DADS7305: MLOPs

Northeastern University

Instructor: Ramin Mohammadi

September 7, 2025

These materials have been prepared and sourced for the course **MLOPs** at Northeastern University. Every effort has been made to provide proper citations and credit for all referenced works.

If you believe any material has been inadequately cited or requires correction, please contact me at:

`r.mohammadi@northeastern.edu`

Thank you for your understanding and collaboration.

Model Resource Management Techniques

Dimensionality Reduction

Dimensionality Effect on Performance

High-dimensional data

Before... when it was all about data mining

- Domain experts selected features
- Designed feature transforms
- Small number of more relevant features were enough

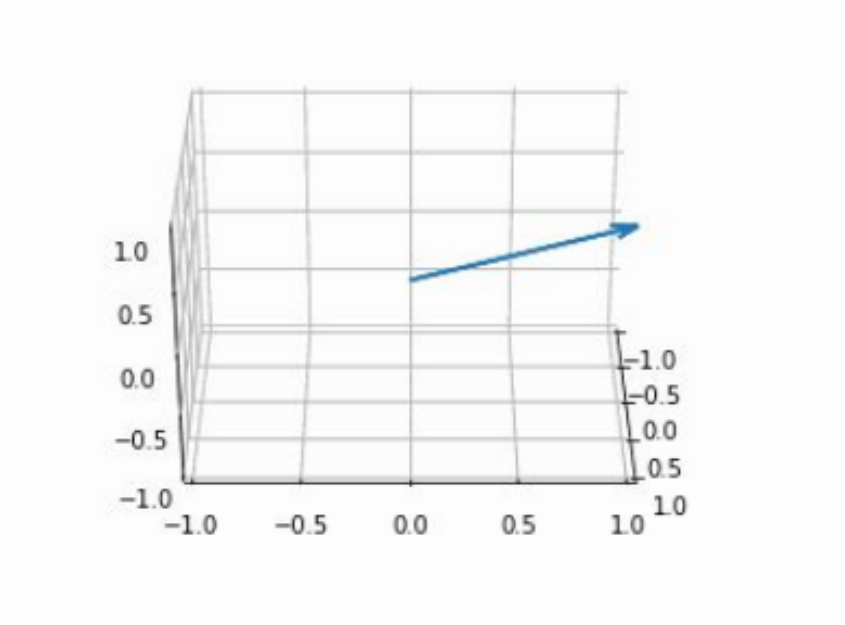
Now ... data science is about integrating everything

- Data generation and storage is less of a problem
- Squeeze out the best from data
- More high-dimensional data having more features

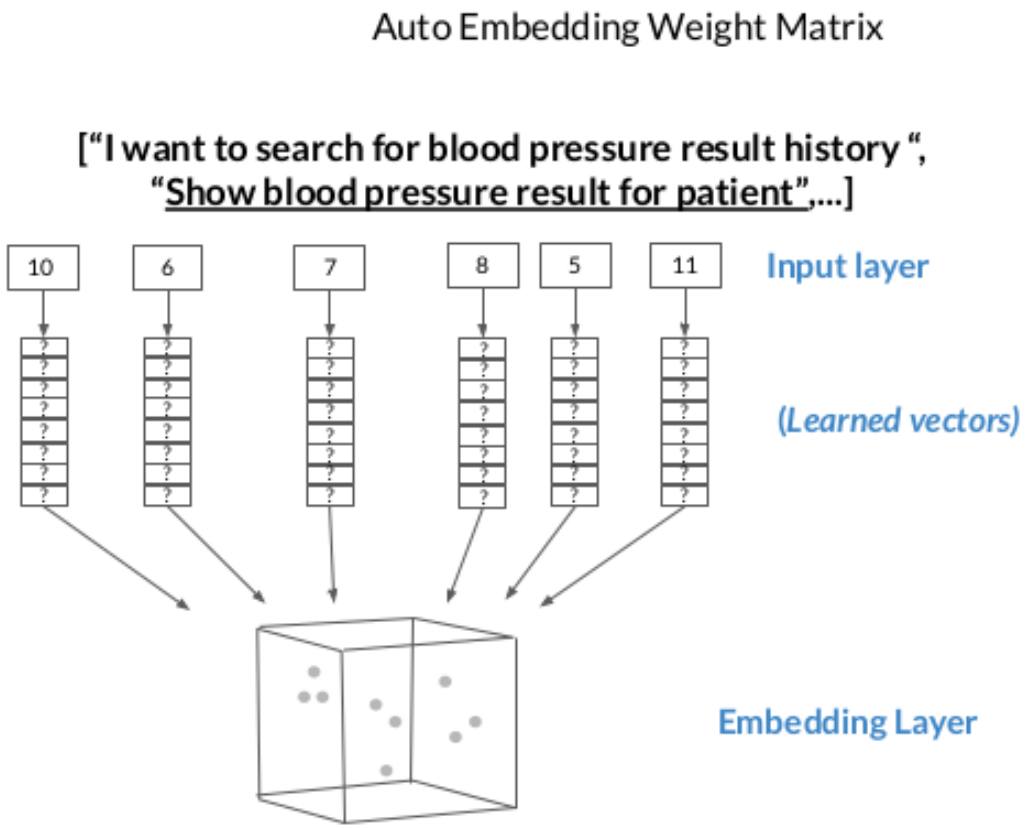
A note about neural networks

- Yes, neural networks will perform a kind of automatic feature selection
- However, that's not as efficient as a well-designed dataset and model
  - Much of the model can be largely "shut off" to ignore unwanted features
  - Even unused parts of the model consume space and compute resources
  - Unwanted features can still introduce unwanted noise
  - Each feature requires infrastructure to collect, store, and manage

High-dimensional spaces



Word embedding - An example



i	1
want	2
to	3
search	4
for	5
blood	6
pressure	7
result	8
history	9
show	10
patient	11
...	
LAST	20

Dimensionality Reduction

Curse of dimensionality

Why is high-dimensional data a problem?

- More dimensions $\rightarrow$ more features
- Risk of overfitting our models
- Distances grow more and more alike
- No clear distinction between clustered objects
- Concentration phenomenon for Euclidean distance

Curse of dimensionality

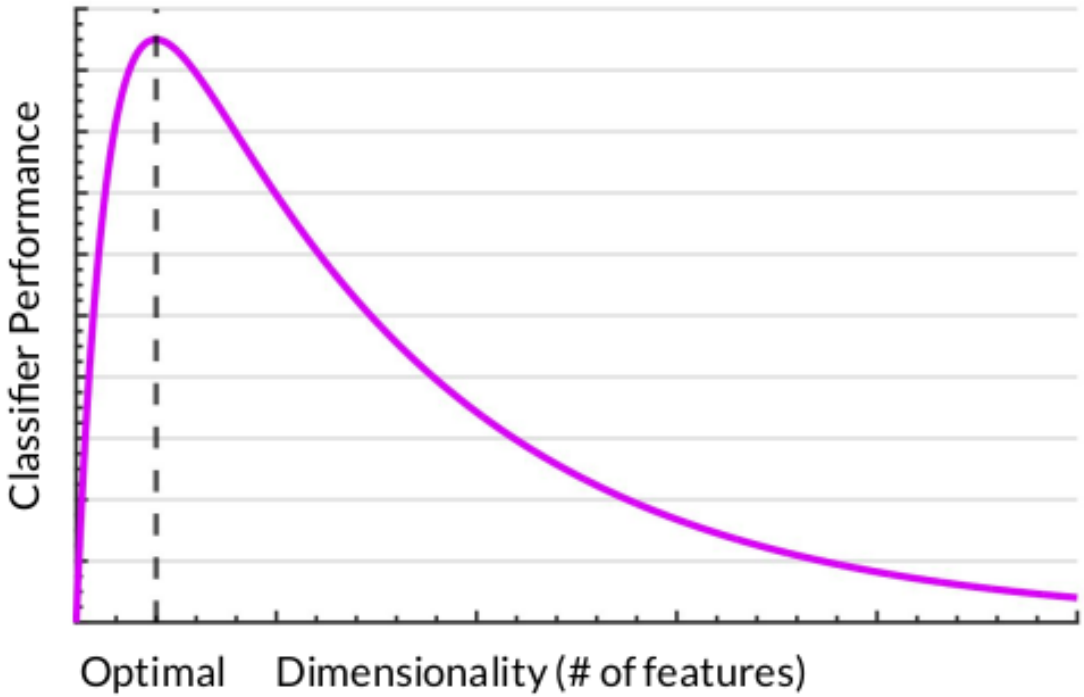
"As we add more dimensions we also increase the processing power we need to train the model and make predictions, as well as the amount of training data required"

— Badreesh Shetty

Why are more features bad?

- Redundant / irrelevant features
- More noise added than signal
- Hard to interpret and visualize
- Hard to store and process data

The performance of algorithms the number of dimensions



Curse of dimensionality in the distance function

1-D

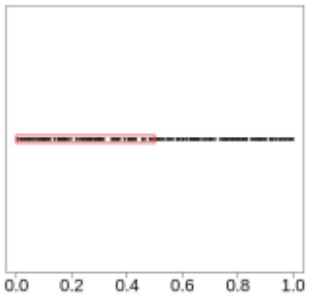
1	2	3	4	5

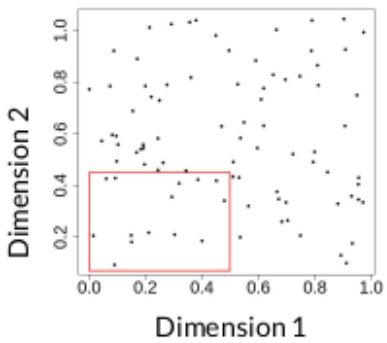
2-D

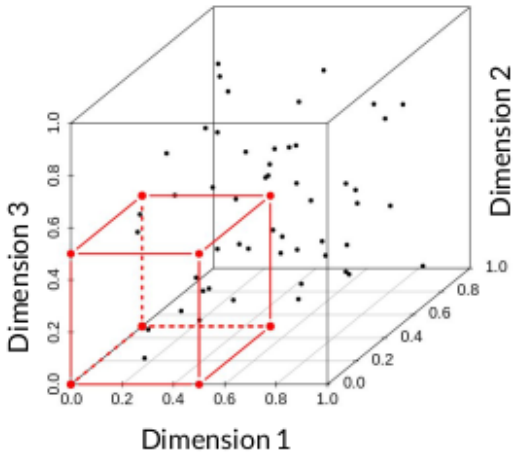
(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)
(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)
(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)
(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)
(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)

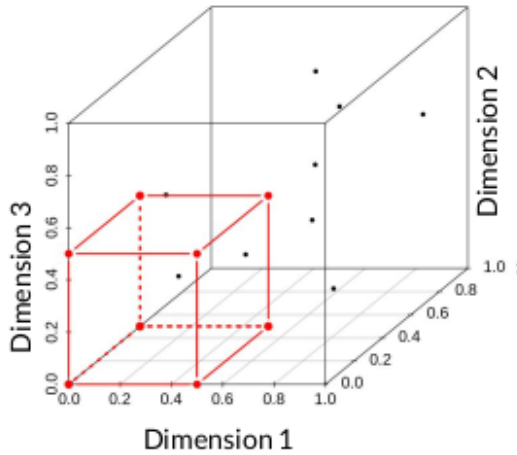
... ...

Increasing sparsity with higher dimensions









The Hughes effect

The more the features, the larger the hypothesis space

The lower the hypothesis space

- the easier it is to find the correct hypothesis
- the less examples you need

Dimensionality Reduction

Manual Dimensionality

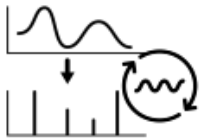
Increasing predictive performance

- Features must have information to produce correct results
- Derive features from inherent features
- Extract and recombine to create new features

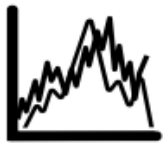
Feature explosion

Initial features

pixels, contours, textures, etc.



samples, spectrograms, etc.



ticks, trends, reversals, etc.

dna, marker sequences, genes, etc.

words, grammatical classes and relations, etc.

Combining features

- Number of features grows very quickly
- Reduce dimensionality

Why reduce dimensionality?

Storage

Computational

Consistency

Visualization

Major techniques for dimensionality reduction

Engineering

Selection

Feature Engineering

Need for manually crafting features

Certainly provides food for thought

Engineer features

- Tabular - aggregate, combine, decompose
- Text-extract context indicators
- Image-prescribe filters for relevant structures

It's an iterative process

Come up with ideas to construct "better" features

Devising features to reduce dimensionality

Select the right features to maximize predictiveness

Evaluate models using chosen features

Manual Dimensionality Reduction

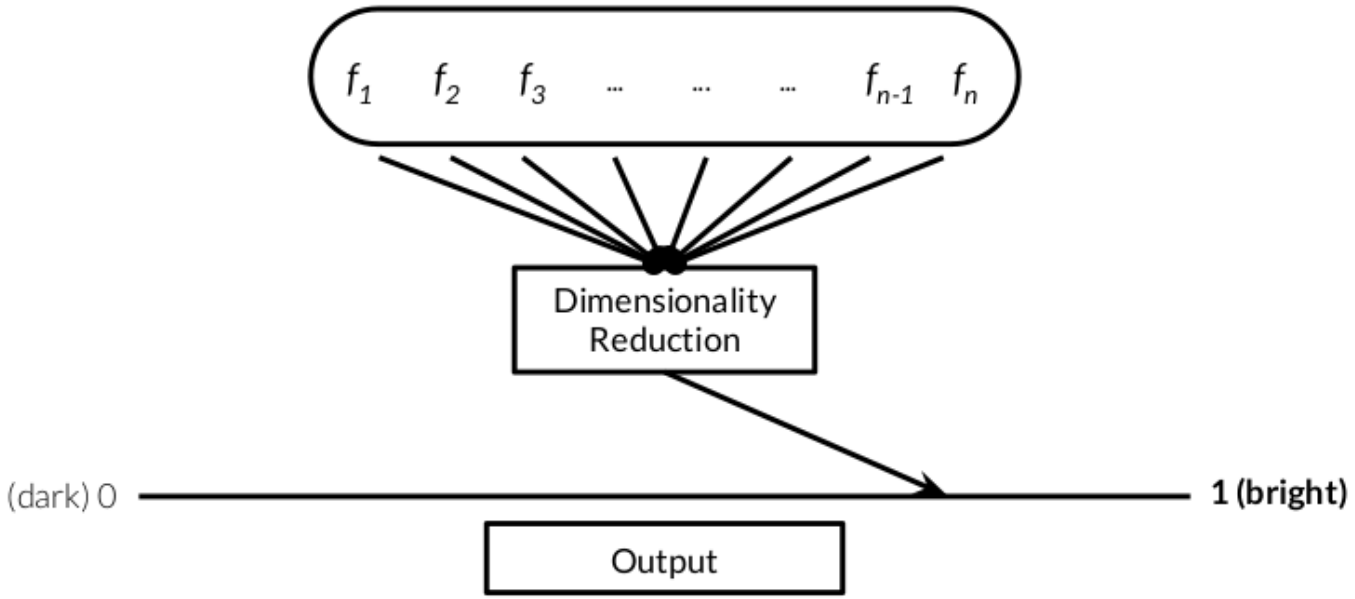
Dimensionality Reduction

Algorithmic Dimensionality

Linear dimensionality reduction

- Linearly project n-dimensional data onto a k-dimensional subspace ($k < n$, often $k \ll n$)
- There are infinitely many k-dimensional subspaces we can project the data onto
- Which one should we choose?

Projecting onto a line



Best k-dimensional subspace for projection

Classification

Maximize separation among classes
Example: Linear discriminant analysis (LDA)

Regression

Maximize correlation between projected data and response variable
Example: Partial least squares (PLS)

Unsupervised

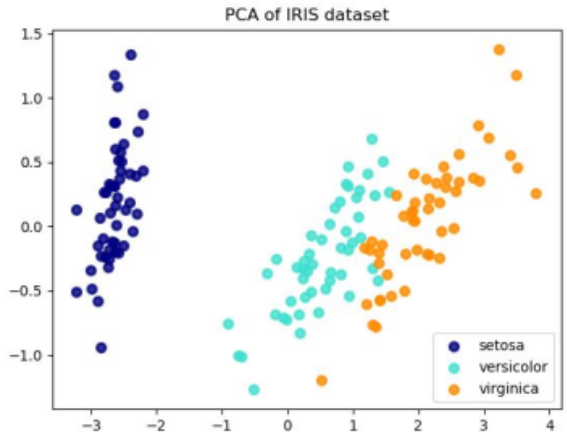
Retain as much data variance as possible
Example: Principal component analysis (PCA)

Dimensionality Reduction

Principal Component Analysis

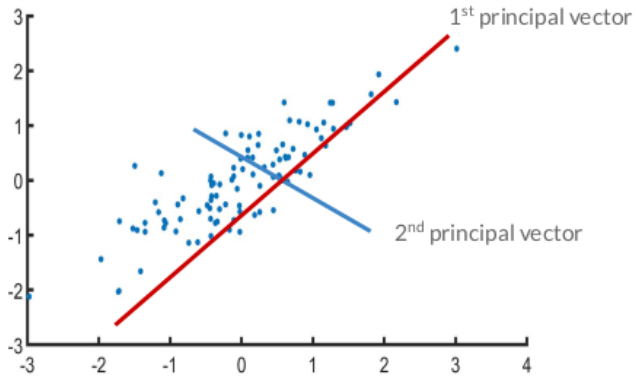
Principal component analysis (PCA)

- PCA is a minimization of the orthogonal distance
- Widely used method for unsupervised & linear dimensionality reduction
- Accounts for variance of data in as few dimensions as possible using linear projections

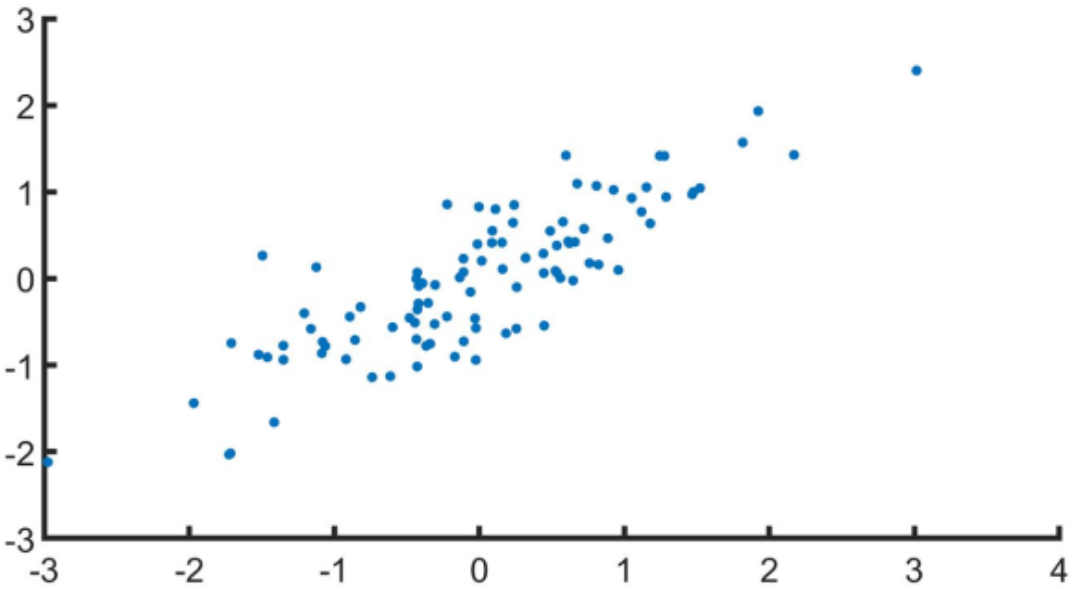


Principal components (PCs)

- PCs maximize the variance of projections
- PCs are orthogonal
- Gives the best axis to project
- Goal of PCA: Minimize total squared reconstruction error

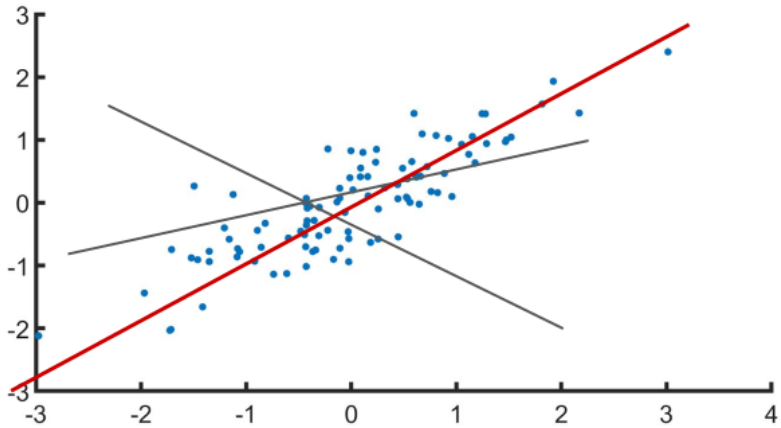


2-D data



PCA Algorithm - First Principal Component

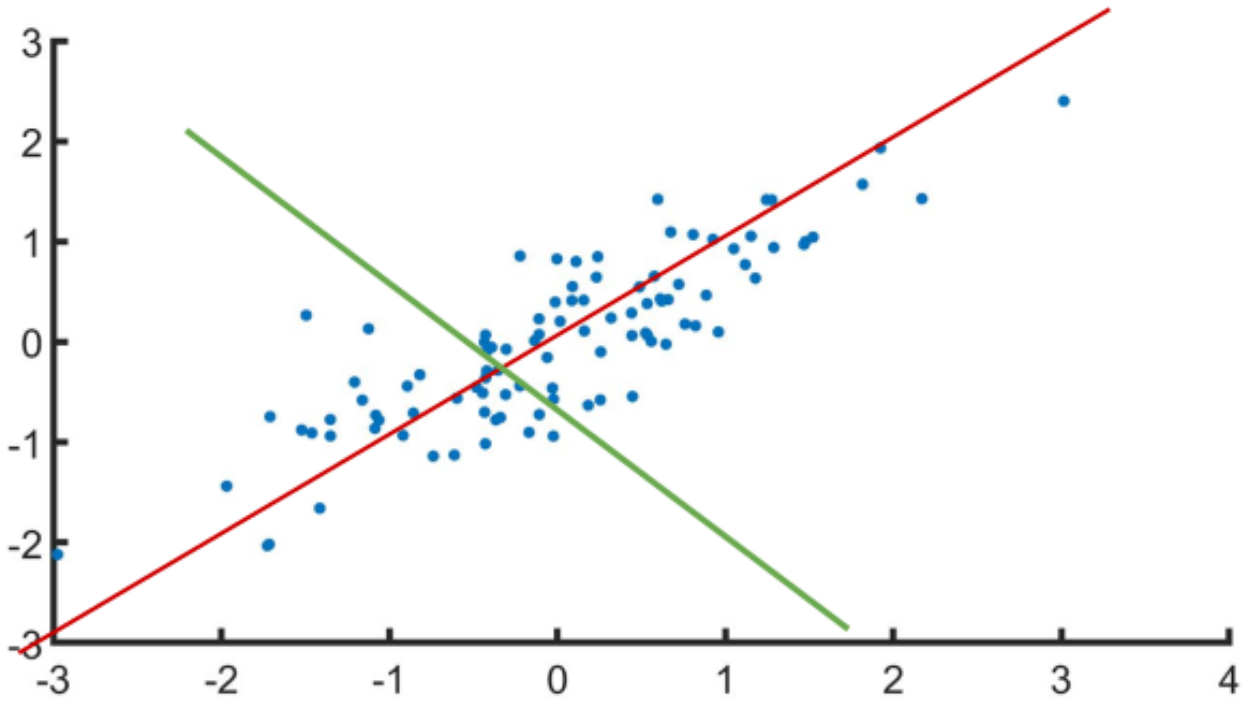
Step 1



Find a line, such that when the data is projected onto that line, it has the maximum variance

