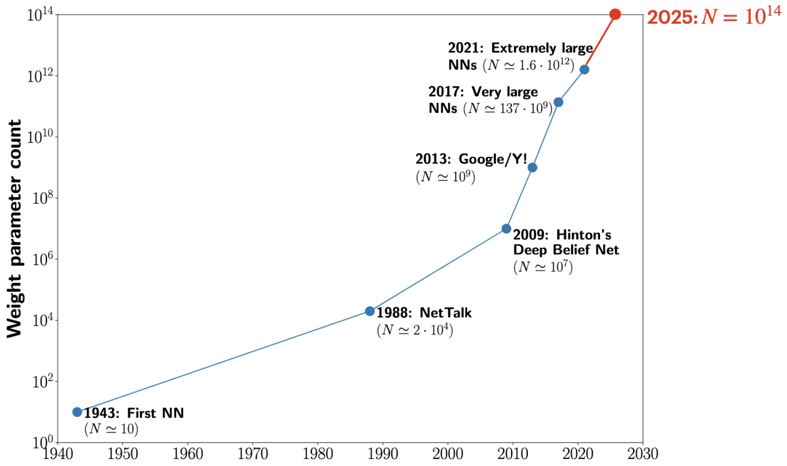
Neural Networks compression

Elisa Riccietti and Theo Mary

January 11, 2026

Context: deep neural networks (DNN) are growing fast

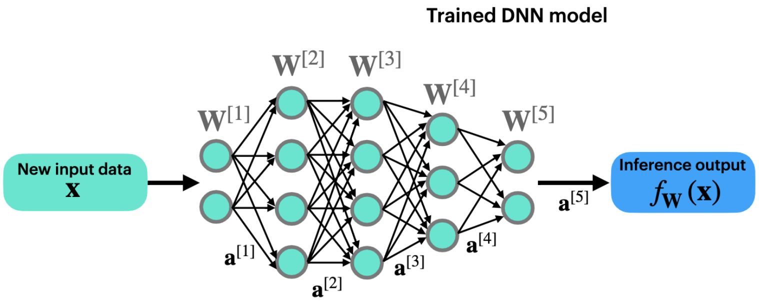


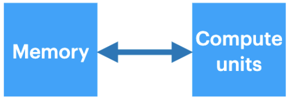
Scaling up

Why? : better performance, more complex tasks ...
... but

- The **computational complexity, memory usage and energy consumption** of deep networks are increasingly growing, both during training and inference
- Limitation for the deployment on resource-constrained devices (mobile, embedded systems) where energy is often a limited resource
- High environmental impact
- Limitation for real-time applications

The computational bottlenecks





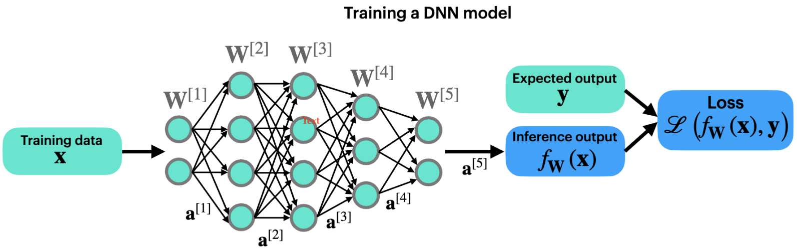
Data movement

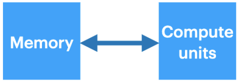
- move input data & model from memory to compute units
- send partial results back to memory

Computations

- vector/matrix manipulations
- done on CPU, GPU, DSP, or custom accelerators (e.g., FPGA, ASIC)

The computational bottlenecks

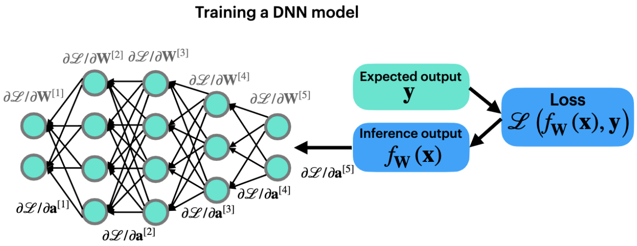


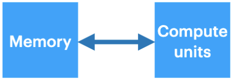


Data movement
- move input data & model from memory to compute units
- send partial results back to memory

Computations
- vector/matrix manipulations
- done on CPU, GPU, DSP, or custom accelerators (e.g., FPGA, ASIC)

The computational bottlenecks

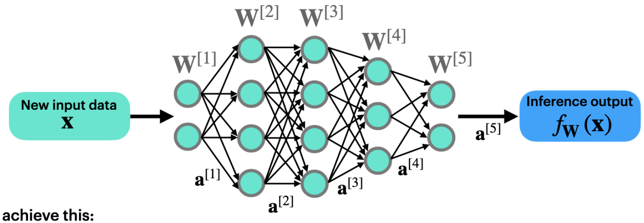

Training a DNN model
$\partial\mathcal{L}/\partial\mathbf{W}^{[1]}$
$\partial\mathcal{L}/\partial\mathbf{W}^{[2]}$
$\partial\mathcal{L}/\partial\mathbf{W}^{[3]}$
$\partial\mathcal{L}/\partial\mathbf{W}^{[4]}$
$\partial\mathcal{L}/\partial\mathbf{W}^{[5]}$
$\partial\mathcal{L}/\partial\mathbf{a}^{[1]}$
$\partial\mathcal{L}/\partial\mathbf{a}^{[2]}$
$\partial\mathcal{L}/\partial\mathbf{a}^{[3]}$
$\partial\mathcal{L}/\partial\mathbf{a}^{[4]}$
$\partial\mathcal{L}/\partial\mathbf{a}^{[5]}$
Expected output $\mathbf{y}$
Inference output $f_{\mathbf{W}}(\mathbf{x})$
Loss $\mathcal{L}\left(f_{\mathbf{W}}(\mathbf{x}),\mathbf{y}\right)$



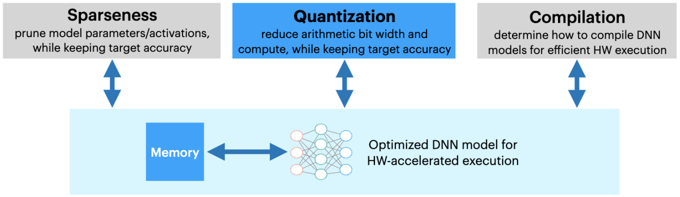
Data movement
- move input data & model from memory to compute units
- send partial results back to memory

Computations
- vector/matrix manipulations
- done on CPU, GPU, DSP, or custom accelerators (e.g., FPGA, ASIC)

Improve performance and power efficiency



Various ways to achieve this:



Sparseness
prune model parameters/activations, while keeping target accuracy

Quantization
reduce arithmetic bit width and compute, while keeping target accuracy

Compilation
determine how to compile DNN models for efficient HW execution

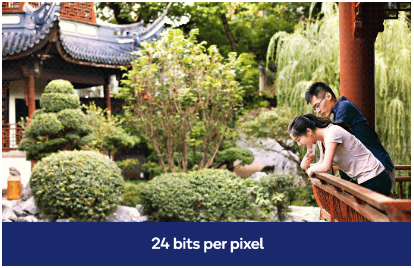
Outline

Outline

Quantization: definition

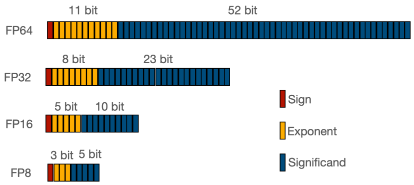
Quantization is the process of constraining an input from a continuous or large set of values (such as the real numbers) to a discrete set (such as the integers).
Example: convert floating-point numbers to lower precision (e.g., 8-bit integers).

