Quantized Machine Learning Framework

Bachelor Project

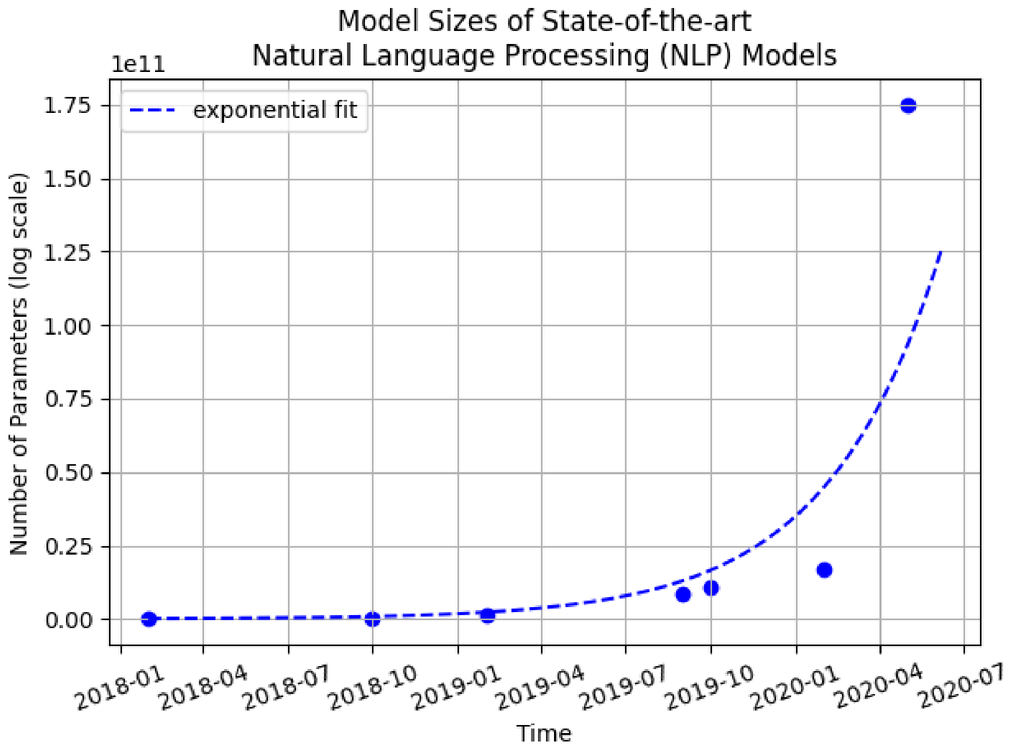
Mathias Bouilloud (supervised by Canberk Sonmez)

PARALLEL SYSTEMS ARCHITECTURE LAB

EPFL

DNN Complexity and Carbon Footprint

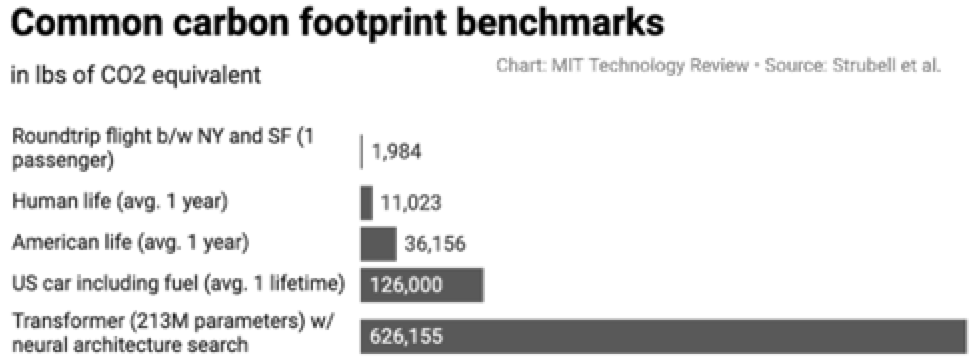
Model size increasing…



…datasets too

CIFAR100 - 50k train, and 10k test images

ImageNet - 1.2M training, and 50k test images



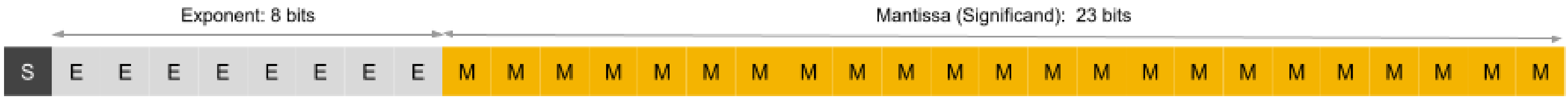
DNN training is not sustainable due to the daunting carbon footprint!

