Quantizing neural networks

Efficient Deep Learning - Session 3

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

2025-02-12 Quantizing neural networks

Quantizing neural networks

Efficient Deep Learning - Session 3

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Course organisation

Sessions

1. Intro Deep Learning,
2. Data Augmentation and Self Supervised Learning,
3. Quantization,
4. Pruning,
5. Factorization,
6. Distillation,
7. Embedded Software and Hardware for DL,
8. Presentations for challenge.

Course organisation
Sessions
1. Intro Deep Learning,
2. Data Augmentation and Self Supervised Learning,
3. Quantization,
4. Pruning,
5. Factorization,
6. Distillation,
7. Embedded Software and Hardware for DL,
8. Presentations for challenge.

Course organisation

Sessions

1. Intro Deep Learning,
2. Data Augmentation and Self Supervised Learning,
3. Quantization,
4. Pruning,
5. Factorization,
6. Distillation,
7. Embedded Software and Hardware for DL,
8. Presentations for challenge.

Course organisation

Sessions

1. Intro Deep Learning,
2. Data Augmentation and Self Supervised Learning,
3. Quantization,
4. Pruning,
5. Factorization,
6. Distillation,
7. Embedded Software and Hardware for DL,
8. Presentations for challenge.

Today's Summary

2025-02-12
Quantizing neural networks
└Today's Summary

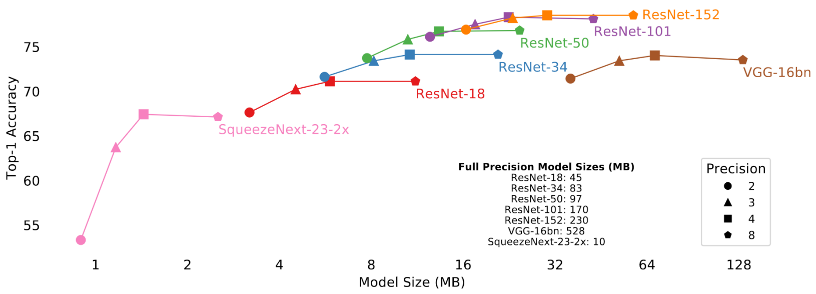
Today's Summary
1 Objectives
2 Quantization : Basics
Floating Point
Integers, Fixed Point
Quantization
3 Quantization : Neural Networks
Quantization Post Training
Quantization Aware Training
4 Quantization in Pytorch

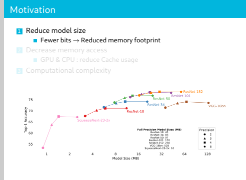
Plan

Plan
1 Objectives
2 Quantization : Basics
Floating Point
Integers, Fixed Point
Quantization
3 Quantization : Neural Networks
Quantization Post Training
Quantization Aware Training
4 Quantization in Pytorch

Motivation

1. Reduce model size
   - Fewer bits → Reduced memory footprint
2. Decrease memory access
   - GPU & CPU : reduce Cache usage
3. Computational complexity

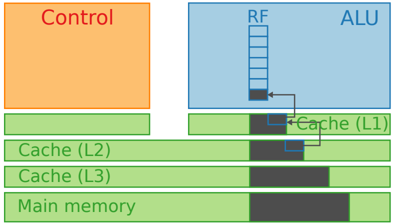


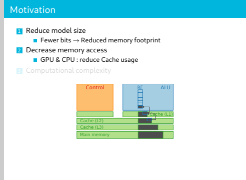


Reducing the memory footprint can be beneficial particularly in the case of embedded systems. On an ESP32, which is commonly used for low power applications, the memory footprint is limited to 520KB.

Motivation

1. Reduce model size
   - Fewer bits $\longrightarrow$ Reduced memory footprint
2. Decrease memory access
   - GPU & CPU : reduce Cache usage
3. Computational complexity

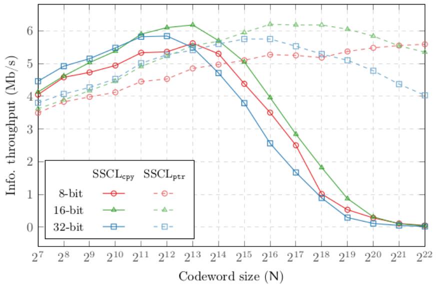


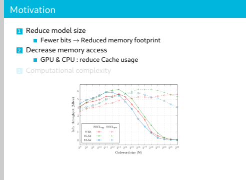


The data bandwidth between the cache and the GPU or the CPU is limited. This can be the bottleneck of the implementation, limiting the throughput or the latency. Therefore, reducing the cache usage is also beneficial for the performance of neural network processing.

