Quantization

Content

- Motivation from tensorflow doc
- Overview of quantization methods from survey paper
- Two methods in more depth
  - One based on two tools
  - The other based on a research paper

From https://www.tensorflow.org/performance/quantization
https://www.tensorflow.org/lite/performance/model_optimization

How to Quantize Neural Networks with TensorFlow

When modern neural networks were being developed, the biggest challenge was getting them to work at all! That meant that accuracy and speed during training were the top priorities. Using floating point arithmetic was the easiest way to preserve accuracy, and GPUs were well-equipped to accelerate those calculations, so it's natural that not much attention was paid to other numerical formats.

These days, we actually have a lot of models being deployed in commercial applications. The computation demands of training grow with the number of researchers, but the cycles needed for inference expand in proportion to users. That means pure inference efficiency has become a burning issue for a lot of teams.

That is where quantization comes in. It's an umbrella term that covers a lot of different techniques to store numbers and perform calculations on them in more compact formats than 32-bit floating point. I am going to focus on eight-bit fixed point, for reasons I'll go into more detail on later.

Why Quantize?

From https://www.tensorflow.org/performance/quantization

Neural network models can take up a lot of space on disk, with the original AlexNet being over 200 MB in float format for example. Almost all of that size is taken up with the weights for the neural connections, since there are often many millions of these in a single model. Because they're all slightly different floating point numbers, simple compression formats like zip don't compress them well. They are arranged in large layers though, and within each layer the weights tend to be normally distributed within a certain range, for example -3.0 to 6.0.

storage

The simplest motivation for quantization is to shrink file sizes by storing the min and max for each layer, and then compressing each float value to an eight-bit integer representing the closest real number in a linear set of 256 within the range. For example with the -3.0 to 6.0 range, a 0 byte would represent -3.0, a 255 would stand for 6.0, and 128 would represent about 1.5. I'll go into the exact calculations later, since there's some subtleties, but this means you can get the benefit of a file on disk that's shrunk by 75%, and then convert back to float after loading so that your existing floating-point code can work without any changes.

latency

Another reason to quantize is to reduce the computational resources you need to do the inference calculations, by running them entirely with eight-bit inputs and outputs. This is a lot more difficult since it requires changes everywhere you do calculations, but offers a lot of potential rewards. Fetching eight-bit values only requires 25% of the memory bandwidth of floats, so you'll make much better use of caches and avoid bottlenecking on RAM access. You can also typically use SIMD operations that do many more operations per clock cycle. In some case you'll have a DSP chip available that can accelerate eight-bit calculations too, which can offer a lot of advantages.

energy

Moving calculations over to eight bit will help you run your models faster, and use less power (which is especially important on mobile devices). It also opens the door to a lot of embedded systems that can't run floating point code efficiently, so it can enable a lot of applications in the IoT world.

Why does it work?

https://petewarden.com/2015/05/23/why-are-eight-bits-enough-for-deep-neural-networks/

I can't see any fundamental mathematical reason why the results should hold up so well with low precision, so I've come to believe that it emerges as a side-effect of a successful training process. When we are trying to teach a network, the aim is to have it understand the patterns that are useful evidence and discard the meaningless variations and irrelevant details. That means we expect the network to be able to produce good results despite a lot of noise. Dropout is a good example of synthetic grit being thrown into the machinery, so that the final network can function even with very adverse data.

accuracy

The networks that emerge from this process have to be very robust numerically, with a lot of redundancy in their calculations so that small differences in input samples don't affect the results. Compared to differences in pose, position, and orientation, the noise in images is actually a comparatively small problem to deal with. All of the layers are affected by those small input changes to some extent, so they all develop a tolerance to minor variations. That means that the differences introduced by low-precision calculations are well within the tolerances a network has learned to deal with. Intuitively, they feel like weebles that won't fall down no matter how much you push them, thanks to an inherently stable structure.

Why does Quantization Work?

accuracy

Training neural networks is done by applying many tiny nudges to the weights, and these small increments typically need floating point precision to work (though there are research efforts to use quantized representations here too).

Taking a pre-trained model and running inference is very different. One of the magical qualities of deep networks is that they tend to cope very well with high levels of noise in their inputs. If you think about recognizing an object in a photo you've just taken, the network has to ignore all the CCD noise, lighting changes, and other non-essential differences between it and the training examples it's seen before, and focus on the important similarities instead. This ability means that they seem to treat low-precision calculations as just another source of noise, and still produce accurate results even with numerical formats that hold less information.

From https://www.tensorflow.org/performance/quantization
https://www.tensorflow.org/lite/performance/model_optimization

Why Not Train in Lower Precision Directly?

There have been some experiments training at lower bit depths, but the results seem to indicate that you need higher than eight bit to handle the back propagation and gradients. That makes implementing the training more complicated, and so starting with inference made sense. We also already have a lot of float models already that we use and know well, so being able to convert them directly is very convenient.

How to declare fixed point variables in code? How to do fixed point math in code?

Fixed Point Math

https://github.com/eteran/cpp-utilities

Fixed.h

This is a Fixed Point math class for c++11. It supports all combinations which add up to a native data types (8.8/16.16/24.8/etc). The template parameters are the number of bits to use as the base type for both the integer and fractional portions, invalid combinations will yield a compiler error; the current implementation makes use of c++11 `static assert` to make this more readable. It should be a nice drop in replacement for native `float` types. Here's an example usage:

```
typedef numeric::Fixed<16, 16> fixed;
fixed f;
```

This will declare a 16.16 fixed point number. Operators are provided though the use of boost::operators. multiplication and division are implemented in free functions named `numeric::multiply` and `numeric::divide` which use `std::enable_if` to choose the best option. If a larger type is available, it will use the accurate and fast scaled math version. If there is not a larger type available, then it will fall back on the slower multiply and emulated divide (which unfortunately has less precision). This system allows the user to specialize the multiplication and division as needed.

Wrapper class with integer arithmetic.

Quantization results comparison

Reduce Precision Method		bitwidth		Accuracy loss vs. 32-bit float (%)
		Weights	Activations	
Dynamic Fixed Point	w/o fine-tuning [121]	8	10	0.4
	w/ fine-tuning [122]	8	8	0.6
Reduce Weight	BinaryConnect [127]	1	32 (float)	19.2
	Binary Weight Network (BWN) [129]	1*	32 (float)	0.8
	Ternary Weight Networks (TWN) [131]	2*	32 (float)	3.7
	Trained Ternary Quantization (TTQ) [132]	2*	32 (float)	0.6
Reduce Weight and Activation	XNOR-Net [129]	1*	1*	11
	Binarized Neural Networks (BNN) [128]	1	1	29.8
	DoReFa-Net [120]	1*	2*	7.63
	Quantized Neural Networks (QNN) [119]	1	2*	6.5
	HWGQ-Net [130]	1*	2*	5.2
Non-linear Quantization	LogNet [135]	5 (conv), 4 (fc)	4	3.2
	Incremental Network Quantization (INQ) [136]	5	32 (float)	-0.2
	Deep Compression [118]	8 (conv), 4 (fc)	16	0
		4 (conv), 2 (fc)	16	2.6

Network: Alexnet   Dataset: Imagenet   Accuracy measured: Top-5 error

From V. Sze, T.-J. Yang, Y.-H. Chen, J. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," arXiv, 2017.

Quantization results: only weights

Reduce Precision Method		bitwidth		Accuracy loss vs. 32-bit float (%)
		Weights	Activations	
Dynamic Fixed Point	w/o fine-tuning [121]	8	10	0.4
	w/ fine-tuning [122]	8	8	0.6
Reduce Weight	BinaryConnect [127]	1	32 (float)	19.2
	Binary Weight Network (BWN) [129]	1*	32 (float)	0.8
	Ternary Weight Networks (TWN) [131]	2*	32 (float)	3.7
	Trained Ternary Quantization (TTQ) [132]	2*	32 (float)	0.6
Reduce Weight and Activation	XNOR-Net [129]	1*	1*	11
	Binarized Neural Networks (BNN) [128]	1	1	29.8
	DoReFa-Net [120]	1*	2*	7.63
	Quantized Neural Networks (QNN) [119]	1	2*	6.5
	HWGQ-Net [130]	1*	2*	5.2
Non-linear Quantization	LogNet [135]	5 (conv), 4 (fc)	4	3.2
	Incremental Network Quantization (INQ) [136]	5	32 (float)	-0.2
	Deep Compression [118]	8 (conv), 4 (fc)	16	0
		4 (conv), 2 (fc)	16	2.6

Network: Alexnet Dataset: Imagenet Accuracy measured: Top-5 error

From V. Sze, T.-J. Yang, Y.-H. Chen, J. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," arXiv, 2017.

Quantization results: both weights and activations

Reduce Precision Method		bitwidth Weights	bitwidth Activations	Accuracy loss vs. 32-bit float (%)
Dynamic Fixed Point	w/o fine-tuning [121]	8	10	0.4
	w/ fine-tuning [122]	8	8	0.6
Reduce Weight	BinaryConnect [127]	1	32 (float)	19.2
	Binary Weight Network (BWN) [129]	1*	32 (float)	0.8
	Ternary Weight Networks (TWN) [131]	2*	32 (float)	3.7
	Trained Ternary Quantization (TTQ) [132]	2*	32 (float)	0.6
Reduce Weight and Activation	XNOR-Net [129]	1*	1*	11
	Binarized Neural Networks (BNN) [128]	1	1	29.8
	DoReFa-Net [120]	1*	2*	7.63
	Quantized Neural Networks (QNN) [119]	1	2*	6.5
	HWGQ-Net [130]	1*	2*	5.2
Non-linear Quantization	LogNet [135]	5 (conv), 4 (fc)	4	3.2
	Incremental Network Quantization (INQ) [136]	5	32 (float)	-0.2
	Deep Compression [118]	8 (conv), 4 (fc)	16	0
		4 (conv), 2 (fc)	16	2.6

Network: Alexnet Dataset: Imagenet Accuracy measured: Top-5 error

From V. Sze, T.-J. Yang, Y.-H. Chen, J. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," arXiv, 2017.

Quantization results: binary values

https://github.com/MatthieuCourbariaux/BinaryConnect https://github.com/allenai/XNOR-Net

Reduce Precision Method		bitwidth		Accuracy loss vs. 32-bit float (%)
		Weights	Activations	
Dynamic Fixed Point	w/o fine-tuning [121]	8	10	0.4
	w/ fine-tuning [122]	8	8	0.6
Reduce Weight	BinaryConnect [127]	1	32 (float)	19.2
	Binary Weight Network (BWN) [129]	1*	32 (float)	0.8
	Ternary Weight Networks (TWN) [131]	2*	32 (float)	3.7
	Trained Ternary Quantization (TTQ) [132]	2*	32 (float)	0.6
Reduce Weight and Activation	XNOR-Net [129]	1*	1*	11
	Binarized Neural Networks (BNN) [128]	1	1	29.8
	DoReFa-Net [120]	1*	2*	7.63
	Quantized Neural Networks (QNN) [119]	1	2*	6.5
	HWGQ-Net [130]	1*	2*	5.2
Non-linear Quantization	LogNet [135]	5 (conv), 4 (fc)	4	3.2
	Incremental Network Quantization (INQ) [136]	5	32 (float)	-0.2
	Deep Compression [118]	8 (conv), 4 (fc)	16	0
		4 (conv), 2 (fc)	16	2.6

Network: Alexnet Dataset: Imagenet Accuracy measured: Top-5 error

From V. Sze, T.-J. Yang, Y.-H. Chen, J. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," arXiv, 2017.

