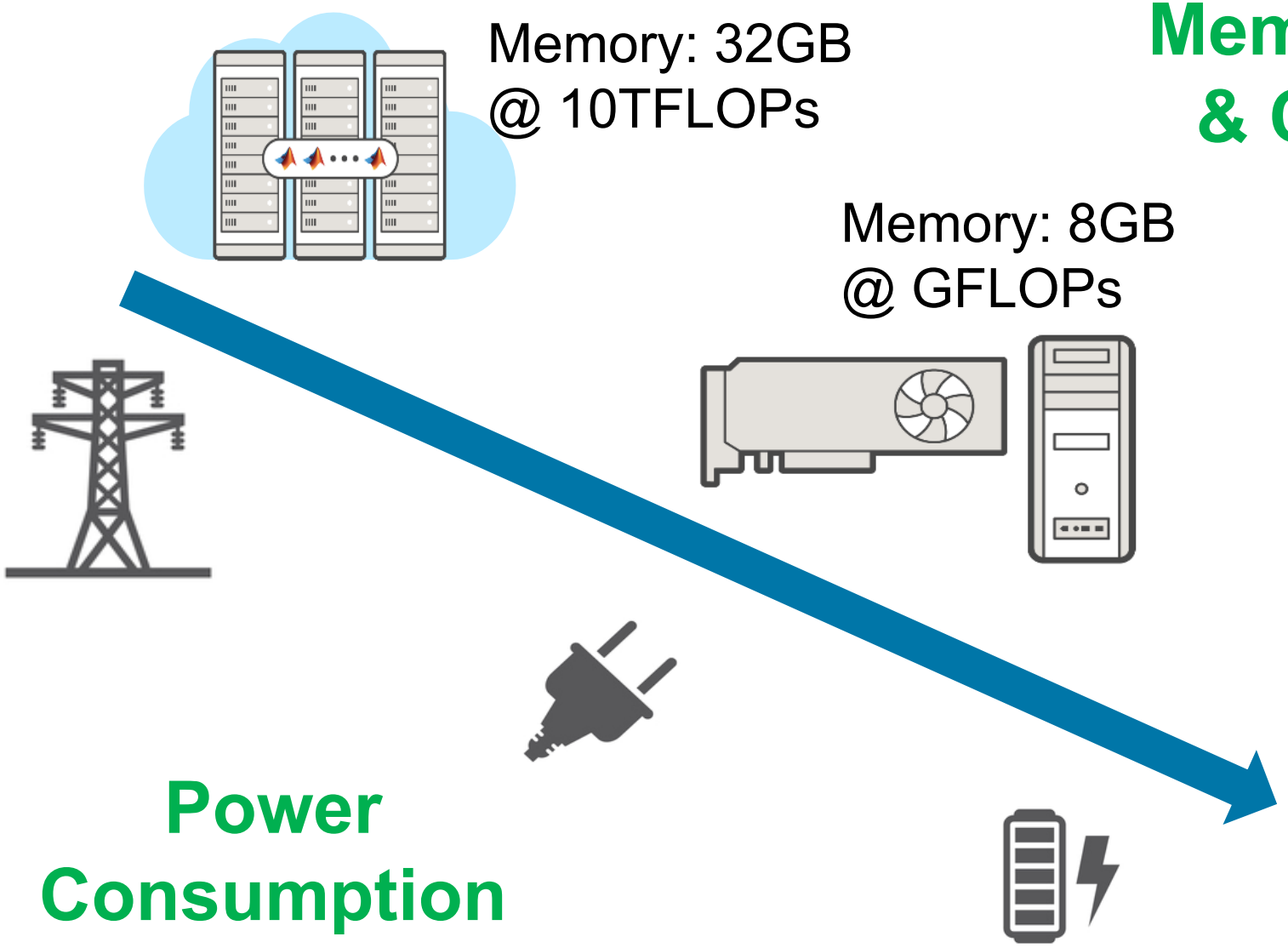
Embedded AI: Compression techniques and performance optimization using automated C/C++ code generation

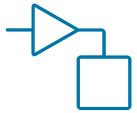
Christoph Stockhammer
MathWorks Application Engineering

Embedded devices are resource-constrained



Embedded AI Workflow





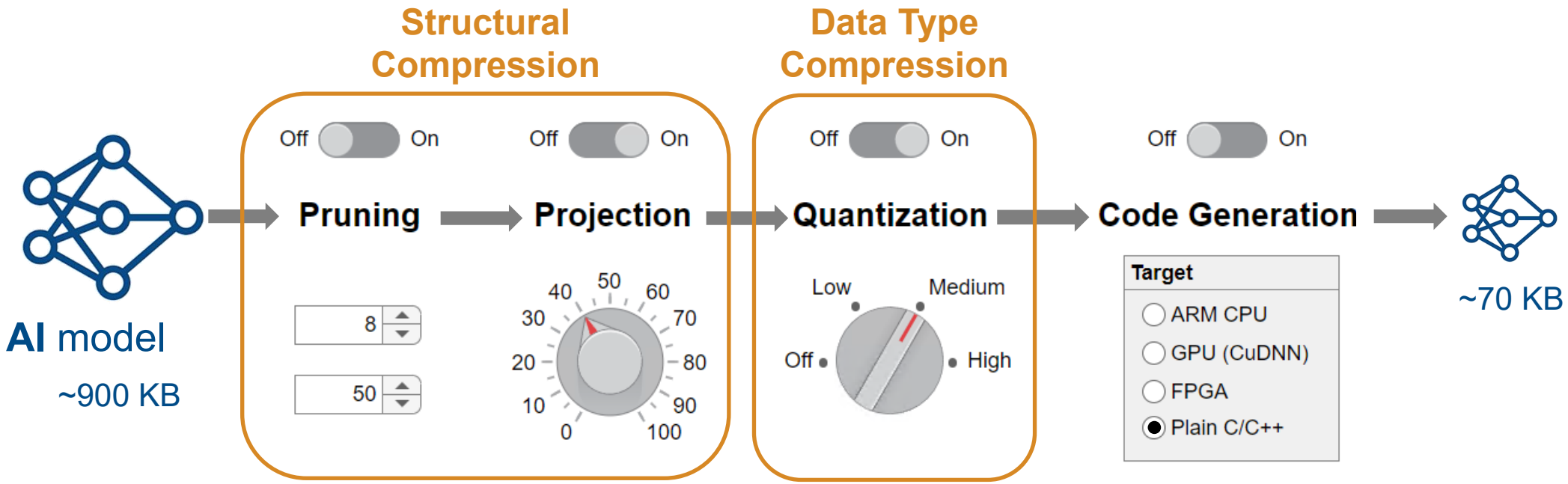

Data Preparation	AI Modeling	Simulation & Test	Compression	Deployment
Toolboxes for **data** pre-processing	**Low-code** AI modeling & training through MATLAB Apps Model import from **TensorFlow**, **PyTorch**, or other frameworks	**Simulink blocks for AI inference** make integration easy **Simulation-based** testing	**Model compression techniques** to reduce model size and speed up inference	**Code generation** from deep learning and machine learning models for embedded targets

PyTorch TensorFlow

3

Compression bridges the gap between AI modelling and embedded deployment

Reduce model footprint and accelerate inference of AI models



4

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

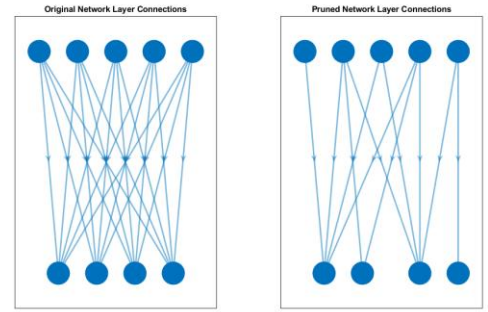
Agenda

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

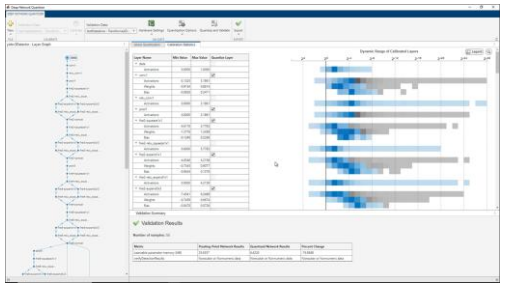
- Code Generation for other Machine Learning Models
  - Overview

Deploy efficient AI to any processor with code generation

Model Compression



Structural



Data Type

Code Generation

CPU

Any CPU
No Library needed

oneDNN
Library

ARM Compute
Library

GPU

µC

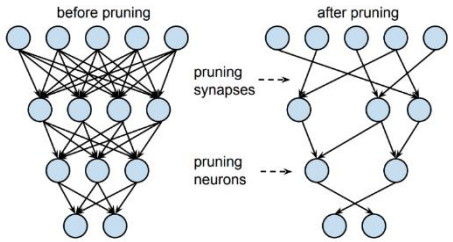
FPGA

Agenda

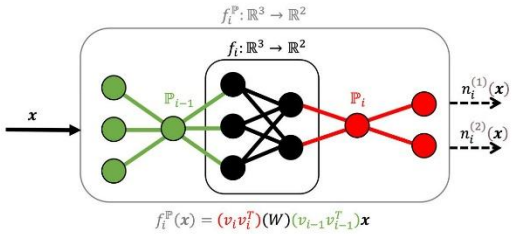
- Code Generation for Deep Learning Networks
  - **Network Compression Techniques**
    - Structural Compression
    - Datatype Compression (incl. Quantization)
  - Techniques for Optimizing Performance
- Code Generation for other Machine Learning Models
  - Overview