Motivation

1. Reduce model size
   - Fewer bits $\rightarrow$ Reduced memory footprint
2. Decrease memory access
   - GPU & CPU : reduce Cache usage
3. Computational complexity

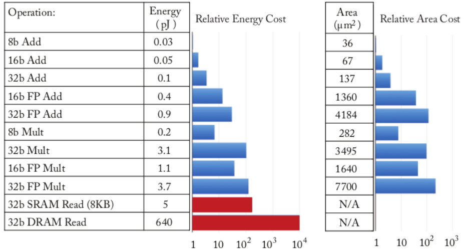




This is an example outside of the scope of Deep Learning. This is a simulation of a channel decoder inside a communication system. On the x-axis is the size of the codeword, which is the size of the main array of data that is processed by the algorithm. On the y-axis is the throughput of the system. The algorithm "SSCLcpy" is there an example of a cache issue, where the array becomes too big to fit in the first level of caches, with a dramatic decrease of the throughput after a certain array size.

Motivation

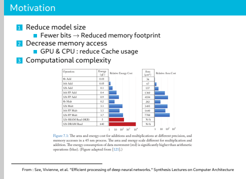
1. Reduce model size
   - Fewer bits → Reduced memory footprint
2. Decrease memory access
   - GPU & CPU : reduce Cache usage
3. Computational complexity



Operation:	Energy (pJ)	Area (μm^2)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

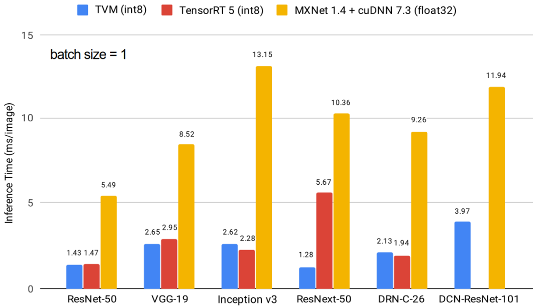
Figure 7.1: The area and energy cost for additions and multiplications at different precision, and memory accesses in a 45 nm process. The area and energy scale different for multiplication and addition. The energy consumption of data movement (red) is significantly higher than arithmetic operations (blue). (Figure adapted from [121].)

From : Sze, Vivienne, et al. "Efficient processing of deep neural networks." Synthesis Lectures on Computer Architecture



These are the relative energy cost of doing different operations on a 45 nm ASIC. The energy cost of floating point operations is higher than the energy cost of integer operations. It takes also more space on the chip, which means that there is a higher parallelization potential with fixed point operations than with floating point ones.

1. Reduce model size
   - Fewer bits → Reduced memory footprint
2. Decrease memory access
   - GPU & CPU : reduce Cache usage
3. Computational complexity



From : `https://github.com/vinx13/tvm-cuda-int8-benchmark/`

This is a concrete example of the benefits of quantization on the inference time of different neural networks, on an NVIDIA GTX 1080, using different frameworks. The inference time is measured in milliseconds. Different frameworks are used, MXNet and cuDNN for the floating point implementation, and TVM and tensorRT for the fixed point implementation. For each neural network, inference time is lower with int8 than with float32.

