



# TinyML Compression



# Portability Trade-offs



Sacrifice **portability**  
across systems for  
**efficiency**.



Specific HW Implementation of a Library

Option 1

Universal Code Portability/Compatibility



Cost (\$)



Power Consumption (W)



Engineering Effort



Option 2

Lower Code Portability



Cost (\$)



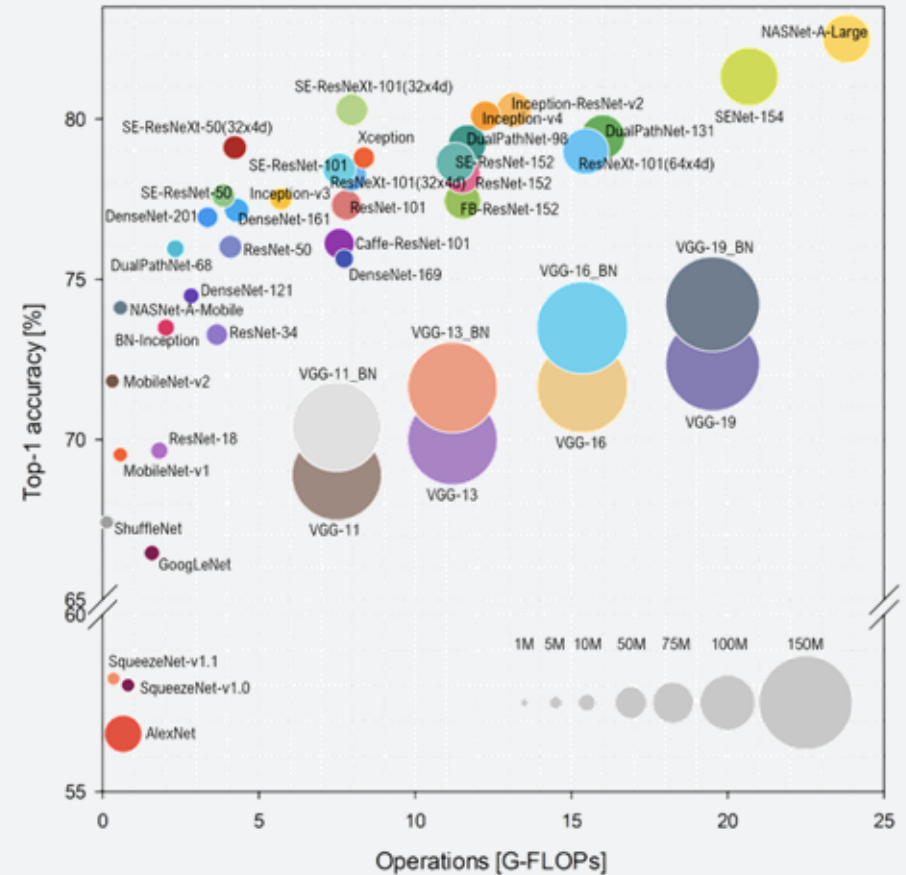
Power (W)



Eng. Effort



# ML Model Evolution

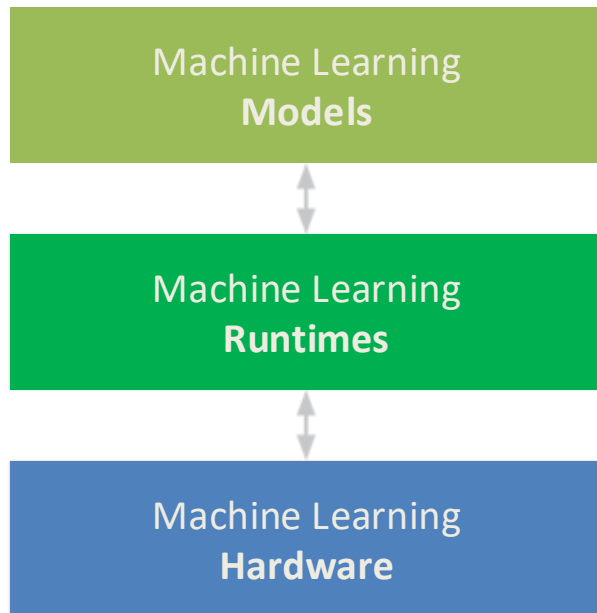


**Source:** S. Bianco, R. Cadene, L. Celona, and P. Napolitano, "Benchmark analysis of representative deep neural network architectures," *IEEE Access*, vol. 6, pp. 64 270–64 277, 2018