Compression bridges the gap between modeling and deployment

Structural Compression

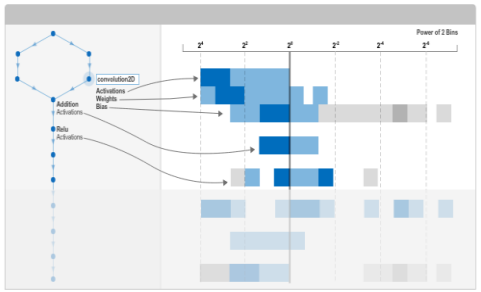


Pruning
convolution layer filters



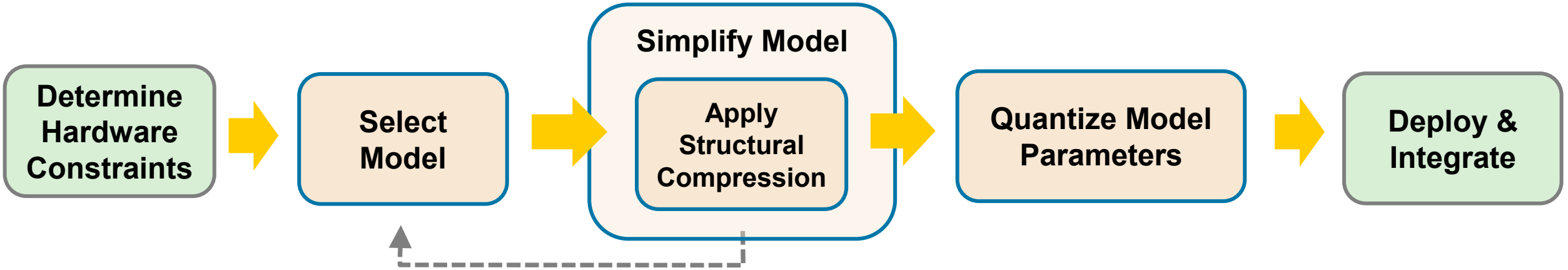
[Projection](#) of
learnables matrices

Datatype Compression



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

Workflow for Compressing Deep Neural Nets

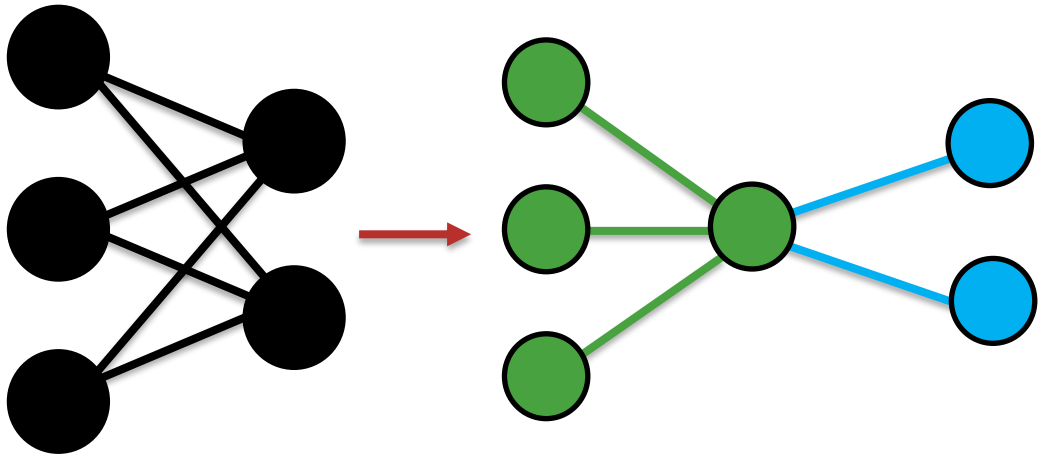


Agenda

- 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

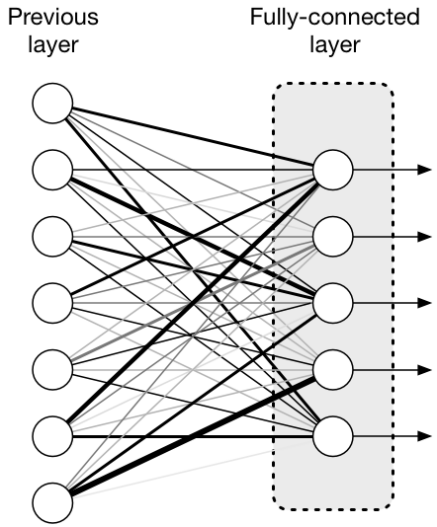
Applies principal component analysis (PCA) to weight matrices and projects them from high-dimension space to low-dimension space

