Friedrich-Alexander-Universität
Technische Fakultät

Energy-efficient Deployment of Deep Learning Applications on Cortex-M based Microcontrollers using Deep Compression

Mark Deutel[1], Philipp Woller[2], Christopher Mutschler[2] and Jürgen Teich[1]
[1] Friedrich-Alexander-Universität Erlangen-Nürnberg
[2] Fraunhofer Institute for Integrated Circuits, Fraunhofer IIS, Nürnberg

Motivation

TinyML – Machine Learning on the Edge

Efficiency

- Processing of data close to the sensor
- Re-usage of (hardware) resources required to drive the sensor

Reliability

- No communication via error prone network required
- Short, predictable “round-trip time"

TinyML

Cost

- Exploitation of already available cheap consumer-grade hardware
- Low energy footprint

Security

- Possibly confidential data is processed on the sensor node
- No connection to external cloud or server required

Motivation

DNN Deployment on Microcontrollers – An easy task?

Deep Neural Network Architectures[1]

Metrics	**AlexNet**	**VGG 16**	**ResNet 50**
# Layers	8	16	54
Total Weights	61 M	138 M	25.5 M
Total MACs*	724 M	15.5 G	3.9 G

1. Sze, Vivienne, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. „Efficient Processing of Deep Neural Networks: A Tutorial and Survey“. 2017.

Target Micro Controllers

Metrics	**Raspberry Pi Pico**	**Arduino Nano 33 BLE Sense**
Processor	ARM Cortex M0+	ARM Cortex M4
Clock Speed	133 MHz	64 MHz
Flash memory	2 MB	1 MB
SRAM	256 KB	256 KB

Significant gap between DNN requirements and available resources

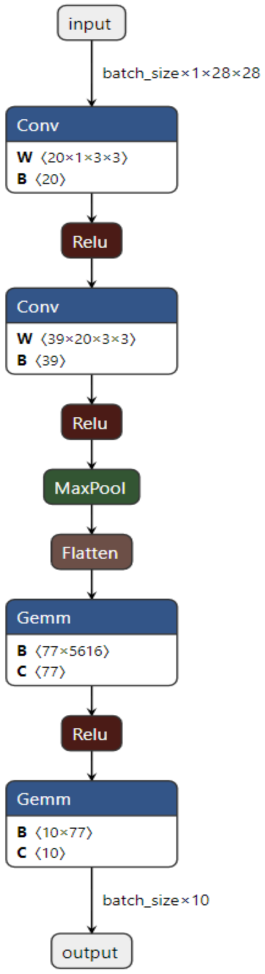
- Low processor speed vs. large number of mathematical operations
- Strict memory limitations vs. large number of weights and big inputs/feature maps
- High precision floating point datatypes vs. hardware often focused on integer arithmetic

* Multiply-Accumulate Operations

Motivation

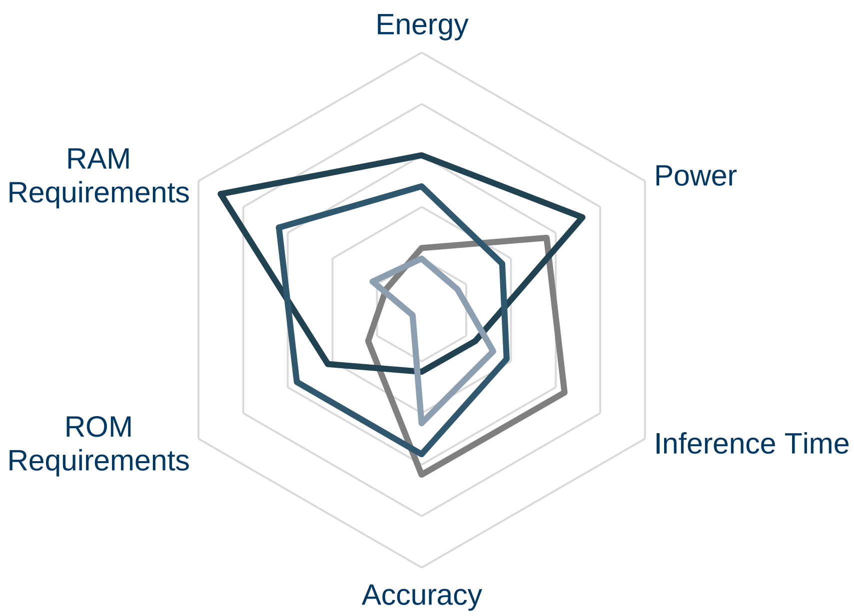
DNN Deployment on Microcontrollers – An Matter of Optimization