Example: `https://www.qualcomm.com/news/onq/2019/03/heres-why-quantization-matters-ai`



Low precision formats

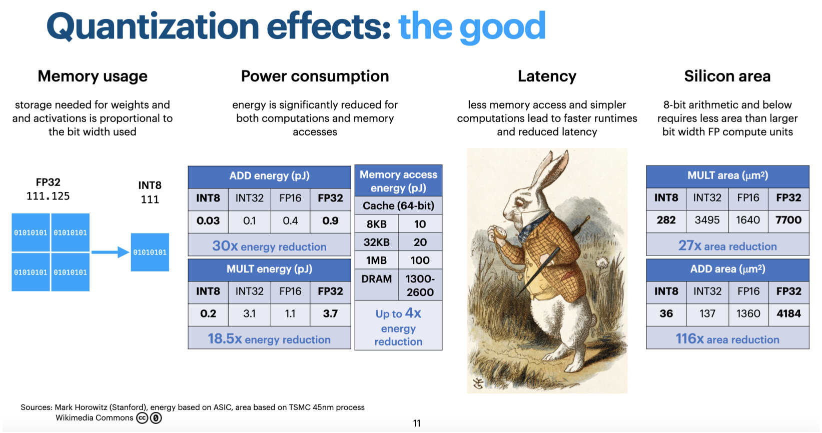
		number of bits			
		signif. (t)	exp.	range	$u = 2^{-t}$
fp128	quadruple	113	15	$10^{\pm 4932}$	1×10^{-34}
fp64	double	53	11	$10^{\pm 308}$	1×10^{-16}
fp32	single	24	8	$10^{\pm 38}$	6×10^{-8}
fp16	half	11	5	$10^{\pm 5}$	5×10^{-4}
bfloat16		8	8	$10^{\pm 38}$	4×10^{-3}
fp8 (e4m3)	quarter	4	4	$10^{\pm 2}$	6×10^{-2}
fp8 (e5m2)		3	5	$10^{\pm 5}$	1×10^{-1}



Modern hardware

- Nvidia tensor cores
- Google TPUs
- FPGAs

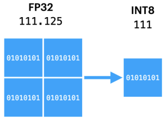
Benefits of quantization



Quantization effects: the good

Memory usage

storage needed for weights and and activations is proportional to the bit width used



Power consumption

energy is significantly reduced for both computations and memory accesses

ADD energy (pJ)			
INT8	INT32	FP16	**FP32**
0.03	0.1	0.4	**0.9**
30x energy reduction			

MULT energy (pJ)			
INT8	INT32	FP16	**FP32**
0.2	3.1	1.1	**3.7**
18.5x energy reduction			

Memory access energy (pJ)	
Cache (64-bit)	
8KB	10
32KB	20
1MB	100
DRAM	1300-2600
Up to 4x energy reduction	

Latency

less memory access and simpler computations lead to faster runtimes and reduced latency

Silicon area

8-bit arithmetic and below requires less area than larger bit width FP compute units

MULT area (μm²)			
INT8	INT32	FP16	**FP32**
282	3495	1640	**7700**
27x area reduction			

ADD area (μm²)			
INT8	INT32	FP16	**FP32**
36	137	1360	**4184**
116x area reduction			

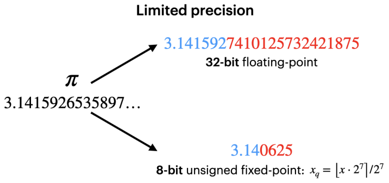
Sources: Mark Horowitz (Stanford), energy based on ASIC, area based on TSMC 45nm process
Wikimedia Commons

11

Interesting formats

- 8-bit Integer (INT8)
  - INT8 quantization is widely used for model deployment in edge devices and mobile platforms.
  - Intel's Neural Network Processor (NNP) and Google's Edge TPUs provide dedicated support for INT8 inference, reducing energy consumption by up to 4x compared to FP32
- 4-bit quantization
  - is an emerging trend, focused on ultra-low-power AI applications.
  - 4-bit quantization is still an area of active research, focusing on improving the trade-off between energy savings and model accuracy.

Quantization: the bad



low precision = low accuracy

Challenge: reduce precision without harming accuracy

Mixed Precision Quantization

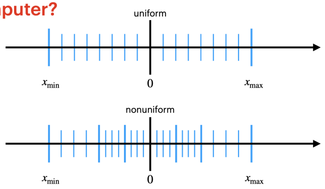
- Mixed precision quantization involves applying different bitwidths to different parts of the model.
- Example: Weights can be quantized to 8 bits while activations remain at 16 bits.
- This helps to balance model accuracy with computational efficiency.
- Particularly useful if not all parameters are equally important for model expressivity

Quantization formats

How do we represent reals on a computer?

➡**two** main families of formats:

- integer/fixed-point (**uniform** quantization)
- floating-point (**nonuniform** quantization)



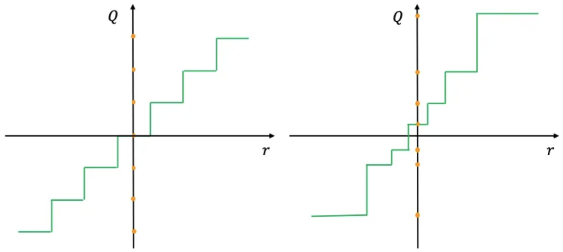


Figure 1

How to quantize?

Limited (dynamic) range

➡ value magnitude range

FP32: $\left(10^{-38}, 10^{38}\right)$

~ 76 orders of magnitude

FP8: $\left(10^{-2}, 10^{2}\right)$

~ 4 orders of magnitude

How to quantize?

Want to quantize X to int8

$$X = \begin{bmatrix} 0.97 & 0.64 & 0.74 & 1.00 \\ 0.58 & 0.84 & 0.84 & 0.81 \\ 0.00 & 0.18 & 0.90 & 0.28 \\ 0.57 & 0.96 & 0.80 & 0.81 \end{bmatrix}$$

We don't directly quantize it, but rather a scaled version to avoid overflows.
Example: for int8 range$=[-128, 127] = [\text{min}_{int8}, \text{max}_{int8}]$

How to quantize?

$$X = \begin{bmatrix} 0.97 & 0.64 & 0.74 & 1.00 \\ 0.58 & 0.84 & 0.84 & 0.81 \\ 0.00 & 0.18 & 0.90 & 0.28 \\ 0.57 & 0.96 & 0.80 & 0.81 \end{bmatrix}$$

Compute scaling factor

$$s = \frac{\max_{int8} - \min_{int8}}{x_{\max} - x_{\min}} = \frac{127 + 128}{x_{\max} - x_{\min}} = 255$$

Scale

$$X_s = s(X - x_{\min}) + \min_{int8}$$

Quantize

$$\text{round}(X_s)$$

Clip the value

$$X \approx \text{clip}(\text{round}(X_s))$$

Outside the representable range: clip function

The basic idea of the clip function is to restrict values to a predefined range (min_val, max_val).
The `clip` function is defined as:

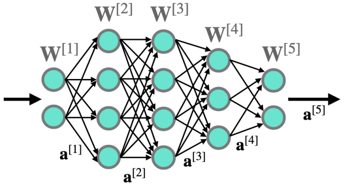
$$\text{clip}(x, \text{min\_val}, \text{max\_val}) = \begin{cases} \text{min\_val} & \text{if } x < \text{min\_val} \\ x & \text{if min\_val} \leq x \leq \text{max\_val} \\ \text{max\_val} & \text{if } x > \text{max\_val} \end{cases}$$

For example:

$$\begin{aligned} \text{clip}(5, 0, 10) &= 5 \quad \text{(within the range)} \\ \text{clip}(-3, 0, 10) &= 0 \quad \text{(clipped to min\_val)} \\ \text{clip}(15, 0, 10) &= 10 \quad \text{(clipped to max\_val)} \end{aligned}$$

Quantization for DNNs

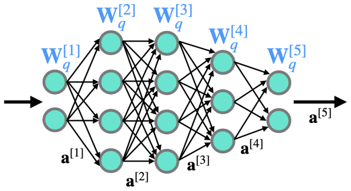
During inference (i.e., for a trained network):



Quantization for DNNs

During inference (i.e., for a trained network):

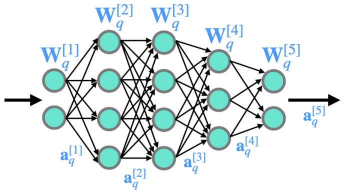
- store network parameters in low precision



Quantization for DNNs

During inference (i.e., for a trained network):

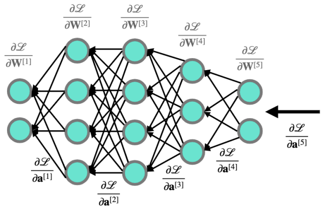
- store network parameters in low precision
- store/compute intermediate signals in low precision



Quantization for DNNs

During inference (i.e., for a trained network):

- store network parameters in low precision
- store/compute intermediate signals in low precision



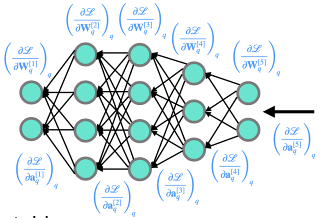
$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[1]}}$
$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[2]}}$
$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[3]}}$
$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[4]}}$
$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[5]}}$
$\frac{\partial \mathscr{L}}{\partial \mathbf{a}^{[5]}}$
$\frac{\partial \mathscr{L}}{\partial \mathbf{a}^{[4]}}$
$\frac{\partial \mathscr{L}}{\partial \mathbf{a}^{[3]}}$
$\frac{\partial \mathscr{L}}{\partial \mathbf{a}^{[2]}}$
$\frac{\partial \mathscr{L}}{\partial \mathbf{a}^{[1]}}$

During training:

Quantization for DNNs

During inference (i.e., for a trained network):

- store network parameters in low precision
- store/compute intermediate signals in low precision



