

Introduction to TinyML: Running Deep Learning Models on low-power Micro-Controllers

Austrian AI Factory Tutorial 2026

Daniel Mueller-Gritschneider, TU Wien

Presenters



Daniel Müller-Gritschneider
(TU Wien)



Philipp van Kempen
(TU München)

With support of

DOMINO Edge AI Flagship Project



With funding from the:



Federal Ministry
of Research, Technology
and Space

Goal of this Tutorial

Give you an introduction and getting started in the field of *tinyML*

Machine
Learning

ML Compilers

Embedded SW

Embedded HW

Machine
Learning

Who already worked with ML frameworks such as TensorFlow, PyTorch or similar?

ML Compilers

Who already worked with ML Toolchains such as TFLite, TVM, IREE, Infineon's ModusToolbox™ ML platform, ST X-CUBE-AI, Edge Impulse or similar?

Embedded SW

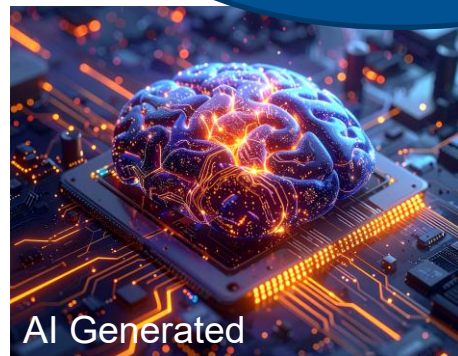
Who already programmed MCUs?

Embedded HW

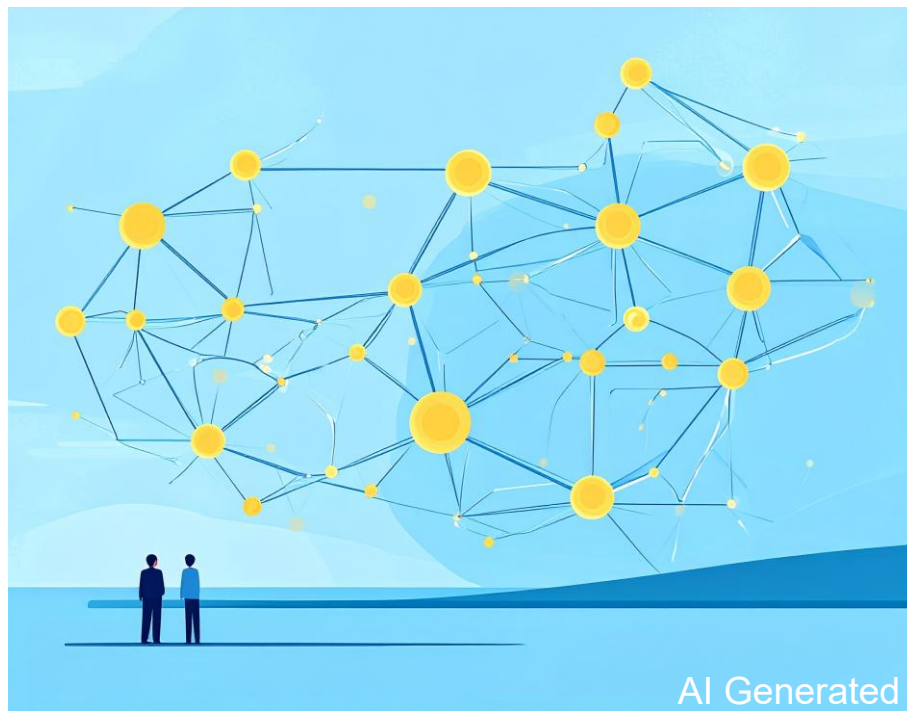
What to expect?

- 09:15 *Join in / Arrival @ AI Factory Austria AI:AT*
- 09:30 **From Data-driven Design to tinyML: Designing Machine Learning Models for Tiny Embedded Systems**
- 10:30 **Deploying Machine Learning Models on Micro-Controller Units (MCUs)**
- 11:10 *Break and Q&A*
- 11:25 **Hands on 1: Keyword Spotting on an MCU using TVM**
- 12:25 *Q&A and Wrap up*
- 12:30 *End of course*

Please feel free to
always ask questions
in between!



PART Ia: From Data-driven Design to tinyML



Data-driven Design – Typical Tasks

(Non)linear Regression

Classification

Object Recognition

Anomaly Detection

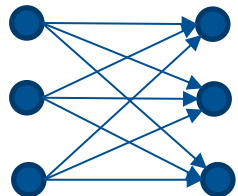
Text2Image Generation (GenAI)

Text2Text Generation (GenAI)

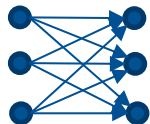
Coding Co-Pilot



ML Models are Heterogeneous



Large language models (LLMs) GPT-4, ChatGPT:
~1.7 trillion parameters – GenAI text2text generation



YOLO family – millions of parameters – Object recognition
e.g. for autonomous driving



Keyword Spotting KWS: 16k-300k parameters – Detects keyword in an audio stream,
e.g. for audio wakeup

Some ML models are small enough to be deployed
on Edge Devices (Edge AI) or even on Micro-Controller Units (tinyML)

Edge AI & tinyML

- Advantages:

- Uplink Bandwidth reduction
- Latency reduction
- Autonomy (smart devices)
- Privacy (Data stays on the device)
- Energy (no expensive data transfers)

- Challenges:

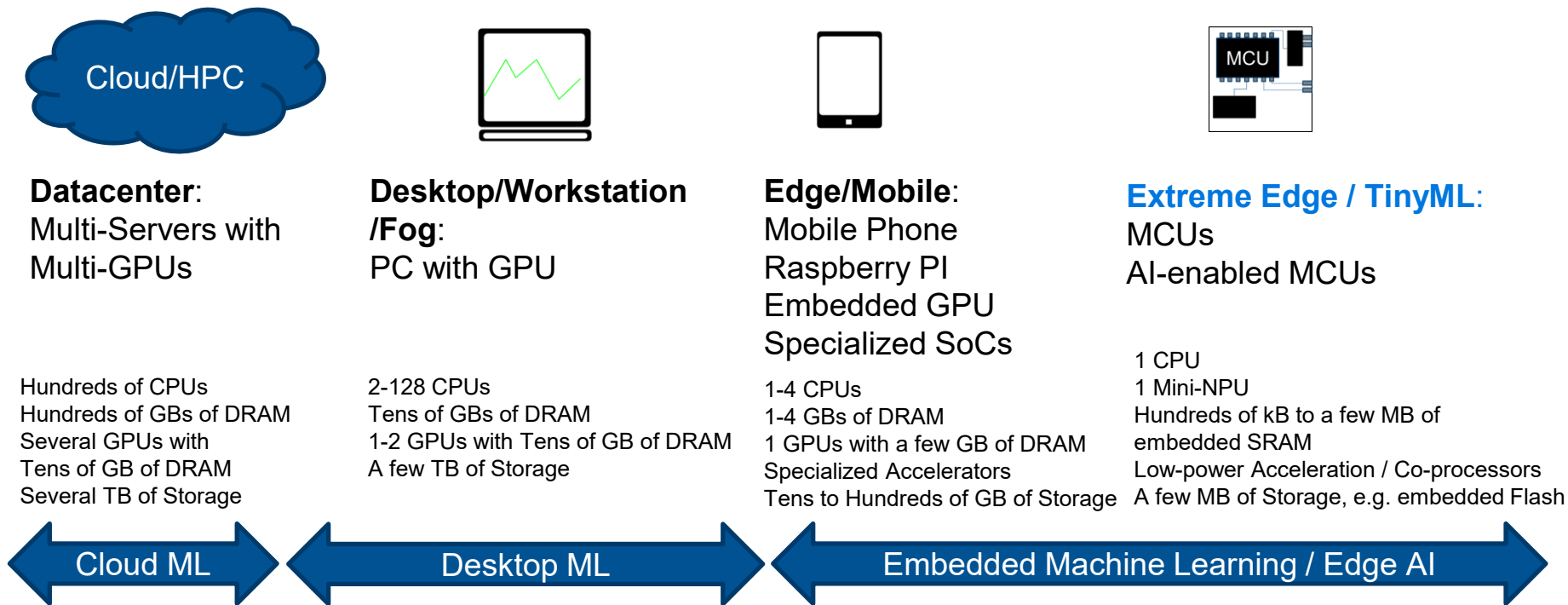
- Edge devices are resource-constrained
- Development of embedded ML applications requires a unique skillset

EdgeAI Applications Domains

- Smart Home
- Autonomous Robots
- Healthcare Monitoring
- Predictive Maintenance
- Smart Agriculture
- Surveillance



ML Platforms are Heterogeneous



Edge vs. Extreme Edge (TinyML) Platforms

Edge GP-GPU	AI-enabled MCU
Jetson Orin Nano*	STM32N6**
six ARM cores	Arm® Cortex®-M55
1024 CUDA cores	ST Neural-ART accelerator
8 GB RAM	4.2 MB RAM
SSD storage support	
15 Watt	200 mW = 0.2 W
67 TOPS	600 GOPS = 0.6 TOPS
4.5 TOPS/W	3 TOPS/W
\$249	\$12.74

Further AI-enabled MCUs

Infineon
PSOC™ Edge E84

TI
TMS320F28P55x

NXP
i.MX RT700

Nordic
Semiconductor
Thingy:91

* <https://www.geeky-gadgets.com/nvidia-jetson-orin-nano-mini-pc>

**05.03.2025 <https://community.st.com/t5/developer-news/stm32n6-highlights-from-the-stm32-summit/ba-p/766783>,
<https://estore.st.com/en/products/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus/stm32-high-performance-mcus/stm32n6-series.html>

TinyMLs – As many variants as there are sensors

- **Audio**
 - Keyword Spotting (KWS) / Audio Wakeup
- **Vision**
 - Video Wakeup (Face Detection)
 - Image Classification
- **Radar**
 - Gesture Recognition
- **Accelerometer**
 - Activity Detection
 - Health care monitoring

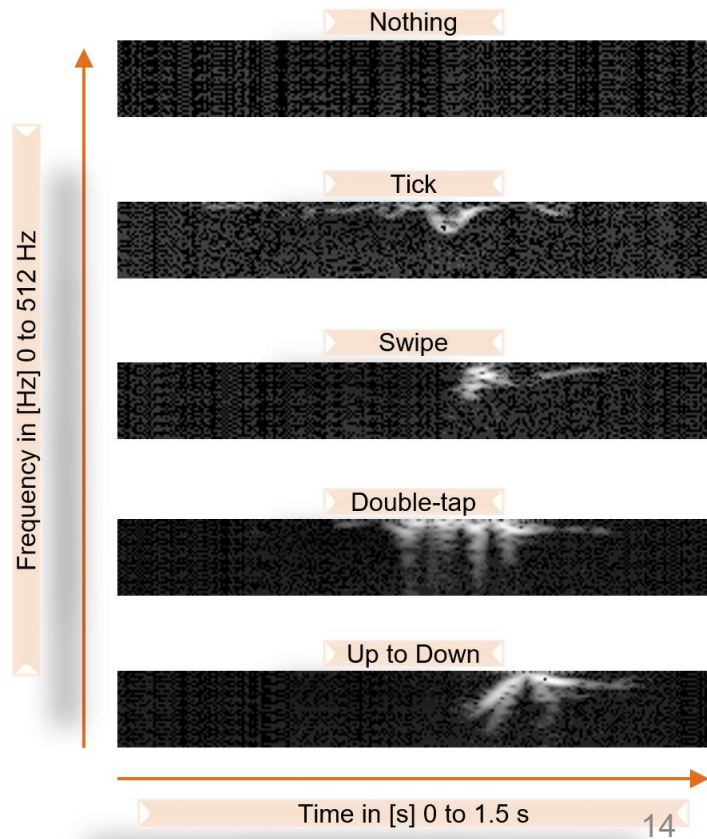
TinyML Example: Radar Gesture Recognition

Radar spectrograms for
different hand gestures [1]

Tiny platform: Infineon XMC1302:
32 MHz CPU,
32 kB Flash, 16 kB RAM

Nothing	20	0	0	0	0
Up – Down	0	19	1	0	0
Swipe	0	0	19	1	0
Tick	0	0	1	19	0
Double Tap	0	0	0	0	20
	Nothing	Up – Down	Swipe	Tick	Double Tap

[1] Pragathi Jayaram: *Real-Time Hand Gesture classification on a MCU with Continuous Wave Radar and a Convolutional Neural Network*, Master thesis 2021



PART Ib: Designing Machine Learning Models for Tiny Embedded Systems

Some ML Terminology

- **Deep Neural Network (DNN):** See next slide
- **ML Model:** A specific ANN for a particular application
- **Dataset:** Collection of data to train the DNN
- **Training:** Trainable parameters are tuned with the dataset
- **Inference:** A trained model is executed on an unknown new data to obtain a new label (e.g. classification result).
- **On-Device Training:** Training is done on an embedded device
- **Deployment:** A code is generated for an ANN that can be executed on the embedded device, usually only for inference.

