Edge AI Deployment: Optimizing and Scaling AI Models on Devices

Alexander D’hoore

Welcome to Day 3: Edge AI Deployment

- Welcome back to the workshop!

- Today is **Day 3**, and we'll focus on:

Deploying AI models on edge devices.

Course Overview

Course Goals

By the end of this course, you'll be able to:

- Understand the opportunities of **edge AI**.
- Select appropriate **edge AI hardware**.

- Optimize models through **ONNX** file format.
- Deploy using **ONNX Runtime** and **TensorRT**.
- Advanced optimization with **Model Optimizer**.

Table of Contents (1/2)

Table of Contents (2/2)

Introduction to Edge AI

What is Edge AI?

- Machine learning **inference on physical devices**
- Close to the source of data: sensors, machines, vehicles
- Operates outside traditional data centers

- **Limited compute, memory and storage**
- Bandwidth may be **unreliable or expensive**
- **Battery-powered** → energy efficiency

How Edge AI Works

- Models are **trained on cloud** or workstation
- Then optimized and **deployed on-device**

- **Inference** happens **locally**, not in the cloud
- Using sensors:
  - Cameras,
  - microphones,
  - accelerometers, etc.

On-Device Training: Rare but Possible

- Some edge devices allow **fine-tuning or adaptation**
  - E.g. personalize to local environment or user
- Limited by:
  - CPU/GPU constraints
  - Power and storage
- Inference remains the **dominant use case**

Where Edge AI Is Used

- **Industrial automation**: defect detection, predictive maintenance
- **Automotive systems**: real-time navigation, safety
- **Healthcare devices**: diagnostics, wearables
- **Remote infrastructure**: monitoring, control

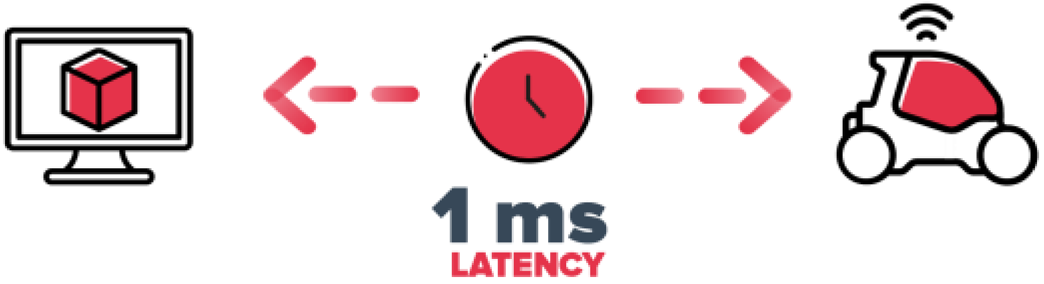
→ Common thread: **continuous local data**

+ Need for **fast, autonomous decisions**

Latency: Real-Time Reactions

- Some systems must respond in **milliseconds**
  - Example: robotics, drones, autonomous vehicles
- Cloud inference: 50–100ms (best case)
- Network delay is **unpredictable**

→ Edge AI offers **microsecond-level latency**, enables fast control loops



Bandwidth: Local Insights, Not Raw Streams

- Edge devices often generate **high-volume data**
  - Cameras, microphones, sensors
- Cloud upload is:
  - Too slow
  - Too costly
  - Sometimes impossible (e.g. rural, satellite)

→ Edge AI extracts and sends **only key results**

→ Optional: upload **rare events** for retraining

Reliability: When the Network Fails

- Many environments have **unstable connectivity**
  - Factories, vehicles, ships, rural areas
- Cloud-only inference = **single point of failur**e

→ Edge AI continues working **offline**

→ Some systems use **fallback**: edge + cloud

Privacy and Compliance

- Local inference keeps **data on the device**
- Reduces:
  - Risk of leaks
  - Need for cloud access control
  - Legal exposure (GDPR, HIPAA, etc.)
- Example:
  - Smart camera → detects activity, not record video

- Sometimes a **legal requirement**
- Can be a **competitive advantage**

Cost: Economics of Edge AI

- Cloud inference incurs **ongoing costs**:
  - Compute, storage, and network usage
  - Costs scale with usage (per request or per byte)

- Edge AI shifts to **fixed hardware cost**
  - One-time investment
  - Predictable long-term budgeting

- **Scalable intelligence without scalable bills**

Power Efficiency and Battery Constraints

- Cloud-based inference = constant transmission = **high energy**
- Edge inference = **no round trip**, less communication overhead

- Edge AI hardware (ARM, NPU):
  - Lower energy consumption
  - Better thermal behavior
- Combined with:
  - Low-power wireless (LoRa, NB-IoT, BLE)
  - Smart sampling or event-based communication

Strategic Deployment: Cloud vs Edge

- **Cloud inference**:
  - Centralized, scalable, fast iteration
  - Risk of vendor lock-in and rising costs
- **Edge inference**:
  - Local control, privacy, and reliability
  - Greater autonomy and offline capability
- **On-premise**:
  - Local server in same facility
  - Useful in hospitals, factories, defense

When Edge AI is **Not** the Right Fit

- Edge AI is powerful — but not universal
- Avoid for:
  - Large-scale language models
  - Collaborative inference
  - Frequent model updates

→ Choose edge only when **justified** by latency, privacy, bandwidth, or autonomy

Advantages of Edge AI

- ✅ Latency: Respond in real time
- ✅ Bandwidth: Reduce data transmission
- ✅ Resilience: Survive network failures
- ✅ Privacy: Keep data local
- ✅ Cost: Lower and more predictable
- ✅ Energy: Enable battery-powered ML
- ✅ Flexibility: Cloud, edge, or hybrid

→ Edge AI = smarter, faster inference **in the real world**

Overview of Edge Hardware and Accelerators

Introduction to Edge Hardware

- **What kind of hardware** will run your model?

- This chapter explores **edge AI hardware**:
  - From microcontrollers to GPUs and NPUs
  - Matching compute to application needs

- Essential for selecting or designing reliable, efficient edge systems.

Industrial Priorities in Hardware Choice

- **Industrial constraints ≠ Consumer constraints**

- Key priorities:
  - Long-term availability (10+ years)
  - Pre-certified modules for CE, FCC, UL…
  - Durability: temperature, vibration, dust…
  - Sourcing risk and supply chain stability

- Even perfect boards can fail,
  if they disappear in 3 years

Certification and Lifecycle Management

- **Regulatory compliance is essential**
  - CE, FCC, UL often legally required
  - Off-the-shelf modules may come **pre-certified**

- **Lifecycle** expectations:
  - Deployment in 2027 → still repairable in 2037
  - Hardware updates = expensive redesigns

Environmental Durability and Sourcing Risk

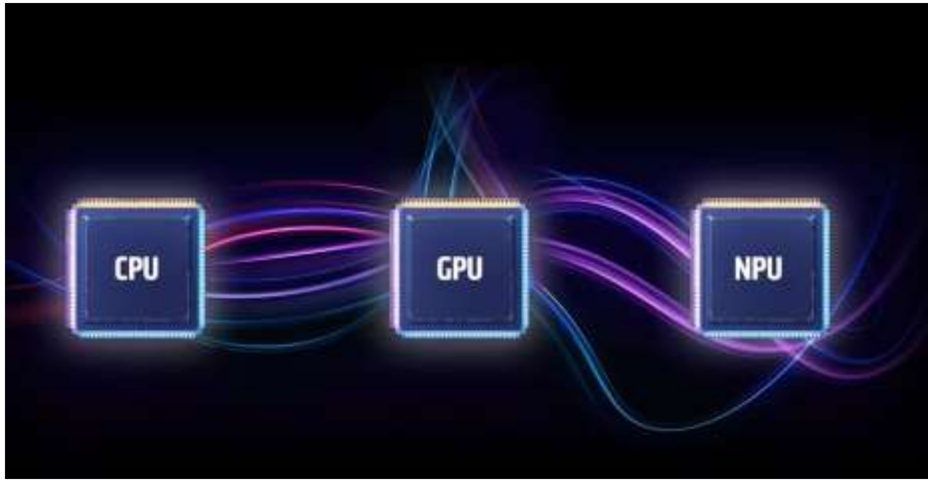
- Real deployments ≠ office conditions
  - Outdoors, in vehicles, factories, etc.
- Industrial environments need:
  - Wide temp range, rugged connectors, sealed enclosures
- **Sourcing matters**:
  - Can you buy 10,000 units next year?
  - Will the part still exist?

Tooling, SDKs, and Ecosystem Maturity

- Good hardware needs **great software**
- Toolchains + docs often **make** or **break** a project
- Mature ecosystems = smooth development
  - Good options: NVIDIA Jetson, Intel OpenVINO
- Cheap boards with bad SDKs = **hidden costs**

CPU, GPU, NPU: What's What?

- **CPU**: Flexible, handles OS, control logic
  - Good for setup, orchestration, signal processing
- **GPU**: Highly **parallel** compute device
  - Great for matrix operations, AI inference
- **NPU**: Dedicated **neural network** accelerator
  - Fast, efficient, application-specific (int8/fp16)



Performance Specs Can Mislead

- A board might advertise **“10 TOPS”**
  → That means **10 trillion operations per second**

- But raw numbers don’t tell the full story:
  - Are all your model’s layers **actually supported**?
  - Is the **memory bandwidth** fast enough to keep up?

- Real-world performance depends on:
  - Operator compatibility, toolchain quality
  - How well your model maps to the hardware

Matching Workload to Hardware

- Choose hardware based on **model type and input data**:

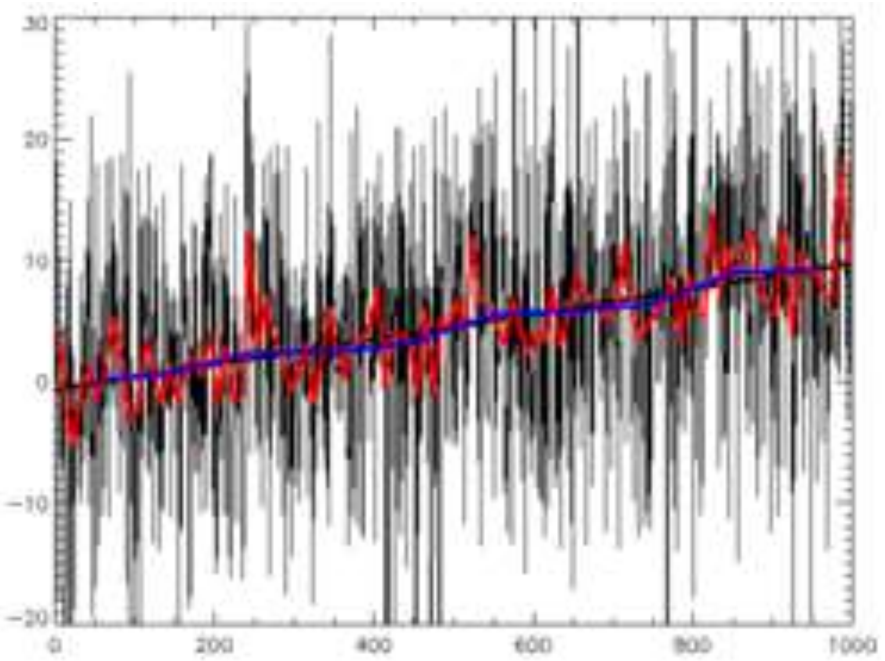
- Time series → low compute
- Audio → medium compute
- Vision CNN → high compute
- Video / Transformers → very high compute

Time Series and TinyML

- **Low sampling rate**, low data volume
- Models: classifiers, anomaly detectors

- Typical hardware:
  - MCUs with KBs of RAM
  - TinyML toolchains

- Use cases:
  - Gesture recognition, vibration analysis



Audio: More Demanding

- **Higher sampling rates** (16–48 kHz)

- Models:
  - Keyword spotting, speaker ID
  - Environmental audio classification

- Typical hardware:
  - Optimized MCUs or entry-level NPUs
  - Full speech recognition
    → Linux-class CPU + NPU

Image/Video: Matrix-Heavy

- **Image classification, detection, segmentation**
  - Object tracking, pose estimation
  - Smart cameras, industrial vision

- Inputs: 224×224 RGB
  → 50k+ pixels

- Requires:
  - Dedicated NPUs
  - Embedded GPUs

Transformers on Edge?

