Network Compression

Aaron Zhao, Imperial College London

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.

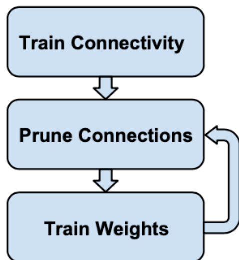
$$W_s = M \odot W$$

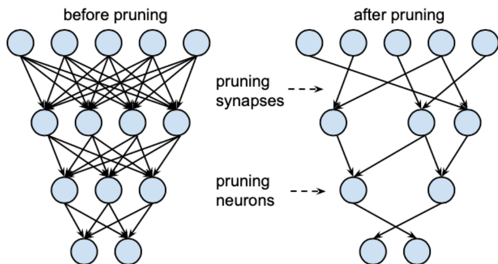
where $\odot$ represents an element-wise Hadamard product between two matrices.

Fine-grained Network Pruning

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

Network Pruning (ii)



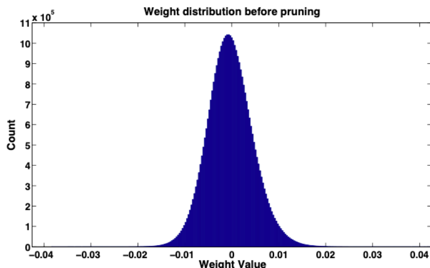


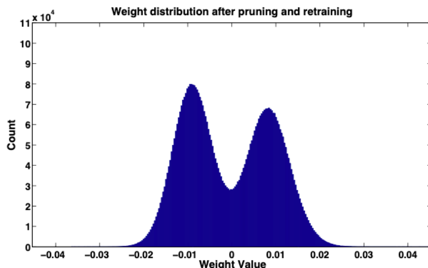
Using normalization terms (L1/L2) in the loss to encourage more sparsity.

$$\mathcal{L}' = \mathcal{L} + \lambda(\|\mathbf{w}\|_n)$$

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

Network Pruning (iii)





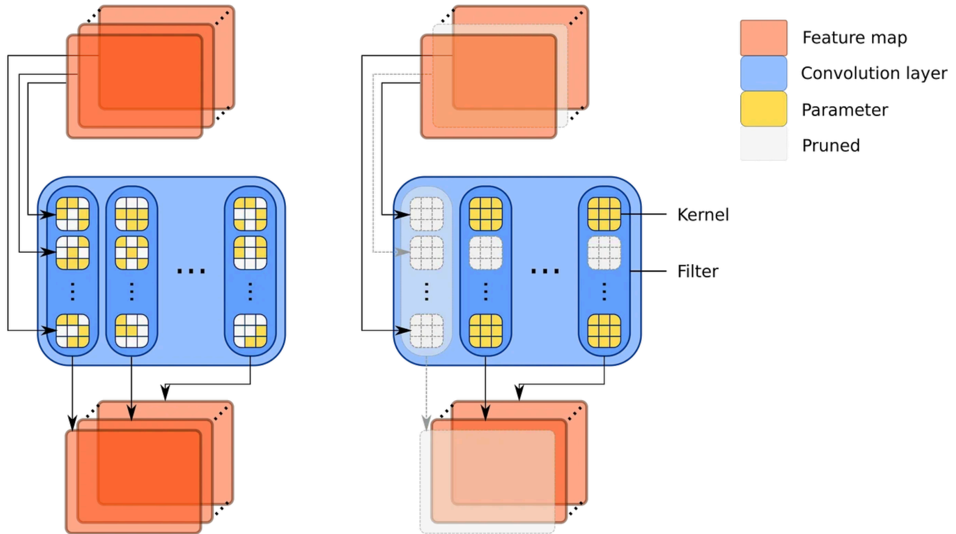
- Ignored the values that are actually zeroed out.
- Original weight distribution has a 'normal' shape

Network Pruning (iv)

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 Pruning



Coarse-grained Pruning (ii)

- 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

Channel 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:

$$s_w(i) = \| \, w[:, i, :, :] \, \|_p$$

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:

$$s_a(i) = \| \, a[i, :, :] \, \|_p$$

Channel pruning (ii)

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

$$s(i) = \alpha s_w(i) + \beta s_a(i)$$

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

$$y' = r \odot y$$

and a regularized loss

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

With a new scoring function

$$s(i) = r[i]$$

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:

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

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!