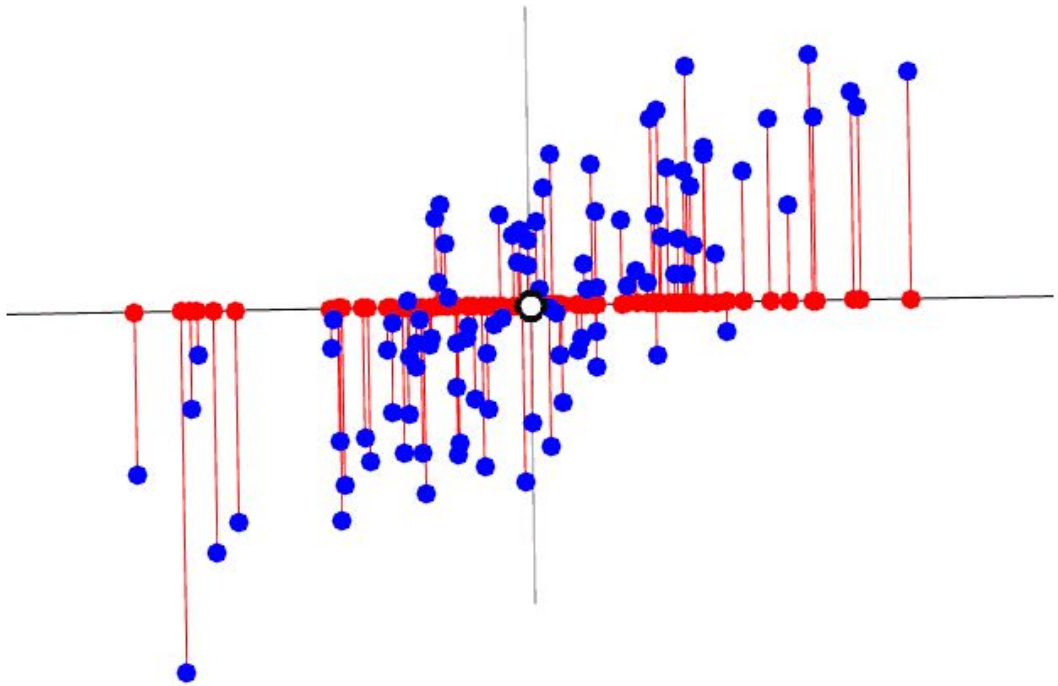
PCA Algorithm - Second Principal Component

Step 2



Find a second line, orthogonal to the first, that has maximum projected variance

PCA Algorithm



PCA: Strengths and Weaknesses

Strengths

- A versatile technique
- Fast and simple
- Offers several variations and extensions (e.g., kernel/sparse PCA)

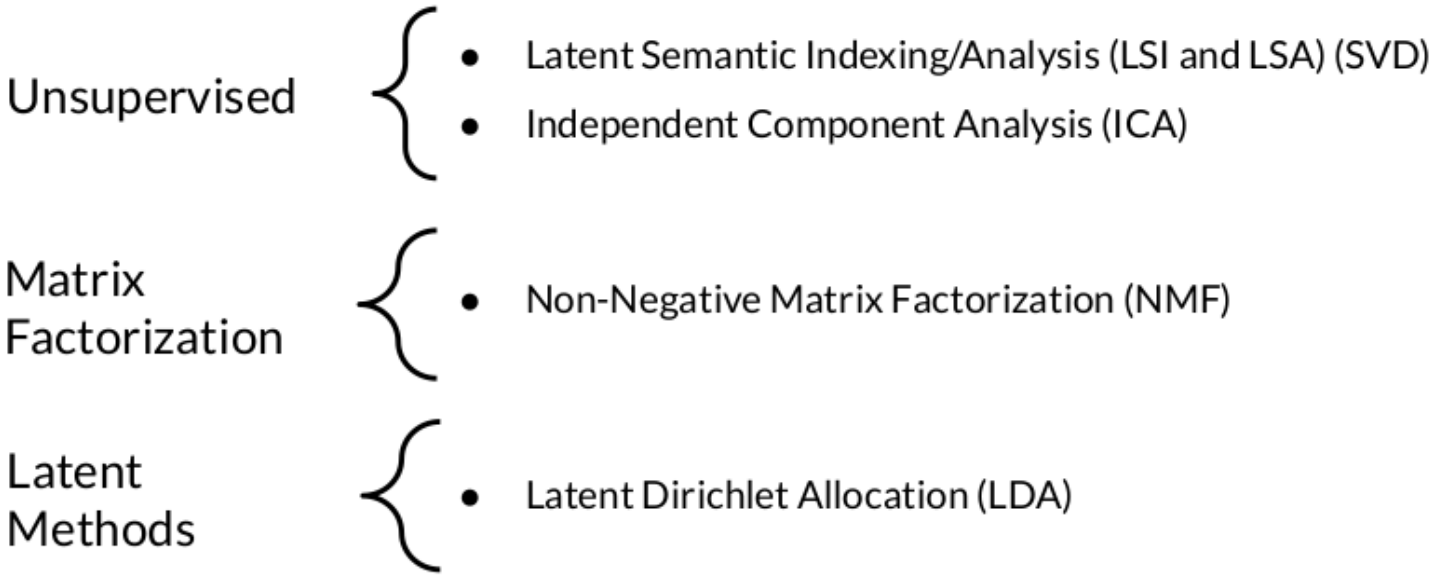
Weaknesses

- Result is not interpretable
- Requires setting threshold for cumulative explained variance

Dimensionality Reduction

Other Techniques

More dimensionality reduction algorithms



Singular value decomposition (SVD)

- SVD decomposes non-square matrices
- Useful for sparse matrices as produced by TF-IDF
- Removes redundant features from the dataset

Independent Components Analysis (ICA)

- PCA seeks directions in feature space that minimize reconstruction error
- ICA seeks directions that are most statistically independent
- ICA addresses higher order dependence

How does ICA work?

- Assume there exists independent signals:

$$S = [s_1(t),\ s_2(t),\ \ldots,\ s_N(t)]$$

- Linear combinations of signals:

$$Y(t) = AS(t)$$

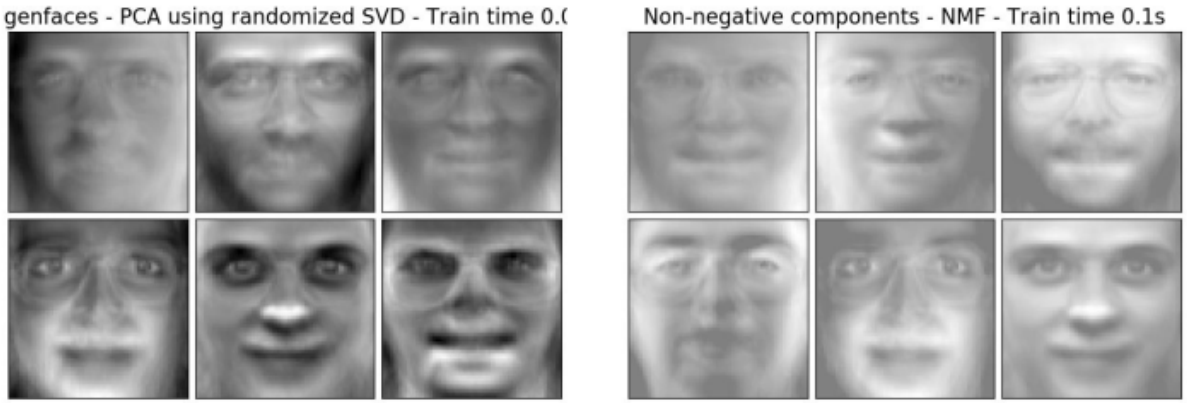
  - Both A and S are unknown
  - A: mixing matrix

- Goal of ICA: recover original signals $S(t)$ from $Y(t)$

Comparing PCA and ICA

	PCA	ICA
Removes correlations	✓	✓
Removes higher order dependence		✓
All components treated fairly?		✓
Orthogonality	✓	

Non-negative Matrix Factorization (NMF)



- NMF models are interpretable and easier to understand
- NMF requires the sample features to be non-negative

Dimensionality Reduction

Mobile, IoT, and Similar Use Cases

Trends in adoption of smart devices

Factors driving this trend

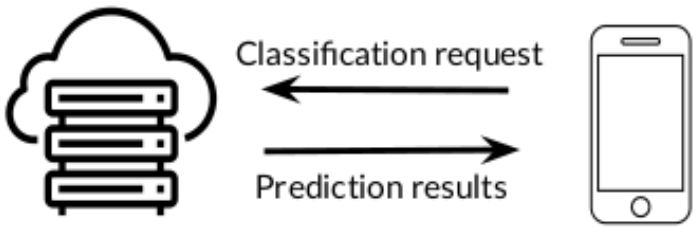
- Demands move ML capability from cloud to on-device
- Cost-effectiveness
- Compliance with privacy regulations

Online ML inference

- To generate real-time predictions you can:
  - Host the model on a server
  - Embed the model in the device
- Is it faster on a server, or on-device?
- Mobile processing limitations?

Mobile inference

Inference on the cloud/server



Pros

- Lots of compute capacity
- Scalable hardware
- Model complexity handled by the server
- Easy to add new features and update the model
- Low latency and batch prediction

Cons

- Timely inference is needed

Mobile inference

On-device Inference

Pro

- Improved speed
- Performance
- Network connectivity
- No to-and-fro communication needed

Cons

- Less capacity
- Tight resource constraints

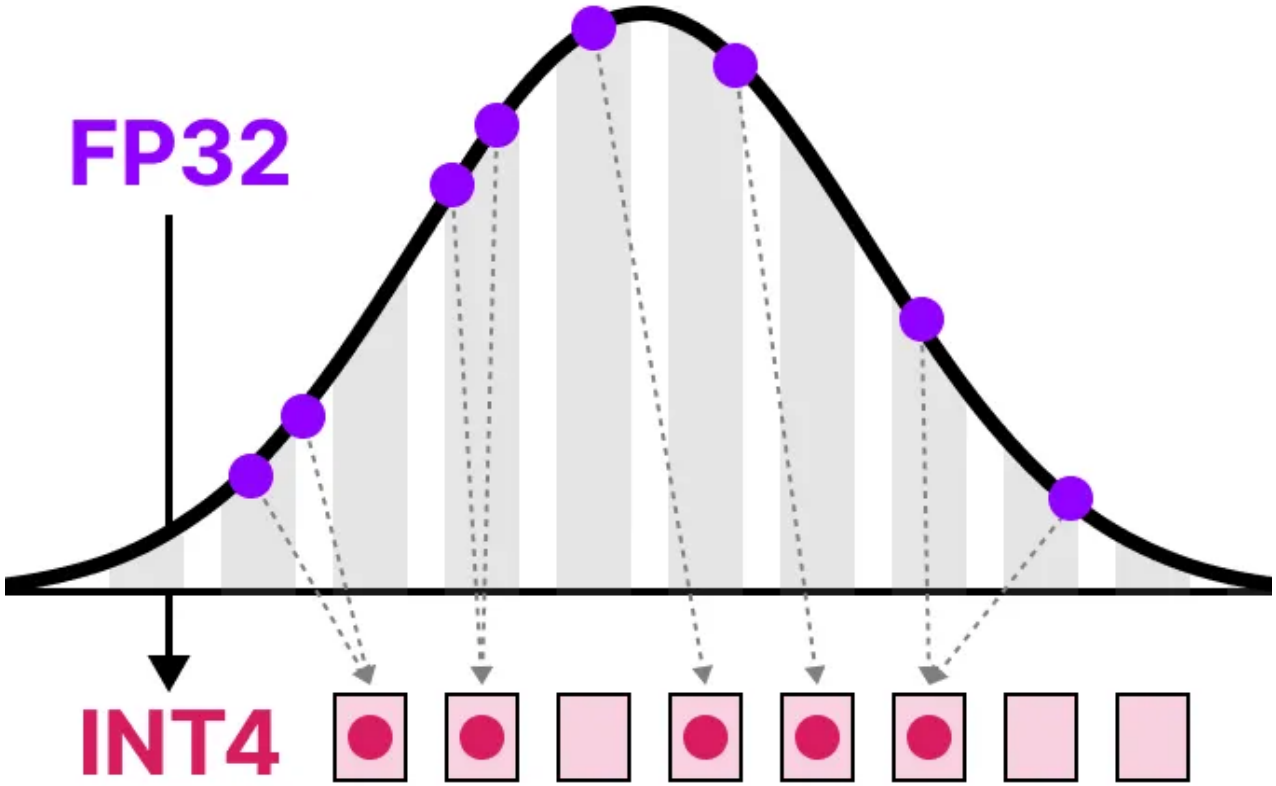
Model deployment

Options	On-device inference	On-device personalization	On-device training	Cloud-based web service	Pretrained models	Custom models
ML Kit	✓	✓		✓	✓	✓
Core ML	✓	✓	✓		✓	✓
TensorFlow Lite *	✓	✓	✓		✓	✓

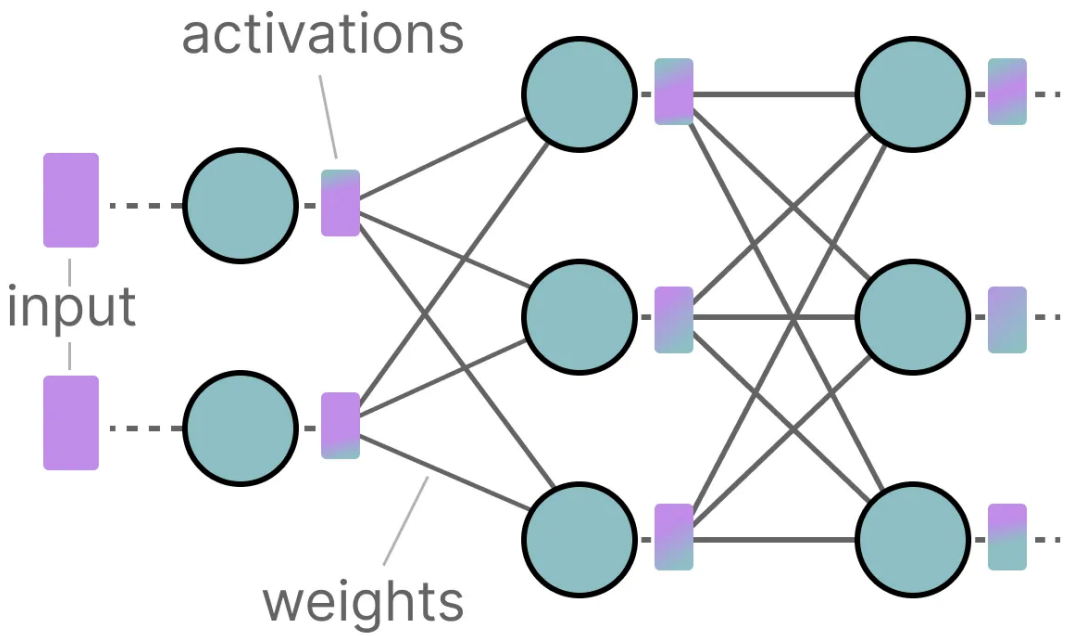
Quantization and Pruning

Benefits and Process of Quantization

Quantization

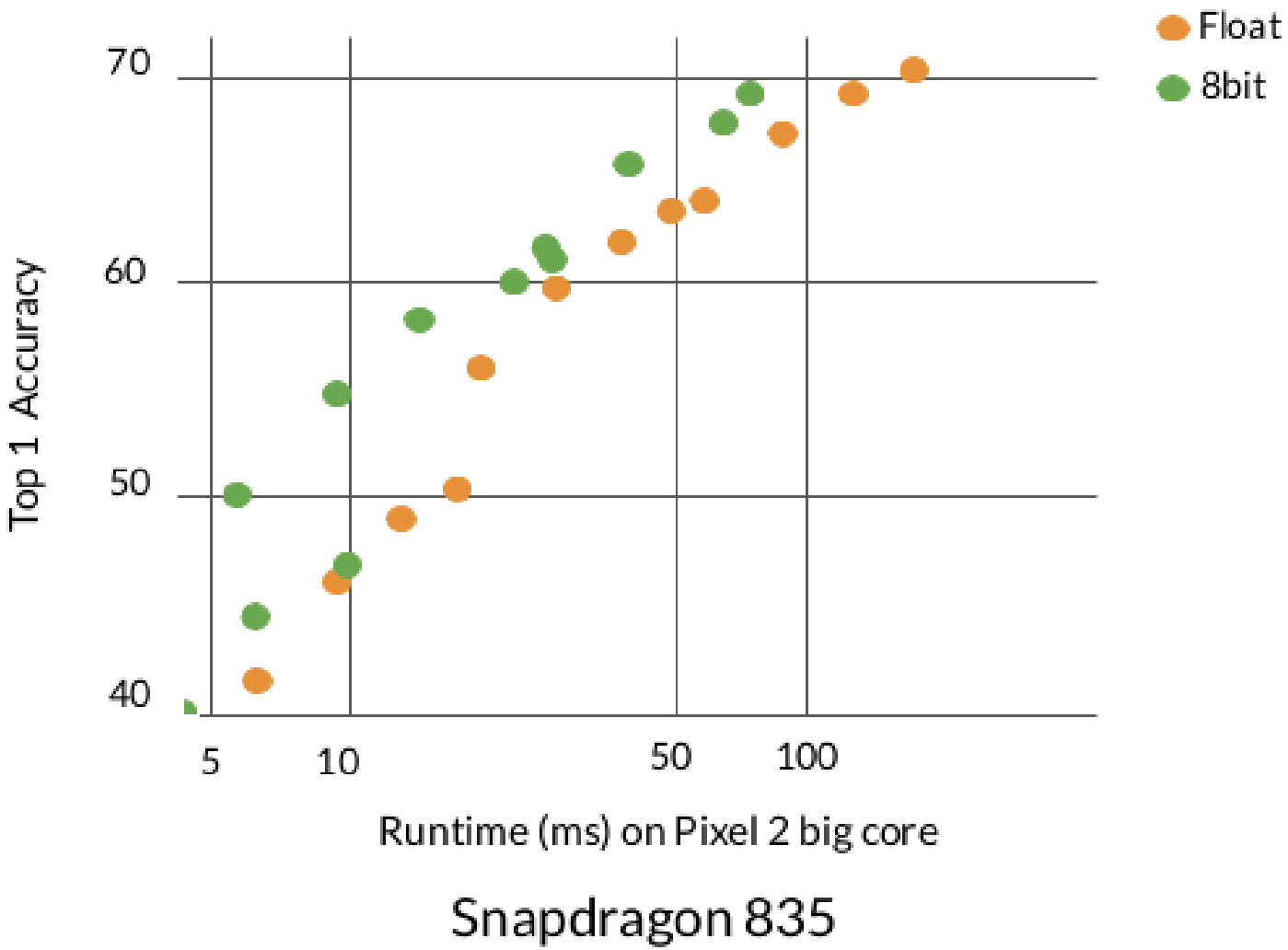


Why quantize neural networks?