$\left(\frac{\partial \mathscr{L}}{\partial \mathbf{W}_q^{[1]}}\right)_q$
$\left(\frac{\partial \mathscr{L}}{\partial \mathbf{W}_q^{[2]}}\right)_q$
$\left(\frac{\partial \mathscr{L}}{\partial \mathbf{W}_q^{[3]}}\right)_q$
$\left(\frac{\partial \mathscr{L}}{\partial \mathbf{W}_q^{[4]}}\right)_q$
$\left(\frac{\partial \mathscr{L}}{\partial \mathbf{W}_q^{[5]}}\right)_q$
$\left(\frac{\partial \mathscr{L}}{\partial \mathbf{a}_q^{[1]}}\right)_q$
$\left(\frac{\partial \mathscr{L}}{\partial \mathbf{a}_q^{[2]}}\right)_q$
$\left(\frac{\partial \mathscr{L}}{\partial \mathbf{a}_q^{[3]}}\right)_q$
$\left(\frac{\partial \mathscr{L}}{\partial \mathbf{a}_q^{[4]}}\right)_q$
$\left(\frac{\partial \mathscr{L}}{\partial \mathbf{a}_q^{[5]}}\right)_q$

During training:

- store/compute back propagated gradients in low precision

Types of rounding

- **Traditional Rounding:** Typically rounds a number to the nearest integer or a nearby fixed point.
  - **Round to Nearest:** Rounds 1.5 to 2 and 2.4 to 2.
  - **Round Down:** Always rounds down (e.g., 2.8 becomes 2).
  - **Round Up:** Always rounds up (e.g., 2.1 becomes 3).
- **Stochastic Rounding:** Rounds based on probability, depending on the fractional part of the number.
  - For example, if a number is 1.7, it might round to 1 with probability 0.3 and to 2 with probability 0.7.
- **Key Difference:** Stochastic rounding introduces randomness, whereas traditional rounding methods are deterministic.

"Stochastic Rounding: Implementation, Error Analysis, and Applications" M. Croci, M. Fasi, N. Higham, T. Mary, M. Mikaitis, 2021

Why Stochastic Rounding?

$$\mathbb{E}(\hat{x}) = x$$

- In standard rounding methods, bias can accumulate, leading to systematic errors in computations.
- Stochastic rounding introduces randomness, which helps to:
  - **Reduce bias:** Prevents systematic overestimation or underestimation of values. This randomness in rounding ensures that the rounding error has an expected value of zero, leading to less bias.
  - **Preserve statistical properties:** Maintains the expected value over a series of rounding operations.
  - **Improve model accuracy:** In neural networks, reduces the impact of rounding errors on training and inference, especially when working with low-precision formats, where small errors can propagate and magnify over multiple layers.

How Stochastic Rounding Works

- Let $x = n + f$, where n is the integer part and f is the fractional part.
- In stochastic rounding:

$$\text{Round}(x) = \begin{cases} n \text{ with probability } 1 - f \\ n + 1 \text{ with probability } f \end{cases}$$

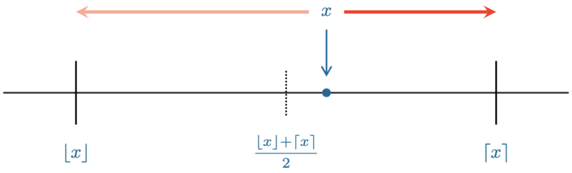


Figure 2.1. Stochastic rounding rounds the real number x to the next smaller number $\lfloor x \rfloor$ in F or to the next larger number $\lceil x \rceil$ in F. In this example, RN rounds x to $\lceil x \rceil$, whereas mode 1 SR can round to either $\lfloor x \rfloor$ or $\lceil x \rceil$ but is more likely to round to $\lceil x \rceil$.

Stochastic Rounding (SR) in Neural Networks

Not a new idea:

Höhfeld M, Fahlman SE. 1992 "Probabilistic rounding in neural network learning with limited precision"

- Useful in NN, especially in low-precision formats like INT8 or FP16.
- In low-precision arithmetic, rounding errors can significantly impact model performance due to the limited number of bits
- SR can outperform traditional rounding methods in certain quantized neural networks:

  Gupta S, Agrawal A, Gopalakrishnan K, Narayanan P. 2015 "Deep Learning with Limited Numerical Precision"

Limitations of RN for low precision

"On Stochastic Roundoff Errors in Gradient Descent with Low- Precision Computation" Xia, L., Massei, S., Hochstenbach, M. E., Koren, B. (2024).

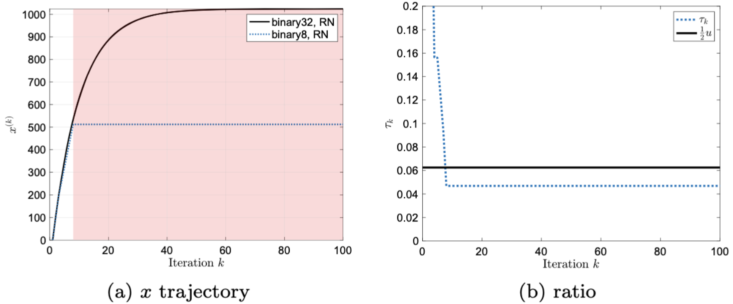
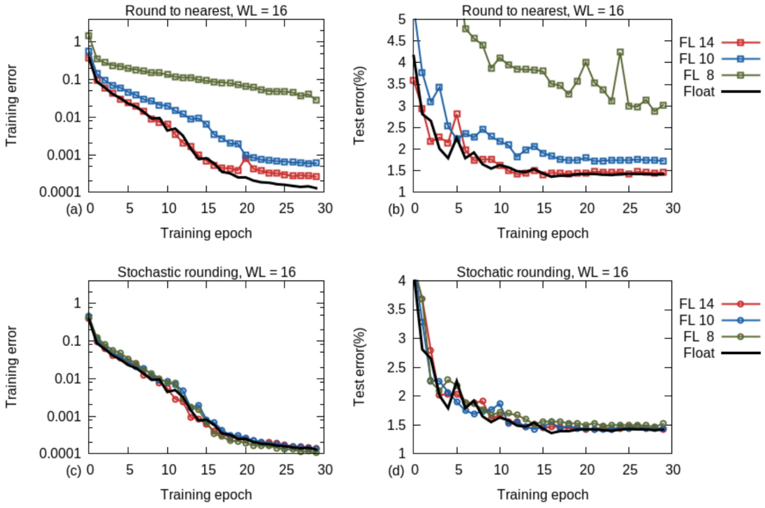


Fig. 2 Minimizing $f(x) = (x - 1024)^2$ using GD with binary8 ($u = 2^{-3}$) and RN, where the red area indicates where stagnation occurs

SR for neural networks

Gupta S, Agrawal A, Gopalakrishnan K, Narayanan P. 2015 "Deep Learning with Limited Numerical Precision"



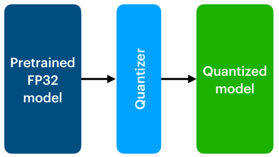
Challenges and Limitations

- **Hardware support:** Not all hardware accelerators natively support stochastic rounding, requiring custom implementations.
- **Higher complexity:** The randomness involved in stochastic rounding can make it more difficult to analyze and debug models.

Quantization-Based Methods for Efficient DNN Inference

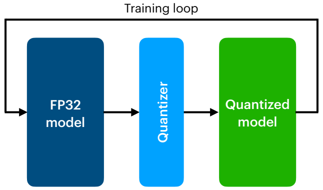
Quantization for efficient DNN inference

Post Training Quantization (PTQ)



- + no need for access to training pipeline
- + data-free or small calibration set used
- + usually fast, with simple API
- - lower accuracy at lower bit widths

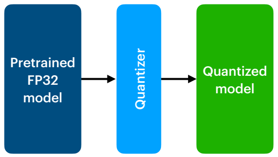
Quantization-Aware Training (QAT)



- - access to training pipeline & labelled data
- - longer training times
- - hyper-parameter tuning needed
- + higher accuracy in general

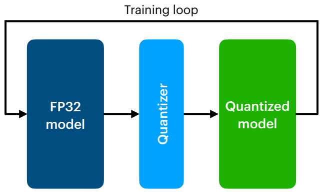
Quantization for efficient DNN inference

Post Training Quantization (PTQ)



- + no need for access to training pipeline
- + data-free or small calibration set used
- + usually fast, with simple API
- - lower accuracy at lower bit widths

Quantization-Aware Training (QAT)

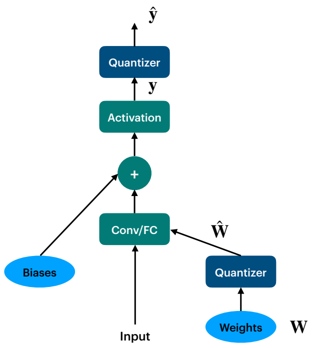


- - access to training pipeline & labelled data
- - longer training times
- - hyper-parameter tuning needed
- + higher accuracy in general

Training with Quantization

- **Quantization-Aware Training (QAT)**: A technique where quantization is simulated during training.
- The model is trained with quantized weights and/or activations to maintain accuracy.
- During training, the gradients are computed as if the model were full precision.

QAT: backward path quantization simulation

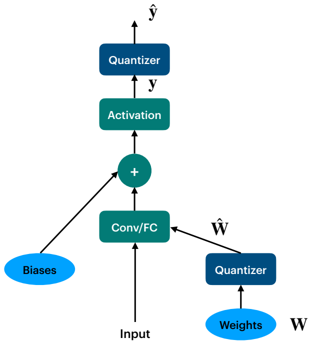


Problem: How can we back propagate through quantization layers?

