TinyML and Efficient Deep Learning Computing

Himanshu Singh

Github

Deep Learning Basics

From efficiency viewpoint, a wide and shallow network is good, because we can execute operation parallelly and make use of full compute available. From accuracy viewpoint network should be deep.

Full Connected layer: $y_i = \sum_j w_{ij} x_j + b_i$

Input feature **X** shape	(n, c_i)
Output feature **Y** shape	(n, c_o)
Weight **W**	(c_o, c_i)
Bias **b**	$(c_o,)$

Convolution Layer:

	1D conv	2D conv	Grouped
Input feature **X** shape	(n, c_i, w_i)	(n, c_i, h_i, w_i)	(n, c_i, h_i, w_i)
Output feature **Y** shape	(n, c_o, w_o)	(n, c_o, h_o, w_o)	(n, c_o, h_o, w_o)
Weight **W**	(c_o, c_i, k_w)	(c_o, c_i, k_h, k_w)	$(g.c_o/g, c_i/g, k_h, k_w)$
Bias **b**	$(c_o,)$	$(c_o,)$	$(c_o,)$

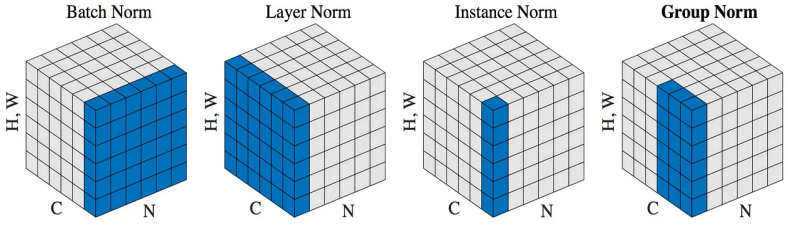
Depthwise Convolution Layer: Independent filter for each channel $g = c_i = c_o$

Normalization Layer

$$\hat{x}_k = \frac{1}{\sigma_i}(x_k - \mu_i)$$

$$\mu_i = \frac{1}{m}\sum_{k \in S_i} x_k$$

$$\sigma_i = \sqrt{\frac{1}{m}\sum_{k \in S_i}(x_k - \mu_i)^2}$$



Then learns a per-channel linear transform (trainable scale γ and shift β) to compensate for possible lost of representational ability.

$$y = \gamma_{ic}\hat{x}_i + \beta_{ic}$$

Activation Function: Sigmoid, ReLU, ReLU6, Leaky ReLU, Swish, Hard Swish, Tanh, GELU, ELU

Efficiency Metrics

Memory Related: #parameters, model size, total/peak #activations

Computation-Related: MAC, FLOP, FLOPS, OP, OPS

Latency and throughput are not related to each other. Suppose an algo take 50ms to process an image, while other algo process 4 images parallelly in 100ms. Throughput of first algo is 20images/s while second is 40 images/s

Latency

$$\text{Latency} \approx max(T_{computation}, T_{memory})$$

$$T_{computation} \approx \frac{\text{No. of operaions in Neural Network Model}}{\text{No. of operation that processor can process per second}}$$ NN Specification / HW Specification

$$T_{memory} \approx T_{\text{data movement of activations}} + T_{\text{data movements of weight}}$$

$$T_{\text{data movement of activations}} \approx \frac{\text{Neural Network Model Size}}{\text{Memory BW of processor}}$$ NN Specification / HW Specification

$$T_{\text{data movements of weight}} \approx \frac{\text{Input activation size + Output Activation size}}{\text{Memory BW of processor}}$$ NN Specification / HW Specification

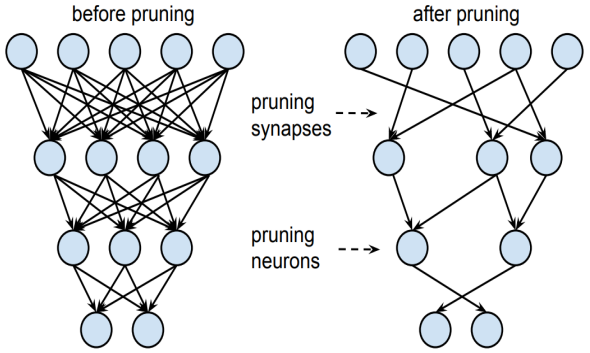
Data movement → More memory reference → More Energy

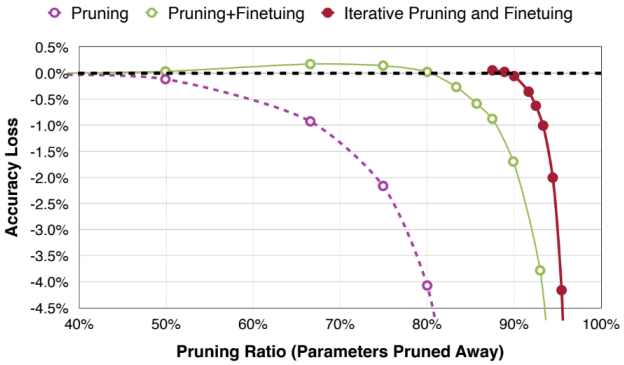
MAC Multiply-Accumulate operation (MAC) a ← a+ b*c

