



MATLAB and PyTorch:

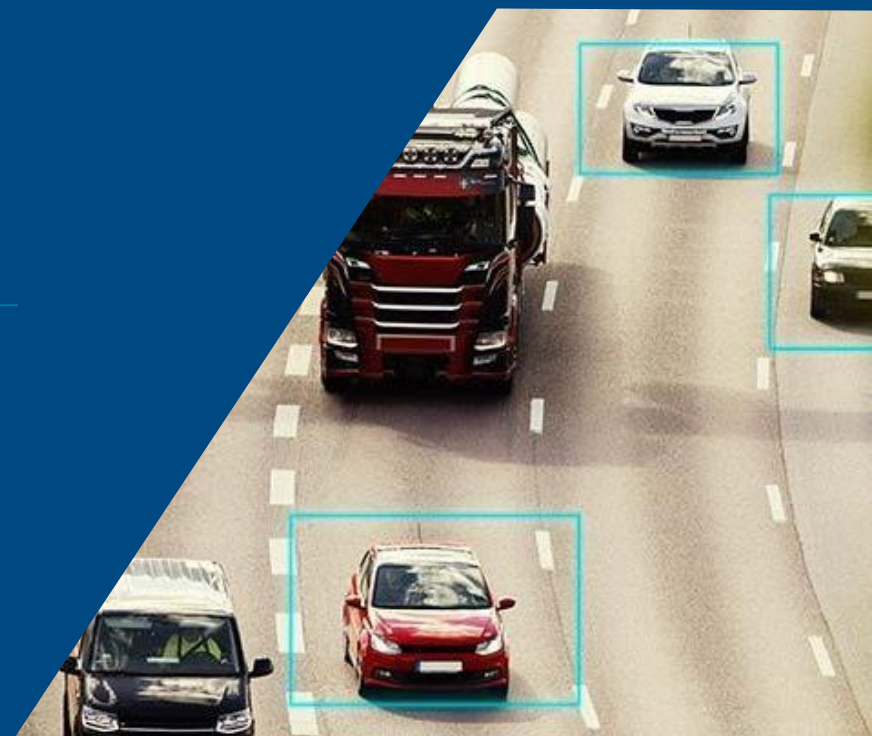
from Co-execution to Deployment

Bill Chou

Product Manager, Code Generation

Jianghao Wang

Product Manager, Deep Learning & AI



Engineers are developing domain-specific AI models to solve problems

Predictive Maintenance



Anomaly Detection



Fault Detection

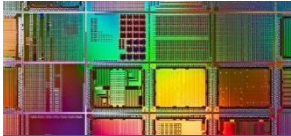


RUL Estimation

Visual Inspection



Defect Detection



Wafer Inspection



Factory Automation

Virtual Sensing



Physical sensor

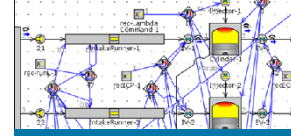


Environment

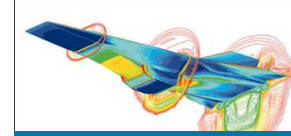


Process Monitoring

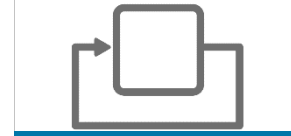
Data-Driven Control



ROM

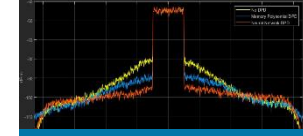


System Identification



Control Design

Wireless



Digital Pre-Distortion



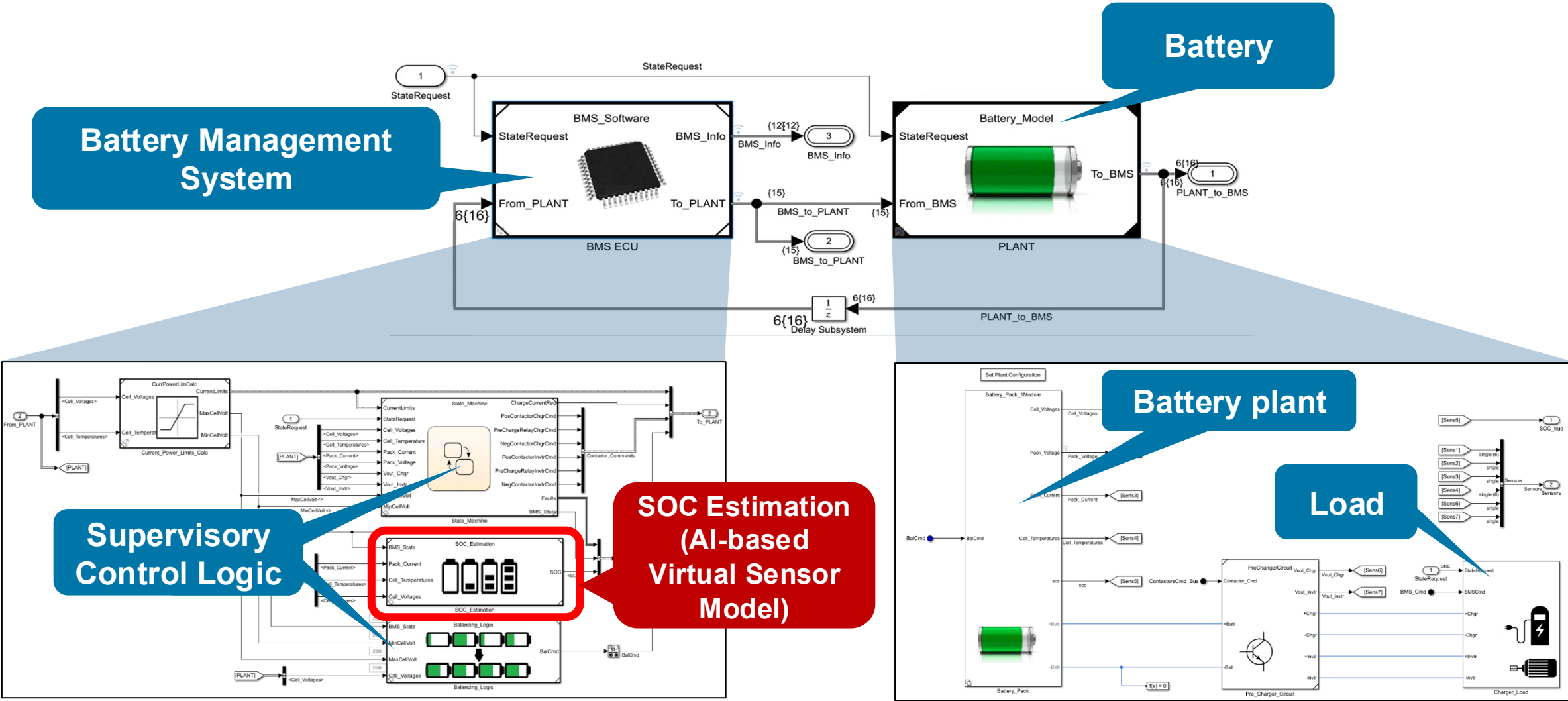
Device Identification



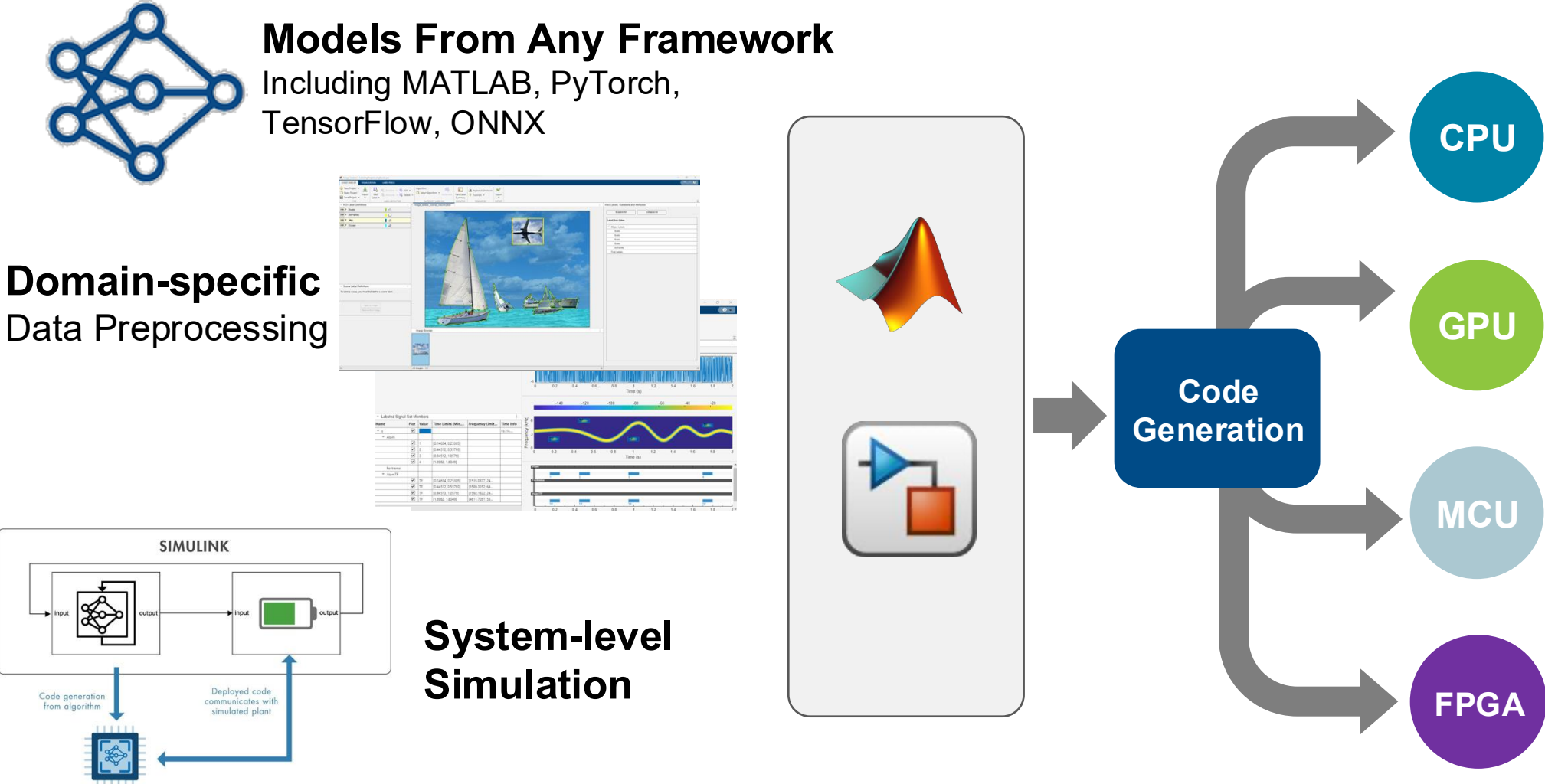
Localization

and
more...