➡round-to-nearest does not have meaningful gradients (i.e., either zero or undefined everywhere)

➡gradient-based training seems impossible

QAT: backward path quantization simulation



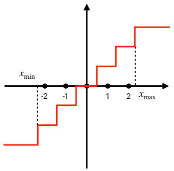
Problem: How can we back propagate through quantization layers?

➡round-to-nearest does not have meaningful gradients (i.e., either zero or undefined everywhere)

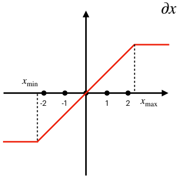
➡gradient-based training seems impossible

Solution: redefine gradient with “straight-through estimator” (STE) [1]

$$\frac{\partial \lfloor x \rceil}{\partial x} = 1$$



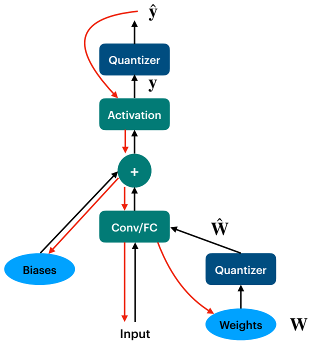
real forward pass



simulated forward pass

[1] Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation, *Bengio et al.*, arXiv:1308.3432, 2013

QAT: backward path quantization simulation



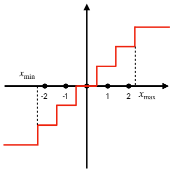
Problem: How can we back propagate through quantization layers?

➡round-to-nearest does not have meaningful gradients (i.e., either zero or undefined everywhere)

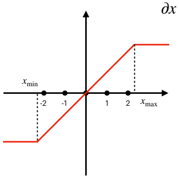
➡gradient-based training seems impossible

Solution: redefine gradient with "straight-through estimator" (STE) [1]

$$\frac{\partial \lfloor x \rceil}{\partial x} = 1$$



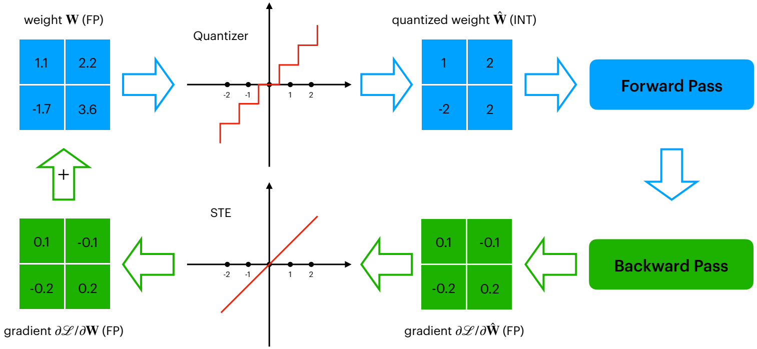
real forward pass



simulated forward pass

[1] Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation, *Bengio et al.*, arXiv:1308.3432, 2013

QAT: backward path quantization simulation



weight $\mathbf{W}$ (FP)
1.1
2.2
-1.7
3.6
Quantizer
-2
-1
1
2
quantized weight $\hat{\mathbf{W}}$ (INT)
1
2
-2
2
Forward Pass
+
STE
-2
-1
1
2
0.1
-0.1
-0.2
0.2
Backward Pass
gradient $\partial\mathcal{L}/\partial\mathbf{W}$ (FP)
gradient $\partial\mathcal{L}/\partial\hat{\mathbf{W}}$ (FP)

Schematic view of a QAT procedure with STE applied (adapted from [1])

[1] A Survey of Quantization Methods for Efficient Neural Network Inference, *Gholami et al.*, arXiv:2103.13630, 2021

Early success of QAT: BinaryConnect

["BinaryConnect: Training Deep Neural Networks with binary weights during propagations", M. Courbariaux, Y. Bengio, JP David, 2016]

- BinaryConnect is a method for training neural networks with binary weights and activations.
- During training, the weights are binarized (approximated to $+1$ or -1).
- The activations can also be binarized, reducing the computational cost of both forward and backward passes.

How BinaryConnect Works

- Keeps double copy of the weights: binary weights W_{bin} and continuous weights W
- Train the model using real-valued weights (for gradient computation), but at each step, constrain the weights to binary values after the weight update step.
- Reduce memory and computation for inference, while still benefiting from continuous gradients during training (essential for SGD to work).

How BinaryConnect Works

- During forward pass:
  - We compute the activations using binary weights
- During backpropagation:
  - The gradients are computed as if the network were using real-valued weights.
  - The continous weights are updated based on these gradients.
- The weights are binarized using the sign function:

$$w_{\text{binary}} = \text{sign}(w)$$

Algorithm: BinaryConnect

Input:

- Training data x_i, y_i for $i = 1, \ldots, N$
- Neural network architecture (e.g., layers, activation functions)
- Number of epochs T
- Learning rate η

Output: Trained binary weights W_{bin} and biases b (usually in full-precision)

Algorithm: BinaryConnect

1. Initialize weights W and biases b with small random values
2. Initialize the binary weights $W_{bin} = \text{sign}(W)$
3. For each epoch $t = 1, \ldots, T$:
   - 3.1 For each training sample (x_i, y_i)
     - 3.1.1 Perform forward pass: calculate activations using the binary weights W_{bin}: $a_i = \sigma(W_{bin}x_i + b)$
     - 3.1.2 Compute the loss $L(x_i, y_i)$ (e.g., cross-entropy or mean squared error)
     - 3.1.3 Perform backward pass (backpropagation): compute the gradients ∇L with respect to continuous weights W
     - 3.1.4 Update continuous weights using the continuous gradients ∇L: $W = \text{clip}(W - \eta \nabla_W L, -1, 1)$, $b = b - \eta \nabla_b L$
     - 3.1.5 Update binary weights: binarize the continuous weights $W_{bin} = \text{sign}(W)$
4. Return the final binary weights W_{bin} and biases b

Backward pass

Key points:

- compute gradients as if the weights were continuous, even though they are binary: $\nabla L(W)$ by using the straight-through estimator (STE) for the sign function.
- Update the weights in full precision and then binarize:

$$W = W - \eta \nabla L(W)$$

Binarizing the weights

Often it is better to binarize stochastically

$$w_{bin} = \begin{cases} +1 & \text{with } p = \text{clip}(w, 0, 1) \\ -1 & \text{with } 1 - p \end{cases}$$

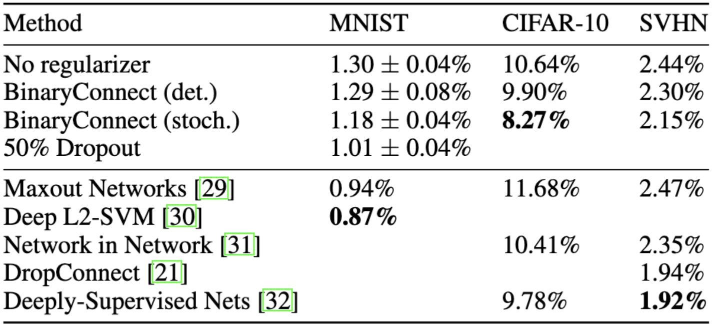
instead that

$$W_{bin} = \text{sign}(W)$$

Advantages of BinaryConnect

- Extreme reduction of memory requirements: binary weights take only 1 bit per weight.
- Faster computations: using binary values accelerates inference and training
- Suitable for embedded systems and mobile devices.
- Often retains near state-of-the-art performance despite the simplifications.

Numerical results



Method	MNIST	CIFAR-10	SVHN
No regularizer	$1.30 \pm 0.04\%$	10.64%	2.44%
BinaryConnect (det.)	$1.29 \pm 0.08\%$	9.90%	2.30%
BinaryConnect (stoch.)	$1.18 \pm 0.04\%$	**8.27%**	2.15%
50% Dropout	$1.01 \pm 0.04\%$		
Maxout Networks [29]	0.94%	11.68%	2.47%
Deep L2-SVM [30]	**0.87%**		
Network in Network [31]		10.41%	2.35%
DropConnect [21]			1.94%
Deeply-Supervised Nets [32]		9.78%	**1.92%**

Challenges and Limitations

- Binarizing weights can lead to reduced model expressiveness and lower performance for complex tasks.
- While STE works well in practice for many cases, it is still an approximation: the learning process may be less efficient or may converge to suboptimal solutions in some scenarios.
- Not all types of neural network architectures are suitable for binary weights (CNN, GANs, transformers..)
- BinaryConnect may require more epochs or larger learning rates to achieve comparable performance to networks with continuous weights.
- The optimization landscape may also be more noisy or less smooth due to the discretization of the weights

Post-Training Quantization

- Post-Training Quantization (PTQ) refers to the process of quantizing a pre-trained model without retraining.
- Typically applied after training a high-precision model (e.g., 32-bit floating point) for deployment on resource-constrained devices.
- It involves converting the model's weights and/or activations to lower bitwidth (e.g., 8-bit integers).