Quantization results: more details

Reduce Precision Method		bitwidth Weights	bitwidth Activations	Accuracy loss vs. 32-bit float (%)
Dynamic Fixed Point	w/o fine-tuning [121]	8	10	0.4
	w/ fine-tuning [122]	8	8	0.6
Reduce Weight	BinaryConnect [127]	1	32 (float)	19.2
	Binary Weight Network (BWN) [129]	1*	32 (float)	0.8
	Ternary Weight Networks (TWN) [131]	2*	32 (float)	3.7
	Trained Ternary Quantization (TTQ) [132]	2*	32 (float)	0.6
Reduce Weight and Activation	XNOR-Net [129]	1*	1*	11
	Binarized Neural Networks (BNN) [128]	1	1	29.8
	DoReFa-Net [120]	1*	2*	7.63
	Quantized Neural Networks (QNN) [119]	1	2*	6.5
	HWGQ-Net [130]	1*	2*	5.2
Non-linear Quantization	LogNet [135]	5 (conv), 4 (fc)	4	3.2
	Incremental Network Quantization (INQ) [136]	5	32 (float)	-0.2
	Deep Compression [118]	8 (conv), 4 (fc)	16	0
		4 (conv), 2 (fc)	16	2.6

Network: Alexnet   Dataset: Imagenet   Accuracy measured: Top-5 error

From V. Sze, T.-J. Yang, Y.-H. Chen, J. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," arXiv, 2017.

Dynamic fixed point

LEPS
Laboratory for Embedded and Programmable Systems

HOME PEOPLE PROJECTS PUBLICATIONS BLOG EVENTS ALUMNI

Home / Ristretto | CNN Approximation

Ristretto | CNN Approximation

Ristretto: CNN approximation tool by LEPS

Created by Philipp Gysel

View On GitHub

Ristretto is an automated CNN-approximation tool which condenses 32-bit floating point networks. Ristretto is an extension of Caffe and allows to test, train and fine-tune networks with limited numerical precision.

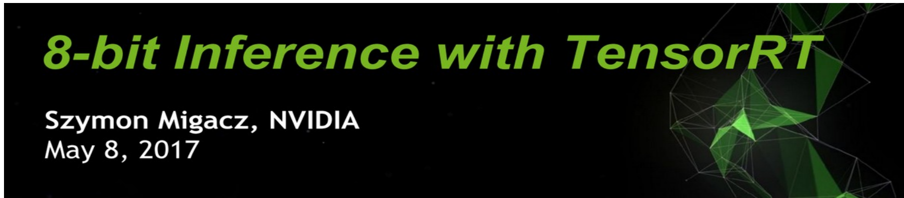
Ristretto In a Minute

- **Ristretto Tool:** The Ristretto tool performs automatic network quantization and scoring, using different bit-widths for number representation, to find a good balance between compression rate and network accuracy.
- **Ristretto Layers**: Ristretto re-implements Caffe-layers and simulates reduced word width arithmetic.
- **Testing and Training:** Thanks to Ristretto's smooth integration into Caffe, network description files can be changed to quantize different layers. The bit-width used for different layers as well as other parameters can be set in the network's prototxt file. This allows to directly test and train condensed networks, without any need of recompilation.

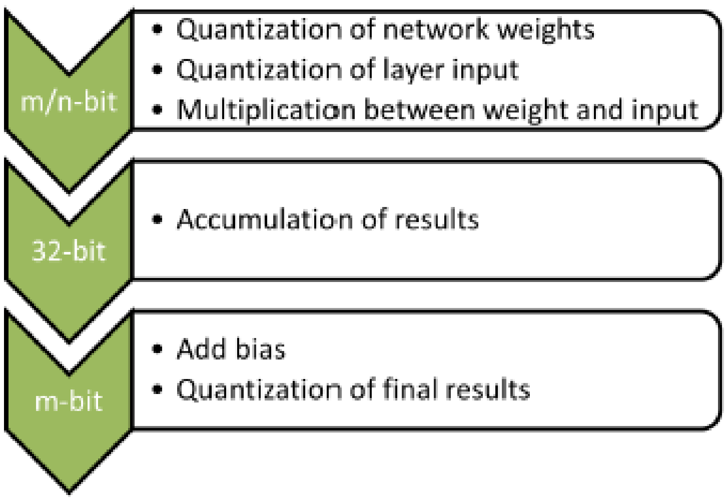
Approximation Schemes

Ristretto allows for three different quantization strategies to approximate Convolutional Neural Networks:

- **Dynamic Fixed Point**: A modified fixed-point format.
- **Minifloat**: Bit-width reduced floating point numbers.
- **Power-of-two parameters**: Layers with power-of-two parameters don't need any multipliers, when implemented in hardware.



Ristretto quantization methods: fixed point, dynamic fixed point, minifloat

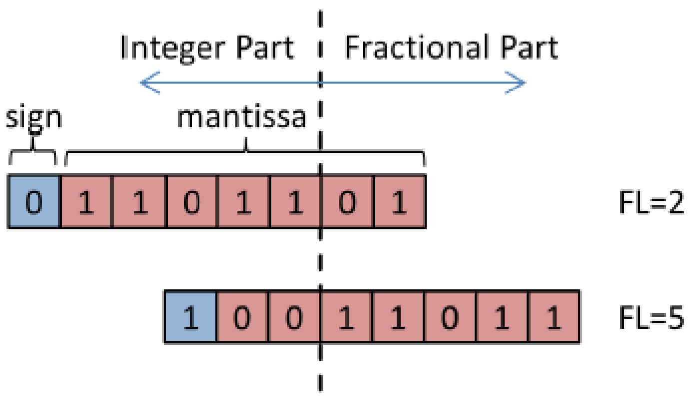


$$round(x) = \begin{cases} \lfloor x \rfloor, & \text{if } \lfloor x \rfloor \leq x \leq x + \frac{\epsilon}{2} \\ \lfloor x \rfloor + \epsilon, & \text{if } \lfloor x \rfloor + \frac{\epsilon}{2} < x \leq x + \epsilon \end{cases}$$

Deterministic rounding

$$round(x) = \begin{cases} \lfloor x \rfloor, & \text{w.p. } 1 - \frac{x - \lfloor x \rfloor}{\epsilon} \\ \lfloor x \rfloor + \epsilon, & \text{w.p. } \frac{x - \lfloor x \rfloor}{\epsilon} \end{cases}$$

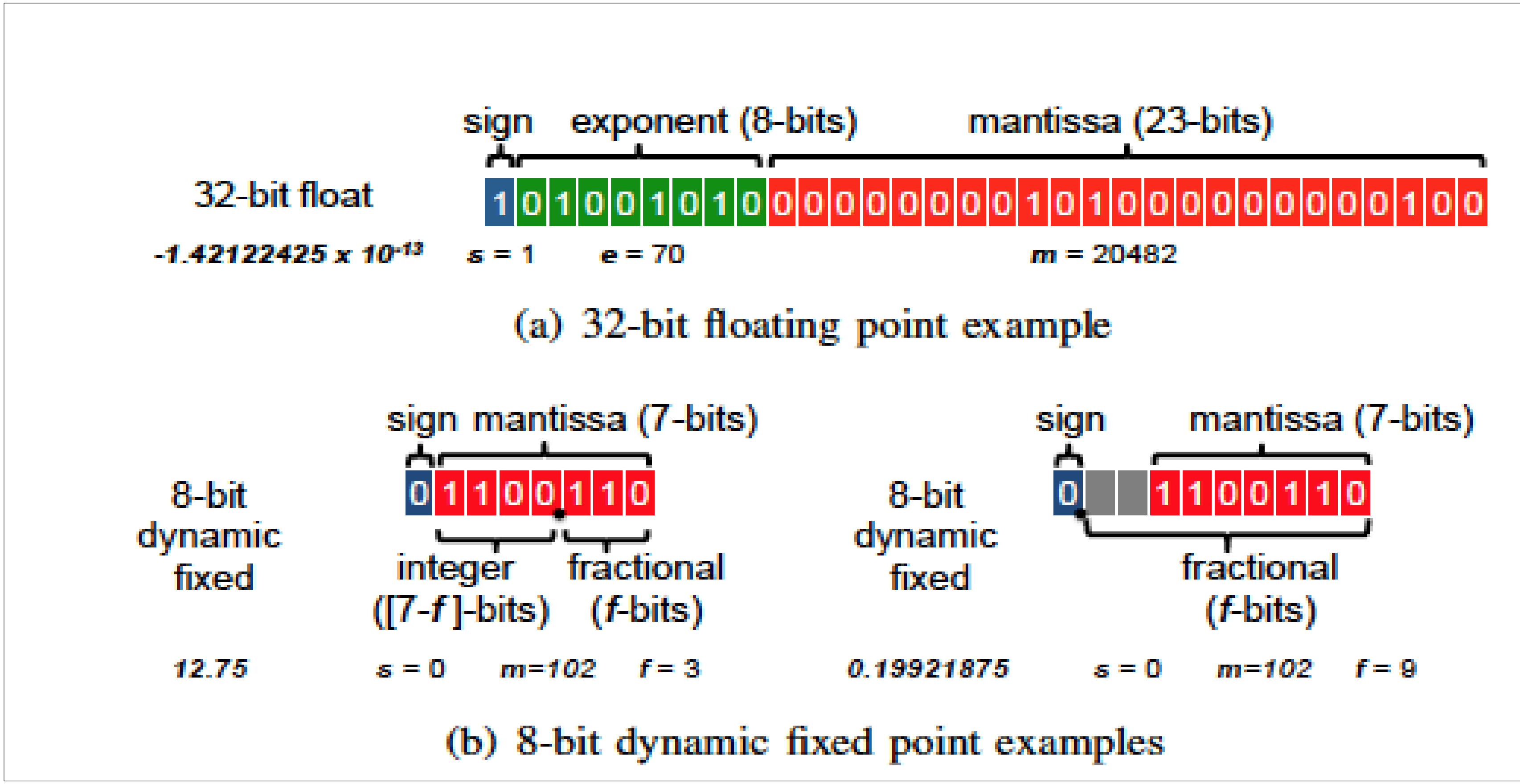
Stochastic rounding



Dynamic fixed point

Dynamic Fixed Point, where to put the point?

