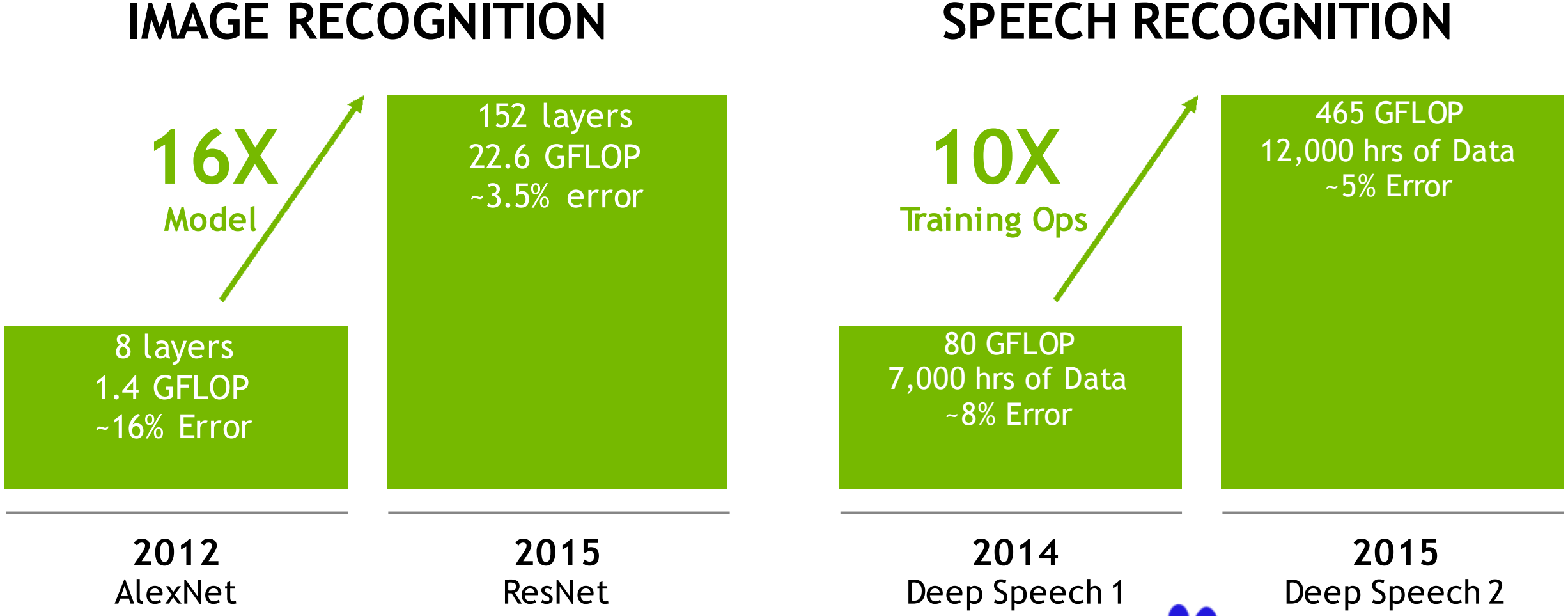
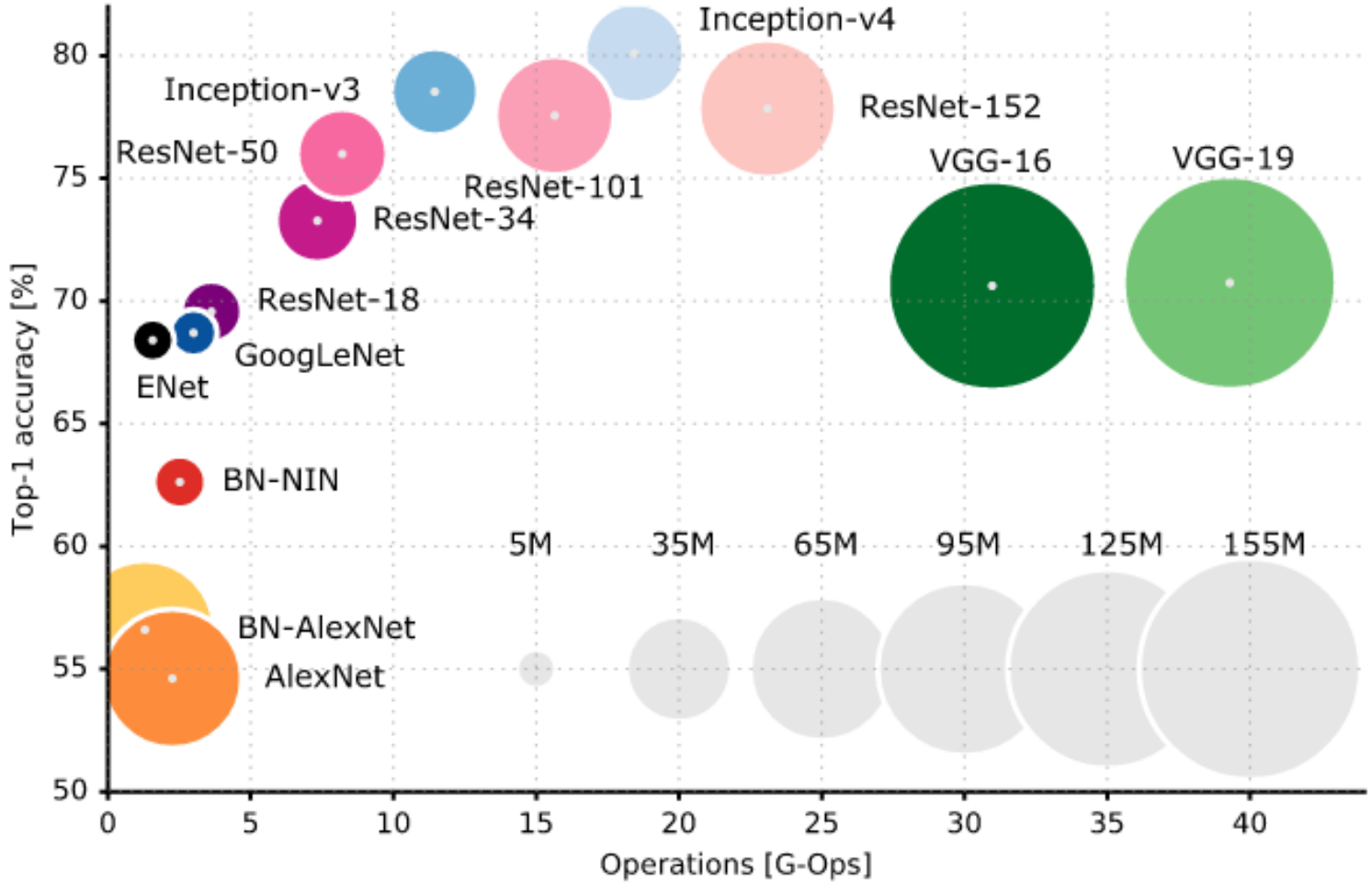
Model Compression and Acceleration

Tian Sheuan Chang
張添烜

Models Are Getting Larger



DL Models Are Getting Larger and More Complex



https://culurciello.github.io/tech/2016/06/04/nets.html

Challenges of Large Model Size

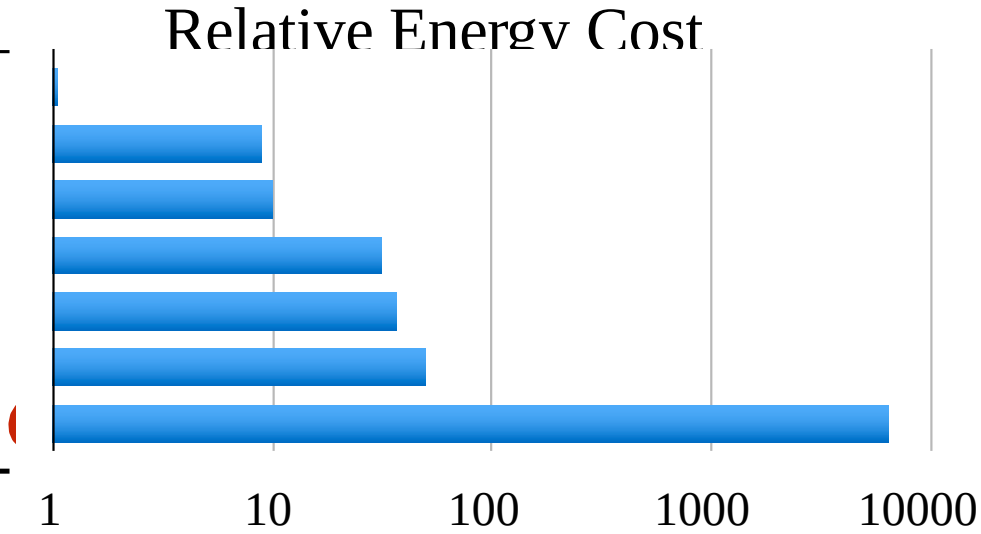
- Real time execution

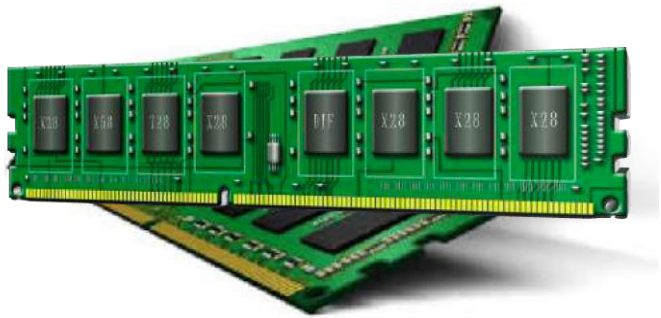
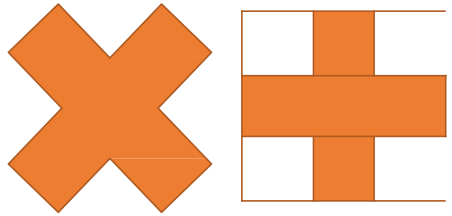
- Power consumption

Where is the Energy Consumed?

larger model => more memory reference => more energy

Operation	Energy [pJ]
32 bit int ADD 32	0.1
bit float ADD	0.9
32 bit Register File 32	1
bit int MULT 32 bit	3.1
float MULT	3.7
32 bit SRAM Cache	5



1  = 1000 

This image is in the public domain

how to make deep learning more efficient?

Outline

原理:神經網路具有高容錯性與可塑性

Static model acceleration	Smaller: fewer weights	**Pruning**: weights, vector, filter, channel
	Lower: **quantization** - from full precision to binary precision	Low precision, weight sharing
		Ternary and binary weights
		Ternary and binary weights+activations
	Less computation	Low rank approximation
		Winograd transformation
	Faster: low complexity architecture	Squeezenet
		Mobilenet
		Shufflenet

Outline

Dynamic execution

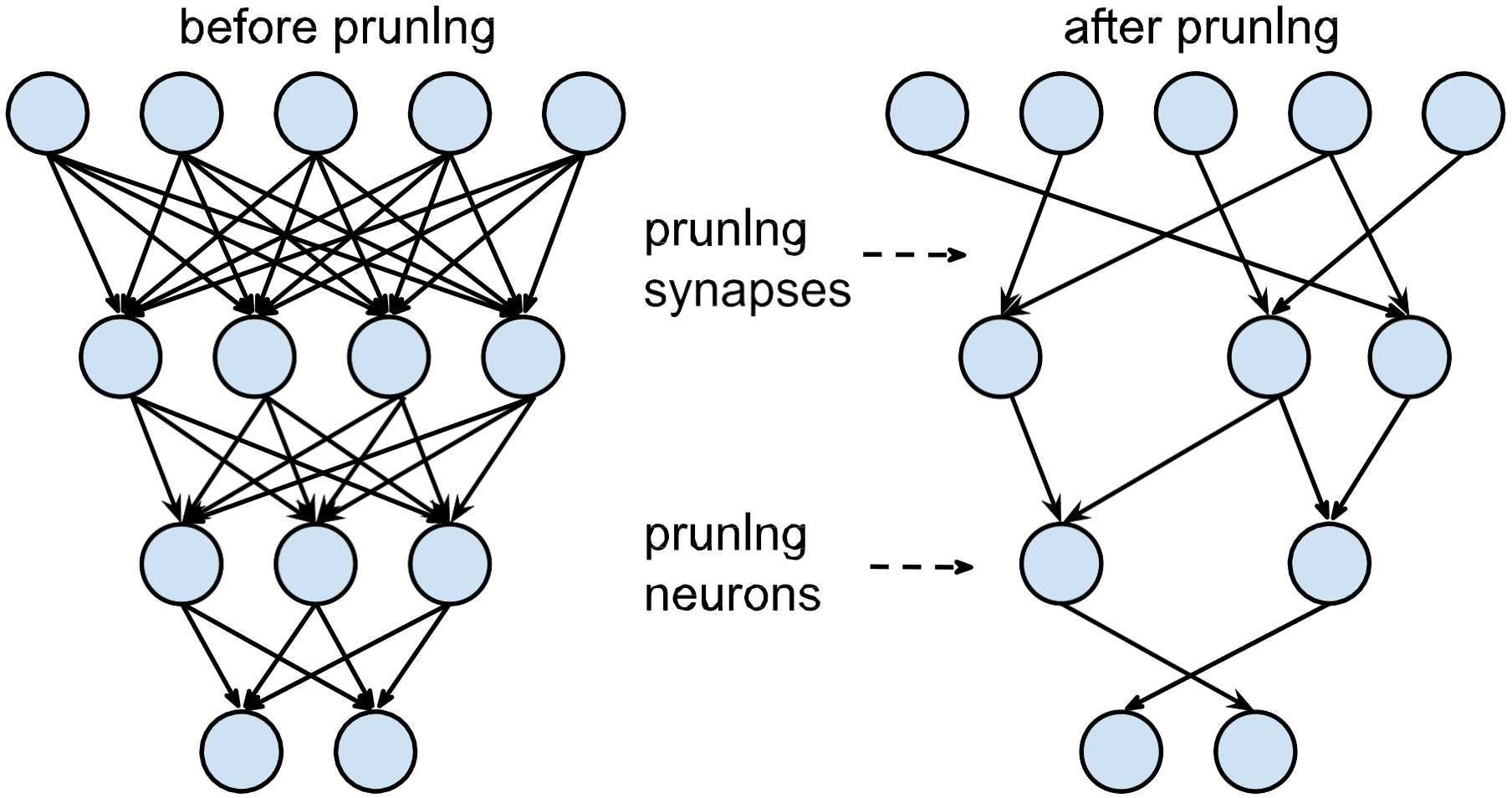
Execution based on input

- SBNet
- SGAD
- Dynamic channel pruning

PRUNING: FEWER WEIGHTS
從既有MODEL開始，先把MODEL 縮小一點

Pruning

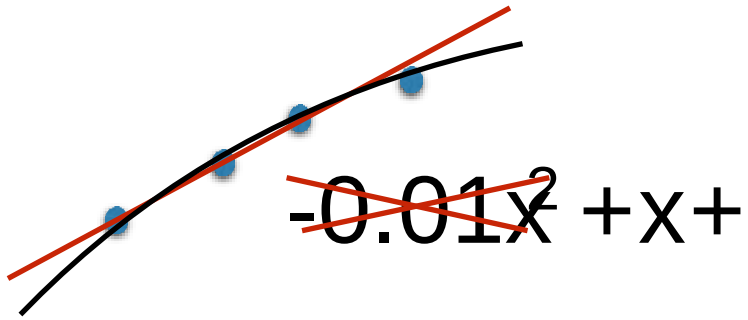
- Concept: Not all weights are created equal importance

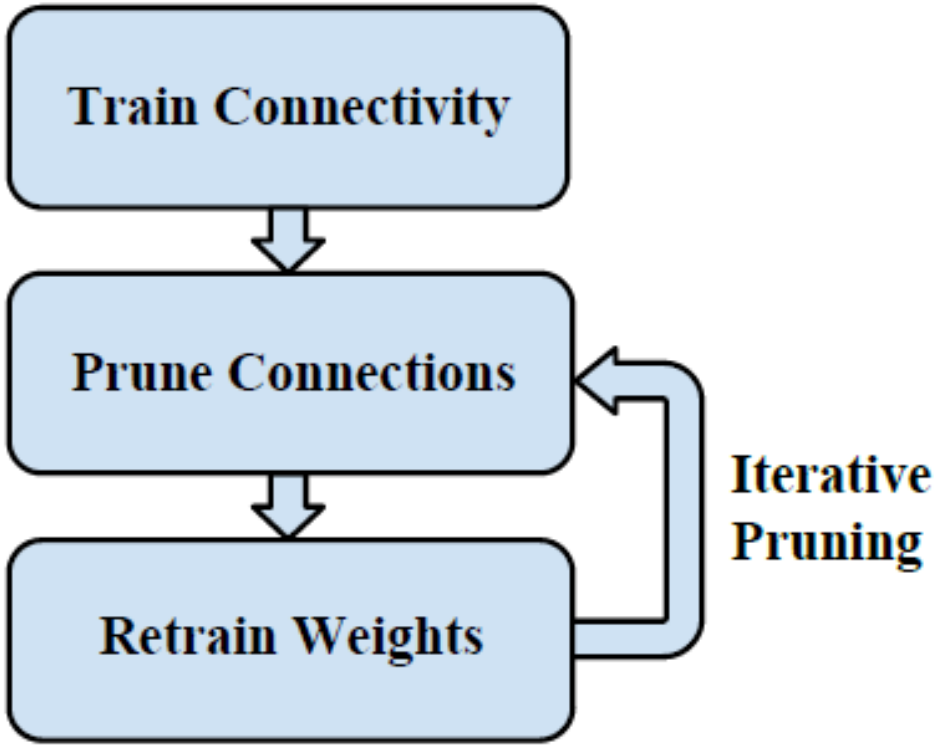


Han et al. Learning both Weights and Connections for Efficient Neural Networks, NIPS'15

Pruning

- Prune unimportance weight
  - Magnitude based method
  - If (w < threshold) w = 0;





Algorithm 1: Pruning Deep Neural Networks

Initialization: $W^{(0)}$ *with* $W^{(0)} \sim N(0, \Sigma)$, $iter = 0$.
Hyper-parameter: *threshold*, δ.
Output : $W^{(t)}$.

Train Connectivity

while *not converged* **do**
  $W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)})$;
  $t = t + 1$;
end

Prune Connections

// initialize the mask by thresholding the weights.
$Mask = \mathbb{1}(|W| > threshold)$;
$W = W \cdot Mask$;

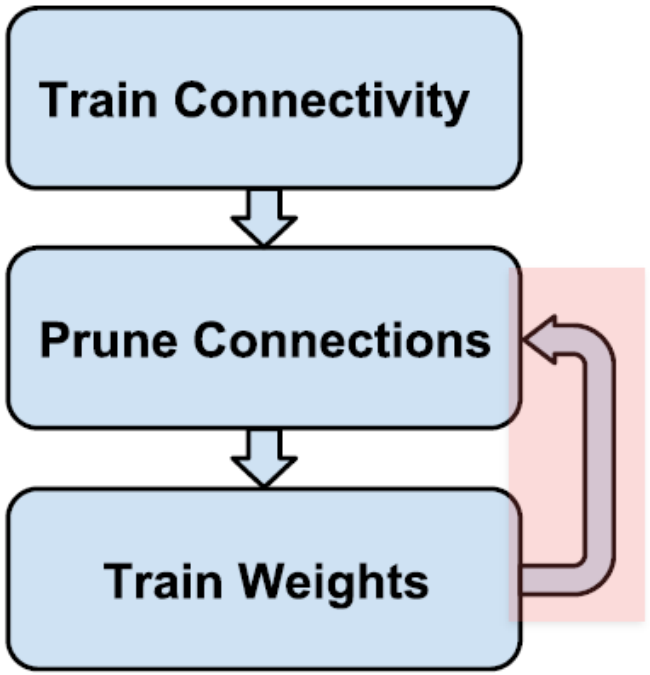
Retrain Weights

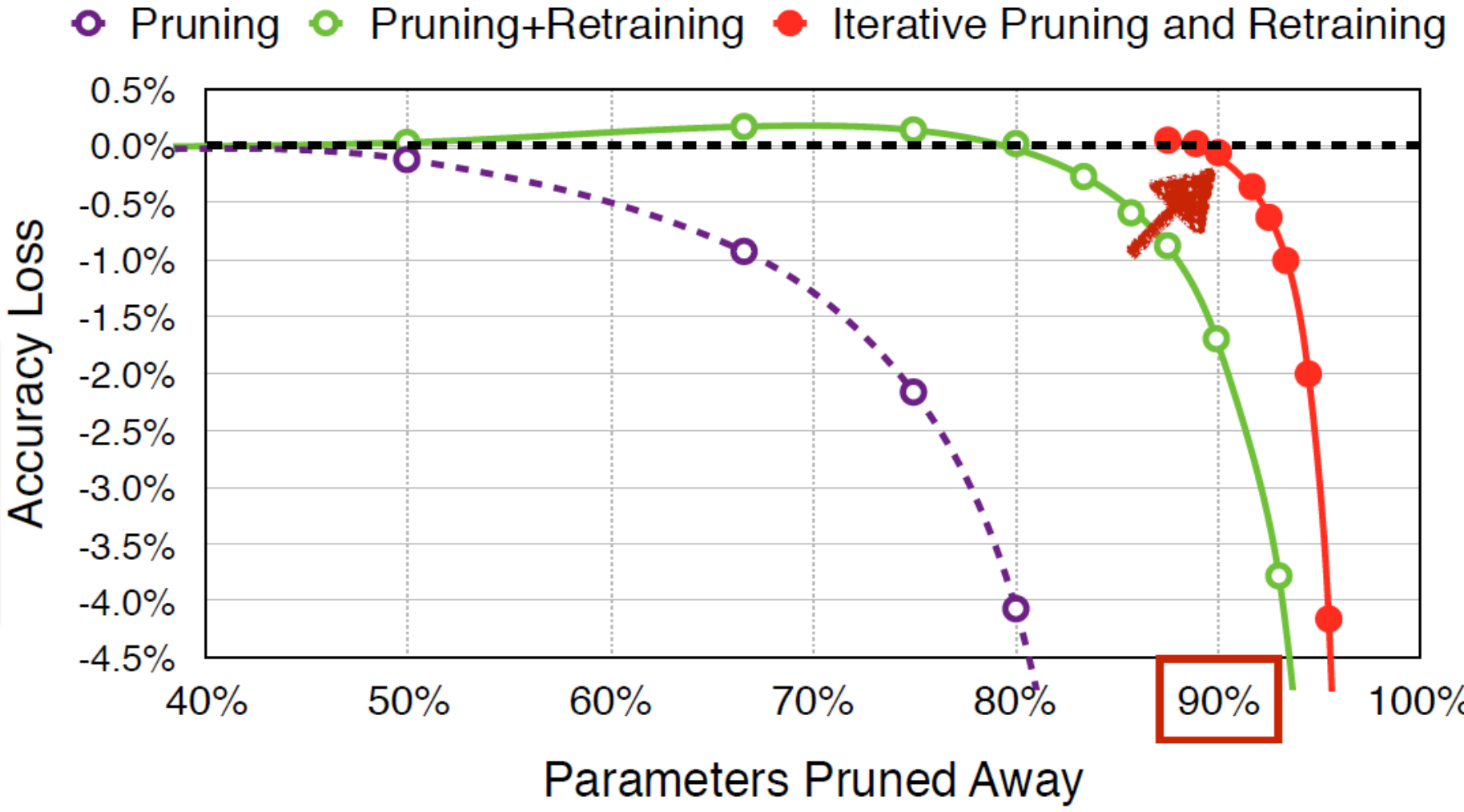
while *not converged* **do**
  $W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)})$;
  $W^{(t)} = W^{(t)} \cdot Mask$;
  $t = t + 1$;
end

Iterative Pruning

$threshold = threshold + \delta[iter++]$;
goto *Pruning Connections;*

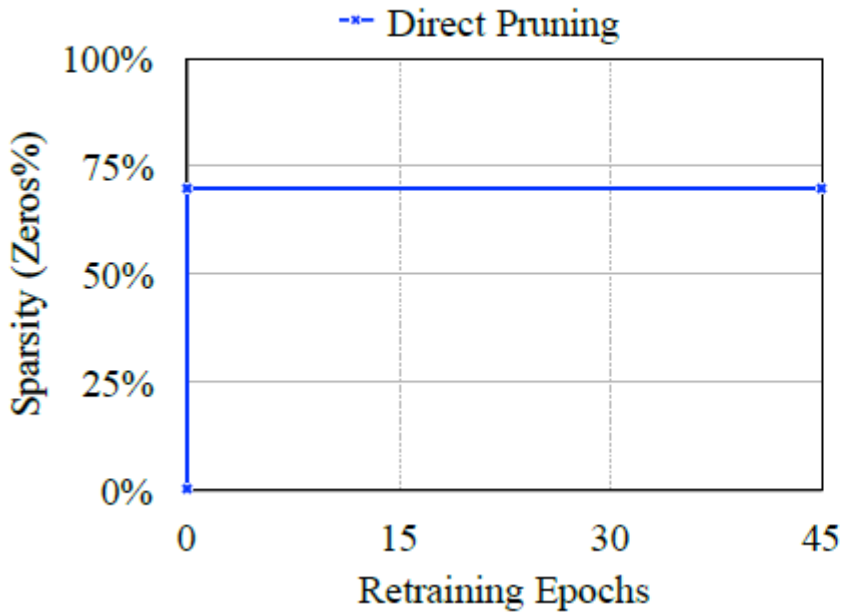
Iteratively Retrain to Recover Accuracy

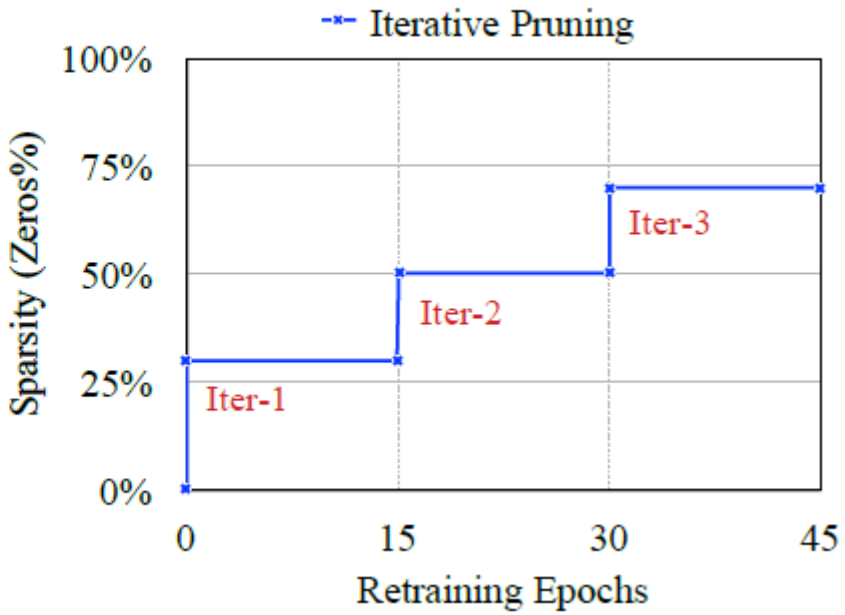




Pruning: pruning procedure

- Iterative pruning
  - better pruning rate than direct pruning for AlexNet

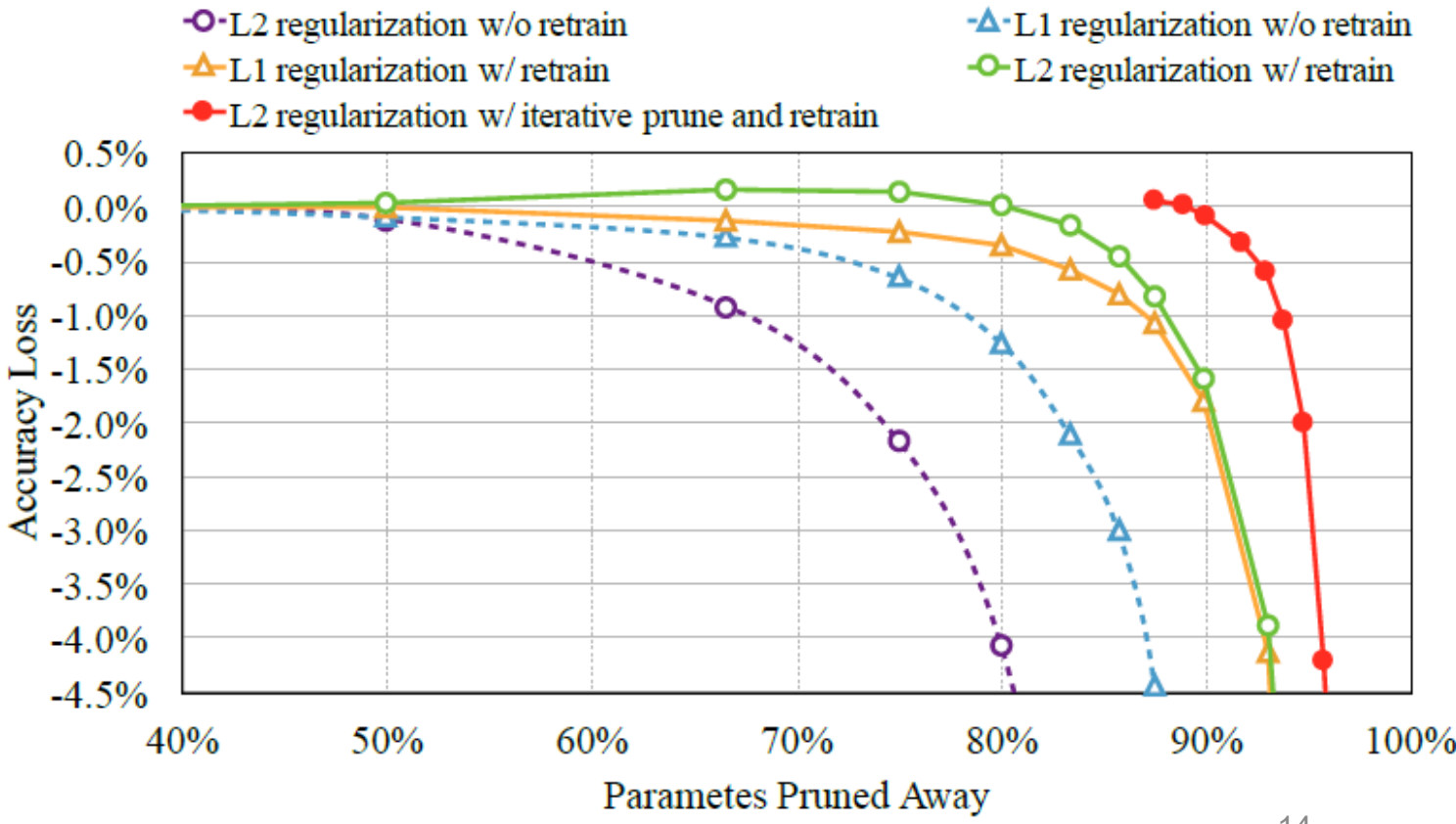




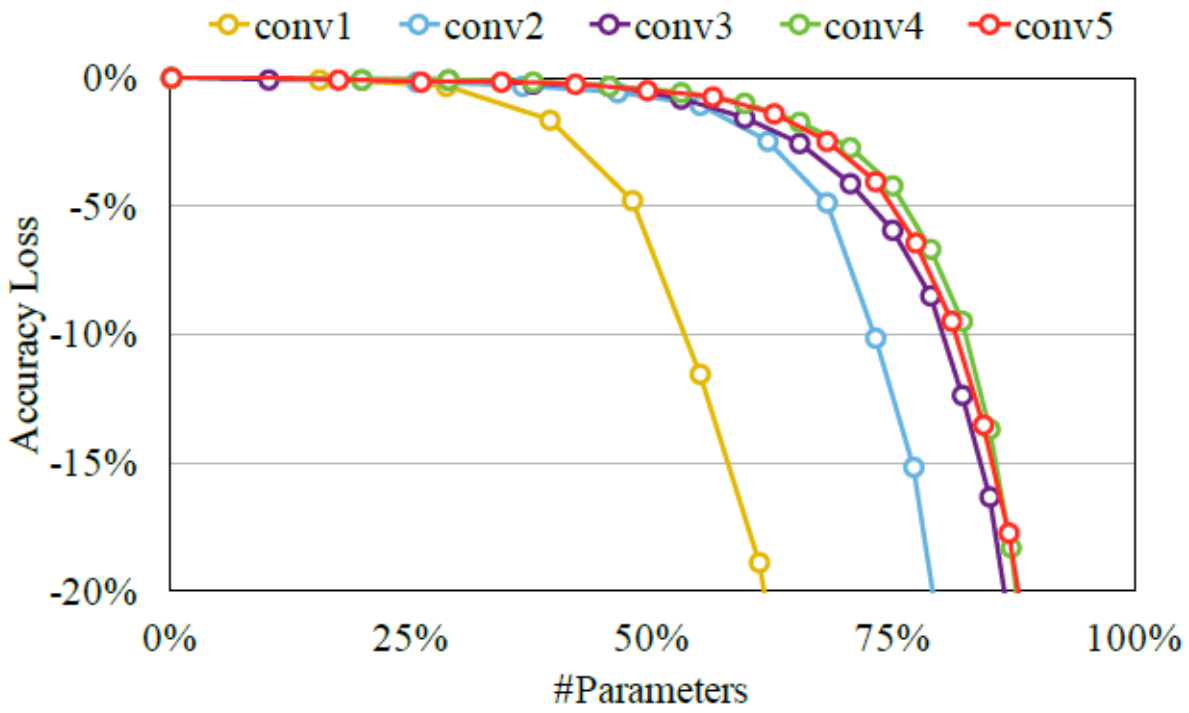
Pruning: pruning procedure

- Free lunch: 2x wo retrain
- 9x w retrain
- L1 or L2 regularization
  - L1 results in more zeros
  - L2 + pruning+ retrain is better
    - No benefit to push more zeros
  - Best: L2+ iterative prune + retrain

砍完，重新訓練才能恢復準確率



Pruning: Sensitivity



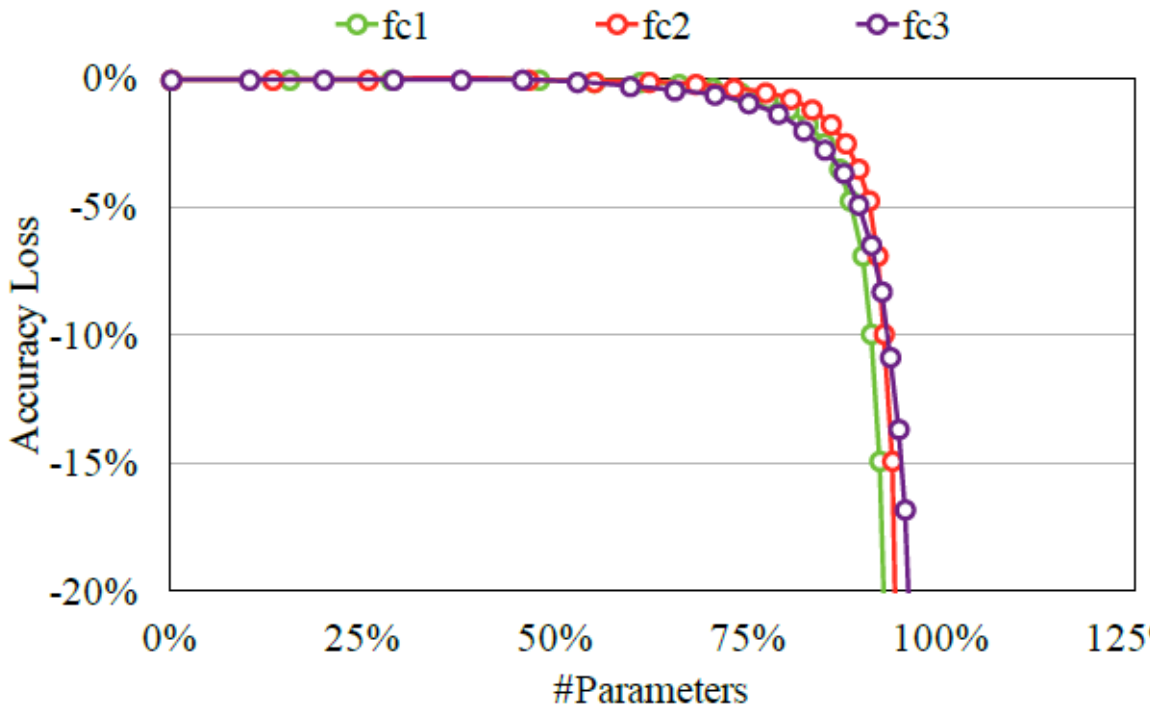
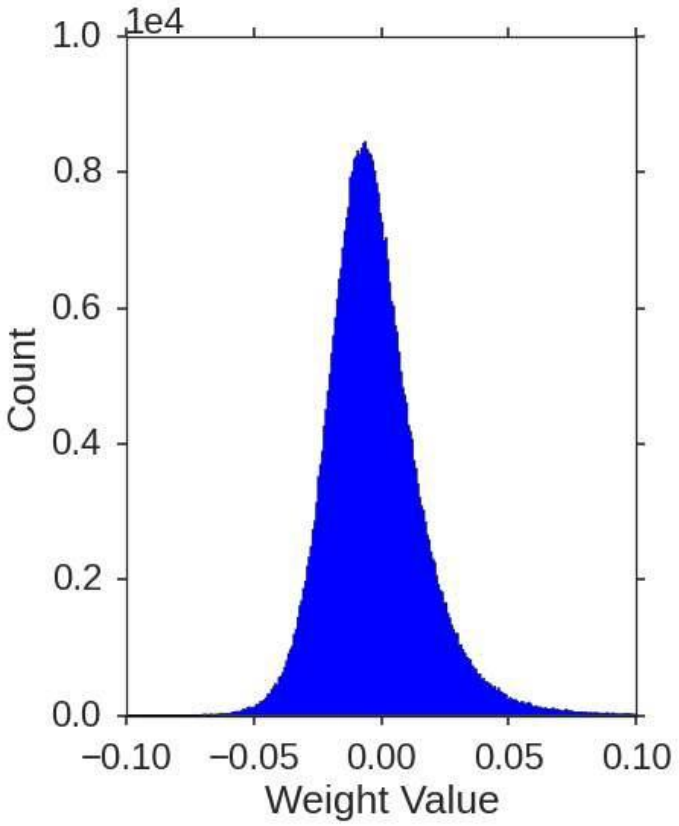


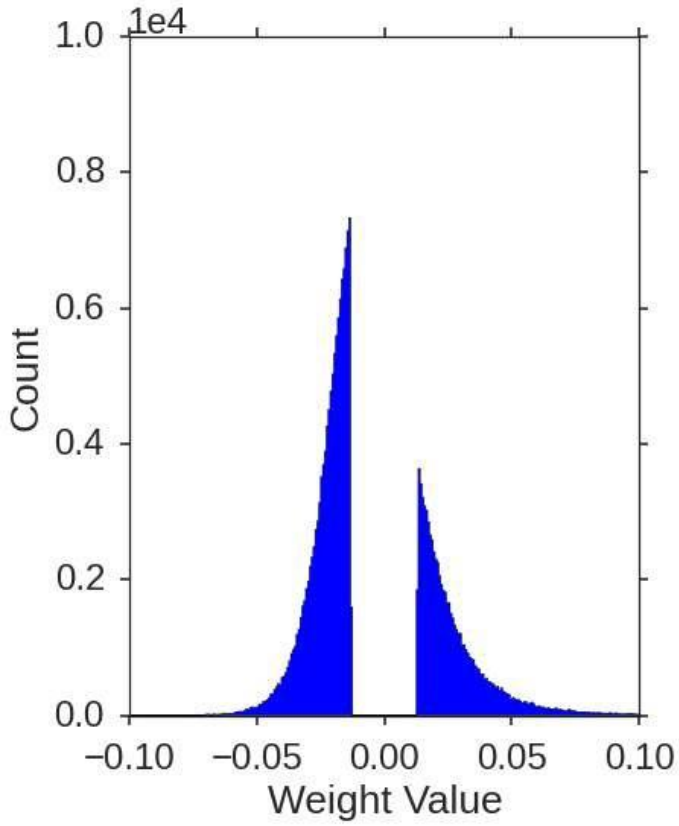
Figure 3.14: Pruning sensitivity for CONV layer (left) and FC layer (right) of AlexNet.

Pruning Changes Weight Distribution

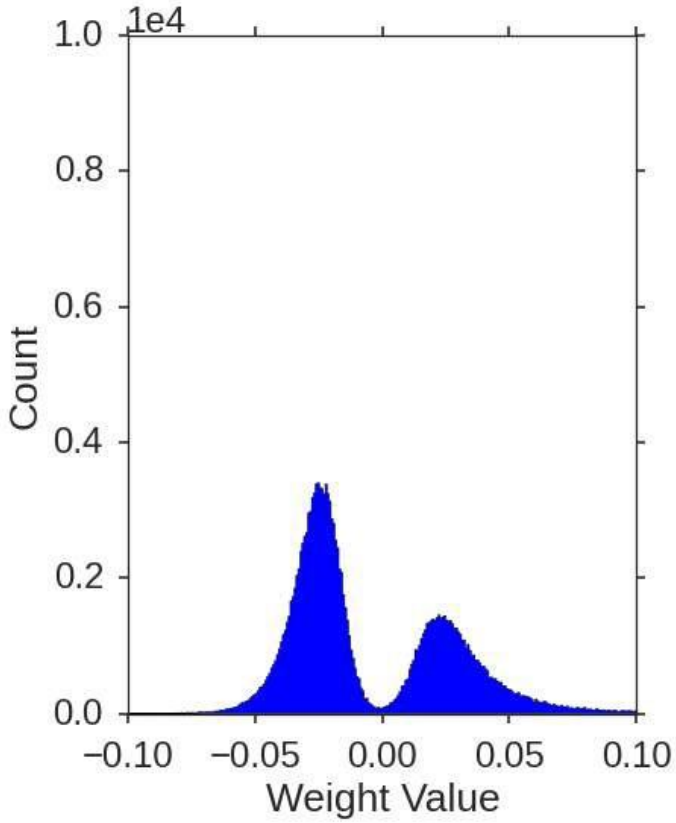
Before Pruning



After Pruning



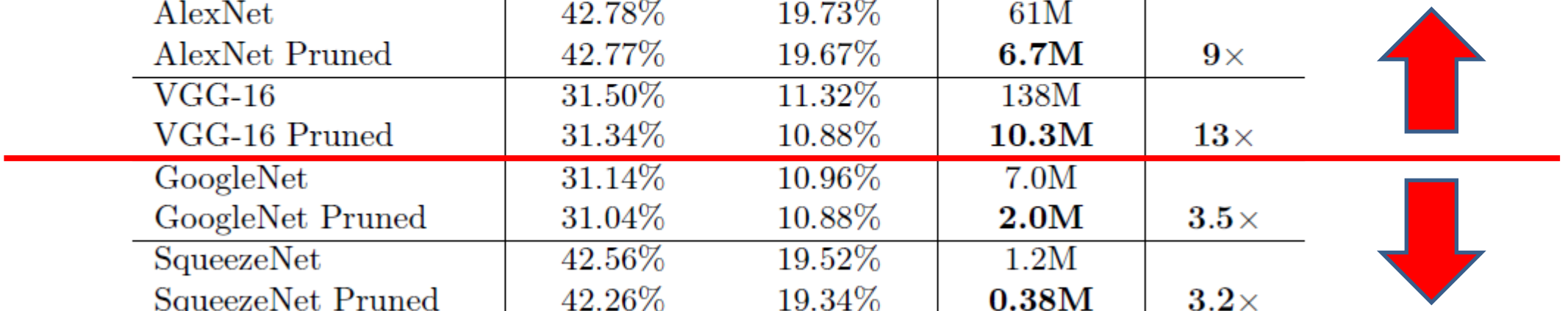
After Retraining



Conv5 layer of Alexnet. Representative for other network layers as well.

Pruning: Results

Network	Top-1 Error	Top-5 Error	Parameters	Pruning Rate
LeNet-300-100	1.64%	-	267K	
LeNet-300-100 Pruned	1.59%	-	**22K**	**12×**
LeNet-5	0.80%	-	431K	
LeNet-5 Pruned	0.77%	-	**36K**	**12×**
AlexNet	42.78%	19.73%	61M	
AlexNet Pruned	42.77%	19.67%	**6.7M**	**9×**
VGG-16	31.50%	11.32%	138M	
VGG-16 Pruned	31.34%	10.88%	**10.3M**	**13×**
GoogleNet	31.14%	10.96%	7.0M	
GoogleNet Pruned	31.04%	10.88%	**2.0M**	**3.5×**
SqueezeNet	42.56%	19.52%	1.2M	
SqueezeNet Pruned	42.26%	19.34%	**0.38M**	**3.2×**
ResNet-50	23.85%	7.13%	25.5M	
ResNet-50 Pruned	23.65%	6.85%	**7.47M**	**3.4×**



FC 層權重數量太多

Fully convolutional networ

FC 層權重數量少

Pruning AlexNet reduces the number of weights by 9× and computation by 3×.

Layer	Weights	FLOP	Activation%	Weight%	FLOP%
conv1	35K	211M	88%	84%	84%
conv2	307K	448M	52%	38%	33%
conv3	885K	299M	37%	35%	18%
conv4	663K	224M	40%	37%	14%
conv5	442K	150M	34%	37%	14%
fc1	38M	75M	36%	9%	3%
fc2	17M	34M	40%	9%	3%
fc3	4M	8M	100%	25%	10%
total	61M	1.5B	54%	**11%**	**30%**

ZERO WEIGHT/ACTIVATION RATIO OF ALEXNET [1]

Layer	Zero Weight [%]	Zero Activation [%]
conv1	15.7	0
conv2	62.1	50.9
conv3	65.4	76.3
conv4	62.8	61.8
conv5	63.1	59.0

Nonzero computation = 0.4*0.4=0.16

Pruning VGG-16 reduces the number of weights by 12× and computation by 5×.

Layer	Weights	FLOP	Activation%	Weight%	FLOP%
conv1_1	2K	0.2B	53%	58%	58%
conv1_2	37K	3.7B	89%	22%	12%
conv2_1	74K	1.8B	80%	34%	30%
conv2_2	148K	3.7B	81%	36%	29%
conv3_1	295K	1.8B	68%	53%	43%
conv3_2	590K	3.7B	70%	24%	16%
conv3_3	590K	3.7B	64%	42%	29%
conv4_1	1M	1.8B	51%	32%	21%
conv4_2	2M	3.7B	45%	27%	14%
conv4_3	2M	3.7B	34%	34%	15%
conv5_1	2M	925M	32%	35%	12%
conv5_2	2M	925M	29%	29%	9%
conv5_3	2M	925M	19%	36%	11%
fc6	103M	206M	38%	4%	1%
fc7	17M	34M	42%	4%	2%
fc8	4M	8M	100%	23%	9%

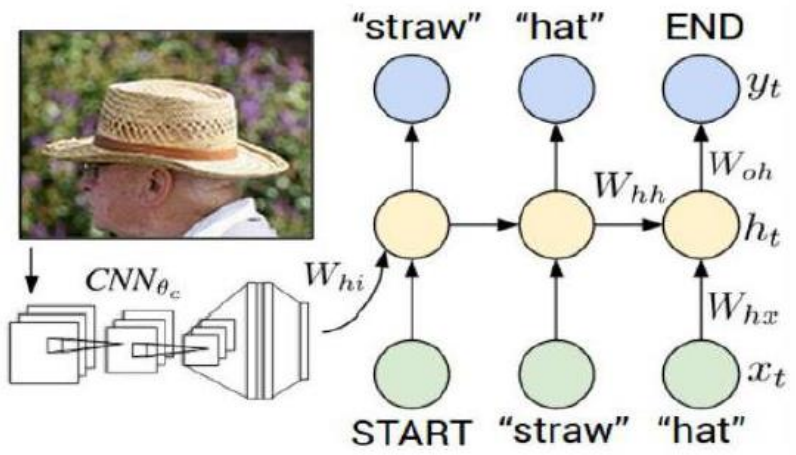
Weight in FC layer can be very sparse

Layer	Weights	FLOP	Activation%	Weight%	FLOP%
conv1	9K	236B	90%	50%	50%
layer1.0.conv1	4K	26B	58%	40%	36%
layer1.0.conv2	37K	231B	64%	30%	18%
layer1.0.conv3	16K	103B	59%	30%	19%
layer1.0.shortcut	16K	103B	59%	40%	36%
layer1.1.conv1	16K	103B	48%	30%	18%
layer1.1.conv2	37K	231B	51%	30%	14%
layer1.1.conv3	16K	103B	75%	30%	15%
layer1.2.conv1	16K	103B	49%	30%	22%
layer1.2.conv2	37K	231B	44%	30%	15%
layer1.2.conv3	16K	103B	80%	30%	13%
layer2.0.conv1	33K	206B	41%	40%	32%
layer2.0.conv2	147K	231B	58%	33%	14%
layer2.0.conv3	66K	103B	50%	30%	17%
layer2.0.shortcut	131K	206B	50%	30%	24%
layer2.1.conv1	66K	103B	61%	30%	15%
layer2.1.conv2	147K	231B	56%	30%	18%
layer2.1.conv3	66K	103B	66%	30%	17%
layer2.2.conv1	66K	103B	54%	30%	20%
layer2.2.conv2	147K	231B	55%	30%	16%
layer2.2.conv3	66K	103B	58%	30%	16%
layer2.3.conv1	66K	103B	49%	30%	17%
layer2.3.conv2	147K	231B	41%	30%	15%
layer2.3.conv3	66K	103B	59%	30%	12%
layer3.0.conv1	131K	206B	32%	40%	23%
layer3.0.conv2	590K	231B	62%	30%	10%
layer3.0.conv3	262K	103B	47%	30%	18%
layer3.0.shortcut	524K	206B	47%	30%	18%
layer3.1.conv1	262K	103B	48%	30%	14%
layer3.1.conv2	590K	231B	43%	30%	14%
layer3.1.conv3	262K	103B	52%	30%	13%
layer3.2.conv1	262K	103B	41%	30%	15%
layer3.2.conv2	590K	231B	40%	30%	12%
layer3.2.conv3	262K	103B	50%	30%	12%
layer3.3.conv1	262K	103B	36%	30%	15%
layer3.3.conv2	590K	231B	39%	30%	11%
layer3.3.conv3	262K	103B	48%	30%	12%
layer3.4.conv1	262K	103B	34%	30%	14%
layer3.4.conv2	590K	231B	35%	30%	10%
layer3.4.conv3	262K	103B	40%	30%	11%
layer3.5.conv1	262K	103B	31%	30%	12%
layer3.5.conv2	590K	231B	36%	30%	9%
layer3.5.conv3	262K	103B	32%	30%	11%
layer4.0.conv1	524K	206B	23%	30%	10%
layer4.0.conv2	2M	231B	38%	30%	7%
layer4.0.conv3	1M	103B	41%	30%	12%
layer4.0.shortcut	2M	206B	41%	30%	10%
layer4.1.conv1	1M	103B	27%	30%	12%
layer4.1.conv2	2M	231B	32%	30%	8%
layer4.1.conv3	1M	103B	56%	30%	10%
layer4.1.conv1	1M	103B	21%	30%	17%
layer4.1.conv2	2M	231B	37%	30%	6%
layer4.1.conv3	1M	103B	100%	30%	11%
fc	2M	4K	100%	20%	20%
total	25.5M	8G	56%	29%	16%

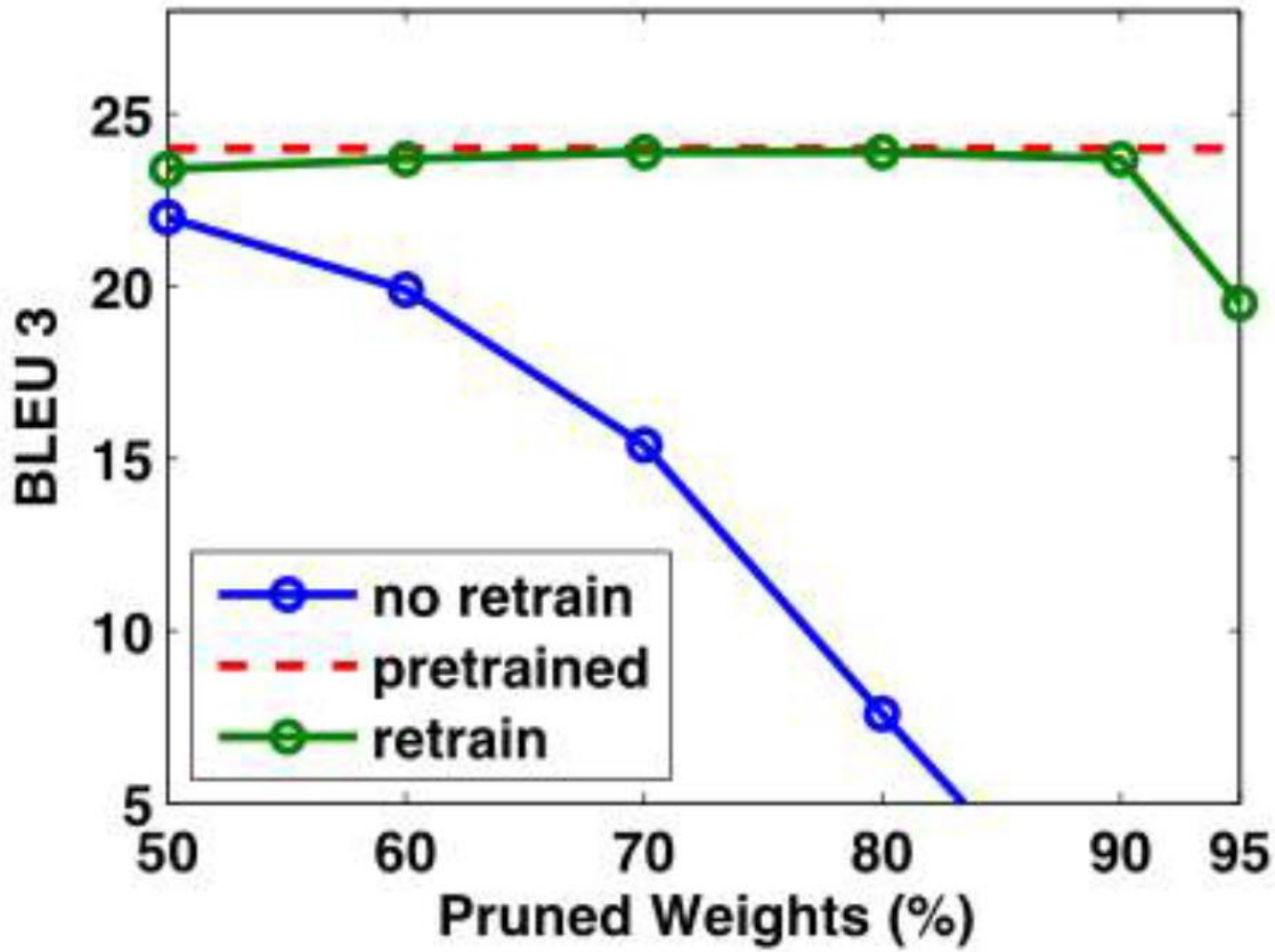
ResNet-50 reduces the number of weights by 3.4X and computation by 6.25X

1. Bypass changes the activation sparsity due to add operation
2. FC layer in ResNet is global average pooling, no reduction in activation

Pruning: RNN and LSTM



*Karpathy et al, "Deep Visual-Semantic Alignments for Generating Image Descriptions", 2015.