Layer	MACs (Batch size n =1)
Linear Layer	c_o*c_i
Convolution	$c_i*k_h*k_w*h_o*w_o*c_o$
Grouped Convolution	$c_i/g*k_h*k_w*h_o*w_o*c_o$
Depthwise Convolution	$k_h*k_w*h_o*w_o*c_o$

One Multiply-accumulate operation is two floating point operations(FLOP)
Floating Point Operations Per Second (FLOPS)

Pruning



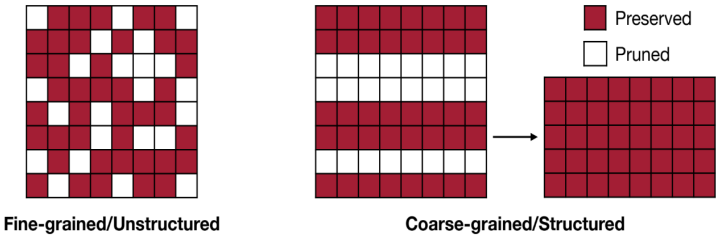


In general, we formulate as follows:
$\arg\min_{W_p} L(x; W_p)$ subject to $\|W_p\|_0 < N$

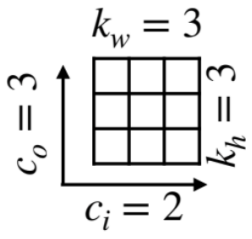
L represents the objective function of neural network, x is input, W is original weights, W_p is pruned weights. N is the target.

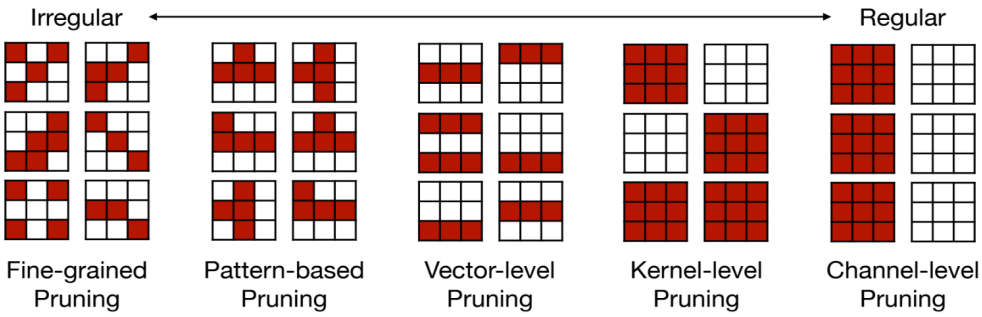
Fine -grained/Unstructured
- More flexible pruning index choice
- Hard to accelerate (irregular)

Coarse-grained/Structured
- Less flexible pruning index choice
- Easy to accelerate



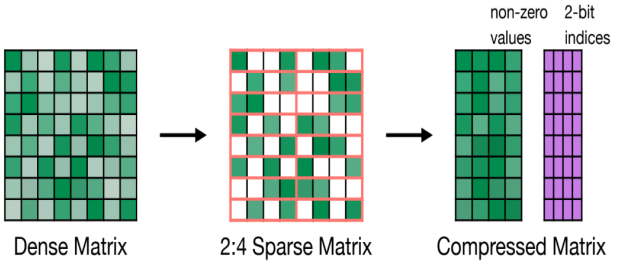
Convolutional layer





Fine grained Pruning
- Flexible Pruning Indices
- Usually larger compression ratio since we can flexibly find “redundant” weights
- Can deliver speed up on some custom hardware (EIE) but not GPU (easily)

Pattern-based Pruning N:M sparsity
- N:M sparsity means that in each contiguous M elements, N of them are pruned
- Classic case is 2:4



- Supported by NVIDIA's Ampere GPU architecture, which deliver 2x speed up.

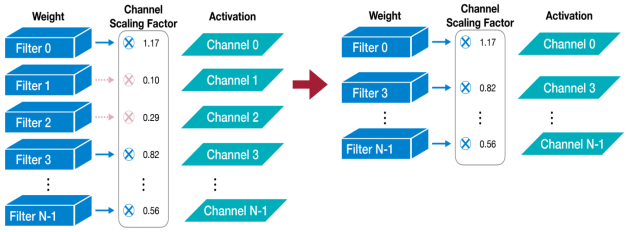
Channel Pruning

- Pro: Direct speed up due to reduced channel numbers (leading to an NN with smaller #channels)
- Con: Smaller compression ratio

Magnitude based Pruning: considers weights with larger aboslute vales are more important than other weights.