# Model Comparisons

Model	Version / Variant	Top-1 Accuracy (ImageNet)	Approx. Model Size / Parameters	Comments
<b>AlexNet</b>	AlexNet (2012)	~ 63 % top-1 reported historically. ( <a href="#">AI Summer</a> )	~ 60 M parameters, ~ hundreds of MB model size. ( <a href="#">Wikipedia</a> )	One of the first deep CNNs, now mostly a teaching tool rather than state-of-the-art.
<b>VGGNet</b>	VGG16 / VGG19	VGG16: ~ 71.3 % top-1 in Keras Applications table. ( <a href="#">keras.io</a> )	~ 138 M parameters (VGG16); ~ 528 MB size in Keras table. ( <a href="#">keras.io</a> )	Simple, uniform architecture; large size & heavy compute.
<b>ResNet</b>	ResNet152V2 (as a recent version)	ResNet152V2: ~ 78.0 % top-1 (Keras table) ( <a href="#">keras.io</a> )	~ 60.4 M parameters (ResNet152/152V2) ( <a href="#">keras.io</a> )	Deep residual networks; very good accuracy vs older nets with more efficient size than VGG.
<b>MobileNet</b>	MobileNetV3 (Large)	~ 75.2 % top-1 for MobileNetV3-Large in original paper. ( <a href="#">arXiv</a> )	~ 5.48 M parameters for V3-Large (source) ( <a href="#">GitHub</a> )	Designed for mobile / edge devices: very efficient size and latency trade-off.



## Model Compression Techniques

Pruning

Quantization

Knowledge

Distillation

...

# Optimization in Tiny Devices

Embedded ML

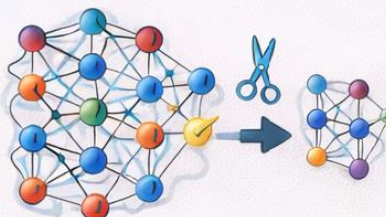


## Quantization



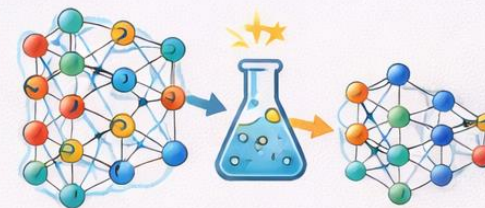
- Reduce model precision to integers instead of floats

## Model Pruning



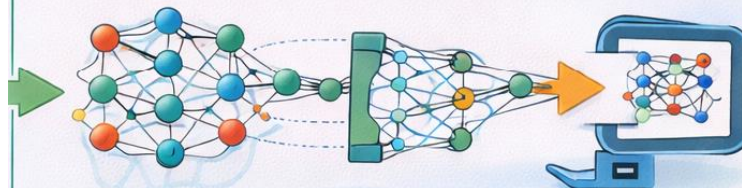
- Remove unimportant neurons & connections

