Model Compression

Shuyue Jia

Goal

- Real time execution (Time efficient)
- Lower Power consumption
- Lower memory consumption
- Smaller size (Less-parametrized in size)
- Performance

部署: 云服务器端？移动端？

训练模型 VS 云服务器端推理 VS 移动端推理：

训练-[模型格式转换-优化-验证-部署]

- **模型实时性:**
  Execution Time (Speed / latency)
  Memory Access Cost (MAC) / Memory Bandwidth
  Parallelism Degree

- **模型大小与能耗:**
  Num of Parameters
  MACs / FLOPs
  Energy Cost

- **性能:**
  Test Error / Accuracy

移动端 推理框架

- Online VS Offline

  Online: 数据上传到Server，然后返回结果 -> 服务器端

  Offline: 直接手机移动端跑 -> 移动端

- CPU: (ARMv7, ARMv8, X86, MIPS, RISC-V) CPU端优化(是否汇编优化)？是否使用Arm NEON，OpenCL以及Hexagon HVX (DSP)等优化？是否设计thread pool？

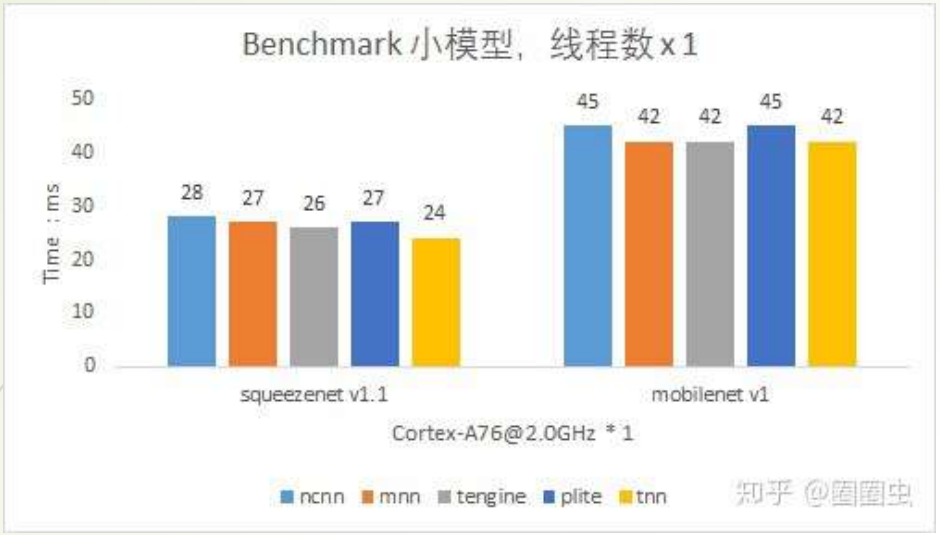
- GPU: (Mali, Adreno, Metal, NV, AMD, PowerVR) 高效的kernel？是否使用OpenGL, Metal等？

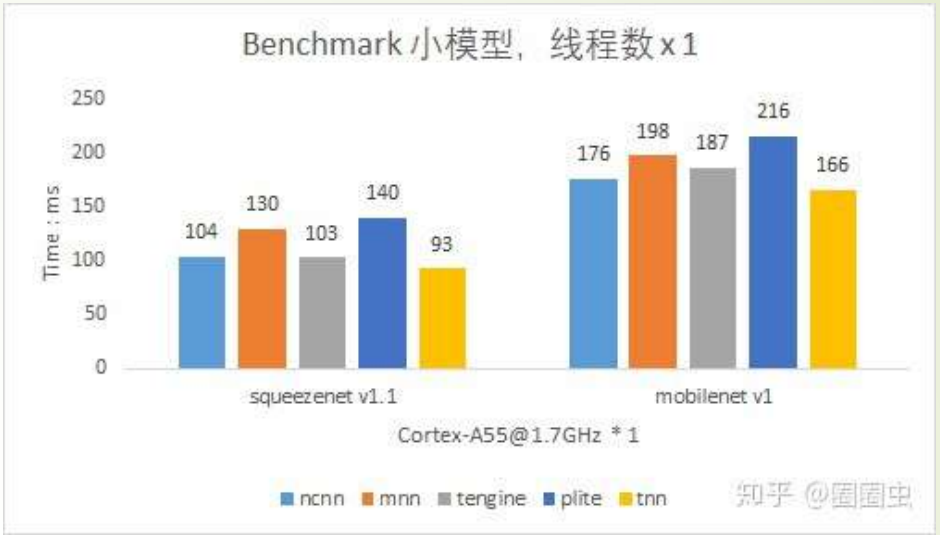
- 模型层面 -> Researchers

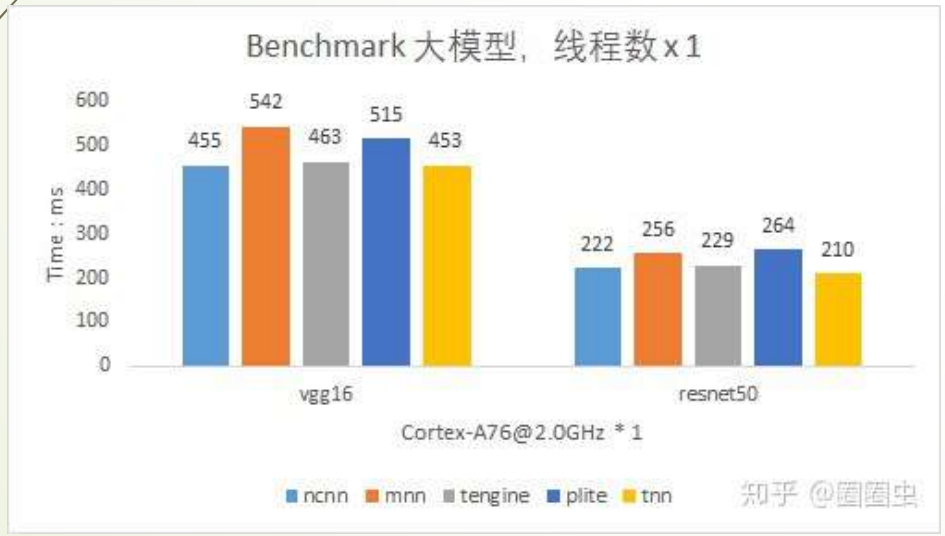
移动端 推理框架

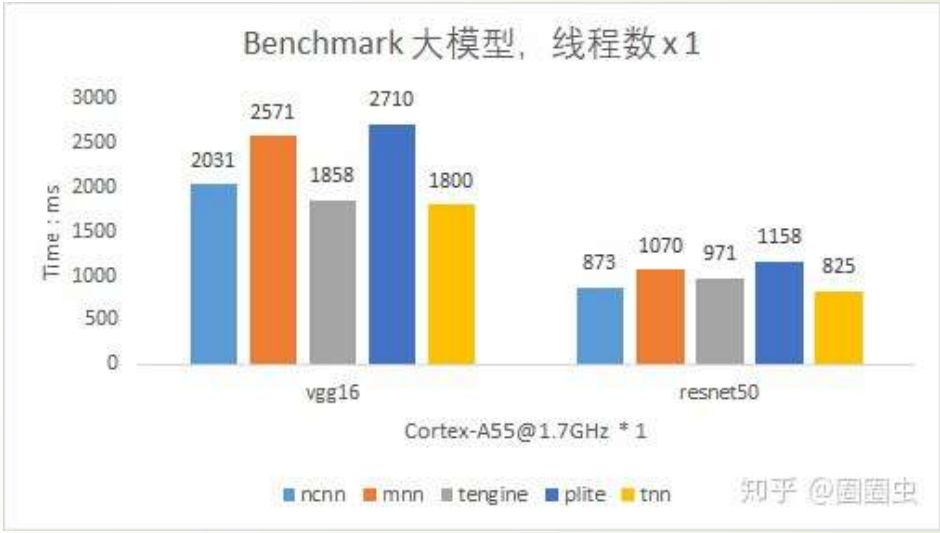
训练-[模型格式转换-优化-验证-部署]