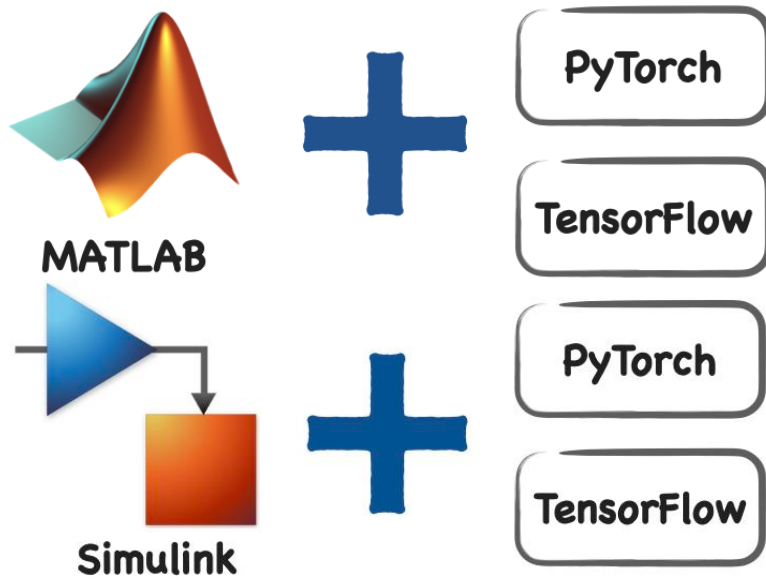
AI is Often a Part of a Larger System



Customers use MATLAB & Simulink Tools to Bridge AI Frameworks and Deployment

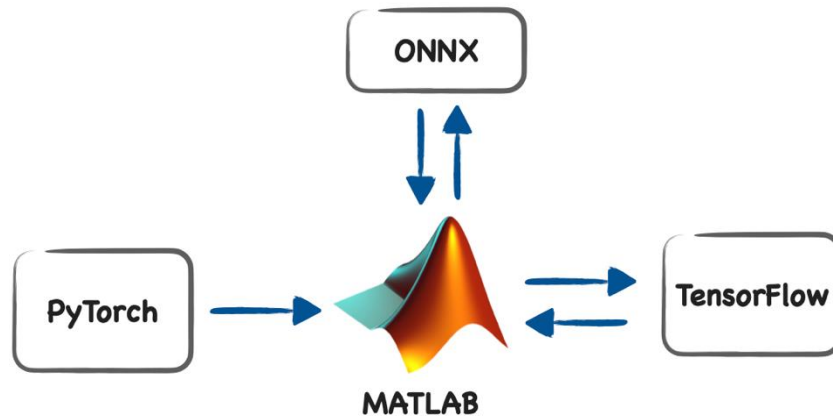


These are several entry points to interoperability. Consider what's needed for your application and workflow



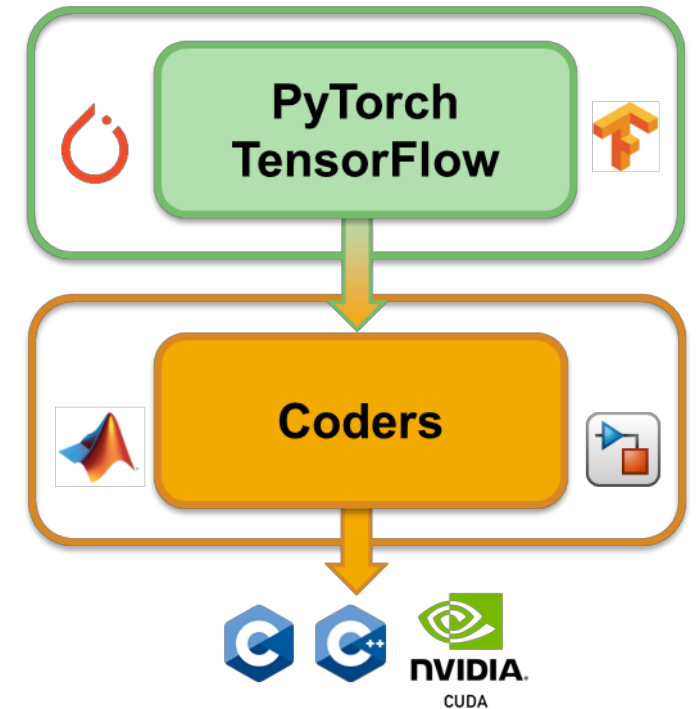
Co-execution

Use multiple frameworks together



Model Exchange

Convert models between frameworks



Direct code generation

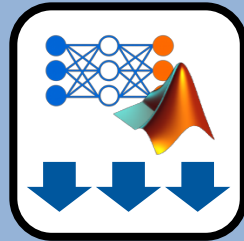
Generate code directly from PyTorch and TensorFlow

Lean Hardware

Cortex-M, Cortex-A/R, DSP

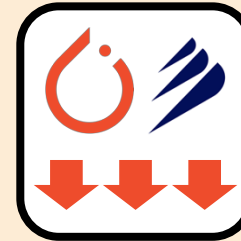
High-Performance Hardware

x86, GPU, Cortex-A/R, DSP



Deep Learning Toolbox Code Generation

Deploy to
Lean Hardware



PyTorch & LiteRT Code Generation

Deploy to
High-Performance
Hardware

Lean-Hardware Deployment: Virtual Sensor for Battery State-of-Charge Estimation

The Application: Virtual Sensor for Battery SoC



Why virtual? Because Physical sensors may be –
*Expensive, Slow, Noisy, Unreliable, Not feasible, Unmanufacturable,
Degrading over time, Requiring redundancy, etc.*