Methods for Post-Training Quantization

- **Quantizing Weights:** Weights can be quantized after training to reduce the model's size.
- **Quantizing Activations:** Activations are also quantized during inference to further reduce computational costs.
- **Calibration:** Calibration is used to select optimal scaling factors for quantization. It typically involves running a small dataset through the model to estimate the range of activations.

Calibration Process

- Calibration helps to determine the scaling factors that best preserve the model's accuracy after quantization.
- Common calibration techniques include:
  - **Min-Max Calibration:** Finds the minimum and maximum values of activations and weights to determine the quantization range.
  - **Histogram-based Calibration:** Uses a histogram of activation values to set more precise scaling factors.
- Calibration is especially important when quantizing activations to prevent a significant accuracy drop.

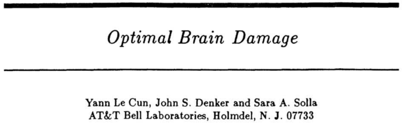
Challenges in Post-Training Quantization

- **Calibration Sensitivity:** The accuracy of the calibration process is critical, and improper calibration can lead to significant performance drops.
- **Automated Calibration Methods:** Advances in machine learning-based calibration techniques that do not require manual tuning.
- **Non-Uniform Distributions:** Some models have highly non-uniform weight distributions, making it harder to quantize efficiently.

Limitations of classical quantization schemes

- Traditional quantization methods usually focus on reducing bit-width uniformly across layers or channels.
- **Limitations**: Such methods can lead to significant accuracy loss because they ignore how sensitive to quantization the parameters in the different layers are.
- **Solution**: Use second-order information (i.e., the Hessian matrix) to guide the quantization process.
- Hessian Matrix: Captures the sensitivity of the model's loss function with respect to the parameters, providing richer information for quantization decisions.

Origins of Hessian aware quantization - 1994


Optimal Brain Damage

Yann Le Cun, John S. Denker and Sara A. Solla
AT&T Bell Laboratories, Holmdel, N. J. 07733

Aim: predict the effect of a parameter x_i on $f(\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix})$ without computing $f(\begin{bmatrix} x_1 \\ \vdots \\ 0 \\ \vdots \\ x_n \end{bmatrix})$ for all i

Idea: Taylor development

$$f(x) - f(\tilde{x}) \approx \nabla f(\tilde{x})^T (x - \tilde{x}) + \frac{1}{2}(x - \tilde{x})^T H(\tilde{x})(x - \tilde{x})$$

We want to find parameters x_i that make $f(x) - f(\tilde{x})$ small to suppress them.

OBD pruning

Simplifying assumptions:

- At convergence: $\nabla f(\tilde{x}) \approx 0$
- $\frac{1}{2}(x-\tilde{x})^T H(\tilde{x})(x-\tilde{x}) = \frac{1}{2}\sum_i H_{i,i}x_i^2 + \frac{1}{2}\sum_{i \neq j} H_{i,j}x_i x_j$
- We neglect the last term to reduce cost
- $\frac{1}{2}\sum_i H_{i,i}x_i^2$ can be efficiently computed by backpropagation with cost similar to that of the gradient

Saliency of a parameter: $s_i = H_{i,i}x_i^2/2$

OBD procedure

Procedure:

1. Choose a reasonable network architecture
2. Train the network until a reasonable solution is obtained
3. Compute the second derivatives for each parameter
4. Compute the saliencies for each parameter
5. Sort the parameters by saliency and delete some low-saliency parameters
6. Iterate to step 2

Numerical results

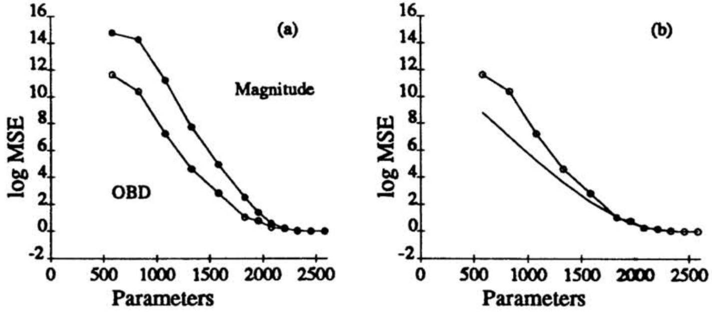


Figure 1: (a) Objective function (in dB) versus number of parameters for OBD (lower curve) and magnitude-based parameter deletion (upper curve). (b) Predicted and actual objective function versus number of parameters. The predicted value (lower curve) is the sum of the saliencies of the deleted parameters.

HAWQ: Hessian AWare Quantization

HAWQ: Hessian AWare Quantization of Neural Networks with Mixed-Precision, Z. Dong, Z. Yao, A. Gholami, M. Mahoney, K. Keutzer, 2019

- HAWQ leverages the Hessian matrix of the loss function to guide the quantization of weights.
- The Hessian matrix provides second-order information about the importance of each weight for the model's performance.
- The idea behind HAWQ is to reduce the precision (bit-width) of the weights that have low sensitivity to the loss, while preserving precision in important weights.
- This method helps minimize accuracy degradation during quantization, while achieving substantial compression.

How to measure sensitivity?

- Compute the eigenvalues of the Hessian of each block in the network.
- **Important**: it is not possible to explicitly form the Hessian since the size of a block can be quite large.
- **Solution**: compute the Hessian eigenvalues without explicitly forming it, using a matrix-free power iteration algorithm

Why the eigenvalues of the Hessian?

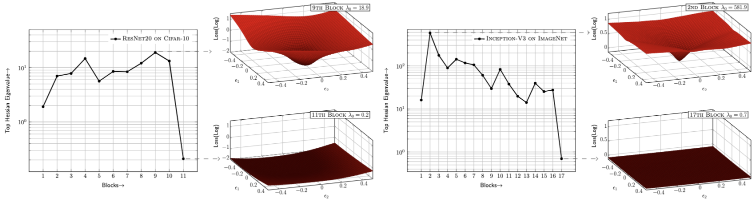


Fig. 1: *Top eigenvalue of each individual block of pre-trained ResNet20 on Cifar-10 (Left), and Inception-V3 on ImageNet (Right). Note that the magnitudes of eigenvalues of different blocks varies by orders of magnitude. See Figure 6 and 7 in appendix for the 3D loss landscape of other blocks.*

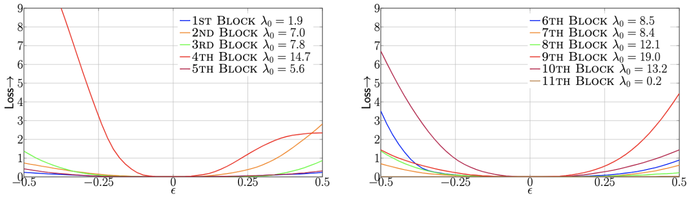
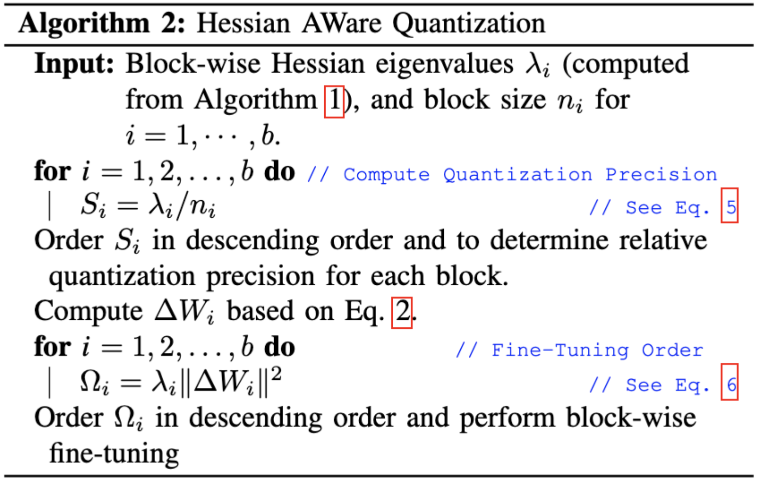


Fig. 2: *1-D loss landscape for different blocks of ResNet20 on Cifar-10. The landscape is plotted by perturbing model weights along the top Hessian eigenvector of each block, with a magnitude of ϵ (i.e., $\epsilon = 0$ corresponds to no perturbation).*

Hessian-Aware Quantization (HAWQ) Overview

1. Train the model with full precision (e.g., FP32).
2. Compute the sensitivity measure based on the Hessian eigenvalues
3. Apply adaptive quantization based on the sensitivity of the weights.
4. Fine-tune the model with quantized weights to minimize accuracy loss (quantization-aware multi-stage re-training)



```
Algorithm 2: Hessian AWare Quantization
Input: Block-wise Hessian eigenvalues $\lambda_i$ (computed
       from Algorithm 1), and block size $n_i$ for
       $i = 1, \cdots, b$.
for $i = 1, 2, \ldots, b$ do // Compute Quantization Precision
    $S_i = \lambda_i / n_i$                          // See Eq. 5
Order $S_i$ in descending order and to determine relative
  quantization precision for each block.
Compute $\Delta W_i$ based on Eq. 2.
for $i = 1, 2, \ldots, b$ do             // Fine-Tuning Order
    $\Omega_i = \lambda_i \|\Delta W_i\|^2$                  // See Eq. 6
Order $\Omega_i$ in descending order and perform block-wise
  fine-tuning
```

$$\Delta W_i = Q(W_i) - W_i$$

Fine tuning intuition : first fine-tune layers that have high curvature, which cause more perturbations after quantization.