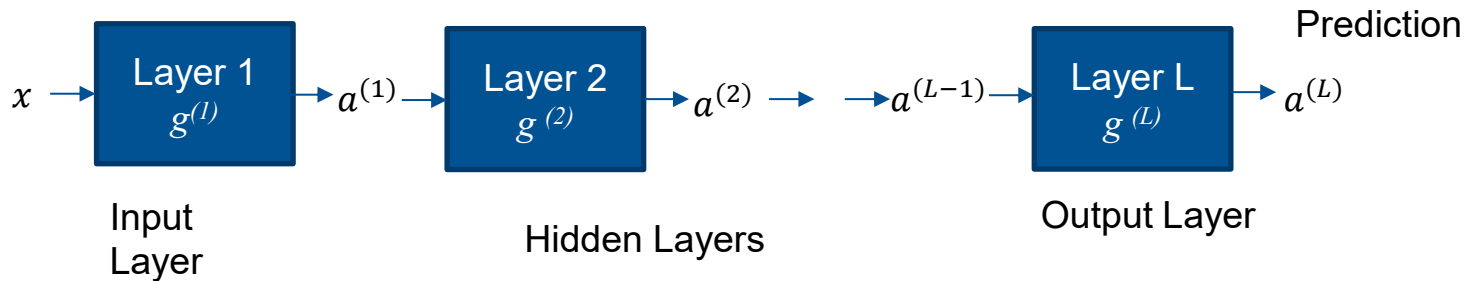
Neural Network Architectures

Layered computation:

$$a^{(l)} = g^{(l)}(a^{l-1})$$

$$a^{(L)} = g^{(L)}(\dots(g^{(2)}(g^{(1)}(x))))$$

Input



Forward pass:

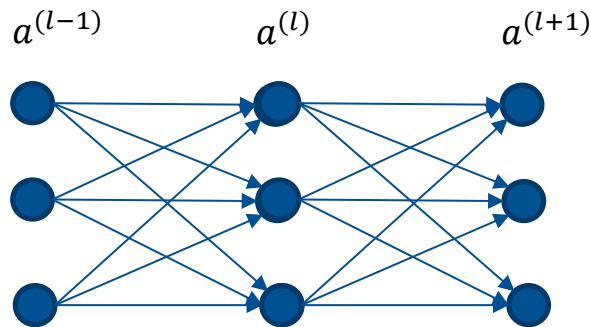
$$a^{(L)} = G(x)$$

Feed-Forward Deep Neural Network

Sequence of Dense/Fully Connected Layers:

- Trainable parameters: Weights: $W^{(l)}$, Bias: $b^{(l)}$
- Activations: $a^{(l)}$
- Activation function: $f^{(l)}$

$$a^{(l)} = g^{(l)}(a^{(l-1)}) = f^{(l)}(W^{(l)}a^{(l-1)} + b^{(l)})$$



$$a^{(L)} = f^{(L)}(W^{(L)}(\dots f^{(2)}(W^{(2)}f^{(1)}(W^{(1)}x + b^{(1)}) + b^{(2)})\dots) + b^{(L)})$$

Training Algorithms

Layered computation:

$$\begin{aligned}a^{(l)} &= g^{(l)}(\dots (g^{(2)}(g^{(1)}(x))) \\a^{(L)} &= f^{(L)}(W^{(L)}(\dots f^{(2)}(W^{(2)}f^{(1)}(W^{(1)}x + b^{(1)}) + b^{(2)})\dots) + b^{(L)})\end{aligned}$$

Training minimizes a Loss function $\mathbf{L}$ over a set of trainable parameters θ for a training dataset with inputs x_i that have known labels (predictions) y_i :

$$\mathbf{L}(a_i^{(L)}, y_i) = \mathbf{L}(G(x_i), y_i)$$

$$\begin{aligned}\min_{\theta} \quad & \sum_i \mathbf{L}(G(x_i), y_i) \\ \text{with } \theta = & [W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \dots, W^{(L)}, b^{(L)}]\end{aligned}$$

This is done using gradient based optimization applying chain rule for derivatives, which need to be computed from the last to the first layer (**backpropagation/backward pass**).

Flow - Model Generation Stage

Signals



Sensor

Collect Data

Database

Data Collection
System

Target Device &
Sensor

Preprocess Data

Data Set Collection & Labeling

Data Preparation & Preprocessing



Design a
ML Model

Neural Architecture
Search (NAS)

Design of ANN
Architecture

Preprocessed
Dataset

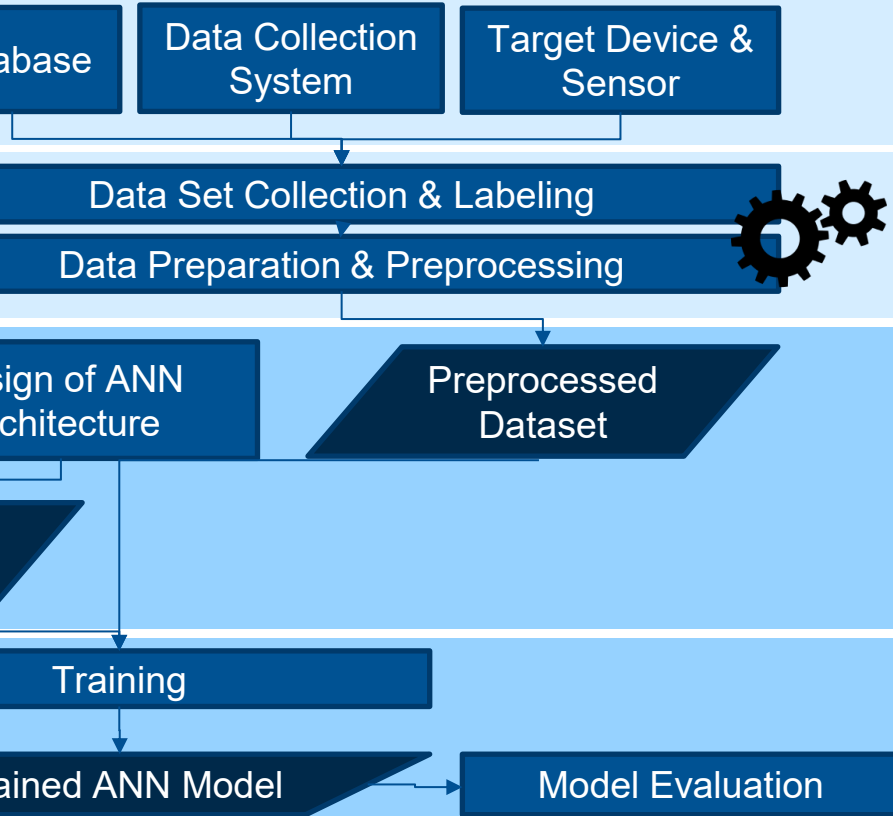
ANN Model
Architecture

Train the
ML Model

Training

Trained ANN Model

Model Evaluation



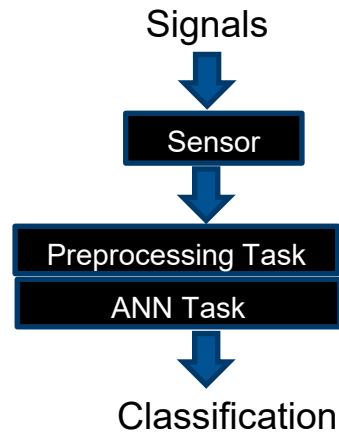
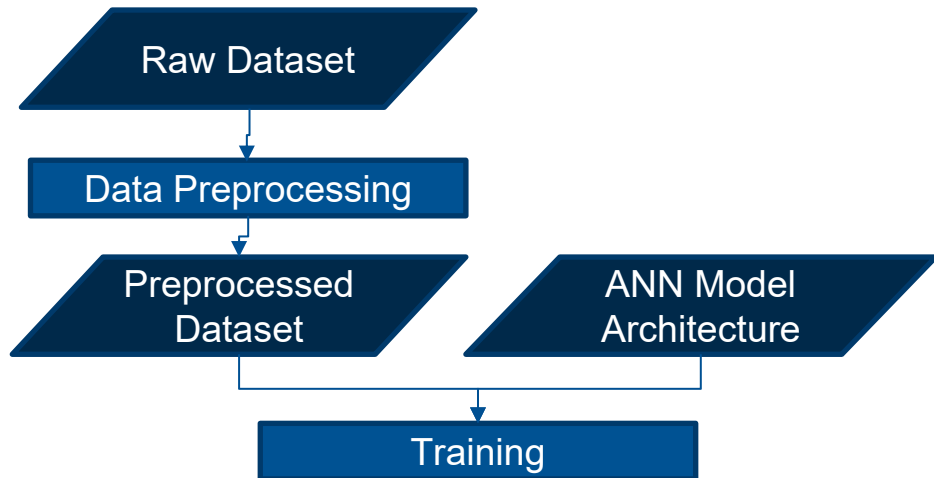
Data Set Collection & Labeling

Three sources:

- Existing Database
- Data Collection System
 - Specialized System with sensors to collect the real-world data
 - System that can produce synthetic data (e.g. by simulation)
- Embedded System with Sensor (Use the Embedded ML System itself)
- Challenges:
 - Data mismatch between collected and used sensor, e.g. different value range, resolution, missing noise and environmental conditions
 - Storage for collected data in the embedded system
 - Labels might not be automatically obtained

Data Preparation & Preprocessing

- Raw data stream from the sensor is often pre-processed before being given to a machine learning model (e.g. an ANN)
- Often DSP-type algorithms, e.g. spectral analysis (Filters, FFT, MFCC)
- Consideration: Must also run on-device for sensor data



Typical pre-processing steps

- Resampling
- Window selection
- Normalization
- Filtering / Noise reduction
- Domain Change

Considerations for ML Model Architecture

- Design of the ANN model architecture must consider target system
 - ROM/RAM Memory resources (weights, activations)
 - Computational power
 - Supported datatypes: FP / integer width
 - Acceleration features (type of layers, layer configurations)
- Design of the ANN model architecture must consider application demands
 - Input resolution
 - Performance: Inferences / second
 - Model accuracy

Layer Selection

Fully connected:

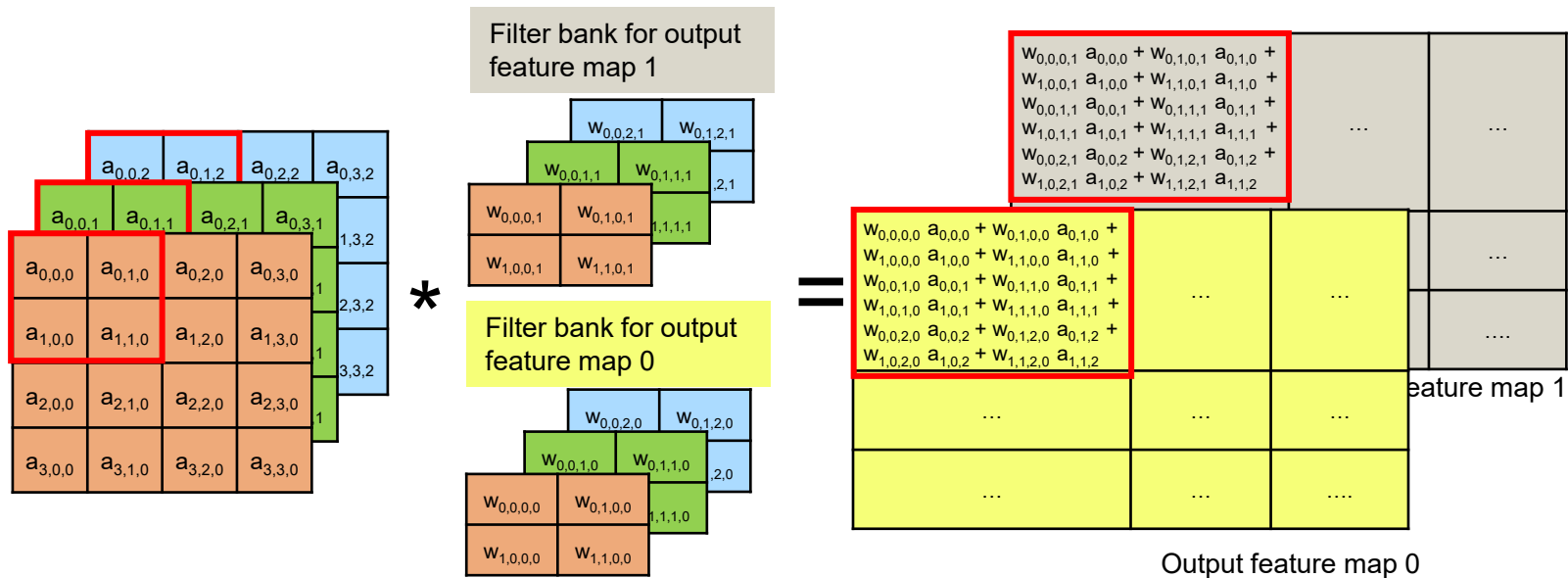
- Matrix Vector multiplication: High number of MACs
- High number of weights (ROM)

Convolutions:

- 4D Filter Bank (Memory depends on Filter Sizes)
- Convolution operation: High number of MACs
- Special Convolutions to bring down memory, MAC demand

Convolution

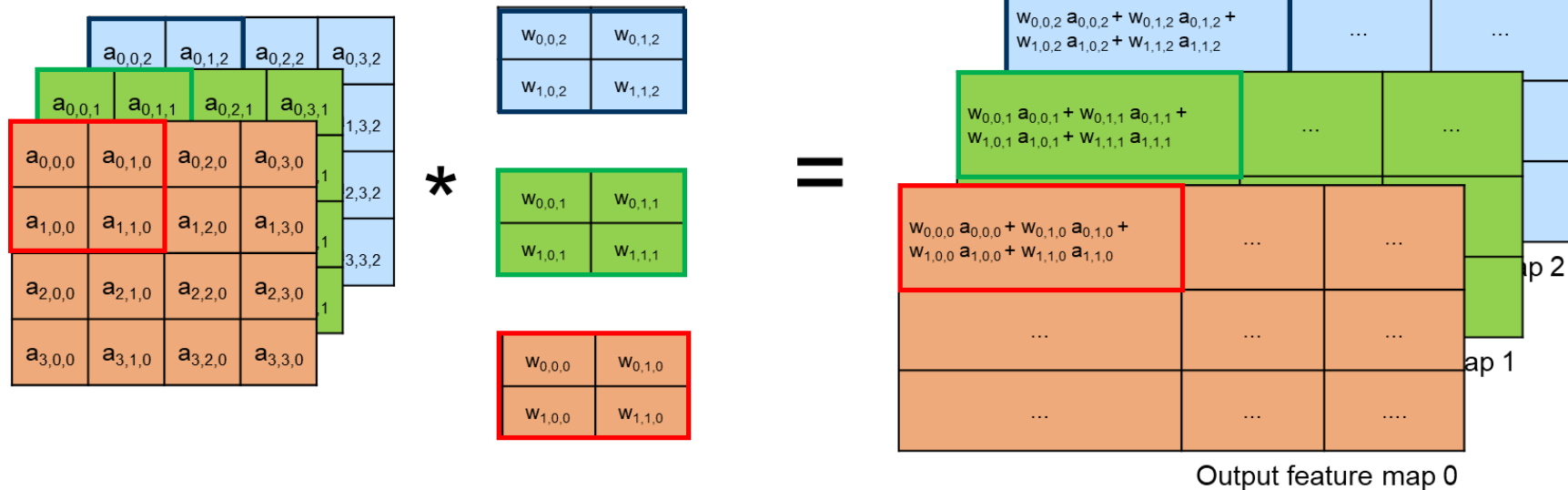
Sliding window over input feature maps (example red and green window position)