Workflow for developing AI-based virtual sensors



Data Preparation

Toolboxes for **domain-specific** pre/post-processing

AI Modeling

Low-code AI Modeling and training through MATLAB Apps

Model import from **PyTorch, TensorFlow, and ONNX**

Simulation & Test

Simulink blocks for AI inference make integration easy

Simulation-based unit testing

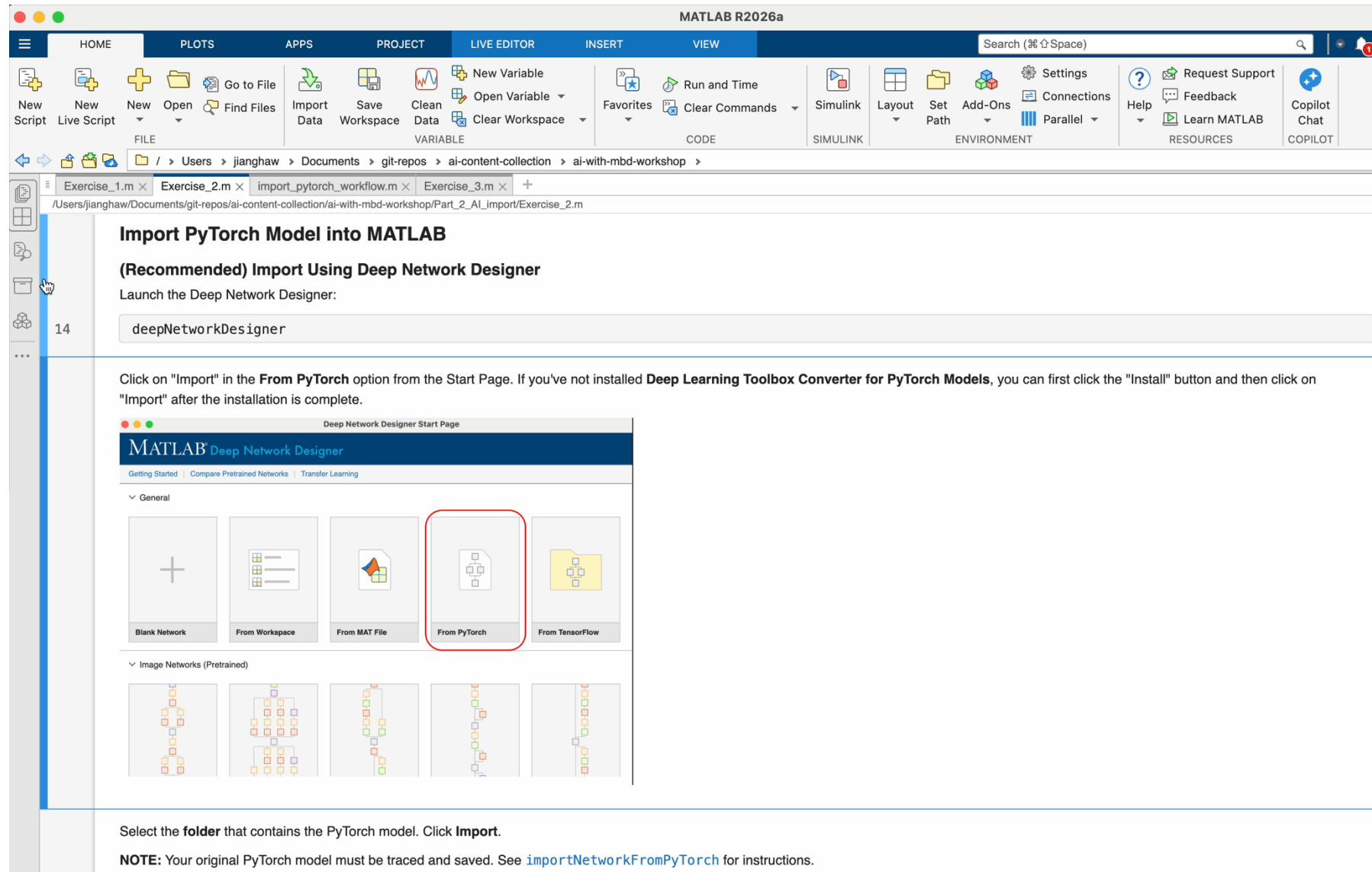
Compression

Model compression techniques to reduce model size and accelerate inference

Deployment

Automatic code generation for embedded targets

Import the PyTorch Model into MATLAB



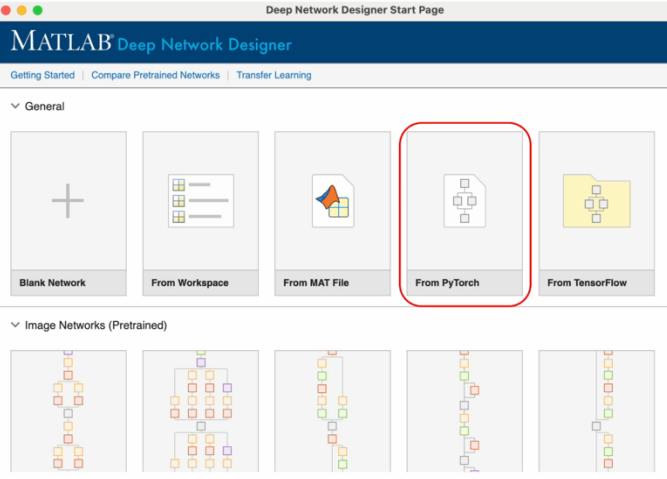
Import PyTorch Model into MATLAB

(Recommended) Import Using Deep Network Designer

Launch the Deep Network Designer:

```
deepNetworkDesigner
```

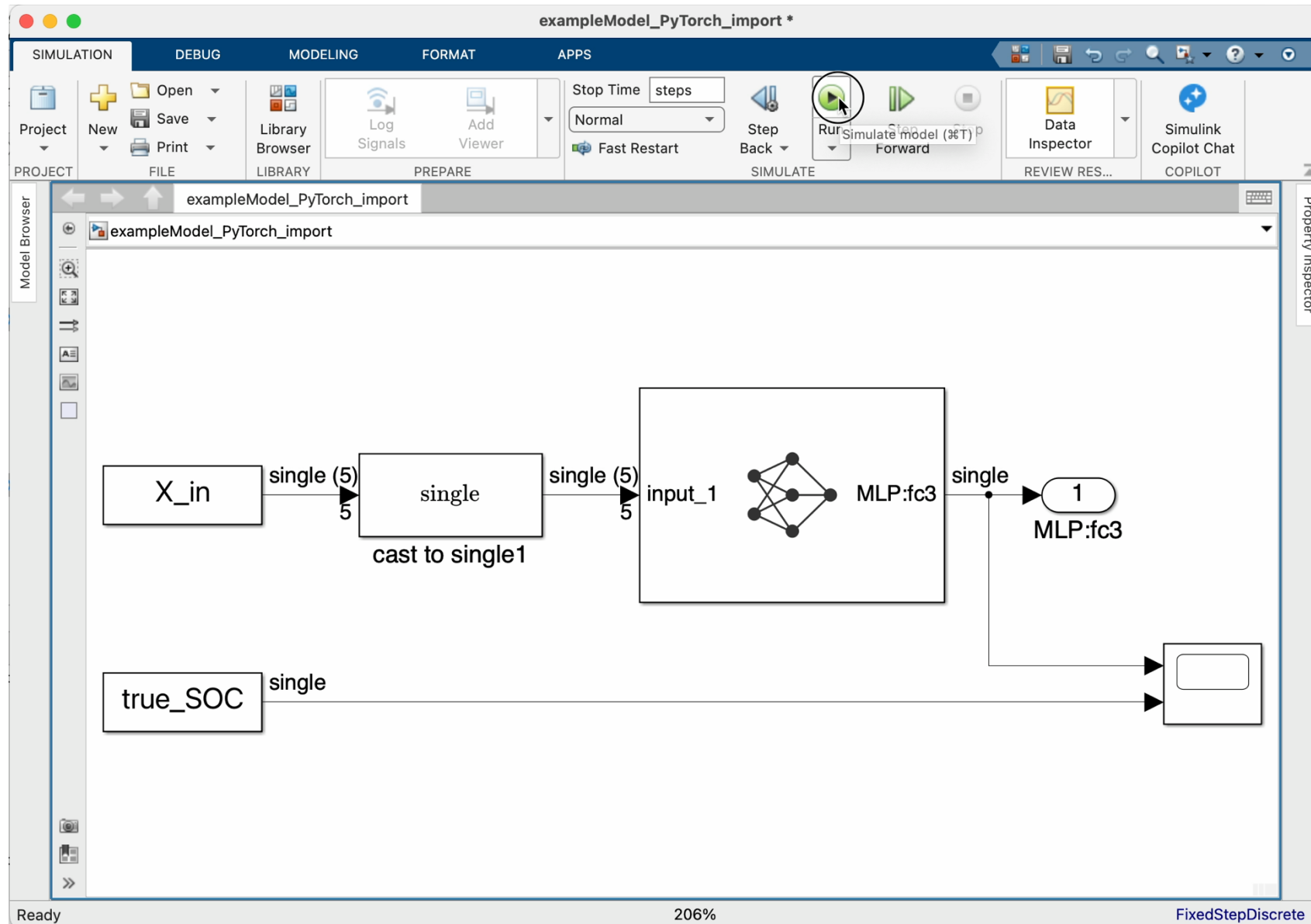
Click on "Import" in the **From PyTorch** option from the Start Page. If you've not installed **Deep Learning Toolbox Converter for PyTorch Models**, you can first click the "Install" button and then click on "Import" after the installation is complete.



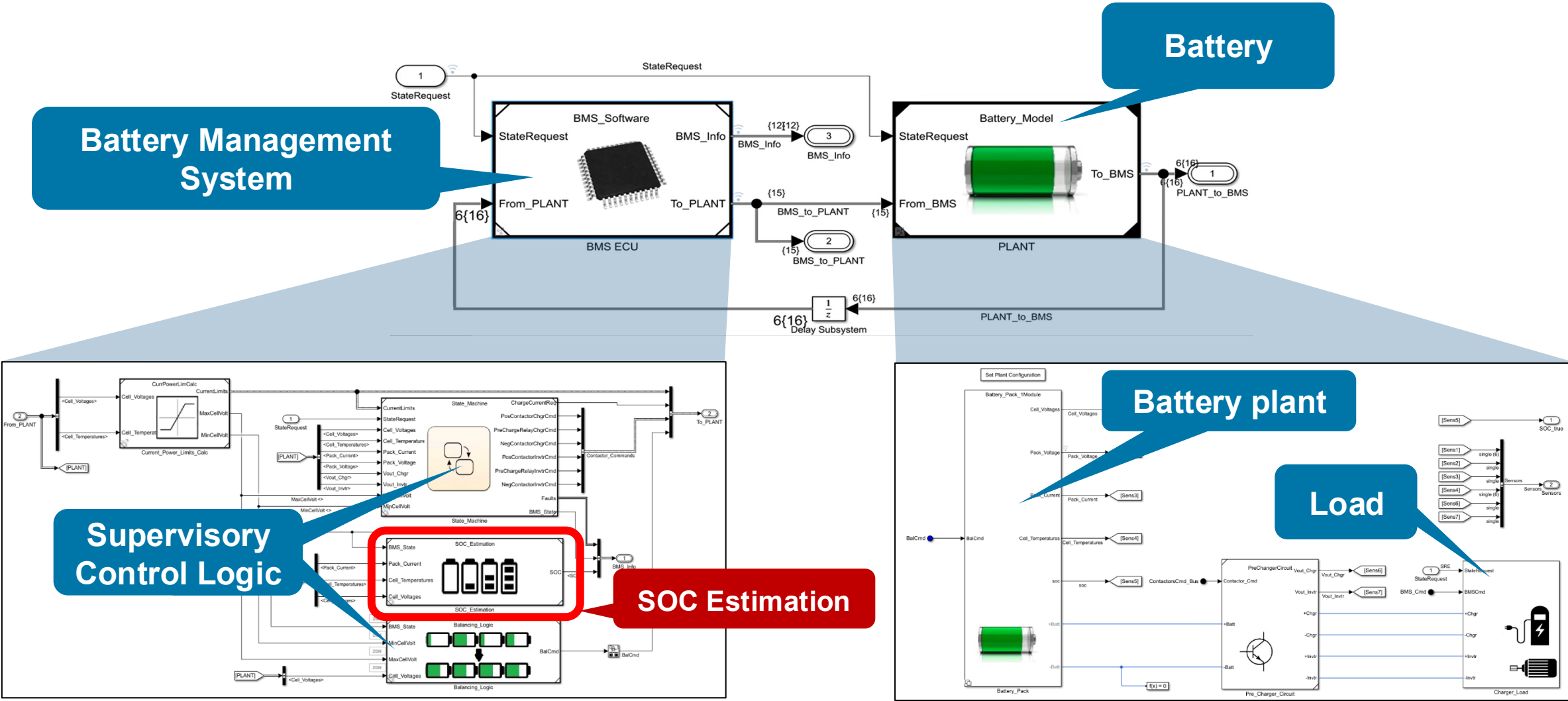
Select the **folder** that contains the PyTorch model. Click **Import**.

NOTE: Your original PyTorch model must be traced and saved. See [importNetworkFromPyTorch](#) for instructions.