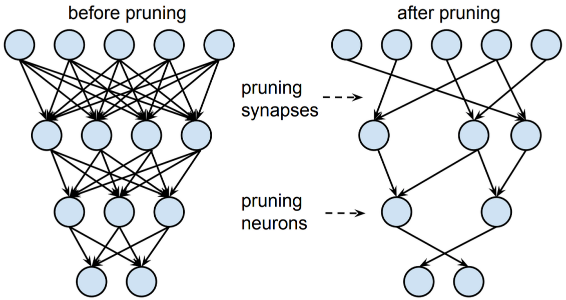
Table VI: *Block seperation and final block precision of ResNet20 on Cifar-10. Here we abbreviate convolutional layer as "Conv," fully connected layer as "FC."*

Block	Layer(s)	Layer Type	Parameter Size	Weight bit	Activation bit
Block 0	Layer 0	Conv	4.32e2	8	8
Block 1	Layer 1-2	Conv	4.61e3	6	4
Block 2	Layer 3-4	Conv	4.61e3	6	4
Block 3	Layer 5-6	Conv	4.61e3	8	4
Block 4	Layer 7-8	Conv	1.38e4	3	4
Block 5	Layer 9-10	Conv	1.84e4	3	4
Block 6	Layer 11-12	Conv	1.84e4	3	4
Block 7	Layer 13-14	Conv	5.53e4	2	4
Block 8	Layer 15-16	Conv	7.37e4	2	4
Block 9	Layer 17-18	Conv	7.37e4	2	4
Block 10	Layer 19	FC	6.40e2	3	8

Outline

Pruning

Definition: Select some neurons and/or weights and suppress them (set to zero)



How to choose?

Magnitude pruning

- Many of the learned weights have small magnitudes and contribute little to the network's performance.
- Magnitude pruning removes weights that have small magnitudes, reducing the complexity of the model.
- Pruning criterion: given a weight w_i, prune if

$$|w_i| < \tau$$

with τ chosen pruning threshold.

Pruning Process

1. Train the full Model
2. Compute Magnitudes of the weights
3. Sort weights based on their absolute magnitudes.
4. Set threshold τ (Top-k pruning, Percentage pruning, Global threshold)
5. Prune the weights that are below the threshold (set them to zero)
6. Retrain the model (fine-tune) for a few more epochs with the found mask. This allows the model to adjust to the new sparsity pattern and recover any lost performance.

Magnitude pruning

Advantage: pruning small weights can also act as a form of regularization, helping the model generalize better by reducing overfitting.

Variants:

- Layer-wise pruning
- Structured Pruning (removes entire neurons, filters (in CNNs), or channels in the network). This leads to a more structured sparsity pattern and can take advantage of hardware optimizations for matrix or tensor operations.
- Iterative Pruning: prune the network iteratively, pruning a small fraction of weights at each step and retraining the model after each pruning phase. This helps in minimizing performance degradation.

Other pruning criteria

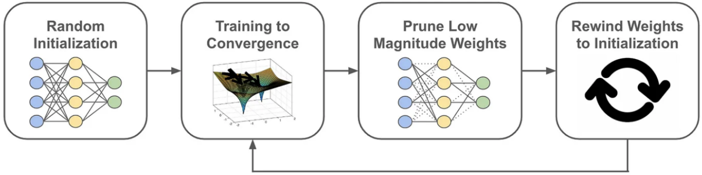
- Gradient pruning: reducing the number of parameters that are updated in each iteration by setting to zero small gradients $\rightarrow$ limitation: saturation (vanishing gradients), better Hessian pruning
- L_1 regularization $\rightarrow$ limitation: tuning of λ

$$regloss = loss + \lambda \|w\|_1$$

Lottery tickets

The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, J.Frankle, M. Carbin, 2019

Objective: Find a subnetwork of a large network, such that, if trained starting from the same w_0 maintains the same performance as the large network

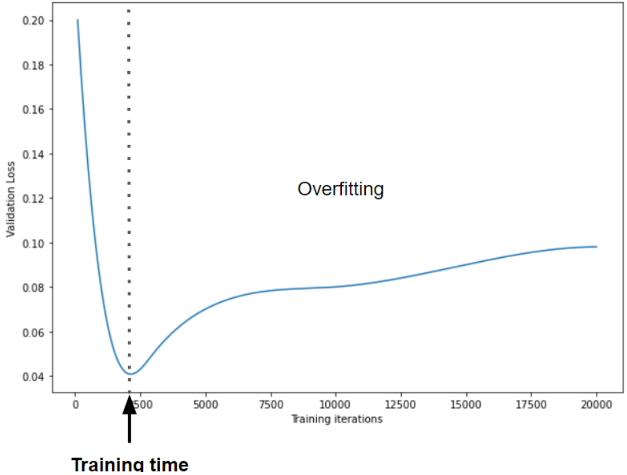


Lottery ticket algorithm

- Choose w_0 random
- Train the full network starting from w_0 and get w^*
- Prune the network based on the magnitude of w^*: select a mask m (binary matrix $0/1$) and set $w_p = w \cdot m$
- Reset $w_p = w_0 \cdot m$
- Train sub network with just weights w_p

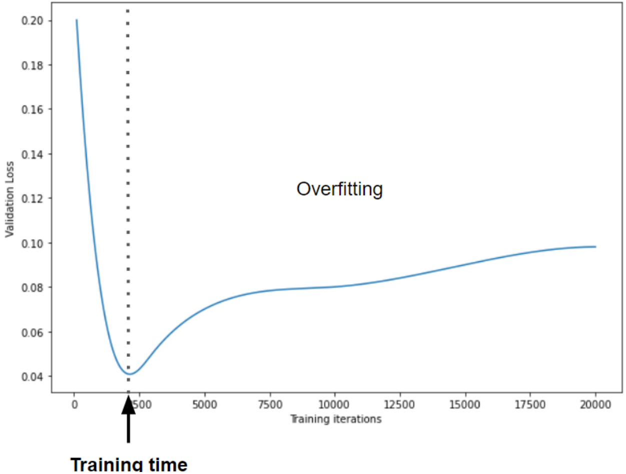
What do we expect from the subnetwork?

Definition of the training time



Definition of the training time

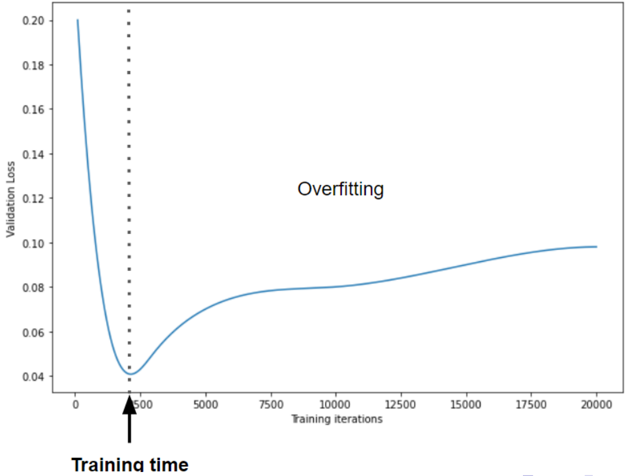
$\boxed{\text{First iteration at which it reaches minimum validation loss}}$



Definition of the training time

Training time $\#\mathrm{Iter}(M, I)$ of model M with initialization I

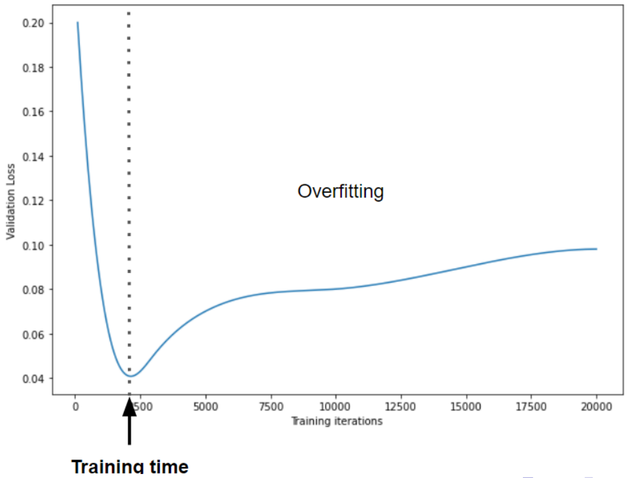
$$\boxed{\text{First iteration at which it reaches minimum validation loss}}$$



Definition of the training time

Training time $\#\mathrm{Iter}(M, I)$ of model M with initialization I given (D, A, H, L) (datasets, learning algorithm, hyperparameters, loss)

$\boxed{\text{First iteration at which it reaches minimum validation loss}}$



Definition of the training time

(M, I) is said to learn faster than (M', I') on (D, A, H, L) if

$$\#\mathrm{Iter}(M, I) \leqslant \#\mathrm{Iter}(M', I')$$

Definition of the training time

(M, I) is said to learn faster than (M', I') on (D, A, H, L) if

$$\#\mathrm{Iter}(M, I) \leqslant \#\mathrm{Iter}(M', I')$$

Remark 1: If the cost of one iteration of (D, A, H, L) for (M', I') is much cheaper than for (M, I), then the actual training time on a machine for M' could be smaller than the one for M.