Pruning: Effect on RNN and LSTM

90%

- **Original**: a basketball player in a white uniform is playing with a ball
- **Pruned 90%**: a basketball player in a white uniform is playing with a basketball

90%

- **Original** : a man is riding a surfboard on a wave
- **Pruned 90%**: a man in a wetsuit is riding a wave on a beach

95%

- **Original** : a soccer player in red is running in the field
- **Pruned <u>95%</u>**: a man in a red shirt and black and white black shirt is running through a field

Learning Structured Sparsity in Deep Neural Networks

Fine grained 需要比較高的sparsity，才有加速效果

- With regularization
- Cost = data loss + structured regularizer + nonstructured regularizer

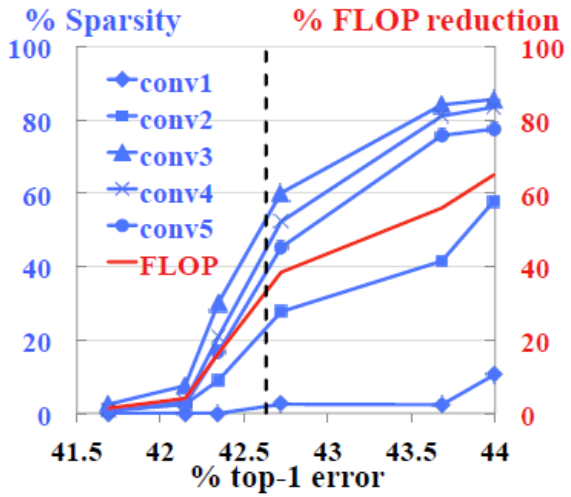


Table 4: Sparsity and speedup of *AlexNet* on ILSVRC 2012

#	Method	Top1 err.	Statistics	conv1	conv2	conv3	conv4	conv5
1	ℓ_1	44.67%	sparsity	67.6%	92.4%	97.2%	96.6%	94.3%
			CPU ×	0.80	2.91	4.84	3.83	2.76
			GPU ×	0.25	0.52	1.38	1.04	1.36
2	SSL	44.66%	column sparsity	0.0%	63.2%	76.9%	84.7%	80.7%
			row sparsity	9.4%	12.9%	40.6%	46.9%	0.0%
			CPU ×	1.05	3.37	6.27	9.73	4.93
			GPU ×	1.00	2.37	4.94	4.03	3.05
3	pruning[7]	42.80%	sparsity	16.0%	62.0%	65.0%	63.0%	63.0%
4	ℓ_1	42.51%	sparsity	14.7%	76.2%	85.3%	81.5%	76.3%
			CPU ×	0.34	0.99	1.30	1.10	0.93
			GPU ×	0.08	0.17	0.42	0.30	0.32
5	SSL	42.53%	column sparsity	0.00%	20.9%	39.7%	39.7%	24.6%
			CPU ×	1.00	1.27	1.64	1.68	1.32
			GPU ×	1.00	1.25	1.63	1.72	1.36

DNNs require a very high non-structured sparsity to achieve a reasonable speedup (The speedups are even negative when the sparsity is low). SSL, however, can always achieve positive speedups

Pruning: Regular Sparsity for HW Load Balance

- Sort the L1 norm of that granularity
  - Discard if smaller
  - E.g. target 30% sparsity
  - Discard smallest 70%

$$S_i = \sum_{w \in G_i} |w|$$

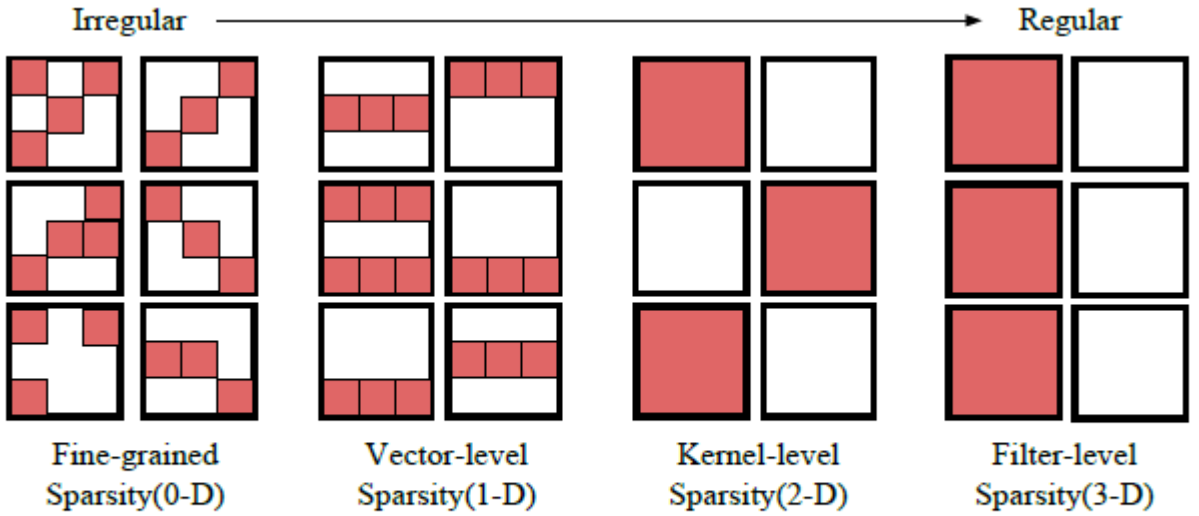


Figure 1: Different sparse structure in a 4-dimensional weight tensor. Regular sparsity makes hardware acceleration easier.

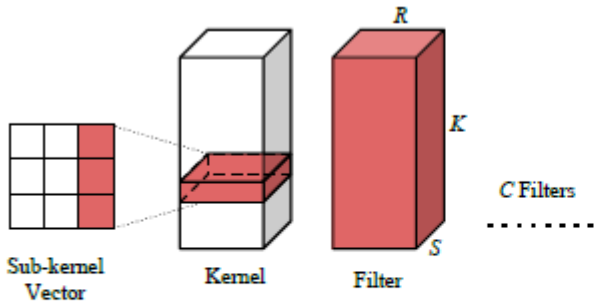


Figure 2: Example of Sub-kernel Vector, Filter and Kernel.

```
Weights = Array(C, K, R, S)

# Case: Dimension-level granularity
Filter(3-Dim) = Weights[c, :, :, :]
Kernel(2-Dim) = Weights[c, k, :, :]
Vector(1-Dim) = Weights[c, k, r, :]
Fine-grain(0-Dim) = Weights[c, k, r, s]
```

Pseudo code: different granularity levels

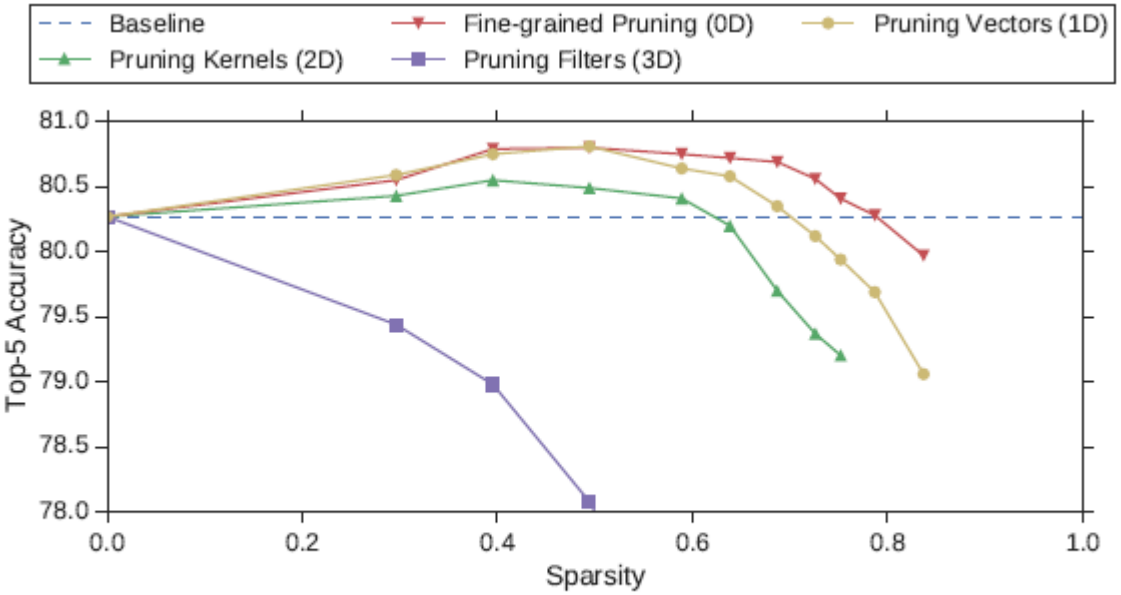


Figure 4: Accuracy-Sparsity Curve of AlexNet obtained by iterative pruning.

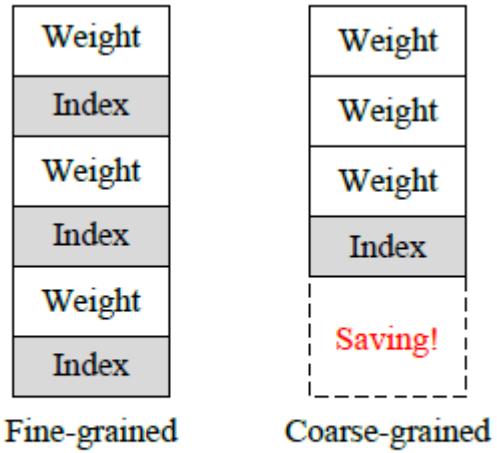


Figure 5: Illustration of index saving.

Table 1: Comparison of accuracies with the same density/sparsity.

Model	Density	Granularity	Top-5
AlexNet	24.8%	Kernel Pruning (2-D)	79.20%
		Vector Pruning (1-D)	79.94%
		Fine-grained Pruning (0-D)	**80.41%**
VGG-16	23.5%	Kernel Pruning (2-D)	89.70%
		Vector Pruning (1-D)	90.48%
		Fine-grained Pruning (0-D)	**90.56%**
GoogLeNet	38.4%	Kernel Pruning (2-D)	88.83%
		Vector Pruning (1-D)	89.11%
		Fine-grained Pruning (0-D)	**89.40%**
ResNet-50	40.0%	Kernel Pruning (2-D)	92.07%
		Vector Pruning (1-D)	92.26%
		Fine-grained Pruning (0-D)	**92.34%**
DenseNet-121	30.1%	Kernel Pruning (2-D)	91.56%
		Vector Pruning (1-D)	91.89%
		Fine-grained Pruning (0-D)	**92.21%**

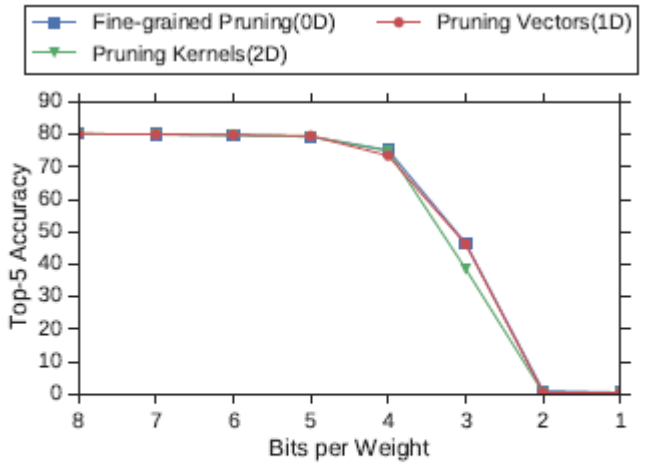


Figure 6: Three curves are almost identical, indicating sparsity structure does not impact quantization.

Pruning: Filter Pruning Result on Recent Models

- Training a pruned model from scratch performs worse
- Threshold criteria
  - L1 norm is better than L2 norm
  - Prune smallest better than random or largest

Table 1: Overall results. The best test/validation accuracy during the retraining process is reported. Training a pruned model from scratch performs worse than retraining a pruned model, which may indicate the difficulty of training a network with a small capacity.

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%

ResNet-56-pruned-A pruning 10% filters while skipping the sensitive layers 16, 20, 38 and 54
ResNet-56-pruned-B skips more layers (16, 18, 20, 34, 38, 54) and prunes layers with p1=60%, p2=30% and p3=10%.

Network Slimming: BN Scaling Factors Channel Sparsity to Channel Pruning

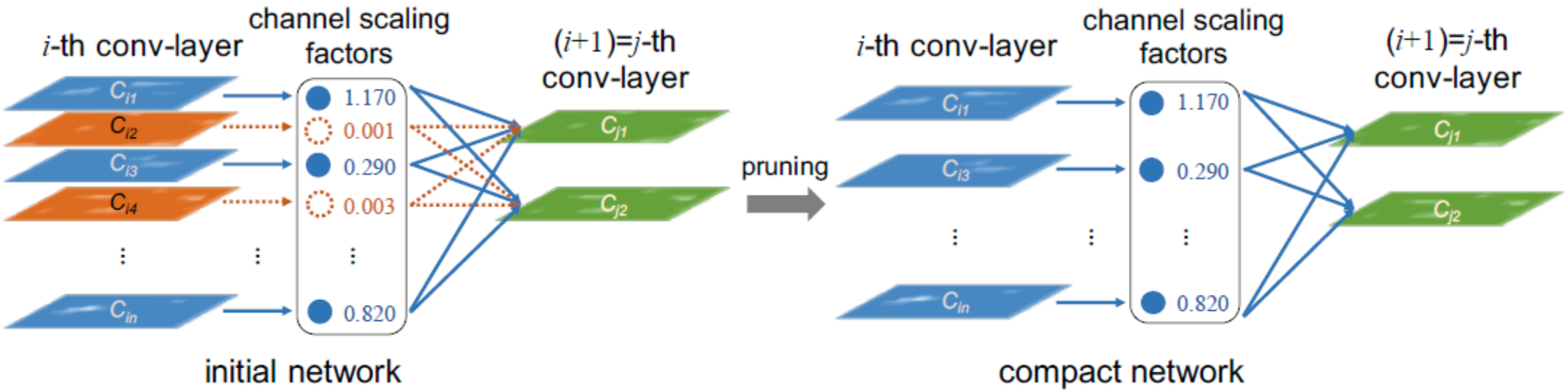


Figure 1: We associate a scaling factor (reused from a batch normalization layer) with each channel in convolutional layers. Sparsity regularization is imposed on these scaling factors during training to automatically identify unimportant channels. The channels with small scaling factor values (in orange color) will be pruned (left side). After pruning, we obtain compact models (right side), which are then fine-tuned to achieve comparable (or even higher) accuracy as normally trained full network.

Channel Sparsity to Channel Pruning

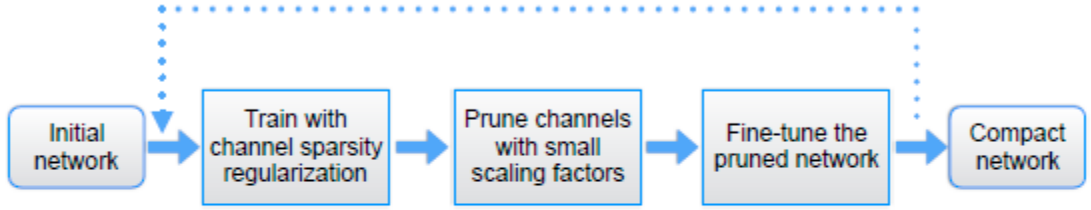


Figure 2: Flow-chart of network slimming procedure. The dotted-line is for the multi-pass/iterative scheme.

- L1 norm penalty for sparsity
- Scaling factor combined with BN to select pruned channels

$$\hat{z} = \frac{z_{in} - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}; \quad z_{out} = \gamma \hat{z} + \beta$$

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

VGG-A	Baseline	50% Pruned
Params	132.9M	23.2M
Params Pruned	-	82.5%
FLOPs	4.57×10^{10}	3.18×10^{10}
FLOPs Pruned	-	30.4%
Validation Error (%)	36.69	36.66

Table 2: Results on ImageNet.

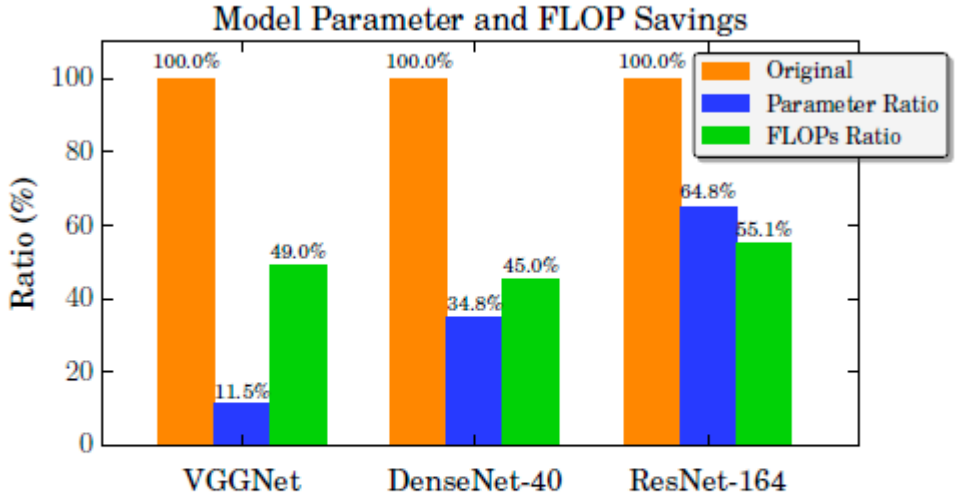


Figure 3: Comparison of pruned models with **lower** test errors on CIFAR-10 than the original models. The blue and green bars are parameter and FLOP ratios between pruned and original models.

(a) Test Errors on CIFAR-10

Model	Test error (%)	Parameters	Pruned	FLOPs	Pruned
VGGNet (Baseline)	6.34	20.04M	-	7.97×10^8	-
VGGNet (70% Pruned)	**6.20**	2.30M	88.5%	3.91×10^8	51.0%
DenseNet-40 (Baseline)	6.11	1.02M	-	5.33×10^8	-
DenseNet-40 (40% Pruned)	**5.19**	0.66M	35.7%	3.81×10^8	28.4%
DenseNet-40 (70% Pruned)	5.65	0.35M	65.2%	2.40×10^8	55.0%
ResNet-164 (Baseline)	5.42	1.70M	-	4.99×10^8	-
ResNet-164 (40% Pruned)	**5.08**	1.44M	14.9%	3.81×10^8	23.7%
ResNet-164 (60% Pruned)	5.27	1.10M	35.2%	2.75×10^8	44.9%

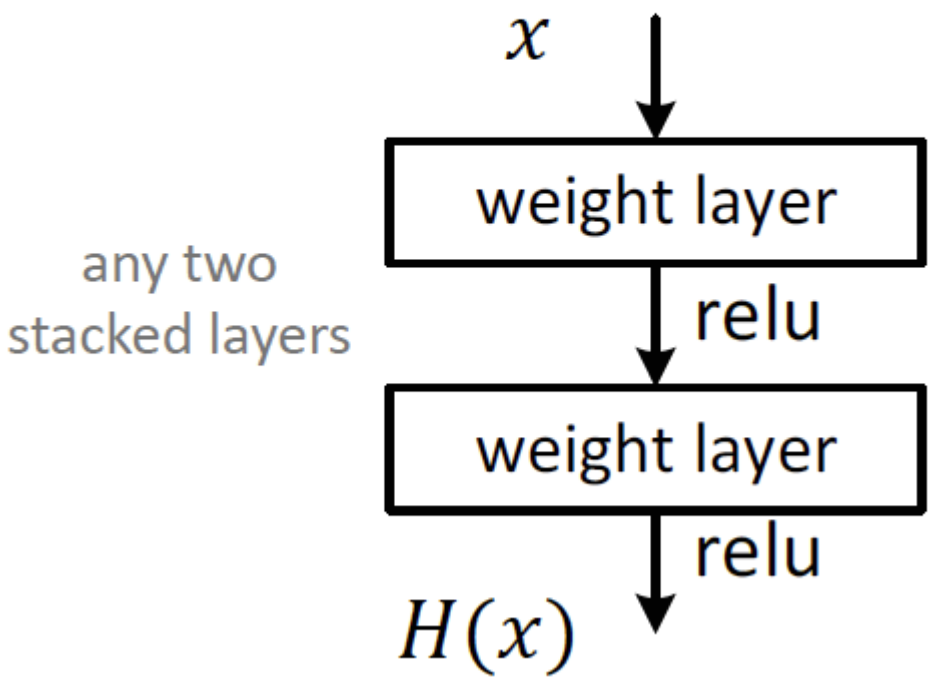
(b) Test Errors on CIFAR-100

Model	Test error (%)	Parameters	Pruned	FLOPs	Pruned
VGGNet (Baseline)	26.74	20.08M	-	7.97×10^8	-
VGGNet (50% Pruned)	**26.52**	5.00M	75.1%	5.01×10^8	37.1%
DenseNet-40 (Baseline)	25.36	1.06M	-	5.33×10^8	-
DenseNet-40 (40% Pruned)	**25.28**	0.66M	37.5%	3.71×10^8	30.3%
DenseNet-40 (60% Pruned)	25.72	0.46M	54.6%	2.81×10^8	47.1%
ResNet-164 (Baseline)	23.37	1.73M	-	5.00×10^8	-
ResNet-164 (40% Pruned)	**22.87**	1.46M	15.5%	3.33×10^8	33.3%
ResNet-164 (60% Pruned)	23.91	1.21M	29.7%	2.47×10^8	50.6%

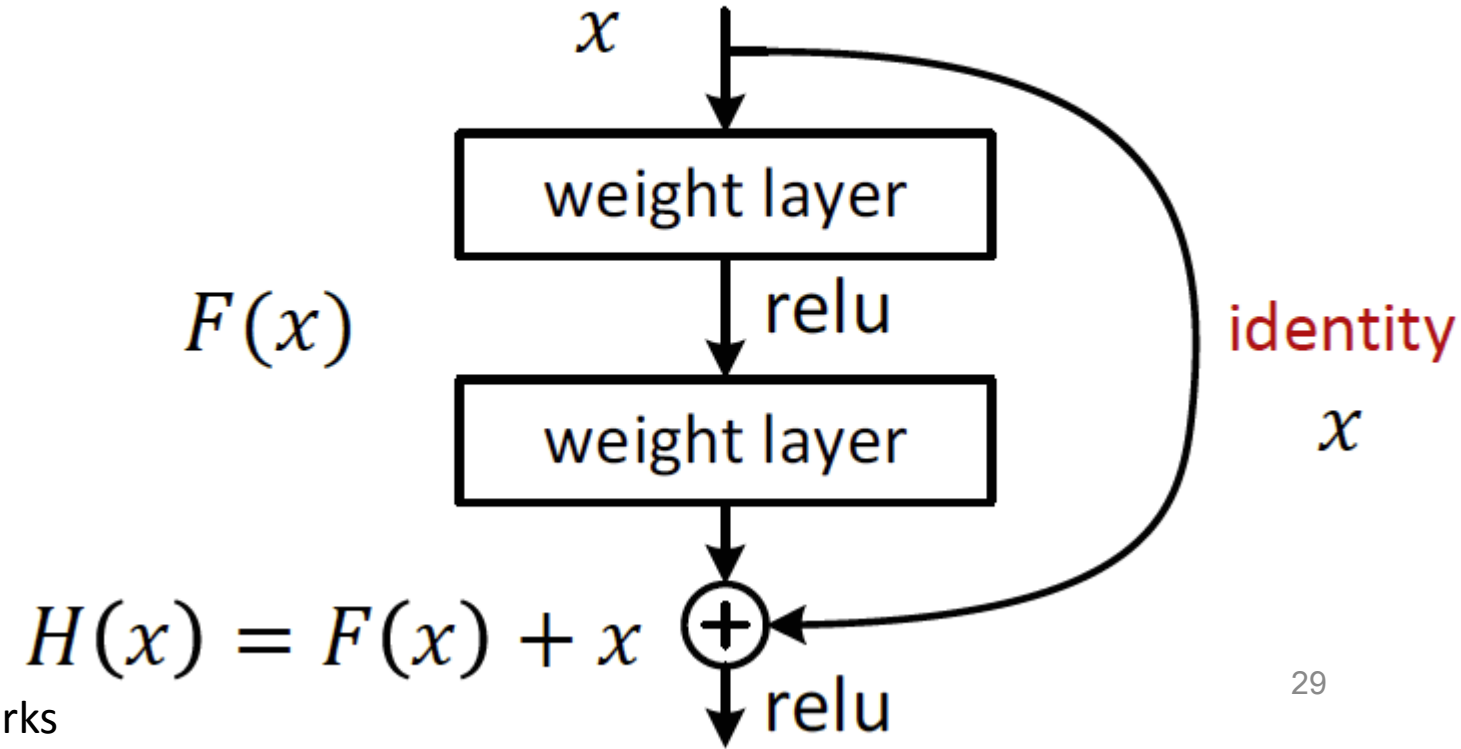
Layer Pruning

- Shortcuts to avoid cuting off the message propagation in the DNN

- Plaint net



- Residual net



Single Shot Network Pruning Based on Connection Sensitivity

- Prune once prior to training 先砍完再訓練，節省時間
  - connection sensitivity as the normalized magnitude of the derivatives
  - measure the sensitivity as to how much influence elements have on the loss function regardless of whether it is positive or negative. This criterion alleviates the dependency on the value of the loss, eliminating the need for pre-training

Algorithm 1 SNIP: Single-shot Network Pruning based on Connection Sensitivity

Require: Loss function L, training dataset $\mathcal{D}$, sparsity level κ ▷ Refer Equation 3

Ensure: $\|\mathbf{w}^*\|_0 \leq \kappa$

1: $\mathbf{w} \leftarrow$ VarianceScalingInitialization ▷ Refer Section 4.2

2: $\mathcal{D}^b = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{b} \sim \mathcal{D}$ ▷ Sample a mini-batch of training data

3: $s_j \leftarrow \frac{|g_j(\mathbf{w}; \mathcal{D}^b)|}{\sum_{k=1}^{m} |g_k(\mathbf{w}; \mathcal{D}^b)|}, \quad \forall j \in \{1 \ldots m\}$ ▷ Connection sensitivity

4: $\tilde{\mathbf{s}} \leftarrow$ SortDescending($\mathbf{s}$)

5: $c_j \leftarrow \mathbb{1}[s_j - \tilde{s}_\kappa \geq 0], \quad \forall j \in \{1 \ldots m\}$ ▷ Pruning: choose top-κ connections

6: $\mathbf{w}^* \leftarrow \arg\min_{\mathbf{w} \in \mathbb{R}^m} L(\mathbf{c} \odot \mathbf{w}; \mathcal{D})$ ▷ Regular training

7: $\mathbf{w}^* \leftarrow \mathbf{c} \odot \mathbf{w}^*$

Architecture	Model	Sparsity (%)	# Parameters	Error (%)	Δ
Convolutional	AlexNet-s	90.0	5.1m → 507k	14.12 → 14.99	+0.87
	AlexNet-b	90.0	8.5m → 849k	13.92 → 14.50	+0.58
	VGG-C	95.0	10.5m → 526k	6.82 → 7.27	+0.45
	VGG-D	95.0	15.2m → 762k	6.76 → 7.09	+0.33
	VGG-like	97.0	15.0m → 449k	8.26 → 8.00	**−0.26**
Residual	WRN-16-8	95.0	10.0m → 548k	6.21 → 6.63	+0.42
	WRN-16-10	95.0	17.1m → 856k	5.91 → 6.43	+0.52
	WRN-22-8	95.0	17.2m → 858k	6.14 → 5.85	**−0.29**
Recurrent	LSTM-s	95.0	137k → 6.8k	1.88 → 1.57	**−0.31**
	LSTM-b	95.0	535k → 26.8k	1.15 → 1.35	+0.20
	GRU-s	95.0	104k → 5.2k	1.87 → 2.41	+0.54
	GRU-b	95.0	404k → 20.2k	1.71 → 1.52	**−0.19**

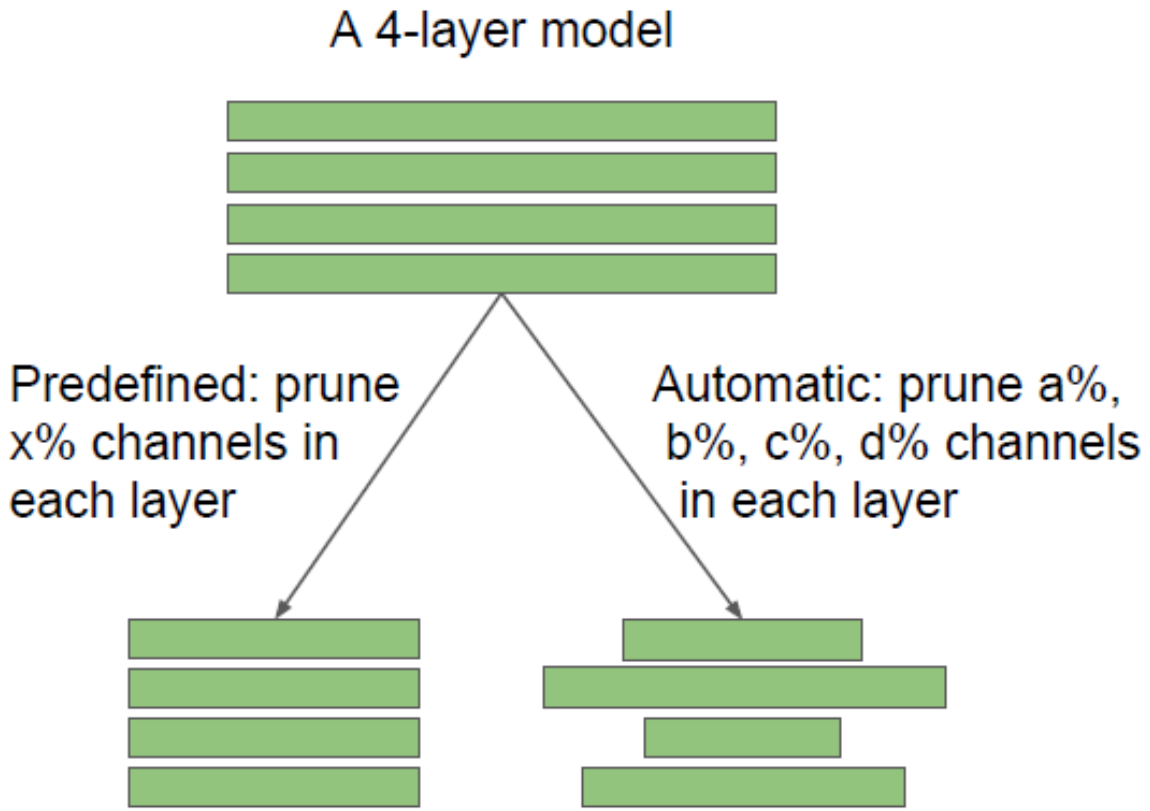
Table 2: Pruning results of the proposed approach on various modern architectures (before → after). AlexNets, VGGs and WRNs are evaluated on CIFAR-10, and LSTMs and GRUs are evaluated on the sequential MNIST classification task. The approach is generally applicable regardless of architecture types and models and results in a significant amount of reduction in the number of parameters with minimal or no loss in performance.

PRUNING AND ARCHITECTURE SEARCH

Rethinking the Value of Network Pruning

- Fine-tuning is not useful
  - comparable or even worse performance than training that model with randomly initialized weights
- Target architecture matters
  - No need for pruning for predefined target network architecture
  - Directly train the target network from scratch
- 3 implications
  - large, over-parameterized model is not necessary for efficient final model
  - learned "important" weights of the large model are not necessarily useful for the small pruned model
  - Pruned architecture, instead of weights, is more important
  - (Pruning as a network architecture search)

Source: Rethinking the Value of Network Pruning



L1-norm based channel pruning[Li, 2017]
ThiNet[Luo, 2017]
Regression based feature reconstruction[He, 2017]

Network slimming [Liu2017]
Sparse structure selection [Huang, 2018]
Non-structured weight pruning [Han, 2015]

Source: Rethinking the Value of Network Pruning

Fine Tuned v.s. Training from Scratch

- Which one is better for small model
  - Pruning + fine-tuned v.s. training from scratch

- Factors
  - Pruning method
    - Structured v.s. non-structured
  - Training time

Source: Rethinking the Value of Network Pruning

Fine Tuned v.s. Training from Scratch

- Scratch-B: longer training time (50% cut, 2X training time)
- Scratch-E: same training time
- The Pruned Model : the list of predefined target models

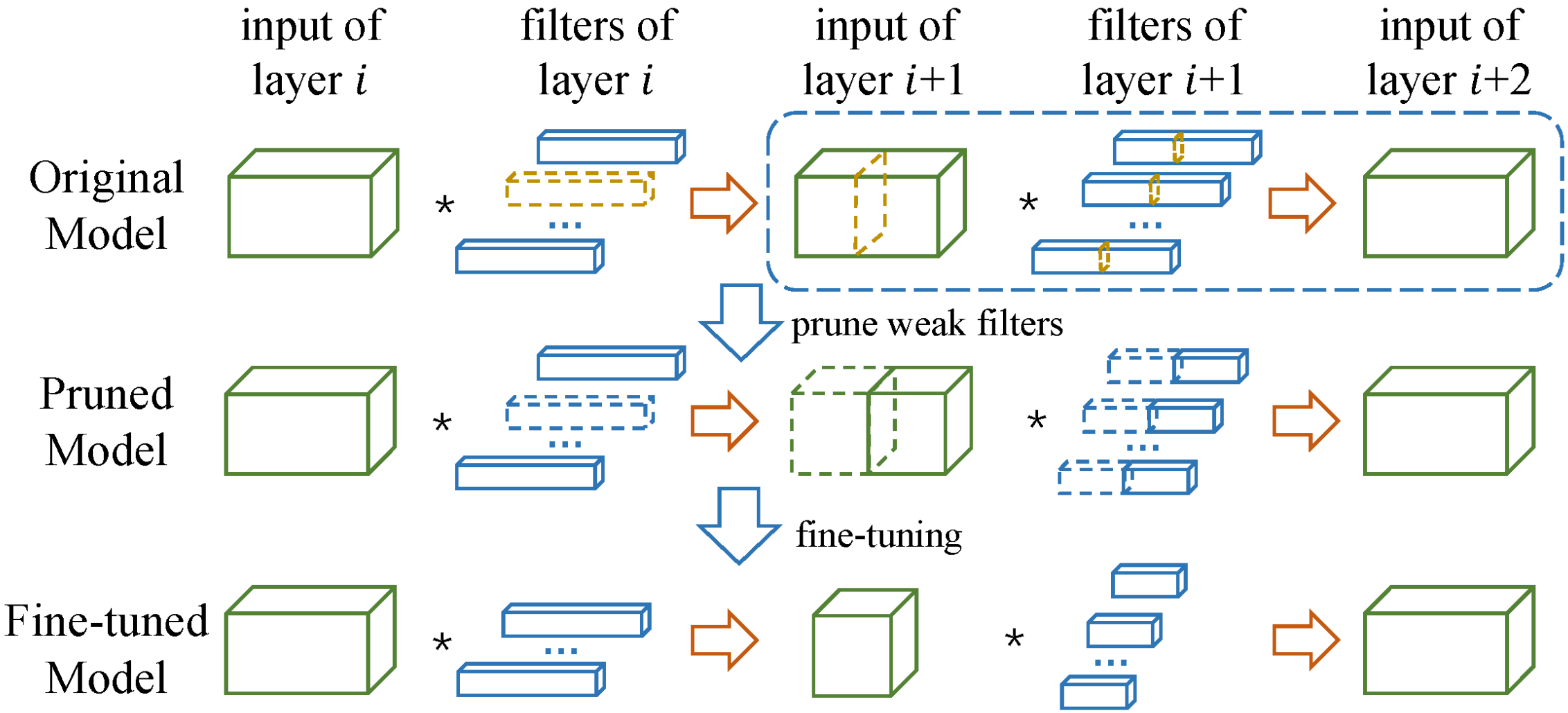
直接用小model，從頭訓練，拉長訓練時間，效果比較好

Dataset	Model	Unpruned	Pruned Model	Fine-tuned	Scratch-E	Scratch-B
CIFAR-10	VGG-16	93.63 (±0.16)	VGG-16-A	93.41 (±0.12)	93.62 (±0.11)	**93.78** (±0.15)
	ResNet-56	93.14 (±0.12)	ResNet-56-A	92.97 (±0.17)	92.96 (±0.26)	**93.09** (±0.14)
			ResNet-56-B	92.67 (±0.14)	92.54 (±0.19)	**93.05** (±0.18)
	ResNet-110	93.14 (±0.24)	ResNet-110-A	93.14 (±0.16)	**93.25** (±0.29)	93.22 (±0.22)
			ResNet-110-B	92.69 (±0.09)	92.89 (±0.43)	**93.60** (±0.25)
ImageNet	ResNet-34	73.31	ResNet-34-A	72.56	72.77	**73.03**
			ResNet-34-B	72.29	72.55	**72.91**

Table 1: Results (accuracy) for L_1-norm based channel pruning (Li et al., 2017). "Pruned Model" is the model pruned from the large model. Configurations of Model and Pruned Model are both from the original paper.

Source: Rethinking the Value of Network Pruning

ThiNet



用少部分的channel 去近似原來更多channel 的結果

both Scratch-E and Scratch-B can almost always achieve better performance than the fine-tuned model, often by a significant margin.

Dataset	Unpruned	Strategy	Pruned Model		Small model
ImageNet	VGG-16		VGG-Conv	VGG-GAP	VGG-Tiny
	71.03	Fine-tuned	−1.23	−3.67	−11.61
	71.51	Scratch-E	−2.75	−4.66	−14.36
		Scratch-B	**+0.21**	**−2.85**	**−11.58**
	ResNet-50		ResNet50-30%	ResNet50-50%	ResNet50-70%
	75.15	Fine-tuned	−6.72	−4.13	−3.10
	76.13	Scratch-E	−5.21	−2.82	−1.71
		Scratch-B	**−4.56**	**−2.23**	**−1.01**

Small model

Table 2: Results (accuracy) for ThiNet (Luo et al., 2017). Names such as "VGG-GAP" and "ResNet50-30%" are pruned models whose configurations are defined in Luo et al. (2017). To accommodate the effects of different frameworks between our implementation and the original paper's, we compare relative accuracy drop from the unpruned large model. For example, for the pruned model VGG-Conv, −1.23 is relative to 71.03 on the left, which is the reported accuracy of the unpruned large model VGG-16 in the original paper; −2.75 is relative to 71.51 on the left, which is VGG-16's accuracy in our implementation.

直接用小model，從頭訓練，拉長訓練時間，效果比較好

Source: Rethinking the Value of Network Pruning

Fine Tuned v.s. Training from Scratch

Dataset	Unpruned	Strategy	Pruned Model
ImageNet	VGG-16		VGG-16-5x
	71.03	Fine-tuned	−2.67
	71.51	Scratch-E	−3.46
		Scratch-B	**−0.51**
	ResNet-50		ResNet-50-2x
	75.51	Fine-tuned	−3.25
	76.13	Scratch-E	−1.55
		Scratch-B	**−1.07**

Table 3: Results (accuracy) for Regression based Feature Reconstruction (He et al., 2017b). Pruned models such as "VGG-16-5x" are defined in He et al. (2017b). Similar to Table 2, we compare relative accuracy drop from unpruned large models.

Regression based Feature Reconstruction (He et al., 2017b) prunes channels by minimizing the feature map reconstruction error of the next layer. Different from ThiNet (Luo et al., 2017), this optimization problem is solved by LASSO regression.

直接用小model，從頭訓練，拉長訓練時間，效果比較好

Source: Rethinking the Value of Network Pruning

Fine Tuned v.s. Training from Scratch

Dataset	Model	Unpruned	Prune Ratio	Fine-tuned	Scratch-E	Scratch-B
CIFAR-10	VGG-19	93.53 (±0.16)	70%	93.60 (±0.16)	93.30 (±0.11)	**93.81** (±0.14)
	PreResNet-164	95.04 (±0.16)	40%	94.77 (±0.12)	94.70 (±0.11)	**94.90** (±0.04)
			60%	94.23 (±0.21)	94.58 (±0.18)	**94.71** (±0.21)
	DenseNet-40	94.10 (±0.12)	40%	94.00 (±0.20)	93.68 (±0.18)	**94.06** (±0.12)
			60%	**93.87** (±0.13)	93.58 (±0.21)	93.85 (±0.25)
CIFAR-100	VGG-19	72.63 (±0.21)	50%	72.32 (±0.28)	71.94 (±0.17)	**73.08** (±0.22)
	PreResNet-164	76.80 (±0.19)	40%	76.22 (±0.20)	76.36 (±0.32)	**76.68** (±0.35)
			60%	74.17 (±0.33)	75.05 (± 0.08)	**75.73** (±0.29)
	DenseNet-40	73.82 (±0.34)	40%	**73.35** (±0.17)	73.24 (±0.29)	73.19 (±0.26)
			60%	72.46 (±0.22)	72.62 (±0.36)	**72.91** (±0.34)
ImageNet	VGG-11	70.84	50%	68.62	70.00	**71.18**

Table 4: Results (accuracy) for Network Slimming (Liu et al., 2017). "Prune ratio" stands for total percentage of channels that are pruned in the whole network. The same ratios for each model are used as the original paper.

直接用小model，從頭訓練，拉長訓練時間，效果比較好

Source: Rethinking the Value of Network Pruning

Fine Tuned v.s. Training from Scratch

Dataset	Model	Unpruned	Pruned Model	Pruned	Scratch-E	Scratch-B
ImageNet	ResNet-50	76.12	ResNet-41	75.44	75.61	**76.17**
			ResNet-32	74.18	73.77	**74.67**
			ResNet-26	71.82	72.55	**73.41**

Table 5: Results (accuracy) for residual block pruning using Sparse Structure Selection (Huang & Wang, 2018). In the original paper no fine-tuning is required so there is a "Pruned" column instead of "Fine-tuned" as before.

Sparse Structure Selection (Huang & Wang, 2018) also uses sparsified scaling factors to prune structures, and can be seen as a generalization of Network Slimming. Other than channels, pruning can be on residual blocks in ResNet or groups in ResNeXt.

Source: Rethinking the Value of Network Pruning

Dataset	Model	Unpruned	Prune Ratio	Fine-tuned	Scratch-E	Scratch-B
CIFAR-10	VGG-19	93.50 (±0.11)	30%	93.51 (±0.05)	**93.71** (±0.09)	93.31 (±0.26)
			80%	93.52 (±0.10)	**93.71** (±0.08)	93.64 (±0.09)
	PreResNet-110	95.04 (±0.15)	30%	95.06 (±0.05)	94.84 (±0.07)	**95.11** (±0.09)
			80%	**94.55** (±0.11)	93.76 (±0.10)	94.52 (±0.13)
	DenseNet-BC-100	95.24 (±0.17)	30%	95.21 (±0.17)	95.22 (±0.18)	**95.23** (±0.14)
			80%	95.04 (±0.15)	94.42 (±0.12)	**95.12** (±0.04)
CIFAR-100	VGG-19	71.70 (±0.31)	30%	71.96 (±0.36)	72.81 (±0.31)	**73.30** (±0.25)
			50%	71.85 (±0.30)	73.12 (±0.36)	**73.77** (±0.23)
	PreResNet-110	76.96 (±0.34)	30%	76.88 (±0.31)	76.36 (±0.26)	**76.96** (±0.31)
			50%	**76.60** (±0.36)	75.45 (±0.23)	76.42 (±0.39)
	DenseNet-BC-100	77.59 (±0.19)	30%	77.23 (±0.05)	77.58 (±0.25)	**77.97** (±0.31)
			50%	77.41 (±0.14)	77.65 (±0.09)	**77.80** (±0.23)
ImageNet	VGG-16	71.59	30%	73.68	72.75	**74.02**
			60%	**73.63**	71.50	73.42
	ResNet-50	76.15	30%	**76.06**	74.77	75.70
			60%	**76.09**	73.69	74.91

Table 6: Results (accuracy) for non-structured pruning (Han et al., 2015). "Prune Ratio" denotes the percentage of parameters pruned in the set of all convolutional weights.

Source: Rethinking the Value of Network Pruning

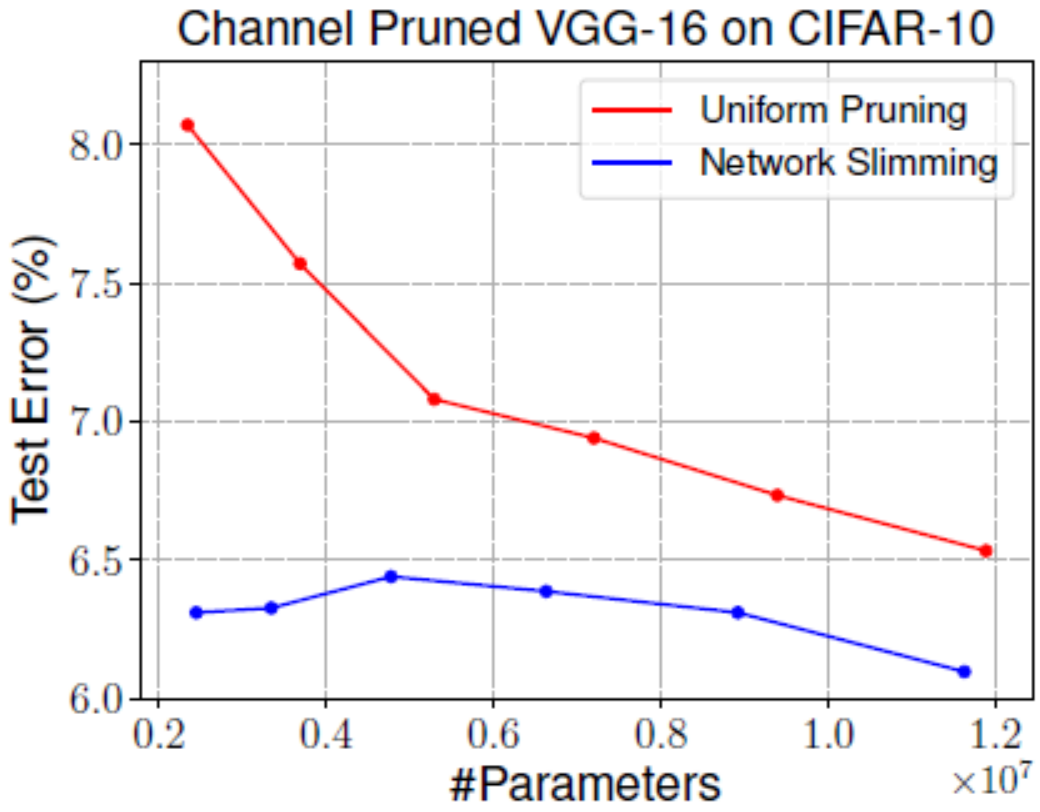
Fine Tuned v.s. Training from Scratch

- Which one is better for small model
  - Pruning + fine-tuned v.s. training from scratch
- Factors
  - training from scratch is better
    - Different pruning methods have the same trend
    - Structured v.s. non-structured
  - 2X training time is better

直接用小model，從頭訓練，拉長訓練時間，效果比較好

Source: Rethinking the Value of Network Pruning

Network Pruning as Architecture Search



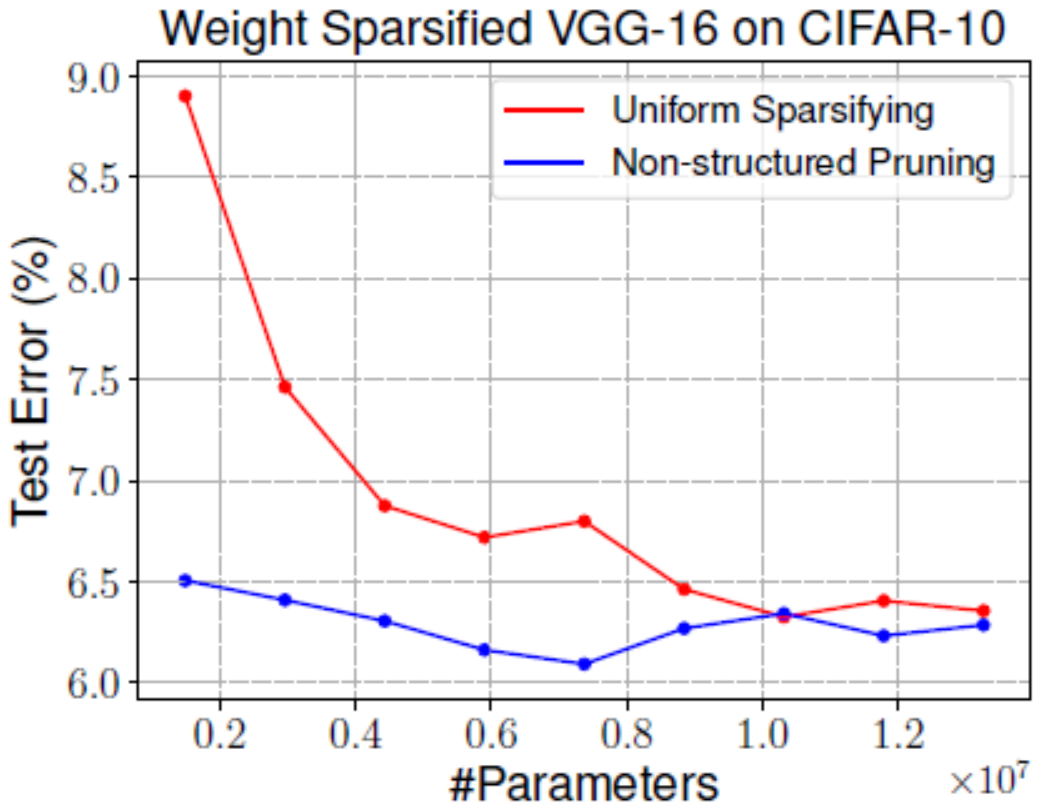


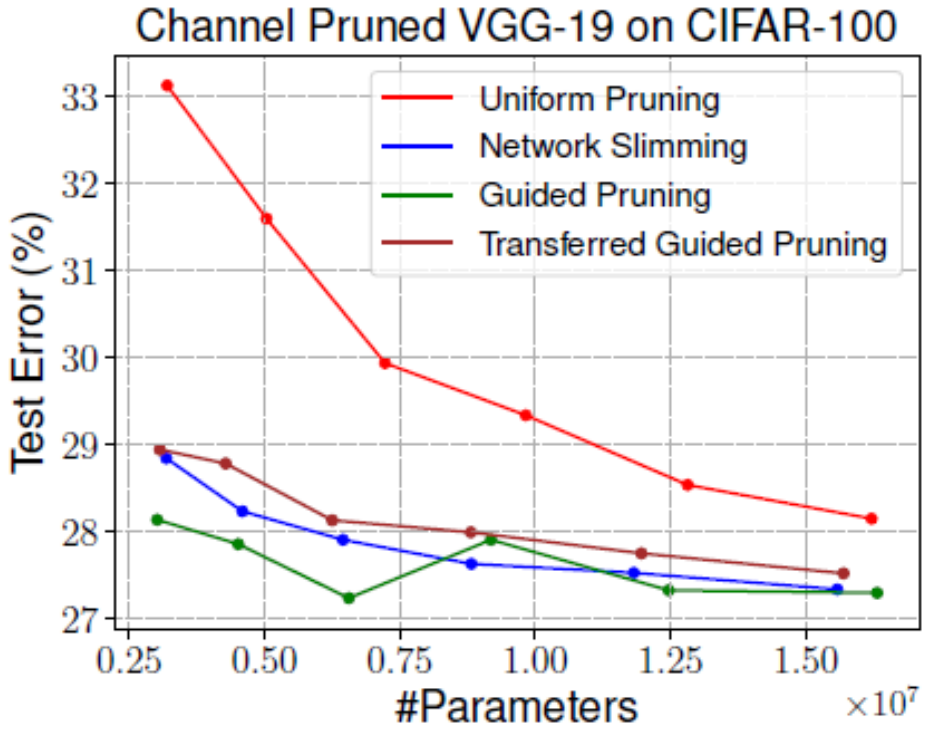
Figure 3: Pruned architectures obtained by different approaches, all *trained from scratch*, averaged over 5 runs. Architectures obtained by automatic pruning methods (*Left:* Network Slimming (Liu et al., 2017), *Right:* Non-stuctured weight pruning (Han et al., 2015)) have better parameter efficiency than uniformly pruning channels or sparsifying weights in the whole network.

不要均匀砍，效果比較好

Source: Rethinking the Value of Network Pruning

Generalizable Design Principles from Pruned Architectures

用小model學到的sparse架構，去設計新的架構(套用相同的avg channel 數等)看效果如何，並給其它架構和dataset使用



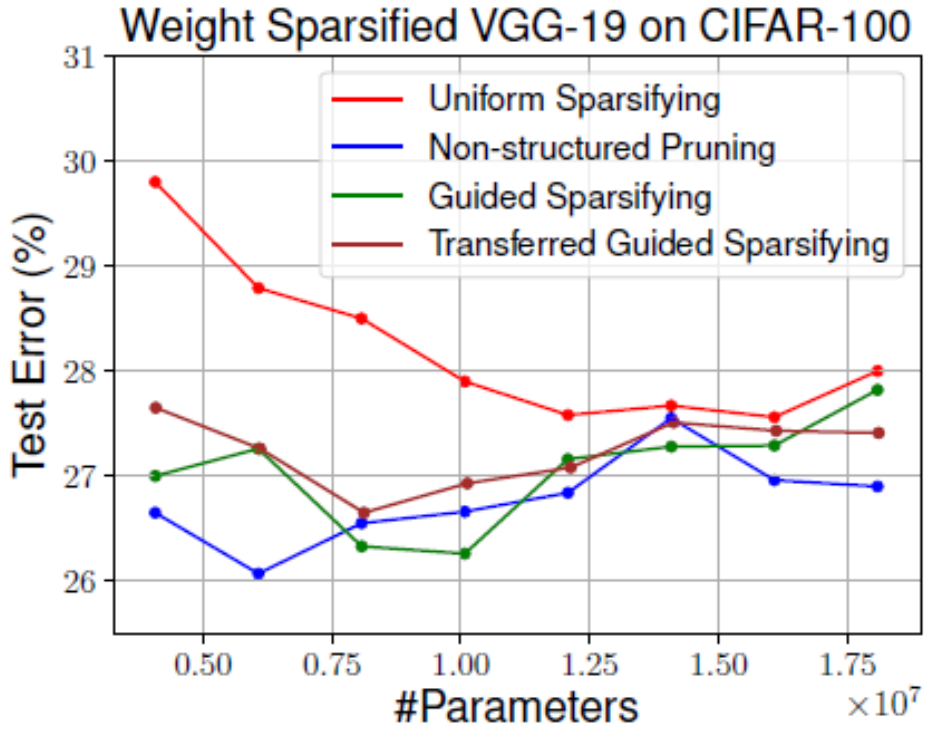


Figure 5: Pruned architectures obtained by different approaches, *all trained from scratch*, averaged over 5 runs. "Guided Pruning/Sparsifying" means using the average sparsity patterns in each layer stage to design the network; "Transferred Guided Pruning/Sparsifying" means using the sparsity patterns obtained by a pruned VGG-16 on CIFAR-10, to design the network for VGG-19 on CIFAR-100. Following the design guidelines provided by the pruned architectures, we achieve better parameter efficiency, even when the guidelines are transferred from another dataset and model.

