Efficient AI

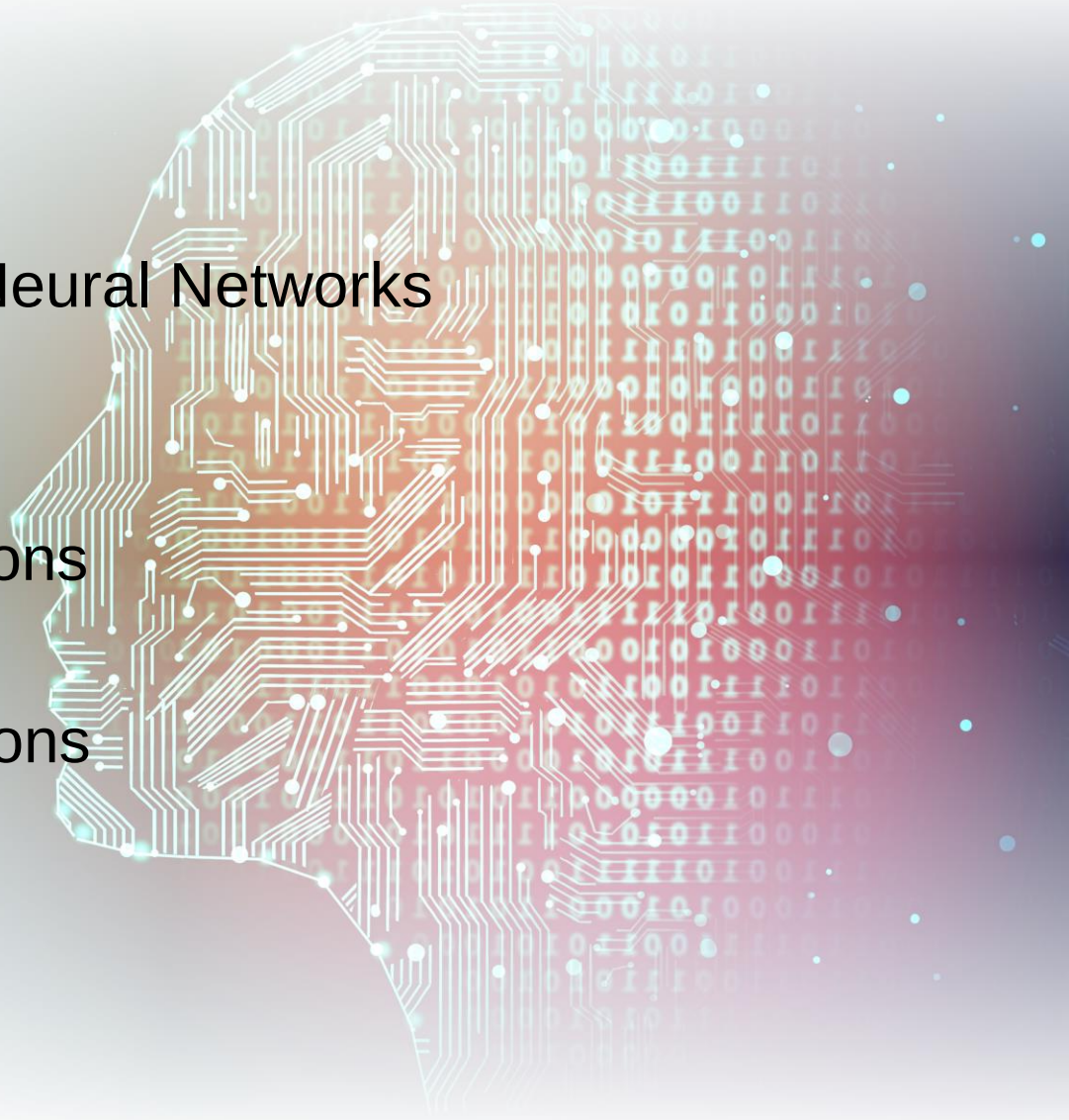
Mastering Shallow Neural Networks from Training to RTL Implementation

Tom Richter, Application Engineer, MathWorks
trichte@mathworks.com

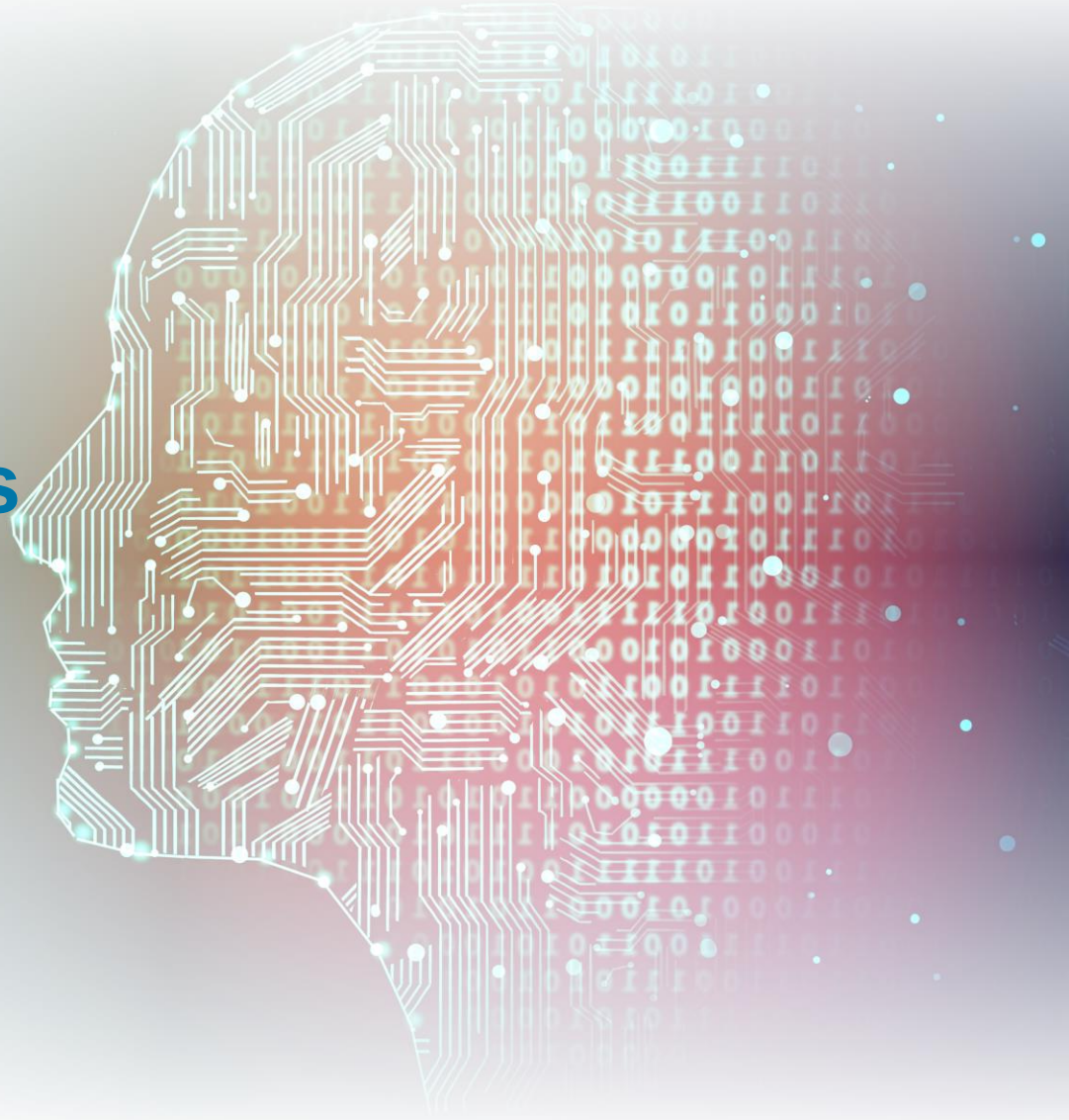
MathWorks®

Outline

- Overview about Neural Networks
- Ways to model, train, and validate Shallow Neural Networks
- Quantization
- RTL and report generation
- Vector-Matrix Multiplication optimization options
- Functional Verification
- Deep Learning Network implementation options
- Questions