Plan

Plan

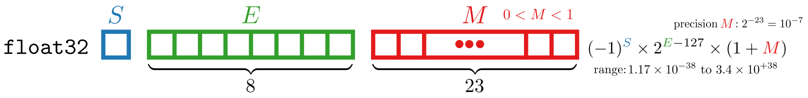
Objectives

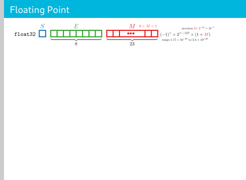
2 Quantization : Basics
- Floating Point
- Integers, Fixed Point
- Quantization

Quantization : Neural Networks
- Quantization Post Training
- Quantization Aware Training

Quantization in Pytorch

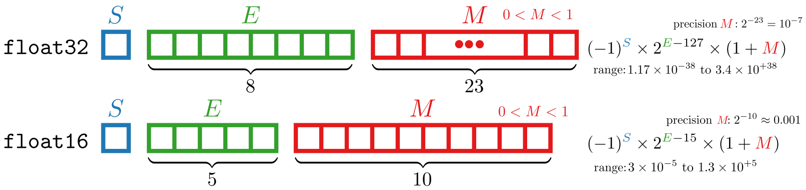
Floating Point

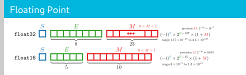




Here are three different floating point representations. S is the sign, E is the exponent, and M is the mantissa. The first representation is the IEEE 754 standard, which is the most common one. The second one is the float16 representation format, which uses only 16 bits. The third one is the bfloat16 representation format, which uses only 16 bits, but with different numbers of bits in the mantissa and in the exponent. It has been designed to work well for neural network training. Performing operations like addition or multiplication of floating point numbers is more complex than with integers.

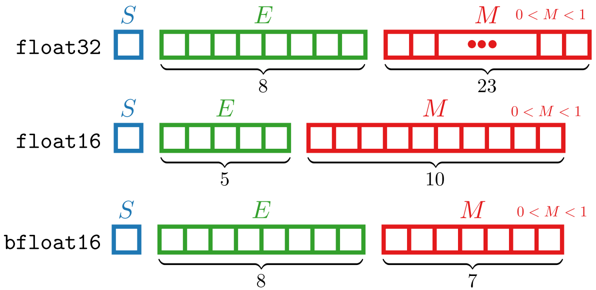
Floating Point





Here are three different floating point representations. S is the sign, E is the exponent, and M is the mantissa. The first representation is the IEEE 754 standard, which is the most common one. The second one is the float16 representation format, which uses only 16 bits. The third one is the bfloat16 representation format, which uses only 16 bits, but with different numbers of bits in the mantissa and in the exponent. It has been designed to work well for neural network training. Performing operations like addition or multiplication of floating point numbers is more complex than with integers.

Floating Point



float32: precision $M : 2^{-23} = 10^{-7}$

$$(-1)^S \times 2^{E-127} \times (1 + M)$$

range: 1.17×10^{-38} to $3.4 \times 10^{+38}$

float16: precision M: $2^{-10} \approx 0.001$

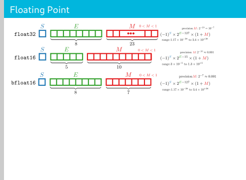
$$(-1)^S \times 2^{E-15} \times (1 + M)$$

range: 3×10^{-5} to $1.3 \times 10^{+5}$

bfloat16: precision M: $2^{-7} \approx 0.001$

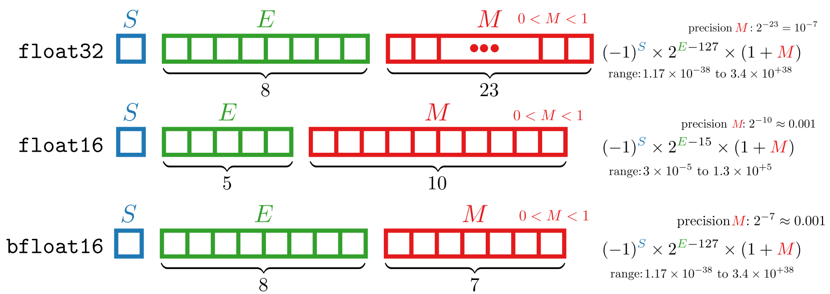
$$(-1)^S \times 2^{E-127} \times (1 + M)$$

range: 1.17×10^{-38} to $3.4 \times 10^{+38}$

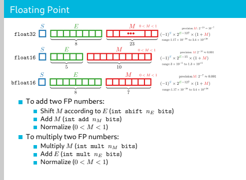


Here are three different floating point representations. S is the sign, E is the exponent, and M is the mantissa. The first representation is the IEEE 754 standard, which is the most common one. The second one is the float16 representation format, which uses only 16 bits. The third one is the bfloat16 representation format, which uses only 16 bits, but with different numbers of bits in the mantissa and in the exponent. It has been designed to work well for neural network training. Performing operations like addition or multiplication of floating point numbers is more complex than with integers.

