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 $f'_{\theta'}$?

Two main families of compression techniques:

- **Network Pruning** — remove redundant weights or structures
  - Fine-grained pruning (element-wise sparsity)
  - Coarse-grained pruning (channel and kernel removal)
  - Pruning at initialization (the lottery ticket hypothesis)
- **Quantization** — reduce the numerical precision of weights and activations
  - Different arithmetic schemes (fixed-point, floating-point, block-based)
  - Quantization-aware training and post-training quantization
  - Extremely low-precision formats (binary and ternary)

Pruning

Network Pruning: Fine-grained

Fine-grained pruning exploits element-wise sparsity: each weight entry has the opportunity to be zeroed out independently.

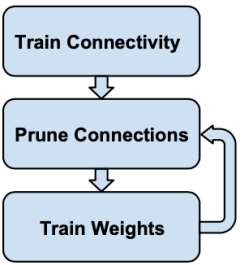
$$W_s = M \odot W$$

where $\odot$ is the element-wise Hadamard product and M is a binary mask.

Key properties:

- Sparsity can be applied to both weights and activations
- Irregular sparsity patterns are hard to exploit on standard hardware
- Re-training after pruning recovers most of the lost accuracy

Network Pruning: Fine-grained (ii)



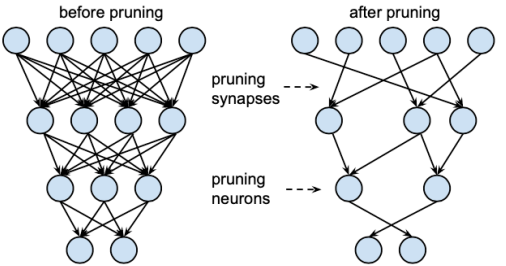


Figure 1: Illustration of fine-grained (element-wise) pruning

Fine-grained Pruning: Encouraging Sparsity

A regularization term is added to the training loss to encourage sparse weights:

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

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

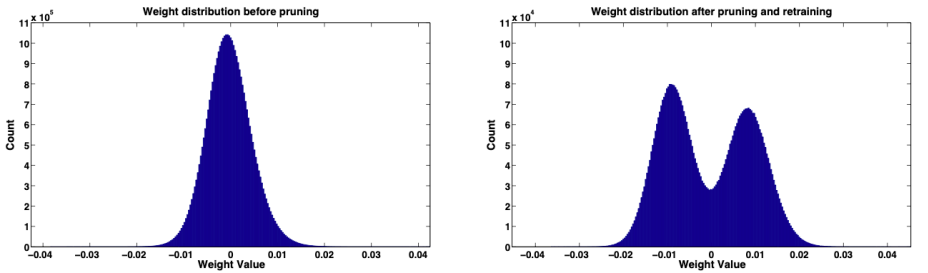


Figure 2: Weight distribution before and after pruning. Pruned weights cluster at zero.

Fine-grained Pruning: Encouraging Sparsity (ii)

- Zeroed-out weights are excluded from statistics
- The original weight distribution has a roughly Gaussian shape

Fine-grained Pruning: Results

Table 1: Fine-grained pruning results with iterative re-training

Network + Dataset	Density	Compression
VGG7 + CIFAR-10	16%	6×
AlexNet + ImageNet	11%	9×
VGG16 + ImageNet	7.5%	12×

- All networks achieve less than 0.1% accuracy drop after pruning with iterative re-training.
- Pruning is most effective for large networks on relatively simple tasks.
- Fine-grained sparsity does **not** always translate to wall-clock speedup on standard hardware.

Coarse-grained Pruning

Coarse-grained pruning removes structured groups of weights, producing hardware-friendly sparse models.

Coarse-grained Pruning (ii)

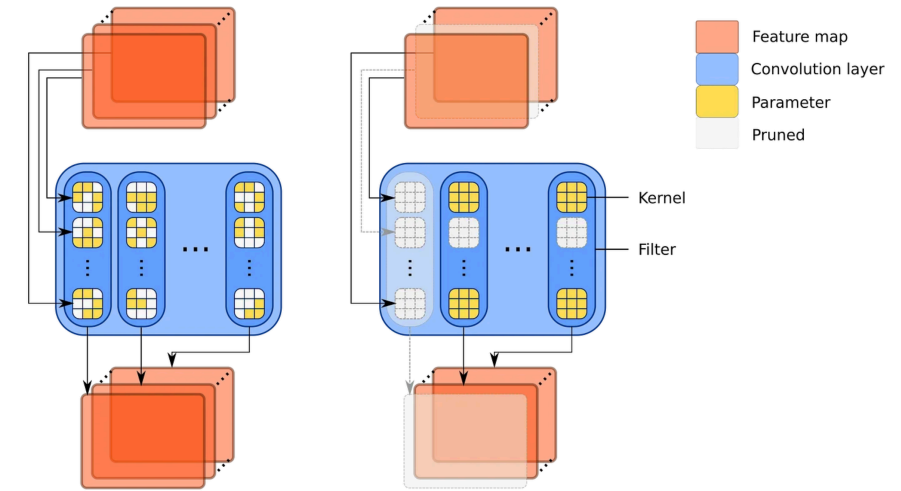


Figure 3: Coarse-grained pruning strategies on a $C_i \times C_o \times K \times K$ weight tensor

