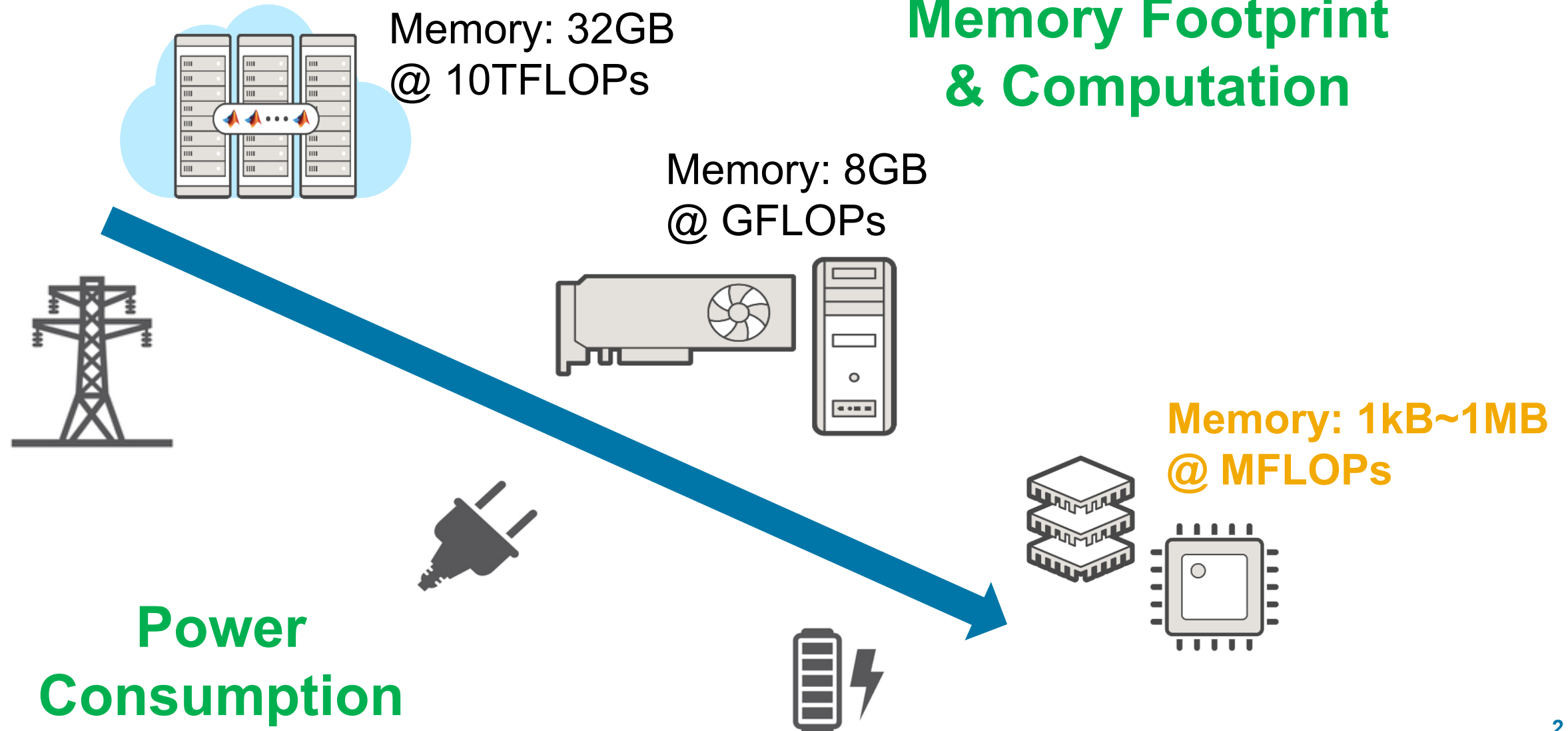


Embedded AI: Compression techniques and performance optimization using automated C/C++ code generation

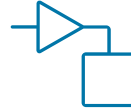
Christoph Stockhammer
MathWorks Application Engineering

Embedded devices are resource-constrained

Memory Footprint & Computation



Embedded AI Workflow



Data Preparation

Toolboxes for **data** pre-processing

AI Modeling

Low-code AI modeling & training through MATLAB Apps

Model import from **TensorFlow**, **PyTorch**, or other frameworks

Simulation & Test

Simulink blocks for AI inference make integration easy

Simulation-based testing

Compression

Model compression techniques to reduce model size and speed up inference

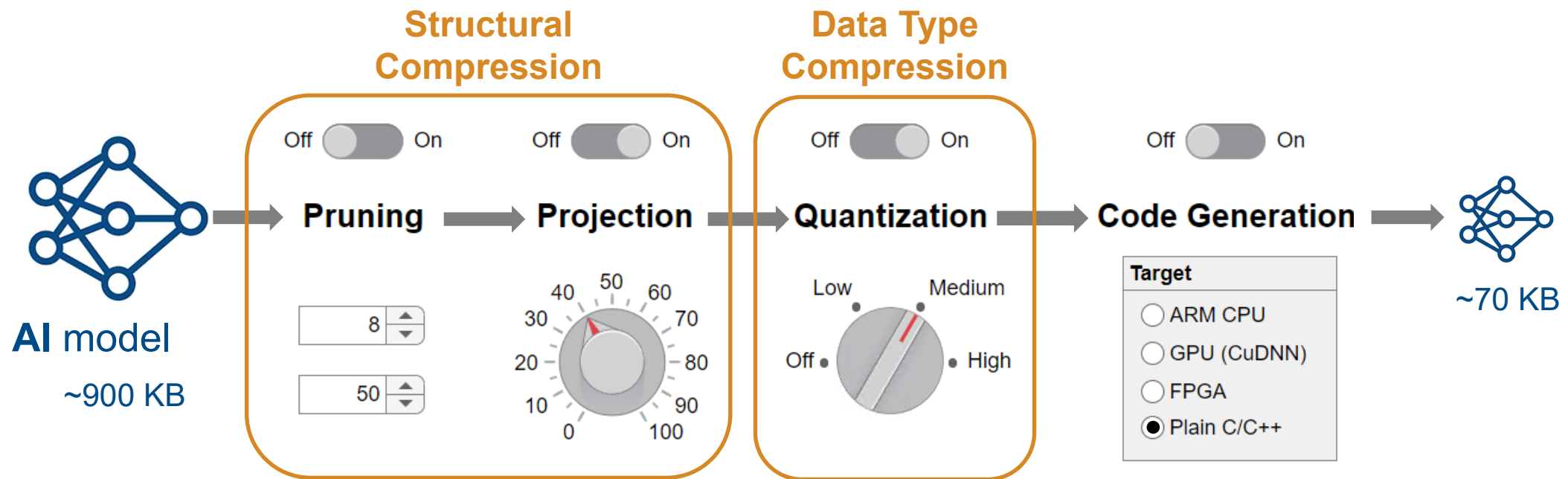
Deployment

Code generation from deep learning and machine learning models for embedded targets



Compression bridges the gap between AI modelling and embedded deployment

Reduce model footprint and accelerate inference of AI models



Agenda

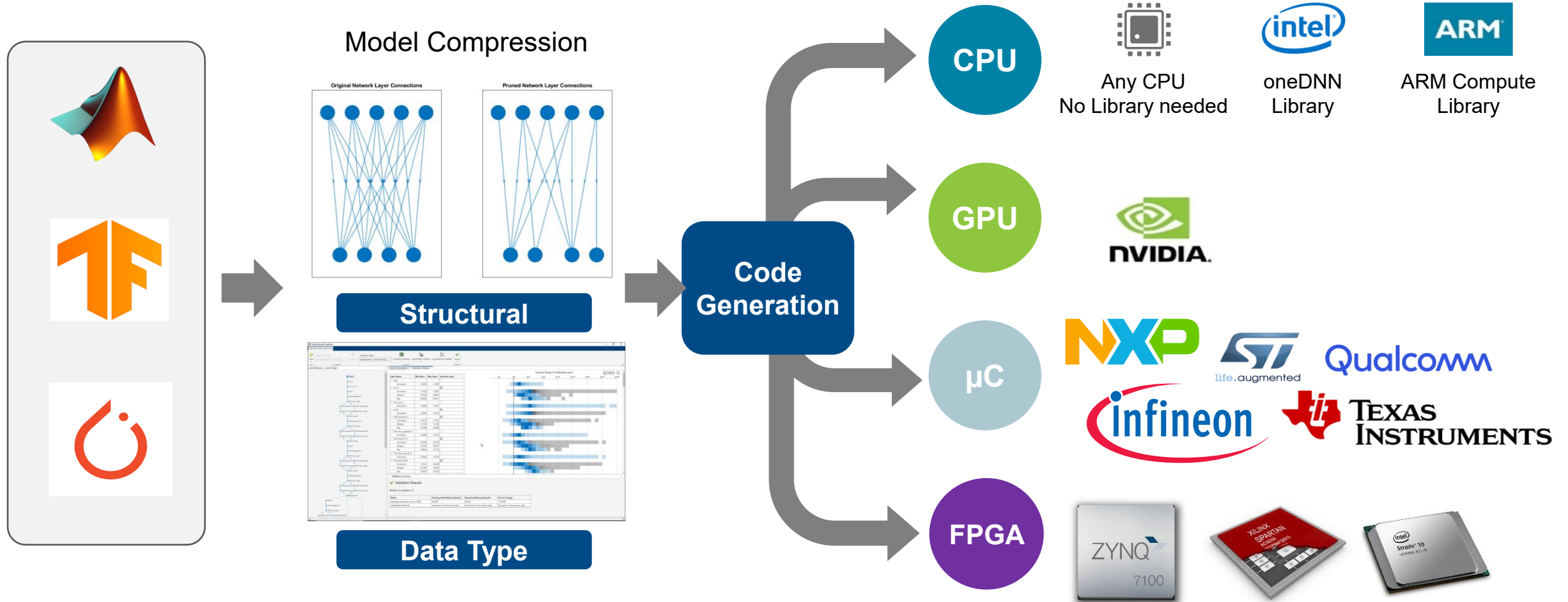
- Code Generation for Deep Learning Networks
 - Network Compression Techniques
 - Structural Compression
 - Datatype Compression (incl. Quantization)
 - Techniques for Optimizing Performance
- Code Generation for other Machine Learning Models
 - Overview