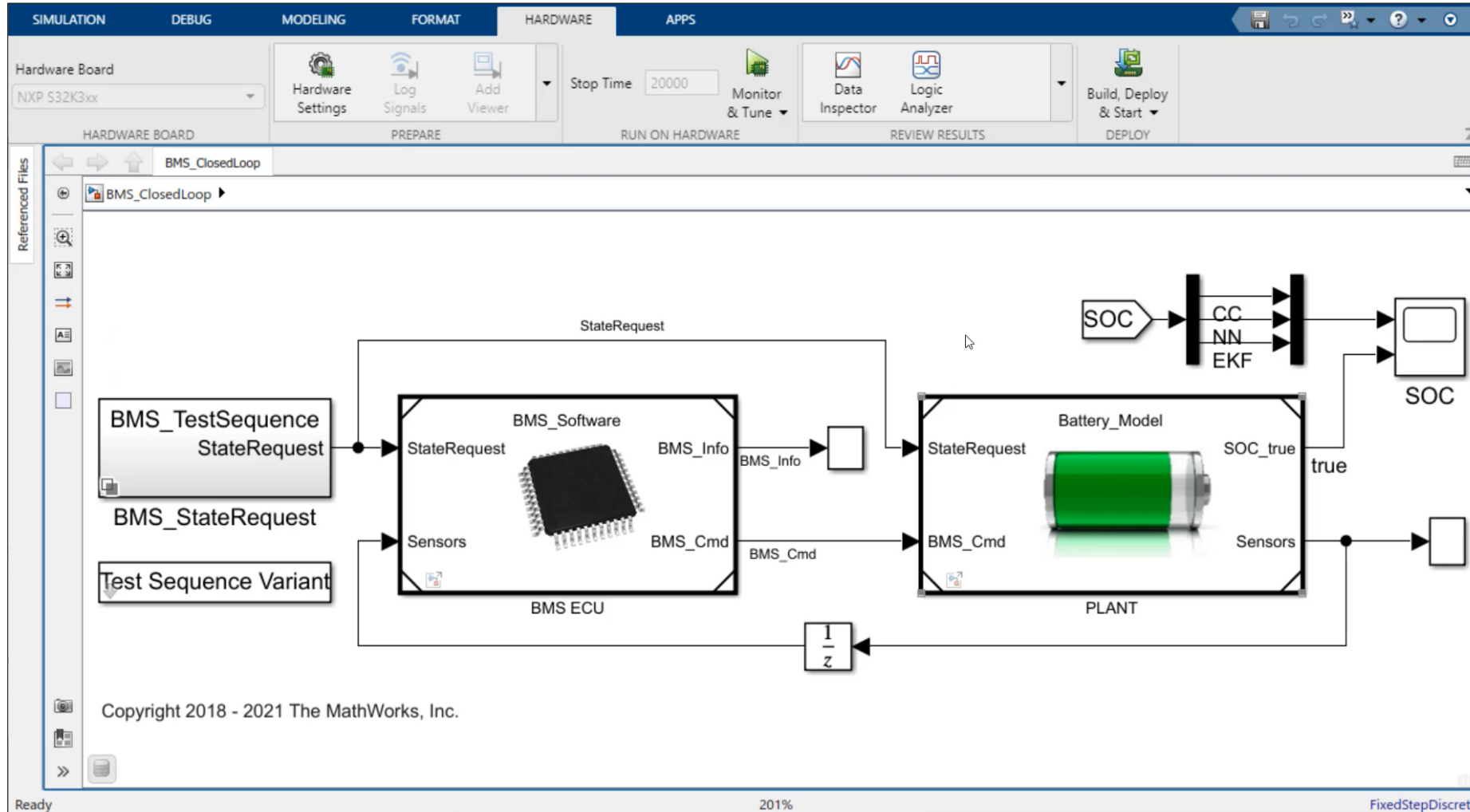
Integrate AI model into Simulink



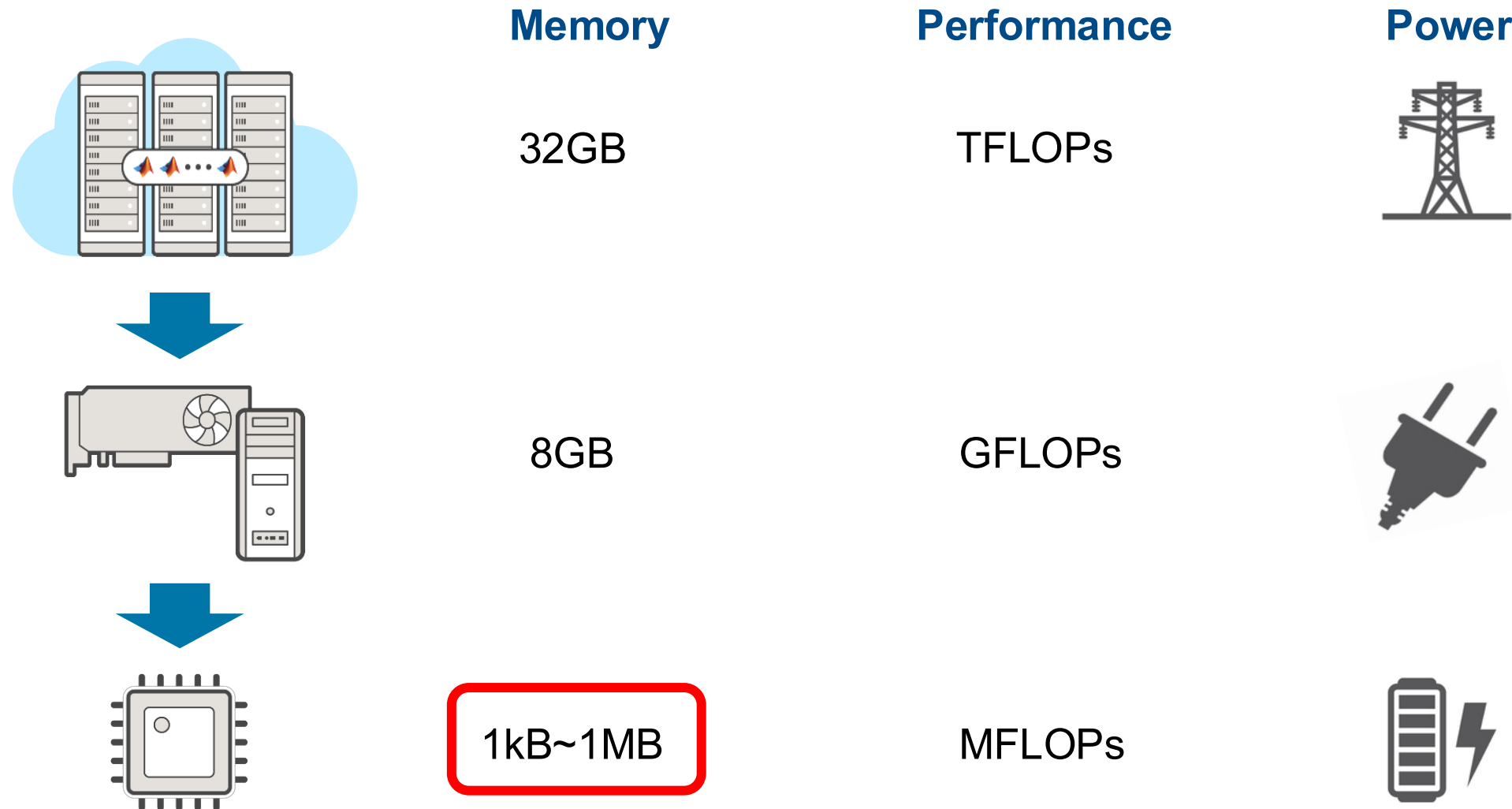
Design a Closed-Loop Battery Management System



Run Closed-Loop System-Level Simulation

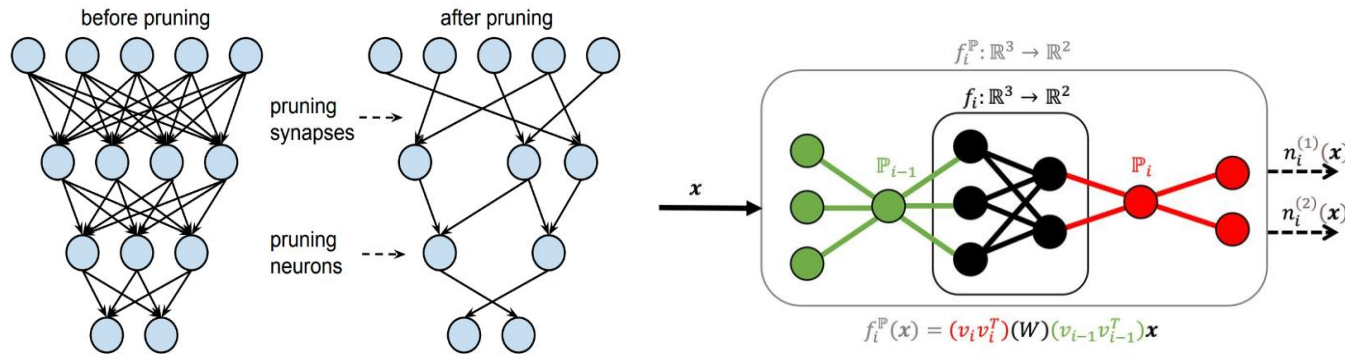


Embedded Devices Have Limited Resources



Compress an AI Model to Reduce Size While Maintaining Accuracy

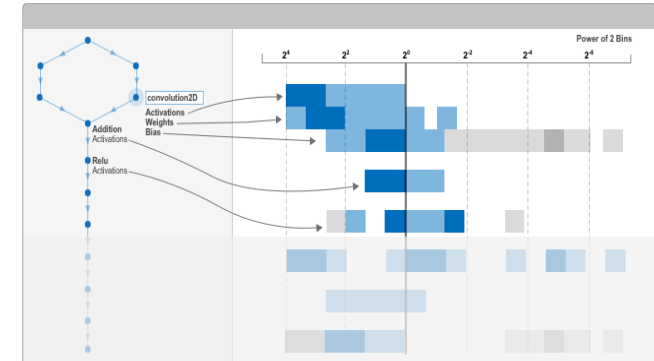
Structural Compression (Deep Learning Models)



