AI Inference Optimization for Resource Constrained Edge Devices

Pavel Macenauer

R&D Manager
AI and Chip Enablement | NXP Semiconductors

Prague Embedded Software Workshop 2026
25 - 27 Jun 2026

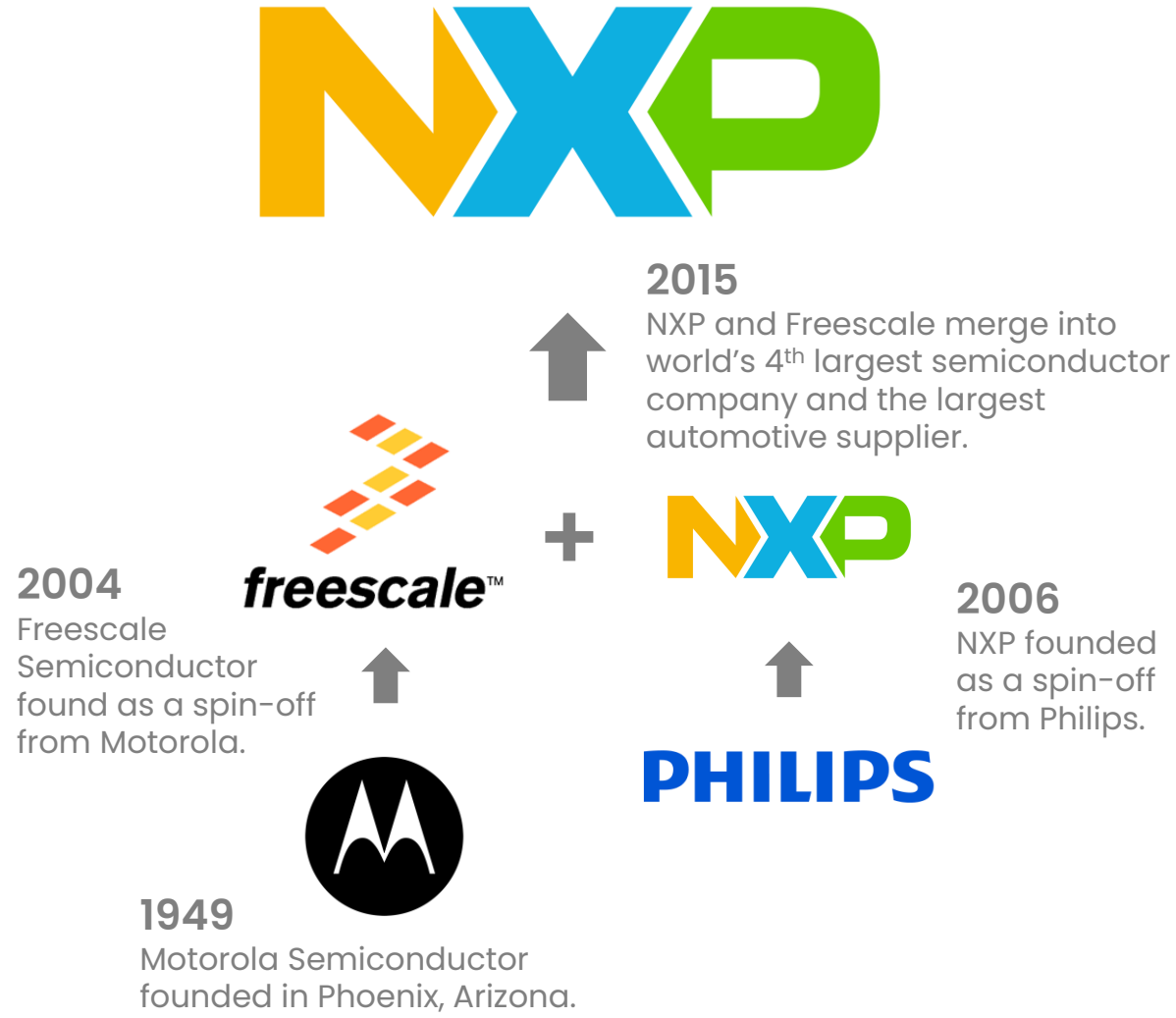
Agenda

About NXP

What is Edge AI?

Deploying AI Models to Edge Devices

Optimizing AI Models and Inference



S&P 500 Nasdaq NFC EDGEVerse I²C BUS

About NXP

NXP Semiconductors N.V. (NASDAQ: NXPI) is a Dutch semiconductor manufacturing and design company with headquarters in **Eindhoven, Netherlands** with 32 169 (2025) employees.

One of World's largest chip suppliers. Focusing on **B2B and mass-market segment** (security, automotive, industrial, IoT).

Presence in the Czech Republic:

- 2 software-focused sites both with 200+ engineers in **Brno** and **Rožnov pod Radhoštěm**.
- Smaller sites at university locations in **Zlín** and **Ostrava**.

Notable Inventions and Customers:

- Co-inventor of **NFC** (with Sony) and inventor of **I²C**.
- NXP chips can be found in products from **Samsung, Garmin, BMW, Ford, Bosch, Google, Meta, Amazon** and many others.

eIQ®

nxp.com/eiq

Suite of products and a software development environment for machine learning.

eIQ® Toolkit
PC tooling for preparation and deployment of neural networks to embedded and mobile devices.

eIQ® Runtime
Inference engines and libraries for microcontrollers and microprocessors.

eIQ® Neutron NPU
Highly scalable architecture for machine learning acceleration integrated into a wide portfolio of microcontrollers and application processors.