- For element-wise pruning, Importance $= |W|$
- For row-wise pruning,L_p-norm magnitude can be defined Importance $= \|W^{(S)}\|_p = \left(\sum_{i \in S} |w_i^p| \right)^{1/p}$

Scaling based Pruning: This pruning methods use scaling factors along with weight magnitudes for pruning. In this method, the trainable scaling factor is multiplied by the output of the channels. These scaling factors are used to calculate the importance of the channels. The filters/ output channels with a small scaling factor magnitude are pruned. The scaling factor can be reused from bach normalization layer



$$z_o = \gamma \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

Second-Order-based Pruning: Minimize the error on loss function introduced by pruning synapses. The induced error can be approximated by Taylor series

$$\delta L = L(x; W) - L(x; W_p = W - \delta W) = \sum_i g_i \delta w_i + \frac{1}{2} \sum_i h_{ii} \delta w_i^2 + \frac{1}{2} \sum_{i \neq j} h_{ij} \delta w_i \delta w_j + O(\|\delta W\|^3)$$

where, $g_i = \frac{\delta L}{\delta w_i}$ $h_{ij} = \frac{\delta^2 L}{\delta w_i \delta w_j}$

Optical Brain damage assume:

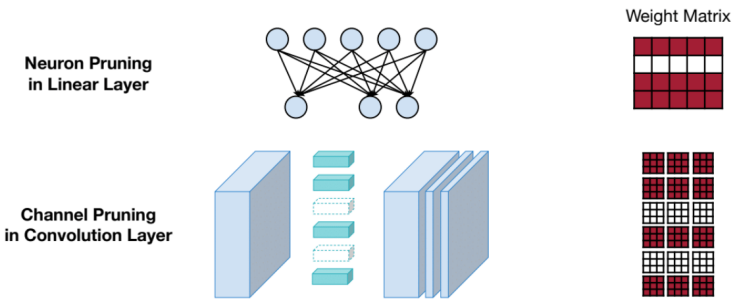
- L is nearly quadratic: last term is neglected
- NN Trainig has converged: first order terms are neglected
- Error caused by deleting each param is independent: cross terms are neglected.

$$\delta L_i = L(x; W) - L(x; W_p | w_i = 0) \approx \frac{1}{2} \sum_i h_{ii} \delta w_i^2$$

Synapses with smaller induced error $|\delta L_i|$ will be removed, Importance$_{w_i} = |\delta L_i| = \frac{1}{2} h_{ii} w_i^2$

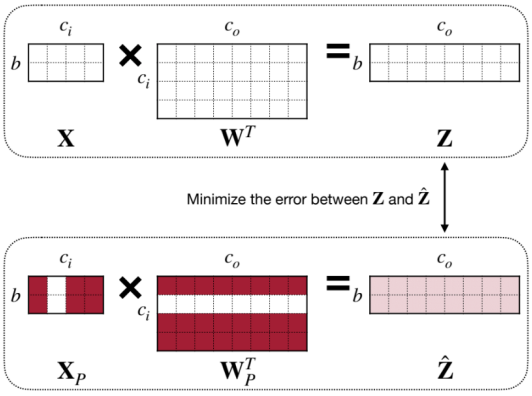
Selection of Neurons to Prune:

Neuron pruning is coarse-grained weight pruning



Percentage of Zero-Based Pruning: ReLU activation will generate zeros in output activation. Similar to magnitude of weight, the Average Percentage of Zero activation (APoZ) can be exploited to measure the importance of neuron

Regression-based Pruning: Instead of considering the pruning error of the objective function, regression-based pruning minimizes the reconstruction error of the corresponding layer's outputs.



$$Z = XW^T = \sum_{c=0}^{c_i-1} X_C W_c^T$$

The problem can be formulated as

$$\arg\min_{W,\beta} \|Z - \hat{Z}\|_F^2 = \|Z - \sum_{c=0}^{c_i-1} \beta_c X_c W_c^T\|_F^2$$

subject to $\|\beta\|_0 \le N_c$

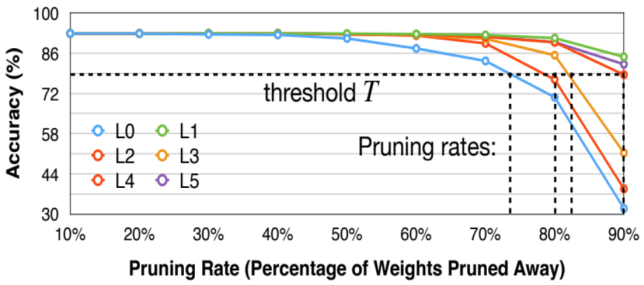
$\beta_c = 0$ means channel c is pruned. N_c is number of nonzero channels.