Floating Point

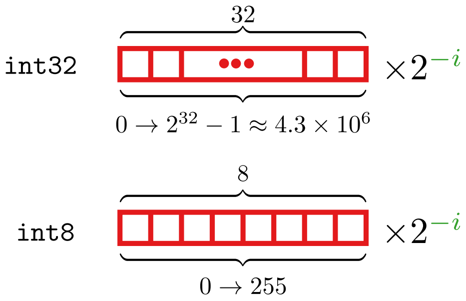


- To add two FP numbers:
  - Shift M according to E (`int shift` n_E `bits`)
  - Add M (`int add` n_M `bits`)
  - Normalize ($0 < M < 1$)
- To multiply two FP numbers:
  - Multiply M (`int mult` n_M `bits`)
  - Add E (`int mult` n_E `bits`)
  - Normalize ($0 < M < 1$)



Here are three different floating point representations. S is the sign, E is the exponent, and M is the mantissa. The first representation is the IEEE 754 standard, which is the most common one. The second one is the float16 representation format, which uses only 16 bits. The third one is the bfloat16 representation format, which uses only 16 bits, but with different numbers of bits in the mantissa and in the exponent. It has been designed to work well for neural network training. Performing operations like addition or multiplication of floating point numbers is more complex than with integers.

Integers, fixed point

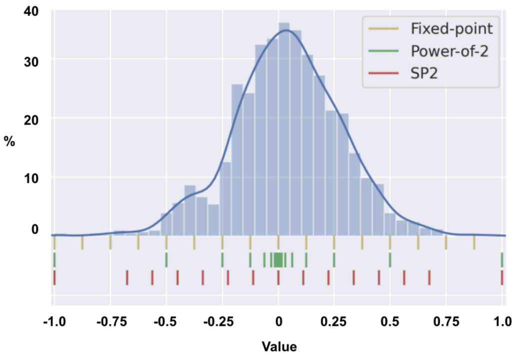


- Fixed point ($-i$)
- Short range
- Simple computation

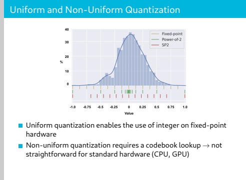
Integers, fixed point
Fixed point ($-i$)
Short range
Simple computation

On the other side, the main way to represent fractional numbers with integers is the fixed point representation. i is the number of fractional bits. The range of the representation is $[-2^{i-1}, 2^{i-1} - 1]$. The representation is simple, but the range is limited.

Uniform and Non-Uniform Quantization

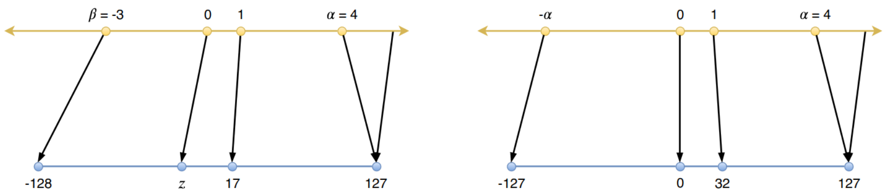


- Uniform quantization enables the use of integer on fixed-point hardware
- Non-uniform quantization requires a codebook lookup $\rightarrow$ not straightforward for standard hardware (CPU, GPU)



There are other ways to use integers to represent real numbers. The different values of the integers can represent arbitrary real numbers in a uniform or non-uniform way.