- Neural networks have many parameters and take up space
- Shrinking model file size
- Reduce computational resources
- Make models run faster and use less power with low-precision

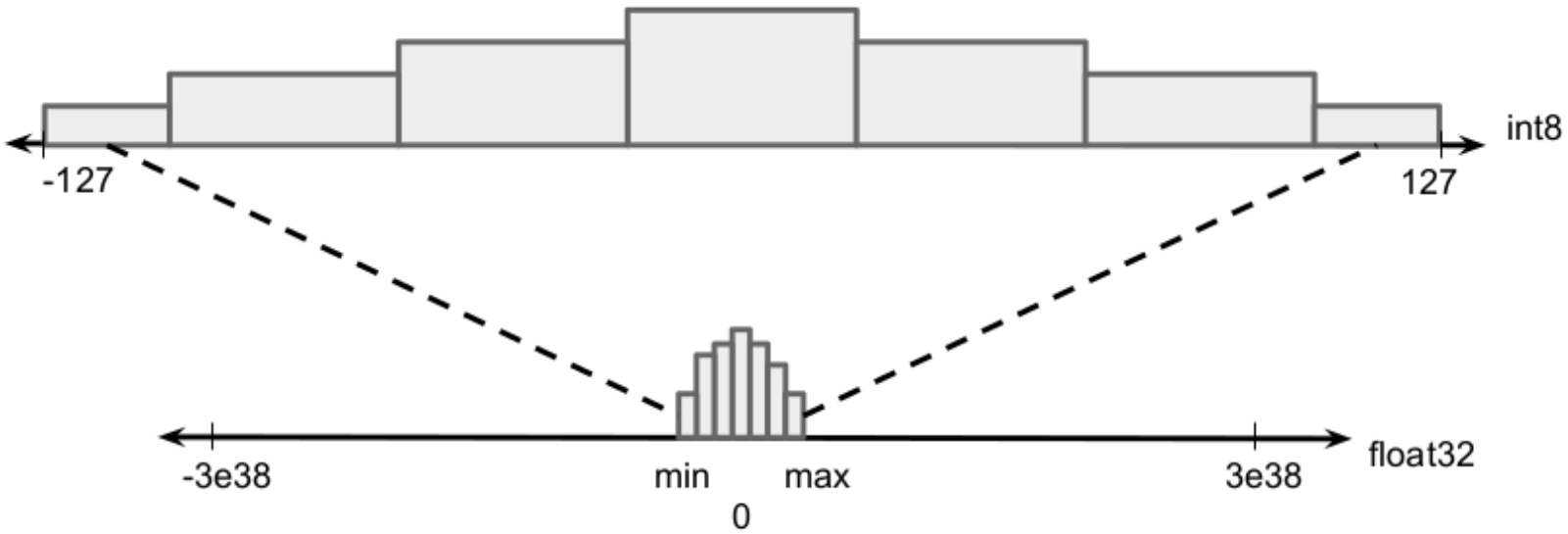
MobileNets: Latency vs Accuracy trade-off



Benefits of quantization

- Faster compute
- Low memory bandwidth
- Low power
- Integer operations supported across CPU/DSP/NPUs

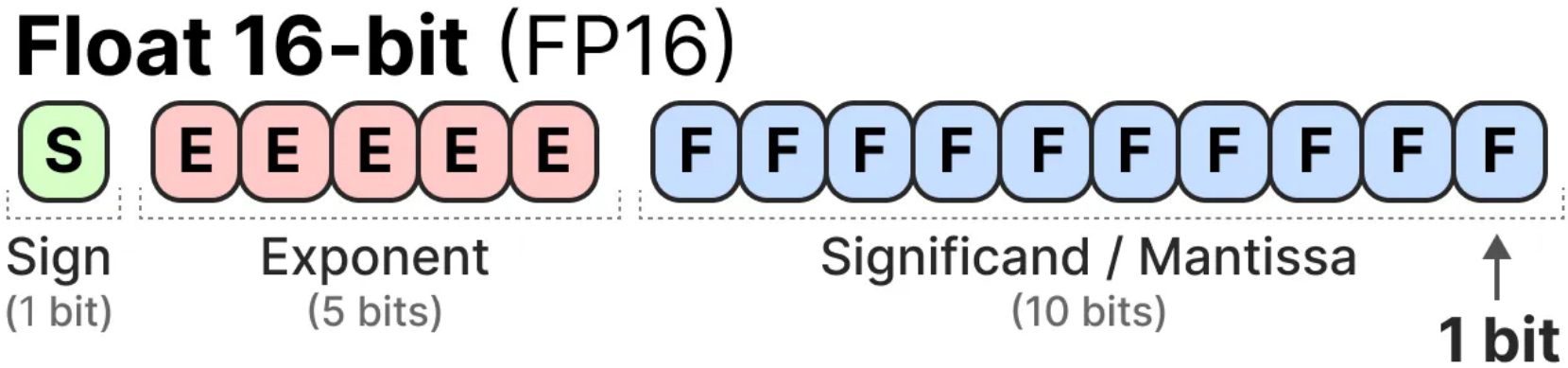
The quantization process



Representation

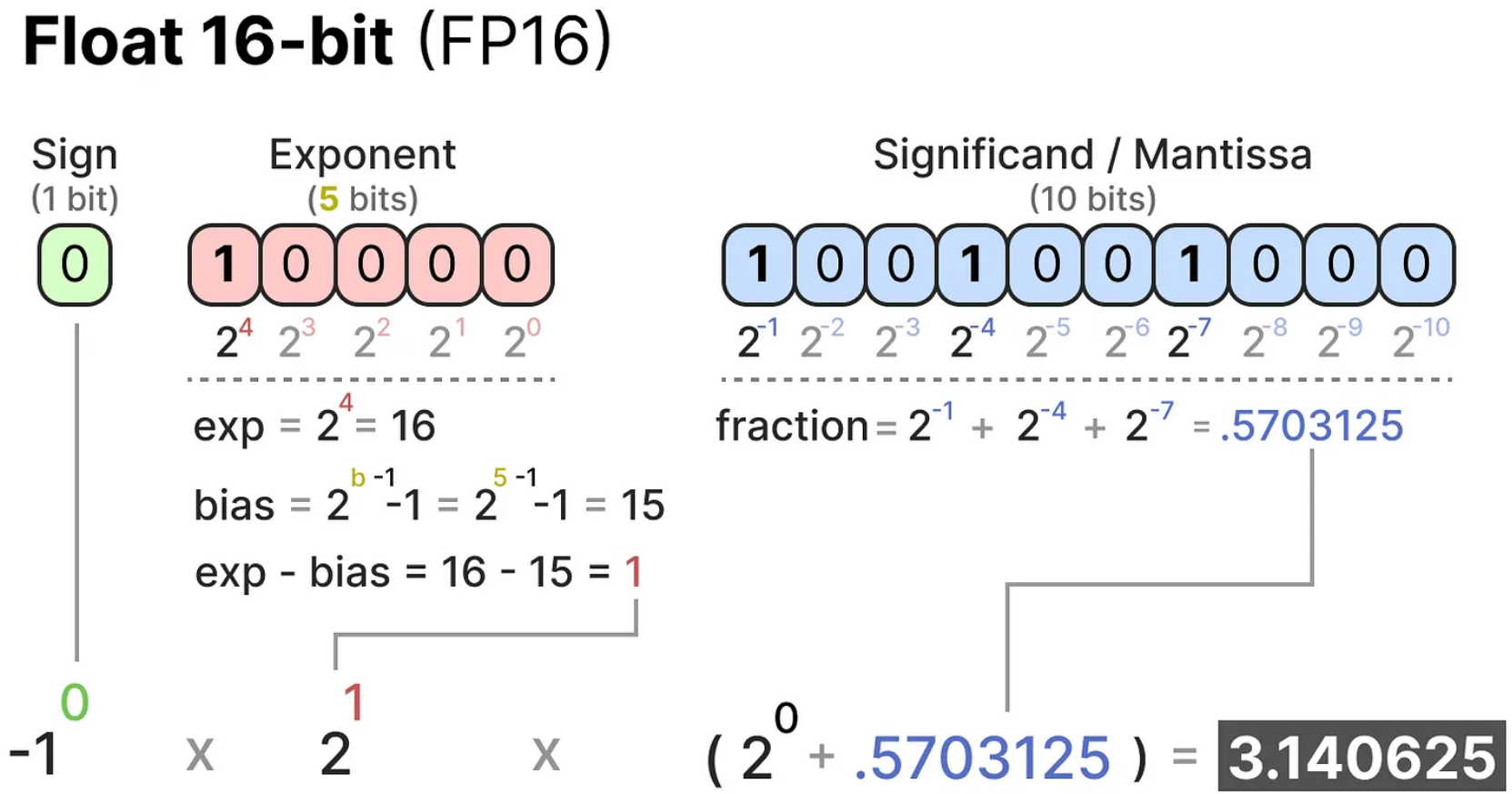
A given value is often represented as a floating point number (or floats in computer science): a positive or negative number with a decimal point.

These values are represented by "bits", or binary digits. The IEEE-754 standard describes how bits can represent one of three functions to represent the value: the sign, exponent, or fraction (or mantissa).



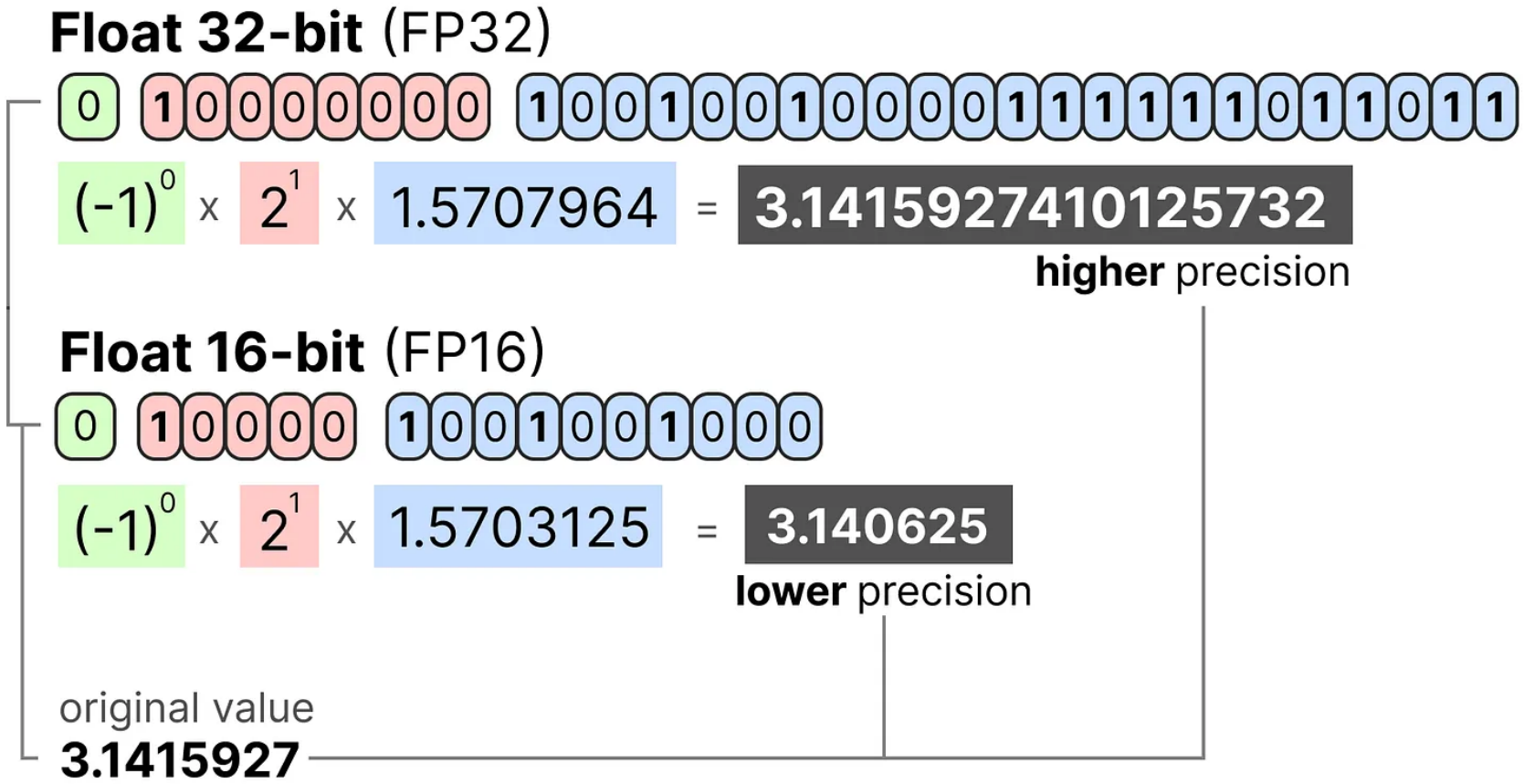
Representation

Together, these three aspects can be used to calculate a value given a certain set of bit values:



Representation

The more bits we use to represent a value, the more precise it generally is:



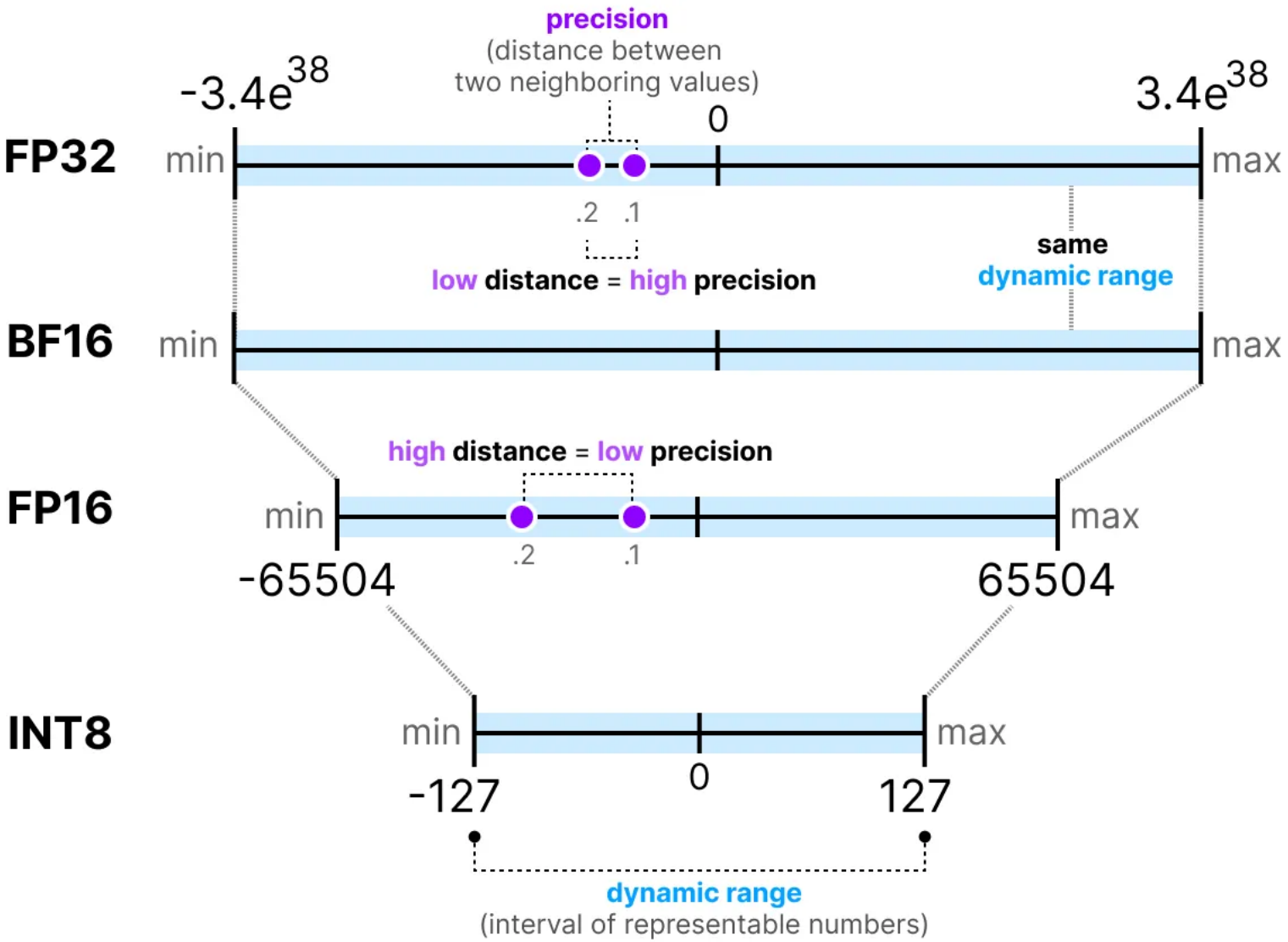
Memory Constraints

The more bits we have available, the larger the range of values that can be represented.

Original	75505.0	1.8e-42
64-bits	75505.0	1.8e-42
32-bits	75505.0	1.80066e-42
16-bits	inf	0.0

The interval of representable numbers a given representation can take is called the dynamic range whereas the distance between two neighboring values is called precision.

Memory Constraints



Memory Constraints

A nifty feature of these bits is that we can calculate how much memory your device needs to store a given value. Since there are 8 bits in a byte of memory, we can create a basic formula for most forms of floating point representation.

$$\text{memory} = \frac{\text{nr\_bits}}{8} \times \text{nr\_params}$$

Now let's assume that we have a model with 70 billion parameters. Most models are natively represented with float 32-bit (often called full-precision), which would require 280GB of memory just to load the model.

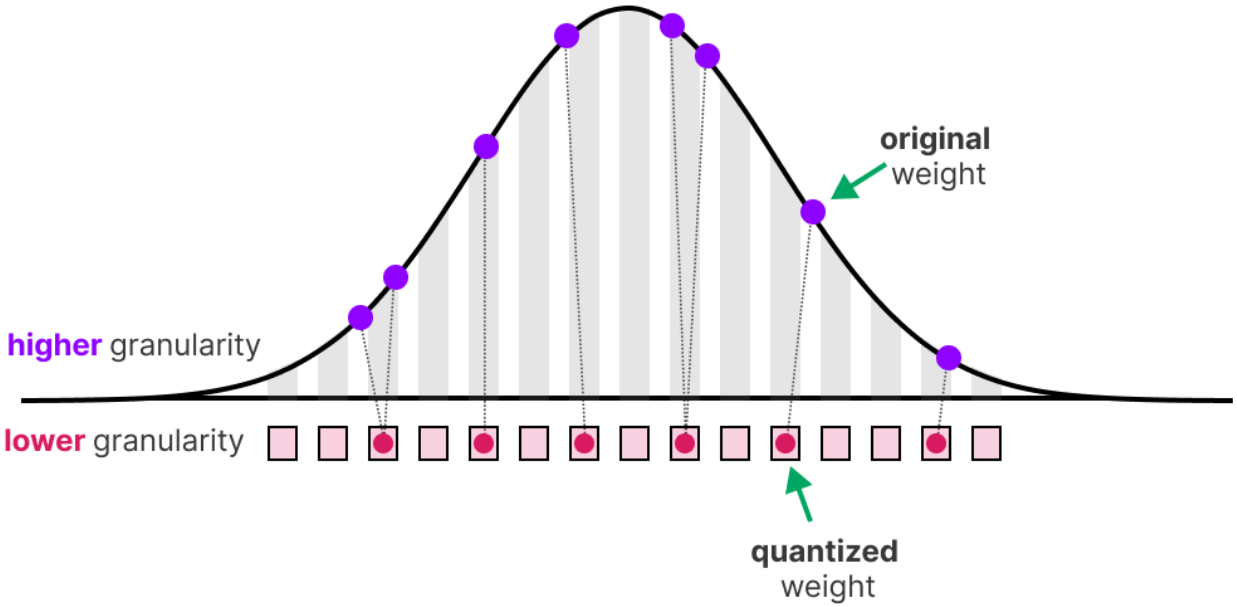
$$\textbf{64-bits} = \frac{64}{8} \times 70\text{B} \approx \textbf{560 GB}$$

$$\textbf{32-bits} = \frac{32}{8} \times 70\text{B} \approx \textbf{280 GB}$$

$$\textbf{16-bits} = \frac{16}{8} \times 70\text{B} \approx \textbf{140 GB}$$

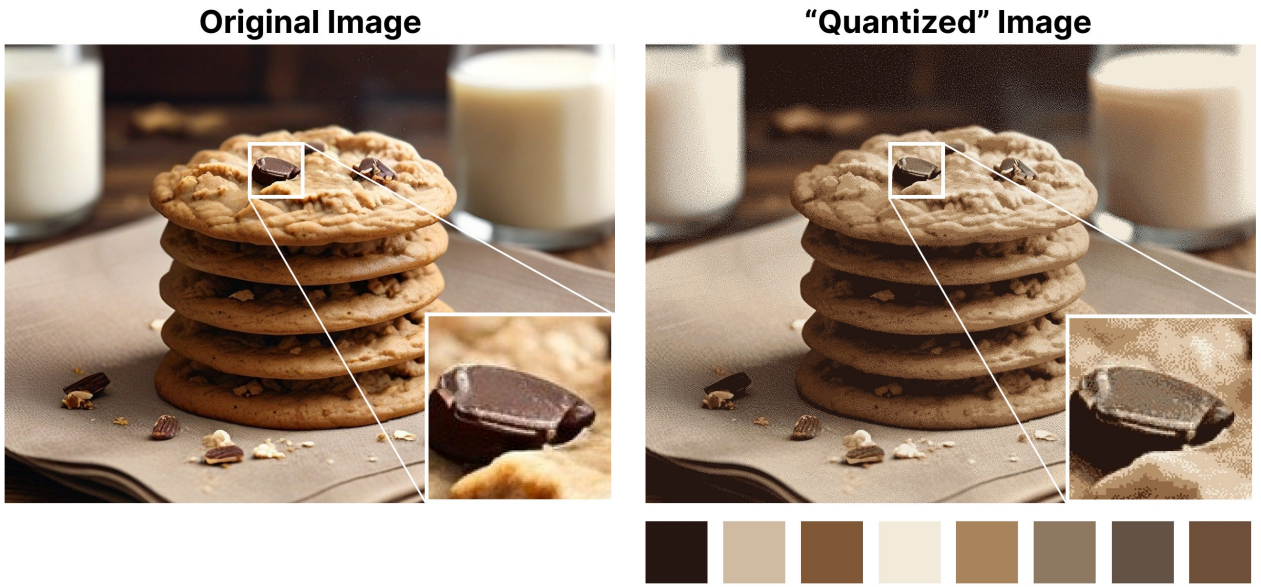
Quantization

Quantization aims to reduce the precision of a model's parameter from higher bit-widths (like 32-bit floating point) to lower bit-widths (like 8-bit integers).



