Network Compression – Part 1

Lecture 9 for Advanced Deep Learning Systems

Aaron Zhao, Imperial College London, a.zhao@imperial.ac.uk

Table of contents

Introduction

Neural Network Compression

Assume we have a pre-trained network f_θ, how can we best approximate it using a much smaller network $f'_{\theta'}$?

We are going to discuss a few popular methods in two lectures

- Network Pruning
    - Fine-grained Pruning
    - Coarse-grained Pruning
    - Pruning at initialization (The lottery ticket hypothesis)
- Quantization
    - Different arithmetic schemes
    - Different tricks to boost the performance of quantization
    - Extremely low-precisions (binary and ternary)

Pruning

Network Pruning

Fine-grained pruning normally refers to a pruning strategy that exploits element-wise sparsity

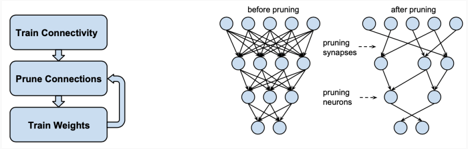
Element-wise sparsity means each entry has the probability to be zeroed out.

$$W_s = M \odot W \tag{1}$$

where $\odot$ represents an element-wise Hadamard product between two matrices.

Fine-grained Network Pruning

- Sparsity on both sides (activation and weight)
- Irregular sparsity is hard to utilize
- Re-training brings back accuracy

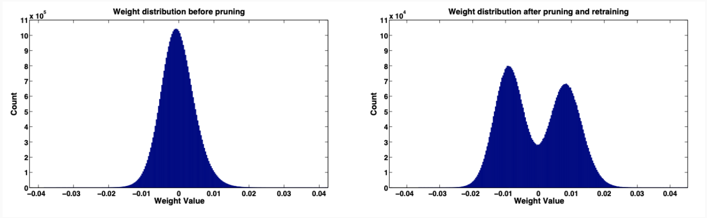


Using normalization terms (L1/L2) in the loss to encourage more sparsity.

$$\mathcal{L}' = \mathcal{L} + \lambda(\|\mathrm{w}\|_n) \tag{2}$$

where $\mathcal{L}$ is the original cross-entropy loss and $||.||_n$ takes the l_n norm.

Fine-grained Network Pruning



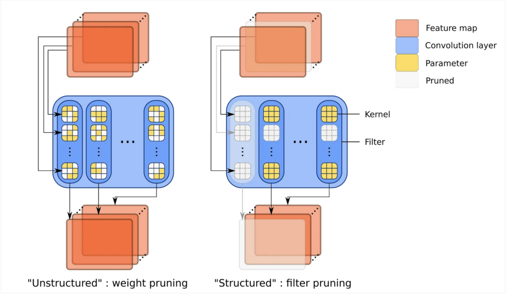
- Ignored the values that are actually zeroed out.
- Original weight distribution has a ‘normal’ shape

Fine-grained Network Pruning

Name	Density	Compression rate
VGG7 + Cifar10	16%	6×
AlexNet + ImageNet	11%	9×
VGG16 + ImageNet	7.5%	12×

- All the above networks have less than 0.1% accuracy drop after pruning with iterative re-training.
- Pruning is effective if you have a large network with an easy task.
- Fine-grained sparsity does not always translate to performance boost.

Coarse-grained Network Pruning


Feature map
Convolution layer
Parameter
Pruned
Kernel
Filter
"Unstructured" : weight pruning
"Structured" : filter pruning

- Channel pruning (Remove channels C_i or C_o from the $C_i \times C_o \times K \times K$ volume)
- Kernel pruning (Remove kernels $K \times K$ from the $C_i \times C_o \times K \times K$ volume)

Channel Pruning

Pruning can be viewed as a problem of finding the correct way to rank the importance of the components (eg. individual weight, filters, kernels).

This can be done through estimating the importance of weights (eg. l_p norm of weight) or activations.