Source: Rethinking the Value of Network Pruning

Pruning or Not Pruning (Small Models)

- Training predefined target models
  - Higher accuracy
  - Faster training
  - No need to implement pruning criteria and procedure
  - No need to tune hyper parameter in the pruning procedure
- Pruning works
  - When a pre-trained large model is already given and little or no training budget is available
  - need to obtain multiple models of different sizes, in this situation one can train a large model and then prune it by different ratios

Source: Rethinking the Value of Network Pruning

CONSTRAINED BASED PRUNING

MorphNet: Constrained based Pruning

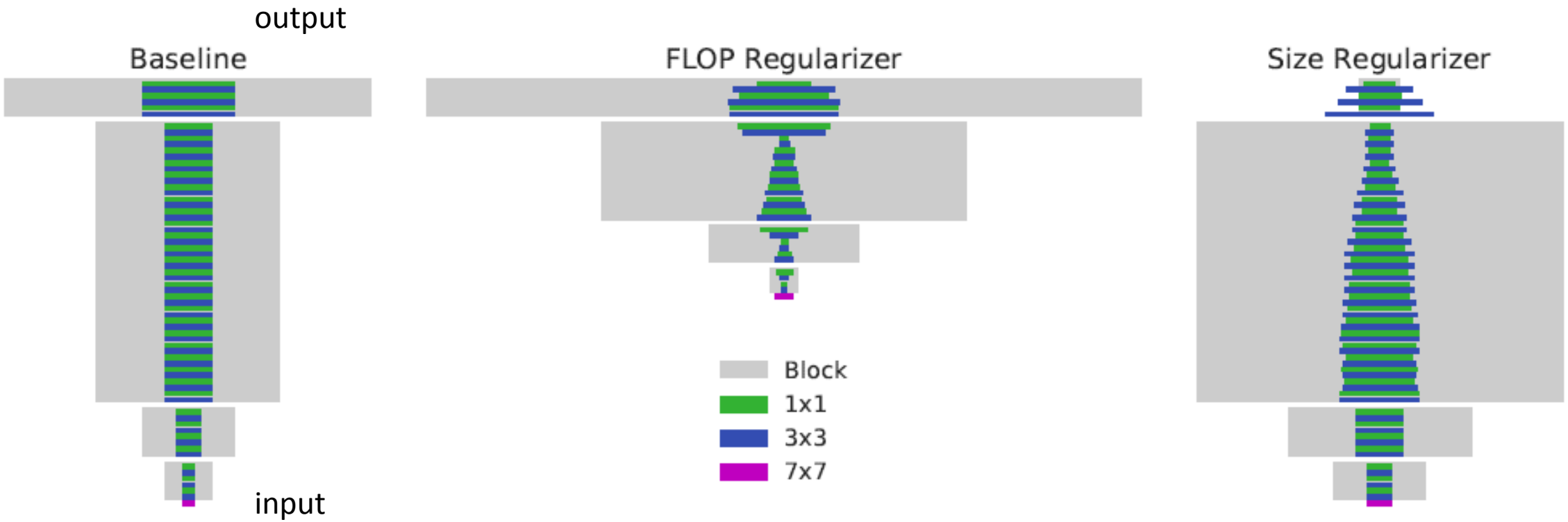


Figure 1. ResNet101 based models with similar performance (around 0.426 MAP on JFT, see Section 5). A structure obtained by shrinking ResNet101 uniformly by a $\omega = 0.5$ factor (left), and structures learned by MorphNet when targeting FLOPs (center) or model size (*i.e.*, number of parameters; right). Rectangle width is proportional to the number of channels in the layer and residual blocks are denoted in gray. 7×7, 3×3, and 1×1 convolutions are in purple, blue and green respectively. The purple bar at the bottom of each model is thus the input layer. Learned structures are markedly different from the human-designed model and from each other. The FLOP regularizer primarily prunes the early, compute-heavy layers. It notably learns to *remove whole layers* to further reduce computational burden. By contrast, the model size regularizer focuses on removal of 3×3 convolutions at the top layers as those are the most parameter-heavy.

MorphNet: only optimize over output widths

- iteratively alternating between a sparsifying regularizer and a uniform width multiplier

Algorithm 1 The MorphNet Algorithm

1: Train the network to find
$\theta^* = \underset{\theta}{\operatorname{argmin}}\{\mathcal{L}(\theta) + \lambda\mathcal{G}(\theta)\}$, for suitable λ.
2: Find the new widths $O'_{1:M}$ induced by θ^*.
3: Find the largests ω such that $\mathcal{F}(\omega \cdot O'_{1:M}) \leq \zeta$.
4: Repeat from Step 1 for as many times as desired, setting
$O^{\circ}_{1:M} = \omega \cdot O'_{1:M}$.
5: **return** $\omega \cdot O'_{1:M}$.

Shrinking:
a sparsifying regularizer
w < 1, uniform width multiplier

Expanind: w > 1
uniform width multiplier

MorphNet: Constraints

- FLOPs and model size are bilinear in the number of inputs and outputs of that layer

$$\mathcal{F}(\text{layer } L) = C(\underbrace{w_L, x_L}_{\text{input}}, \underbrace{y_L, z_L}_{\text{output}}, \underbrace{f_L, g_L}_{\text{Filter size}}) \cdot I_L O_L$$

  - FLOPS $C(w, x, y, z, f, g) = 2yzfg$
  - Model size $C(w, x, y, z, f, g) = fg$
  - After pruning, indicate which neuron is zero

$$\mathcal{F}(\text{layer } L) = C \sum_{i=0}^{I_L - 1} A_{L,i} \sum_{j=0}^{O_L - 1} B_{L,j}$$

  - Total constrained quality

$$\mathcal{F}(O_{1:M}) = \sum_{L=1}^{M+1} \mathcal{F}(\text{layer } L)$$

A, B: number of alived channels in a layer

MorphNet: Regularization

- Similar to network slimming, use γ in BN for channel pruning

$$\mathcal{G}(\theta) = \sum_{L=1}^{M+1} \mathcal{G}(\theta, \text{layer } L).$$

$$\mathcal{G}(\theta, \text{layer } L) = C \sum_{i=0}^{I_L-1} |\gamma_{L-1,i}| \sum_{j=0}^{O_L-1} B_{L,j} + C \sum_{i=0}^{I_L-1} A_{L,i} \sum_{j=0}^{O_L-1} |\gamma_{L,j}|,$$

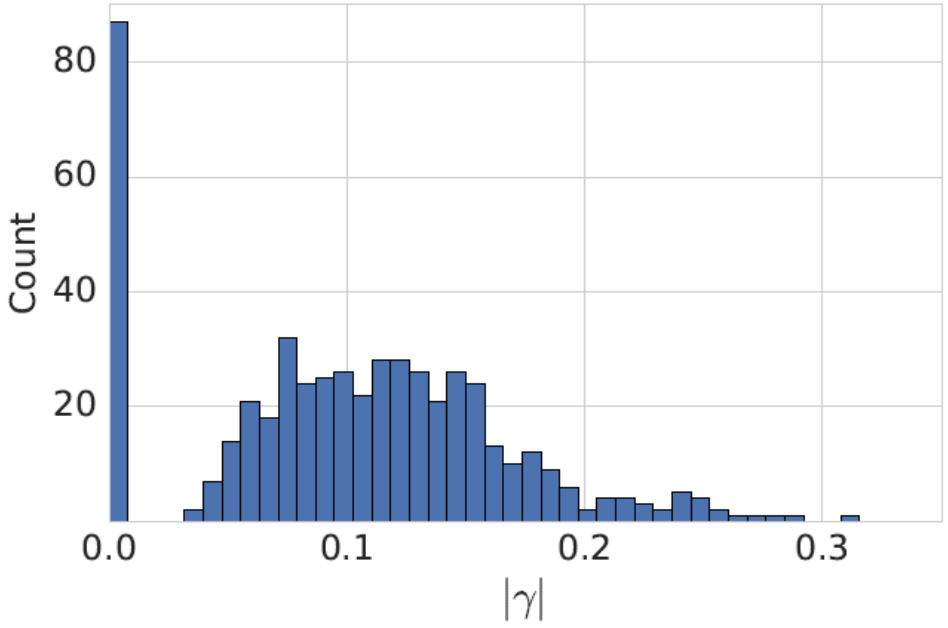


Figure 2. A histogram of γ for one of the ResNet101 bottleneck layers when trained with a FLOP regularizer. Some of the $|\gamma|$'s are zeroed out, and are separated by a clear gap from the nonzero $|\gamma|$'s.

MorphNet: Preserving Network Topology

- Input of residual block is the sum of multiple layers
  - Separated regularization results in topology change due to different sparsity in different layers
  - Group all neurons tied in skip connection via a Group LASSO
    - j-th output of L1 will be grouped with the j-th output of L2

MorphNet

- Inception V2 on ImageNet

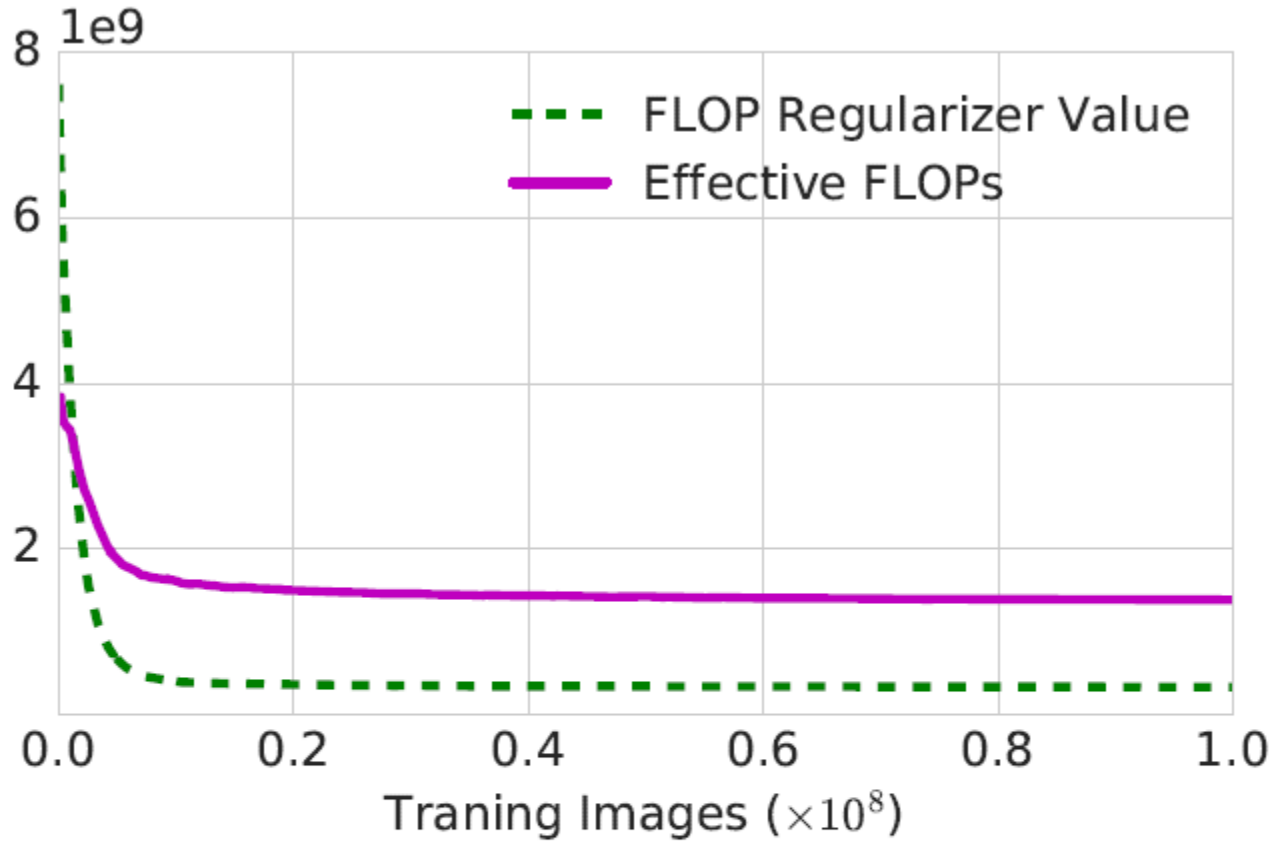
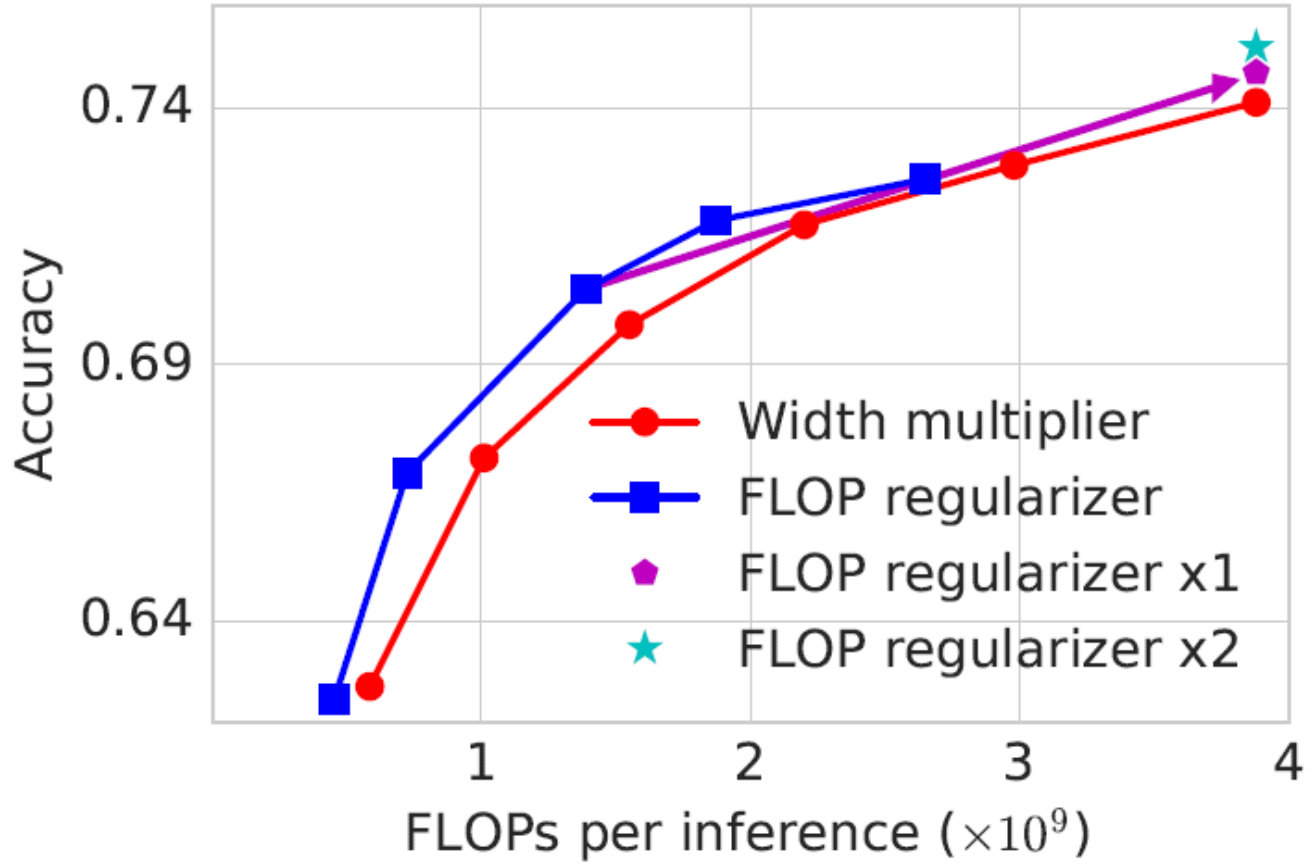


Figure 3. Rapid convergence of of the FLOP regularization (green, dashed) and projected number of FLOPs (purple) for ImageNet trained with a FLOP regularizer strength of $\lambda = 1.3 \cdot 10^{-9}$. The projected number of FLOPs is computed by assuming all $|\gamma| < 0.01$ are zeroed-out.



Iteration	ω	Dropout	Weights	Accuracy
0	NA	0	$1.12 \cdot 10^7$	74.1%
1	1.69	10%	$1.61 \cdot 10^7$	74.7%
2	1.57	20%	$1.55 \cdot 10^7$	75.2%

evaluation accuracy for various downsized
V2 using both a naïve width multiplier
arsifying FLOP regularizer (blue squares).
sult of re-expanding one of the networks
regularizer to match the FLOP cost of the
original network (pentagon point). A further increase in accuracy is achieved by performing the sparsifying and expanding process a second time (star point).

MorphNet: Improved Performance at No Cost

Network	Baseline	MorphNet	Relative Gain
Inception V2	74.1	75.2	+1.5%
MobileNet 50%	57.1	58.1	+1.78%
MobileNet 25%	44.8	45.9	+2.58%
ResNet101	0.477	0.487	+2.1%
AudioResNet	0.182	0.186	+2.18%

Table 2. The result of applying MorphNet to a variety of datasets and model architectures while maintaining FLOP cost.

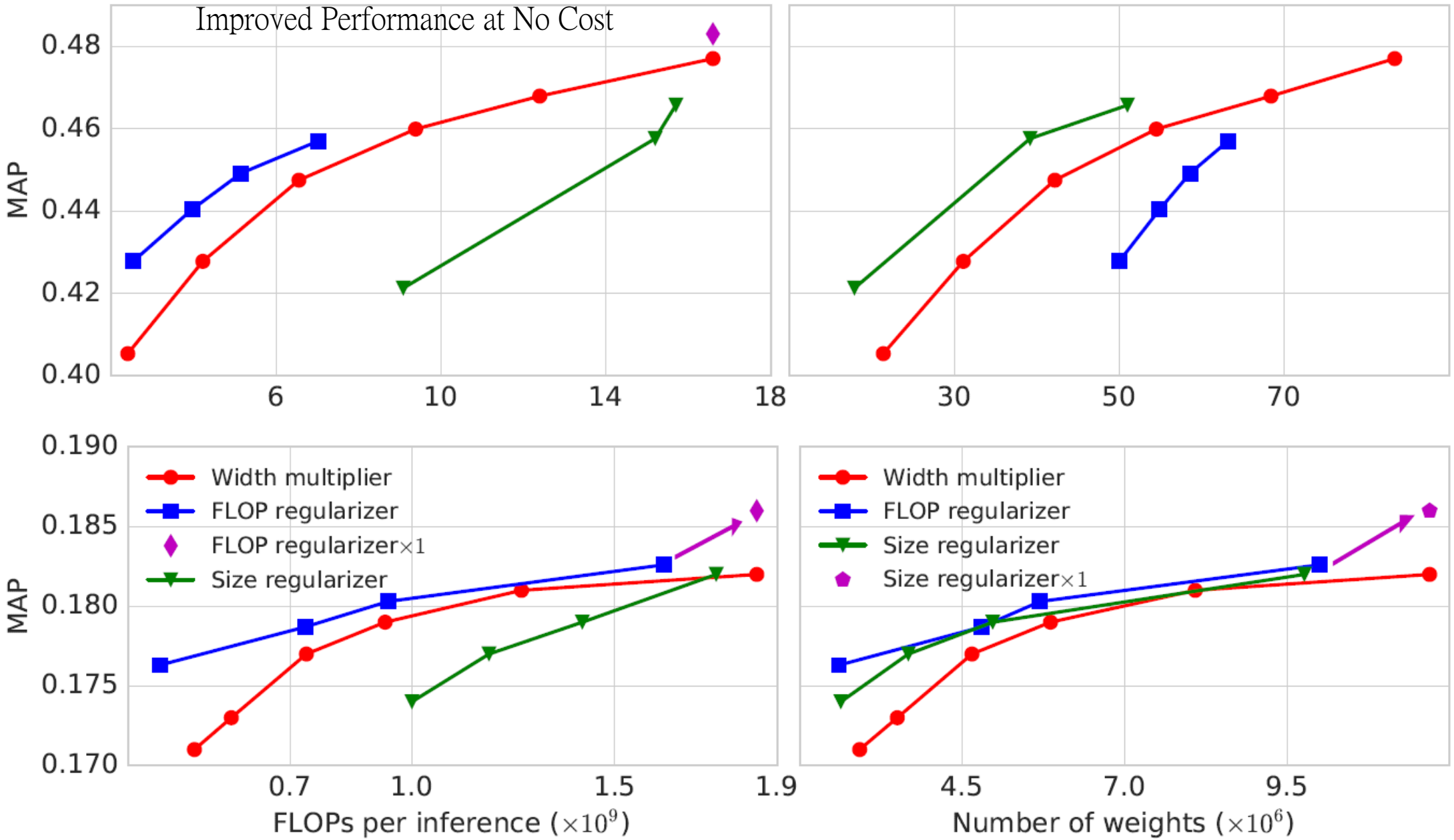
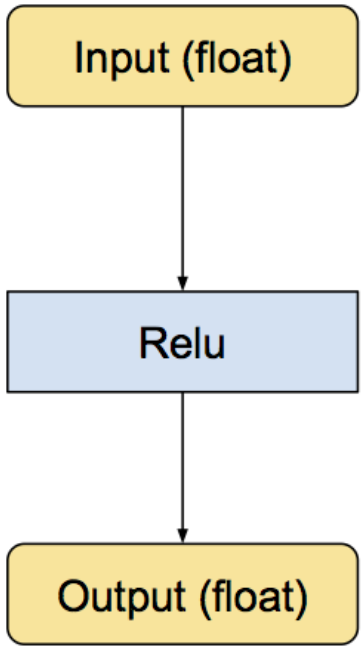


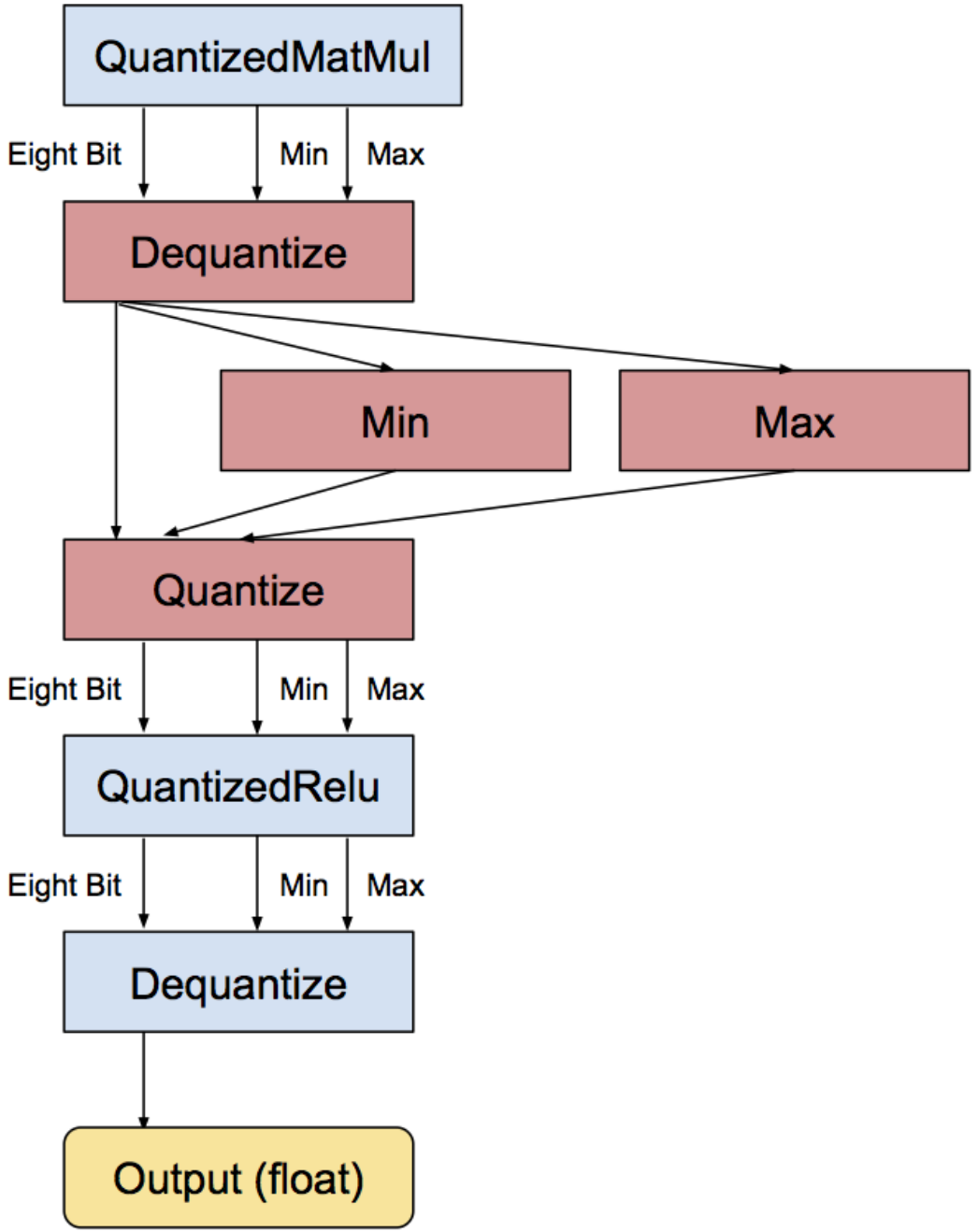
Figure 5. MAP vs. FLOPs (left) and MAP vs. model-size (right) curves on JFT (top) and AudioSet (bottom). The magenta points in the

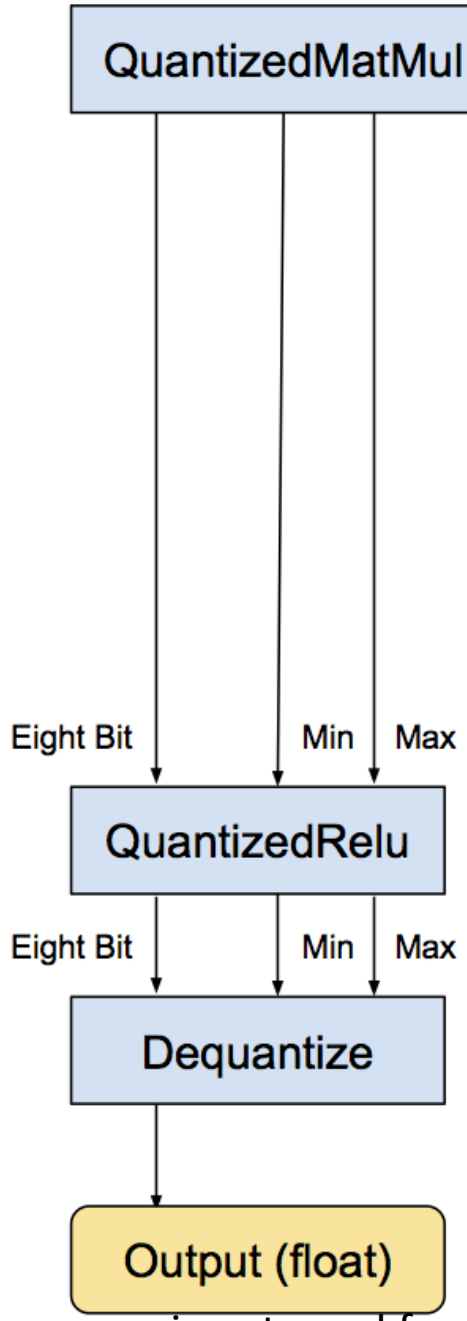
WEIGHT QUANTIZATION
32BIT浮點到8-BIT或4-BIT

Quantization

- 32-bit float to lower precision
- Quantization tool in Tensorflow (internal 8-bit quantization)
  - https://www.tensorflow.org/performance/quantization
  - Optimized matrix multiplications gemmlowp



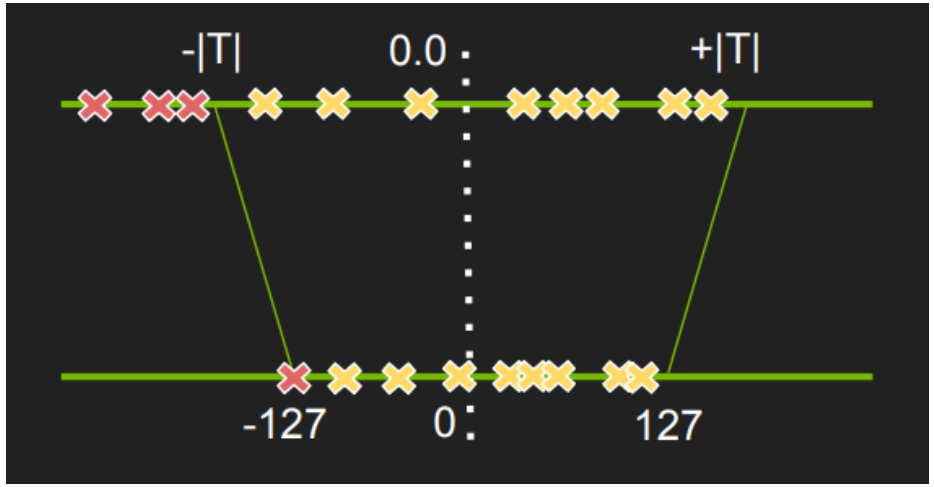




remove unnecessary conversions to and from float for consecutive seq

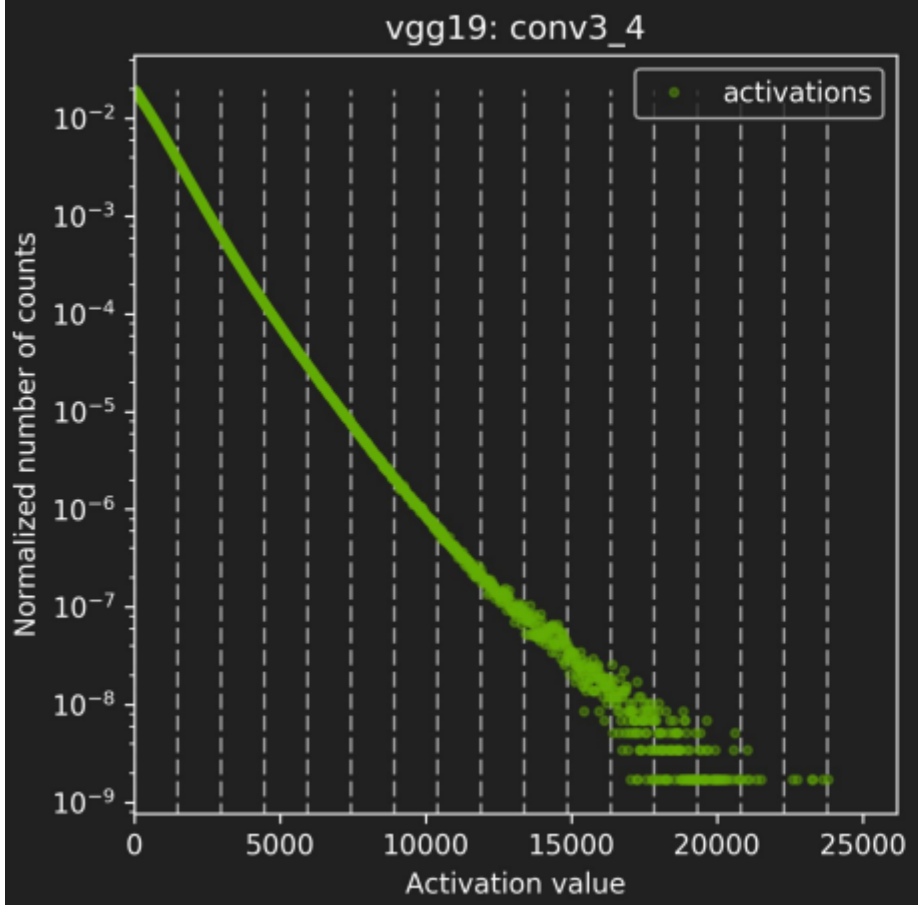
Nvidia TensorRT Quantization

- Concept
  - INT8 model encodes the same information as the original FP32 model
- Measure loss of information by KL divergence
  - P, Q - two discrete probability distributions.
  - `KL_divergence(P,Q):= SUM(P[i] * log(P[i] / Q[i] ), i)`
  - Intuition: KL divergence measures the amount of information lost when approximating a given encoding

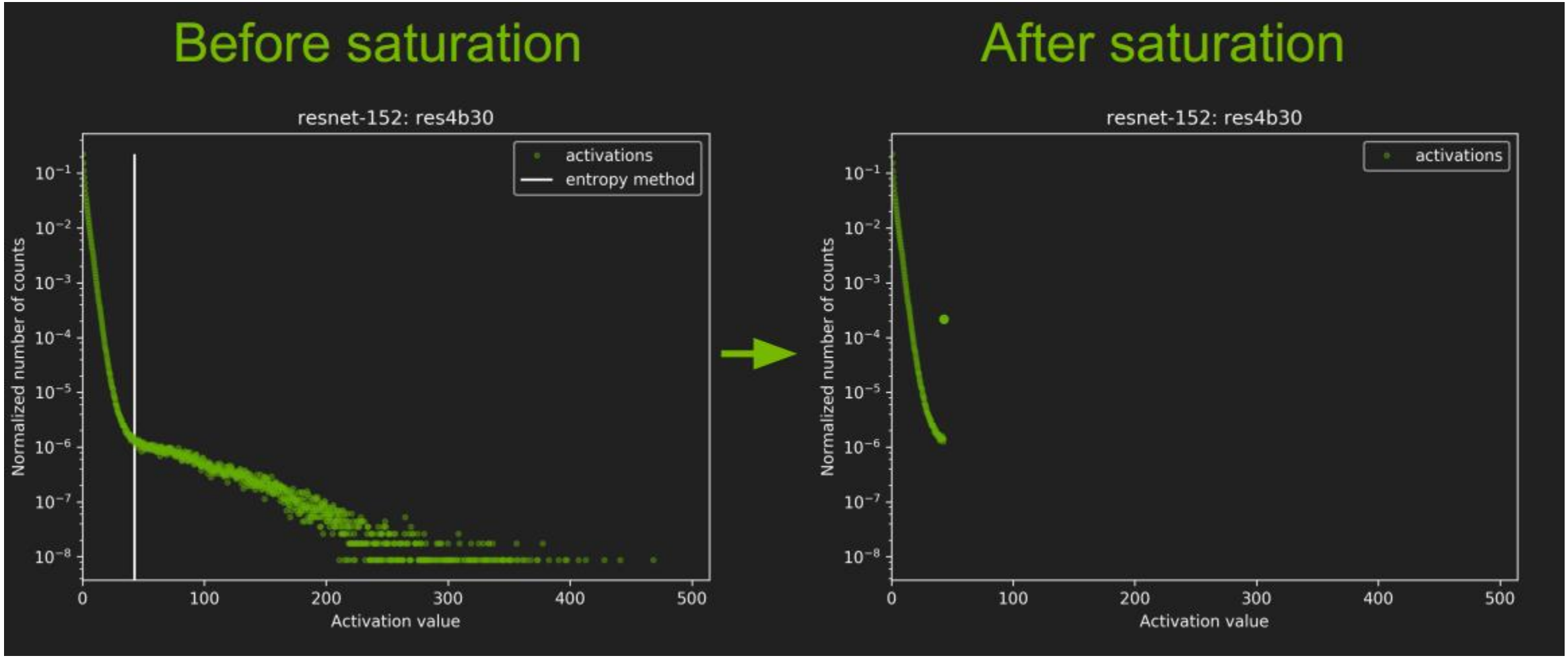


http://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf

Solution: Calibration

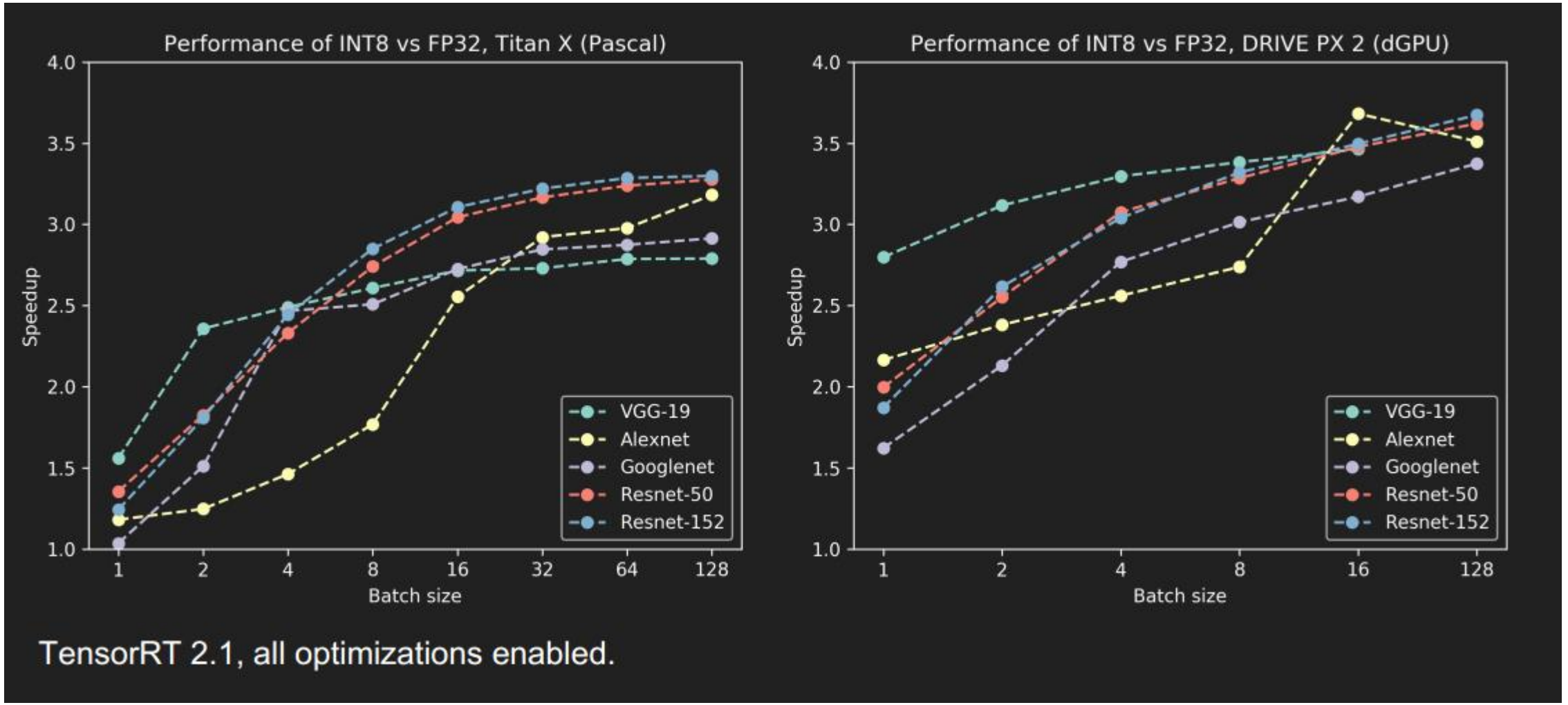


- Run FP32 inference on Calibration Dataset.
- For each Layer:
  - collect histograms of activations.
  - generate many quantized distributions with different saturation thresholds.
  - pick threshold which minimizes KL_divergence(ref_distr, quant_distr).
- Entire process takes a few minutes on a typical desktop workstation.

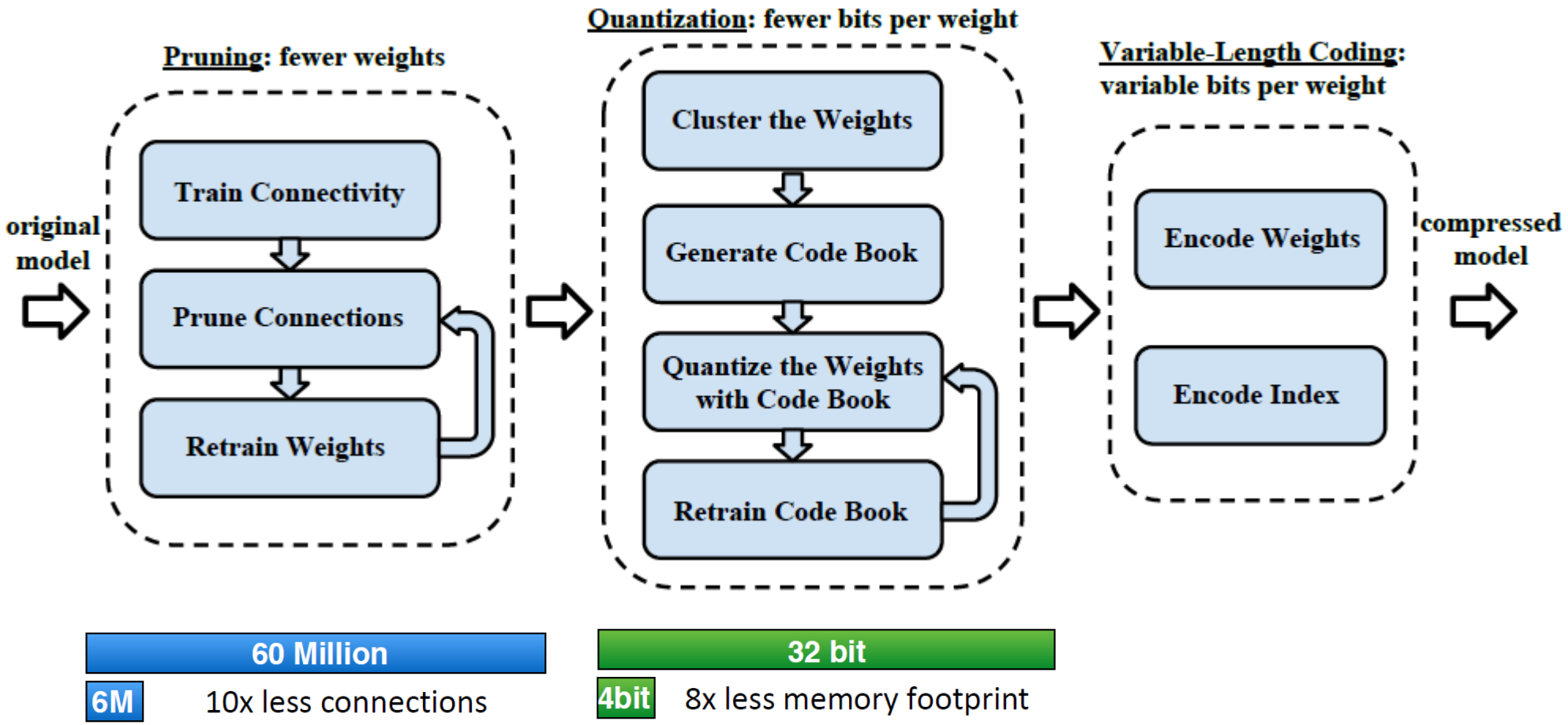

Before saturation
resnet-152: res4b30
activations
entropy method
Normalized number of counts
Activation value
After saturation
resnet-152: res4b30
activations
Normalized number of counts
Activation value

	FP32		INT8					
			Calibration using 5 batches		Calibration using 10 batches		Calibration using 50 batches	
NETWORK	Top1	Top5	Top1	Top5	Top1	Top5	Top1	Top5
Resnet-50	73.23%	91.18%	73.03%	91.15%	73.02%	91.06%	73.10%	91.06%
Resnet-101	74.39%	91.78%	74.52%	91.64%	74.38%	91.70%	74.40%	91.73%
Resnet-152	74.78%	91.82%	74.62%	91.82%	74.66%	91.82%	74.70%	91.78%
VGG-19	68.41%	88.78%	68.42%	88.69%	68.42%	88.67%	68.38%	88.70%
Googlenet	68.57%	88.83%	68.21%	88.67%	68.10%	88.58%	68.12%	88.64%
Alexnet	57.08%	80.06%	57.00%	79.98%	57.00%	79.98%	57.05%	80.06%
NETWORK	Top1	Top5	Diff Top1	Diff Top5	Diff Top1	Diff Top5	Diff Top1	Diff Top5
Resnet-50	73.23%	91.18%	0.20%	0.03%	0.22%	0.13%	0.13%	0.12%
Resnet-101	74.39%	91.78%	-0.13%	0.14%	0.01%	0.09%	-0.01%	0.06%
Resnet-152	74.78%	91.82%	0.15%	0.01%	0.11%	0.01%	0.08%	0.05%
VGG-19	68.41%	88.78%	-0.02%	0.09%	-0.01%	0.10%	0.03%	0.07%
Googlenet	68.57%	88.83%	0.36%	0.16%	0.46%	0.25%	0.45%	0.19%
Alexnet	57.08%	80.06%	0.08%	0.08%	0.08%	0.07%	0.03%	-0.01%

TensorRT 2.1, all optimizations enabled. ILSVRC2012 validation dataset, batch = 25 images.
Accuracy was measured on 500 batches which were not used for the calibration.

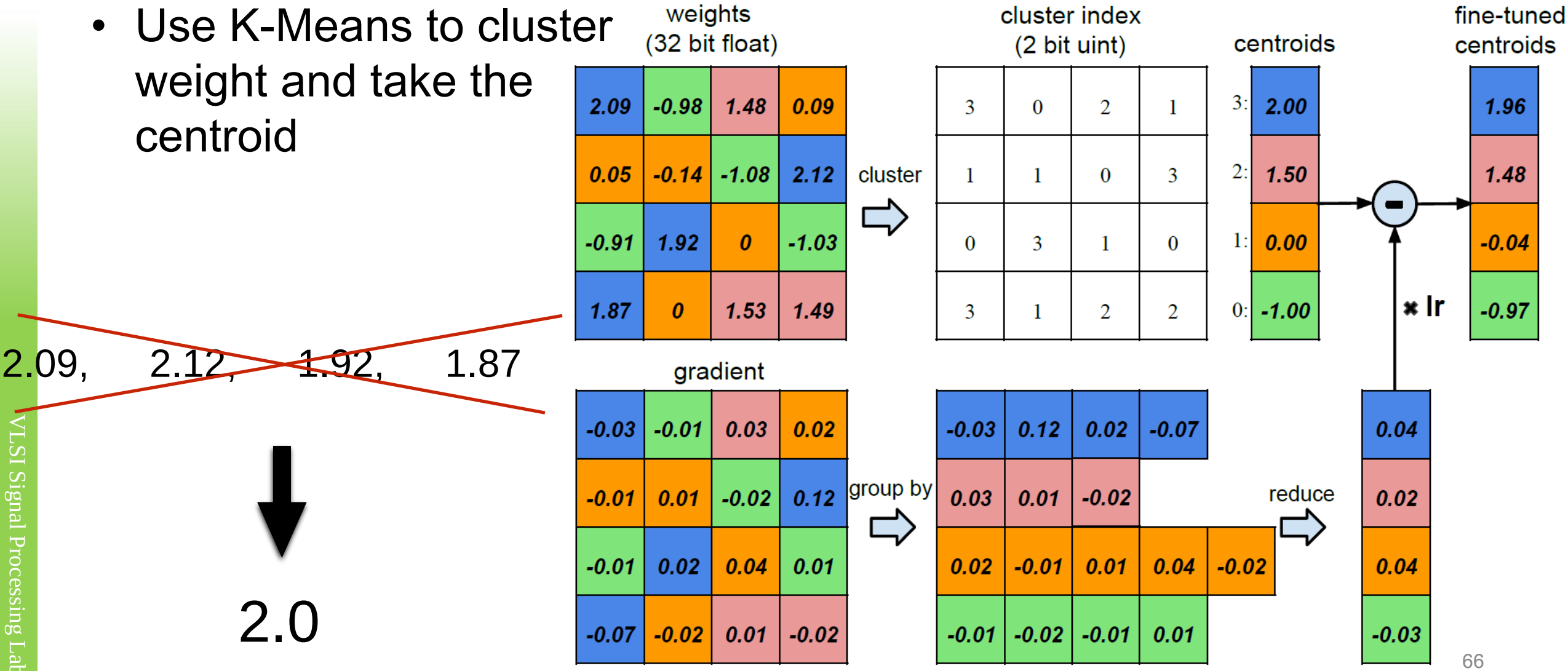

Performance of INT8 vs FP32, Titan X (Pascal)
Speedup
Batch size
VGG-19
Alexnet
Googlenet
Resnet-50
Resnet-152
Performance of INT8 vs FP32, DRIVE PX 2 (dGPU)
Speedup
Batch size
VGG-19
Alexnet
Googlenet
Resnet-50
Resnet-152
TensorRT 2.1, all optimizations enabled.

Deep Compression



Weight Sharing: One form of Quantization

- Use K-Means to cluster weight and take the centroid



~~2.09, 2.12, 1.92, 1.87~~

2.0

Trained Quantization Changes Weight Distribution

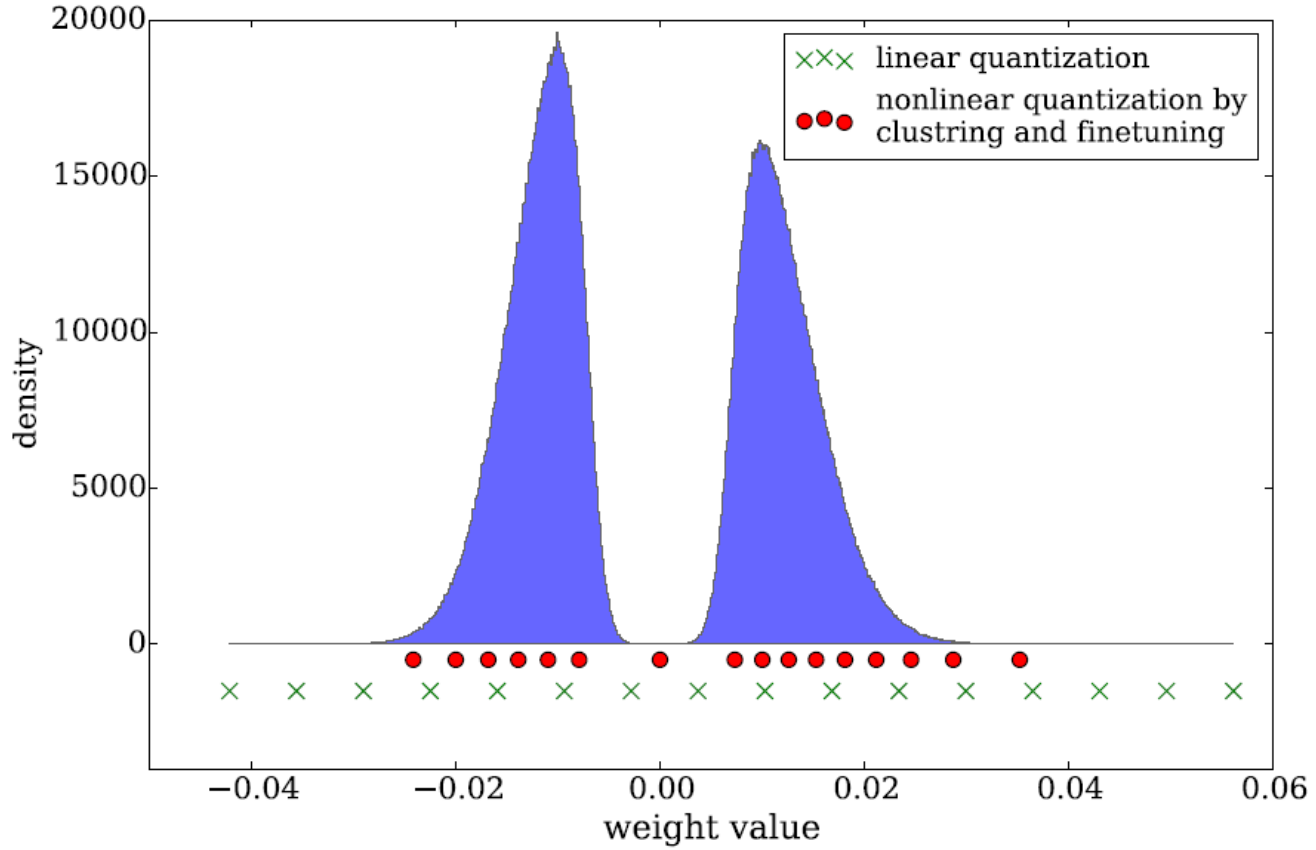


Figure 4.4: Distribution of weights and codebook before (green) and after fine-tuning (red).

Huffman Coding

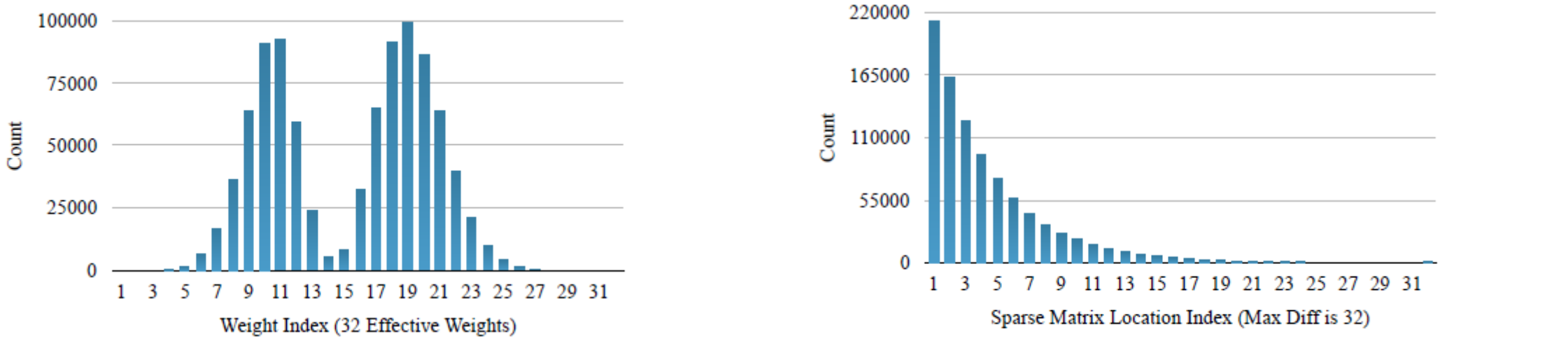
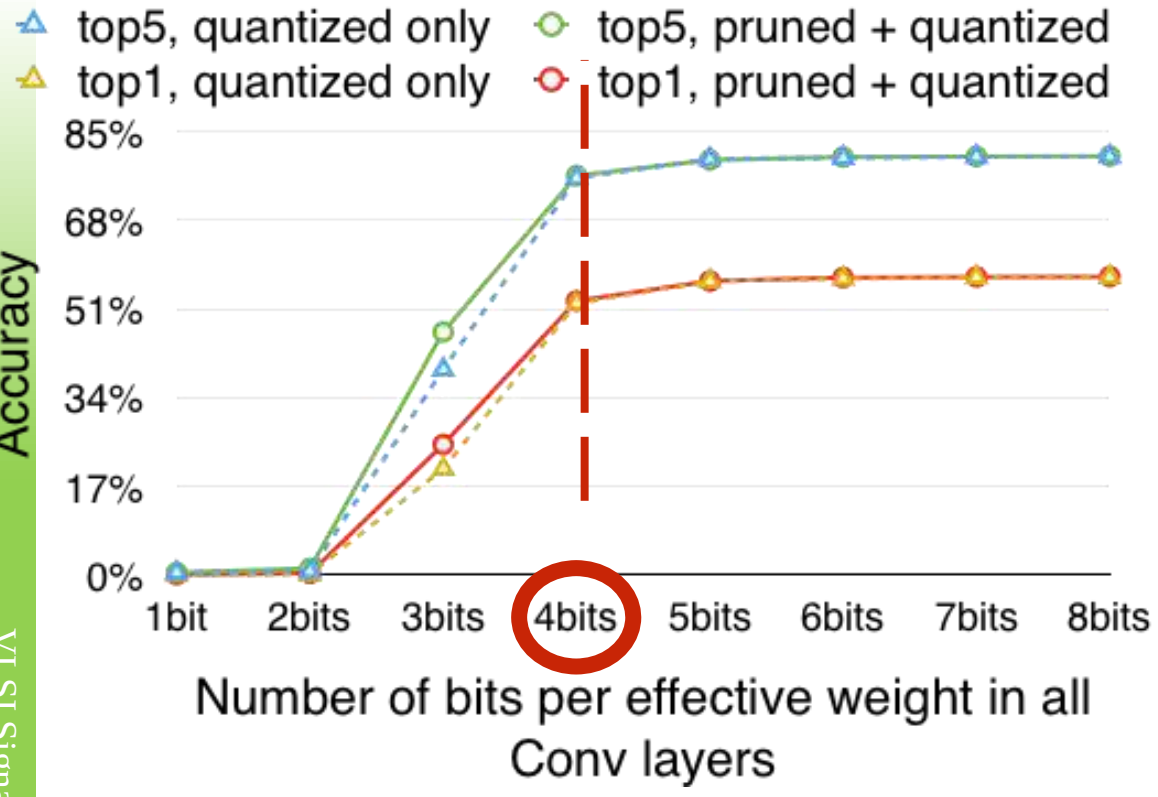
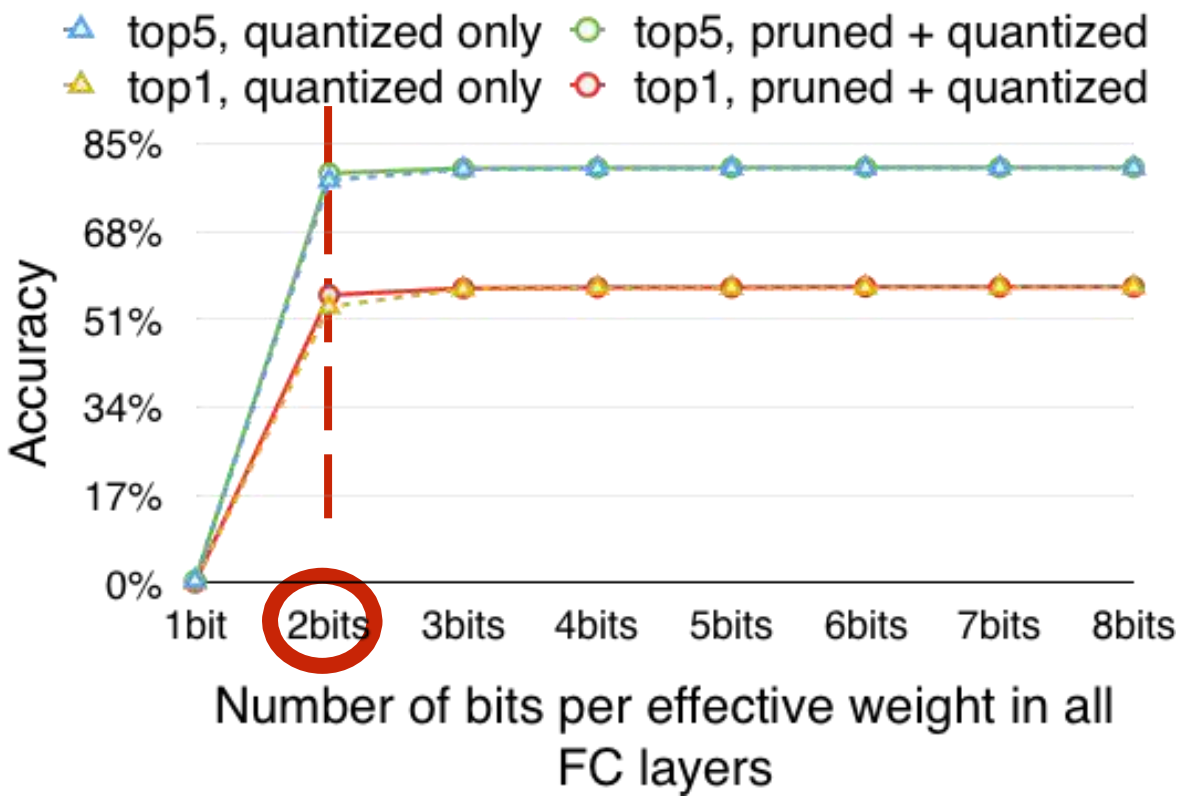


Figure 4.8: The non-uniform distribution for weight (Top) and index (Bottom) gives opportunity for variable-length coding.