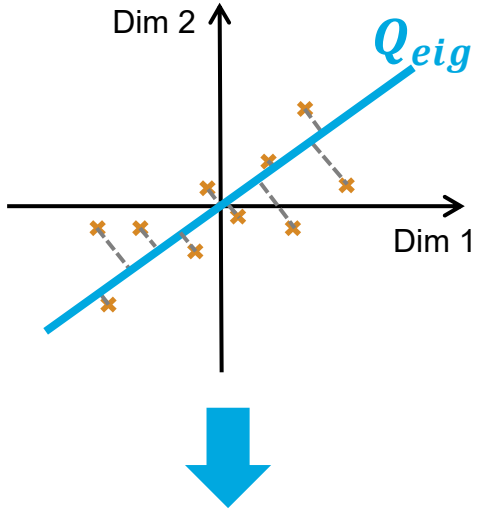


Technical article on network projection technique

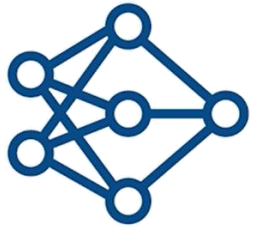
Reduce model sizes using projection

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



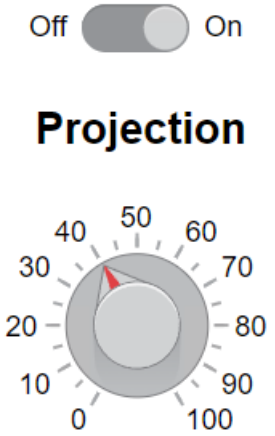


projected layer



AI model

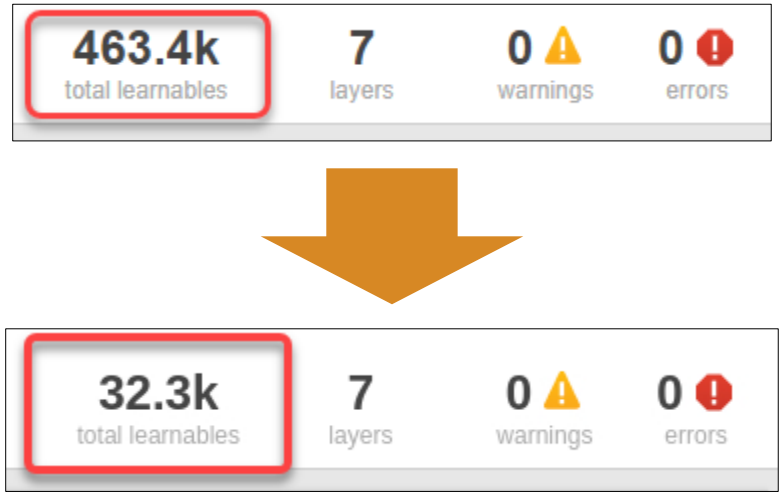
e.g. 1MB



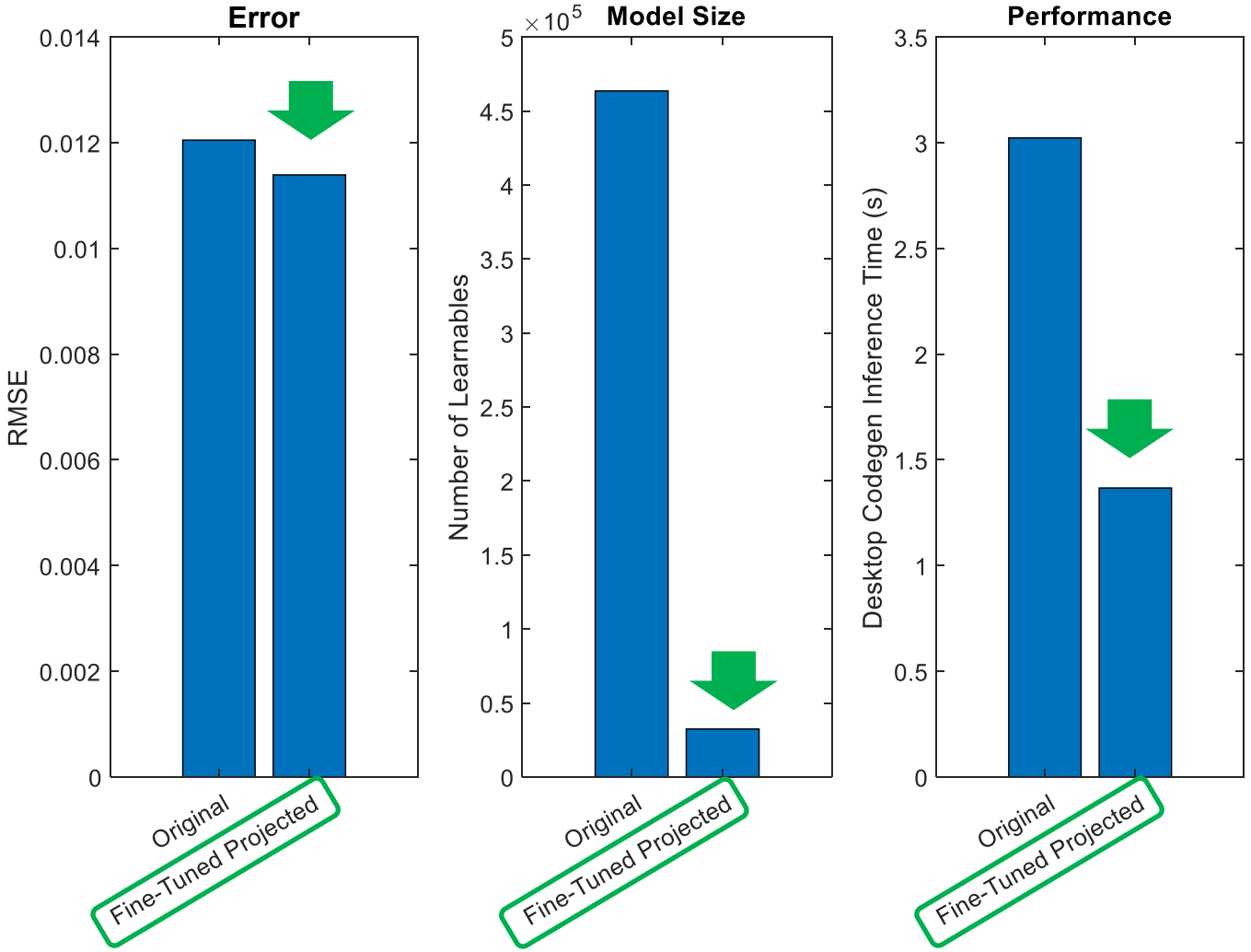
e.g. 600 KB

Projection Results – case study

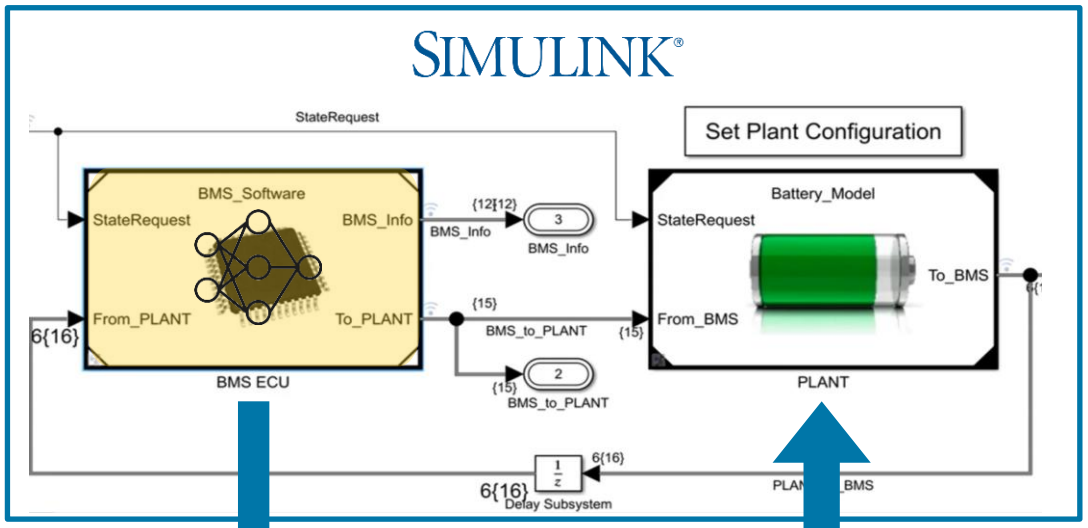
Example: LSTM Network for Battery State of Charge (SoC) Prediction



Number of total learnables decreased

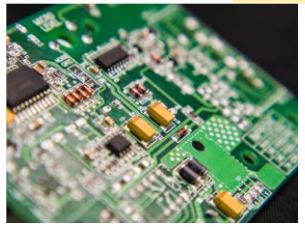


Workflow: Processor-in-the-Loop (PIL) Simulations



Code generation from algorithm

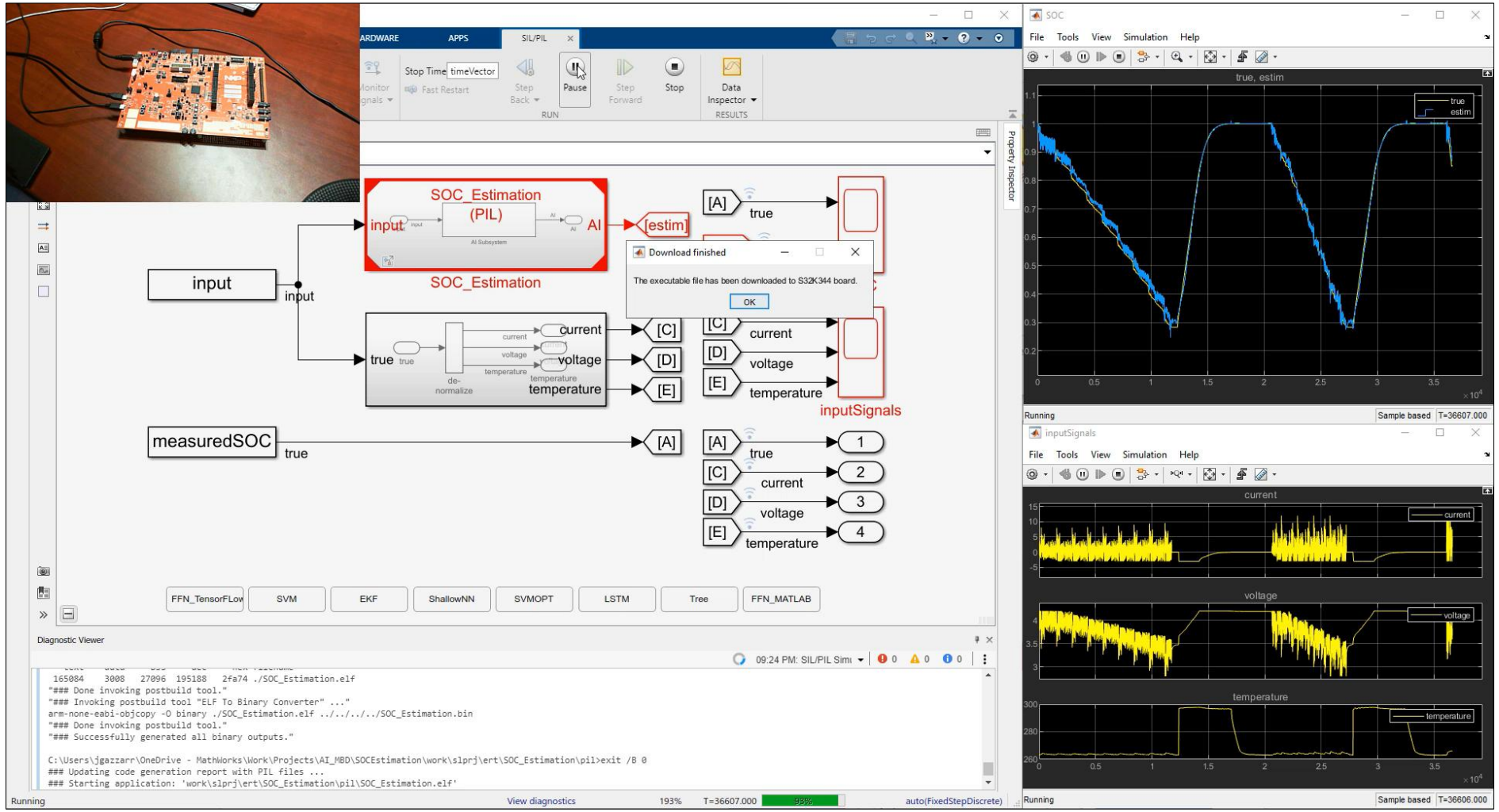
Deployed code communicates with simulated plant