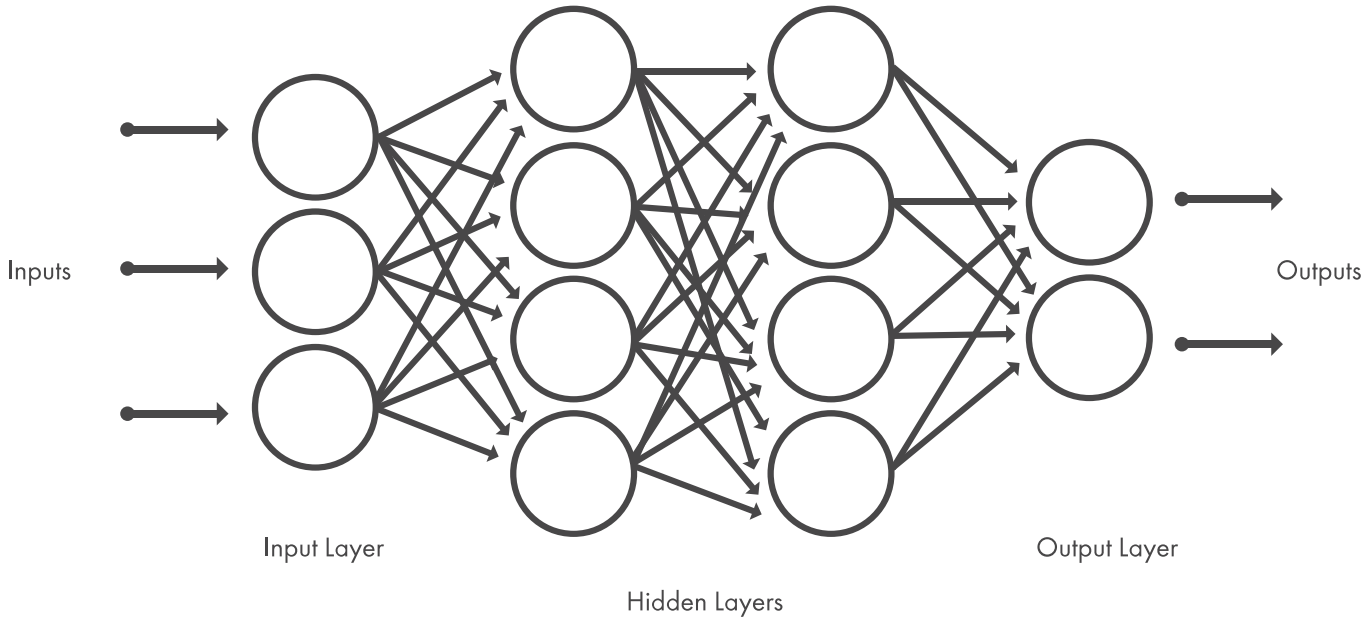
Overview about Neural Networks

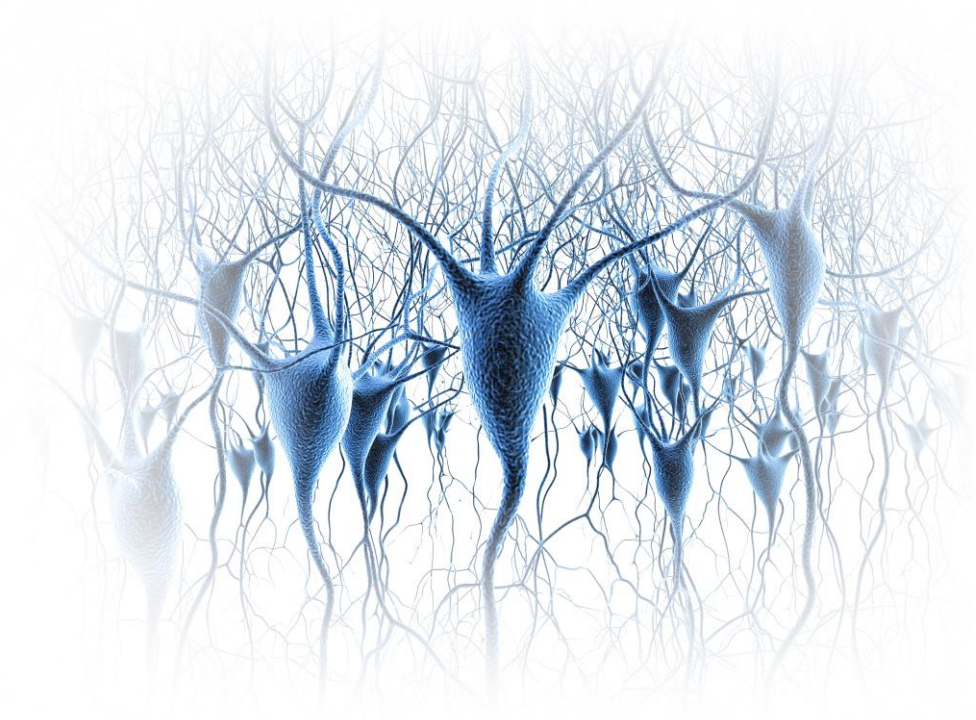


Neural Networks

In Machine Learning, a Neural Network is a model inspired by the structure and function of biological neural networks in brains.

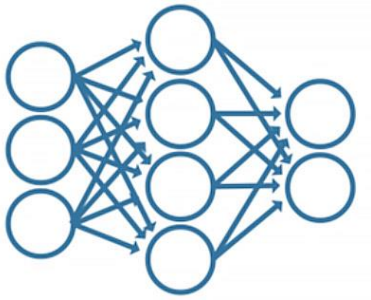
- Nodes model neurons, these are connected by edges which model synapses
- Nodes/Neurons are aggregated into layers: Input, Hidden, and Output
- Outputs of Neurons are the sums of weighted inputs through a non-linear activation function



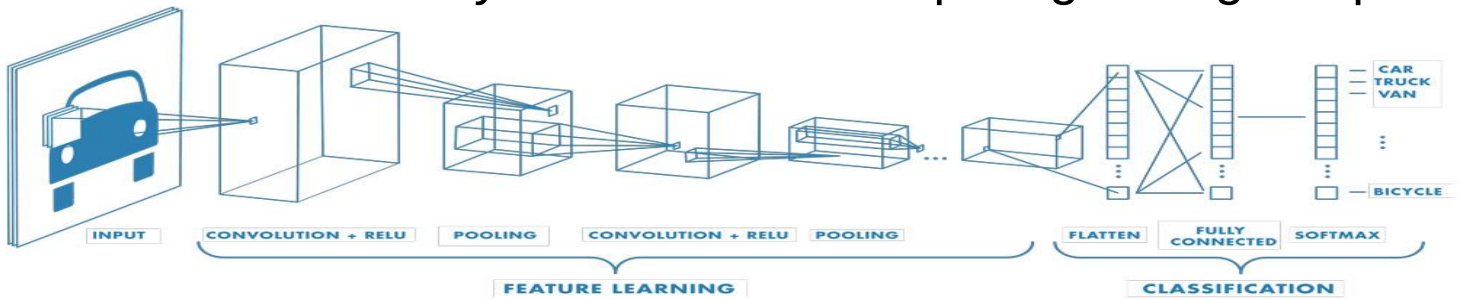


Kinds of Neural Networks

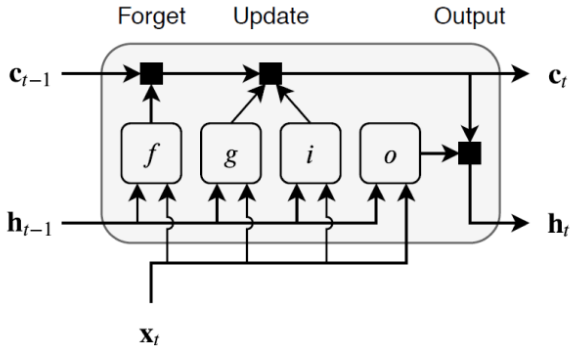
There are numerous neural networks, each featuring its own distinct structure and purpose. For example:



- Feedforward Neural Networks (FNN)
  - using Fully Connected Layers with an Activation Function
- Convolutional Neural Networks (CNN)
  - also requires Convolutional Layers which are computing sliding dot-products



- Recurrent Neural Networks (RNN)
  - Having internal states which are used to compute the next state
- Long Short-Term Memory networks (LSTM)
  - As RNNs, having internal states but can forget as well



Shallow vs. Deep Neural Network

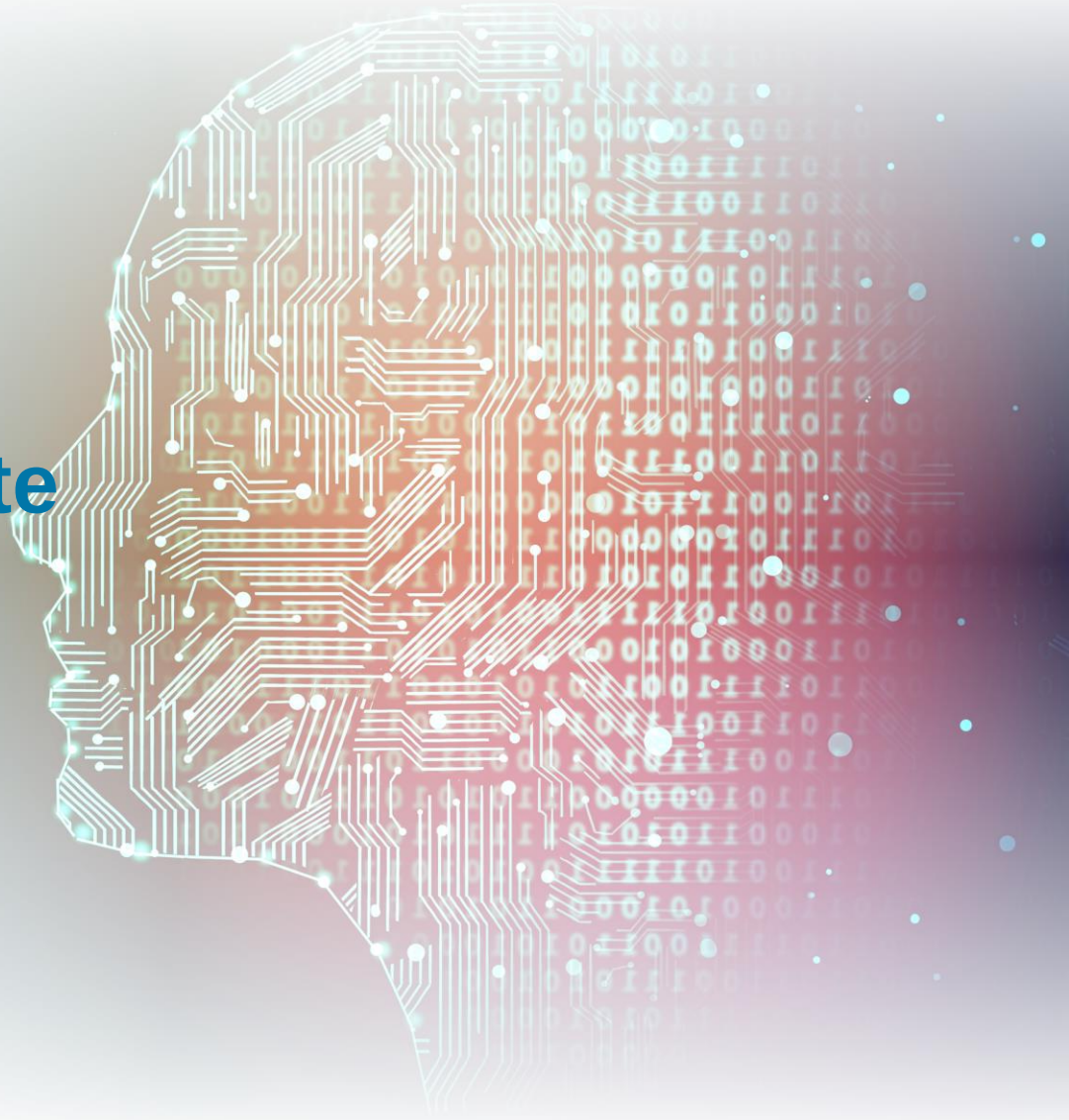
When becomes a Neural Network a Deep Neural Network?