If we use weights, consider $w \in \mathcal{R}^{C_i \times C_o \times K \times K}$, the scoring function for each output channel can be:

$$s_w(i) = ||w[:, i, :, :]||_p \tag{3}$$

Channel Pruning

If we consider activations, consider $y = wx$ and $y \in \mathcal{R}^{C_o \times K \times K}$, the scoring function for each channel can be:

$$s_a(i) = ||a[i, :, :]||_p \tag{4}$$

Notice now we can construct more complex scoring by considering both the importance of weights and activations. Consider to free hyperparameters α and β:

$$s(i) = \alpha s_w(i) + \beta s_a(i) \tag{5}$$

Channel Pruning: Network Slimming

Another approach (Network Slimming) is to associate a scaling variable $r \in \mathcal{R}^{C_o}$ with the output values, and let SGD decide which channel is more important. We use r as a proxy to measure the importance of channels

Train with

$$y' = r \odot y \tag{6}$$

and a regularized loss

$$\mathcal{L}' = \mathcal{L} + \lambda \sum_{i=1}^{C_o} (\|r[i]\|_p) \tag{7}$$

With a new scoring function

$$s(i) = r[i] \tag{8}$$

Channel Pruning: Connecting it to BN

Batch normalization has been adopted by most modern CNNs as a standard approach to achieve fast convergence and better generalization.

This is normally inserted after convolutional layers:

$$y = \gamma(\frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}) + \beta \tag{9}$$

γ and β are trainable parameters, where μ and σ are approximated through moving values (eg. moving means).

γ values are directly used for ranking in Network Slimming!

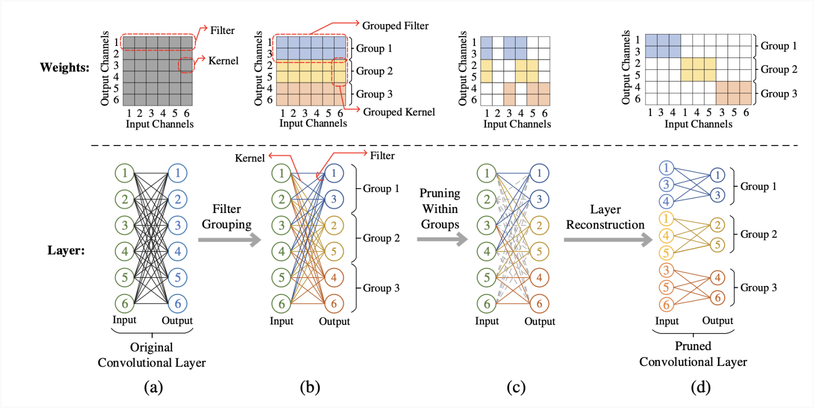
Kernel Pruning

Kernel pruning removes kernels $K \times K$ from the $C_i \times C_o \times K \times K$ volume

Naively remove kernels brings a problem: the resulted computation pattern is also Irregular!

The only way to maintain this regularity is to remove kernels so that the remaining ones form equal-sized groups!

Kernel Pruning



Kernel Pruning

After reshaping, this is similar to grouped convolution, or similar spirit to depth-wise convolution.

Original computation is $C_i = 6, C_o = 6$, we have in total 36 kernels.

Now is 3 sets of $C_i' = 3, C_o' = 2$ convolutions, we have in total $3 \times 3 \times 2 = 18$, this means $2\times$ decrease in terms of FLOPs.

Pruning at initialization

We discussed how trained networks can be pruned. What about training?

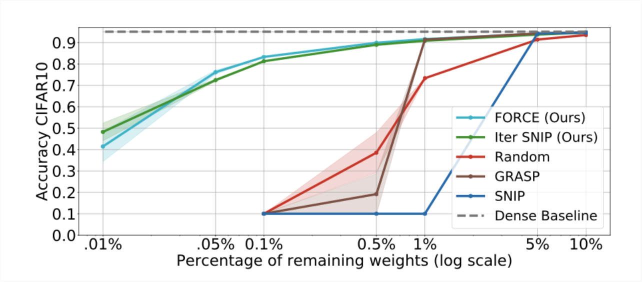
We can prune at initialization (sparse training).

The lottery ticket hypothesis: dense, randomly-initialized, feed-forward networks contain subnetworks (winning tickets) that—when trained in isolation— reach test accuracy comparable to the original network in a similar number of iterations.