Coarse-grained Pruning (iii)

- **Channel pruning** — remove entire input (C_i) or output (C_o) channels
- **Kernel pruning** — remove entire $K \times K$ kernels from the weight volume

Structured removal directly reduces tensor dimensions, enabling efficient dense computation on standard hardware.

Channel Pruning: Scoring Functions

Channel pruning requires ranking channels by importance. Two common signals:

Weight-based score — given $w \in \mathcal{R}^{C_i \times C_o \times K \times K}$:

$$s_w(i) = \parallel w[:, i, :, :] \parallel_p$$

Activation-based score — given output activation $a \in \mathcal{R}^{C_o \times K \times K}$:

$$s_a(i) = \parallel a[i, :, :] \parallel_p$$

Both signals can be combined with two free hyperparameters α and β:

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

The p-norm choice (e.g. l_1 or l_2) controls sensitivity to outliers.

Channel Pruning: Network Slimming

Network Slimming learns channel importance automatically via a trainable scaling vector $r \in \mathcal{R}^{C_o}$:

$$y' = r \odot y$$

The loss is augmented with an l_p sparsity penalty on r:

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

After training, the channel importance score is simply:

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

Channels with small $|r[i]|$ are pruned. SGD drives unimportant channels towards zero automatically.

Channel Pruning: Connecting to Batch Normalization

Batch Normalization (BN) is standard in modern CNNs for fast convergence and better generalization. It is inserted after convolutional layers:

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

where γ and β are learnable parameters, and μ, σ are computed from the running statistics.

Key insight: The BN scale parameter γ plays exactly the same role as the Network Slimming scaling vector r. In practice, γ values are **directly reused** as the channel importance scores, requiring no additional parameters.

Kernel Pruning

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

Challenge: Naively removing arbitrary kernels produces irregular computation patterns that cannot be efficiently accelerated.

Solution: Remove kernels in groups so that the remaining kernels form equal-sized blocks, equivalent to grouped convolution.

Kernel Pruning (ii)

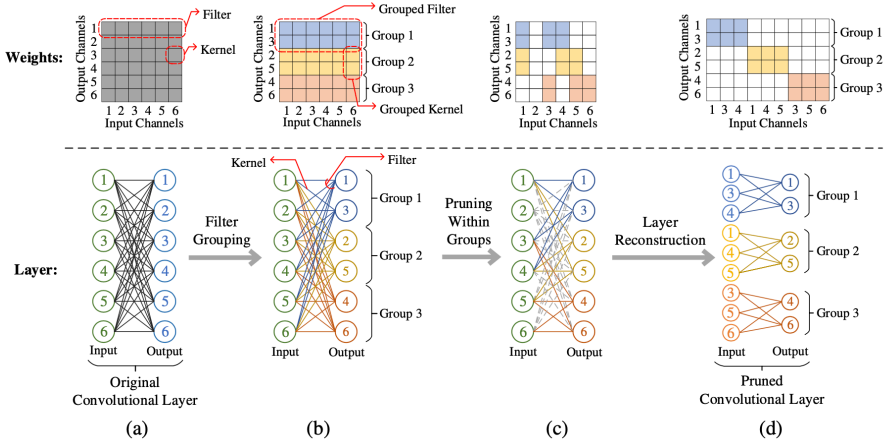


Figure 4: Kernel pruning with equal-sized groups yields regular computation

Kernel Pruning (iii)

After reshaping, the original $C_i = 6, C_o = 6$ convolution (36 kernels) becomes 3 groups of $C_i' = 3, C_o' = 2$ convolutions ($3 \times 3 \times 2 = 18$ kernels), achieving a $2\times$ reduction in FLOPs.

Pruning at Initialization

So far we have pruned **trained** networks. Can we identify the right subnetwork **before** training?

The Lottery Ticket Hypothesis (Frankle & Carlin, 2018):

> 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.

Implication: There exists a sparse mask that, applied at initialization, yields a competitive model — we just need to find it.

Finding Winning Tickets

Metrics for identifying winning tickets at initialization:

- **Weight magnitude** — proxy for importance after training
- **Gradient norm** — sensitivity of the loss to each weight
- **SNIP** — saliency based on the Hessian of the loss w.r.t. weights at initialization

Iterative variants (iterative SNIP, FORCE): allow pruned parameters to **resurrect** at later stages, enabling better exploration of the sparse subnetwork landscape.