Range vs. precision



(a) 32-bit floating point example

(b) 8-bit dynamic fixed point examples

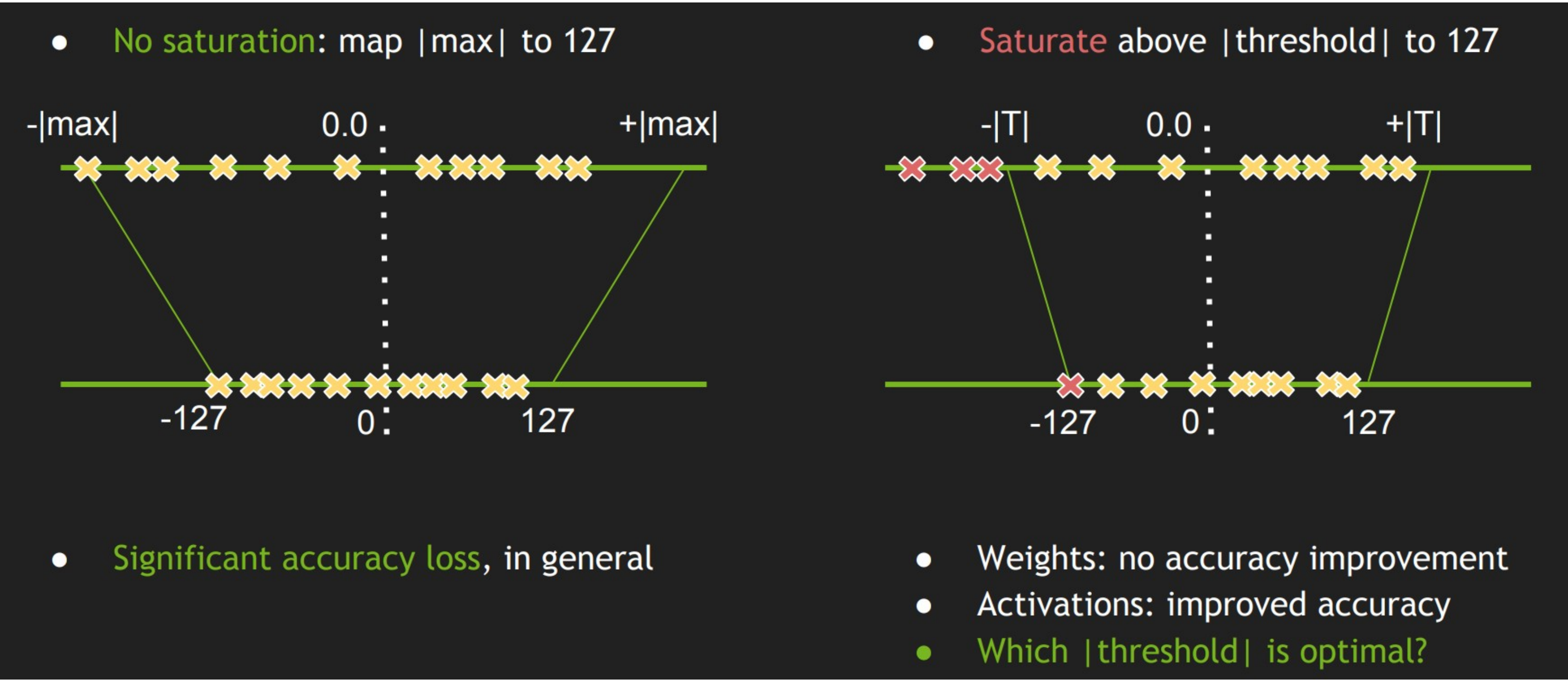
$$(-1)^s \times m \times 2^{(e-127)}$$

IEEE 754 standard floating point

$$(-1)^s \times m \times 2^{-f}$$

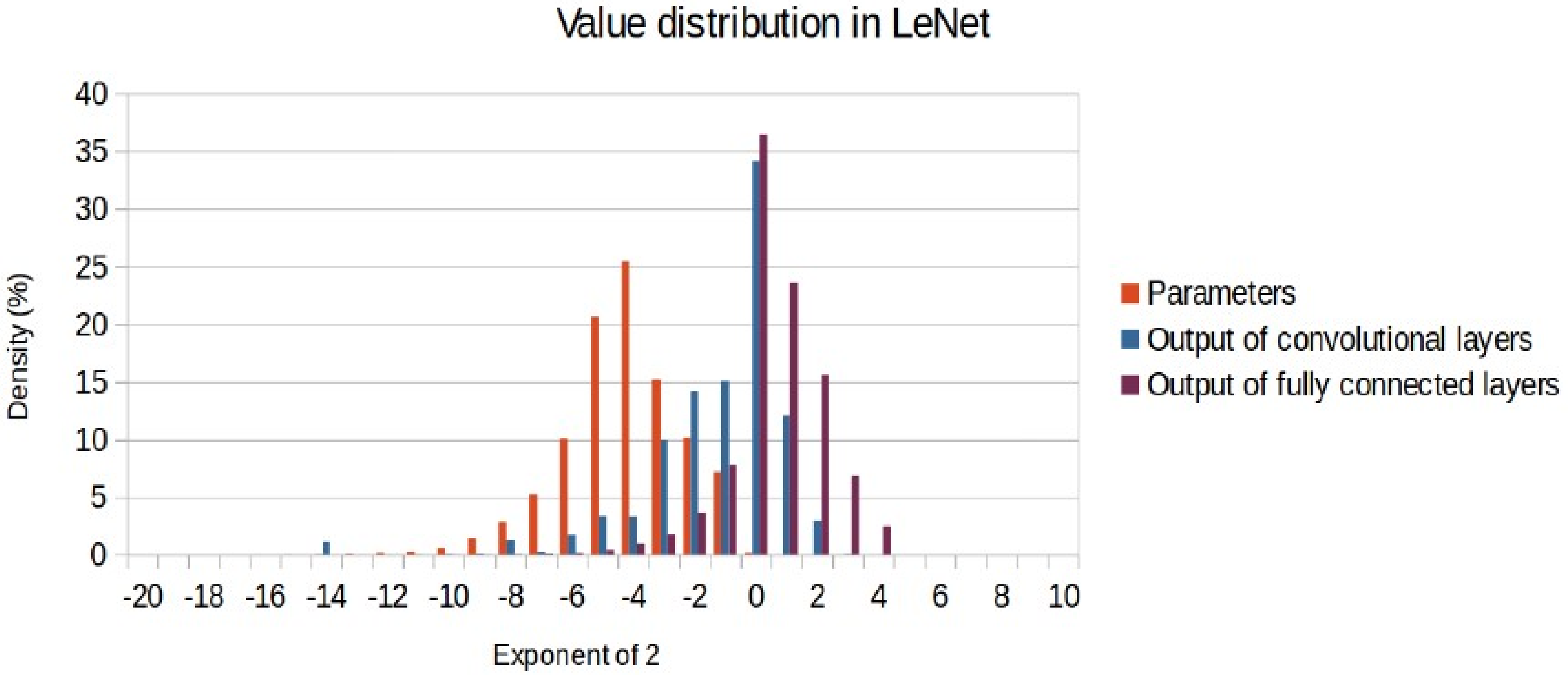
Dynamic Fixed point

TensorRT: where to saturate?

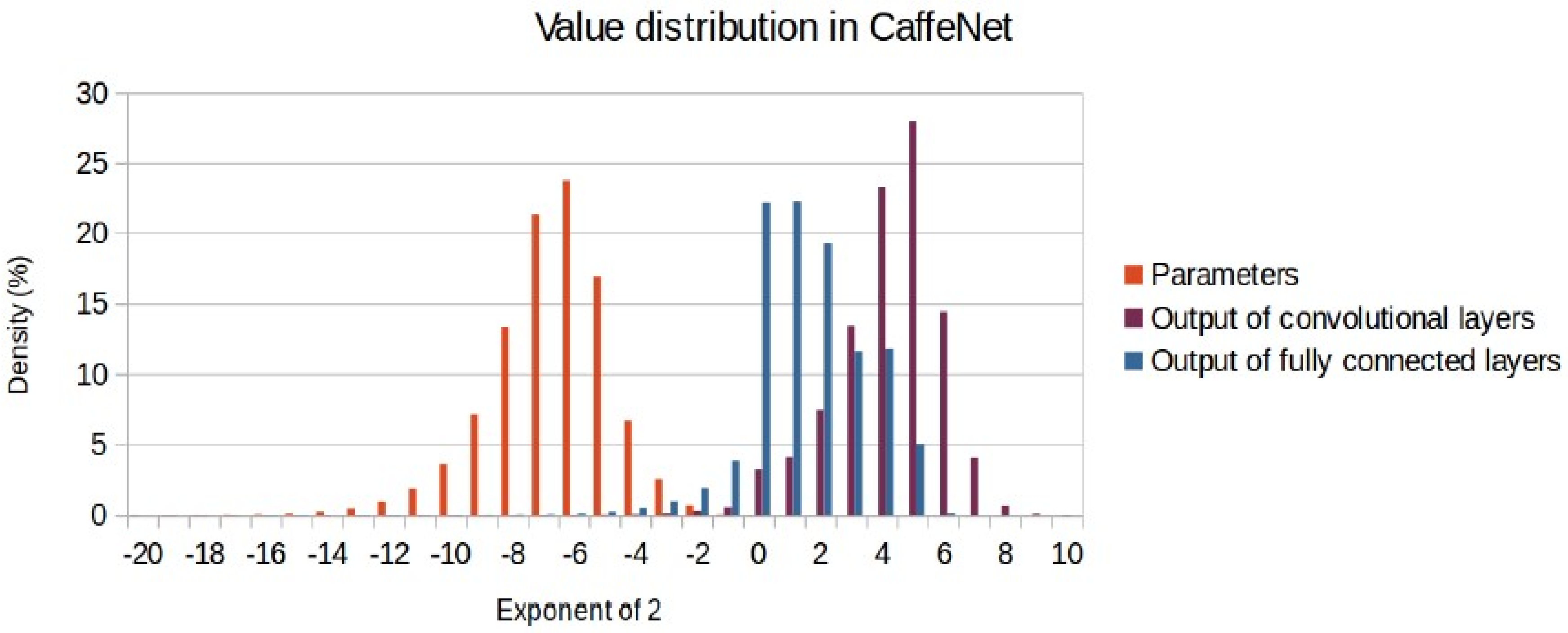


Symmetric linear quantization

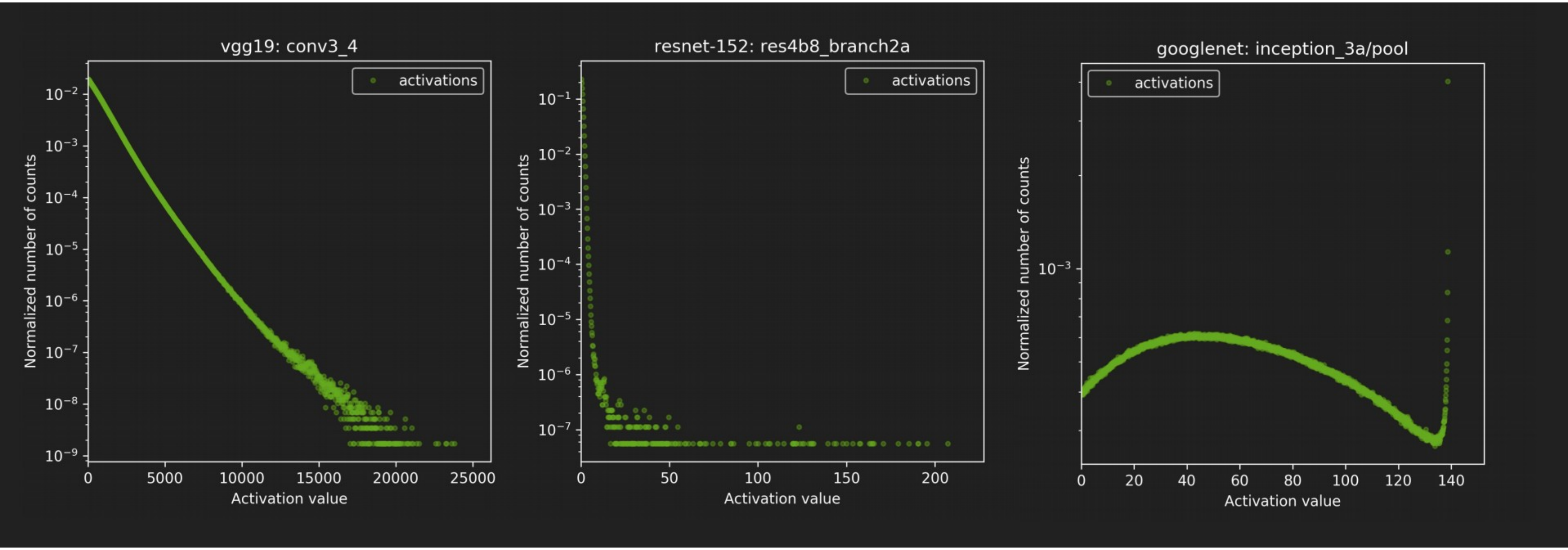
Empirical observations: Dynamic range of parameters and activations in LENET