Major contributors to opensource

TensorFlow ONNX GitHub docker

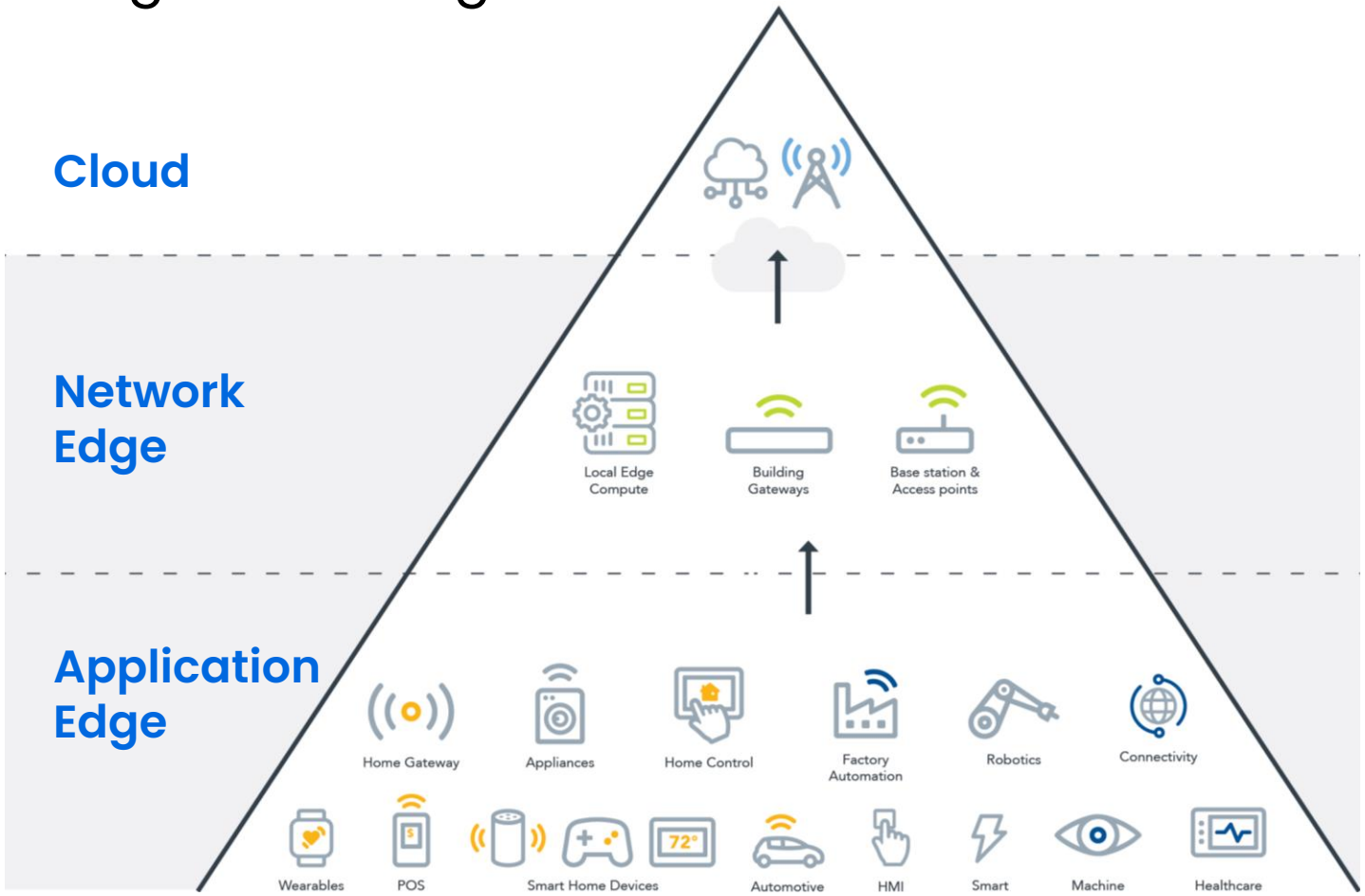
eIQ® Toolkit applications

- AI Toolkit
- Time Series Studio
- GenAI Flow
- AI Hub
- Agentic AI Framework

What is Edge AI ?

Edge AI is an area of AI which focuses on **deployment and inference of AI models locally**, close to where the data is generated and collected, rather than relying on centralized cloud servers for processing — enabling:

- **real-time decision-making**
- **reduced latency**
- **lower power** consumption
- enhanced data **privacy**
- improved **reliability** even in environments with **limited or no connectivity**



Edge Device Challenges

Performance – Latency and Memory

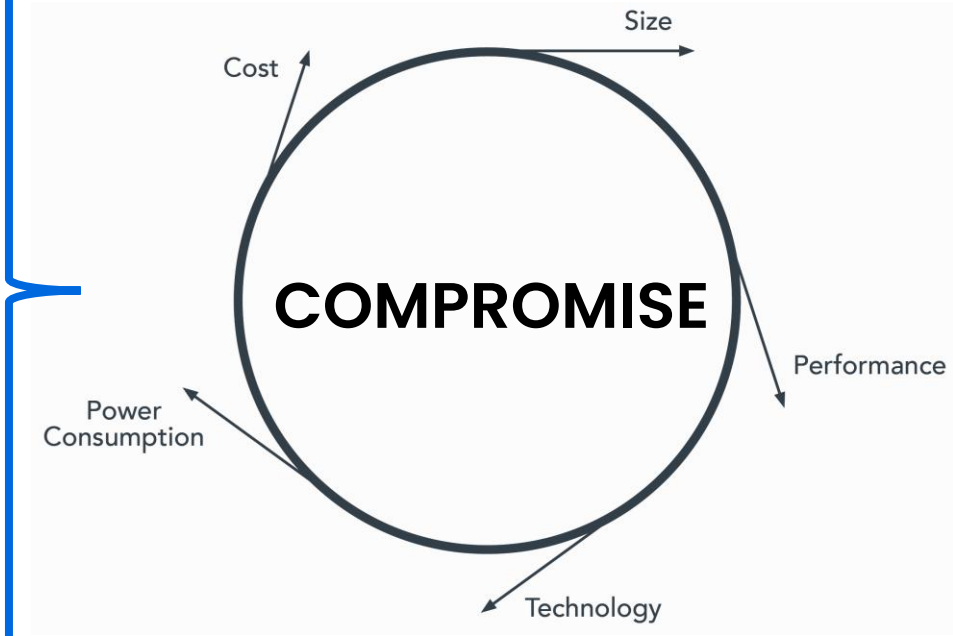
Edge devices must often provide a response within a **guaranteed** time frame compared to a desktop computer where response time is not critical. They also have **memory constraints**.

Power Consumption

Edge devices are powered by **batteries**, run in **ultra-low power** modes and may require to run for years without recharging or replacing the battery. Cooling also plays an important role.

Technology, size and design cost

Unlike in a desktop environment which is almost always driven by performance, edge devices can only fit limited technology. **Die size** plays an important role.



Challenges of Running a Model on the Edge

Running a model on CPU (x86/aarch64) typically **works right away**. Latency or power are not optimal.

Splitting workload between different platforms (CPU/GPU/NPU/DSP).

Quantization of the model to utilize the accelerator (4/8/16b int) decreases accuracy.

NPU driver does not fully support the **inference framework** (PyTorch, TensorFlow).

- Lacks operator implementation support.
- Different operator definitions.
- Unsupported datatypes (mixed schemas).

The **N3-64 NPU** (eIQ Neutron NPU configuration) consists of 64 MAD blocks for up to **64 MACs/cycle** for 8b x 8b operations. With a maximum frequency of **325 MHz**, the NPU provides up to **41.6 GOPS** (325 x 2 x 64 = 41.6).