Quantization

To illustrate this effect, we can take any image and use only 8 colors to represent it:

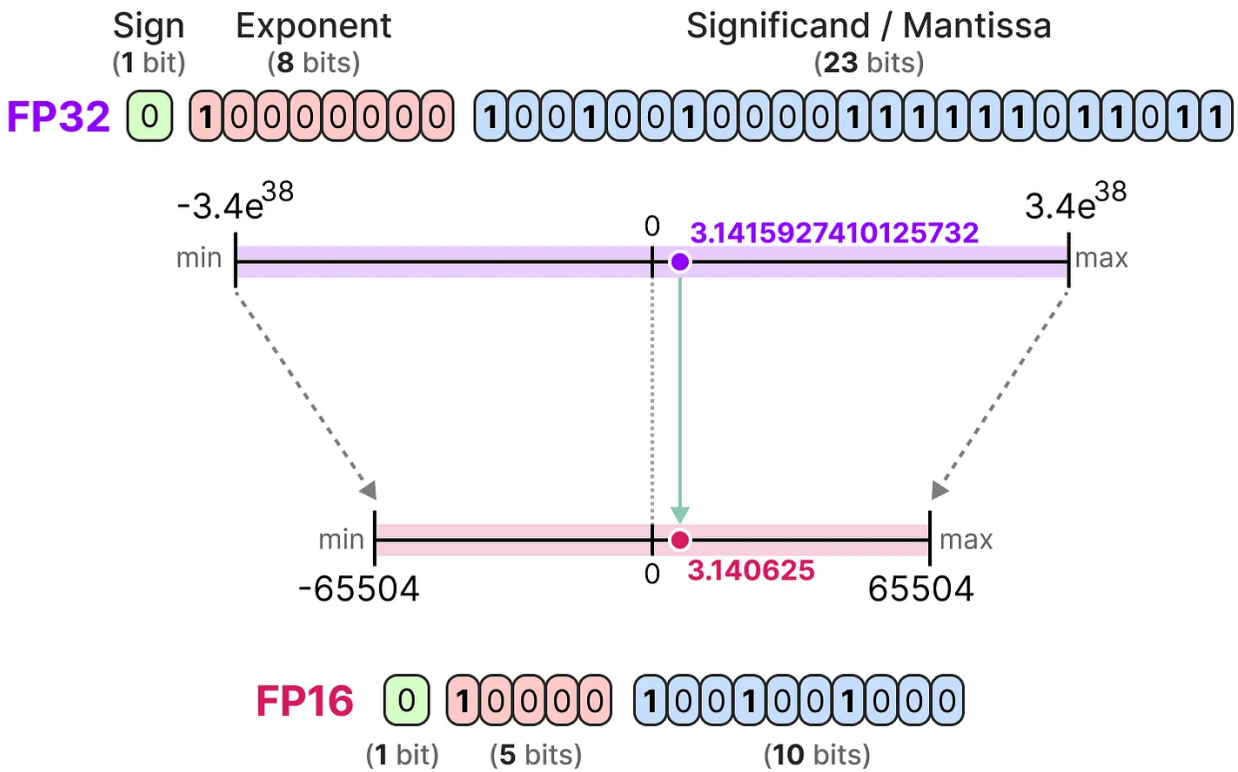


Quantization and Pruning

Common Data Types

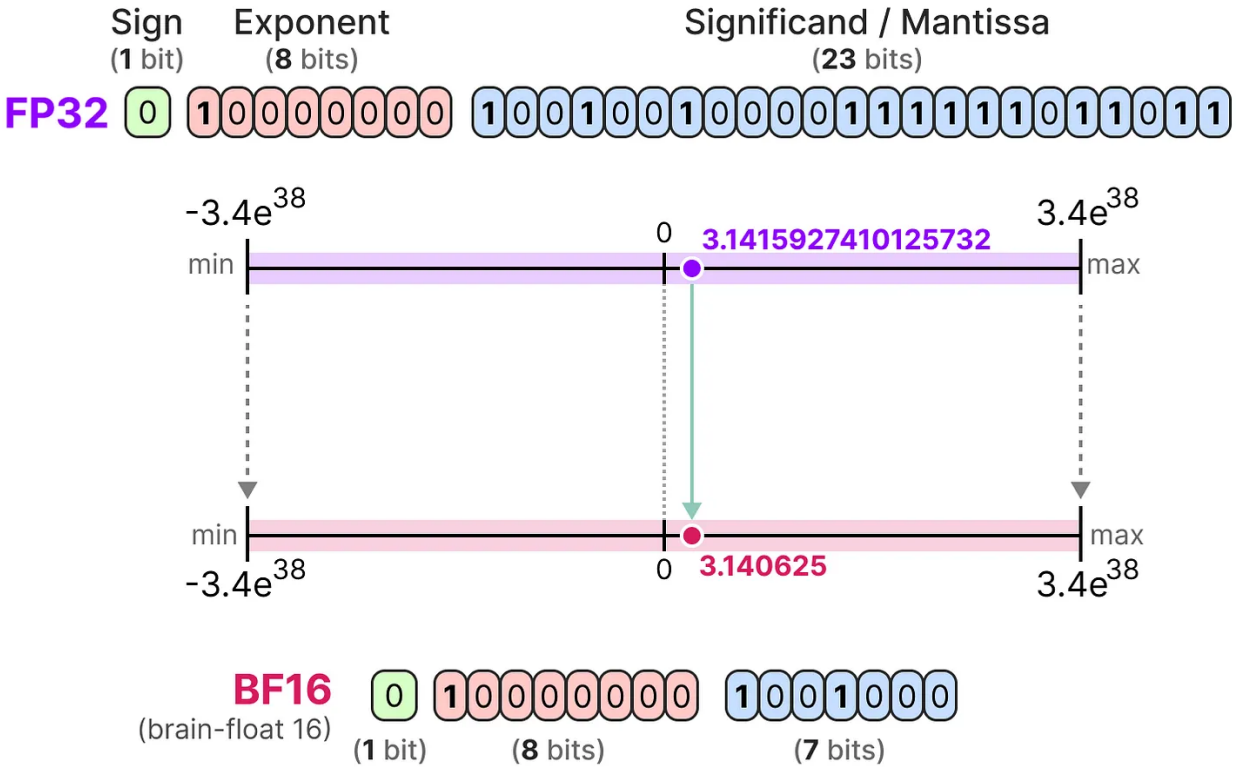
FP16

Let's look at an example of going from 32-bit to 16-bit (called half precision or FP16) floating point:



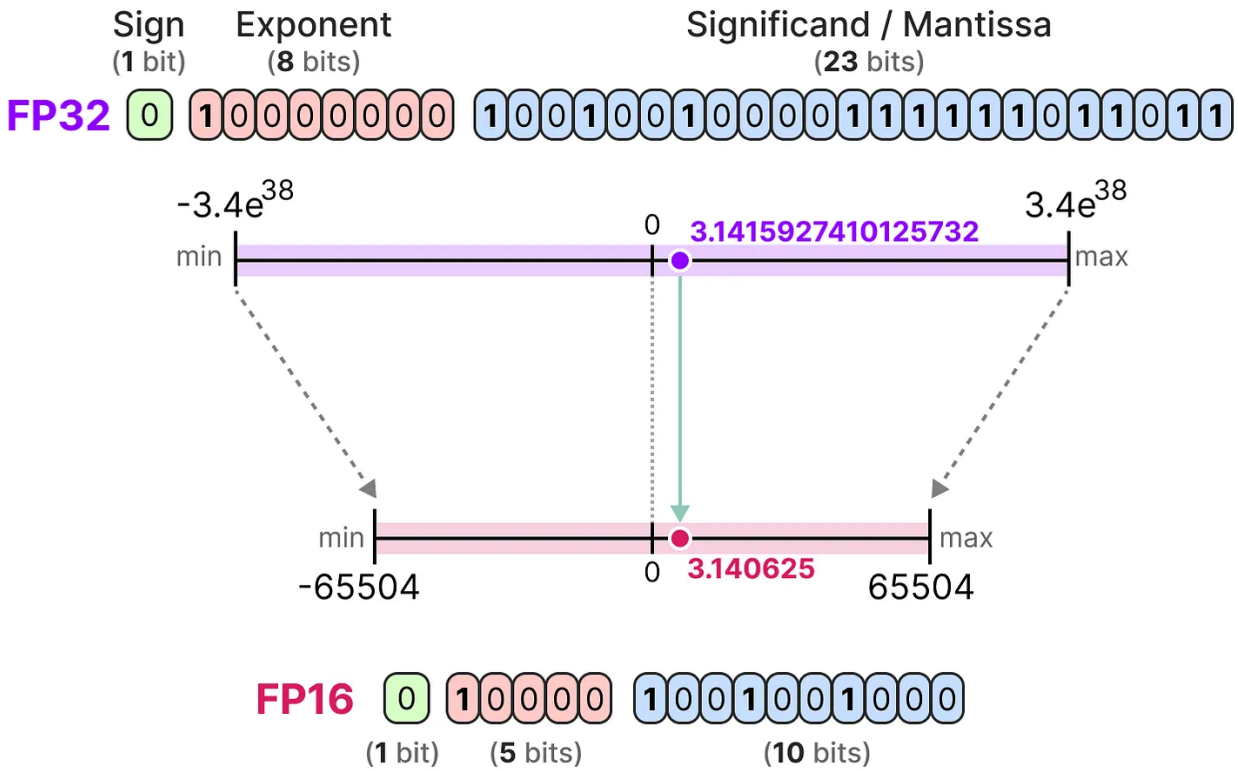
BF16

To get a similar range of values as the original FP32, bfloat 16 was introduced as a type of "truncated FP32":



INT8

When we reduce the number of bits even further, we approach the realm of integer-based representations rather than floating-point representations. To illustrate, going FP32 to INT8, which has only 8 bits, results in a fourth of the original number of bits:



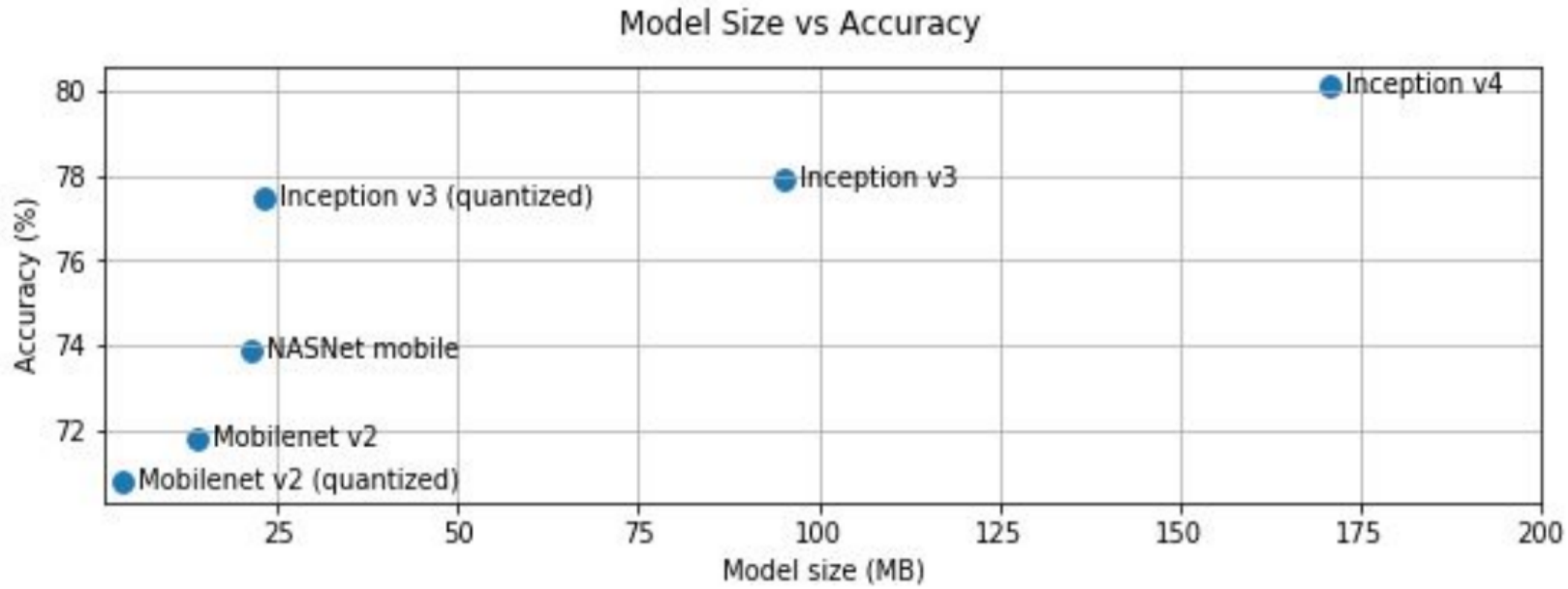
What parts of the model are affected?

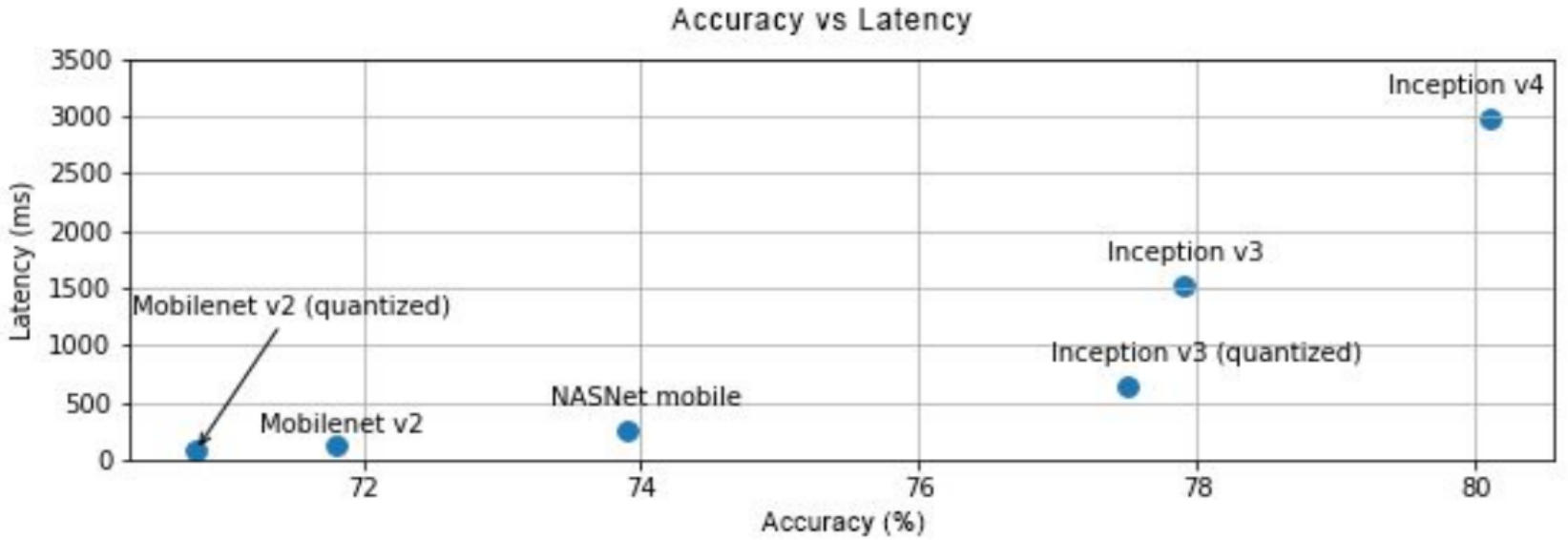
- Static values (parameters)
- Dynamic values (activations)
- Computation (transformations)

Trade-offs

- Optimizations impact model accuracy
  - Difficult to predict ahead of time
- In rare cases, models may actually gain some accuracy
- Undefined effects on ML interpretability

Choose the best model for the task





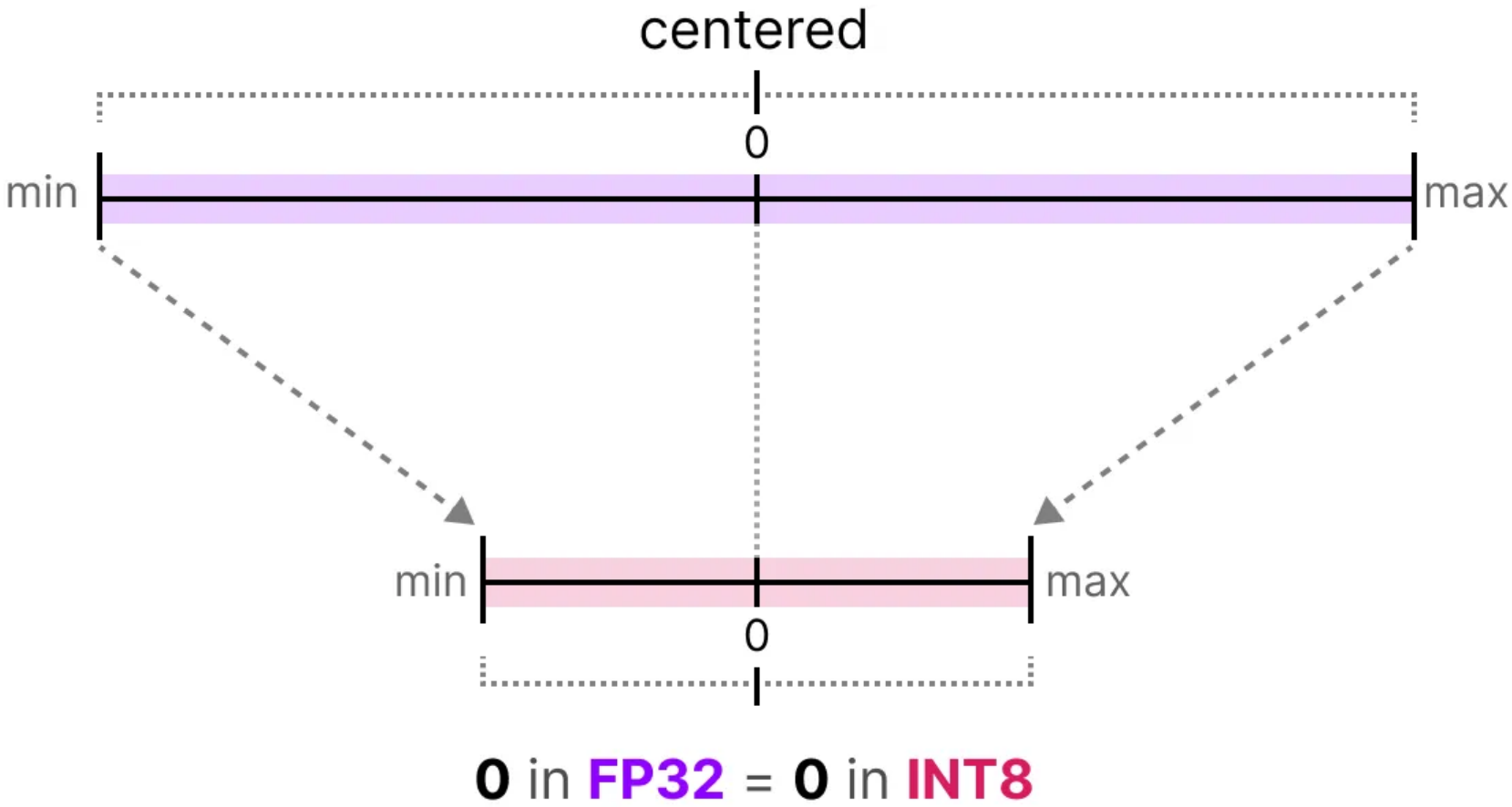
Quantization

Symmetric Quantization

Symmetric Quantization

In symmetric quantization, the range of the original floating-point values is mapped to a symmetric range around zero in the quantized space. In the previous examples, notice how the ranges before and after quantization remain centered around zero.

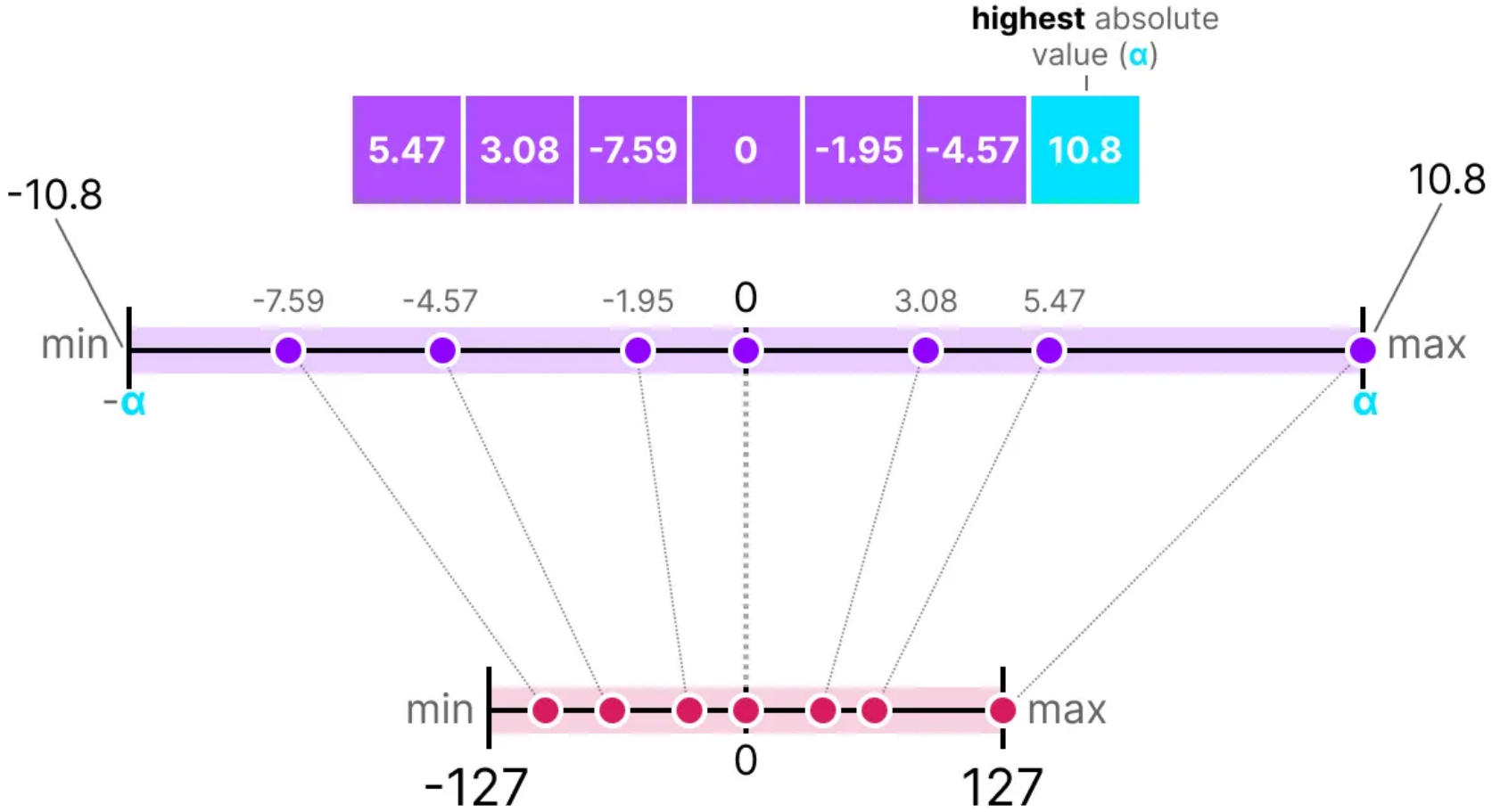
This means that the quantized value for zero in the floating-point space is exactly zero in the quantized space.



absmax Quantization

A nice example of a form of symmetric quantization is called absolute maximum (absmax) quantization.

Given a list of values, we take the highest absolute value (α) as the range to perform the linear mapping.



absmax Quantization

We first calculate a scale factor (s) using:

- b is the number of bytes that we want to quantize to (8),
- α is the highest absolute value,

Then, we use the s to quantize the input x:

$$s = \frac{2^{b-1}-1}{\alpha} \quad \text{(scale factor)}$$

$$x_{quantized} = \text{round}\left(s \cdot x\right) \quad \text{(quantization)}$$

Filling in the values would then give us the following:

$$s = \frac{127}{10.8} = 11.76 \quad \text{(scale factor)}$$

$$x_{quantized} = \text{round}\left(11.76 \cdot \blacksquare\right) \quad \text{(quantization)}$$

To retrieve the original FP32 values, we can use the previously calculated scaling factor (s) to dequantize the quantized values.