Depthwise Convolution

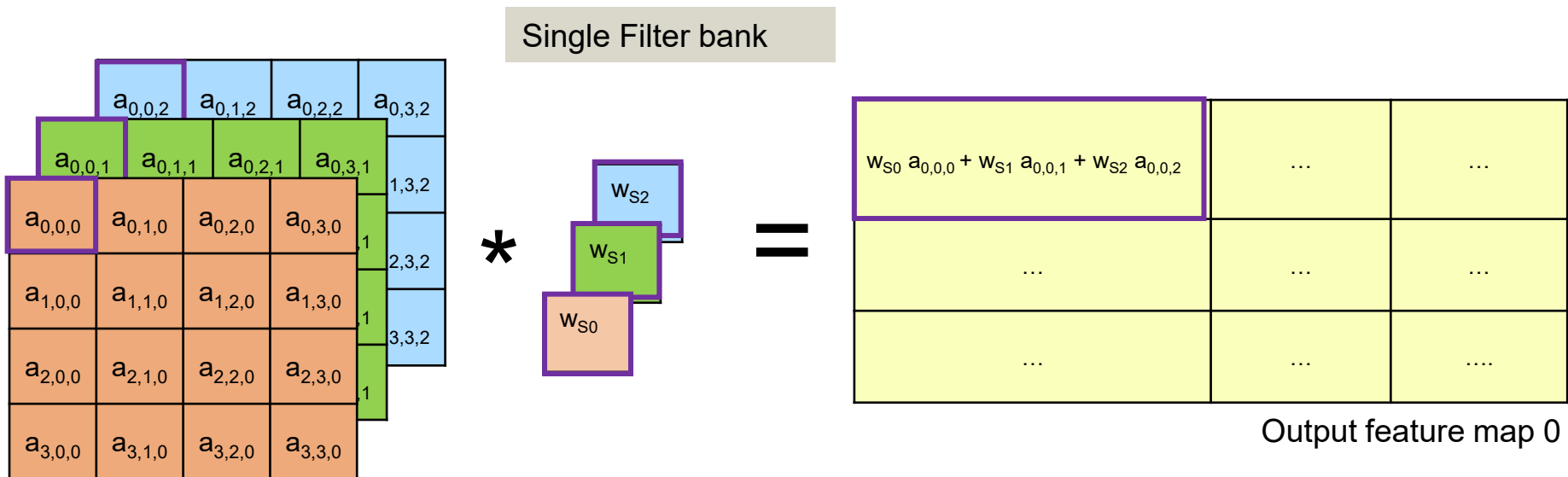
Single filter for each input channel (no sum over channels)

Single bank of filters applied individually per channel



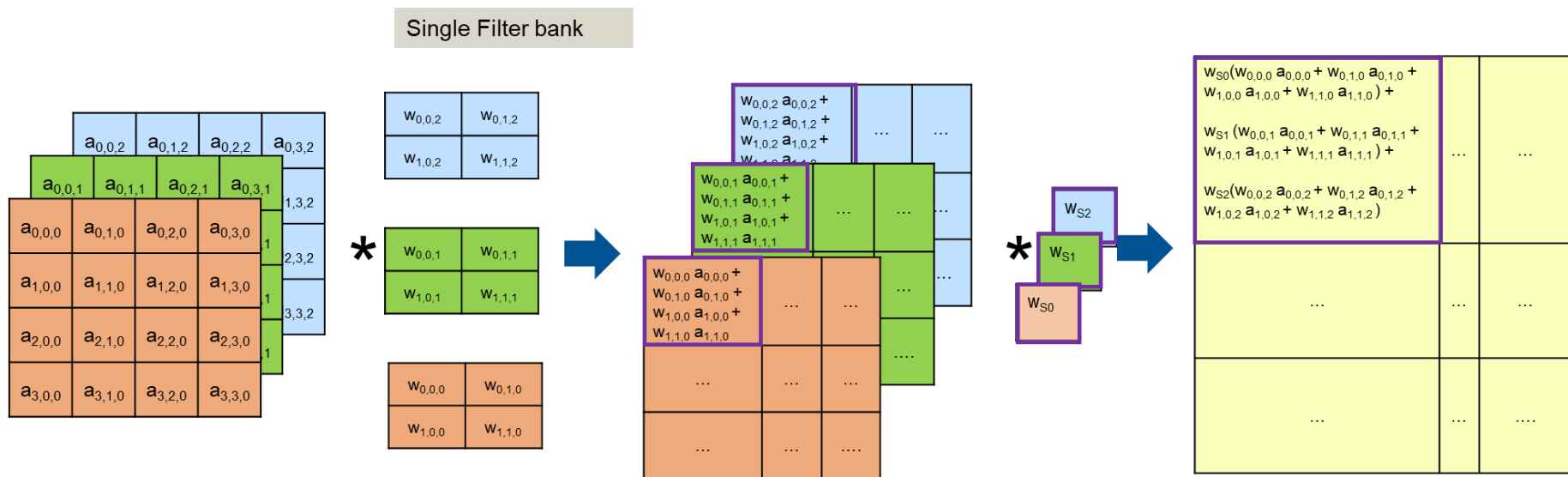
Pointwise Convolution

Filter Size 1x1: Only sum over channels



Depthwise Separable Convolution

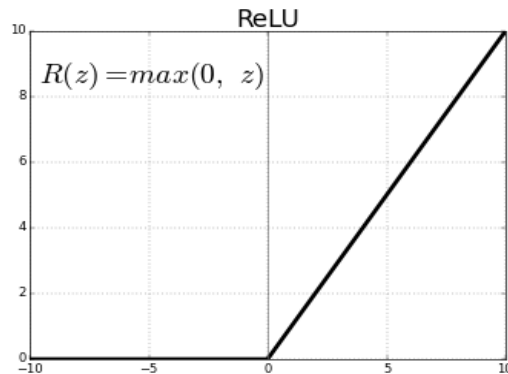
Combination of Depthwise and Pointwise Convolution



Activation Functions (Hidden Layers)

- Nonlinear functions hard to compute in HW (e.g. tanh)
 - Efficient algorithms
 - Piecewise linear approximations
- Many activations as output from hidden layers (activation function must be executed very often)
 - Often simple activation functions e.g. ReLU

```
muriscv_nn_status muriscv_nn_relu_q7(q7_t *data, const uint16_t size) {  
    for (uint16_t i = 0; i < size; i++) {  
        if (data[i] < 0) data[i] = 0;  
    }  
    return MURISCV_NN_MATH_SUCCESS;  
}
```



Activation Functions (Final Layer)

Activation function of final layer often depends on task

- **Regression:** Depends on regression problem
- **Binary Classification:** Sigmoid function maps any input to an output ranging from **0 to 1**.
- **Multi-class Classification:** Softmax converts a vector of real values to a vector of categorical/class probabilities. The elements of the output are in range $(0, 1)$ and sum to 1.

Model Types

- Feed-forward DNN: Several dense layer
- CNN: Combination of Conv layers + Pooling for feature extraction followed by Dense Layers for classification
- Auto-Encoder: Encoder-Decoder Architecture
- Recursive Neural Network (RNN): Feedback paths inside the NN (makes use of past information)
- Transformers: Special Attention Layers
- Many special model types with special layers: ResNET, ...

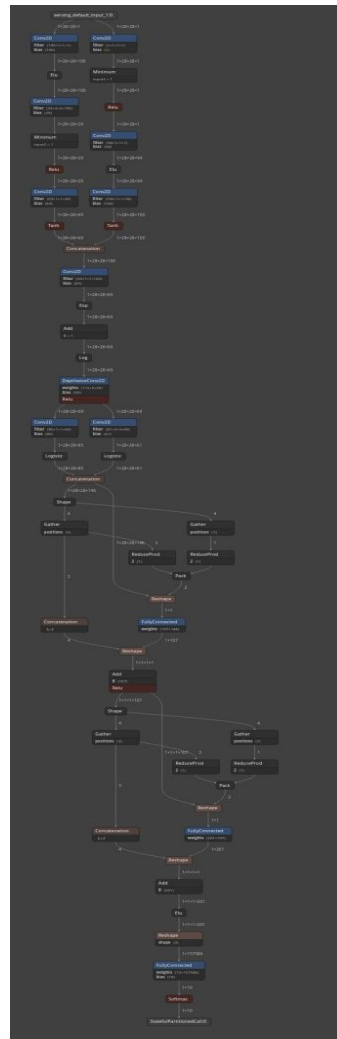
Model Definition

ML Frameworks: PyTorch / Tensor Flow (Keras)

```
# Step 1: Define the keras model
model1 = tf.keras.Sequential()
model1.add(tf.keras.layers.Dense(10, activation='relu', input_shape=(2,)))
model1.add(tf.keras.layers.Dense(20, activation='relu'))
model1.add(tf.keras.layers.Dense(20, activation='relu'))
# model1.add(tf.keras.layers.Dense(3, activation='softplus'))
model1.add(tf.keras.layers.Dense(3, activation='softmax'))
```

NAS / AutoML

- Neural Architecture Search (NAS)
 - Algorithms that systematically explore different ANN model architectures in an automatic way
 - Computationally very expensive (training of many candidates to evaluate the accuracy)
- AutoML
 - Toolflows that try to automate full process of data-driven development



Training

Training of the ANN model is done on a powerful machine (GPU)

Trained model is deployed on the embedded device

Embedded device executes the trained model (inference task)

Training:

- Selection of the hyperparameters
- Optimization of the trainable parameters of the ANN model
- Using usually a backpropagation algorithm



Model Evaluation

- **Model Training Target:**

Minimize error on validation set with the aid of the training set

- Make the error for the samples in the training set small
- Make the gap between the training and validation error small

- **Model Evaluation Target:** Does the model perform well for the given embedded task?

Confusion Matrix (Binary Classification)

Binary classification with two classes 0/1 (positive/negative):

	NN predicts class 0 (positive)	NN predicts class 1 (negative)
Labeled class 0 (positive)	Nr. of true positives (<i>tp</i>)	Nr. of false negatives (<i>fn</i>)
Labeled class 1 (negative)	Nr. of false positives (<i>fp</i>)	Nr. of true negatives (<i>tn</i>)

Model Evaluation for Binary Classification

Accuracy:

$$a = \frac{tp+tn}{tp+tn+fp+fn}$$

How many samples were classified correctly?

Precision:

$$p = \frac{tp}{tp+fp}$$

How many of the samples classified as positives were really positive?

Recall:

$$r = \frac{tp}{tp+fn}$$

How many of the positive samples were found over the total amount of positive samples?

F1 score:

$$F1 = 2 \frac{p \cdot r}{p+r}$$

Combined value for precision and recall.

Confusion matrix

Prediction

Positive Negative

**Ground
Truth**

Positive

tp

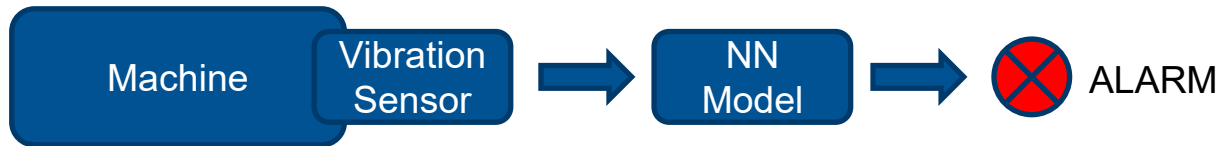
fn

Negative

fp

tn

Case Study: Anomaly Detection (1/6)



Positive (Alarm): Machine has an anomaly: Conduct predictive maintenance

Negative: Machine has no anomaly

Test Set:

- Negative samples with no anomaly: 1000
- Positive samples with anomaly: 20

We have three trained NN candidates: **NN1**, **NN2** and **NN3**

Case Study: Anomaly Detection (2/6)

NN1 prediction	
	Positive Negative
Positive	1 19
Negative	2 998

Accuracy: $a = \frac{1+998}{1+998+19+2} = 0,98$

Precision: $p = \frac{1}{1+2} = 0,33$

Recall: $r = \frac{1}{1+19} = 0,05$

F1 score: $F1 = 2 \frac{0,05 \cdot 0,33}{0,05+0,33} = 0,09$

NN2 prediction	
	Positive Negative
Positive	19 1
Negative	200 800

Accuracy: $a = \frac{19+800}{19+800+1+200} = 0,80$

Precision: $p = \frac{19}{19+200} = 0,09$

Recall: $r = \frac{19}{18+2} = 0,95$

F1 score: $F1 = 2 \frac{0,95 \cdot 0,09}{0,95+0,09} = 0,16$

Case Study: Anomaly Detection (3/6)

Accuracy of **NN1** and **NN2** has little meaning (high for both NNs)

NN1:

Medium precision: Many alarms are false

Low recall: Most anomalies are overlooked

➡ Low F1 score

NN2:

Low precision: Most alarms are false

High recall: Most anomalies are detected

➡ Low F1 score

Case Study: Anomaly Detection (4/6)

NN3 prediction	
	Positive Negative
Positive	18 2
Negative	40 960

Accuracy: $a = \frac{18+960}{18+960+40+2} = 0,96$

Precision: $p = \frac{18}{18+40} = 0,31$

Recall: $r = \frac{18}{18+2} = 0,9$

F1 score: $F1 = 2 \frac{0,9 \cdot 0,31}{0,9+0,31} = 0,46$

Accuracy of NN3 lower than NN1

NN3:

Average precision:

Many alarms are still false

High recall:

Most anomalies are detected

➡ Average F1 score

Is NN3 suitable for the task?