DNN Architecture, Dataset



DNN Compression and Deployment

Deployment on Target System



Deployment of DNNs on Microcontrollers

A Fully-Automated End-to-End Pipeline for DNN Deployment

DNN Training | Post Processing

Dataset, Model, Hyperparameter

Neural Network Pruning

Pruning Strategy

→ How to prune?
→ What to prune?
→ When to prune?

Weight Quantization

Quantization Model

→ How to perform?
→ When to apply?

Deployment on Target

Data Representation

→ Optimized Memory Layout

Target Code Generation

→ Convert Graph to Code
→ Graph Optimizations
→ Algorithmic Optimization

Deployable Model

Suggest next set of Hyperparameters (parameter space) using Multi-Objective Optimization based on performance metrics (objective space: accuracy, ROM, RAM, FLOPS)

Agenda

- Deployment of DNNs on Microcontroller Targets
  - Network Pruning
  - Weight Quantization
  - Microcontroller Deployment
- Evaluation of End-To-End Training, Compression and Deplyoment
  - Pruning and Quantization
  - From Size Reduction to Memory Savings
  - Deployment on Microcontrollers
- Conclusion

Deployment of DNNs on Microcontroller Targets

Network Pruning

Pruning Strategy

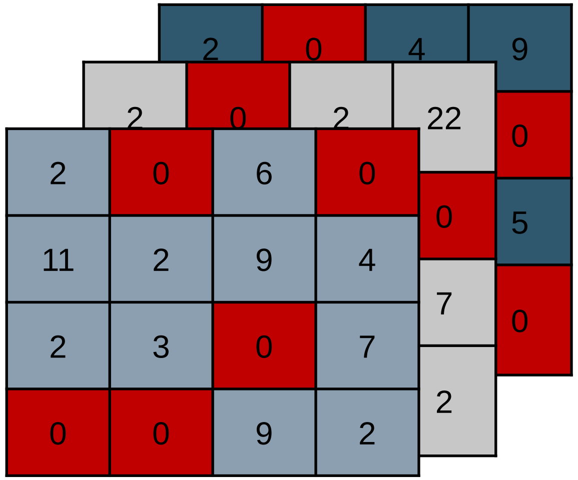
Understanding: Neural Networks are extremely over-parametrized and have a lot of redundancies in their parameters

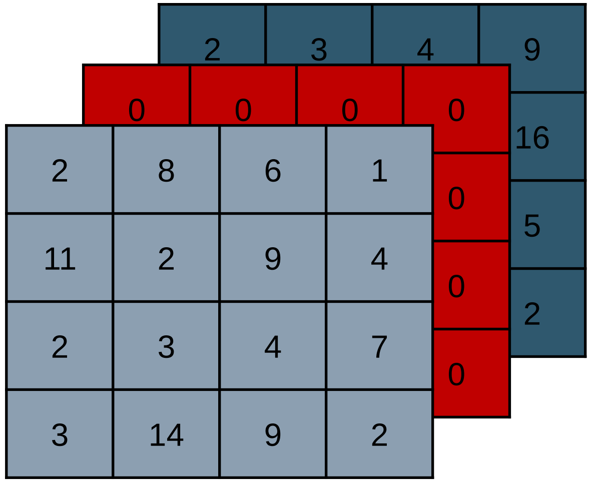
Element-Wise (Unstructured) Pruning[1]

- Set single weights to zero
- Sparse data structures remain at the end of training

Structured Pruning[2]

- Set whole structures of weights to zero
- Structures (and their dependencies) can be removed at the end of training





1. LeCun, Yann, John S Denker, and Sara A Solla. "Optimal Brain Damage", 1989.
2. Anwar, Sajid, Kyuyeon Hwang, and Wonyong Sung. "Structured Pruning of Deep Convolutional Neural Networks". 2015.