NXP S32K148 board

Arm® Cortex®-M

PIL execution on ARM Cortex-M7 processor



Projection can be applied to a variety of neural networks

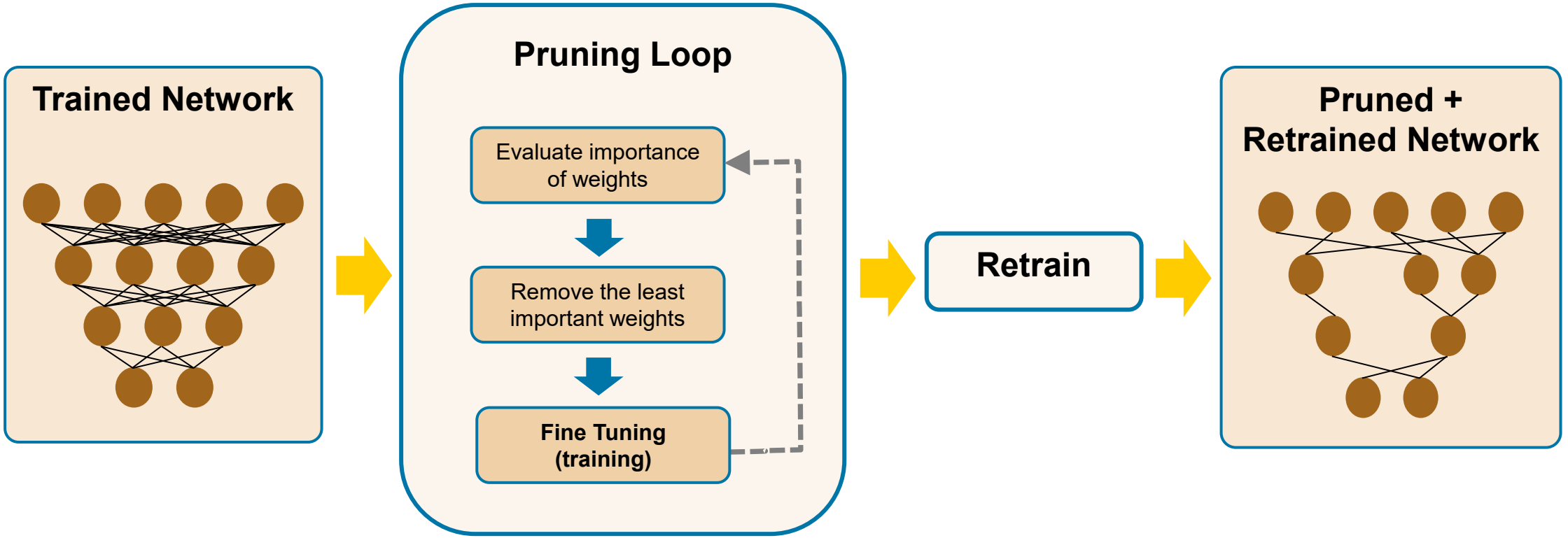
- Supported layer types:
  - **LSTM layer**
  - **GRU layer**
  - **FullyConnected layer**
  - **Convolution2D layer**