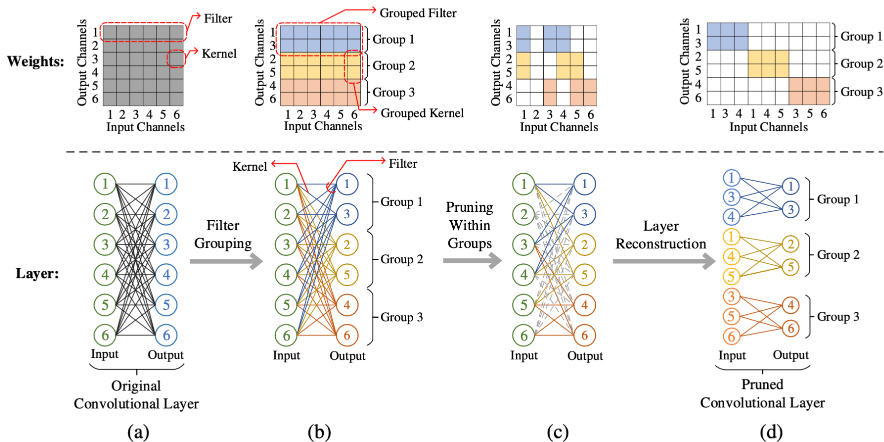
Kernel Pruning

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 (ii)



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

Kernel Pruning (iii)

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?

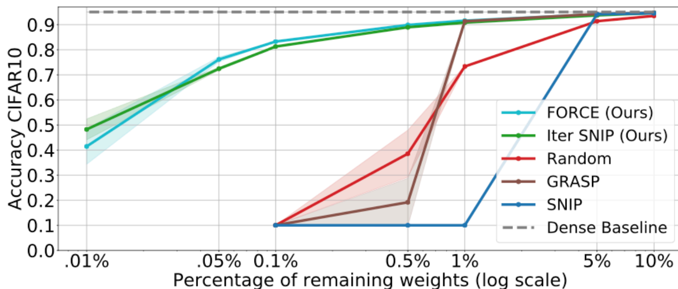
We can prune at initialization (sparse training).

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:

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

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 $M = 10$.

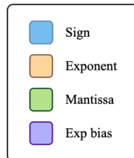
`MiniFloat: E,M` this allows custom exponent and mantissa widths.

Non-linear Arithmetic Various block-based arithmetic (BFP, BL)

Non-linear Arithmetic (ii)

IEEE Float32 (FP32)

1-bit sign, 8-bit exponent, 23-bit mantissa



IEEE Float16 (FP16)

1-bit sign, 5-bit exponent, 10-bit mantiisa

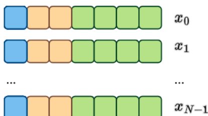
MiniFloat / Denormed Minifloat (DMF)

1-bit sign, 4-bit exponent, 3-bit mantiisa

Block Minifloat (BM)

1-bit sign, E-bit exponent, M-bit mantissa

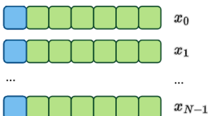
B-bit shared exponent bias



Block Floating Point (BFP)

1-bit sign, M-bit mantissa

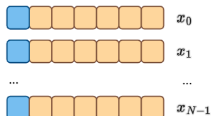
E-bit shared exponent



Block Logorithm (BL)

1-bit sign, E-bit exponent

B-bit shared exp bias



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!

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

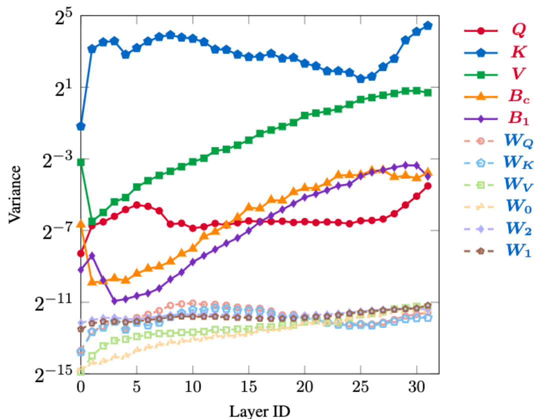
```
25  class MyRound(InplaceFunction):
26      @staticmethod
27      def forward(ctx, input):
28          ctx.input = input
29          return input.round()
30
31      @staticmethod
32      def backward(ctx, grad_output):
33          grad_input = grad_output.clone()
34          return grad_input
35
36
37  my_clamp = MyClamp.apply
```

Mixed precision quantization

- 8 matrix multiplications in total in an Attention layer
- 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: $\quad Q_i \leftarrow X_n W_{Q_i}$ ①
4: $\quad K_i \leftarrow X_n W_{K_i}$ ②
5: $\quad V_i \leftarrow X_n W_{V_i}$ ③
6: $\quad A_i \leftarrow \frac{Q_i K_i^T}{\sqrt{d_k}}$ ④
7: $\quad \hat{A}_i \leftarrow softmax(A_i, axis \leftarrow -1)$
8: $\quad B_i \leftarrow \hat{A}_i V_i$ 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 quantization (ii)

Mixed Precision Search with Bayesian Search

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