Network Pruning

Pruning Heuristics

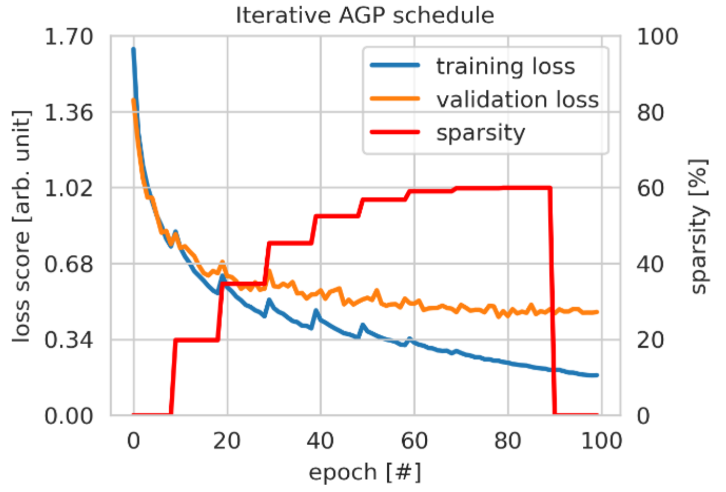
- Are used as an approximation to decide which structures/elements to remove.
  - **L-Norm**[2] based approximations
    - Higher magnitude or norm of elements/structures implies higher importance
  - **Gradient**[1] based approximates
    - Steeper backpropagation gradient implies higher learning activity
  - **Average percentage of Zeros**[3] (ApoZ) in activation tensors
    - Exploits sparsity in activation tensors introduced by ReLU activation functions
  - More recently: Approaches based on **explainable AI**[4]

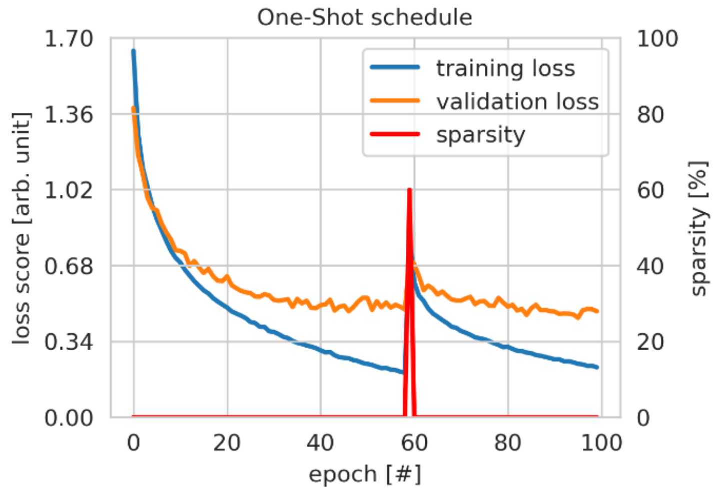
1. Han, Song, Jeff Pool, John Tran, und William J. Dally. “Learning both Weights and Connections for Efficient Neural Networks“. 2015.
2. Molchanov, Pavlo, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. “Pruning Convolutional Neural Networks for Resource Efficient Inference“. 2017.
3. Hu, Hengyuan, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. “Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures“. 2016.
4. Sabih, Muhammad, Frank Hannig, and Juergen Teich. “Utilizing explainable AI for quantization and pruning of deep neural networks.“ 2020.

Network Pruning

Pruning Schedule

- Defines **when** and **how often** pruning is applied during training
- **Iterative Pruning:**
  - Prune multiple times during training
  - Increase sparsity starting with a low value
  - Automated Gradual Pruning[1] (AGP) algorithm
- **One-Shot Pruning:**
  - Prune one time at the end of training
  - Enforce all sparsity at once





1. Zhu, Michael, and Suyog Gupta. “To prune, or not to prune: exploring the efficacy of pruning for model compression“. 2017.