- 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

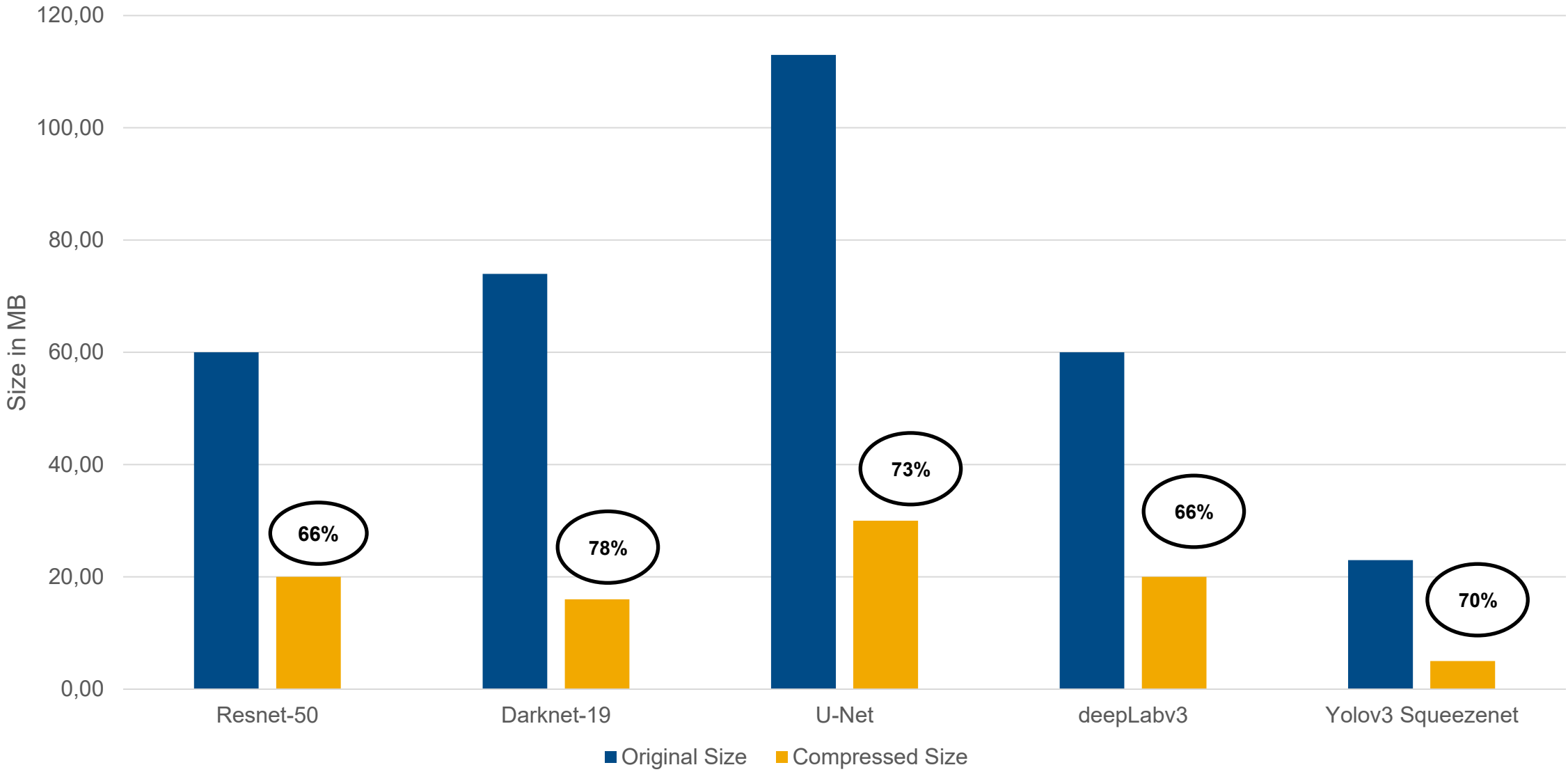
Taylor Pruning for 2D convolutions

Remove **redundant** convolution filters

Supported networks
- Semantic segmentation
- Classifiers
- Object detectors



Taylor Pruning Applied to Popular Networks

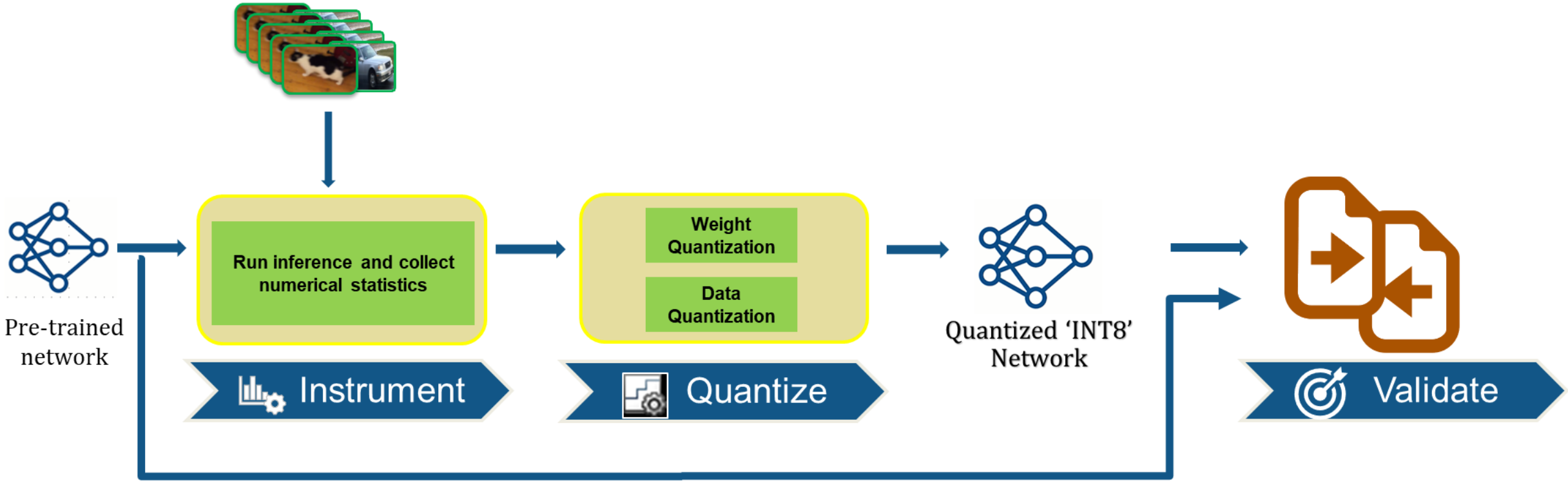


Agenda

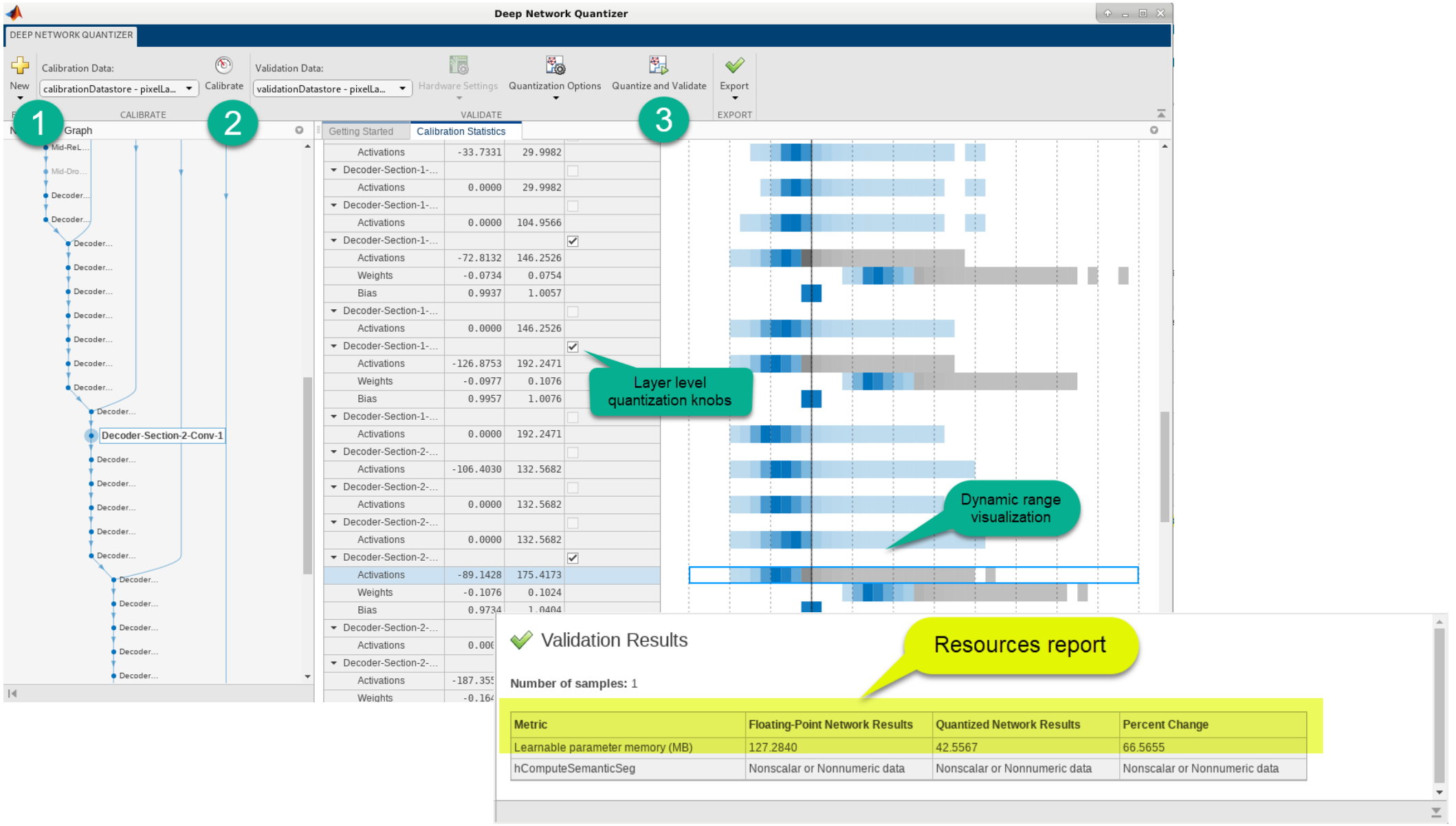
- Code Generation for Deep Learning Networks
  - Network Compression Techniques
    - Structural Compression
    - **Datatype Compression (incl. Quantization)**
  - Techniques for Optimizing Performance

- Code Generation for other Machine Learning Models
  - Overview

Deep Learning Quantization Workflow



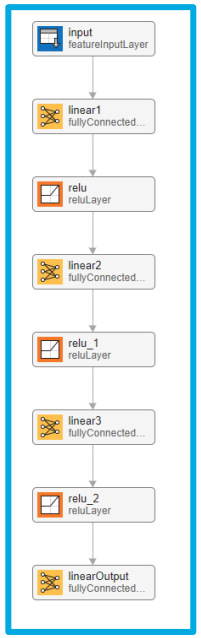
Deep Network Quantization App



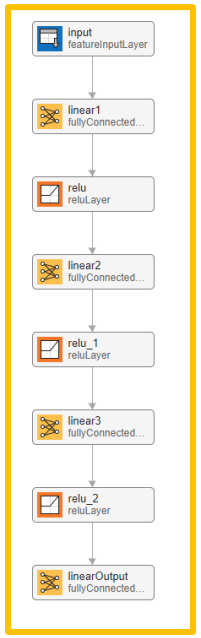
Metric	Floating-Point Network Results	Quantized Network Results	Percent Change
Learnable parameter memory (MB)	127.2840	42.5567	66.5655
hComputeSemanticSeg	Nonscalar or Nonnumeric data	Nonscalar or Nonnumeric data	Nonscalar or Nonnumeric data

INT8 simulation and code generation with Simulink Layer Blocks

dlnetwork → `dq = dlquantizer(net, 'ExecutionEnvironment', 'MATLAB')` → `prepareNetwork` → `calibrate` → `quantize` → Quantized dlnetwork → Simulink Model with Quantization