Case Study: Anomaly Detection (5/6)

- On average one anomaly after 30 days of machine operation
- One monitoring sample from vibration sensor per minute
- On average: $30 * 24 * 60$ negative samples : 43.200 negative samples

Negative samples: $tn = 960$ $fp = 40$

- *Probability of false alarm for false positive sample: $P(fp) = 40/1000$*

„Pi-Mal-Daumen“ (Pi-times-thumb - German saying for rough estimation):

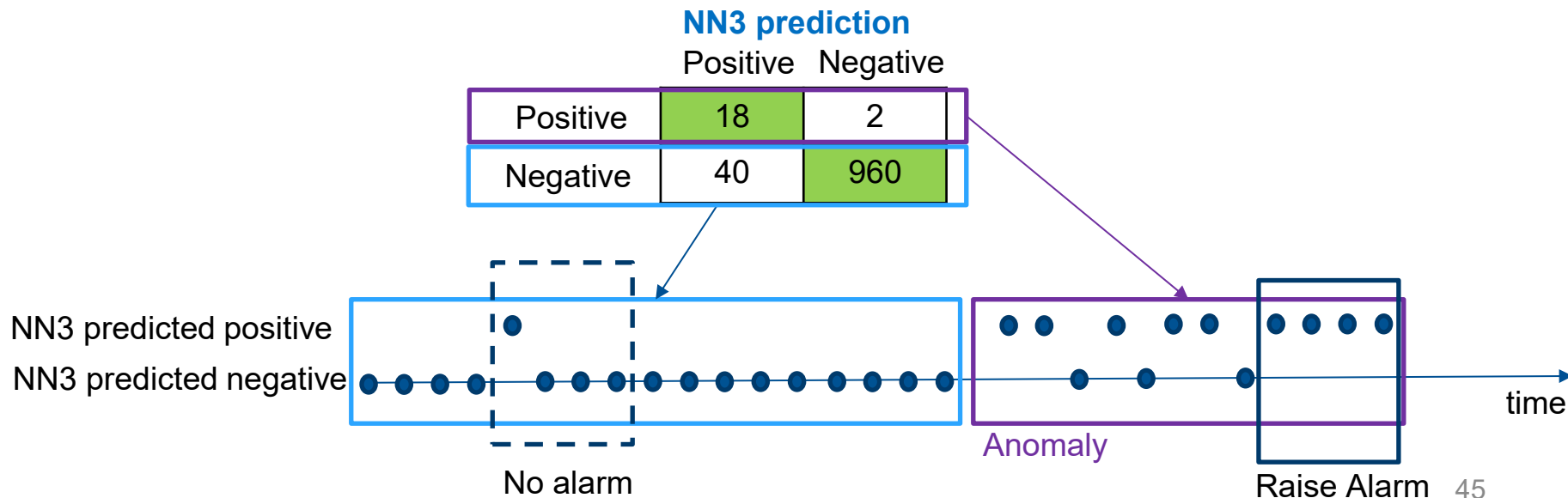
- On average we have $(40/1000) * 43.200 = 40 * 43.2 = 1.728$ false alarms within the 30 days

Solution?

Case Study: Anomaly Detection (6/6)

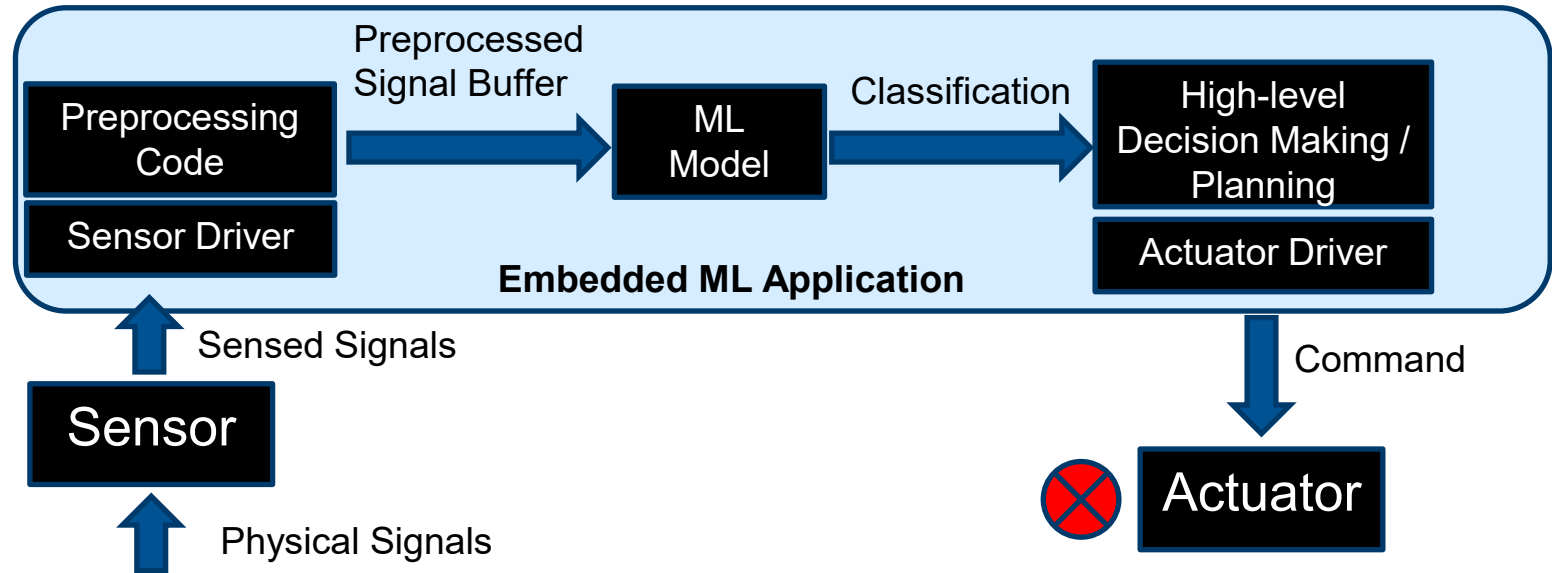
Possible solution:

- Use temporal correlation: Several samples measured during anomaly
- High-level decision logic: to raise alarm, many consecutive samples needs to be positive (with high probability).

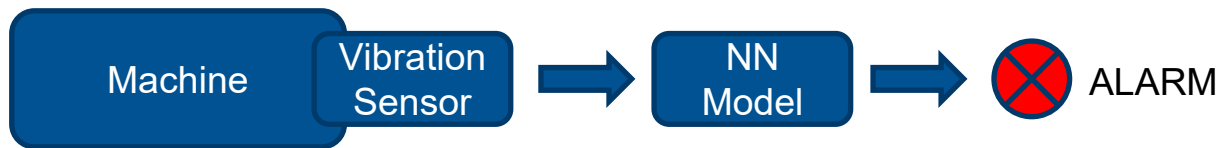


ML Model is Embedded into Application

Typical SW Structure of an Embedded Machine Learning Application
Needs to be considered for training ML model



Case Study 2: Anomaly Detection (1/2)



Positive (Alarm): Machine has an anomaly: Conduct predictive maintenance
Negative: Machine has no anomaly

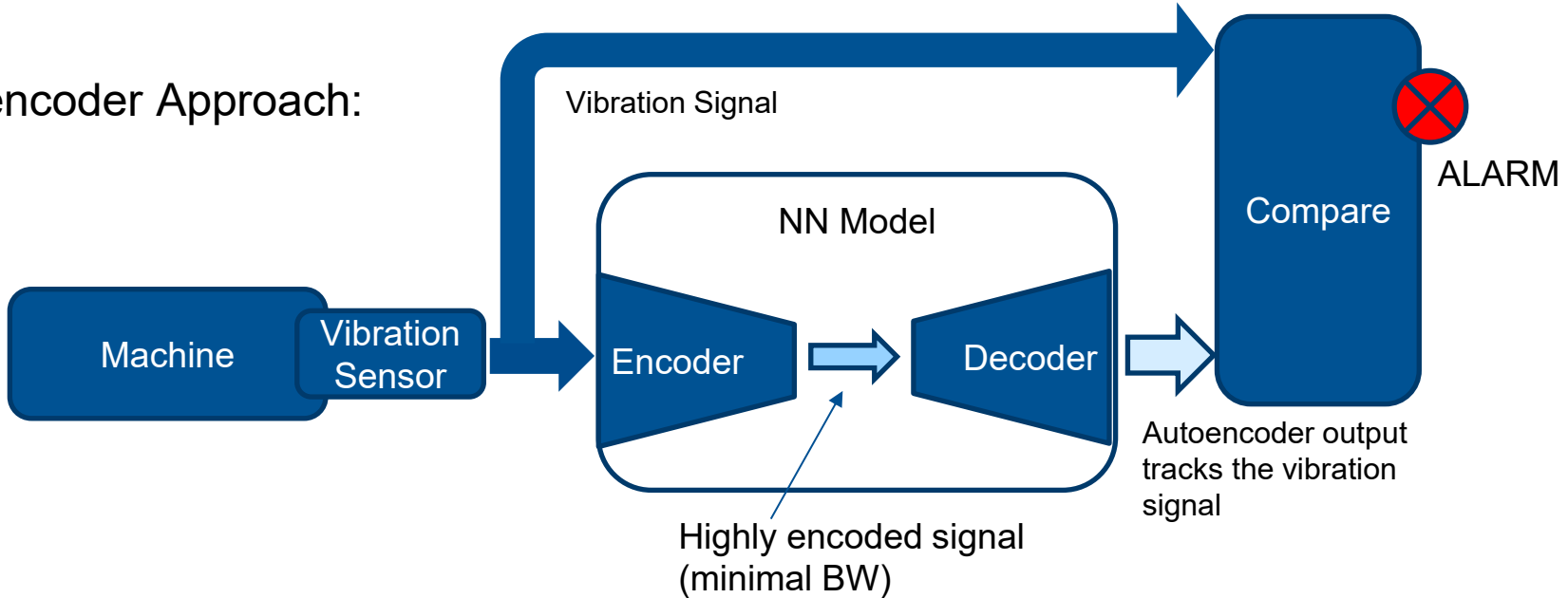
Test Set:

- Negative samples with no anomaly: 1000
- Positive samples with anomaly: **NONE**

Can we still solve the task in a data-driven way?

Case Study 2: Anomaly Detection (2/2)

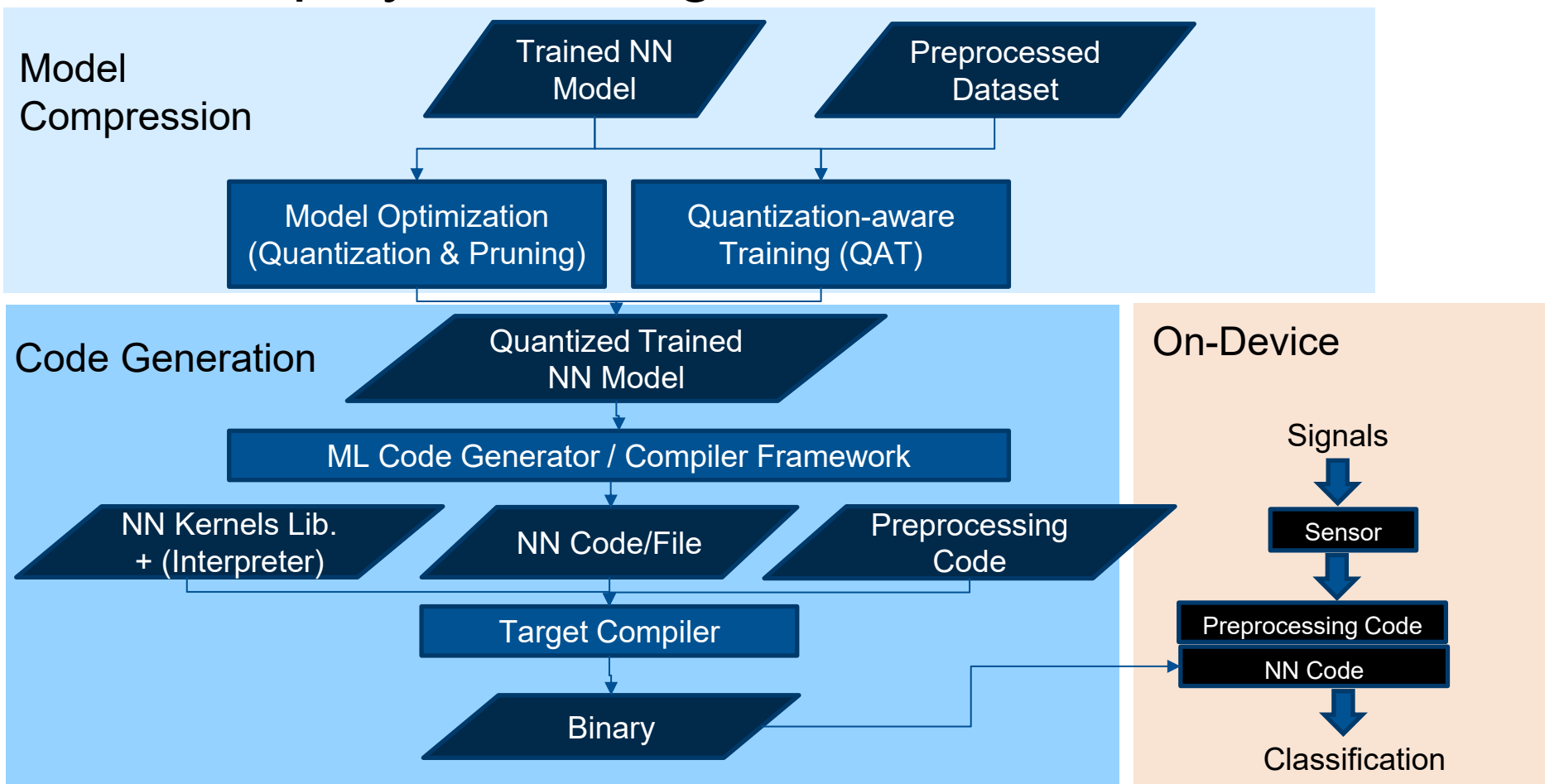
Autoencoder Approach:



- Autoencoder trained so that its output follows the vibration signal based on encoded signal for normal operation.
- Anomaly (unseen signal): Autoencoder output cannot follow vibration signal.