	Shallow Neural Network	Deep Neural Network
Number of hidden layers	a few (maybe 1-5)	many (tens or hundreds)
Complexity of the layers	low	high
Learning capability	limited	great
Risk of overfitting	lower	higher
Memory for parameters	low (normally less than 1 KB)	high (often more than 1 MB)
Examples	Regression networks (FNNs)	CNNs for image recognition

A direct implementation is possible and beneficial

A processor or FSM implementation is necessary together with RAM access

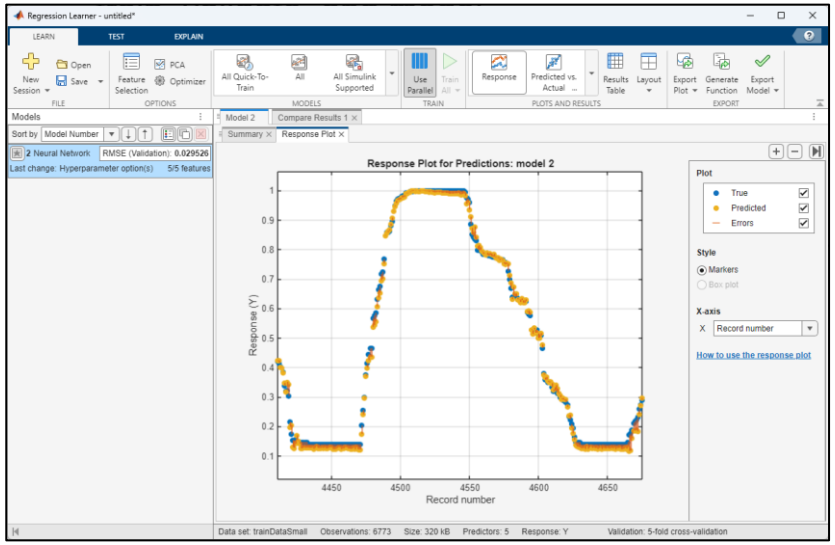
Ways to model, train, and validate Shallow Neural Networks



How getting the Neural Network?

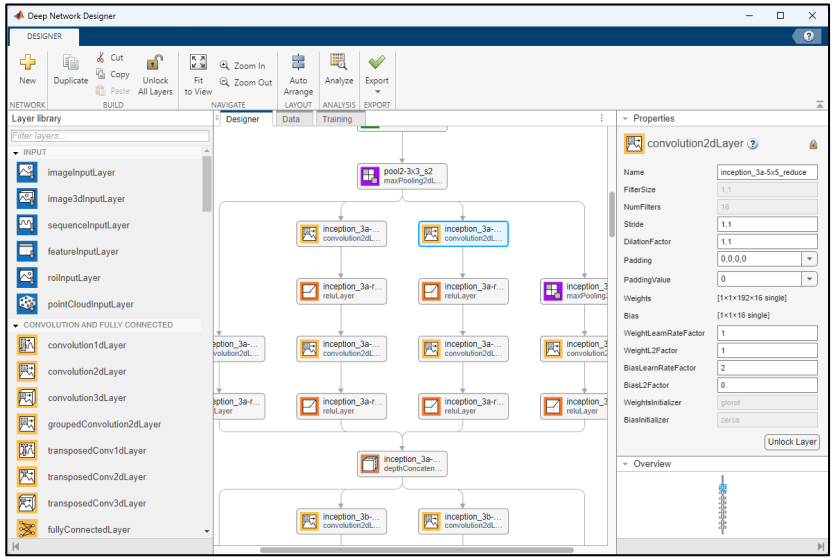
Statistics and Machine Learning Toolbox to define, train, validate, and model

- Shallow Regression Networks or
- Shallow Classification Networks



Deep Learning Toolbox for designing and implementing deep neural networks with algorithms, pretrained models, and apps. Features:

- Build, edit, combine
- Load, import, analyze,
- Train and monitor (inclusive transfer learning)
- Quantize
- Export networks



Example – Battery State of Charge (SoC)

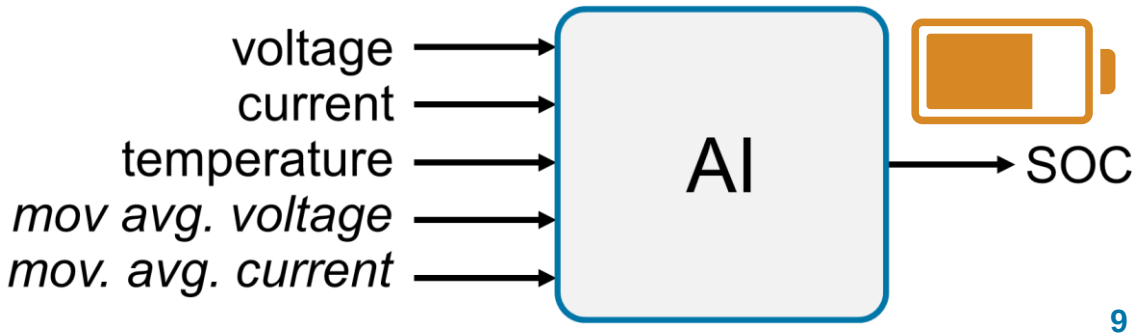
SoC estimation remains a significant challenge because of

- Nonlinear temperature and battery health
- Traditional approaches require precise parameters and battery knowledge

A data-driven approach using Neural Networks

- Requires minimal knowledge of the battery and its nonlinear characteristics
- Can predict the SoC as good as complex traditional methods (e.g.: Kalman)

Example is based on: [Deploy Neural Network Regression Model to FPGA/ASIC Platform]



Battery SoC – Load Data, Define, and Train a Neural Network

The original battery data used for training and validation comes from:

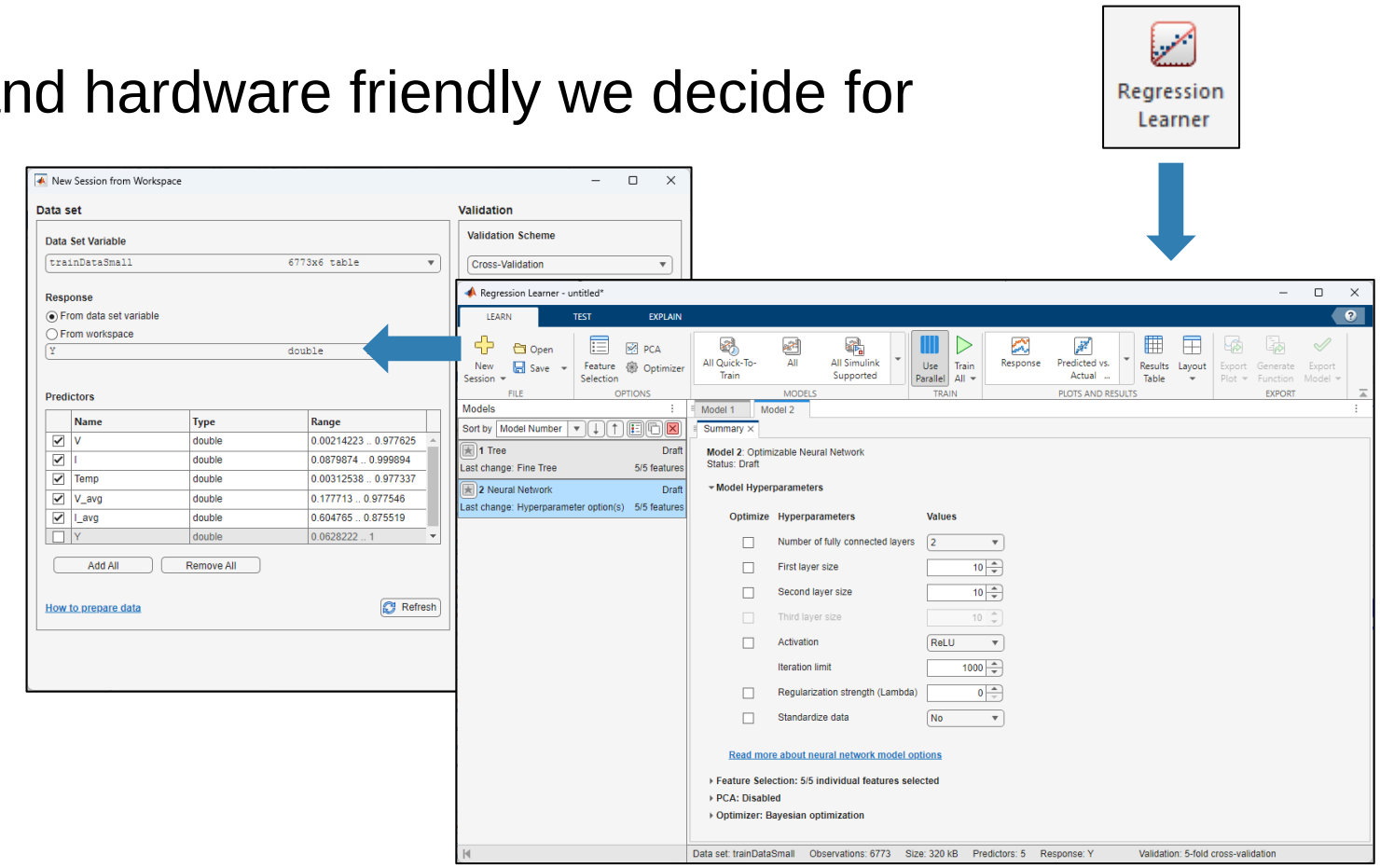
[LG 18650HG2 Li-ion Battery Data and Example - McMaster University]

For keeping the network small and hardware friendly we decide for

- 2 Hidden layers with sizes 10
- Activations using RELU

Training can be achieved using

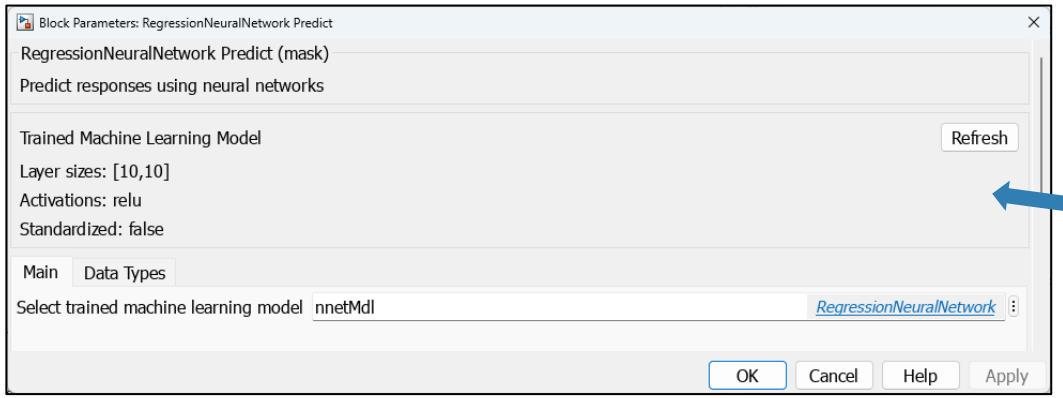
- Regression Learner App, or
- Function **`fitrnet`**



