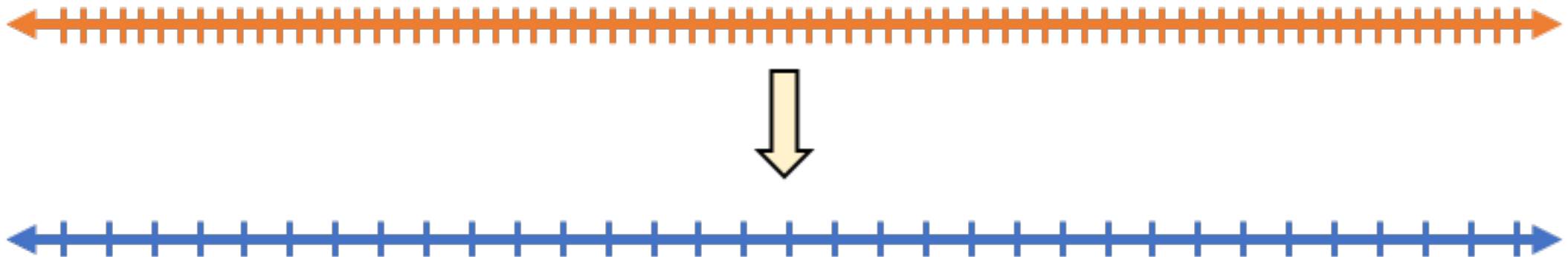


DL Optimizations for IoT Devices

Quantization

Quantization

- Standard DL training and inference are performed with 32-bit float weights and activations.
- Both tasks, but especially inference, can however be performed at reduced precision (i.e. *quantizing* data) with very little accuracy loss
 - Remember, deep learning models are error tolerant!



Quantization

- Quantization can be applied either to **weights only** or to **weights + activations**
 - Using smaller float formats (e.g. 16-bit mini-float)
 - Using low-precision integers (can also be seen as fixed-point)
 - Using custom data formats.

Benefits of Quantization

- The benefits of quantization are two-fold
 1. As we have seen in a previous lecture, data transfers to and from memory are fundamental contributors to the total execution time and energy consumption of DL models. For example, going from 32-bit to 8-bit data can theoretically **reduce both the time and energy of memory transfers** by a factor of 4.
 2. If the underlying HW platform supports it, using low precision data can also **improve the efficiency of the computation phase**, since arithmetic operations can be executed on smaller, faster and more efficient hardware.

Quantization - Edge

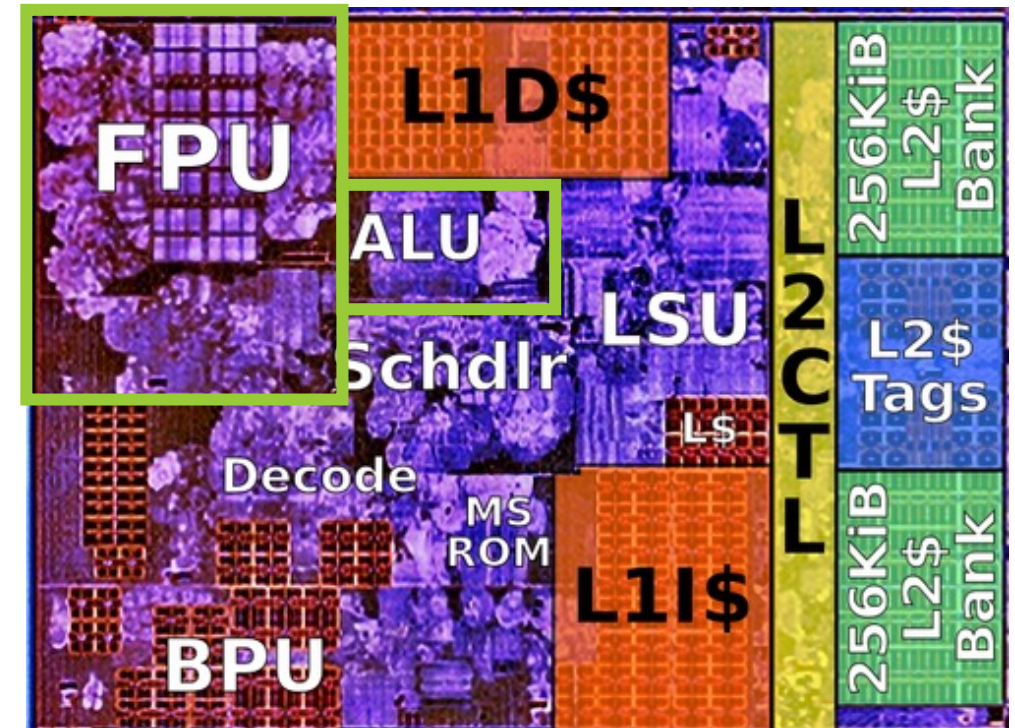
- To deploy DL models on IoT edge devices, a popular quantized data format is **8-bit integer** (fixed point).

- **Why integer?**

- ALUs are smaller and more efficient than FPUs
- Many microcontrollers don't even have a FPU

(*) In fact, this is not a microcontroller's CPU.

Example of a CPU: AMD Zen (*)

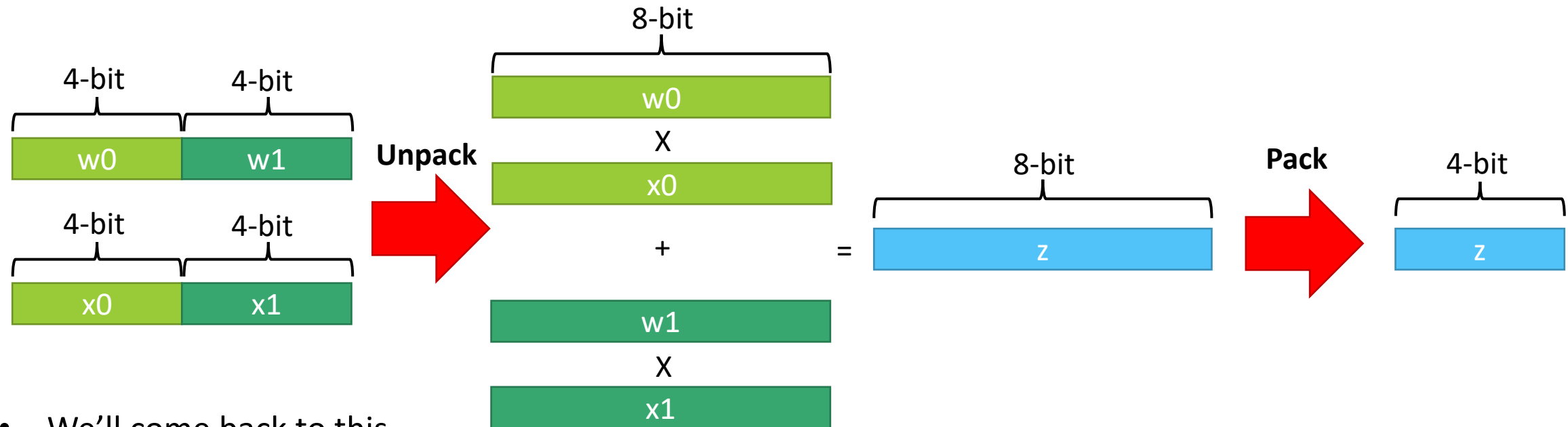


Quantization - Edge

- To deploy DL models on IoT edge devices, a popular quantized data format is **8-bit integer** (fixed point).
- **Why 8-bit?**
 - Most CPUs have 1 byte as the **minimum unit of processing**. Instruction only work on single or multiple bytes, not sub-bytes.
 - RAM memories are **byte-addressed**
- Also, 8-bit quantization is (empirically) a “*sweet spot*” in terms of accuracy for many complex tasks, e.g. image classification.

Quantization - Edge

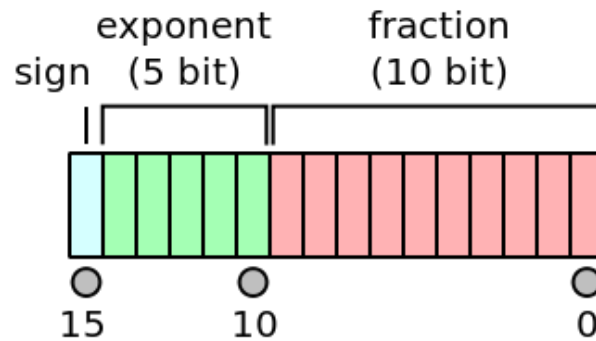
- Using less than 8-bits (and sometimes even less than 16) requires:
 - Custom hardware support or...
 - ...complex packing/unpacking of data



- We'll come back to this...

Quantization - Cloud

- Quantization in the cloud is often done using a **16-bit minifloat** format (i.e. half precision):



- The goal in this case is to increase execution speed while maintaining high accuracy. Energy is relevant, but somewhat secondary.
- Minifloats are **more complex** to process than integers, but can represent a higher **dynamic range** for the same bit-width, and therefore typically result in **higher accuracy**.

Quantization Formats - Summary

- **16-bit minifloats** (aka half-precision floats) basically come for free:
 - Can be used for both training and inference with no drop in accuracy in most cases
 - But they're still floats
 - Increasingly adopted for training/inference in the cloud. Less common at the edge
- **Int-8** is the current standard at the edge:
 - Drop in accuracy often minimal (especially with Quantization-Aware Training).
 - Most ISAs have byte-level instructions (although not always byte-level SIMD arithmetic)
- **Sub-byte** integer quantization is attracting attention:
 - 1-bit quantization (BNN) is already popular for simple tasks
 - 4-bit is increasingly used, despite its packing/unpacking overheads.

Benefits of Quantization - Summary

- Storage and memory:
 - Reduce ROM/disk occupation
 - Reduce RAM occupation
 - Increase data-transfers bandwidth (e.g., with SIMD load/store)
- Compute:
 - Speed-up thanks to ALU vs FPU
 - Further speed-up thanks to SIMD operations
 - Enable deployment on FPU-less Microcontrollers

Basic Integer Quantization

Integer Quantization

- In general, there are many ways to represent real numbers using an integer.
 - No universal standard as for floats
 - We'll only scratch the surface here...
- At the basis of the most common integer quantization approaches is a format sometimes called **dynamic fixed point**.

Dynamic Fixed Point

- **Fixed point:** all elements of a tensor share the same “exponent” or scaling factor (Δ). Each integer represents a multiple of the scaling factor

- Example:

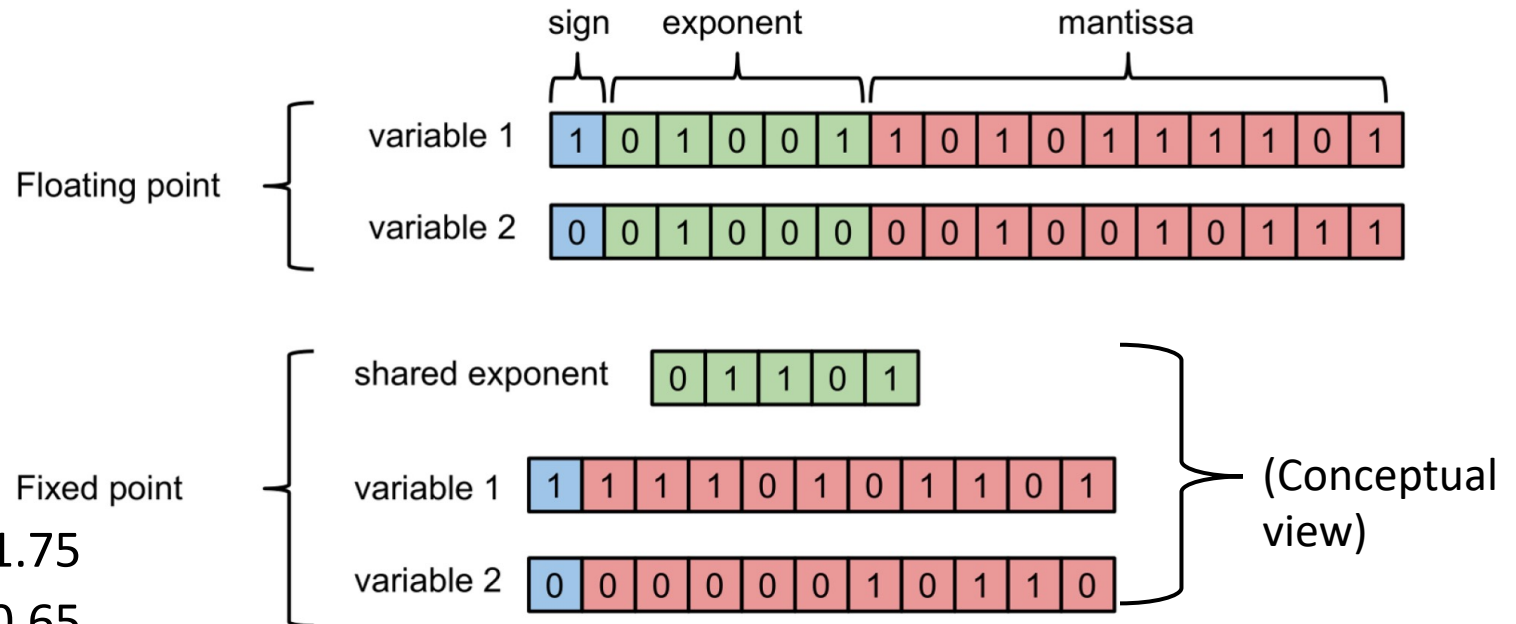
- $\Delta = 0.05$

- $X1(\text{int representation}) = 35$

- $X2(\text{int representation}) = 13$

- $X1(\text{actual value}) = 35 * 0.05 = 1.75$

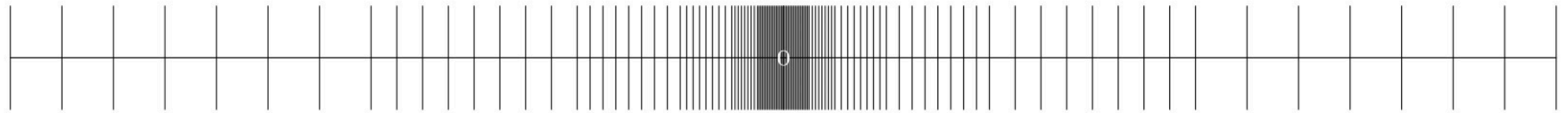
- $X2(\text{actual value}) = 13 * 0.05 = 0.65$



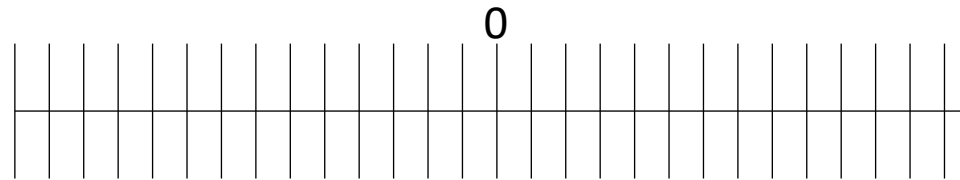
Dynamic Fixed Point

- Differently from floats, fixed point numbers have a uniform density in the entire range, which (for a given bit-width) is determined by Δ . The bit-width determines the number of “ticks” in the range.

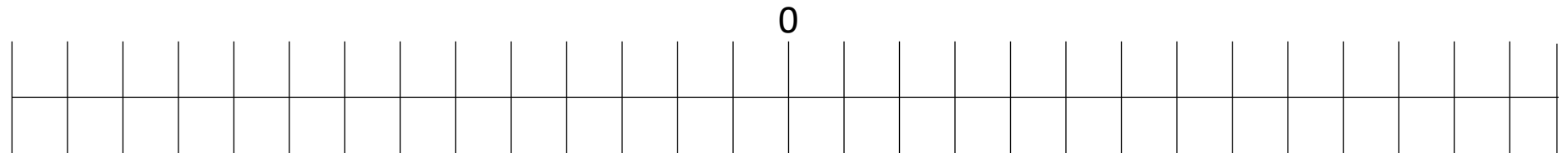
Float



Fixed (Δ_1)

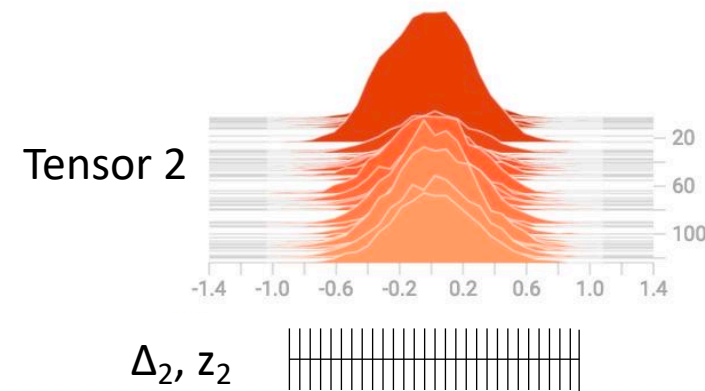
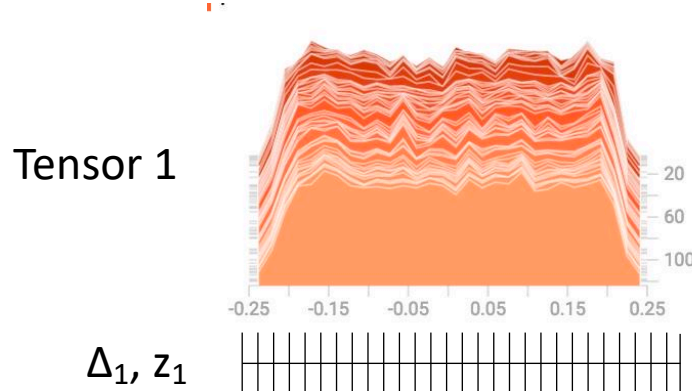


Fixed ($\Delta_2 > \Delta_1$)



Dynamic Fixed Point

- Why “dynamic” fixed point? Because we use a different set of quantization parameters **for each tensor** (or even for each channel of the same tensor)
 - Quantization parameters = scaling factor (Δ) and zero offset (see later)
- Example: all weights of a given Dense layer share the same parameters, but different layers can use different parameters
 - So that we don't lose too much precision when different layers have different weights (or activations) distributions.



Affine Quantizer

- Asymmetric integer quantization, to deal with non-centered distributions of activations (e.g. think about ReLU). It is the format used by TFLite (see [here](#)).

$$real_value = (int8_value - z) * \Delta \qquad int8_value = \frac{real_value}{\Delta} + z$$

- The int8_value is in two's complement
- Δ and z are set per-tensor or per-channel (for weights).
- In particular z is always 0 for weights, and ranges in $[-128, 127]$ for activations.

Affine Quantizer

- With the affine quantizer, 0.0 (*real_value*) is always represented exactly as *z*

$$\textit{int8_value} = \frac{\textit{real_value}}{\Delta} + z$$

- Important because 0.0 in NNs has a special meaning (e.g. padding, “dead neuron”, etc).
 - Accuracy worsens significantly if 0 cannot be represented.

Affine Quantizer

- Why is z always 0 for weights?
 - Weights distributions tend to be more symmetric compared to activations.
 - But most importantly, z can be heavily optimized for activations, not for weights...
- Case 1: both asymmetric

$$\begin{aligned}w_{i,real} &= (w_{i,int} - z_w) * \Delta_w, & x_{i,real} &= (x_{i,int} - z_x) * \Delta_x \\out_{real} &= f\left(\sum^N (w_{i,real} * x_{i,real})\right) = f\left(\sum^N ((w_{i,int} - z_w) * \Delta_w * (x_{i,int} - z_x) * \Delta_x)\right) = \\&= f\left(\Delta_w \Delta_x \left(\sum^N (w_{i,int} * x_{i,int} - w_{i,int} * z_x - \underbrace{x_{i,int} * z_w}_{\text{Requires an additional MUL for each activation Since xi is variable!}}) + N * z_x * z_w)\right)\right)\end{aligned}$$

Affine Quantizer

- Case 1: symmetric weights

$$w_{i,real} = w_{i,int} * \Delta_w, \quad x_{i,real} = (x_{i,int} - z_x) * \Delta_x$$

$$out_{real} = f\left(\sum^N (w_{i,real} * x_{i,real})\right) = f\left(\sum^N (w_{i,int} * \Delta_w * (x_{i,int} - z_x) * \Delta_x)\right) =$$

$$= f\left(\Delta_w \Delta_x \sum^N (w_{i,int} * x_{i,int} - w_{i,int} * z_x)\right)$$

Weights are constant at runtime.
This entire summation can be pre-computed.
No additional MUL required, just a **single** subtraction

$$= f\left(\Delta_w \Delta_x \sum^N (w_{i,int} * x_{i,int}) - \Delta_w \Delta_x \sum^n (w_{i,int} * z_x)\right)$$

Integer Quantization

- How to determine Δ and z for a given tensor (**a**)?
- The basic way is to make sure the entire range (a_{\min} , a_{\max}) is correctly mapped, where a_{\min} and a_{\max} are the minimum and maximum (float) values that the tensor can have. This corresponds to:

Activations	$\Delta = \frac{a_{\max} - a_{\min}}{2^{Nbit} - 1}$	$z = \frac{a_{\max} + a_{\min}}{2}$
Weights	$\Delta = \frac{2 * \max(a_{\max} , a_{\min})}{2^{Nbit} - 1}$	$z = 0$

-1 because TFLite prefers to use symmetric two's complement values, e.g. [-127, 127] for 8bit

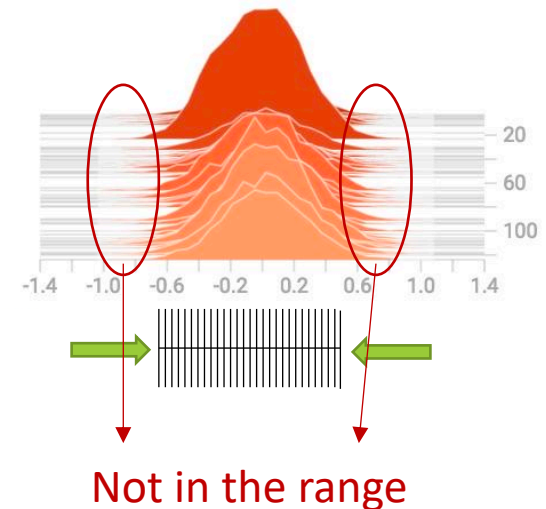
Integer Quantization

- How to determine Δ and z for a given tensor (**a**)?
- Some tools prefer to use a smaller Δ , thus worsening the approximation of some outlier weights, in exchange for more precision around z :
 - Can sometimes yield higher accuracy (empirical).