Agenda

- **Code Generation for Deep Learning Networks**
 - Network Compression Techniques
 - Structural Compression
 - Datatype Compression (incl. Quantization)
 - Techniques for Optimizing Performance

- Code Generation for other Machine Learning Models
 - Overview

Deploy efficient AI to any processor with code generation

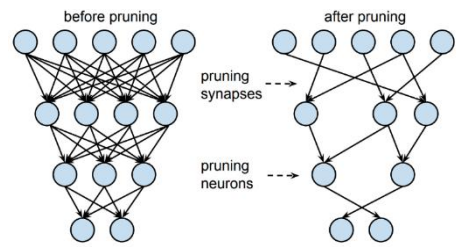
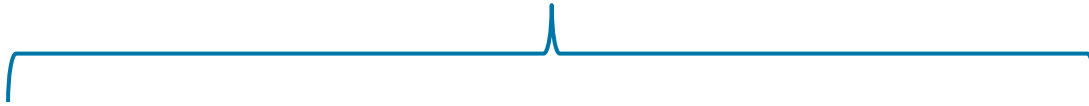


Agenda

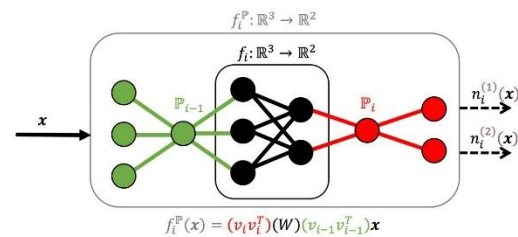
- Code Generation for Deep Learning Networks
 - **Network Compression Techniques**
 - Structural Compression
 - Datatype Compression (incl. Quantization)
 - Techniques for Optimizing Performance
- Code Generation for other Machine Learning Models
 - Overview

Compression bridges the gap between modeling and deployment

Structural Compression

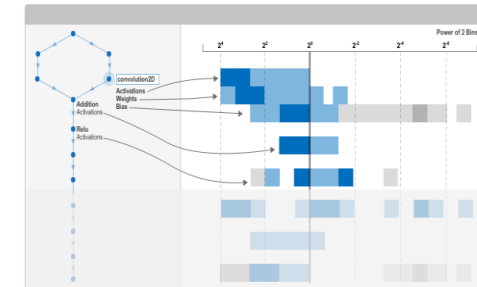
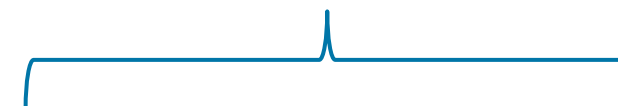


Pruning
convolution layer filters



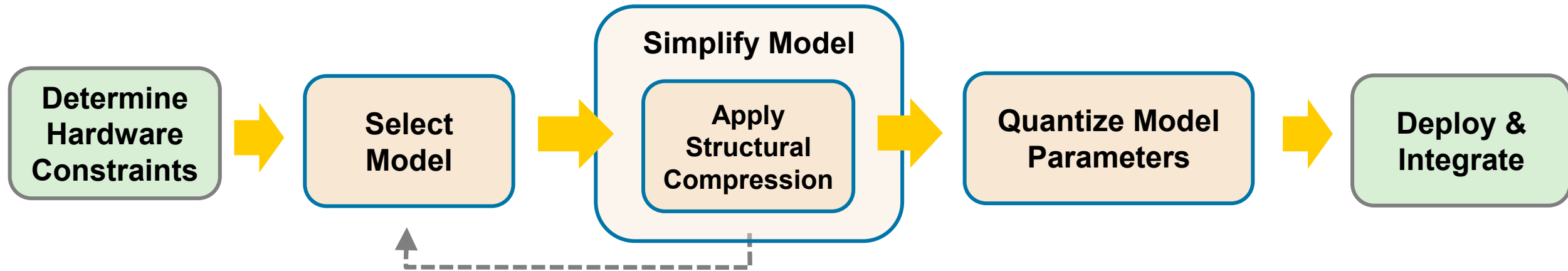
Projection of
learnables matrices

Datatype Compression



Quantization of network
weights to lower precision
datatypes (INT8/INT16,
bfloat16)

Workflow for Compressing Deep Neural Nets



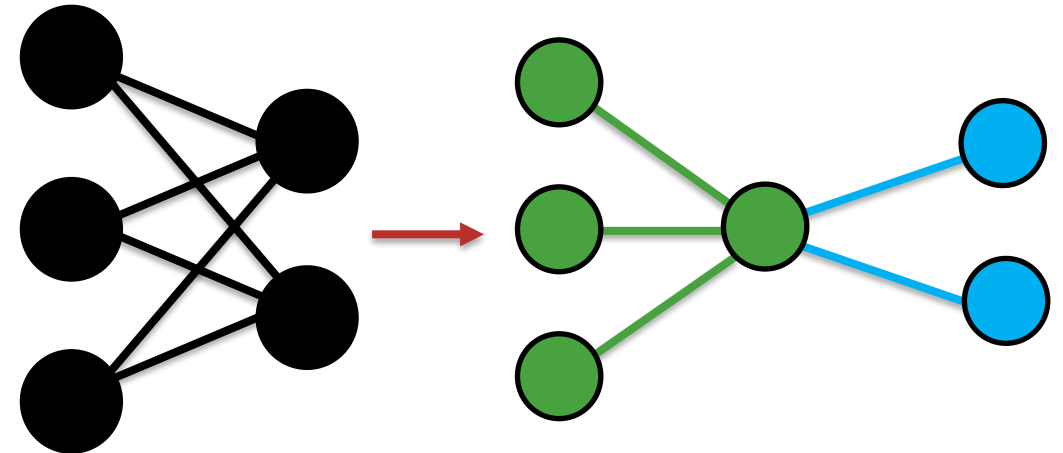
Agenda

- Code Generation for Deep Learning Networks
 - Network Compression Techniques
 - **Structural Compression**
 - Datatype Compression (incl. Quantization)
 - Techniques for Optimizing Performance

- Code Generation for other Machine Learning Models
 - Overview

Neural Network Projection

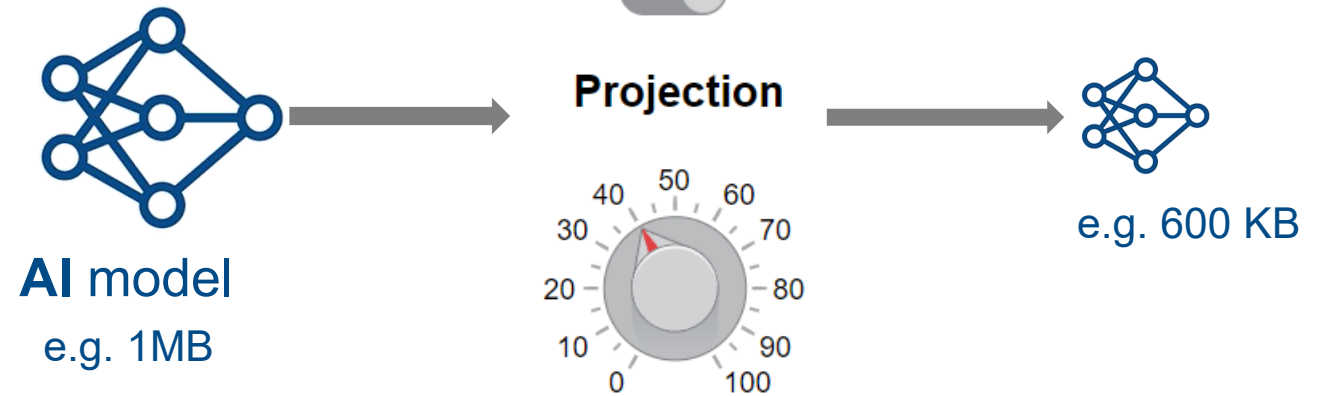
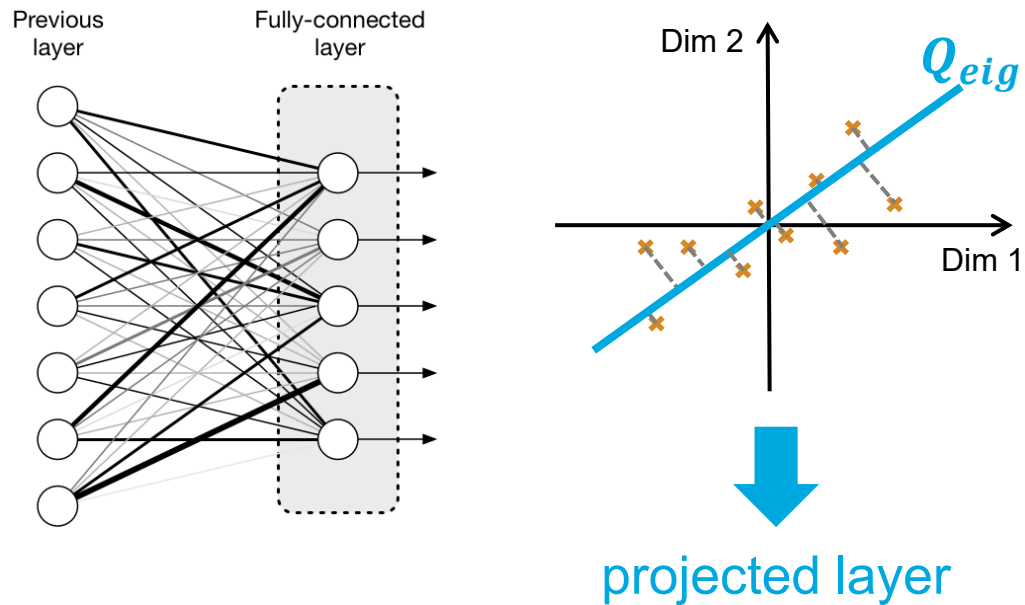
Applies principal component analysis (PCA) to weight matrices and projects them from high-dimension space to low-dimension space