Definition of the training time

(M, I) is said to learn faster than (M', I') on (D, A, H, L) if

$$\#\mathrm{Iter}(M, I) \leqslant \#\mathrm{Iter}(M', I')$$

Remark 1: If the cost of one iteration of (D, A, H, L) for (M', I') is much cheaper than for (M, I), then the actual training time on a machine for M' could be smaller than the one for M. For instance if M' is a smaller model than M, then an iteration for M' is likely to be cheaper than for M: be cautious if $\#\mathrm{Iter}(M, I) \leqslant \#\mathrm{Iter}(M', I')$.

Definition of the training time

(M, I) is said to learn faster than (M', I') on (D, A, H, L) if

$$\#\mathrm{Iter}(M, I) \leqslant \#\mathrm{Iter}(M', I')$$

Remark 1: If the cost of one iteration of (D, A, H, L) for (M', I') is much cheaper than for (M, I), then the actual training time on a machine for M' could be smaller than the one for M. For instance if M' is a smaller model than M, then an iteration for M' is likely to be cheaper than for M: be cautious if $\#\mathrm{Iter}(M, I) \leqslant \#\mathrm{Iter}(M', I')$.

Remark 2: Doing gradient-descent, if M' is a subnet of M and if it is trained by computing all the gradients of M and then zeroing the ones not in M', then an iteration for M' should have the same cost as for M. Remark 1 does not apply in this case.

Notations

Notations: $s \in [0, 1]$ = level of sparsity, M_s = subnet of M of sparsity s

Average random sparse subnets learn slower and are less accurate (empirical)

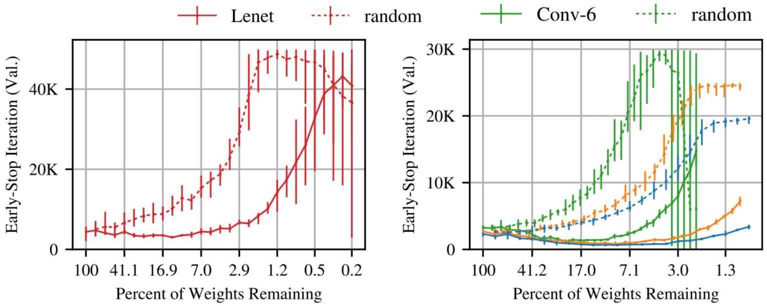


Figure: Dashed lines show the average training time for random subnets with random initializations of given sparsity of a fixed original model.

$$\#\text{Iter}(\text{original model}) \leqslant \#\text{Iter}(\text{random } M_s)$$

Average random sparse subnets learn slower and are less accurate (empirical)

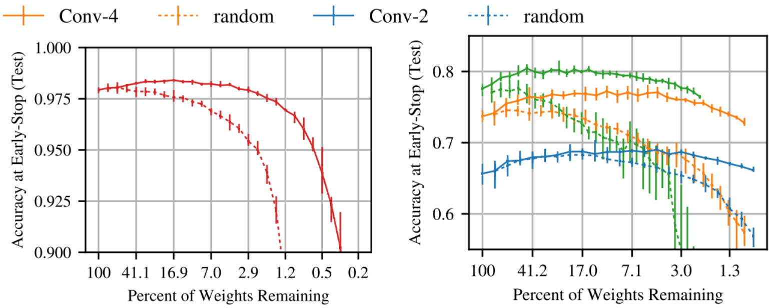


Figure: Dashed lines show the average top 1 accuracy for random subnets with random initializations of given sparsity of a fixed original model.

$$\text{Top1}(\text{trained random } M_s) \leqslant \text{Top1}(\text{trained original model})$$

Average random sparse subnets learn slower and are less accurate (empirical)

Average sparse subnets learn slower than the average trained full model: for M and s considered in the experiments and empirical means:

$$\mathbb{E}_I \#\mathrm{Iter}(M, I) \leqslant \mathbb{E}_{M_s, I} \#\mathrm{Iter}(M_s, I)$$

$$\max_I \#\mathrm{Iter}(M, I) \leqslant \min_{M_s, I} \#\mathrm{Iter}(M_s, I) \text{ for } s \text{ not too close from } 100\%$$

Average random sparse subnets learn slower and are less accurate (empirical)

Average sparse subnets learn slower than the average trained full model: for M and s considered in the experiments and empirical means:

$$\mathbb{E}_I \#\mathrm{Iter}(M, I) \leqslant \mathbb{E}_{M_s,I} \#\mathrm{Iter}(M_s, I)$$
$$\max_I \#\mathrm{Iter}(M, I) \leqslant \min_{M_s,I} \#\mathrm{Iter}(M_s, I) \text{ for } s \text{ not too close from } 100\%$$

Average sparse subnets are less accurate than the average trained full model: for M and s considered in the experiments and empirical means:

$$\mathbb{E}_I \mathrm{Top1}(M, I) \geqslant \mathbb{E}_{M_s,I} \mathrm{Top1}(M_s, I)$$
$$\min_I \mathrm{Top1}(M, I) \geqslant \max_{M_s,I} \mathrm{Top1}(M_s, I) \text{ for } s \text{ not too close from } 100\%$$

Challenge

Is it possible to find early on during training a sparse subnet that trains faster than the original model without accuracy degradation?

Definition of lottery tickets: (M_s, I) versus (M, I)

Lottery ticket: Fix (D, A, H, L). Consider a model and an initialization (M, I). A lottery ticket is a submodel (M_s, I) of sparsity s of (M, I).

Definition of lottery tickets: (M_s, I) versus (M, I)

Lottery ticket: Fix (D, A, H, L). Consider a model and an initialization (M, I). A lottery ticket is a submodel (M_s, I) of sparsity s of (M, I).

- **Average lottery ticket learns slower than the average full model:**

$$\mathbb{E}_I \#\mathrm{Iter}(M, I) \leqslant \mathbb{E}_{M_s, I} \#\mathrm{Iter}(M_s, I), \quad \forall M, \forall s.$$

Definition of lottery tickets: (M_s, I) versus (M, I)

Lottery ticket: Fix (D, A, H, L). Consider a model and an initialization (M, I). A lottery ticket is a submodel (M_s, I) of sparsity s of (M, I).

- **Average lottery ticket learns slower than the average full model:**

$$\mathbb{E}_I \#\mathrm{Iter}(M, I) \leqslant \mathbb{E}_{M_s, I} \#\mathrm{Iter}(M_s, I), \quad \forall M, \forall s.$$

- **Average lottery ticket is less accurate than the average trained full model:**

$$\mathbb{E}_I \mathrm{Top1}(M, I) \geqslant \mathbb{E}_{M_s, I} \mathrm{Top1}(M_s, I), \quad \forall M, \forall s.$$

Definition: Winning tickets

Winning lottery ticket: Fix $(D, A, H, L, \mathrm{Top1})$ with $\mathrm{Top1}$ a measure of accuracy. Consider a model and an initialization (M, I). A lottery ticket is a submodel (M_s, I) of sparsity s is **winning** if:

- **it learns faster** than the original model: $\#\mathrm{Iter}(M_s, I) \leqslant \#\mathrm{Iter}(M, I)$
- **it is more accurate** than the original model: $\mathrm{Top1}(M, I) \leqslant \mathrm{Top1}(M_s, I)$

Algorithm to find winning tickets

Algorithm of Iterative Pruning to find winning tickets:

- Train original model
- Layer-wise, prune p = 20% of the weights with the smallest magnitude (p/2% for the output layer)
- Iterate until desire sparsity is achieved

Novelty: IMP find subnets that can be trained efficiently from the start for unprecedented small level of sparsity without degradation of accuracy

Empirical performance of tickets found by Iterative Pruning

Learns faster[1] than the original model for $3.6\% \leqslant s \leqslant 100\%$:
$\mathbb{E}_I \#\mathrm{Iter}(\mathrm{IMP}(M, s), I) \leqslant \mathbb{E}_I \#\mathrm{Iter}(M, I)$