- In-frequent weights: use more bits to represent
- Frequent weights: use fewer bits to represent

How Many Bits Are Needed?





Deep Compression: Results (Pruning + Weight Sharing + Huffman Coding)

Table 4.1: Deep Compression saves 17× to 49× parameter storage with no loss of accuracy.

Network	Top-1 Error	Top-5 Error	Model Size	Compress Rate
LeNet-300-100	1.64%	-	1070 KB	
LeNet-300-100 Compressed	1.58%	-	**27 KB**	40×
LeNet-5	0.80%	-	1720 KB	
LeNet-5 Compressed	0.74%	-	**44 KB**	**39**×
AlexNet	42.78%	19.73%	240 MB	
AlexNet Compressed	42.78%	19.70%	**6.9 MB**	**35**×
VGG-16	31.50%	11.32%	552 MB	
VGG-16 Compressed	31.17%	10.91%	**11.3 MB**	49×
Inception-V3	22.55%	6.44%	91 MB	
Inception-V3 Compressed	22.34%	6.33%	**4.2 MB**	**22**×
ResNet-50	23.85%	7.13%	97 MB	
ResNet-50 Compressed	23.85%	6.96%	**5.8 MB**	**17**×

Table 4.5: Compression statistics for AlexNet. P: pruning, Q: quantization, H: Huffman coding.

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

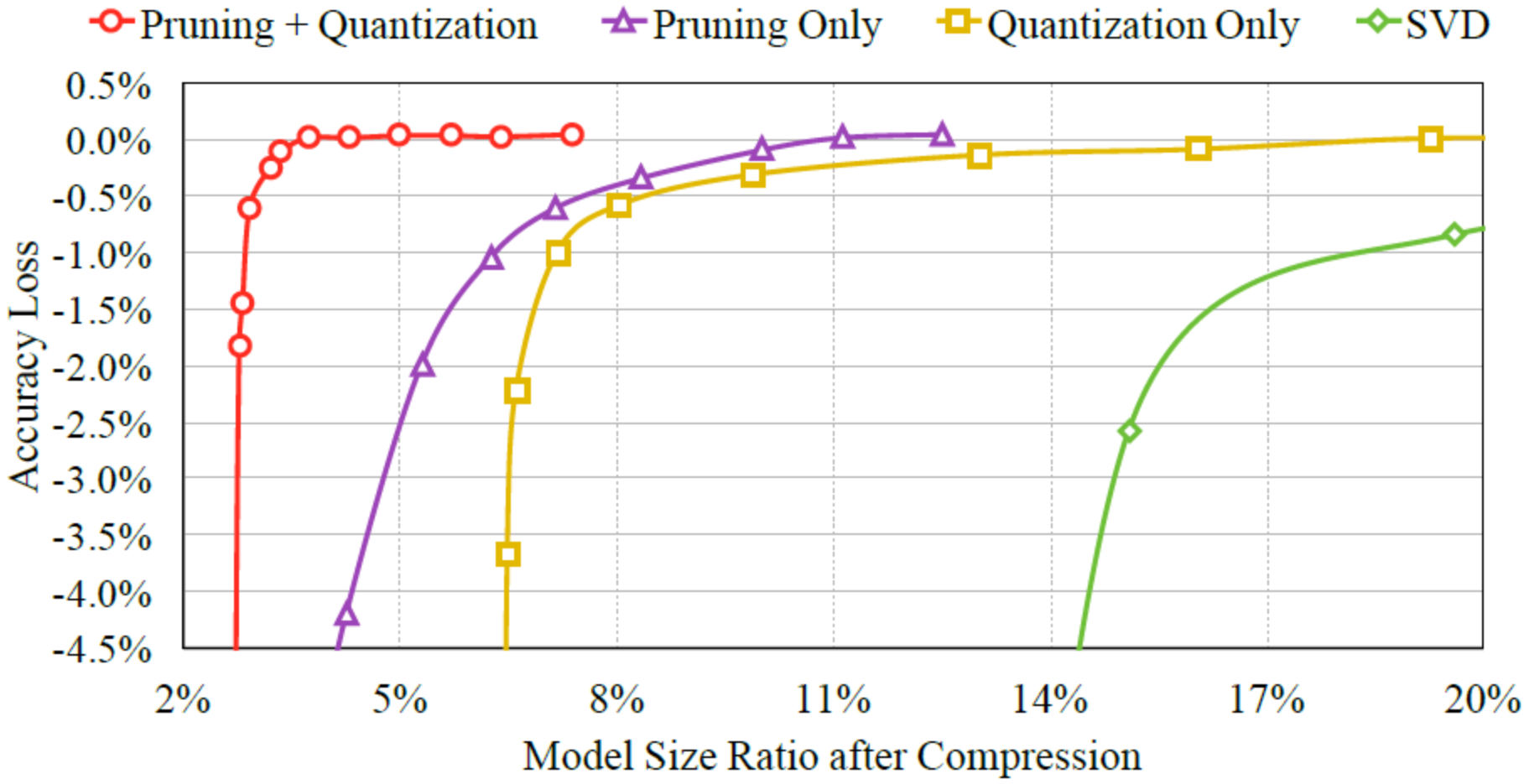


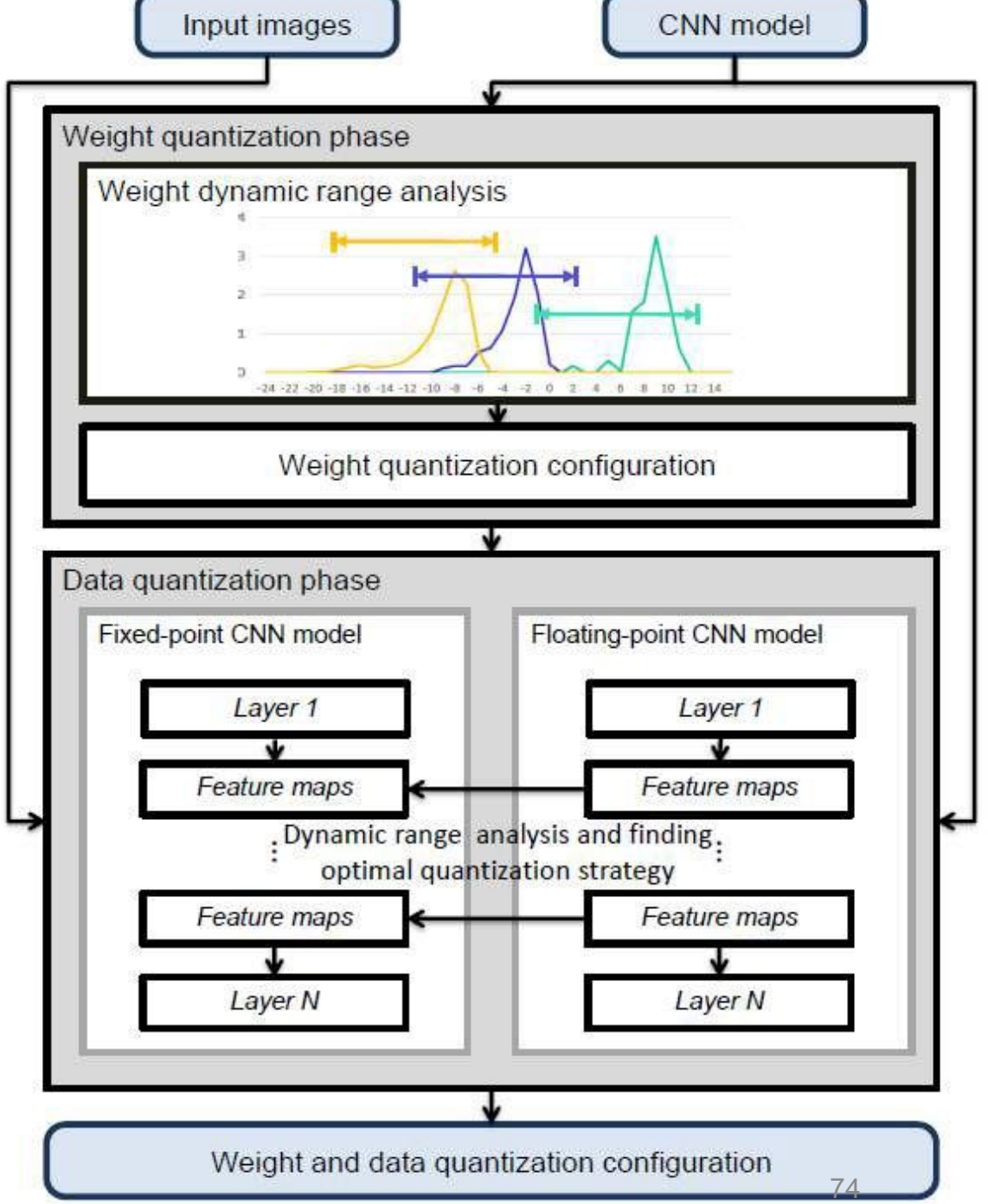
Figure 4.9: Accuracy vs. compression rates under different compression methods. Pruning and quantization works best when combined.

Meaning for 30x to 50X compression

- Complex DNNs can be put in mobile applications (<100MB total)
  - 1GB network (250M weights) becomes 20-30 MB
- Memory bandwidth reduced by 30-50X
  - Particular for FC layers in real application with no reuse
- Memory working set fits in on-chip SRAM
  - 5pj/word access v.s 640 ps/word

Quantize Weight and Activation

- Train with float
- Quantizing the weight and activation:
  - Gather the statistics for weight and activation
  - Choose proper radix point position
- Fine-tune in float format
- Convert to fixed-point format



Qiu et al. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network, FPGA'16

Dynamic precision data quantization results

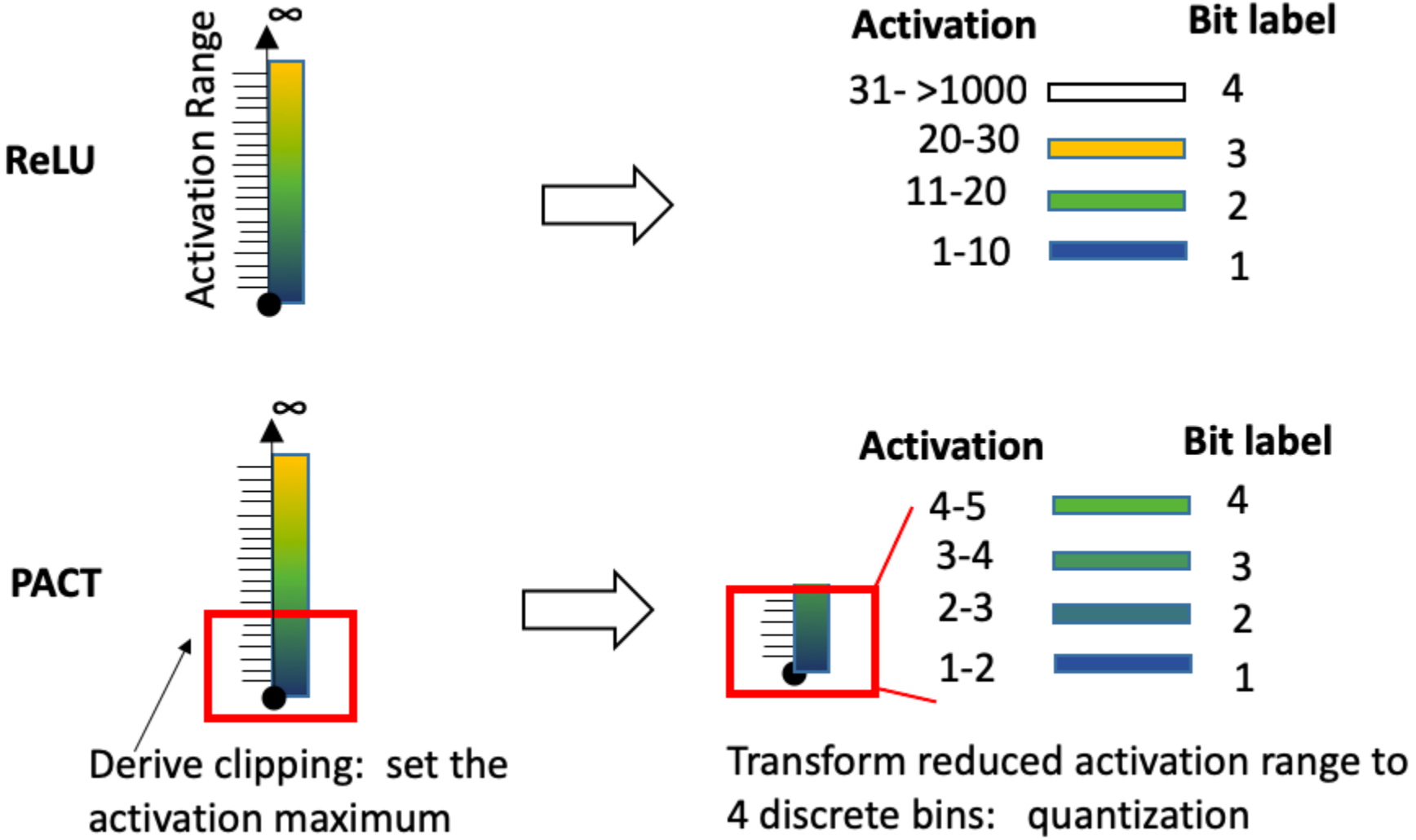
Network	VGG16						
Data Bits	Single-float	16	16	8	8	8	8
Weight Bits	Single-float	16	8	8	8	8	8 or 4
Data Precision	N/A	2^{-2}	2^{-2}	Impossible	$2^{-5}/2^{-1}$	Dynamic	Dynamic
Weight Precision	N/A	2^{-15}	2^{-7}	Impossible	2^{-7}	Dynamic	Dynamic
Top-1 Accuracy	68.1%	68.0%	53.0%	Impossible	28.2%	66.6%	67.0%
Top-5 Accuracy	88.0%	87.9%	76.6%	Impossible	49.7%	87.4%	87.6%

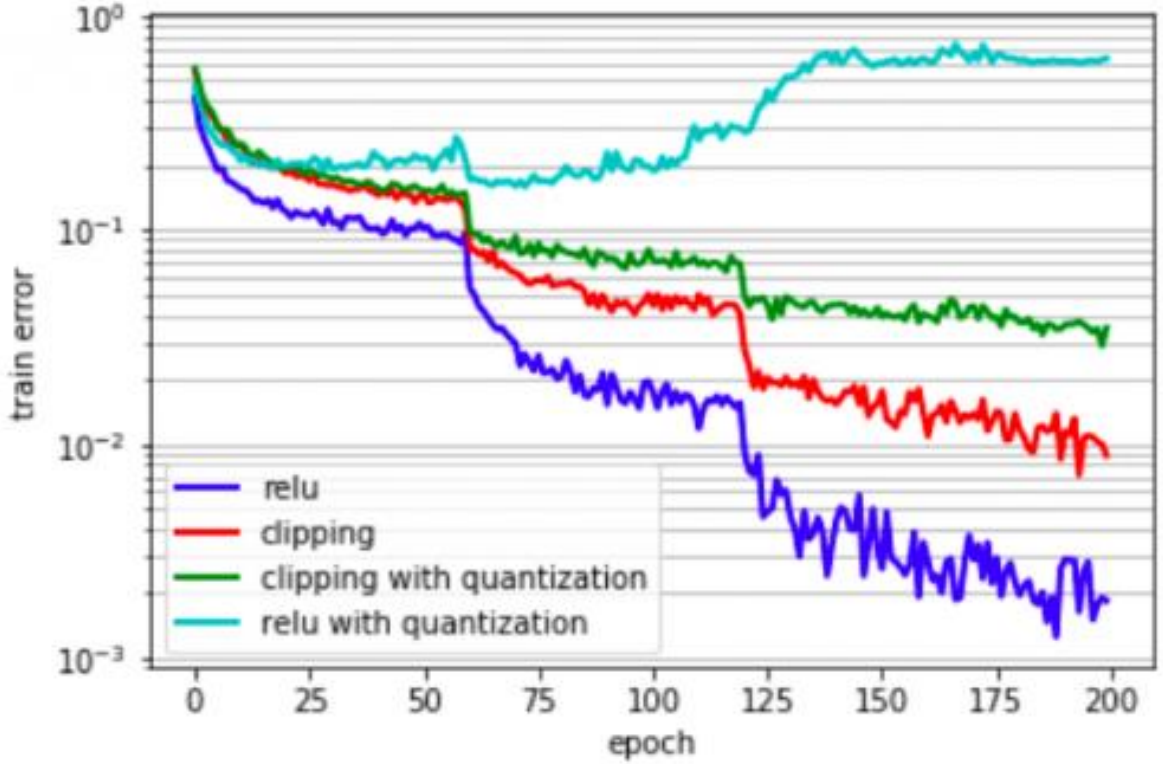
Network	CaffeNet			VGG16-SVD		
Data Bits	Single-float	16	8	Single-float	16	8
Weight Bits	Single-float	16	8	Single-float	16	8 or 4
Data Precision	N/A	Dynamic	Dynamic	N/A	Dynamic	Dynamic
Weight Precision	N/A	Dynamic	Dynamic	N/A	Dynamic	Dynamic
Top-1 Accuracy	53.9%	53.9%	53.0%	68.0%	64.6%	64.1%
Top-5 Accuracy	77.7%	77.1%	76.6%	88.0%	86.7%	86.3%

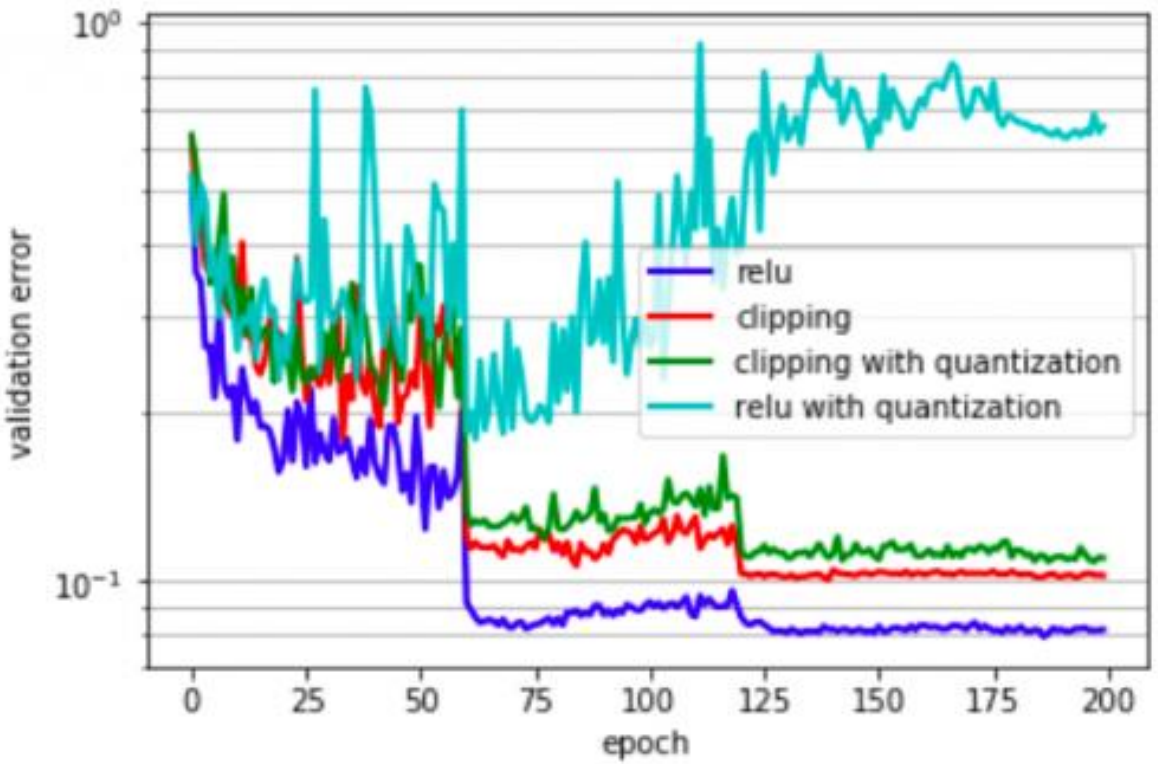
Qiu et al. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network, FPGA'16

Highly Accurate Deep Learning Inference with 2-bit Precision

- Current quantization limited by ReLU
  - relies on high dynamic range and precision to capture the unbounded continuous range of activation values
- PArameterized Clipping acTivation (PACT)
  - a maximum activation value is automatically derived for the unlimited range of activation values as part of the model training itself


ReLU
Activation Range
∞
Activation
Bit label
31- >1000
4
20-30
3
11-20
2
1-10
1
PACT
∞
Activation
Bit label
4-5
4
3-4
3
2-3
2
1-2
1
Derive clipping: set the activation maximum
Transform reduced activation range to 4 discrete bins: quantization

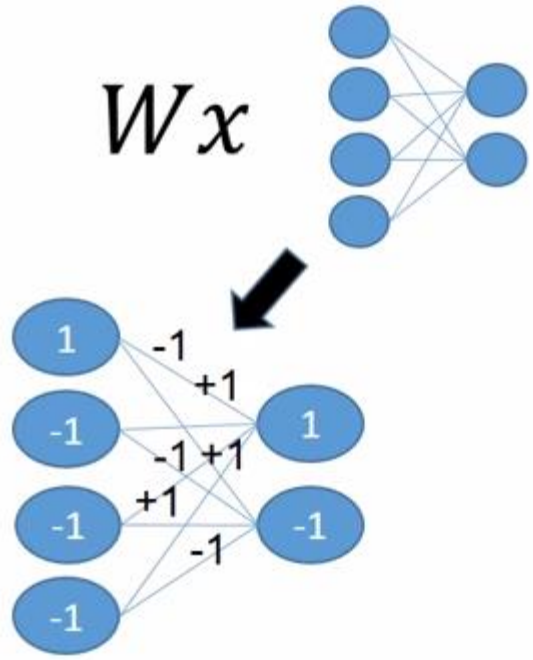

train error
epoch
relu
dipping
clipping with quantization
relu with quantization


validation error
epoch
relu
dipping
clipping with quantization
relu with quantization

BINARIZATION AND TERNARIZATION
量化極致版: 只用+/-1 或 +/-1, 0
FOR WEIGHT AND ACTIVATION

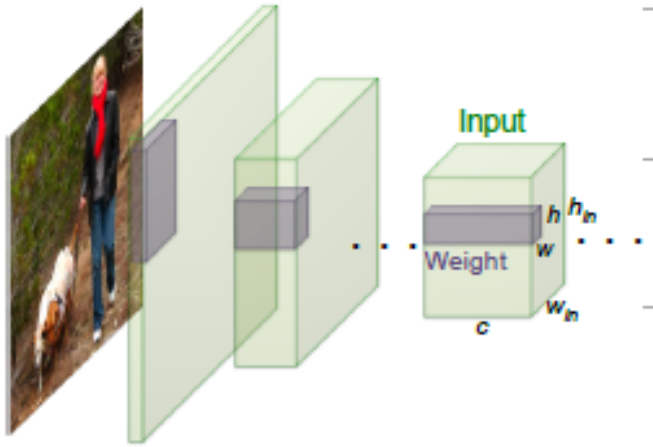
Motivation for Binary/Ternary Neural Networks

- Challenge
  - Run DNN on low power devices
  - Speedup DNNs at run-time
- Main computational bottleneck
  - Multiply-Accumulate (MAC) operation at matrix vector products
- Solution
  - Binary/ternary weights and activations to +/-1 or 0
  - Expensive MAC => cheap XNOR + PopCount



Binary/Ternary Neural Network

- Binary: +1 / -1
- Ternary: +1 / 0/ -1
- Binary/ternary neural network
  - Quantize weight or activation to binary or ternary value
  - Ternary value is better due to better representation



	Network Variations	Operations used in Convolution	Memory Saving (Inference)	Computation Saving (Inference)	Accuracy on ImageNet (AlexNet)
Standard Convolution	Real-Value Inputs: 0.11 -0.21 ... -0.34; -0.25 0.61 ... 0.52 — Real-Value Weights: 0.12 -1.2 ... 0.41; -0.2 0.5 ... 0.68	+ , − , ×	1x	1x	%56.7
Binary Weight	Real-Value Inputs: 0.11 -0.21 ... -0.34; -0.25 0.61 ... 0.52 — Binary Weights: 1 -1 ... 1; -1 1 ... 1	+ , −	~32x	~2x	%56.8
BinaryWeight Binary Input (XNOR-Net)	Binary Inputs: 1 -1 ... -1; -1 1 ... 1 — Binary Weights: 1 -1 ... 1; -1 1 ... 1	XNOR , bitcount	~32x	~58x	%44.2

Recent Results

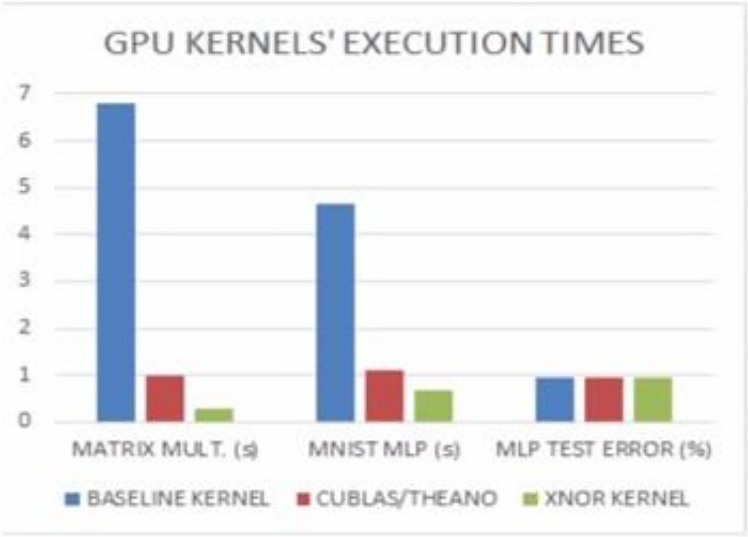
- Comparison on ImageNet with AlexNet
- For binary/ternary, the first and last layer are still full precision

Reduce Precision Method		Bitwidth		Accuracy loss vs. 32-bit float (%)
		Weights	Activations	
Dynamic Fixed Point	w/o fine-tuning [118]	8	10	0.4
	w/ fine-tuning [119]	8	8	0.6
Reduce weight	Ternary Weight Networks (TWN) [129]	2*	32 (float)	3.7
	Trained Ternary Quantization (TTQ) [130]	2*	32 (float)	0.6
	BinaryConnect [126]	1	32 (float)	19.2
	Binary Weight Network (BWN) [128]	1*	32 (float)	0.8
Reduce weight and activation	Quantized Neural Networks (QNN) [116]	1	2*	6.5
	Binarized Neural Networks (BNN) [127]	1	1	29.8
	XNOR-net [128]	1*	1	11
Non-linear Quantization	Incremental Network Quantization (INQ) [131]	5	32 (float)	-0.2
	LogNet [115]	5 (conv), 4 (fc)	4	3.2
	Deep Compression [114]	8 (conv), 4 (fc)	16	0
		4 (conv), 2 (fc)	16	2.6

Source: Efficient Processing of Deep Neural Networks: A Tutorial and Survey

Binarized Neural Network

- 32 x less memory.
- Speed-up GPU implementations of BNNs using a dedicated XNOR kernel:
  - 23 x faster than the baseline kernel
  - 3.4 x faster than cuBLAS.



- Much lower power possible

Operation	MUL	ADD
8bit Integer	0.2pJ	0.03pJ
32bit Integer	3.1pJ	0.1pJ
16bit Floating Point	1.1pJ	0.4pJ
32 bit Floating Point	3.7pJ	0.9pJ

(Horowitz, 2014)

BNN on ResNet (Open Source on MxNet)

- BMXNet

	Architecture	Test Accuracy (Binary/Full Precision)	Model Size (Binary/Full Precision)
MNIST	Lenet	0.97/0.99	206kB/4.6MB
CIFAR-10	ResNet-18	0.86/0.90	1.5MB/44.7MB

Table 1: Classification test accuracy of binary and full precision models trained on MNIST and CIFAR-10 dataset. No pre-training or data augmentation was used.

Listing 1: LeNet

```
def get_lenet():
 data = mx.symbol.Variable('data')
 # first conv layer
 conv1 = mx.sym.Convolution(...)
 tanh1 = mx.sym.Activation(...)
 pool1 = mx.sym.Pooling(...)
 bn1 = mx.sym.BatchNorm(...)
 # second conv layer
 conv2 = mx.sym.Convolution(...)
 bn2 = mx.sym.BatchNorm(...)
 tanh2 = mx.sym.Activation(...)
 pool2 = mx.sym.Pooling(...)
 # first fullc layer
 flatten = mx.sym.Flatten(...)
 fc1 = mx.symbol.FullyConnected(..)
 bn3 = mx.sym.BatchNorm(...)
 tanh3 = mx.sym.Activation(...)
 # second fullc
 fc2 = mx.sym.FullyConnected(...)
 # softmax loss
 lenet = mx.sym.SoftmaxOutput(...)
 return lenet
```

Listing 2: Binary LeNet

```
def get_binary_lenet():
 data = mx.symbol.Variable('data')
 # first conv layer
 conv1 = mx.sym.Convolution(...)
 tanh1 = mx.sym.Activation(...)
 pool1 = mx.sym.Pooling(...)
 bn1 = mx.sym.BatchNorm(...)
 # second conv layer
 ba1 = mx.sym.QActivation(...)
 conv2 = mx.sym.QConvolution(...)
 bn2 = mx.sym.BatchNorm(...)
 pool2 = mx.sym.Pooling(...)
 # first fullc layer
 flatten = mx.sym.Flatten(...)
 ba2 = mx.symbol.QActivation(..)
 fc1 = mx.symbol.QFullyConnected(..)
 bn3 = mx.sym.BatchNorm(...)
 tanh3 = mx.sym.Activation(...)
 # second fullc
 fc2 = mx.sym.FullyConnected(...)
 # softmax loss
 lenet = mx.sym.SoftmaxOutput(...)
 return lenet
```

Listing 3: Baseline xnor GEMM Kernel

```
void xnor_gemm_baseline_no_omp(int M, int N, int K,
                               BINARY_WORD *A, int lda,
                               BINARY_WORD *B, int ldb,
                               float *C, int ldc){
  for (int m = 0; m < M; ++m) {
    for (int k = 0; k < K; k++) {
      BINARY_WORD A_PART = A[m*lda+k];
      for (int n = 0; n < N; ++n) {
        C[m*ldc+n] += __builtin_popcountl(~(A_PART ^ B[k*ldb+n]));
      }
    }
  }
}
```

A single assembly command to support popcount in x86 and ARM v7
Leverage processor cache hierarchies by blocking and packing the data, use unrolling techniques and OpenMP for parallelization

Gated XNOR Network

- Binary gradient as well

COMPARISONS WITH STATE-OF-THE-ART ALGORITHMS AND NETWORKS.

Methods	Datasets		
	MNIST	CIFAR10	SVHN
BNNs [19]	98.60%	89.85%	97.20%
TWNs [17]	99.35%	92.56%	N.A
BWNs [16]	98.82%	91.73%	97.70%
BWNs [17]	99.05%	90.18%	N.A
Full-precision NNs [17]	99.41%	92.88%	N.A
GXNOR Networks	**99.32%**	**92.50%**	**97.37%**

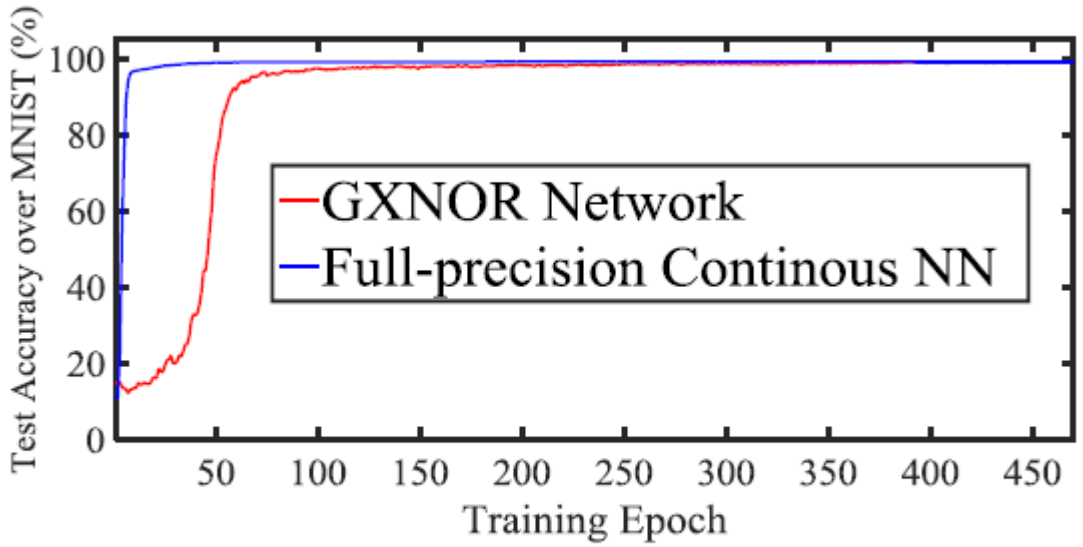


Fig. 7. **Training curve.** GXNOR network can achieve comparable final accuracy, but converges slower than full-precision continuous NN.

Binarized Neural Networks

- Training algorithm

Require: a minibatch of inputs and targets (a_0, a^*), previous weights W, previous BatchNorm parameters θ, weights initialization coefficients from (Glorot & Bengio, 2010) γ, and previous learning rate η.

Ensure: updated weights W^{t+1}, updated BatchNorm parameters θ^{t+1} and updated learning rate η^{t+1}.

{1. Computing the parameters gradients:}

{1.1. Forward propagation:}

for $k = 1$ to L **do**

$W_k^b \leftarrow \text{Binarize}(W_k)$

$s_k \leftarrow a_{k-1}^b W_k^b$

$a_k \leftarrow \text{BatchNorm}(s_k, \theta_k)$

if $k < L$ **then**

$a_k^b \leftarrow \text{Binarize}(a_k)$

end if

end for

{1.2. Backward propagation:}

{Please note that the gradients are not binary.}

Compute $g_{a_L} = \frac{\partial C}{\partial a_L}$ knowing a_L and a^*

for $k = L$ to 1 **do**

if $k < L$ **then**

$g_{a_k} \leftarrow g_{a_k^b} \circ 1_{|a_k| \leq 1}$

end if

$(g_{s_k}, g_{\theta_k}) \leftarrow \text{BackBatchNorm}(g_{a_k}, s_k, \theta_k)$

$g_{a_{k-1}^b} \leftarrow g_{s_k} W_k^b$

$g_{W_k^b} \leftarrow g_{s_k}^\top a_{k-1}^b$

end for

{2. Accumulating the parameters gradients:}

for $k = 1$ to L **do**

$\theta_k^{t+1} \leftarrow \text{Update}(\theta_k, \eta, g_{\theta_k})$

$W_k^{t+1} \leftarrow \text{Clip}(\text{Update}(W_k, \gamma_k \eta, g_{W_k^b}), -1, 1)$

$\eta^{t+1} \leftarrow \lambda \eta$

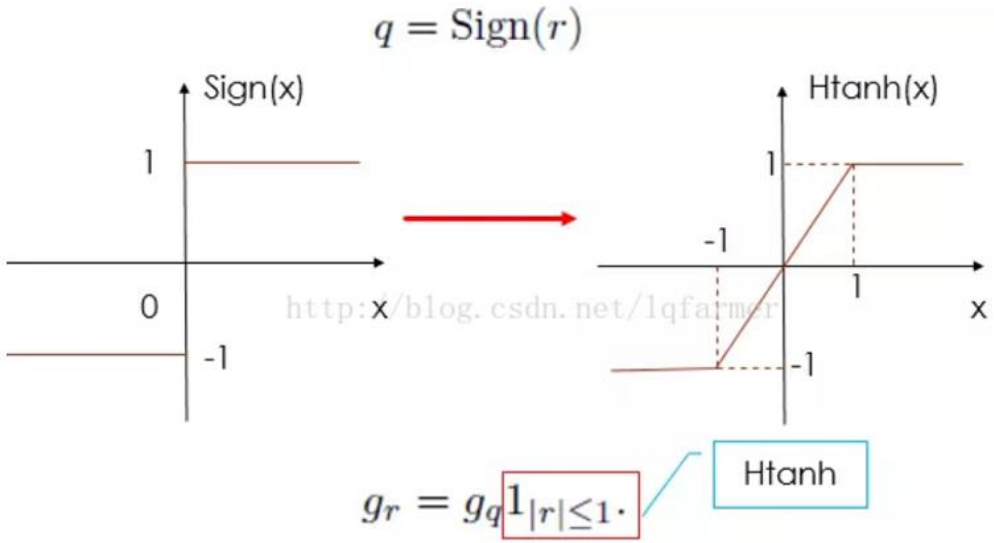
end for

Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or -1

Binarized Neural Networks: Training

- Binarization: stochastic or deterministic
  - Almost the same accuracy
- Weights in two types
  - full precision: backward pass
  - Binary: forward pass (binarized version of full precision weight)
- Gradient estimator for discrete cost
  - Straight through estimator

$$g_r = g_q 1_{|r|\leq 1}.$$



Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or -1

How to Train BNN with High Accuracy

- Learning rate
  - Full precision: start from 0.01 or 0.05 with BN
    - Too large for BNN, resulting in learning curve fluctuation due to sign change
  - BNN: 0.0001
- Scale factor
  - XNOR-net adopts a real value scale factor during binarization
    - Too complex and inefficient
  - Use PReLU instead
    - Move scale factors of weight to activation function
- Regularizer
  - Move weight to +/-1 instead of 0

binarization

$$w^b = \begin{cases} +1 & w \geq 0, \\ -1 & otherwise. \end{cases}$$

$$J(\mathbf{W}, \mathbf{b}) = L(\mathbf{W}, \mathbf{b}) + \lambda \sum_{l=1}^{L} \sum_{i=1}^{N_l} \sum_{j=1}^{M_l} (1 - (\mathbf{W}_{l,ij})^2)$$

- Last layer
  - binarization could lead to significant accuracy drop
    - No binarization is usually adopted
  - Add a learnable scale layer after this final binarized layer
    - Initialized to 0.001,
    - before sending results to softmax function

Table 4: Comparison between our method and DoReFa-net after binarizing the last layer. For fair comparison, activations of our method are also binarized with 2 bits.

Methods	Last layer	Scale layer	Comp. rate	Accuracy(%)
DoReFa-net	Full	No	10.3×	– /47.1
DoReFa-net	Binary	–	31.4×	– /40.3
BNN	Binary	Yes	27.1×	70.4/45.8

Table 5: Results before and after multiple binarizations. A denotes activation.

Methods	Bits of A	Comp. rate	Accuracy(%)
BNN	1	27.1×	64.6/39.3
BNN	2	27.1×	70.4/45.8
BNN (expand)	2	23.6×	75.6/51.4

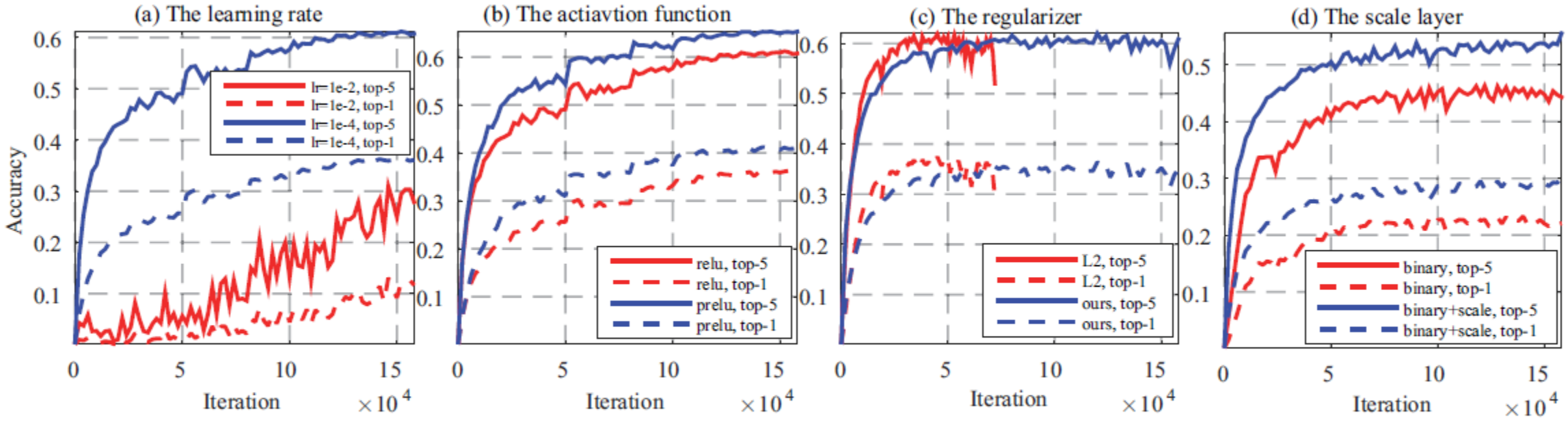


Figure 3: The effectiveness of our proposed strategies on training BNN. (a) The learning rate. (b) The activation function. (c) The regularizer. (d) The scale layer.

Table 6: Comparison of different methods on the ImageNet dataset. The compression rates inside/outside brackets are the results compared with the baseline/AlexNet model. bef. and aft. denote the model size before and after binarization, respectively.

Methods	Base model	Bits of A	Last layer	Comp. rate	Model size (bef.)	Model size (aft.)	Accuracy(%)
BNN	AlexNet	1	Full	10.3×	232MB	22.6MB	50.4/27.9
XNOR-net	AlexNet	1	Full	10.3×	232MB	22.6MB	69.2/44.2
XNOR-net	ResNet-18	1	Full	70×(13.4×)	44.6MB	3.34MB	73.2/51.2
DoReFa-net	AlexNet	2	Full	10.3×	232MB	22.6MB	– /49.8
Our method	AlexNet	2	Binary	31.2×	232MB	7.43MB	71.1/46.6
Our method	NIN-net	2	Binary	189×(23.6×)	29MB	1.23MB	75.6/51.4

Algorithm 1 Training a L−layers BinaryNet.

Require: a minibatch of inputs and targets (A_0, Y^*), previous weights W, previous PReLU parameters P, previous BatchNorm parameters θ, regularization term coefficient λ, and previous learning rate η.

Ensure: updated weights W^{t+1}, updated PReLU parameters P^{t+1}, updated BatchNorm parameters θ^{t+1}.

1. Computing the parameters gradients:

1.1 Forward propagation:

for $l := 1$ to L **do**

$B_l \leftarrow sign(W_l)$

$A^b_{l-1} \leftarrow A_{l-1}$ # $A^b_l = \sum_{i=1}^{m}(\alpha_{l,i} H_{l,i})$

$O_l \leftarrow \sum_{i=1}^{m}(\alpha_{l-1,i} xnor - popcnt(B_l, H_{l-1,i}))$

$S_l \leftarrow PReLU(O_l, P_l)$

$A_l \leftarrow BatchNorm(S_l, \theta_l)$

if $l = L$ **then**

$Y_L \leftarrow Scale(A_L)$ # *scale layer*

$C \leftarrow Loss(Y_L, Y^*)$

end if

end for

1.2 Backward propagation:

Compute $g_{A_L} = \frac{\partial C}{\partial A_L}$ knowing A_L and Y^*

for $l := L$ to 1 **do**

$(g_{S_l}, g_{\theta_l}) \leftarrow BackBatchNorm(g_{A_l}, S_l, \theta_l)$

$(g_{O_l}, g_{P_l}) \leftarrow BackPReLU(g_{S_l}, O_l, P_l)$

$g_{W_l} \leftarrow g_{O_l} A^{b\intercal}_{l-1}$

$g_{A_{l-1}} \leftarrow (B_l^\intercal g_{O_l}) \circ 1_{|A_{l-1}| \preceq 1}$

end for

2. Accumulating the parameters gradients:

for $l := 1$ to L **do**

$\theta_l^{t+1} \leftarrow Update(\theta_l, \eta, g_{\theta_l})$ # *using Adam solver*

$P_l^{t+1} \leftarrow Update(P_l, \eta, g_{P_l})$

$W_l^{t+1} \leftarrow Clip(Update(W_l, \eta, g_{W_l}), -1, 1)$

end for

Incremental Network Quantization

- Method
  - weight partition, group-wise quantization, re-training.

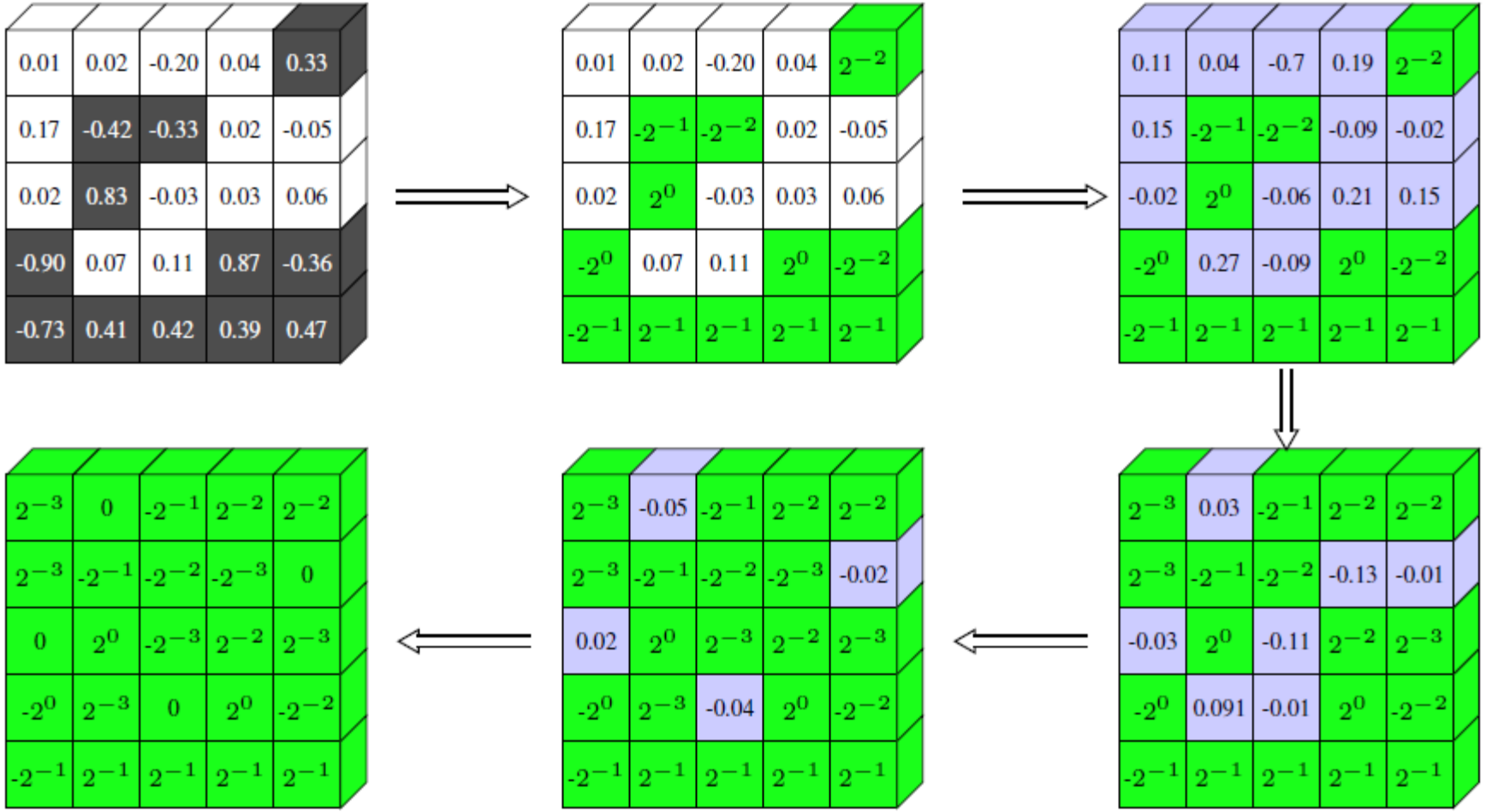
Algorithm 1 Incremental network quantization for lossless CNNs with low-precision weights.

Input: X: the training data, $\{\mathbf{W}_l : 1 \leq l \leq L\}$: the pre-trained full-precision CNN model, $\{\sigma_1, \sigma_2, \cdots, \sigma_N\}$: the accumulated portions of weights quantized at iterative steps

Output: $\{\widehat{\mathbf{W}}_l : 1 \leq l \leq L\}$: the final low-precision model with the weights constrained to be either powers of two or zero

1: Initialize $\mathbf{A}_l^{(1)} \leftarrow \emptyset$, $\mathbf{A}_l^{(2)} \leftarrow \{\mathbf{W}_l(i,j)\}$, $\mathbf{T}_l \leftarrow \mathbf{1}$, for $1 \leq l \leq L$
2: **for** $n = 1, 2, \ldots, N$ **do**
3: Reset the base learning rate and the learning policy
4: According to σ_n, perform layer-wise weight partition and update $\mathbf{A}_l^{(1)}$, $\mathbf{A}_l^{(2)}$ and $\mathbf{T}_l$
5: Based on $\mathbf{A}_l^{(1)}$, determine $\mathbf{P}_l$ layer-wisely
6: Quantize the weights in $\mathbf{A}_l^{(1)}$ by Equation (4) layer-wisely
7: Calculate feed-forward loss, and update weights in $\{\mathbf{A}_l^{(2)} : 1 \leq l \leq L\}$ by Equation (8)
8: **end for**

Incremental Network Quantization



Incremental Network Quantization

- On imageNet

Network	Bit-width	Top-1 error	Top-5 error	Decrease in top-1/top-5 error
AlexNet ref	32	42.76%	19.77%	
AlexNet	5	**42.61%**	**19.54%**	0.15%/0.23%
VGG-16 ref	32	31.46%	11.35%	
VGG-16	5	**29.18%**	**9.70%**	2.28%/1.65%
GoogleNet ref	32	31.11%	10.97%	
GoogleNet	5	**30.98%**	**10.72%**	0.13%/0.25%
ResNet-18 ref	32	31.73%	11.31%	
ResNet-18	5	**31.02%**	**10.90%**	0.71%/0.41%
ResNet-50 ref	32	26.78%	8.76%	
ResNet-50	5	**25.19%**	**7.55%**	1.59%/1.21%

Model	Bit-width	Top-1 error	Top-5 er
ResNet-18 ref	32	31.73%	11.31%
INQ	5	31.02%	10.90%
INQ	4	31.11%	10.99%
INQ	3	31.92%	11.64%
INQ	2 (ternary)	33.98%	12.87%

Method	Bit-width(Conv/FC)	Compression ratio	Decrease in top-1/top5 erro
Han et al. (2016) (P+Q)	8/5	27×	0.00%/0.03%
Han et al. (2016) (P+Q+H)	8/5	35×	0.00%/0.03%
Han et al. (2016) (P+Q+H)	8/4	-	-0.01%/0.00%
Our method (P+Q)	5/5	**53**×	**0.08%/0.03%**
Han et al. (2016) (P+Q+H)	4/2	-	-1.99%/-2.60%
Our method (P+Q)	4/4	**71**×	**-0.52%/-0.20%**
Our method (P+Q)	3/3	**89**×	**-1.47%/-0.96%**

Binary Ensemble Neural Network: More Bits per Network or More Networks per Bit?

Table 6. Comparison with state-of-the-arts on ImageNet using ResNet-18 (W-weights, A-activation)

Method	W	A	Top-1
Full-Precision DNN [27, 43]	32	32	69.3%
XNOR-Net [50]	1	1	48.6%
ABC-Net [43]	1	1	42.7%
BNN [31, 50]	1	1	42.2%
BENN-SB-3, Bagging (ours)	1	1	**53.4%**
BENN-SB-3, Boosting (ours)	1	1	**53.6%**
BENN-SB-6, Bagging (ours)	1	1	**57.9%**
BENN-SB-6, Boosting (ours)	1	1	**61.0%**

WRPN: Wide Reduced-Precision Networks

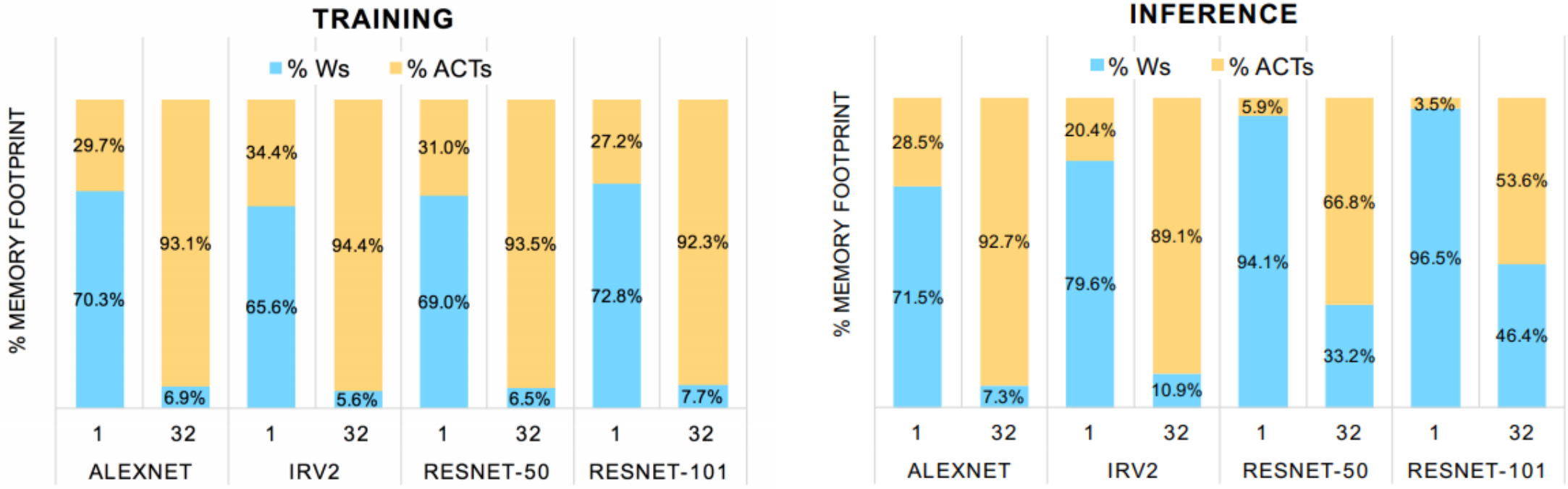


Figure 1: Memory footprint of activations (ACTs) and weights (W) during training and inference for mini-batch sizes 1 and 32.