A possible solution: Quantization

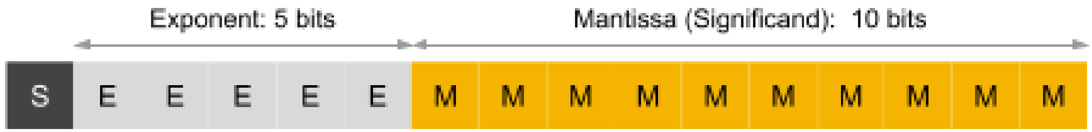
- Machine Learning algorithms defined over real numbers
- Computers approximate real values by quantizing them
- **Quantization:** mapping from continuous reals numbers to a discrete set
- Quantization determines the energy required to train a DNN model
  - FP16 encoding is **4** times more power efficient than FP32 encoding
  - INT8 encoding is **18** times more power efficient than FP32 encoding
- **DNNs are resilient to noise, enabling more power-efficient quantization schemes**

Conventional Encodings

- ML hardware/frameworks traditionally used **FP32 (single-precision)** encoding:
  - Sufficiently accurate representation
  - Inefficient hardware in terms of power and area
  - High memory requirements



- FP16: half precision



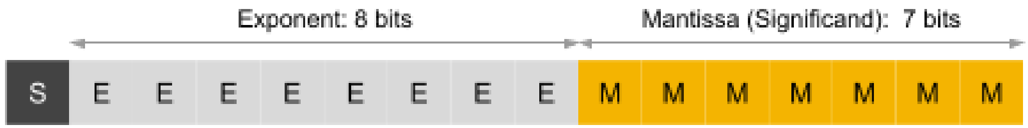
- Integer: int8, int4, …

Conventional encodings are not tailored towards ML applications.

ML-specific Encodings

PARALLEL SYSTEMS ARCHITECTURE LAB

- bfloat16: Brain Floating Point (*Google*)



- Cfloat8 (*Tesla*)

Format	Sign bit?	No. of Mantissa bits	No. of Exponent bits	Exponent Bias Value
CFloat8_1_4_3	Yes	1 + 3	4	Unsigned 6-bit integer
CFloat8_1_5_2	Yes	1 + 2	5	Unsigned 6-bit integer

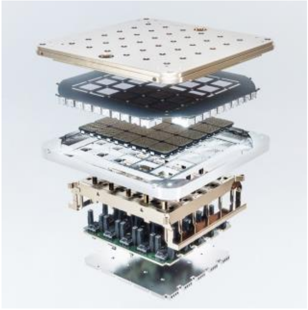
- Others, more complex (BFP, FlexPoint, MSFP)

A plethora of different encodings are introduced/being introduced!

Where to apply quantization?

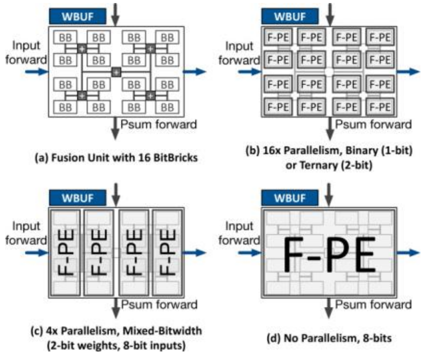
- Custom hardware platforms require the **entire** DNN in a certain encoding:

TPUv1 (int8)



Dojo Chip (Cfloat8)

NVDIA A100 (bfloat16)



BitFusion (2b, 4b, 6b, 8b)

Arduino (int8)

Where to apply quantization?