- In this case, the quantization procedure has two steps:

$$tmp = \frac{real_value}{\Delta} + z$$

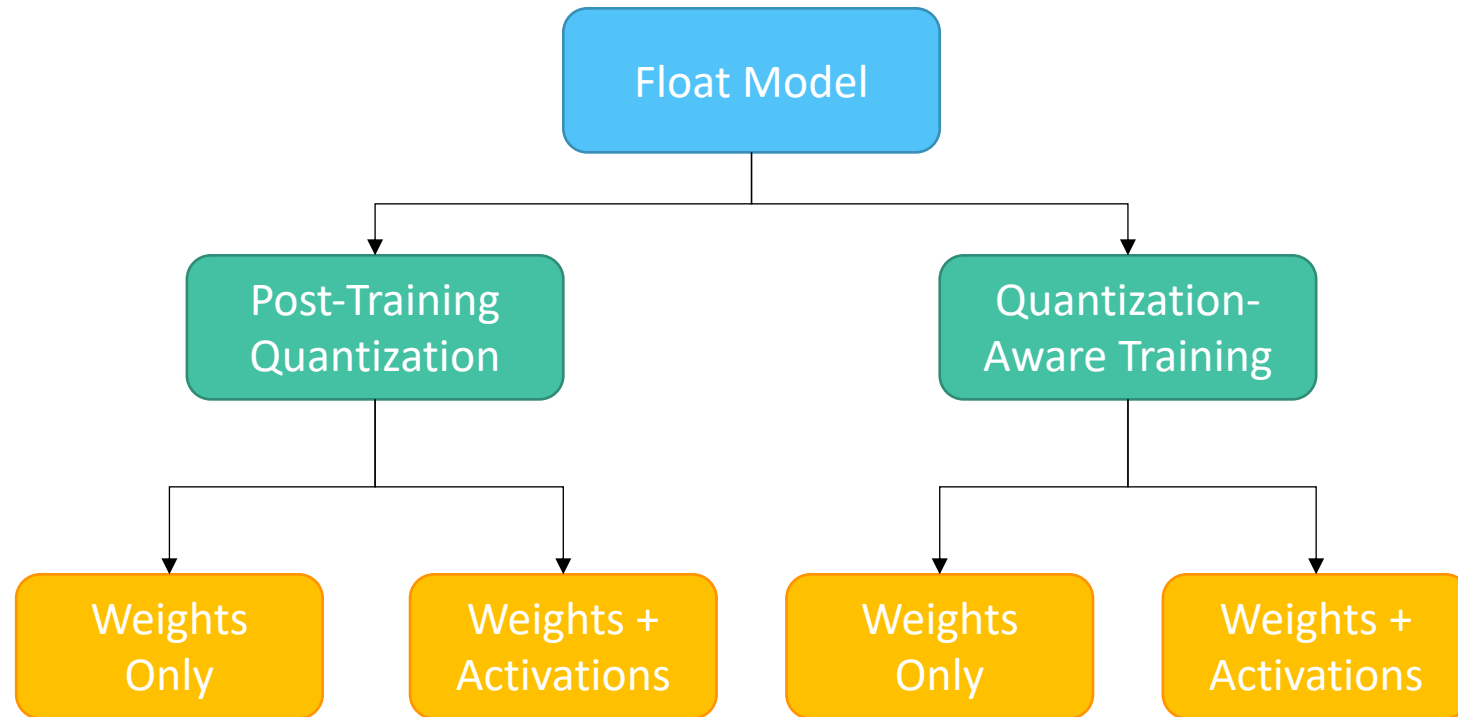
$$int8_{value} = clamp(tmp, -127, 127) = \min(\max(-127, tmp), 127)$$



Quantization Strategies

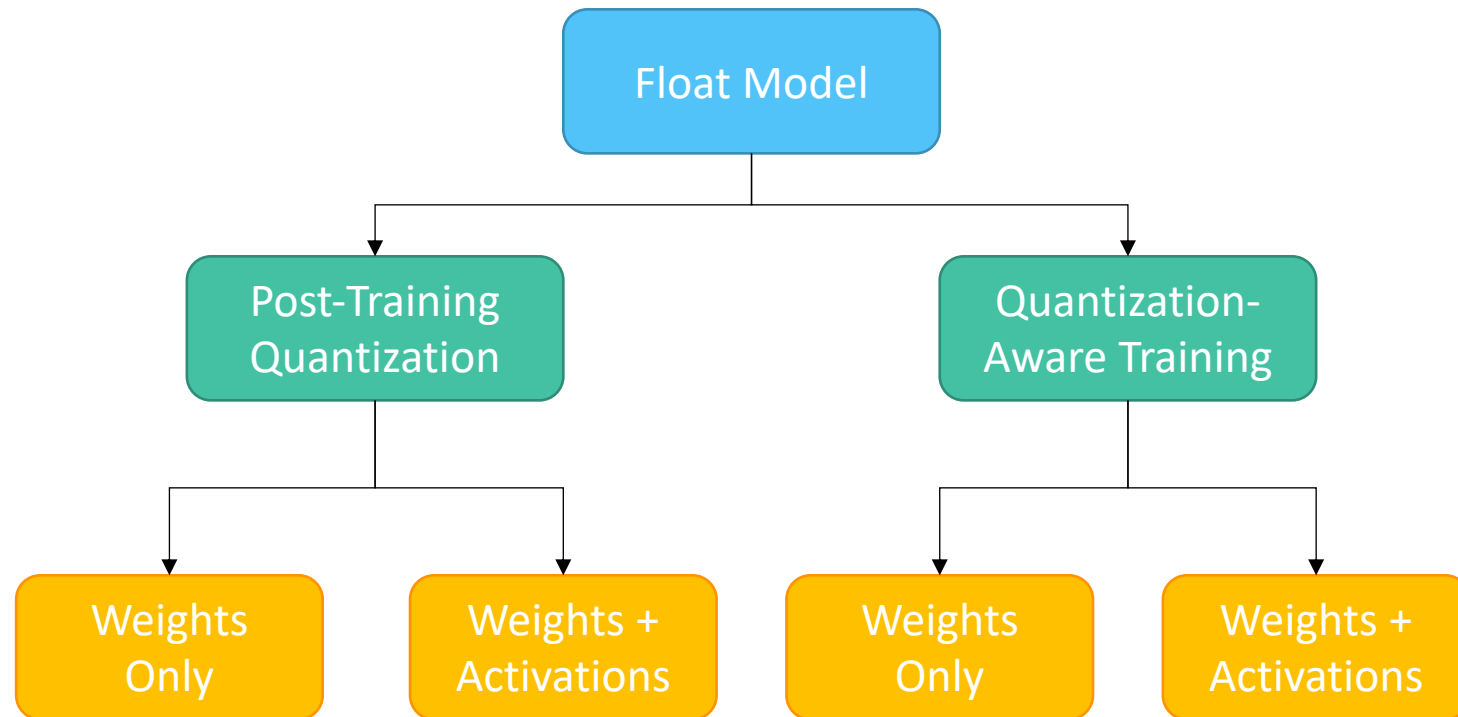
Quantization: Strategies

- Quantization can be applied either post-training or at training time, to weights only or weights + activations.



Quantization: Strategies

- Quantization can be applied either post-training or at training time, to weights only or weights + activations.

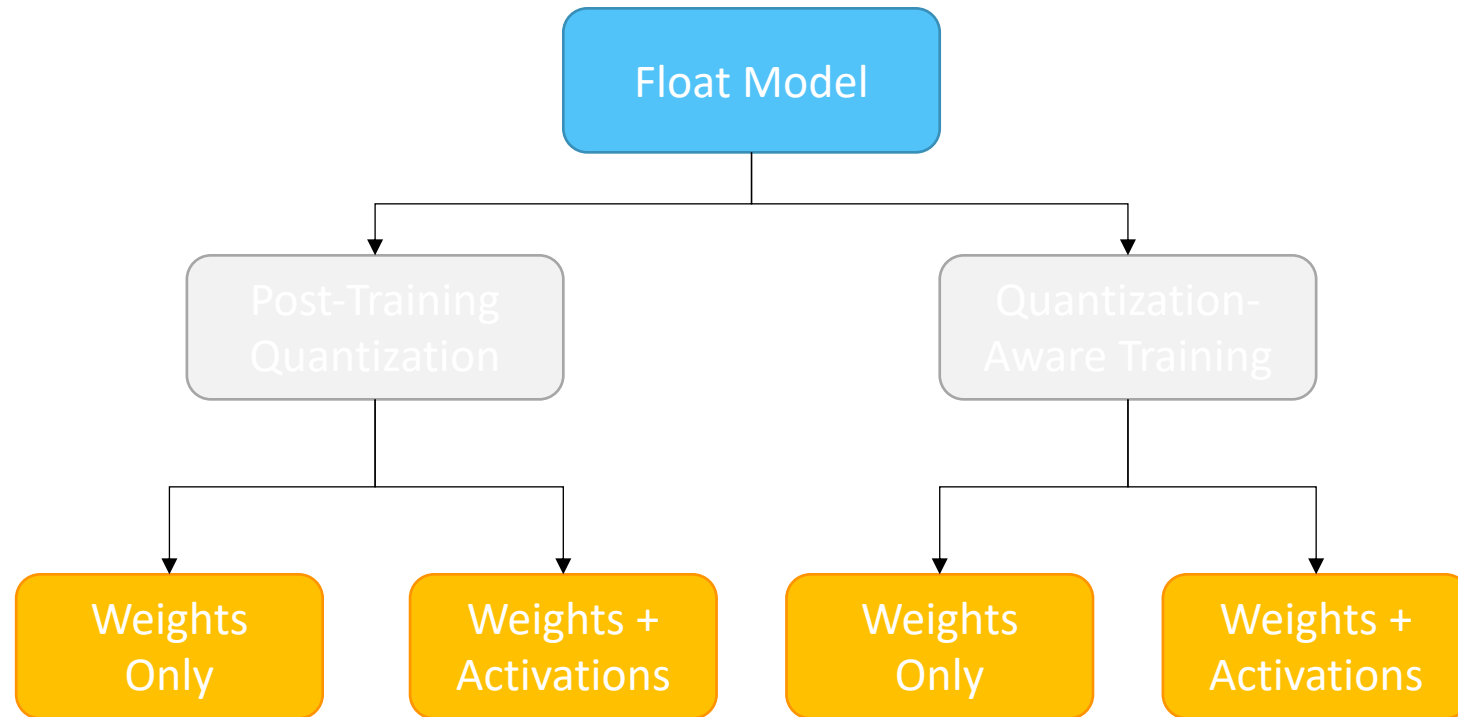


WHEN/HOW

WHAT

Quantization: Strategies

- Quantization can be applied either post-training or at training time, to weights only or weights + activations.

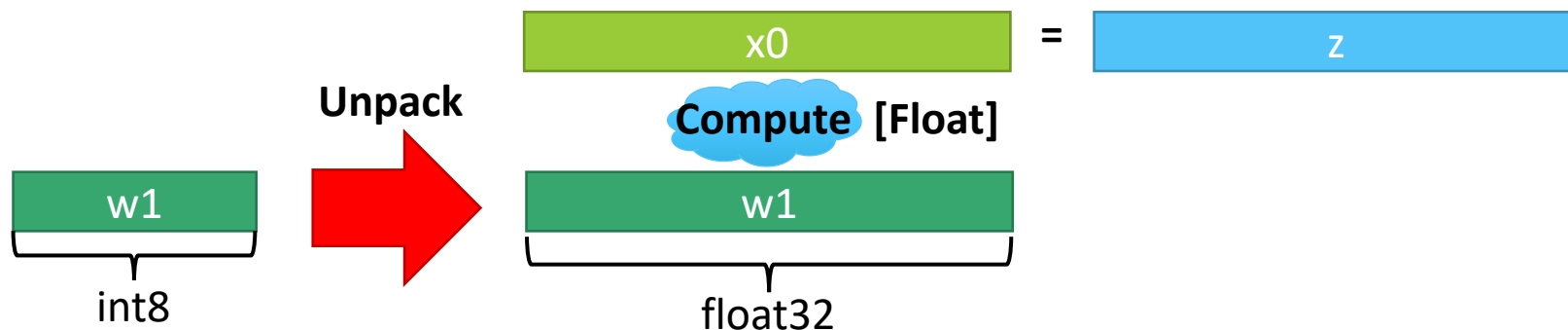


Quantization (Weights Only)

- Weights are converted to integers (or equivalently, fixed point numbers). Currently, TFLite supports quantization to **8-bit integers** only (16-bits are supported by the experimental package).
- The simplest approach only quantizes the weights. Inputs/outputs and activations are still stored in float32 format

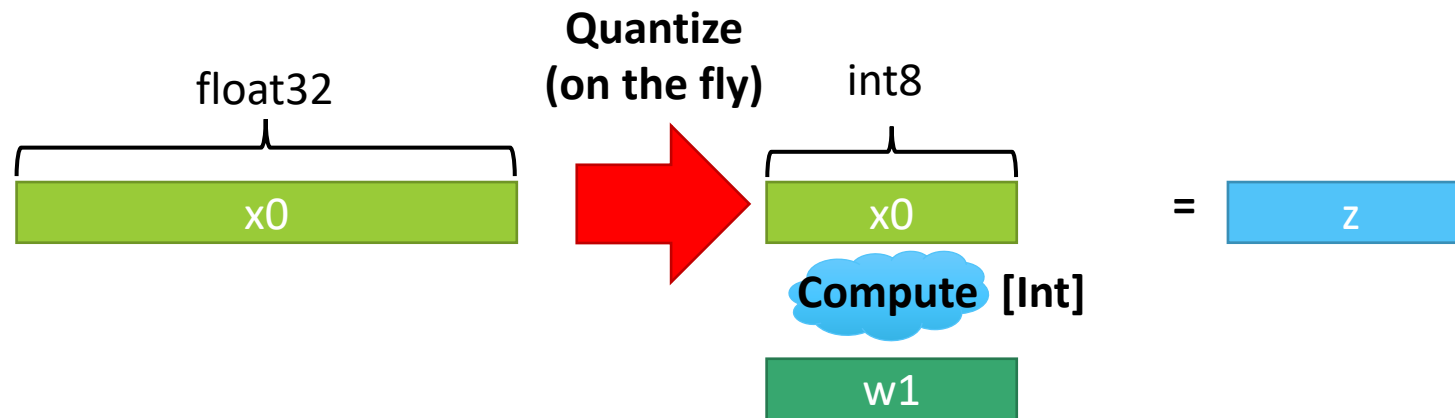
Quantization (Weights Only)

- **Basic version**: during inference, weights are **unpacked from int8 to float32** and operations are performed in float.
 - 4x model size reduction on disk! (32 → 8)
 - **No advantage in terms of working memory or latency!!**
- Once loaded in memory, weights are unpacked to 32bits and the converted versions are then cached to reduce latency.
 - First inference is surely slower than float
 - Following inferences require the same working memory as the float model.