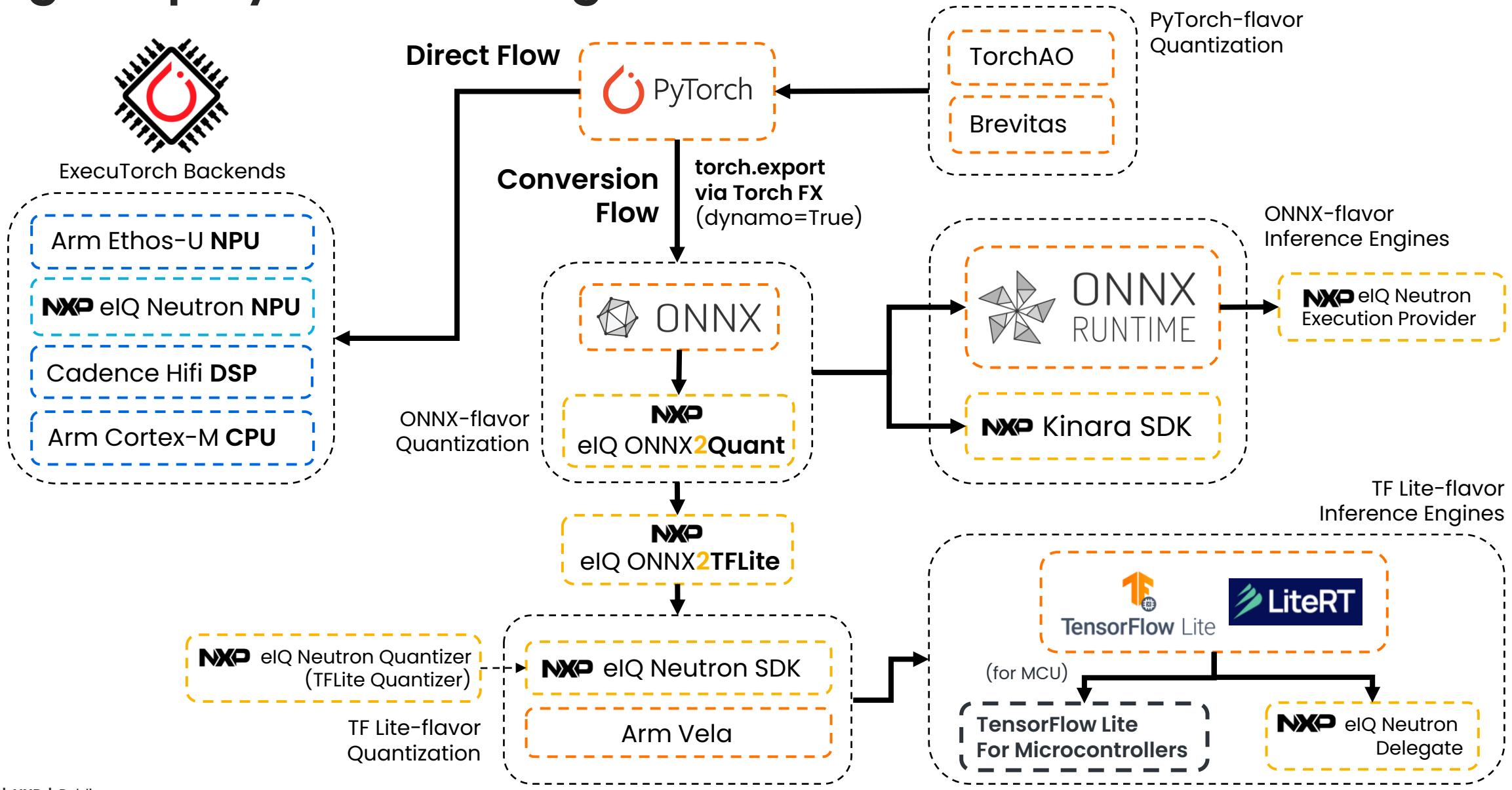
Model	Arm Cortex-M33 only	eIQ Neutron NPU fully offloaded	Increase
MLPerf Tiny AD01	2.3 ms	0.134 ms	**17.5x**
MLPerf Tiny KWS	28.5 ms	0.384 ms	**74.1x**
MLPerf Tiny Resnet	121.2 ms	0.717 ms	**169.1x**
MLPerf Tiny VWW	91.4 ms	0.966 ms	**94.6x**

CPU vs. NPU Acceleration using TF Lite Micro on i.MX RT 700 @ 325 MHz (2025)
See https://mlcommons.org/ for benchmarking details.

Platform	Latency	Notes
ARM Cortex-M33 CPU fp32 I/O	77694.018 ms	-
eIQ Neutron NPU fp32 I/O	257.163 ms	-
eIQ Neutron NPU int8 I/O	38.917 ms	fully quantized to int8 including I/O
ARM Cortex-M33 CPU + eIQ Neutron NPU fp32 I/O	124.617 ms	2.1x faster vs. NPU-only variant (justifying mixed backends)

Experimental ExecuTorch Backend latency for fp32/int8 on i.MX RT700 @ 325 MHz
MobileNet v2 224x224x3; ExecuTorch 1.0+ (2026)

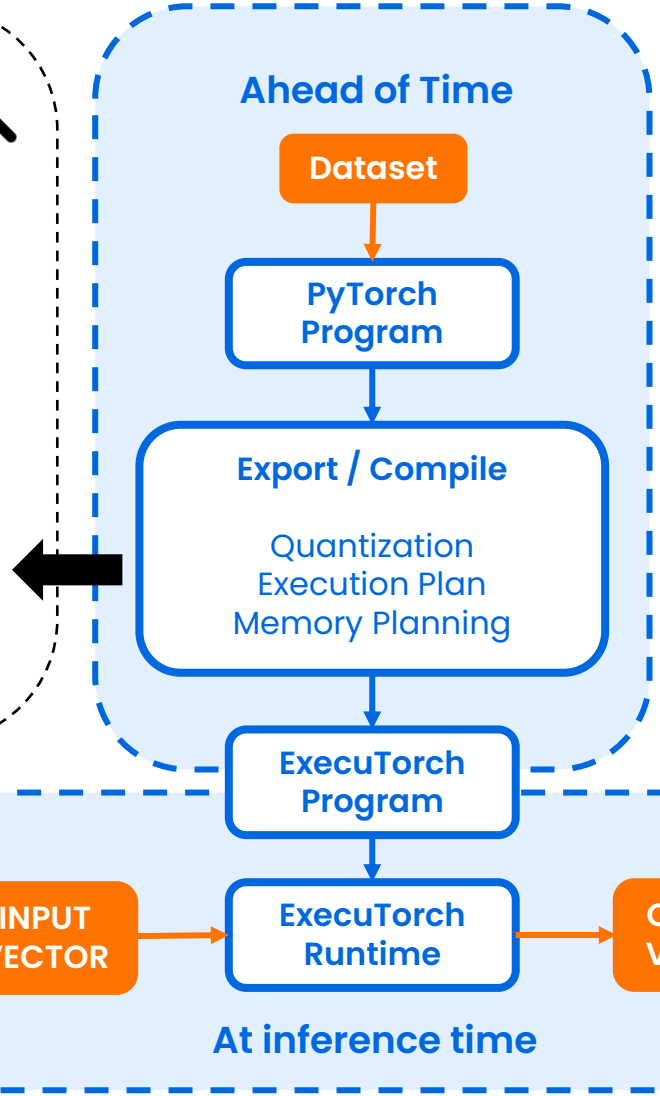
Edge Deployment Strategies



From PyTorch to ExecuTorch

Export / Compile Lowering:

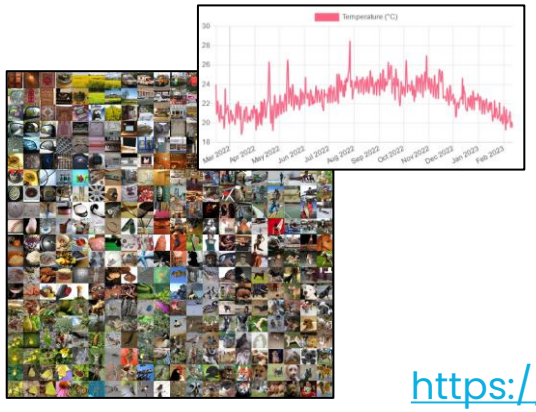
1. **PyTorch Program (nn.Module)**
   PyTorch Python program for training.
2. **ATen Dialect**
   Exported IR graph and
   entry point to ExecuTorch.
3. **Edge Dialect**
   Specializations useful for edge devices which can further run on different hardware.
4. **(optional) Backend Dialect**
   Special variant of edge dialect introducing hardware-awareness.