- [2017] **NCNN**/FeatherCNN: float32, CPU优化, GPU速度无优化, C++接口[link]
- [2017] TensorFlow Lite: Android, IOS, Linux SoCs IoT设备, Python接口
- [2017] Paddle Lite: CPU+GPU均优化, Python接口 [link]
- [2017] Tengine-Lite: 支持CPU与GPU，C++接口
- [2018] **MACE**: CPU+GPU均优化, Python接口
- [2018] Facebook QNNPACK, [2017] 九言科技 Prestissimo
- [2018] **PocketFlow**: 自动模型压缩
- [2019] **MNN**: CPU+GPU均优化, Python接口
- [2020] **TNN**: CPU+GPU均优化, 低精度优化, 内存优化，C++接口 [link]
- **CoreML**: 局限于苹果设备

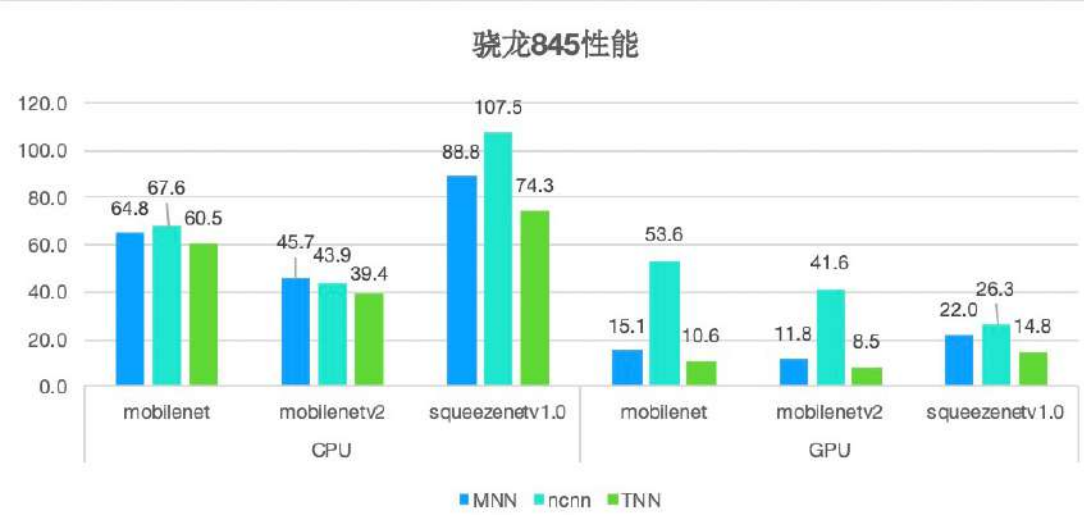


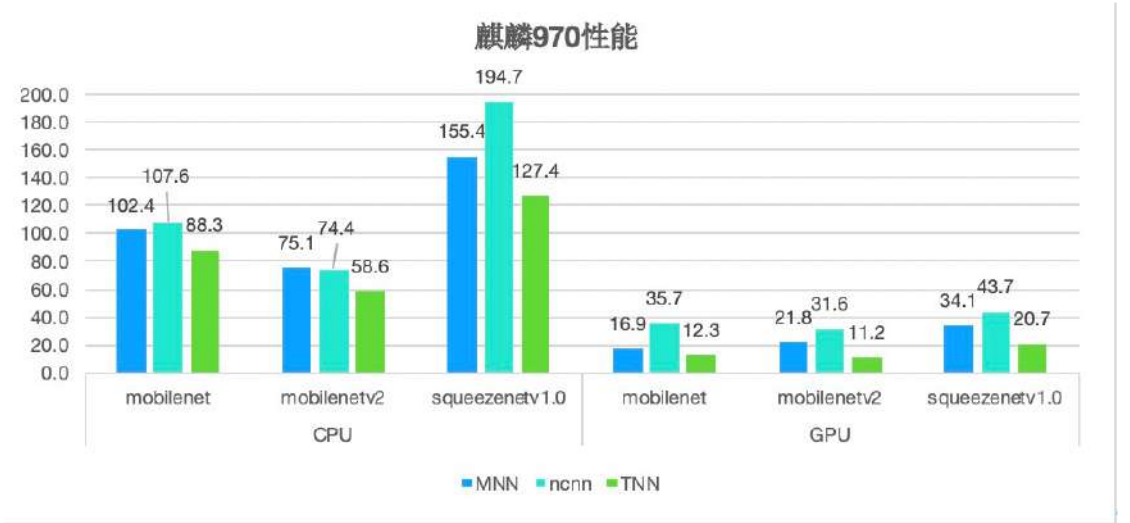


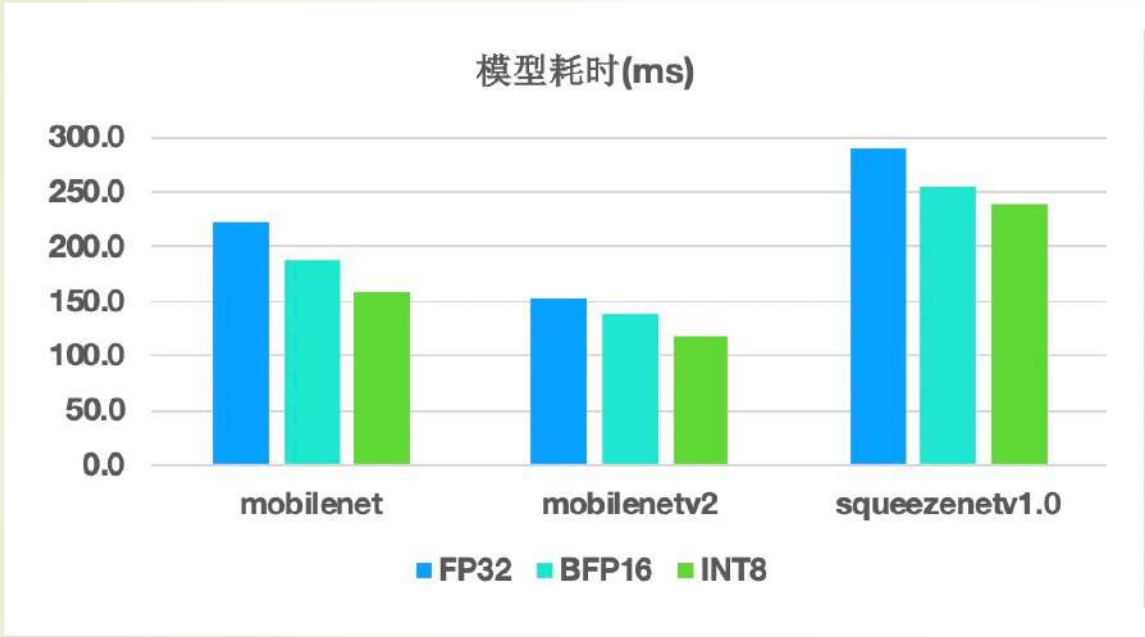


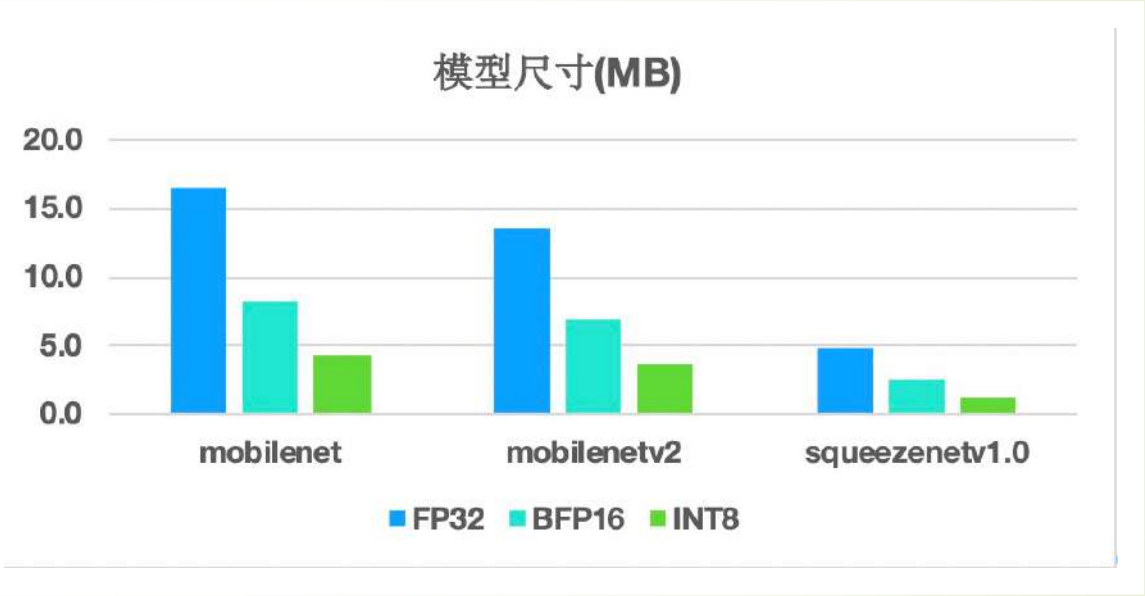


腾讯ncnn, 腾讯TNN, 阿里mnn, 开放智能Tengine, 百度Paddle-Lite, 小米MACE


骁龙845性能
120.0
100.0
80.0
60.0
40.0
20.0
0.0
64.8
67.6
60.5
45.7
43.9
39.4
88.8
107.5
74.3
15.1
53.6
10.6
11.8
41.6
8.5
22.0
26.3
14.8
mobilenet
mobilenetv2
squeezenetv1.0
mobilenet
mobilenetv2
squeezenetv1.0
CPU
GPU
MNN
ncnn
TNN


麒麟970性能
200.0
180.0
160.0
140.0
120.0
100.0
80.0
60.0
40.0
20.0
0.0
102.4
107.6
88.3
75.1
74.4
58.6
155.4
194.7
127.4
16.9
35.7
12.3
21.8
31.6
11.2
34.1
43.7
20.7
mobilenet
mobilenetv2
squeezenetv1.0