Quantization (Weights Only)

- Advanced version (called “dynamic range quantization”): only available for some ops. **Weights are kept in 8-bit format**, and activations (always stored as floats) are **quantized on-the-fly** so that operations can be performed with integer HW.



- Reduces the working memory (no unpacking). May be slower or faster depending on the underlying HW

Quantization (Weights Only)

- To perform weights-only quantization in TFLite, we just need to request the “default optimizations” to the converter:

```
converter = tf.lite.TFLiteConverter.from_saved_model("my_model")
```



```
converter.optimizations = [tf.lite.Optimize.DEFAULT]
```

```
quant_model = converter.convert()
```

Quantization (Weights + Activations)

- In this case, we want to use only 8-bit integers for both **weights and activations**, and for both **storage and computation**.
- **Problem:** we are not quantizing “on the fly”, so we need to fix Δ and z for the activation tensors.
 - w_{\min} and w_{\max} for each weights tensor can be extracted from the float model
 - x_{\min} and x_{\max} for activation tensors, instead, depend on the inputs!

Quantization (Weights + Activations)

- **Solution (for post-training quantization):** provide the converter with a set of **representative inputs** which will be used to compute x_{\min} and x_{\max} (and consequently Δ and z) for input/output and activation tensors.
 - Not needed with Quantization-Aware Training (see later)
- Typically, a small set of inputs (100—1000) is enough to correctly set the parameter.
- Outliers can be squashed within the range with *clamp()*

Quantization (Weights + Activations)

- We need to define a “generator function” for providing these representative data:

```
def representative_data_gen():  
    for input_value in train_ds.take(100):  
        yield [input_value]
```



The function returns but the next invocation restarts from the current state (sort of)

Quantization (Weights + Activations)

- And set it in the converter options:

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]
```



```
converter.representative_dataset = representative_data_gen
```

```
tflite_model_quant = converter.convert()
```

Quantization (Weights + Activations)

- With the conversion instructions of the previous slide, computations will be in int8 **only if TFLite includes an integer implementation** of that particular op. Unsupported ops will be executed in float.
- Moreover, inputs and outputs are still received/produced in float, for compatibility with applications that use the original float model.
- Ok for a high-end device (e.g. mobile), not for a microcontroller without FPU.

Quantization (Weights + Activations)

- Additional parameters in the converter can be used to force it to use only int8 ops, as well as int8 inputs and outputs (and generate an error otherwise).

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data_gen

# Ensure that if any ops can't be quantized, the converter throws an error
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]

# Set the input and output tensors to uint8
converter.inference_input_type = tf.uint8
converter.inference_output_type = tf.uint8

tflite_model_quant = converter.convert()
```

Quantization (Weights + Activations)

- The quantization parameters (Δ and z) for inputs and outputs can be extracted in the TFLite interpreter `input_details`:

```
input_details = interpreter.get_input_details()[0]
input_dtype = input_details["dtype"]
input_scale, input_zero_point = input_details["quantization"]
```

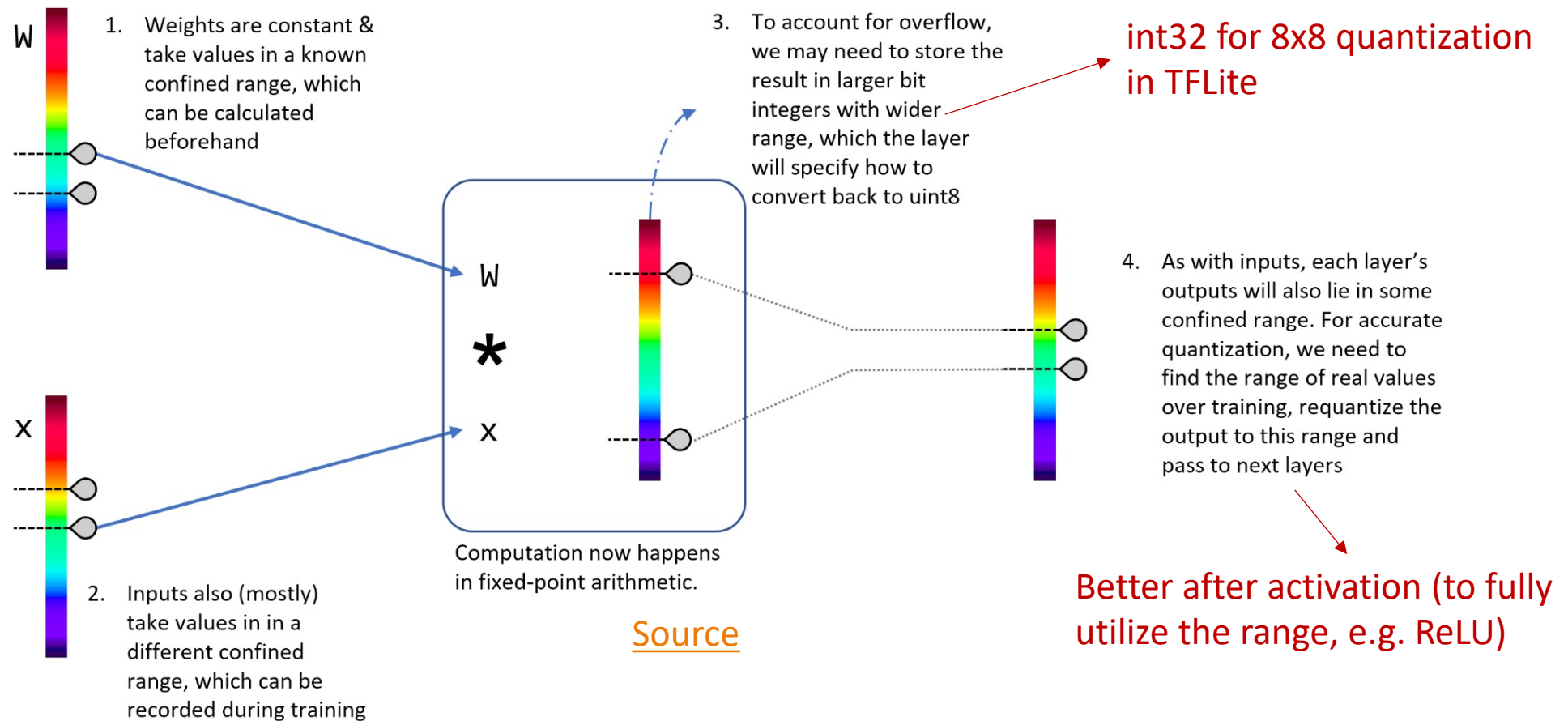
Quantization (16x8 Experimental)

- Tflite also supports an experimental integer-only quantization scheme in which weights are stored and manipulated as int8, whereas activations use int16.
- **Advantage:** increased accuracy with only a slightly increase in model size
 - Typically weights size >> activations size

```
converter.target_spec.supported_ops = [  
    tf.lite.OpsSet.EXPERIMENTAL_TFLITE_BUILTINS_ACTIVATIONS_INT16_WEIGHTS_INT8  
]
```

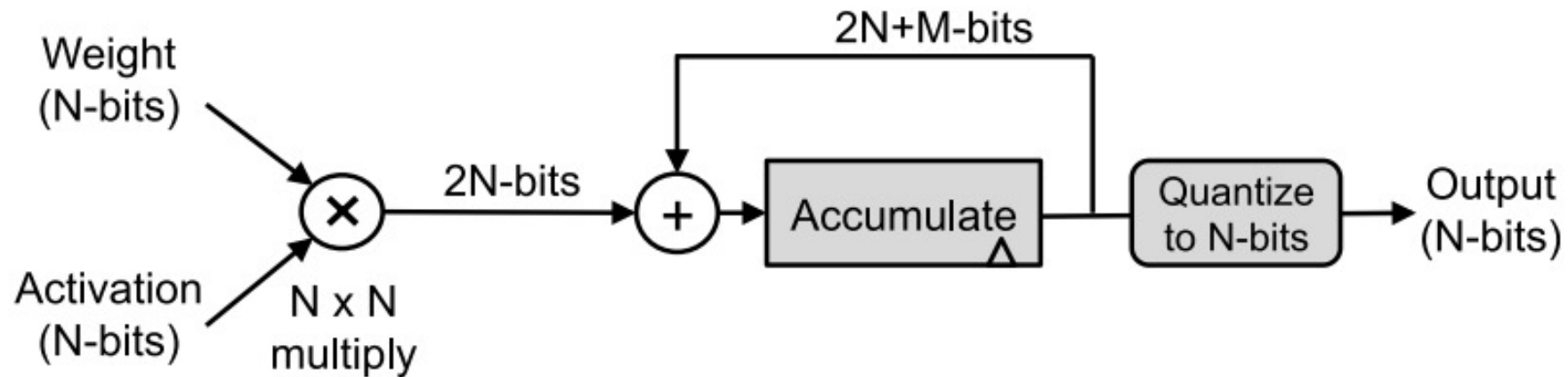
Integer Quantization

- Internal operations:



Integer Quantization

- Another view (theoretical bit-widths):



Float16 Quantization (Weights)

- Almost no impact on accuracy
- 2x reduction in model size compared to float32.
- Some hardware, like modern GPUs, can compute natively in float16 → **Quantization Speedup**
- If native float16 operations are not available, weights are unpacked to float32 before the first inference (model size reduction with minimal impact on latency)

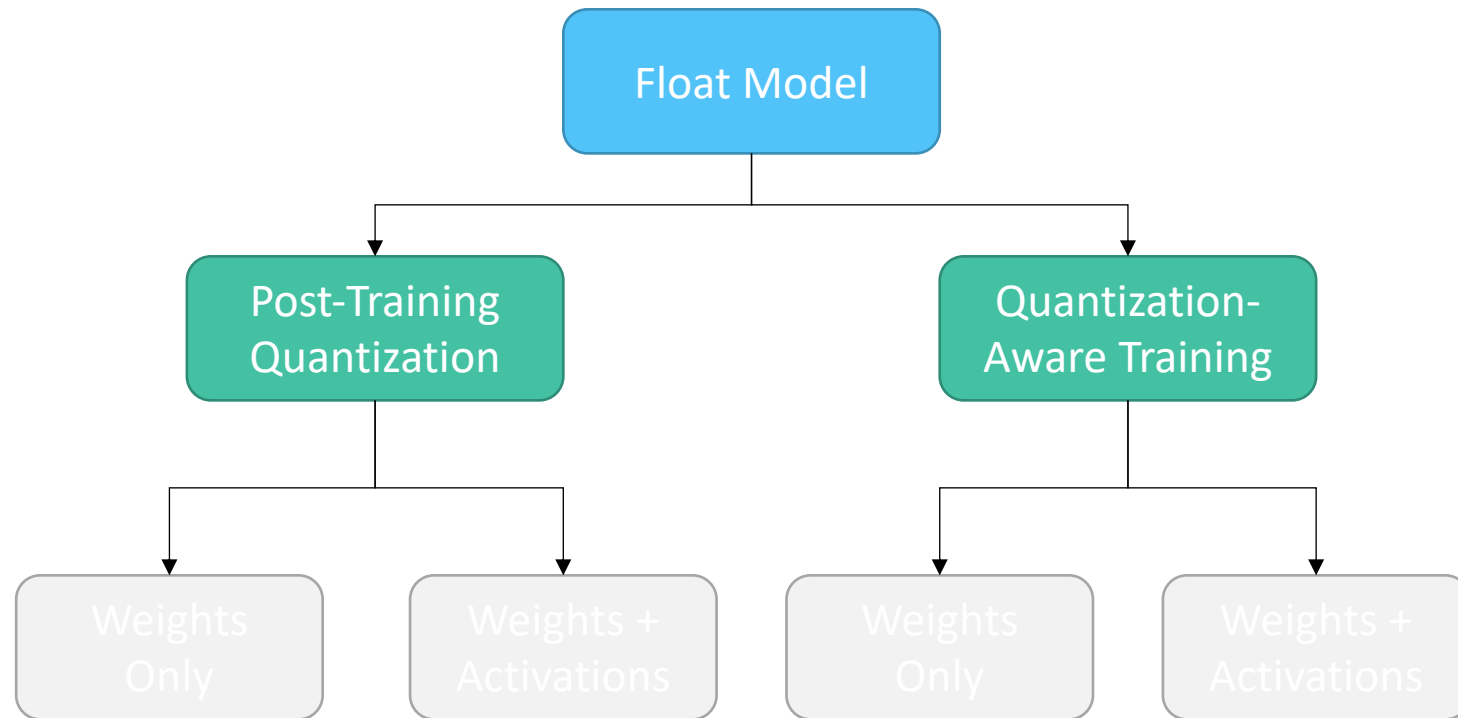
Float16 Quantization (Weights)