Training is done in PyTorch typically on server-class machines, GPU farms, in the cloud, etc.

Inference (run) happens on the device. ExecuTorch runs the generated *.pte file.

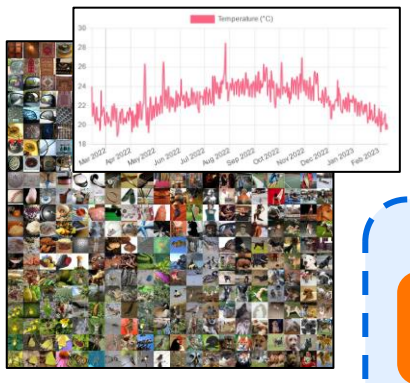
https://github.com/pytorch/executorch/tree/main/backends/nxp

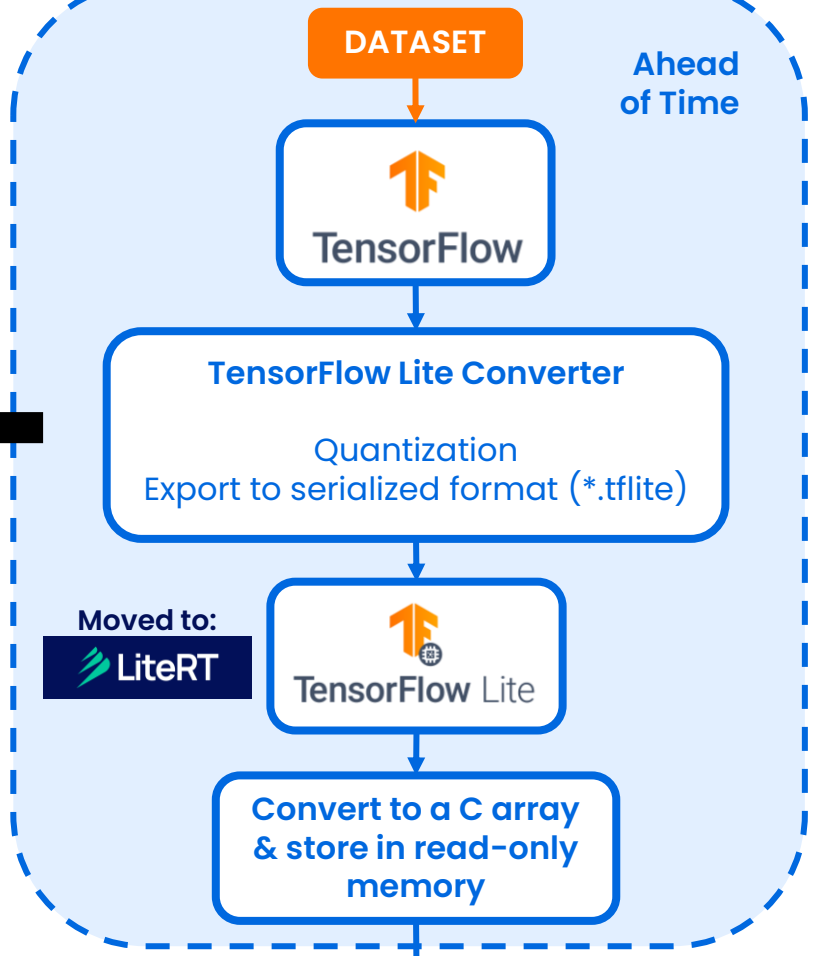


https://github.com/pytorch/executorch/tree/main/backends/nxp

From TensorFlow to TF Lite Micro

```python
import tensorflow as tf

# Create a model using high-level tf.keras.* APIs
model = tf.keras.models.Sequential([
    # Layers
])

# Compile and train the model.
model.compile(optimizer='sgd', loss='mean_squared_error')
model.fit(x=[-1, 0, 1], y=[-3, -1, 1], epochs=5)
# (to generate a SavedModel)
tf.saved_model.save(model, "saved_model_keras_dir")

# Convert the model.
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the model.
with open('model.tflite', 'wb') as f:
  f.write(tflite_model)
```





Protocol Buffers (Protobuf) is an open-source cross-platform data format used to serialize structured data. It is useful in developing programs that communicate with each other over a network or for storing data.