mobilenet
mobilenetv2
squeezenetv1.0
CPU
GPU
MNN
ncnn
TNN


模型耗时(ms)
300.0
250.0
200.0
150.0
100.0
50.0
0.0
mobilenet
mobilenetv2
squeezenetv1.0
FP32
BFP16
INT8


模型尺寸(MB)
20.0
15.0
10.0
5.0
0.0
mobilenet
mobilenetv2
squeezenetv1.0
FP32
BFP16
INT8

CPU: 2.4GHz i5 GPU: GTX1060 6G RTX 3090	Method	Time (CPU/GPU) (s)	Num of Params (M)	MACs (G)
FR	C3DVQA	318.35 / 3.30 (GPU: 3090)	0.2272	1423.3642
NR	(传统) V-BLIINDS	CPU: 88.24 (Feature)	-	-
	(传统) VIIDEO	CPU: 79.15	-	-
	VSFA	423.68 / 18.89 / 5.17	0.5565	0.1296
	(传统) TLVQM	CPU: 52.98	-	-
	CNN-TLVQM	51.06 / 39.31 (Feature)	23.570693	Ref: ResNet50: 4.12
	(传统) RAPIQUE	22.61 (Feature)	-	-

video_height: 540, video_width: 960, video_channel: 3, video_length: 240, 8 Sconds, 30 FPS

Model Complexity

- Num of Parameters (Params):
  - Conv: $(K_{in} \times K_{out} \times C_{in} + 1) \times C_{out}$
  - FC: $N_{in} \times N_{out} + N_{out}$
- Floating Point Operations (FLOPs) (注意区别于FLOPS):
  - Conv: $2K_{in} \times K_{out} \times C_{in} \times C_{out} \times H_{out} \times W_{out}$
  - FC: $(2N_{in} - 1)N_{out}$
- Multiply-accumulate Operations (MACC/MACs): ≈ ½ FLOPs