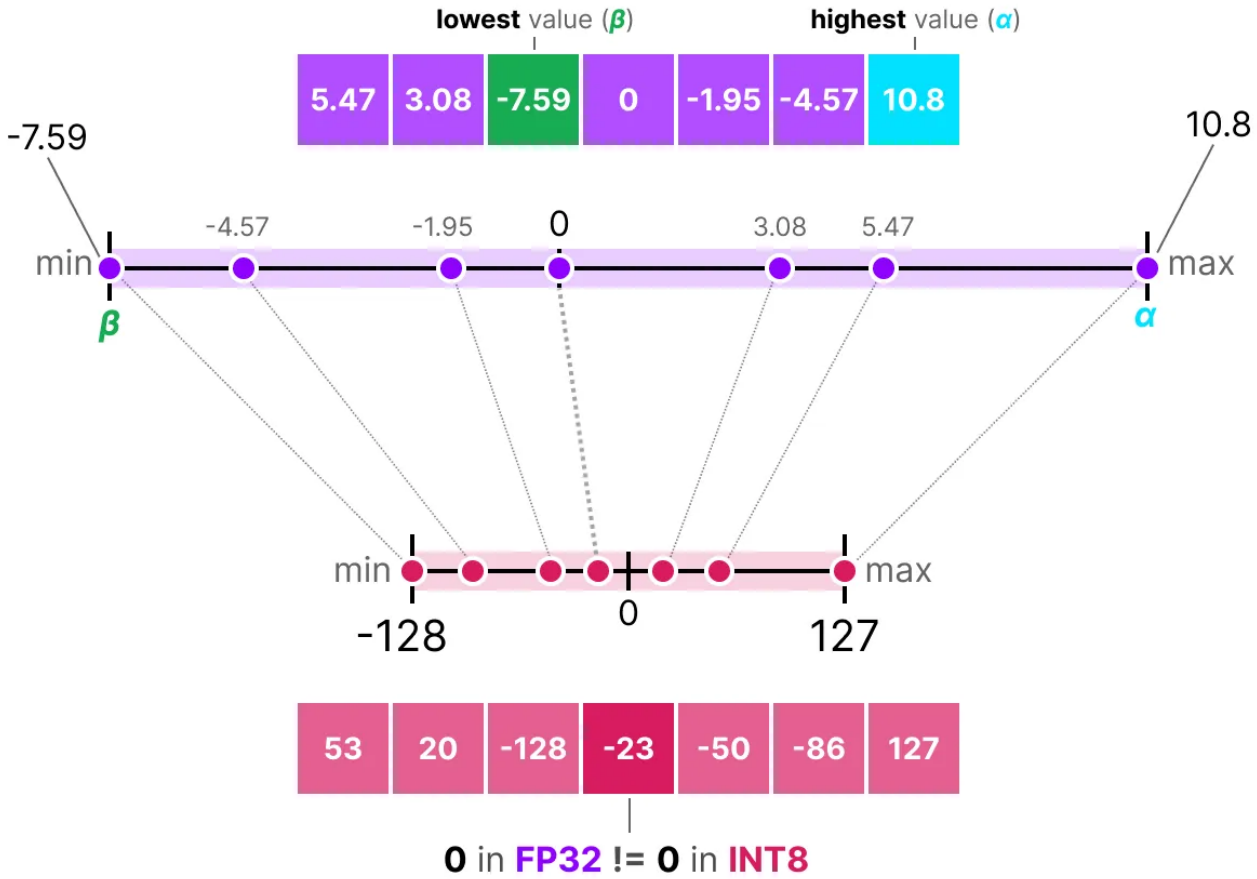
$$x_{dequantized} = \frac{\blacksquare}{s} \quad \text{(dequantize)}$$

Quantization

Asymmetric Quantization

Asymmetric Quantization

Asymmetric quantization, in contrast, is not symmetric around zero. Instead, it maps the minimum (β) and maximum (α) values from the float range to the minimum and maximum values of the quantized range.
The method we are going to explore is called zero-point quantization.



Assymetric Quantization

Notice how the 0 has shifted positions? That's why it's called asymmetric quantization. The min/max values have different distances to 0 in the range [-7.59, 10.8].

Due to its shifted position, we have to calculate the zero-point for the INT8 range to perform the linear mapping. As before, we also have to calculate a scale factor (s) but use the difference of INT8's range instead [-128, 127]

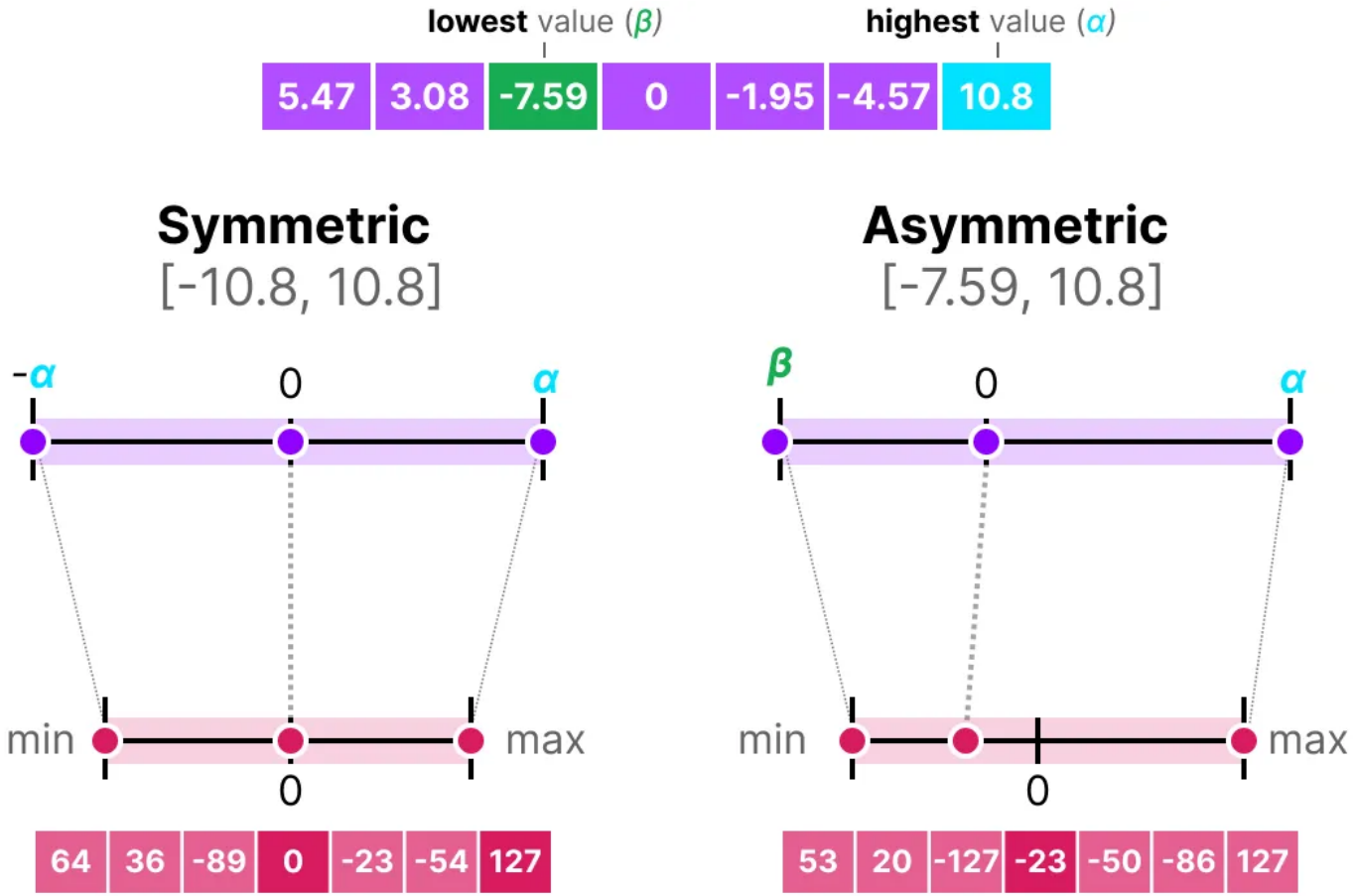
$$s = \frac{128 - -127}{\alpha - \beta} \quad \text{(scale factor)}$$

$$z = \text{round}\left(-s \cdot \beta\right) - 2^{b-1} \quad \text{(zeropoint)}$$

$$x_{quantized} = \text{round}\left(s \cdot x + z\right) \quad \text{(quantization)}$$

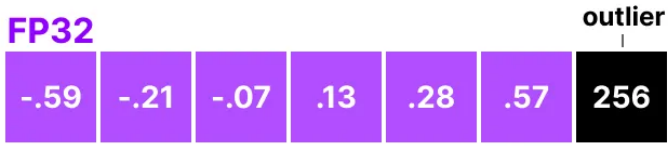
$$x_{dequantized} = \frac{\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare - z}{s} \quad \text{(dequantize)}$$

(A)Symmetric Quantization



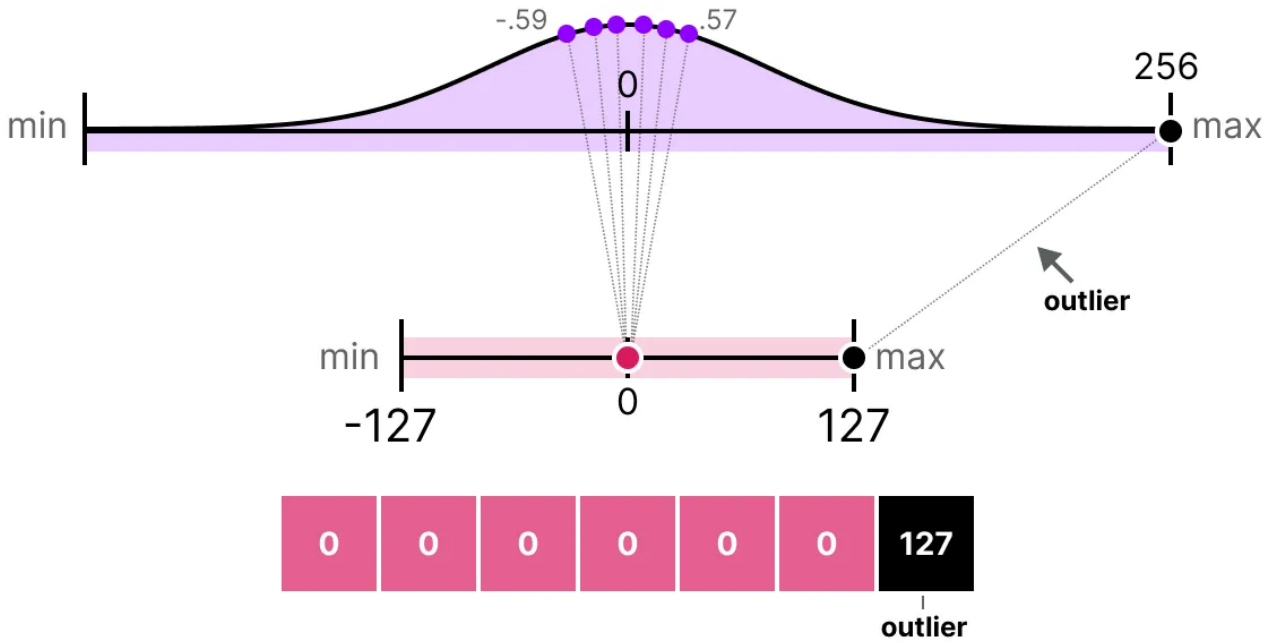
Range Mapping and Clipping

Imagine that you have a vector with the following values:

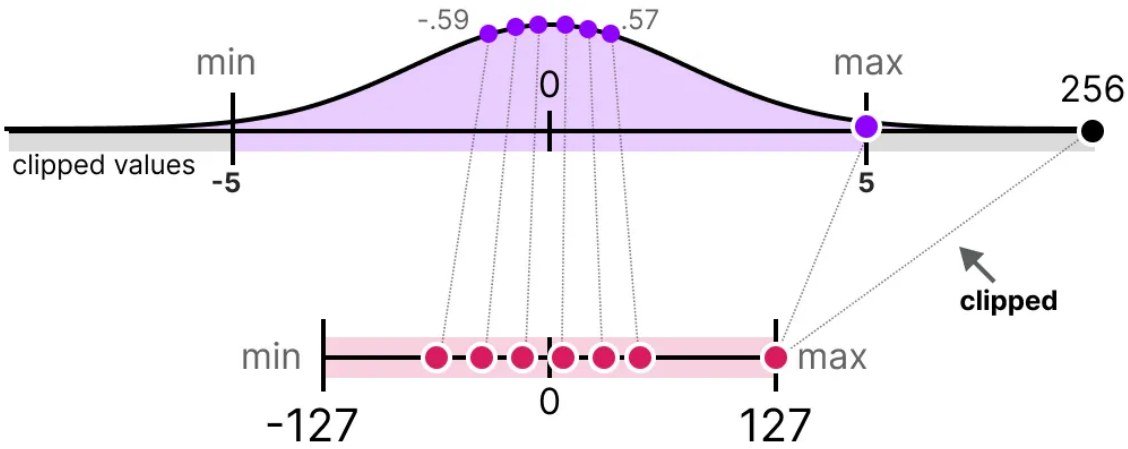


Note how one value is much larger than all others and could be considered an outlier.

If we were to map the full range of this vector, all small values would get mapped to the same lower-bit representation and lose their differentiating factor:



Instead, we can choose to clip certain values. Clipping involves setting a different dynamic range of the original values such that all outliers get the same value.
In the example below, if we were to manually set the dynamic range to [-5, 5] all values outside that will either be mapped to -127 or to 127 regardless of their value:



The major advantage is that the quantization error of the non-outliers is reduced significantly. However, the quantization error of outliers increases.

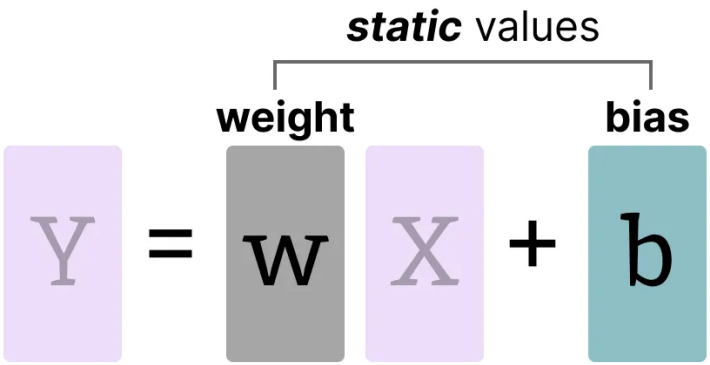
Calibration

In quantization, choosing the numeric range for values is called **calibration**.

- A naive approach might pick a fixed range (e.g., $[-5, 5]$), but this can introduce unnecessary quantization error.
- The goal of calibration is to select a range that captures as many real values as possible while minimizing error.
- Different types of parameters (e.g., weights vs. activations) may require different calibration strategies.

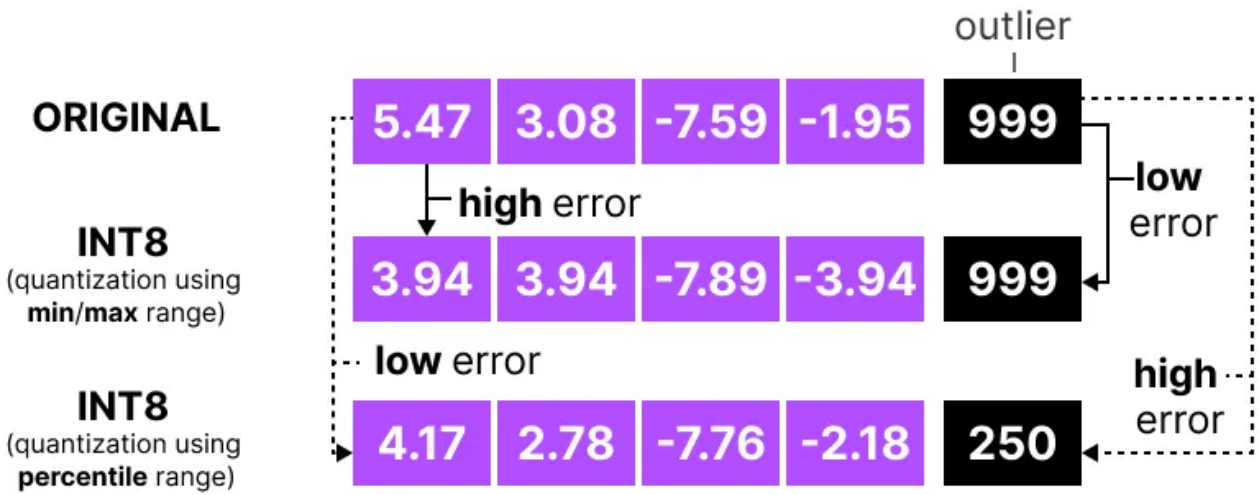
Weight and Biases

We can view the weights and biases of an LLM as static values since they are known before running the model. For instance, the $\sim 20GB$ file of Llama 3 consists mostly of its weight and biases.



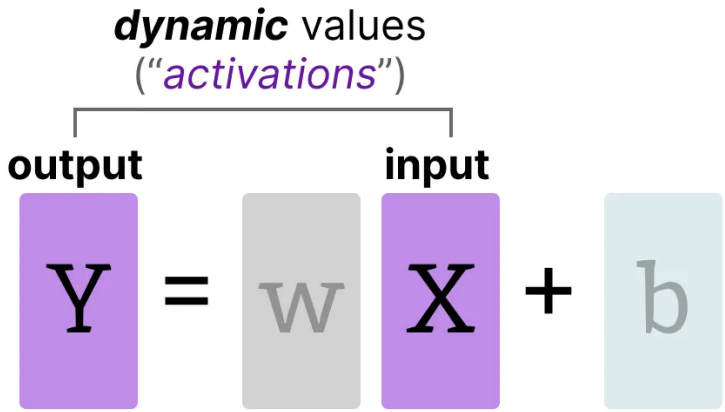
Since there are significantly fewer biases (millions) than weights (billions), the biases are often kept in higher precision (such as INT16), and the main effort of quantization is put towards the weights.
For weights, which are static and known, calibration techniques for choosing the range include:

- Manually chosing a percentile of the input range
- Optimize the mean squared error (MSE) between the original and quantized weights.
- Minimizing entropy (KL-divergence) between the original and quantized values



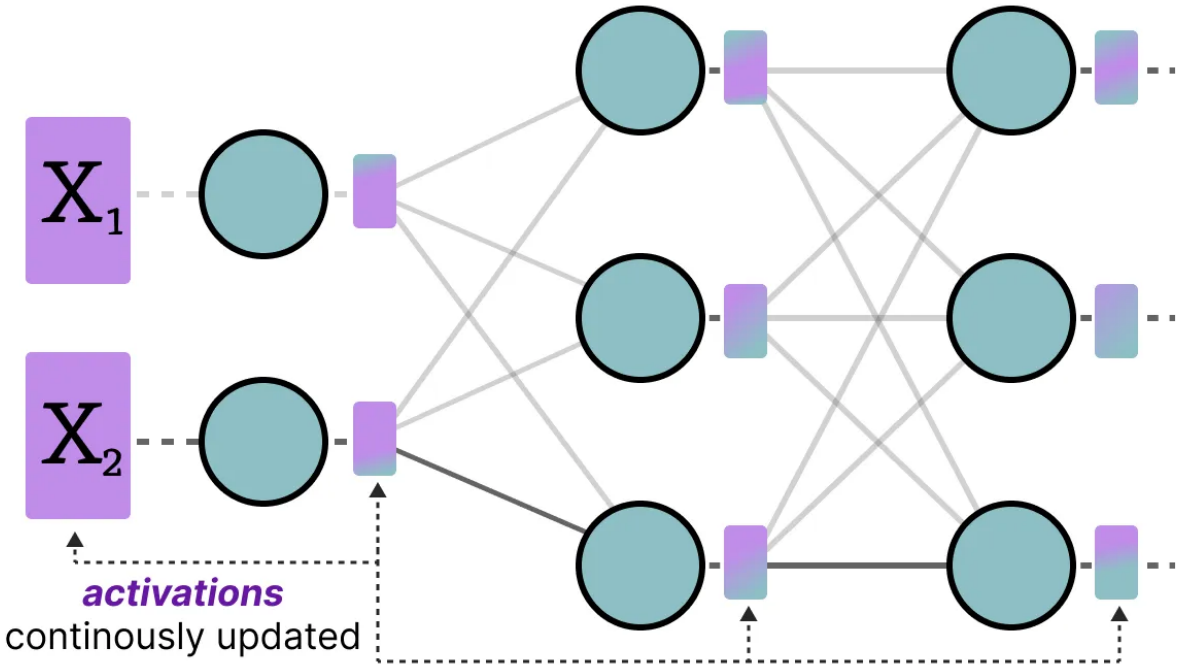
Activations

The input that is continuously updated throughout the LLM is typically referred to as "activations".



Unlike weights, activations vary with each input data fed into the model during inference, making it challenging to quantize them accurately.

Since these values are updated after each hidden layer, we only know what they will be during inference as the input data passes through the model.



Quantization

Post Training Quantization

Post-training quantization

It involves quantizing a model's parameters (both weights and activations) after training the model.

Quantization of the weights is performed using either symmetric or asymmetric quantization.

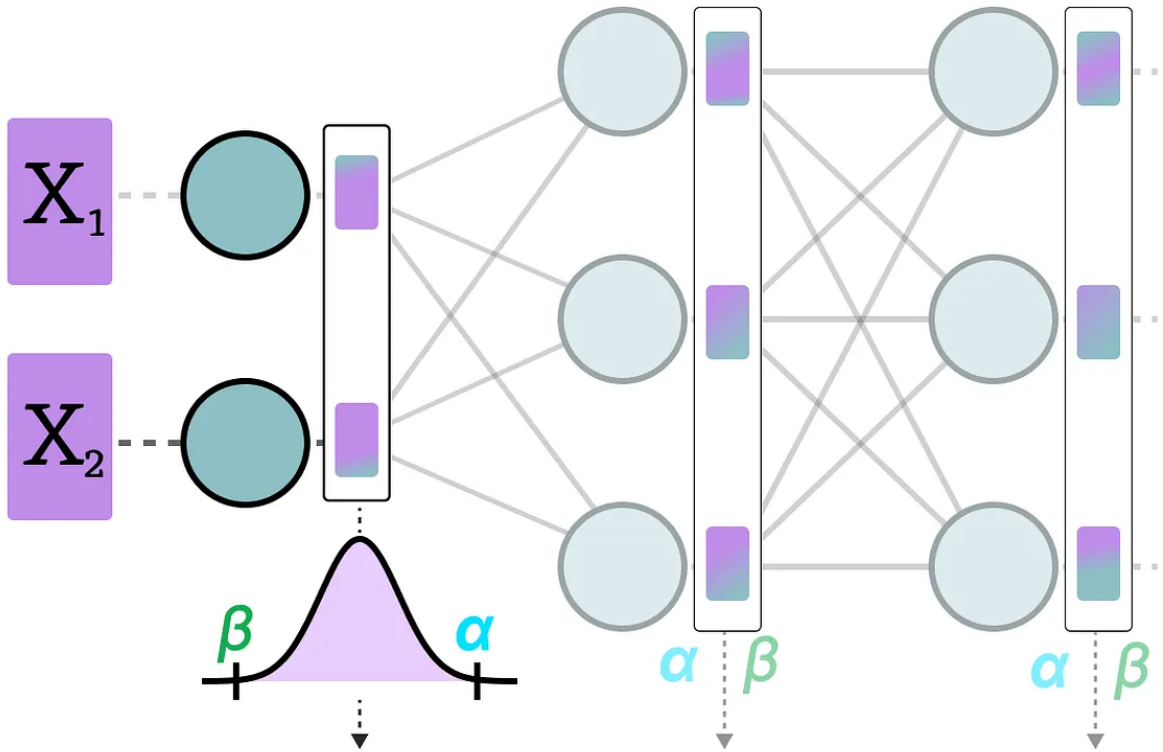
Quantization of the activations, however, requires inference of the model to get their potential distribution since we do not know their range.