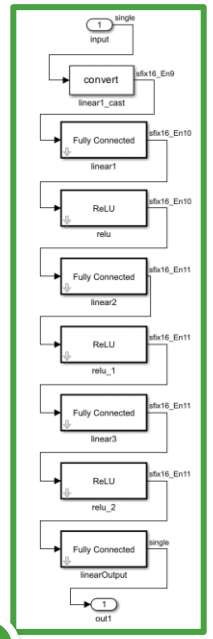


`dlquantizer`
or
Deep Network Quantizer App



Fixed-Point Simulation in MATLAB

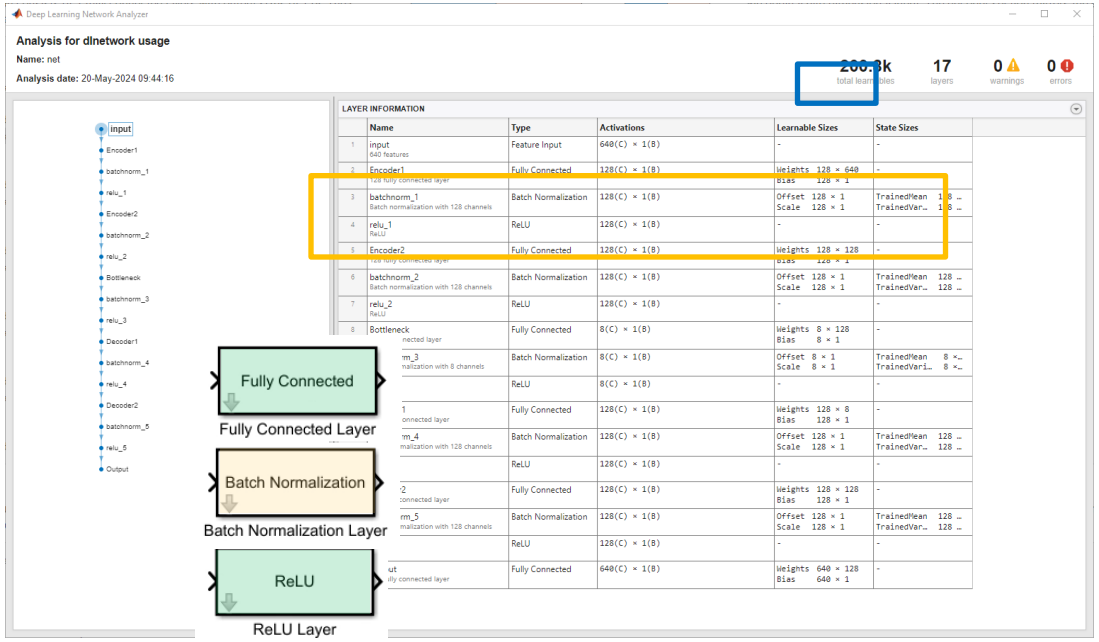
`exportNetworkToSimulink`

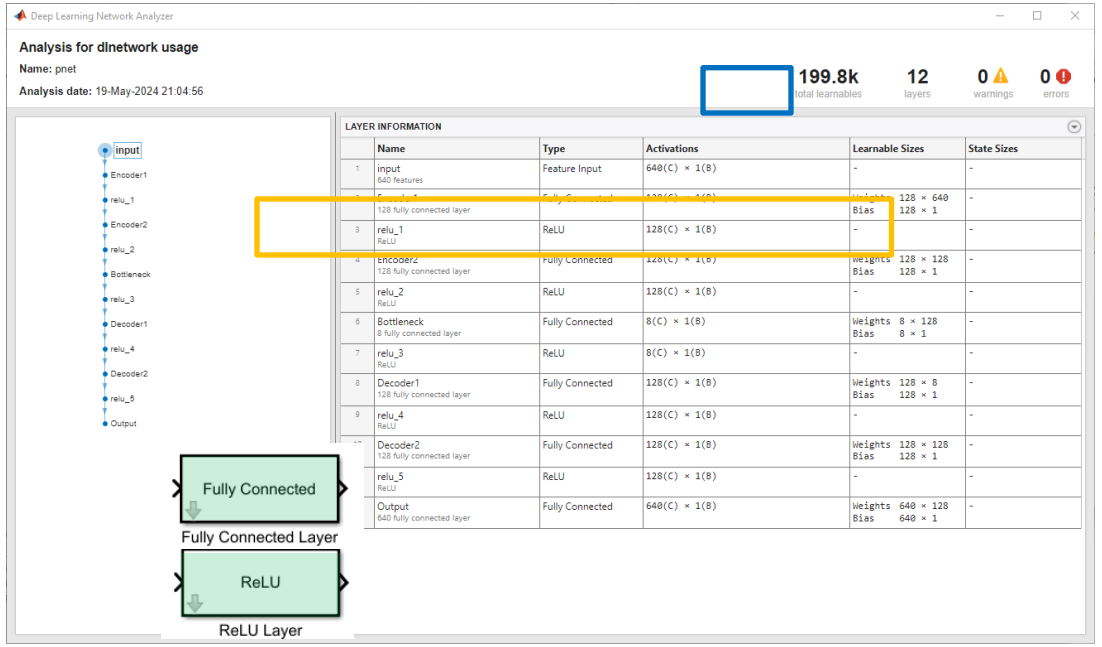


Fixed-Point Simulation and Code Generation in Simulink

Step 1: Prepare Network

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

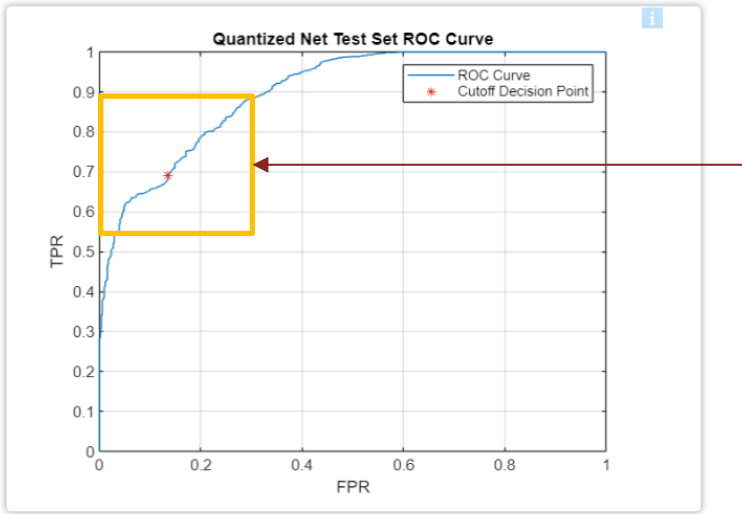




Benefits to Preparation	Export to Simulink	Improved accuracy	Avoid error conditions	Layer fusion	Projection to quantization

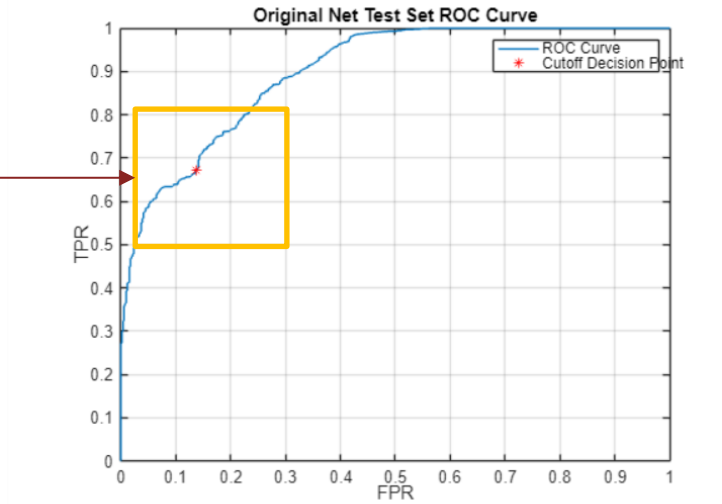
Step 2: Use calibration data for quantization