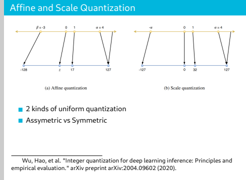
Affine and Scale Quantization

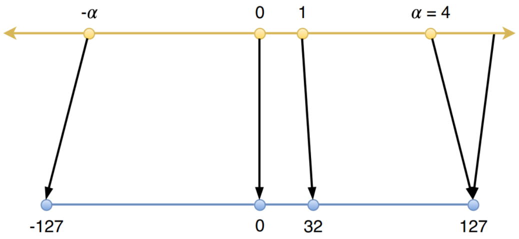


(a) Affine quantization (b) Scale quantization

- 2 kinds of uniform quantization
- Assymetric vs Symmetric

Wu, Hao, et al. "Integer quantization for deep learning inference: Principles and empirical evaluation." arXiv preprint arXiv:2004.09602 (2020).



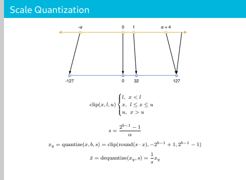


$$\mathrm{clip}(x,l,u)\begin{cases} l, & x < l \\ x, & l \leq x \leq u \\ u, & x > u \end{cases}$$

$$s = \frac{2^{b-1}-1}{\alpha}$$

$$x_q = \mathrm{quantize}(x,b,s) = \mathrm{clip}(\mathrm{round}(s \cdot x), -2^{b-1}+1, 2^{b-1}-1)$$

$$\hat{x} = \mathrm{dequantize}(x_q, s) = \frac{1}{s}x_q$$



The upper arrow represents the real axis, the lower one represent the quantized axis. Scale quantizaiton is a category of uniform quantization, where real number 0 is represented as the integer 0. It is also symmetric, as there is the same number of positive and negative values on the quantized axis. Only one parameter is then needed to define the quantization, which is the real number α that corresponds to the maximum integer value. The axes are then split uniformly in $2^b - 1$ intervals. The *quantize* and *dequantize* functions are defined accordingly.

$$y_{ij} = \sum_{k=1}^{p} x_{ik} \cdot w_{kj} \approx$$

$$\sum_{k=1}^{p} \text{dequantize}(x_{q,ik}, s_{q,ik}) \cdot \text{dequantize}(w_{q,kj}, s_{w,kj}) =$$

$$\sum_{k=1}^{p} \frac{1}{s_{x,ik}} x_{q,ik} \cdot \frac{1}{s_{w,kj}} w_{q,kj}$$

And, in order to use integer multiplication, the scaling factor s must be independent of k :

$$\frac{1}{s_{x,i} \cdot s_{w,j}} \sum_{k=1}^{p} x_{q,ik} \cdot w_{q,kj}$$

Scale Quantization

$$y_{ij} = \sum_{k=1}^{p} x_{ik} \cdot w_{kj} \approx$$

$$\sum_{k=1}^{p} \text{dequantize}(x_{q,ik}, s_{q,ik}) \cdot \text{dequantize}(w_{q,kj}, s_{w,kj}) =$$

$$\sum_{k=1}^{p} \frac{1}{s_{x,ik}} x_{q,ik} \cdot \frac{1}{s_{w,kj}} w_{q,kj}$$

And, in order to use integer multiplication, the scaling factor s must be independent of k :

$$\frac{1}{s_{x,i} \cdot s_{w,j}} \sum_{k=1}^{p} x_{q,ik} \cdot w_{q,kj}$$

This is the important part about scale quantization. When the scaling factor is constant over a part of the data (weights and activations), the scaling to go back to floating point values can be done a posteriori, after all of the computations have been done with integers.

Plan

2025-02-12

Quantizing neural networks
└ Quantization : Neural Networks
└ Plan

Plan

1 Objectives

2 Quantization : Basics
- Floating Point
- Integers, Fixed Point
- Quantization

3 Quantization : Neural Networks
- Quantization Post Training
- Quantization Aware Training

4 Quantization in Pytorch

Quantization Post Training : Weights

Quantize a set of parameters

- Select the number of quantization bits,
- Select the set of parameters to quantize,
- Determine range of values,
- Determine the scaling factor (and zero if affine).

Different weight sets can be considered

- Whole network,
- per layer,
- per neuron.

Finer sets segmentation $\rightarrow$ better accuracy.

Quantization Post Training : Weights
Quantize a set of parameters
- Select the number of quantization bits,
- Select the set of parameters to quantize,
- Determine range of values,
- Determine the scaling factor (and zero if affine).
Different weight sets can be considered
- Whole network,
- per layer,
- per neuron.
Finer sets segmentation $\rightarrow$ better accuracy.