Empirical observations: Dynamic range of weights and parameters in CaffeNet



TensorRT: empirical observations

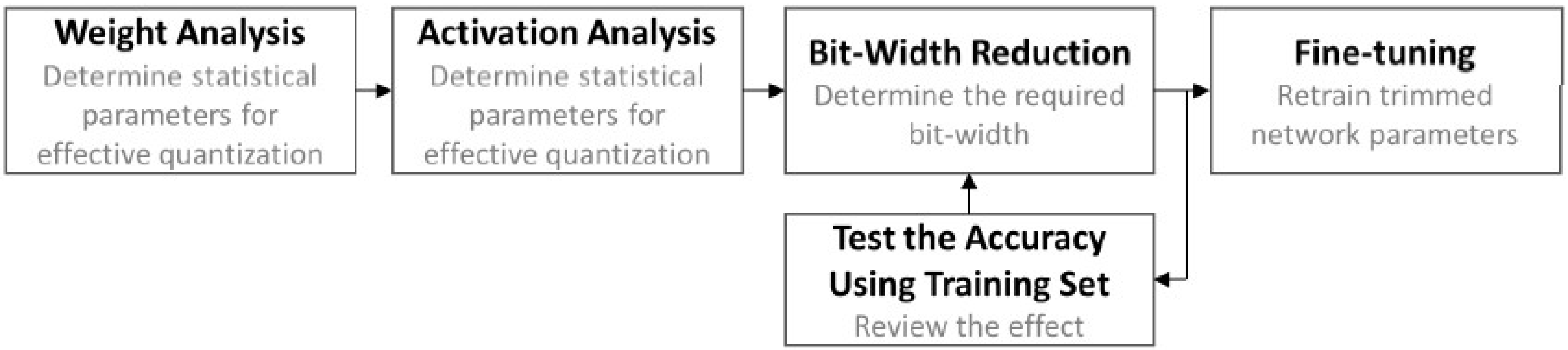


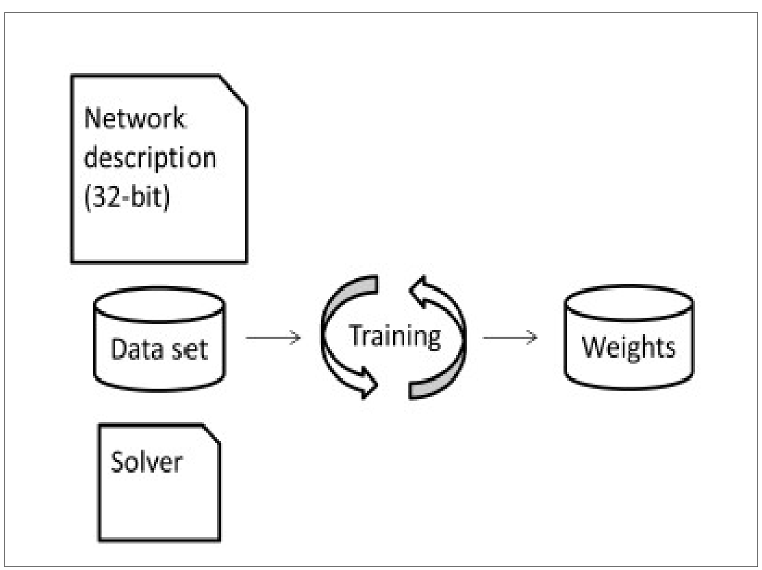
Empirical Observations

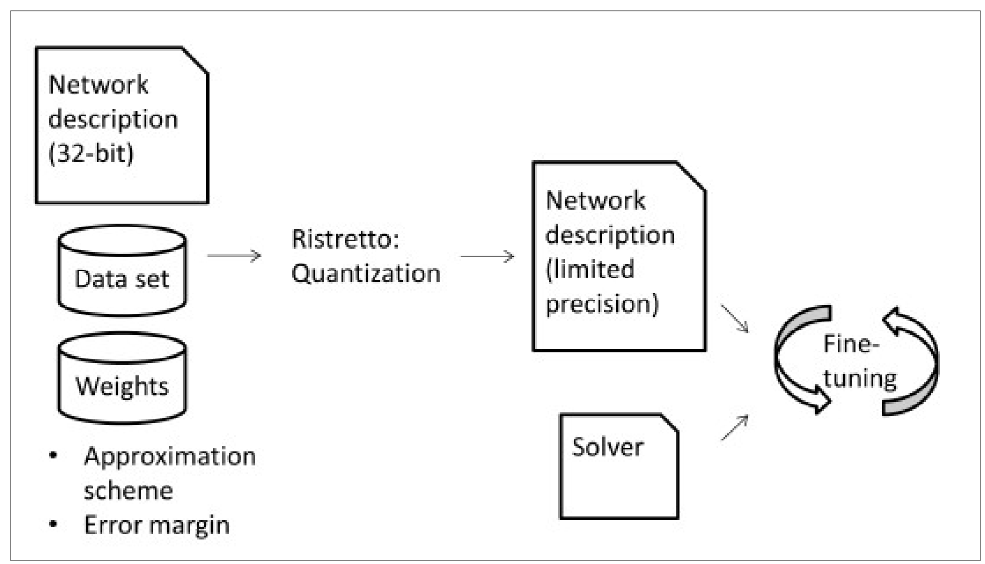
- Different networks have different range of weights and activations
- Weights and activations within same network have different numerical ranges

Network	Layer outputs	CONV parameters	FC parameters	32-bit baseline	Fixed point accuracy
LeNet (Exp 1)	4-bit	4-bit	4-bit	99.15%	98.95% (98.72%)
LeNet (Exp 2)	4-bit	2-bit	2-bit	99.15%	98.81% (98.03%)
Full CIFAR-10	8-bit	8-bit	8-bit	81.69%	81.44% (80.64%)
CaffeNet	8-bit	8-bit	8-bit	56.90%	56.00% (55.77%)
SqueezeNet	8-bit	8-bit	8-bit	57.68%	57.09% (55.25%)
GoogLeNet	8-bit	8-bit	8-bit	68.92%	66.57% (66.07%)

Ristretto: a quantization tool built on Caffe



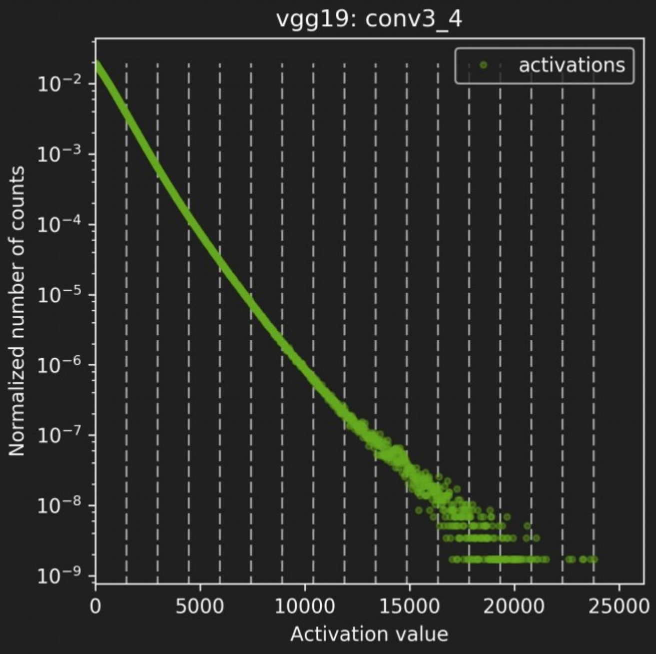




For TensorRT

- You will need:
  - Model trained in FP32.
  - Calibration dataset.
- TensorRT will:
  - Run inference in FP32 on calibration dataset.
  - Collect required statistics.
  - Run calibration algorithm → optimal scaling factors.
  - Quantize FP32 weights → INT8.
  - Generate “CalibrationTable” and INT8 execution engine.

For TensorRT



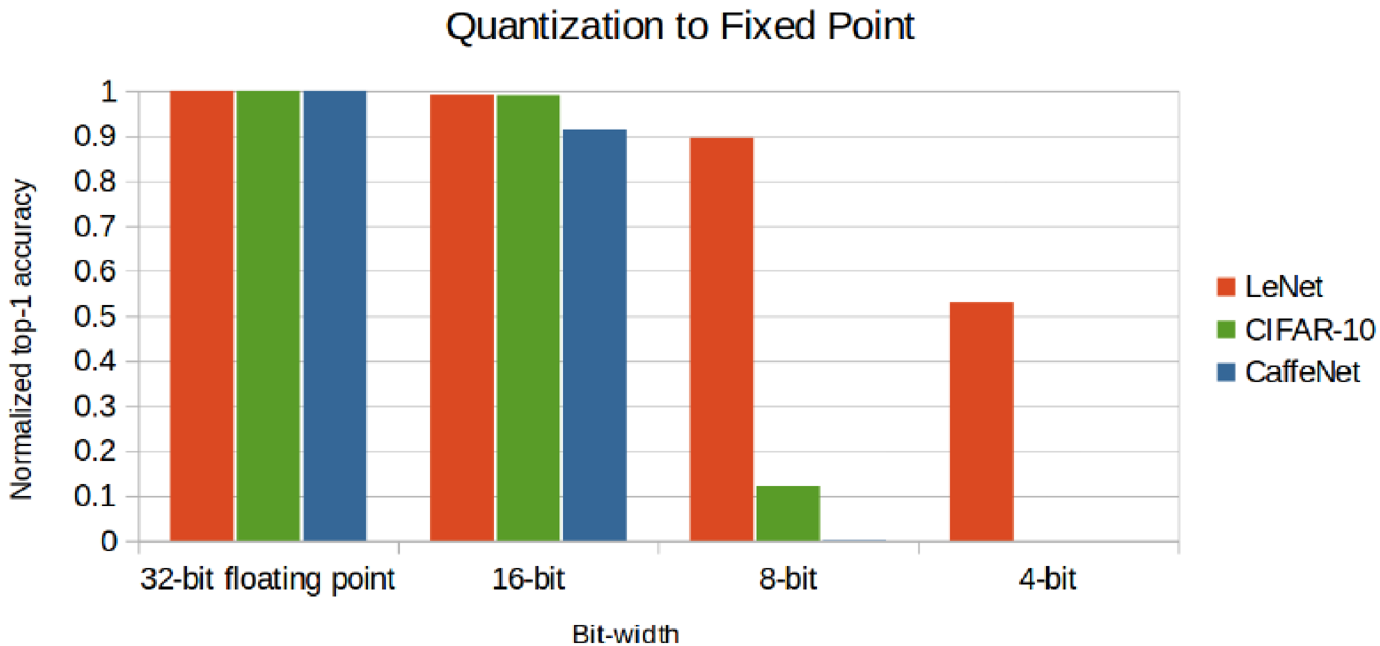
- Run FP32 inference on Calibration Dataset.
- For each Layer:
  - collect histograms of activations.
  - generate many quantized distributions with different saturation thresholds.
  - pick threshold which minimizes KL_divergence(ref_distr, quant_distr).
- Entire process takes a few minutes on a typical desktop workstation.