## Knowledge Distillation



- Train a small model to mimic a
- a Large, accurate teacher Model

## Model Compression



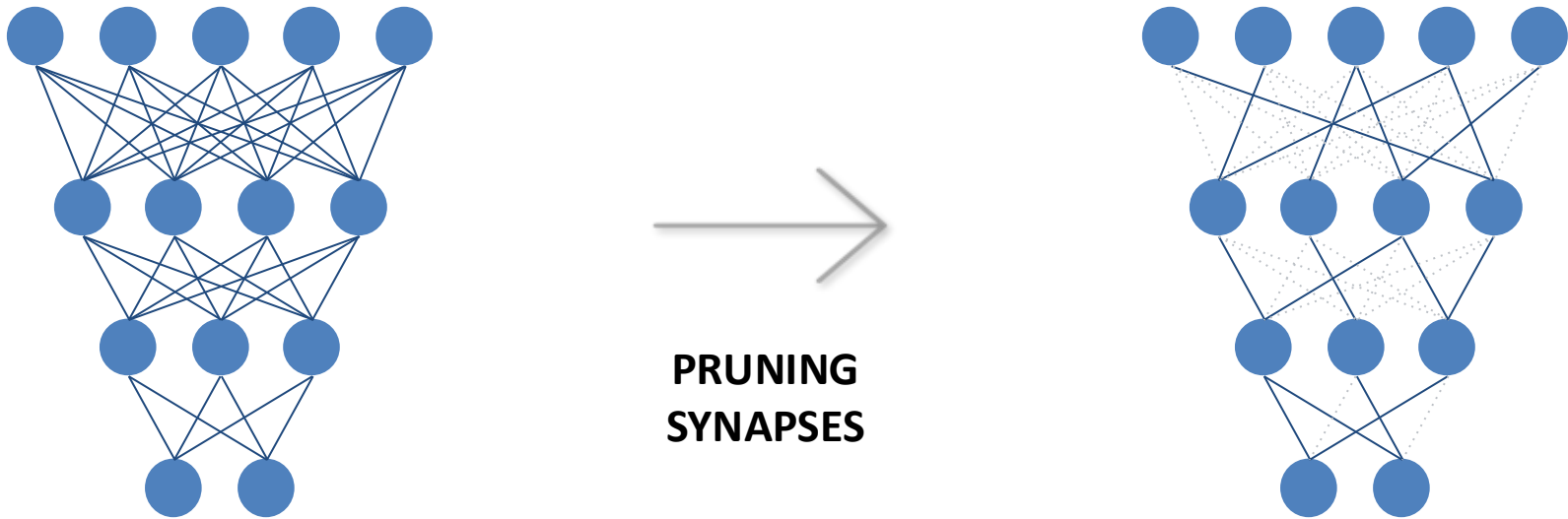
- Combine techniques to shrink models
- Quantized & pruned
- Distillation applied

“Smaller models run faster, use less memory, and consume less power”



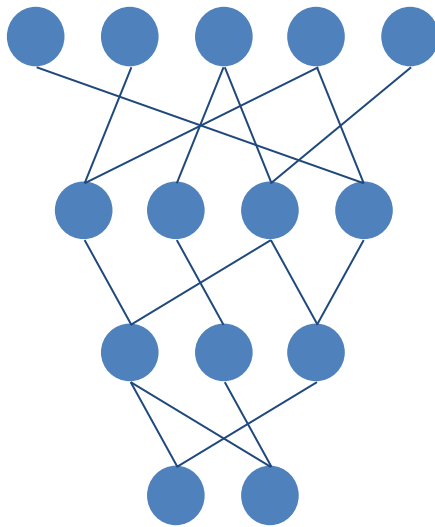
# Pruning

Pruning is one model compression technique that allows the model to be optimized for real-time inference for resource-constrained devices.