[Technical article on network projection technique](#)

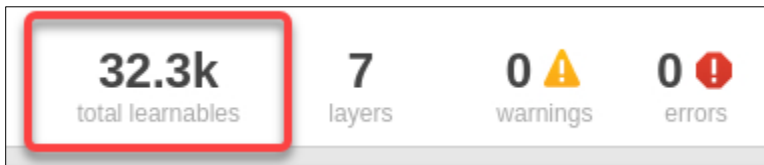
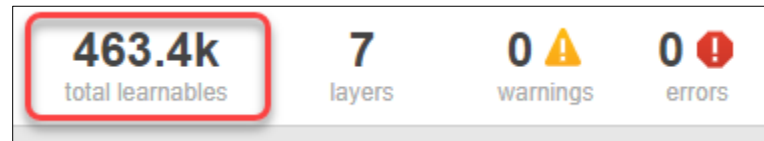
Reduce model sizes using projection

Assumption: High-dimensional space of input and output neurons is redundant

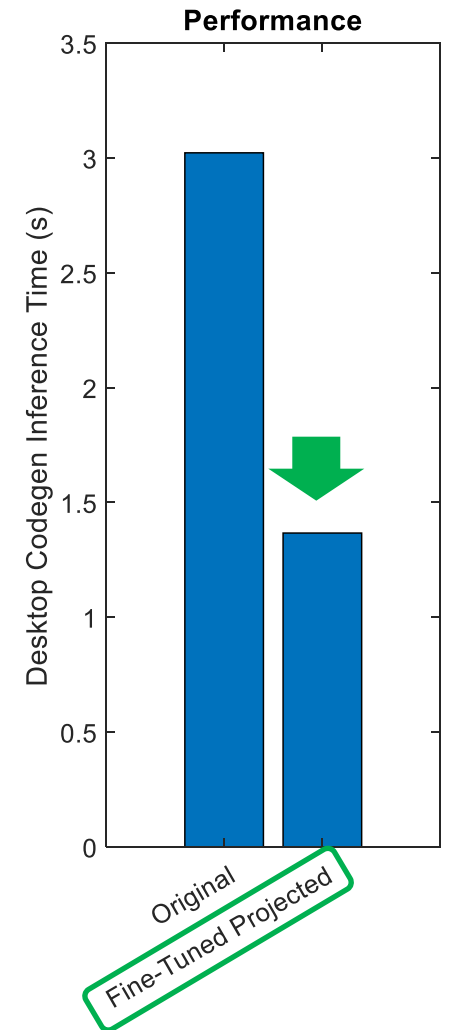
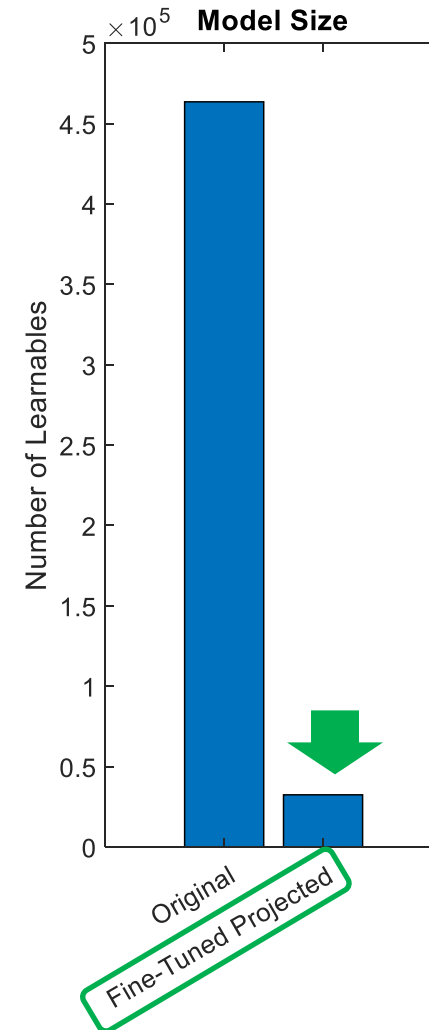
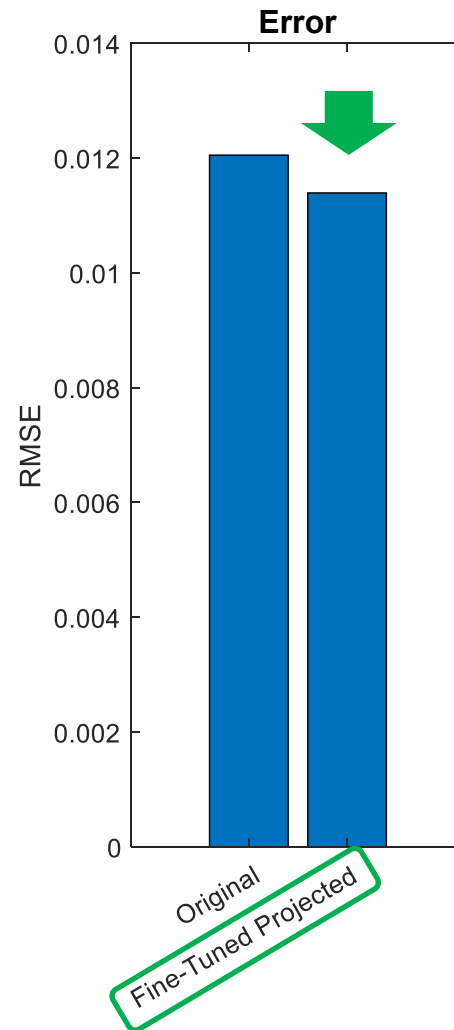


Projection Results – case study

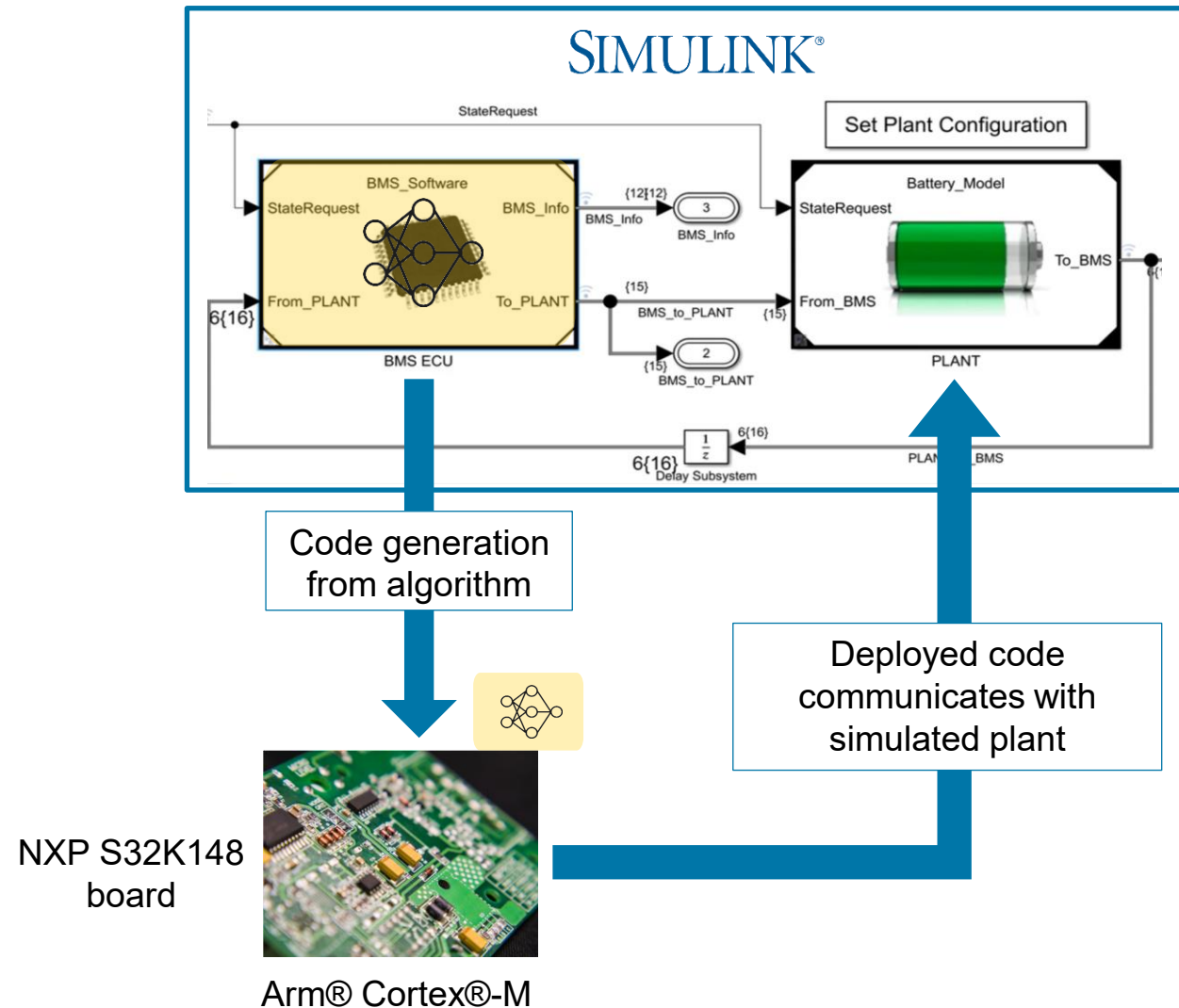
Example: LSTM Network for Battery State of Charge (SoC) Prediction



Number of total learnables decreased



Workflow: Processor-in-the-Loop (PIL) Simulations



PIL execution on ARM Cortex-M7 processor

The screenshot displays the MATLAB/Simulink environment during the execution of a PIL (Pure Integer Library) on an ARM Cortex-M7 processor. The main window shows the Simulink model for 'SOC Estimation (PIL)'. The model includes an 'input' block, a 'SOC Estimation (PIL)' block, and a 'measuredSOC' block. The output of the PIL block is 'estim', which is compared against 'true' values. The model also outputs 'current', 'voltage', and 'temperature' signals.