Reference: https://zhuanlan.zhihu.com/p/266275736

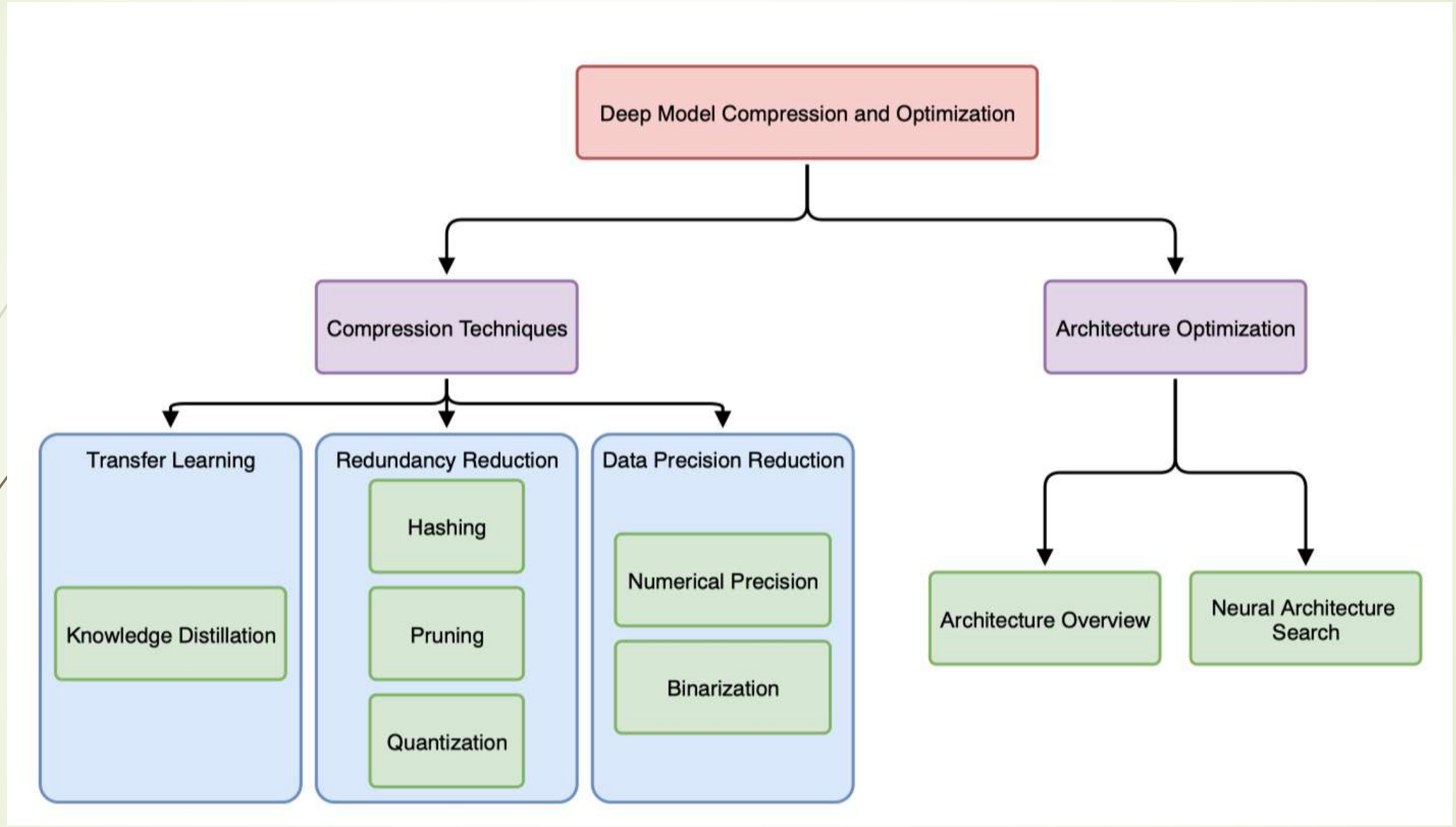
Current Main-stream Methods

Static Model Acceleration

- Smaller: **Pruning**: pixel (weight), vector, channel, filter
- Lower: **quantization**: scalar, vector / **Binarization**
- Less computation - Low rank approximation, Winograd transformation
- Faster: **Architecture Design** - Squeezenet, Mobilenet, Shufflenet

Dynamic execution

- SBNet
- SGAD
- Dynamic channel pruning


Deep Model Compression and Optimization
Compression Techniques
Architecture Optimization
Transfer Learning
Knowledge Distillation
Redundancy Reduction
Hashing
Pruning
Quantization
Data Precision Reduction
Numerical Precision
Binarization
Architecture Overview
Neural Architecture Search

Pruning

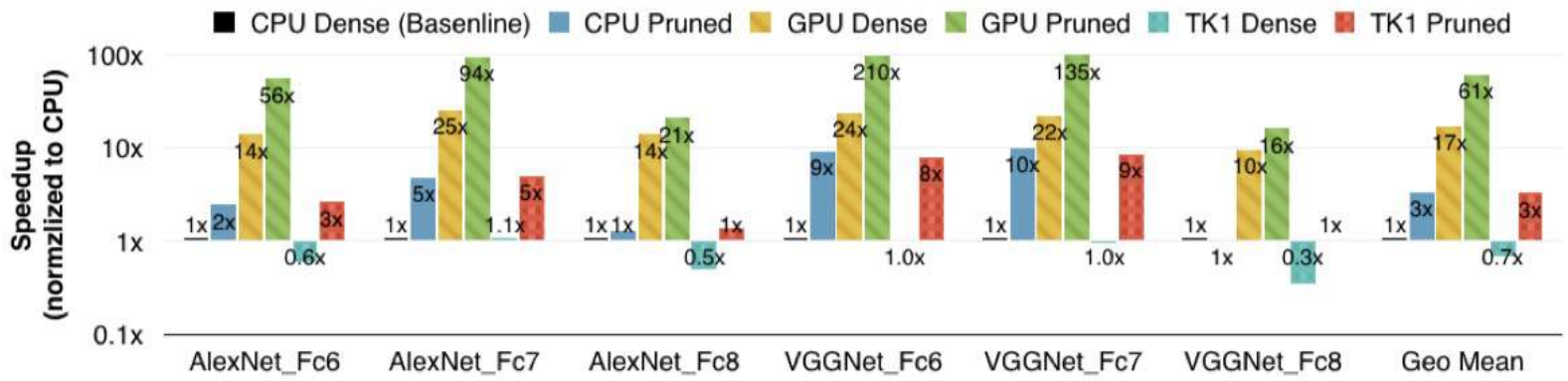


Figure 9: Compared with the original network, pruned network layer achieved 3× speedup on CPU, 3.5× on GPU and 4.2× on mobile GPU on average. Batch size = 1 targeting real time processing. Performance number normalized to CPU.

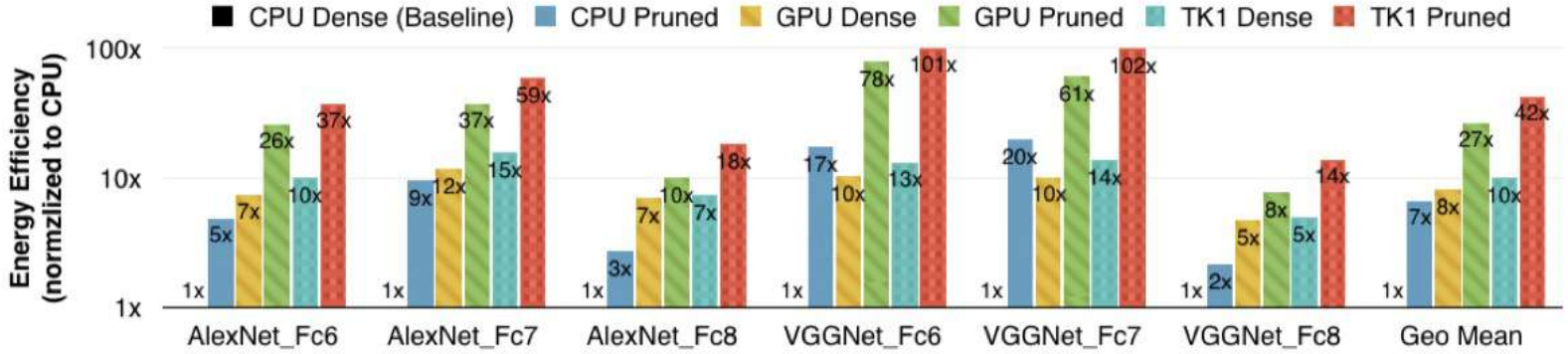


Figure 10: Compared with the original network, pruned network layer takes 7× less energy on CPU, 3.3× less on GPU and 4.2× less on mobile GPU on average. Batch size = 1 targeting real time processing. Energy number normalized to CPU.