PART II: Deploying Machine Learning Models on Tiny Embedded Systems

Flow – Deployment Stage



Quantization

- Model are usually trained with floating-point (FP) precision (float, double).
- Inference (execution of trained model on embedded device)
 - Full precision (FP) computation (multiplication, addition) expensive
 - HW Floating Point Units expensive (area, energy)
- For inference the model is transferred to a quantized variant
- Integer computations (less expensive)
- Many challenges: Rounding, Overflow, Rescaling, Shifting
- Simple Example (8bit integer [-128 ... 127]):

$$z^l = \begin{bmatrix} 1.4 & 150.5 \\ 8.3 & 2.6 \end{bmatrix} a^{l-1} + \begin{bmatrix} 3.7 \\ 2.4 \end{bmatrix} \quad \longrightarrow \quad z^l = \begin{bmatrix} 1 & 127 \\ 8 & 3 \end{bmatrix} a^{l-1} + \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

Quantization

Quantization Formats

Many possible formats:

- **Integer formats** for weights/activations usually given by bit-width: x-bit
- On many embedded processors (byte-type quantization simplest 8bit, 16bit)
 - Byte /int8 (8bit) quantization range: [-128 ... 127]
 - Use of SIMD instructions
 - Accumulation variables usually use larger size
- Sub-byte integer quantization: < 8bit
 - Binary quantization w in $\{0,1\}$
 - Ternary quantization w in $\{-1,0,1\}$
- Reduced-precision floating-point (many specialized formats)

Pruning

Unstructured pruning: Small weight values are set to zero

- Skip computation with zero values (might require additional logic in program)
- Simple example:

$$z^l = \begin{bmatrix} 155 & 1 & 8 \\ 17 & 38 & 234 \\ 5 & 12 & 3 \end{bmatrix} a^{l-1} + \begin{bmatrix} 66 \\ 7 \\ 7 \end{bmatrix} \xrightarrow{\text{Unstructured Pruning}} z^l = \begin{bmatrix} 155 & 0 & 8 \\ 17 & 38 & 234 \\ 0 & 12 & 0 \end{bmatrix} a^{l-1} + \begin{bmatrix} 66 \\ 0 \\ 7 \end{bmatrix}$$

Structured pruning: A column, row, kernel is removed from the operator

- Operator is modified
- Example:

$$z^l = \begin{bmatrix} 155 & 1 & 8 \\ 17 & 38 & 234 \\ 5 & 12 & 3 \end{bmatrix} a^{l-1} + \begin{bmatrix} 66 \\ 7 \\ 7 \end{bmatrix} \xrightarrow{\text{Structured Pruning}} z^l = \begin{bmatrix} 155 & 1 & 8 \\ 17 & 38 & 234 \\ 0 & 0 & 0 \end{bmatrix} a^{l-1} + \begin{bmatrix} 66 \\ 7 \\ 0 \end{bmatrix}$$

Structured Pruning

$$\xrightarrow{\text{Structured Pruning}} z^{*l} = \begin{bmatrix} 155 & 1 & 8 \\ 17 & 38 & 234 \end{bmatrix} a_{53}^{l-1} + \begin{bmatrix} 66 \\ 7 \end{bmatrix}$$

Quantization-aware Training

- Pruning and quantization usually lead to accuracy loss
- Quantization-aware training:
 - Tries to retrain the network for minimizing accuracy loss
 - Dataset needs to be available
- Example algorithm idea:
 1. Train network with full precision
 2. Quantize and fix some weights
 3. Retrain other weights (might shift due to quantized weights to regain accuracy)
 4. Goto 2. until all weights quantized

Tasks of Code Generation

Kernel Generation and Optimization:

- Generate kernel codes that implement the computation of a layer
- Optimize kernels (e.g. fusing layers) for efficient execution on the target HW

Storage for read only data:

- Decide on format how to store weight tensors
- Plan memory to store weight tensors and layer configurations

Storage for read-write data:

- Decide on format how to store activation tensors
- Plan memory to store the activation tensors (memory reuse)

Model Graph Generation / Runtime Setup:

- Layer Scheduling
- Generate code or include runtime that executes the model graph (calls layer kernels on the tensors)

Convolution Kernel (in C from CMSIS-NN)

```
#else
(void)bufferA;
/* Run the following code as reference implementation for Cortex-M0 and Cortex-M3 */
int i, j, k, l, m, n;
int conv_out;
int in_row, in_col;

if (ch_im_in % 4 != 0 || ch_im_out % 2 != 0)
{
    /* check if the input dimension meets the constraints */
    return ARM_CMSIS_NN_ARG_ERROR;
}

for (i = 0; i < ch_im_out; i++)
{
    for (j = 0; j < dim_im_out; j++)
    {
        for (k = 0; k < dim_im_out; k++)
        {
            conv_out = (bias[i] << bias_shift) + NN_ROUND(out_shift);
            for (m = 0; m < dim_kernel; m++)
            {
                for (n = 0; n < dim_kernel; n++)
                {
                    // if-for implementation
                    in_row = stride * j + m - padding;
                    in_col = stride * k + n - padding;
                    if (in_row >= 0 && in_col >= 0 && in_row < dim_im_in && in_col < dim_im_in)
                    {
                        for (l = 0; l < ch_im_in; l++)
                        {
                            conv_out += Im_in[(in_row * dim_im_in + in_col) * ch_im_in + l] *
                                wt[i * ch_im_in * dim_kernel * dim_kernel + (m * dim_kernel + n) * ch_im_in + l];
                        }
                    }
                }
            }
            Im_out[i + (j * dim_im_out + k) * ch_im_out] = (q7_t)__SSAT((conv_out >> out_shift), 8);
        }
    }
}
}
```

Conv_out: Accumulation variable 32-bit

Bias

Load Multiply 8bit weight and
activations and accumulate in 32-bit

Requantize and
write to 8-bit output

Convolution Kernel (Optimized CMSIS-NN)

Parts of the optimized CMSIS Kernel for ARM

```
1  /* mid part */
2  for (; i_out_x < dim_im_out - padding; i_out_x++)
3  {
4      /* This part implements the im2col function */
5      for (i_ker_y = i_out_y * stride - padding; i_ker_y < i_out_y * stride - padding + dim_kernel; i_ker_y++)
6      {
7          arm_q7_to_q15_reordered_no_shift((q7_t *)Im_in +
8              (i_ker_y * dim_im_in + i_out_x * stride - padding) * ch_im_in,
9              pBuffer,
10              ch_im_in * dim_kernel);
11          pBuffer += ch_im_in * dim_kernel;
12      }
13
14      if (pBuffer == bufferA + 2 * ch_im_in * dim_kernel * dim_kernel)
15      {
16          pOut = arm_nn_mat_mult_kernel_q7_q15_reordered(
17              wt, bufferA, ch_im_out, ch_im_in * dim_kernel * dim_kernel, bias_shift, out_shift, bias, pOut);
18          /* counter reset */
19          pBuffer = bufferA;
20      }
21  }
22
```

Img2col
transformation

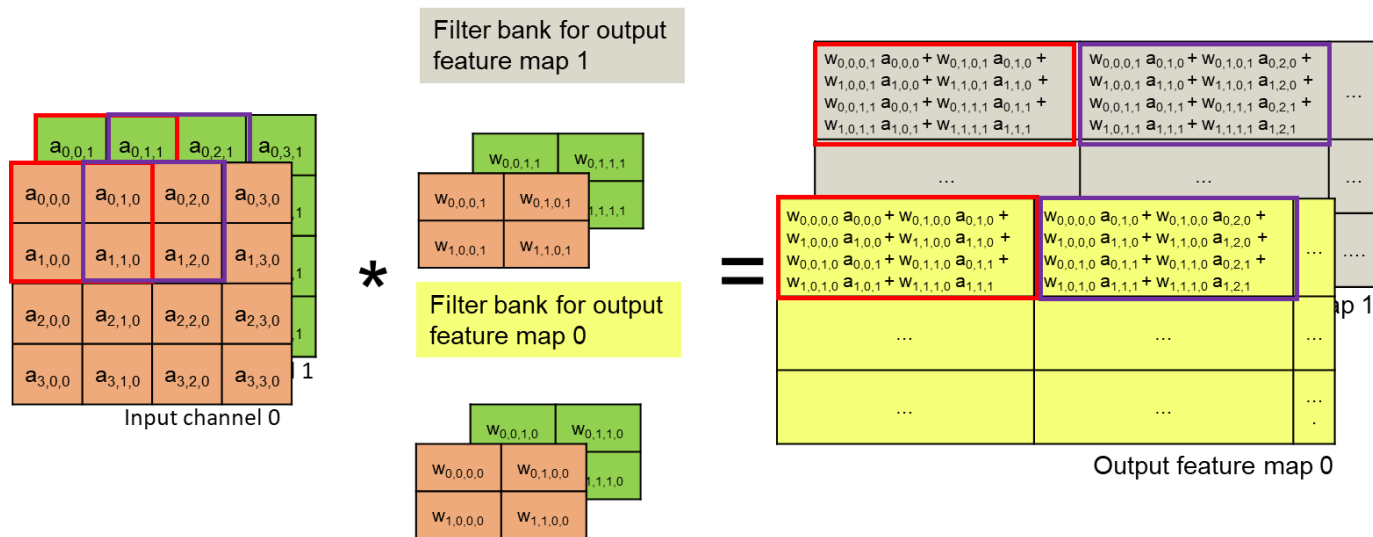
Optimized
Matmult
implementation
for target

Image to Column (Img2Col) Transformation

- For many targets there exist a very optimized implementation of matrix-matrix-multiply computation e.g. accelerators, for CPUs with some SIMD support, GPUs, but also single-issue CPUs
- Img2Col transforms a convolution operation into a matrix-matrix-multiply operation
- Img2Col requires to build up a batch matrix, which is larger than the original activation tensor, because it holds duplicates of some values
- Usually Img2Col is not done on the full input activation tensor but inside the convolution loop on some part of the tensor in order to avoid building up the full batch matrix

Example for Img2Col (1/5)

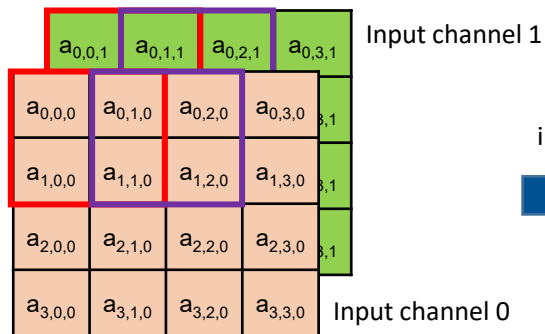
For reference: This is the Standard Convolution



Example for Img2Col (2/5)

Step 1 for Img2Vol: Create a line-based batch matrix

Each line holds the activation values under one kernel position for all channels



batch 1	batch 2	
$a_{0,0,0}$	$a_{0,1,0}$	...
$a_{0,1,0}$	$a_{0,2,0}$	...
$a_{1,0,0}$	$a_{1,1,0}$	...
$a_{1,1,0}$	$a_{1,2,0}$	...
$a_{0,0,1}$	$a_{0,1,1}$	...
$a_{0,1,1}$	$a_{0,2,1}$	...
$a_{1,0,1}$	$a_{1,1,1}$	...
$a_{1,1,1}$	$a_{1,2,1}$	...

Example for Img2Col (3/5)

Step 2: Create a column-based filter matrix. (Can be done already offline, is already existing with just storing weight tensor in ROM memory)

Filter bank for output feature map 1 (FM1)

	$w_{0,0,1,1}$	$w_{0,1,1,1}$
$w_{0,0,0,1}$	$w_{0,1,0,1}$	$w_{1,1,1,1}$
$w_{1,0,0,1}$	$w_{1,1,0,1}$	

Filter bank for output feature map 0 (FM0)

	$w_{0,0,1,0}$	$w_{0,1,1,0}$
$w_{0,0,0,0}$	$w_{0,1,0,0}$	$w_{1,1,1,0}$
$w_{1,0,0,0}$	$w_{1,1,0,0}$	

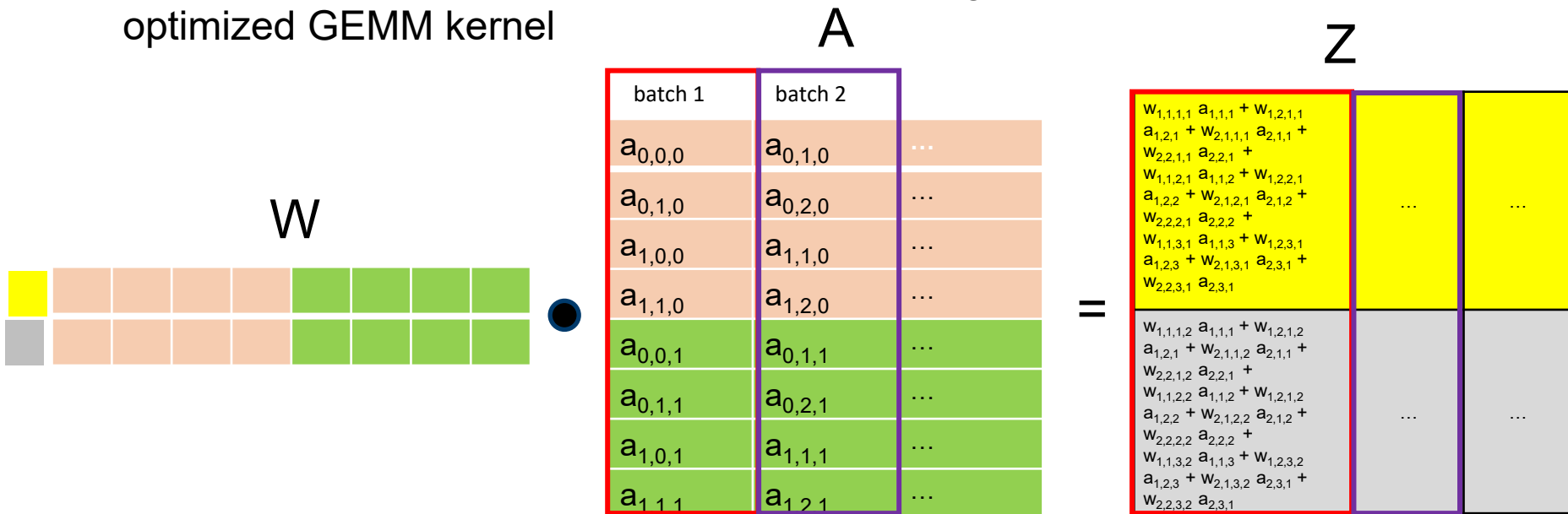
Img2col_weights



	$w_{0,0,0,0}$	$w_{0,1,0,0}$	$w_{1,0,0,0}$	$w_{1,1,0,0}$	$w_{0,0,1,0}$	$w_{0,1,1,0}$	$w_{1,0,1,0}$	$w_{1,1,1,0}$
	$w_{0,0,0,1}$	$w_{0,1,0,1}$	$w_{1,0,0,1}$	$w_{1,1,0,1}$	$w_{0,0,1,1}$	$w_{0,1,1,1}$	$w_{1,0,1,1}$	$w_{1,1,1,1}$

Example for Img2Col (4/5)

Step 3: Run a matrix-matrix multiplication with target-specific optimized GEMM kernel



Example for Img2Col (5/5)

Step 4: Reshape the output to recover the output feature maps using the inverse col2img transformation.