A 'Download finished' dialog box is visible, indicating that the executable file has been downloaded to the S32C344 board. The diagnostic viewer at the bottom shows the following terminal output:

```

C:\Users\jgazzarr\OneDrive - MathWorks\Work\Projects\AI_MBD\SOCestimation\work\sprj\ert\SOC_Estimation\pil\exit /B 0
### Updating code generation report with PIL files ...
### Starting application: 'work\sprj\ert\SOC_Estimation\pil\SOC_Estimation.elf'
165084 3008 27096 195188 2fa74 ./SOC_Estimation.elf
### Done invoking postbuild tool.
### Invoking postbuild tool "ELF To Binary Converter" ...
arm-none-eabi-objcopy -O binary ./SOC_Estimation.elf ../.././././SOC_Estimation.bin
### Done invoking postbuild tool.
### Successfully generated all binary outputs.

```

On the right side, three plots are shown, comparing 'true' values (blue line) with 'estim' values (yellow line). The top plot shows 'true, estim' values over time. The middle plot shows 'current' values over time. The bottom plot shows 'voltage' and 'temperature' values over time. The x-axis for all plots is time in seconds, ranging from 0 to 3.5 x 10⁴.

Projection can be applied to a variety of neural networks

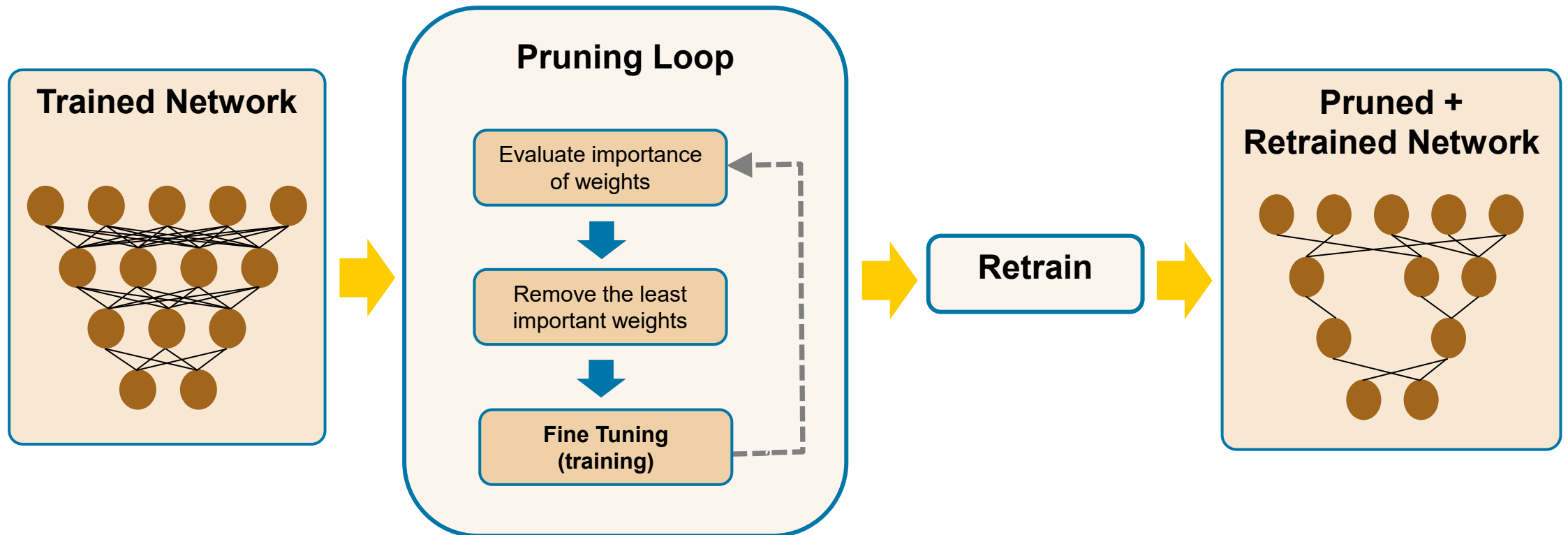
- Supported layer types:
 - **LSTM layer**
 - **GRU layer**
 - **FullyConnected layer**
 - **Convolution2D layer**
- Try projection on:
 - Recurrent Neural Networks (RNNs)
 - Convolutional Neural Networks (CNNs)
 - Object Detectors
 - Fully-connected Networks (aka MLP)
 - Other networks with large fully connected layers

Taylor Pruning for 2D convolutions

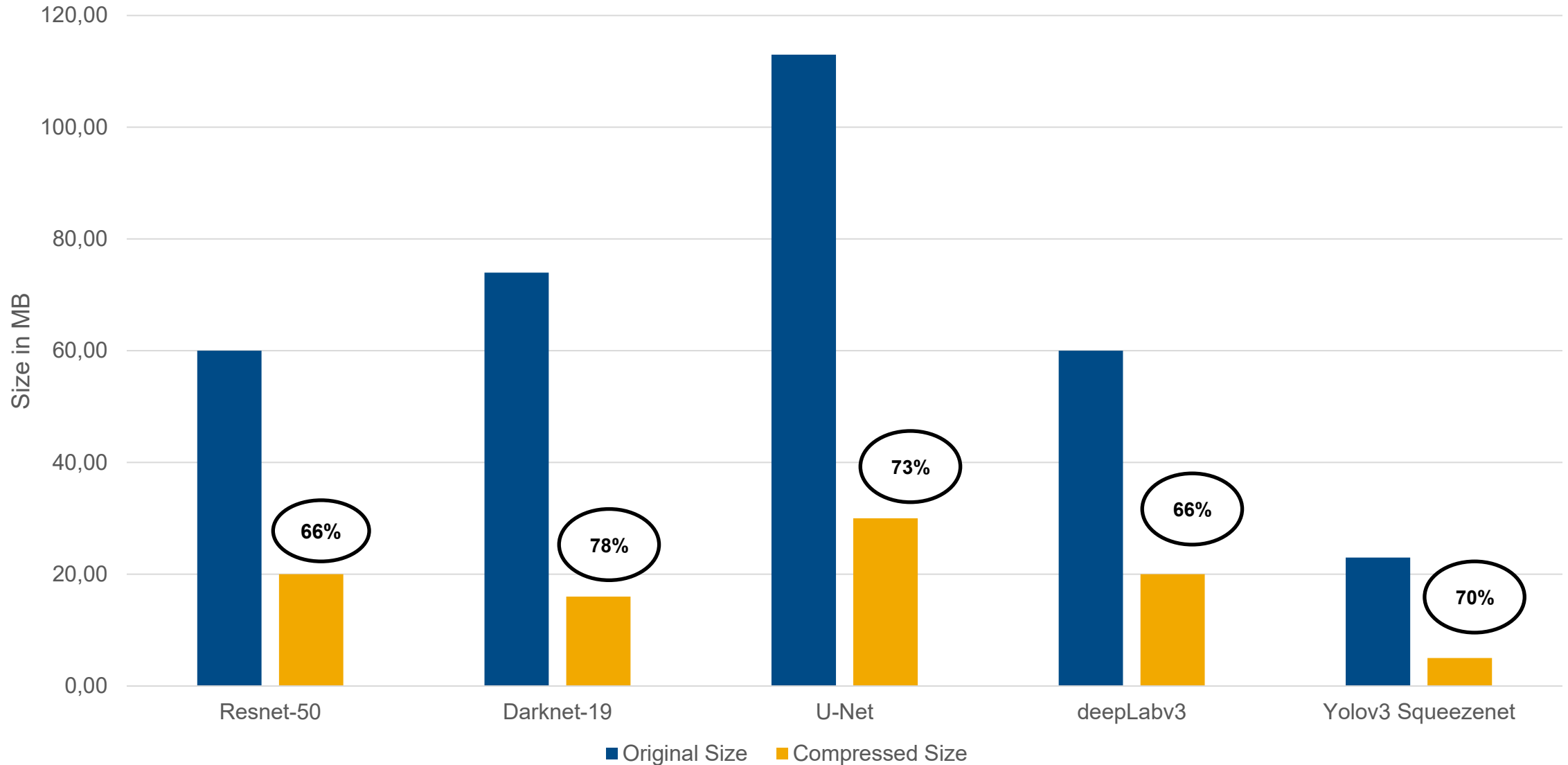
Remove **redundant** convolution filters

Supported networks

- Semantic segmentation
- Classifiers
- Object detectors



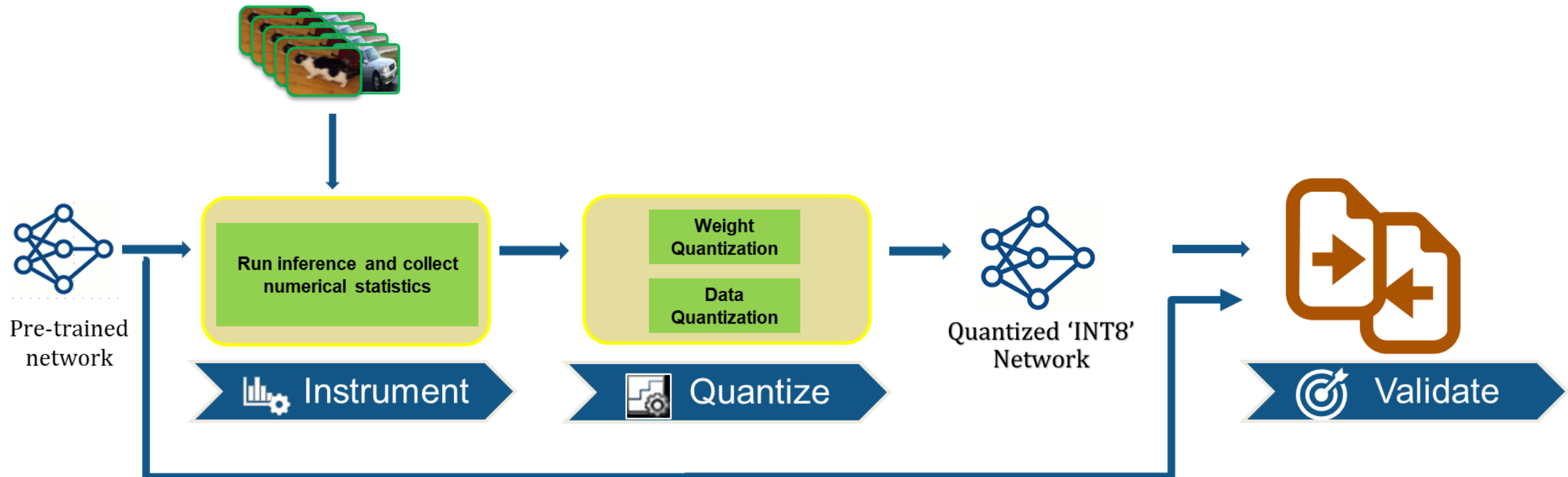
Taylor Pruning Applied to Popular Networks