Table 1: AlexNet top-1 validation set accuracy % as precision of activations (A) and weight(W) changes. All results are with end-to-end training of the network from scratch. – is a data-point we did not experiment for.

	32b A	8b A	4b A	2b A	1b A
32b W	57.2	54.3	54.4	52.7	–
8b W	–	54.5	53.2	51.5	–
4b W	–	54.2	54.4	52.4	–
2b W	57.5	50.2	50.5	51.3	–
1b W	56.8	–	–	–	44.2

Table 2: AlexNet 2x-wide top-1 validation set accuracy % as precision of activations (A) and weights (W) changes.

	32b A	8b A	4b A	2b A	1b A
32b W	60.5	58.9	58.6	57.5	52.0
8b W	–	59.0	58.8	57.1	50.8
4b W	–	58.8	58.6	57.3	–
2b W	–	57.6	57.2	55.8	–
1b W	–	–	–	–	48.3

Table 3: Compute cost of AlexNet 2x-wide vs. 1x-wide as precision of activations (A) and weights (W) changes.

	32b A	8b A	4b A	2b A	1b A
32b W	3.9x	2.4x	2.2x	2.1x	2.0x
8b W	2.4x	1.0x	0.7x	0.6x	0.6x
4b W	2.2x	0.7x	0.5x	0.4x	0.3x
2b W	2.1x	0.6x	0.4x	0.2x	0.2x
1b W	2.0x	0.6x	0.3x	0.2x	0.1x

Network	Method	Precision (w,a)	Accuracy (% top-1)	Accuracy (% top-5)
ResNet-18	baseline	32,32	69.76	89.08
ResNet-18	**Apprentice**	**4,8**	**70.40**	**-**
ResNet-18	**FAQ (This paper)**	**8,8**	**70.02**	**89.32**
ResNet-18	**FAQ (This paper)**	**4,4**	**69.78±0.04**	**89.11±0.03**
ResNet-18	Joint Training	4,4	69.3	-
ResNet-18	UNIQ	4,8	67.02	-
ResNet-18	Distillation	4,32	64.20	-
ResNet-34	baseline	32,32	73.30	91.42
ResNet-34	**FAQ (This paper)**	**8,8**	**73.71**	**91.63**
ResNet-34	**FAQ (This paper)**	**4,4**	**73.31**	**91.32**
ResNet-34	UNIQ	4,32	73.1	-
ResNet-34	Apprentice	4,8	73.1	-
ResNet-34	UNIQ	4,8	71.09	-
ResNet-50	baseline	32,32	76.15	92.87
ResNet-50	**FAQ (This paper)**	**8,8**	**76.52**	**93.09**
ResNet-50	**FAQ (This paper)**	**4,4**	**76.25**	**93.00**
ResNet-50	**EL-Net**	**4,4**	**75.9**	**92.4**
ResNet-50	IOA	8,8	74.9	-
ResNet-50	Apprentice	4,8	74.7	-
ResNet-50	UNIQ	4,8	73.37	-
ResNet-152	baseline	32,32	78.31	
ResNet-152	**FAQ (This paper)**	**8,8**	**78.54**	
ResNet-152	**FAQ (This paper)**	**4,4**	**78.35**	
Inception-v3	baseline	32,32	77.45	
Inception-v3	**FAQ (This paper)**	**8,8**	**77.60**	
Inception-v3	FAQ (This paper)	4,4	77.33	
Inception-v3	IOA	8,8	74.2	
Densenet-161	baseline	32,32	77.65	
Densenet-161	**FAQ (This paper)**	**4,4**	**77.90**	
Densenet-161	**FAQ (This paper)**	**8,8**	**77.84**	
VGG-16bn	baseline	32,32	73.36	
VGG-16bn	**FAQ (This paper)**	**4,4**	**73.87**	
VGG-16bn	**FAQ (This paper)**	**8,8**	**73.66**	

Regularizing Activation Distribution for Training Binarized Deep Networks

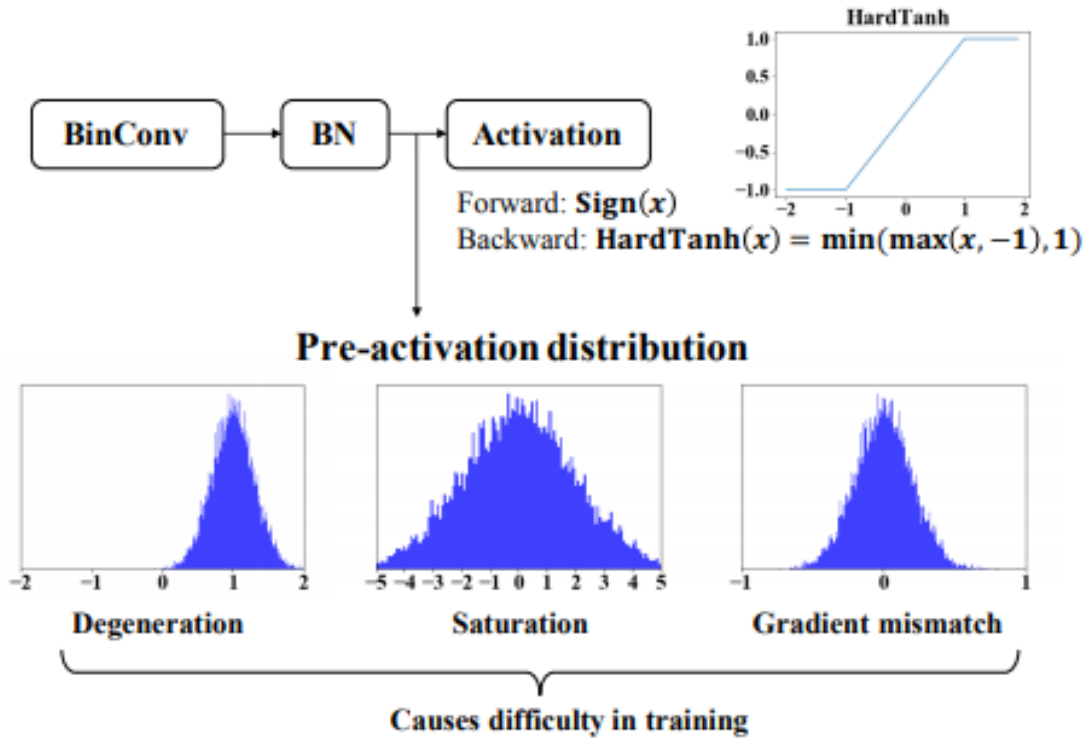


Figure 1: The basic Conv-BN-Act structure for BNN (BinConv: binary convolution; BN: batch normalization). The pre-activation distribution may exhibit from degeneration, saturation or gradient mismatch problem that causes difficulty in training.

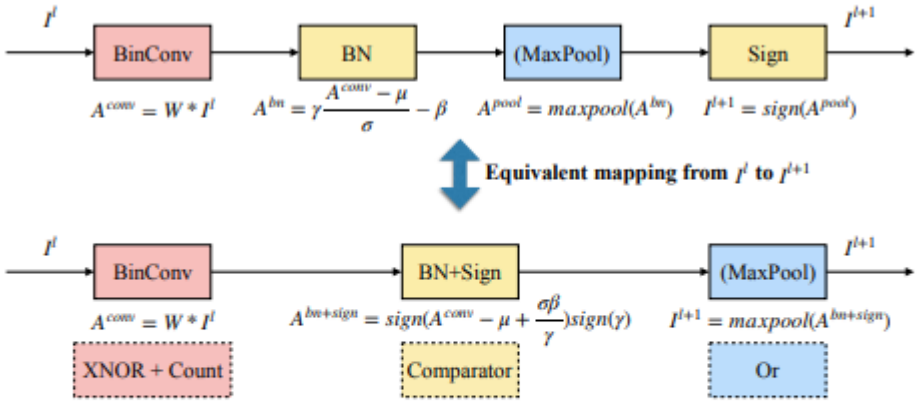


Figure 2: Basic block for convolutional BNN [22]. The activations I^l and weights W are binarized to ± 1. The inference of this block can be implemented on hardware with only logical operators.

Then, in the training phase, we add the distribution loss for the b-th batch of input data:

$$L_{DL}^{b} = \sum_{l,c} L_{DL}^{b,l,c} = \sum_{l,c} L_{D}^{b,l,c} + L_{S}^{b,l,c} + L_{M}^{b,l,c}, \qquad (2)$$

Table 4: Comparison with prior art using 1-bit weights and activations, in terms of accuracy and computation energy on different datasets. The best results are shown in bold face.

Dataset	Model	Pure-logical	Energy cost	Accuracy
CIFAR-10	BNN [22]	Yes	1×	87.13%
	XNOR-Net [38]	No	4.5×	87.38%
	LAB [19]	No	4.5×	87.72%
	BNN-DL	**Yes**	1×	**89.90%**
SVHN	BNN [22]	Yes	1×	96.50%
	XNOR-Net [38]	No	4.5×	96.57%
	LAB [19]	No	4.5×	96.64%
	BNN-DL	**Yes**	1×	**97.23%**
CIFAR-100	BNN [22]	No	1×	60.40%
	DQ-2bit [37]	No	-	49.32%
	BNN-DL	**No**	1×	**68.17%**

Table 5: Robustness to the selection of optimizer, learning rate, and network structure. CIFAR-10 is used for illustrating the results.

	BNN	BNN-DL
Momentum	66.02%	89.37%
Nesterov	68.66%	89.22%
Adam	88.12%	89.62%
RMSprop	87.39%	90.24%
lr_{init}=1e-1	82.19%	89.60%
lr_{init}=5e-3	88.12%	89.62%
lr_{init}=2e-4	85.62%	88.73%
VGG	88.12%	89.62%
ResNet-18	85.71%	90.47%

LOW RANK APPROXIMATION

Low Rank Approximation

- Layer responses lie in a low-rank subspace
- Decompose a convolutional layer with d filters with filter size $k \times k \times c$ to
  - A layer with d' filters ($k \times k \times c$)
  - A layer with d filter ($1 \times 1 \times d$')

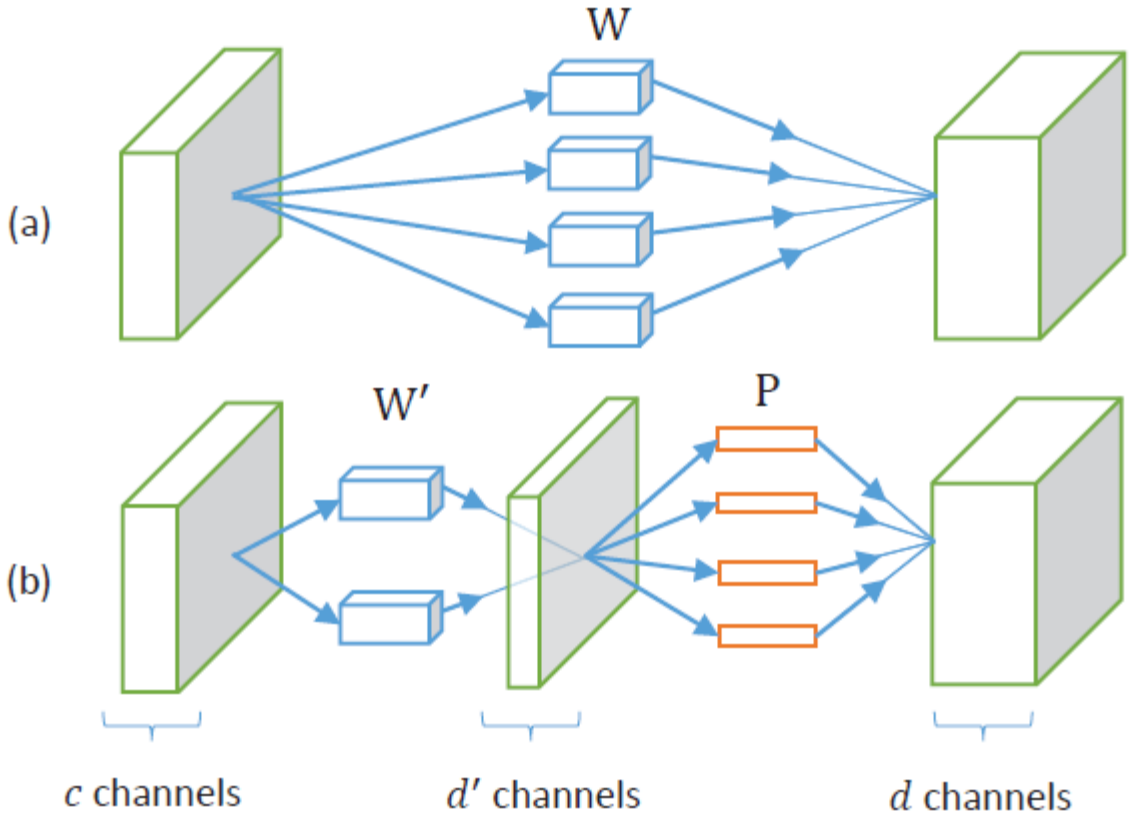


Figure 1. Illustration of the approximation. (a) An original layer with complexity $O(dk^2c)$. (b) An approximated layer with complexity reduced to $O(d'k^2c) + O(dd')$.

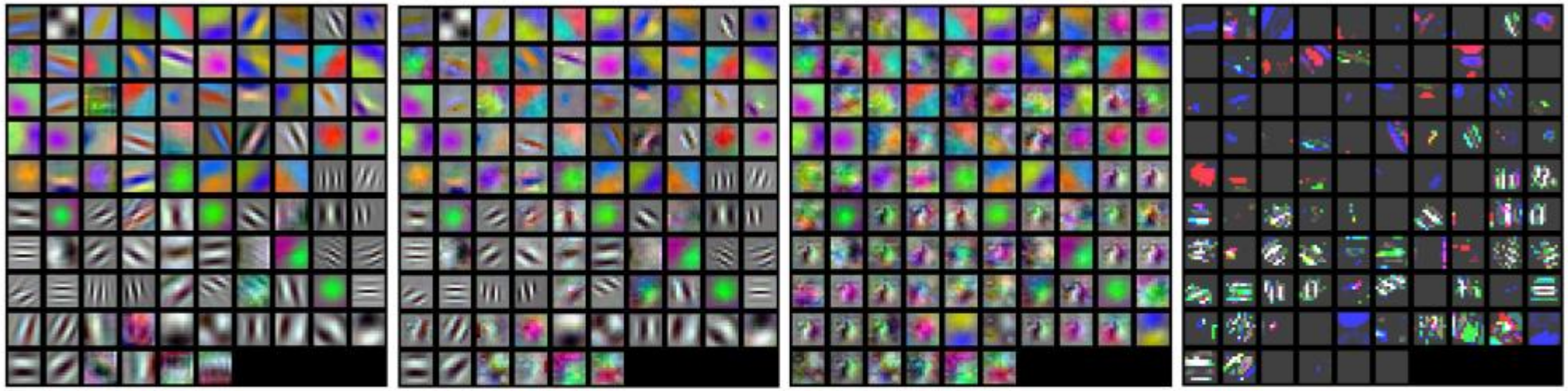
Zhang et al Efficient and Accurate Approximations of Nonlinear Convolutional Networks CVPR'15

speedup	rank sel.	Conv1	Conv2	Conv3	Conv4	Conv5	Conv6	Conv7	err. ↑ %
2×	no	32	110	199	219	219	219	219	1.18
2×	**yes**	32	83	182	211	239	237	253	**0.93**
2.4×	no	32	96	174	191	191	191	191	1.77
2.4×	**yes**	32	74	162	187	207	205	219	**1.35**
3×	no	32	77	139	153	153	153	153	2.56
3×	**yes**	32	62	138	149	166	162	167	**2.34**
4×	no	32	57	104	115	115	115	115	4.32
4×	**yes**	32	50	112	114	122	117	119	**4.20**
5×	no	32	46	83	92	92	92	92	6.53
5×	**yes**	32	41	94	93	98	92	90	**6.47**

Zhang et al Efficient and Accurate Approximations of Nonlinear Convolutional Networks CVPR'15

On Compressing Deep Models by Low Rank and Sparse Decomposition

- Observations
  - weight filters tend to be both low-rank and sparse
  - weight filters usually share smooth components in a low-rank subspace, and also remember some important information represented by weights that are sparsely scattered outside the low-rank subspace.
  - The resulting feature maps also contain the smooth components [30] and the spiky changes that represent the uniqueness of each feature
- Proposed
  - unified framework integrating the low-rank and sparse decomposition of weight matrices with the feature map reconstructions
  - greedy bilateral decomposition (GreBdec) for deep model compression



(a) (b) (c) (d)

Figure 1. (a) Filters in the first layer of AlexNet (from Caffe Model Zoo). (b) Low-rank and sparse approximation using the proposed method. Here, rank is 12 (for L) and ratio of non-zero entries is 12.5% (for S). We reduce the number of parameters by $4\times$. (c) Low-rank components of the approximated filters. (d) Sparse components of the approximated filters. Here, for better visualization, we add a constant upon the whole sparse matrix.

Algorithm 1 Greedy Bilateral Decomposition (GreBdec)

Input: X, W, Y, target rank r, rank step size Δr, objective function f, and power K. Initial rank r_0 and initial $V = V_0$.

Output: U, V and S.

while *unconvergence* **do**
  for $i = 0$ **to** K **do**
    Update U and V using equation (7).
  end for
  Update S using last equation in (5).
  Calculate the top Δr right singular vectors v or random projections of $U^\top \frac{\partial \mathcal{L}}{\partial UV}$.
  Set $V := [V, v]$.
end while

Table 1. Compression rates for deep networks. O: Reference network. C: Compressed network. R: Compression rate. $\#W$: Total number of weights in the networks.

Network	Top-1	Top-5	#W	R
AlexNet(O)	57.22%	80.27%	61M	10
AlexNet(C)	57.26%	80.31%	6M	
VGG-16(O)	68.50%	88.68%	138M	15
VGG-16(C)	68.75%	89.06%	9.7M	
GoogLeNet(O)	68.70%	88.90%	7M	4.5
GoogLeNet(C)	67.30%	88.11%	1.5M	

Table 5. Comparison of overall compression rates. O: Reference model. R: Compression rate. #W: The total number of weights in the networks.

Network	Top-1	Top-5	#W	R
AlexNet (O)	57.22%	80.27%	61M	1×
Tai et al. [27]	–	79.66%	12.2M	5×
Kim et al. [14]	–	78.33%	11M	5.46×
Han et al. [8]	57.23%	80.33%	6.7M	9×
GreBdec	57.26%	80.31%	6M	**10**×
VGG-16 (O)	68.50%	88.68%	138M	1×
Tai et al. [27]	–	90.31%	50.2M	2.75×
Kim et al. [14]	–	89.40%	127M	1.09×
Han et al. [8]	68.66%	89.12%	10.3M	13×
GreBdec	68.75%	89.06%	9.7M	**15**×
GoogLeNet (O)	68.70%	88.90%	6.9M	1×
Tai et al. [27]	–	91.79%	2.4M	2.84×
Kim et al. [14]	–	88.66%	4.7M	1.28×
GreBdec	67.30%	88.11%	1.5M	**4.5**×

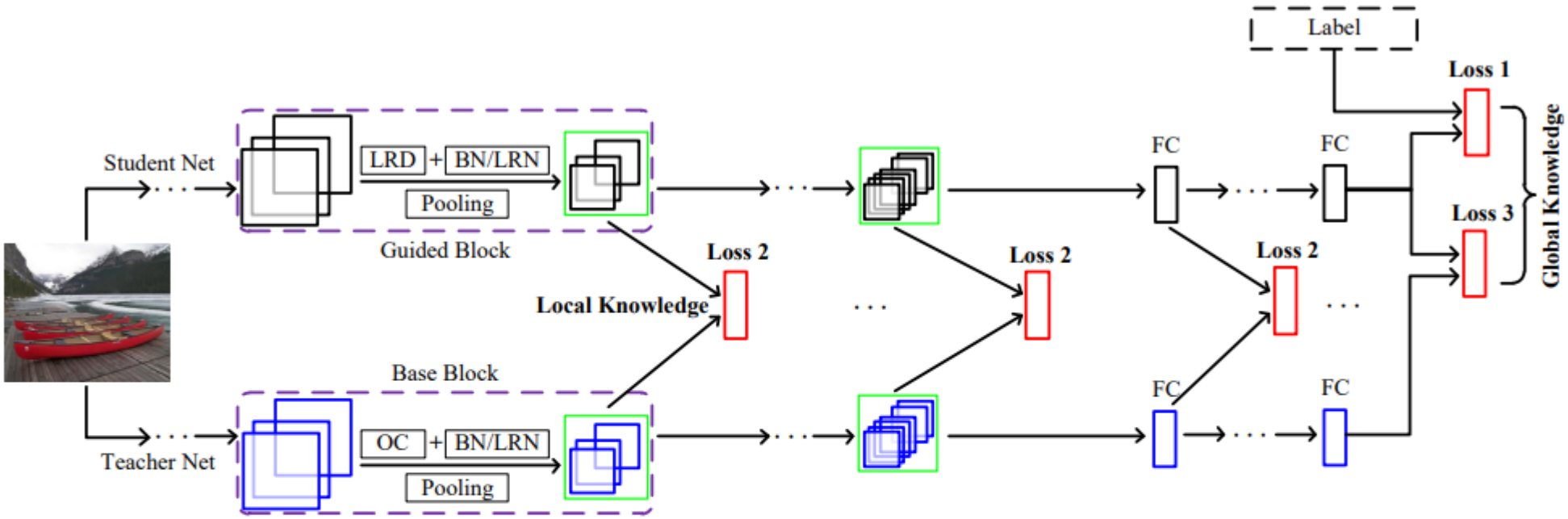


Fig. 1. The framework of the proposed LRDKT. A low-rank decomposition (LRD) based compression scheme is first constructed to form a guided block in the student network. Next, the "local" and "global" knowledge is transferred by KT in a unified way, which deals with the performance degradation caused by LRD compression. The based and guided blocks are defined in Sec. 3.3.

WINOGRAD TRANSFORMATION

Convolution

Definition: Convolution in the time domain is equivalent to pointwise multiply in the frequency domain.

$$f * g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\}$$

$\mathcal{F}\{f\}$ and $\mathcal{F}\{g\}$ are the Fourier transforms of f and g

The asterisk denotes convolution, not multiplication.

$$f = \begin{bmatrix} 0 & 0 & 1 & 2 \end{bmatrix}$$

$$g = \begin{bmatrix} 10 & 20 & 30 \end{bmatrix}$$

$$f * g = \begin{bmatrix} 30 & 80 \end{bmatrix}$$

Dot Product Approach

$$out[0] = 0 * 10 + 0 * 20 + 1 * 20 = 30$$

$$out[1] = 0 * 10 + 1 * 20 + 2 * 20 = 80$$

- 6 Multiplications
- 4 Additions

Shmuel Winograd (Winograd FFTs)

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} \quad (5)$$

where

$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$

$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

- Data (d's): 4 ADDs
- Filter (g's): 3 ADDs, 2 MULs
- Outputs (m's): **4 MULs**, **4 ADDs**

Winograd Convolution

- Winograd's minimal filtering algorithm for small filter sizes (3x3)
  - computing m outputs with an r-tap FIR filter, which we call F(m, r), requires $\mu(F(m, r)) = m + r - 1$ multiplications
  - minimal 2D algorithms: for computing m × n outputs with an r × s filter, which we call F(m × n, r × s). These require $\mu(F(m \times n, r \times s)) = \mu(F(m, r))\mu(F(n, s)) = (m + r - 1)(n + s - 1)$

$$\mu(F(2,3)) = 2 + 3 - 1 = 4$$

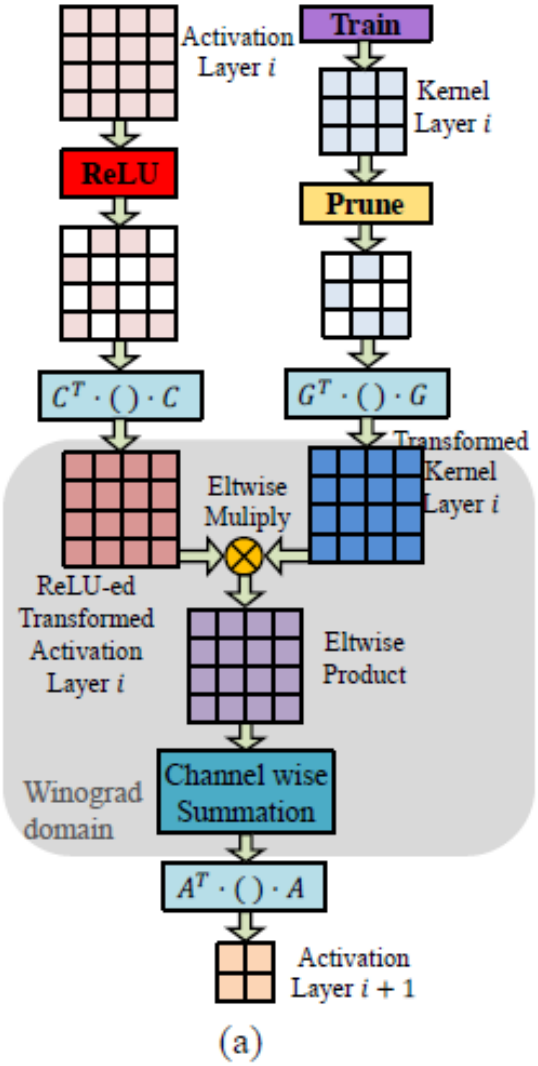
4 mul v.s. 6 mul

F(2×2, 3×3) uses 4×4 = 16 multiplications
standard algorithm uses 2 × 2 × 3 × 3 = 36
an arithmetic complexity reduction of 36/16 = **2.25**

N	cuDNN		F(2x2,3x3)		Speedup
	msec	TFLOPS	msec	TFLOPS	
1	12.52	3.12	5.55	7.03	2.26X
2	20.36	3.83	9.89	7.89	2.06X
4	104.70	1.49	17.72	8.81	5.91X
8	241.21	1.29	33.11	9.43	7.28X
16	203.09	3.07	65.79	9.49	3.09X
32	237.05	5.27	132.36	9.43	1.79X
64	394.05	6.34	266.48	9.37	1.48X

Table 5. cuDNN versus $F(2 \times 2, 3 \times 3)$ performance on VGG Network E with fp32 data. Throughput is measured in Effective TFLOPS, the ratio of direct algorithm GFLOPs to run time.

Winograd Convolution



Producing 4 output pixels:

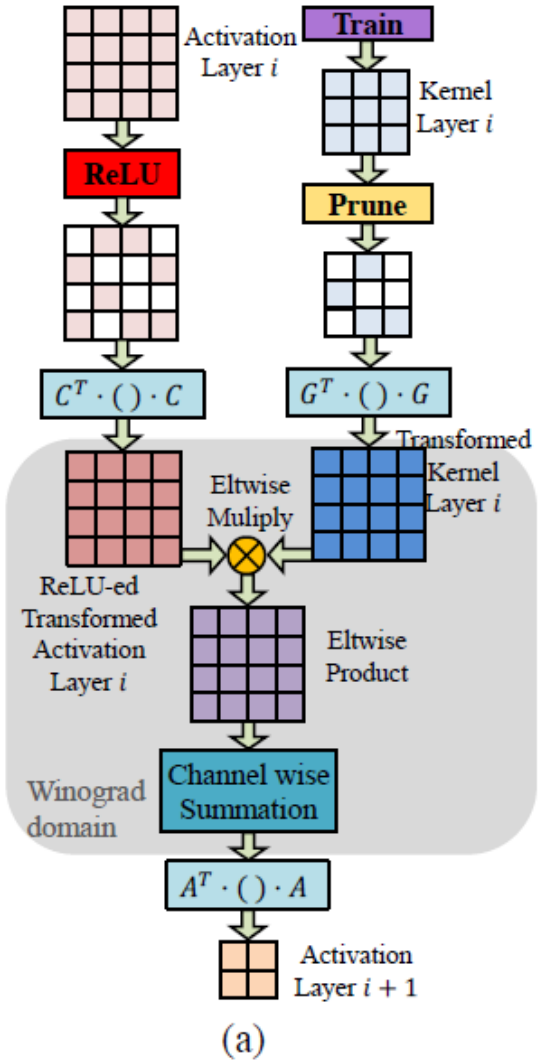
Direct Convolution:

- 4*9=36 multiplications (**1x**)

Winograd convolution:

- 4*4=16 multiplications (**2.25x** less)

Winograd Convolution v.s. Sparse



Producing 4 output pixels:

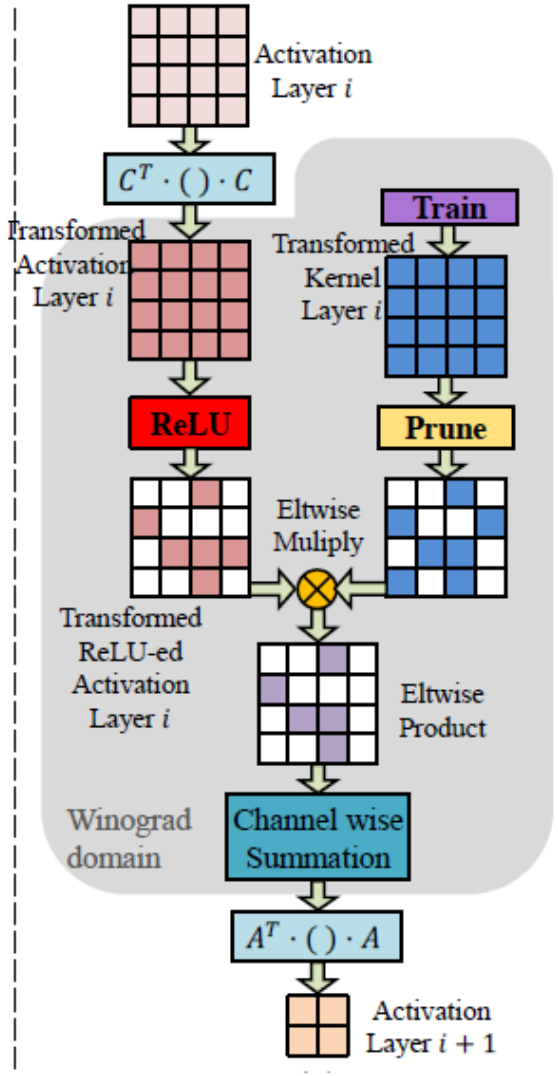
Direct Convolution:

- 4*9=36 multiplications (**1x**)
- sparse weight [NIPS'15] (**3x**)
- sparse activation (relu) (**3x**)
- Overall saving: **9x**

Winograd convolution:

- 4*4=16 multiplications (**2.25x** less)
- dense weight (**1x**)
- dense activation (**1x**)
- Overall saving: **2.25x**

Winograd Convolution + Sparse



Producing 4 output pixels:

Direct Convolution:

- 4*9=36 multiplications (**1x**)
- sparse weight [NIPS'15] (**3x**)
- sparse activation (relu) (**3x**)
- Overall saving: **9x**

Winograd convolution:

- 4*4=16 multiplications (**2.25x** less)
- sparse weight (**2.5x**)
- dense activation (**2.25x**)
- Overall saving: **12x**

- Pruned Winograd-transformed weights
- Moving ReLU to the Winograd domain

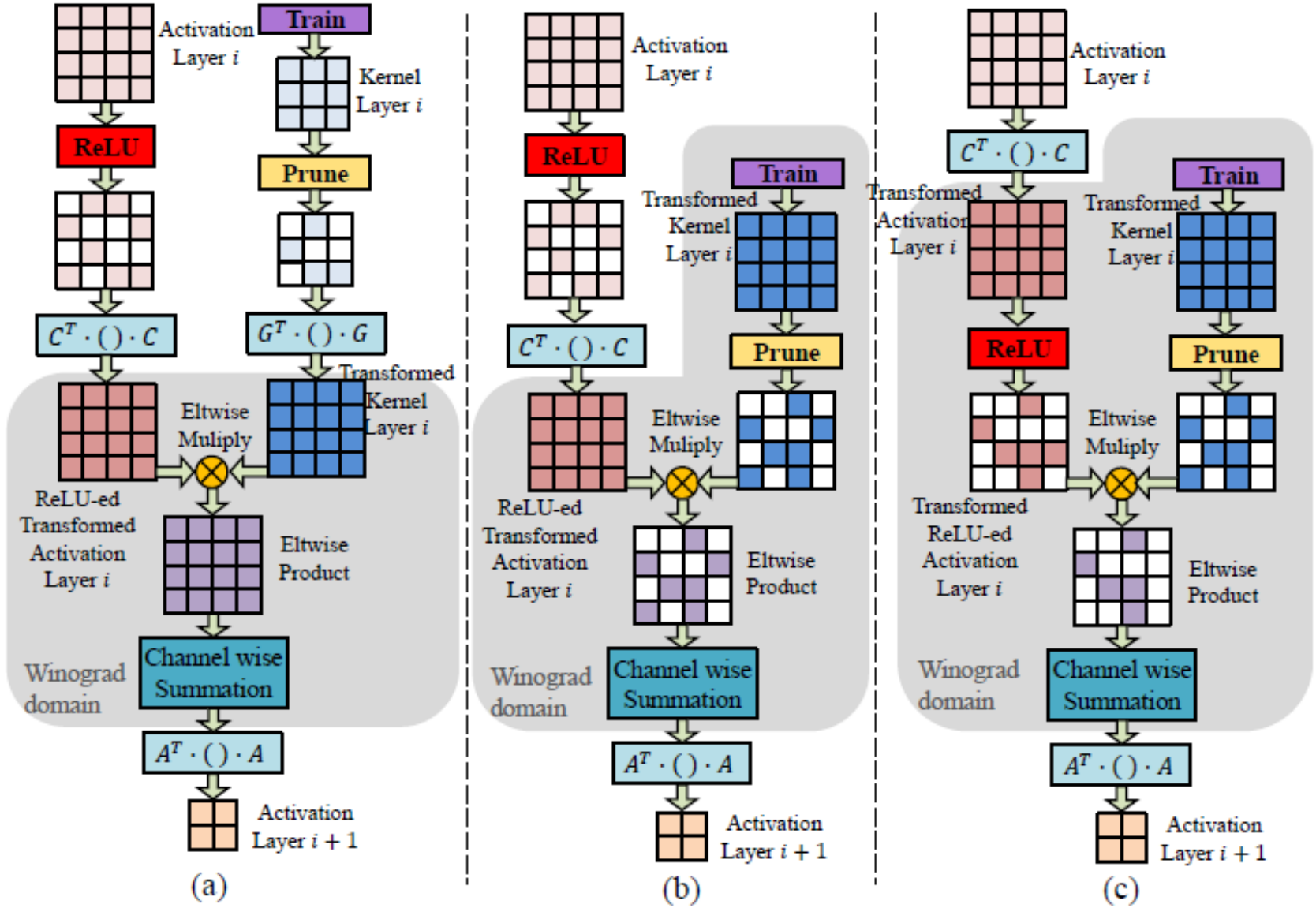
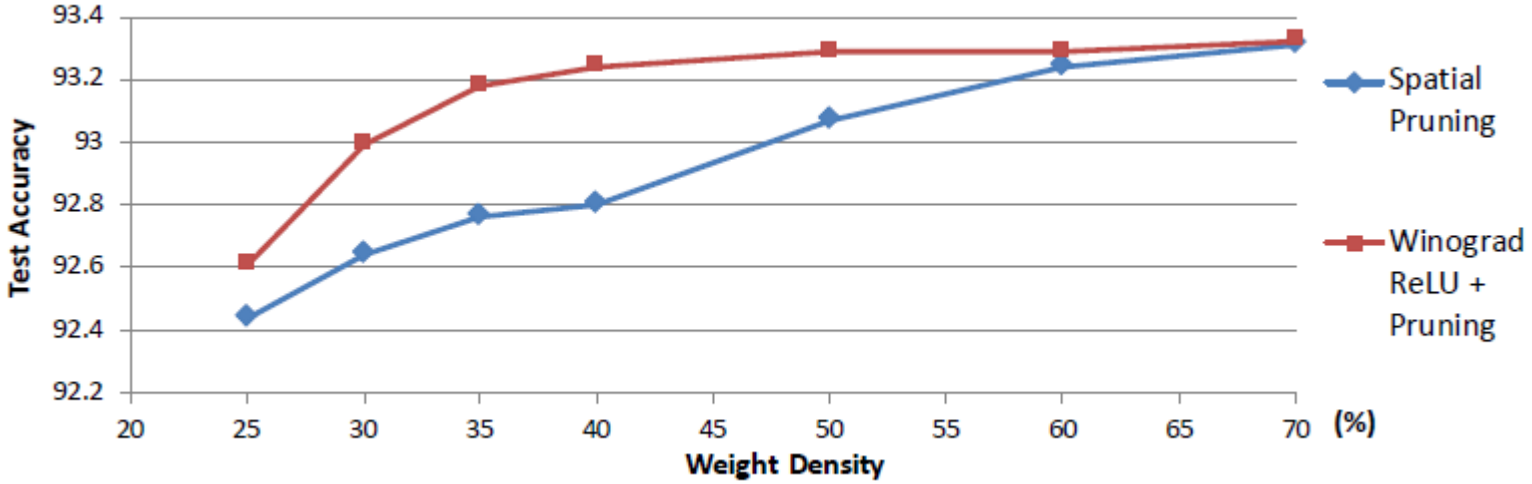
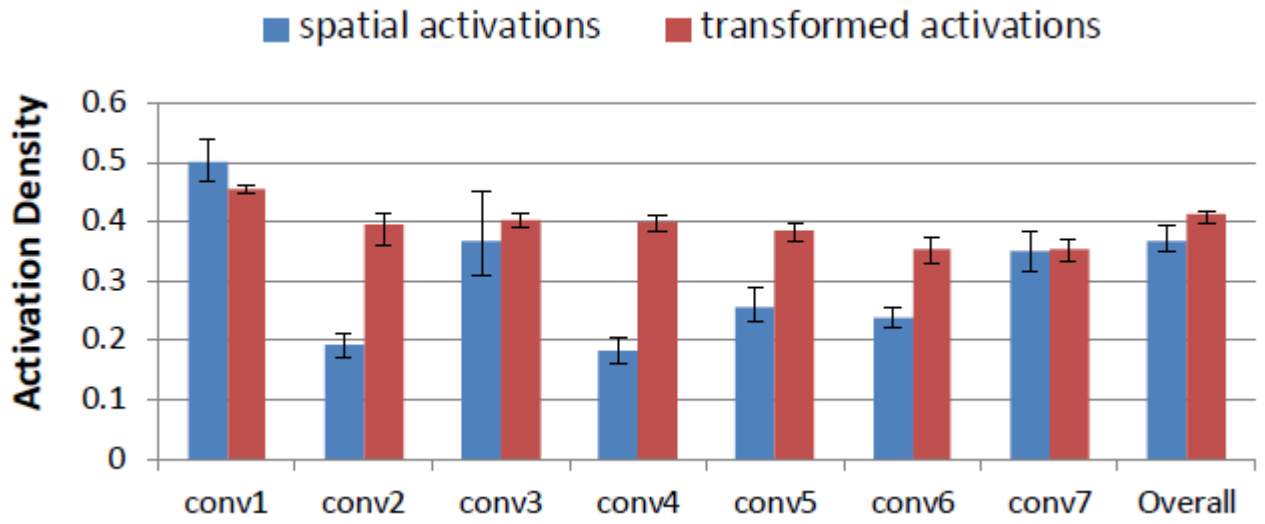


Figure 1: Combining Winograd convolution with sparse weights and activations. (a) Original Winograd-based convolution proposed by Lavin (2015) fills in the non-zeros in both the weights and activations. (b) Pruning the 4×4 transformed kernel restores sparsity to the weights. (c) Moving the ReLU layer after Winograd transformation also restores sparsity to the activations.

Winograd Convolution + Sparse

pruned to 40% density without significant (> 0:1%) loss of accuracy





overall activation density of 41:1% compared to 36:9% density for the spatial activations

LOW COMPLEXITY OR COMPACT MODEL FROM SCRATCH

Problem of Sparse CNN

- Irregular weight and activation distribution
  - Bad for modern SIMD computation and cache architecture
- Solution
  - Structure sparsity as shown before
  - Transfer learning: teacher student model
  - Compact architecture from scratch

Compact models

- Low complexity layers with 1x1 for channel compression
  - Squeezenet: fewer parameters
- Group convolution on channels based method
  - ResneXT
- Separable convolution: depthwise + 1x1 (group or not)
  - Mobilenet v1/v2
  - Xception
  - shufflenet v1/v2

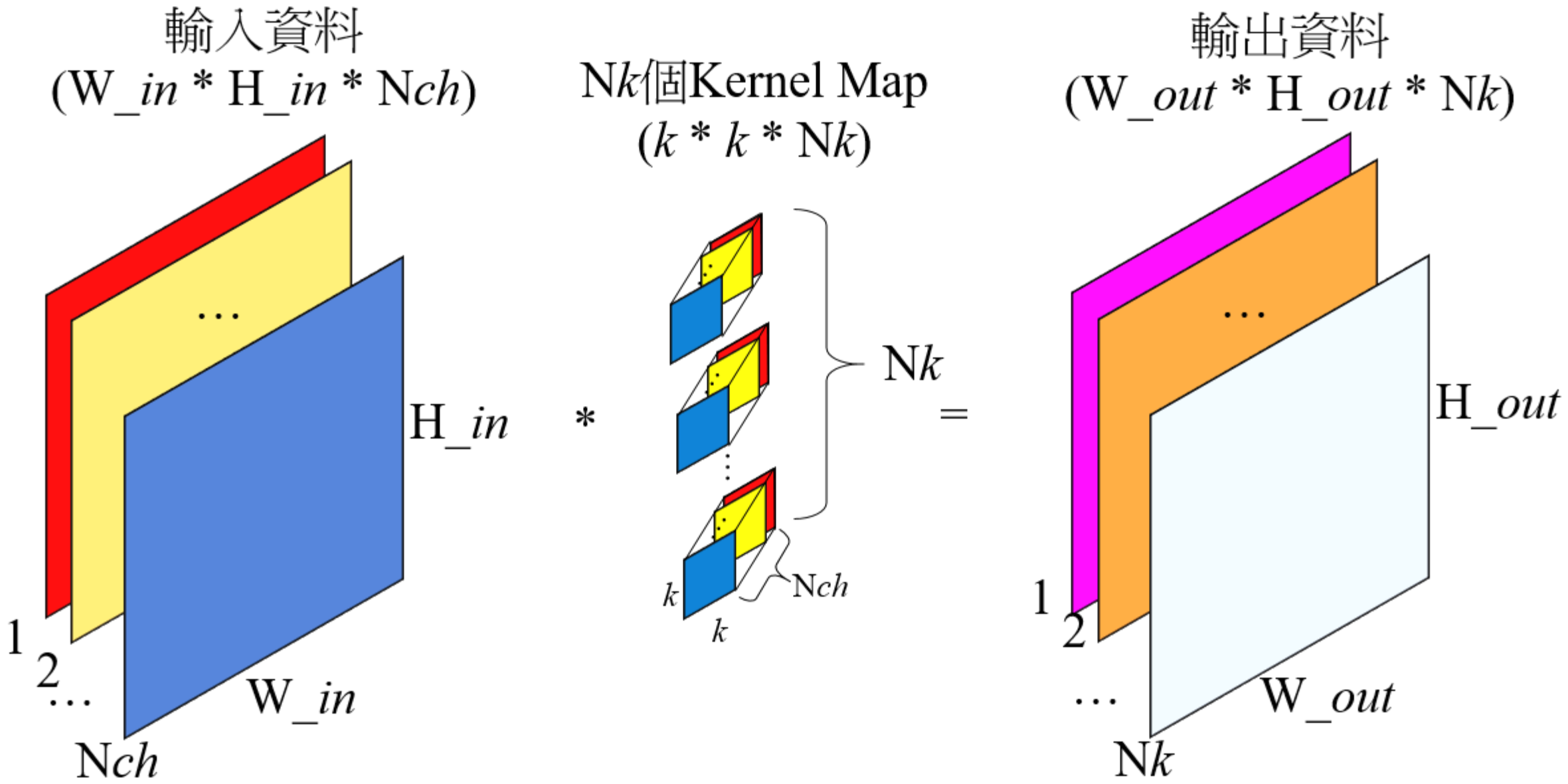
$$W\_in * H\_in * Nch * k * k * Nk$$

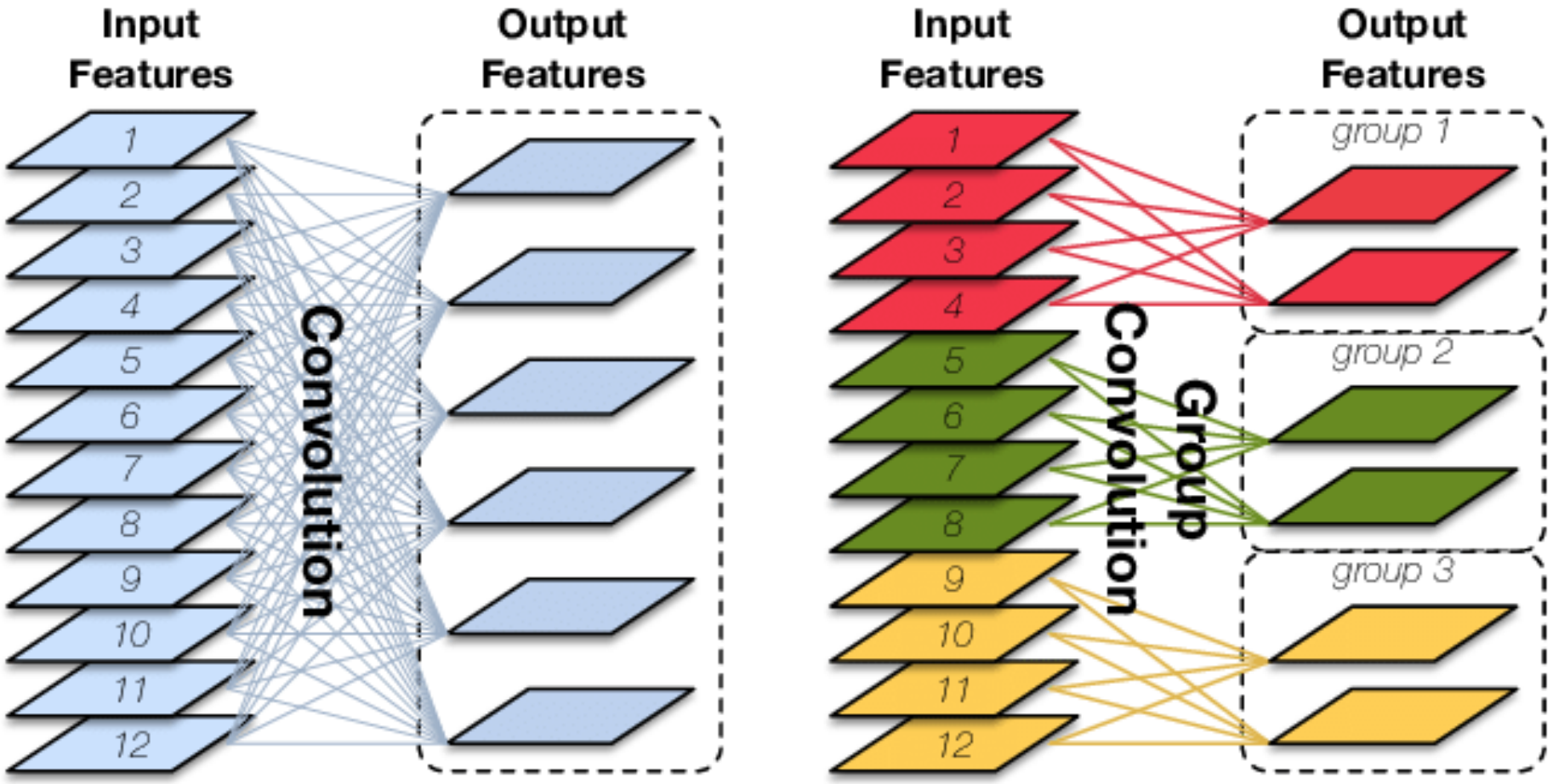
Input size

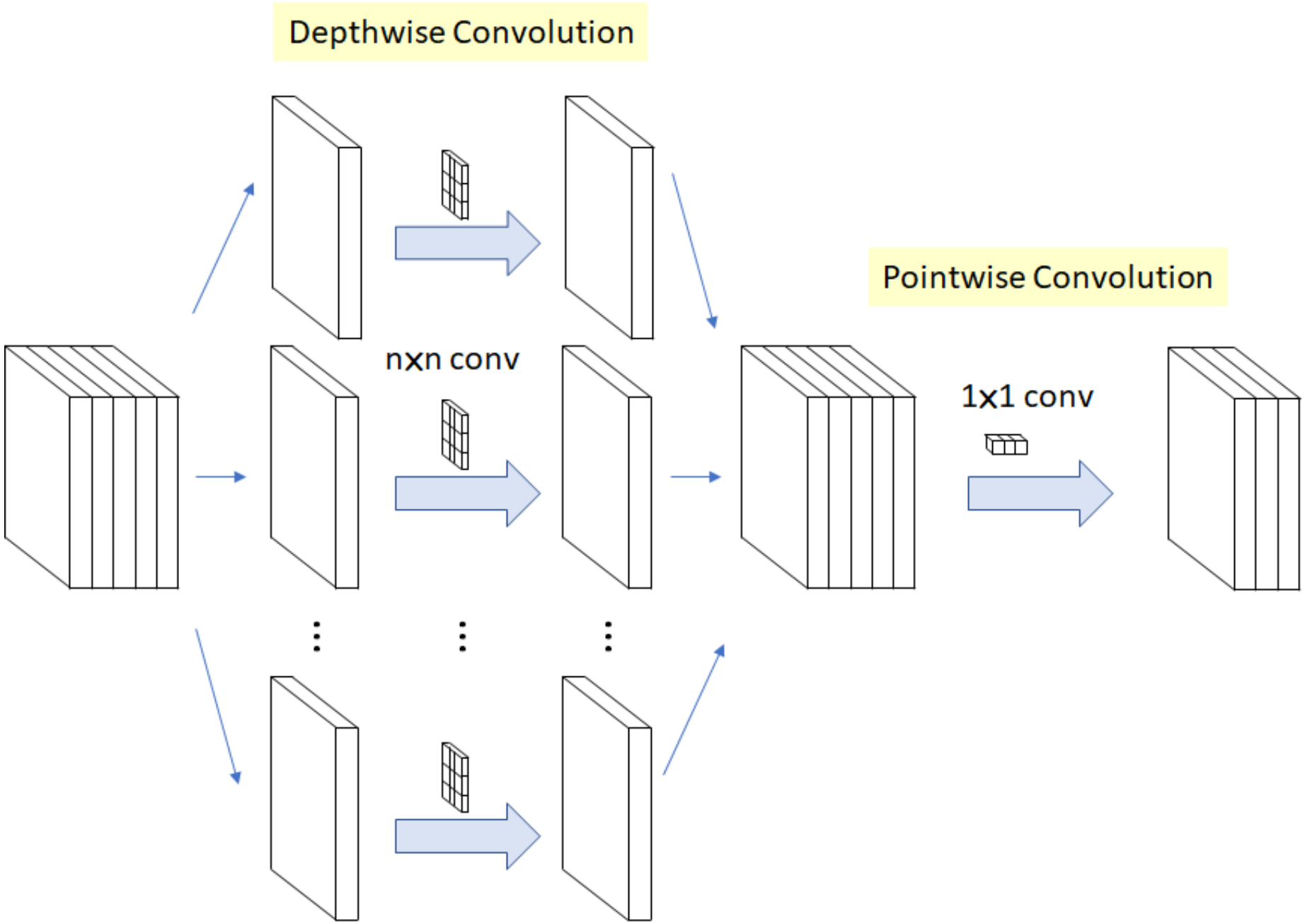
Input Channel No.

Kernel size

output Channel no.