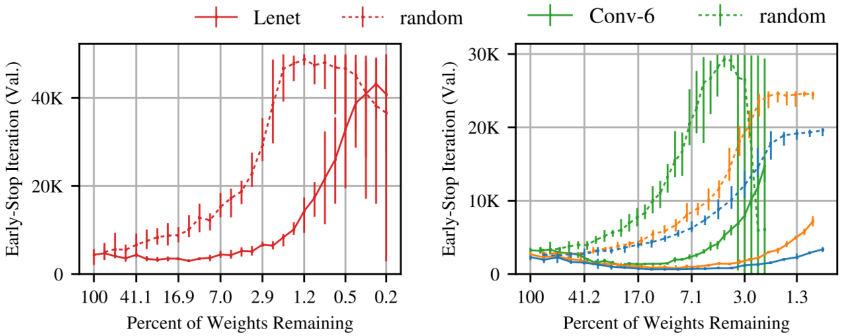
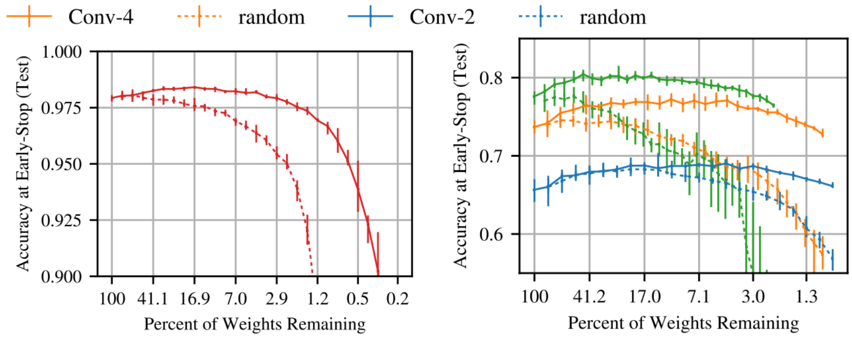


Figure: Dashed lines: random M_s, I. Solid lines: random $\mathrm{IMP}(M, s), I$.

[1]The training time decreases from $s = 100\%$ to $s = 21\%$ at which point early-stopping occurs 38% earlier than for the original model, then it increases

Empirical performance of tickets found by Iterative Pruning

More accurate[2] than the original model for $3.6\% \leqslant s \leqslant 100\%$:
$\mathbb{E}_I \mathrm{Top1}(\mathrm{IMP}(M, s), I) \geqslant \mathbb{E}_I \mathrm{Top1}(M, I)$



[2]The accuracy increases from $s = 100\%$ to $s = 13.5\%$ where it gained 0.3%, then it decreases

Structured sparsification

Aim: replace dense weight matrices with structured ones (e.g., sparse, low-rank, Fourier transform).

These methods have not seen widespread adoption:

- in end-to-end training due to unfavorable efficiency-quality tradeoffs,
- in dense-to-sparse fine-tuning of pretrained models due to lack of tractable algorithms to approximate a given dense weight matrix

Monarch networks

["Monarch: Expressive Structured Matrices for Efficient and Accurate Training", T. Dao et all, 2022]

Monarch matrices:

- hardware-efficient (they are parameterized as products of two block-diagonal matrices for better hardware utilization)
- expressive (they can represent many commonly used transforms).
- The problem of approximating a dense weight matrix with a Monarch matrix, though nonconvex, has an analytical optimal solution.

Monarch matrices

Definition 3.1. Let $n = m^2$. An $n \times n$ *Monarch matrix* has the form:

$$\mathbf{M} = \mathbf{P}\mathbf{L}\mathbf{P}^\top\mathbf{R},$$

where $\mathbf{L}$ and $\mathbf{R}$ are block-diagonal matrices, each with m blocks of size $m \times m$, and $\mathbf{P}$ is the permutation that maps $[x_1, \ldots, x_n]$ to $[x_1, x_{1+m}, \ldots, x_{1+(m-1)m}, x_2, x_{2+m}, \ldots,$ $x_{2+(m-1)m}, \ldots, x_m, x_{2m}, \ldots, x_n]$.

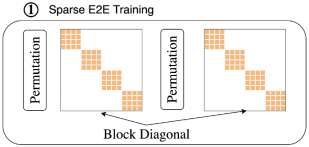


Figure 2: Monarch matrices are parametrized as products of two block-diagonal matrices up to permutation, allowing efficient multiplication algorithm that leverages batch matrix multiply.

We can interpret $P = P^T$ as follows: it reshapes a vector x of size n as a matrix of size $m \times m$, transposes the matrix, then converts back into a vector of size n.

Link with butterfly matrices

- Let B butterfly matrix of size n where n is a power of 4.
- $B = \underbrace{B_1 \dots B_{\log_2(n)/2}}_{L'} \underbrace{B_{\log_2(n)/2} \dots B_{\log_2(n)}}_{R}$
- R is block-diagonal with $m = \sqrt{n}$ dense blocks, each block of size $m \times m$
- L' is composed of $m \times m$ blocks of size $m \times m$, where each block is a diagonal matrix:

$$L' = \begin{bmatrix} D_{11} & \dots & D_{1m} \\ \dots & \dots & \dots \\ D_{m1} & \dots & D_{mm} \end{bmatrix}$$

- L' can be written as block-diagonal with the same structure as R after permuting the rows and columns.
- $L = PL'P^T$: up to permuting rows and columns, L' is also a block-diagonal matrix of m dense blocks, each of size $m \times m$.
- B butterly implies B monarch

Projection algorithm

Special case of matrix factorization algorithm - cf. Cours 10 Sparse matrix factorization

$\rightarrow$ Solve a series of block SVDs

Compression - end-to-end (E2E)

Replacing dense matrices with Monarch matrices in Vision Transformer ViT, MLP-Mixer (ImageNet), and GPT-2 (WikiText-103) can speed up training by up to 2× without sacrificing model quality

Table 1: The performance of Monarch matrices and ViT / MLP-Mixer on ImageNet, including the number of parameters and FLOPs. We measure the Top-1 accuracy and the training time speedup compared to the corresponding dense model.

Model	ImageNet acc.	Speedup	Params	FLOPs
Mixer-S/16	74.0	-	18.5M	3.8G
Monarch-Mixer-S/16	73.7	1.7×	7.0M	1.5G
Mixer-B/16	77.7	-	59.9M	12.6G
Monarch-Mixer-B/16	77.8	1.9×	20.9M	5.0G
ViT-S/16	79.4	-	48.8M	9.9G
Monarch-ViT-S/16	79.1	1.9×	19.6M	3.9G
ViT-B/16	78.5	-	86.6M	17.6G
Monarch-ViT-B/16	78.9	2.0×	33.0M	5.9G

Table 2: Performance of Monarch matrices and GPT-2-Small/Medium on WikiText-103, including the # of parameters and FLOPs. Monarch achieves similar perplexity (ppl) but 2.0× faster.

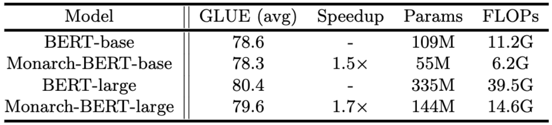
Model	PPL	Speedup	Params	FLOPs
GPT-2-Small	20.6	-	124M	106G
Monarch-GPT-2-Small	20.7	1.8×	72M	51G
GPT-2-Medium	20.9	-	355M	361G
Monarch-GPT-2-Medium	20.3	2.0×	165M	166G

Compression - Denso-to-sparse

Procedure: BERT pretrained weights, approximate them with Monarch matrices, and finetune the resulting model on the 9 GLUE tasks (collection of nine natural language understanding tasks).

Result: Monarch finetuned model with similar quality to the dense BERT model, but with 1.7× faster finetuning speed.

Table 8: The performance of Monarch matrices in finetuning BERT on GLUE.

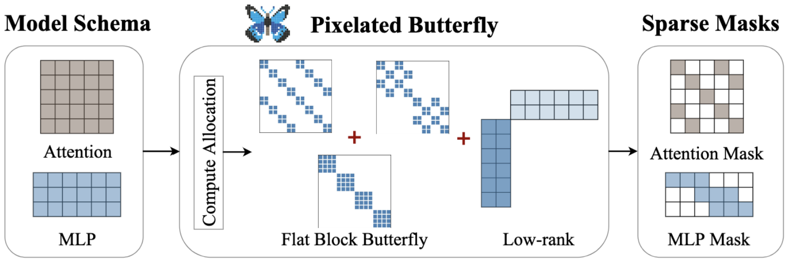


Model	GLUE (avg)	Speedup	Params	FLOPs
BERT-base	78.6	-	109M	11.2G
Monarch-BERT-base	78.3	1.5×	55M	6.2G
BERT-large	80.4	-	335M	39.5G
Monarch-BERT-large	79.6	1.7×	144M	14.6G

Pixelated butterfly

Pixelated Butterfly: Simple and Efficient Sparse Training for Neural Network Models, Dao et all, 2022

- An approach similar to the previous one, but that uses butterfly factorizations
- As classical butterfly matrices are not hardware efficient, they propose variants of butterfly (block and flat) to take advantage of modern hardware.



Recent development : "Fast inference with Kronecker-sparse matrices" A. Gonon, L. Zheng, P. Carrivain, Q. Le, 2024 (GPU matrix multiplication algorithms specialized for Kronecker-sparse matrices)