protobuf
Protocol Buffers

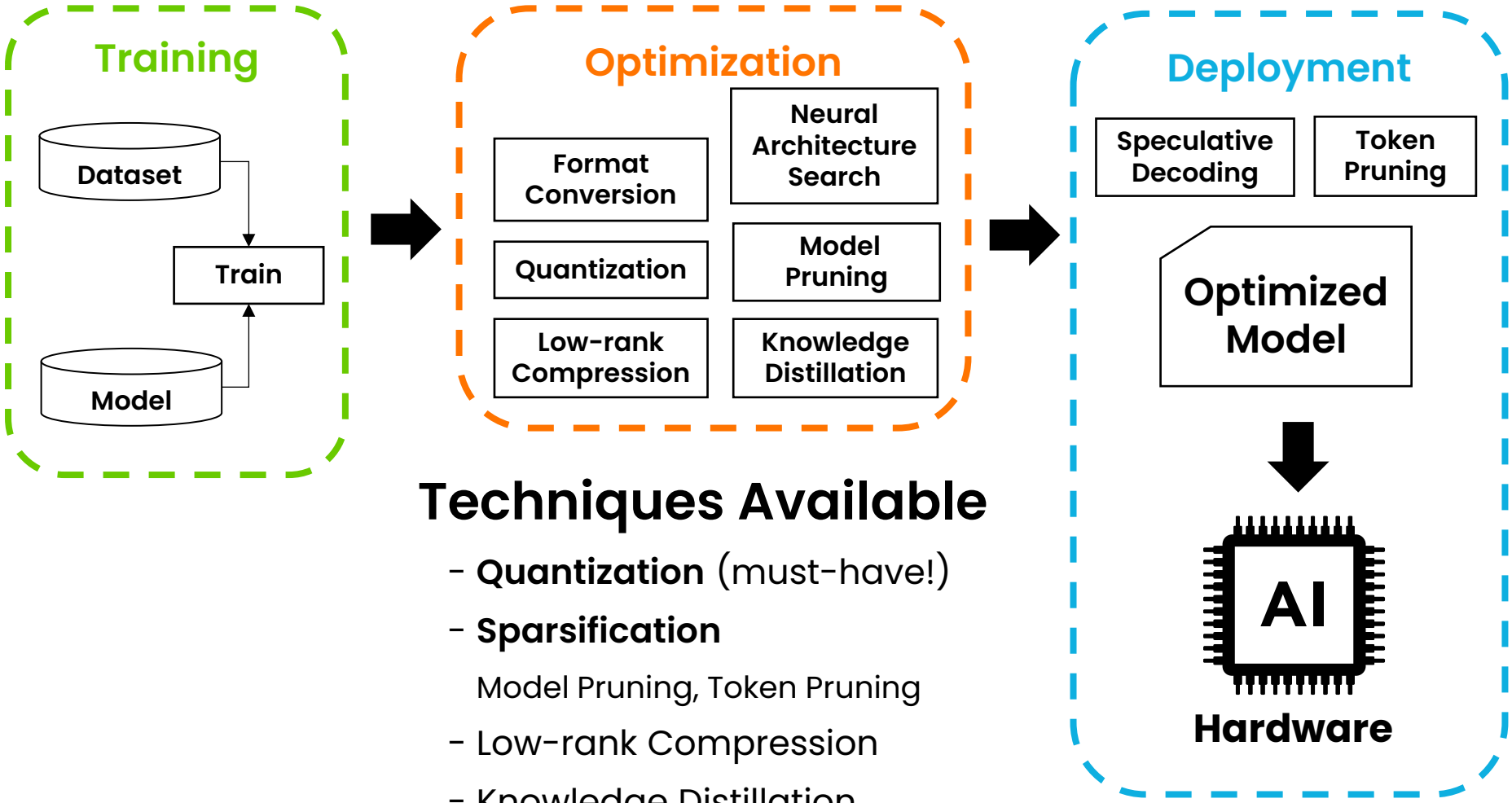
https://github.com/protocolbuffers/protobuf

FlatBuffers is an efficient cross platform serialization library for C++, C#, C, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust and Swift. It was originally created at Google for game development and other performance-critical applications.

FlatBuffers

https://github.com/google/flatbuffers

Optimizing AI Inference is key for efficient deployment!



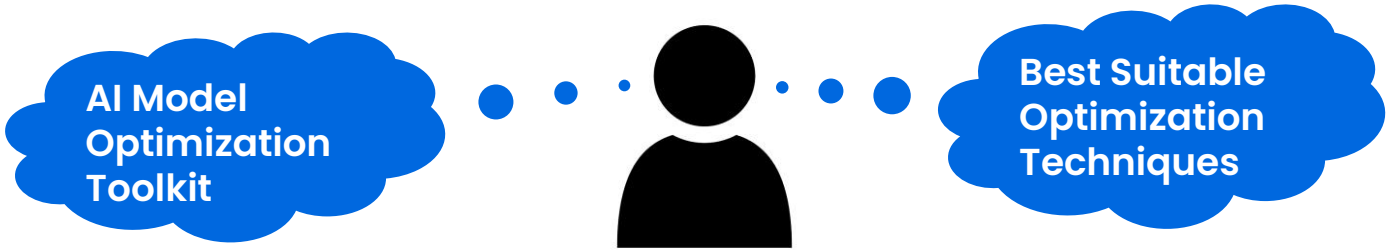
Techniques Available

- **Quantization** (must-have!)
- **Sparsification**

  Model Pruning, Token Pruning
- Low-rank Compression
- Knowledge Distillation
- Neural Architecture Search
- Speculative Decoding

ONNX Optimization Framework - eIQ Olive

"Given a model and targeted hardware, Olive (abbreviation of **O**nnx **LIVE**) composes the **best suitable optimization techniques** to output the most efficient ONNX model(s) for inferencing on the cloud or edge, while taking a set of constraints such as accuracy and latency into consideration."



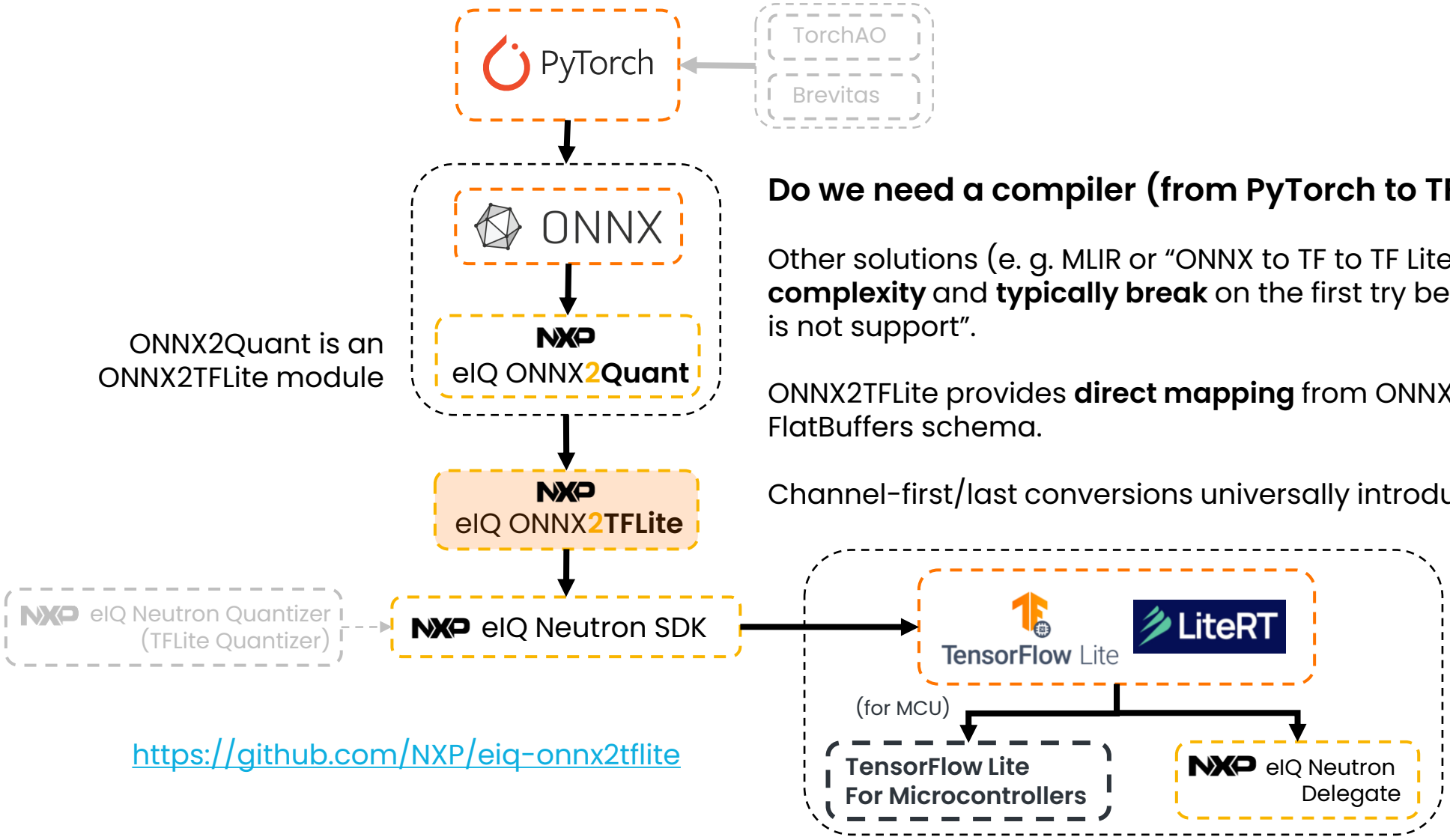
"Olive: The **AI Model Optimization Toolkit** for the ONNX Runtime"