- Various **parts of the DNN** algorithm can use custom encodings

- Weights-only:
  - BinaryConnect (Matthieu Courbariaux, 2016)

- Activations-only:
  - BitWise Information Bottleneck (Xichuan Zhou, 2020)

- Layer-wise:
  - Adaptive Quantization for DNNs (Yiren Zhou, 2017)

Where to apply quantization?

PARALLEL SYSTEMS ARCHITECTURE LAB

- Distributed learning scales ML algorithms by partitioning computation
- Data-parallel distributed ML requires accumulating **gradients** across workers
- Communication is a bottleneck

AWS EC2 Instance Type	GPU Model	GPU-to-GPU Link	GPU-to-GPU Bandwidth	Network Bandwidth between Instances	Ratio
P4 p4d.24xlarge	NVIDIA A100	NVSwitch	600 GB/s	400 Gbps	12
P3 p3.16xlarge	NVIDIA V100	NVLink	50 GB/s	100 Gbps	4

- Sharing quantized gradients is a way of reducing communication overhead:
  - TernGrad (Wei Wen , 2017) encodes gradients as {-1, 0, 1}
  - *1-bit SGD* (Frank Seide, 2014) encodes gradients as {0, 1}

Problem and Goal

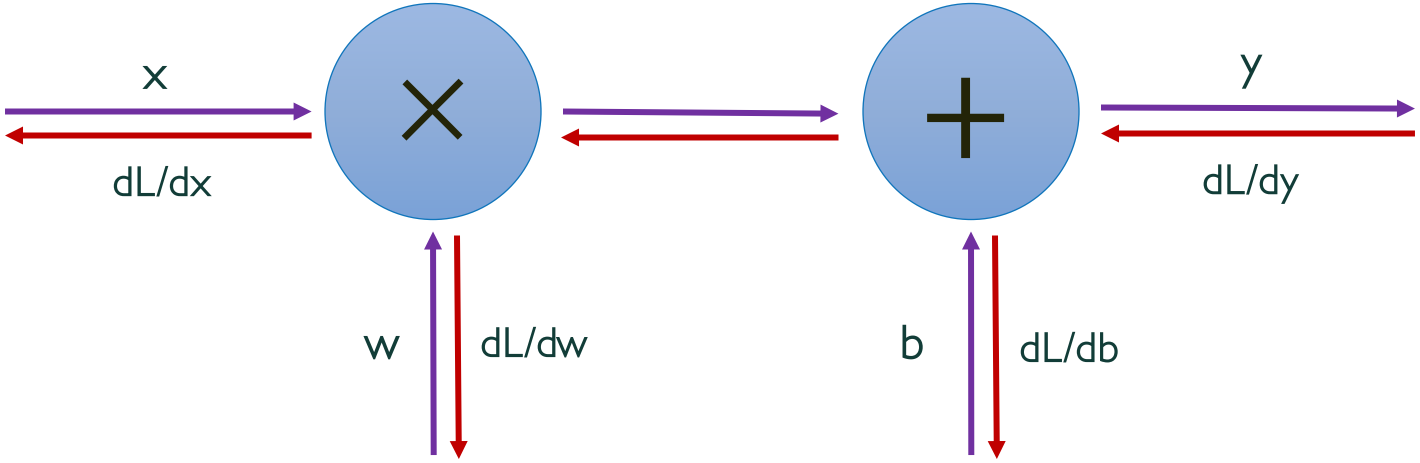
- There is a lack of an ML framework emulating various quantization schemes

- In this project, we aim to develop such a framework, which is:
  - Universally applicable to any quantization scheme
  - Accessible with a clean and user-friendly API
  - Customizable for various parts of DNN processing

- **Q**uantized **M**achine **L**earning framework (**QML**):
  - Based on PyTorch
  - Provides abstractions for describing quantization
  - Applicable to linear and convolutional layers

Overview

- A simple fully-connected layer:



- Can represent linear or 2d convolutional layers

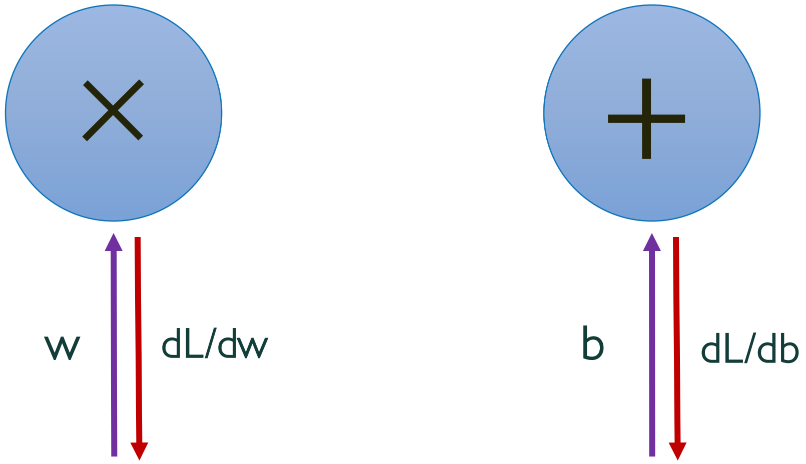
Overview: Activation

- Activations (input/output) can be quantized, along with their gradients

- The following quantizer naming convention is used:
  - **qx** for input and **qdx** for its gradient
  - **qy** for input and **qdy** for its gradient

Overview: Parameter

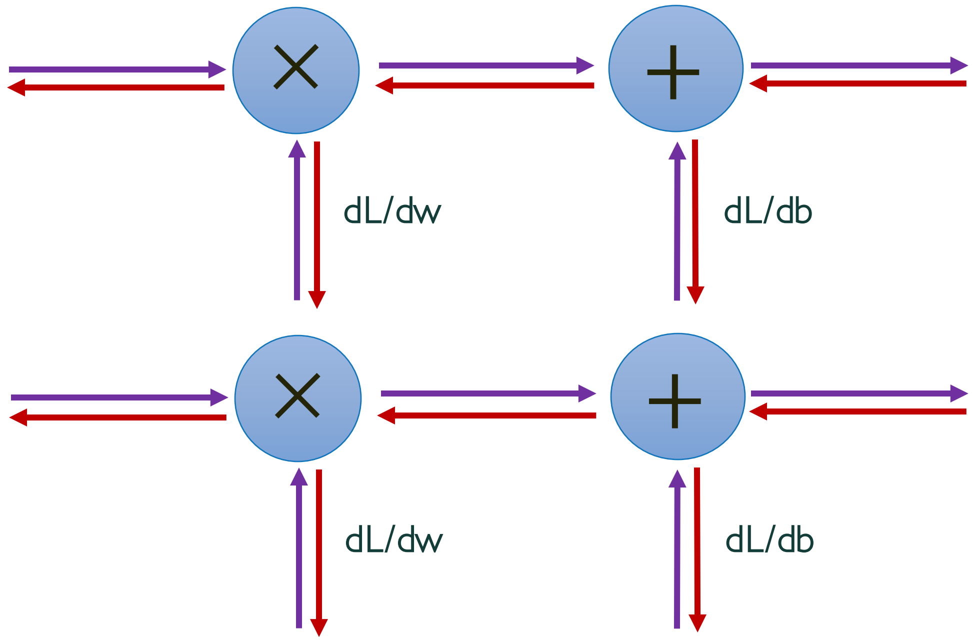
- Parameters (weight/bias) can be quantized, along with their gradients



- The following quantizer naming convention is used:
  - **qw** for weight and **qdw** for its gradient
  - **qb** for bias and **qdb** for its gradient

Overview: Aggregation

- Distributed learning is emulated with quantization, processing is done on a single machine
- Parameters gradient (weight/bias) can be accumulated using custom accumulators



- The following accumulator naming convention is used:
  - aw for weight
  - ab for bias

Workflow

QML provides a simple 4-step workflow:

- Implement a quantizer
- Implement an accumulator
- Configure the framework
- Execute training

Quantizers

Quantizers inherit from Quantizer and redefine 2 methods:

```
class Quantizer:

    def quantize(self, t: Tensor) -> Tuple[Tensor, object]:
        """
        returns a quantized version of the input tensor and a context
        """
        raise NotImplementedError

    def dequantize(self, t: Tensor, ctx: object) -> Tensor:
        """
        returns the original tensor from its quantized version using the context
        """
        raise NotImplementedError

    ...
```