Solve:

- Fix W, solve β for selection selection
- Fix β, solve W to minimize reconstruction error

Pruning Ratio Non-uniform pruning is better than uniform shrinking

Analyse the sensitivity of each layer. We need different pruning ratio for each layer since different layer have different sensitivity. Some layers are more sensitive. Some layers are more redundant

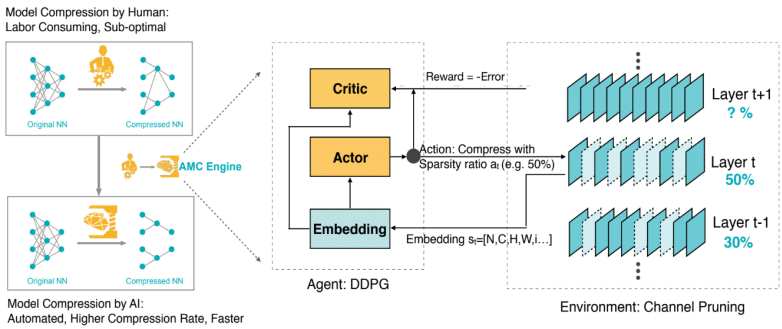


1. Pick layer L_i in the model
   - Prune on the layer L_i with pruning ratio $r \in \{0, 0.1, 0.2, \ldots, 0.9\}$
   - Observe the accuracy degrade ΔAcc_i^r for each pruning ratio
2. Repeat process for all layers
3. Pick degradation threshold T such that the overall pruning rate is desired

OPTIMAL? Maybe not! We don't consider interaction between layers.

Automatic Pruning AMC: AutoML for Model Compression

AMC uses following setup for the reinforcement learning problem:

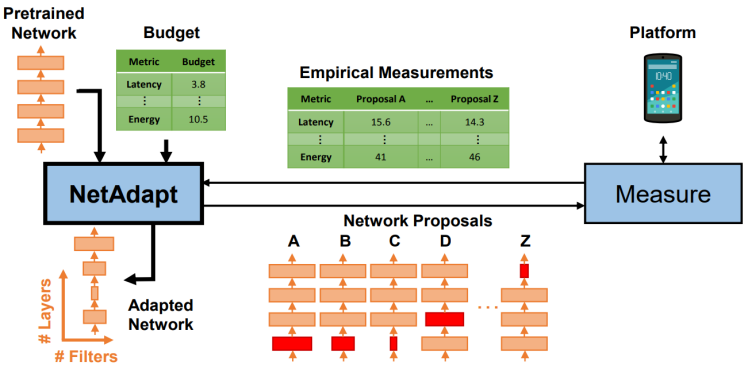


- State: features including layer indices, channel numbers, kernel sizes, FLOPs, ...
- Action: A continuous number (pruning ratio) $a \in [0, 1)$
- Agent: DDPG agent, since it support continuous action output
- Reward $R = \begin{cases} -Error, & \text{if satisfies constrains} \\ -\infty, & \text{if not} \end{cases}$

NetAdapt

Find per-layer pruning ratio to meet global resource constraint (e.g. latency energy)

For each iteration, reduce latency by certain amount ΔR



1. For each layer L_k (k in A-Z)
   - Prune layer s.t. latency reduction meets ΔR
   - Short-term fine-tune model; measure accuracy after fine tuning
2. Choose and prune the layer with highest accuracy

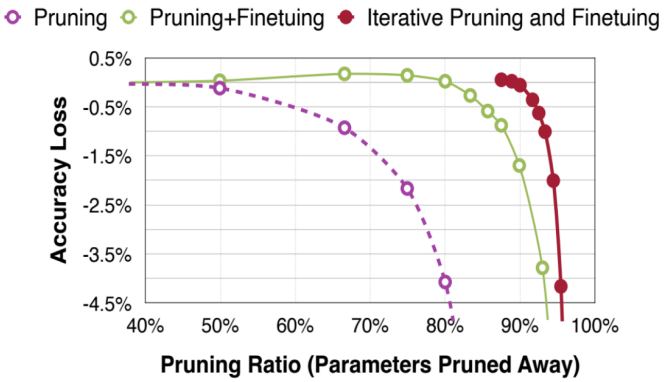
Repeat until total latency reduction satisfies the constraint. Long term fine-tune to recover accuracy.

Finetuning Pruned Neural Networks

Learning rate for fine-tuning is usually 1/100 or 1/10 of the original learning rate.

Iterative pruning gradually increases the target sparsity in each iteration e.g. boost pruning ratio from 5x to 9x on AlexNet compared to single-step aggressive pruning.

When training neural network or fine-tuning quantized neural network, regularization is added to loss term to (a) penalize non-zero parameters (b) encourage smaller parameters.



Examples:

1) Magnitude-based Fine-grained Pruning applies L2 regularization on weights

2) Network Slimming applies smooth L1 regularization on channels scaling factors.

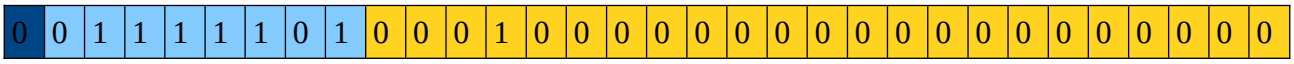
Quantization

32-bit floating point number in IEEE 754

Sign 8bit Exponent 23bit Fraction

$$(-1)^{sign} \times (1 + \text{Fraction}) \times 2^{\text{Exponent} - 127}$$

$$\mathbf{0.265625} = 1.0625 \times 2^{-2} = (1 + 0.0625) \times 2^{125 - 127}$$



- Subnormal number, when exponent bits are frozen to 0 $(-1)^{sign} \times \text{Fraction} \times 2^{1-127}$
- Normal number, when exponent bits are frozen to 1
  - If signed bit is 0 and fraction bits are all 0, it represent +∞
  - If signed bit is 1 and fraction bits are all 0, it represent -∞
  - If fractions bits are non-zero, it represent NaN (Not a Number)