Agenda

- Code Generation for Deep Learning Networks
 - Network Compression Techniques
 - Structural Compression
 - **Datatype Compression (incl. Quantization)**
 - Techniques for Optimizing Performance
- Code Generation for other Machine Learning Models
 - Overview

Deep Learning Quantization Workflow



Deep Network Quantization App

1 Graph

2 CALIBRATE

3 VALIDATE

Validation Data: validationDatastore - pixelLa...

Hardware Settings Quantization Options Quantize and Validate Export

Getting Started Calibration Statistics

Layer level quantization knobs

Dynamic range visualization

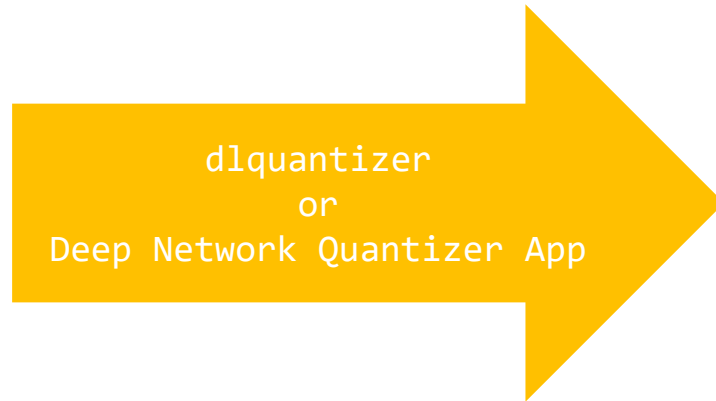
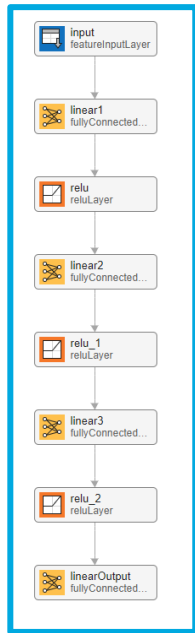
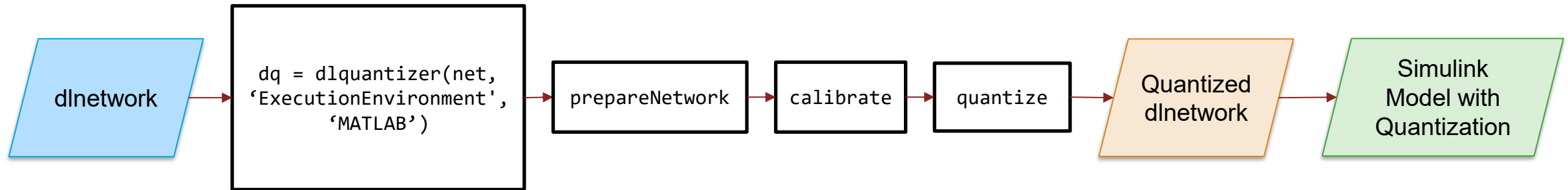
Validation Results

Resources report

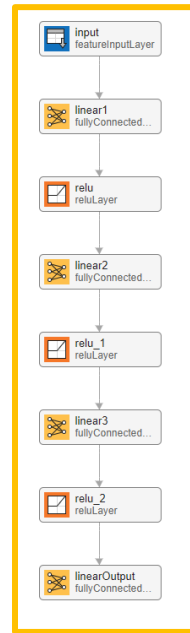
Number of samples: 1

Metric	Floating-Point Network Results	Quantized Network Results	Percent Change
Learnable parameter memory (MB)	127.2840	42.5567	66.5655
hComputeSemanticSeg	Nonscalar or Nonnumeric data	Nonscalar or Nonnumeric data	Nonscalar or Nonnumeric data

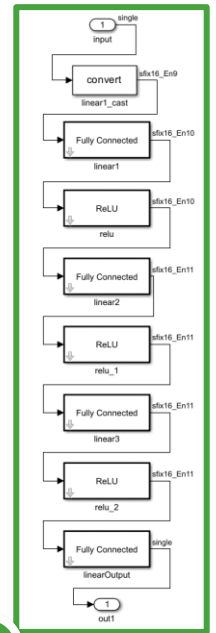
INT8 simulation and code generation with Simulink Layer Blocks



Fixed-Point Simulation
in MATLAB

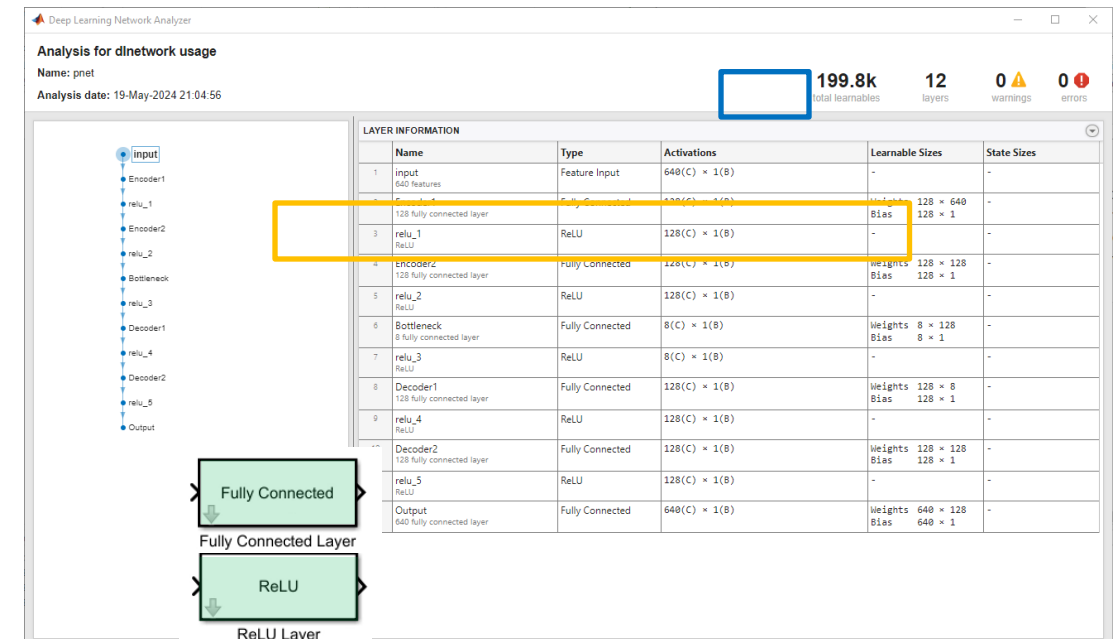
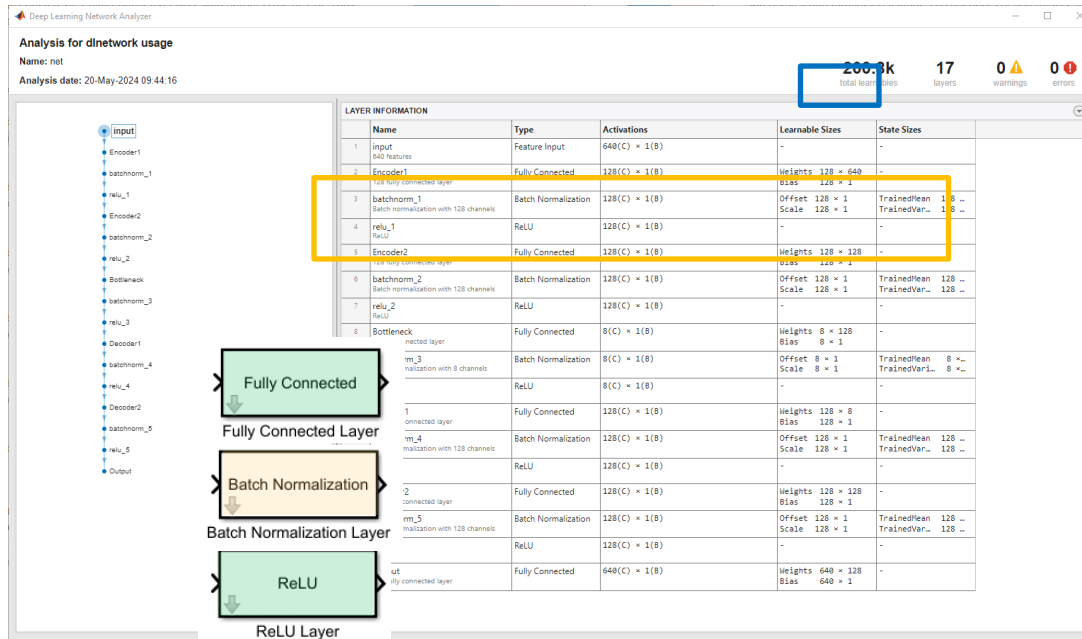