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

Quantization

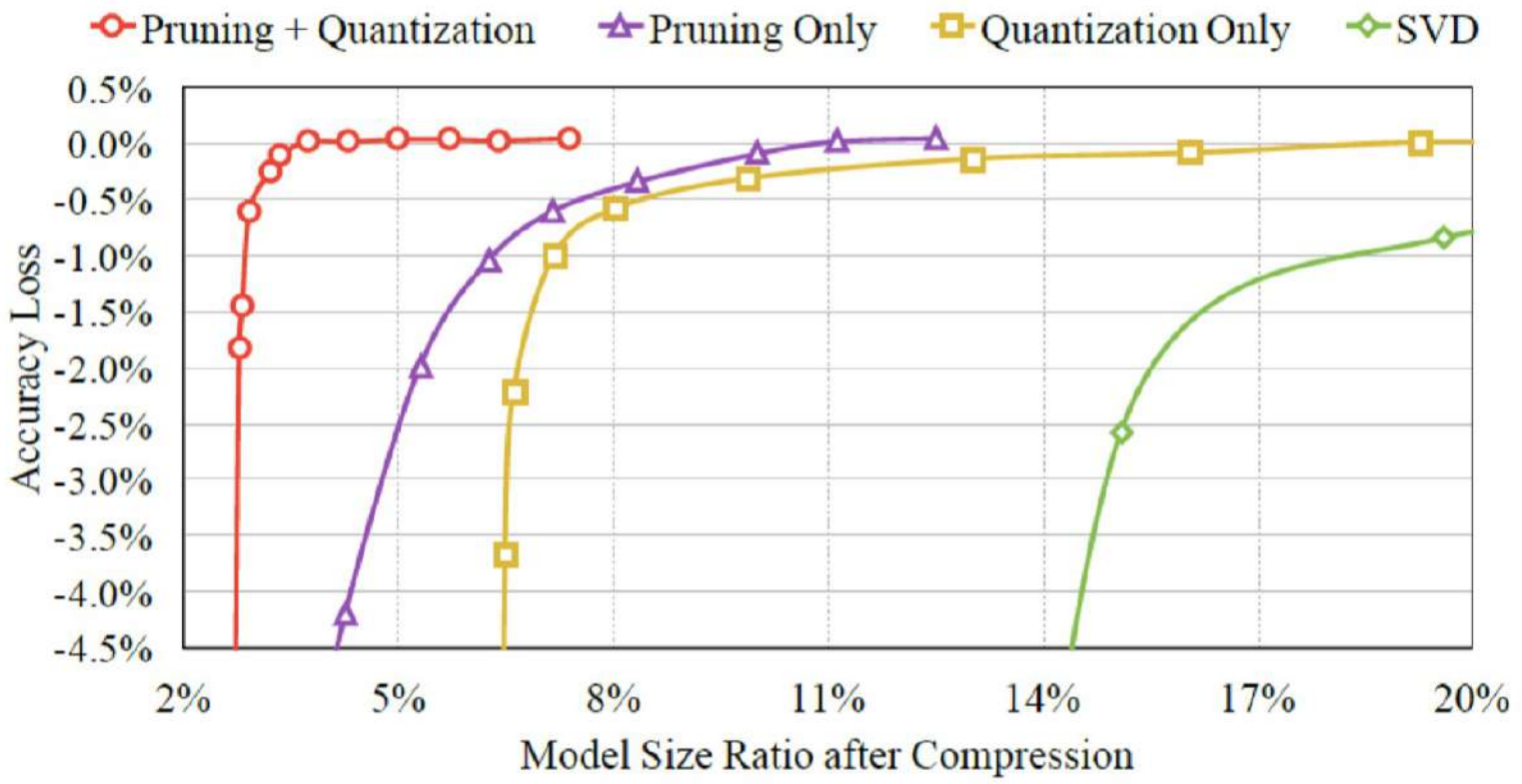


Figure 4.9: Accuracy vs. compression rates under different compression methods. Pruning and quantization works best when combined.

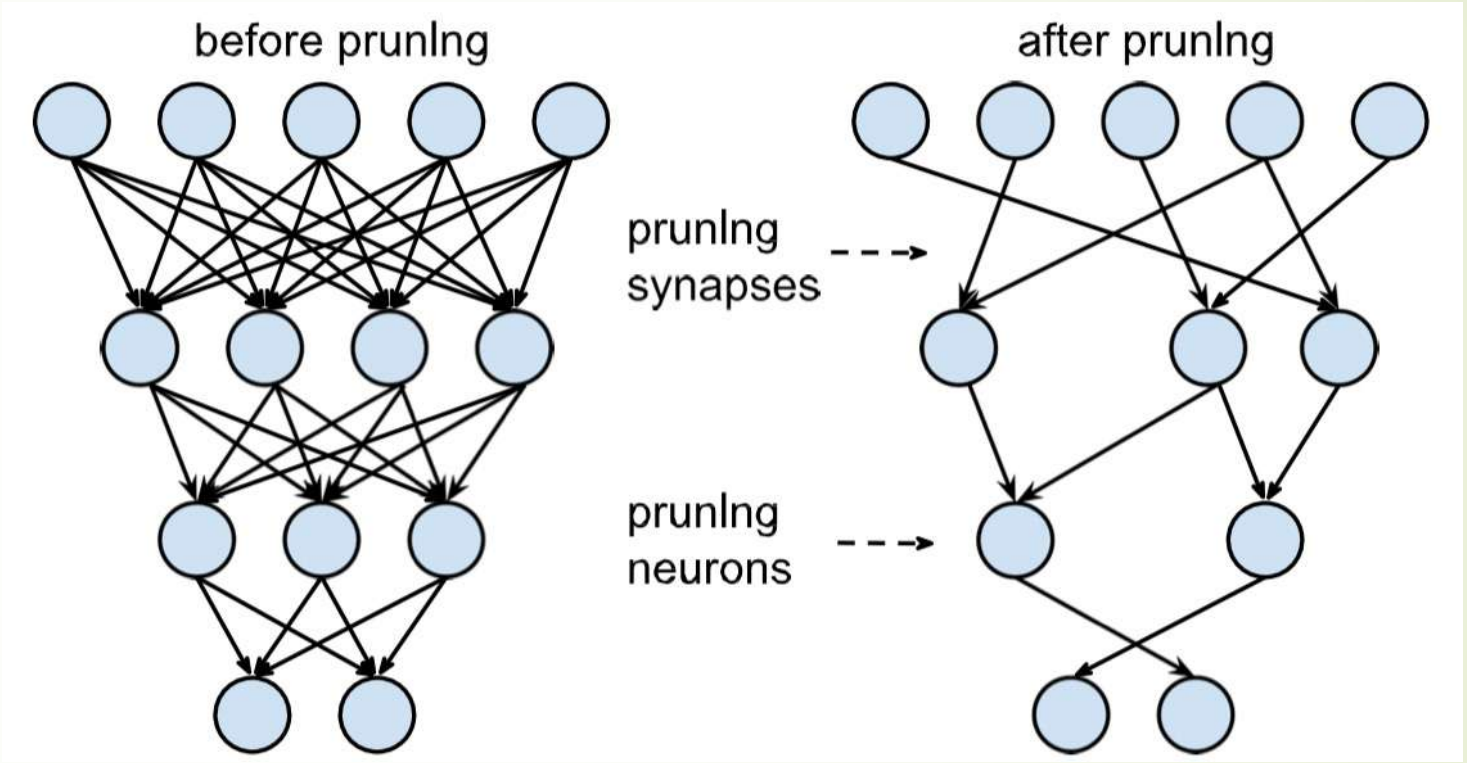
Pruning + Quantization:
Size: 30 – 50 Times compression
Memory Bandwidth: Reduced by 30 – 50 X
Binarization: 50 – 60 X faster, 30 X smaller

Network Pruning

- Not all weights are created equal importance
- Prune unimportance weight:

  If (w < threshold) w = 0
- Define Importance:

  L1 (better) or L2 Norm
- **Iterative pruning**



Some Conclusions

(1) 训练好模型，Pruning完，重新微调，才能恢复准确率。

(2) (**Weight Pruning**) Iterative pruning比direct pruning能取得更好的效果；
L2 regularization+ iterative prune + retrain效果好

(3) (**Weight Pruning**) 对于CNN模型来说，比如说AlexNet, 浅层的Conv对Pruning比较Sensitive (25%左右), 深层的Conv不太sensitive (50%左右), FC对Pruning也不sensitive (75%左右)