- **High resource requirements**:
  - RAM, memory access, attention support
- Models:
  - Whisper, ViT, LLaMA
- Most platforms can't run them efficiently
  - Some support emerging (e.g. Qualcomm, Hailo)
  - Often needs **desktop GPU**

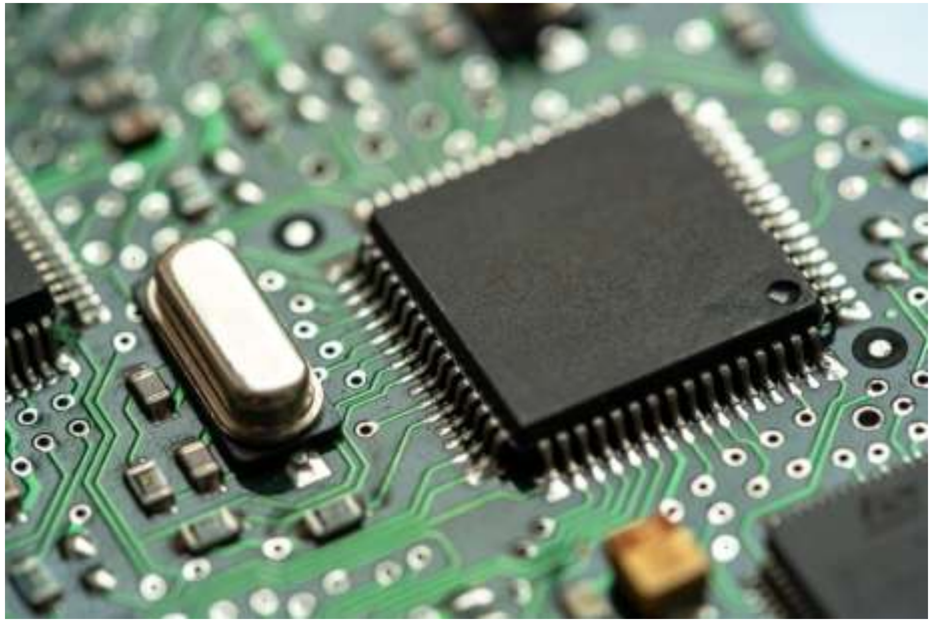
Let’s Look at Some Hardware

Microcontrollers for TinyML

- Microcontrollers (MCUs) are the **smallest and most common** compute platforms in the world.

- Can be used for:
  - Industrial monitoring
  - Vibration analysis
  - Anomaly detection

- Powering the rise of **TinyML**:
  ML models on KB–MB memory devices



hogeschool vives KU LEUVEN

MCU Example: STM32 Series

- Widely used in industry
  - **Cortex-M0** (simple)
  - **STM32H7** (high-end with FP support)
  - **STM32N6**: NPU for real-time vision

→ STM32N6 runs **YOLOv8 at 30 FPS** within a battery-friendly package

→ Previously impossible on microcontrollers

Embedded Linux SoCs (With or Without NPU)

- When microcontrollers aren't enough:
  - Input is richer
  - Models are bigger
  - You need **Linux**

- Embedded Linux **system-on-chips** offer:
  - MPUs with 100s of MBs to GBs of RAM
  - Advanced I/O
  - Real-time AI on-device

What Linux SoCs Enable

- Enable you to run full **Linux** operating system

- Support complex workloads such as:
  - **Robotics** with real-time control loops
  - **Computer vision** for detection or tracking
  - **Audio processing** like speech commands
  - **Human–machine interfaces** (displays)

→ Think: smart cameras, inspection robots, AI-powered control units

Linux Chips with NPUs

- Many modern SoCs come with **dedicated NPUs**
- Offload inference from CPU

- Key examples:
  - **NXP i.MX 8M Plus**: 2.3 TOPS, great docs
  - **TI TDA4VM**: 8 TOPS, real-time AI
  - **Rockchip RK3588**: octa-core CPU + 6 TOPS
  - **STM32MP25**: 1.35 TOPS vision chip

Single-Board Computers and Modules

- Not every product starts from scratch

- Use **single-board computers (SBCs)**
- Or **compute modules (SOMs)**

- Pre-integrated hardware
- Speeds up development and prototyping

SBCs: Fast, Flexible Development

- **Full computers** on one PCB
- Run Linux out of the box
- Include HDMI, USB, Ethernet…

- Examples:
  - **Radxa ROCK 5** (Rockchip + 8K + AI)
  - **Toradex, SolidRun, Advantech** (industrial)
  - **Raspberry Pi 5** (less AI, more community)

Modules: For Embedded Products

- **Compute modules** plug into **carrier boards**
  - System-on-module (SOM), computer-on-module (COM) …

- Split high-frequency logic (module)
  from application-specific I/O (carrier)

- **Custom I/O** on the carrier board:
  - Motor control
  - Camera input
  - Industrial buses

Add-On Module Examples

- **Toradex Aquila**: NXP/TI SoCs, long-term support
- **SolidRun SoMs**: modular i.MX8 boards
- **Raspberry Pi Compute Module 5**

→ Ideal when you need
integration + certification + lifecycle

The NVIDIA Jetson Series

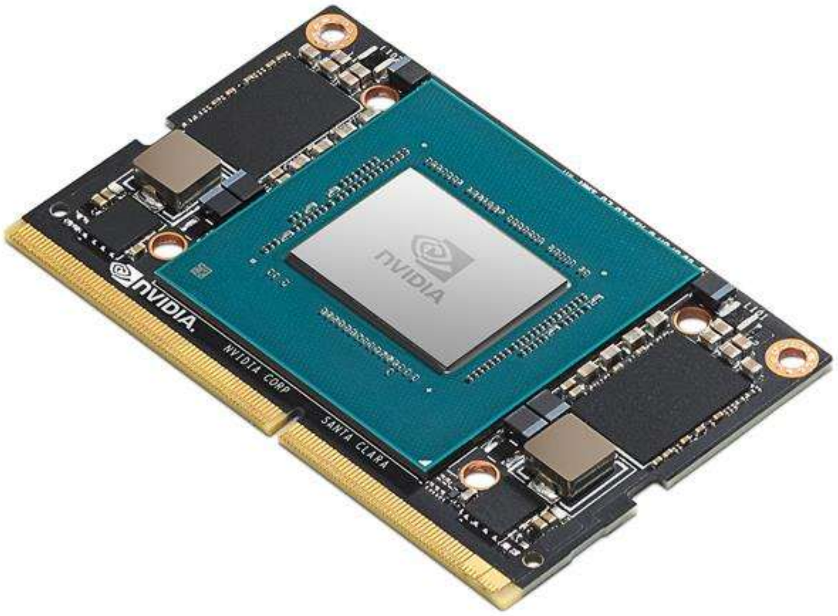
- Jetson = **Embedded AI Powerhouse**
  - **Multi-core ARM CPUs** for general-purpose compute
  - **Embedded NVIDIA GPUs** for high-throughput parallel processing
  - **Dedicated AI accelerators (DLA)** for efficient deep learning inference

- Built for demanding edge use cases:
  - Autonomous robotics
  - Multi-camera video analytics
  - Real-time vision, planning, decision-making

Jetson Form Factor and Integration

- Jetson = **compute module,** sits on **carrier board**

- Dev kits = reference carrier + module
- Carrier boards:
  - Use 3rd-party carriers (Connect Tech, Forecr)
  - Build your own custom carrier



Jetson Orin Lineup

- **Jetson Orin Nano**:
  - Up to **67 TOPS,** Power: 7–25W
- **Jetson Orin NX**:
  - Up to **157 TOPS,** Power: 10–40W
- **Jetson AGX Orin**:
  - Up to **275 TOPS,** Power: 15–60W

→ Choose based on model size and real-time needs

Jetson = Hardware + Software

- Every Jetson runs **JetPack**:
  - CUDA
  - cuDNN
  - TensorRT
  - DeepStream

- ✅ **Datacenter** tools
- ✅ Optimized for **embedded**
- ✅ Full **PyTorch** support

External AI Accelerators (Hailo)

- Not every system has a good GPU/NPU

- Solution: **External AI accelerators**
  - PCIe or M.2 form factor
  - Plug into existing systems
  - Add neural inference without a redesign

→ A leading example: **Hailo modules**

Hailo Overview

- **Hailo-8**:
  - Up to **26 TOPS**
  - Power: ~2.5W
  - PCIe / M.2
- **Hailo-8L**:
  - Up to **13 TOPS**
  - Lower power
- **Hailo-15**:
  - Combines ARM + NPU
  - Smart vision chip

VLAIO TETRA MLOps4ECM

Hailo in the Edge AI Ecosystem

- ✅ Compact, passively cooled NPU
- ✅ Strong **industrial adoption** (Kontron, Advantech, Toradex…)
- ✅ Raspberry Pi AI Kit includes Hailo-8L
- ✅ Excellent SDK: supports ONNX
- ✅ Powerful **Hailo Dataflow Compiler**

→ Compact, powerful, and easy to integrate

FPGAs for Deterministic AI

- Most accelerators = fixed-function (CPU, GPU, NPU)
- **FPGAs = reconfigurable logic**
  - Build **custom hardware** per application
- Ideal for:
  - Real-time signal processing
  - Deterministic vision pipelines

→ Great for edge AI with
hard real-time requirements

Popular FPGA Platforms

- **AMD (Xilinx)** and **Intel (Altera)** dominate

- Examples:
  - **AMD Kria K26**:
    - Zynq UltraScale+ MPSoC
    - ARM + FPGA fabric
    - Target: embedded vision
  - **Intel Agilex 5**:
    - ARM + DSP + AI blocks
    - Mid-range automation & robotics

Intel-Compatible Edge Devices

- Not all edge AI runs on ARM
  → Example: **Intel N100** (6W TDP, solid performance)

- ✅ Full Linux/Windows
- ✅ Mainstream toolchains
- ✅ Off-the-shelf I/O and enclosures
- ✅ Wide industrial vendor support
- ❌ Higher power usage than ARM

Intel OpenVINO for AI Inference

- Intel's **OpenVINO toolkit**:
  - Optimizes models for Intel CPUs, GPUs
  - Supports: ONNX, PyTorch, TensorFlow

- **Excellent** library for:
  - Visual inspection
  - Predictive maintenance
  - Anomaly detection

→ Avoids need for external
AI chips in many workloads

Industrial PCs (IPCs)

- When you need more power:
  - Go from embedded boards → **Industrial PCs**

- IPCs = PC hardware in **rugged form**
  - Dust-proof, fanless, vibration-resistant
  - DIN rail mountable
  - Real-time fieldbus support

Industrial PC Capabilities

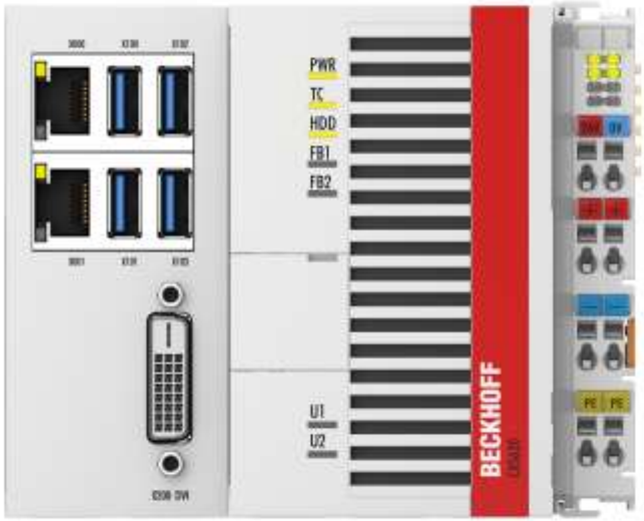
- CPUs: Intel/AMD (high single/multi-core perf)
- OS: Linux or Windows

- AI Acceleration: **high-end GPUs**
  - NVIDIA RTX cards (for vision or transformers)

→ In factories, inspection, machine control

PLCs with AI Capabilities

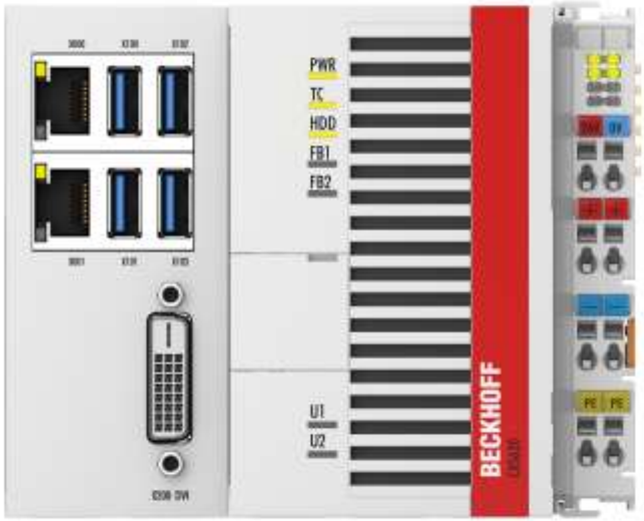
- PLCs = heart of **real-time automation**
  - Deterministic timing, safety interlocks
  - Often programmed in **Structured Text**
- New hybrid architectures:
  - RTOS + Linux/Windows on same device
  - Combines control + inference on one box
- Vendors: Siemens, Beckhoff, Schneider…



AI on the PLC?