Quantization Post Training : Activation

Quantize a set of parameters

- Select the number of quantization bits,
- Select the se t of activations to quantize,
- Determine range of values **according to the training set or a subset**,
- Determine the scaling factor (and zero if affine).

Different weight sets can be considered

- Whole network,
- per layer,
- per neuron.

Finer sets segmentation → better accuracy.

2025-02-12

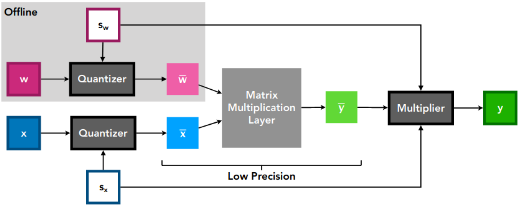
Quantization Post Training : Activation
Quantize a set of parameters
- Select the number of quantization bits,
- Select the se t of activations to quantize,
- Determine range of values **according to the training set or a subset**,
- Determine the scaling factor (and zero if affine).
Different weight sets can be considered
- Whole network,
- per layer,
- per neuron.
Finer sets segmentation → better accuracy.

Quantization Aware Training

- Train the network while quantizing the weights and / or the activations,
- Quantization Aware Techniques yield way better accuracy,
- Especially for extremely low-bit precision (2-3-4 bit precision).

Quantization Aware Training
- Train the network while quantizing the weights and / or the activations,
- Quantization Aware Techniques yield way better accuracy,
- Especially for extremely low-bit precision (2-3-4 bit precision).

Learned Step Size Quantization



- s quantizer step size
- Q_P and Q_N, the number of positive and negative quantization levels

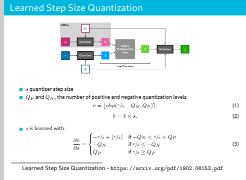
$$\bar{v} = \lfloor clip(^v/_s, -Q_N, Q_P) \rceil, \quad (1)$$

$$\hat{v} = \bar{v} \times s. \quad (2)$$

- s is learned with :

$$\frac{\partial \hat{v}}{\partial s} = \begin{cases} -^v/_s + \lfloor ^v/_s \rceil & \text{if } -Q_N < {^v/_s} < Q_P \\ -Q_N & \text{if } {^v/_s} \leq -Q_N \\ Q_P & \text{if } {^v/_s} \geq Q_P \end{cases} \quad (3)$$

Learned Step Size Quantization - `https://arxiv.org/pdf/1902.08153.pdf`



LSQ is a technique that allows to learn the step size s of the quantizer. The step size is learned with a gradient descent algorithm. Weights quantization can be done offline, which means that no additional computation is required during inference.

Quantization while Learning - Binary Connect

Algorithm 1 SGD training with BinaryConnect. C is the cost function for minibatch and the functions binarize(w) and clip(w) specify how to binarize and clip weights. L is the number of layers.

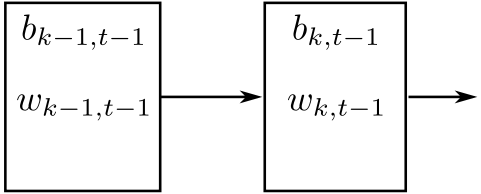
Require: a minibatch of (inputs, targets), previous parameters w_{t-1} (weights) and b_{t-1} (biases), and learning rate η.
Ensure: updated parameters w_t and b_t.
1. Forward propagation:
$w_b \leftarrow \text{binarize}(w_{t-1})$
For $k = 1$ to L, compute a_k knowing a_{k-1}, w_b and b_{t-1}
2. Backward propagation:
Initialize output layer's activations gradient $\frac{\partial C}{\partial a_L}$
For $k = L$ to 2, compute $\frac{\partial C}{\partial a_{k-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and w_b
3. Parameter update:
Compute $\frac{\partial C}{\partial w_b}$ and $\frac{\partial C}{d b_{t-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and a_{k-1}
$w_t \leftarrow \text{clip}(w_{t-1} - \eta \frac{\partial C}{\partial w_b})$
$b_t \leftarrow b_{t-1} - \eta \frac{\partial C}{\partial b_{t-1}}$

Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David. "Binaryconnect: Training deep neural networks with binary weights during propagations." Advances in neural information processing systems. 2015. `https://arxiv.org/pdf/1511.00363.pdf`

Binary Connect is a technique that allows to train a network with binary weights. The weights are binarized during the forward pass and then the gradients are backpropagated. The weights are updated with a gradient descent algorithm. During training, the weights are binarized, which means that the network is not differentiable. To overcome this issue, Straight Through Estimator (STE) is used. It consists of replacing the gradient of the binarization function by the gradient of the identity function.