Pruning at initialization

We can explore different metrics such as weight magnitude, gradient norm, or hessian of the loss with respect to the weights (SNIP).

Or we can iteratively apply metrics (iterative SNIP, FORCE): iterative process that allows exploration by allowing already pruned parameters to resurrect at later stages.



Quantization

Network Quantization

Quantisation methods allow parameters to be represented with much narrower bit-widths than the 32-bit long floating-point numbers

Converting numbers to fixed-point representations drastically reduces computation and memory requirements.

An n-bit fixed-point number with a binary point position p can represent a value $\hat{x}$ with:

$$\hat{x} = 2^{-p} \times m_n m_{n-1} \dots m_1, \tag{10}$$

Notice fixed-point arithmetic is a linear arithmetic.

Non-linear Arithmetic

Standard floating-point

A standard IEEE floating-point number is defined as a 4-tuple, (s, e, m, b). $s \in \{0, 1\}$ is the sign bit, $e \in \mathbb{N}$ is the exponent field; $b \in \mathbb{N}$ is the exponent bias; and $m \in \mathbb{N}$ is the mantissa.

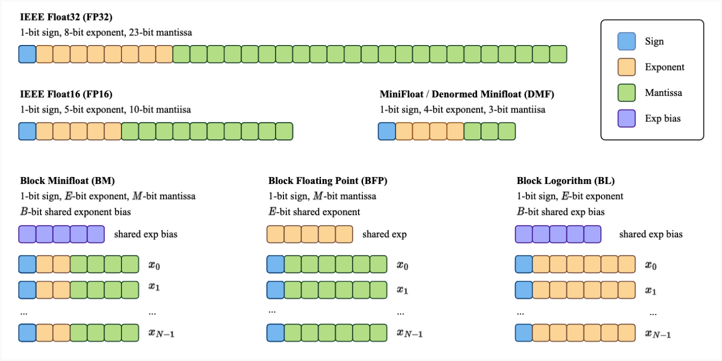
`float32` (FP32) number has $E = 8$ and $M = 23$, where the other bit is used as the sign bit.

`float16` (FP16) has $E = 5$ and $M = 10$.

`MiniFloat:  E,M` this allows custom exponent and mantissa widths.

Non-linear Arithmetic

Various block-based arithmetic (BFP, BL)



IEEE Float32 (FP32)
1-bit sign, 8-bit exponent, 23-bit mantissa
IEEE Float16 (FP16)
1-bit sign, 5-bit exponent, 10-bit mantiisa
MiniFloat / Denormed Minifloat (DMF)
1-bit sign, 4-bit exponent, 3-bit mantiisa
Sign
Exponent
Mantissa
Exp bias
Block Minifloat (BM)
1-bit sign, E-bit exponent, M-bit mantissa
B-bit shared exponent bias
shared exp bias
x_0
x_1
...
x_{N-1}
Block Floating Point (BFP)
1-bit sign, M-bit mantissa
E-bit shared exponent
shared exp
x_0
x_1
...
x_{N-1}
Block Logorithm (BL)
1-bit sign, E-bit exponent
B-bit shared exp bias
shared exp bias
x_0
x_1
...
x_{N-1}

Non-linear Arithmetic

Method	Config	E	M	B
Fixed-point	W8A8	-	7	-
MiniFloat	W8A8	4	3	-
DMF	W8A8	4	3	-
BFP	W8A8	8	7	-
BFP	W6A6	8	5	-
BFP	W4A4	8	3	-
BM	W8A8	4	3	8
BL	W8A8	7	-	8

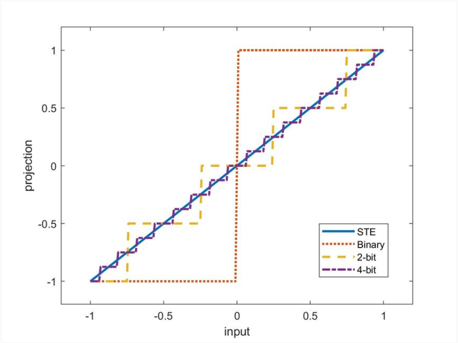
PTQ and QAT

PTQ (Post-training Quantization): normally zero-shot, directly quantized pre-trained network without finetuning.