$w_{1,1,1,1} a_{1,1,1} + w_{1,2,1,1} a_{1,2,1} +$ $w_{2,1,1,1} a_{2,1,1} + w_{2,2,1,1} a_{2,2,1} +$ $w_{1,1,2,1} a_{1,1,2} + w_{1,2,2,1} a_{1,2,2} +$ $w_{2,1,2,1} a_{2,1,2} + w_{2,2,2,1} a_{2,2,2} +$ $w_{1,1,3,1} a_{1,1,3} + w_{1,2,3,1} a_{1,2,3} +$ $w_{2,1,3,1} a_{2,3,1} + w_{2,2,3,1} a_{2,3,1}$	...	...
$w_{1,1,1,2} a_{1,1,1} + w_{1,2,1,2} a_{1,2,1} +$ $w_{2,1,1,2} a_{2,1,1} + w_{2,2,1,2} a_{2,2,1} +$ $w_{1,1,2,2} a_{1,1,2} + w_{1,2,2,2} a_{1,2,2} +$ $w_{2,1,2,2} a_{2,1,2} + w_{2,2,2,2} a_{2,2,2} +$ $w_{1,1,3,2} a_{1,1,3} + w_{1,2,3,2} a_{1,2,3} +$ $w_{2,1,3,2} a_{2,3,1} + w_{2,2,3,2} a_{2,3,1}$	...	...

col2img



$w_{0,0,0,1} a_{0,0,0} + w_{0,1,0,1} a_{0,1,0} +$ $w_{1,0,0,1} a_{1,0,0} + w_{1,1,0,1} a_{1,1,0} +$ $w_{0,0,1,1} a_{0,0,1} + w_{0,1,1,1} a_{0,1,1} +$ $w_{1,0,1,1} a_{1,0,1} + w_{1,1,1,1} a_{1,1,1}$	$w_{0,0,0,1} a_{0,1,0} + w_{0,1,0,1} a_{0,2,0} +$ $w_{1,0,0,1} a_{1,1,0} + w_{1,1,0,1} a_{1,2,0} +$ $w_{0,0,1,1} a_{0,1,1} + w_{0,1,1,1} a_{0,2,1} +$ $w_{1,0,1,1} a_{1,1,1} + w_{1,1,1,1} a_{1,2,1}$	...
...	...	...
$w_{0,0,0,0} a_{0,0,0} + w_{0,1,0,0} a_{0,1,0} +$ $w_{1,0,0,0} a_{1,0,0} + w_{1,1,0,0} a_{1,1,0} +$ $w_{0,0,1,0} a_{0,0,1} + w_{0,1,1,0} a_{0,1,1} +$ $w_{1,0,1,0} a_{1,0,1} + w_{1,1,1,0} a_{1,1,1}$	$w_{0,0,0,0} a_{0,1,0} + w_{0,1,0,0} a_{0,2,0} +$ $w_{1,0,0,0} a_{1,1,0} + w_{1,1,0,0} a_{1,2,0} +$ $w_{0,0,1,0} a_{0,1,1} + w_{0,1,1,0} a_{0,2,1} +$ $w_{1,0,1,0} a_{1,1,1} + w_{1,1,1,0} a_{1,2,1}$	...
...	...	...
...	...	...

map 1

Output feature map 0

GEMM Algorithm

Basic linear algebra algorithm for matrix-matrix multiply

Optimized versions exist for many hardware platforms e.g.

- Considering block-wise computation depending on cache sizes
- Exploiting SIMD instructions (or vector instructions)

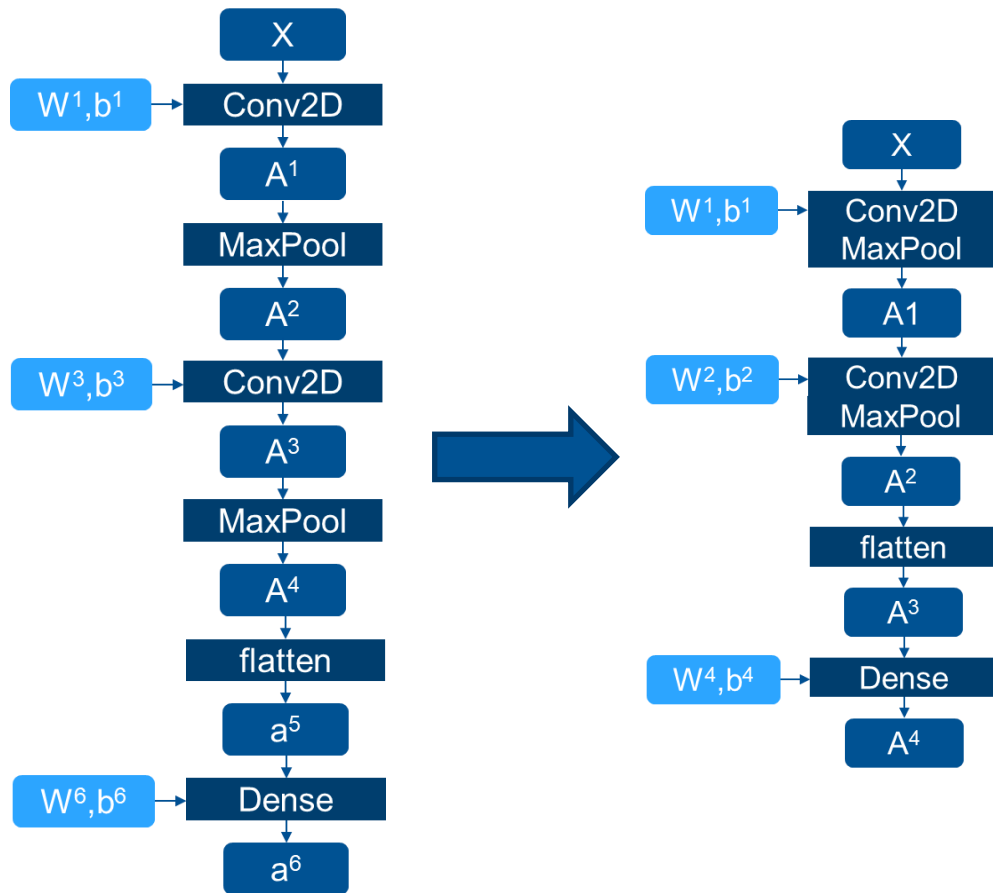
GEMM is seen as „*at the heart of deep learning*“ especially when acceleration is considered.

Further reading:

<https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/>

Layer Fusing (1/2)

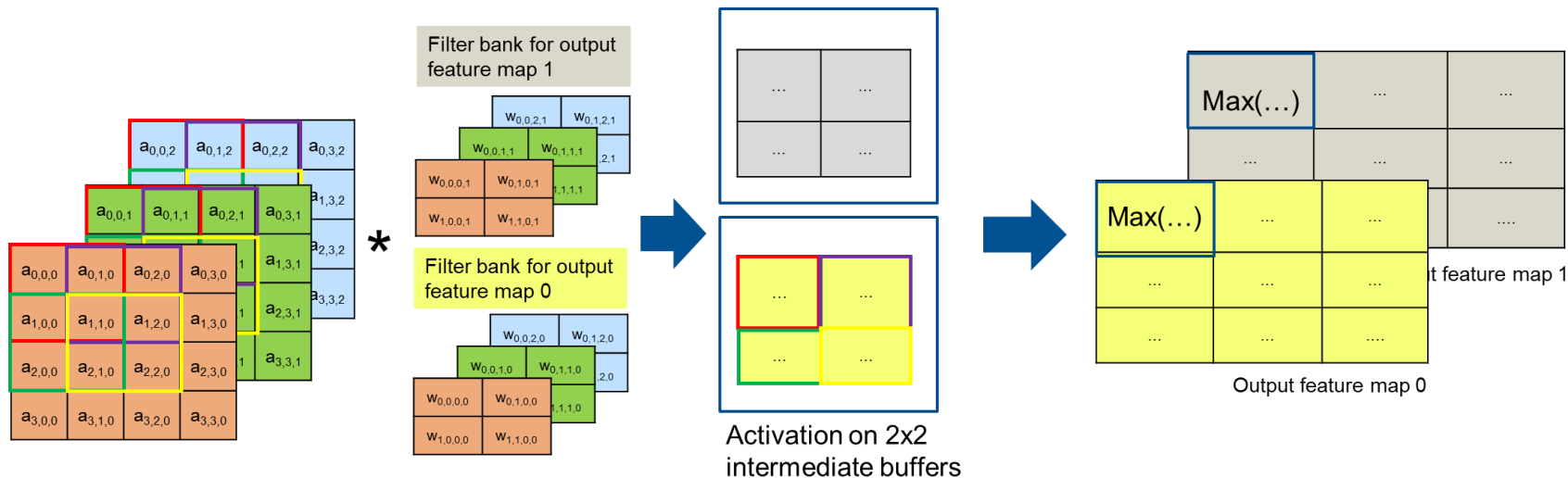
- It can be more efficient to compute several layers together (fusing)
- Intermediate Tensor does not need to be written
- Number of loops can be reduced.
- Works when second layer requires only some output of the first layer present.



Layer Fusing (2/2)

Example: Fusing Conv2D with MaxPool (2,2)

1. Compute outputs of convolutions with activation function for kernel positions that are in one MaxPool window (Example 2x2 max window)
2. Run the maxpool on the partly computed outputs and save result as final result after both operations



Tasks of Code Generation

Kernel Generation and Optimization:

- Generate kernel codes that implement the computation of a layer
- Optimize kernels (e.g. fusing layers) for efficient execution on the target HW

Storage for read only data:

- Decide on format how to store weight tensors
- Plan memory to store weight tensors and layer configurations

Storage for read-write data:

- Decide on format how to store activation tensors
- Plan memory to store the activation tensors (memory reuse)

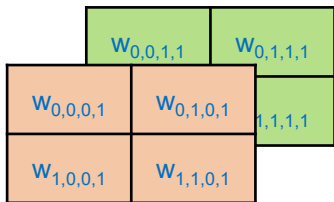
Model Graph Generation / Runtime Setup:

- Layer Scheduling
- Generate code or include runtime that executes the model graph (calls layer kernels on the tensors)

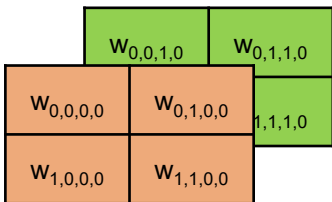
Kernel Tensor Memory Layouts (2/4)

Byte-quantized weights:

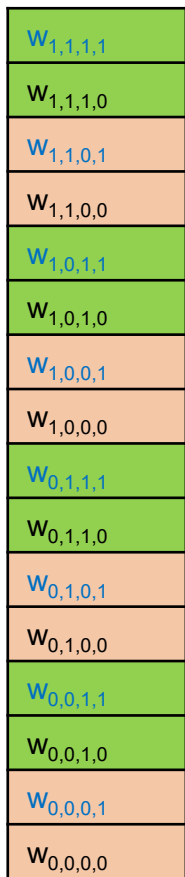
Filter bank for output feature map 1 (FM1)



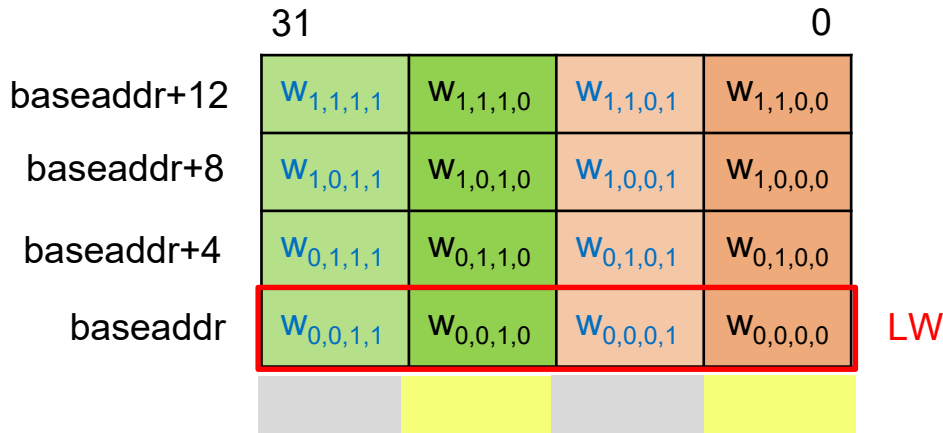
Filter bank for output feature map 0 (FM0)



Flatten
into 1D



Layout for 8-bit quantized (HWIO) weights
in 32bit memory



Load Byte **LB** : Loads one weight

Load Word **LW**: Loads 4 weights at once ->
mask & shift operations required or SIMD DSP
instruction

Tasks of Code Generation

Kernel Generation and Optimization:

- Generate kernel codes that implement the computation of a layer
- Optimize kernels (e.g. fusing layers) for efficient execution on the target HW

Storage for read only data:

- Decide on format how to store weight tensors
- Plan memory to store weight tensors and layer configurations

Storage for read-write data:

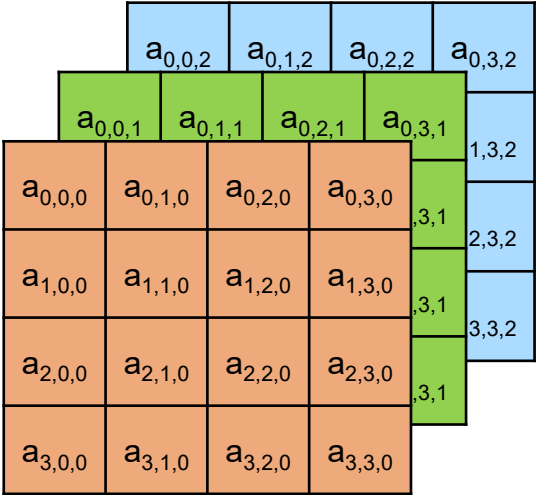
- Decide on format how to store activation tensors
- Plan memory to store the activation tensors (memory reuse)

Model Graph Generation / Runtime Setup:

- Layer Scheduling
- Generate code or include runtime that executes the model graph (calls layer kernels on the tensors)

Activation Tensor Memory Formats (3/3)

Example:



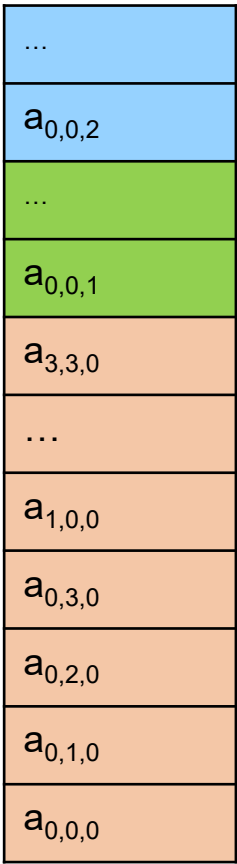
img_buffer[2*4*4+0]

img_buffer[1*4*4+0]

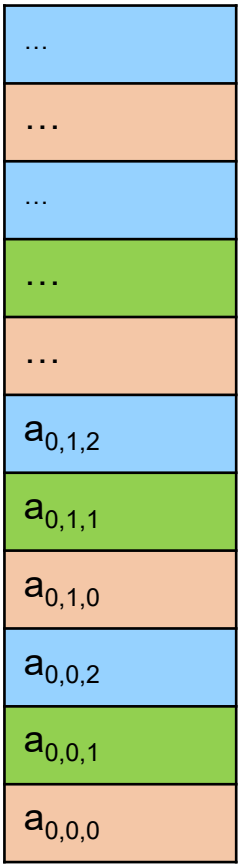
img_buffer[1*4+0]

img_buffer[0]

Channel first
CHW



Channel last
HWC



Memory Planner (Tensor Arena)

Tensor Arena:

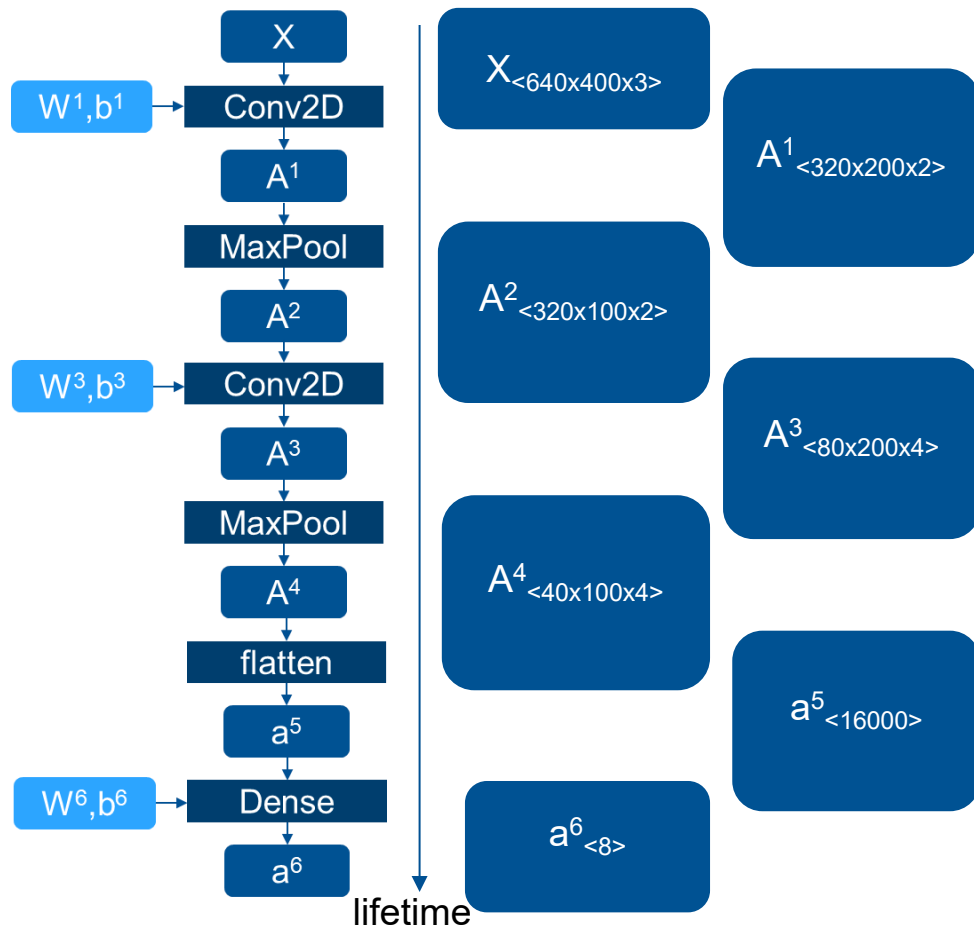
- Memory section reserved for memory buffers, which hold the tensors that store the activations of the layers
- Buffers can overlap if the activation tensors are not live at the same time
 - Sharing of memory space (very important for TinyML)
 - Requires memory planner

More Reading:

- Y. Pisarchyk and J. Lee, “Efficient memory management for deep neural net inference,” CoRR, vol. abs/2001.03288, 2020. [Online]. Available: <https://arxiv.org/abs/2001.0328>
- <https://blog.tensorflow.org/2020/10/optimizing-tensorflow-lite-runtime.html>

Tensor Lifetime Analysis

- Tensors are live from the generation until their last use.
- Output and input of a layer must be live during execution of that layer.



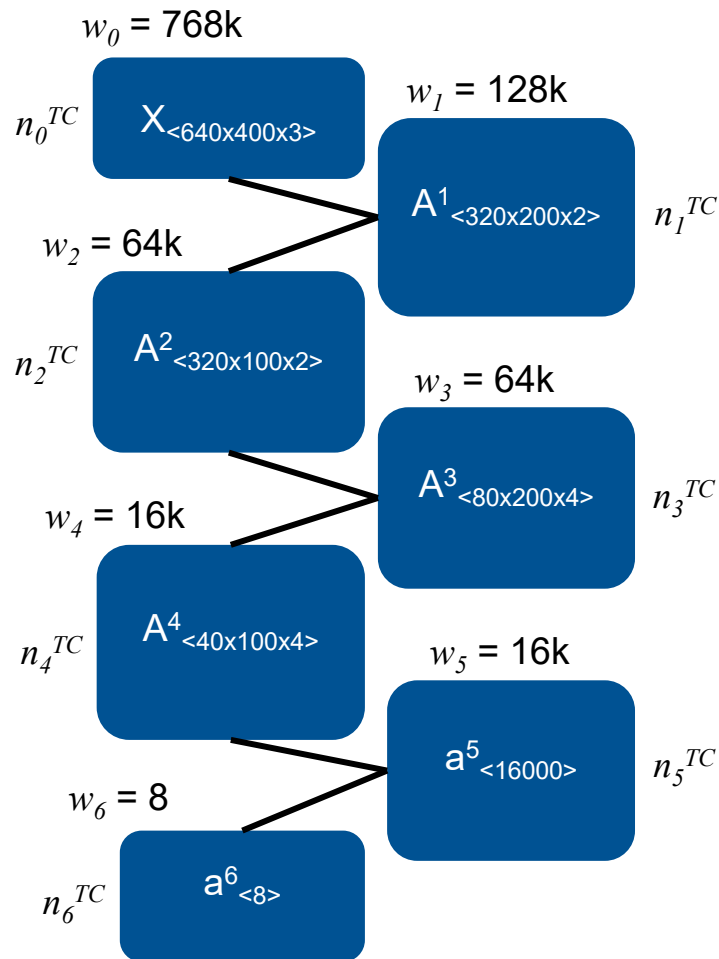
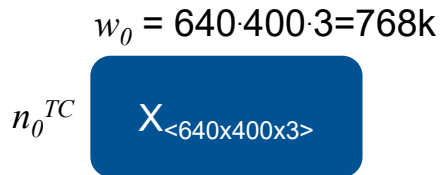
Tensor Conflict Graph

Undirected Interval Graph $G^{TC}(N^{TC}, E^{TC})$

Nodes n_i^{TC} : Tensors

Node weights w_i : Tensor sizes

Edges $e=(n_i^{TC}, n_j^{TC})$: Two nodes are connected if the lifetime of the tensors overlap



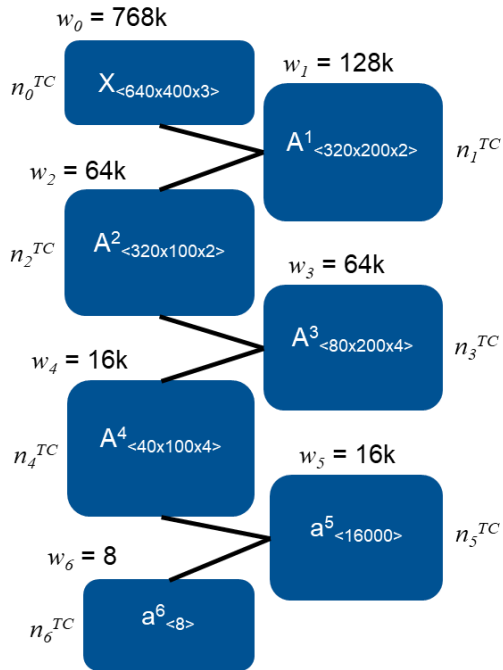
Greedy by Size Memory Planning (1/3)

Greedy by Size Algorithm using Shared Buffers:

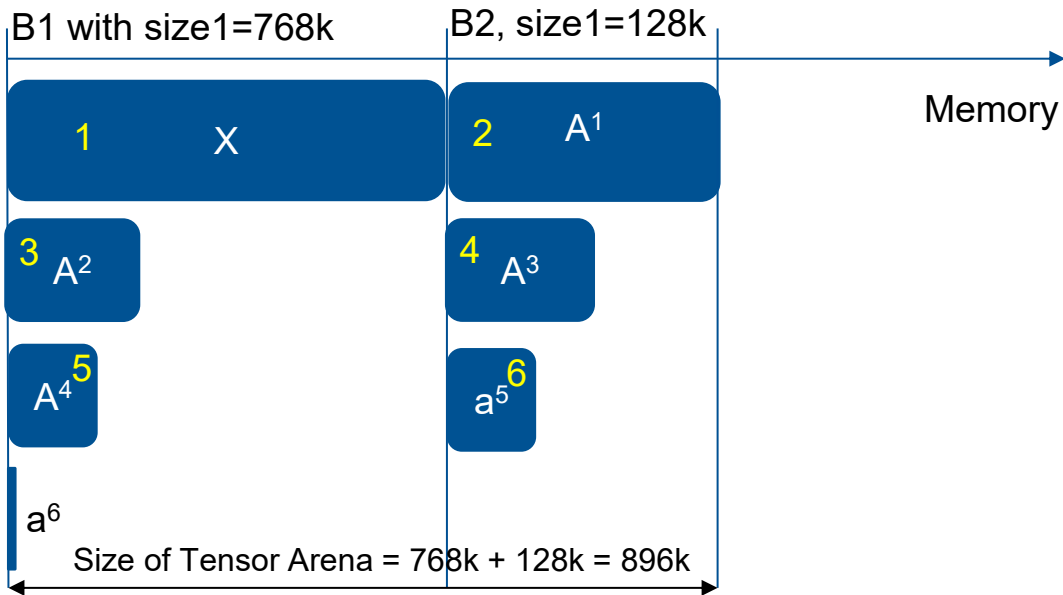
```
MemoryPlannerGreedyBySize( $G^{TC}(N^{TC}, E^{TC})$ ) {  
    Sort nodes  $n_{TG\_i}$  in list L by decreasing  $w_i$   
    Initialize empty List of Shared Buffers  $L_B$   
    repeat {  
        Pop next node  $n_{TG\_i}$  from L  
        if there exist suitable Shared Buffers  $B_j$  in  $L_B$  with  
            size_j  $\geq w_i$  and  
            forall  $n_{TG\_k}$  in  $B_j$  there is no conflict ( $n_{TG\_i}, n_{TG\_k}$ ) in ETC  
        {  
            Insert  $n_{TG\_k}$  in smallest suitable  $B_j$   
        }  
        else {  
            Create new  $B_j$  with size size_j =  $w_i$  and insert in  $L_B$   
        }  
    } until (End of List L reached)  
}
```

Greedy by Size for Memory Planning (2/3)

Example 1



$$L = [n_0^{TC}, n_1^{TC}, n_2^{TC}, n_3^{TC}, n_4^{TC}, n_5^{TC}, n_6^{TC}]$$



Base pointers for tensor buffers in B1: = Start of Tensor Arena
 Base pointers for tensor buffers in B2: = Start of Tensor Arena + 768k

Tasks of Code Generation

Kernel Generation and Optimization:

- Generate kernel codes that implement the computation of a layer
- Optimize kernels (e.g. fusing layers) for efficient execution on the target HW

Storage for read only data:

- Decide on format how to store weight tensors
- Plan memory to store weight tensors and layer configurations

Storage for read-write data:

- Decide on format how to store activation tensors
- Plan memory to store the activation tensors (memory reuse)

Model Graph Generation / Runtime Setup:

- Layer Scheduling
- Generate code or include runtime that executes the model graph (calls layer kernels on the tensors)

Open Source ML Deployment Frameworks

Tensor Flow Lite / TFLM (Google) [<https://www.tensorflow.org/lite/microcontrollers>]

- Offers a runtime (interpreter) that can execute a quantized model graph
- Quantized model graph is stored in a flatbuffer file (.tflite)
- Provides reference (hardcoded) kernel library that the interpreter calls
- Usually kernel library is replaced with efficient inference library (e.g. for ARM: CMSIS-NN)

Apache TVM / MicroTVM [<https://tvm.apache.org/>]

- Compiler-based optimization and kernel generation based on high- and low-level IR
- Offers several backends for different targets either interpreter-based or static code generation
- Support auto-tuning for HW-specific optimizations and HW accelerator support via BYOC interface

(Tiny)IREE [<https://github.com/iree-org/iree>]

- Based on LLVM's MLIR (Multi Level IR) Project
- Highly flexible and modular
- Lightweight runtime or bytecode interpreter

Commercial Deployment Flows

Incomplete list!