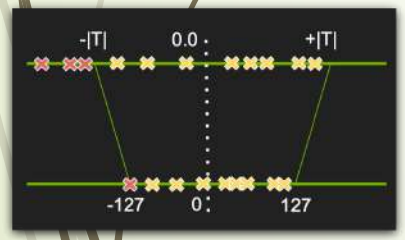
(4) (**Weight Pruning**) Fine grained level pruning需要有比较高的sparsity, 才有加速效果

(5) (**Filter Pruning**) Pruning小的参数值(L1 Norm better)要比随机Pruning效果好。

(6) (**Channel Pruning**) 大模型Pruning后变为小模型，从头开始训练，然后拉长训练时间，效果好

https://ocw.nctu.edu.tw/course/Lecture%20model%20acceleration%20and%20compression%2020191222.pdf

Weight and **Activation** Quantization



- Scalar Quantization: **reducing numerical precision**
- Vector Quantization: **remove redundancies**
- Definition:

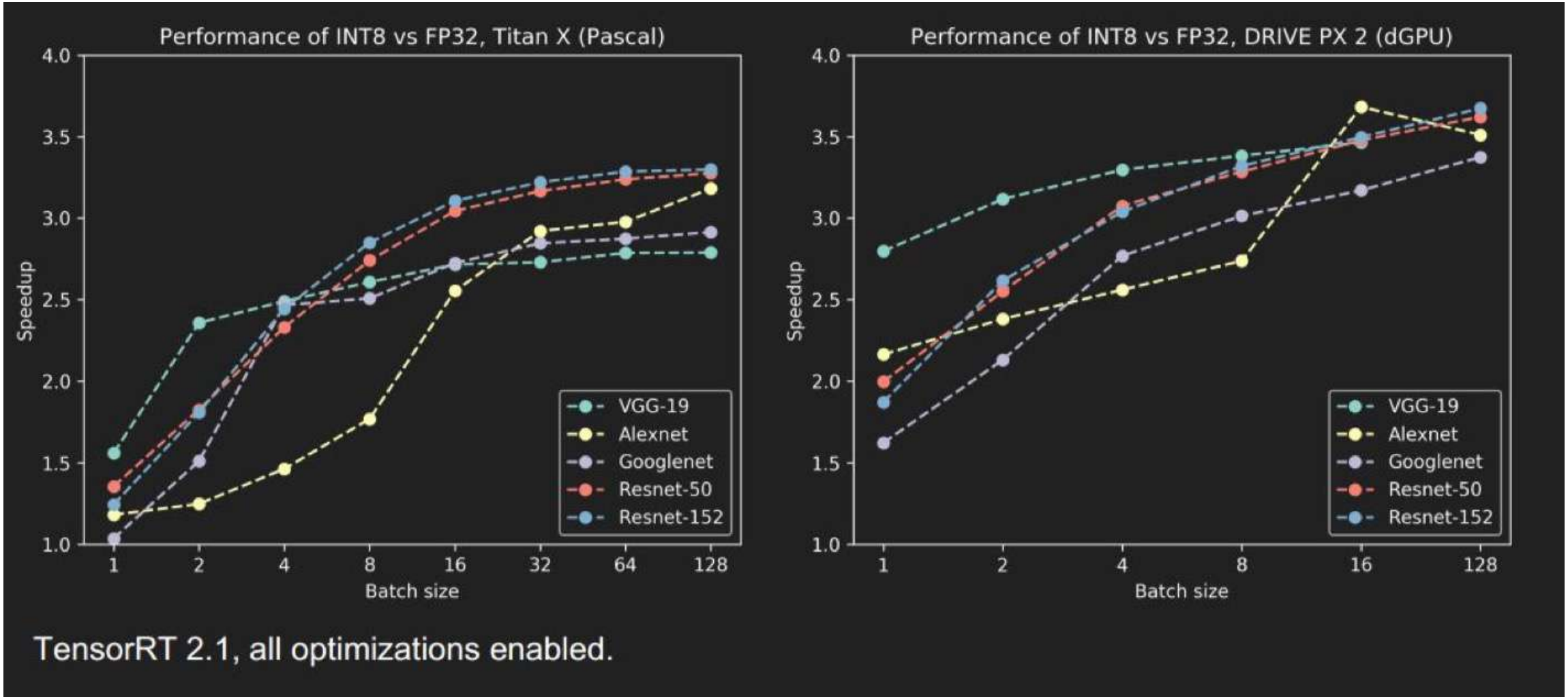
  32-bit -> 16-bit / 8-bit / 4-bit / binary / ternary

  float 16 / int 8

  encodes the **same information** as the original FP32 model
- TensorRT Method:

  (1) **KL divergence:** measures the amount of information lost when approximating a given encoding

  (2) **Calibration**: Quantized Distortions with different Saturation Threshold: Activation Values


Performance of INT8 vs FP32, Titan X (Pascal)
Speedup
Batch size
4.0
3.5
3.0
2.5
2.0
1.5
1.0
1
2
4
8
16
32
64
128
VGG-19
Alexnet
Googlenet
Resnet-50
Resnet-152
Performance of INT8 vs FP32, DRIVE PX 2 (dGPU)
Speedup
Batch size
4.0
3.5
3.0
2.5
2.0
1.5
1.0
1
2
4
8
16
128
VGG-19
Alexnet
Googlenet
Resnet-50
Resnet-152
TensorRT 2.1, all optimizations enabled.

Some Conclusions

(1) Weights Representation:

-> Conv: 4 bits; FC: 2 bits

(2) Pruning + Quantization

-> 30 – 50 X Size Compression, Memory Bandwidth: Reduced by 30 – 50 X

(3) Binary (+/-1)/ ternary (+/-1, 0) weights and activations Quantization:

-> 20, 50 – 60 X faster, 30 X smaller

(4) Low-rank Approximation: 2 – 5 X faster

(5) Winograd Convolution: 1.3 – 2.25 less multiplications

https://ocw.nctu.edu.tw/course/Lecture%20model%20acceleration%20and%20compression%2020191222.pdf

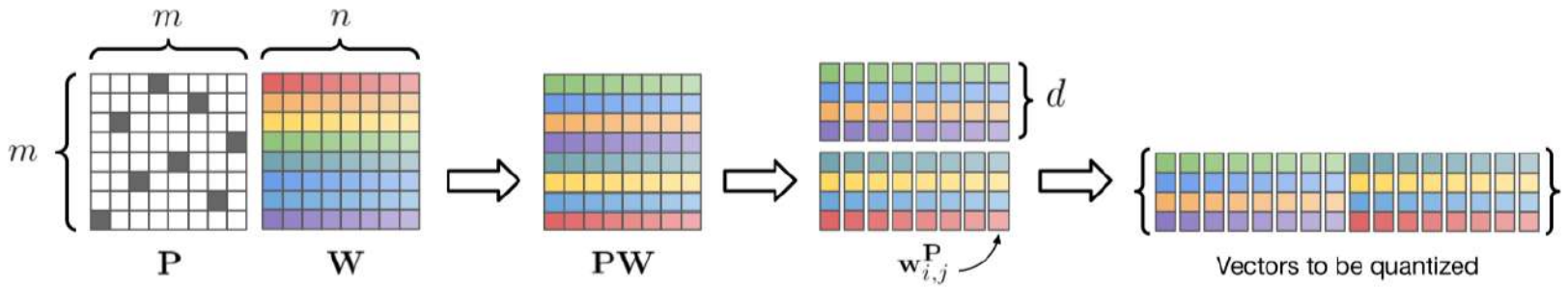


Figure 1: Permutation optimization of a fully-connected layer. Our goal is to find a permutation **P** of the weights **W** such that the resulting subvectors are easier to compress.