1. **On-PLC Inference**
   - Fast, deterministic, model size limited
2. **Nearby Module/IPC**
   - More powerful, adds latency
3. **Cloud or Central Server**
   - Most complex models, least real-time

→ Choose based on control loop timing vs model complexity



Edge Hardware Landscape

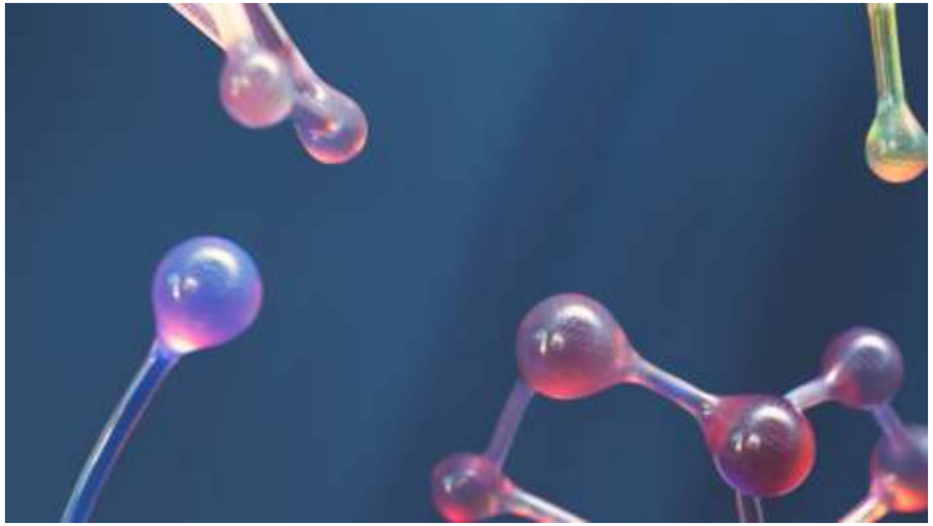
- **MCUs**: ultra-low power, TinyML
- **SoCs**: balance of Linux and AI
- **SBCs/Modules**: fast time to market
- **Jetson**: high-performance embedded AI
- **Hailo**: external acceleration, flexible
- **FPGAs**: custom logic, deterministic control
- **Intel x86**: mature, flexible platforms
- **IPCs**: rugged, GPU-ready edge systems
- **PLCs**: real-time control with AI support

Model Optimization Techniques

Why Optimize Models for the Edge?

- Most models are trained on **server hardware**
  - Large GPUs, plenty of RAM, fast I/O

- But edge devices face real limits:
  - Small memory footprint
  - Slow CPU/GPU
  - Hard latency deadlines



Reality Check: Inference-Time Memory

- Model weights are usually stored in **FP32**
  - 4 bytes per parameter
  - 1M parameters = **4 MB** weight file
- But inference also needs:
  - **Activation maps** (layer outputs)
  - **Temporary buffers**, which are much larger
- Vision models often use **5–10× more RAM** at inference time

ResNet-50: A Classic Heavyweight

- ~25 million parameters → ~**100 MB** (FP32)
- Inference-time memory: **hundreds of MB**
- Fine on desktop GPUs

- On **Jetson Orin Nano (4 GB RAM)**:
  - GPU memory on Jetson is **shared** with operating system
  - Reaching limits when combined with camera input or other tasks

YOLOv5s: Light, but Still Demanding

- ~7.5 million parameters (~**30 MB** in FP32)

- Real-time inference on:
  - Jetson Orin: ✅ OK
  - Raspberry Pi 4: ⚠️ **CPU-only > 500 ms/frame**

- Needs acceleration for real-time performance

MobileNetV2: Designed for Efficiency

- ~3.5 million parameters → ~**14 MB**

- On **Radxa Rock 3**:
  - Automotive-grade single-board computer
  - INT8 quantized model runs in real time (60Hz)
  - Inference time cut by **>50%** with quantization

→ **Smaller model, smarter design = edge performance**

Who’s Doing the Optimization?

1. Training Frameworks

- Where models are **built and trained**
- Usually used in the cloud or on dev machines
- **PyTorch**:
  - Most widely used framework today
  - Pythonic, flexible, **dominant** in research and prototyping
- **TensorFlow / Keras**:
  - Still widely adopted, especially in production ecially mobile
  - Less popular in current research circles

→ These are not deployment-optimized, but they're the **starting point**

2. General-Purpose Runtimes (1/2)

- Cross-platform engines, not hardware specific

- **ONNX Runtime**:
  - **Cross-platform**, high-performance
  - Supports CPU, GPU, and NPU backends
  - Automatically applies quantization, operator fusion, graph rewrites
- **TensorFlow Lite (TFLite)**:
  - Lightweight, **mature** inference engine from TensorFlow
  - Targets mobile, embedded Linux, and **microcontrollers** (Cortex-M)
  - Includes quantization tooling and runtime acceleration

2. General-Purpose Runtimes (2/2)

- **TorchScript**:
  - PyTorch's native model export format
  - Used for deployment in mobile apps and C++ environments
  - Easier to use than ONNX for simple deployment within **PyTorch ecosystem**
- **ExecuTorch**:
  - New runtime from PyTorch team for edge devices
  - Supports Android, iOS, and some microcontrollers
  - Leverages PyTorch 2 export tools
  - Promising, but **not yet mature** or widely used

3. Vendor-Specific SDKs and Toolchains (1/2)

- Provide **maximum performance** for specific hardware
- **TensorRT (NVIDIA)**:
  - Converts ONNX models into GPU-optimized engines
  - Supports FP16, INT8, layer fusion, memory tuning
- **OpenVINO (Intel)**:
  - Targets CPUs, iGPUs, and FPGAs
  - Includes quantization, pruning-aware tools, strong runtime
- **TI EdgeAI SDK**:
  - For TI SoCs (e.g. TDA series)
  - Hardware-aware quantization and model compilation

3. Vendor-Specific SDKs and Toolchains (2/2)

- **STM32Cube.AI**:
  - Compiles models to **efficient C code** for STM32 MCUs
  - Applies static memory allocation and INT8 quantization
- **Hailo SDK**:
  - For Hailo-8 external NPUs, supports ONNX models
- **RKNPU Toolchain**:
  - Targets Rockchip SoCs with NPUs, optimized execution graphs

- and many more…

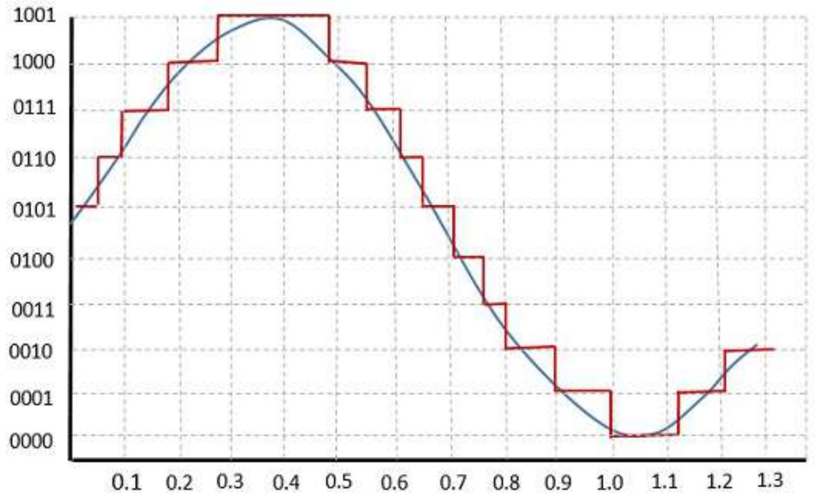
What Kinds of Optimizations Are There?

Numeric Precision and Data Types

- Most models use **32-bit floating point (FP32)**

- Edge devices benefit from **smaller formats**:
  - **FP16**: Half-size, faster on GPUs
  - **INT8**: 1 byte per value, major speed gains
  - **BF16** (Brain Float 16): TPUs, Intel CPUs
  - **INT4**: Extreme compression

→ **FP16 and INT8** are the workhorses of edge

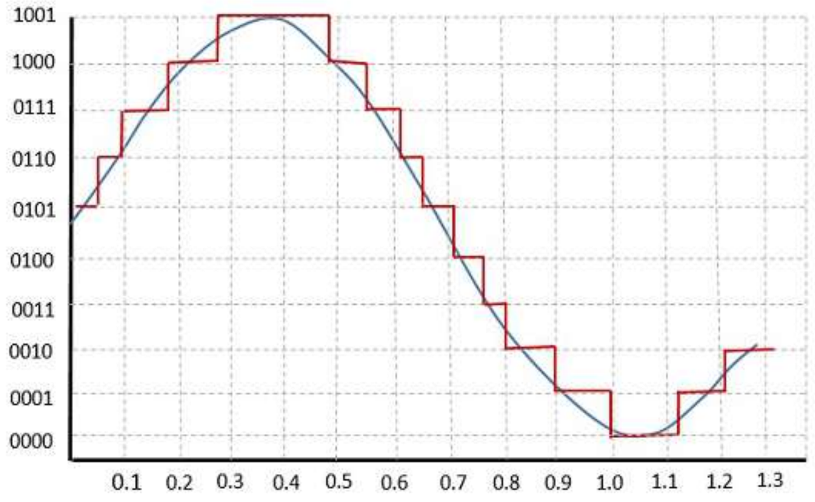


What is Quantization?

- Converts weights/activations to **lower-precision integers**
  - Saves memory (4× smaller models)
  - Speeds up inference (2 to 4× faster)
  - Enables deployment to constrained devices

- Most common: **INT8 quantization**
  - Native support on NPUs, CPUs, and MCUs

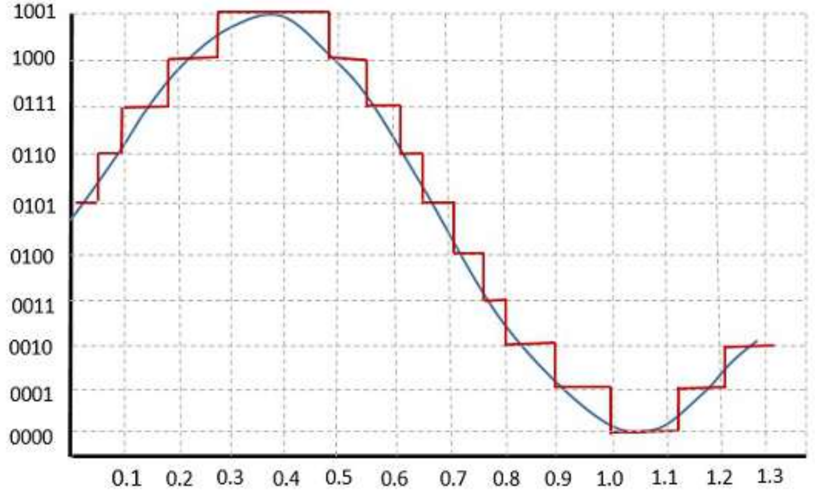
→ **Quantization is key** to efficient edge AI



Post-Training Quantization (PTQ)

- Apply quantization **after training** — no retraining needed
- Converts FP32 → INT8
  - Uses **scaling factors** to map float to int
  - Keeps model structure the same
- Accuracy drops possible, but often small

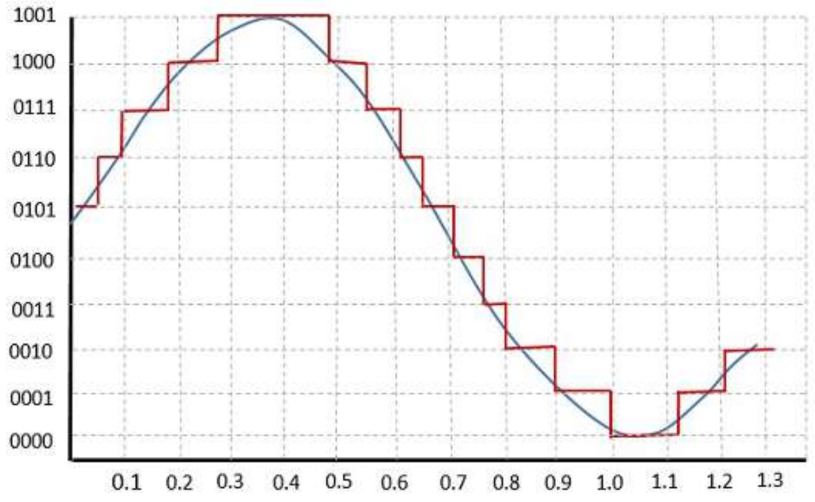
→ Fast and easy way to reduce model size



PTQ: Dynamic vs Static Quantization

- **Dynamic Quantization**:
  - Quantize weights only
  - Activations quantized on-the-fly
  - Works well for LSTMs, transformers
- **Static Quantization**:
  - Quantize both weights and activations
  - Requires **calibration dataset**
  - Better accuracy for vision models

→ Tradeoff: simplicity vs performance



VLAIO TETRA MLOps4ECM

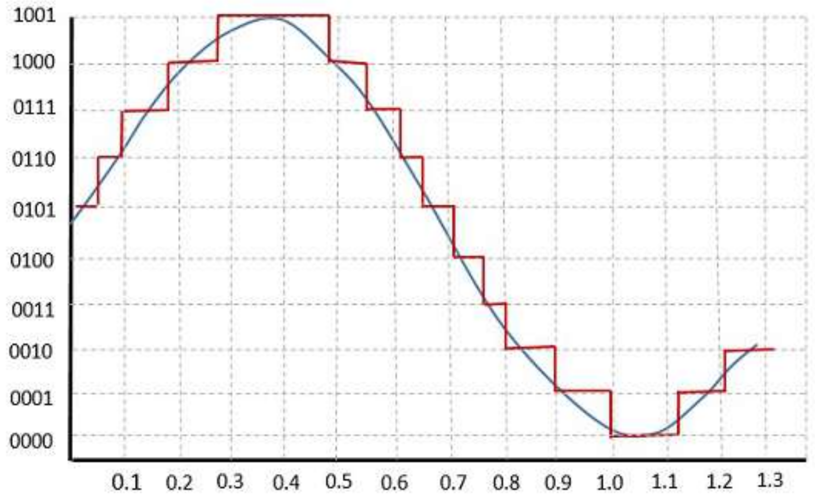
Quantization Formula & Strategies

$$\text{float\_value} = \text{scale} \times (\text{int\_value} - \text{zero\_point})$$