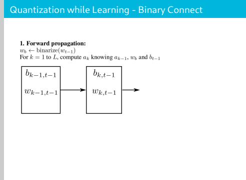
Quantization while Learning - Binary Connect

1. Forward propagation:
$w_b \leftarrow \text{binarize}(w_{t-1})$
For $k = 1$ to L, compute a_k knowing a_{k-1}, w_b and b_{t-1}

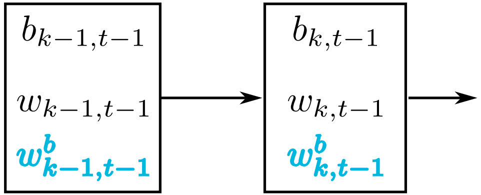


2025-02-12

Quantizing neural networks
└ Quantization : Neural Networks
└ Quantization Aware Training
└ Quantization while Learning - Binary Connect



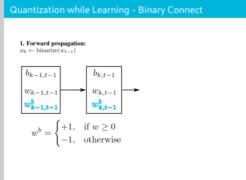
1. Forward propagation:
$w_b \leftarrow \text{binarize}(w_{t-1})$



$$w^b = \begin{cases} +1, & \text{if } w \geq 0 \\ -1, & \text{otherwise} \end{cases}$$

2025-02-12

Quantizing neural networks
└Quantization : Neural Networks
└Quantization Aware Training
└Quantization while Learning - Binary Connect

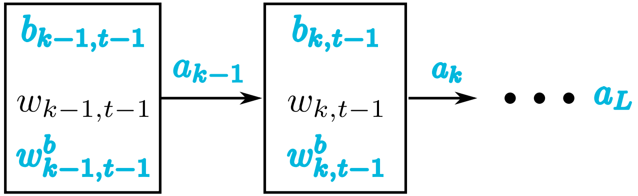


Quantization while Learning - Binary Connect

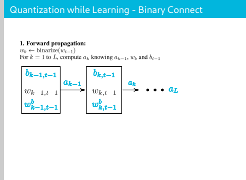
1. Forward propagation:

$w_b \leftarrow \text{binarize}(w_{t-1})$

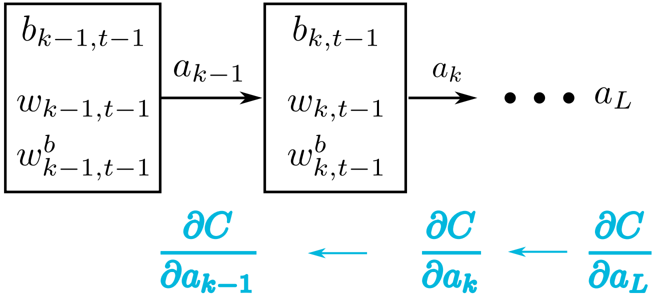
For $k = 1$ to L, compute a_k knowing a_{k-1}, w_b and b_{t-1}



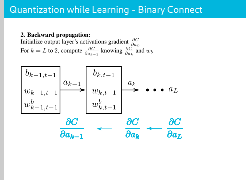
2025-02-12



2. Backward propagation:
Initialize output layer's activations gradient $\frac{\partial C}{\partial a_L}$
For $k = L$ to 2, compute $\frac{\partial C}{\partial a_{k-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and w_b