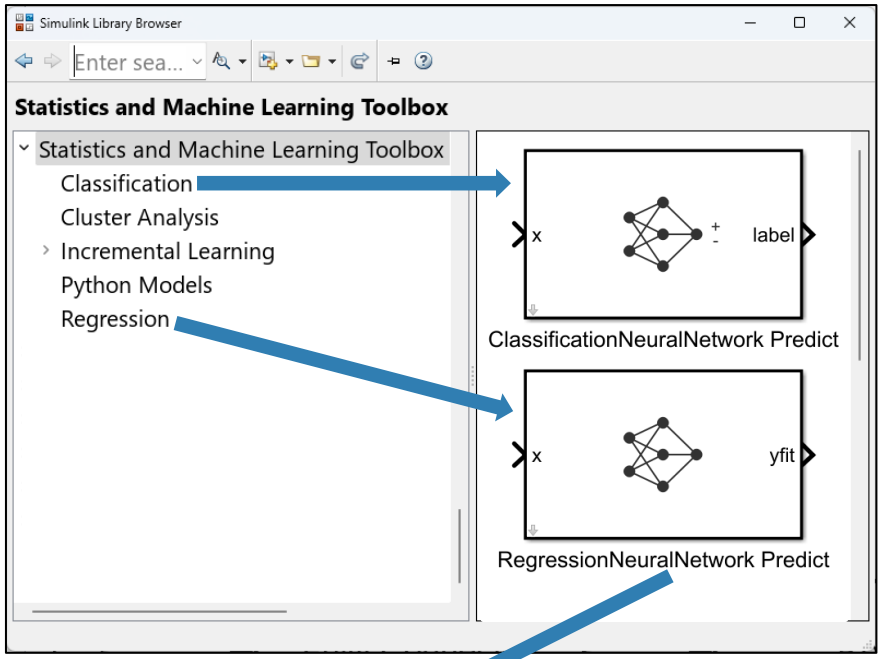
Battery SoC – Import Network Model to Simulink for Prediction

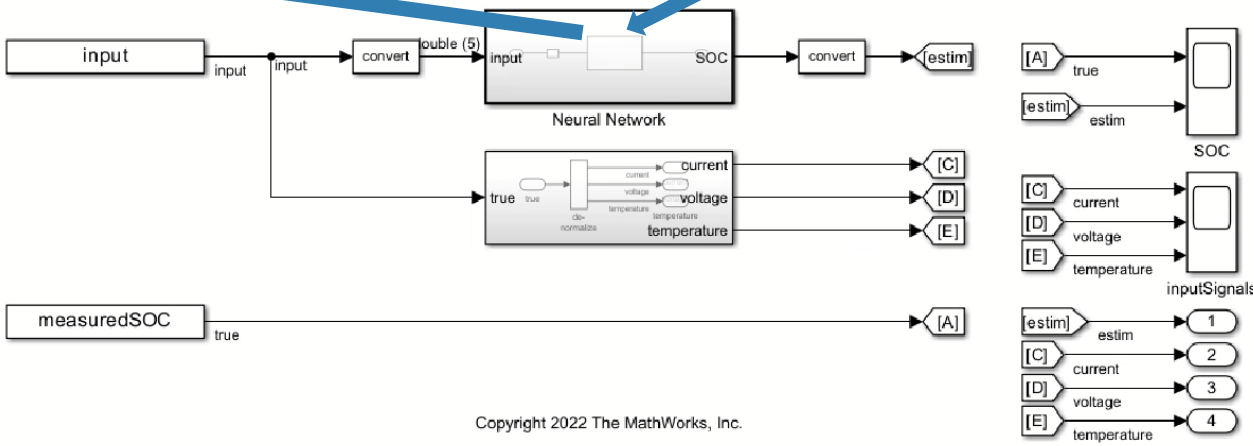
The Statistics and Machine Learning Toolbox Library comes with two blocks:

A Testbench model can be used with the Regression Neural Network Predict block.

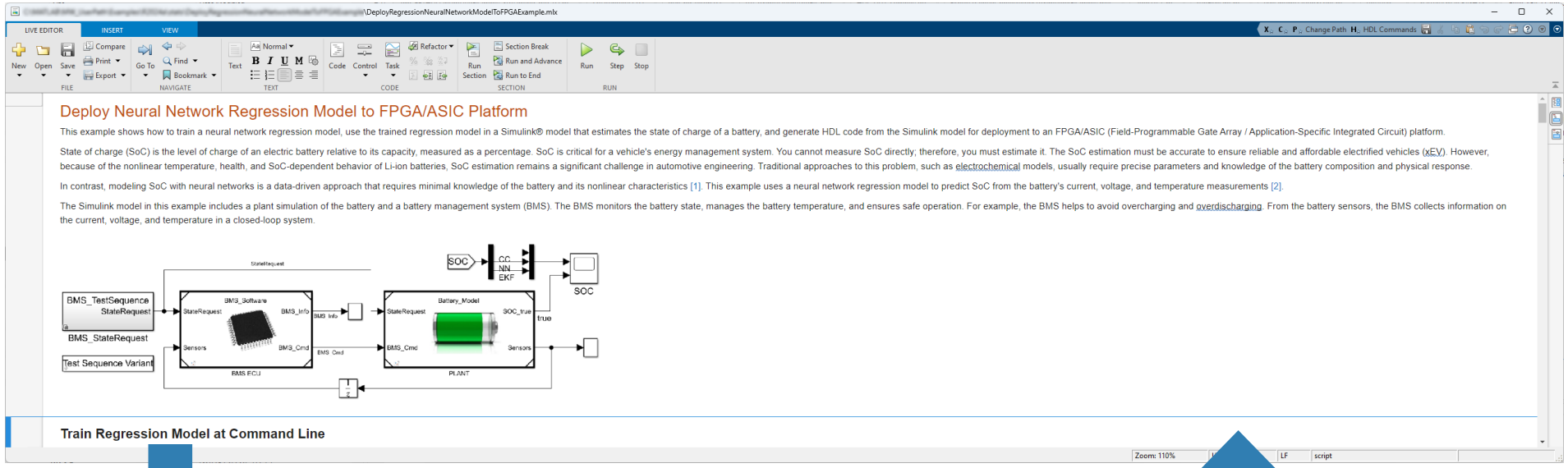


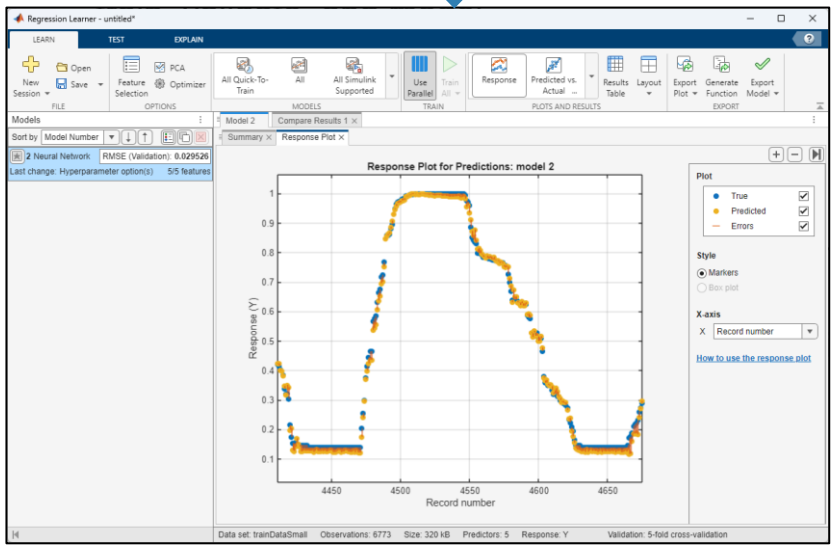
The trained Regression FNN model is the parameter for the block.