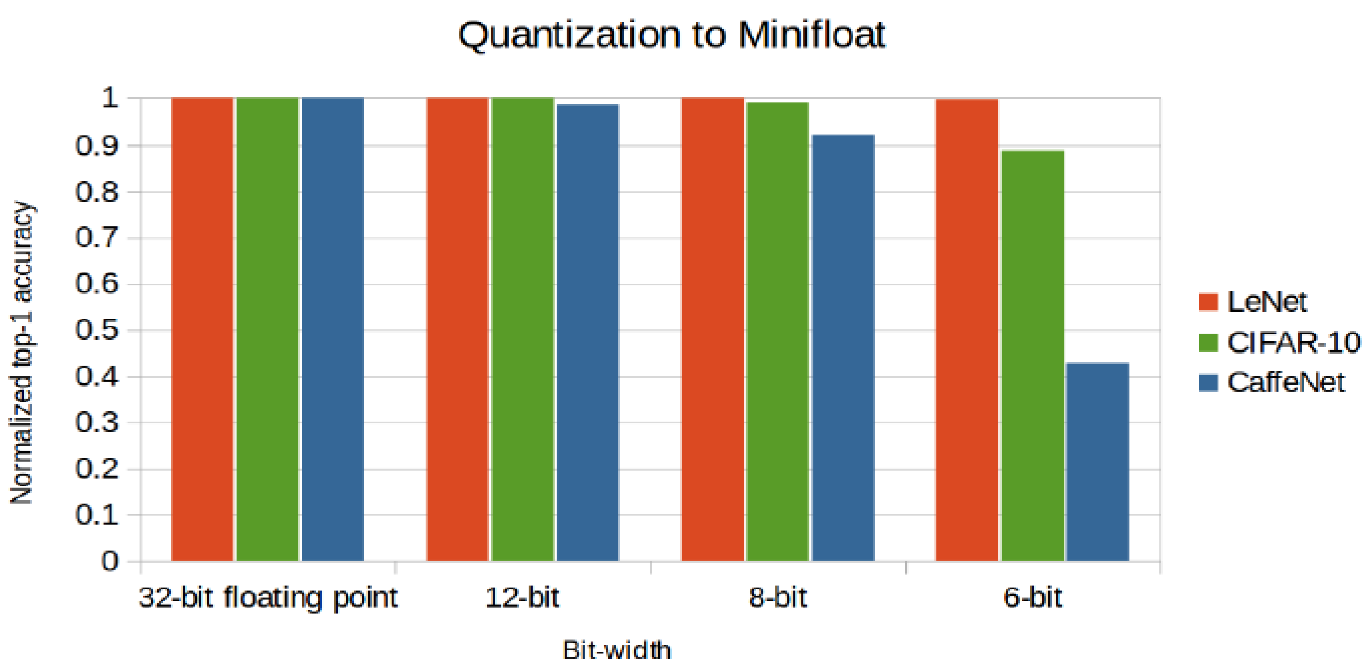
Different networks need different bit-widths and formats,
depends on quantization method

Network	Baseline accuracy	Fixed point bit-width	Fixed point format	Fixed point accuracy
LeNet	99.15%	8-bit	Q4.4	98.88% (88.90%)
CIFAR-10	81.69%	16-bit	Q8.8	81.38% (80.94%)
CaffeNet top-1	56.90%	16-bit	Q9.7	52.48% (52.13%)

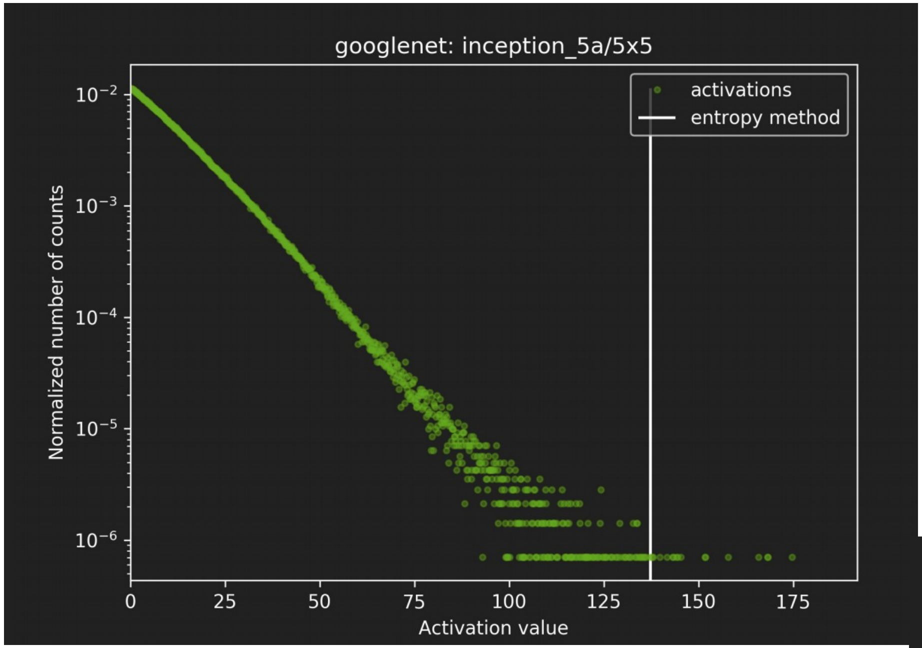
Network	32-bit accuracy	Minifloat bit-width	Minifloat accuracy	Exponent bits, mantissa bits
LeNet	99.15%	8-bit	99.20% (99.20%)	4-bit, 3-bit
CIFAR-10	81.69%	8-bit	80.85% (80.47%)	5-bit, 2-bit
CaffeNet top-1	56.90%	8-bit	52.52% (52.30%)	5-bit, 2-bit

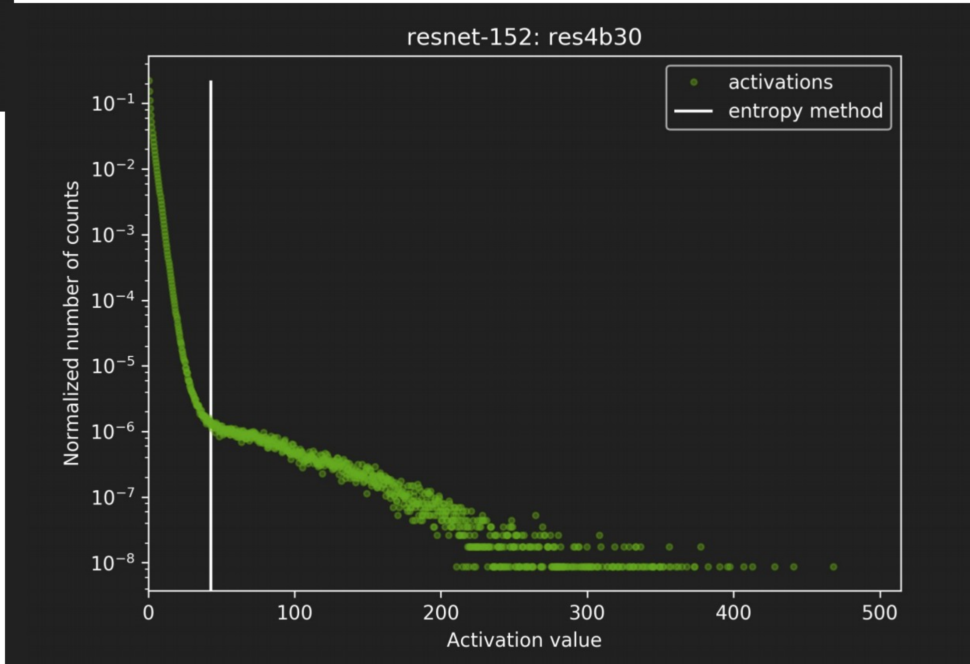
Different networks need different bit-widths (depends on quantization method)





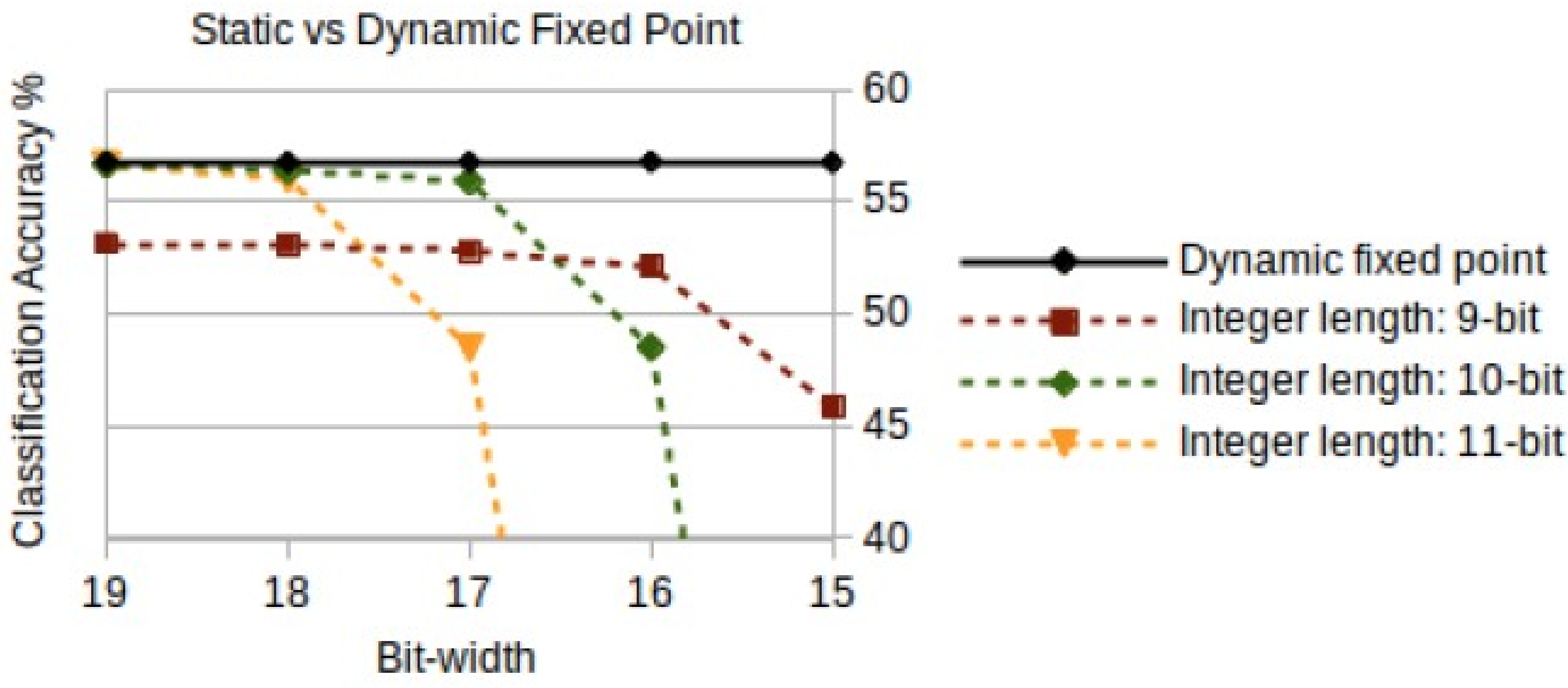
TensorRT: network specific thresholds



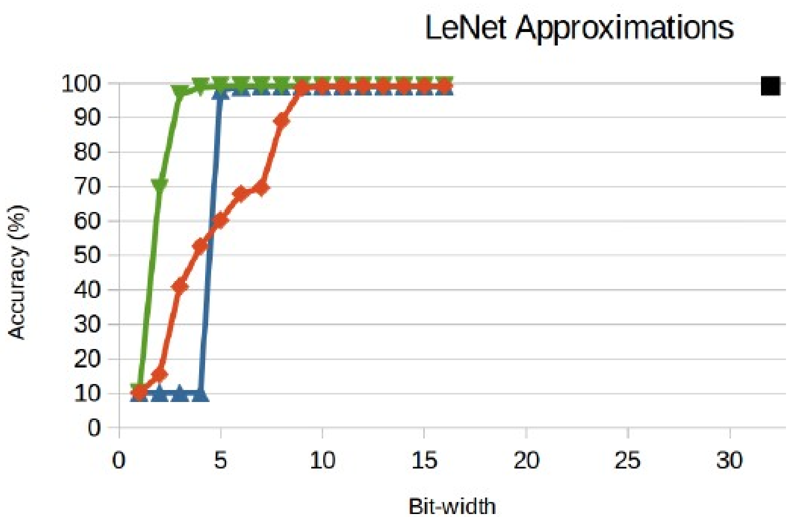


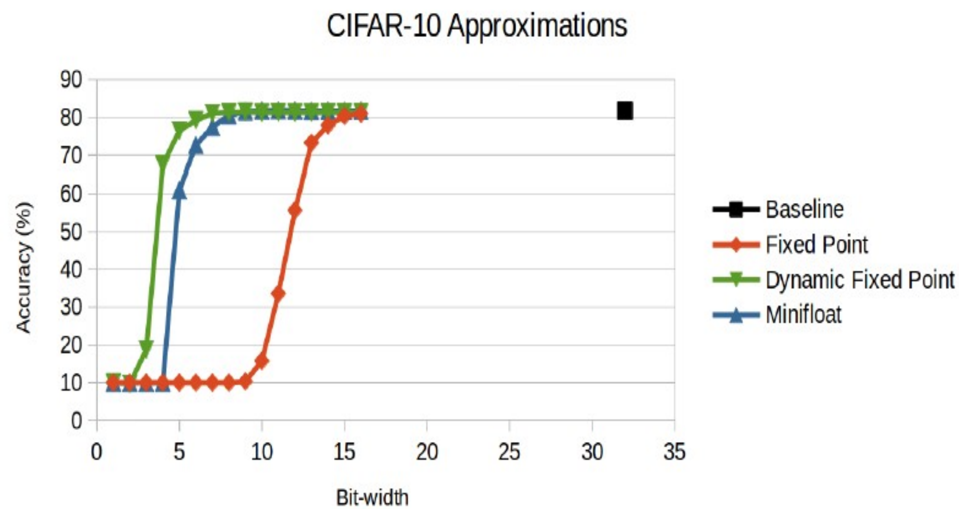
Dynamic fixed point better than static fixed point