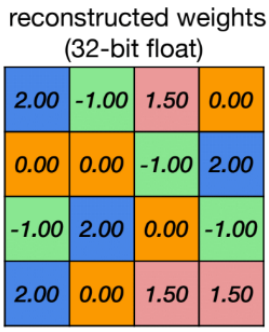
Mantissa is used for precision while exponent is used for dynamic range. Google Brain BF16 representation, Nvidia FP8(E4M3), Nvidia FP8 (E5M2) also comes to picture.

Format	Exponent (bits)	Fraction (bits)
IEEE754 32-bit (IEEE FP32)	8	23
IEEE754 16-bit (IEEE FP16)	5	10
Google Brain Float (BF16)	8	7
Nvidia FP8 (E4M3)	4	3
Nvidia FP8 (E5M2)	5	2

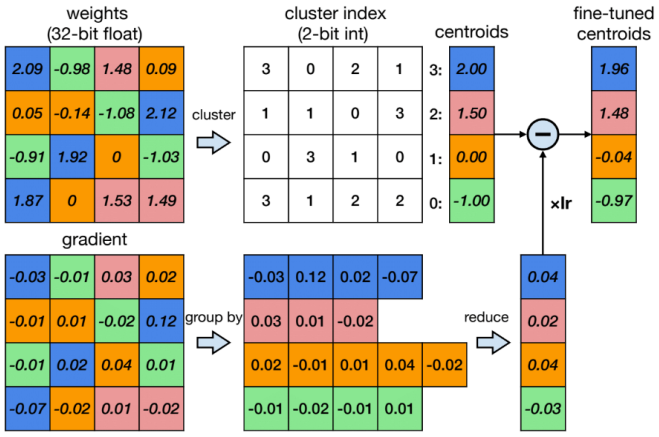
FP8(E4M3) does not have ∞, $S.1111.111_2$ is used for NaN (Largest value 448

FP8(E5M2) does have ∞ ($S.11111.00_2$) and Nan ($S.11111.XX_2$) (Largest value 57334.)

K-Means-based weight Quantization



Storage (4x4 weight matrix):
Intially 32bit x 16 = 512 bit
After quantization: Indexes 2 bit x16 + codebook 32bits x 4 =
= 160bit
Reduction 3.2x
Assume N bit quantization, and #param M >> 2^N
Initially 32bit x M = 32M bit
After quantization: Indexes N bit x M + codebook 32bits x 2^N =
≈ NM bit
Reduction 32/N



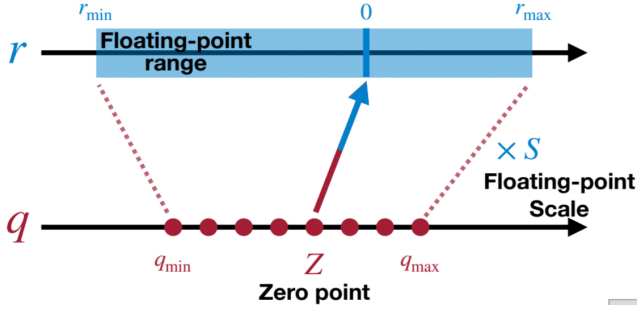
Fine-tuning Quantized weights
First do a group-by operation on gradients.
Reduce them to single value (summing or take mean) multiply them with a learning rate and shifts the centroids.
Summary:
The weights are decompressed using lookup table (i.e. codebook) during runtime inference.
K-means-based weight Quantization only saves storage cost of neural network model. All the computation and memory access are still floating.

Linear Quantization

An affine mapping of integers to real numbers.

$$r = S(q - Z)$$

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}} \qquad Z = \text{round}\left(q_{min} - \frac{r_{min}}{S}\right)$$



Consider following matrix multiplication

$$Y = WX$$
$$S_Y(q_Y - Z_Y) = S_W(q_W - Z_W).S_X(q_X - Z_X)$$
$$q_Y = \frac{S_W S_X}{S_Y}(q_W - Z_W)(q_X - Z_X) + Z_Y$$
$$q_Y = \frac{S_W S_X}{S_Y}(q_W q_X - Z_W q_X - Z_X q_W + Z_W Z_X) + Z_Y$$

Use N bit Integer Multiplication and 32-bit Integer Addition/Subtraction.
Pre-compute $-Z_X q_W + Z_W Z_X$ No run-time overhead.