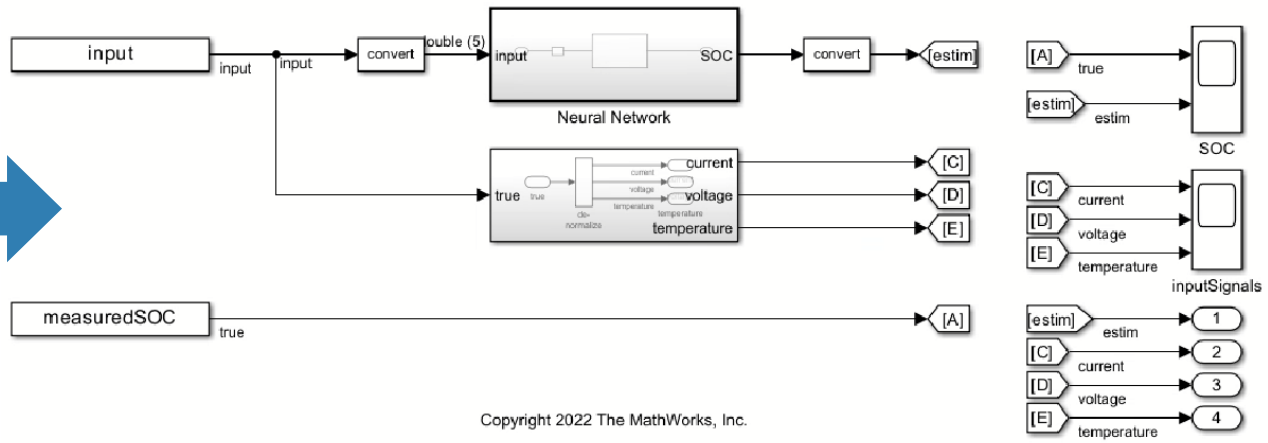




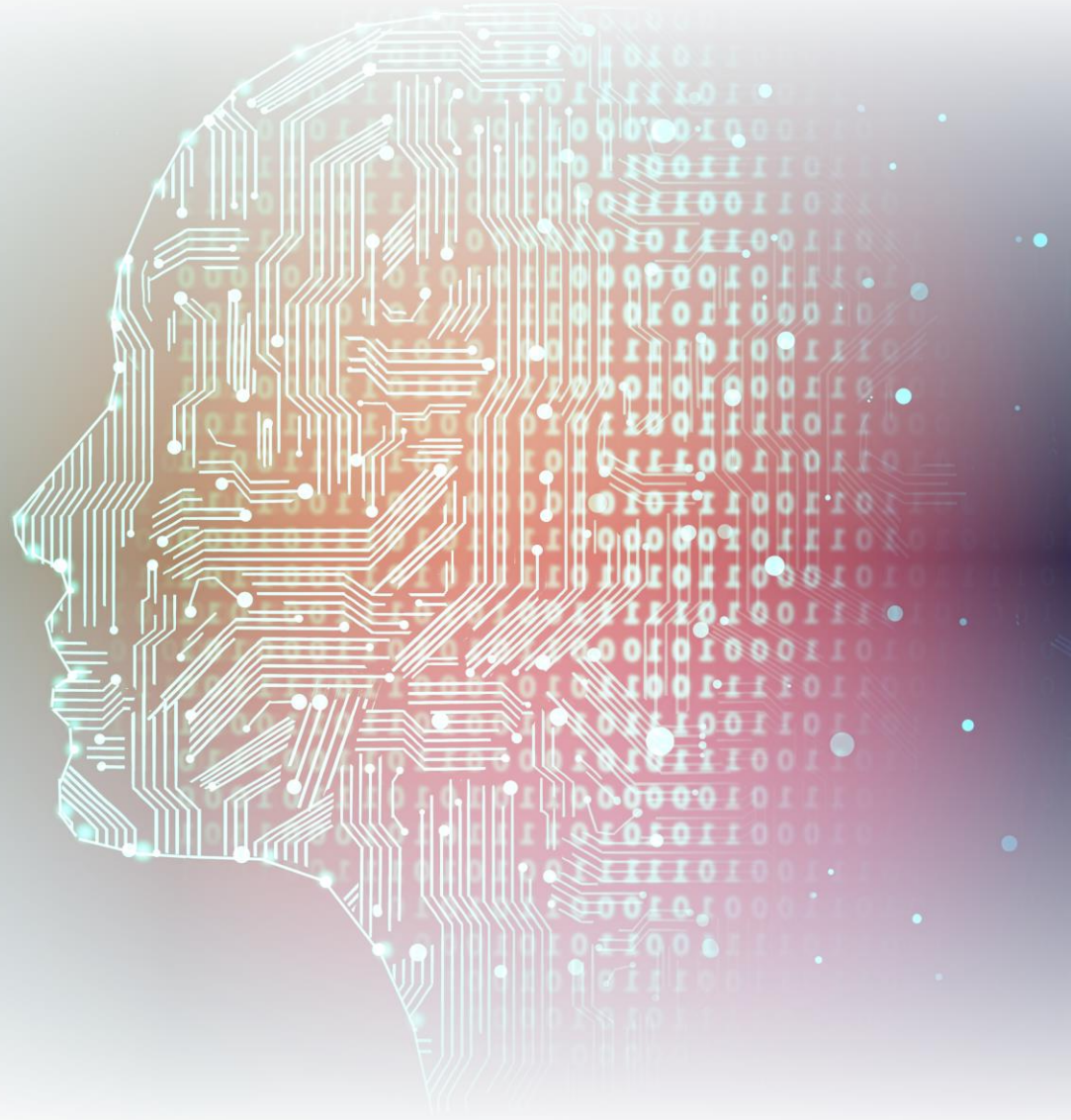
Demo – Battery SoC – From Training to Model





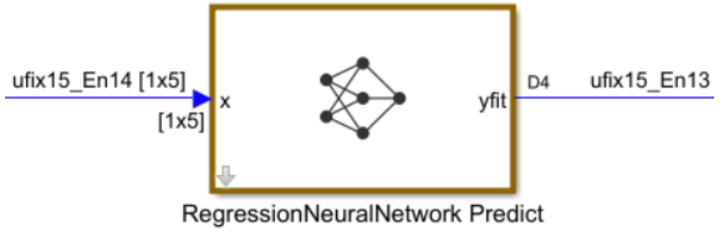


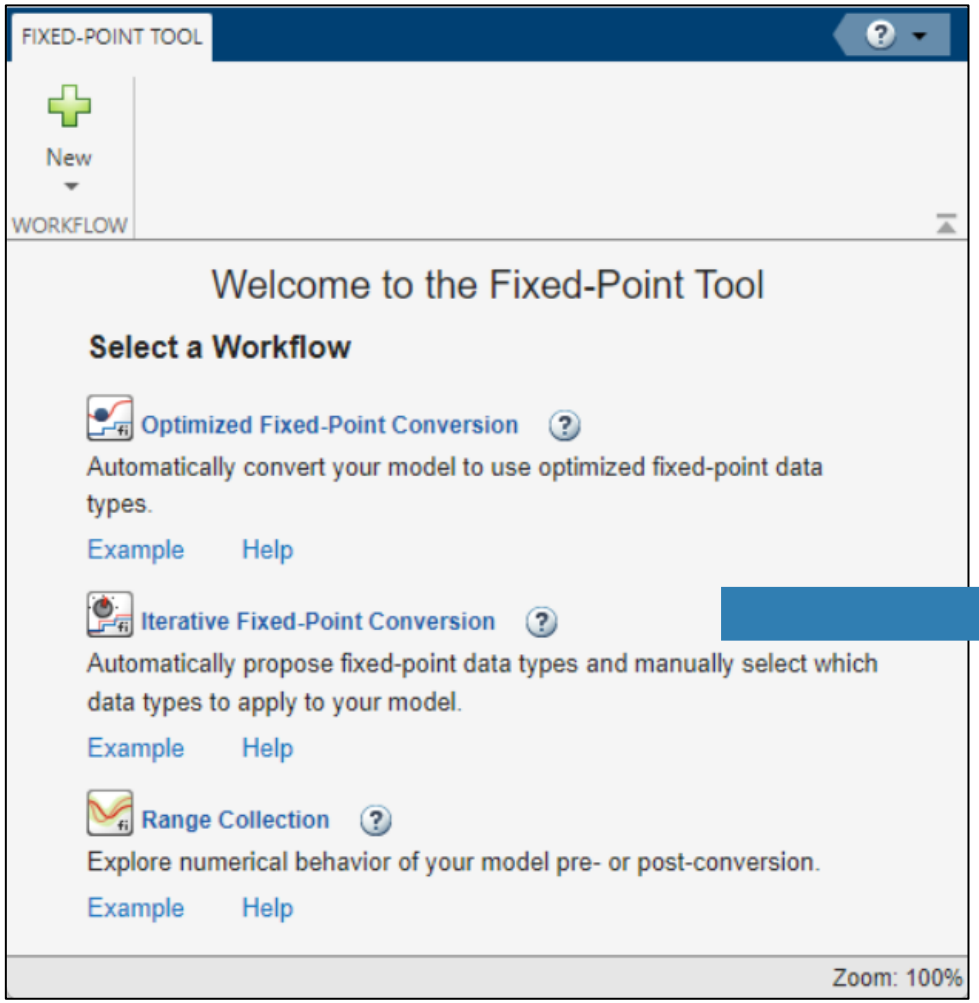
Quantization



Quantization or Fixed-Point Conversion

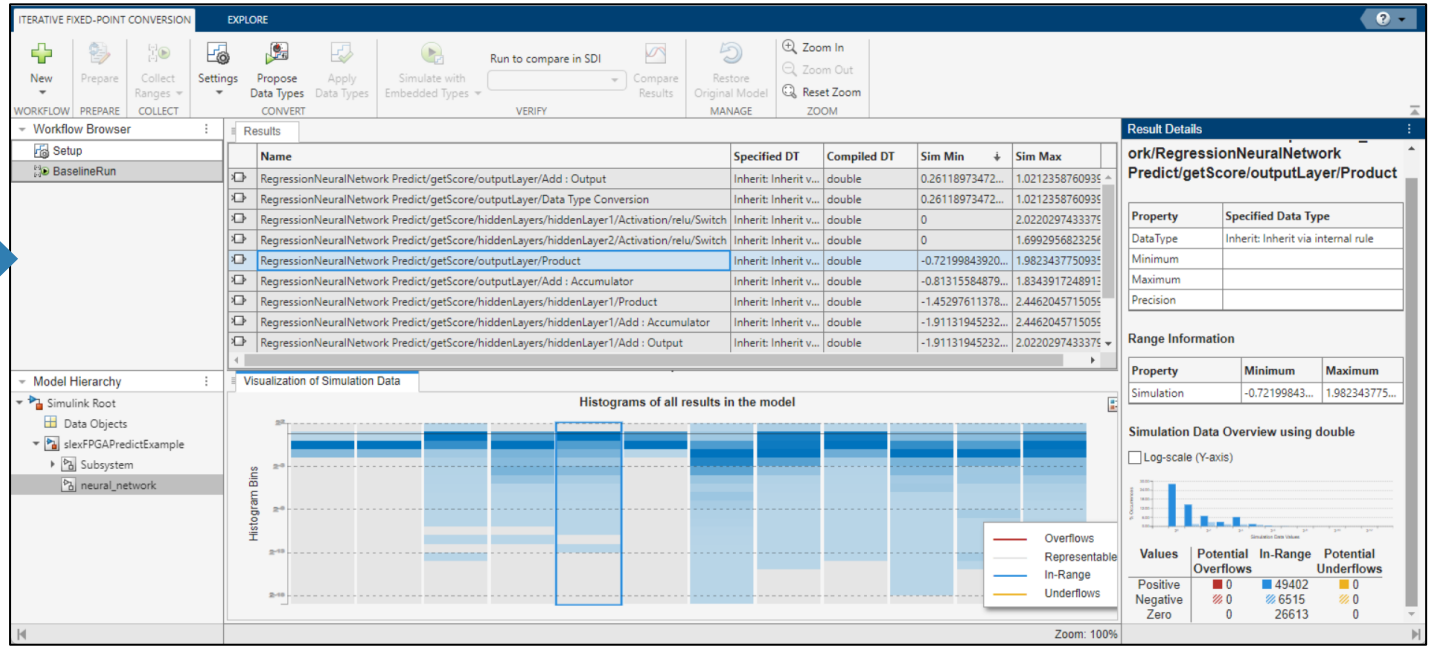
There exist several options for Fixed-Point conversion:






```
% Set options
opt = fxpOptimizationOptions('AllowableWordLengths', 10:24);
addTolerance(opt, sud, 1, 'AbsTol', 10e-2);
% Run conversion
result = fxpopt(model, sud, opt);
% Explore results
explore(result);
```