Fixed-Point Simulation
and Code Generation
in Simulink



Step 1: Prepare Network

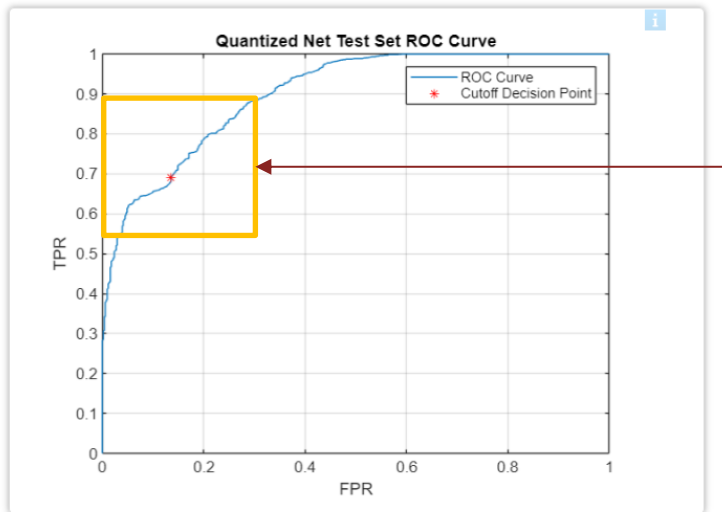
```
>> dq = dlquantizer(net, ExecutionEnvironment='MATLAB');
>> dq.prepareNetwork();
```



- Benefits to Preparation
- Export to Simulink
- Improved accuracy
- Avoid error conditions
- Layer fusion
- Projection to quantization

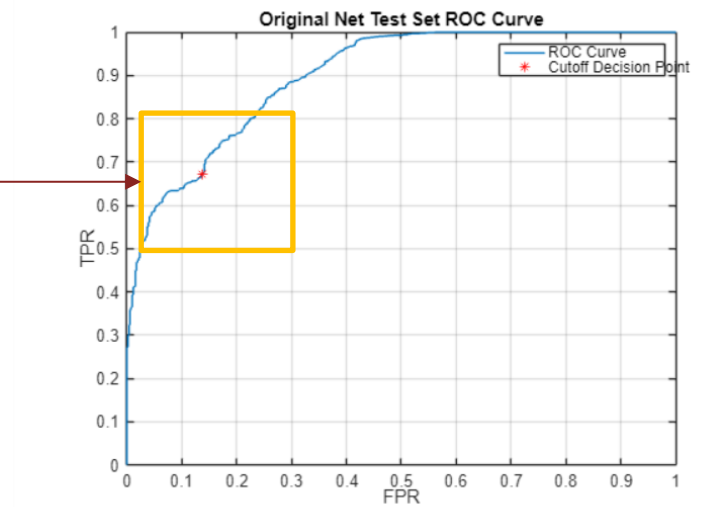
Step 2: Use calibration data for quantization

```
>> dq.calibrate(representativeData);
>> qnet = dq.quantize(ExponentScheme='MinMax');
>> ypred = predict(qnet, testData);
```



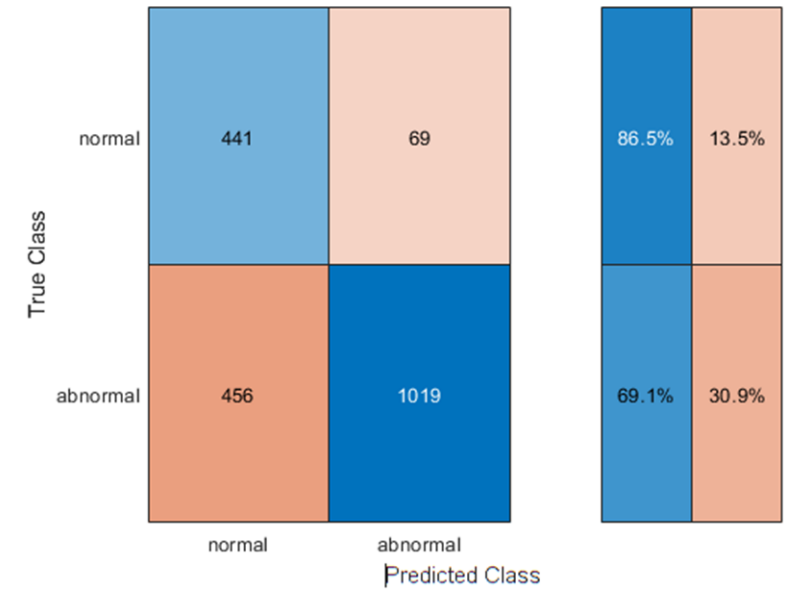
AUC

AUC = single
0.8994



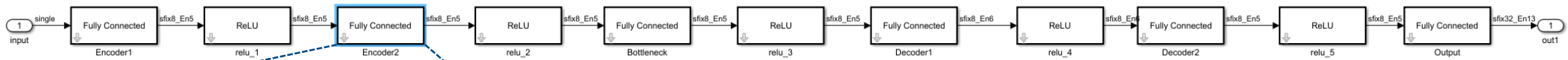
AUC

AUC = single
0.8987



Step 3: Simulate in Simulink

```
>> exportNetworkToSimulink(qnet);
```



Block Parameters: Encoder2

Fully Connected Layer (mask) (link)
Fully connected layer

Main Data Types Execution

Output minimum: [] Output maximum: []

Output data type: `fixdt(1,8,5)`

Lock data type settings against changes by the fixed-point tools

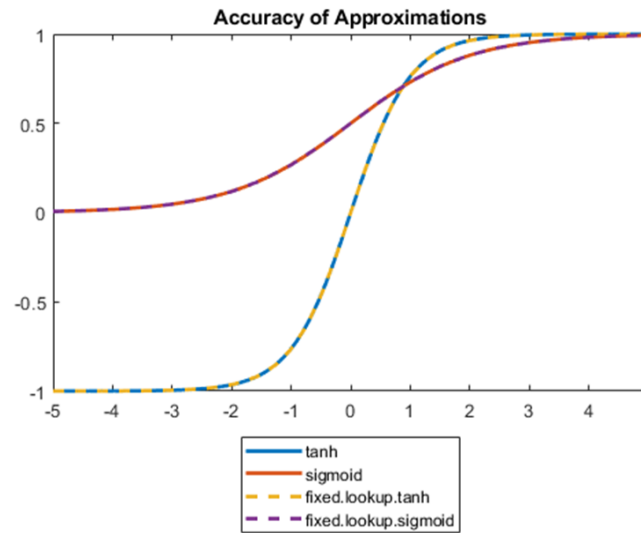
Integer rounding mode: Floor

Saturate on integer overflow

Additional data type

Data Type	Minimum	Maximum
Weights: <code>fixdt(1,8,8)</code>	[]	[]
Bias: <code>fixdt(1,32,13)</code>	[]	[]
Matrix multiply: <code>fixdt(1,32,13)</code>	[]	[]

OK Cancel Help Apply



Numerically comparable

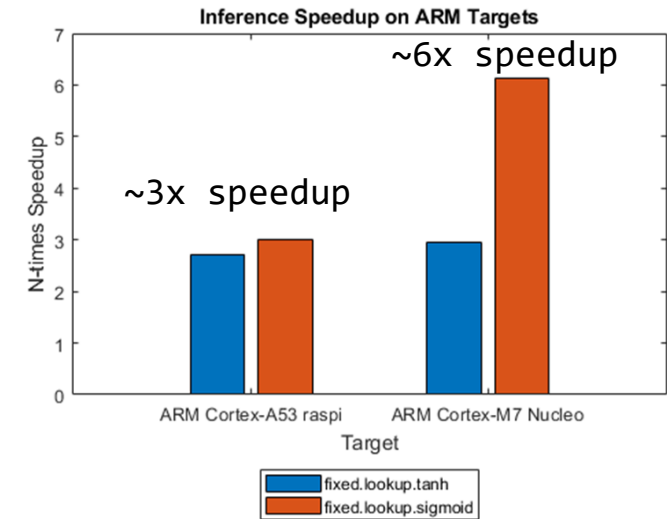
Step 4: Generate C or C++ Code for ARM Cortex A or Cortex M