2025-02-12

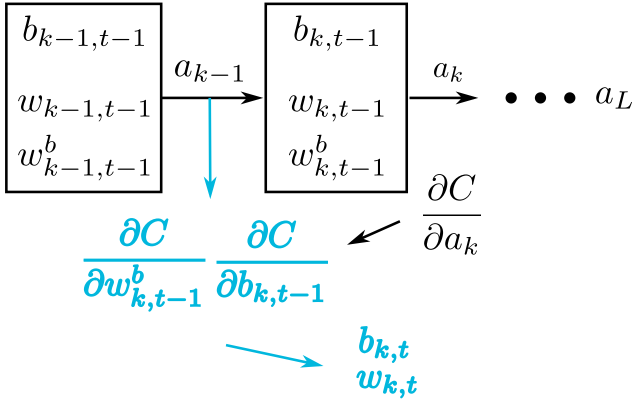


3. Parameter update:

Compute $\frac{\partial C}{\partial w_b}$ and $\frac{\partial C}{db_{t-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and a_{k-1}

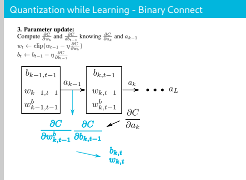
$w_t \leftarrow \text{clip}(w_{t-1} - \eta \frac{\partial C}{\partial w_b})$

$b_t \leftarrow b_{t-1} - \eta \frac{\partial C}{\partial b_{t-1}}$



2025-02-12

Quantizing neural networks
└Quantization : Neural Networks
└Quantization Aware Training
└Quantization while Learning - Binary Connect



Binarization : Stochastic vs Deterministic

- Deterministic

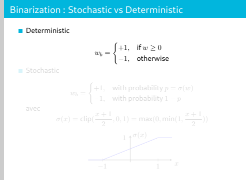
$$w_b = \begin{cases} +1, & \text{if } w \geq 0 \\ -1, & \text{otherwise} \end{cases}$$

- Stochastic

$$w_b = \begin{cases} +1, & \text{with probability } p = \sigma(w) \\ -1, & \text{with probability } 1 - p \end{cases}$$

avec

$$\sigma(x) = \text{clip}(\frac{x+1}{2}, 0, 1) = \max(0, \min(1, \frac{x+1}{2}))$$



Two different ways to binarize the weights exist. The first one is deterministic, which means that the weights are binarized with a threshold. The second one is stochastic, which means that the weights are binarized with a probability.

Binarization : Stochastic vs Deterministic

- Deterministic

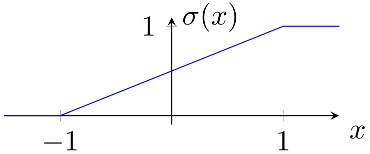
$$w_b = \begin{cases} +1, & \text{if } w \geq 0 \\ -1, & \text{otherwise} \end{cases}$$

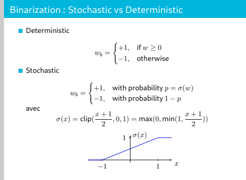
- Stochastic

$$w_b = \begin{cases} +1, & \text{with probability } p = \sigma(w) \\ -1, & \text{with probability } 1 - p \end{cases}$$

avec

$$\sigma(x) = \text{clip}(\frac{x+1}{2}, 0, 1) = \max(0, \min(1, \frac{x+1}{2}))$$





Two different ways to binarize the weights exist. The first one is deterministic, which means that the weights are binarized with a threshold. The second one is stochastic, which means that the weights are binarized with a probability.

Plan

Plan

1 Objectives

2 Quantization : Basics
- Floating Point
- Integers, Fixed Point
- Quantization

3 Quantization : Neural Networks
- Quantization Post Training
- Quantization Aware Training

4 Quantization in Pytorch

Quantization in Pytorch

1. Dynamic Quantization
2. Static Quantization
3. Quantization Aware Training

https://pytorch.org/docs/stable/quantization.html

And for our need :
https://pytorch.org/tutorials/prototype/pt2e_quant_ptq.html

Quantization in Pytorch
1. Dynamic Quantization
2. Static Quantization
3. Quantization Aware Training
https://pytorch.org/docs/stable/quantization.html
And for our need :
https://pytorch.org/tutorials/prototype/pt2e_quant_ptq.html

Quantization in Pytorch is a new feature. It is still in beta version. A group of students from last year worked on this topic. They implemented a post quantization algorithm in order to quantize the network with 8-bit weights and activations.