- **Scale** = adjusts float to int range
- **Zero-point** = optional offset (for asymmetric quantization)
- **Symmetric quantization** = zero-point = 0 (simpler, faster)
- **Per-tensor quantization**: 1 scale/zero-point per layer
- **Per-channel quantization**: 1 scale per output channel
  - More accurate, especially for CONV layers

PTQ Support in Frameworks

- **PyTorch**:
  - torch.quantization supports dynamic & static PTQ
- **ONNX Runtime**:
  - Includes quantization toolkit
- **TensorRT**:
  - Uses calibration steps to compute ranges
- **OpenVINO**:
  - NNCF toolkit with strong calibration support



→ All major runtimes support PTQ workflows

Quantization-Aware Training (QAT)

- Simulates quantization **during training**
  - Inserts fake quantization operators
  - Model learns to handle rounding noise

- Gradients still use FP32
- Final model runs in INT8 with **better accuracy**

→ Ideal for sensitive models or tight accuracy budgets

QAT in Practice

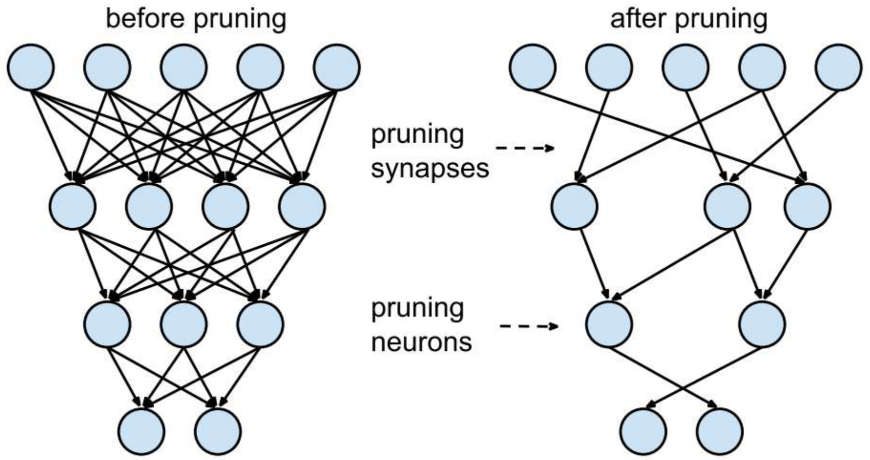
- **PyTorch**: Use prepare_qat() → train → convert()
- **ONNX Runtime**: Can run QAT-trained models (from PyTorch)
- **TensorRT**: Supported with NVIDIA Model Optimizer library
- **OpenVINO**: Integrated with NNCF: simulate quantization + export

→ Adds complexity, but enables **high-accuracy INT8 inference**

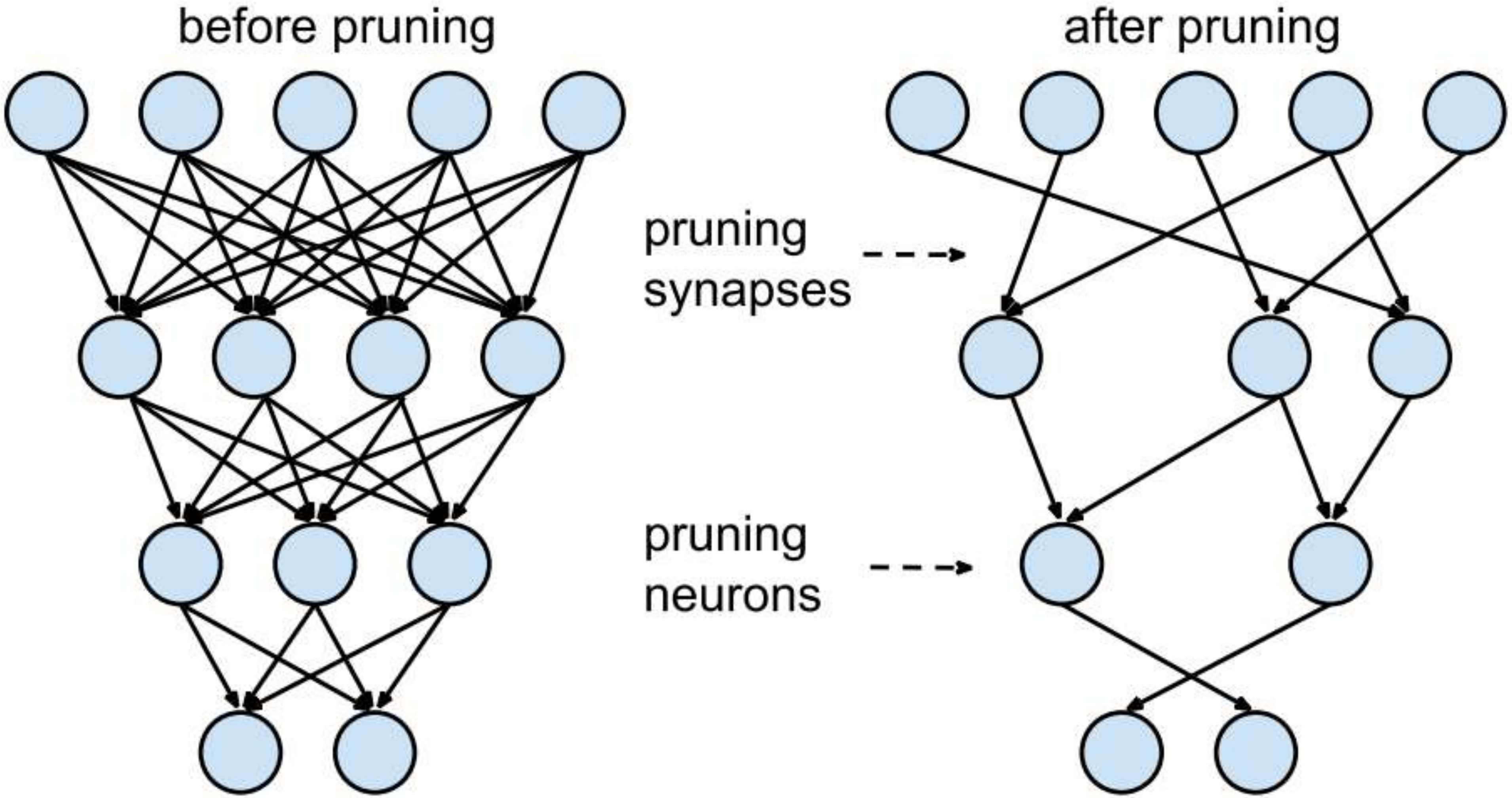
→ Use QAT when PTQ **accuracy drop is unacceptable**

Pruning: Slimming Down the Network

- Pruning = removing parts of a model
- Many weights/activations are **not important**
- Goal: **smaller, faster models** with minimal accuracy loss



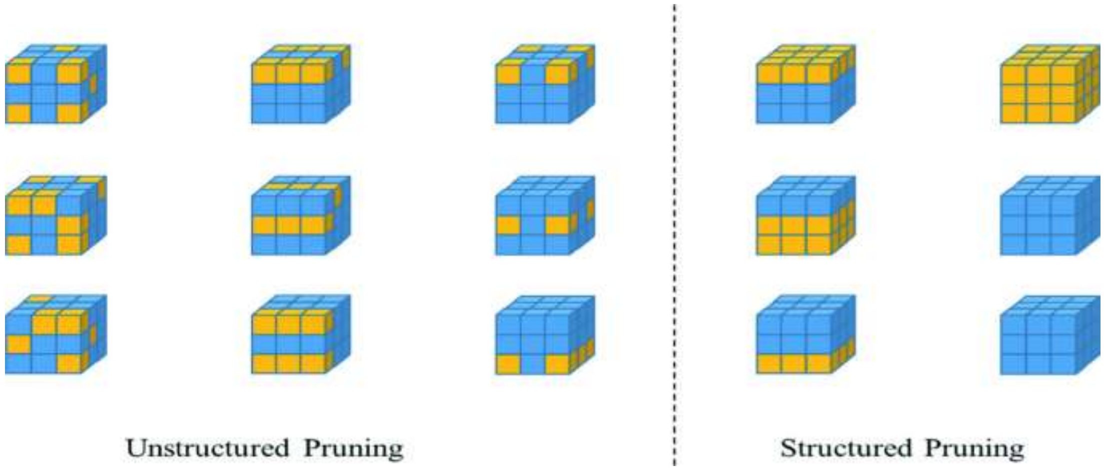
VLAIO TETRA MLOps4ECM

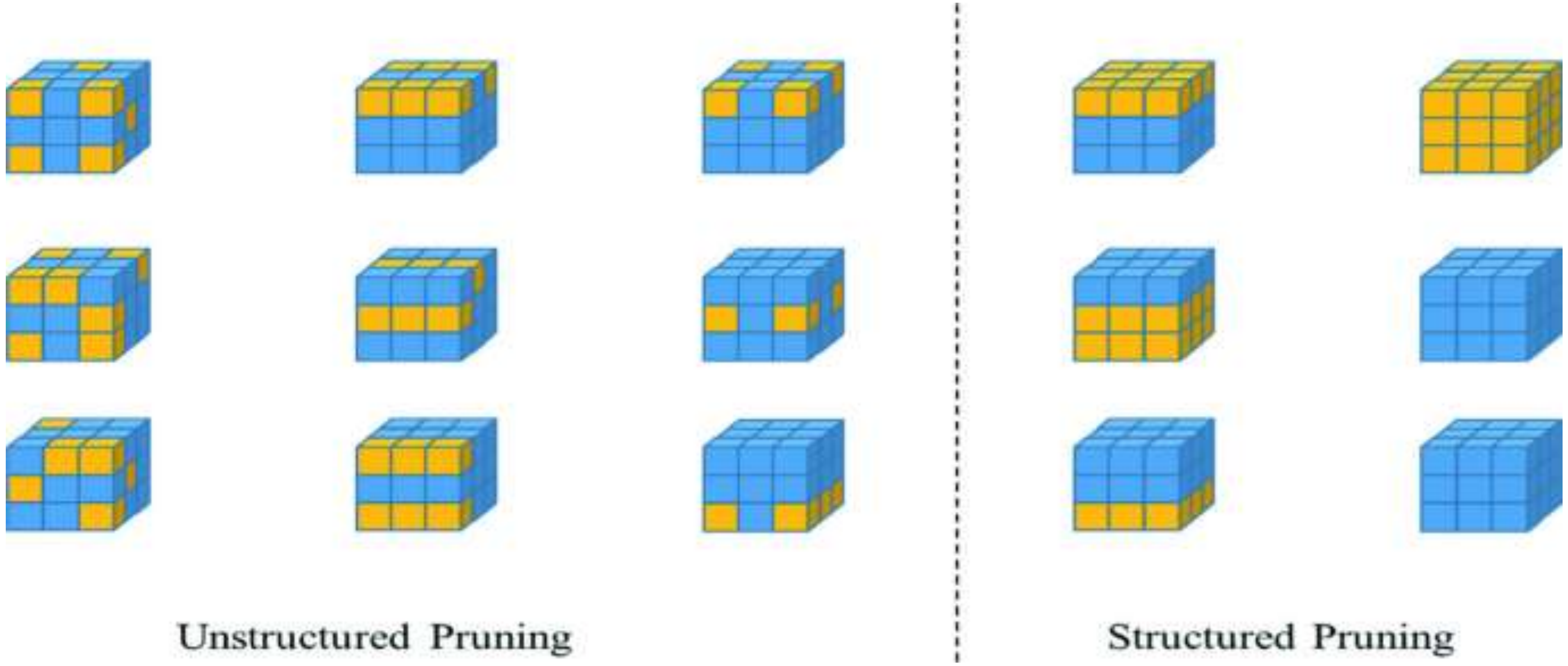

before pruning
after pruning
pruning
synapses
pruning
neurons

Unstructured vs Structured Pruning

- **Unstructured**:
  - Removes weights near zero
  - Model shape unchanged
  - ⚠️ **No speedup** on most hardware

- **Structured**:
  - Removes filters, channels, or blocks
  - Model becomes **smaller and faster**
  - Greater risk of accuracy drop