- Conversion from IEEE single precision float to 16bit minifloat follows a set of rules that take into account all special cases of the format (e.g. subnormal numbers, +-infinity, etc.). We'll not go into details.
- In TFLite, float16 quantization is enforced with the following options:

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]  
converter.target_spec.supported_types = [tf.float16]
```

Quantization: Strategies

- Quantization can be applied either post-training or at training time.



Post-Training Quantization

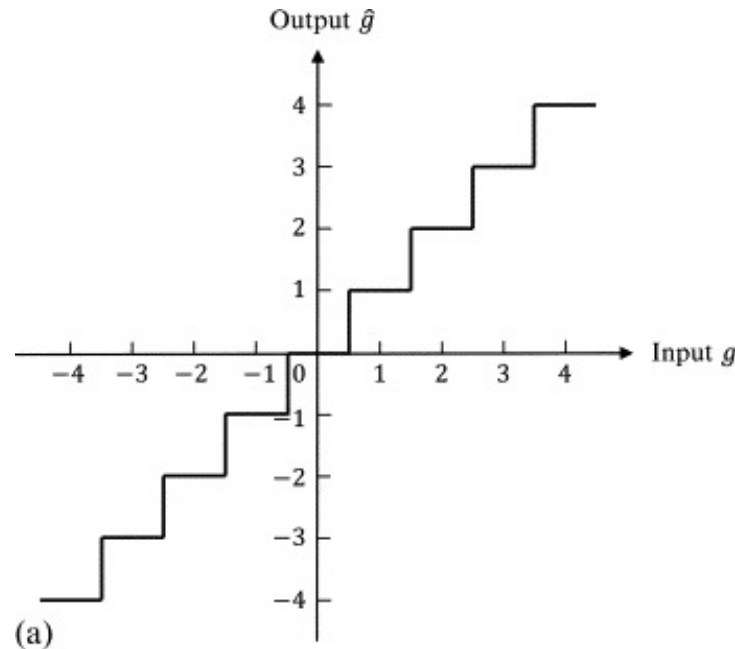
- The model is trained normally, in float32
- After training, weights (and optionally activations) are converted to a lower precision format (e.g. int8)
- **Pros:** It is the simplest approach. It can be applied to pre-trained models, with no re-training and (in case of weights-only quantization) even without input data available.
- **Cons:** it can cause a significant accuracy drop on complex tasks (1-10%).

Quantization-Aware Training

- For large models (more redundant) post-training quantization works fine, obtaining a significant model size and latency/energy reduction with little impact on accuracy.
- However, the accuracy of smaller models can be affected significantly by the “approximation” of weights and activations
- The obvious way to recover this accuracy drop is to **introduce quantization already during training**. In this way, gradient-descent-based training algorithms can take into account the fact that weights and activations can only assume a limited set of values, and adjust the former accordingly.

Quantization-Aware Training

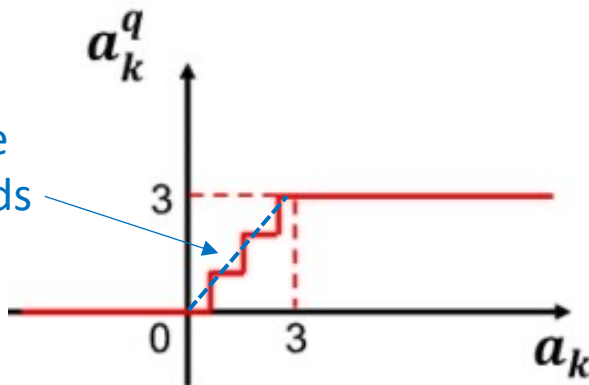
- However, quantization operations create **problems with gradients**:
 - The actual gradient is zero in all differentiable points.
 - In practice, using quantization during back-prop makes small weight updates impossible, and therefore training to convergence more difficult.



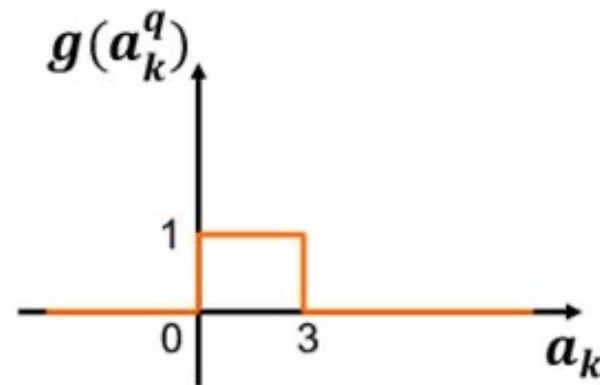
Quantization-Aware Training

- Solution: **Straight-Through Estimators (STEs)**
- Use **quantization only in the forward pass**, and replace it with a differentiable function (e.g. **identity**) in the **backward pass**.
- Example for 2-bit quantization:

Approximated by the
blue line in backwards
passes



Quantization function

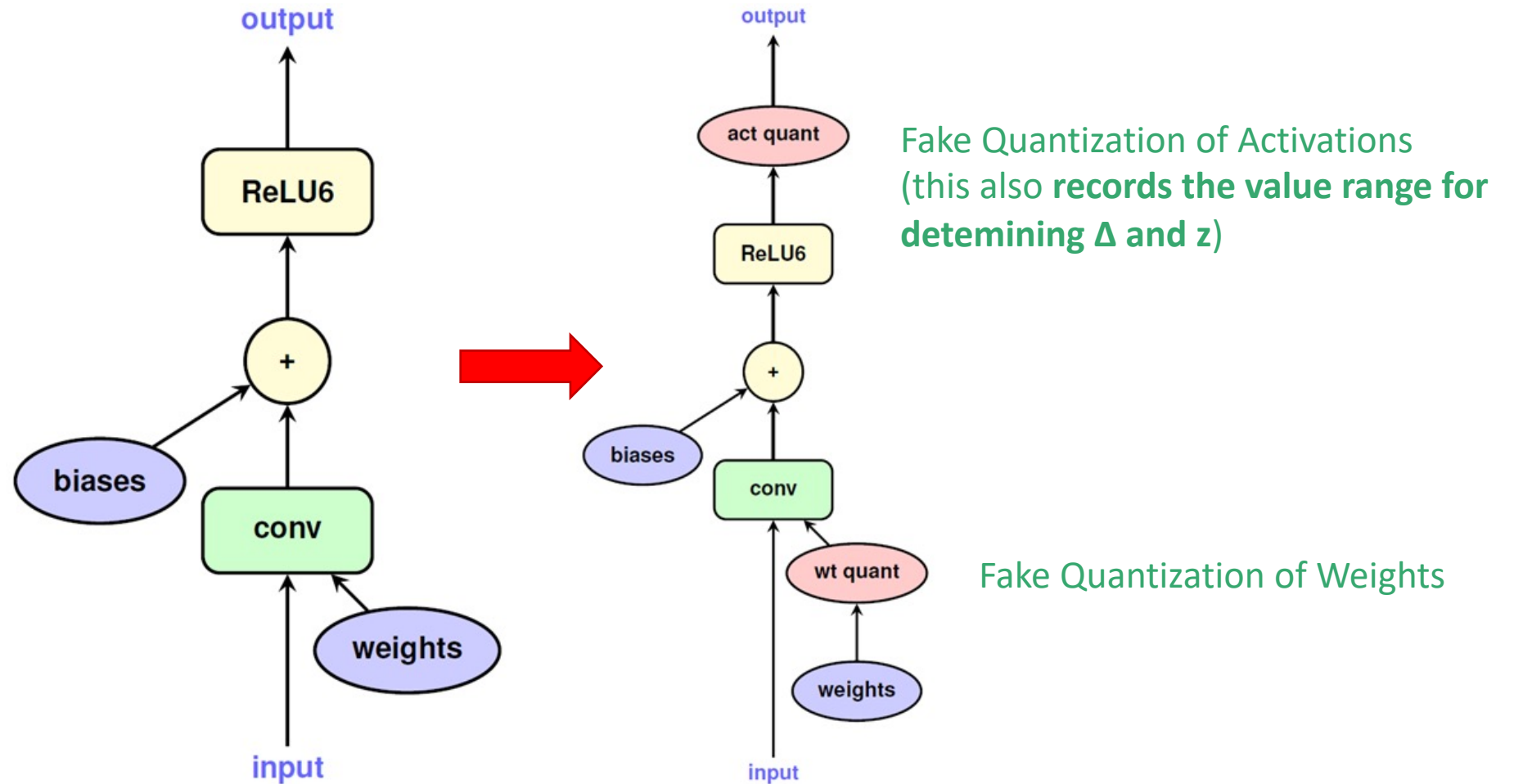


Quantization gradient

Quantization-Aware Training

- In practice, training is still performed in float, but adding “fake quantization” ops (nodes) to the model graph
- In the forward pass, these nodes **simulate the effect of quantization**, rounding the inputs and outputs of an operation as if it was performed on integer data
 - NB: The actual computation is still done in float.
- In the backward pass, the STE approach is used to back-propagate gradients through fake quantization nodes

Quantization-Aware Training



Quantization-Aware Training

- In TF jargon, a model with fake quantization nodes is called “**quantization-aware**” model and is created as follows:

```
import tensorflow_model_optimization as tfmot  
  
quant_aware_model = tfmot.quantization.keras.quantize_model(base_model)
```



A float keras model, possibly pre-trained
(Only Sequential/Functional APIs are supported)

Quantization-Aware Training

- You can also quantize only some selected layers (for better accuracy) based on the layer name or its type:

```
def apply_quantization_to_dense(layer):  
    if isinstance(layer, tf.keras.layers.Dense):  
        return tfmot.quantization.keras.quantize_annotate_layer(layer)  
    return layer
```

```
annotated_model = tf.keras.models.clone_model(base_model,  
        clone_function=apply_quantization_to_dense)
```

```
quant_aware_model =  
    tfmot.quantization.keras.quantize_apply(annotated_model)
```

Quantization-Aware Training

- The **quantization-aware** model can be then fine-tuned normally using `fit()`

```
quant_aware_model.fit(...)
```

- After that, we can turn it into an actual **quantized** model for TFLite as usual:

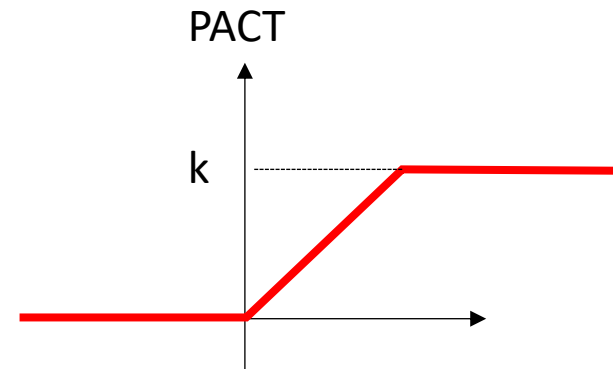
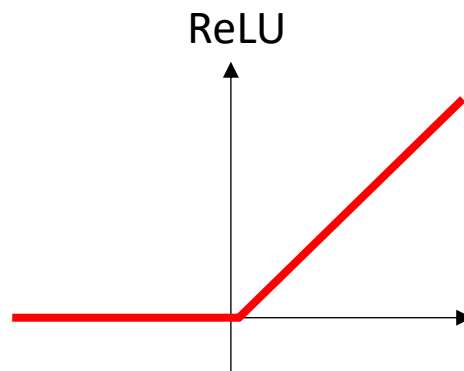
```
converter = tf.lite.TFLiteConverter.from_keras_model(quant_aware_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
```