Empirically, the scale $\frac{S_W S_X}{S_Y}$ is always (0,1) $\frac{S_W S_X}{S_Y} = 2^{-n} M_0$, where $M_0 \in [0.5, 1).2^{-n}$ is simple bit shift operation and multiplication with M0 is fixed-point multiplication.

$$q_Y = \underbrace{\frac{S_W S_X}{S_Y}}_{\text{Rescale to N-bit int}} (q_W q_X - Z_W q_X \underbrace{- Z_X q_W + Z_W Z_X}_{\text{Precompute}}) + \underbrace{Z_Y}_{\text{N-bit int addition}}$$

If we set zero point $Z = 0, Z_W = 0$

$$q_Y = \frac{S_W S_X}{S_Y}(q_W q_X - Z_X q_W) + Z_Y$$

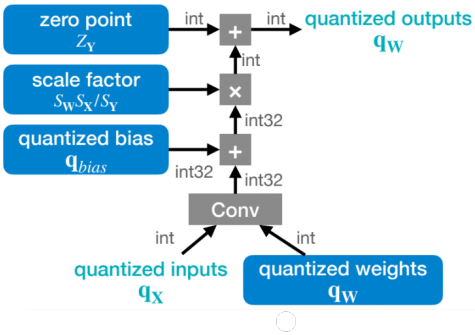
Now consider fully connected layer with bias

$$Y = WX + b$$
$$S_Y(q_Y - Z_Y) = S_W(q_W - Z_W).S_X(q_X - Z_X) + S_b(q_b - Z_b)$$
$$\downarrow Z_W = 0$$
$$S_Y(q_Y - Z_Y) = S_W S_X(q_W q_X - Z_X q_W) + S_b(q_b - Z_b)$$
$$\downarrow Z_b = 0, S_b = S_w S_X$$
$$S_Y(q_Y - Z_Y) = S_W S_X(q_W q_X - Z_X q_W + q_b)$$
$$q_Y = \frac{S_W S_X}{S_Y}(q_W q_X + \underbrace{q_b - Z_X q_W}_{\text{Precompute}}) + Z_Y$$
$$\downarrow q_{bias} = q_b - Z_X q_W$$
$$q_Y = \frac{S_W S_X}{S_Y}(q_W q_X + q_{bias}) + Z_Y$$

Note: q_b and q_{bias} are 32 bits

For Convolution, results will be the same

$q_Y = \frac{S_W S_X}{S_Y}(\text{Conv}(q_W q_X + q_{bias})) + Z_Y$, both q_b and q_{bias} are 32 bits

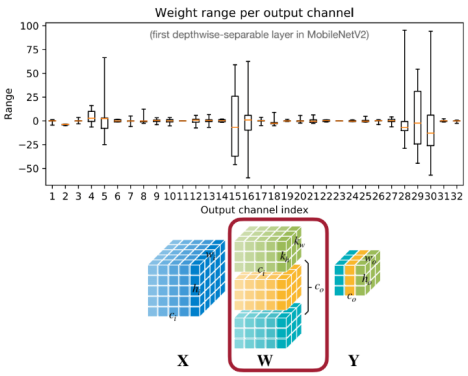


		K-Means-based Quantization	Linear Quantization	Binary/Ternary Quantization
Storage	Floating-Point Weights	Integer Weights, Floating Point Codebook	Integer Arithmetic	Binary/Ternary Weights
Computation	Floating-Point Arithmetic	Floating Point Arithmetic	Integer Arithmetic	Bit Operations

Post-Training Quantization

Quantization Granularity:

- Per-Tensor Quantization
- Per-Channel Quantization
- Group Quantization
  - Per-Vector Quantization
  - Shared Micro-Exponent (MX) data type



Per-Tensor Quantization

Using single scale S for whole weight tensor. Works wells for large models, accuracy drops for small models.

Failure: Large differences in ranges of weight for different output channels

Solution: Per-Channel Quantization

Per-Channel Quantization

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	-1.53	1.49

Per-Channel Quantization

$$|r|_{max\,0}=2.09\;S_0=2.09$$
$$|r|_{max\,1}=2.12\;S_0=2.12$$
$$|r|_{max\,0}=1.92\;S_0=1.92$$
$$|r|_{max\,0}=1.87\;S_0=1.87$$

1	0	1	0
0	0	-1	1
0	1	0	-1
1	0	1	1

Quantized

2.09	0	2.09	0
0	0	-2.12	2.12
0	1.92	0	-1.92
1.87	0	1.87	1.87

Reconstructed

$$\|W-S\odot q_W\|=2.08$$

Per-Tensor Quantization

$$|r|_{max}=2.12$$
$$S=\frac{|r|_{max}}{q_{max}}=\frac{2.12}{2^{2-1}-1}=2.12$$

1	0	1	0
0	0	-1	1
0	1	0	0
1	0	1	1

Quantized

2.12	0	2.12	0
0	0	-2.12	2.12
0	2.12	0	0
2.12	0	2.12	2.12

Reconstructed

$$\|W-S\odot q_W\|=2.28$$

VS-Quant: Per-vector Scale Quantization

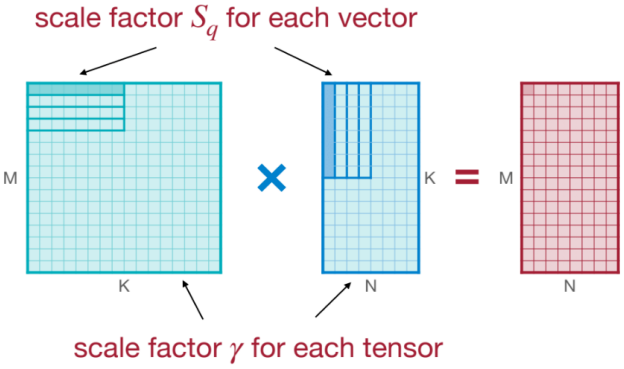
$$r=S(q-Z)\rightarrow r=\gamma.S_q(q-Z)$$