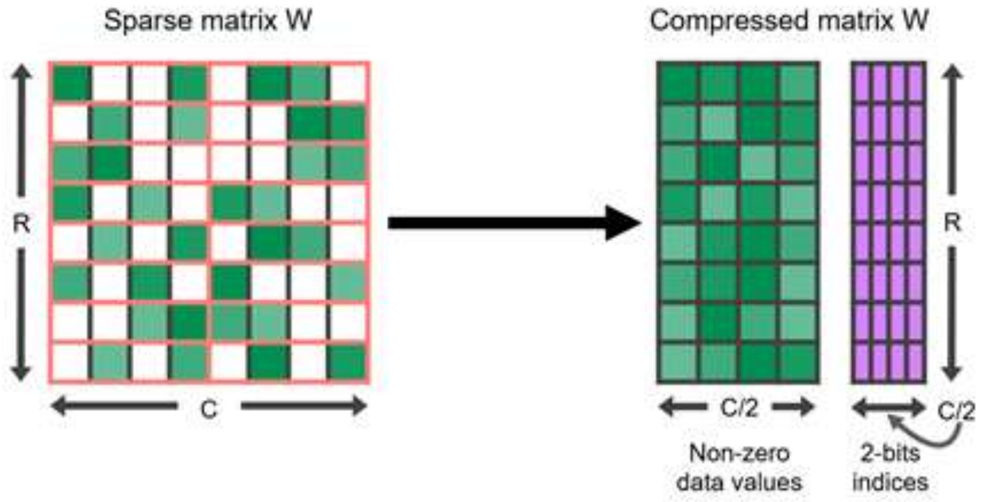


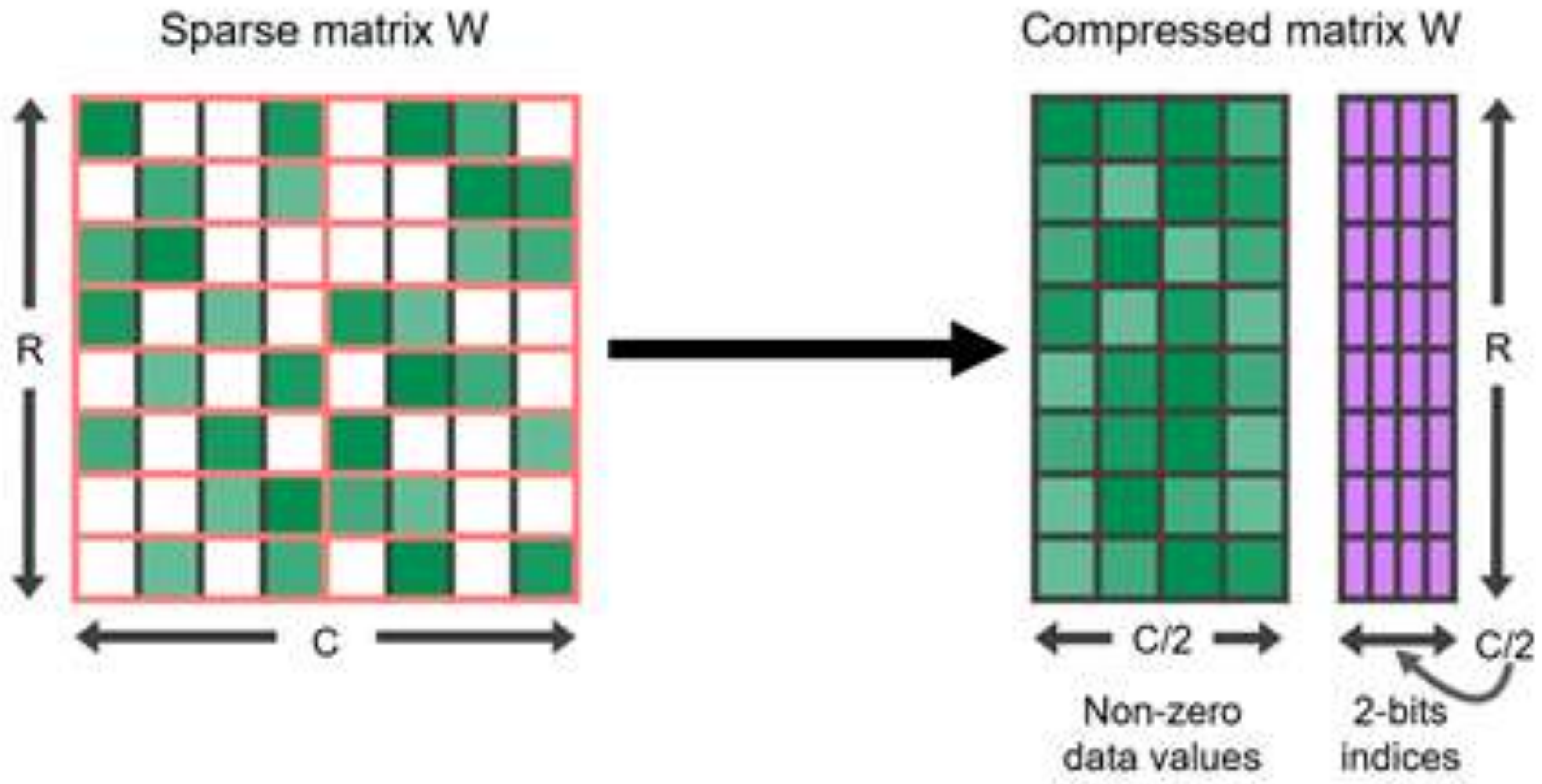

Unstructured Pruning
Structured Pruning

Special Case: 2:4 Sparsity Pruning

- Format where **2 of every 4 weights = 0**
- Supported by modern NVIDIA GPUs, some other NPUs
- Benefits:
  - Compression + real runtime acceleration
  - More structured than unstructured pruning

→ Requires **hardware + compiler support**




Sparse matrix W
R
C
Compressed matrix W
R
C/2
C/2
Non-zero data values
2-bits indices

Pruning in Practice

- **PyTorch**: torch.nn.utils.prune for structured/unstructured pruning
- **TensorRT**: Supports pruning and 2:4 sparsity with Model Optimizer
- **OpenVINO**: Pruning-aware training via NNCF

→ Toolchain support **varies** by target hardware

Graph-Level Optimizations

- Deployment tools **transform the model graph** for performance
  - Graph = operations + data dependencies
- Optimizations reduce memory use, kernel overhead, and latency

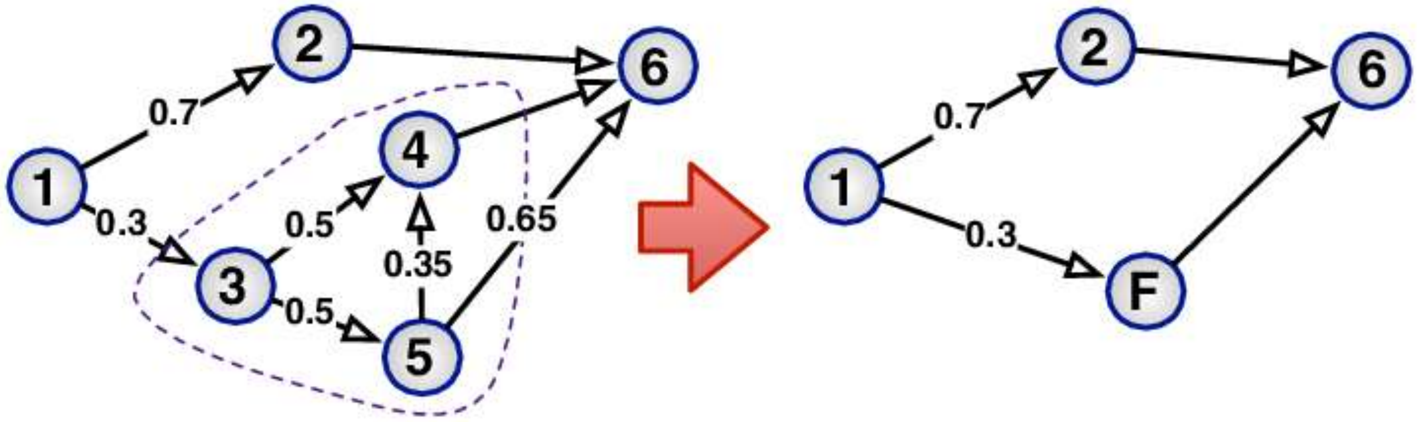
→ All done **automatically** at export or runtime

- You don't have to apply these manually

Operator Fusion

- Combines multiple operations into one:
  - Example: **Conv + BatchNorm + ReLU**

- Benefits:
  - Fewer memory accesses
  - Less kernel launch overhead



More Graph Optimizations

- **Constant folding**: pre-compute static operators
- **Dead code elimination**: remove unused branches
- **Transpose folding**: reduce layout changes (e.g. NHWC ↔ NCHW)
- **Memory reuse**: reuse intermediate buffers to save RAM
- **Layout optimization**: align memory layout with hardware
- **Static scheduling**: pre-plan execution order + buffers

→ Often invisible but powerful

Which Tools Apply These?

- **ONNX Runtime**:
  - Graph optimizations enabled by default
- **TensorRT**:
  - Aggressive fusion + memory planning
  - Uses **benchmarking** to find optimal graph
- **OpenVINO**:
  - Tailored fusion passes for Intel CPUs/GPUs

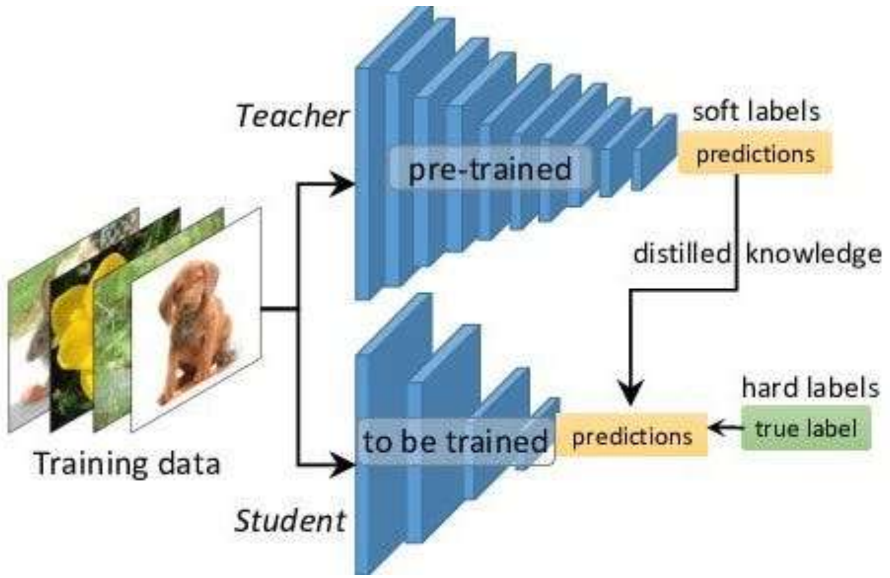
→ Export quality + runtime = **huge difference** in performance

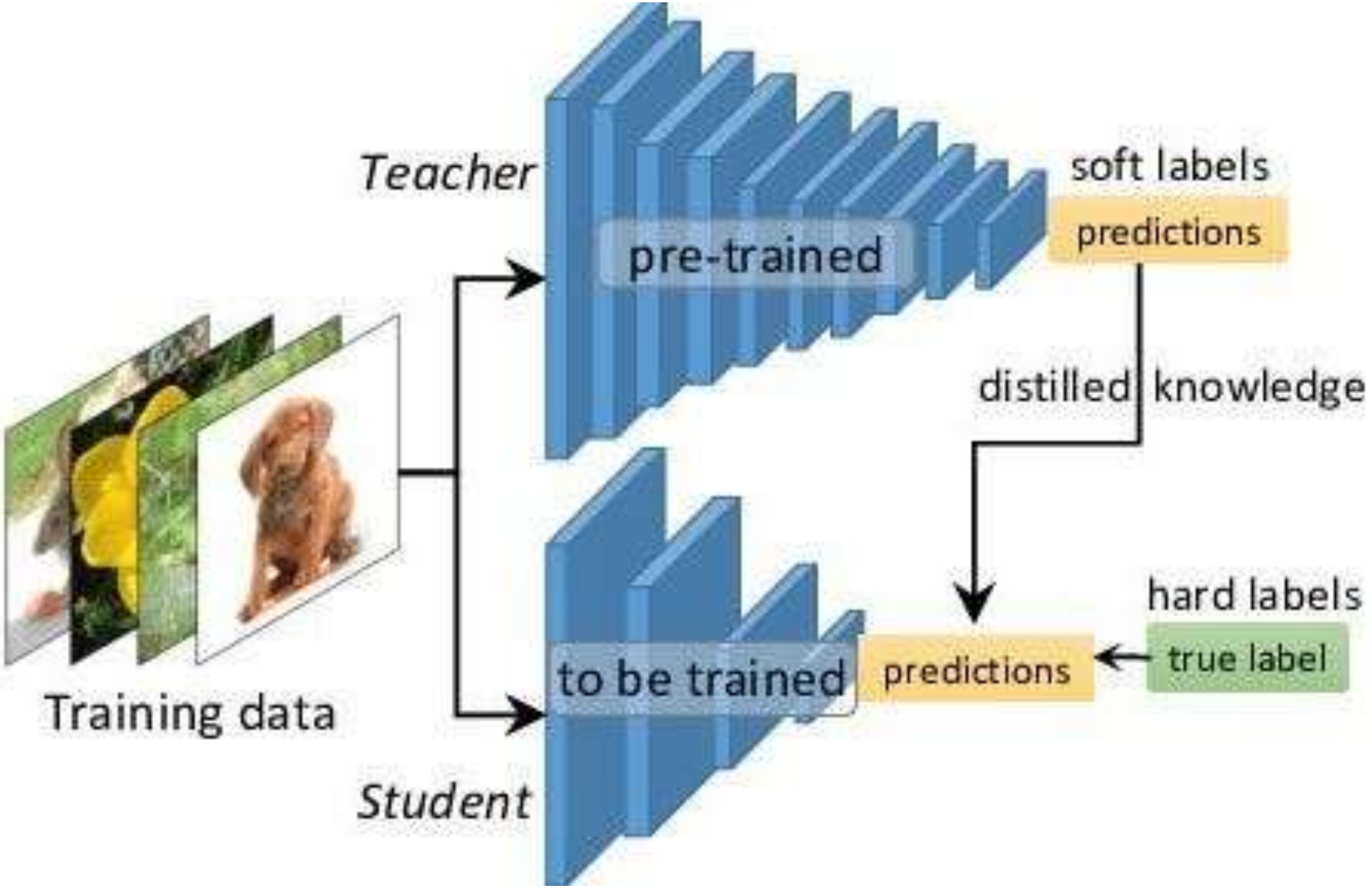
Knowledge Distillation: Core Idea

- Teach a **small model (student)** using predictions from a **larger model (teacher)**