Weight Quantization

Quantization Schema

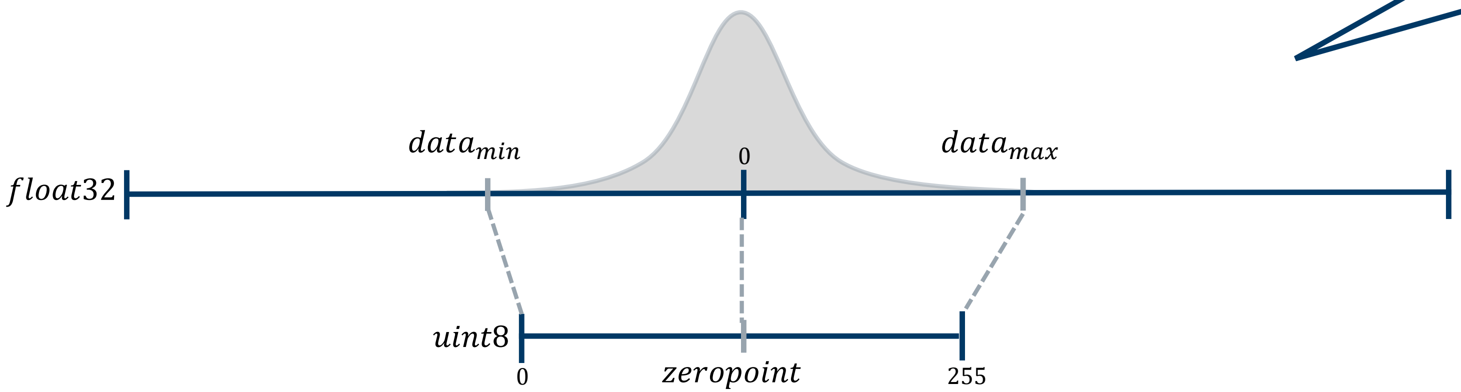
Instead of using high-precision floating-point arithmetic to store trained weights, map them to integer space instead

- Affine, linear mapping from **32-bit floating point** space to **8-bit unsigned integer** space using (trainable) **scale** and **zero point** parameters[1]
- Both weight and activations tensors can be quantized (i.e. partial and full quantization)

$$val_{uint8} = \text{clamp}\left(\left\lceil \frac{val_{fp32}}{scale} \right\rfloor + zeropoint\right)$$

$$scale = \frac{data_{max} - data_{min}}{255}, 0 \leq zeropoint \leq 255$$

Standardization of activation tensors is a great idea



1. see https://onnxruntime.ai/docs/performance/quantization.html

Weight Quantization

Quantization Strategy

Post Training Static Quantization[1]

- Perform Quantization once training has finished
- Use evaluation dataset from training to approximate zero points and scales

+ Easy to add into an existing training and deployment process
+ Inexpensive and fast to perform

- Quantization parameters are only approximated (using a set of sample inputs)
- Quantization not considered during training

Quantization Aware Training[2]

- Fake quantized trainable weights and activations during training
- Use quantized parameters during forward passes to emulate quantization

+ Network becomes more robust towards quantization
+ Better approximation of quantization parameters

- Training becomes more expensive
- Has to be integrated into training process

1. Krishnamoorthi, Raghuraman. „Quantizing deep convolutional networks for efficient inference: A whitepaper“. 2018.
2. Jacob, Benoit, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. „Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference“. 2017.

Microcontroller Deployment

Overview

Conversion to Intermediate Representation

- **Convert graph-based** (ONNX) model to **target-platform-based** representation
  - **Weight** tensors → **Data** segment (Flash utilization)
  - **Network** structure → “Function” **Code** segment (Flash utilization)
  - **Activation** tensors → **Heap** (SRAM consumption)