Finding Winning Tickets (ii)

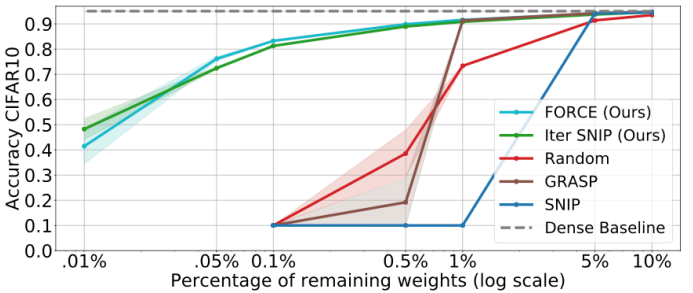


Figure 5: Comparison of pruning-at-initialization methods

Quantization

Network Quantization

Quantization represents weights (and activations) with much narrower bit-widths than the standard 32-bit floating-point, drastically reducing computation and memory.

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

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

Fixed-point arithmetic is linear — multiplication of two n-bit numbers produces a $2n$-bit result, requiring truncation back to n bits.

Common notation: **WxAy** means x-bit weights and y-bit activations (e.g. W8A8 = 8-bit weights and activations).

Non-linear Arithmetic: Floating-Point

Standard IEEE floating-point is defined by a 4-tuple (s, e, m, b):

- $s \in \{0, 1\}$ — sign bit
- $e \in \mathbb{N}$ — exponent field (width E bits)
- $b \in \mathbb{N}$ — exponent bias
- $m \in \mathbb{N}$ — mantissa (width M bits)

Common formats:

- `float32` (FP32): $E = 8$, $M = 23$
- `float16` (FP16): $E = 5$, $M = 10$
- **MiniFloat** (E, M): custom widths for hardware efficiency

Floating-point provides **higher dynamic range** than fixed-point but involves non-linear spacing between representable values.

Non-linear Arithmetic: Block-based Formats

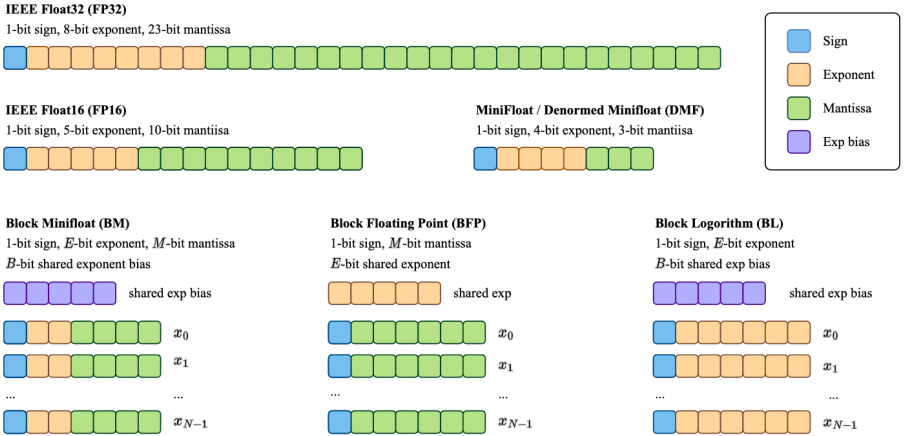


Figure 6: Comparison of fixed-point, floating-point, and block-based arithmetic formats

Non-linear Arithmetic: Block-based Formats (ii)

Block-based formats share a common exponent across a group of values, combining the dynamic range of floating-point with the simplicity of fixed-point within each block.

Arithmetic Format Comparison

Table 2: Arithmetic format configurations: E = exponent bits, M = mantissa bits, B = block size

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

Two main paradigms for applying quantization to a pre-trained model:

Post-training Quantization (PTQ)

- Applied directly to a pre-trained model with **no** or minimal fine-tuning
- Typically uses a small calibration dataset to choose quantization parameters
- Fast and data-efficient, but accuracy may degrade at very low bit-widths

Quantization-aware Training (QAT)

- Simulates quantization **during** fine-tuning so the model learns to compensate
- Generally achieves better accuracy than PTQ, especially at W4A4 or lower
- Requires access to training data and additional compute

Straight-through Estimator

Quantization (e.g. rounding) is used in the **forward pass** of QAT, but the rounding function has zero gradient almost everywhere — making standard backpropagation impossible.

Straight-through Estimator (STE): define a custom backward pass that treats the rounding operation as the identity:

$$\frac{\partial \mathcal{L}}{\partial x} \approx \frac{\partial \mathcal{L}}{\partial \hat{x}}$$

This "passes the gradient straight through" the non-differentiable quantizer.