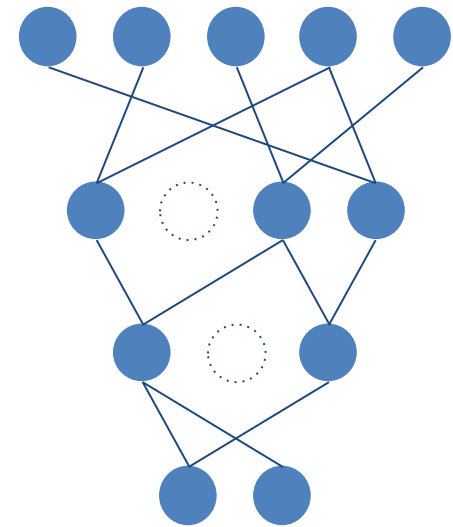




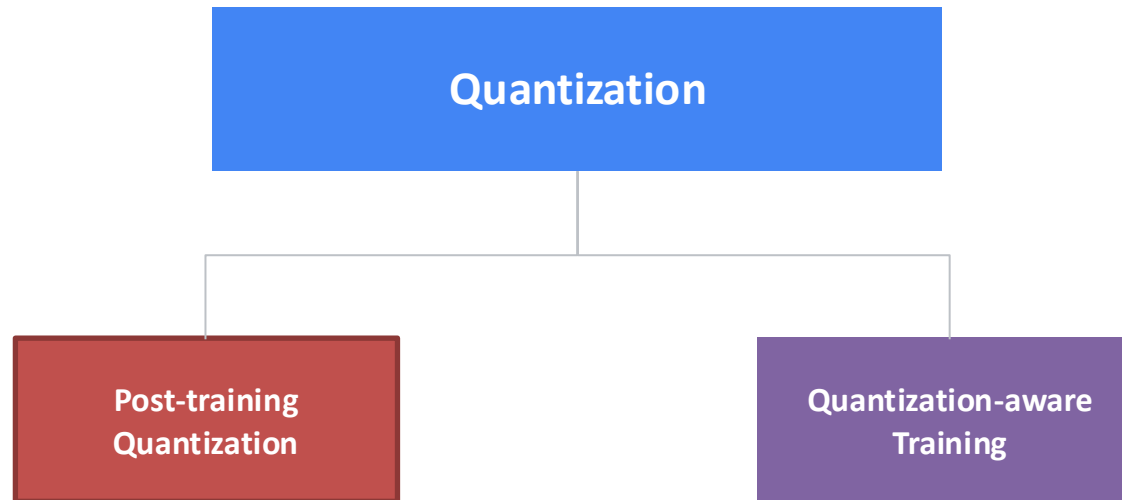
# Pruning



**PRUNING  
NEURONS**







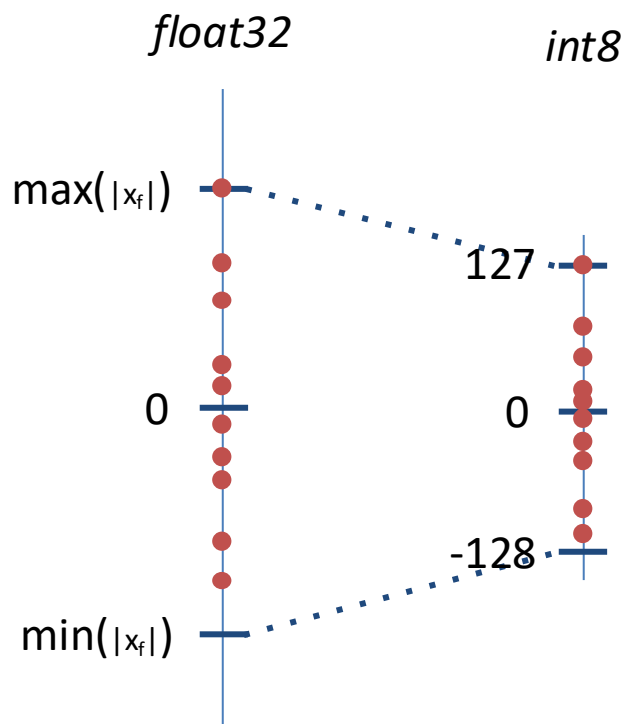
Post-training Optimization Tool (POT) is the fastest and easiest way to get a quantized model. A conversion technique that can reduce model size while also improving CPU and hardware accelerator latency, with little degradation in model accuracy.