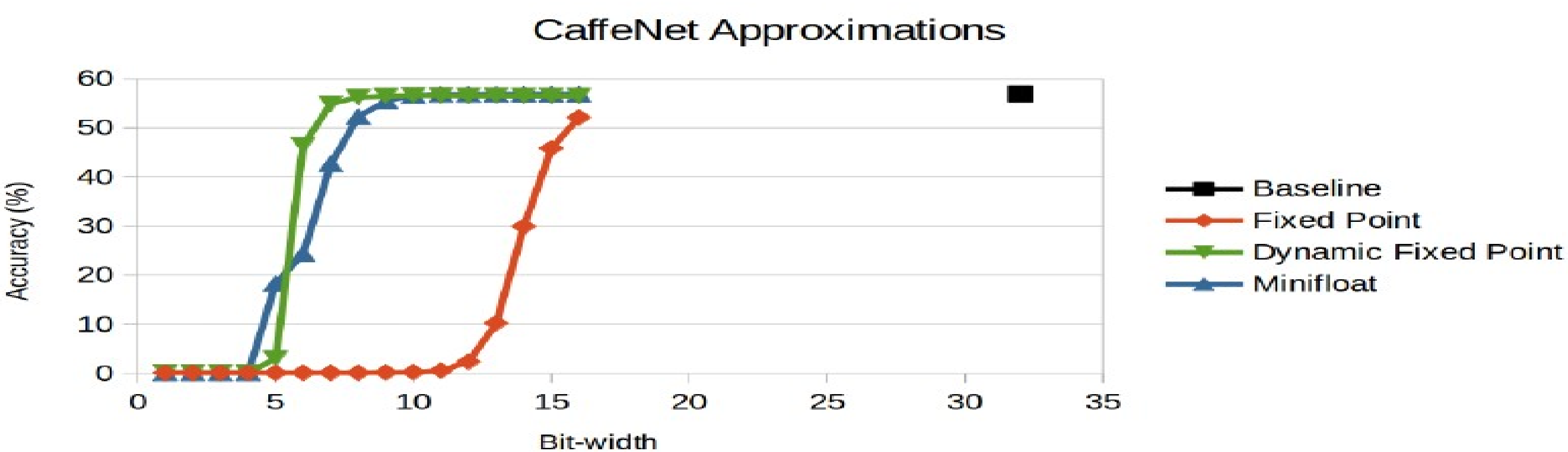
-> exploits different layers within the network having different numerical ranges



Dynamic fixed point performs best



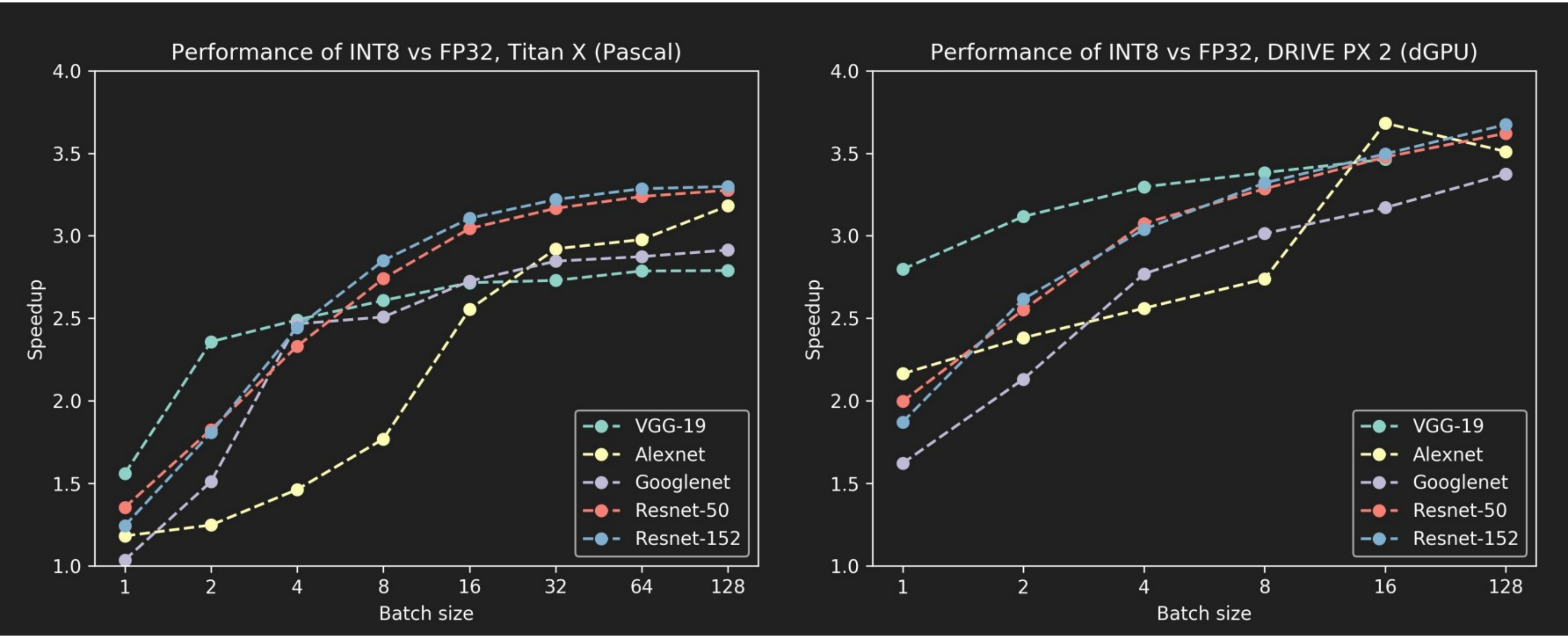




TensorRT accuracy

	FP32		INT8					
			Calibration using 5 batches		Calibration using 10 batches		Calibration using 50 batches	
NETWORK	**Top1**	**Top5**	**Top1**	**Top5**	**Top1**	**Top5**	**Top1**	**Top5**
Resnet-50	73.23%	91.18%	73.03%	91.15%	73.02%	91.06%	73.10%	91.06%
Resnet-101	74.39%	91.78%	74.52%	91.64%	74.38%	91.70%	74.40%	91.73%
Resnet-152	74.78%	91.82%	74.62%	91.82%	74.66%	91.82%	74.70%	91.78%
VGG-19	68.41%	88.78%	68.42%	88.69%	68.42%	88.67%	68.38%	88.70%
Googlenet	68.57%	88.83%	68.21%	88.67%	68.10%	88.58%	68.12%	88.64%
Alexnet	57.08%	80.06%	57.00%	79.98%	57.00%	79.98%	57.05%	80.06%
NETWORK	**Top1**	**Top5**	**Diff Top1**	**Diff Top5**	**Diff Top1**	**Diff Top5**	**Diff Top1**	**Diff Top5**
Resnet-50	73.23%	91.18%	**0.20%**	**0.03%**	**0.22%**	**0.13%**	**0.13%**	**0.12%**
Resnet-101	74.39%	91.78%	**-0.13%**	**0.14%**	**0.01%**	**0.09%**	**-0.01%**	**0.06%**
Resnet-152	74.78%	91.82%	**0.15%**	**0.01%**	**0.11%**	**0.01%**	**0.08%**	**0.05%**
VGG-19	68.41%	88.78%	**-0.02%**	**0.09%**	**-0.01%**	**0.10%**	**0.03%**	**0.07%**
Googlenet	68.57%	88.83%	**0.36%**	**0.16%**	**0.46%**	**0.25%**	**0.45%**	**0.19%**
Alexnet	57.08%	80.06%	**0.08%**	**0.08%**	**0.08%**	**0.07%**	**0.03%**	**-0.01%**

TensorRT performance



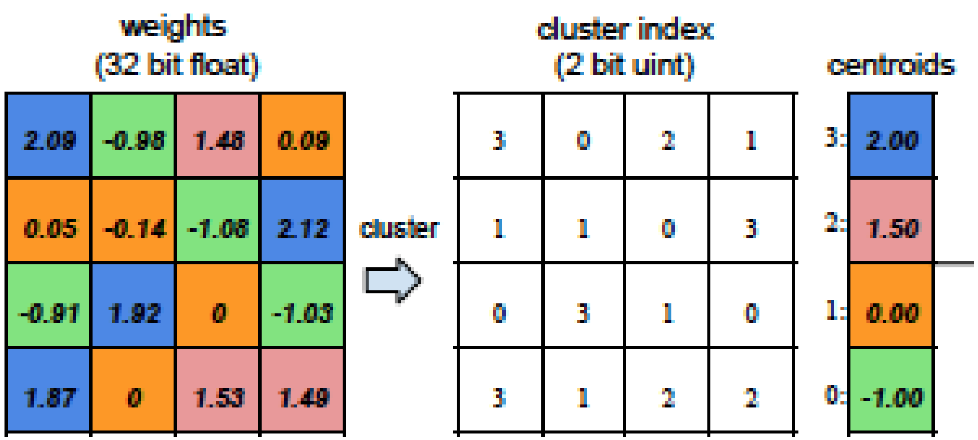
Deep Compression: Compressing Deep Neural Networks with Pruning, **Trained Quantization** and Huffman Coding