Permute, Quantize, and Fine-tune: Efficient Compression of Neural Networks

Julieta Martinez*[,1] Jashan Shewakramani*[,1,2] Ting Wei Liu*[,1,2]
Ioan Andrei Bârsan[1,3] Wenyuan Zeng[1,3] Raquel Urtasun[1,3]

[1]Uber Advanced Technologies Group [2]University of Waterloo [3]University of Toronto
{julieta,jashan,tingwei.liu,andreib,wenyuan,urtasun}@uber.com

Abstract

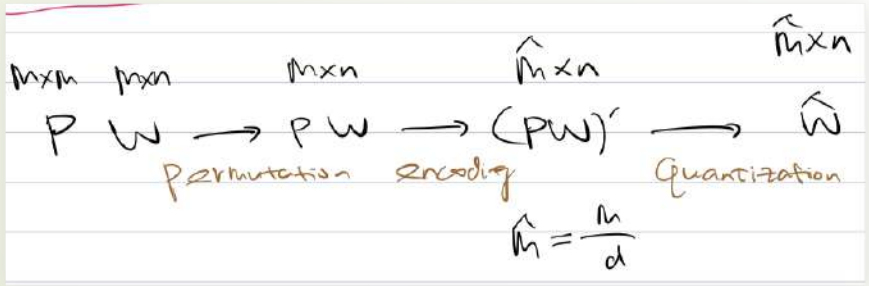
Compressing large neural networks is an important step for their deployment in resource-constrained computational platforms. In this context, vector quantization is an appealing framework that expresses multiple parameters using a single code, and has recently achieved state-of-the-art network compression on a range of core vision and natural language processing tasks. Key to the success of vector quantization is deciding which parameter groups should be compressed together. Previous work has relied on heuristics that group the spatial dimension of individual convolutional filters, but a general solution remains unaddressed. This is desirable for pointwise convolutions (which dominate modern architectures), linear layers (which have no notion of spatial dimension), and convolutions (when more than one filter is compressed to the same codeword). In this paper

are often overparameterized [5], which implies that it is possible to compress them – thereby reducing their memory and computation demands – without much loss in accuracy.

Scalar quantization is a popular approach to network compression where each network parameter is compressed individually, thereby limiting the achievable compression rates. To address this limitation, a recent line of work has focused on vector quantization (VQ) [13, 47, 54], which compresses multiple parameters into a single code. Conspicuously, these approaches have recently achieved state-of-the-art compression-to-accuracy ratios on core computer vision and natural language processing tasks [10, 48].

A key advantage of VQ is that it can naturally exploit redundancies among groups of network parameters, for example, by grouping the spatial dimensions of convolutional filters in a single vector to achieve high compression rates. However, finding which network parameters should be com-

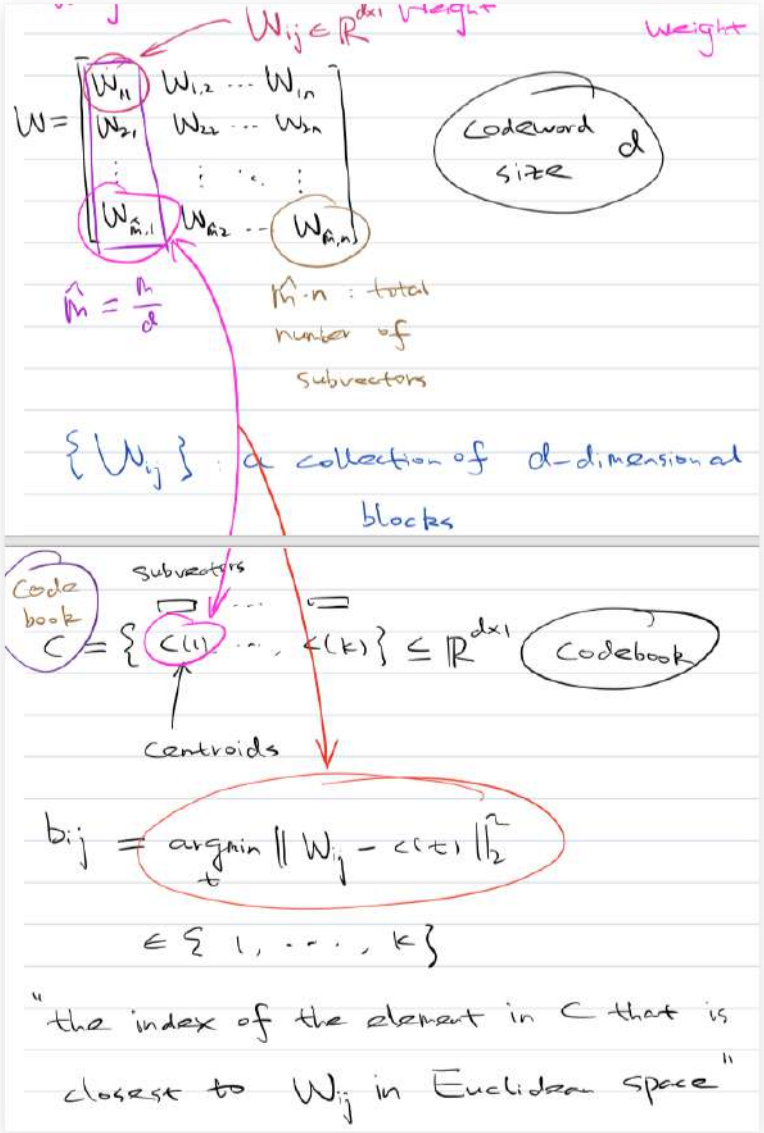
- **Motivation**: exploits the *invariance of neural networks* under permutation of their weights for the purpose of vector compression.
- **Why** $\widehat{W} \approx W$: Remove redundancies; Activations of the layer should be similar.
- **Why Permutation**: minimize the spread of the vectors by minimizing the determinant of the covariance of the resulting subvectors (Rate-distortion Theory)

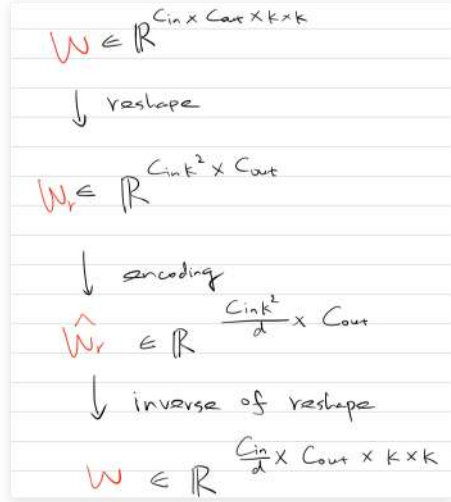


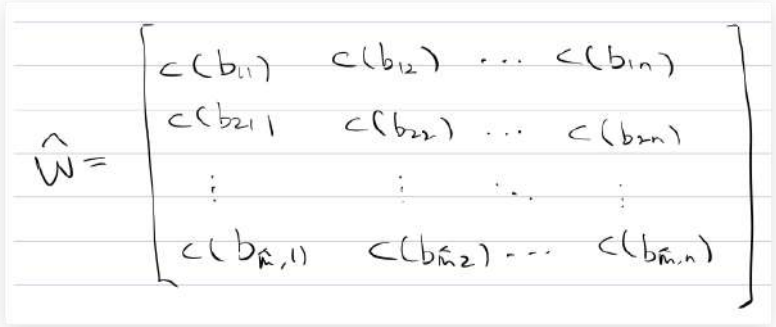
Algorithm 1 SR-C: Stochastic relaxation of k-means.