- Student learns **correct labels**, and also:
  - Teacher's **confidence distribution**
  - Output behavior on ambiguous inputs

→ Improves student accuracy



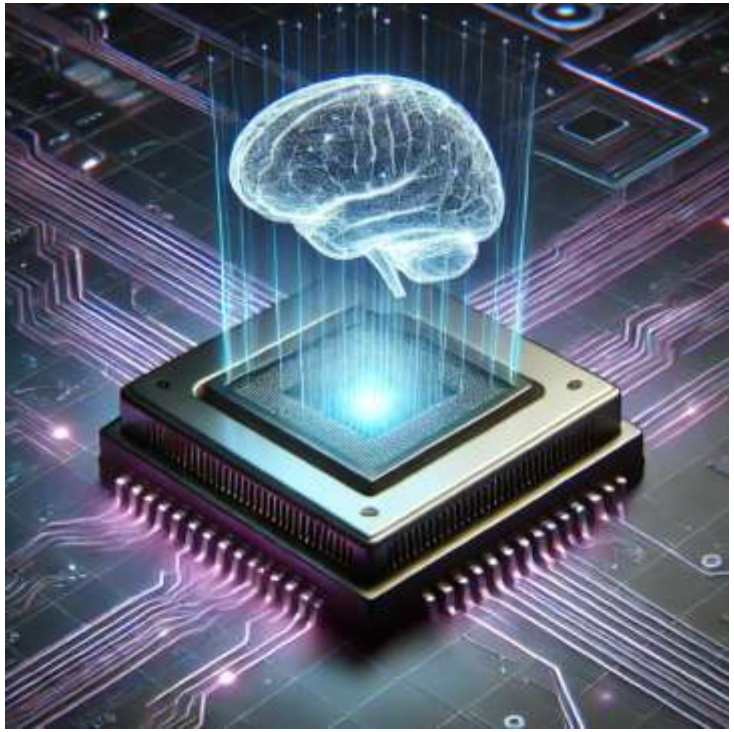

Teacher
pre-trained
soft labels
predictions
distilled knowledge
Training data
to be trained
predictions
hard labels
true label
Student

Synthetic Data: What and Why

- Synthetic data = **artificially generated training data**
  - Not an optimization technique, strictly speaking

- Data comes from:
  - Simulators (Unity, Isaac Sim, CARLA)
  - 3D renderers (Blender)
  - Generative models (Stable Diffusion)
  - Large language models (GPT, Llama)

→ Can **recover accuracy lost** to optimization

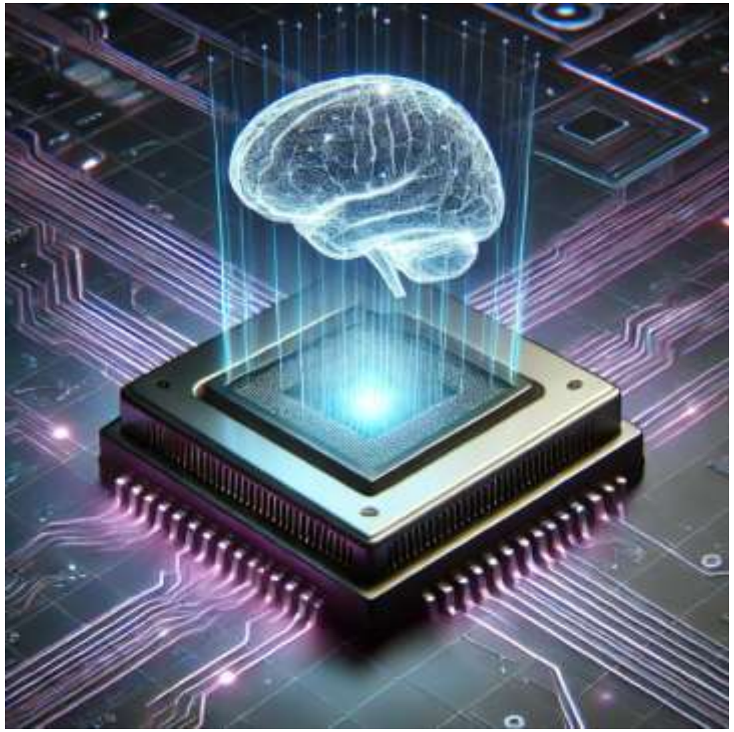


Synthetic Data + Edge AI = Future

- Use **foundation models** to:
  - Generate labeled samples (“red pipe on gray background”)
  - Auto-label unlabeled images or logs

- Small, optimized networks can:
  - Be trained on synthetic datasets
  - Run faster inference in the field

→ Synthetic + distilled + optimized
= **the future of edge AI**

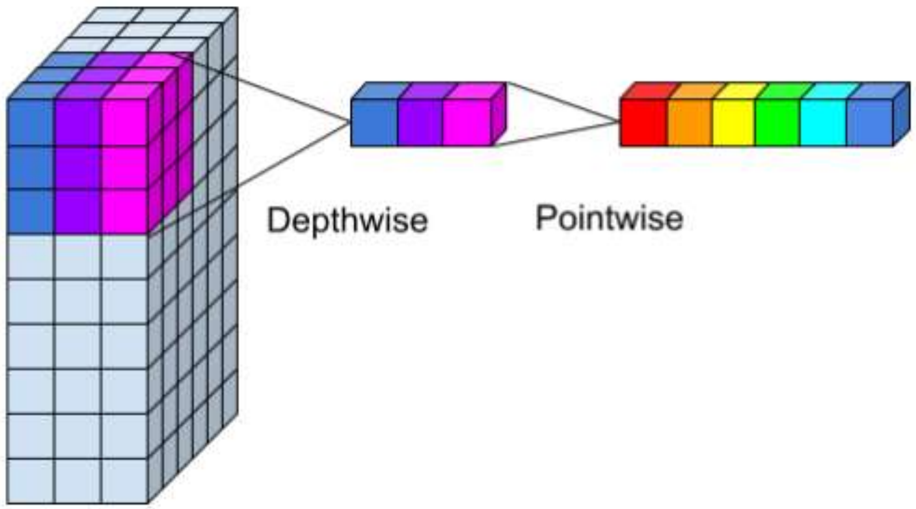


Efficient Model Architectures

Lightweight Vision Models

- Lightweight models are **designed for efficiency**
  - Prioritize: smaller size, fewer FLOPs, faster inference

- ✅ Depthwise-separable convolutions
- ✅ Inverted residuals + bottlenecks
- ✅ Squeeze-and-excitation blocks



Popular Lightweight Vision Models

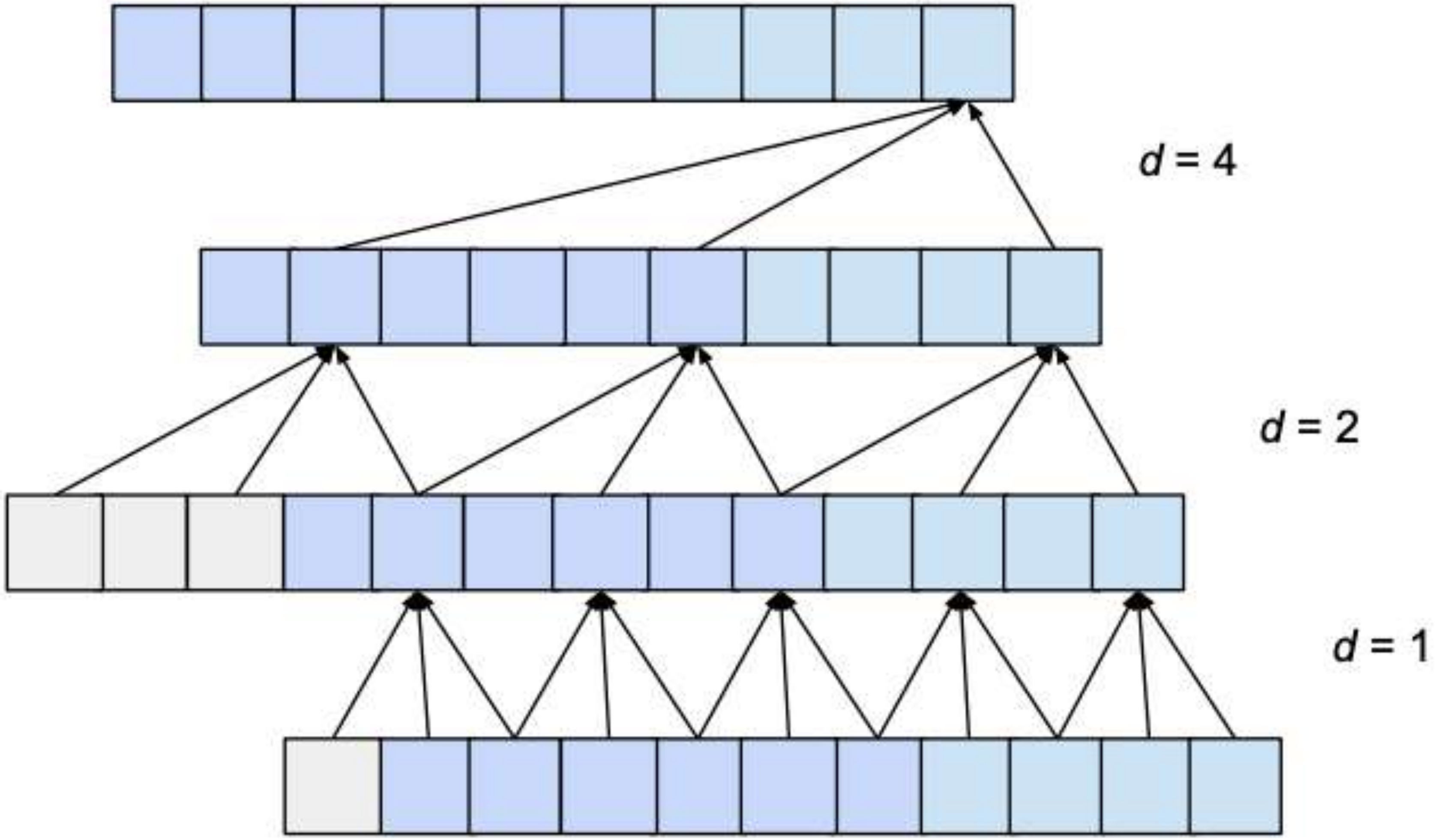
- **MobileNetV2/V3**: 2.5M–3.4M params, fast and widely supported
- **EfficientNet-Lite**: Great accuracy/size ratio, ONNX-ready
- **YOLO-Tiny**: Real-time object detection on CPUs and GPUs
- **MobileNet-SSD**: Compact detector, OpenVINO ready
- **DeepLabv3+ (MobileNet)**: Lightweight segmentation

→ Start here, then optimize further

Models for Time-Based Inputs

- **CNN-1D**: Simple, fast, ideal for low-frequency signals
- **Small RNNs (LSTM/GRU)**: Long memory, efficient inference
- **TCNs**: Dilated convolutions, good for anomaly detection
- **Spectrogram-based CNNs**: Transform audio into images
- **CRNNs**: Combine CNN + RNN, good for sound events

→ TCNs + Spectrogram CNNs are **go-to choices** today


$d = 4$
$d = 2$
$d = 1$

Neural Architecture Search (NAS): Worth It?