γ is a floating-point coarse grained scale factor

S_q is integer per-vector scale factor

This achieves a balance between accuracy and HW efficiency by:

- More expensive floating point scale factors at coarser granularity
- Less expensive integer scale factors at finer granularity



Multi-Level Quantization Scheme

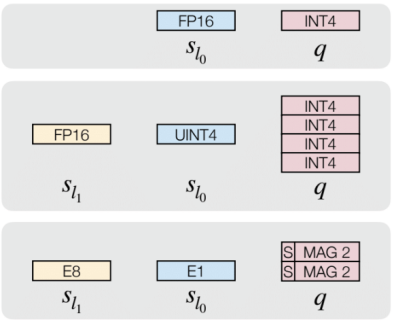
$$r=(q-Z).s_{l_0}.s_{l_1}\ldots$$

r: real number value

q: quantized value

z: zero point (z = 0 is symmetric quantization)

s: scale factors of different levels



Quantization Approach	Data Type	L0 Group size	L0 Scale Data Type	L1 Group Size	L1 Scale Data Type	Effective Bit Width
Per Channel Quant	INT4	Per Channel	FP16	-	-	4
VSQ	INT4	16	UINT4	Per Channel	FP16	4+4/16=4.25
MX4	S1M2	2	E1M0	16	E8M0	3+1/2+8/16=4
MX6	S1M4	2	E1M0	16	E8M0	5+1/2+8/16=6
MX9	S1M7	2	E1M0	16	E8M0	8+1/2+8/16=9

Dynamic Range for Activation Quantization

$$\hat{r}^{(t)}_{max,min} = \alpha . r^{(t)}_{max,min} + (1-\alpha).\hat{r}^{(t-1)}_{max,min}$$

Type1: During Training

- Exponential Moving Averages (EMA)
  - observed ranges are smoothed across thousands of training steps

Type2: By running a few "calibration" batches of samples on trained FP32 model

- spending dynamic range on the outlier hurts the representation ability
- use mean of min/max of each sample in the batches
- Analytical calculations

  Minimize the mean-square error between inputs X and (reconstructed) quantized inputs Q(X)

$$\min_{|r|} E[(X - Q(x))^2]$$

  Minimize loss of information: Loss of information is measured by KL divergence

Adaptive Rounding for Weight Quantization

Instead of $\lfloor w \rceil$, we want to choose $\{\lfloor w \rfloor, \lceil w \rceil\}$ to get best reconstruction. We took learning-based method to find quantized value $\widetilde{w} = \lfloor \lfloor w \rfloor + \delta \rceil, \delta \in [0,1]$

$$\underset{V}{argmin} \|Wx - \widetilde{W}x\|_F^2 + \lambda f_{reg}(V)$$

$$\underset{V}{argmin} \|Wx - \lfloor \lfloor W \rfloor + h(V) \rceil x\|_F^2 + \lambda f_{reg}(V)$$

V is random variable as same size of input. h() is function to map the range (0,1), such as rectified sigmoid. $f_{reg}(V)$ is regularization encourages h(V) to be binary

<u>Quantization-Aware Training</u>

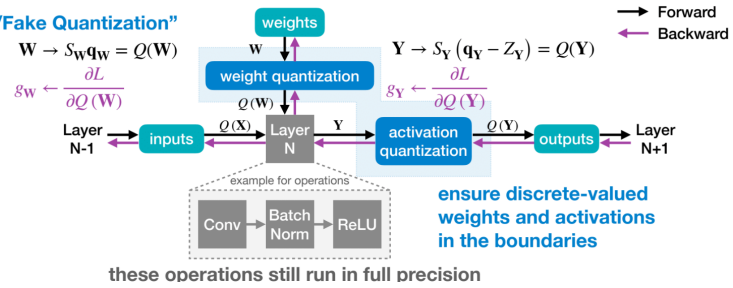
A full precision copy of weights is maintained throughout the training. A small gradients are accumulated without loss of precision. Once model is trained, only quantized weights are used for inference.