There are two forms of quantization of the activations:

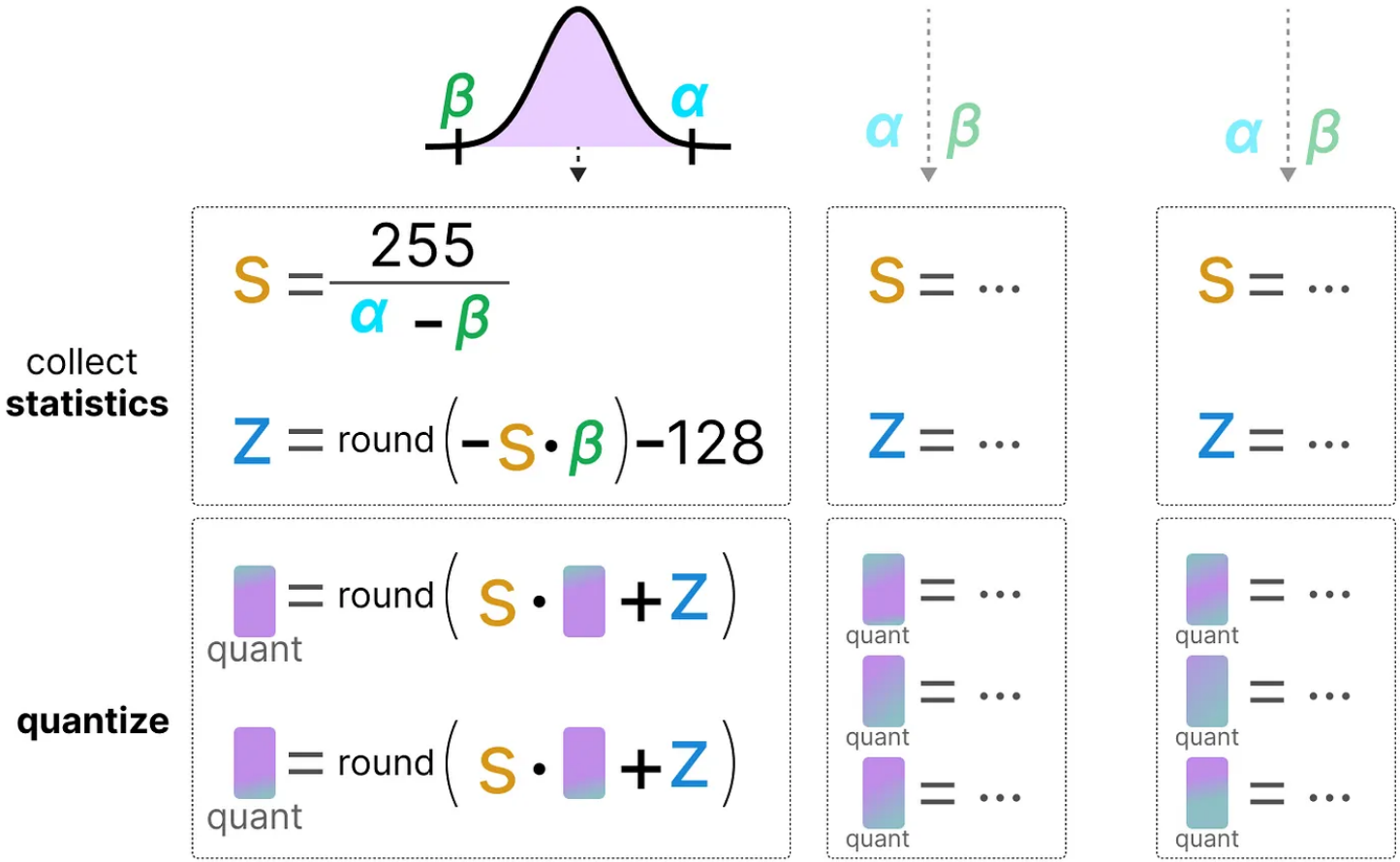
- Dynamic Quantization
- Static Quantization

Dynamic Quantization

After data passes a hidden layer, its activations are collected:



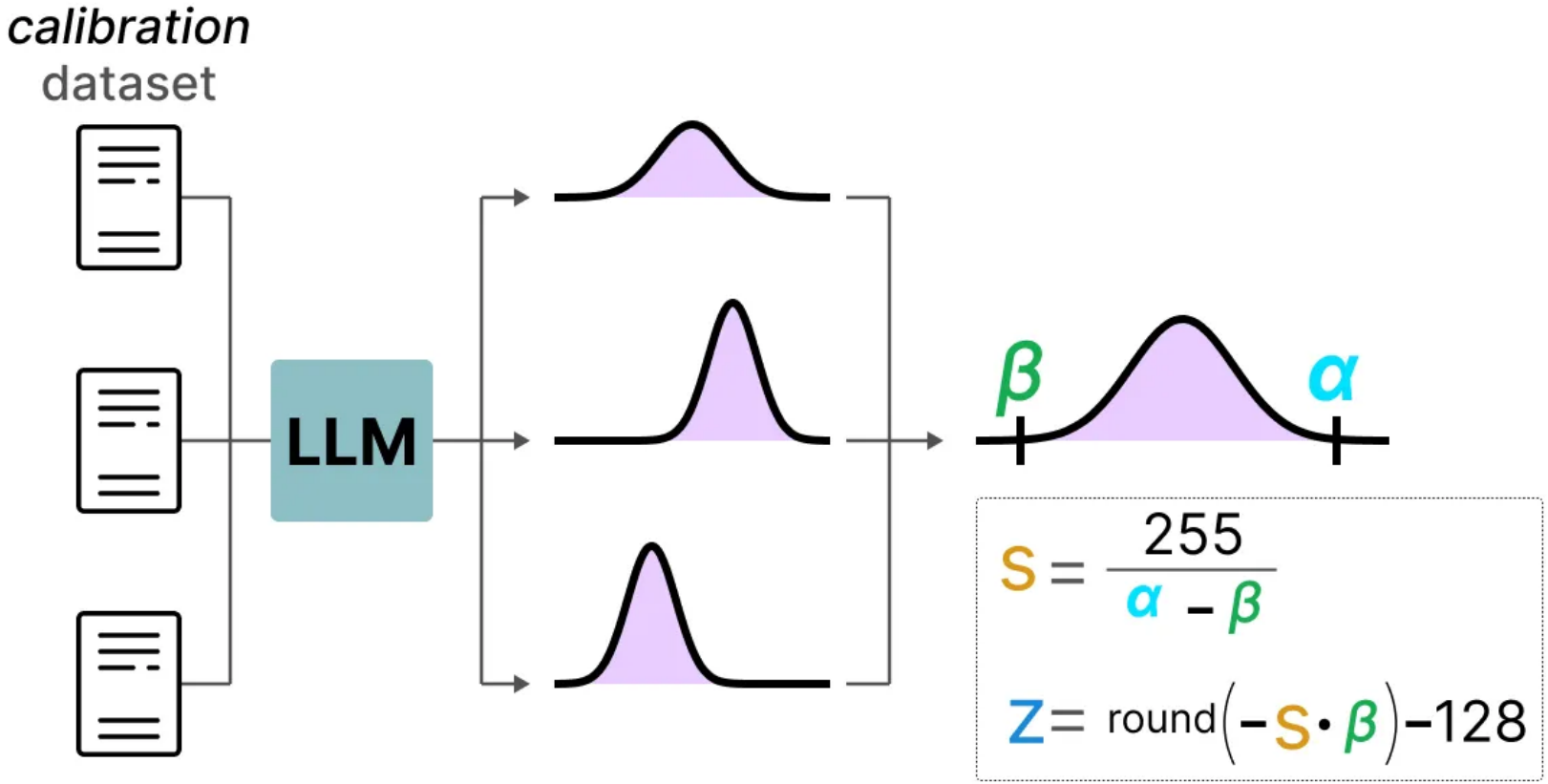
This distribution of activations is then used to calculate the zeropoint (z) and scale factor (s) values needed to quantize the output



The process is repeated each time data passes through a new layer. Therefore, each layer has its own separate z and s values and therefore different quantization schemes.

Static Quantization

In contrast to dynamic quantization, static quantization does not calculate the zeropoint (z) and scale factor (s) during inference but beforehand.
To find those values, a calibration dataset is used and given to the model to collect these potential distributions.



After these values have been collected, we can calculate the necessary s and z values to perform quantization during inference.

When you are performing actual inference, the s and z values are not recalculated but are used globally over all activations to quantize them.

In general, dynamic quantization tends to be a bit more accurate since it only attempts to calculate the s and z values per hidden layer. However, it might increase compute time as these values need to be calculated.

In contrast, static quantization is less accurate but is faster as it already knows the s and z values used for quantization.

Going below 8-bit quantization

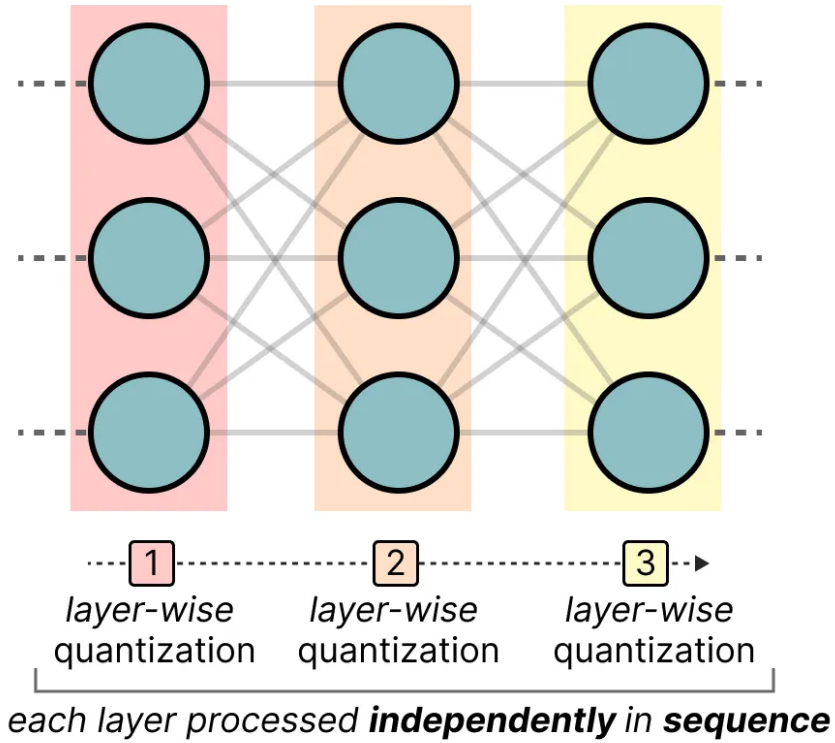
- Reducing precision below 8-bits is challenging because quantization error grows with each lost bit.
- Despite this, research has shown effective strategies for pushing models to 6-bit, 4-bit, and even 2-bit precision.
- Going lower than 4-bits is usually not recommended due to severe accuracy degradation.

Two widely used methods (shared on HuggingFace):

- **GPTQ** – full model quantization optimized for running on GPU.
- **GGUF** – flexible format that allows offloading layers to the CPU.

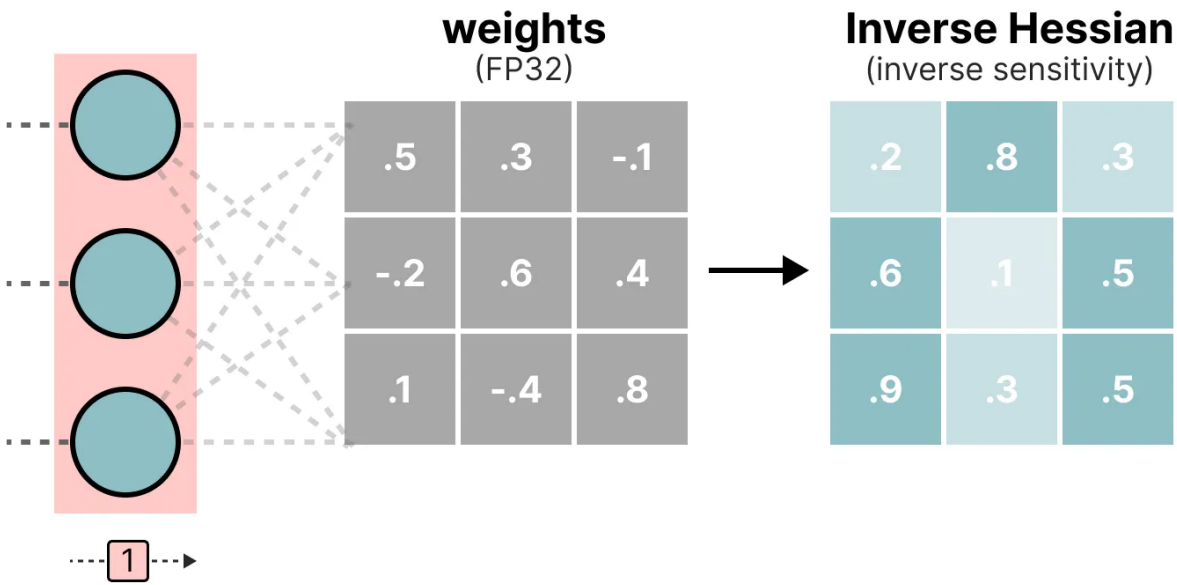
GPTQ: 4-bit Quantization

- GPTQ is one of the most widely used methods for 4-bit quantization in practice.
- Uses **asymmetric, layer-wise quantization**:
  - Each layer is quantized independently before moving to the next.



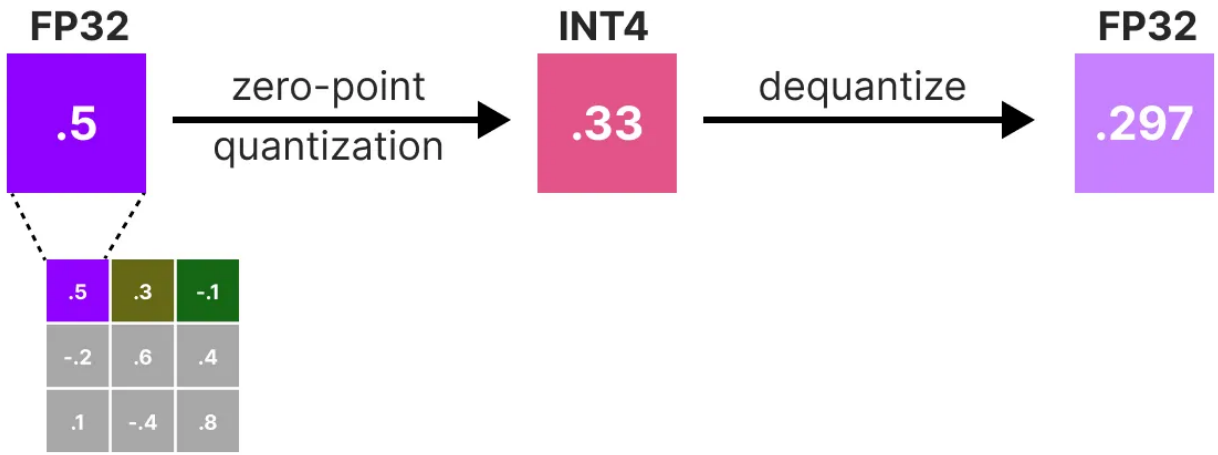
GPTQ: 4-bit Quantization

- During this process:
  - Layer weights are transformed using the **inverse Hessian**.
  - The Hessian (second-order derivative of the loss) measures sensitivity of outputs to changes in weights.
  - Its inverse highlights the relative **importance of each weight** in the layer.
- Simplified: GPTQ preserves the most important weights while reducing precision.



GPTQ: 4-bit Quantization

Next, we quantize and then dequantize the weight of the first row in our weight matrix:



This process allows us to calculate the quantization error (q) which we can weigh using the inverse-Hessian (h_1) that we calculated beforehand. Essentially, we are creating a weighted-quantization error based on the importance of the weight:

$$q = \frac{x_1 - x_1}{h_1}$$

(hessian-weighted quantization error)

$$q = \frac{.5 - .297}{.2} = .203$$

GPTQ: 4-bit Quantization

Next, we redistribute this weighted quantization error over the other weights in the row. This allows for maintaining the overall function and output of the network.
For example, if we were to do this for the second weight, namely .3(x_2), we would add the quantization error (q) multiplied by the inverse-Hessian of the second weight (h_2)

$$x_2 = x_2 + q \cdot h_2 \qquad \text{(update weight)}$$

$$x_2 = .3 + .203 \cdot .8$$

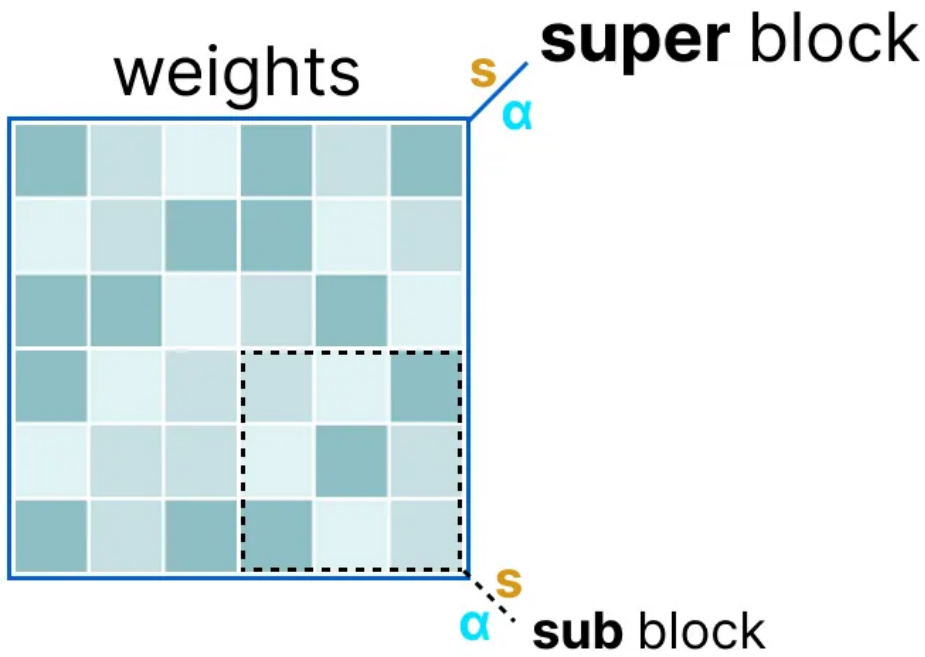
GPTQ: 4-bit Quantization

- The process iterates by redistributing the weighted quantization error until all values are quantized.
- This works effectively because:
  - Weights are often correlated.
  - When one weight has quantization error, related weights are adjusted using the inverse Hessian.
- **Implementation tricks** (for speed and performance):
  - Dampening factor applied to the Hessian
  - "Lazy batching" for efficiency
  - Precomputation via the Cholesky method
- **TIP:** Explore **EXL2** if you want a method optimized for performance and faster inference.
- **NOTE:** Highly recommend checking the YouTube lecture on GPTQ for deeper insights.

GGUF: Quantization with CPU Offloading

- While GPTQ is well-suited for running a full LLM on GPU, you may not always have enough VRAM available.
- **GGUF** allows flexible offloading; any layer of the LLM can be executed on the CPU.
- This enables hybrid inference using both CPU and GPU together.
- GGUF evolves frequently, and its efficiency depends on the bit-width of quantization.

GGUF



General principle:

- The weights of each layer are divided into **super-blocks**, each containing multiple **sub-blocks**.
- From these blocks, two key parameters are extracted:
  - Scale factor (s)
  - Alpha (α)

To quantize a given "sub" block, we can use the absmax quantization we used before. Remember that it multiplies a given weight by the scale factor (s):

$$X_{quantized} = S \cdot X \qquad \text{(absmax quantization)}$$

The scale factor is calculated using the information from the "sub" block but is quantized using the information from the "super" block which has its own scale factor:

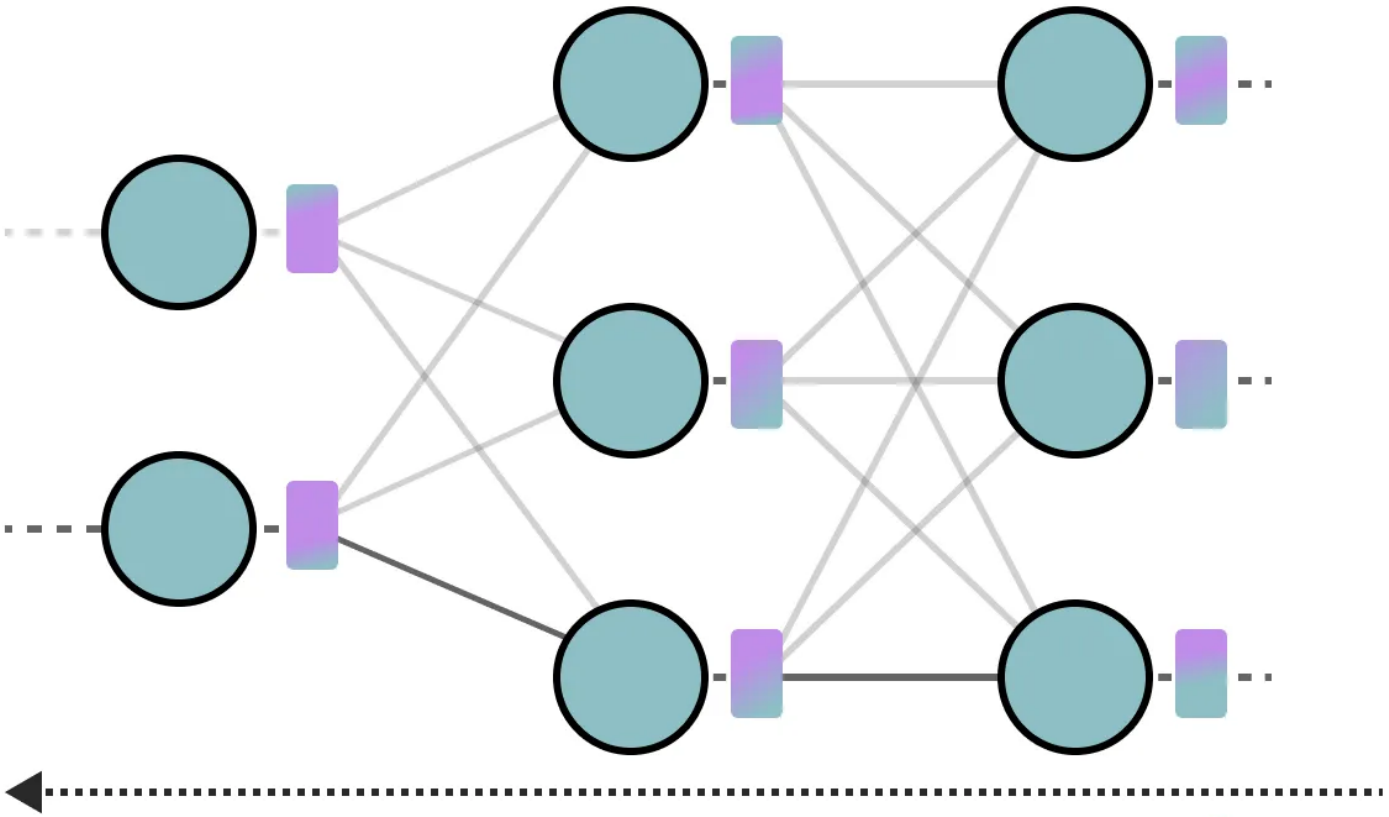
$$X_{quantized} = \underbrace{S_{sub}}_{\substack{\textit{quantized using} \\ S_{super}}} \cdot X \qquad \text{(absmax quantization)}$$

This block-wise quantization uses the scale factor (*s_super*) from the "super" block to quantize the scale factor (*s_sub*) from the "sub" block.