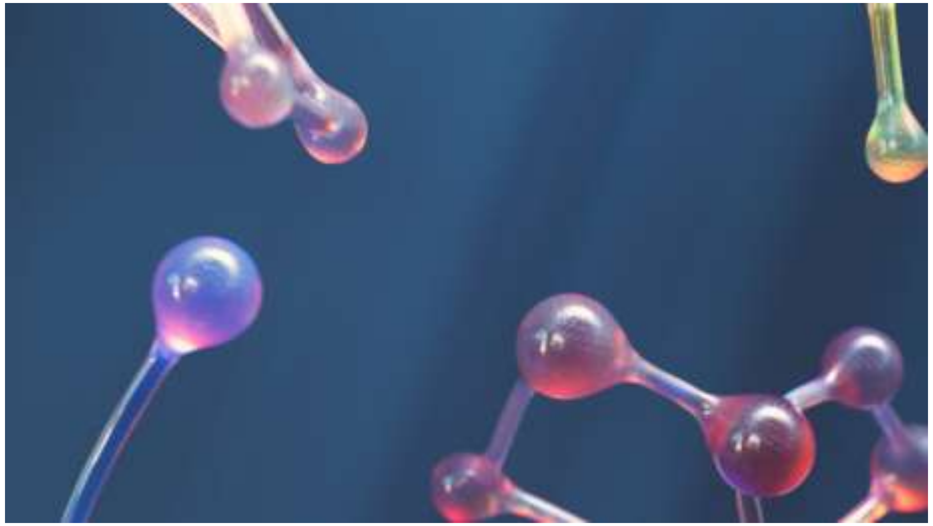
- NAS = **automated search** for optimal architectures
- Promises better accuracy – efficiency trade-offs

- ❌ Requires **huge compute** (train 100s–1000s of models)
- ❌ Complex, costly, and hard to validate
- ✅ Luckily, **MobileNetV3**, **EfficientNet** = NAS-designed

→ For most teams: skip the search, **use existing NAS models**

Optimizing for the Edge

- We've covered the core techniques of edge AI:
  - Quantization (post-training and during)
  - Pruning (structured/sparsity)
  - Knowledge distillation
  - Graph-level optimization
  - Lightweight architectures



→ These aren't academic tricks, they're essential for running AI at the edge

Exporting Models with the ONNX Format

From Training to Inference (ONNX)

- Training frameworks (e.g. PyTorch, TensorFlow):
  - Gradient computation
  - Debugging and visualization
  - Rely on **Python**, which isn't ideal

- These features are ideal for research, **not for deployment**.

- For real-time inference, we need something **lighter and faster**

Export Format Options

- **TorchScript** model file
  - PyTorch-native, not cross-framework
  - Works in C++, mobile apps
- **TensorFlow Lite** model file
  - Only for TensorFlow Lite (Micro)
  - Optimized for mobile, microcontrollers
- **ONNX (Open Neural Network Exchange)**
  - Cross-framework standard, widely used
  - Compatible with many inference engines

→ We focus on **ONNX** for its flexibility

What Is ONNX?

- **Open Neural Network Exchange** (ONNX) is:
  - A portable file format for machine learning models
  - Created by Microsoft and Facebook (2017)
  - Widely supported by almost all ML tools

Export models once, run anywhere!

ONNX File Contents

An .onnx file contains:

- The **computation graph** (layers/operations)
- The **learned weights** (parameters)
- Input/output specs (shapes, dtypes)

→ Portable, compact,
ready for deployment

ONNX vs ONNX Runtime

- **ONNX** = file format (model storage)
- **ONNX Runtime** = inference engine (model execution)

- You can run ONNX in many runtimes:
  - TensorRT
  - OpenVINO
  - And more…

→ ONNX is the **file format**,
runtimes are the **interpreters**

Exporting PyTorch Models to ONNX

- PyTorch has **built-in support** for exporting to ONNX.
- Use the torch.onnx.export() function.

- Exporting involves:
  1. Defining or loading a model
  2. Preparing a dummy input
  3. Calling the export function with key options

Step 1: Define or Load Your Model

```
class TinyCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1, 8, 3, 1, 1)
        self.fc = nn.Linear(8 * 28 * 28, 10)
    def forward(self, x):
        x = torch.relu(self.conv(x))
        x = x.view(x.size(0), -1)
        return self.fc(x)

model = TinyCNN()
model.eval()  # Eval mode for ONNX export!
```

Step 2: Prepare Dummy Input

```
dummy_input = torch.rand(1, 1, 28, 28)
```

- Required for tracing the **computation graph**
- Must match the input shape expected by the model
- Can be **random**, the data itself doesn't matter

→ Often one sample with batch size 1

Step 3: Export the Model

```
torch.onnx.export(
    model, dummy_input, "tiny_cnn.onnx",
    input_names=["input"], output_names=["output"],
    dynamic_axes={"input": {0: "batch"}, "output": {0: "batch"}},
    opset_version=20,
)
```

- Creates a .onnx file containing:
  - Graph structure
  - Learned parameters
  - Input/output names

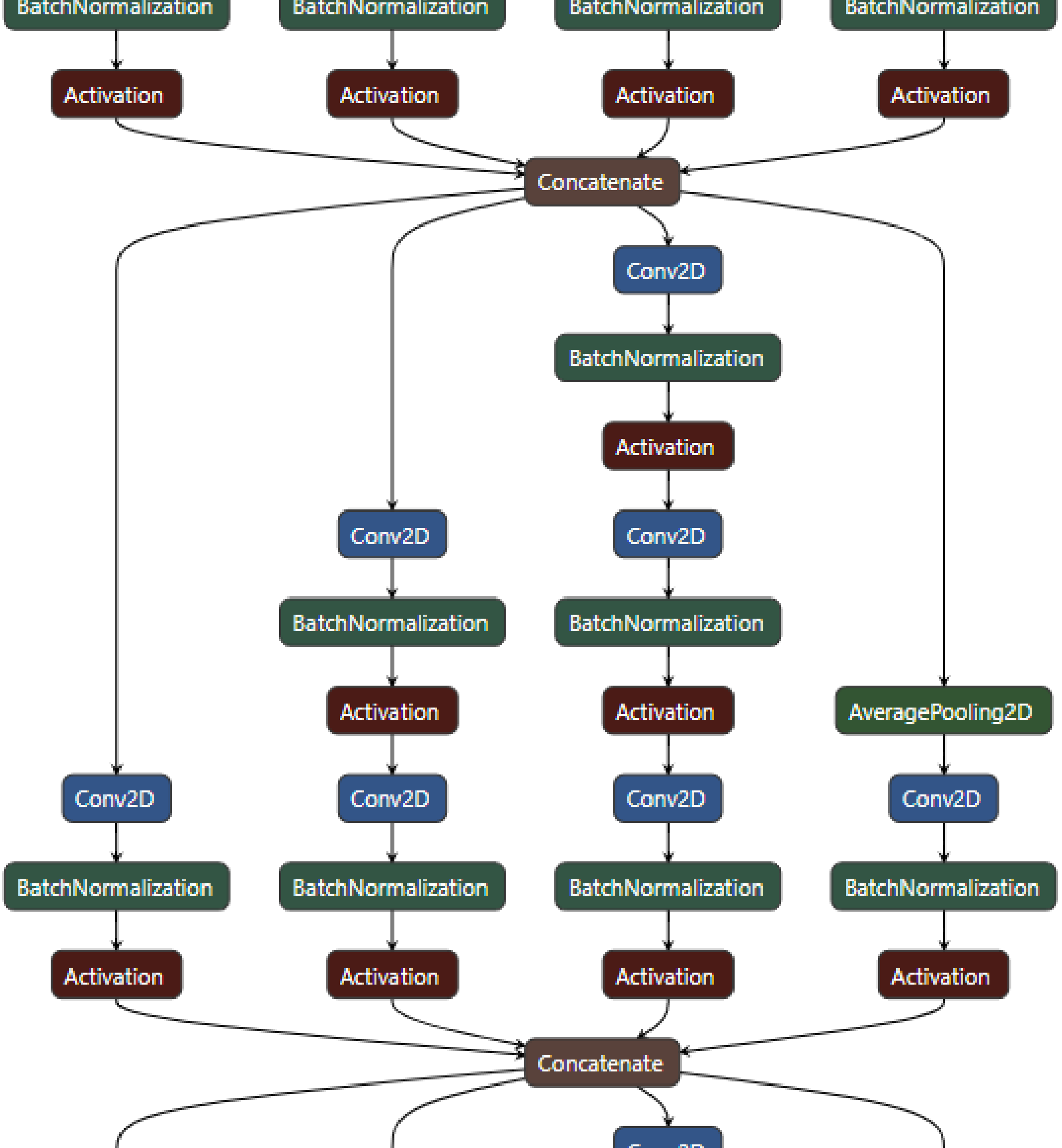
Key Export Options to Know

- **input_names / output_names**
  - Assigns names to the input and output nodes of the graph
- **dynamic_axes**
  - Enables variable **batch sizes** or input lengths
- **opset_version**
  - Controls which ONNX operators are used
  - PyTorch supports up to **version 20** (as of 2025)

Visualizing with Netron

- **Netron**: graphical model viewer
  - Opens .onnx files in the browser
  - Great for sanity checks and optimization review
- Lets you:
  - See full computation graph
  - Click to inspect layers and shapes
  - Compare versions side by side

Visit: https://netron.app/


BatchNormalization
BatchNormalization
BatchNormalization
BatchNormalization
Activation
Activation
Activation
Activation
Concatenate
Conv2D
BatchNormalization
Activation
Conv2D
Conv2D
BatchNormalization
BatchNormalization
Activation
Activation
AveragePooling2D
Conv2D
Conv2D
Conv2D
Conv2D
BatchNormalization
BatchNormalization
BatchNormalization
BatchNormalization
Activation
Activation
Activation
Activation
Concatenate

Efficient Inference with ONNX Runtime

What Is ONNX Runtime?

- A fast, lightweight inference engine for ONNX models
- Developed by Microsoft — **open-source** and **cross-platform**

- Designed to run models efficiently on:
  - CPUs, GPUs, embedded devices

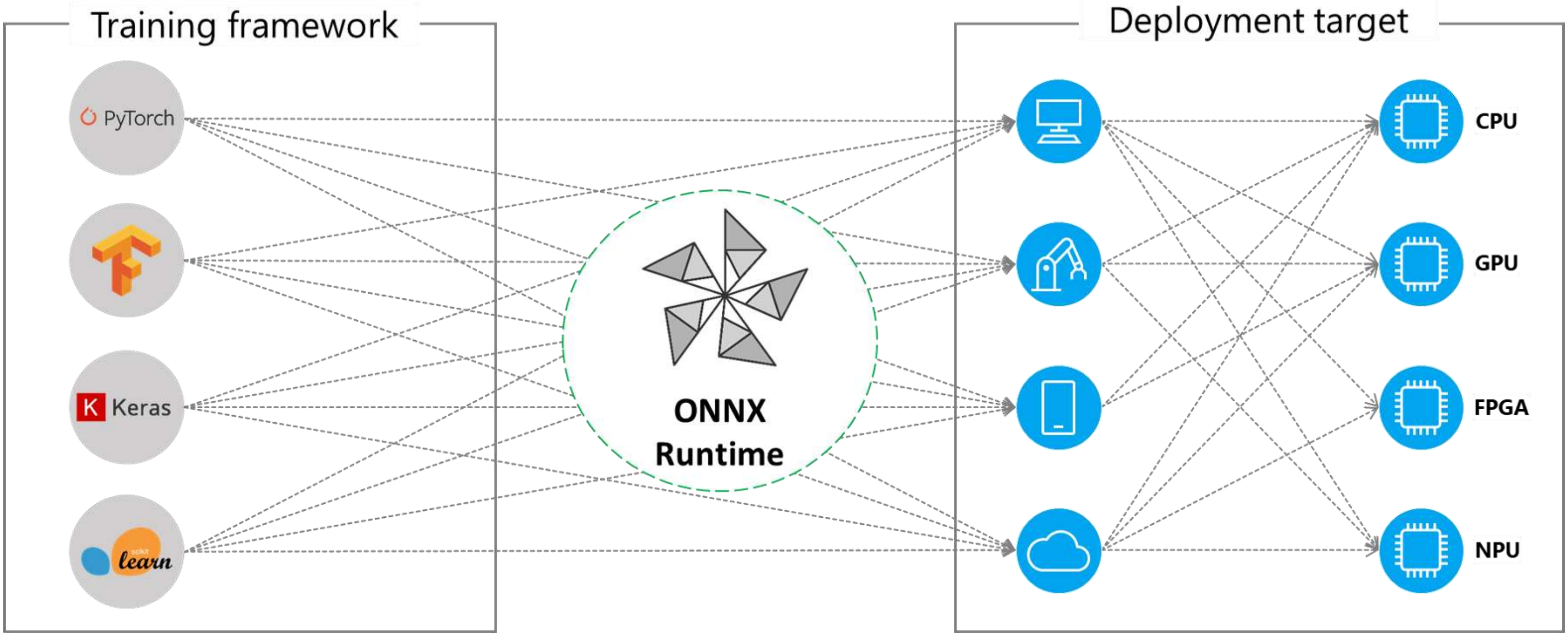
- Can be embedded into:
  - Edge applications
  - Mobile apps
  - Server APIs

Why Use ONNX Runtime?

- **Optimized for inference**
  - Strips away training overhead
  - Focuses on speed, memory efficiency, and portability

- **Hardware-agnostic**
  - Not tied to one vendor or device
  - Same model can run on many platforms

- Supports **Python**, but also:
  - C++, Java, C#, JavaScript, and others


Training framework
PyTorch
Keras
learn
ONNX
Runtime
Deployment target
CPU
GPU
FPGA
NPU

Execution Providers in ONNX Runtime

- Backends that run parts of the model on **specific hardware**.
- ONNX Runtime delegates calculations to these providers.