Problem: Quantization is discrete-values, thus derivate is 0 almost everywhere. The NN will learn nothing since gradient become 0 and weights wont get updated.

$$g_w = \frac{\delta L}{\delta W} = \frac{\delta L}{\delta Q(W)} \cdot \frac{\delta Q(W)}{\delta W} = 0$$

<u>Straight-Through Estimator(STE)</u> passes the gradients through quantization as if it has been identity function

$$g_w = \frac{\delta L}{\delta W} = \frac{\delta L}{\delta Q(W)}$$



Binary/Ternary Quantization

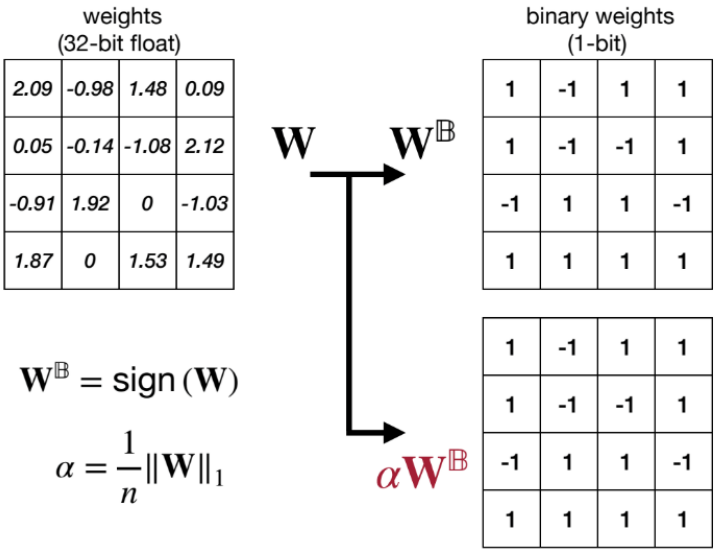
Weight Binarization

Deterministic Binarization: directly computes the bit value based on threshold, usually 0, resulting in a sign function

$$q = sign(r) = \begin{cases} +1, & r \geq 0 \\ -1, & r < 0 \end{cases}$$

Stochastic Binarization: use global stats tics or value of input data to determine the probability of being +1 or -1 e.g. Binary Connect (BC)

$$q = sign(r) = \begin{cases} +1, & \text{with probability } p = \sigma(r) \\ -1, & \text{with probability } 1-p \end{cases}, \text{where } \sigma(r) = min\left(max\left(\frac{r+1}{2}, 0\right), 1\right)$$



AlexNet-based Network	ImageNet Top-1 Accuracy Delta
BinaryConnect	-21.2%
Binary Weight Network (BWN)	0.2%

$$\|\mathbf{W} - \mathbf{W}^{\mathbb{B}}\|_F^2 = 9.28$$

scale
(32-bit float)

$$\times \ 1.05 = \frac{1}{16}\|\mathbf{W}\|_1$$

$$\|\mathbf{W} - \alpha\mathbf{W}^{\mathbb{B}}\|_F^2 = 9.24$$

Weights and Activations Binarization

$$y_i = \sum_j W_{ij} x_j$$
$$= 1\times 1 + (-1)\times 1 + 1\times(-1) + (-1)\times 1$$
$$= 1 + (-1) + (-1) + (-1)$$
$$= -2$$

$$y_i = -n + 2.\sum_j W_{ij} \, xnor \, x_j$$
$$1 \, xnor \, 1 + 0 \, xnor \, 1 + xnor \, 0 + 0 \, xnor \, 1$$
$$1 + 0 + 0 + 0 = 1$$
$$1\times 2 - 4 = -2$$

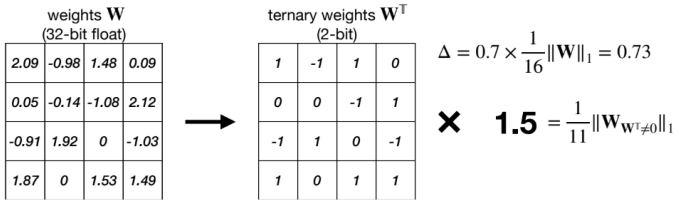
$$y_i = -n + popcount(W_i \, xnor \, x) \ll 1$$

Input	Weight	Operations	Memory	Computation
$\Re$	$\Re$	$+\times$	1x	1x
$\Re$	B	$+-$	~32x less	~2x less
B	B	xnor, popcount	~ 32x less	~ 58x less

Ternary Weight Quantization

Weights are quantized to +1, -1 and 0

$$q = \begin{cases} r_t, & r > \Delta \\ 0, & |r| \leq \Delta, \\ -r_t, & r < -\Delta \end{cases} \text{where } \Delta = 0.7 \times \mathbb{E}(|r|), r_t = \mathbb{E}_{|r|>\Delta}(|r|)$$



$$\Delta = 0.7 \times \frac{1}{16}\|\mathbf{W}\|_1 = 0.73$$

$$\times \ 1.5 = \frac{1}{11}\|\mathbf{W}_{\mathbf{W}^T \neq 0}\|_1$$

Trained Ternary Quantization (TTQ)

Instead of using fixed scale r_t, TTQ introduce two trainable parameters w_p and w_n to represent the positive and negative scales in quantization

$$q = \begin{cases} w_p, & r > \Delta \\ 0, & |r| \le \Delta \\ -w_n, & r < -\Delta \end{cases}$$