https://microsoft.github.io/Olive/
https://github.com/microsoft/Olive

Fork

https://github.com/nxp/eiq-olive

eIQ ONNX2TFLite



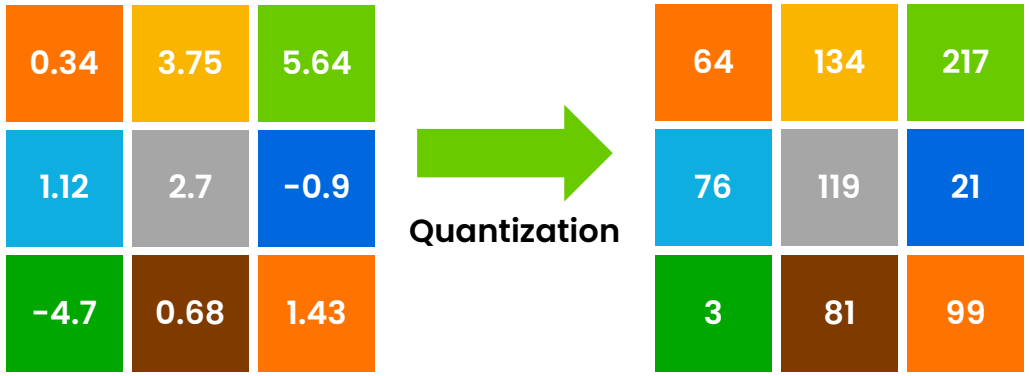
https://github.com/NXP/eiq-onnx2tflite

Do we need a compiler (from PyTorch to TF Lite)?

Other solutions (e. g. MLIR or "ONNX to TF to TF Lite") have **high complexity** and **typically break** on the first try because "something is not support".

ONNX2TFLite provides **direct mapping** from ONNX to TF Lite FlatBuffers schema.

Channel-first/last conversions universally introduce **transposes**.



Quantization

Mapping a large set of numbers to a smaller set.

An analog signal to a discrete signal.

Removing outliers.

A number represented using more bits to a number represented using less bits

- Typically, 32/16-bit float to 16-bit float or to 32/16/8-bit integer.
- 4-bit or binary neural networks are mostly experimental.
- **8-bit int ADD** consumes **30x** less energy than a 32-bit float ADD
- **8-bit int MUL** consumes **18.5x** less energy than a 32-bit float MUL

Post-training Quantization

- 2 types – Quant-Aware Training and **Post-Training Quantization**
- For int8 values maps to [-128, 127] - zero point = 0 which allows HW to skip multiplication of zero point by the activation value
  - Unsigned int8 can also be used, but usually has worse performance due to being asymmetric

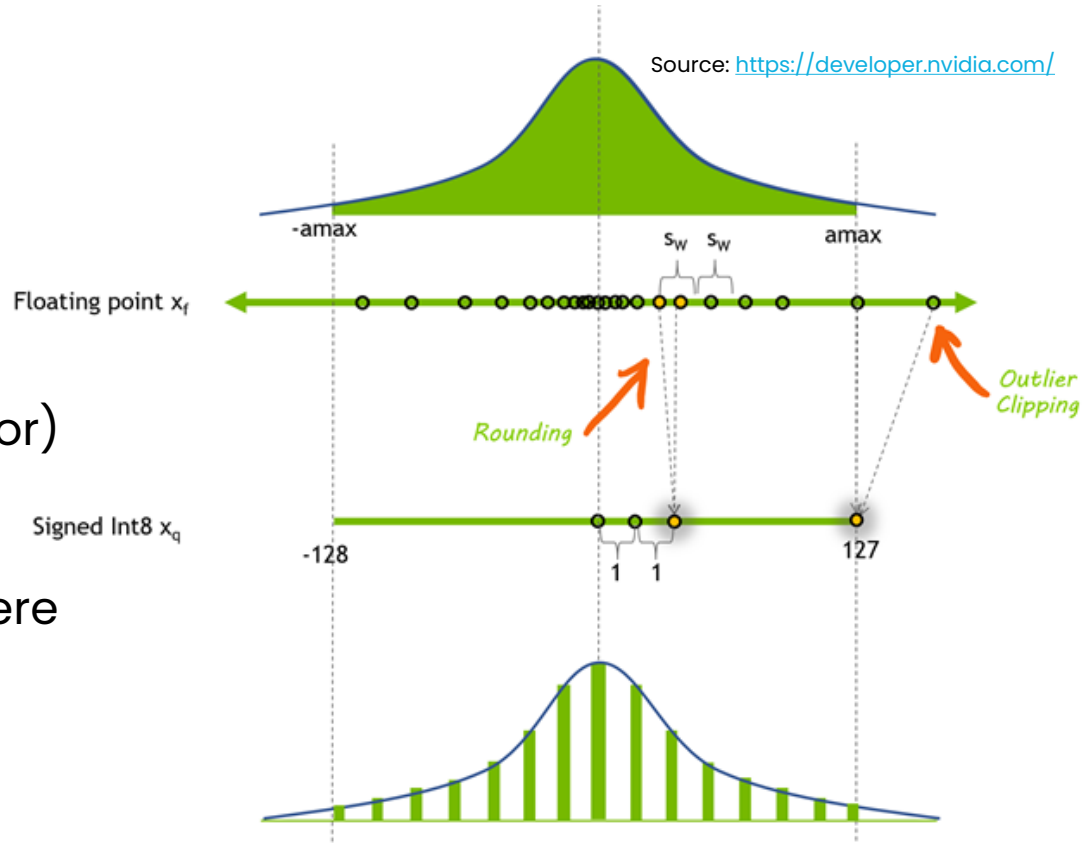
$$real\_value = (int8\_value - zero\_point)\ x\ scale$$

Quant-Aware Training shows minimal benefit to accuracy for CNNs, however lately with **Transformers**, **Large Language Models**, **Squeeze-and-Excite** blocks or **Hard-Swish** activation function, we start seeing its application where even bit-level errors make a difference.