procedure SR-C($\{\mathbf{w}^{\mathbf{P}_t}_{i,j}\}, \mathbf{\Sigma}_t, k, T, \gamma$)
  $\mathbf{B}_t \leftarrow$ INITIALIZECODES(k)
  for $\tau \leftarrow 1, \dots, T$ **do**
    # Add scheduled noise to subvectors
    for $\mathbf{w}^{\mathbf{P}_t}_{i,j} \in \{\mathbf{w}^{\mathbf{P}_t}_{i,j}\}$ **do**
      $\mathbf{x}_{i,j} \sim \mathcal{N}(\mathbf{0}, \mathrm{diag}(\mathbf{\Sigma}_t))$
      $\hat{\mathbf{w}}^{\mathbf{P}_t}_{i,j} \leftarrow \mathbf{w}^{\mathbf{P}_t}_{i,j} + (\mathbf{x}_{i,j} \times (1 - (\tau/I))^{\gamma})$ Add noise
    end for
    # Noisy codebook update
    $\mathcal{C}_t \leftarrow \arg\min_{\mathcal{C}} \sum_{i,j} \|\hat{\mathbf{w}}^{\mathbf{P}_t}_{i,j} - \mathbf{c}(b_{i,j})\|_2^2$ Update Centroids
    # Regular codes update
    $\mathbf{B}_t \leftarrow \arg\min_{\mathbf{B}} \sum_{i,j} \|\mathbf{w}^{\mathbf{P}_t}_{i,j} - \mathbf{c}(b_{i,j})\|_2^2$
  end for
  return $\mathbf{B}_t, \mathcal{C}_t$
end procedure

$$E_t = \min_{\mathbf{P}_t, \mathbf{B}_t, \mathcal{C}_t} \frac{1}{\hat{m}n} \left\| \widehat{\mathbf{W}}_t - \mathbf{P}_t \mathbf{W}_t \right\|_F^2$$


$W_{ij} \in \mathbb{R}^{d \times 1}$ weight
weight
$W =$
codeword size d
$\hat{m} = \frac{m}{d}$
$\hat{m} \cdot n$: total number of subvectors
$\{W_{ij}\}$: a collection of d-dimensional blocks
Code book
Subvectors
$C = \{c(1), \ldots, c(k)\} \subseteq \mathbb{R}^{d \times 1}$
Codebook
Centroids
$b_{ij} = \underset{t}{\operatorname{argmin}} \| W_{ij} - c(t) \|_2^2$
$\in \{1, \ldots, k\}$
"the index of the element in C that is closest to W_{ij} in Euclidean space"


$W \in \mathbb{R}^{C_{in} \times C_{out} \times k \times k}$
reshape
$W_r \in \mathbb{R}^{C_{in}k^2 \times C_{out}}$
encoding
$\hat{W}_r \in \mathbb{R}^{\frac{C_{in}k^2}{d} \times C_{out}}$
inverse of reshape
$W \in \mathbb{R}^{\frac{C_{in}}{d} \times C_{out} \times k \times k}$


$\hat{W} = \begin{bmatrix} c(b_{11}) & c(b_{12}) & \cdots & c(b_{1n}) \\ c(b_{21}) & c(b_{22}) & \cdots & c(b_{2n}) \\ \vdots & \vdots & \ddots & \vdots \\ c(b_{\hat{m},1}) & c(b_{\hat{m}2}) & \cdots & c(b_{\hat{m},n}) \end{bmatrix}$

Low Complexity or Compact Model

- SqueezeNet: 50 X size compression
  - 1 X 1 Pointwise – (3 X 3 Conv + 1 X 1 Conv)
- MobileNet:
  - V1: Depthwise separable Conv → 3 X 3 Depthwise Conv + 1 X 1 Pointwise Conv

    -> 1 / Kernel_size^2 Params

    -> 3 X 3 DW后的ReLU6去掉后，Quantization好做
  - V2: Inverted residual + Linear bottleneck

    -> 1 X 1 Pointwise + 3 X 3 Depthwise + 1 X 1 Pointwise

    -> Short connection + wide middle layer
  - V3: V2 + Squeeze-and-Excite
- ShuffleNet:
  - V1: Point-wise Group Conv + Channel Shuffle

    -> 1/ Group Params
  - V2

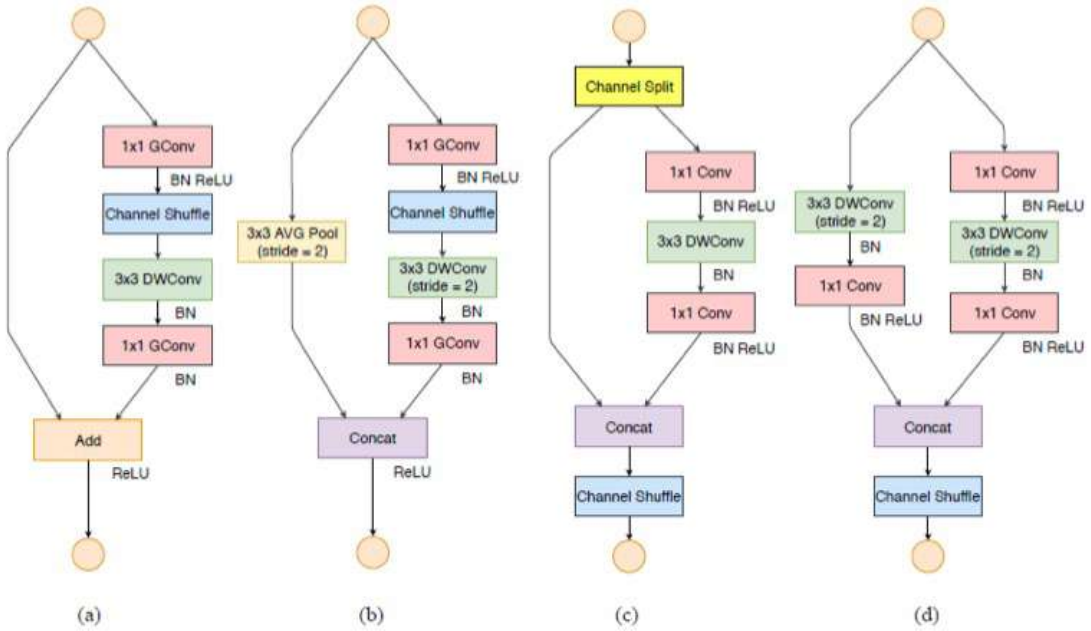


Fig. 3: Building blocks of ShuffleNet v1 [15] and this work. (a): the basic ShuffleNet unit; (b) the ShuffleNet unit for spatial down sampling (2×); (c) our basic unit; (d) our unit for spatial down sampling (2×). **DWConv**: depthwise convolution. **GConv**: group convolution.

ShuffleNet V2

- Short-cut Element-wise计算的部分：

  不用add，直接concat

- Excessive group convolution increases MAC

  不做Group Conv：直接做channel split，随后直接PW-DW-PW，到最后concat后再做channel Shuffle

谢谢！