Pruning
convolutional neural
networks

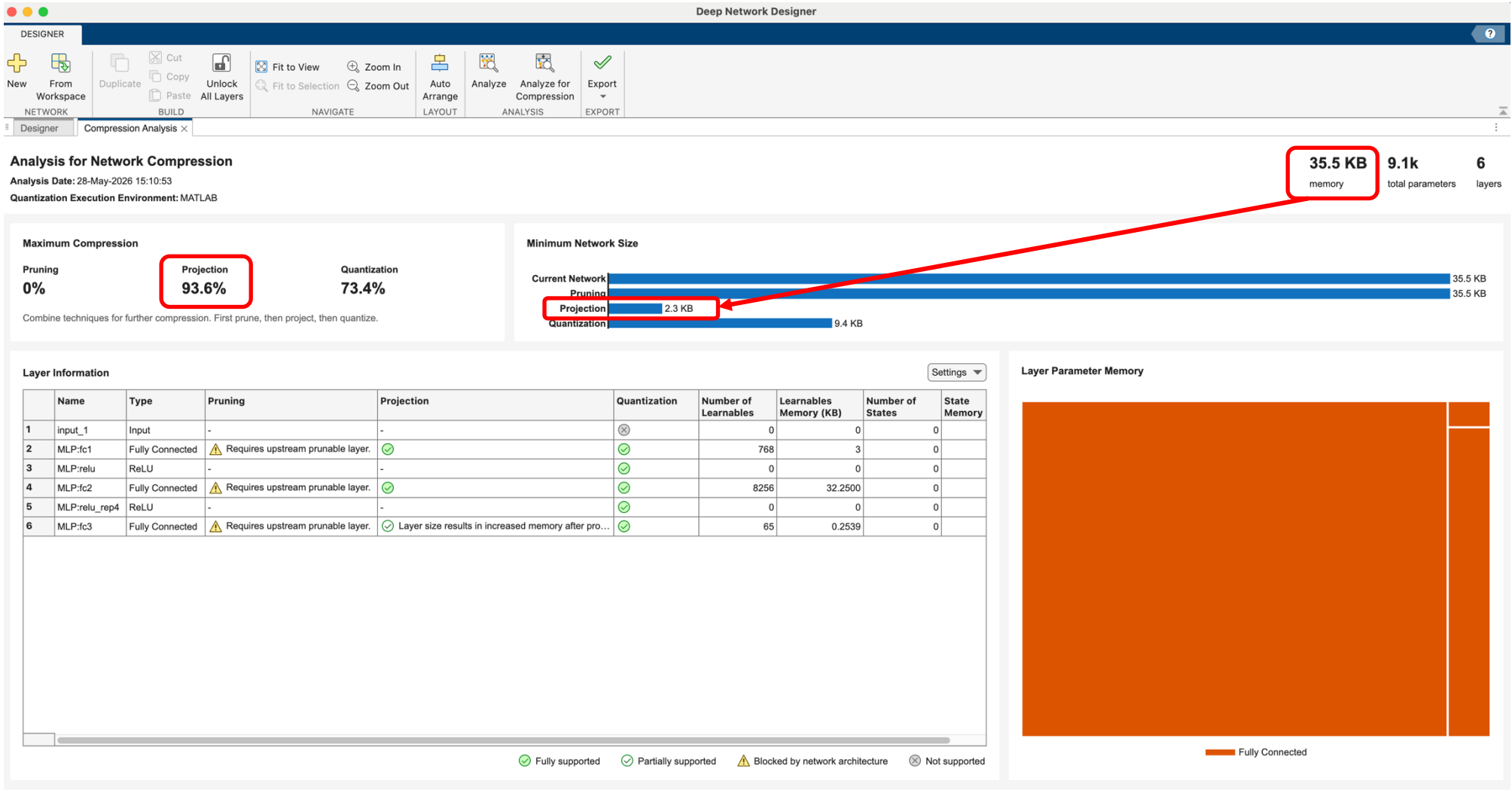
Projection of deep
neural networks

Datatype Compression (Machine Learning + Deep Learning Models)

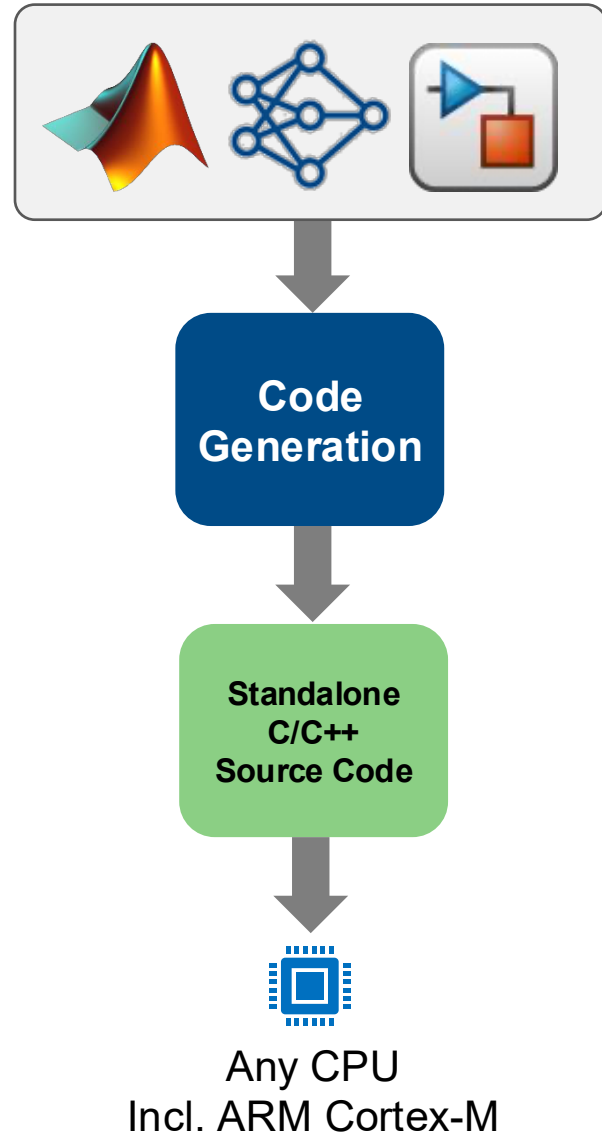


Quantization of network
weights to lower precision
datatypes (bfloat16, int8)

Compress an AI Model to Reduce Size While Maintaining Accuracy



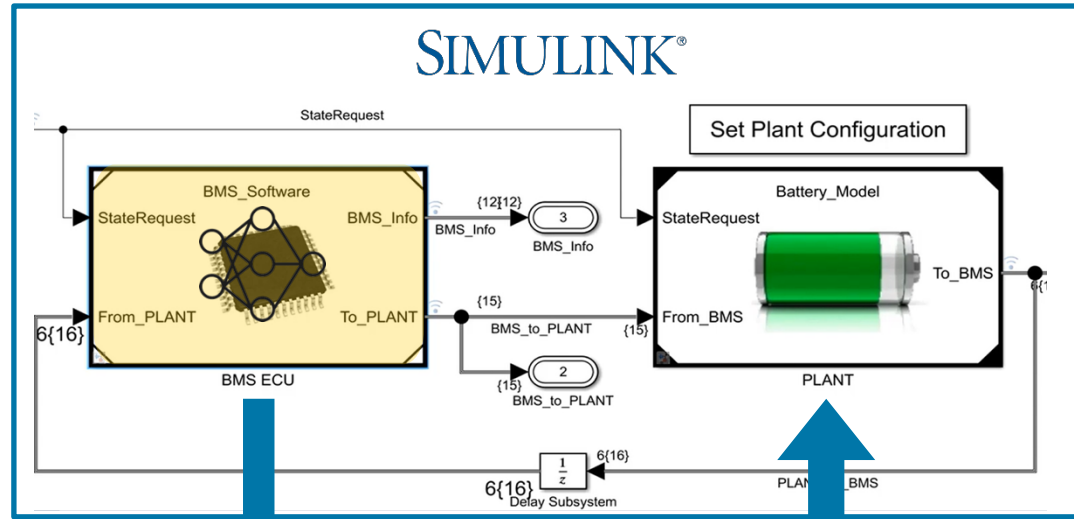
Generate Standalone C Code for ARM Cortex-M



Code Generation

- Configure for your target
- Embedded Coder generates optimized standalone C source code
 - No dependencies on Python
 - No third-party libraries required on target

Verify Code with Processor-in-the-Loop Testing



Code generation
from algorithm



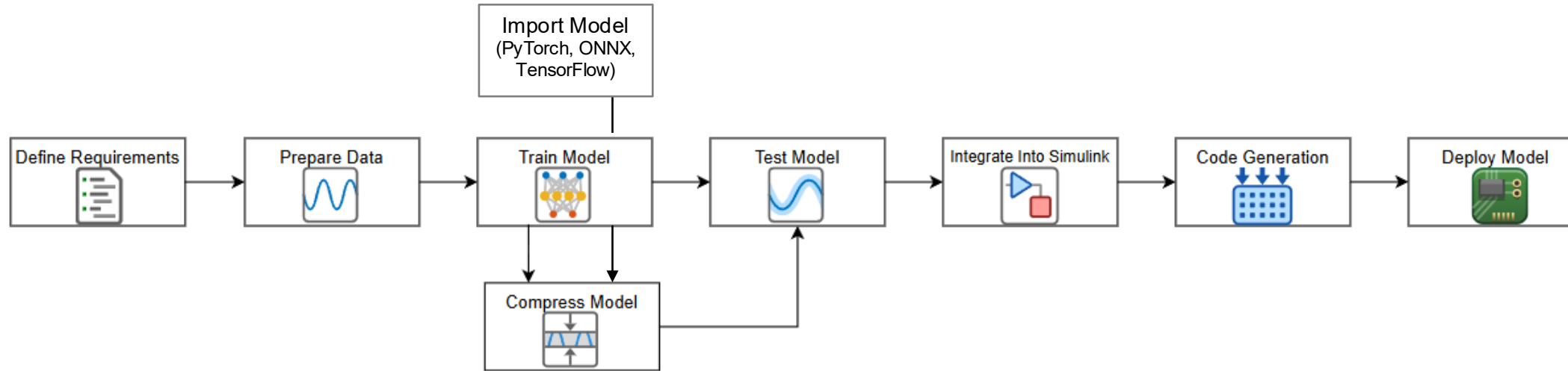
Arm® Cortex®-M

Deployed code
communicates with
simulated plant

Code Verification

- Software-in-the-loop testing on host
- Processor-in-the-loop testing on target
- Hardware-in-the-loop testing on standalone target

Lean Hardware Deployment Path: Leverage the Full MATLAB and Simulink Ecosystem



- Key Benefits:
 - Model surgery, fine-tuning, visualization, analysis and compression tools
 - INT8 code generation for quantized networks
 - Full system-in-the-loop/processor-in-the-loop verification and simulation

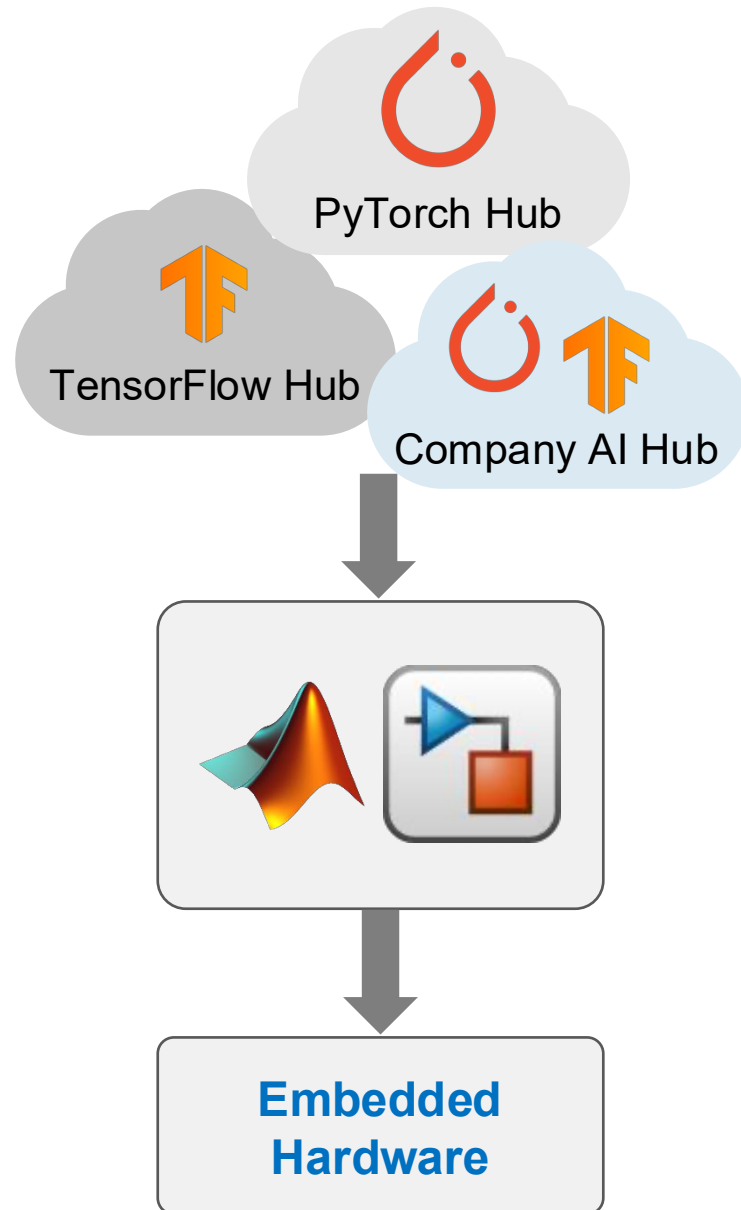
High-Performance Hardware Deployment: AI-Powered Visual Inspection

The Application: AI-Powered Visual Inspection



- Manufacturing
- Retail self check-out
- Autonomous vehicle
- Robotics
- ...