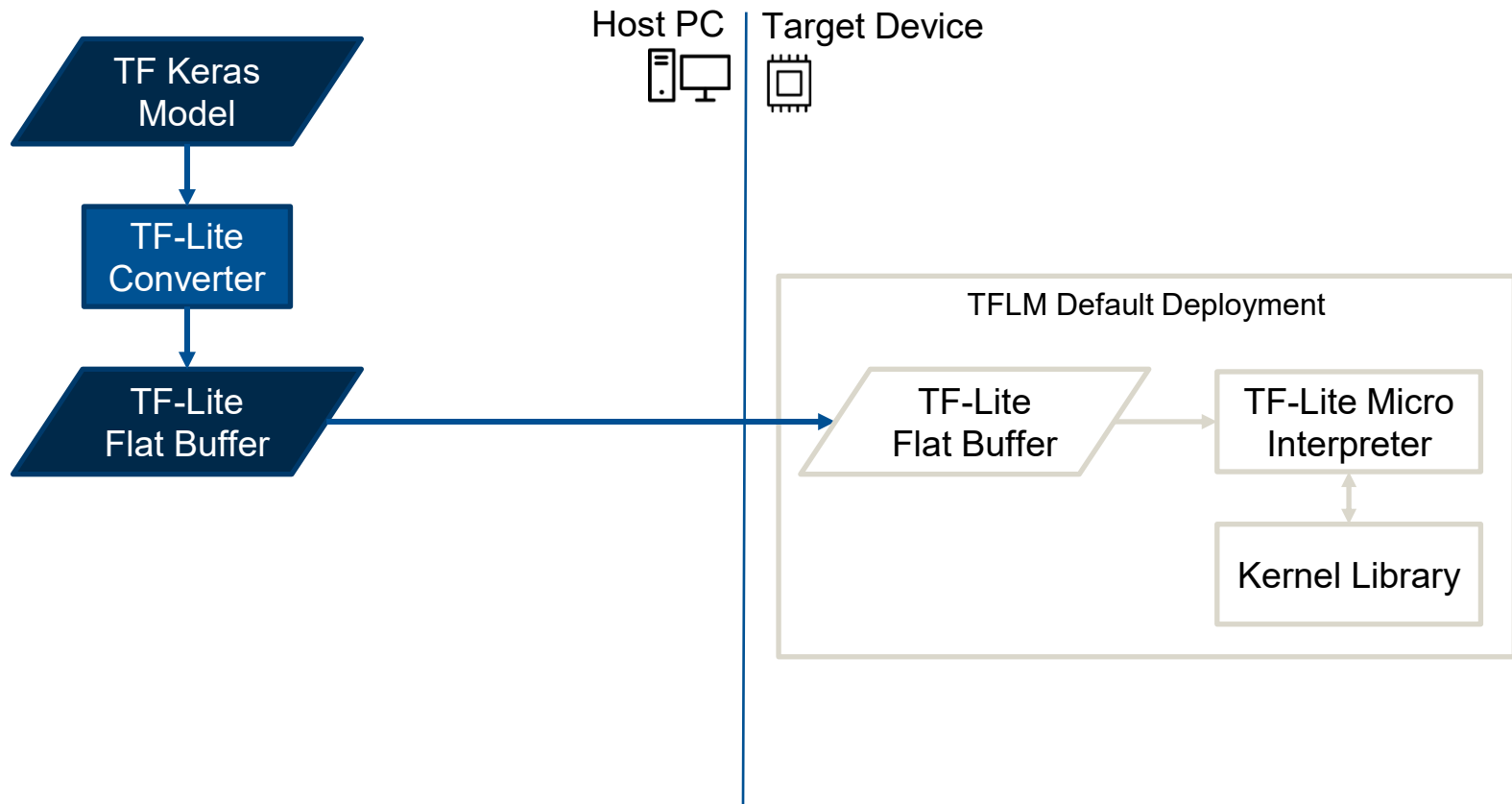
MCU-Vendor deployment tools:

- Infineon's ModusToolbox™ ML platform
- ST X-CUBE-AI
- ...

AutoESL tools:

- Imagimob AI
- SensiML
- Edge Impulse
- ...

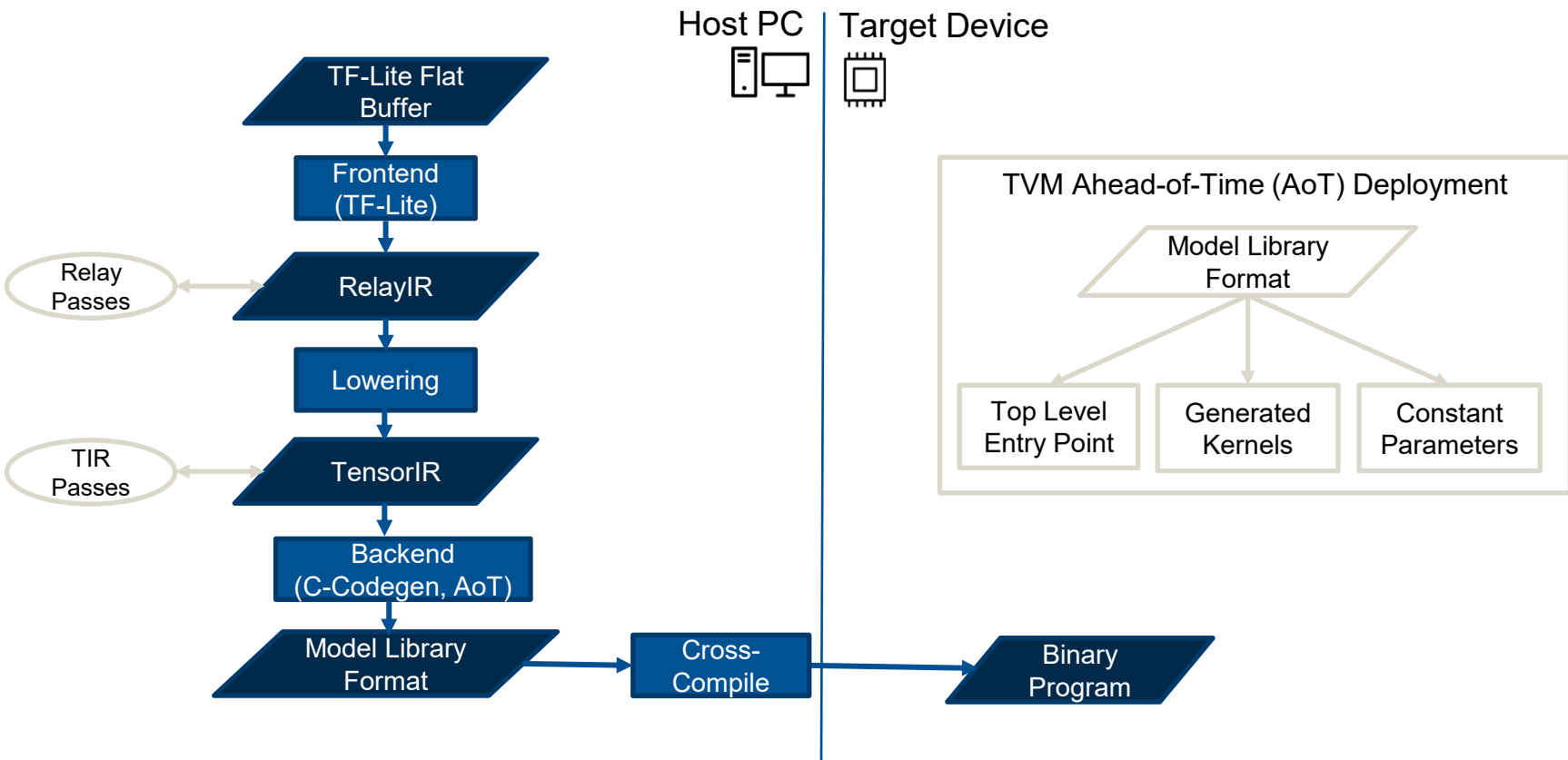
Code Generation Flow (TFLM) (1/2)



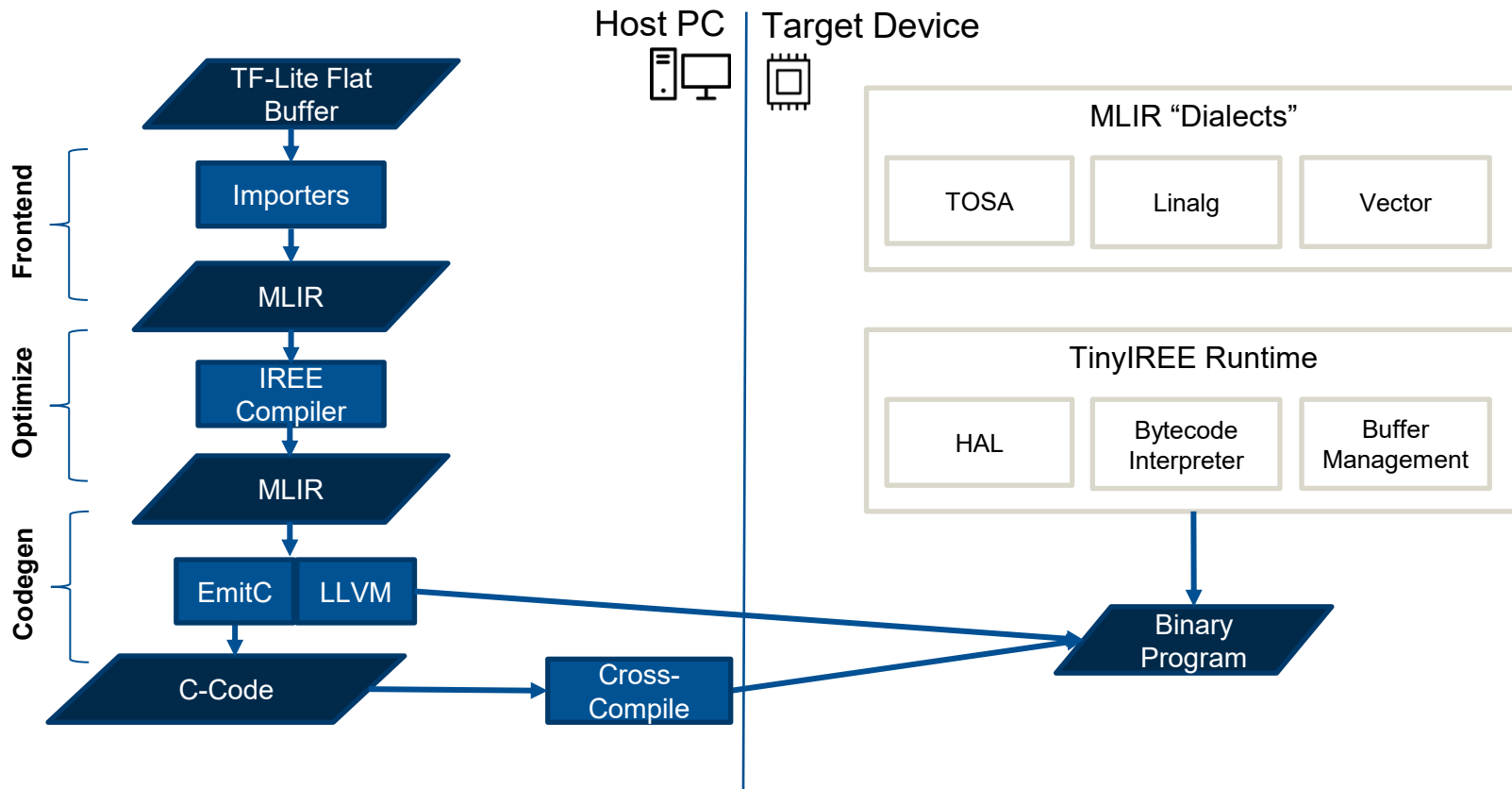
Code Generation Flow (TFLM) (2/2)

- TensorFlow / Keras models are converted into TF-Lite format (.tflite)
 - Includes automated quantization
 - Loads model structure
 - Does memory planning
 - Calls into kernel library

Code Generation Flow (TVM)



Code Generation Flow (TinyIREE)



Tasks of Code Generation

Kernel Generation and Optimization:

- Generate kernel codes that implement the computation of a layer
- Optimize kernels (e.g. fusing layers) for efficient execution on the target HW

Storage for read only data:

- Decide on format how to store weight tensors
- Plan memory to store weight tensors and layer configurations

Storage for read-write data:

- Decide on format how to store activation tensors
- Plan memory to store the activation tensors (memory reuse)

Model Graph Generation / Runtime Setup:

- Layer Scheduling
- Generate code or include runtime that executes the model graph (calls layer kernels on the tensors)

What to expect?

- 09:15 *Join in / Arrival @ AI Factory Austria AI:AT*
- 09:30 **From Data-driven Design to tinyML: Designing Machine Learning Models for Tiny Embedded Systems**
- 10:30 **Deploying Machine Learning Models on Micro-Controller Units (MCUs)**
- 11:10 *Break and Q&A*
- 11:25 **Hands on 1: Keyword Spotting on an MCU using TVM**
- 12:25 *Q&A and Wrap up*
- 12:30 *End of course*

Please feel free to
always ask questions
in between!