Quantizers: Floating Point

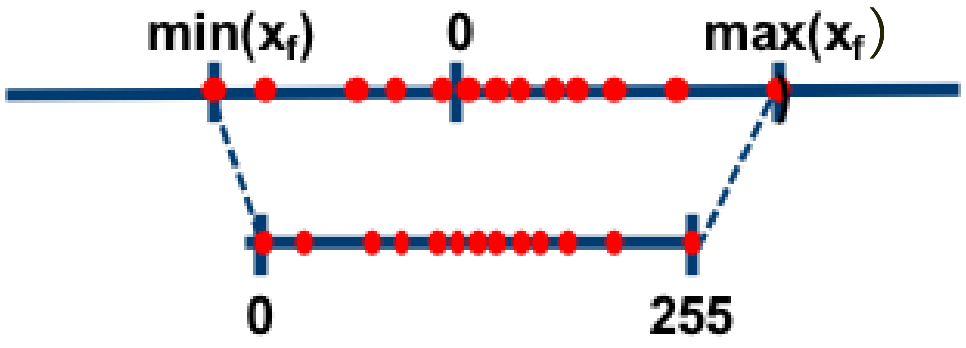
PARALLEL SYSTEMS ARCHITECTURE LAB

Decreases the number of mantissa bits, keeping the same number of exponent bits

```
class FloatingPointQuantizer(Quantizer):
    def __init__(self, m: int) -> None:
        super().__init__()
        self._m = m
        self._2m = math.pow(2.0, m)
        self._2mm = math.pow(2.0, -m)

    def quantize(self, t: Tensor) -> Tuple[Tensor, object]:
        m, e = t.frexp()
        m = m * 2  # fp convention
        e = e - 1  # re align
        sign = m.sign()
        m = sign * m
        q: Tensor = (m * self._2m).floor() * self._2mm
        return (sign * q * torch.pow(2.0, e), None)

    def dequantize(self, t: Tensor, ctx: object) -> Tensor:
        return t
```

Quantizers: Integer

Maps floating point numbers between a minimum and a maximum to integers




```python
class IntQuantizer(Quantizer):
    def __init__(self, m: int) -> None:
        super().__init__()
        self._m = m

    def quantize(self, t: Tensor) -> Tuple[Tensor, object]:
        shape = t.size()
        t_f = t.view(shape[0], -1)

        tmin = torch.min(t_f, dim=1)[0].view(-1, 1)
        tmax = torch.max(t_f, dim=1)[0].view(-1, 1)

        result = (t_f - tmin) / (tmax - tmin) * ((1 << self._m) - 1)
        return result.round().view(shape), (tmin, tmax)

    def dequantize(self, t: Tensor, ctx: object) -> Tensor:
        tmin, tmax = ctx

        shape = t.size()
        t_f = t.view(shape[0], -1)

        result = t_f / ((1 << self._m) - 1) * (tmax - tmin) + tmin

        return result.view(shape)
```

Quantizers: Tiled

PARALLEL SYSTEMS ARCHITECTURE LAB

Apply a given quantizer to each tile of the input tensor

Ex: 2x2

$$\begin{pmatrix} \boxed{\begin{matrix} a & b \\ e & f \end{matrix}} & \begin{matrix} c & d \\ g & h \end{matrix} \\ \begin{matrix} i & j \\ m & n \end{matrix} & \begin{matrix} k & l \\ o & p \end{matrix} \end{pmatrix} \to \left(\begin{pmatrix} a & b \\ e & f \end{pmatrix} \begin{pmatrix} c & d \\ g & h \end{pmatrix} \begin{pmatrix} i & j \\ m & n \end{pmatrix} \begin{pmatrix} k & l \\ o & p \end{pmatrix} \right)$$

```python
class TiledQuantizer(Quantizer):
    def __init__(self, dims: List[int], q: Quantizer) -> None:
        super().__init__()
        self._dims = dims
        self._q = q

    def quantize(self, t: Tensor) -> Tuple[Tensor, object]:
        original_shape = t.size()
        tt = t.view(-1, *self._dims, t.size()[2:])

        result, ctx = self._q.quantize(tt)

        return result.view(original_shape), ctx

    def dequantize(self, t: Tensor, ctx: object) -> Tensor:
        original_shape = t.size()
        tt = t.view(-1, *self._dims, t.size()[2:])

        result = self._q.dequantize(tt)

        return result.view(original_shape)
```