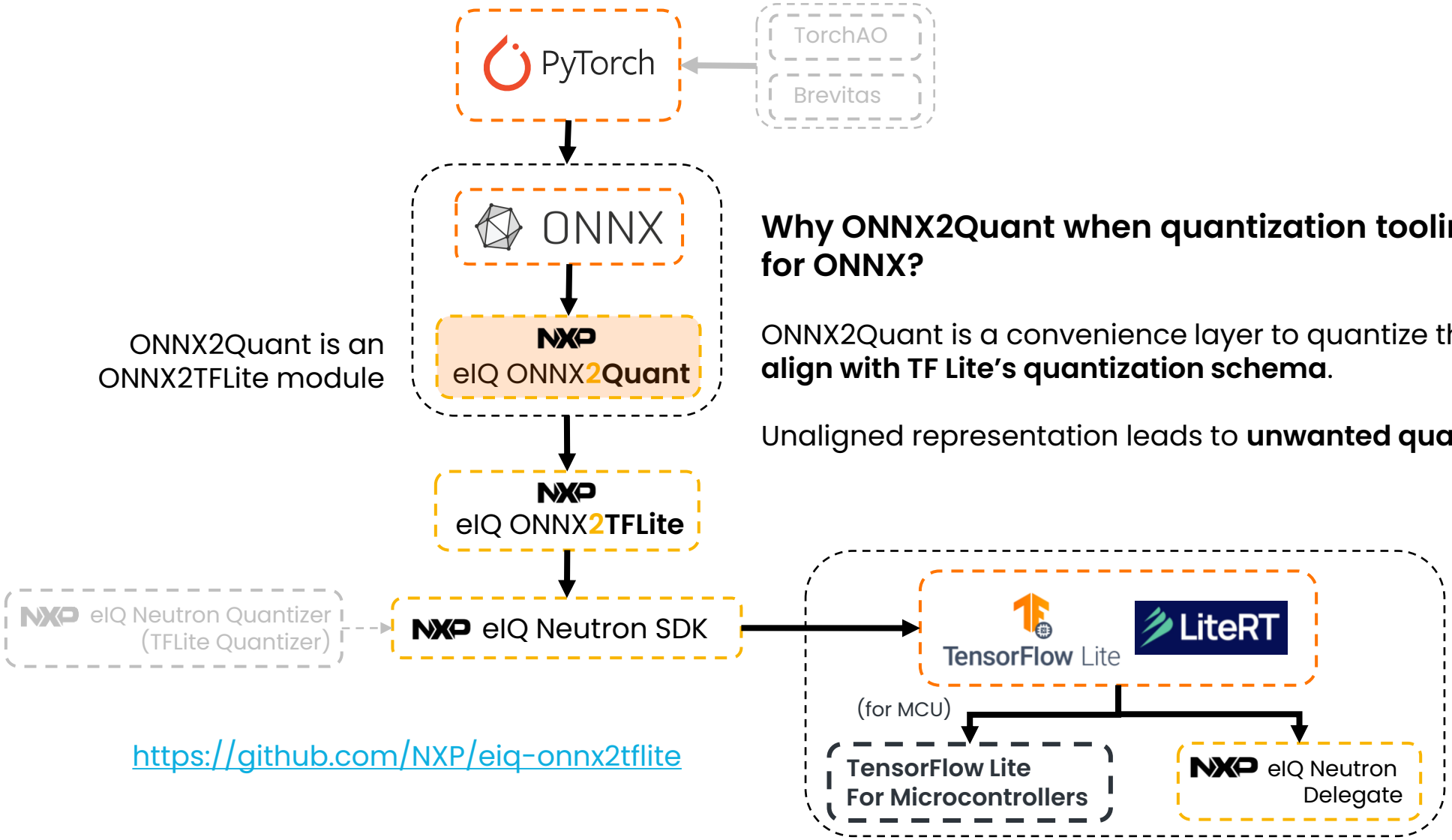
Calibration

1. providing a representative dataset
2. calculating distribution
3. clipping and rounding

- **Per-tensor**
  min/max range is selected across all channels (whole tensor)
- **Per-channel**
  min/max range is determined per every channel (typically only done on convolutions and depthwise convolutions where it makes a significant difference in accuracy)



eIQ ONNX2Quant in eIQ ONNX2TFLite



https://github.com/NXP/eiq-onnx2tflite

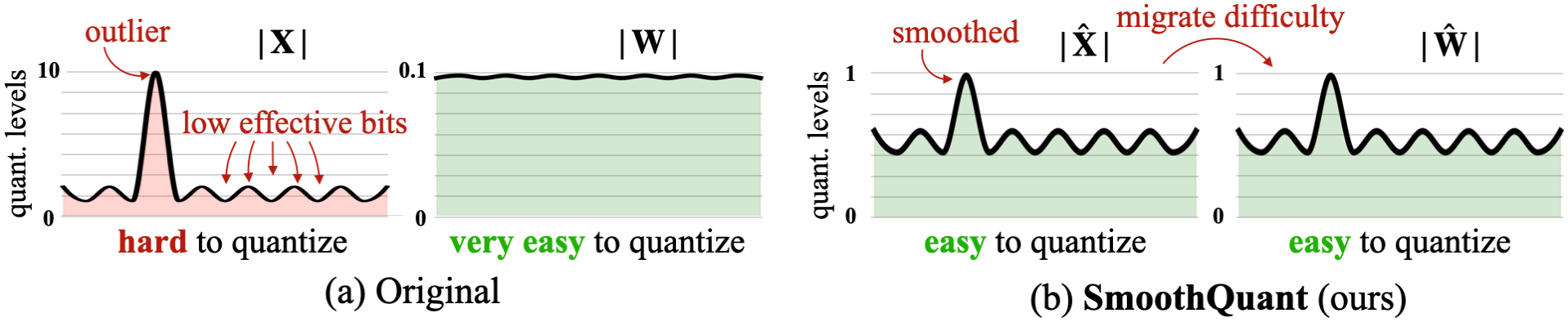
Why ONNX2Quant when quantization tooling already exists for ONNX?

ONNX2Quant is a convenience layer to quantize the ONNX model to **align with TF Lite's quantization schema**.

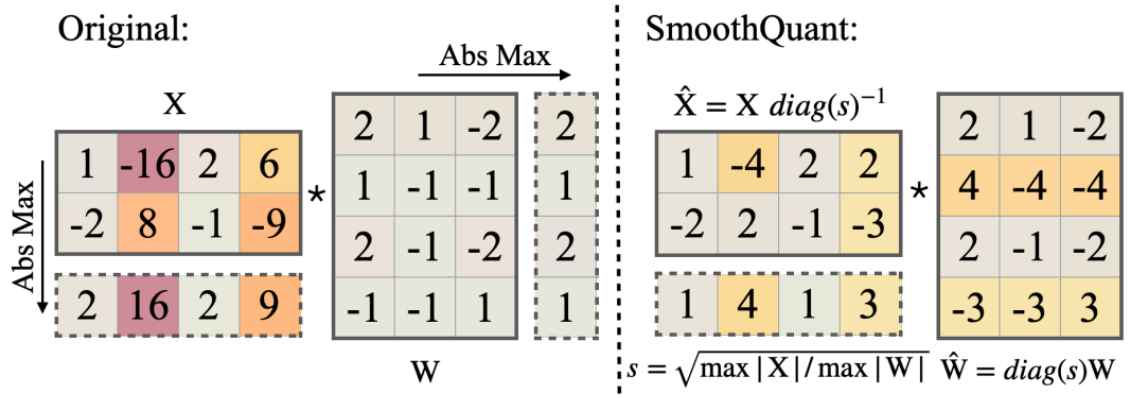
Unaligned representation leads to **unwanted quant & dequant** ops.