Song Han, Huizi Mao, William J. Dally

ICLR 2016

K-means clustering based quantization

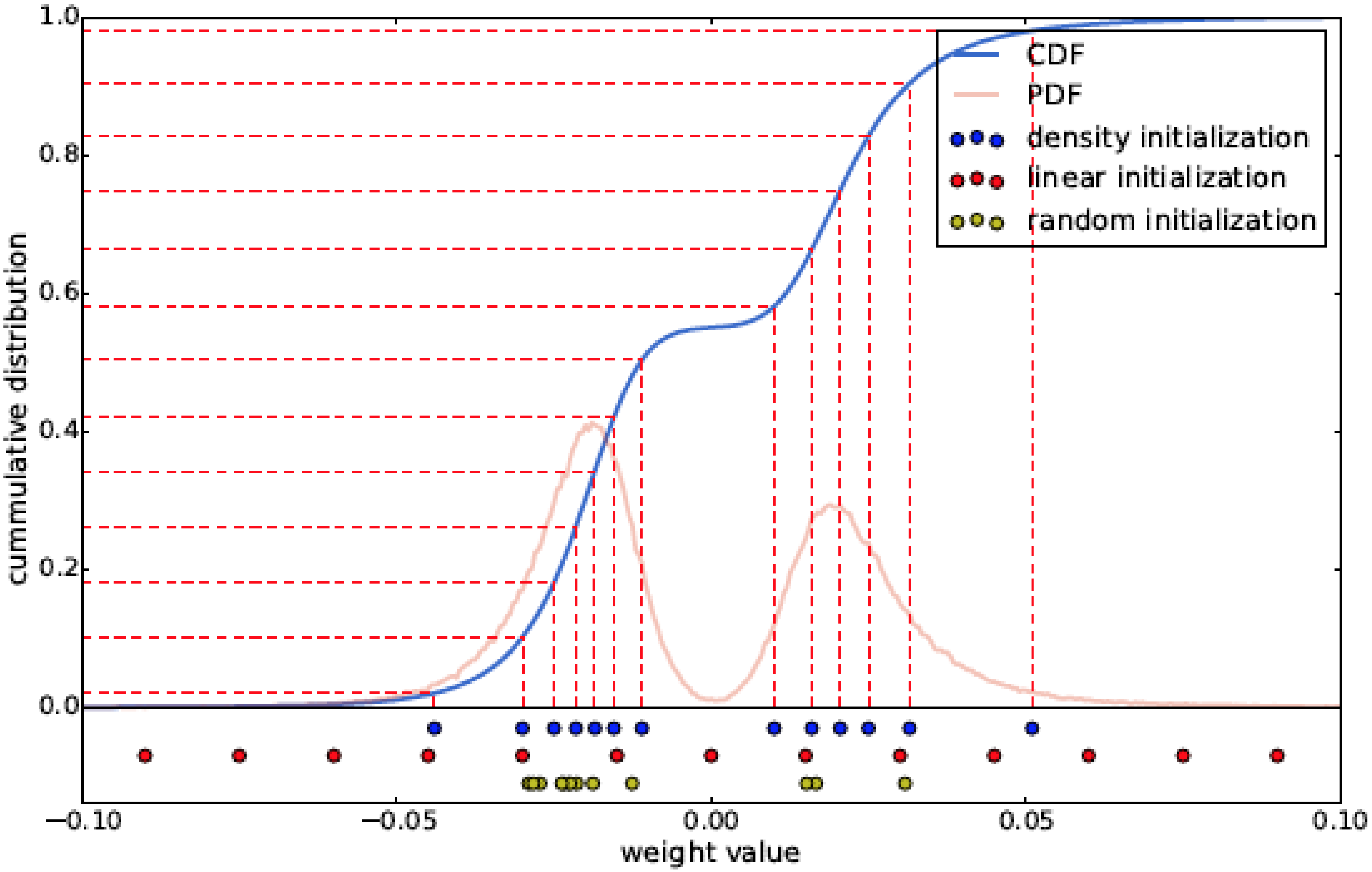
- Bit-width representation of each weight doesn't change, so each weight is still the same size
- Each weight represented by the mean value of the cluster it belongs to
- Number of unique weights change, storage requirements reduce due to this weight sharing



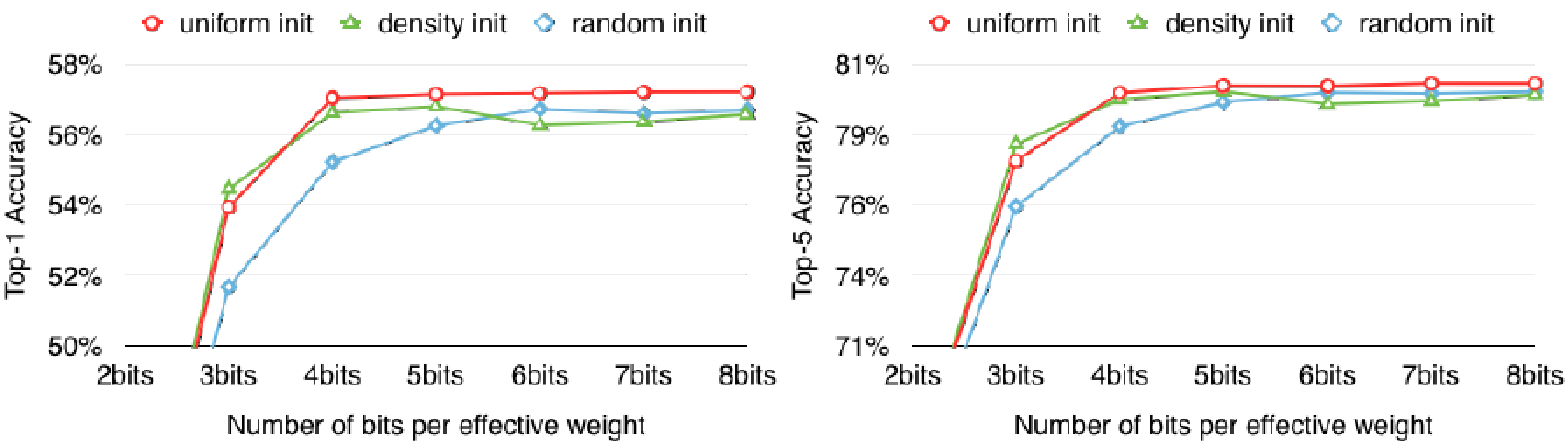
$$\arg\min_{C} \sum_{i=1}^{k} \sum_{w \in c_i} |w - c_i|^2$$

Minimizes within cluster sum of squares

Cluster initialization

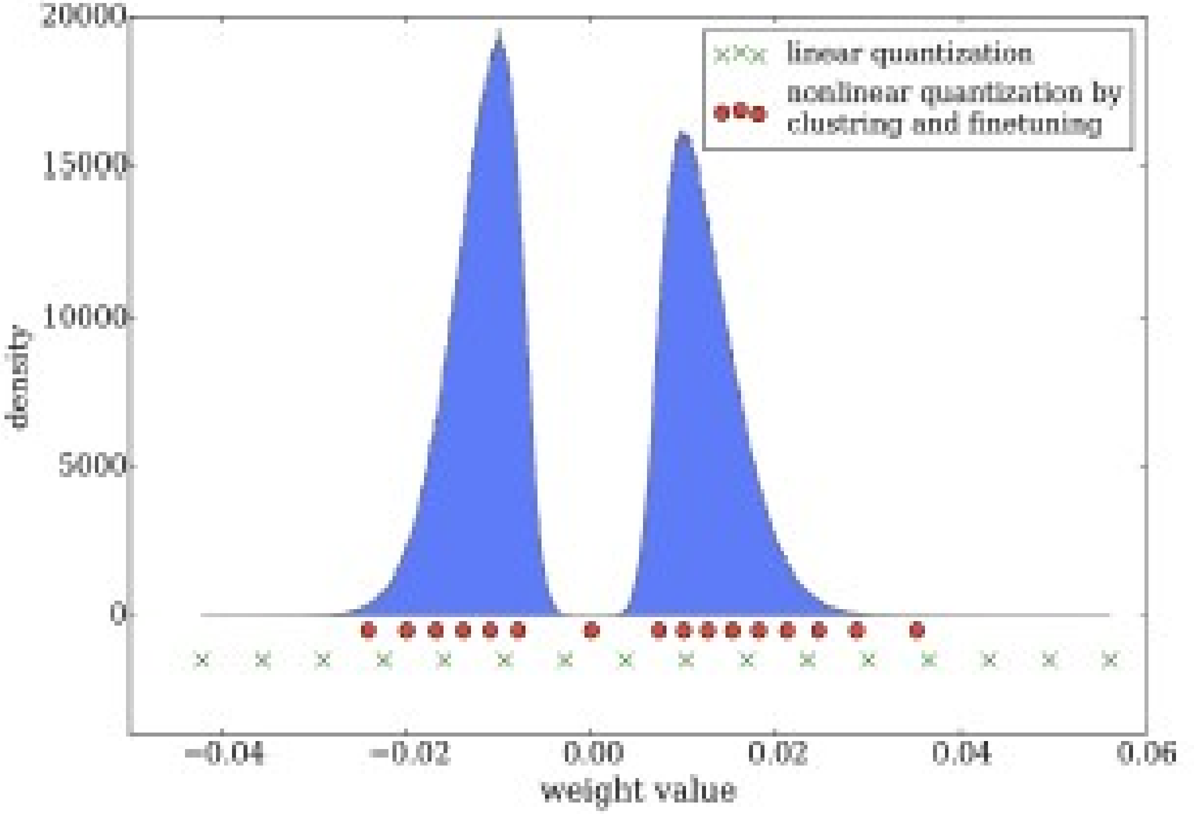


Linear Initialization gives best results



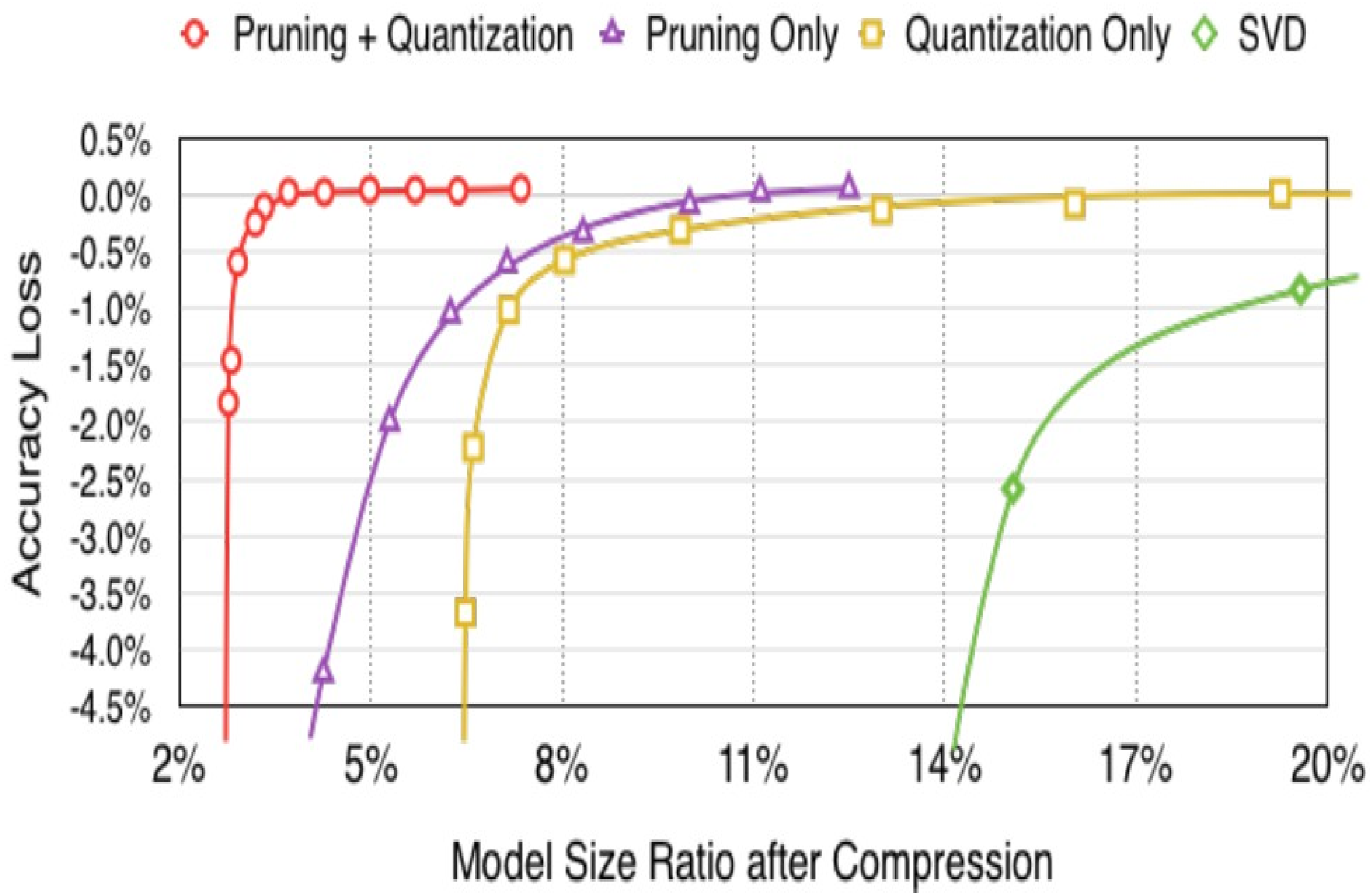
Larger weights play a more important role than smaller weights (Han et al., 2015), but there are fewer of these large weights. Thus for both random initialization and density-based initialization, very few centroids have large absolute value which results in poor representation of these few large weights. Linear initialization does not suffer from this.