```
quantized_tflite_model = converter.convert()
```

- TF also allows you to customize the quantization bit-width and parameters for each layer, to create custom quantizers, etc. We'll not go into details (see info [here](#))

QAT - Clipped Activations

- Using bounded activations keeps the activations range limited, thus helping quantization. Example: ReLU-6 vs standard ReLU.
- One advanced variant is the PArametrized Clipping acTivation (PACT), i.e. a ReLU-k where k is learned during quantization-aware training by back-propagation.

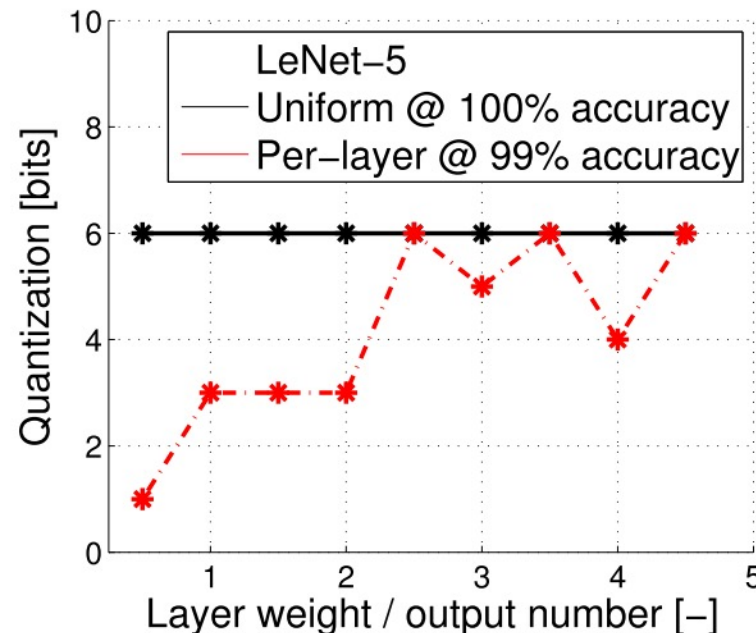


Advanced Quantization

Fixed-Precision vs Mixed-Precision

Quantization: Fixed- vs Mixed-Precision

- **Fixed-precision:** while Δ and z change per-tensor (or channel), the bit-width N is fixed for the entire network
- **Mixed-precision:** uses a different N layer-wise or channel-wise.
 - N_w (weights) can also be different from N_x (activations)
 - Possibly higher compression for the same accuracy



[Source] B. Moons. Energy-efficient ConvNets through approximate computing, 2016

Mixed-Precision Quantization

- Various search algorithms (greedy heuristics, simulated annealing, genetic, etc.) can be used to determine the optimal bit-width per each layer.
- Typically done during training to let the model “recover” from the approximations introduced by smaller bit-widths.
- Not yet supported by TF/TFLite

Advanced Quantization

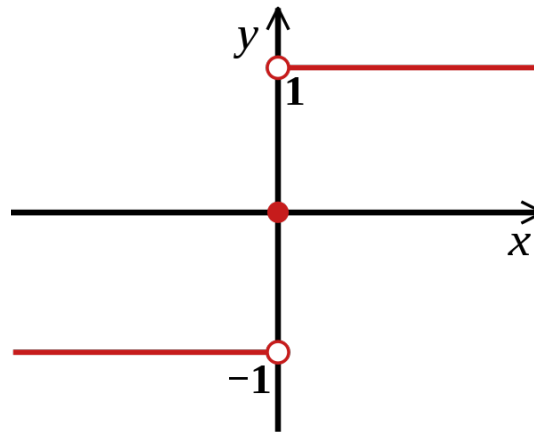
Binary Neural Networks

Binary Neural Networks

- Extreme form of quantization, where data are constrained to have a binary value.
 - Initially used only for **weights**, ([BinaryConnect](#)) then extended also to **activations** ([XNOR-Net](#))
- **Obvious advantage**: 1-bit vs 32 (float) → 32x reduction in model size
- **Less obvious advantage**: MAC operations can be replaced by much simpler bit manipulations. Significant speed-up and energy efficiency improvement, also on general purpose HW.
- **Drawback**: accuracy can drop significantly on complex tasks

Binary Neural Networks

- Weights and activations are constrained to have one of two values: $w, x: \{-1, 1\}$
 - In practice, **-1 is represented by logic-0** for HW simplicity.
 - More advanced variants use $\{-\alpha, \alpha\}$, where α is set per-tensor.
- Conversion is done using the $\text{sign}()$ function
 - Binarization output is +1 if the corresponding input value is ≥ 0 and -1 otherwise



Binary Neural Networks

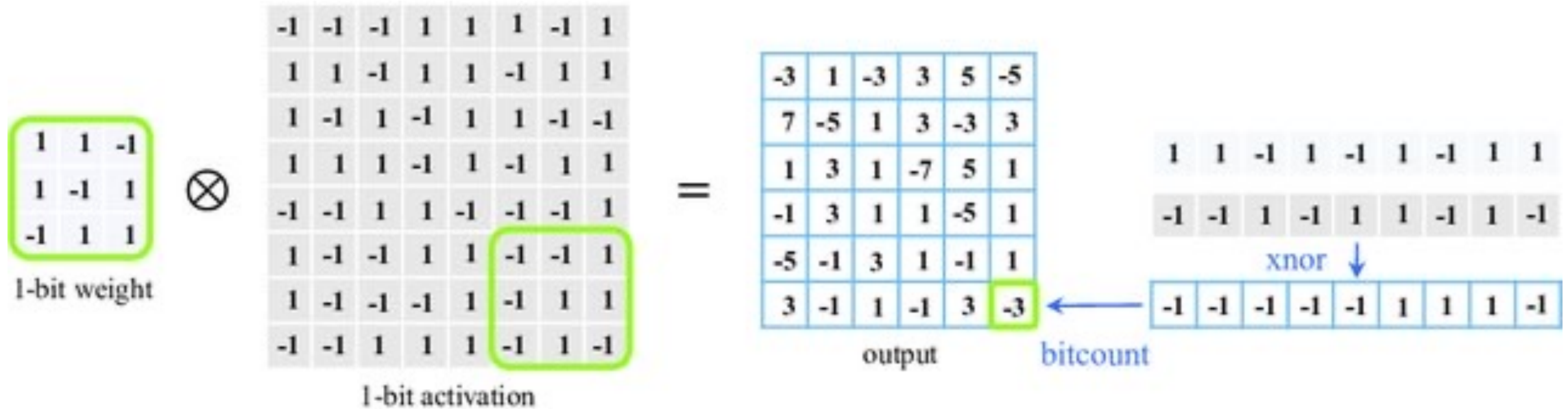
- **Multiplication** of binary weights represented in this format becomes a **binary XNOR** (logic function equal to 1 if two bits are equal, 0 otherwise).

Value of A	Value of B	Value of A * B	Logic A	Logic B	Logic A XNOR Logic B
-1	-1	1	0	0	1
-1	1	-1	0	1	0
1	-1	-1	1	0	0
1	1	1	1	1	1

Binary Neural Networks

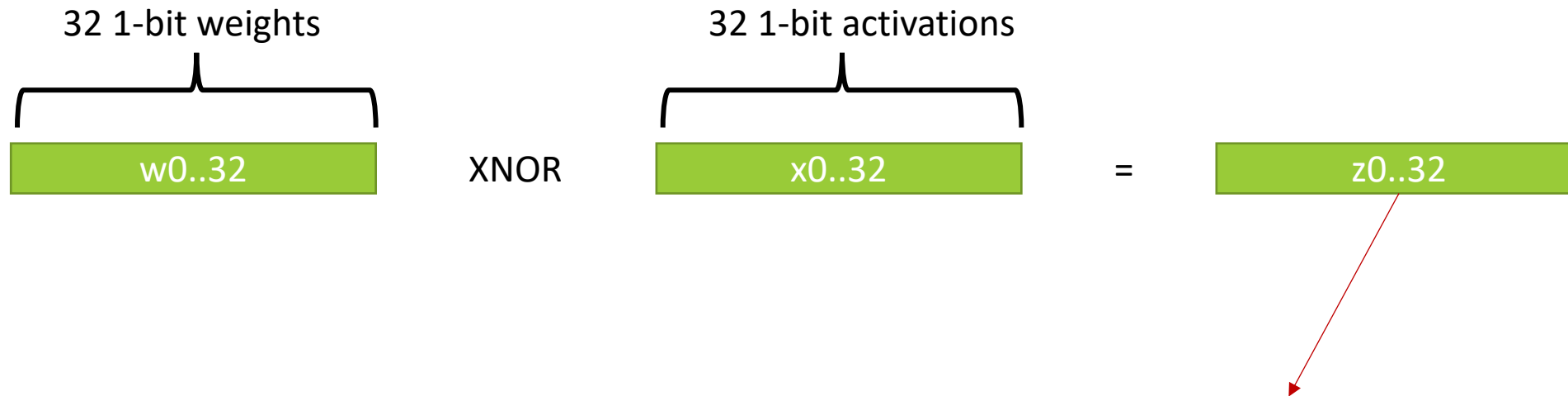
- **Accumulation** reduces to counting the number of 1s (*bitcount* or *popcount*):
- Assume we want to accumulate the results of N bit-wise XNORs (multiplications)
- Let us call:
 - p = count of 1s
 - n = count of 0s (-1s),
- $Accum = p - n = p - (N - p) = 2 * p - N$