Straight-through Estimator (ii)

```
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
```

Figure 7: STE approximates the gradient of the rounding function (red) with the identity (blue)

Mixed Precision Quantization

Different layers and operations have very different sensitivity to quantization. **Mixed precision** assigns different bit-widths to different components.

Example: Attention layer

- 8 matrix multiplications per attention layer
- Each multiplication has a different statistical distribution of values

Mixed Precision Quantization (ii)

Algorithm 1 Transformer layer

Require: X ▷ Input features
Require: H ▷ Number of heads

$\boldsymbol{X_n} \leftarrow LayerNorm(X)$
for $i \in [0, H)$ **do**
  $\boldsymbol{Q_i} \leftarrow \boldsymbol{X_n} W_{Q_i}$ ①
  $\boldsymbol{K_i} \leftarrow \boldsymbol{X_n} W_{K_i}$ ②
  $\boldsymbol{V_i} \leftarrow \boldsymbol{X_n} W_{V_i}$ ③
  $A_i \leftarrow \frac{\boldsymbol{Q_i} \boldsymbol{K_i}^T}{\sqrt{d_k}}$ ④
  $\hat{\boldsymbol{A}}_i \leftarrow softmax(A_i, axis \leftarrow -1)$
  $B_i \leftarrow \hat{\boldsymbol{A}}_i \boldsymbol{V_i}$ ⑤
end for
$\boldsymbol{B_c} \leftarrow concat(B_0, \ldots, B_{H-1})$
$B_0 \leftarrow \boldsymbol{B_c} W_0 + b_0$ ⑥
$\boldsymbol{B_n} \leftarrow LayerNorm(B_0 + X)$
$\boldsymbol{B_1} \leftarrow ReLU(\boldsymbol{B_n} W_1 + b_1)$ ⑦
$B_2 \leftarrow \boldsymbol{B_1} W_2 + b_2$ ⑧
$O \leftarrow B_2 + B_0 + X$
return O

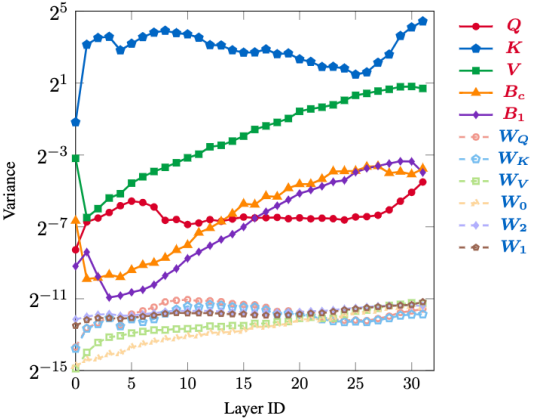


Figure 8: Per-layer quantization sensitivity in an Attention layer

Mixed Precision Search

Bayesian search (Optuna-based):

- Search over a very large configuration space (bit-width per layer/operation)
- Sampled results show that several layers strongly prefer high-precision components
- Fitness combines accuracy and hardware efficiency (latency, memory)

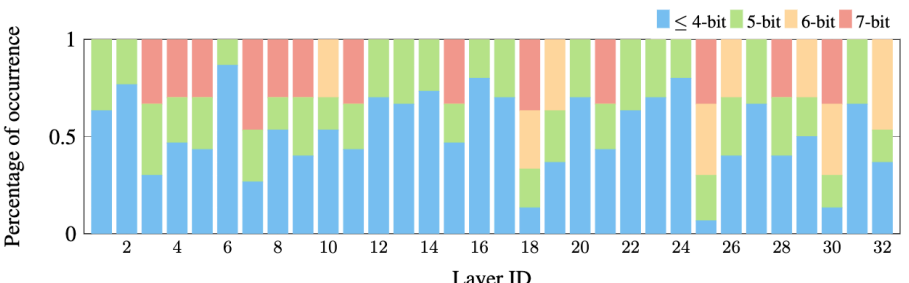


Figure 9: Mixed precision search results: each bar represents the chosen bit-width per operation

Summary

Network Pruning:

- **Fine-grained** — element-wise sparsity; high compression but irregular patterns
- **Coarse-grained** — channel/kernel removal; hardware-friendly; scoring via weights, activations, or BN γ
- **Pruning at initialization** — lottery ticket hypothesis; find winning tickets before training

Quantization:

- **Fixed-point** — linear, simple, efficient on hardware
- **Floating-point / MiniFloat** — higher dynamic range, non-linear spacing
- **Block-based (BFP, BM, BL)** — shared exponent per block
- **PTQ** — zero-shot, fast; **QAT** — fine-tune aware, better accuracy
- **STE** — enables gradient flow through non-differentiable quantizers
- **Mixed precision** — per-layer bit-width optimized via search

Summary (ii)

Key takeaway: Pruning and quantization are complementary; combining both often yields the best compression-accuracy trade-off.