輸入資料
(W_in * H_in * Nch)
Nk個Kernel Map
(k * k * Nk)
輸出資料
(W_out * H_out * Nk)
H_in
W_in
Nch
1
2
...
*
Nk
=
k
k
Nch
H_out
W_out
Nk


Input Features
Output Features
1
2
3
4
5
6
7
8
9
10
11
12
Convolution
Input Features
Output Features
group 1
group 2
group 3
Group Convolution


Depthwise Convolution
nxn conv
Pointwise Convolution
1x1 conv

SqueezeNet

- Strategy
  - Replace 3x3 filters with 1x1 filters: 9x fewer parameters
    - make the majority of these filters 1x1
  - Decrease the number of input channels to 3x3 filters
    - total quantity of parameters in this layer is (number of input channels) * (number of filters) * (3*3)
  - Downsample late in the network
    - large activation maps (due to delayed downsampling) can lead to higher classification accuracy

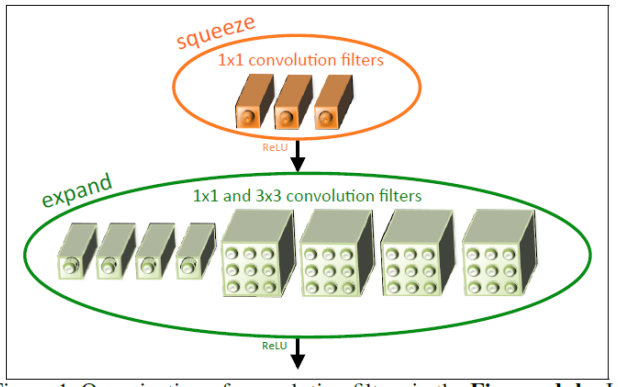
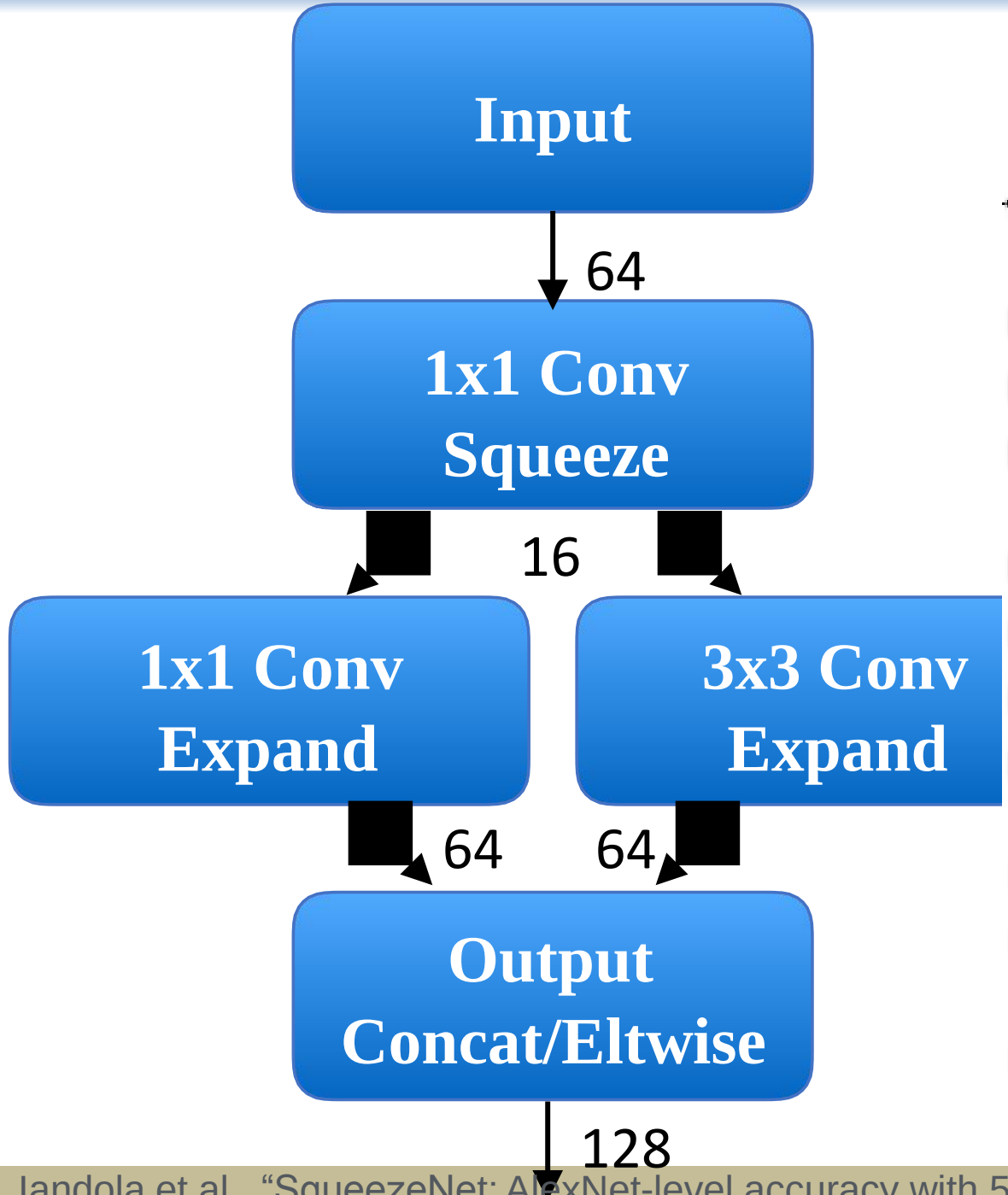
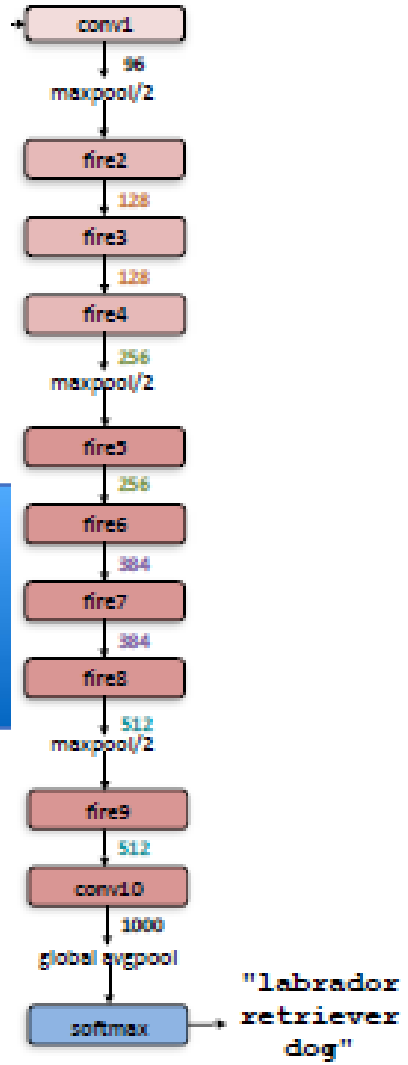


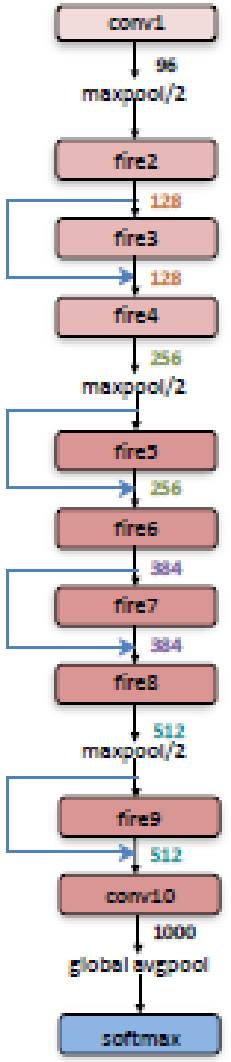
Figure 1. Organization of convolution filters in the **Fire module**. In this example, $s_{1x1} = 3$, $e_{1x1} = 4$, and $e_{3x3} = 4$. We illustrate the convolution filters but not the activations.

$$W\_in * H\_in * Nch * k * k * Nk$$

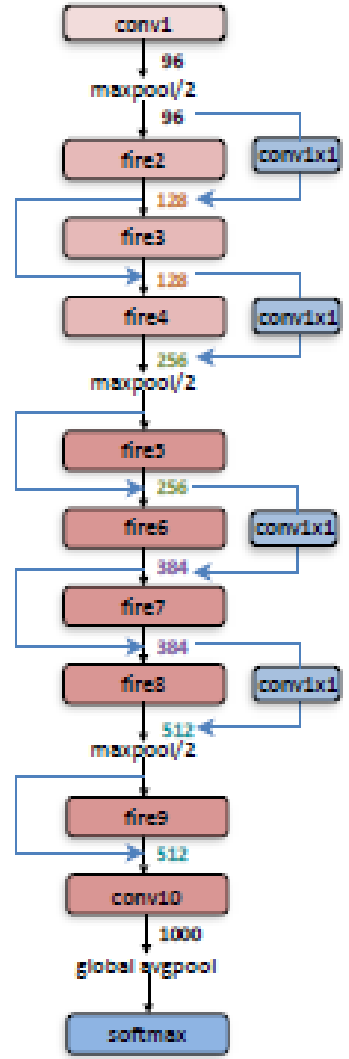
Iandola et al, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size", arXiv 2016







with simple bypass



with complex bypass

Iandola et al, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size", arXiv 2016

Sqeezenet

- Good compression, but still needs a lot of computation
  - MAC 861M ~= AlexNet

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%

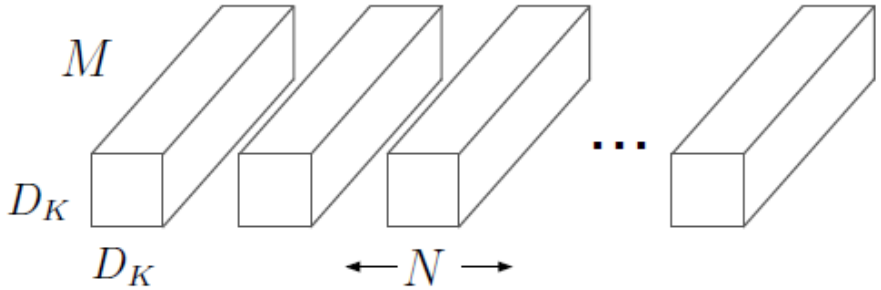
Iandola et al., "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size", arXiv 2016

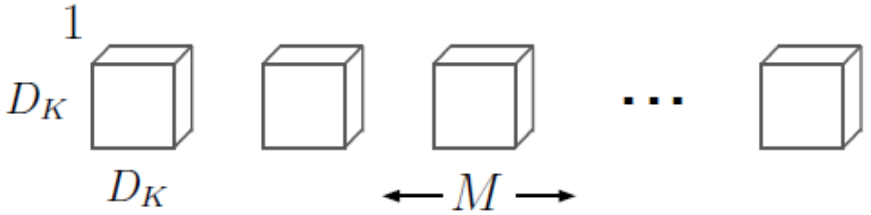
MobileNet

- M: input depth, N: output depth, D_k: kernel size, D_F: input feature size
- Standard convolution
  - 2D + depth
- MobileNet: 2D (per depth) => all depth
  - Depthwise separable convolution +
  - Concatenate output
  - 1x1 convolution (pointwise convolution)

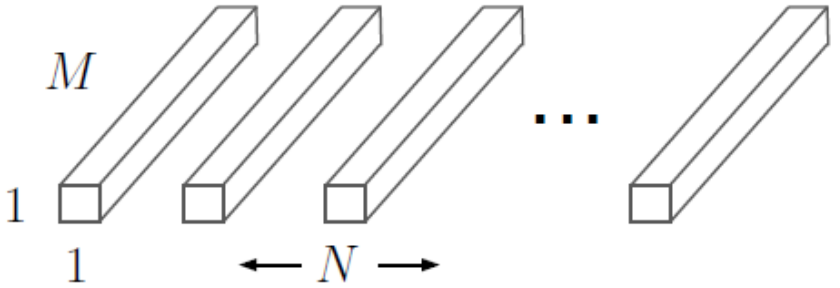
$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2}$$

W_in * H_in * Nch * k * k * Nk



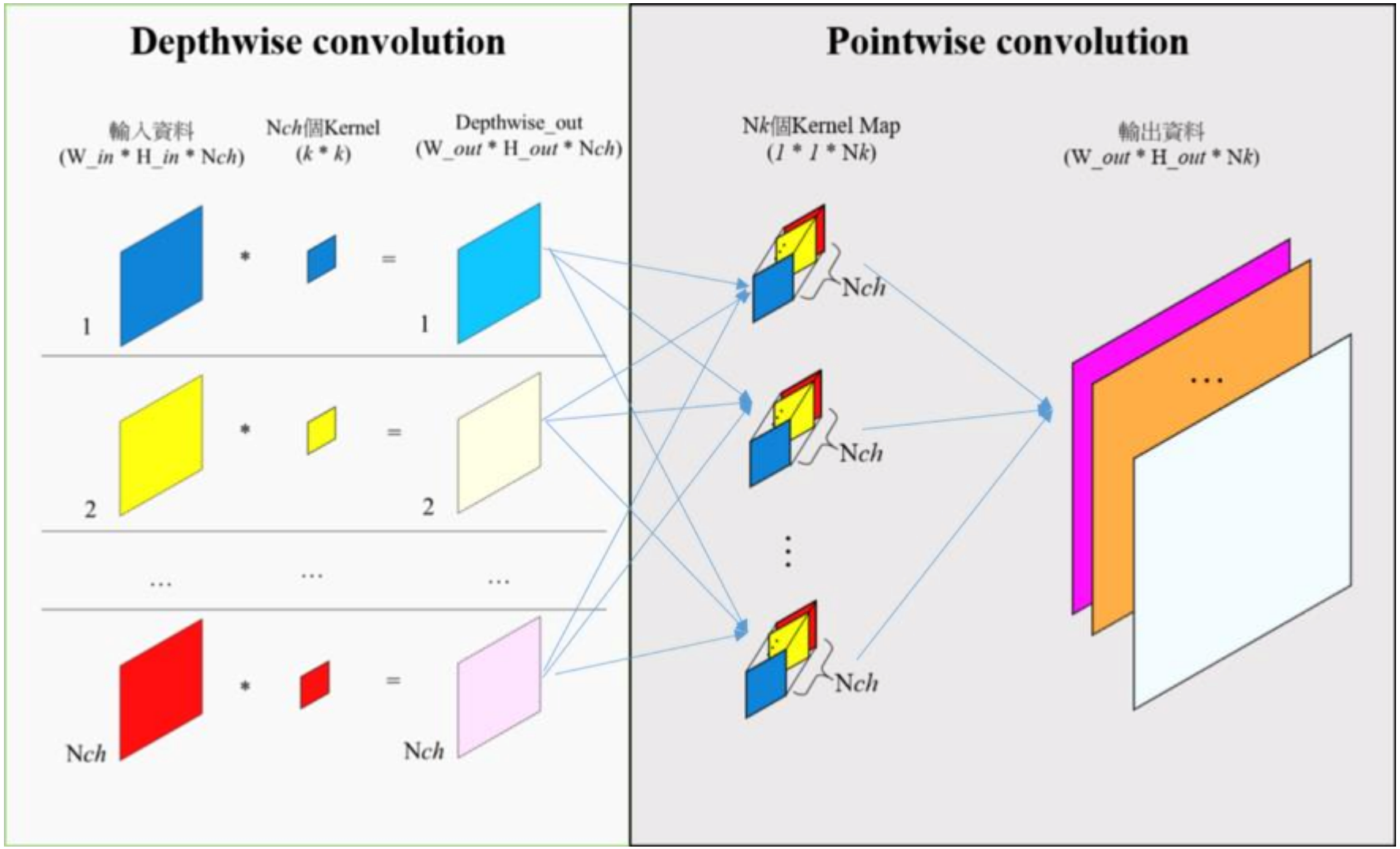


(b) Depthwise Convolutional Filters



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

Figure 2. The standard convolutional filters in (a) are replaced by two layers: depthwise convolution in (b) and pointwise convolution in (c) to build a depthwise separable filter.


Depthwise convolution
Pointwise convolution
輸入資料
(W_in * H_in * Nch)
Nch個Kernel
(k * k)
Depthwise_out
(W_out * H_out * Nch)
Nk個Kernel Map
(1 * 1 * Nk)
輸出資料
(W_out * H_out * Nk)
1
2
Nch
Nch
Nch
Nch
Nch

MobileNet

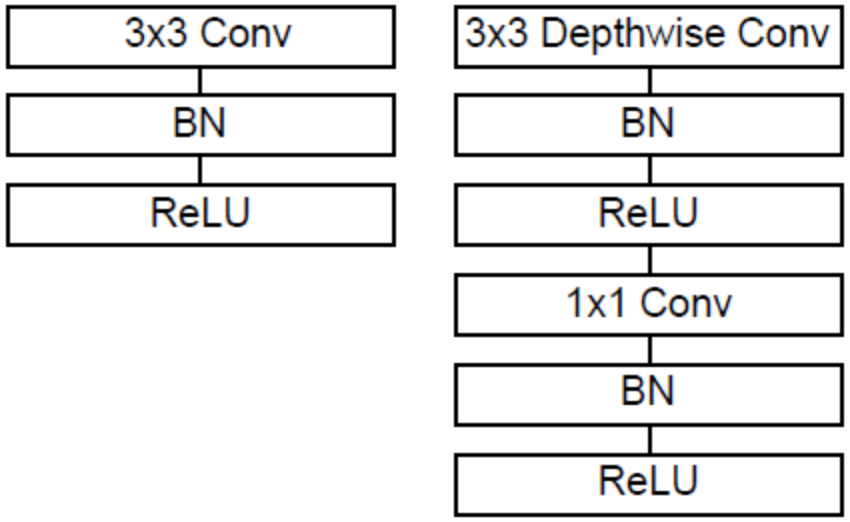


Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

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%

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	$1 \times 1 \times 1000$

MobileNet:

- Width Multiplier: Thinner Models
- Resolution Multiplier: Reduced Representation
- Both have Quadratic effect $D_K \cdot D_K \cdot \alpha M \cdot \rho D_F \cdot \rho D_F + \alpha M \cdot \alpha N \cdot \rho D_F \cdot \rho D_F$

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

MobileNet: Results

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

1/7 fewer parameters

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

Tall and thin model is better

Table 6. MobileNet Width Multiplier

Width Multiplier	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
0.75 MobileNet-224	68.4%	325	2.6
0.5 MobileNet-224	63.7%	149	1.3
0.25 MobileNet-224	50.6%	41	0.5

Table 7. MobileNet Resolution

Resolution	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
1.0 MobileNet-192	69.1%	418	4.2
1.0 MobileNet-160	67.2%	290	4.2
1.0 MobileNet-128	64.4%	186	4.2

MobileNet: Results

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

Table 9. Smaller MobileNet Comparison to Popular Models

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
0.50 MobileNet-160	60.2%	76	1.32
Squeezenet	57.5%	1700	1.25
AlexNet	57.2%	720	60

Table 13. COCO object detection results comparison using different frameworks and network architectures. mAP is reported with COCO primary challenge metric (AP at IoU=0.50:0.05:0.95)

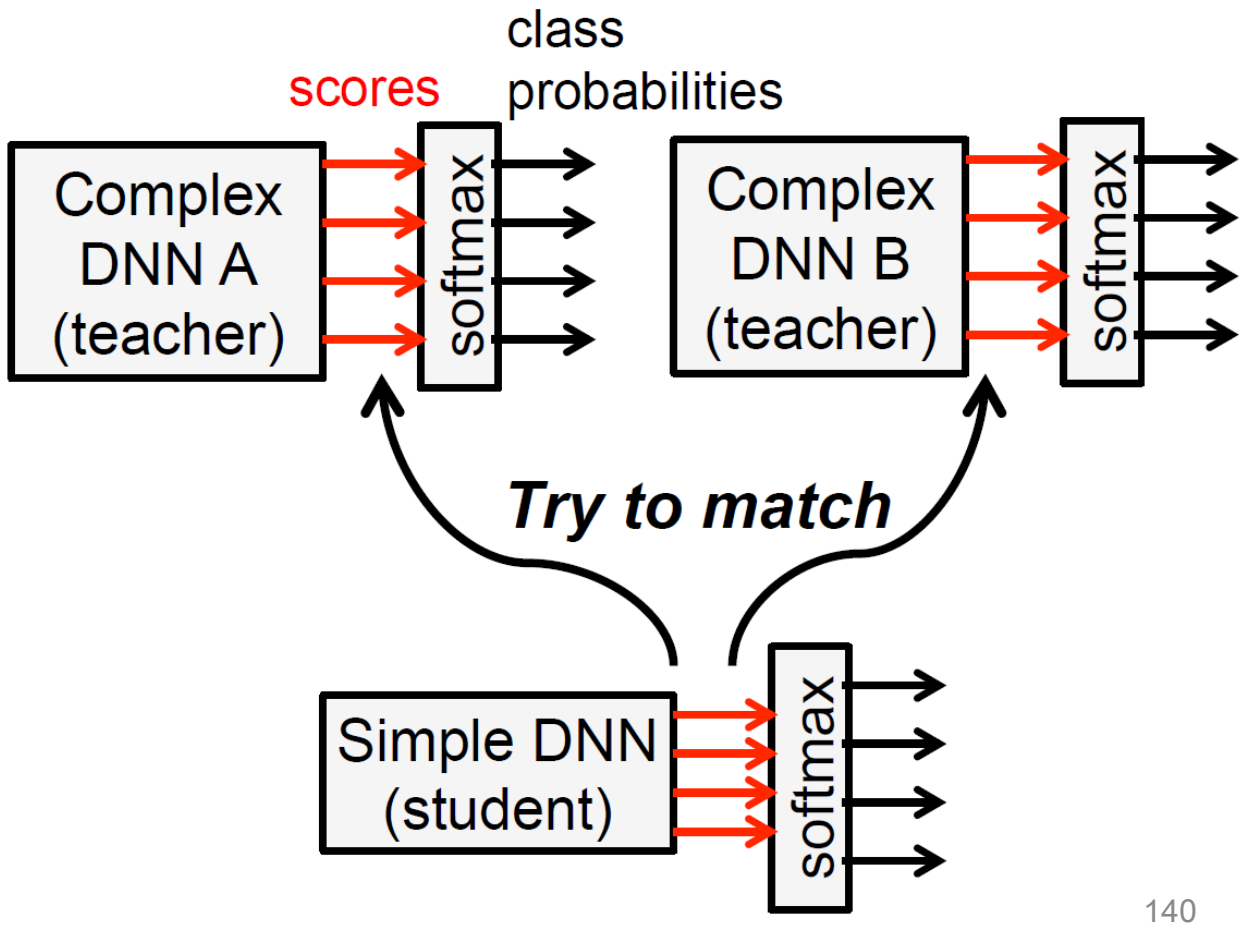
Framework Resolution	Model	mAP	Billion Mult-Adds	Million Parameters
SSD 300	deeplab-VGG	21.1%	34.9	33.1
	Inception V2	22.0%	3.8	13.7
	MobileNet	19.3%	1.2	6.8
Faster-RCNN 300	VGG	22.9%	64.3	138.5
	Inception V2	15.4%	118.2	13.3
	MobileNet	16.4%	25.2	6.1
Faster-RCNN 600	VGG	25.7%	149.6	138.5
	Inception V2	21.9%	129.6	13.3
	Mobilenet	19.8%	30.5	6.1

Table 14. MobileNet Distilled from FaceNet

Model	1e-4 Accuracy	Million Mult-Adds	Million Parameters
FaceNet [25]	83%	1600	7.5
1.0 MobileNet-160	79.4%	286	4.9
1.0 MobileNet-128	78.3%	185	5.5
0.75 MobileNet-128	75.2%	166	3.4
0.75 MobileNet-128	72.5%	108	3.8

Knowledge Distillation from Large model to Small Model

- Concept: Use a small student model to learn a large teacher model
  - Learn the score instead of final class



Source: Hardware Architectures for Deep Neural Networks

- MobileNet Example

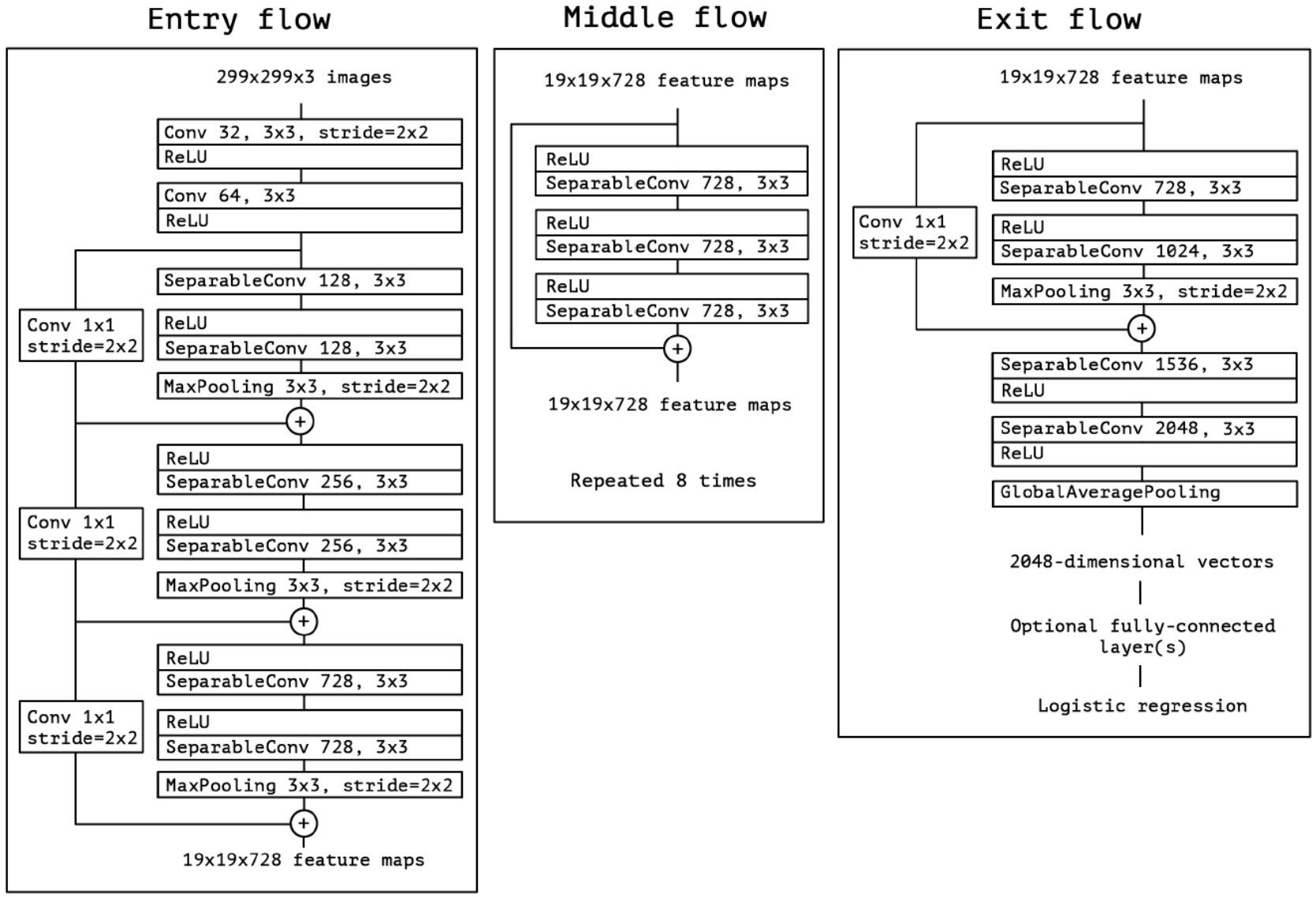
Table 12. Face attribute classification using the MobileNet architecture. Each row corresponds to a different hyper-parameter setting (width multiplier α and image resolution).

Width Multiplier / Resolution	Mean AP	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	88.7%	568	3.2
0.5 MobileNet-224	88.1%	149	0.8
0.25 MobileNet-224	87.2%	45	0.2
1.0 MobileNet-128	88.1%	185	3.2
0.5 MobileNet-128	87.7%	48	0.8
0.25 MobileNet-128	86.4%	15	0.2
Baseline	86.9%	1600	7.5

Table 14. MobileNet Distilled from FaceNet

Model	1e-4 Accuracy	Million Mult-Adds	Million Parameters
FaceNet [25]	83%	1600	7.5
1.0 MobileNet-160	79.4%	286	4.9
1.0 MobileNet-128	78.3%	185	5.5
0.75 MobileNet-128	75.2%	166	3.4
0.75 MobileNet-128	72.5%	108	3.8

Xception: Inception with Depthwise Convolution



Xception: Deep L

Figure 2. A simplified Inception module.

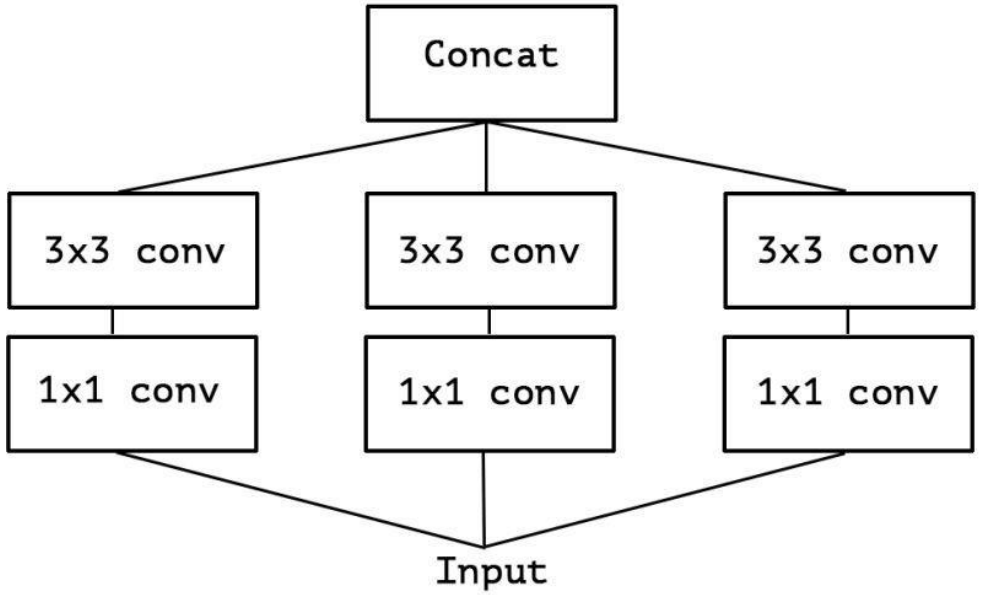


Figure 3. A strictly equivalent reformulation of the simplified Inception module.

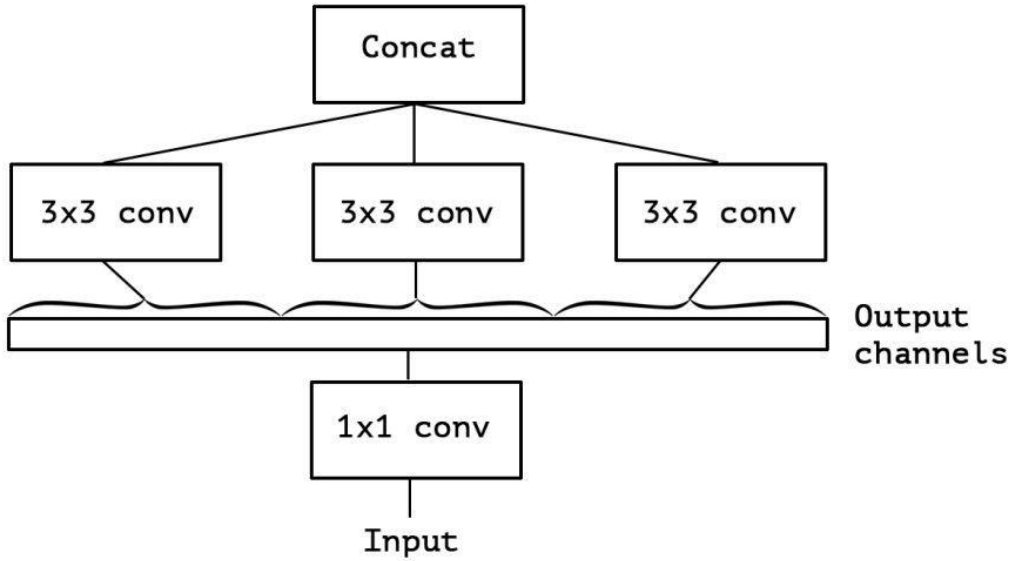
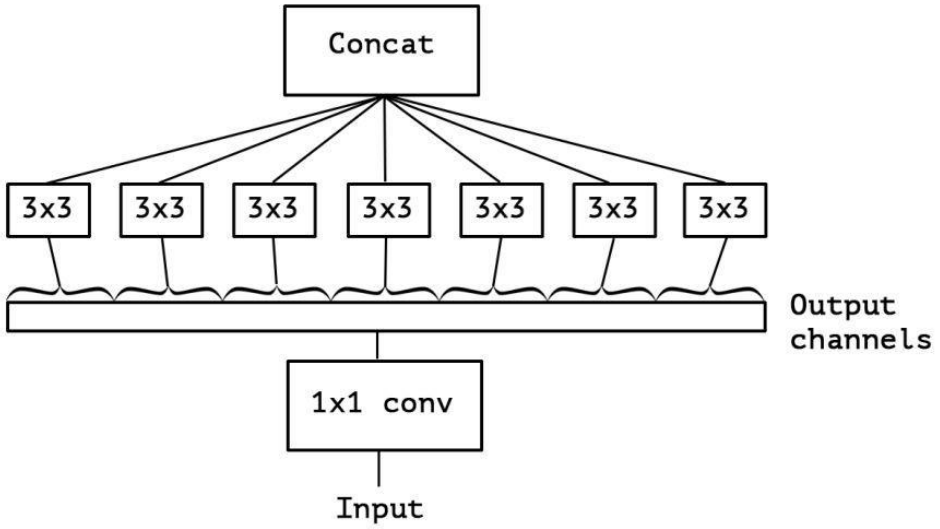
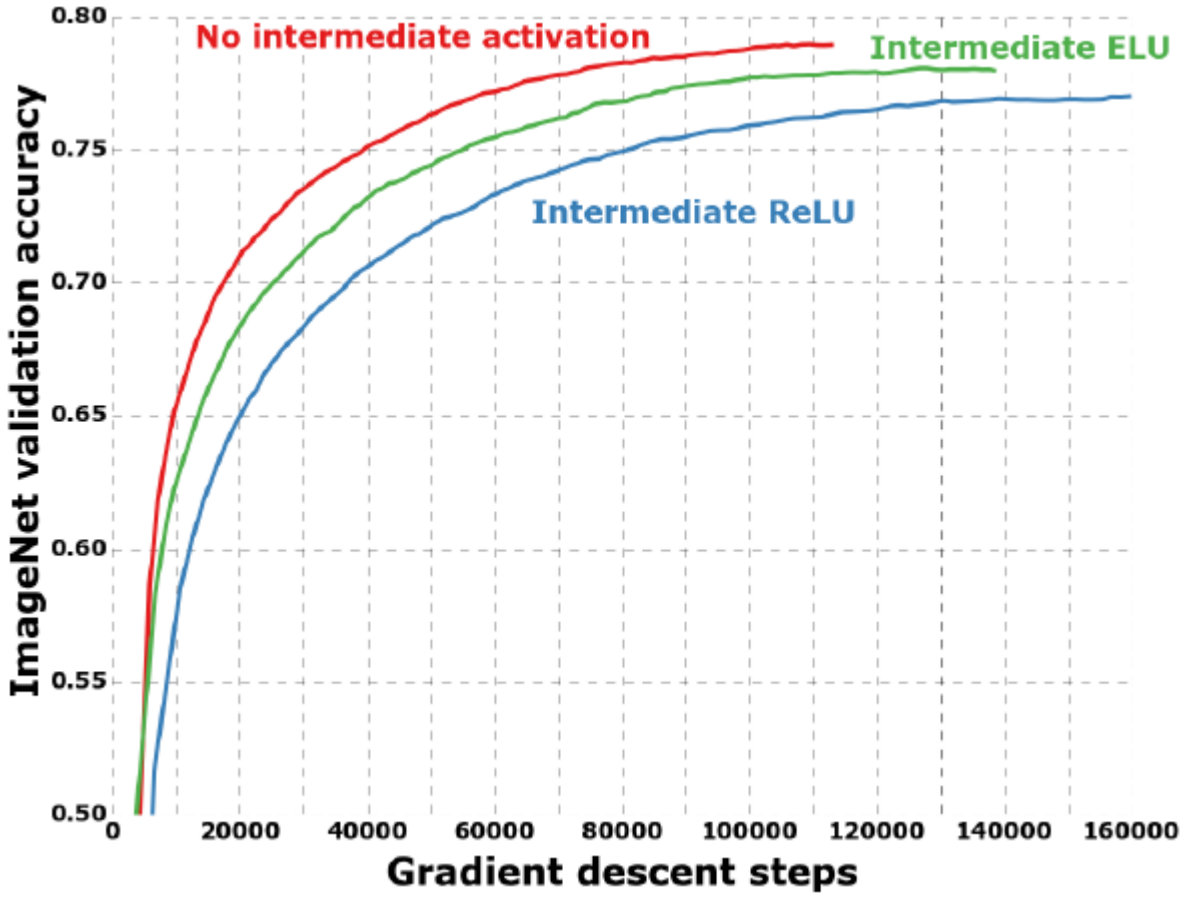


Figure 4. An "extreme" version of our Inception module, with one spatial convolution per output channel of the 1x1 convolution.



Xception

- No intermediate ReLU for depthwise convolution gets better results



ResNeXt: ResNet (Group Conv. Version)

Grouped convolutions

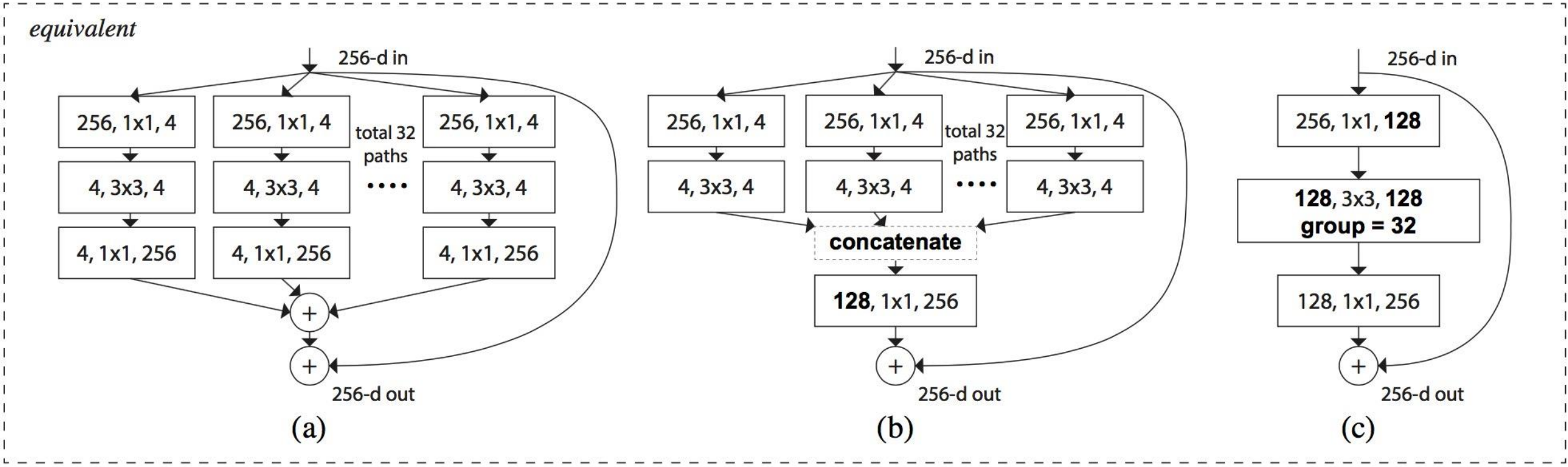


Figure 3. Equivalent building blocks of ResNeXt. **(a)**: Aggregated residual transformations, the same as Fig. 1 right. **(b)**: A block equivale to (a), implemented as early concatenation. **(c)**: A block equivalent to (a,b), implemented as grouped convolutions [23]. Notations in **bo** text highlight the reformulation changes. A layer is denoted as (# input channels, filter size, # output channels).

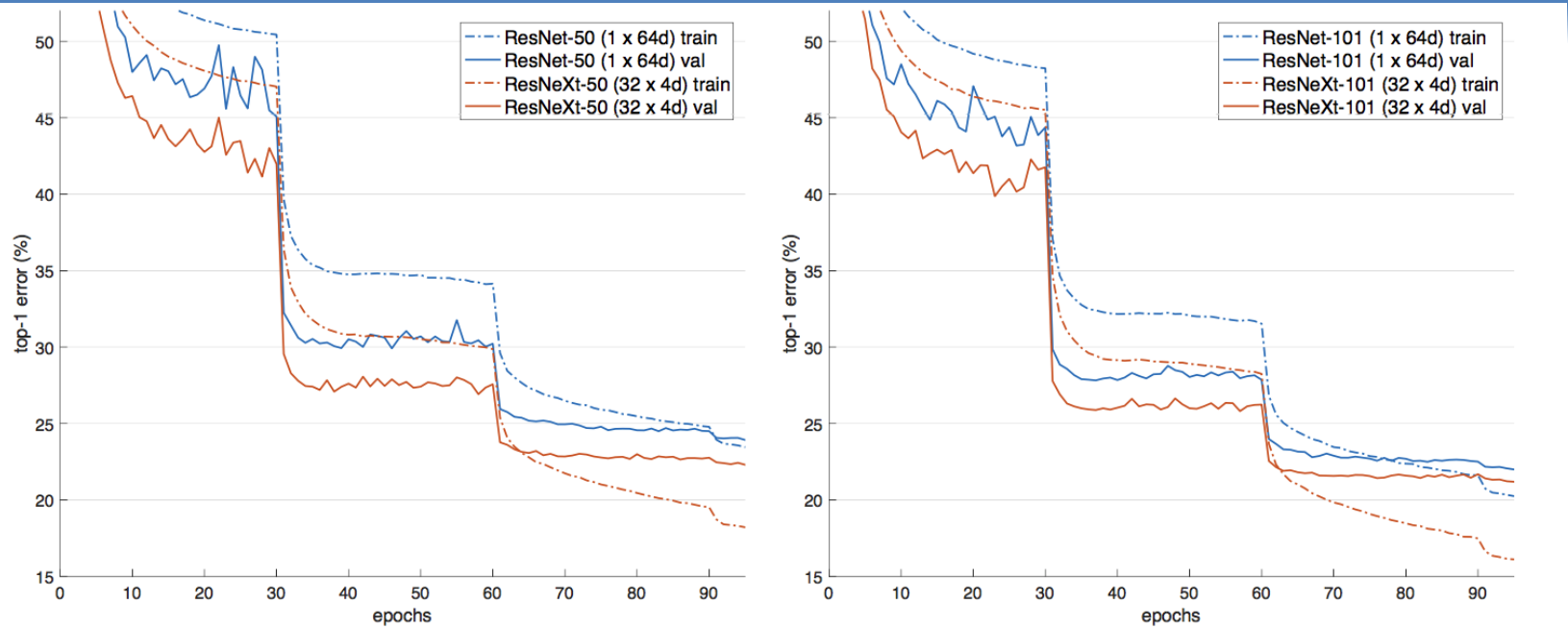


Figure 5. Training curves on ImageNet-1K. (**Left**): ResNet/ResNeXt-50 with preserved complexity (∼4.1 billion FLOPs, ∼25 million parameters); (**Right**): ResNet/ResNeXt-101 with preserved complexity (∼7.8 billion FLOPs, ∼44 million parameters).

	setting	top-1 error (%)
ResNet-50	1 × 64d	23.9
ResNeXt-50	2 × 40d	23.0
ResNeXt-50	4 × 24d	22.6
ResNeXt-50	8 × 14d	22.3
ResNeXt-50	32 × 4d	**22.2**
ResNet-101	1 × 64d	22.0
ResNeXt-101	2 × 40d	21.7
ResNeXt-101	4 × 24d	21.4
ResNeXt-101	8 × 14d	21.3
ResNeXt-101	32 × 4d	**21.2**

Table 3. Ablation experiments on ImageNet-1K. (**Top**): ResNet-50 with preserved complexity (∼4.1 billion FLOPs); (**Bottom**): ResNet-101 with preserved complexity (∼7.8 billion FLOPs). The error rate is evaluated on the single crop of 224×224 pixels.

	setting	top-1 err (%)	top-5 err (%)
1× complexity references:			
ResNet-101	1 × 64d	22.0	6.0
ResNeXt-101	32 × 4d	21.2	5.6
2× complexity models follow:			
ResNet-**200** [14]	1 × 64d	21.7	5.8
ResNet-101, wider	1 × **100**d	21.3	5.7
ResNeXt-101	**2** × 64d	20.7	5.5
ResNeXt-101	**64** × 4d	**20.4**	**5.3**

Table 4. Comparisons on ImageNet-1K when the number of FLOPs is increased to 2× of ResNet-101's. The error rate is evaluated on the single crop of 224×224 pixels. The highlighted factors are the factors that increase complexity.

ShuffleNet

- Problem of mobilenet: expensive pointwise (1x1) convolution
  - Concept: channel sparsity
  - 1st Approach: pointwise group convolution
  - Side effect: how to get features across group
  - 2nd Approach: channel shuffle
- Strategy
  - 1x1 convolution => group convolution

ShuffleNet

- divide the channels in each group into several subgroups, then
- feed each group in the next layer with different subgroups

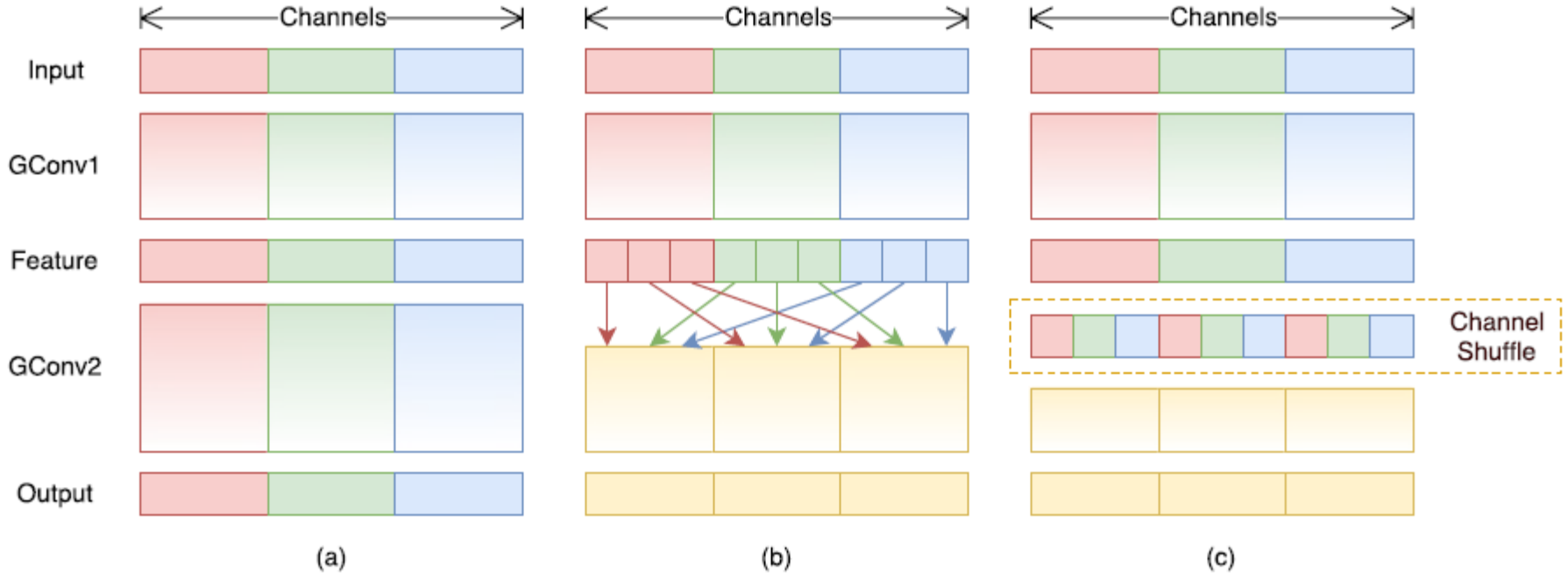


Figure 1: Channel shuffle with two stacked group convolutions. GConv stands for group convolution. a) two stacked convolution layers with the same number of groups. Each output channel only relates to the input channels within the group. No cross talk; b) input and output channels are fully related when GConv2 takes data from different groups after GConv1; c) an equivalent implementation to b) using channel shuffle.

ShuffleNet

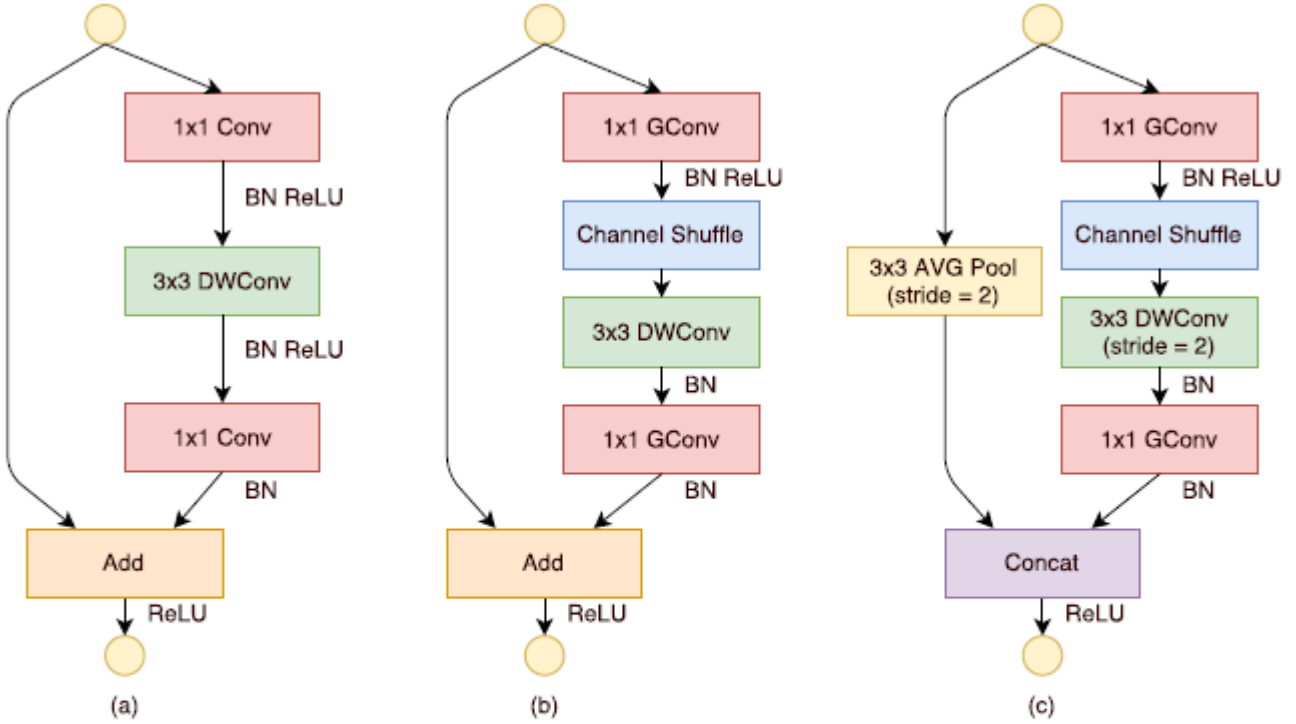


Figure 2: ShuffleNet Units. a) bottleneck unit [9] with depthwise convolution (DWConv) [3, 12]; b) ShuffleNet unit with pointwise group convolution (GConv) and channel shuffle; c) ShuffleNet unit with stride = 2.

ShuffleNet

Table 1: ShuffleNet architecture

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[1]	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[2]					143M	140M	137M	133M	137M

Importance of Pointwise Group Convolutions

- Scale factor *s* on channel number
- Smaller models tend to benefit more from groups
- Arch2: remove two units in stages 3 and widen each feature map.

Table 2: Classification error vs. number of groups g (*smaller number represents better performance*)

Model	Complexity (MFLOPs)	Classification error (%)				
		$g = 1$	$g = 2$	$g = 3$	$g = 4$	$g = 8$
ShuffleNet 1×	140	35.1	34.2	**34.1**	34.3	34.7
ShuffleNet 0.5×	38	46.1	45.1	44.4	**43.7**	43.8
ShuffleNet 0.25×	13	56.7	56.3	55.6	54.5	**53.7**
ShuffleNet 0.5× (arch2)	40	45.7	44.3	43.8	43.2	**42.7**
ShuffleNet 0.25× (arch2)	13	56.5	55.3	55.5	54.3	**53.3**

Xception-like

Table 4: Classification error vs. various structures (%, *smaller number represents better performance*)

Complexity (MFLOPs)	VGG-like[4]	ResNet	Xception-like	ResNeXt	ShuffleNet (ours)
140	56.0	38.7	35.1	34.3	**34.1** ($1\times$, $g = 3$)
38	-	48.9	46.1	46.3	**43.7** ($0.5\times$, $g = 4$)
13	-	61.6	56.7	59.2	**53.7** ($0.25\times$, $g = 8$)
40 (arch2)	-	48.5	45.7	47.2	**42.7** ($0.5\times$, $g = 8$)
13 (arch2)	-	61.3	56.5	61.0	**53.3** ($0.25\times$, $g = 8$)

Table 5: ShuffleNet vs. MobileNet [12] on ImageNet Classification

Model	Complexity (MFLOPs)	Cls err. (%)	Δ err. (%)
1.0 MobileNet-224	569	29.4	-
ShuffleNet $2\times$ ($g = 3$)	524	**29.1**	0.3
0.75 MobileNet-224	325	31.6	-
ShuffleNet $1.5\times$ ($g = 3$)	292	**31.0**	0.6
0.5 MobileNet-224	149	36.3	-
ShuffleNet $1\times$ ($g = 3$)	140	**34.1**	2.2
0.25 MobileNet-224	41	49.4	-
ShuffleNet $0.5\times$ (arch2, $g = 8$)	40	**42.7**	6.7
ShuffleNet $0.5\times$ (shallow, $g = 3$)	40	45.2	4.2

ShuffleNet

Table 6: Complexity comparison

Model	Cls err. (%)	Complexity (MFLOPs)
VGG-16 [27]	28.5	15300
ShuffleNet 2× ($g = 3$)	29.1	**524**
PVANET [18] (*our impl.*)	35.3	557
ShuffleNet 1× ($g = 3$)	34.1	**140**
AlexNet [19]	42.8	720
SqueezeNet [13]	42.5	833
ShuffleNet 0.5× (arch2, $g = 8$)	42.7	**40**

Table 7: Object detection results on MS COCO (*larger numbers represents better performance*)

Model	mAP [.5, .95] (300× image)	mAP [.5, .95] (600× image)
ShuffleNet 2× ($g = 3$)	**18.0%**	**24.5%**
ShuffleNet 1× ($g = 3$)	14.3%	19.8%
1.0 MobileNet-224 [12]	16.4%	19.8%

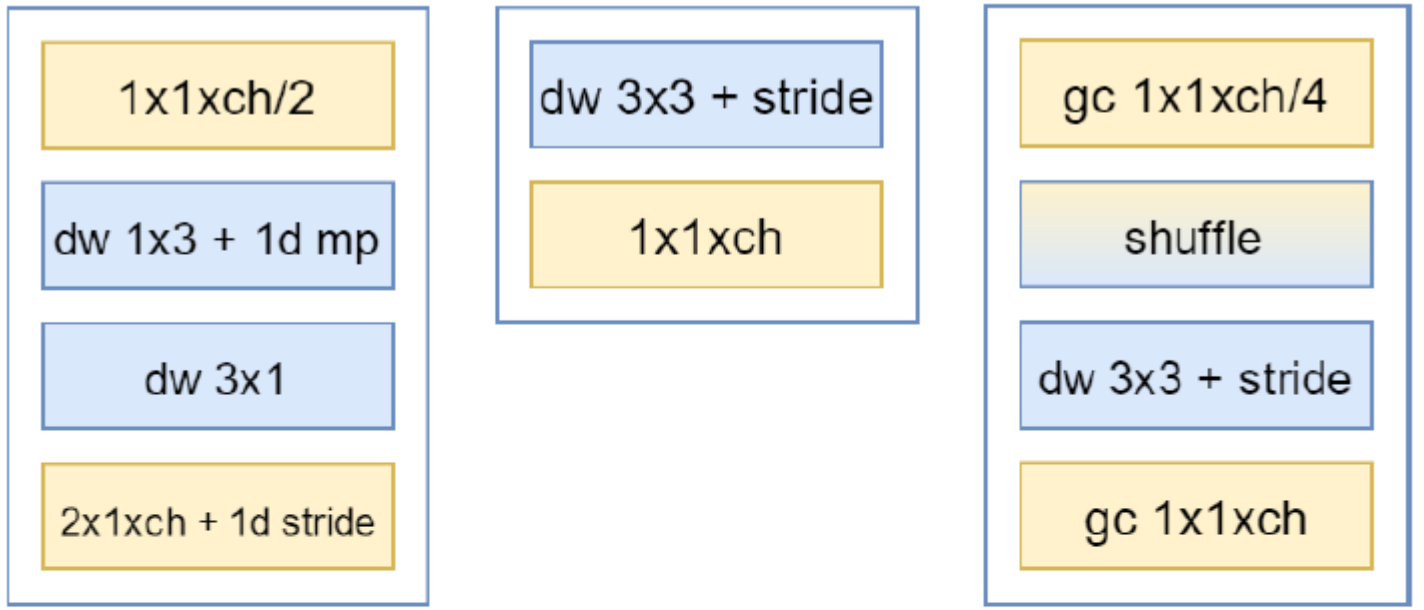
Table 8: Actual inference time on mobile device (*smaller number represents better performance*)

Model	Cls err. (%)	FLOPs	224 × 224	480 × 640	720 × 1280
ShuffleNet 0.5× (arch2, $g = 3$)	43.8	40M	15.2ms	87.4ms	260.1ms
ShuffleNet 1× ($g = 3$)	34.1	140M	37.8ms	222.2ms	684.5ms
AlexNet [19]	42.8	720M	184.0ms	1156.7ms	3633.9ms

How Group Convolution Saves Parameters

- Standard convolution
  - Parameters number $N_{in}*N_{out}*k*k \sim= N^2*k^2$
- Divide channel into g group
  - Parameter number $(N/g)^2*g*k^2$

EffeNet



(a) An EffNet block (ours) (b) A MobileNet block (c) A ShuffleNet block

Fig. 1: A comparison of MobileNet and ShuffleNet with our EffNet blocks. 'dw' means depthwise convolution, 'mp' means max-pooling, 'ch' is for the number of output channels and 'gc' is for group convolutions. Best seen in colour.