QAT (Quantization-aware Training): quantize and then fine-tune the quantized network.

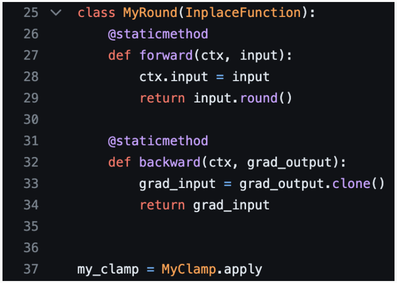
Straight-through Estimator for Quantization

Quantization is normally used in the forward pass, and QAT requires training it. However, functions such as ‘round’ is strictly non-differentiable!



Straight-through Estimator for Quantization

This means you have to design your own 'round' function with custom back propagation.



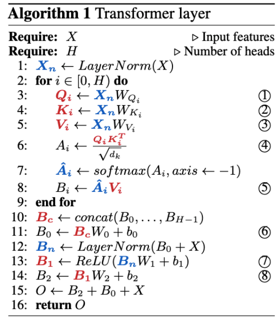
```
class MyRound(InplaceFunction):
    @staticmethod
    def forward(ctx, input):
        ctx.input = input
        return input.round()

    @staticmethod
    def backward(ctx, grad_output):
        grad_input = grad_output.clone()
        return grad_input


my_clamp = MyClamp.apply
```

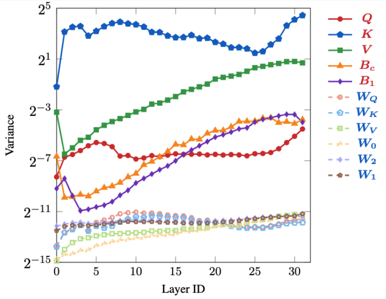
Mixed Precision Quantization

- 8 matrix multiplications in total
- Each multiplication has very different statistical property



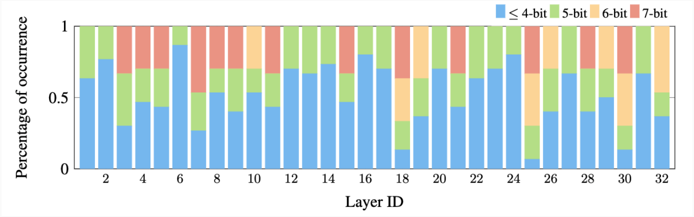
Algorithm 1 Transformer layer

```
Require: $X$                                  ▷ Input features
Require: $H$                                  ▷ Number of heads
 1: $X_n \leftarrow LayerNorm(X)$
 2: for $i \in [0, H)$ do
 3:     $Q_i \leftarrow X_n W_{Q_i}$                           ①
 4:     $K_i \leftarrow X_n W_{K_i}$                           ②
 5:     $V_i \leftarrow X_n W_{V_i}$                           ③
 6:     $A_i \leftarrow \frac{Q_i K_i^T}{\sqrt{d_k}}$                      ④
 7:     $\hat{A}_i \leftarrow softmax(A_i, axis \leftarrow -1)$
 8:     $B_i \leftarrow \hat{A}_i V_i$                            ⑤
 9: end for
10: $B_c \leftarrow concat(B_0, \ldots, B_{H-1})$
11: $B_0 \leftarrow B_c W_0 + b_0$                            ⑥
12: $B_n \leftarrow LayerNorm(B_0 + X)$
13: $B_1 \leftarrow ReLU(B_n W_1 + b_1)$                       ⑦
14: $B_2 \leftarrow B_1 W_2 + b_2$                            ⑧
15: $O \leftarrow B_2 + B_0 + X$
16: return $O$
```



Mixed Precision Search with Bayesian Search

- Optuna-based search
- Sampled from a very large search space, several layers prefer to have high-precision components.


≤ 4-bit
5-bit
6-bit
7-bit
Percentage of occurrence
Layer ID