Automate with a Script or interactive using an App



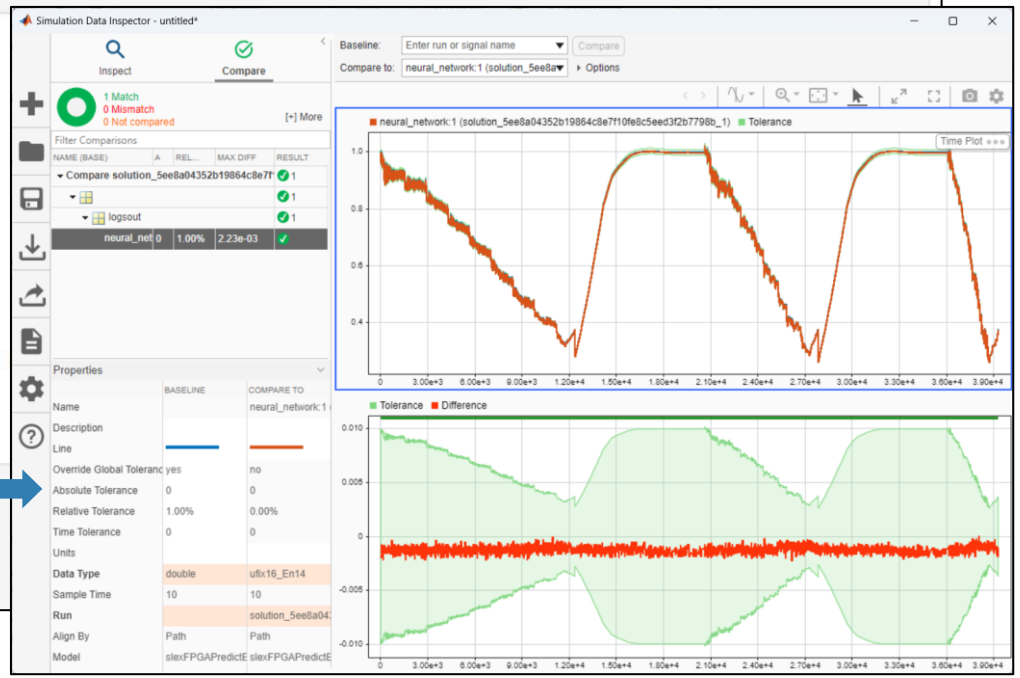
Demo – Battery SoC – Convert Simulink Model to Fixed-Point

Document the data type conversion steps by using the command line interface:

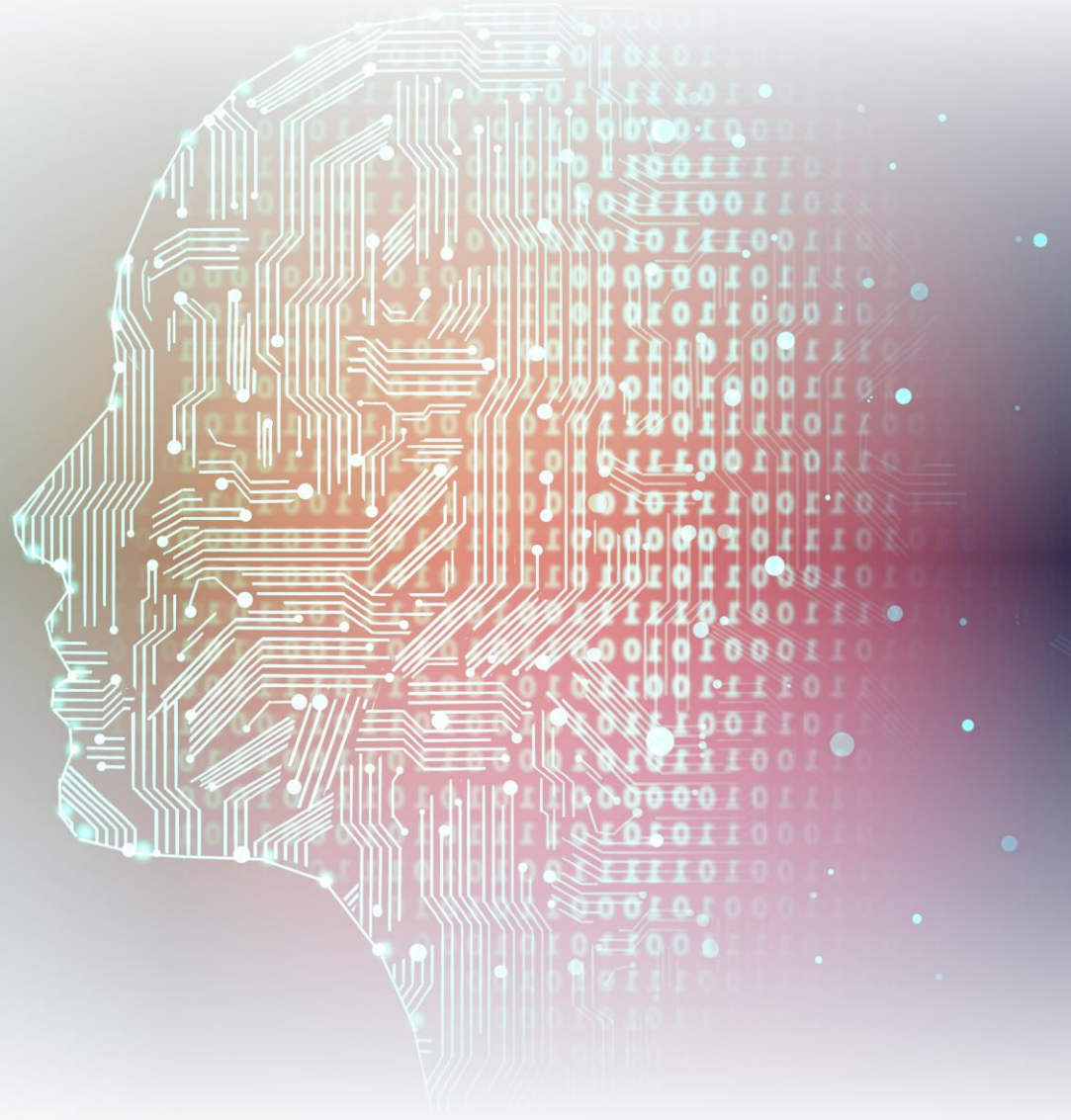
```
mdlInfo = get_param(modelName,'DataLoggingOverride');

sud = [modelName, '/neural_network'];
options = fxpOptimizationOptions();
options.AdvancedOptions.PerformNeighborhoodSearch = false; % Perform a neighborhood search to further optimize.
options.AdvancedOptions.SafetyMargin = 100;                % Safety margin to allow for higher ranges.
options.AllowableWordLengths = 13:18;                      % Word lengths that can be used.

addTolerance(options, sud, 1, 'RelTol', 0.01);             % Relative tolerance for the subsystem output
result = fxpopt(modelName, sud, options);
```

```
+ Starting data type optimization...
+ Checking for unsupported constructs.
+ Preprocessing
+ Modeling the optimization problem
        - Constructing decision variables
+ Running the optimization solver
        - Evaluating new solution: cost 650, does not meet the behavioral constraints.
        - Evaluating new solution: cost 698, does not meet the behavioral constraints.
        - Evaluating new solution: cost 746, does not meet the behavioral constraints.
        - Evaluating new solution: cost 794, meets the behavioral constraints.
        - Updated best found solution, cost: 794
+ Optimization has finished.
+ Fixed-point implementation that satisfies the behavioral constraints found. The best found solution
        - Total cost: 794
        - Maximum absolute difference: 0.002230
        - Use the explore method of the result to explore the implementation.
```

```
solution = explore(result);

set_param(modelName,'DataLoggingOverride',mdlInfo)         % Reset settings changed by fxpopt
set_param(modelName,'SaveOutput','on')
```



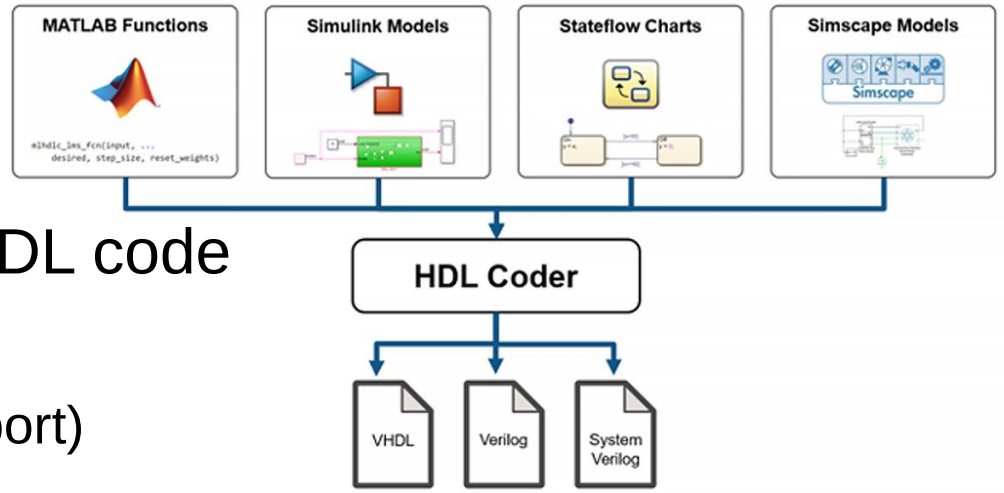
RTL and Report Generation



RTL Generation, Optimization, and Verification

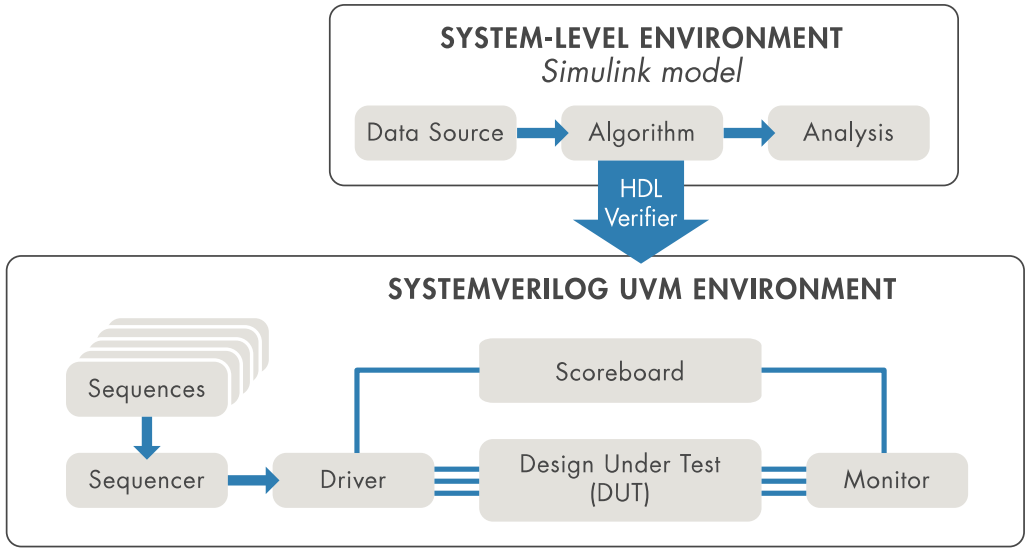
RTL Generation:

- Portable for FPGAs and ASICs
- Synthesizable Verilog, SystemVerilog, and VHDL code
- Specific features are:
  - Two-way traceability between Model and Code (Report)
  - Timing and Area optimization options
  - Floating-Point Support for IEEE-754 double, single, and half-precision data types with
    - Denormal Numbers
    - Exceptions such as NaN, Inf, and Zero
    - Customizable latency options
    - Many supported math and trigonometric functions



Functional Verification:

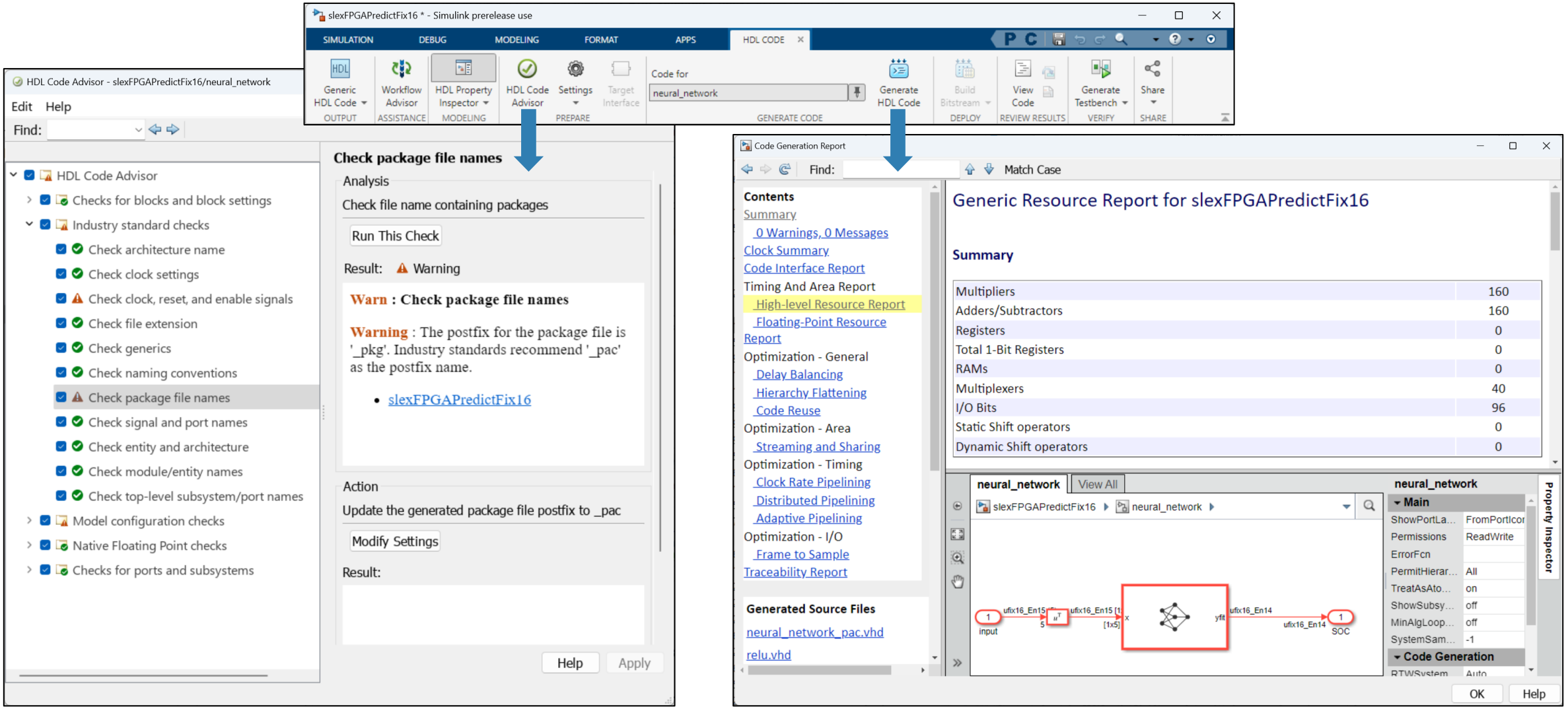
- Generate testbenches (Co-Simulation, SystemVerilog and UVM)



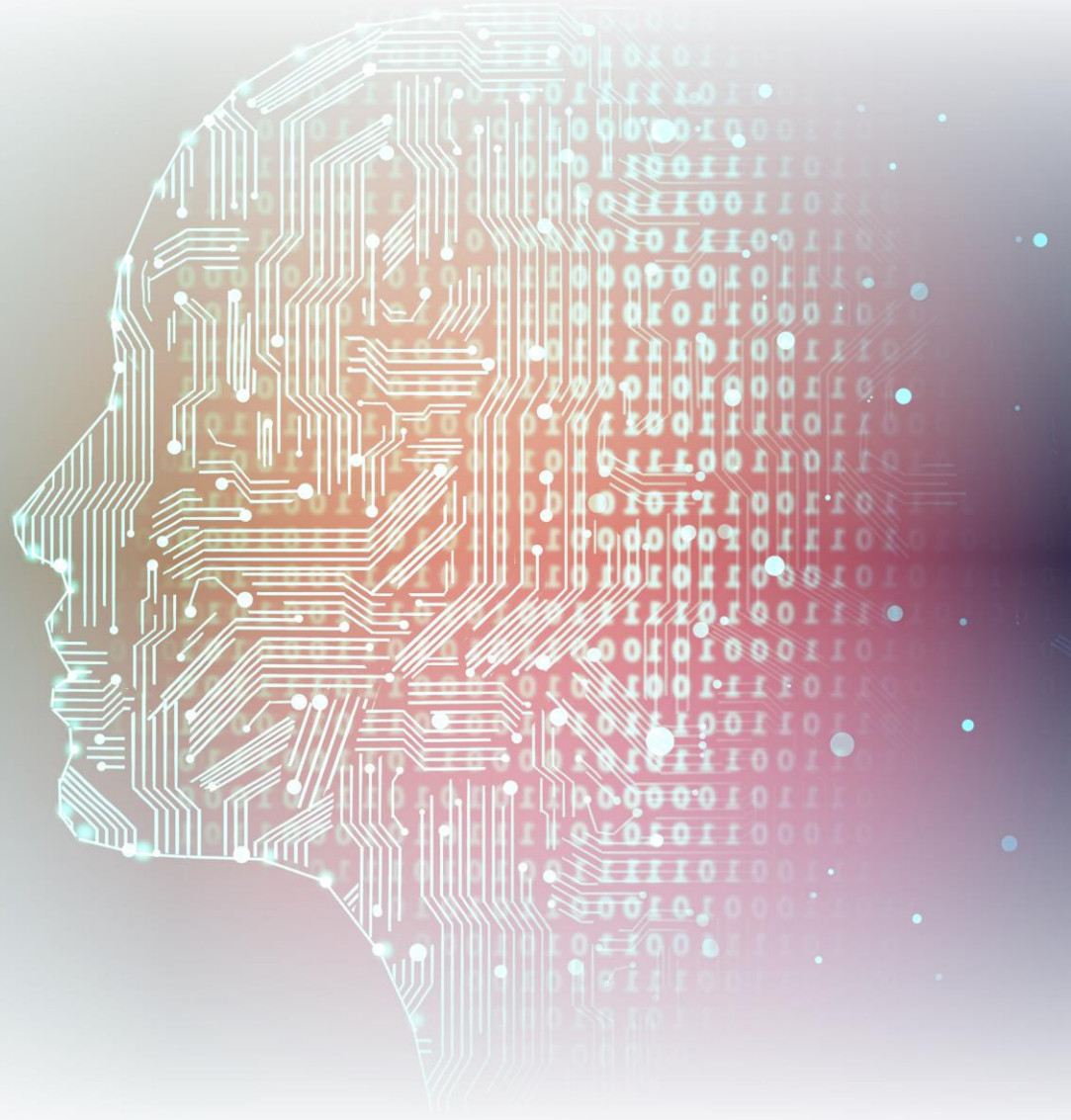
Demo – Battery SoC – RTL Generation

First, prepare the model.

Second, generate code and reports.