Quantization aware training *emulates inference-time quantization*, creating a model that downstream tools will use to produce actually quantized models. The quantized models use lower-precision (e.g. 8-bit instead of 32-bit float), leading to benefits during deployment.



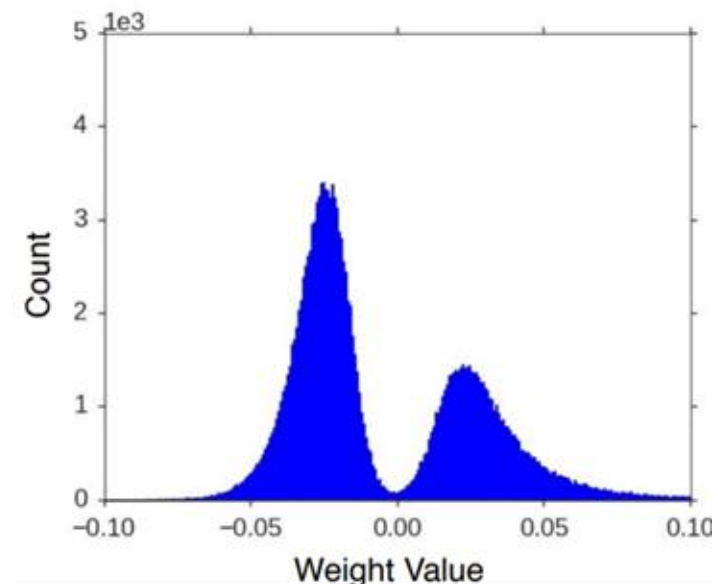
# Quantization

Quantization is an optimization that works by reducing the precision of the numbers used to represent a model's parameters, which by default are 32-bit floating point numbers. This results in a smaller model size, better portability and faster computation.



Why it works?

Weight distribution for AlexNet shows how most weight values are concentrated in a small range.