Why Direct Code Generation?



Scenario

- Large models from Hubs
- Keep source of truth in PyTorch/TensorFlow/ONNX
- Models already compressed
- Need to integrate with larger engineered system and deploy to embedded

Benefits:

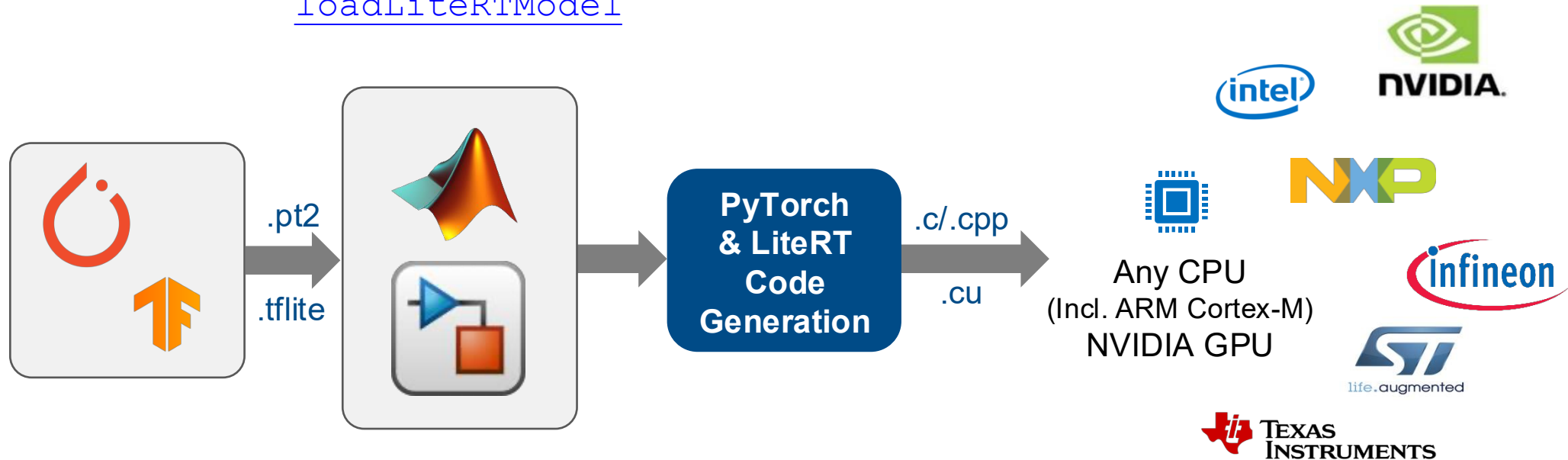
- Wider model/layer support
- More direct workflow
- Model truth stays in PyTorch

PyTorch & LiteRT Code Generation Workflow

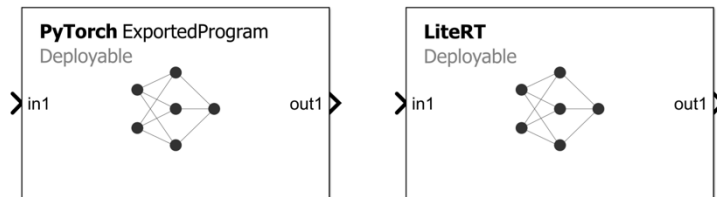
MATLAB Coder API

[loadPyTorchExportedProgram](#)

[loadLiteRTModel](#)



Simulink Block



Generated Code has

- No dependencies on Python
- No third-party libraries required on target

Supported Models & Generated Code

Types	Models
Image Classification	ResNet50, MobileNetV2, EfficientNet, Inceptionv3, MNASNet, DenseNet
Object Detection	YOLOv8, YOLOv11, YOLOv26
Segmentation	SAMv2, Depth Anything
Vision Transformers	DINOv2
Language & Audio	RoBERTa, Whisper
Generative Networks	DCGAN, CycleGAN
Sequence Models	LSTMs, GRU, BiLSTM

Generated Code:

- Standalone C, C++, or CUDA
- Optimizations
 - OpenMP
 - SIMD
 - Code replacement libraries
- Data types: Single, FP16

Code Verification:

- Software-in-the-loop
- Processor-in-the-loop
- Hardware-in-the-loop

Bottle Counting on a Manufacturing Line

Perform instance segmentation and object detection using a You Only Look Once v26 (YOLO v26) model in the PyTorch ExportedProgram (.pt2) format.

```
img = imread("Bottle.jpg");
pt2Model = loadPyTorchExportedProgram("yolo26n-seg-2out.pt2");
out = yoloSegmentImagePt2_v26(img);
```

Generate C/C++ code for the entry-point function targeting the CPU

```
cfg = coder.config('lib');
codegen -config cfg yoloSegmentImagePt2_v26 -args {img} -report|
```



```
function out = yoloSegmentImagePt2_v26(in)
    %#codegen

    if size(in, 3) == 1
        in = repmat(in, [1 1 3]);
    end

    origSize = size(in, 1:2);
    inputSize = [640 640];
    img = im2single(imresize(in, inputSize));

    persistent yoloSegModel
    if isempty(yoloSegModel)
        yoloSegModel = loadPyTorchExportedProgram("yolo26n-seg-2out.pt2");
    end

    outs = cell(1, 2);
    [outs{:}] = yoloSegModel.invoke(permute(img, [4 3 1 2]));
    detections = squeeze(outs{1});
    masks = permute(squeeze(outs{2}), [2 3 1]);
    scores = detections(:, 5);
    keep = scores > 0.5;

    bboxes_xyxy = detections(keep, 1:4);
    labels = detections(keep, 6) + 1;
    maskCoeffs = detections(keep, 7:38);

    bboxes_xyxy(:, [1 3]) = bboxes_xyxy(:, [1 3]) * origSize(2) / inputSize(2);
    bboxes_xyxy(:, [2 4]) = bboxes_xyxy(:, [2 4]) * origSize(1) / inputSize(1);
    binaryMasks = processMasks(masks, maskCoeffs, bboxes_xyxy, origSize);

    bboxes_xywh = bboxes_xyxy;
    bboxes_xywh(:, 3) = bboxes_xyxy(:, 3) - bboxes_xyxy(:, 1);
    bboxes_xywh(:, 4) = bboxes_xyxy(:, 4) - bboxes_xyxy(:, 2);
    bboxes_xywh = max(floor(bboxes_xywh), 1);

    classes = createClasses(labels);

    RGB = insertObjectMask(in, binaryMasks, LineColor=[1 1 1], LineWidth=1);
    out = insertObjectAnnotation(RGB, 'rectangle', bboxes_xywh, classes);

end
```

Self-Checkout Object Detection

Perform instance segmentation and object detection using a You Only Look Once v26 (YOLO v26) model in the PyTorch ExportedProgram (.pt2) format.

```
img = imread("Grocery.jpg");
pt2Model = loadPyTorchExportedProgram("yolo26n-seg-2out.pt2");
out = yoloSegmentImagePt2_v26(img);
```