Quantization-aware training (QAT)

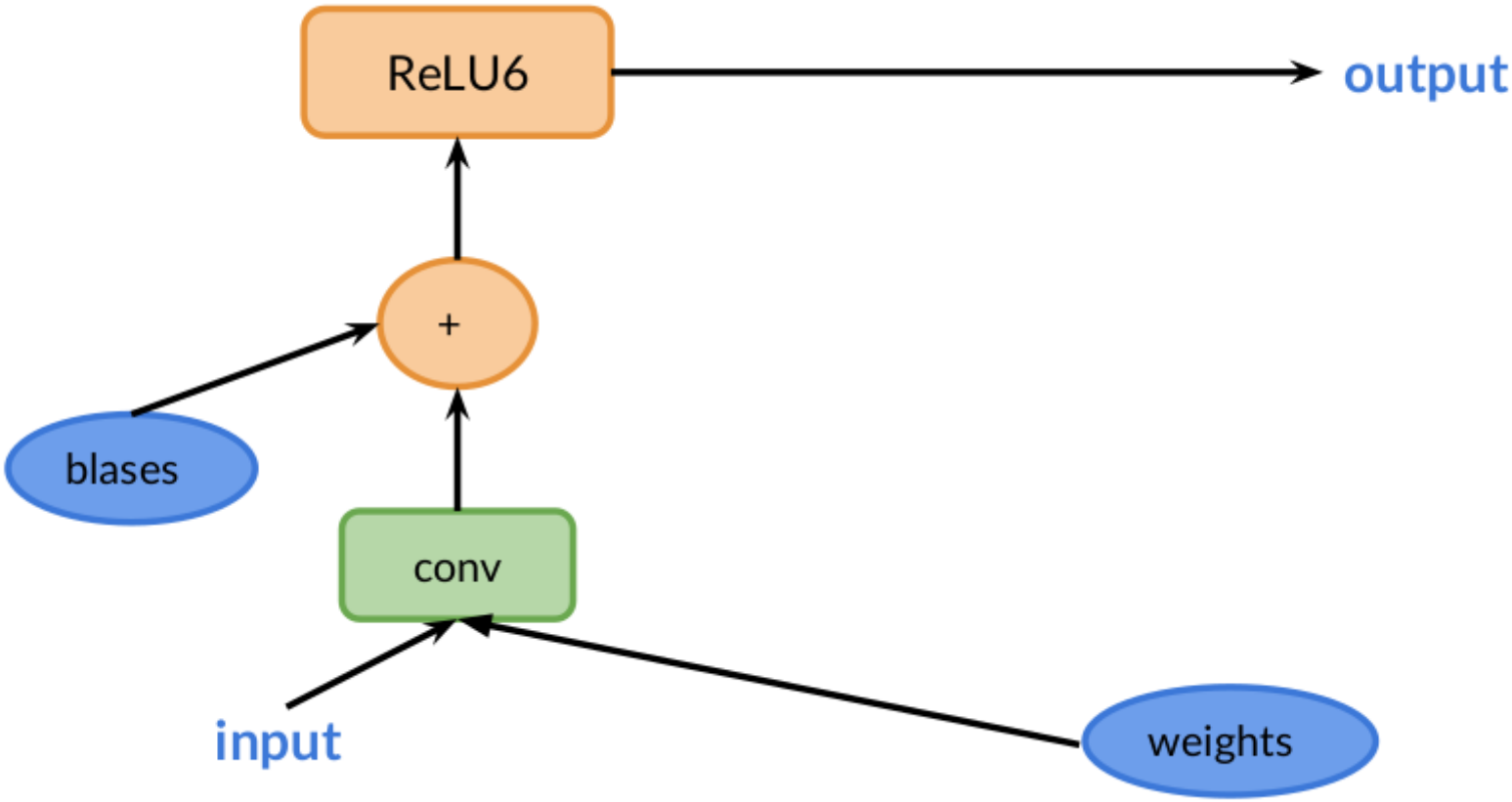
- Inserts **fake quantization (FQ)** nodes during the forward pass.
- Rewrites the computation graph to **simulate quantized inference**.
- Helps reduce the accuracy loss that typically comes from post-training quantization.
- The final trained model includes all metadata needed to be quantized according to specification.

Quantization-aware training (QAT)



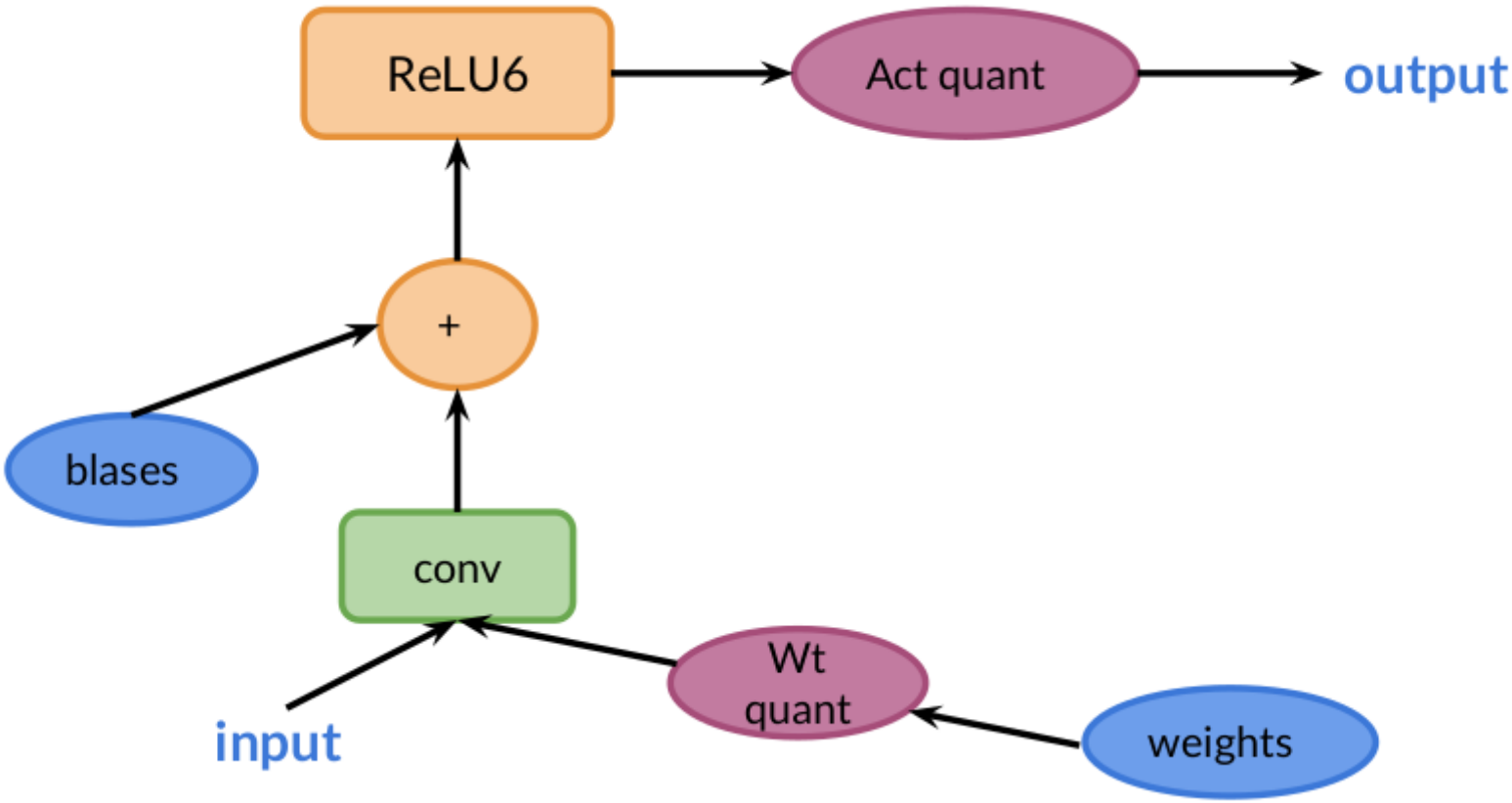
Quantization-aware training (QAT)

Adding the quantization emulation operations

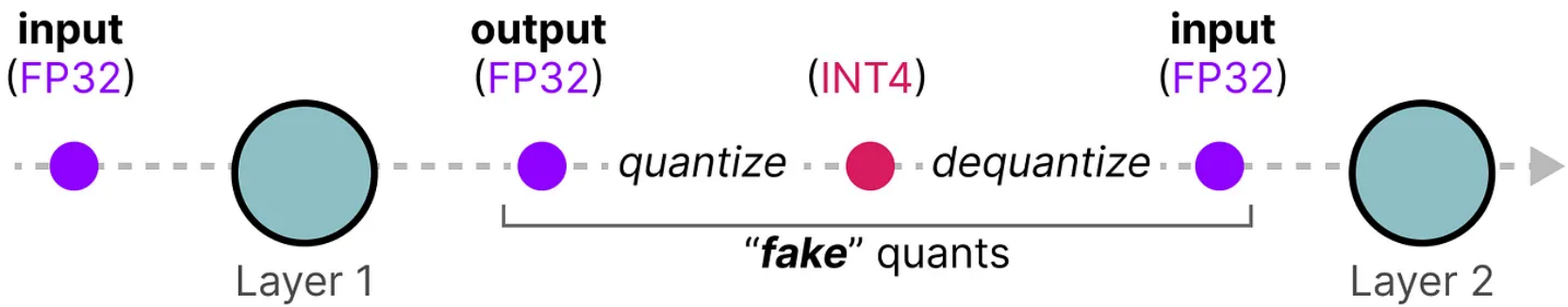


Quantization-aware training (QAT)

Adding the quantization emulation operations

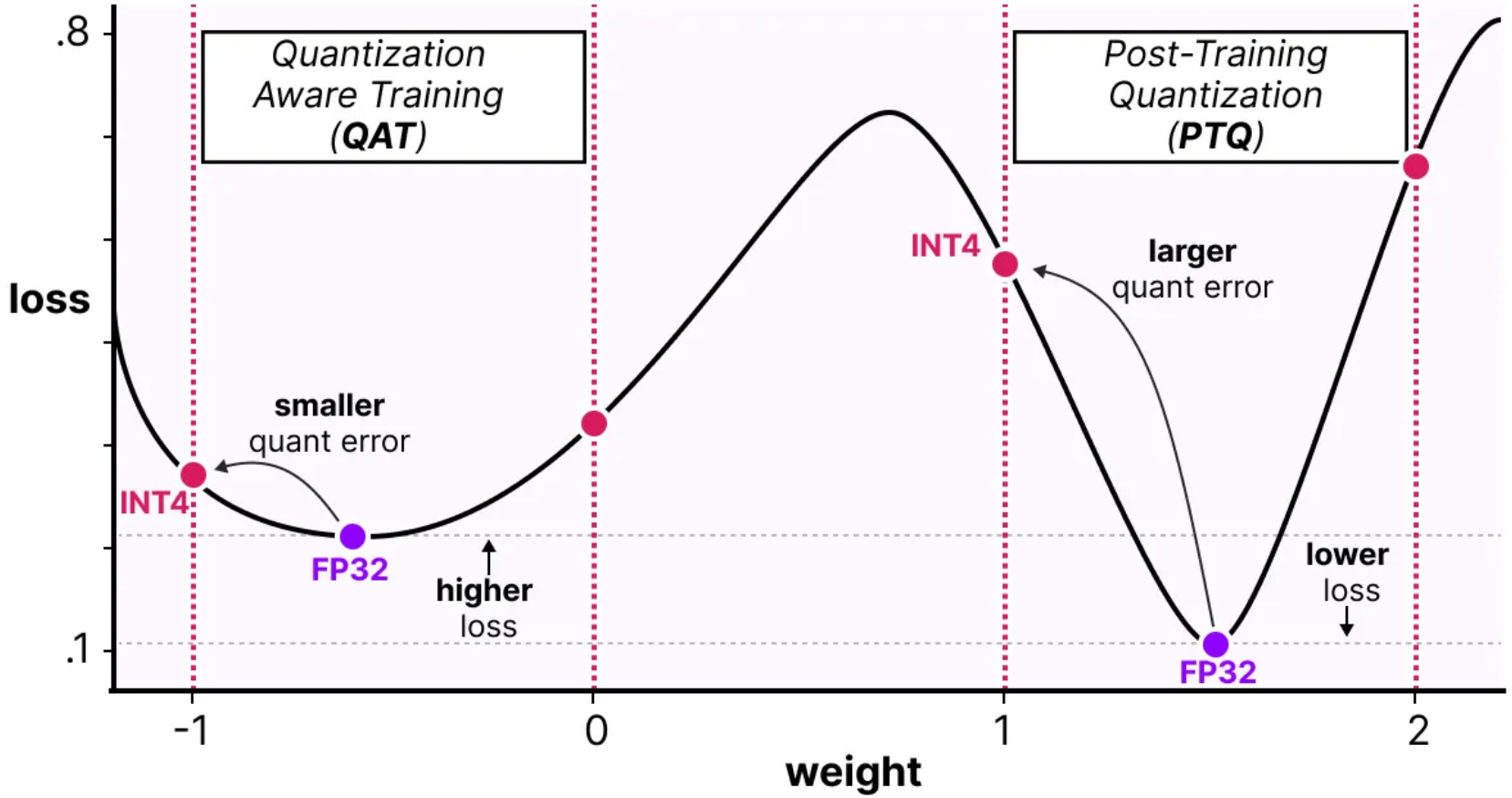


During training, so-called "fake" quants are introduced. This is the process of first quantizing the weights to, for example, INT4 and then dequantizing back to FP32:



This process allows the model to consider the quantization process during training, the calculation of loss, and weight updates.

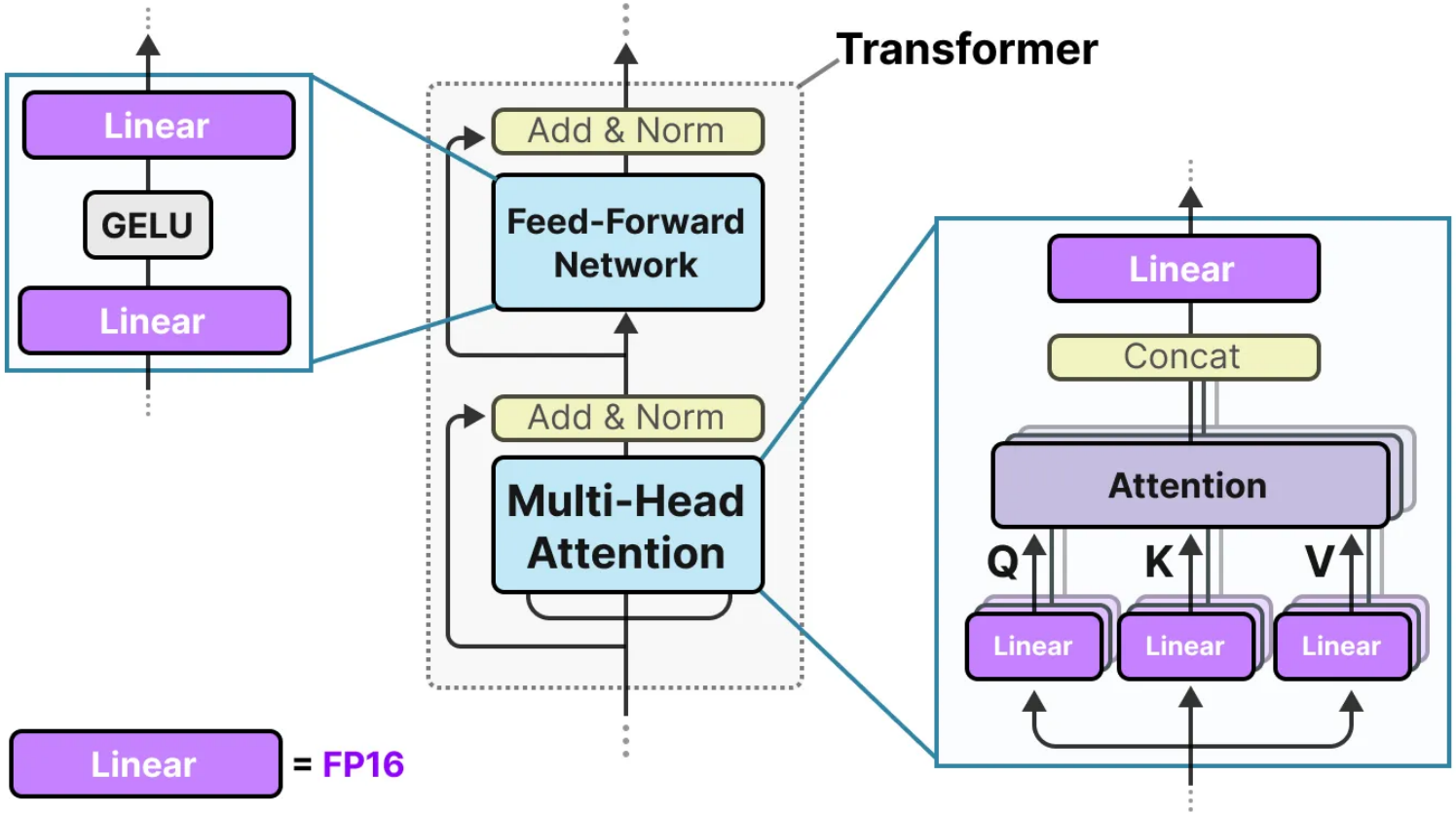
QAT attempts to explore the loss landscape for "wide" minima to minimize the quantization errors as "narrow" minima tend to result in larger quantization errors.



For example, imagine if we did not consider quantization during the backward pass. We choose the weight with the smallest loss according to gradient descent. However, that would introduce a larger quantization error if it's in a "narrow" minima.

BitNet

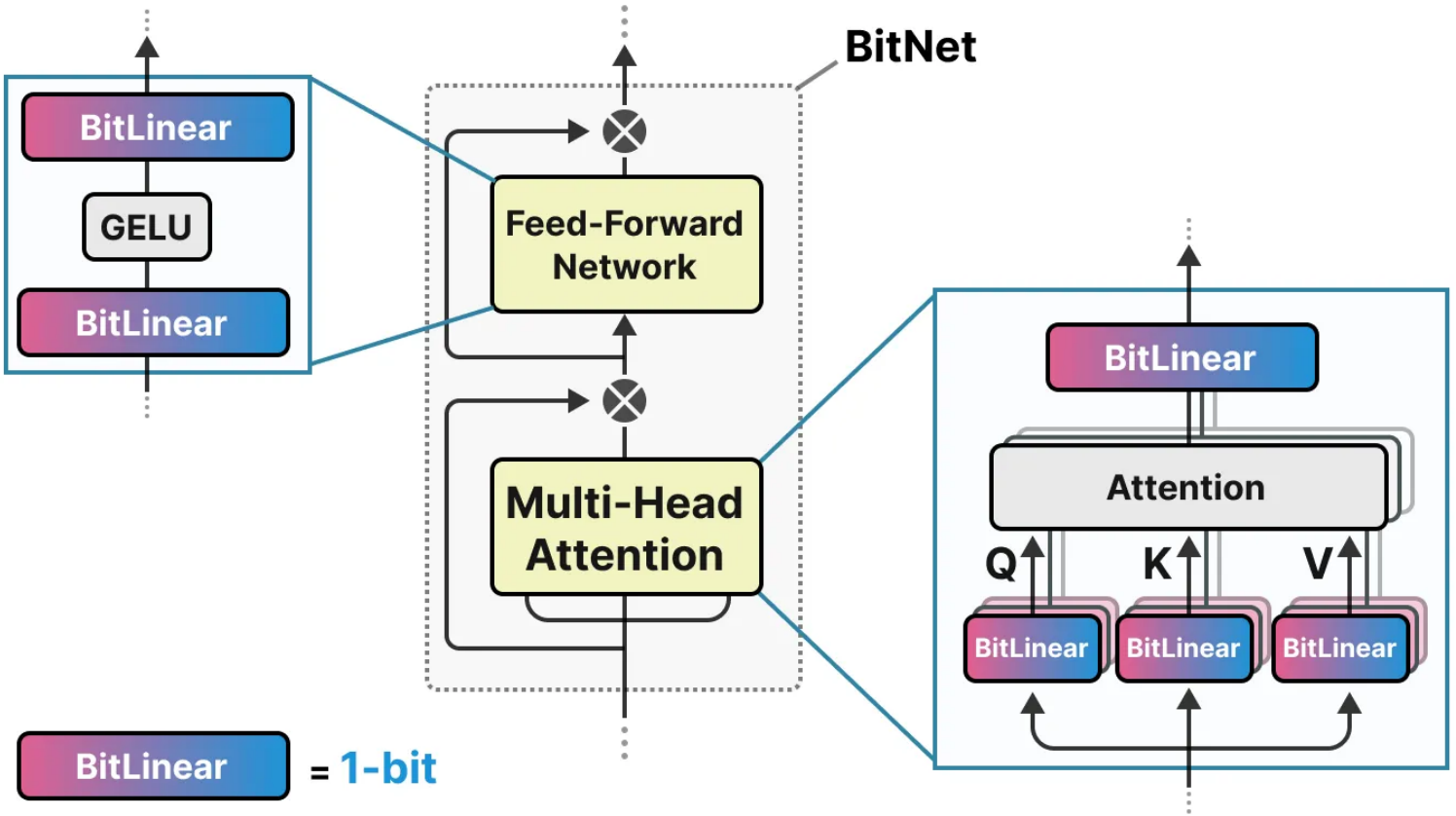
representing the weights of a model single 1-bit, using either -1 or 1 for a given weight.3
It does so by injecting the quantization process directly into the Transformer architecture.