Accumulators

Accumulators inherit from GradientAccumulators and redefine 1 method:

```
class GradientAccumulator:

    def accumulate(self, t: Tensor) -> Tensor:
    """
    Takes a gradient tensor of shape (N, *) and accumulates it over the first
    dimension, returning a tensor of shape (1, *)
    """
    raise NotImplementedError

    ...
```

Accumulators: Share Quantized Gradients

- HsumAccumulator: used to emulate reduced precision gradient sharing

- 3 steps:
  - Sum gradients for each worker, - i.e compute local gradient
  - Apply quantization to each local gradient
  - Sums the results to compute global gradient

- Takes a list of tuple, defining the aggregation scheme:

  [
  (samples_per_worker, id_quantizer),
  (nb_workers, **your_quantizer**)
  ]

Configure the Framework

PARALLEL SYSTEMS ARCHITECTURE LAB

```
class CNN_LeNet(Module):

    def __init__(self):
        super(CNN_LeNet, self).__init__()

        self.conv_pool_stack = Sequential(
            Conv2d(1, 6, 3, 1, padding=1),
            MaxPool2d(2),
            Conv2d(6, 16, 3, 1, padding=1),
            MaxPool2d(2)
        )

        self.linear_relu_stack = Sequential(
            Linear(784, 120),
            ReLU(),
            Linear(120, 84),
            ReLU(),
            Linear(84, 10),
        )

    def forward(self, x):
        x = self.conv_pool_stack(x)
        x = x.reshape((x.shape[0], -1))
        return self.linear_relu_stack(x)
```

```
class CNN_LeNet(Module):

    def __init__(self):
        super(CNN_LeNet, self).__init__()

        self.conv_pool_stack = Sequential(
            QConv2d(..., quantizerBundle, accumulatorBundle),
            MaxPool2d(2),
            QConv2d(..., quantizerBundle, accumulatorBundle),
            MaxPool2d(2)
        )

        self.linear_relu_stack = Sequential(
            QLinear(..., quantizerBundle, accumulatorBundle),
            ReLU(),
            QLinear(..., quantizerBundle, accumulatorBundle),
            ReLU(),
            QLinear(..., quantizerBundle, accumulatorBundle),
        )

    def forward(self, x):
        x = self.conv_pool_stack(x)
        x = x.reshape((x.shape[0], -1))
        return self.linear_relu_stack(x)
```

Configure the Framework

- These additional parameters are objects carrying the quantizers and accumulators that are used while training

- They are filled in following way:

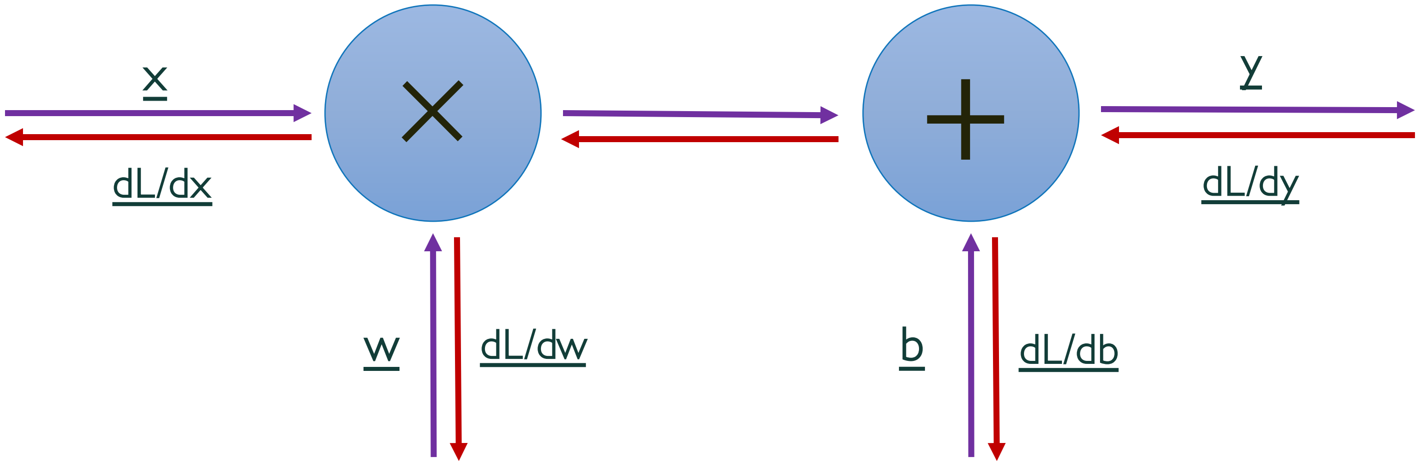
```
quantizerBundle = PrefixBundle(default = quantizer)
accumulatorBundle = PrefixBundle(default = accumulator)

quantizerBundle.qx = quantizer1   # to quantize activations
accumulatorBundle.aw = accumulator1 # to accumulate weights
```