Provider	Hardware	Notes
CPU	All platforms	Always works, slower
CUDA	NVIDIA GPUs	Fast, needs CUDA/cuDNN
TensorRT	NVIDIA GPUs	Very fast, startup overhead
OpenVINO	Intel (CPU, GPU)	Optimized for Intel stack
XNNPACK	ARM, x86 CPUs	Good on mobile and edge

Example: Full Inference Pipeline

```
import onnxruntime as ort
import numpy as np

session = ort.InferenceSession("model.onnx")
input_name = session.get_inputs()[0].name

batch = np.random.rand(2, 1, 28, 28).astype(np.float32)
outputs = session.run(None, {input_name: batch})

print("Predictions:", outputs[0])
```

Graph-Level Optimizations

- ONNX Runtime doesn't execute the graph **as-is**
- It applies **automatic graph-level optimizations**
  - Fuses patterns (Conv + BN + ReLU)
  - Optimizes reshapes/transposes
  - Precomputes constant subgraphs
  - Removes identity operators
  - And many more…

→ Boosts **inference speed** and reduces **memory use**

Quantization in ONNX Runtime

- ONNX Runtime supports **Post-Training Quantization (PTQ)**
  - It does not support Quantization-Aware Training (QAT)
  - Although you can QAT with PyTorch and run with ONNX Runtime
- PTQ converts weights (and activations) to **INT8**.
- Two main flavors:
  - ✅ **Dynamic Quantization**
  - ✅ **Static Quantization**

→ Shrinks model size and boosts inference speed with minimal effort.

Dynamic Quantization

- Simplest form of quantization.

- Converts **weight tensors** from FP32 to INT8.
- Activations stay in FP32 — quantized on the fly at runtime.

- **No calibration data** required.

Dynamic Quantization: Code Example

```python
from onnxruntime.quantization import quantize_dynamic

quantize_dynamic(
    model_input="model_processed.onnx",
    model_output="model_dynamic.onnx",
    per_channel=True,  # Optional but helpful
)
```

- per_channel: Improves accuracy, especially for Conv layers.

Static Quantization

- Quantizes **both weights and activations**.
- Requires a **calibration dataset** to measure activation ranges.
- More setup, but better performance

1. Create a **calibration data reader**.
2. Provide representative input samples.
3. Run the quantize_static() function.

→ You'll get an INT8 model optimized for runtime memory and speed.

Static Quantization: Code Example

```python
from onnxruntime.quantization import quantize_static
from onnxruntime.quantization import CalibrationDataReader

class MyDataReader(CalibrationDataReader):
    def __init__(self): pass
    def get_next(self): return next(self.data, None)

quantize_static(
    model_input="model_processed.onnx",
    model_output="model_static.onnx",
    calibration_data_reader=MyDataReader(),
    per_channel=True)
```

Recap: Dynamic vs Static

Dynamic Quantization

- ✅ Easiest to apply
- ✅ No calibration needed
- ❌ Less effective on CNNs

Static Quantization

- ✅ Best performance on edge hardware
- ✅ Fully INT8 model
- ❌ Needs calibration data

Float16: A GPU-Friendly Optimization

- Not all models run on ARM or MCUs

- On **GPUs**, float16 (FP16) is a great middle ground:
  - ✅ Faster than FP32
  - ✅ Safer than INT8
  - ✅ No retraining or calibration needed

→ Use FP16 when deploying on Jetson, industrial PCs or GPU servers

Converting to Float16 in ONNX

```python
import onnx
from onnxconverter_common import float16

model = onnx.load("model_fp32.onnx")
model_fp16 = float16.convert_float_to_float16(model)
onnx.save(model_fp16, "model_fp16.onnx")
```

- Converted model runs faster on GPU, with **half the memory**.

ONNX Runtime Pipeline

You now know how to:

- ✅ Export models from **PyTorch** to **ONNX**
- ✅ Optimize **graphs** at runtime
- ✅ Apply dynamic and static **quantization**
- ✅ Convert models to **float16**
- ✅ Run inference on real hardware

High-Performance Deployment with TensorRT

What Is TensorRT?

- **TensorRT** is NVIDIA’s high-performance inference engine.

- It compiles trained models into **optimized GPU executables**.
- Works with models exported to **ONNX** format.

- Supports a wide range of NVIDIA devices:
  - Desktop RTX, data center GPUs, **Jetson**.

What TensorRT Does

- **Runs models faster** than PyTorch/TF/ONNX Runtime.

- Uses **graph compilation**, not layer-by-layer interpretation.
- Converts a model into a **static inference engine**

- Produces hardware-specific binaries for **maximum speed**.
- Ideal for production-grade deployments on NVIDIA GPUs.

Graph Compilation: The Secret Sauce

- Compilation occurs once, when building the engine.

- Process:
  1. **Parse** the model graph
  2. **Optimize** the graph structure
  3. **Benchmark tactics** per layer
  4. **Plan memory**, serialize engine

→ Inference becomes fast, low-latency, and efficient.

Step 1: Graph Parsing

- ONNX model is parsed into a DAG (directed acyclic graph).

- Validates operator support, shapes, and connectivity.
- Unsupported operators cause **build failure**.
- Most common operators from PyTorch/TF are supported.

→ Optional: add custom plugins for unsupported layers.

Step 2: Graph-Level Optimization

- TensorRT applies static graph rewrites:
  - **Operator fusion** (e.g. Conv + ReLU)
  - **Constant folding** (precompute static operators)
  - **Dead node elimination**
  - **Precision calibration** for FP16/INT8

→ Reduces kernel count, memory usage, and latency.

Step 3: Tactic Selection

- For each op, TensorRT tries **many GPU kernels** (tactics).
  - A single layer (e.g. Conv2D) may have 30+ kernel variants.
- Benchmarks tactics to find the **fastest one**.
- Based on:
  - Input shapes
  - GPU architecture
  - Precision settings

→ This makes build time longer, but inference **blazing fast**.

Step 4: Memory Planning & Serialization

- Allocates memory for:
  - Inputs and outputs
  - Intermediate tensors
- Uses **smart reuse** to reduce peak memory.
- Produces a **serialized engine file**:
  - Loadable without recompilation

→ Ready for production deployment

Building and Running TensorRT Engines

- After exporting a model to ONNX, the next step is to **build a TensorRT engine**.

- Steps:
  1. Load the ONNX model
  2. Compile it into an engine
  3. Run inference using GPU memory

→ All done in Python using **TensorRT + PyCUDA**.

Step 1: Load the ONNX Model

```
logger = trt.Logger(trt.Logger.INFO)
builder = trt.Builder(logger)
flags = 1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
network = builder.create_network(flags)
parser = trt.OnnxParser(network, logger)

with open("model.onnx", "rb") as f:
    if not parser.parse(f.read()):
        raise RuntimeError("Failed to parse ONNX")
```

Step 2: Build the Engine

```python
config = builder.create_builder_config()
config.max_workspace_size = 1 << 30  # 1 GB

if builder.platform_has_fast_fp16:
    config.set_flag(trt.BuilderFlag.FP16)

engine = builder.build_engine(network, config)
```

- Output: GPU-optimized **inference engine**

Step 3: Inference with the Engine

```
context = engine.create_execution_context()
d_input = cuda.mem_alloc(input_data.nbytes)
d_output = cuda.mem_alloc(output_data.nbytes)

cuda.memcpy_htod(d_input, input_data)
context.execute_v2([int(d_input), int(d_output)])
cuda.memcpy_dtoh(output_data, d_output)
```

- **Note**: manual memory transfers to/from GPU

Serializing Engines

```python
engine_bytes = engine.serialize()
with open("model.engine", "wb") as f:
    f.write(engine_bytes)
```

- Saves fully optimized engine to disk
- **Avoids rebuild** time during app startup

→ Required for fast startup in production.

Deserializing Engines

```python
runtime = trt.Runtime(trt.Logger(trt.Logger.INFO))
with open("model.engine", "rb") as f:
    engine = runtime.deserialize_cuda_engine(f.read())
context = engine.create_execution_context()
```

- Load precompiled engine — no ONNX needed
- Works **only on compatible GPUs**

→ Fast and lightweight deployment option.

TensorRT - Lower Precision

- Lower precision = faster inference + smaller models

- ✅ **FP32**: default, precise, slowest
- ✅ **FP16**: faster, **halves memory**, no calibration needed
- ✅ **INT8**: fastest, needs **quantization** metadata
- 🧪 **FP8**: experimental, Hopper GPUs only

→ Reducing bit width improves performance and power efficiency.

Advanced Techniques with Model Optimizer

What Is Model Optimizer?

- ModelOpt = NVIDIA’s **Model Optimizer** library for PyTorch
- Works before export**, during training**
- Optimizes model **before ONNX or TensorRT**
- Focuses on model-level compression,
  not just runtime optimization.

Where Model Optimizer Fits

PyTorch

↓

(ModelOpt: optional)

↓

ONNX export

↓

TensorRT engine build

↓

GPU inference

What Model Optimizer Enables

- Compression-aware **training** and structure refinement.

- Supports techniques like:
  - Quantization-aware training (QAT)
  - Structured pruning
  - 2:4 sparsity patterns
  - Knowledge distillation
  - NAS-assisted architecture search

→ All applied directly to your **PyTorch model**.

Model Optimizer Pipeline

1. Train a model in **PyTorch** (normal LR)
2. Apply **Model Optimizer**:
   - Quantization-aware training
   - Structured pruning or 2:4 sparsity
3. **Fine-Tune** the model (lower LR, less epochs)
4. Export to **ONNX** (as always)
5. Build a **TensorRT** engine (flags: INT8, sparse)

→ Deploy to NVIDIA GPU (desktop, Jetson, server)

Quantization-Aware Training (QAT)

```python
import modelopt.torch.quantization as mtq

def calibrate(model):
    model.eval()
    with torch.no_grad():
        for x in calibration_data:
            model(x)

model_q = mtq.quantize(model, mtq.INT8_DEFAULT_CFG, calibrate)
```

- Simulates INT8 using **fake quantization layers**
- Enables immediate INT8 export or QAT fine-tuning

Structured Pruning

```
pruned_model, _ = mtp.prune(
    model=model,
    dummy_input=torch.rand(1, 1, 28, 28),
    constraints={"flops": "50%"},
    mode="fastnas",
    config={
        "data_loader": train_loader,
        "score_func": evaluate_accuracy,
```

- Targets 50% FLOPs reduction
- fastnas = fast search for performant submodels

How Pruning Works

- Traces model with dummy input
- Evaluates subnetwork performance using your score_func
- Physically removes channels (e.g. from conv layers)

- Supports constraints:
  - "flops": Number of computations
  - "params": Number of parameters

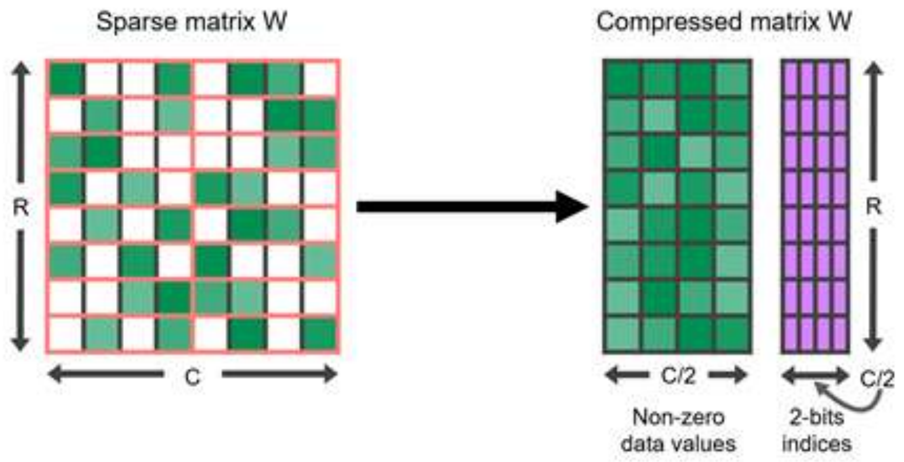
→ Final model is structurally **leaner**

2:4 Sparsity Pruning

- Enforces a hardware-friendly pattern:
  - For every **4 weights**, **2 are zero**

- Supported from NVIDIA Ampere (2020)
- Enables **sparse Tensor Core kernels**

→ Delivers **1.5–2× speedup** without model redesign



2:4 Sparsity with Model Optimizer

```
import modelopt.torch.sparsify as mts

sparse_model = mts.sparsity(model, “sparse_magnitude")
```

- Modifies weights to match **2:4 pattern**
- Can be applied **post-training** or **during training**

→ Sparse kernels = **real performance gains**

Advanced Features and Frontiers

- **AutoQuantize (Model Optimizer)**
  - Chooses best precision per layer: INT8, FP8 or FP16
- **New Precision Formats: FP8, INT4**
  - Ultra-low memory, high-speed inference
- **Runtime Adaptation & LoRA on the Edge**
  - Lightweight fine-tuning techniques like LoRA

- And so much more…

You’ve Reached the Last Layer

From Training to Edge Inference

You now know how to:

- 🎯 Select and train models for **constrained hardware**
- ⚙️ Optimize structure and precision for deployment
- 📦 Export to **ONNX** and build efficient inference engines
- 🚀 Accelerate with **ONNX Runtime** or **NVIDIA TensorRT**
- 🧠 Use **ModelOpt** to squeeze out speed and memory savings

→ This isn't just theory — it's a full path to **production**

Thank You & Good Luck!

- Keep learning as hardware evolves. 🎓
- Remember: **deployment is where real-world ML starts.**

Good luck and build something great! 🚀