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



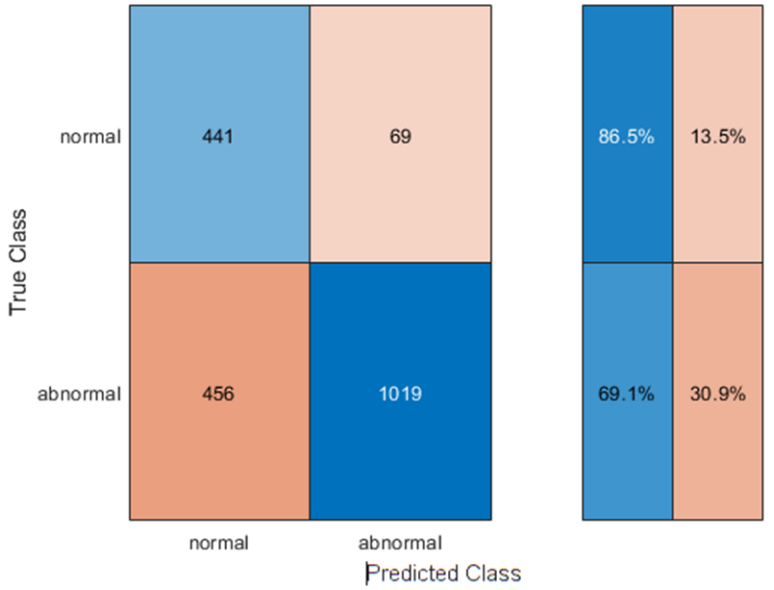
AUC

```
AUC = single
    0.8994
```



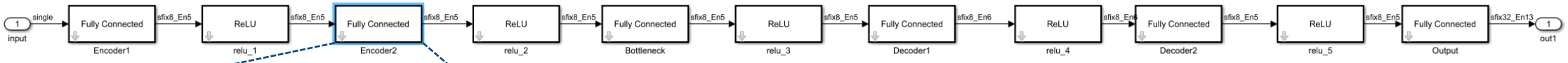
AUC

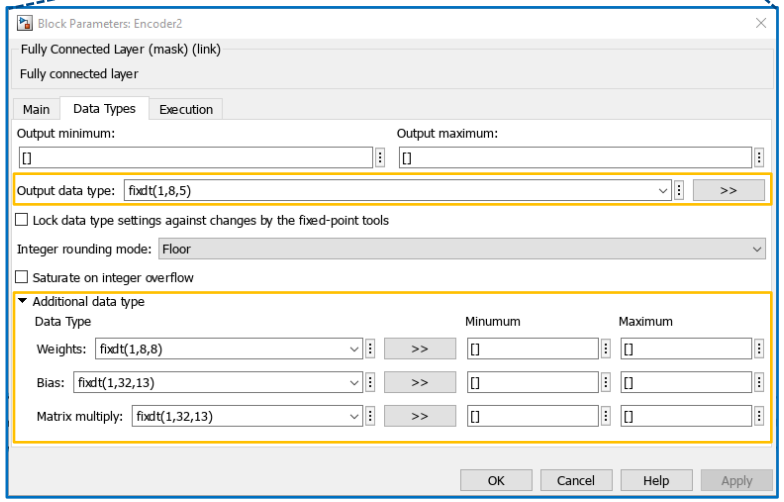
```
AUC = single
    0.8987
```

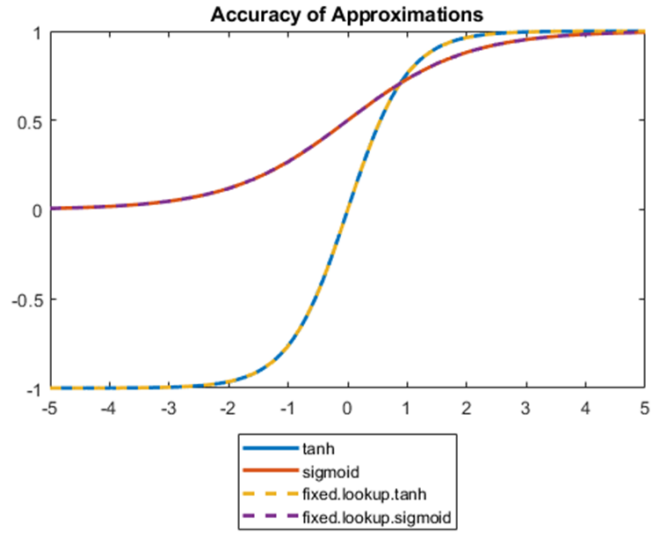


Step 3: Simulate in Simulink

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

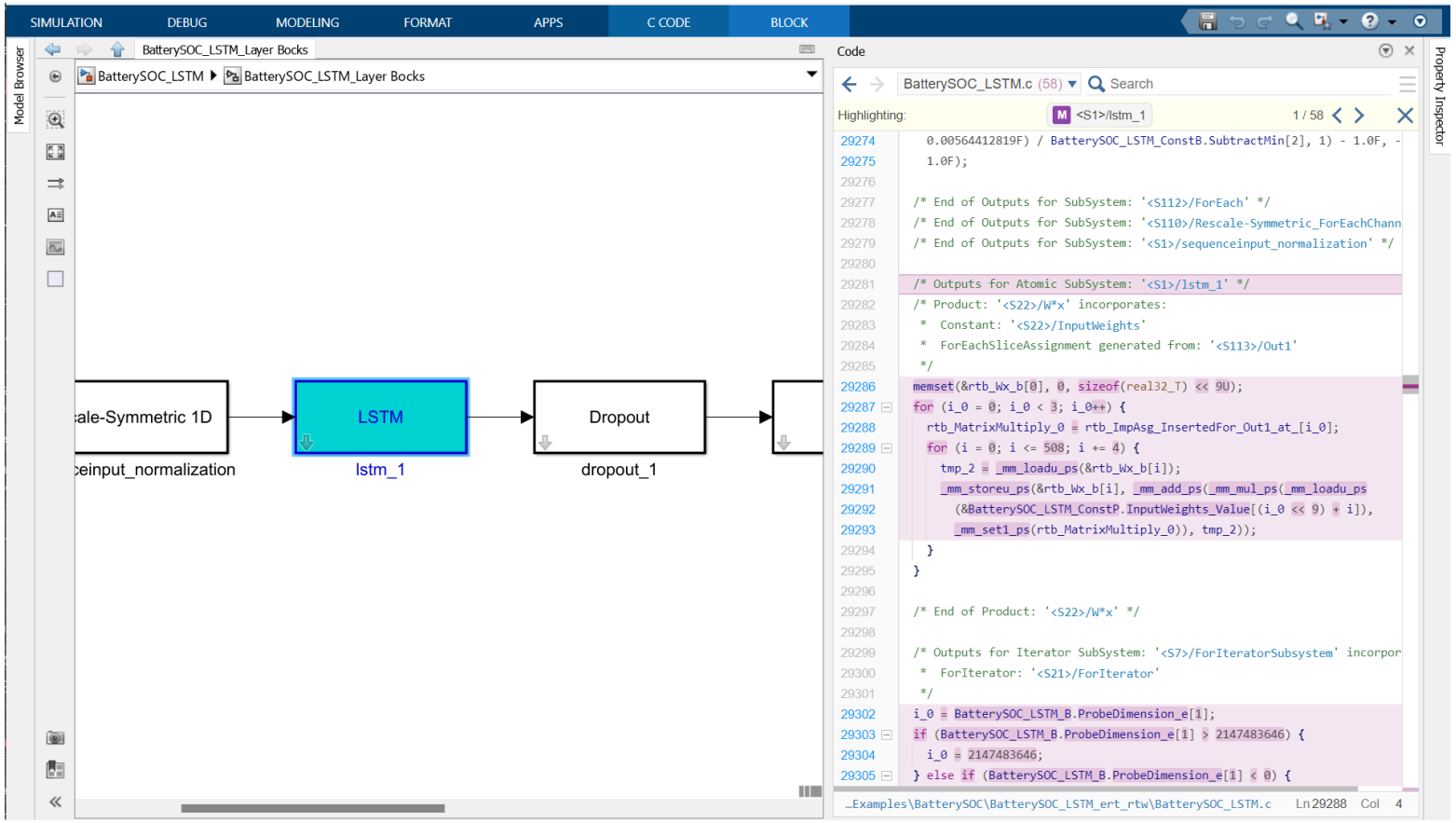


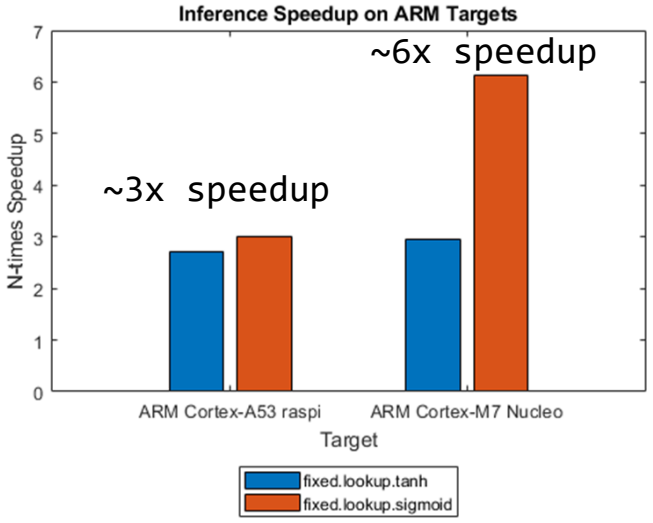




Numerically comparable

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



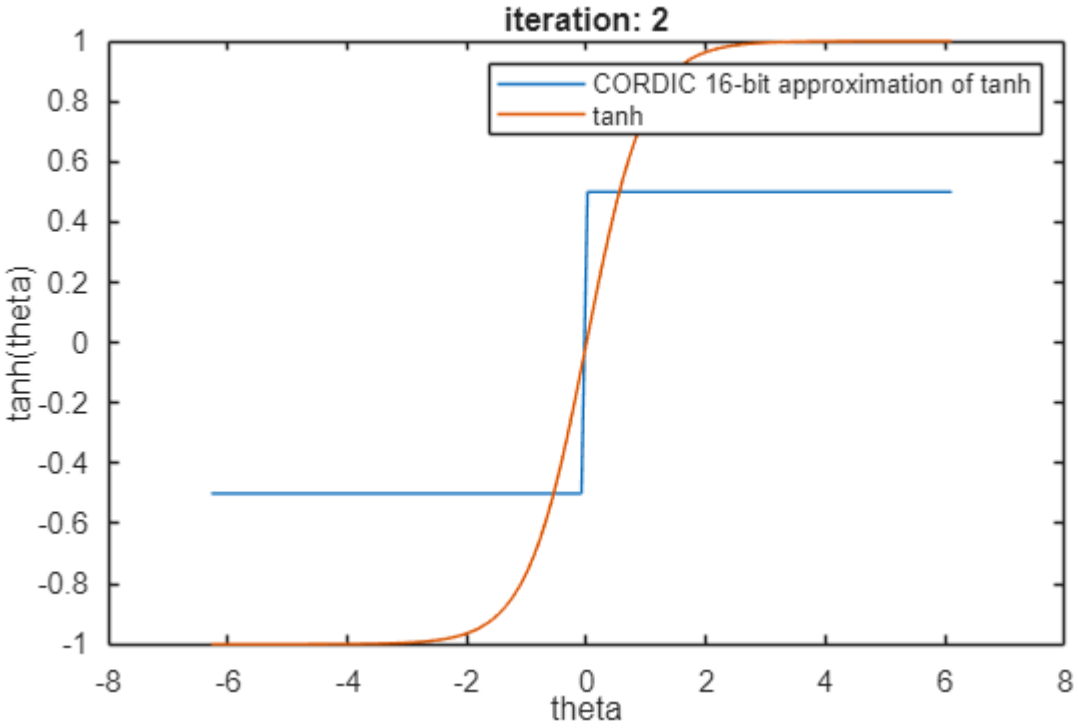


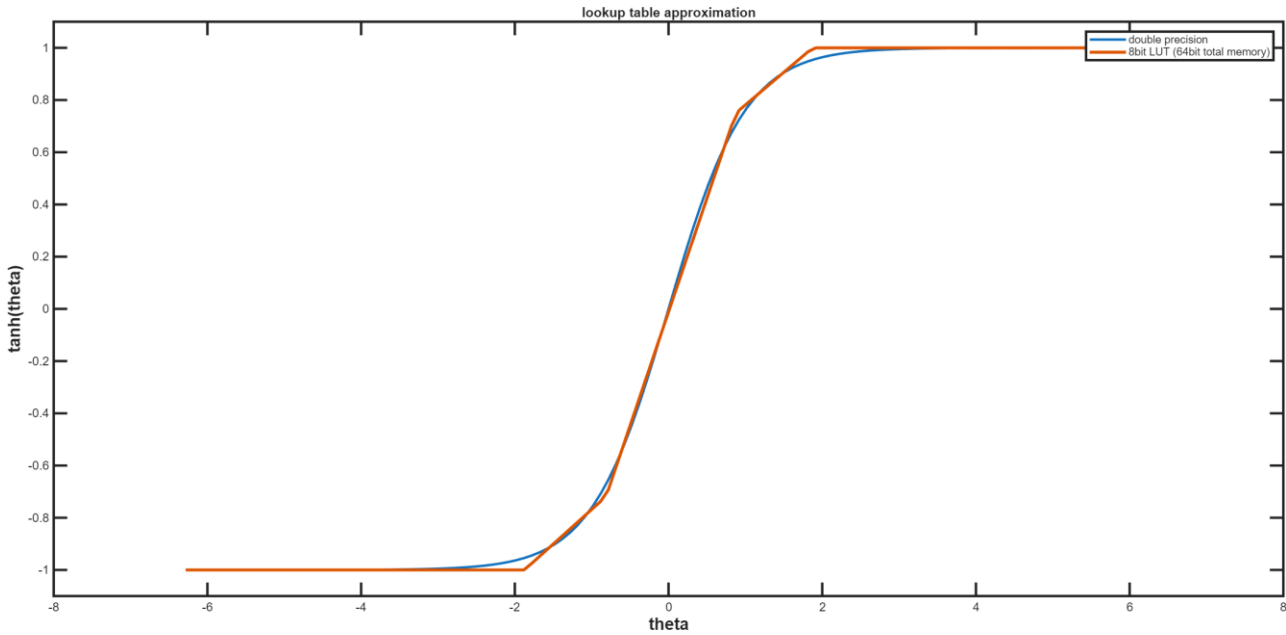
Faster on target

A word on activation functions

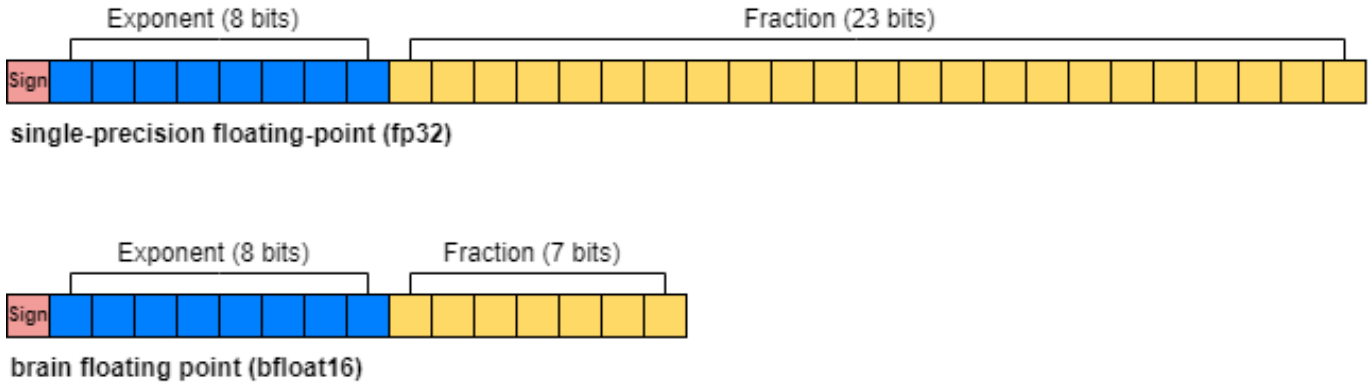