Generate C/C++ code for the entry-point function targeting the CPU

```
cfg = coder.config('lib');
codegen -config cfg yoloSegmentImagePt2_v26 -args {img} -report
```



```
function out = yoloSegmentImagePt2_v26(in)
%#codegen

if size(in, 3) == 1
    in = repmat(in, [1 1 3]);
end

origSize = size(in, 1:2);
inputSize = [640 640];
img = im2single(imresize(in, inputSize));

persistent yoloSegModel
if isempty(yoloSegModel)
    yoloSegModel = loadPyTorchExportedProgram("yolo26n-seg-2out.pt2");
end

outs = cell(1, 2);
[outs{:}] = yoloSegModel.invoke(permute(img, [4 3 1 2]));
detections = squeeze(out{1});
masks = permute(squeeze(out{2}), [2 3 1]);
scores = detections(:, 5);
keep = scores > 0.5;

bboxes_xyxy = detections(keep, 1:4);
labels = detections(keep, 6) + 1;
maskCoeffs = detections(keep, 7:38);

bboxes_xyxy(:, [1 3]) = bboxes_xyxy(:, [1 3]) * origSize(2) / inputSize(2);
bboxes_xyxy(:, [2 4]) = bboxes_xyxy(:, [2 4]) * origSize(1) / inputSize(1);
binaryMasks = processMasks(masks, maskCoeffs, bboxes_xyxy, origSize);

bboxes_xywh = bboxes_xyxy;
bboxes_xywh(:, 3) = bboxes_xyxy(:, 3) - bboxes_xyxy(:, 1);
bboxes_xywh(:, 4) = bboxes_xyxy(:, 4) - bboxes_xyxy(:, 2);
bboxes_xywh = max(floor(bboxes_xywh), 1);

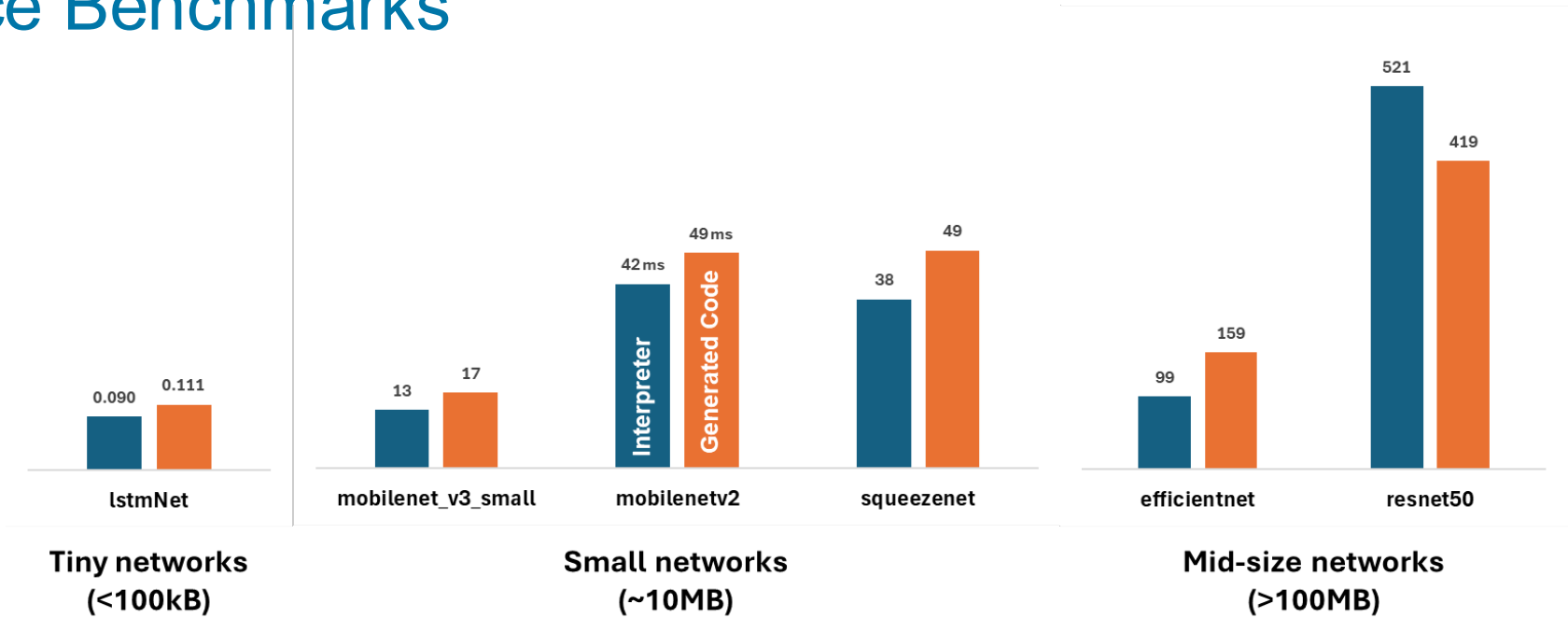
classes = createClasses(labels);

RGB = insertObjectMask(in, binaryMasks, LineColor=[1 1 1], LineWidth=1);
out = insertObjectAnnotation(RGB, 'rectangle', bboxes_xywh, classes);

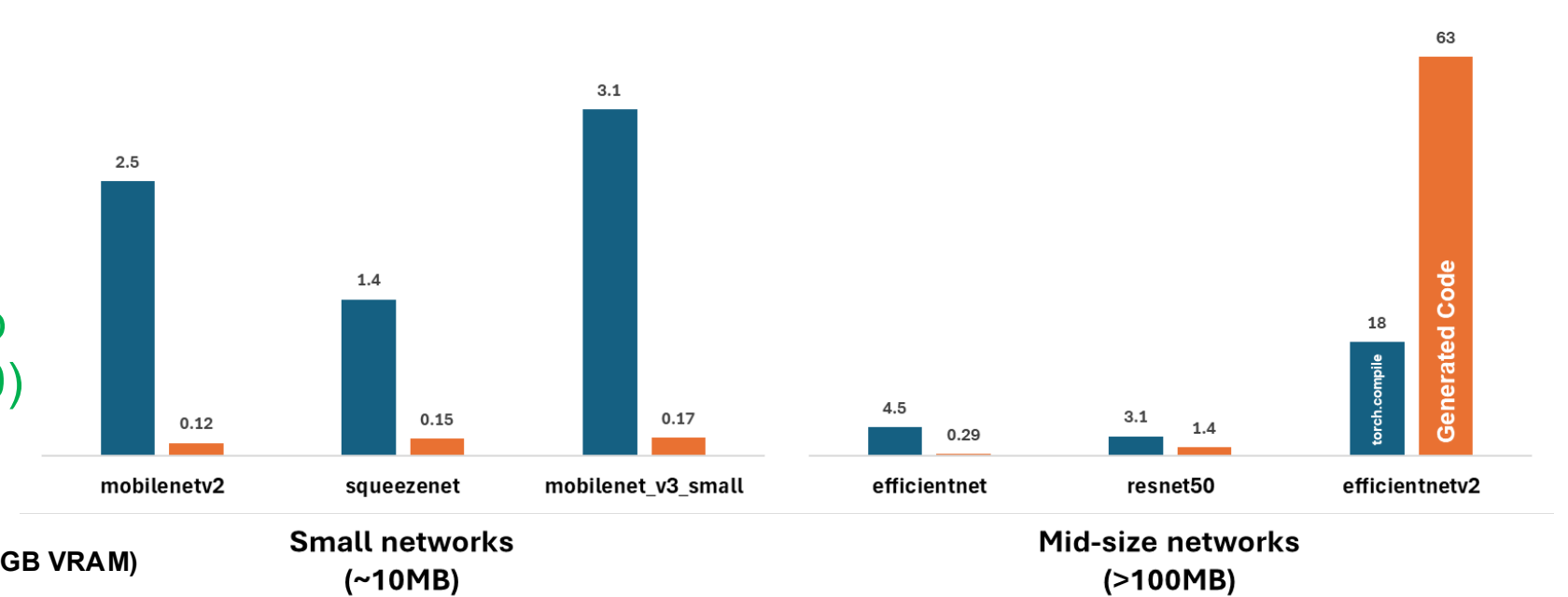
end
```

Performance Benchmarks

Good
Performance on
Raspberry Pi 4
(ARM Cortex-A)



Strong
Performance on
NVIDIA Desktop
GPU (RTX 4080)



Simulink Integration: Autonomous Vehicle Perception

YOLOv26Pt2VideoSimulink - Simulink

MATHWORKS-X4U5T

SIMULATION DEBUG MODELING FORMAT APPS C++ CODE BLOCK

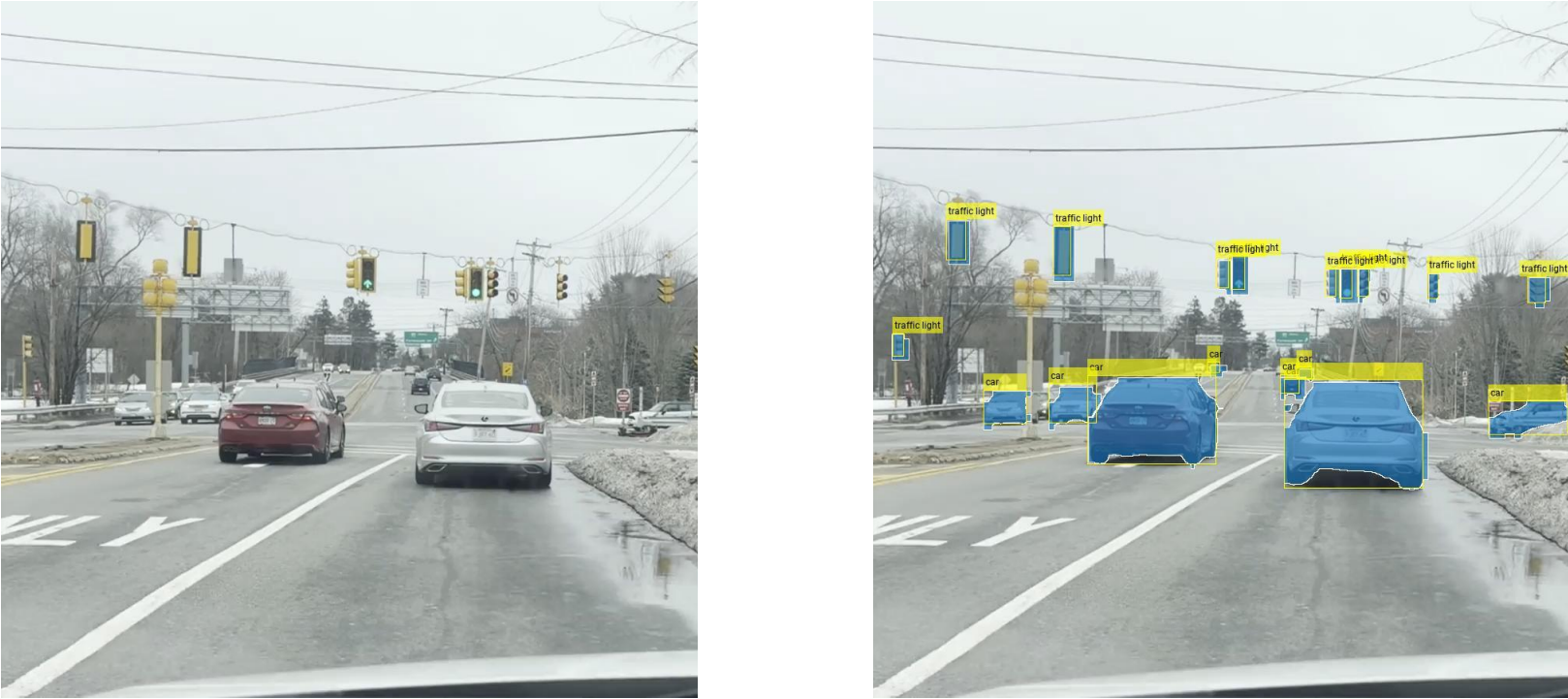
Stop Time 5 Normal Fast Restart Step Back Run Step Forward Stop Data Inspector Logic Analyzer Bird's-Eye Scope Simulation Manager