Memory Allocation

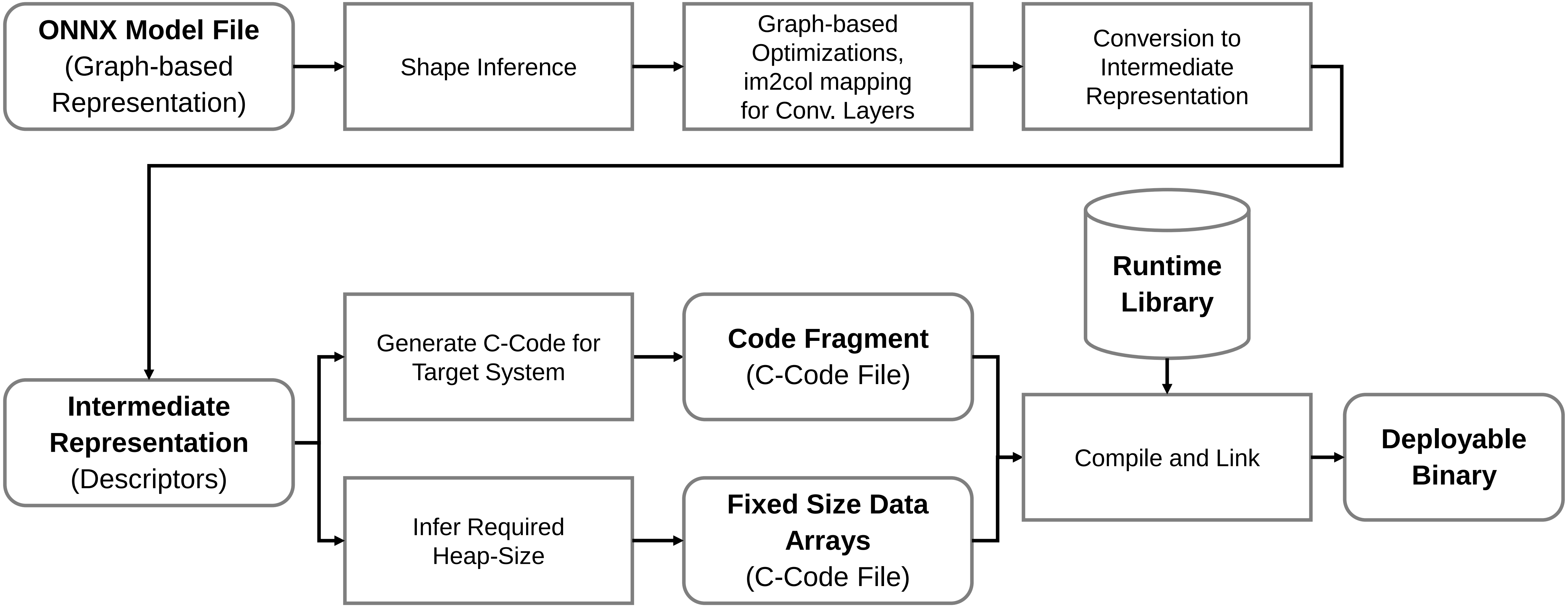
- **Allocate byte arrays** for both **weight and activation tensors**
- Dynamic memory allocation for activation tensors can be avoided by simulating offline
  → No malloc required at runtime

Code Generation + Compilation

- **Emit compilable C-code** representing the DNN (“ahead-of-time”)
- **Compile and link** with runtime library (optimized for each target platform)
- Implements common DNN layers based on the ONNX specification