 - Since 2 and N are constants, equivalent to p

Binary Neural Networks



Binary Neural Networks

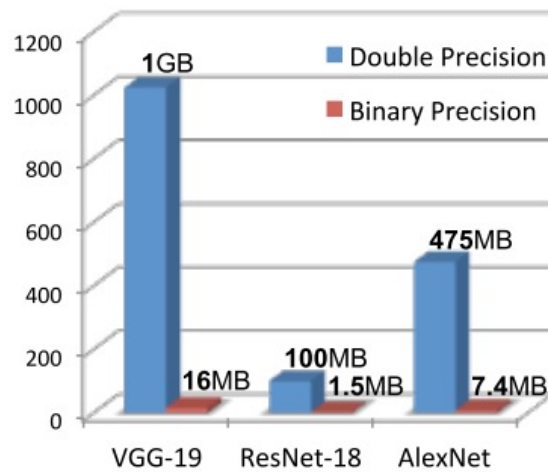
- BNNs are also easy to implement on general purpose HW (differently from other sub-byte quantization formats).
- All general purpose CPUs have instructions for bit-wise operations on words.



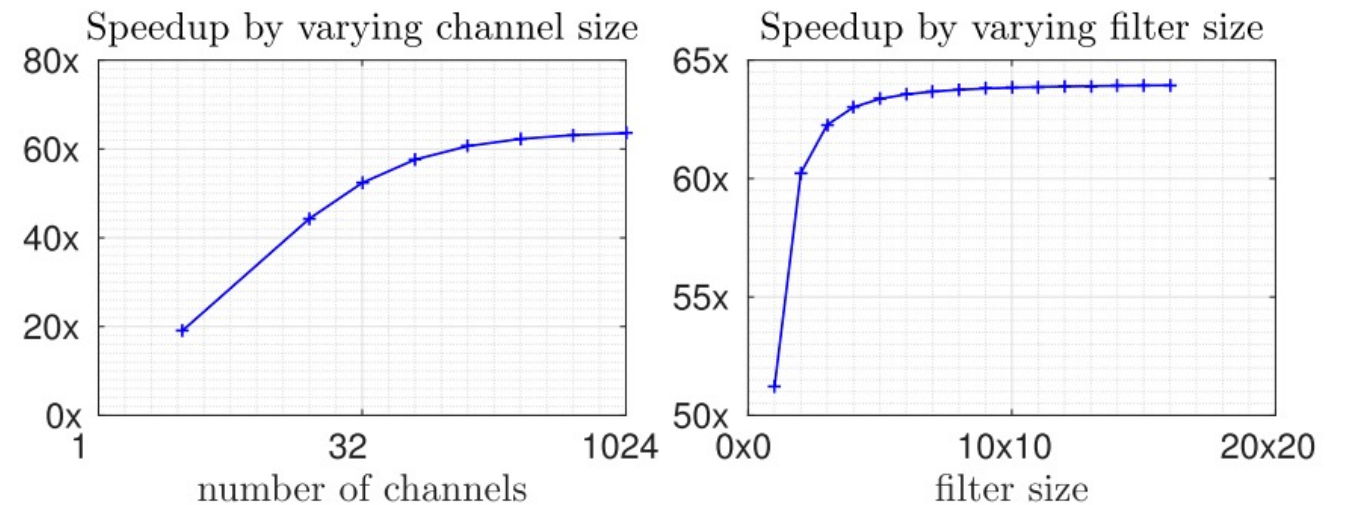
32 multiplications can be performed in a single instruction (even without considering multi-core)

Binary Neural Networks

- BNN results:



Memory Occupation

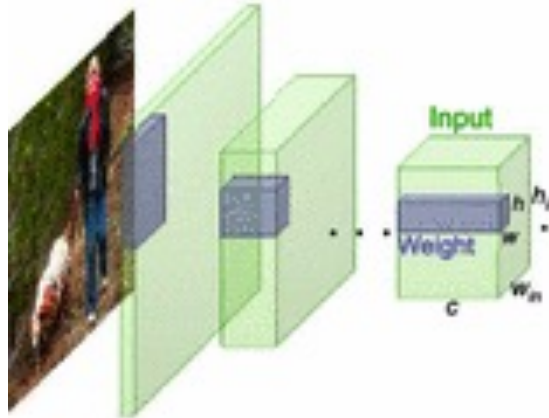


Speed-up

[Source] M. Rastegari et al. XNOR-Net

Binary Neural Networks

- BNN results (cont'd):



	Network Variations		Operations used in Convolution	Memory Saving (Inference)	Computation Saving (Inference)	Accuracy on ImageNet (AlexNet)
Standard Convolution	Real-Value Inputs 0.11 -0.21 ... -0.34 -0.25 0.61 ... 0.52	Real-Value Weights 0.12 -0.2 ... 0.41 -0.2 0.5 ... 0.44	+ , - , ×	1x	1x	%56.7
Binary Weight	Real-Value Inputs 0.11 -0.21 ... -0.34 -0.25 0.61 ... 0.52	Binary Weights 1 1 1 -1 1 1	+ , -	~32x	~2x	%56.8
BinaryWeight Binary Input (XNOR-Net)	Binary Inputs 1 -1 ... -1 -1 1 ... 1	Binary Weights 1 1 1 -1 1 1	XNOR , bitcount	~32x	~58x	%44.2

[Source] M. Rastegari et al. XNOR-Net

Binary Neural Networks

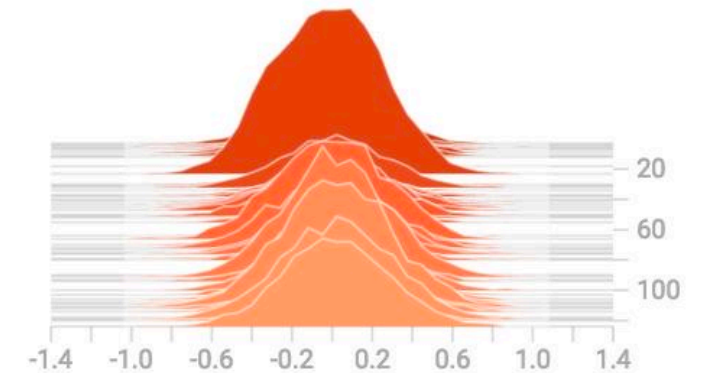
- Not yet supported in TF/TFLite

Advanced Quantization

Non-uniform Quantization

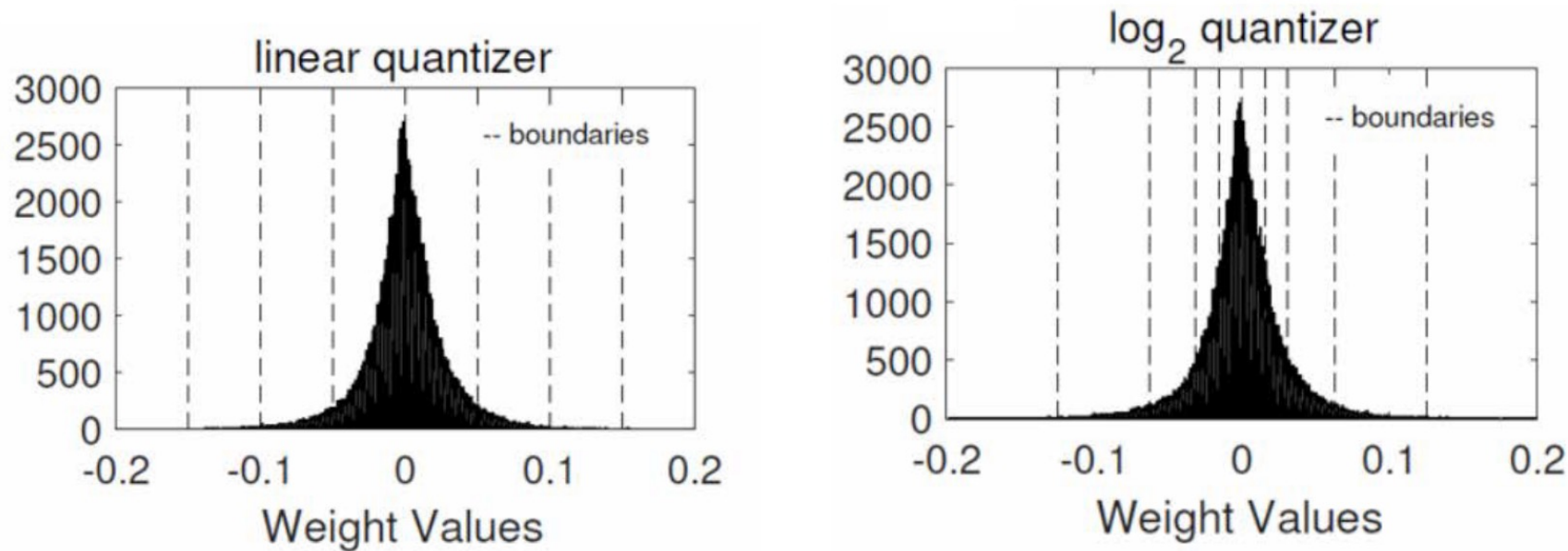
Non-Uniform Quantization

- Distributions of weights and activations are almost never uniform.
- Non-uniformly spaced quantization points may improve accuracy
- We'll quickly overview two approaches:
 - Log-domain quantization
 - Learned quantization (called *weight clustering* or *weight sharing*).



Log-domain quantization

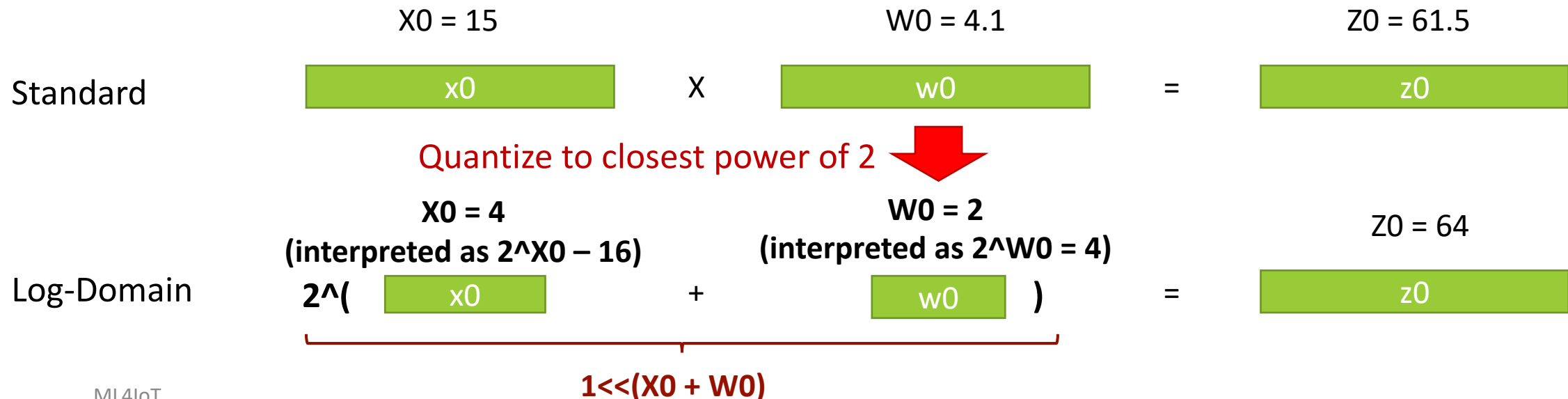
- The main idea of log-domain quantization is to have more quantization levels close to 0, since that is where most weight values fall.
- This is achieved using a logarithmic mapping, rather than a linear one.



[Source] E. H. Lee et al. LogNet

Log-domain quantization

- In particular, one HW-friendly approach is to let weights only assume values that are **powers of 2** (0, 1, 2, 4, etc.)
- In this ways, multiplications can be replaced by shifts and sums which are more efficient in HW.



Log-domain quantization

- In general:
 - $\widetilde{w}_i = \text{Quant}(\log_2(w_i)), \widetilde{x}_i = \text{Quant}(\log_2(x_i))$
 - $w^T x \cong \sum_{i=1}^n 2^{\widetilde{w}_i + \widetilde{x}_i} = \sum_{i=1}^n \text{Bitshift}(1, \widetilde{w}_i + \widetilde{x}_i)$
- Returning to logarithmic space simply involves finding the leftmost 1 in the result (+ rounding)

Log-domain quantization

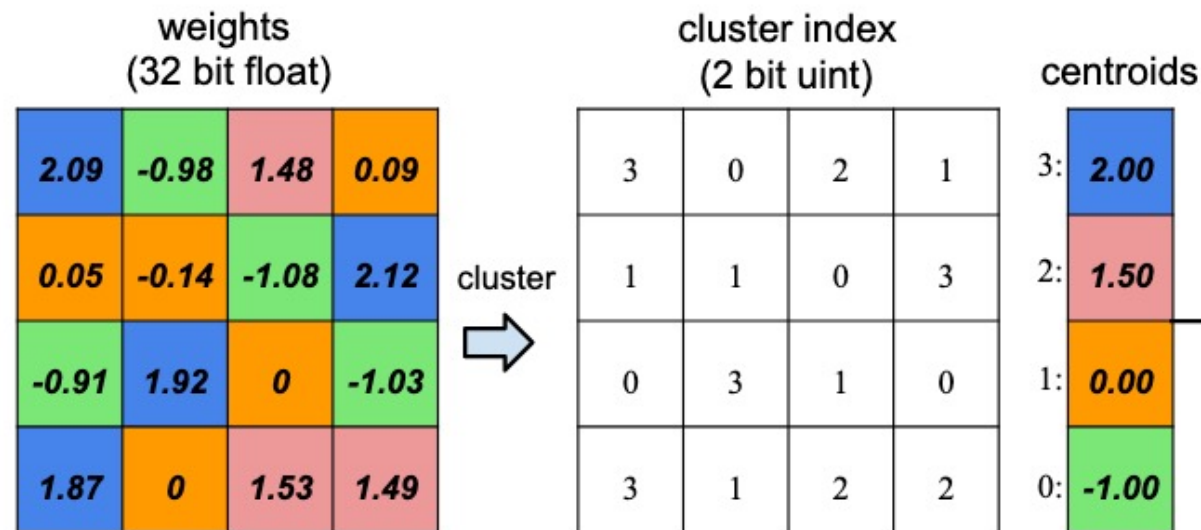
- Not yet supported in TF/TFLite

Weights Clustering

- It can be considered a form of quantization in which rather than reducing the precision of each weight/activation value, we reduce the **number of distinct values allowed**.
- Store the allowed values in a **codebook** (or *look-up table*)
 - Store weight as codebook index → reduce storage size