Framework Results

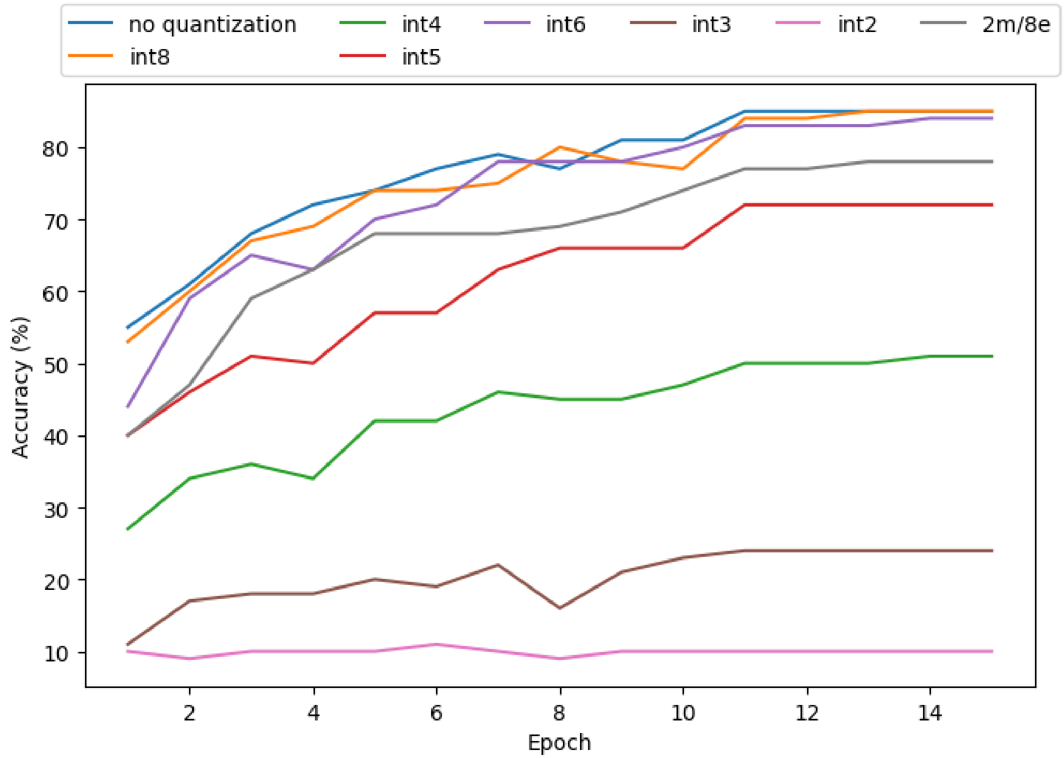
Quantization: Scheme

PARALLEL SYSTEMS ARCHITECTURE LAB

- All **relevant tensors** are quantized using a custom encoding



Quantization: Results



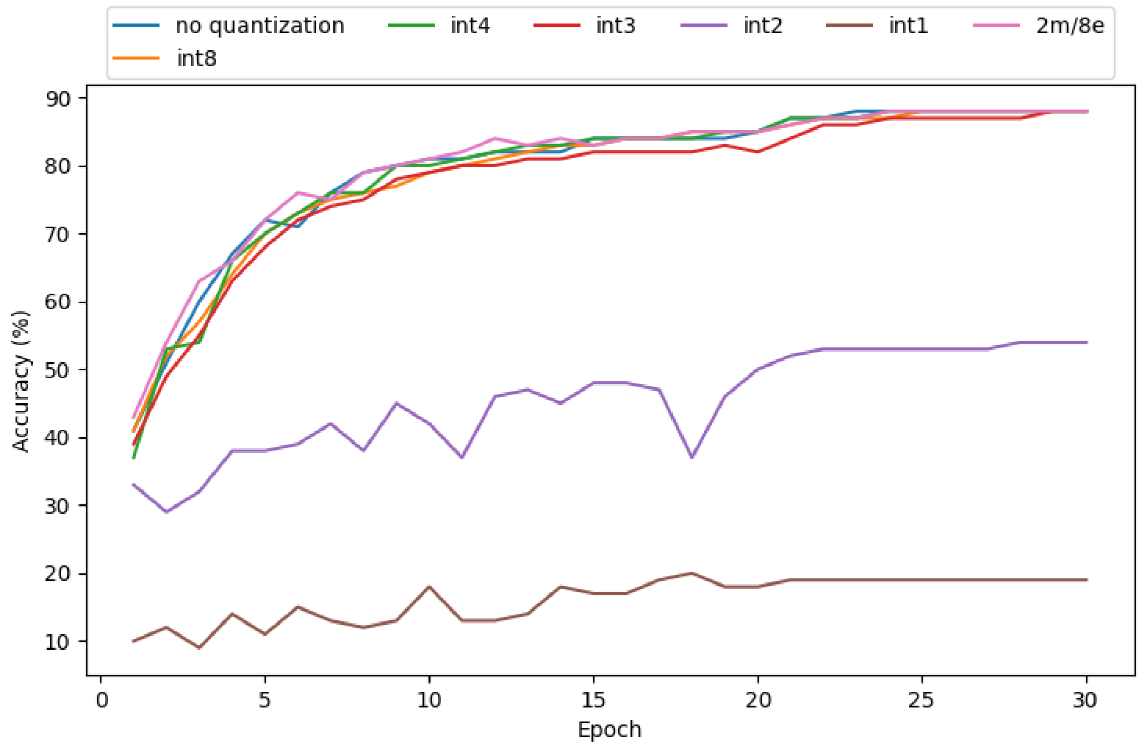
Quantization	10 Epochs	15 Epochs
None	81	85
2 mantissa/8 exponent	74	78
Int8	77	85
Int6	80	84
Int5	66	72
Int4	47	51
Int3	23	24
Int2	10	10

Quantized Resnet18's accuracy on CIFAR10

Aggregation: Setup

- Global mini-batch size: 512 samples
- Number of workers: 8
- Each worker processing 64 samples
- All workers do FP32 arithmetic
- Gradients are exchanged in a customized encoding

Aggregation: Results



Quantization	10 epochs	20 epochs	30 epochs
None	80	85	88
2 mantissa/8 exponent	81	83	88
Int8	79	85	88
Int6	79	84	88
Int5	79	85	88
Int4	80	85	88
Int3	79	84	88
Int2	44	52	54
Int1	16	18	20

Reduced gradient precision Resnet18's accuracy on CIFAR10

Framework Overhead

Quantization	Execution time (minutes)
None	24
FP	42
Int	45

Accumulation	Execution time (minutes)
None	53
FP	63
Int	64

- Execution time increases as overhead is added by quantization/aggregation calculations
- The overhead is proportional to the quantizer/accumulators complexity and to the number of quantized datapaths

Conclusion: Limitations & Future Work

PARALLEL SYSTEMS ARCHITECTURE LAB

- Quantization emulated, nothing tested on dedicated hardware
- Distributed learning emulated, other problems might happen
- Just for DNNs composed of linear and 2D convolutional layers
- Extend to more types of layers
- Extend framework with new quantizers and accumulators

Thank You!

For more information:

Mathias Alexandre Bouilloud: mathias.bouilloud@epfl.ch

https://github.com/parsa-epfl/q7d-nn

Thanks for listening!