Microcontroller Deployment

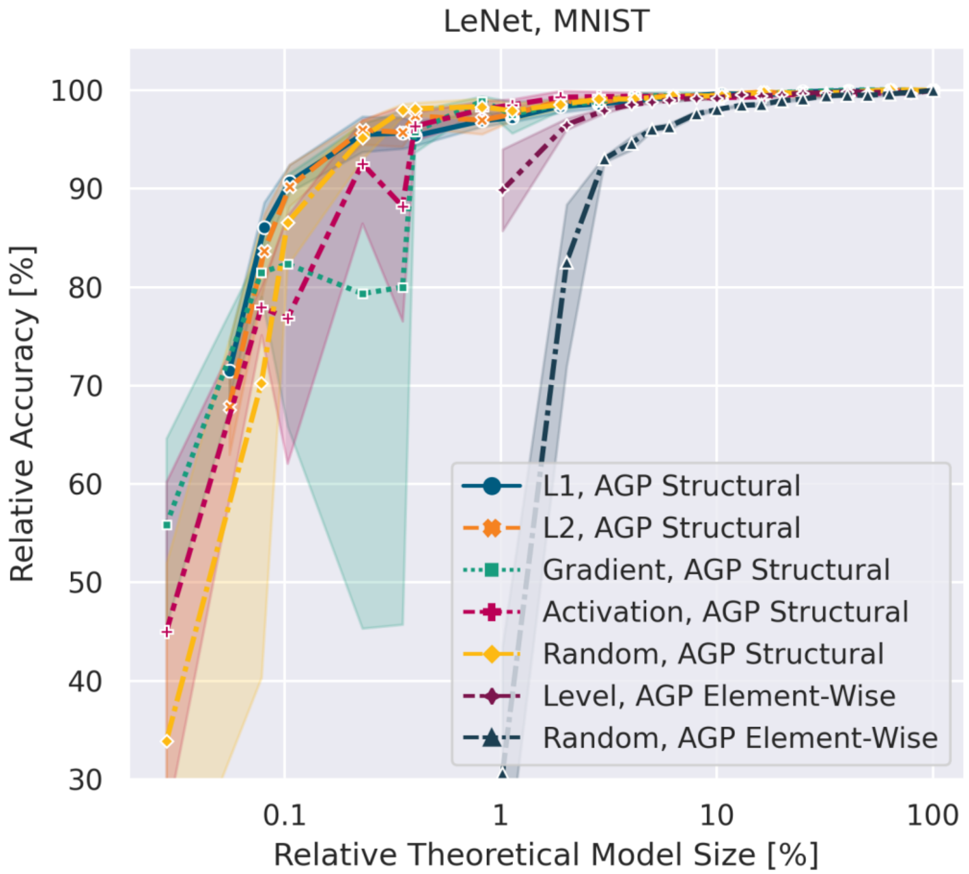
Deployment Pipeline

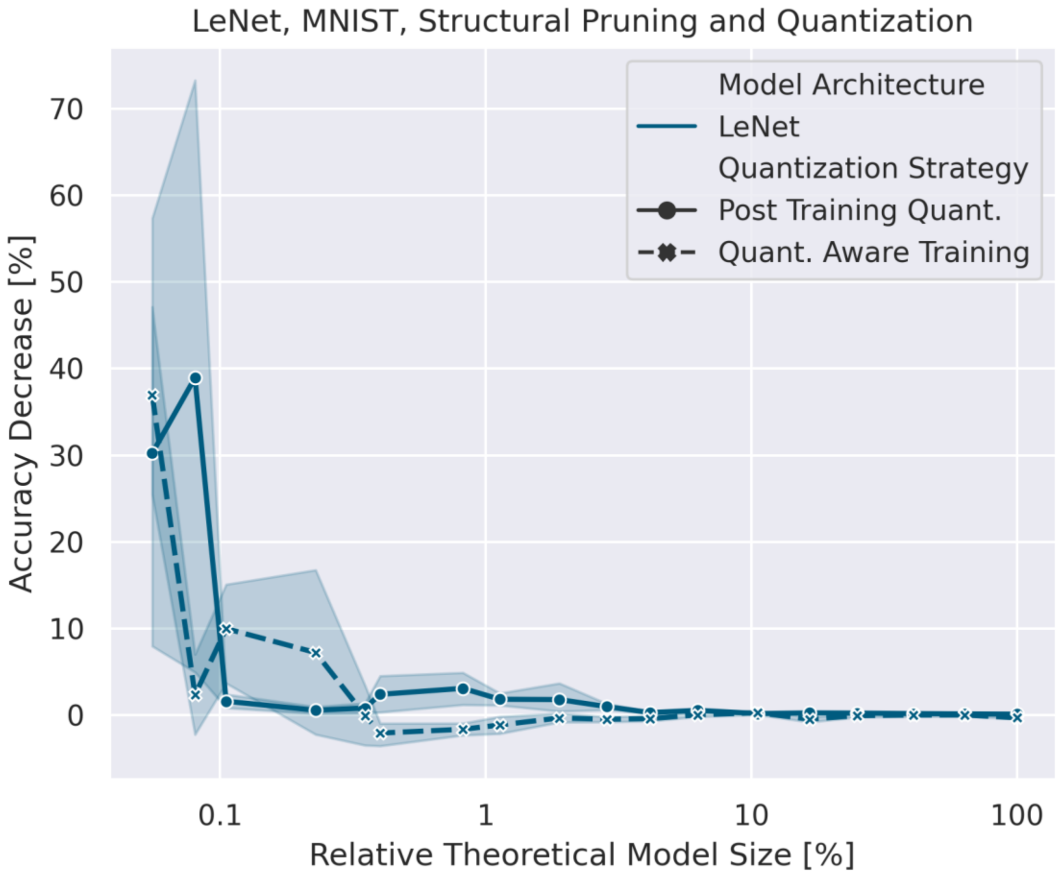


Evaluation

Evaluation

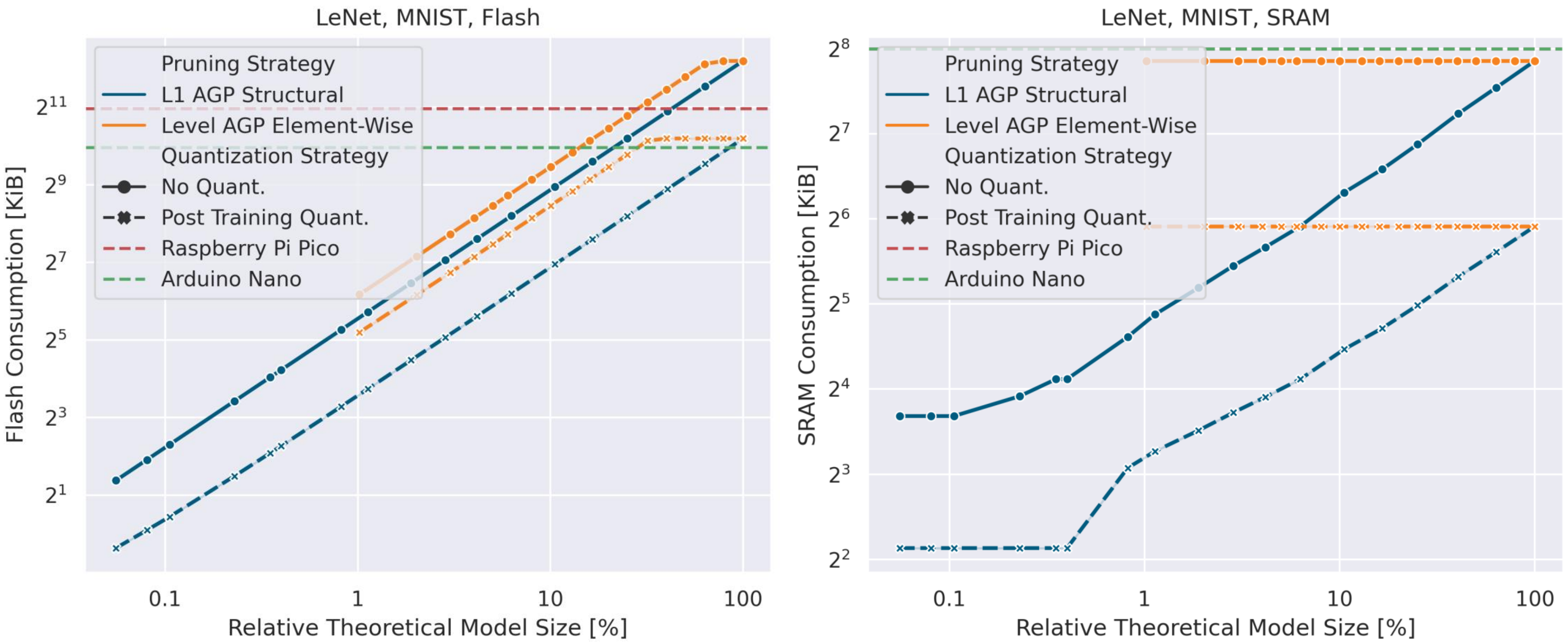
Pruning and Quantization of DNNs





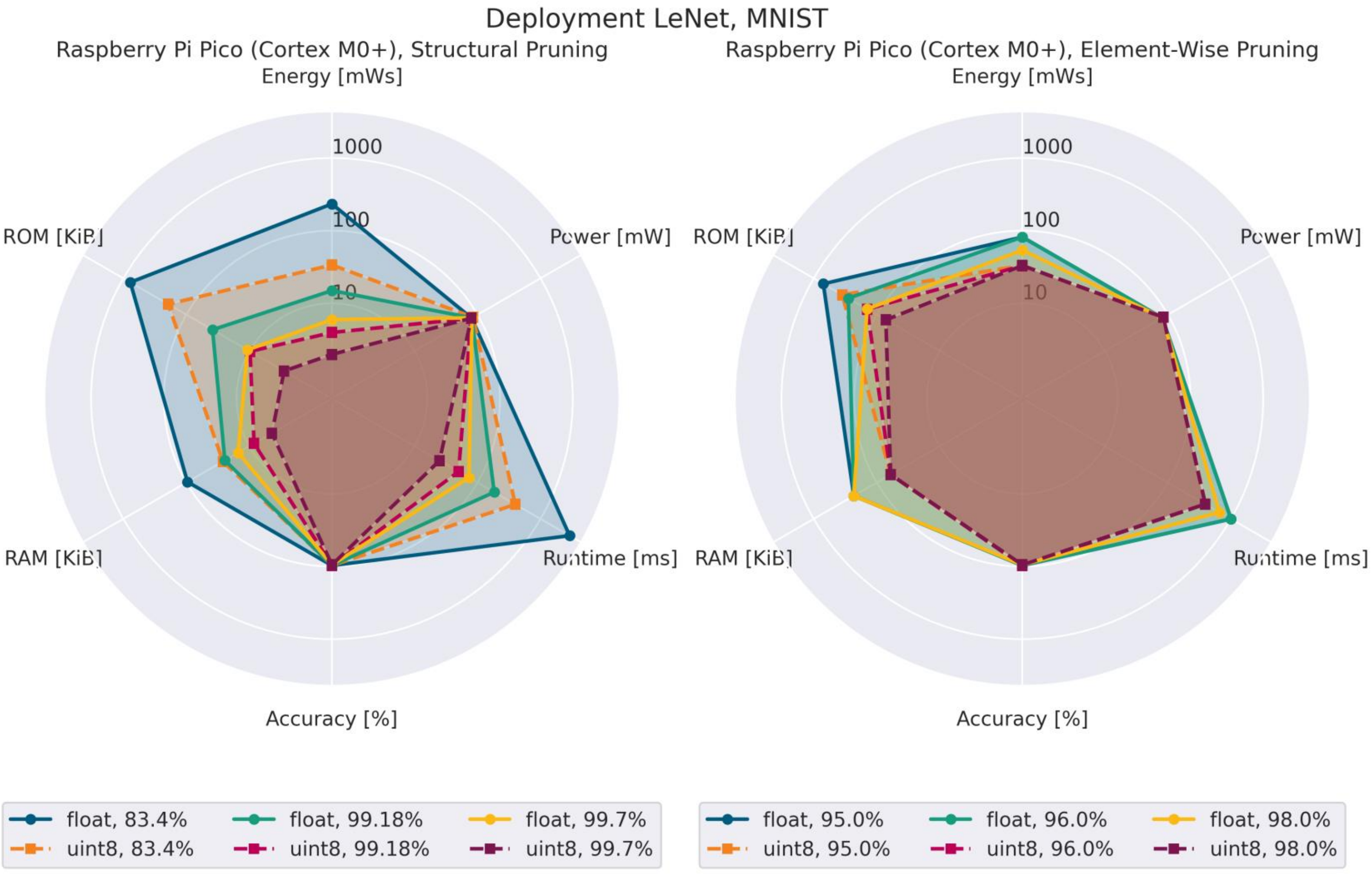
Evaluation

From Size Reduction to Memory Savings



Evaluation

Deployment on Microcontrollers



Conclusion

Conclusion

- **DNN Compression** and **Deployment Pipeline:**
  - Three stages: Network Pruning, Weight Quantization, Deployment
  - Target: Cortex-M processors
- **Conclusions/Recommendations:**
  - Pruning can be combined with quantization to save memory and latency.
    - Structural Pruning in combination with Post Training Static Quantization (PTSQ)
    - Quantization Aware Training is a good alternative in situations where PTSQ fails
  - Model compression combined with algorithmic and instruction set-based optimizations enables efficient deployment on microcontrollers
  - Most energy can be saved by decreasing latency, the smallest system that can meet the target requirements.

Thank you for your attention!

Contact: mark.deutel@fau.de