SmoothQuant

- Legacy quantization algorithms **do not work** with modern LLMs and most Transformers.
- **Smoothquant** introduces a **hyperparameter** α as a smooth factor to calculate the conversion per-channel scale and balance the **quantization difficulty of activation and weight** for W8A8 quantization producing a a mathematically equivalent scaling transformation.



- Includes an algorithm to search for optimal α.
- Outlier values are **~100x larger than most of the activation values** but they appear only in a small fraction of the channels.
- More advanced algorithms such as **RPTQ** (Reorder-based Post-training Quantization) are to be explored.



SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models
Official SmoothQuant Github Repository

Pruning

Model Pruning **removes model parameters** to reduce computational complexity and memory requirements.

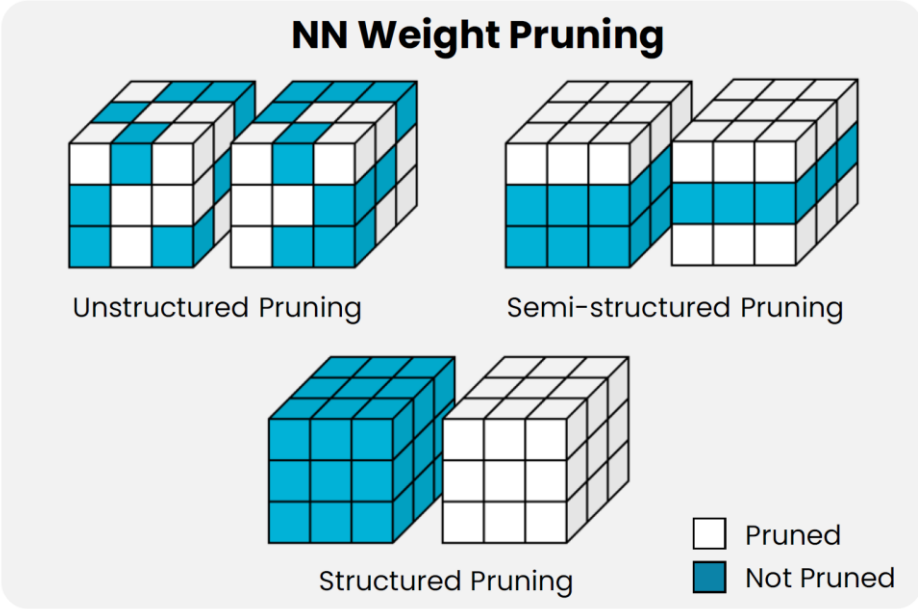
Motivation:

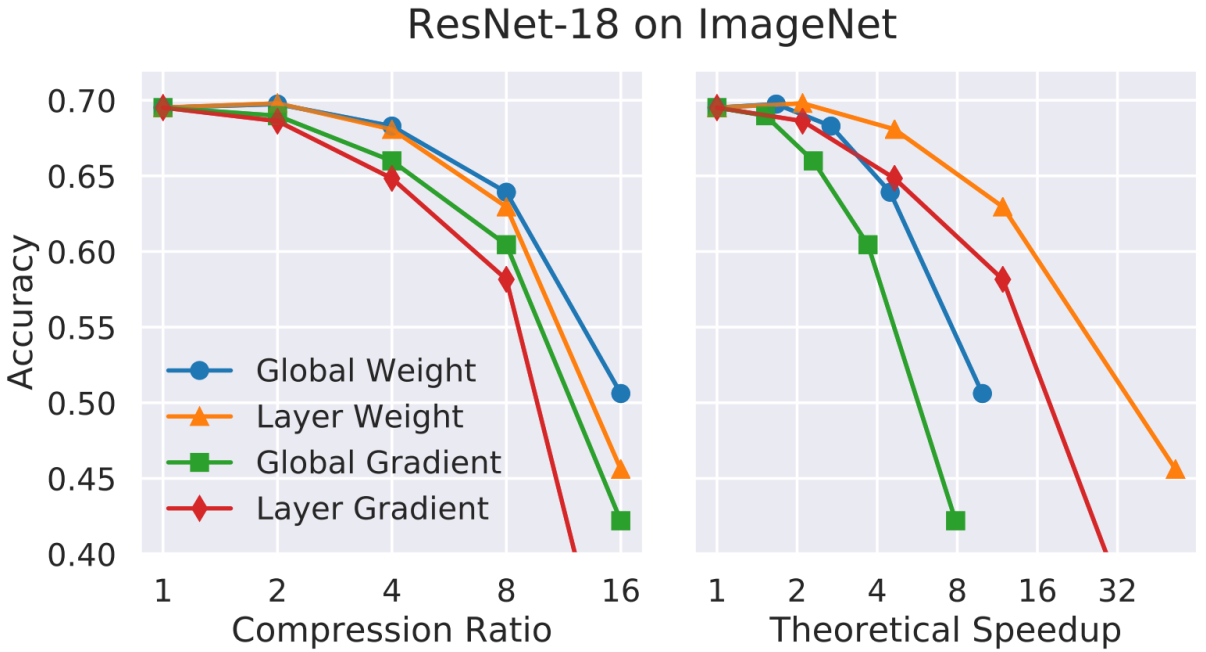
- Weights in a NN can be removed without substantial impact (redundant parameters).

1. Start from a pre-trained model
2. Remove weights
3. Fine-tune iteratively for various compression ratios.

Setting up pruning:

- **How to prune?** Criteria for pruning weights (e.g., weight magnitude, gradients, etc.)
- **What to prune (granularity)?** Unstructured/Structured/Semi-structured
- **Uniformity?** local vs. global





D. Blalock, et al., "What is the state of Neural Network Pruning?," MLSys '20

Comparison of Pruning Methods

Type	Ability to Preserve Task Performance	Benefits	Special SW/HW Requirements
Unstructured *	**High**	Model Size	HW Decompression Engine
Semi-Structured	**Mid**	Model Size Latency	HW Engine / Sparse NN Kernels
Structured	**Low**	Model Size Latency	None

* Unstructured pruning compression supported on **i.MX93** and **i.MX95** (w/ NXP's eIQ Neutron NPU)

Get in touch

Pavel Macenauer

pavel.macenauer@nxp.com

nxp.com

https://www.linkedin.com/in/pavel-macenauer