The screenshot shows the MATLAB Simulink environment with a neural network model on the left and its generated C code on the right. The model consists of the following blocks: Rescale-Symmetric 1D, sequenceinput_normalization, LSTM (lstm_1), Dropout (dropout_1), and an output block. The C code window shows the implementation of the LSTM block, with the following key sections highlighted:

```

29274 0.00564412819F) / BatterySOC_LSTM_ConstB.SubtractMin[2], 1) - 1.0F, -
29275 1.0F);
29276
29277 /* End of Outputs for SubSystem: '<S112>/ForEach' */
29278 /* End of Outputs for SubSystem: '<S110>/Rescale-Symmetric_ForEachChan
29279 /* End of Outputs for SubSystem: '<S1>/sequenceinput_normalization' */
29280
29281 /* Outputs for Atomic SubSystem: '<S1>/lstm_1' */
29282 /* Product: '<S22>/W*x' incorporates:
29283 * Constant: '<S22>/InputWeights'
29284 * ForEachSliceAssignment generated from: '<S113>/Out1'
29285 */
29286 memset(&rtb_Wx_b[0], 0, sizeof(real32_T) << 9U);
29287 for (i_0 = 0; i_0 < 3; i_0++) {
29288     rtb_MatrixMultiply_0 = rtb_ImpAsg_InsertedFor_Out1_at_[i_0];
29289     for (i = 0; i <= 508; i += 4) {
29290         tmp_2 = _mm_loadu_ps(&rtb_Wx_b[i]);
29291         _mm_storeu_ps(&rtb_Wx_b[i], _mm_add_ps(_mm_mul_ps(_mm_loadu_ps
29292 (&BatterySOC_LSTM_ConstP.InputWeights_Value[(i_0 << 9) + i]),
29293 _mm_set1_ps(rtb_MatrixMultiply_0)), tmp_2));
29294     }
29295 }
29296
29297 /* End of Product: '<S22>/W*x' */
29298
29299 /* Outputs for Iterator SubSystem: '<S7>/ForIteratorSubsystem' incorpor
29300 * ForIterator: '<S21>/ForIterator'
29301 */
29302 i_0 = BatterySOC_LSTM_B.ProbeDimension_e[1];
29303 if (BatterySOC_LSTM_B.ProbeDimension_e[1] > 2147483646) {
29304     i_0 = 2147483646;
29305 } else if (BatterySOC_LSTM_B.ProbeDimension_e[1] < 0) {

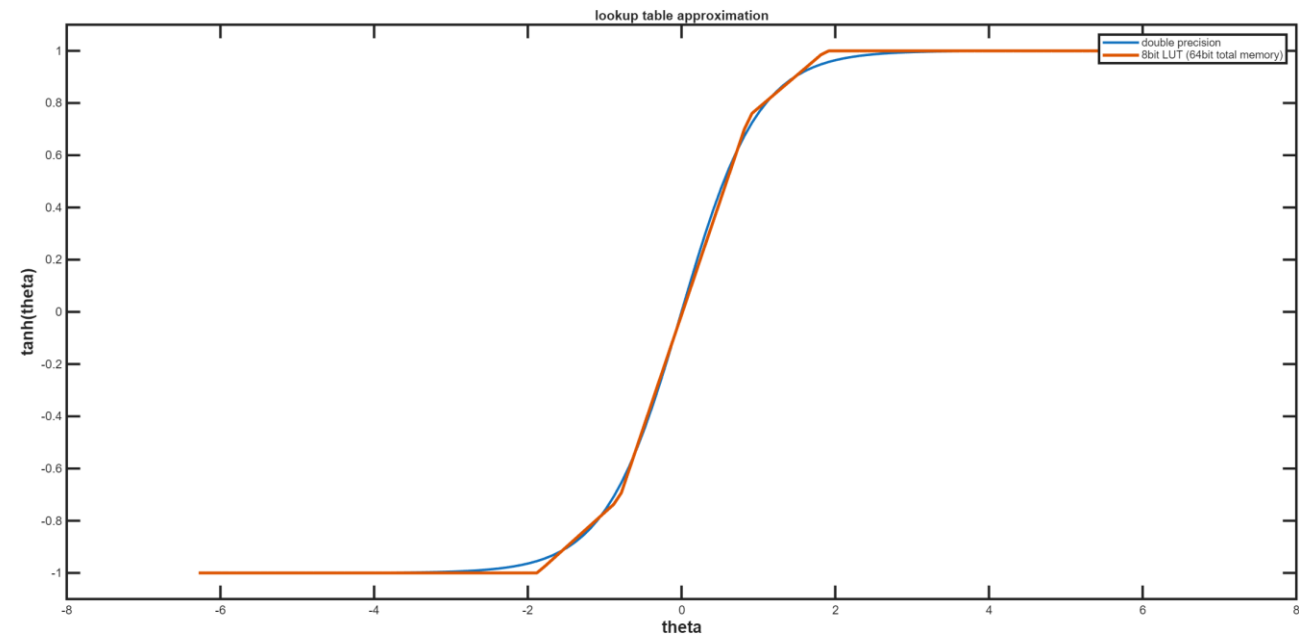
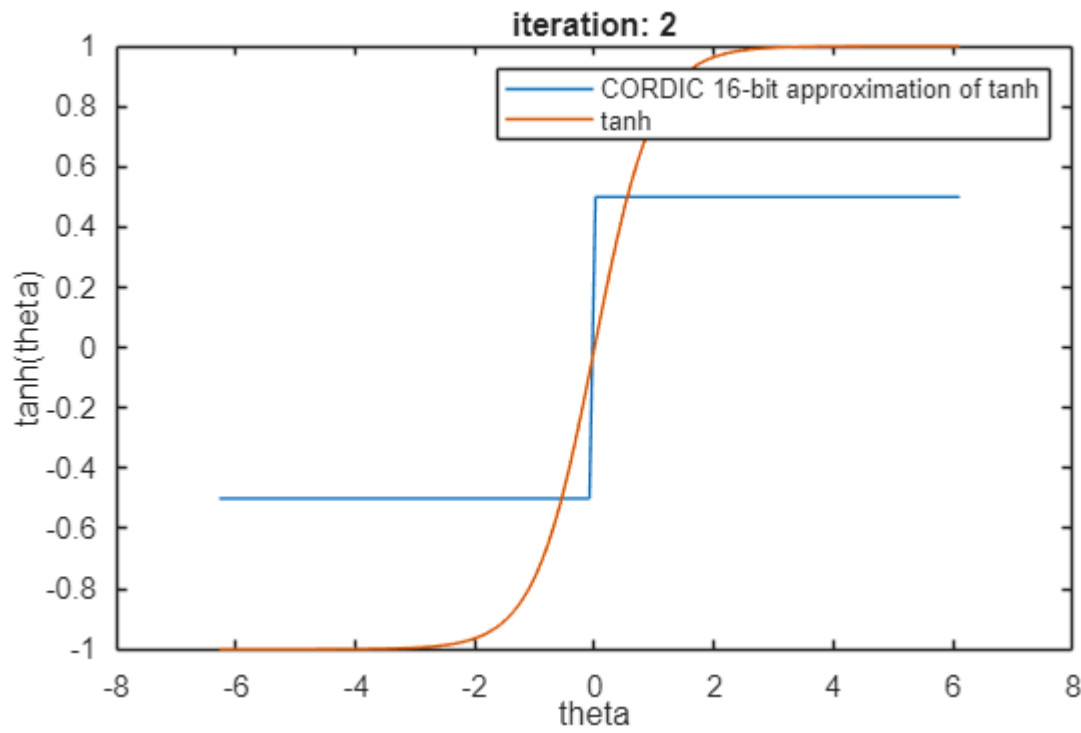
```



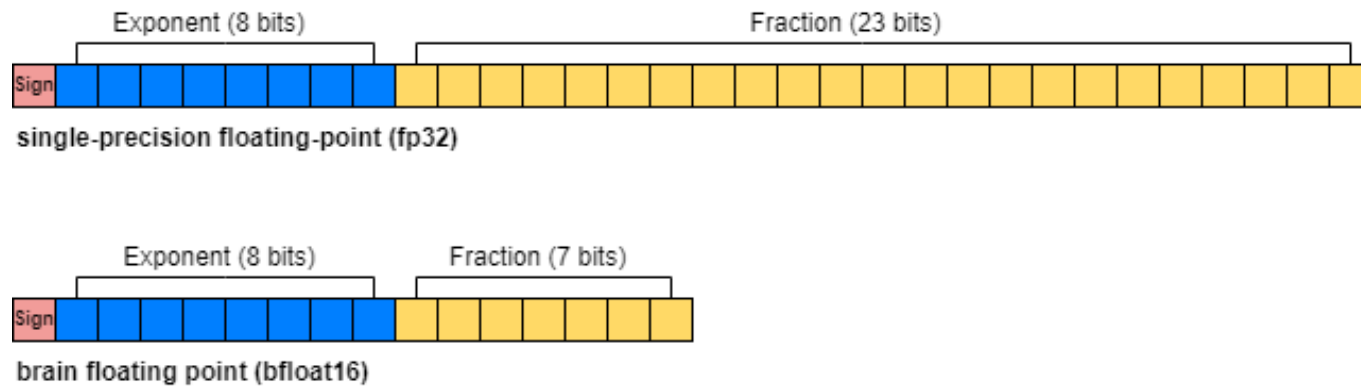
Faster on target

A word on activation functions

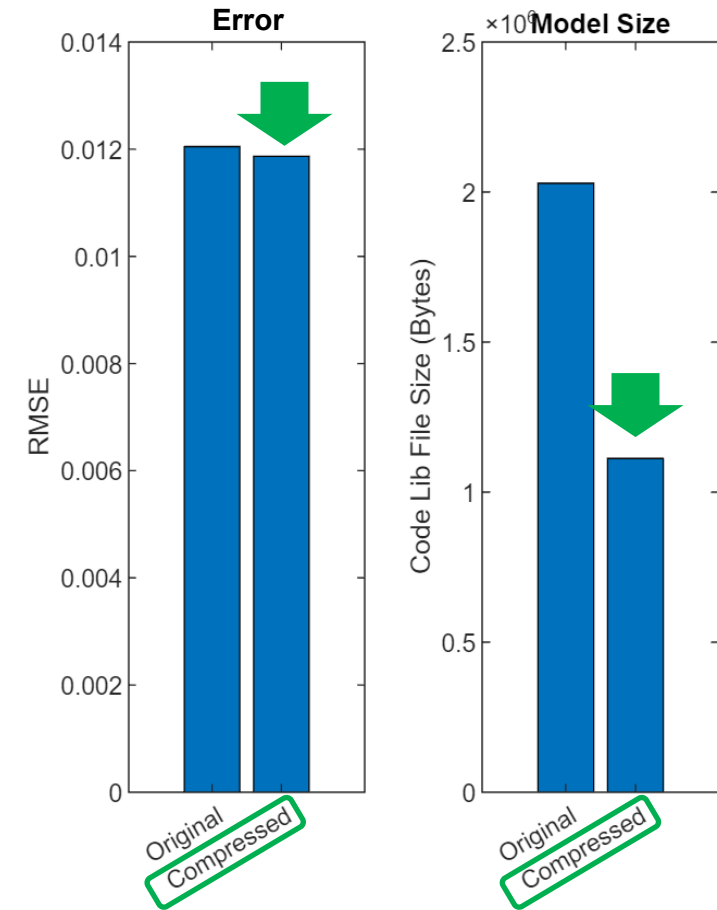
- Relu is straightforward to do in fixed-point, but what about tanh, sigmoid, ... ?
- In practice, typically we use look-up table approximations
 - ⇒ can control the required accuracy and memory cost
- Another option are CORDIC algorithms (for sigmoid/tanh)