Weights Clustering

- Idea: replace each float weight with its **closest allowed value** (cluster centroid). Then, store only the values of centroids plus one index per weight.



[Source] S. Han et al. Deep Compression

Weights Clustering

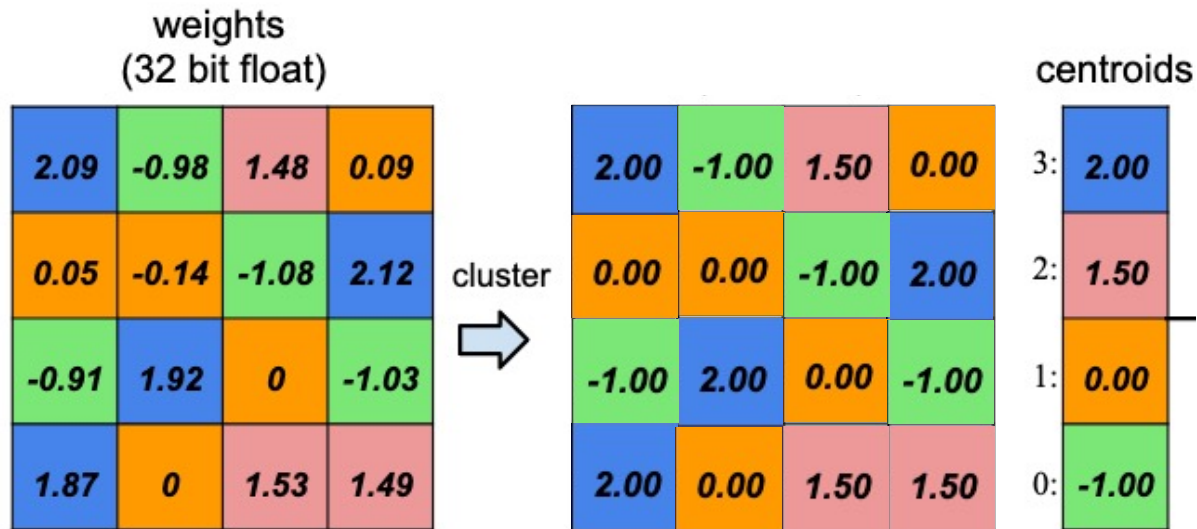
- Example: 4x4 matrix with 4 allowed values:
 - Original storage: $4 \times 4 \times 32\text{bit} = 512\text{bit}$
 - Clustered storage: $4 \times 32\text{bit}$ (centroids) + $4 \times 4 \times \log_2(4)$ (indexes) = 160bit
- Model size reduction, but possibly slower inference (2-step weight access).

In theory, we only need 2bit to store 4 distinct indices



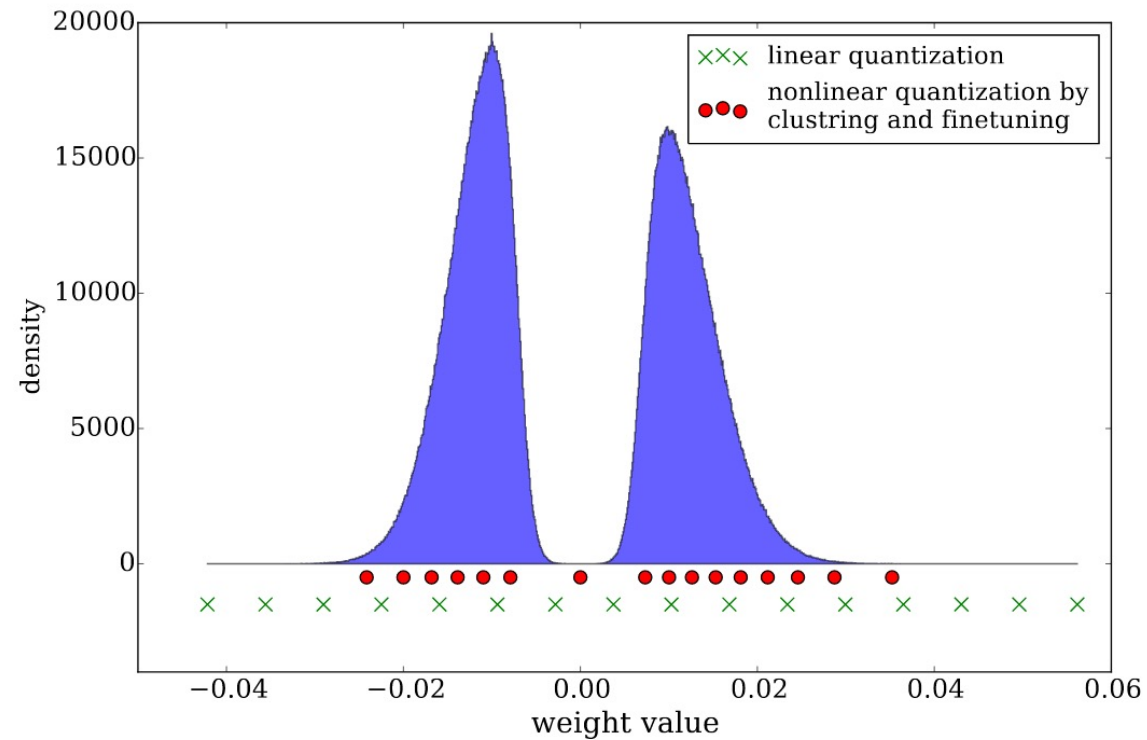
Weights Clustering

- Even if we don't store the indexes but simply replace each weight with the closest centroid, we can still reduce the model size
- The model will still contain 4x4x32bit weights, but only 4 distinct values. A standard compression algorithm (e.g. zip) can take advantage of this.



Weights Clustering

- Example:



[Source] S. Han et al. Deep Compression

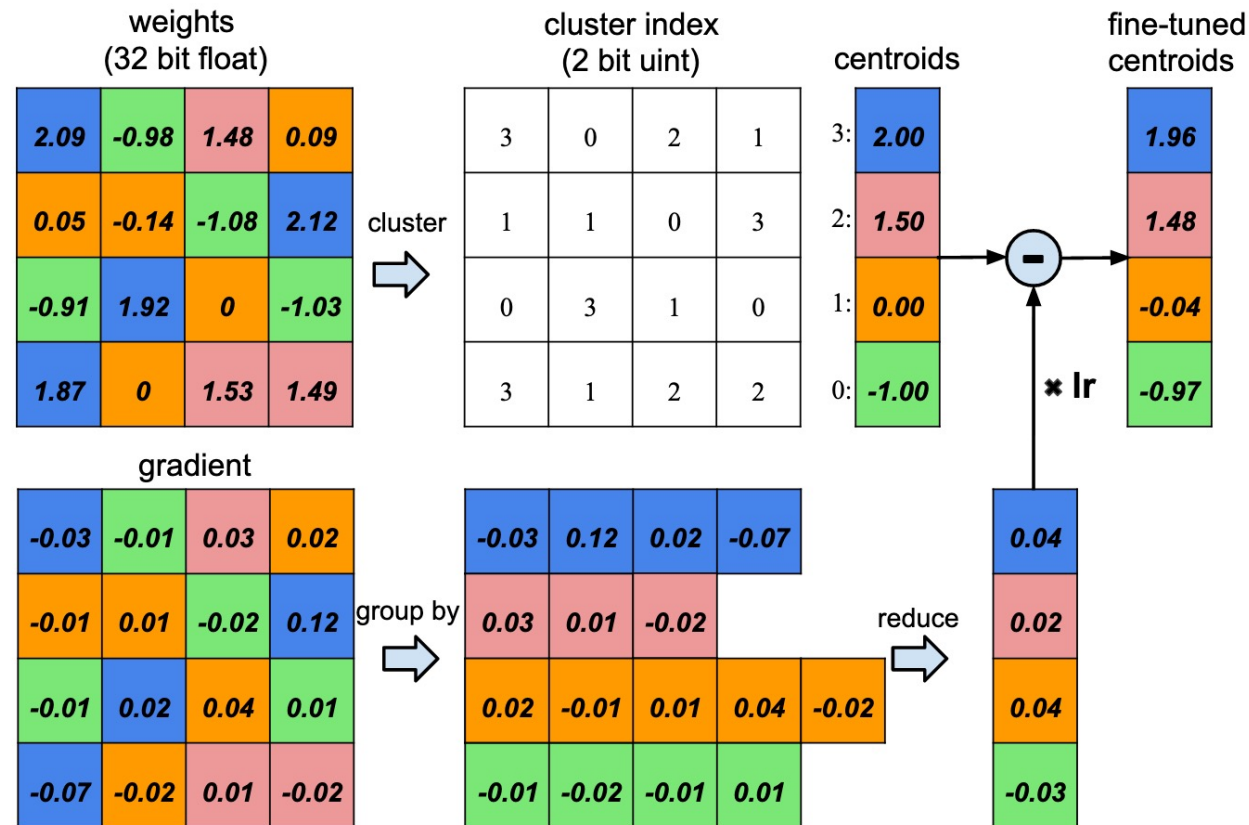
Weights Clustering

- The key issue is how to determine the centroids during training. The most famous algorithm (which is the one recently introduced in TF) is by Song Han et al and works as follows:
 1. First, the **centroids are initialized** using a **pre-trained float model**.
 - Different initialization methods are available.
 - The “Linear” method uses centroids uniformly spaced between the min and max value of each weight tensor, and is a good starting point.
 2. Then, a k-means clustering is applied to obtain the **initial centroids**.

Weights Clustering

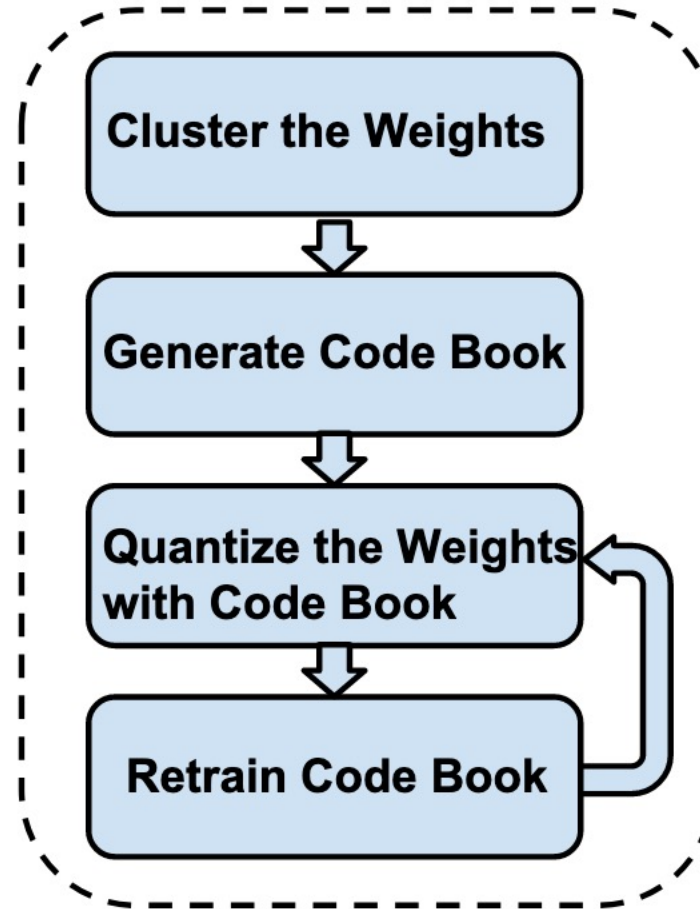
3. The model is then **fine-tuned** as follows:
 - a) In the forward pass, each weight is replaced with the closest centroid
 - b) In the backward pass, the gradients of all weights that were assigned to the same cluster are summed together, and used to update the centroid for the next iteration.

Weights Clustering



Weights Clustering

- Recap:



Weights Clustering

- TF has recently added support for weights clustering.
- Weights clustering is added to a pre-trained model as follows:
 - Then, you need to fine-tune the model with fit()
 - As for QAT, you can also selectively fine-tune only some of the layers.

```
import tensorflow_model_optimization as tfmot

clustered_model = tfmot.clustering.keras.cluster_weights(
    model,
    number_of_clusters=4,
    cluster_centroids_init = tfmot.clustering.keras.CentroidInitialization.LINEAR
)
```

Weights Clustering

- After fine-tuning, you should remove the additional wrappers used by TF to enforce weight sharing.

```
final_model = tfmot.clustering.keras.strip_clustering(clustered_model)
```

- In the current implementation, to see the benefits of weights clustering you need to compress the model (e.g. with gzip).

Weights Clustering

- Weight clustering can be combined with post—training quantization, so that centroids are stored as int8 rather than float32.
- Further model size reduction...