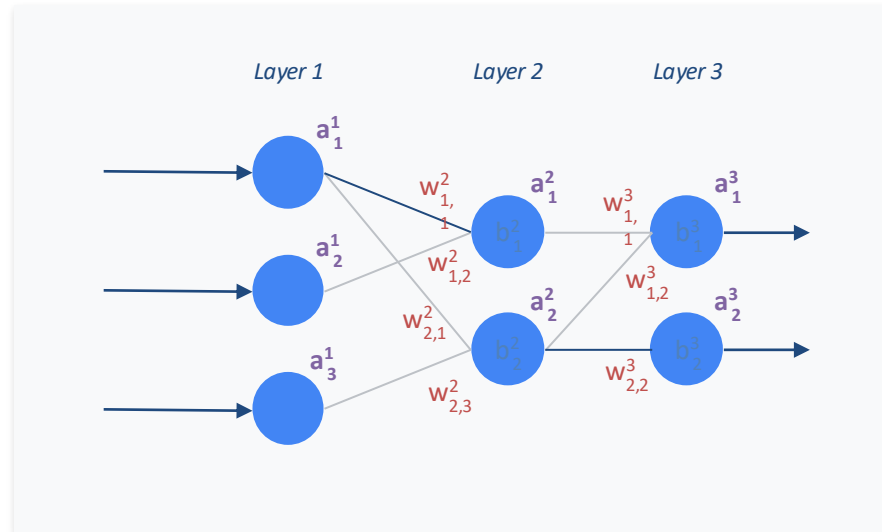




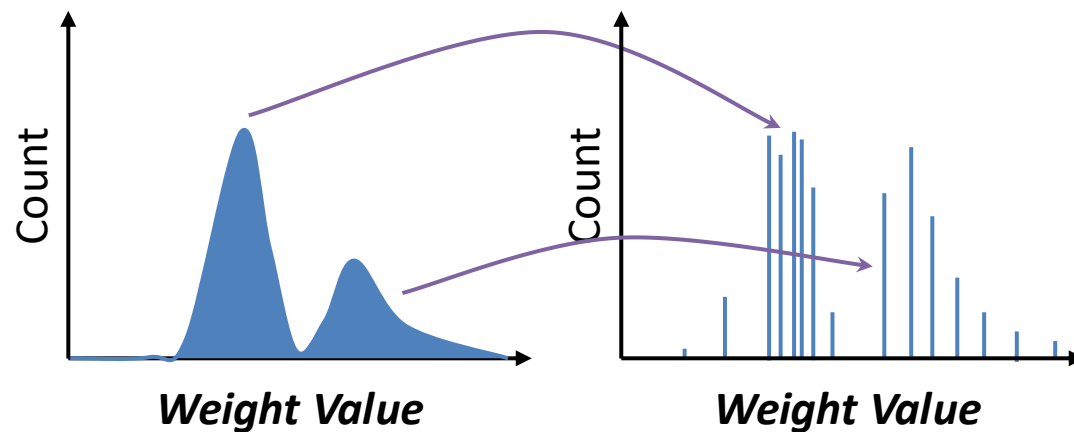
# What to Quantize?

Quantize

Weights
Biases
Activations

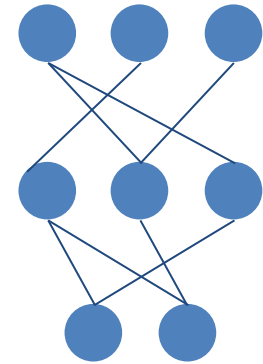
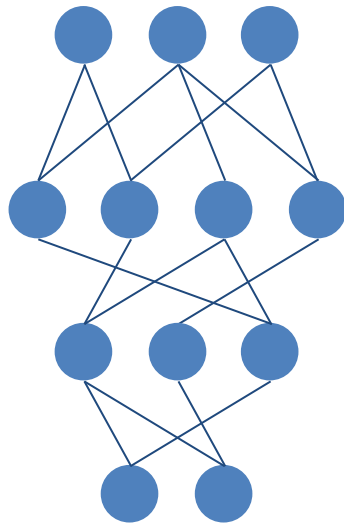


Reduce Precision (Discretize)