Model Compression with bfloat16 Data Type Conversion



- Reduce memory footprint by ~50% without much loss in accuracy
- The process does not need data or preprocessing step or retraining



Generate bfloat16 Code for Deep Learning Networks

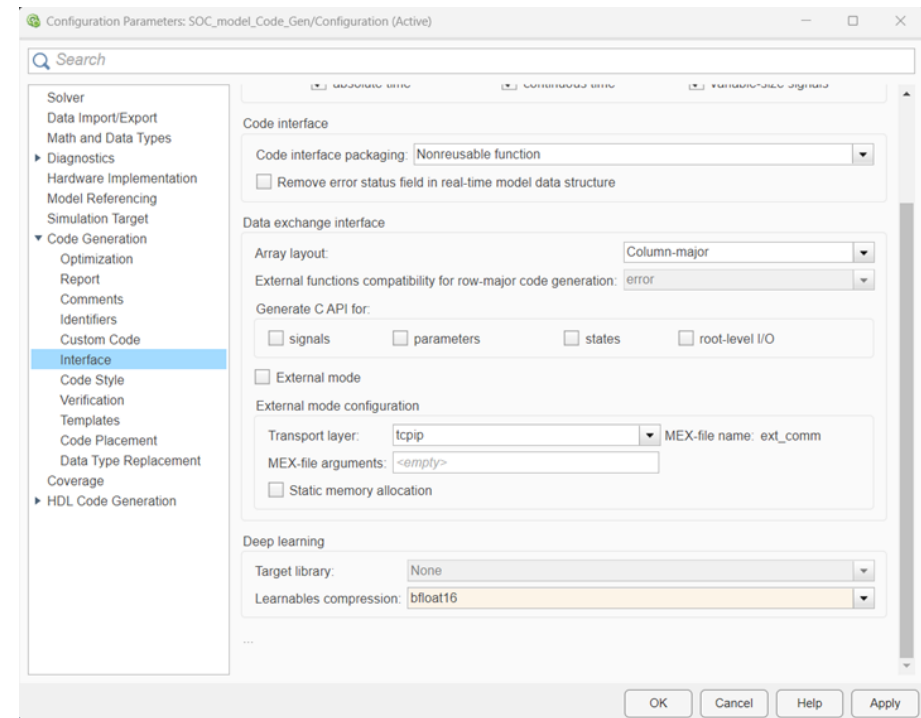
- Supported for plain C/C++ code generation

```
>> cfg = coder.config('lib');
```

```
>> cfg.DeepLearningConfig.LearnablesCompression = 'bfloat16';
```

- Supported layers and classes

- fullyConnectedLayer
- gruLayer (and gruProjectedLayer)
- lstmLayer (and lstmProjectedLayer)
- bilstmLayer
- groupedConvolution2dLayer
- convolution2dLayer (R2025a)



Agenda

- Code Generation for Deep Learning Networks
 - Network Compression Techniques
 - Structural Compression
 - Datatype Compression (incl. Quantization)
 - **Techniques for Optimizing Performance**
- Code Generation for other Machine Learning Models
 - Overview

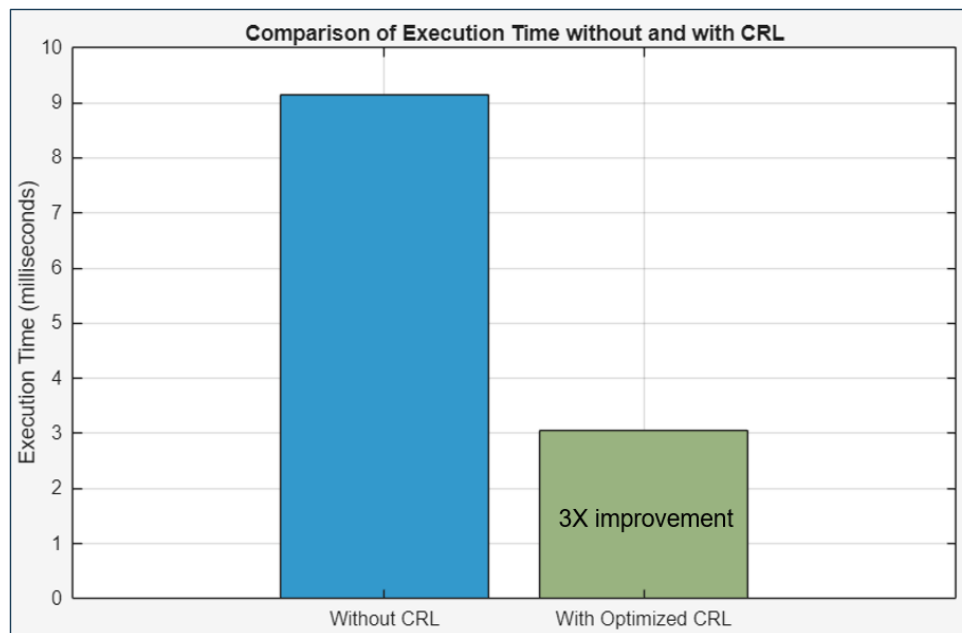
Techniques for Optimizing Performance

- **Vectorization** and **multi-threading improve** the performance of edge AI
 - Both improve resource utilization, leading to faster task completion.
- Vectorization executes the same instruction on multiple data elements simultaneously
- Multi-threading divides a workload into threads for concurrent execution

Vectorization support for ARM Cortex-A and -M

- Vectorization through SIMD intrinsics in code replacement libraries

```
>> cfg = coder.config('lib');  
>> cfg.HardwareImplementation.ProdHWDeviceType = 'ARM Compatible -> ARM Cortex M';  
>> cfg.CodeReplacementLibrary = 'ARM Cortex M (CMSIS)';
```



Running a digit classification network (MLP) on ARM Cortex M (STM32F746G-Discovery), with and without usage of code replacement libs

ARM Cortex A with Neon instructions and Intel with AVX instructions

```

for (k = 0; k < K; k++) {
    float32x4_t a;
    float32x4_t b;
    float32x4_t b_a;
    float32x4_t b_b;
    float32x4_t c_b;
    float32x4_t d_b;
    a = vld1q_f32(&A[idxA]);
    b_a = vld1q_f32(&A[idxA + 4]);
    b = vdupq_n_f32(B[idxB]);
    b_b = vdupq_n_f32(B[idxB + LDB]);
    c_b = vdupq_n_f32(B[idxB + (LDB << 1)]);
    d_b = vdupq_n_f32(B[idxB + LDB * 3]);
    c = vmlaq_f32(c, a, b);
    b_c = vmlaq_f32(b_c, b_a, b);
    c_c = vmlaq_f32(c_c, a, b_b);
    d_c = vmlaq_f32(d_c, b_a, b_b);
    e_c = vmlaq_f32(e_c, a, c_b);
    f_c = vmlaq_f32(f_c, b_a, c_b);
    g_c = vmlaq_f32(g_c, a, d_b);
    h_c = vmlaq_f32(h_c, b_a, d_b);
    idxA += LDA;
    idxB++;
}

```

```

for (k = 0; k < K; k++) {
    __m256 a;
    __m256 b;
    __m256 b_a;
    __m256 b_b;
    __m256 c_b;
    __m256 d_b;
    a = _mm256_loadu_ps(&A[idxA]);
    b_a = _mm256_loadu_ps(&A[idxA + 8]);
    b = _mm256_set1_ps(B[idxB]);
    b_b = _mm256_set1_ps(B[idxB + LDB]);
    c_b = _mm256_set1_ps(B[idxB + (LDB << 1)]);
    d_b = _mm256_set1_ps(B[idxB + LDB * 3]);
    c = _mm256_add_ps(c, _mm256_mul_ps(a, b));
    b_c = _mm256_add_ps(b_c, _mm256_mul_ps(b_a, b));
    c_c = _mm256_add_ps(c_c, _mm256_mul_ps(a, b_b));
    d_c = _mm256_add_ps(d_c, _mm256_mul_ps(b_a, b_b));
    e_c = _mm256_add_ps(e_c, _mm256_mul_ps(a, c_b));
    f_c = _mm256_add_ps(f_c, _mm256_mul_ps(b_a, c_b));
    g_c = _mm256_add_ps(g_c, _mm256_mul_ps(a, d_b));
    h_c = _mm256_add_ps(h_c, _mm256_mul_ps(b_a, d_b));
    idxA += LDA;
    idxB++;
}

```

Multi-Threading support in the generated code

- For targets that support multi-threading, extend calls to the OpenMP API in the generated code. For example, for a Raspberry Pi:

```
>> cfg = coder.config('lib');  
>> cfg.Hardware = coder.Hardware('Raspberry Pi');  
>> cfg.CodeReplacementLibrary = "GCC ARM Cortex-A";  
>> cfg.EnableOpenMP = true;
```

Agenda

- **Code Generation for Deep Learning Networks**
 - Network Compression Techniques
 - Structural Compression
 - Datatype Compression (incl. Quantization)
 - Techniques for Optimizing Performance
- **Code Generation for other Machine Learning Models**
 - Overview

C/C++ Code Generation for other machine Learning Algorithms

- Machine Learning Algorithms Supported for C/C++ Code Generation
 - **Anomaly Detection:** Isolation Forest, One-class SVM
 - **Classification:** Binary Decision Tree, Discriminant Analysis, Ensemble, Incremental Linear, Nearest Neighbor (KNN), Linear, Multiclass (ECOC), Naive Bayes, SVM
 - **Regression:** Ensemble, Gaussian Process (GP), Linear, Incremental Linear, Tree and Tree Ensembles, SVM
 - **Nearest Neighbor Searcher:** ExhaustiveSearcher, KDTree
 - **Incremental (On-Device) Learning:** Regression and Classification

Key takeaways

- Deploy AI models to any hardware using automatic C/C++ code generation
- Reduce model size using compression techniques: projection & quantization
- Optimize performance of generated code via vectorization & multi-threading

MathWorks
**AUTOMOTIVE
CONFERENCE**
Europe

14 April 2026 | Mövenpick Hotel Stuttgart Messe

Register at mathworks.com/mac