EffNet: Observations

- Not adopted due to performance loss
  - Group convolutions
  - Strides and pooling prone to aliasing
  - Residual connections (only beneficial in deeper networks)
- Adopted
  - Bottleneck factor of two (larger factor hurts)
  - Separable convolution
  - Early 2x1 pooling
  - EffNet in the first layer as well
    - Save 30% complexity of that layer

Table 1: Data flow analysis of selected models. One could intuitively understand how an aggressive data compression in early stages would harm accuracies. Compression factors of 4 or more are marked in red. *gc*4 means convolution in 4 groups. Best seen in colour.

Baseline		MobileNet [5]		ShuffleNet [6]		EffNet (Ours)	
Layer	Floats Out	Layer	Floats Out	Layer	Floats Out	Layer	Floats Out
3x3x64 + mp	16384	3x3x64 + mp	16384	3x3x64 + mp	16384	1x1x32	32768
						dw 1x3 + 1d mp	16384
						dw 3x1	16384
						2x1x64 + 1d stride	16384
3x3x128 + mp	8192	dw 3x3 + stride	4096	gc4 1x1x32	8192	1x1x64	16384
		1x1x128	8192	dw 3x3 + stride	2048	dw 1x3 + 1d mp	8192
				gc4 1x1x128	8192	dw 3x1	8192
						2x1x128 + 1d stride	8192
3x3x256 + mp	4096	dw 3x3 + stride	2048	gc4 1x1x64	4096	1x1x128	8192
		1x1x256	4096	dw 3x3 + stride	1024	dw 1x3 + 1d mp	4096
				gc4 1x1x256	4096	dw 3x1	4096
						2x1x256 + 1d stride	4096
Fully Connected	10	Fully Connected	10	Fully Connected	10	Fully Connected	10

Table 2: A comparison of the large and smaller on the Cifar10 dataset

	Mean Accuracy	Mil. FLOPs	Factor
Baseline	82.78%	80.3	1.00
EffNet large	**85.02%**	**79.8**	**0.99**
MobileNet	77.48%	5.8	0.07
ShuffleNet	77.30%	**4.7**	**0.06**
EffNet (ours)	**80.20%**	11.4	0.14

Table 3: A comparison of the large and smaller on the SVHN dataset

	Mean Accuracy	kFLOPs	Factor
Baseline	91.08%	3,563.5	1.00
EffNet large	**91.12%**	**3,530.7**	**0.99**
MobileNet	85.64%	773.4	0.22
ShuffleNet	82.73%	733.1	0.21
EffNet (ours)	**88.51%**	**517.6**	**0.14**

Table 5: A comparison of MobileNet v2 and EffNet on the Cifar10 dataset for various expansion rates

Ex. Rate		Mean Acc.	Mil. Flops	Fact.
	Baseline	82.78%	80.3	1.00
6	EffNet	**83.20%**	44.1	0.55
	MobileNet v2	79.10%	**42.0**	**0.52**
4	EffNet	**82.45%**	31.1	0.39
	MobileNet v2	78.91%	**29.2**	**0.36**
2	EffNet	**81.67%**	18.1	0.22
	MobileNet v2	76.47%	**16.4**	**0.20**
6	mob_imp	84.25%	44.0	0.55

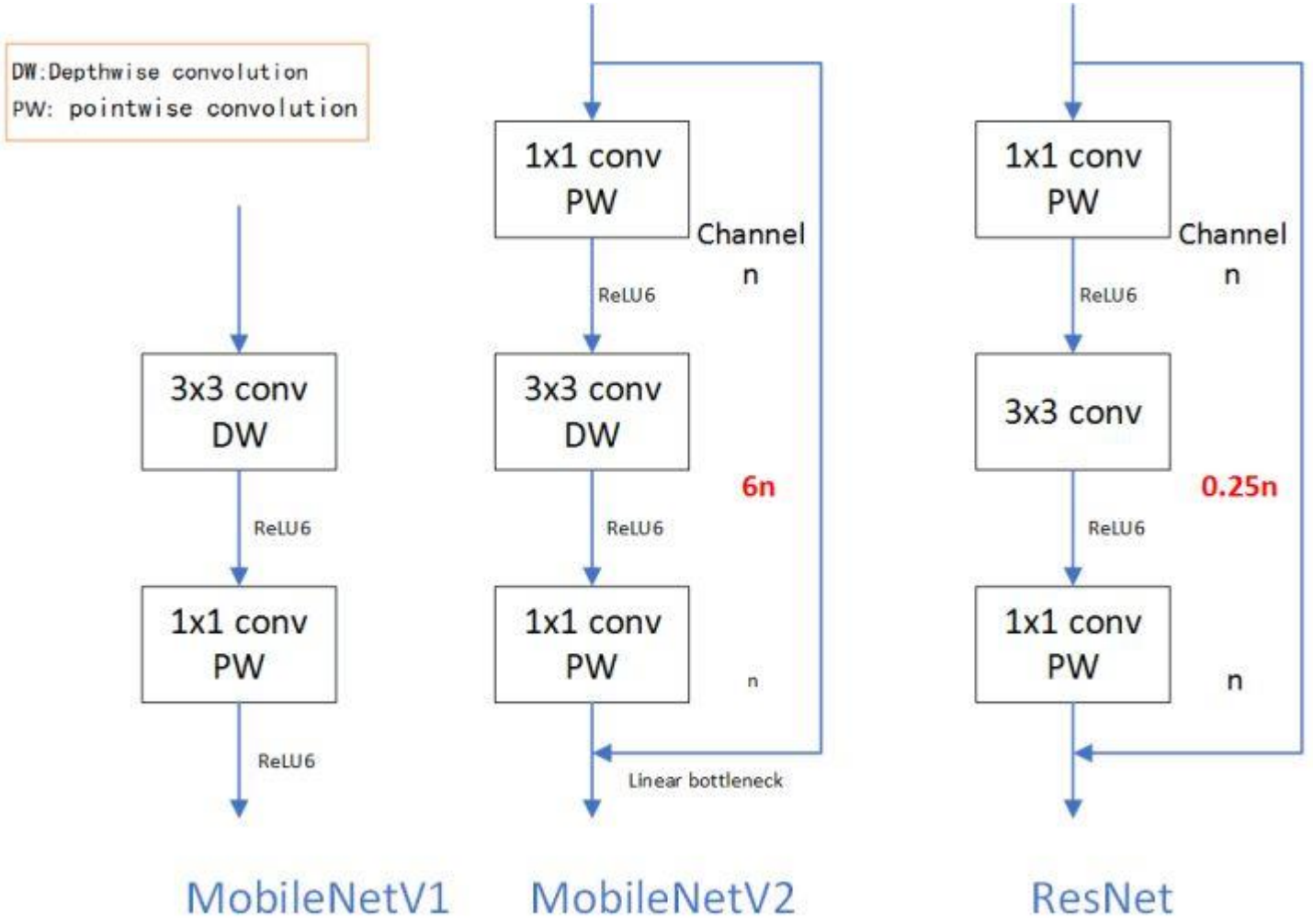
Problems of MobileNet v1 & Why MobileNet v2

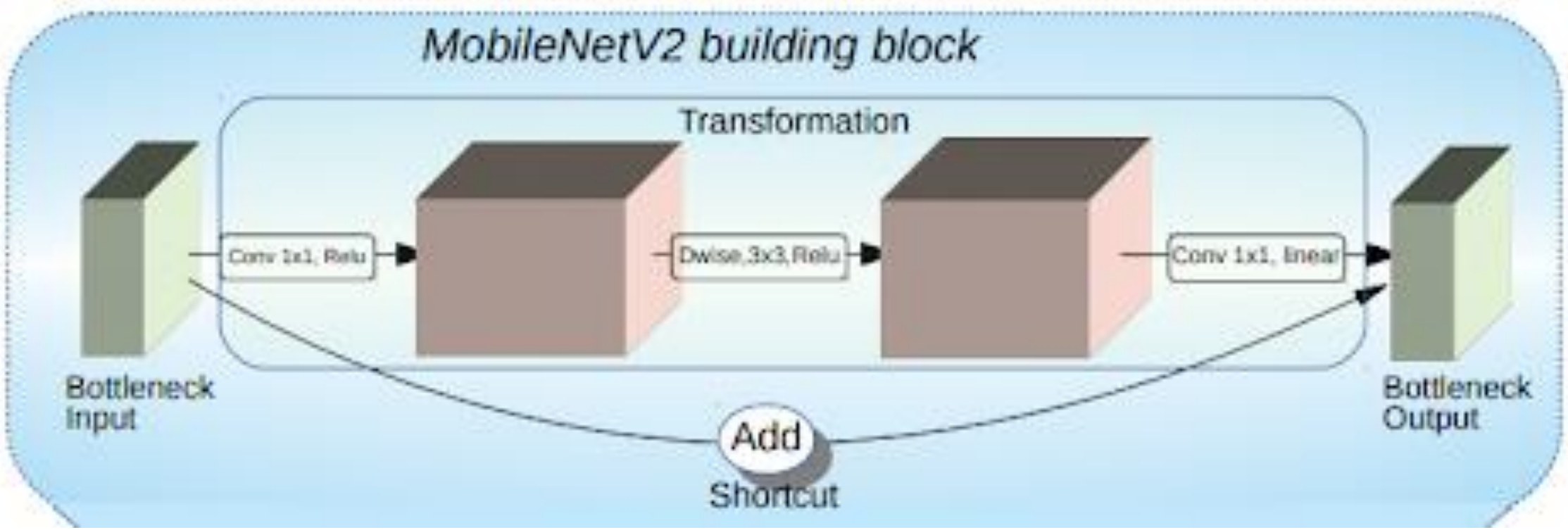
MobileNet v1

- Structure
  - Structure is too simple like VGG, low c/p ratio (performance v.s. parameters)
  - Move to better one like ResNet, DenseNet
- Potential problems of depth wise convolution
  - Easy to be degenerated due to smaller kernel dimension and ReLU

MobileNet v2

- Inverted residual
  - Short connection + wide middle layer
- Linear bottleneck
  - Reduce effect of ReLU (ReLU is not good for depthwise convolution_


DW:Depthwise convolution
PW: pointwise convolution
3x3 conv
DW
ReLU6
1x1 conv
PW
ReLU6
MobileNetV1
1x1 conv
PW
Channel
n
ReLU6
3x3 conv
DW
6n
ReLU6
1x1 conv
PW
n
Linear bottleneck
MobileNetV2
1x1 conv
PW
Channel
n
ReLU6
3x3 conv
0.25n
ReLU6
1x1 conv
PW
n
ResNet


MobileNetV2 building block
Transformation
Conv 1x1, Relu
Dwise,3x3,Relu
Conv 1x1, linear
Bottleneck Input
Bottleneck Output
Add
Shortcut

- inverted residual with linear bottleneck
  - inverted residual structure where the shortcut connections are between the thin bottleneck
  - intermediate expansion layer uses lightweight depthwise convolution
  - Remove nonlinearitis in the narrow layer to keep representation power

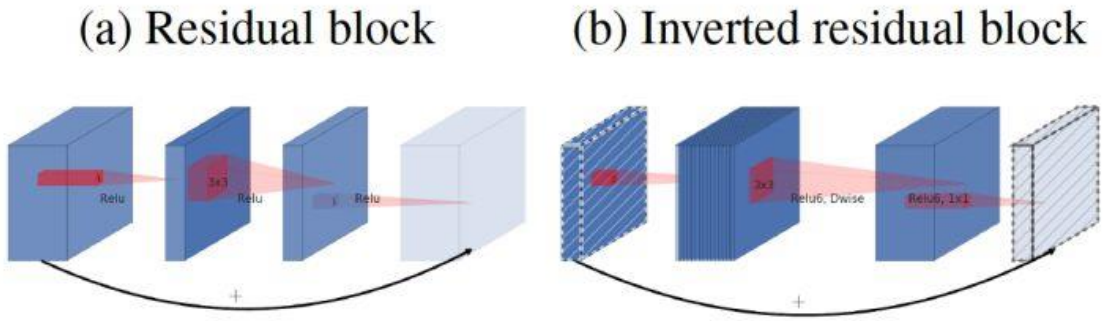
Expressness v.s. Capacity

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	$\frac{h}{s} \times \frac{w}{s} \times k'$

Table 1: *Bottleneck residual block* transforming from k to k' channels, with stride s, and expansion factor t.

Capacity
Expressness
Capacity

- Inverted residual allows separation of expressness and capacity
  - t=0, shortcut
  - t<1, classical residual
  - t>1, inverted residual, the most useful



MobileNet v2

- Inverted residual

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	-	

Network	Top 1	Params	MAdds	CPU
MobileNetV1	70.6	4.2M	575M	113ms
ShuffleNet (1.5)	71.5	**3.4M**	292M	-
ShuffleNet (x2)	73.7	5.4M	524M	-
NasNet-A	74.0	5.3M	564M	183ms
MobileNetV2	**72.0**	**3.4M**	**300M**	**75ms**
MobileNetV2 (1.4)	**74.7**	6.9M	585M	**143ms**

Table 4: Performance on ImageNet, comparison for dif ferent networks. As is common practice for ops, w count the total number of Multiply-Adds. In the las column we report running time in milliseconds (ms) fo a single large core of the Google Pixel 1 phone (using TF-Lite). We do not report ShuffleNet numbers as effi cient group convolutions and shuffling are not yet sup ported.

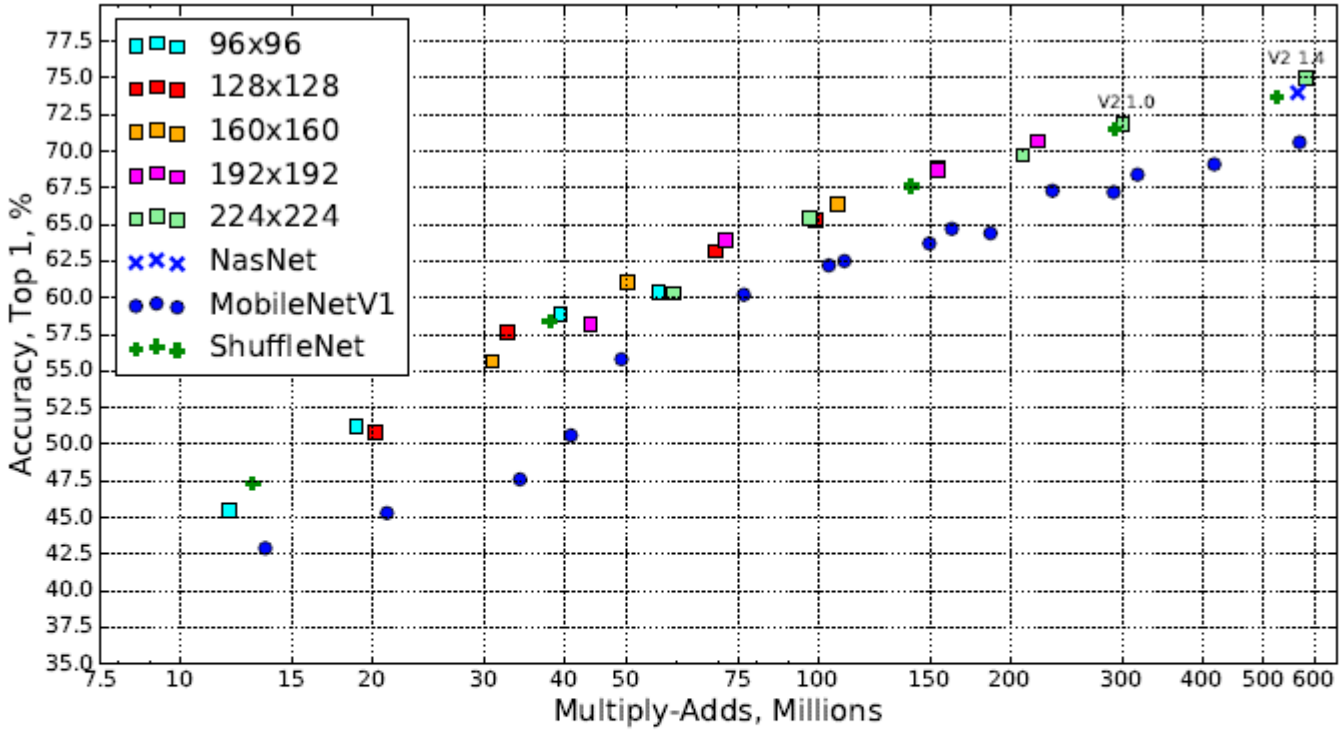
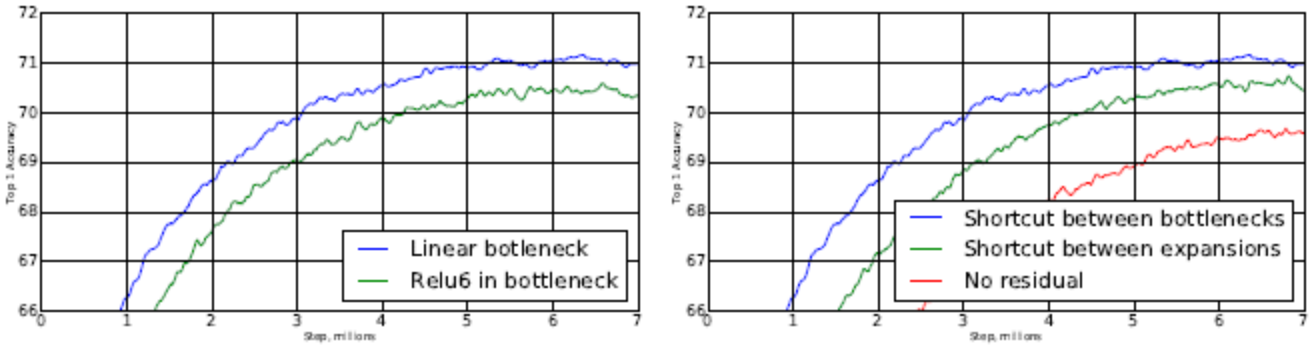


Figure 5: Performance curve of MobileNetV2 vs MobileNetV1, ShuffleNet, NAS. For our networks we use multipliers 0.35, 0.5, 0.75, 1.0 for all resolutions, and additional 1.4 for for 224. Best viewed in color.



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

(b) Impact of variations in residual blocks.

Figure 6: The impact of non-linearities and various types of shortcut (residual) connections.

	Params	MAdds
SSD[34]	14.8M	1.25B
SSDLite	**2.1M**	**0.35B**

Table 5: Comparison of the size and the computational cost between SSD and SSDLite configured with MobileNetV2 and making predictions for 80 classes.

Network	mAP	Params	MAdd	CPU
SSD300[34]	23.2	36.1M	35.2B	-
SSD512[34]	26.8	36.1M	99.5B	-
YOLOv2[35]	21.6	50.7M	17.5B	-
MNet V1 + SSDLite	22.2	5.1M	1.3B	270ms
MNet V2 + SSDLite	22.1	**4.3M**	**0.8B**	200ms

Table 6: Performance comparison of MobileNetV2 + SSDLite and other realtime detectors on the COCO dataset object detection task. MobileNetV2 + SSDLite achieves competitive accuracy with significantly fewer parameters and smaller computational complexity. All models are trained on `trainval35k` and evaluated on `test-dev`. SSD/YOLOv2 numbers are from [35]. The running time is reported for the large core of the Google Pixel 1 phone, using an internal version of the TF-Lite engine.

Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions

- Followed by a 1x1 convolution for shift

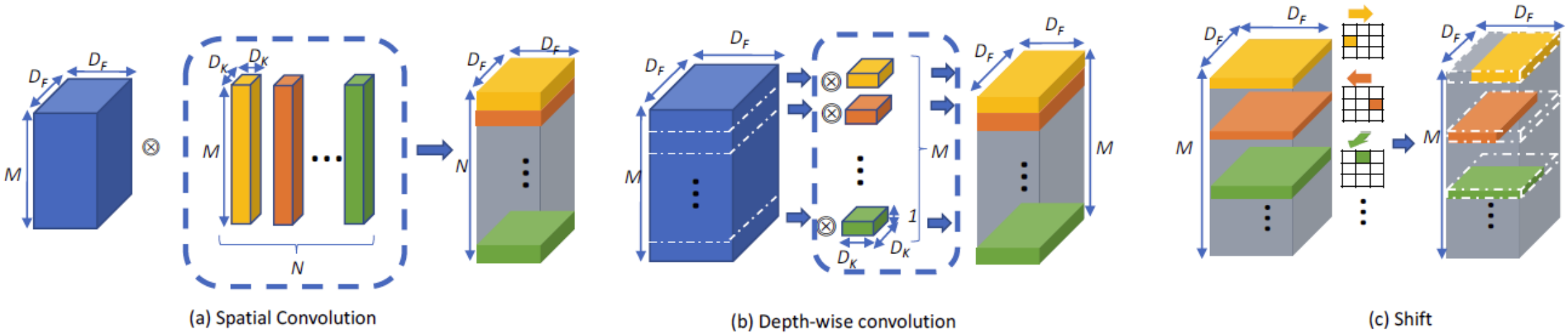


Figure 2: Illustration of (a) spatial convolutions, (b) depth-wise convolutions and (c) shift. In (c), the 3x3 grids denote a shift matrix with a kernel size of 3. The lighted cell denotes a 1 at that position and white cells denote 0s.

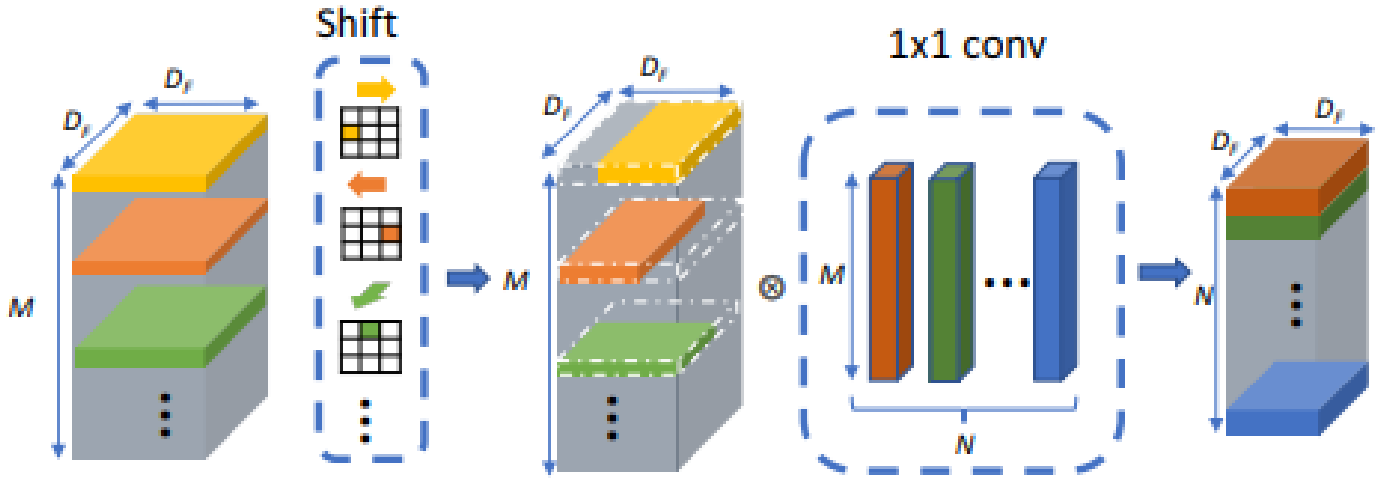
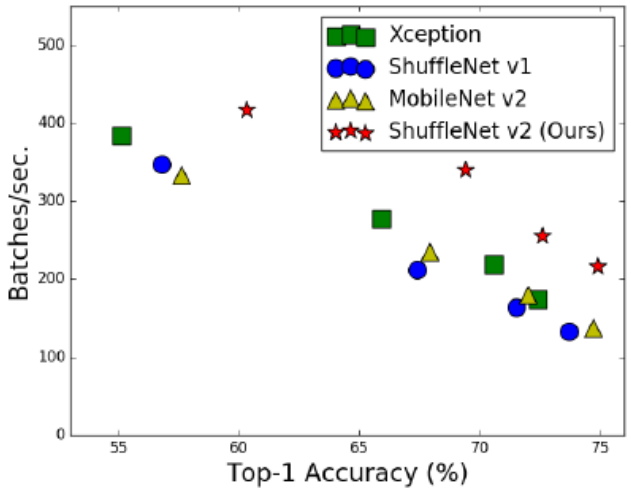


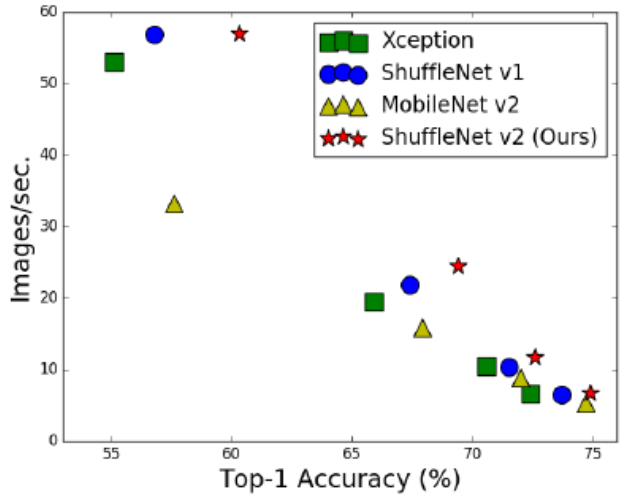
Figure 1: Illustration of a shift operation followed by a 1x1 convolution. The shift operation adjusts data spatially and the 1x1 convolution mixes information across channels.

ShuffleNet v2

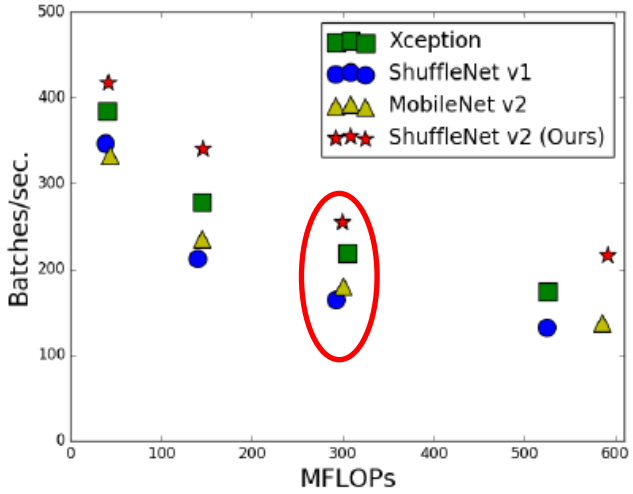
- accuracy, speed and FLOPs
- Metrics for complexity
  - Indirect: FLOPS
  - Direct: execution time/ speed/ latency
  - Same flops but different speed



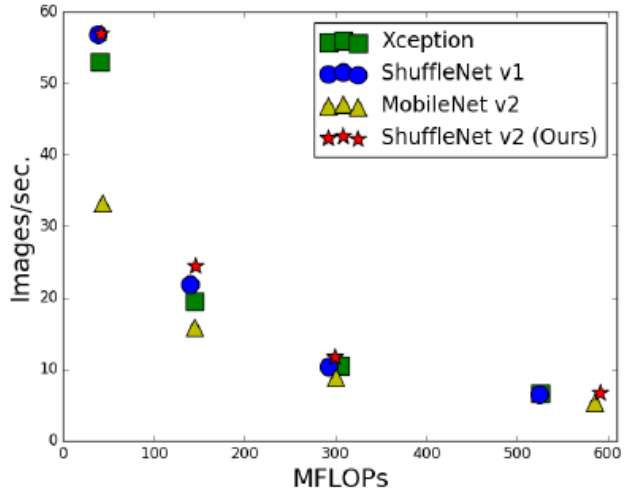
(a) GPU



(b) ARM



(c) GPU



(d) ARM

- Metrics for complexity
  - Indirect: FLOPS
  - Direct: execution time/ speed/ latency
  - Same flops different speed
    - CUDNN specially optimized for 3x3
    - We cannot certainly think that 3x3 conv is 9 times slower than 1x1 conv
- What missed
  - Memory access cost
    - Significant part in low complexity layer: Group convolution
    - Bottleneck for strong computing devices, eg. GPU
  - Degree of parallelism
    - Layers with high parallelism run faster than one with low parallelism

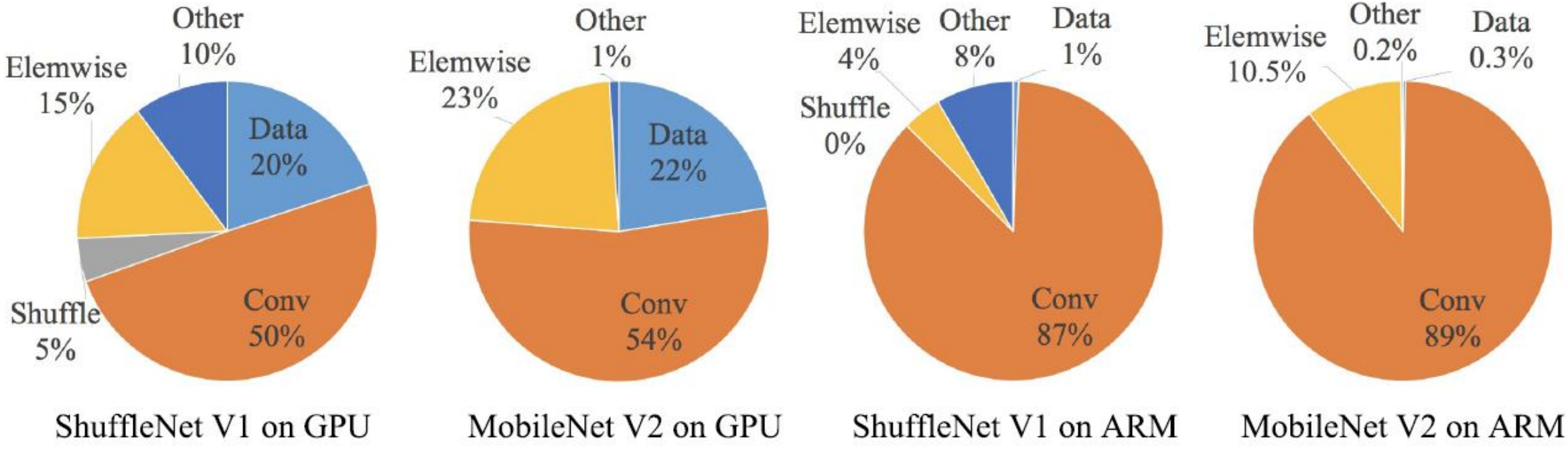


Fig. 2: Run time decomposition on two representative state-of-the-art network architectures, *ShuffeNet v1* [15] ($1\times$, $g = 3$) and *MobileNet v2* [14] ($1\times$).

Four Guideline for Network Designs

- G1: Equal channel width minimizes memory access cost
  - Assume depthwise conv + 1x1
  - Focus on channel no. of 1x1 convolution (account for most of the complexity)

		GPU (Batches/sec.)				ARM (Images/sec.)		
c1:c2	(c1,c2) for ×1	×1	×2	×4	(c1,c2) for ×1	×1	×2	×4
1:1	(128,128)	1480	723	232	(32,32)	76.2	21.7	5.3
1:2	(90,180)	1296	586	206	(22,44)	72.9	20.5	5.1
1:6	(52,312)	876	489	189	(13,78)	69.1	17.9	4.6
1:12	(36,432)	748	392	163	(9,108)	57.6	15.1	4.4

Table 1: Validation experiment for **Guideline 1**. Four different ratios of number of input/output channels ($c1$ and $c2$) are tested, while the total FLOPs under the four ratios is fixed by varying the number of channels. Input image size is 56×56.

卷積層的輸入輸出特徵通道數對MAC指標的影響。結論是卷積層的輸入和輸出特徵通道數相等時MAC最小，此時模型速度最快

- FLOPS
  - 1x1, B = hwc1c2
- MAC (memory access cost)
  - MAC = feature map + kernel = hw(c1+c2) + c1c2
- By mean value theorem

$$\mathrm{MAC} \geq 2\sqrt{hwB} + \frac{B}{hw}.$$

  - Substitue MAC and B to above equation, get (c1-c2) ^2>=0
    - C1 = C2 is the lower bound
  - C1: c2 = 1: 1 has the lowest execution time

- G2) Excessive group convolution increases MAC.

$$\begin{aligned}\text{MAC} &= hw(c_1 + c_2) + \frac{c_1 c_2}{g} \\ &= hwc_1 + \frac{Bg}{c_1} + \frac{B}{hw},\end{aligned}$$

- MAC increases with the growth of g if others fixed

在B不變時，g越大，MAC也越大

- C = c1 + c2
- Same flops

		GPU (Batches/sec.)				CPU (Images/sec.)		
g	c for ×1	×1	×2	×4	c for ×1	×1	×2	×4
1	128	2451	1289	437	64	40.0	10.2	2.3
2	180	1725	873	341	90	35.0	9.5	2.2
4	256	1026	644	338	128	32.9	8.7	2.1
8	360	634	445	230	180	27.8	7.5	1.8

Table 2: Validation experiment for **Guideline 2**. Four values of group number g are tested, while the total FLOPs under the four values is fixed by varying the total channel number c. Input image size is 56×56.

卷積的group操作對MAC的影響。結論是過多的group操作會增大MAC，從而使模型速度變慢

- G3) Network fragmentation reduces degree of parallelism.

	GPU (Batches/sec.)			CPU (Images/sec.)		
	c=128	c=256	c=512	c=64	c=128	c=256
1-fragment	2446	1274	434	40.2	10.1	2.3
2-fragment-series	1790	909	336	38.6	10.1	2.2
4-fragment-series	752	745	349	38.4	10.1	2.3
2-fragment-parallel	1537	803	320	33.4	9.1	2.2
4-fragment-parallel	691	572	292	35.0	8.4	2.1

Table 3: Validation experiment for **Guideline 3**. *c* denotes the number of channels for *1-fragment*. The channel number in other fragmented structures is adjusted so that the FLOPs is the same as *1-fragment*. Input image size is 56×56.

關於模型設計的分支數量對模型速度的影響。結論是模型中的分支數量越少，模型速度越快。

- G4) Element-wise operations are non-negligible
  - Depthwise conv is regarded as element wise operation (low FLOPs but high MAC).

		GPU (Batches/sec.)			CPU (Images/sec.)		
ReLU	short-cut	c=32	c=64	c=128	c=32	c=64	c=128
yes	yes	2427	2066	1436	56.7	16.9	5.0
yes	no	2647	2256	1735	61.9	18.8	5.2
no	yes	2672	2121	1458	57.3	18.2	5.1
no	no	2842	2376	1782	66.3	20.2	5.4

Table 4: Validation experiment for **Guideline 4**. The ReLU and shortcut operations are removed from the "bottleneck" unit [4], separately. c is the number of channels in unit. The unit is stacked repeatedly for 10 times to benchmark the speed.

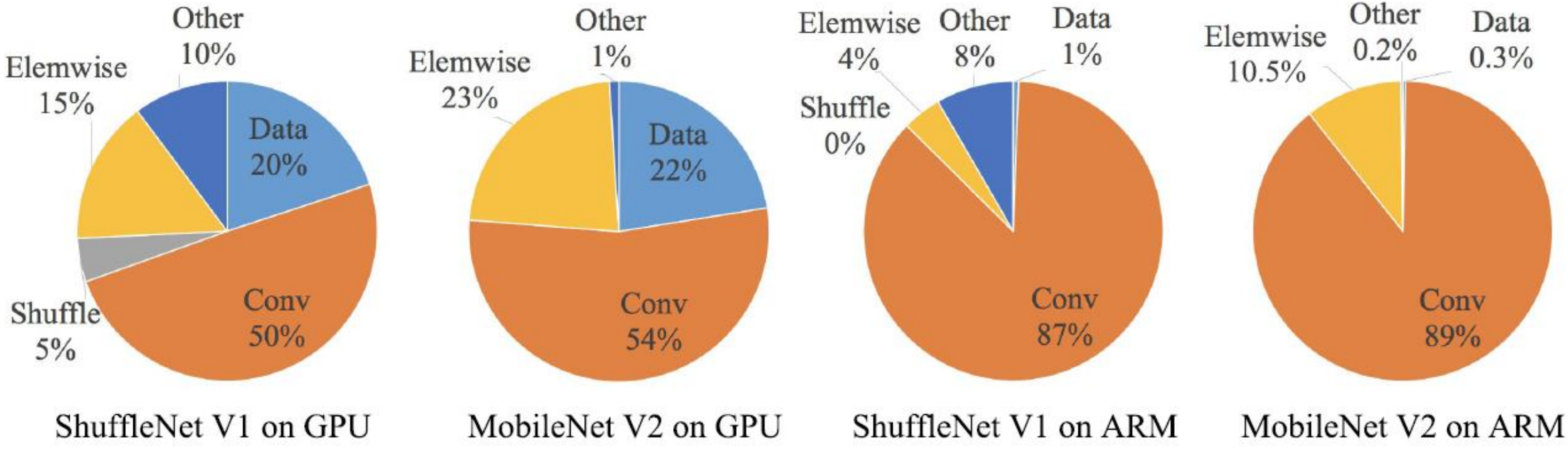


Fig. 2: Run time decomposition on two representative state-of-the-art network architectures, *ShuffeNet v1* [15] ($1\times$, $g = 3$) and *MobileNet v2* [14] ($1\times$).

Some discussions

- Shuffnet v1
  - Heavily depends on group convolutions (against G2)
  - And bottleneck like blocks (against G1)
- MobileNet v2
  - Inverted bottleneck (against G1)
  - Depthwise and ReLUs on thick feature maps (against G4)
- Nasnet
  - Fragmented structure (against G3)

ShuffleNet v2

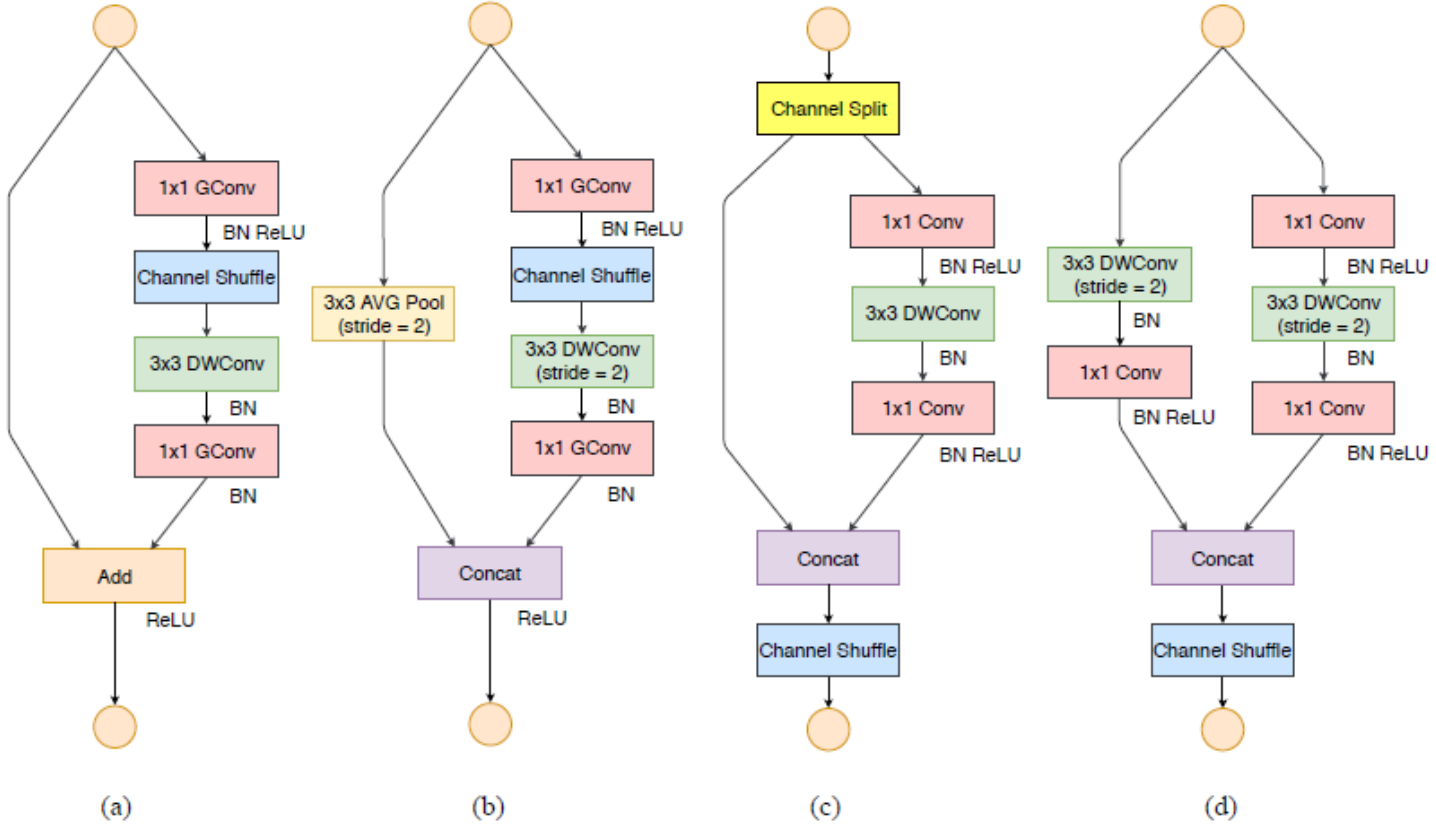


Fig. 3: Building blocks of ShuffleNet v1 [15] and this work. (a): the basic ShuffleNet unit; (b) the ShuffleNet unit for spatial down sampling (2×); (c) our basic unit; (d) our unit for spatial down sampling (2×). **DWConv**: depthwise convolution. **GConv**: group convolution.

Layer	Output size	KSize	Stride	Repeat	Output channels			
					0.5×	1×	1.5×	2×
Image	224×224				3	3	3	3
Conv1	112×112	3×3	2	1	24	24	24	24
MaxPool	56×56	3×3	2					
Stage2	28×28		2	1	48	116	176	244
	28×28		1	3				
Stage3	14×14		2	1	96	232	352	488
	14×14		1	7				
Stage4	7×7		2	1	192	464	704	976
	7×7		1	3				
Conv5	7×7	1×1	1	1	1024	1024	1024	2048
GlobalPool	1×1	7×7						
FC					1000	1000	1000	1000
FLOPs					41M	146M	299M	591M
# of Weights					1.4M	2.3M	3.5M	7.4M

Model	FLOPs	Top-1 err. (%)
ShuffleNet v2-50 (ours)	**2.3G**	**22.8**
ShuffleNet v1-50 [15] (our impl.)	**2.3G**	25.2
ResNet-50 [4]	3.8G	24.0
SE-ShuffleNet v2-164 (ours, with residual)	**12.7G**	**18.56**
SENet [8]	20.7G	18.68

Table 6: Results of large models. See text for details.

Model	mmAP(%)				GPU Speed (Images/sec.)			
FLOPs	40M	140M	300M	500M	40M	140M	300M	500M
Xception	21.9	29.0	31.3	32.9	178	131	101	83
ShuffleNet v1	20.9	27.0	29.9	32.9	152	85	76	60
MobileNet v2	20.7	24.4	30.0	30.6	146	111	94	72
ShuffleNet v2 (ours)	22.5	29.0	31.8	33.3	**188**	**146**	**109**	**87**
ShuffleNet v2* (ours)	**23.7**	**29.6**	**32.2**	**34.2**	183	138	105	83

Table 7: Performance on COCO object detection. The input image size is 800 × 1200. *FLOPs* row lists the complexity levels at 224 × 224 input size. For GPU speed evaluation, the batch size is 4. We do not test ARM because the *PSRoI Pooling* operation needed in [34] is unavailable on ARM currently.

Model	Complexity (MFLOPs)	Top-1 err. (%)	GPU Speed (Batches/sec.)	ARM Speed (Images/sec.)
ShuffleNet v2 0.5× (ours)	41	**39.7**	417	**57.0**
0.25 MobileNet v1 [13]	41	49.4	**502**	36.4
0.4 MobileNet v2 [14] (our impl.)*	43	43.4	333	33.2
0.15 MobileNet v2 [14] (our impl.)	39	55.1	351	33.6
ShuffleNet v1 0.5× (g=3) [15]	38	43.2	347	56.8
DenseNet 0.5× [6] (our impl.)	42	58.6	366	39.7
Xception 0.5× [12] (our impl.)	40	44.9	384	52.9
IGCV2-0.25 [27]	46	45.1	183	31.5
ShuffleNet v2 1× (ours)	146	**30.6**	341	**24.4**
0.5 MobileNet v1 [13]	149	36.3	**382**	16.5
0.75 MobileNet v2 [14] (our impl.)**	145	32.1	235	15.9
0.6 MobileNet v2 [14] (our impl.)	141	33.3	249	14.9
ShuffleNet v1 1× (g=3) [15]	140	32.6	213	21.8
DenseNet 1× [6] (our impl.)	142	45.2	279	15.8
Xception 1× [12] (our impl.)	145	34.1	278	19.5
IGCV2-0.5 [27]	156	34.5	132	15.5
IGCV3-D (0.7) [28]	210	31.5	143	11.7

MobileNetV3

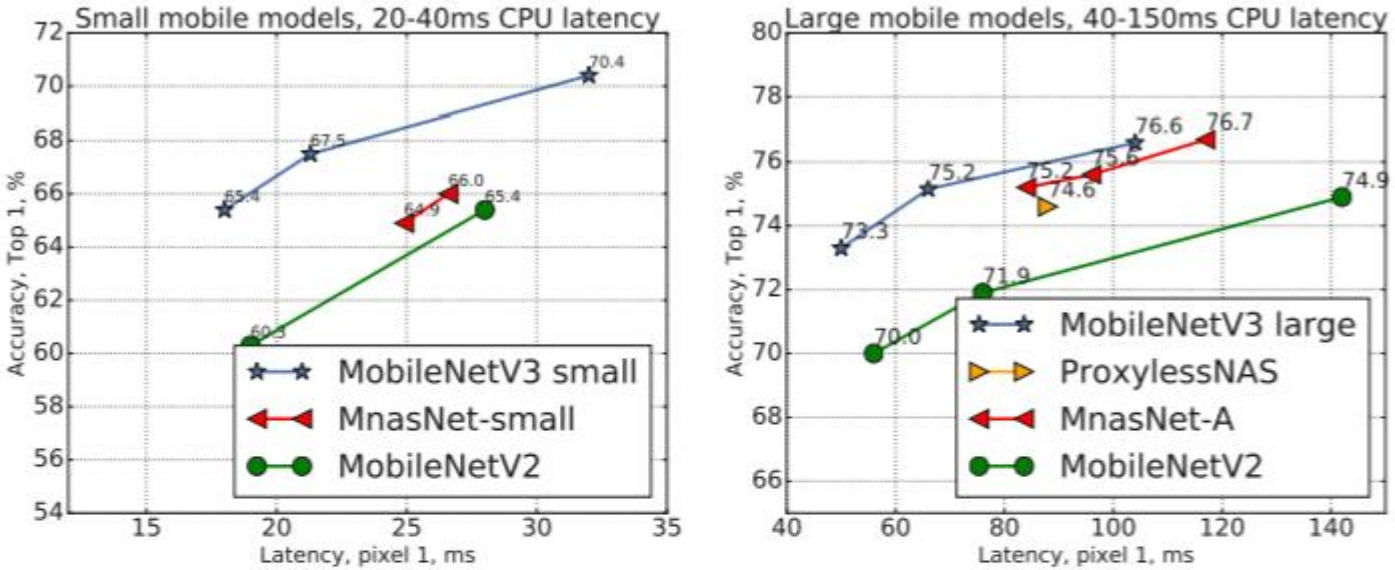


Figure 1. The trade-off between Pixel 1 latency and top-1 ImageNet accuracy. All models use the input resolution 224. V3 large and V3 small use multipliers 0.75, 1 and 1.25 to show optimal frontier. All latencies were measured on a single large core of the same device using TFLite[1]. MobileNetV3-Small and Large are our proposed next-generation mobile models.

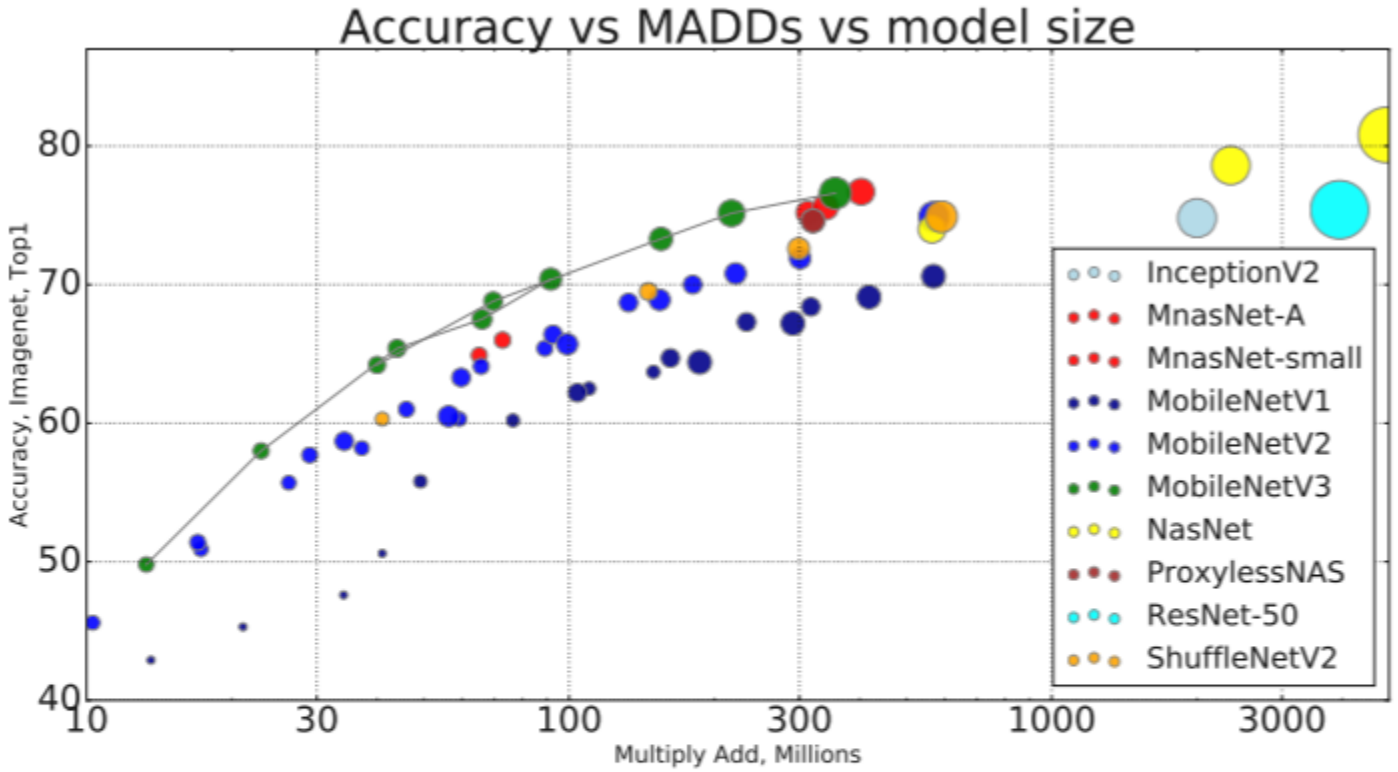


Figure 2. The trade-off between MAdds and top-1 accuracy. This allows to compare models that were targeted different hardware or software frameworks. All MobileNetV3 are for input resolution 224 and use multipliers 0.35, 0.5, 0.75, 1 and 1.25. See section 6 for other resolutions. Best viewed in color.

- MobileNet v1
  - Depthwise separable conv
- MobileNet v2
  - linear bottleneck
  - inverted residual structure
- MNasNet
  - lightweight attention modules based on squeeze and excitation into the bottleneck structure.
- MobileNet v3
  - Combination
  - Modifed swish (hard sigmoid)

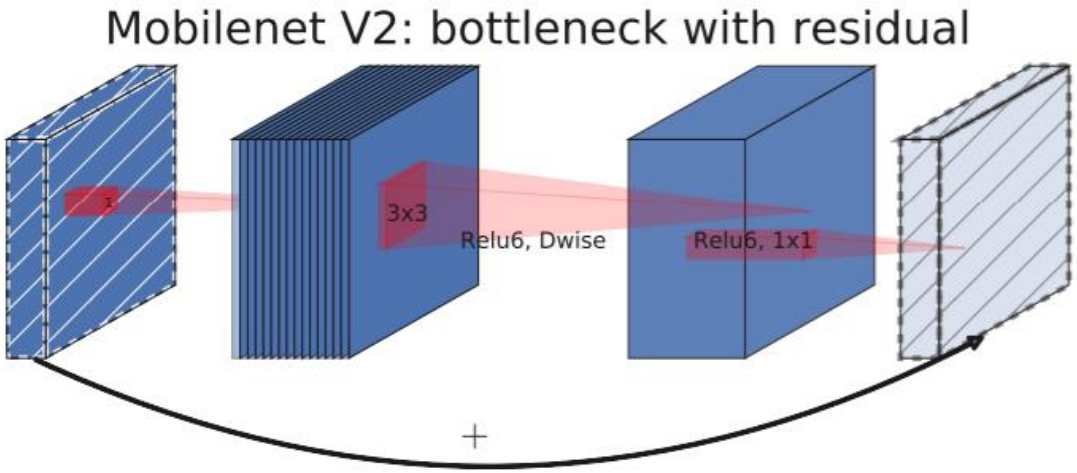


Figure 3. MobileNetV2 [39] layer (Inverted Residual and Linear Bottleneck). Each block consists of narrow input and output (bottleneck), which don't have nonlinearity, followed by expansion to a much higher-dimensional space and projection to the output. The residual connects bottleneck (rather than expansion).

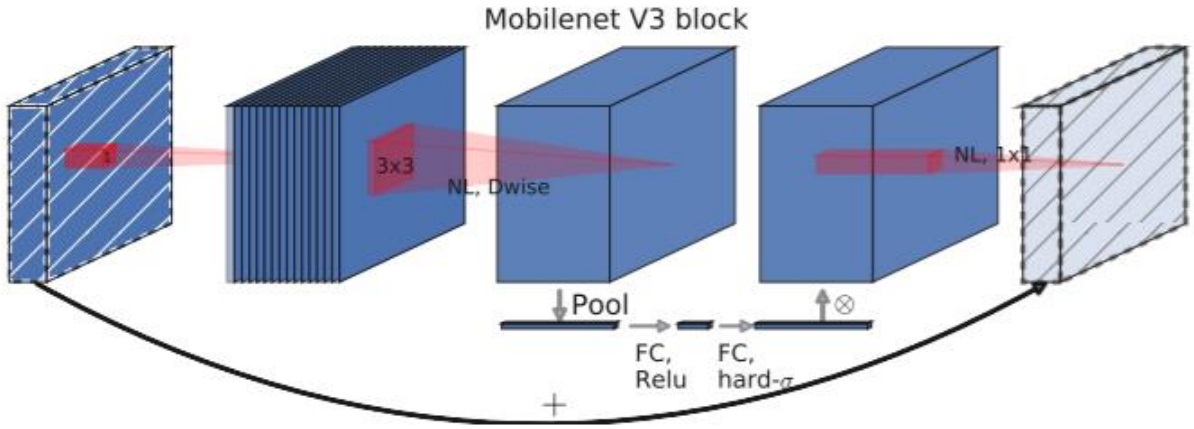


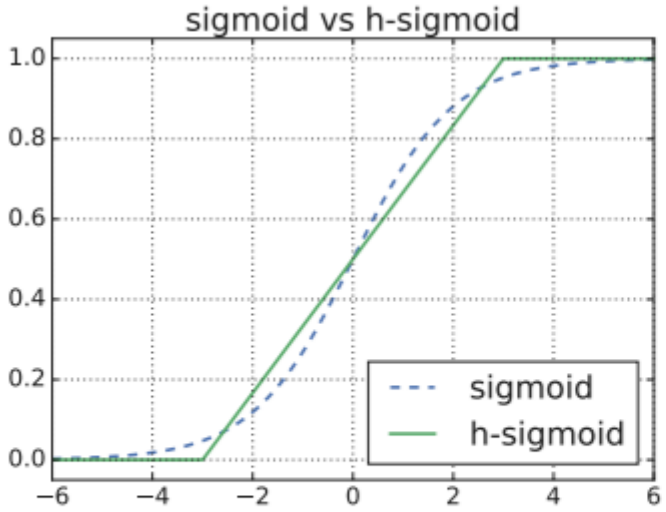
Figure 4. MobileNetV2 + Squeeze-and-Excite [20]. In contrast with [20] we apply the squeeze and excite in the residual layer. We use different nonlinearity depending on the layer, see section 5.2 for details.