These linear layers are generally represented with higher precision, like FP16, and are where most of the weights reside.

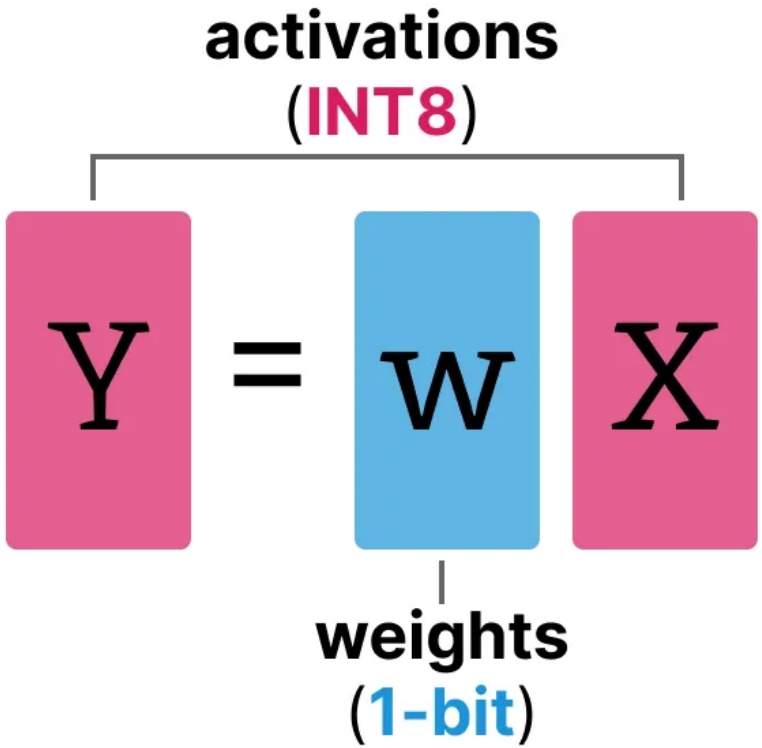
BitNet

BitNet replaces these linear layers with something they call the BitLlinear:

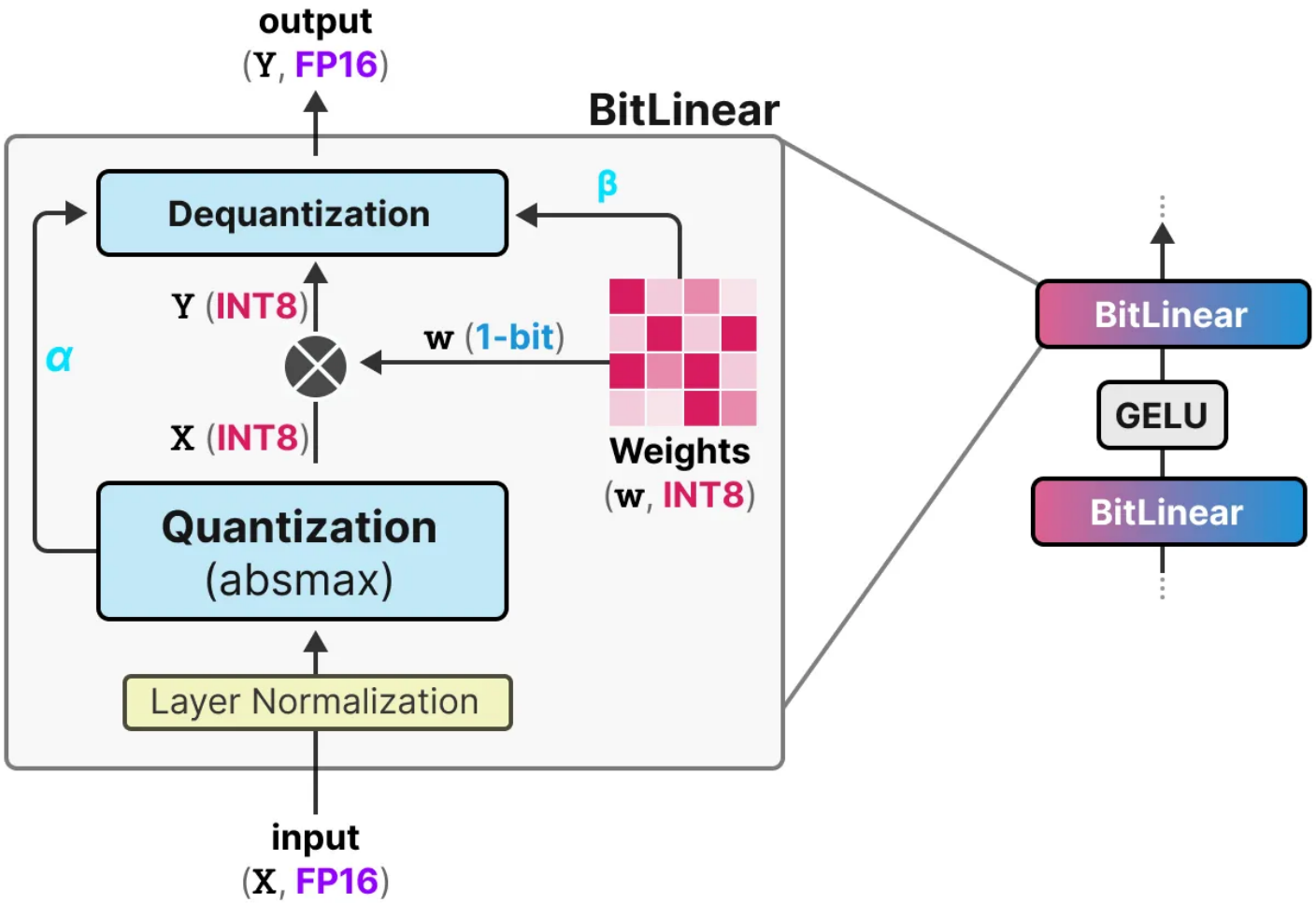


A BitLinear layer works the same as a regular linear layer and calculates the output based on the weights multiplied by the activation.

In contrast, a BitLinear layer represents the weights of a model using 1-bit and activations using INT8:



A BitLinear layer, like Quantization-Aware Training (QAT) performs a form of "fake" quantization during training to analyze the effect of quantization of the weights and activations:

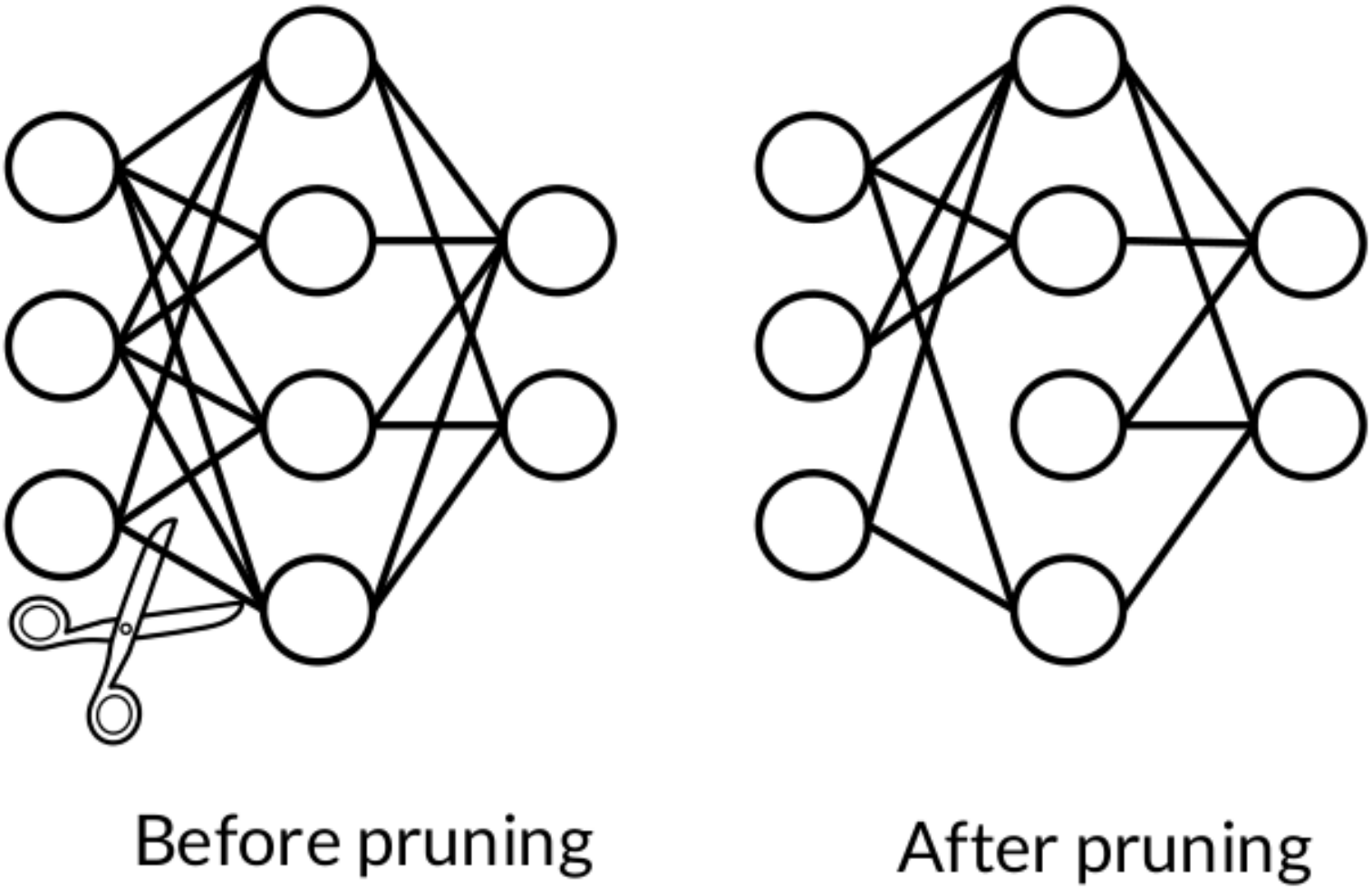


This procedure is relatively straightforward and allows models to be represented with only two values, either -1 or 1.

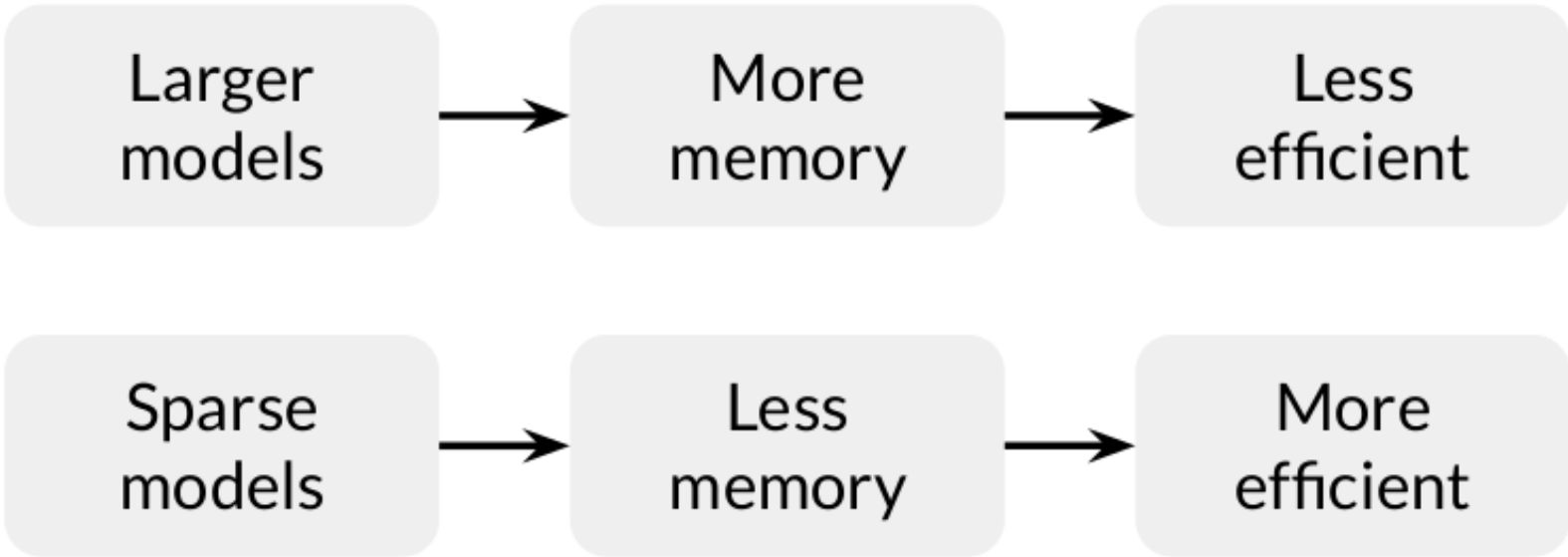
Quantization and Pruning

Pruning

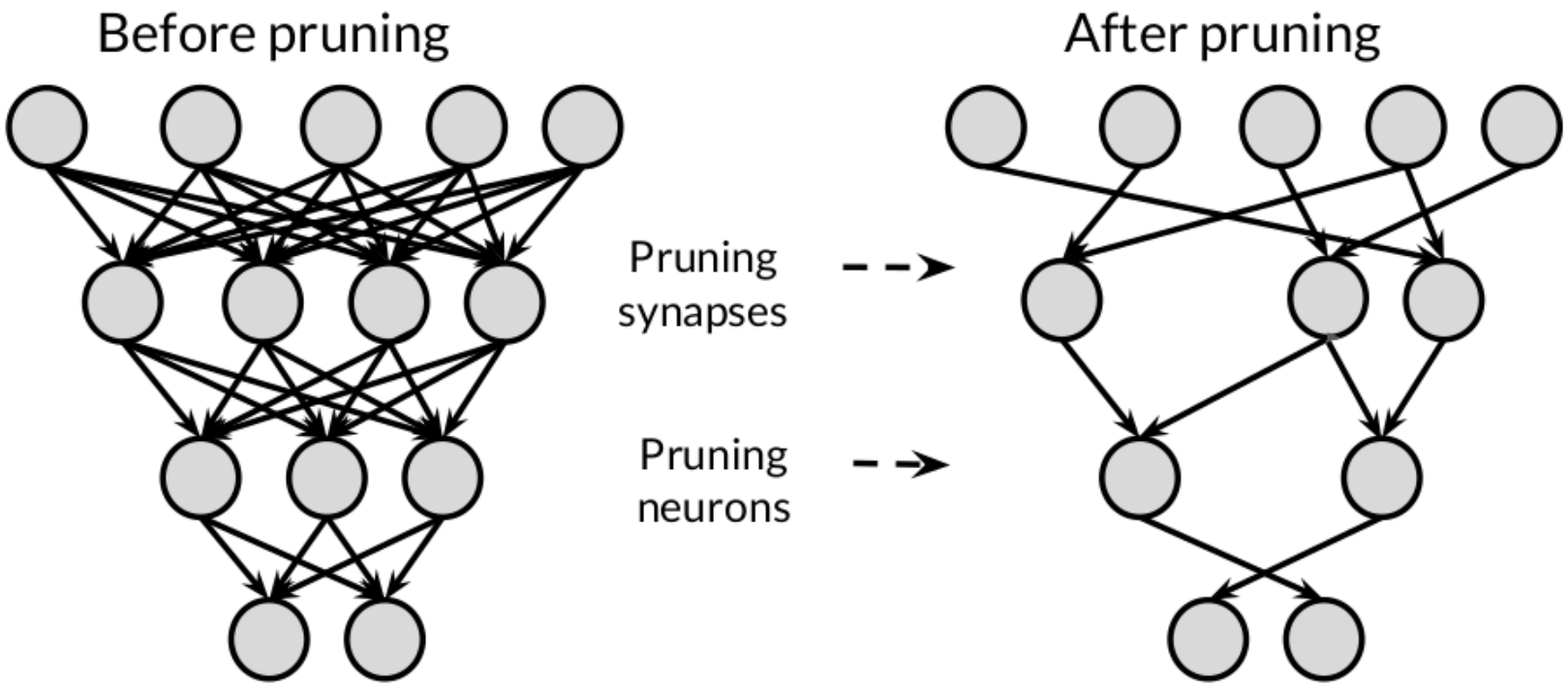
Connection pruning



Model sparsity



Origins of weight pruning



The Lottery Ticket Hypothesis

$$p = \frac{1}{3000000}$$

$$\bar{p} = 1 - p$$

$$p_n = 1 - (1 - p)^n$$

Finding Sparse Neural Networks

"A randomly-initialized, dense neural network contains a subnetwork that is initialized such that — when trained in isolation — it can match the test accuracy of the original network after training for at most the same number of iterations."

— Jonathan Frankle and Michael Carbin

Pruning research is evolving

- The new method did not perform well at large scale
- The new method failed to identify the randomly initialized winners
- Pruning remains an active area of research

Eliminate connections based on their magnitude

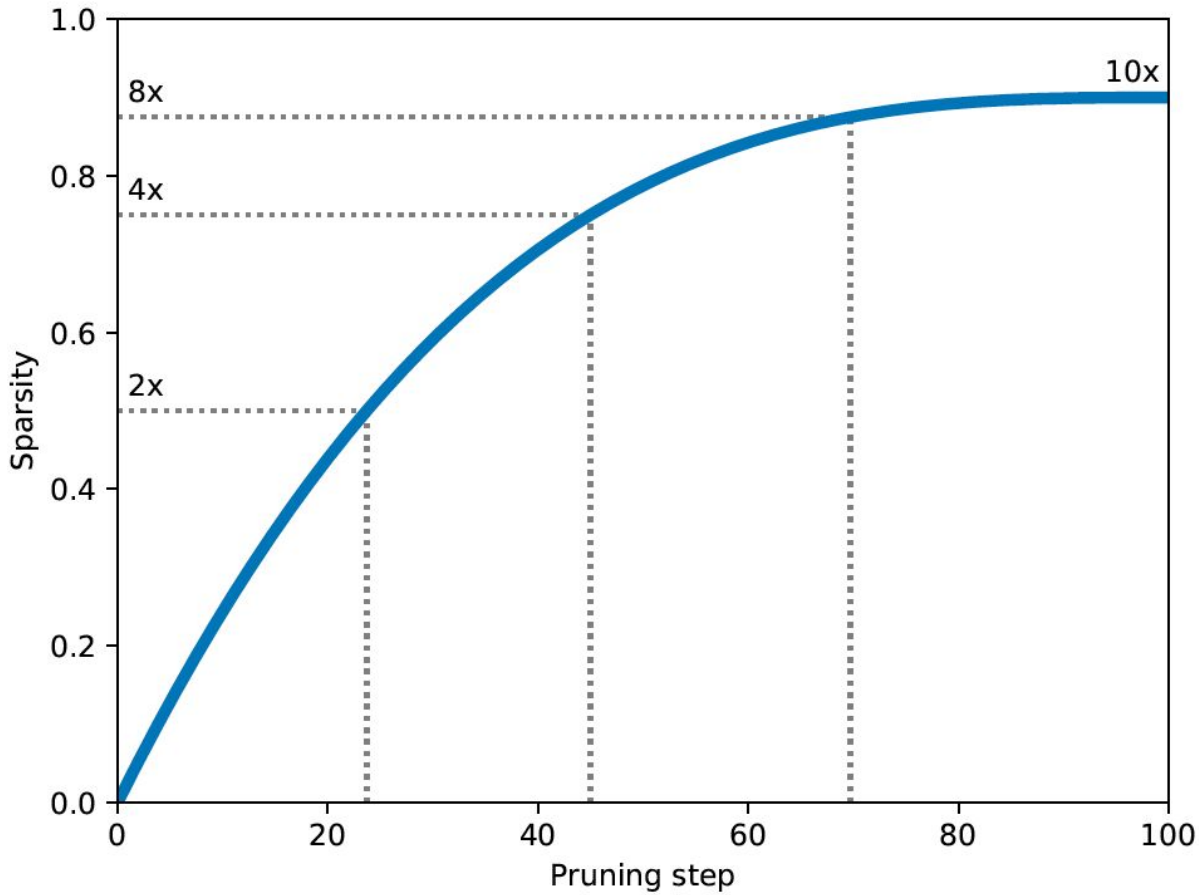
3	2	7	4
9	6	3	8
4	4	1	3
2	3	2	5

0	2	0	4
0	6	3	0
4	0	0	3
0	3	0	5

0	0	7	4
9	6	0	0
0	0	1	3
2	3	0	0

Tensors with no sparsity (left), sparsity in blocks of 1x1 (center), and the sparsity in blocks 1x2 (right)

Apply sparsity with a pruning routine



Example of sparsity ramp-up function with a schedule to start pruning from step 0 until step 100, and a final target sparsity of 90%.

What's special about pruning?

- Better storage and/or transmission
- Gain speedups in CPU and some ML accelerators
- Can be combined with quantization for additional benefits
- Unlock performance improvements

Results across different models and tasks

Model	Non-sparse Top-1 acc.	Sparse acc.	Sparsity
Inception V3	78.1%	78.0%	50%
		76.1%	75%
		74.6%	87.5%
Mobilenet V1 224	71.04%	70.84%	50%

Model	Non-sparse BLEU	Sparse BLEU	Sparsity
GNMT EN-DE	26.77	26.86	80%
		26.52	85%
		26.19	90%
GNMT DE-EN	29.47	29.50	80%
		29.24	85%
		28.81	90%

Labs for This Week

Objective

Briefly describe the learning goal for this week's lab(s).

Lab Activities:

- Lab 8: [Quantization and Pruning] — [Quantization Tutorial]
- Lab 8: [Docker Container] — [Docker Tutorial]
- Lab 8: [LLMs] - [Quantization]

Submission Deadline: [Before the next class]

- Assignment 8: [Docker] — [Create a container of your choice]
- Assignment 8: [Feature Selection] — [Run the lab on Feature Selection]
- Assignment 8: [LLMs] — [Quantize a model of your choice]

 DeepLearning.AI

 The People + AI Guidebook

 A Visual Guide to Quantization