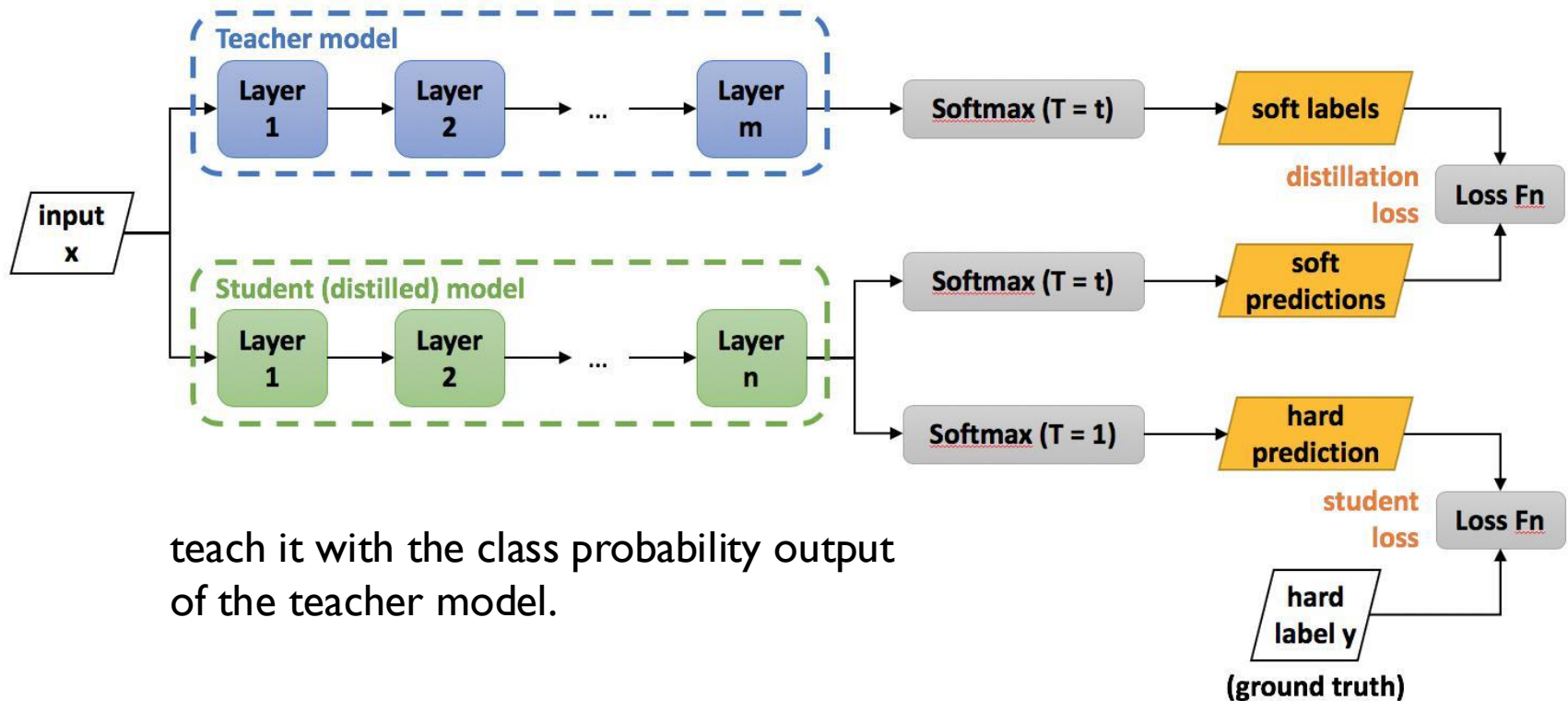


# Knowledge Distillation



**knowledge distillation** is the process of transferring knowledge from a large model to a smaller one.

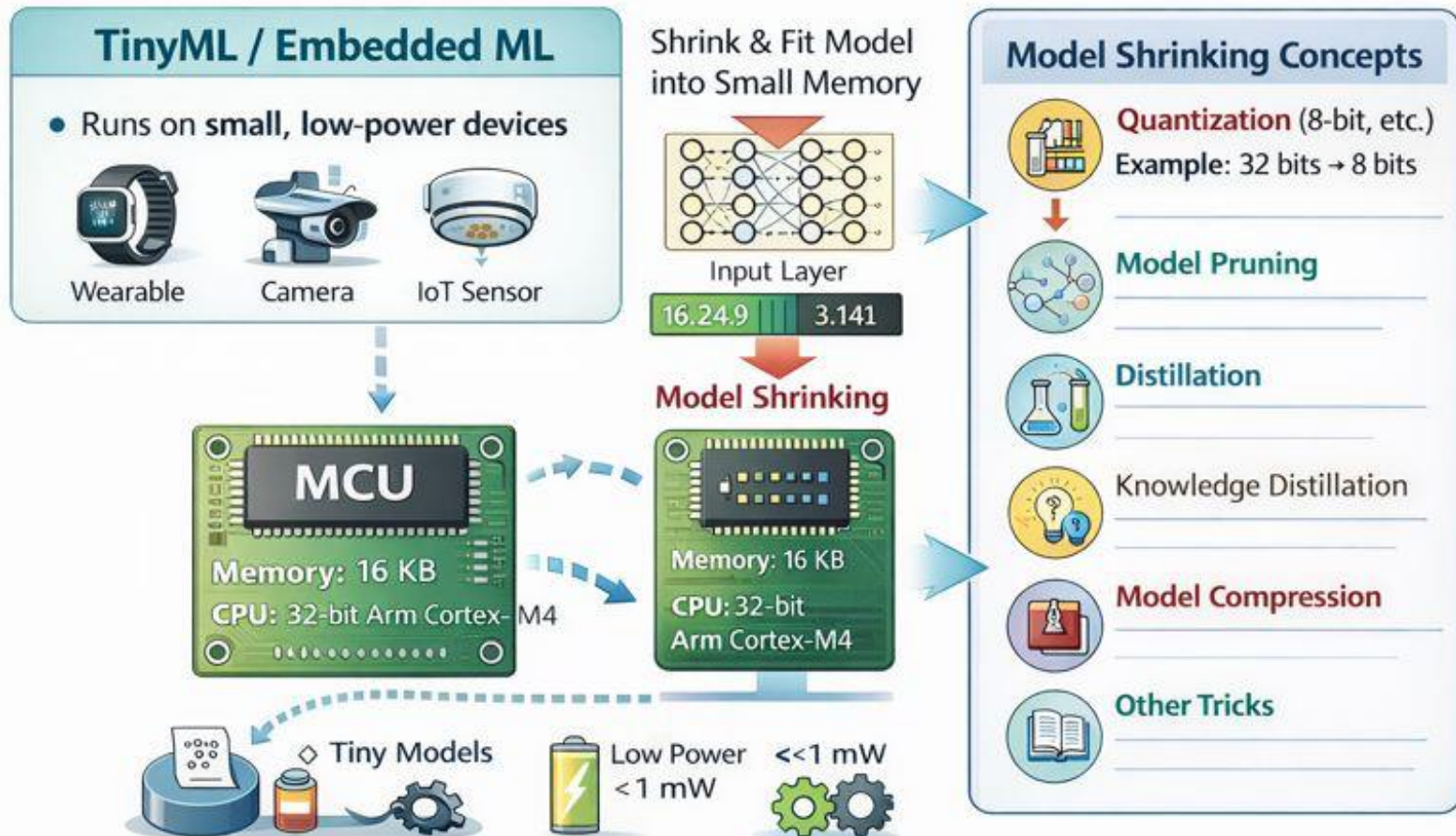
# Transferring Knowledge



teach it with the class probability output of the teacher model.

# TinyML

Embedded ML





# Deploy in the Tiny Device

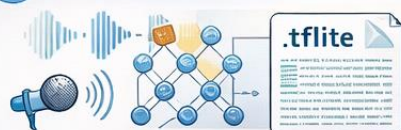
Embedded ML



## Audio ML Workflow: TFLite Model to Arduino (M5Core2)



### 1 Train Audio Classifier (PC)



- Train audio classifier with TensorFlow, output <model.tflite>

### 2 Quantize Model (INT8)

Quantize model to INT8  
(reduce size for microcontroller)



- Use "xxd" to convert to C/C++ header/source array

### 3 Convert TFLite to C Array (.cc /.h)

```
xxd -i model_int8.tflite > model_data.cc  
const unsigned char model_int8_tflite[] = { .... ;  
const int model_int8_tflite_len = ..... ;
```

### Arduino Sketch (M5Core2)

- Add model\_data.cc, config memory (arena)
- Libraries (TFLM), feature extraction (MFCC), inference loop

### 4 Arduino Sketch (M5Core2)

- Add model\_data.cc, config memory (arena)
- libraries (TFLM)
- I2S mic (16 kHz)
- Log-mel / MFCC (40)
- Keep same as training

Audio Classification on M5Core2

### 5 Audio Capture & Feature Extraction



- i2sint1, 16 kHz
- Log-mel / MFCC
- i2d-mic (16 kHz)
- Log-mel / MFCC (40)
- Keep same as training

### 6 ML Inference with TFLite Micro

- Run inferences in loop:
- Feed MFCC features into
- Read output scores/label

#### Checklist:

- ✓ Train/save TFLite model
- ✓ Quantize it to INT8
- ✓ Upload & run inference on M5Core2

### 7 Deployment on Arduino M5Core2

- ✓ Deploy C array model
- ✓ Quantize it to INT8
- ✓ Convert it to C array
- ✓ Upload & run inference on M5Core2