Input	Operator	exp size	#out	SE	NL	s
$224^2 \times 3$	conv2d	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	-	RE	1
$112^2 \times 16$	bneck, 3x3	64	24	-	RE	2
$56^2 \times 24$	bneck, 3x3	72	24	-	RE	1
$56^2 \times 24$	bneck, 5x5	72	40	✓	RE	2
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 3x3	240	80	-	HS	2
$14^2 \times 80$	bneck, 3x3	200	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	480	112	✓	HS	1
$14^2 \times 112$	bneck, 3x3	672	112	✓	HS	1
$14^2 \times 112$	bneck, 5x5	672	160	✓	HS	2
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	conv2d, 1x1	-	960	-	HS	1
$7^2 \times 960$	pool, 7x7	-	-	-	-	1
$1^2 \times 960$	conv2d 1x1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv2d 1x1, NBN	-	k	-	-	1

Table 1. Specification for MobileNetV3-Large. SE denotes whether there is a Squeeze-And-Excite in that block. NL denotes the type of nonlinearity used. Here, HS denotes h-swish and RE denotes ReLU. NBN denotes no batch normalization. s denotes

Input	Operator	exp size	#out	SE	NL	s
$224^2 \times 3$	conv2d, 3x3	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	✓	RE	2
$56^2 \times 16$	bneck, 3x3	72	24	-	RE	2
$28^2 \times 24$	bneck, 3x3	88	24	-	RE	1
$28^2 \times 24$	bneck, 5x5	96	40	✓	HS	2
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	120	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	144	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	288	96	✓	HS	2
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	conv2d, 1x1	-	576	✓	HS	1
$7^2 \times 576$	pool, 7x7	-	-	-	-	1
$1^2 \times 576$	conv2d 1x1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv2d 1x1, NBN	-	k	-	-	1

Table 2. Specification for MobileNetV3-Small. See table 1 fo notation.



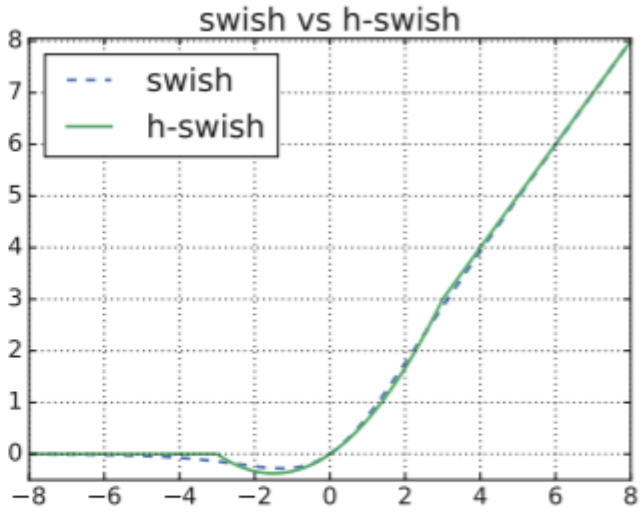


Figure 6. Sigmoid and swish nonlinearities and ther "hard" counterparts.

$$\text{swish}\, x = x \cdot \sigma(x) \qquad \text{h-swish}[x] = x \frac{\text{ReLU6}(x+3)}{6}$$

Network	Top-1	Madds	Params	
V3-Large 1.0	**75.2**	**219**	5.4M	
V3-Large 0.75	73.3	155	4M	
Mnasnet-A1	75.2	315	3.9	
Proxyless[5]	74.6	320	4M	
V2 1.0	72.0	300	3.4M	
V3-Small 1.0	67.4	66	2.9M	2
V3-Small 0.75	65.4	44	2.4M	
Mnasnet [43]	64.9	65.1	1.9M	2
V2 0.35	60.8	59.2	1.6M	

Table 3. Floating point performance on Pixel family of phones ("P-n" denotes a Pixel-n phone). All latencies are in ms. The inference latency is measured using a single large core with a batch size of 1.

Network	Top-1	P-1	P-2	P-3
V3-Large 1.0	**73.8**	57	55.9	38.9
V2 1.0	70.9	62	53.6	38.4
V3-Small	**64.9**	21	20.5	13.9
V2 0.35	57.2	20.6	18.6	13.1

Table 4. Quantized performance. All latencies are in ms. The inference latency is measured using single large core on the respective Pixel 1/2/3 device.

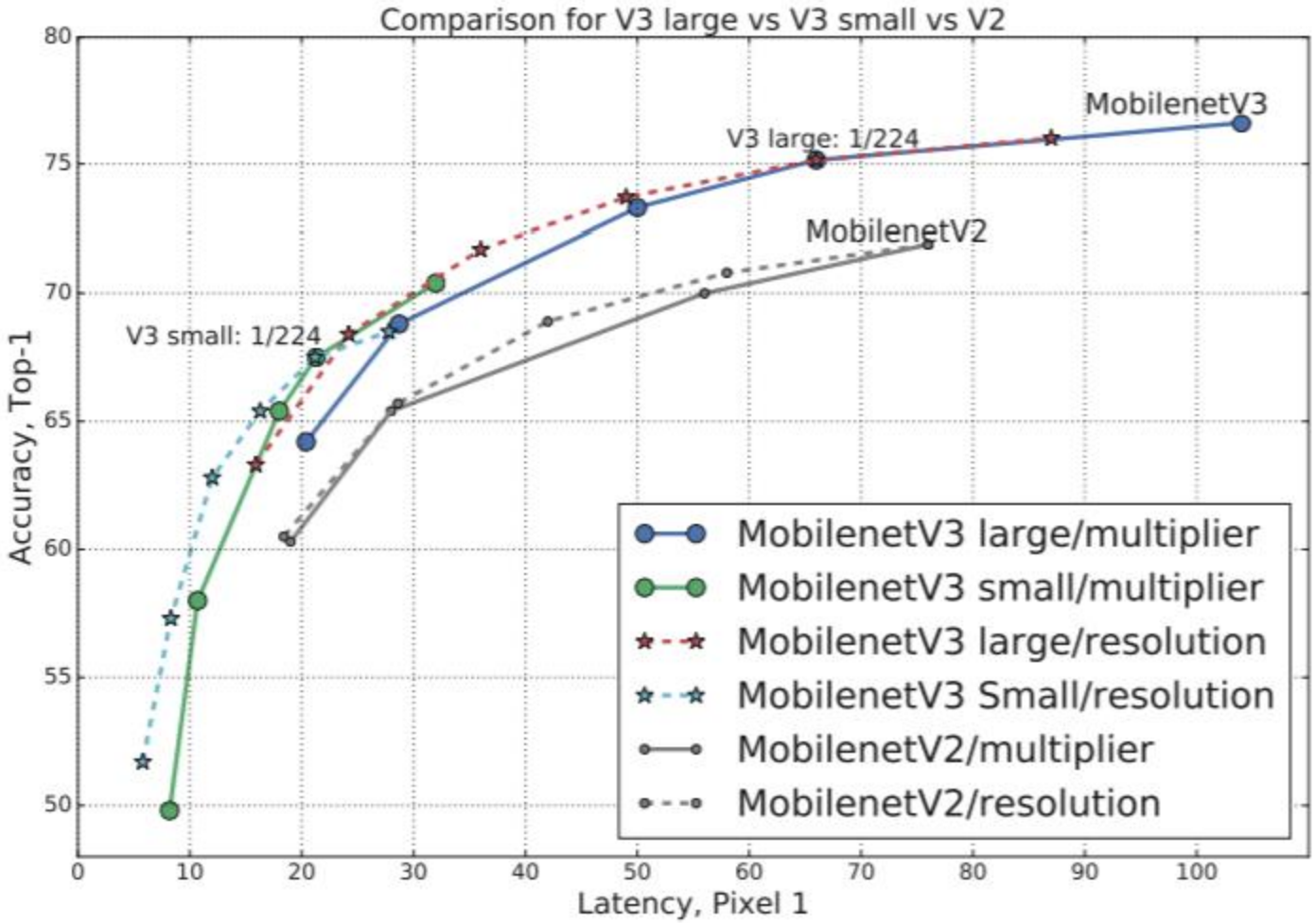


Figure 7. Performance of MobilenetV3 as a function of different multipliers and resolutions. In our experiments we have used multipliers 0.35, 0.5, 0.75, 1.0 and 1.25, with a fixed resolution of 224, and resolutions 96, 128, 160, 192, 224 and 256 with a fixed depth multiplier of 1.0. Best viewed in color.

N	Backbone	RF2	SH	F	mIOU	Params	Madds	CPU (f)	CPU (h)
1	V2	-	×	256	72.84	2.11M	21.29B	4.20s	1.07s
2	V2	✓	×	256	72.56	1.15M	13.68B	3.23s	819ms
3	V2	✓	✓	256	72.97	1.02M	12.83B	3.16s	808ms
4	V2	✓	✓	128	72.74	0.98M	12.57B	3.12s	797ms
5	V3	-	×	256	72.64	3.60M	18.43B	4.17s	1.07s
6	V3	✓	×	256	71.91	1.76M	11.24B	3.01s	765ms
7	V3	✓	✓	256	72.37	1.63M	10.33B	2.96s	750ms
8	V3	✓	✓	128	72.36	1.51M	9.74B	2.87s	730ms
9	V2 0.5	✓	✓	128	68.57	0.28M	4.00B	1.61s	402ms
10	V2 0.35	✓	✓	128	66.83	0.16M	2.54B	1.31s	323ms
11	V3-Small	✓	✓	128	68.38	0.47M	2.90B	1.38s	349ms

Table 7. Semantic segmentation results on Cityscapes *val* set. **RF2**: **R**educe the **F**ilters in the last block by a factor of **2**. V2 0.5 and V2 0.35 are MobileNetV2 with depth multiplier = 0.5 and 0.35, respectively. **SH: S**egmentation **H**ead, where × employs the R-ASPP while ✓ employs the proposed LR-ASPP. **F:** Number of **F**ilters used in the Segmentation Head. **CPU (f):** CPU time measured on a single large core of Pixel 3 (floating point) w.r.t. a **full-resolution** input (i.e., 1024 × 2048). **CPU (h):** CPU time measured w.r.t. a **half-resolution** input (i.e., 512 × 1024). Row 8, and 11 are our MobileNetV3 segmentation candidates.

Sparse Networks from Scratch: Faster Training without Losing Performance

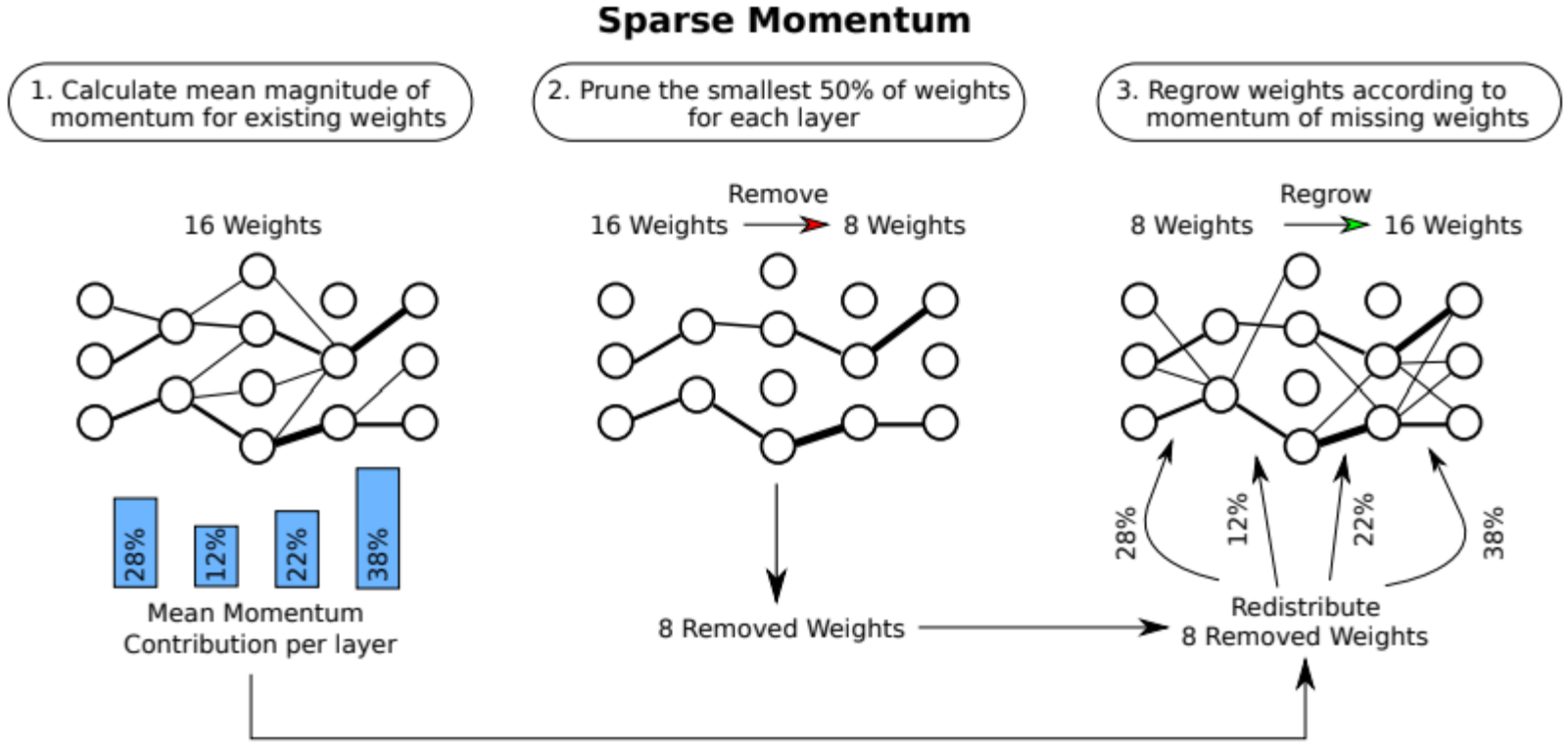


Figure 1: Sparse Momentum is applied at the end of each epoch: (1) take the magnitude of the exponentially smoothed gradient (momentum) of each layer and normalize to 1; (2) for each layer, remove $p = 50\%$ of the weights with the smallest magnitude; (3) across layers, redistribute the removed weights by adding weights to each layer proportionate to the momentum of each layer; within a layer, add weights starting from those with the largest momentum magnitude. Decay p.

Algorithm 1: Sparse momentum algorithm in NumPy notation.

Data: Layer i to k with: Momentum $\mathbf{M}_i$, Weight $\mathbf{W}_i$, binary $\mathbf{Mask}_i$; pruning rate p

TotalMomentum $\leftarrow 0$, TotalNonzero $\leftarrow 0$
/* (a) Calculate mean momentum contributions of all layers. */
for $i \leftarrow 0$ **to** k **do**
  MeanMomentum$_i \leftarrow$ mean(abs($\mathbf{M}_i[\mathbf{W}_i \neq 0]$))
  TotalMomentum $\leftarrow$ TotalMomentum + MeanMomentum$_i$
  NonZero$_i$ = sum($\mathbf{W}_i \neq 0$)
  TotalNonzero $\leftarrow$ TotalNonzero + NonZero$_i$
end
for $i \leftarrow 0$ **to** k **do**
  LayerContribution$_i \leftarrow$ MeanMomentum$_i$/TotalMomentum
  $p_i \leftarrow$ getPruneRate($\mathbf{W}_i, p$)
  NumRegrowth$_i \leftarrow$ floor($p_i \cdot$ TotalNonzero $\cdot$ LayerContribution$_i$)
end
/* (b) Prune weights by finding the NumRemove*th* smallest weight. */
for $i \leftarrow 0$ **to** k **do**
  NumRemove$_i \leftarrow$ NonZero$_i \cdot p$
  PruneThreshold $\leftarrow$ sort(abs($\mathbf{W}_i[\mathbf{W}_i \neq 0]$))[NumRemove$_i$]
  $\mathbf{Mask}_i[\mathbf{W}_i <$ PruneThreshold$] \leftarrow 0$ // Stop gradient flow.
  $\mathbf{W}_i[\mathbf{W}_i <$ PruneThreshold$] \leftarrow 0$
end
/* (c) Enable gradient flow of weights with largest momentum magnitude. */
for $i \leftarrow 0$ **to** k **do**
  RegrowthThreshold$_i \leftarrow$ sort(abs($\mathbf{M}_i[\mathbf{W}_i == 0]$))[NumRegrowth$_i$]
  $\mathbf{Z}_i = \mathbf{M}_i \cdot (\mathbf{W}_i == 0)$ // Only consider the momentum of missing weights.
  $\mathbf{Mask}_i \leftarrow \mathbf{Mask}_i \mid (\mathbf{Z}_i >$ RegrowthThreshold$_i)$ // | is the boolean OR operator
end
$p \leftarrow$ decayPruneRate(p)
applyMask()

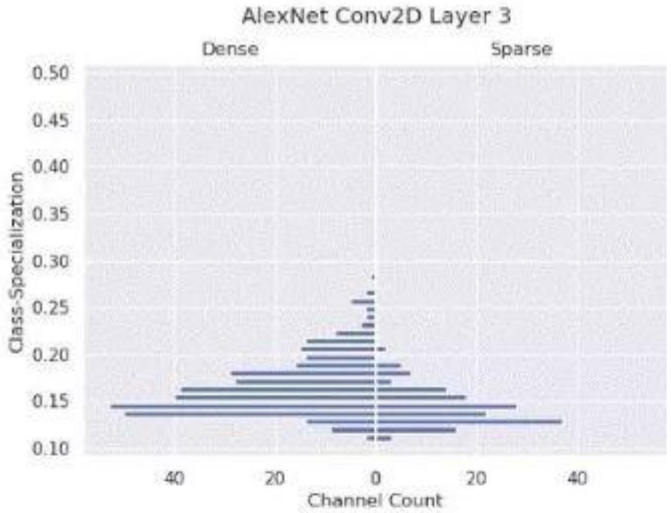
Table 2: CIFAR-10 test set error (±standard error) for dense baselines, Sparse Momentum and SNIP.

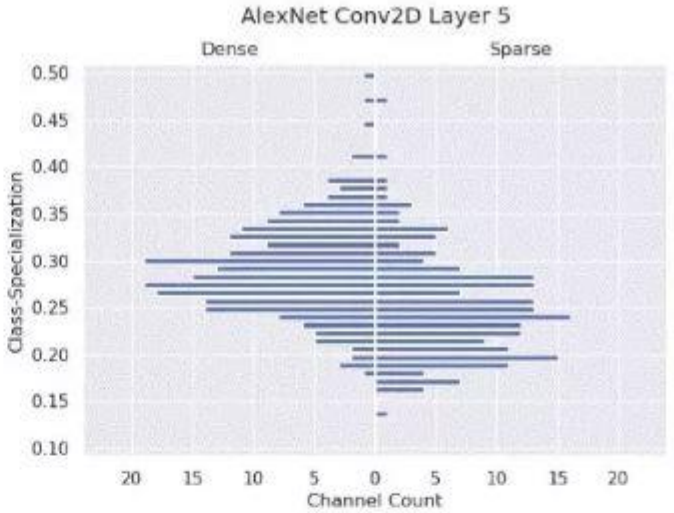
		Sparse Error (%)		
Model	Dense Error (%)	SNIP	Momentum	Weights (%)
AlexNet-s	12.95±0.056	14.99	**14.35**±0.057	10
AlexNet-b	12.85±0.068	14.50	**13.93**±0.048	10
VGG16-C	6.49±0.038	7.27	**6.77**±0.056	5
VGG16-D	6.59±0.050	7.09	**6.49**±0.045*	5
VGG16-like	6.50±0.054	8.00	**6.71**±0.046	3
WRN-16-8	4.57±0.022	6.63	**5.66**±0.054	5
WRN-16-10	4.45±0.040	6.43	**4.59**±0.043*	5
WRN-22-8	4.26±0.032	5.85	**4.96**±0.042	5

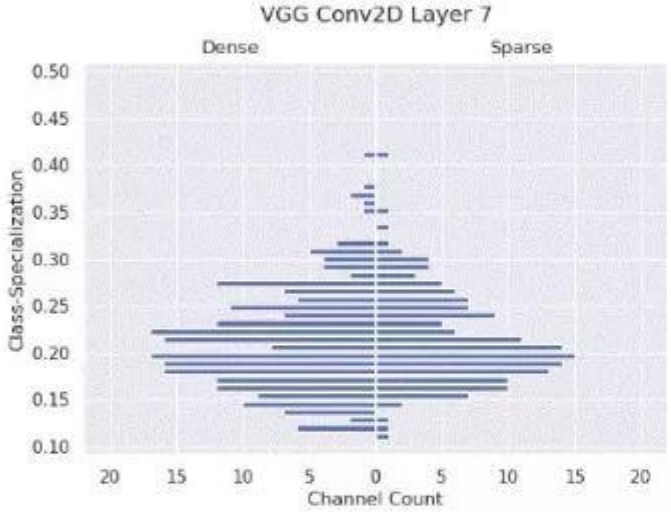
* 95% confidence intervals overlap with dense model.

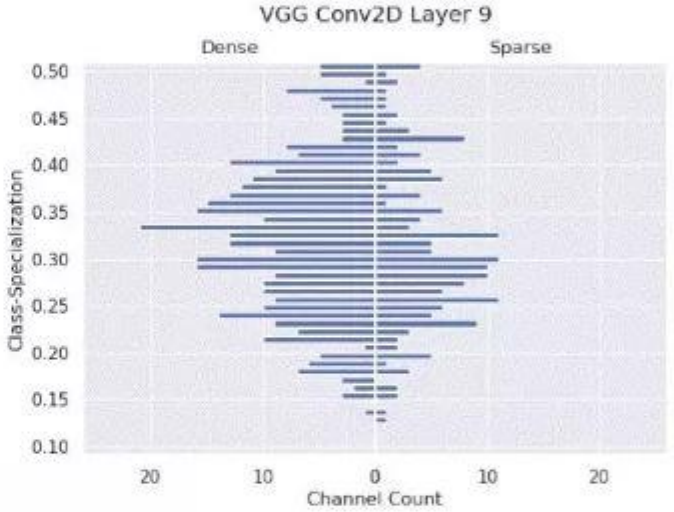
Table 3: ImageNet results for sparse momentum. Other results are from Mostafa and Wang (2019).

	Accuracy (%)			
Model	Top-1	Top-5	Top-1	Top-5
Dense baseline (He et al., 2016)	79.3	94.8	79.3	94.8
	10% weights		20% Weights	
Static sparse (Mostafa and Wang, 2019)	67.8	88.4	71.6	90.4
Thin Dense (Mostafa and Wang, 2019)	70.7	89.9	72.4	90.9
DeepR (Bellec et al., 2018)	70.2	90.0	71.7	90.6
Compressed sparse (Mostafa and Wang, 2019)	70.3	90.0	73.2	91.5
Sparse Evolutionary Training (Mocanu et al., 2018)	70.4	90.1	72.6	91.2
Dynamic Sparse (Mostafa and Wang, 2019)	71.6	90.5	73.3	92.4
Sparse momentum	**73.1**	**91.5**	**74.9**	**92.5**

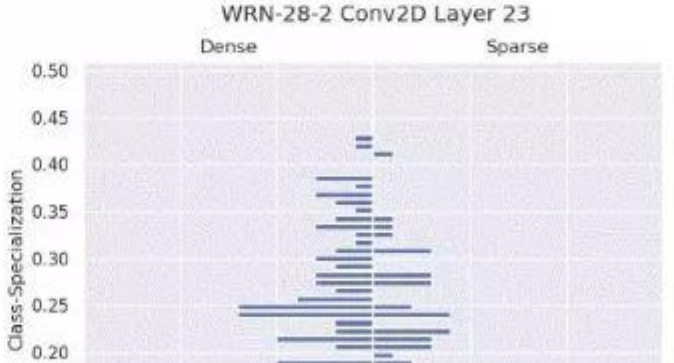

AlexNet Conv2D Layer 3
Dense
Sparse
Class-Specialization
Channel Count


AlexNet Conv2D Layer 5
Dense
Sparse
Class-Specialization
Channel Count


VGG Conv2D Layer 7
Dense
Sparse
Class-Specialization
Channel Count


VGG Conv2D Layer 9
Dense
Sparse
Class-Specialization
Channel Count

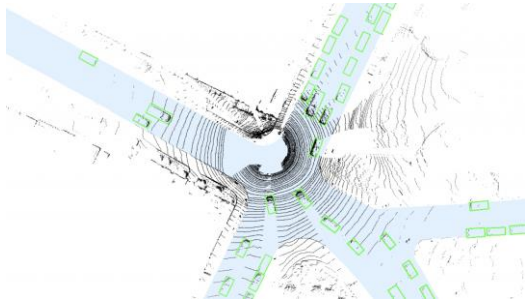

WRN-28-2 Conv2D Layer 5
Dense
Sparse
Class-Specialization


WRN-28-2 Conv2D Layer 23
Dense
Sparse
Class-Specialization

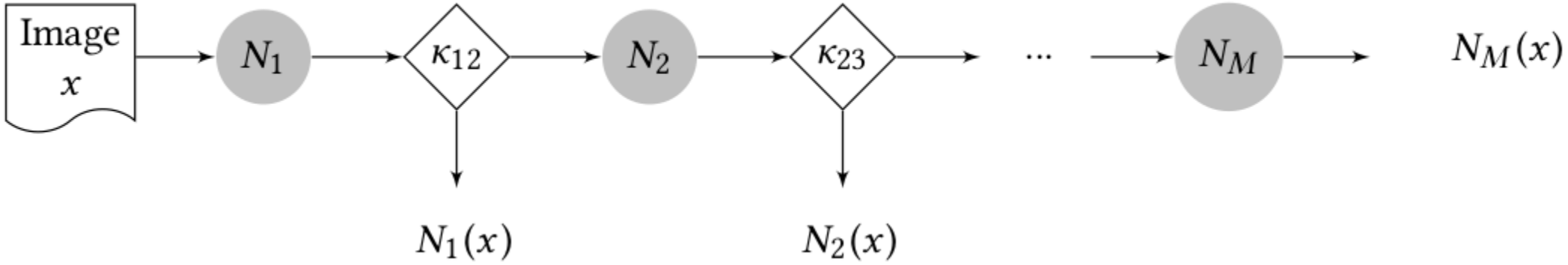
DYNAMIC EXECUTION BASED ON INPUT

Concept

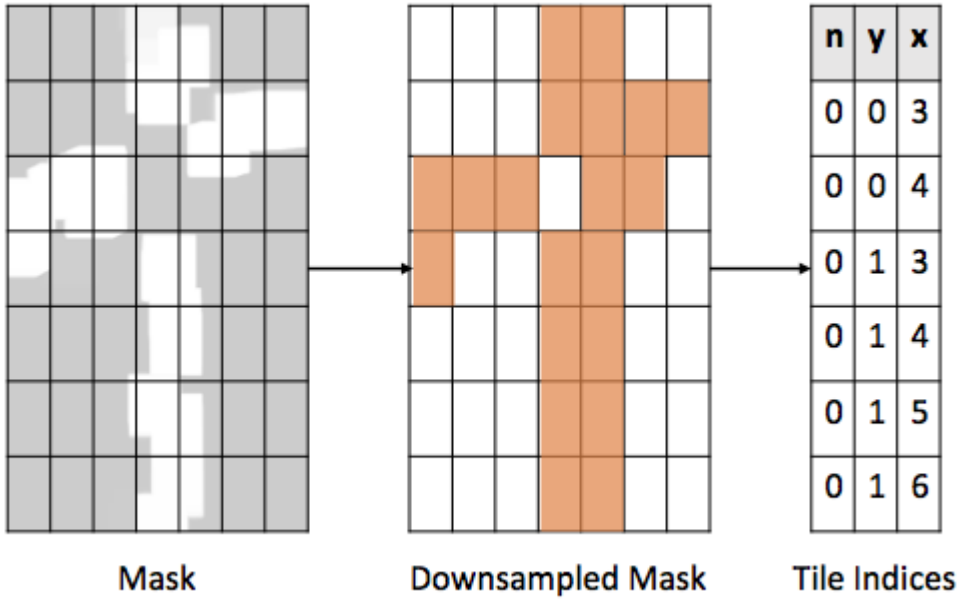
- Skip invalid input and its computations
  - LiDAR for traffic detection
  - Skip objects outside the road

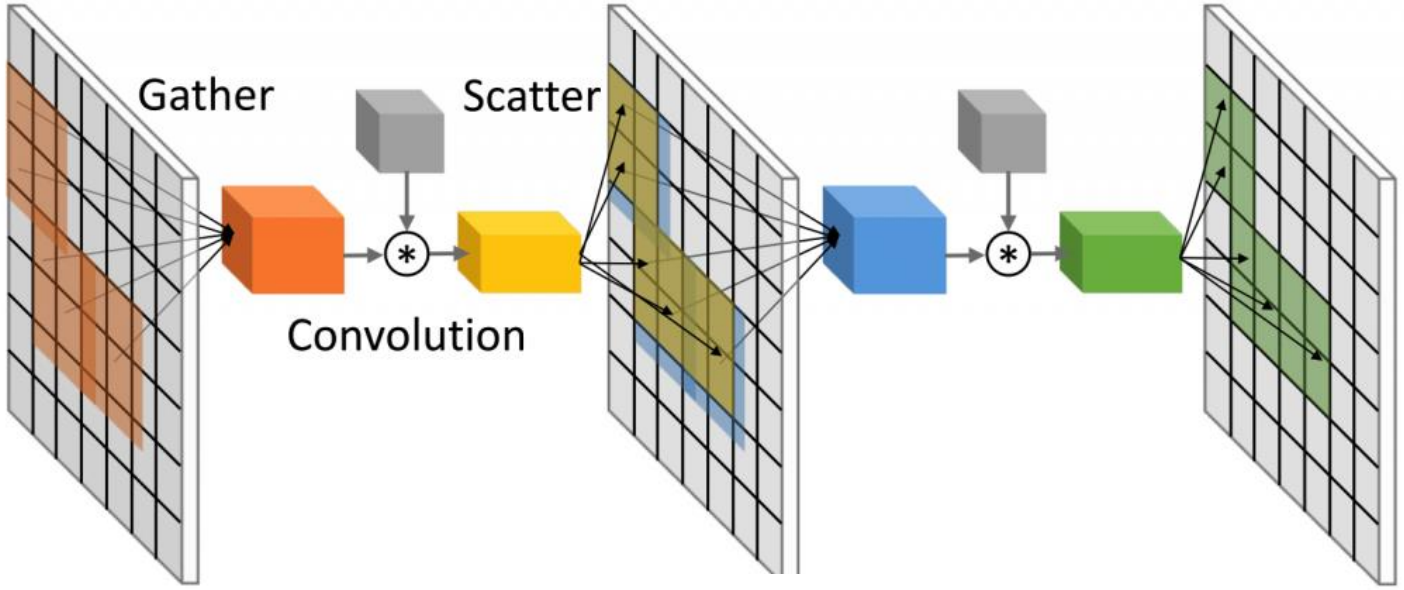


- Not all input are created equal
  - Easy input just needs simpler architecture



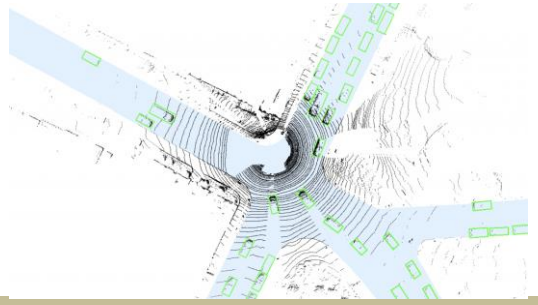
SBNet

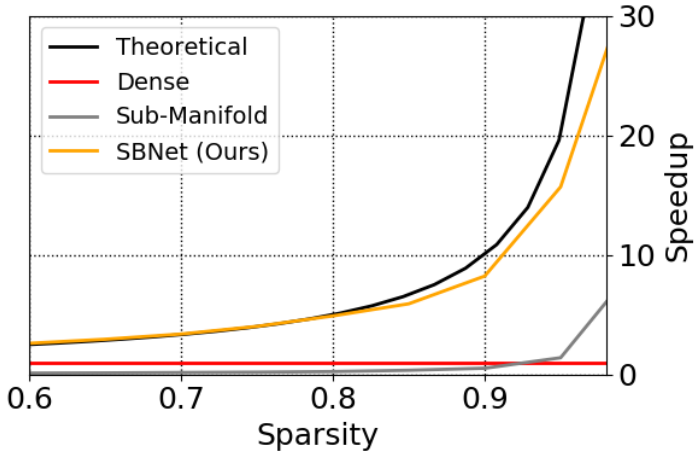




To exploit sparsity in the activations of CNNs, SBNet first converts a computation mask to a tile index list.

Gather along batch dim





SGAD: Soft-Guided Adaptively-Dropped Neural Network

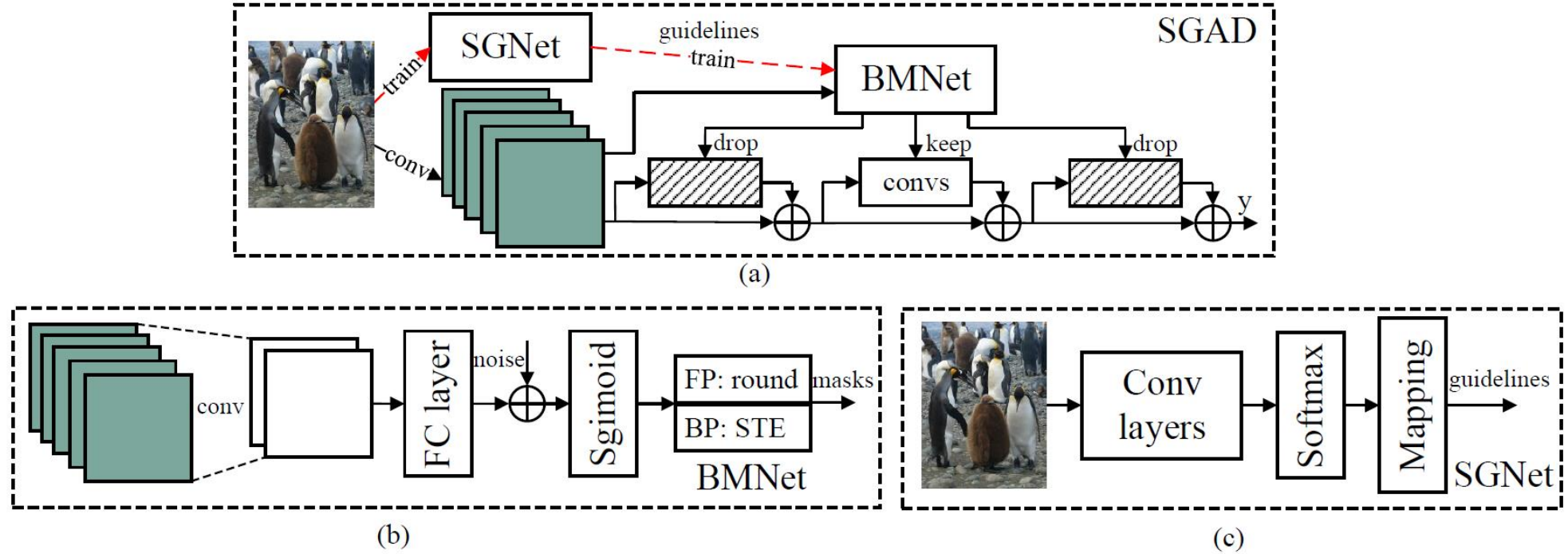


Figure 2: Architecture of the proposed SGAD. (a). The overview of the presented algorithm. The activations after the first single convolutional layer of ResNet is denoted by green blocks. The red dot lines mean that the SGNet will be active only in the training phase. (b) Architecture of BMNet. The binary rounding and sinc function-based estimator are used in the phase of forward and backward propagation, respectively. (c). Architecture of SGNet. The block named *Conv layers* denotes a group of convolutional layers which can be flexibly adjusted.

- SGNet
  - Indicate the difficulty of classification (~ softmax output)

$$var^n = \frac{1}{M}\sum_{i=1}^{M}[(\frac{e^{s_i^n}}{\sum_j e^{s_j^n}} - \frac{1}{M})^2] = \frac{\sum_i e^{2s_i^n}}{M(\sum_j e^{s_j^n})^2} - \frac{1}{M^2} \in [0, \frac{1}{M})$$

- BMNet: binary mask to indicate to skip block
  - Regularizer loss

$$R^m = \frac{1}{N}\sum_{n=1}^{N} \parallel rat_s^n - (1 - \frac{1}{L}\sum_{j=1}^{L} m_j^n(z_n^1; \theta_m)) \parallel_1$$

Expected drop ratio for a input

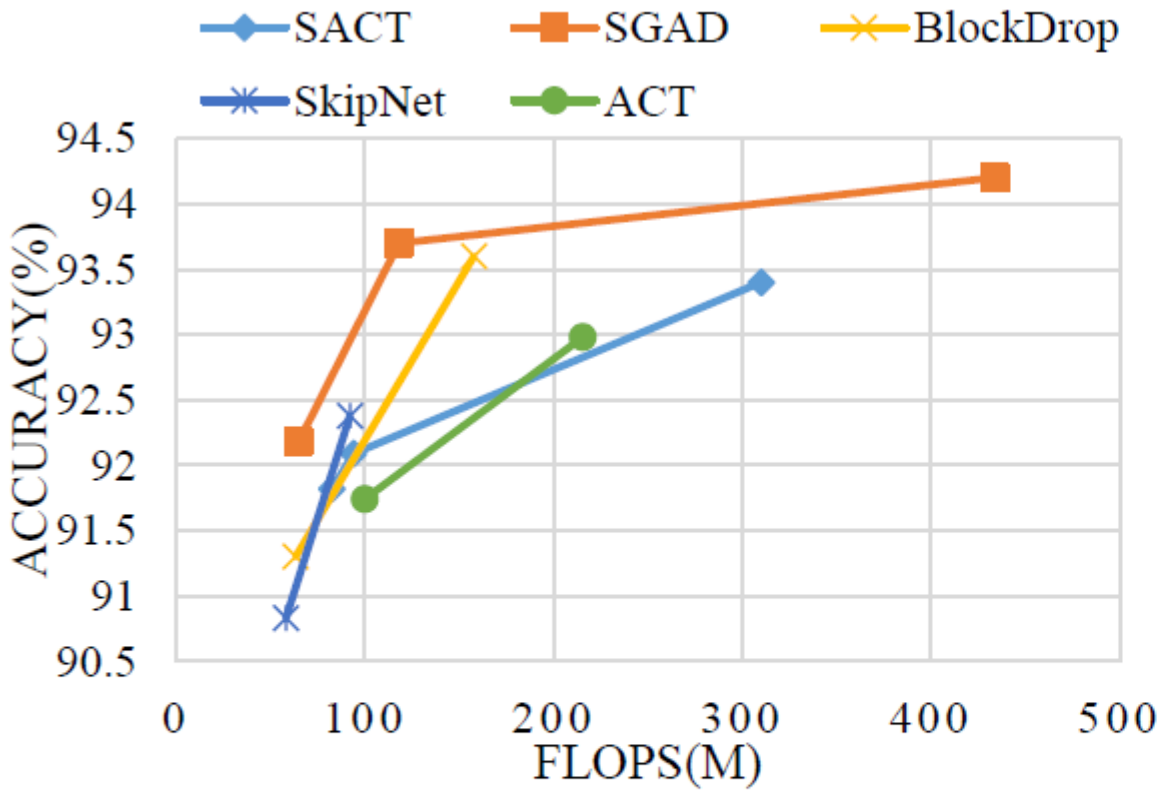
Average drop ratio in a batch

$$rat_s^n = 1 - L^{1-scale \times var^n}/L,\ n \in \{1, 2, \cdots, N\}, \quad scale = -\frac{Mln(s_{max})}{ln(L)}$$

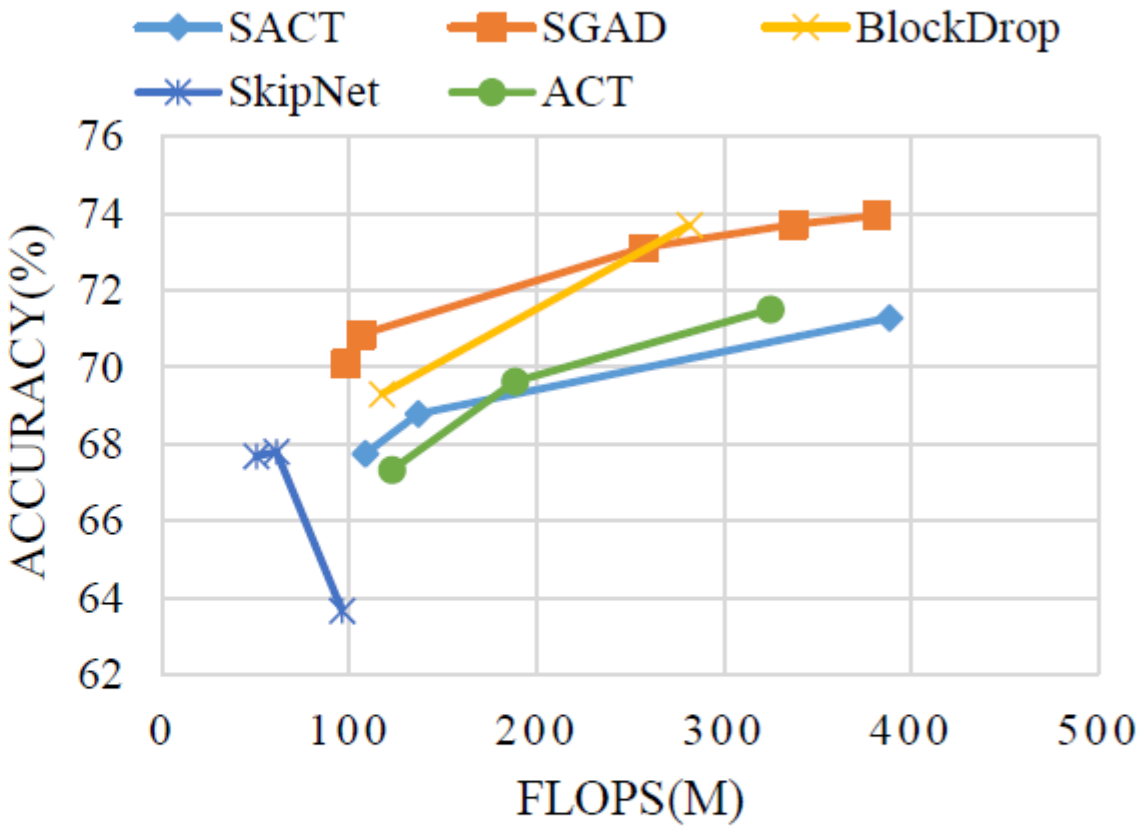
- Smax: allowed maximum drop ratio
- SGNet can be removed and only the BMNet and the ResNet are needed after training.

Table 1: Results of SGADs and ResNets. For the original ResNets, n-FLOPs=1.0.

Dataset	Layers	ResNet	MF-SGAD, $s_{max} = 0.2$		LF-SGAD, $s_{max} = 0.8$	
		accuracy	accuracy	n-FLOPs	accuracy	n-FLOPs
CIFAR-10	32	93.02%	93.11%	0.86	92.18%	0.47
	110	94.57%	94.20%	0.86	**93.70%**	**0.23**
CIFAR-100	32	70.38%	70.85%	0.77	70.09%	0.71
	110	73.94%	73.94%	0.94	73.94%	0.75



(a) CIFAR-10



(b) CIFAR-100

Figure 3: Comparisons with state-of-the-art. (a) Results on CIFAR-10. (b) Results on CIFAR-100. The solid lines and dashed lines have different baselines.

Dropping ratio relate to Magnitude of Grad.

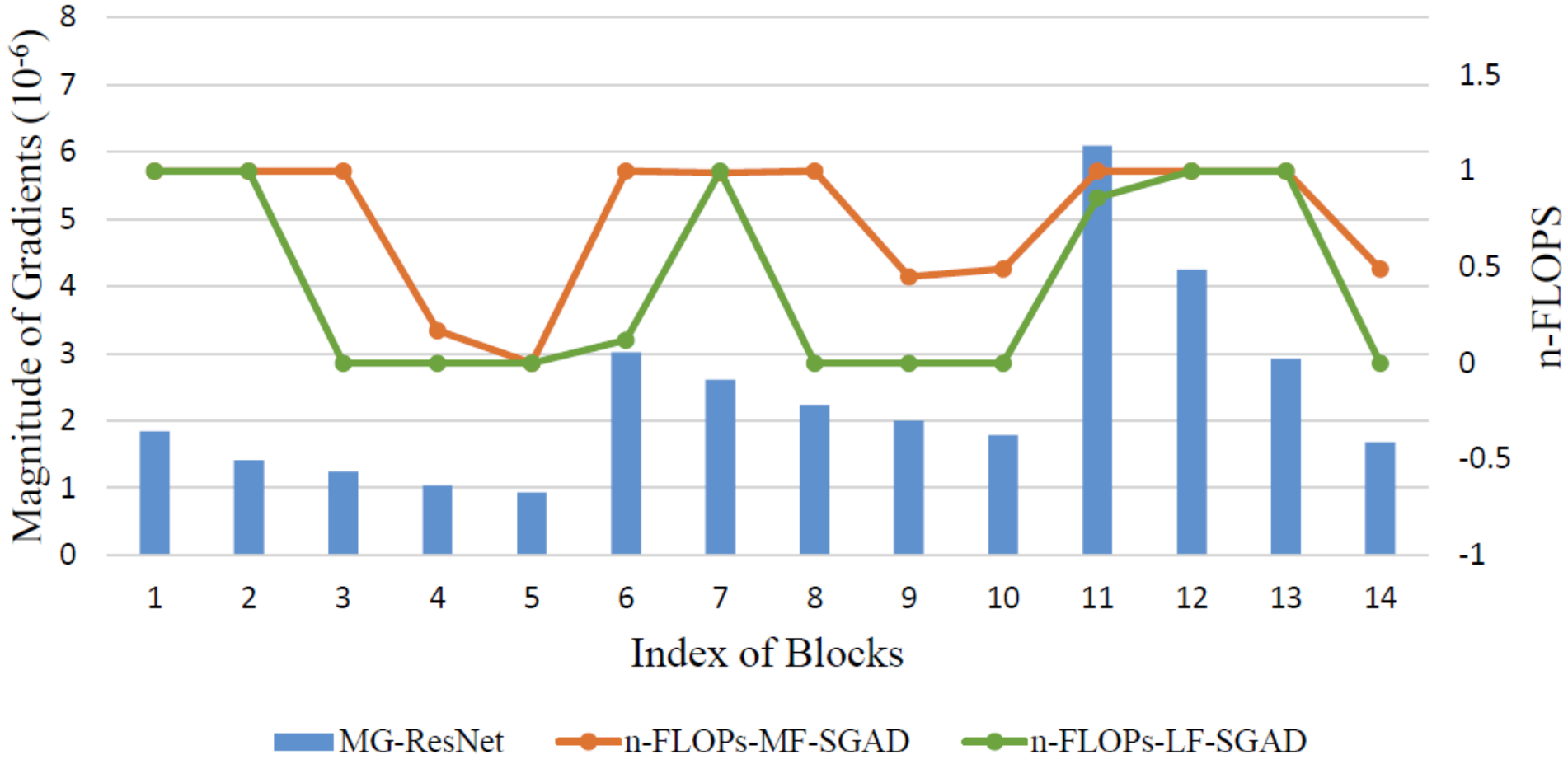


Figure 4: Comparisons of magnitude of gradients and normalized flops of each blocks. The n-FLOPS-MF-SGAD and the n-FLOPS-LF-SGAD denote the n-FLOPS of MF-SGADs and LF-SGAD, respectively. The magnitudes of gradients are given by the mean value of L1 normalization of gradients after the 160-th epoch.

Dynamic Channel Pruning

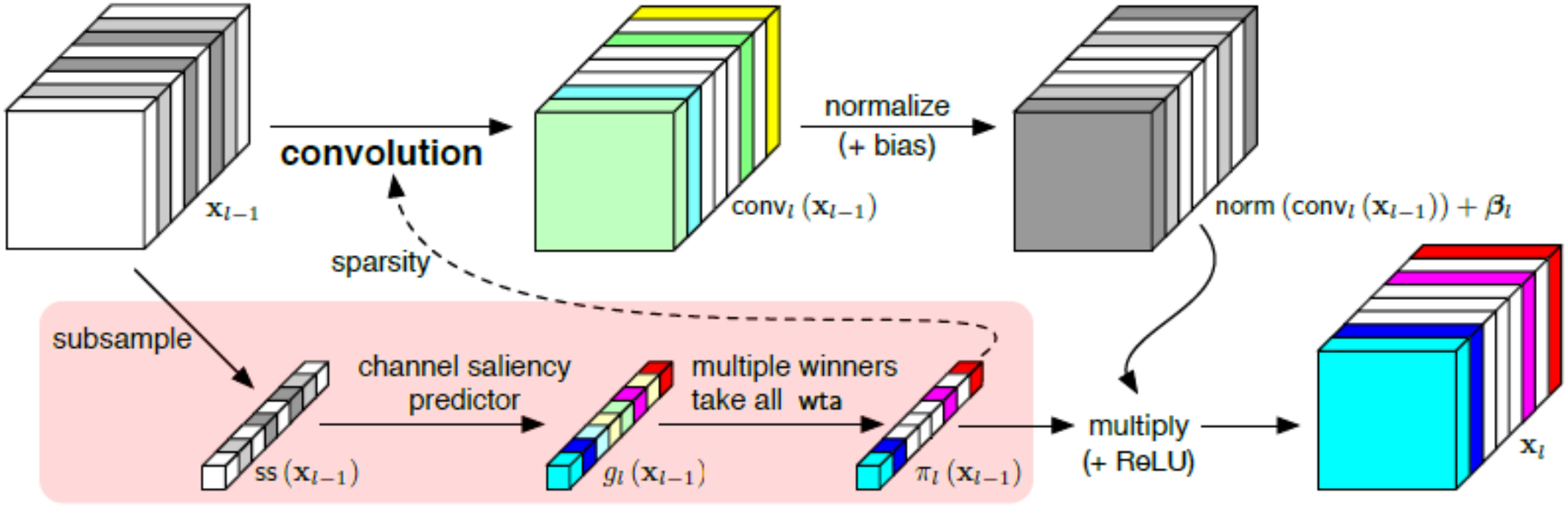
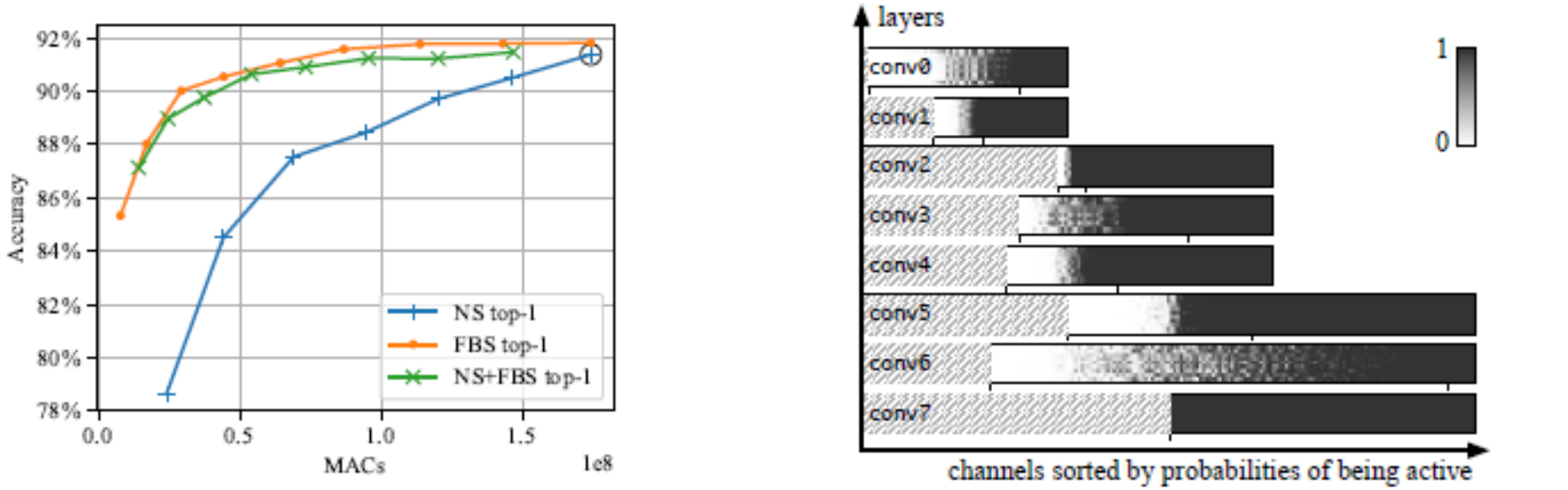


Figure 2: A high level view of a convolutional layer with FBS. By way of illustration, we use the l^{th} layer with 8-channel input and output features, where channels are colored to indicate different saliencies, and the white blocks (▯) represent all-zero channels.

4.1 CIFAR-10



(a) M-CifarNet accuracy/MACs trade-off

(b) Channel skipping probabilites

Figure 3: Experimental results on M-CifarNet. We compare in (a) the accuracy/MACs trade-off between FBS, NS and FBS+NS. The baseline is emphasized by the circle ◯. The heat map in (b) reveals the individual probability of skipping a channel for each channel (x-axis), when an image of a category (y-axis) is shown to the network with $d = 1$.

Method	Dynamic	Baseline Top-1	Baseline Top-5	Accelerated Top-1	Accelerated Top-5	MAC saving
Soft Filter Pruning (He et al., 2018a)		29.72	10.37	32.90	12.22	1.72×
Network Slimming (Liu et al. (2017), our implementation)		31.02	11.32	32.79	12.61	1.39×
Discrimination-aware Channel Pruning (Zhuang et al., 2018)		30.36	11.02	32.65	12.40	1.89×
Low-cost Collaborative Layers (Dong et al., 2017)	✓	30.02	10.76	33.67	13.06	1.53×
Channel Gating Neural Networks (Hua et al., 2018)	✓	30.98	11.16	32.60	12.19	1.61×
Feature Boosting and Suppression (FBS)	✓	**29.29**	**10.32**	**31.83**	**11.78**	**1.98×**

Table 1: Comparisons of error rates of the baseline and accelerated ResNet-18 models.

Method	Dynamic	Δ top-5 errors (%) 3×	4×	5×
Filter Pruning (Li et al. (2017), reproduced by He et al. (2017))		—	8.6	14.6
Perforated CNNs (Figurnov et al., 2016)		3.7	5.5	—
Network Slimming (Liu et al. (2017), our implementation)		1.37	3.26	5.18
Runtime Neural Pruning (Lin et al., 2017)	✓	2.32	3.23	3.58
Channel Pruning (He et al., 2017)		0.0	1.0	1.7
AutoML for Model Compression (He et al., 2018b)		—	—	1.4
ThiNet-Conv (Luo et al., 2017)		0.37	—	—
Feature Boosting and Suppression (FBS)	✓	0.04	**0.52**	**0.59**

Table 2: Comparisons of top-5 error rate increases for VGG-16 on ILSVRC2012 validation set under 3×, 4× and 5× speed-up constraints. The baseline has a 10.1% top-5 error rate. Results from He et al. (2017) only show numbers with one digit after the decimal point.

Summary

- Fancy pruning methods do not contribute a lot
- Learning based pruning is the trend
  - Add constraints as a regularization term
- Fine tuning or not
  - Training from scratch with longer training time achieve similar or better results
- Learned dynamic execution based on input