FILE LIBRARY PREPARE SIMULATE REVIEW RESULTS

Model Browser

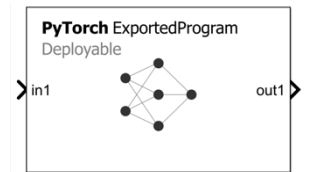
YOLOv26Pt2VideoSimulink

YOLOv26 Object Detection and Segmentation



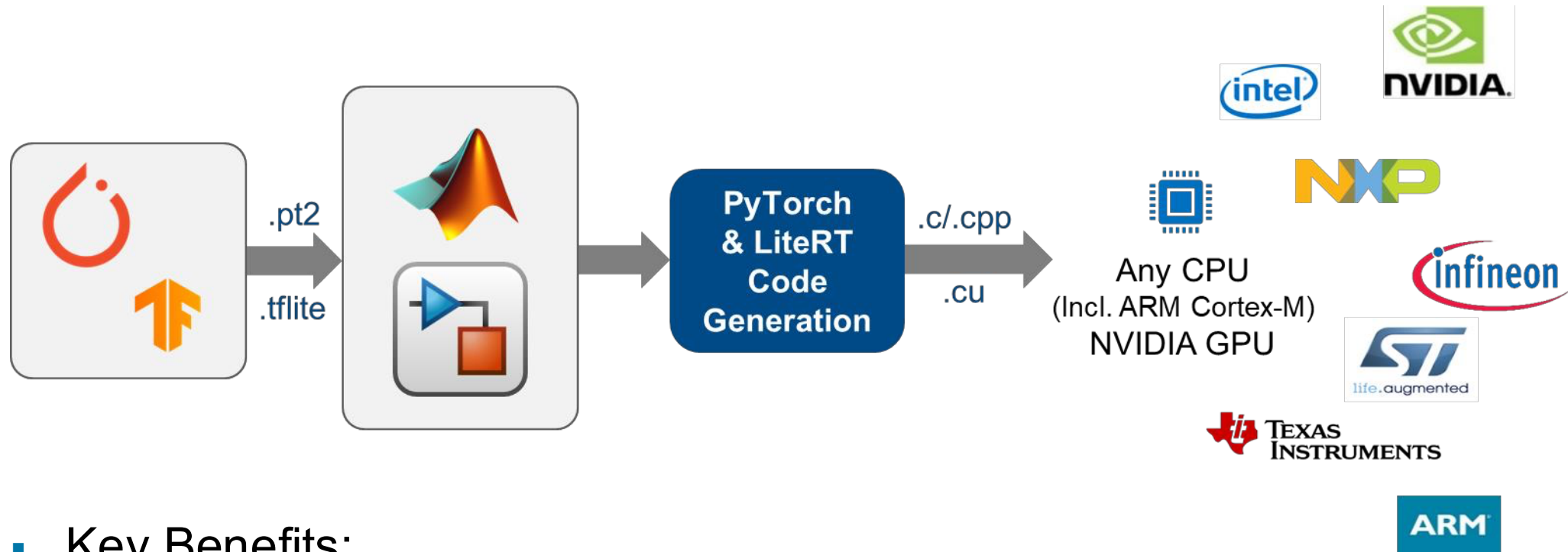
Code Mappings - Component Interface

Ready 96% auto(FixedStepDiscrete)



- Model Reference
- Rapid Accelerator Mode
- SIL
- PIL
- ...

High-Performance Hardware Deployment Path: Direct Code Generation from PyTorch for Engineered System



- Key Benefits:
 - Wider model and layer support
 - Simpler, more direct workflow from PyTorch to code
 - No Python or 3P dependencies in generated code
 - Scales from standalone inference to full Simulink system simulation

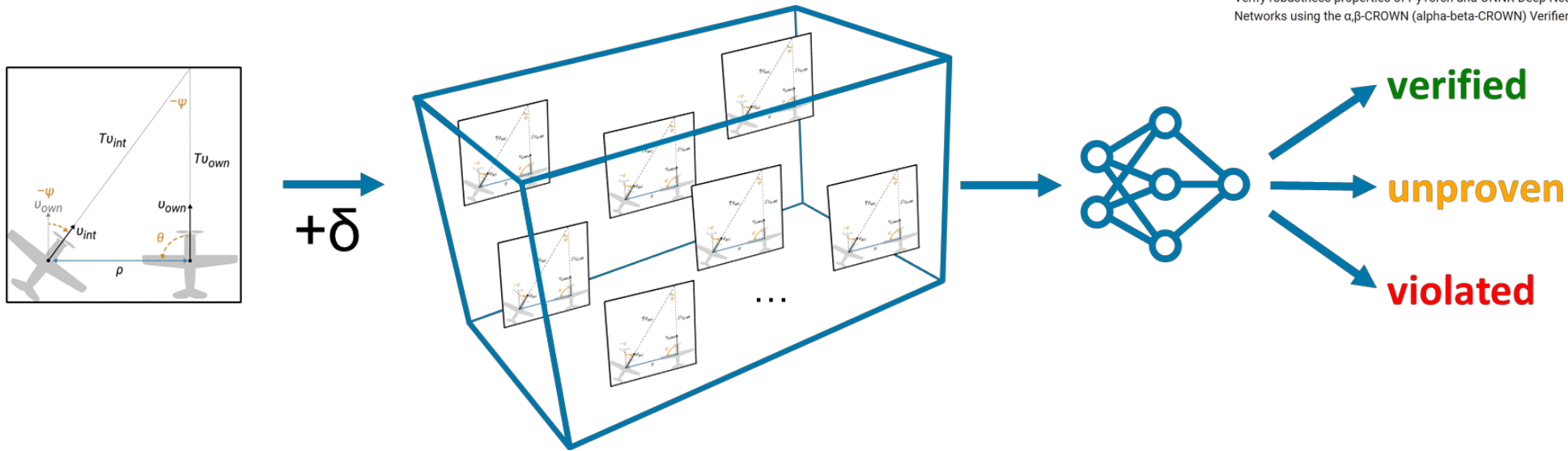
Verify AI Models for Safety-Critical Systems



Deep Learning Toolbox Interface for
alpha-beta-CROWN Verifier

by MathWorks Deep Learning Toolbox Team **STAFF**

Verify robustness properties of PyTorch and ONNX Deep Neural
Networks using the $\alpha\beta$ -CROWN (alpha-beta-CROWN) Verifier



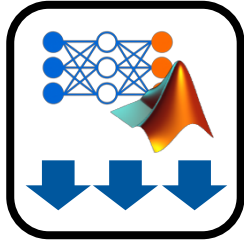
Why Verify?

- Safety-critical systems
- Automotive, aerospace, medical
- Regulatory requirements
- Adversarial robustness

Capabilities:

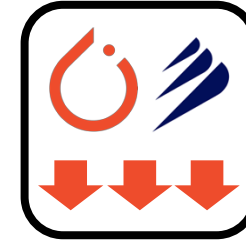
- Robustness verification
- Adversarial attack analysis
- Coverage metrics
- Formal guarantees

Choosing the Right Workflow



Deep Learning Toolbox + Code Generation

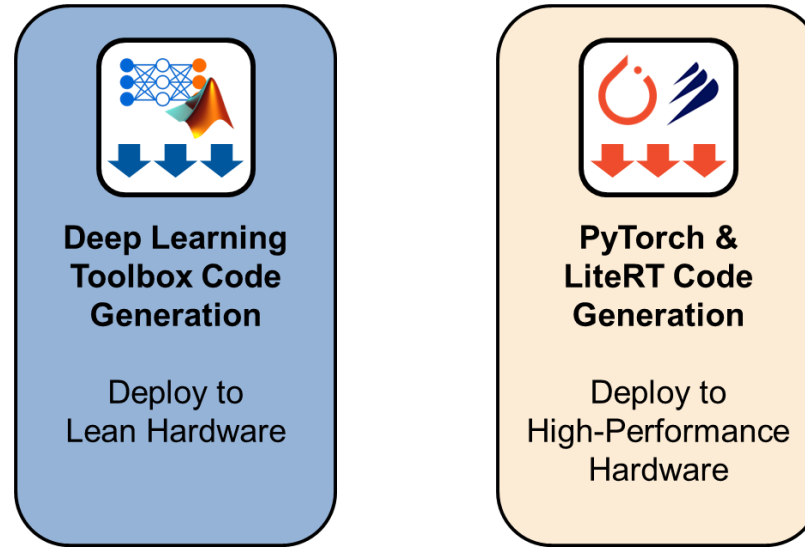
- Compression using MathWorks tools
- Need model surgery
- Targeting MCUs
- Want INT8 code generation



PyTorch & LiteRT Code Generation

- Larger model, already compressed
- Keep source-of-truth in PyTorch
- Wider layer support needed
- Targeting GPU, ARM Cortex-A, Industrial PCs
- Want direct workflow

Summary & Key Takeaways



- MATLAB & Simulink integrate deeply with PyTorch and TensorFlow
- Full system-level simulation and verification before deployment
- One environment from data preparation to production code
- Two proven deployment paths for different hardware targets
- Choose the workflow that matches your hardware and model complexity

Resources & Next Steps

- Documentation
 - [Deep Learning Toolbox](#)
 - [MATLAB Coder](#)
 - [GPU Coder](#)
 - [Embedded Coder](#)
- [PyTorch & LiteRT code generation examples](#)
- [Battery SoC estimation example](#)
- Product trial:
www.mathworks.com/trial

Train Deep Learning Network for Battery State of Charge Estimation R2026a

Since R2024b

Step 3 of 8 in Battery State of Charge Estimation Using Deep Learning

Open in MATLAB ▼

Copy Command

This example shows how to train a deep neural network to predict battery state of charge (SOC).

SOC is the level of charge of an electric battery relative to its capacity, measured as a percentage. This example shows how to train a deep learning network to predict battery state of charge given the temperature, voltage, and current.

This example is step three in a series of examples that take you through a battery state of charge estimation workflow. You can run each step independently or work through the steps in order. This example follows the [Prepare Data for Battery State of Charge Estimation Using Deep Learning](#) example. For more information about the full workflow, see [Battery State of Charge Estimation Using](#)

Generate Code for Segmentation and Object Detection Using YOLO v11 LiteRT Model R2026a

Since R2026a

This example shows how to generate a CUDA® executable for a You Only Look Once v11 (YOLO v11) model. The generated CUDA code does not contain dependencies to the NVIDIA® cuDNN or TensorRT™ deep learning libraries. This example uses a YOLO v11 model that was trained by [Ultralytics](#) on the [COCO](#) data set. YOLO v11 is one of the best performing real-time object detectors and is considered as improvement to the existing YOLO variants such as YOLO v8, YOLO v9 and YOLO v10. The instance segmentation variation of the YOLO v11 model provides masks or contours that outline each detected object that helps in segmenting the objects from the rest of the image.

Third-Party Prerequisites

This example requires a CUDA-enabled NVIDIA® GPU and a compatible driver.

For non-MEX builds, such as static, dynamic libraries or executables, this example also requires:

- The NVIDIA CUDA toolkit.
- Environment variables for the compilers and libraries. For more information, see [Third-Party Hardware](#) (GPU Coder) and [Setting Up the Prerequisite Products](#) (GPU Coder).

This example uses:

- MATLAB Coder
- GPU Coder
- MATLAB Coder Support Package for PyTorch
- LiteRT Models
- Computer Vision Toolbox

Open in Installed MATLAB

Copy Command

that you prepared in the previous step.
e of Charge Estimation Using Deep Learning.
array containing the temperature, voltage, and
uts, where each data set is a cell array containing

Execute Code

Visualize the segmented image by running the generated executable.

```
out = yoloSegmentImage_mex(I);  
imshow(out);
```

Get ▼