Quantization after pruning

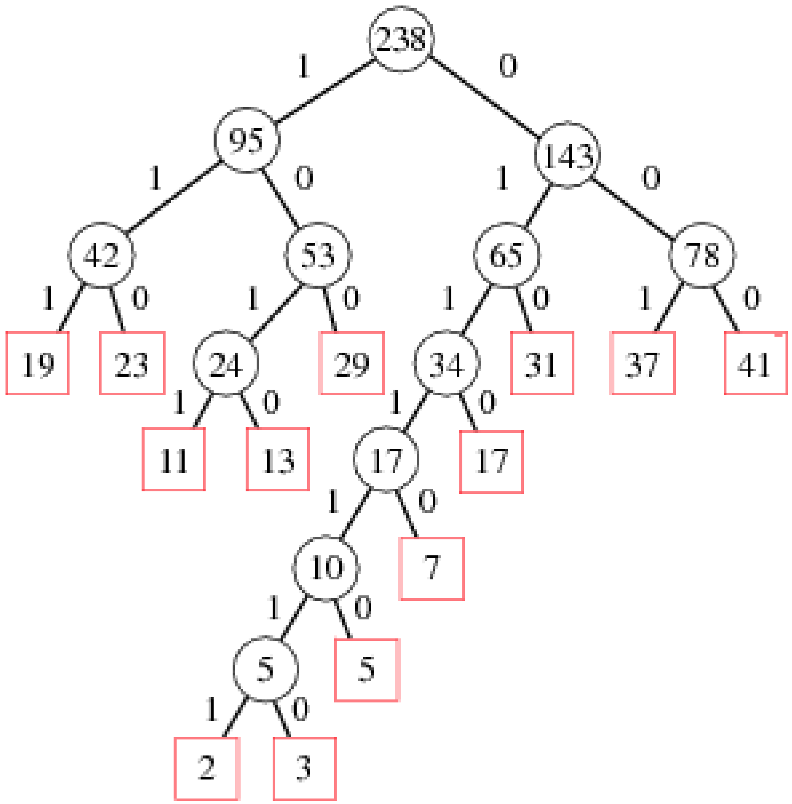


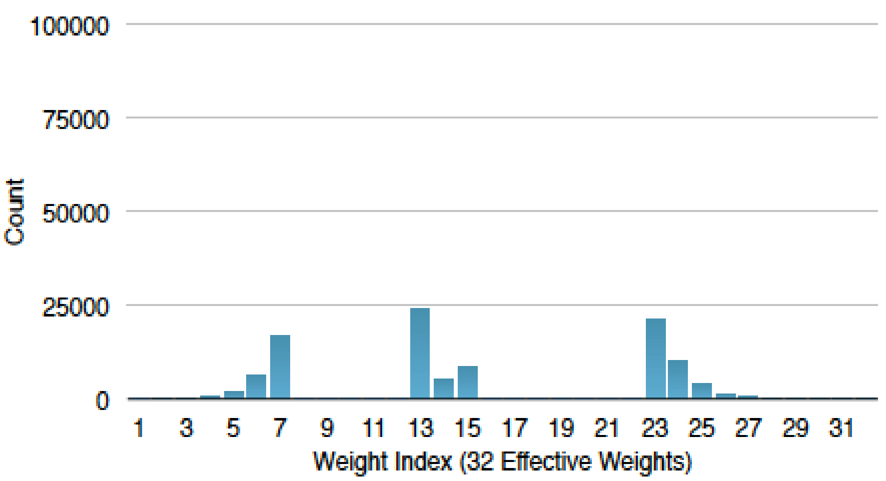
Unsupervised K-Means clustering of weights benefits from pruning.
Pruning retains more important weights, that can be grouped into more meaningful clusters.

Benefits of pruning



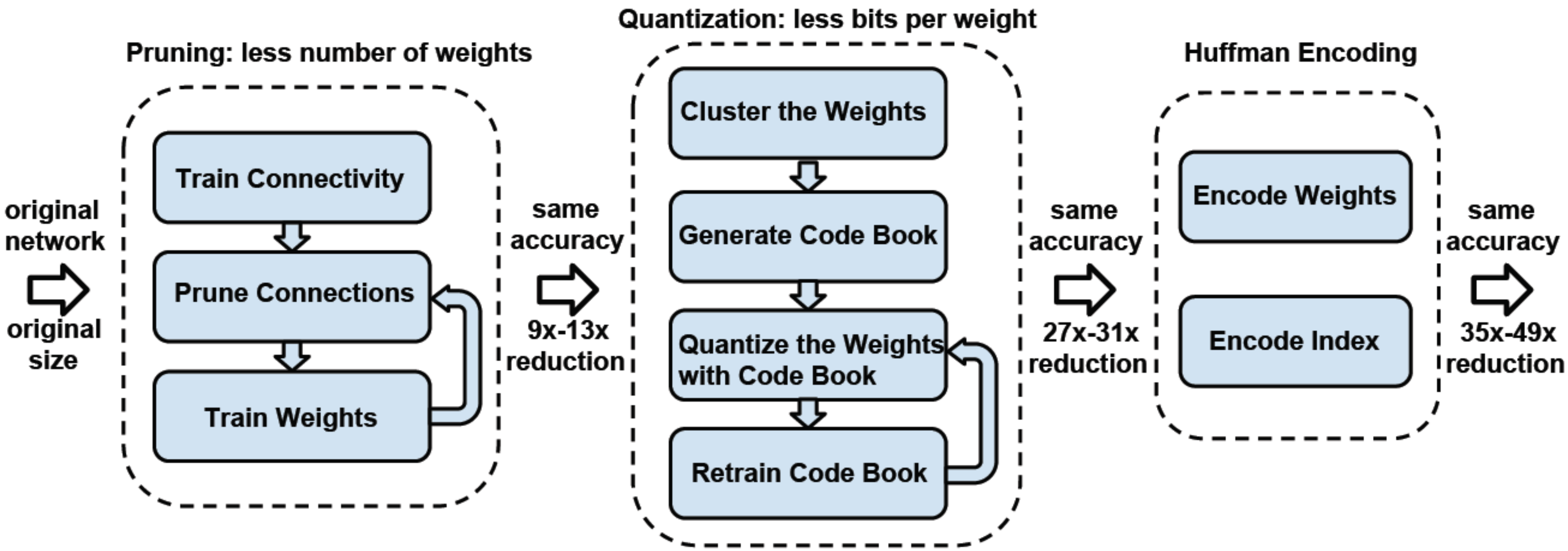
Huffman encoding



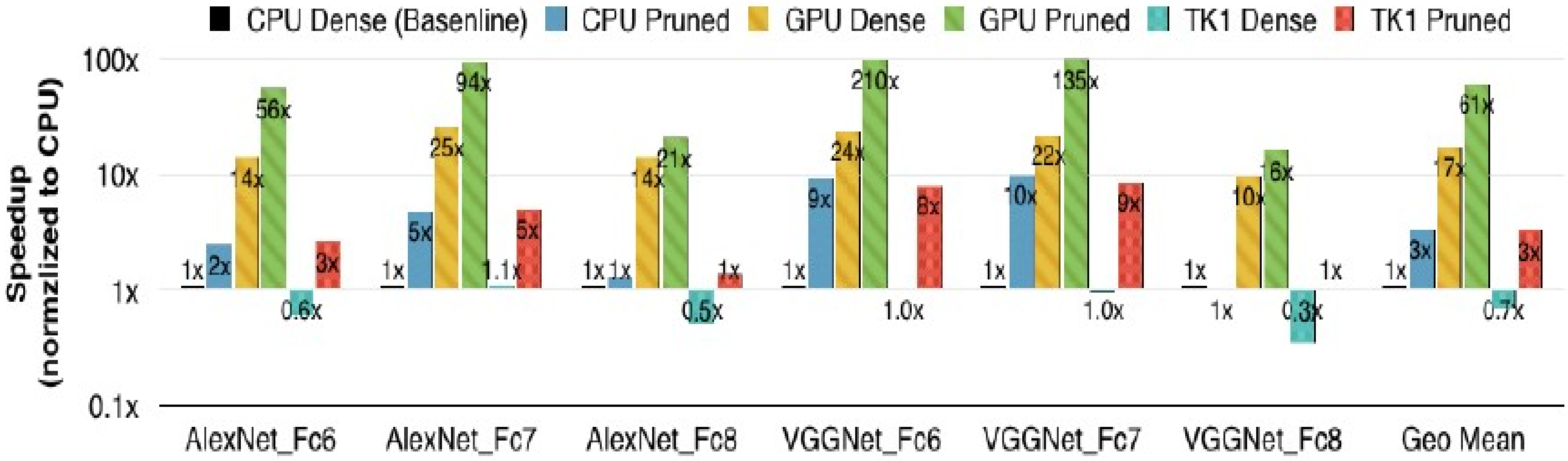


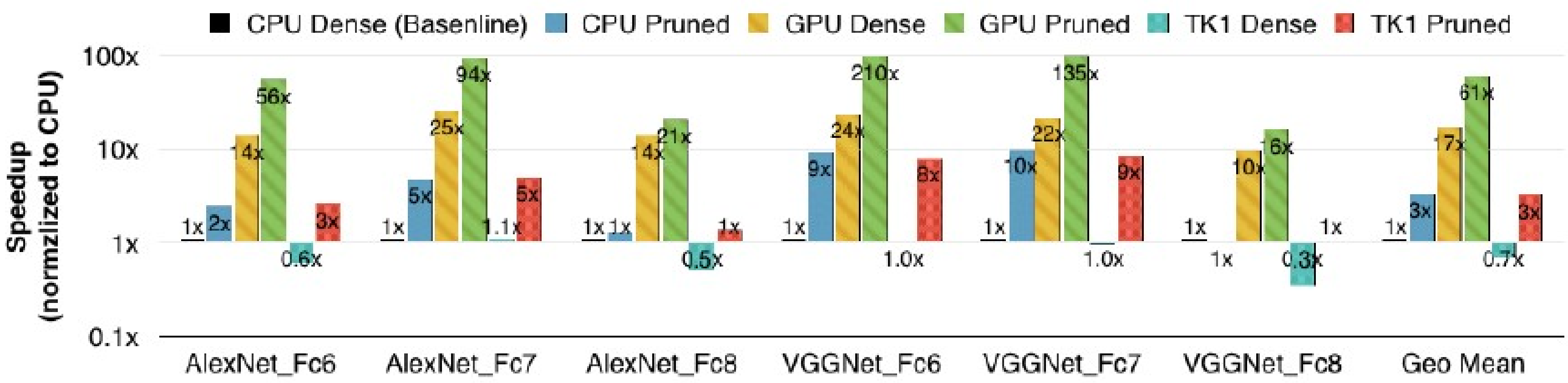
Opportunity for Huffman Encoding, as seen from the empirical values of weights

Overall flow



FC layer performance improvements





Accuracy and data storage

-> network weights and activations/ quantization bit widths.

Latency/energy

-> hardware platform

(CPU with SIMD vs. GPU vs. FPGA? has floating point co-processor unit? 8-bit vs. 16 bit vs. 32 bit addressing ……)

Floating vs. fixed point performance, hardware dependency

(1) http://www.ti.com/lit/wp/spry061/spry061.pdf

(2) https://www.eetimes.com/document.asp?doc_id=1275364

(3) https://stackoverflow.com/questions/25351114/is-fixed-point-math-faster-than-floating-point-on-armv7-a

(4) https://community.arm.com/processors/f/discussions/1466/neon-fixed-point-coding-and-fixed-vs-floating-point-operations-performance-comparison

(5) challenges of fixed point: http://www.artist-embedded.org/docs/Events/2009/EmbeddedControl/SLIDES/FixPoint.pdf slide 20

(6) https://blogs.mentor.com/colinwalls/blog/2012/09/10/the-floating-point-argument/

(7) https://dsp.stackexchange.com/questions/4835/relative-merits-of-fixed-point-vs-floating-point-computation