- Relu is straightforward to do in fixed-point, but what about tanh, sigmoid, … ?
- In practice, typically we use look-up table approximations

  => can control the required accuracy and memory cost
- Another option are CORDIC algorithms (for sigmoid/tanh)

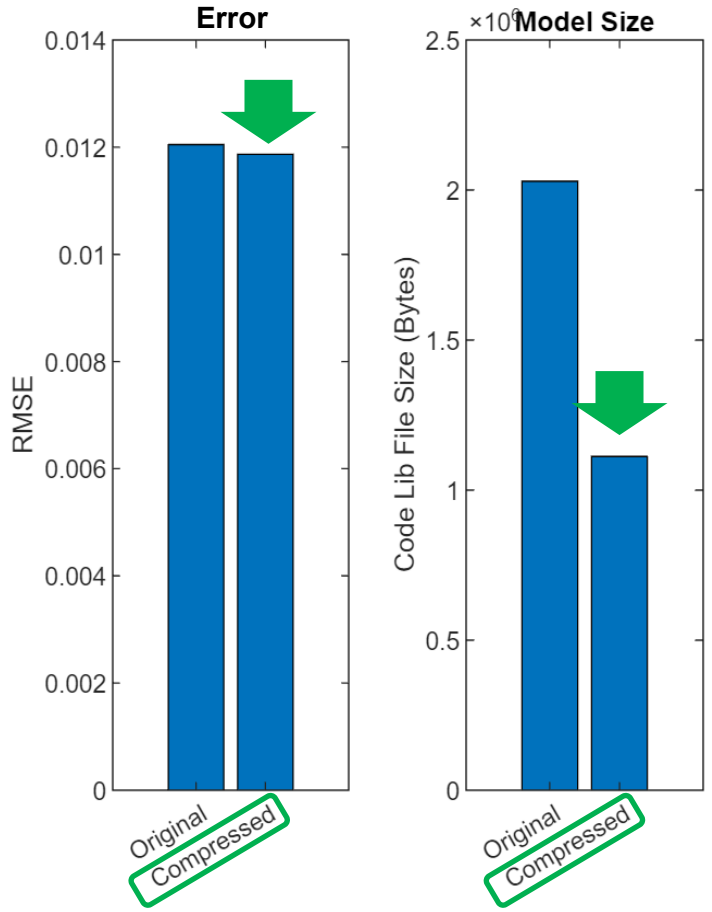




Model Compression with `bfloat16` Data Type Conversion



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

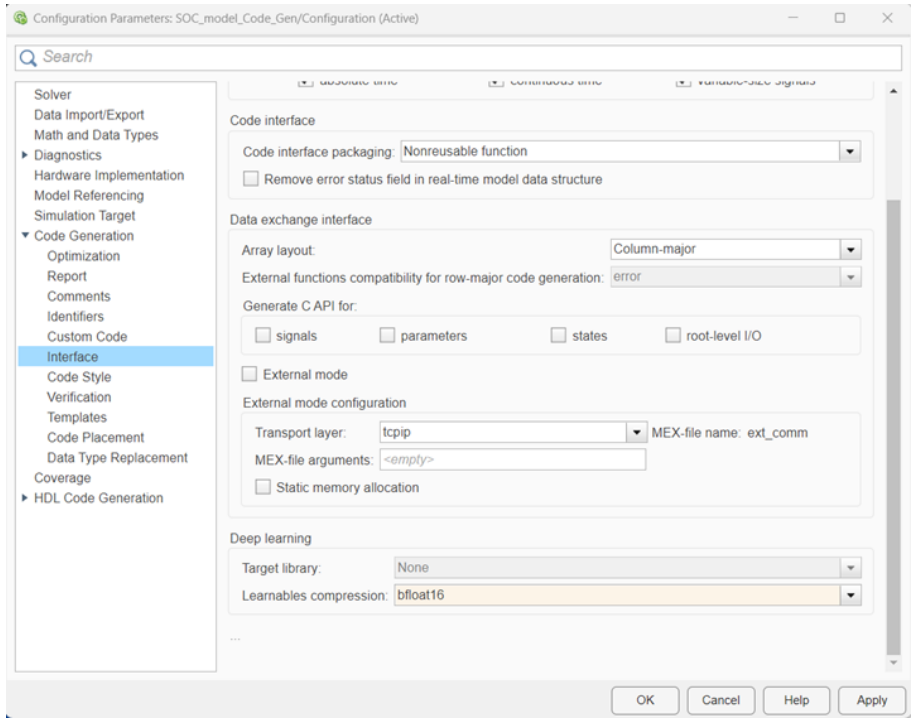


Generate bfloat16 Code for Deep Learning Networks

- 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

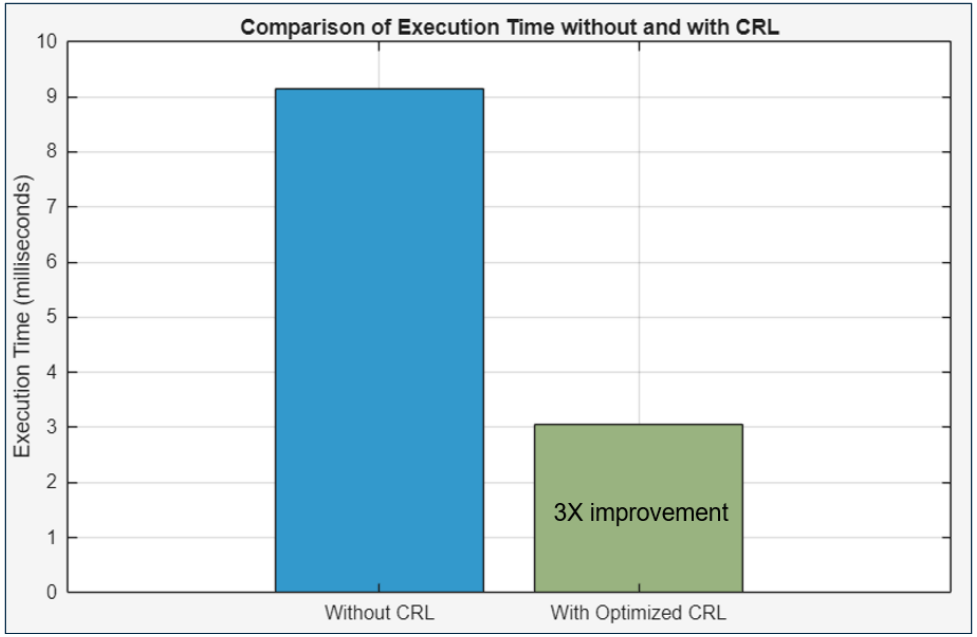
Techniques for Optimizing Performance

- **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, KDTTree
  - **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**

MathWorks