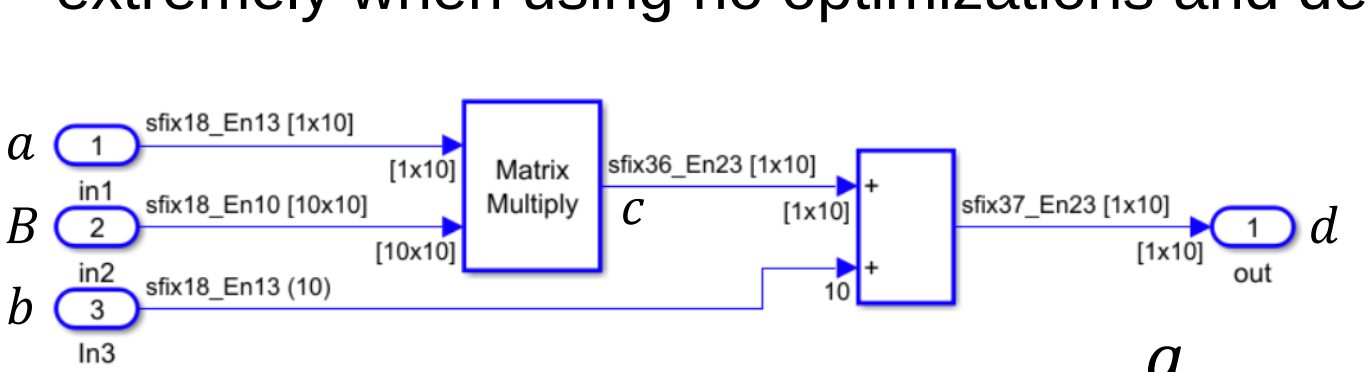


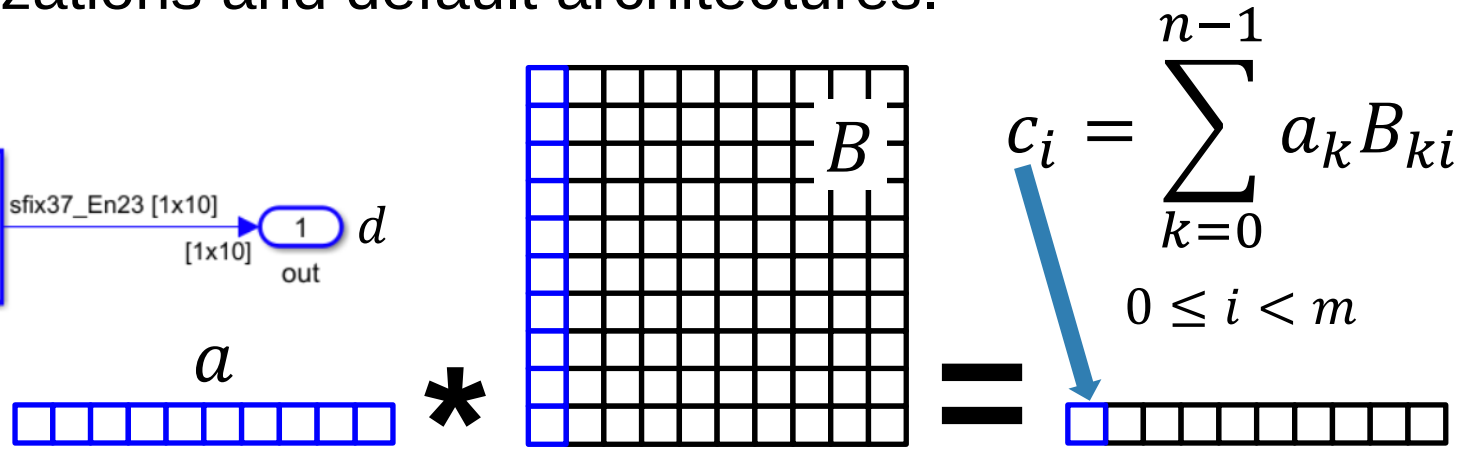
Vector-Matrix Multiplication Optimization Options



Vector-Matrix Multiplication with Bias – Default

Vector-Matrix or even Matrix-Matrix multiplication can increase the area usage extremely when using no optimizations and default architectures.





- The product 1xn * nxm = 1*m, requires n*m multipliers and (n-1)*m adders. For n = m = 10, that is 100 multipliers and 90 adders.
- For the Bias, additional m adders are required (10 adders)

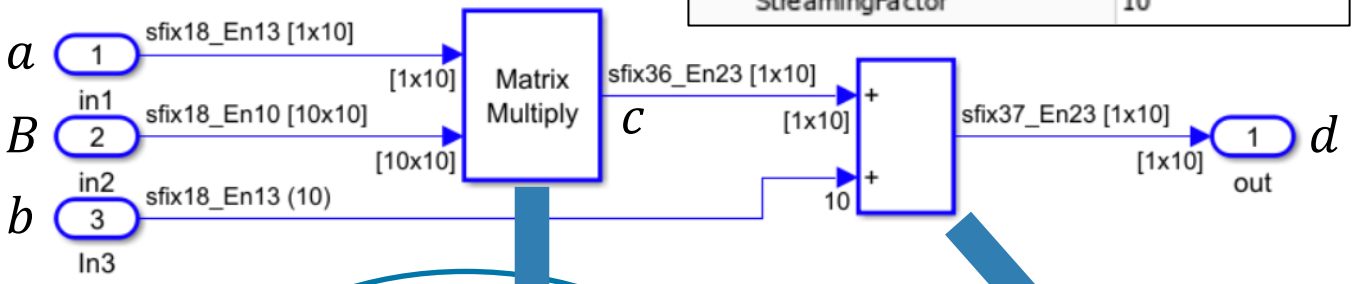
Multipliers	160
Adders/Subtractors	160
Registers	0
Total 1-Bit Registers	0
RAMs	0
Multiplexers	40
I/O Bits	96
Static Shift operators	0
Dynamic Shift operators	0

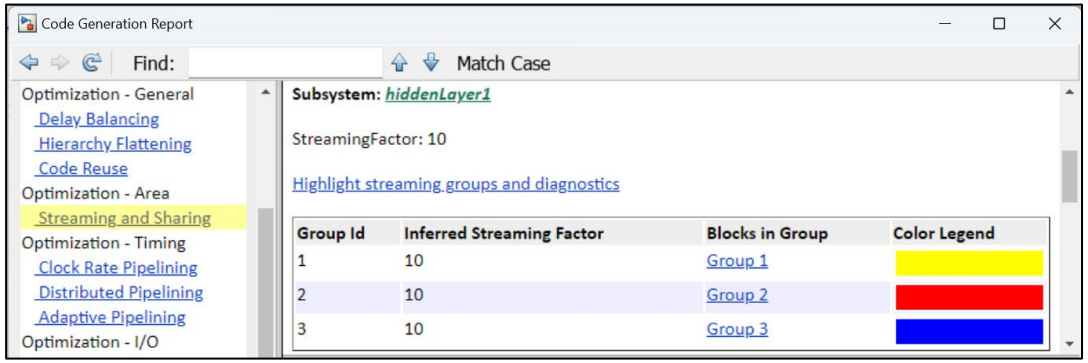
However, do you need such a high throughput here?

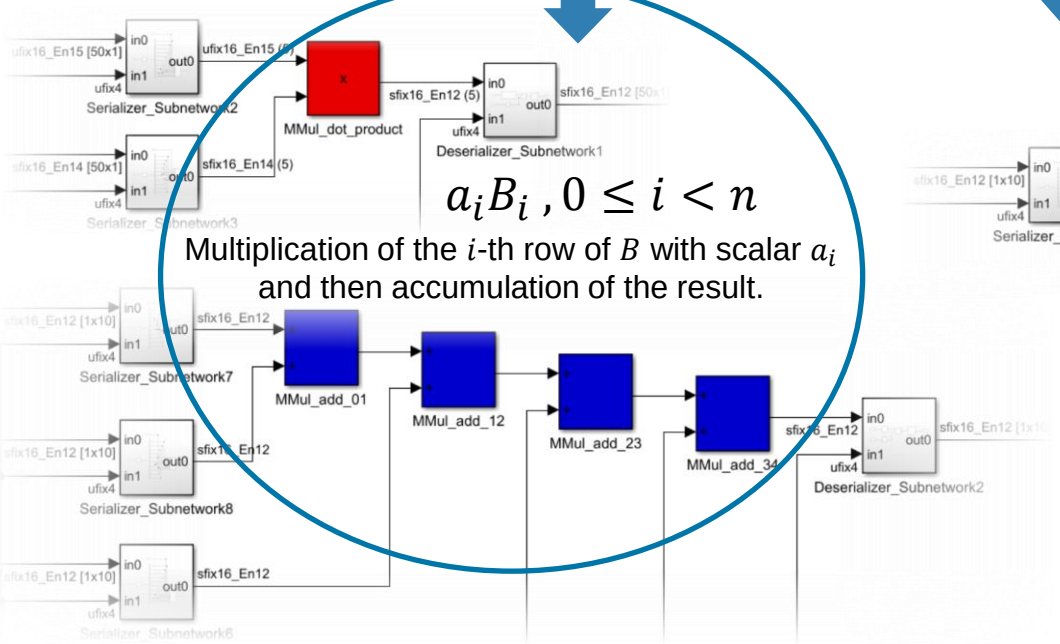
Vector-Matrix Multiplication with Bias – Streaming

With the Optimization features Streaming or Sharing you can reduce the hardware costs in exchange of reducing the maximal throughput.

SharingFactor	0
StreamingFactor	10

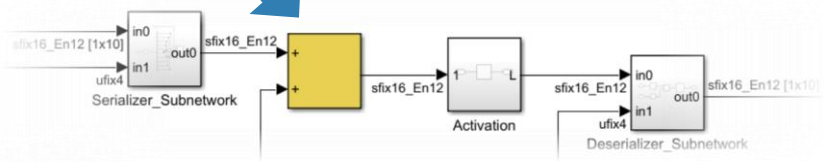






$a_i B_i \, , 0 \leq i < n$

Multiplication of the i-th row of B with scalar a_i and then accumulation of the result.

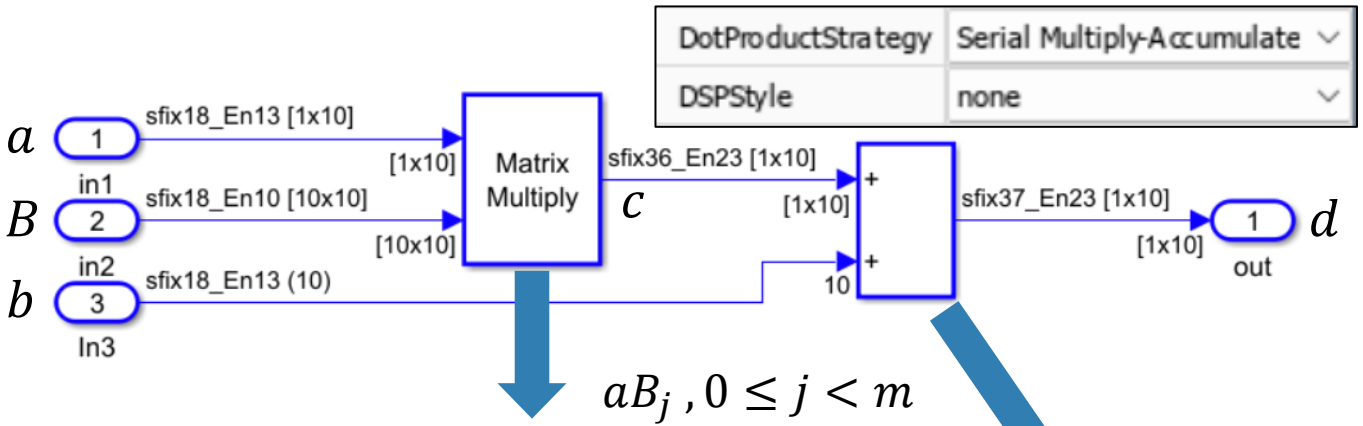


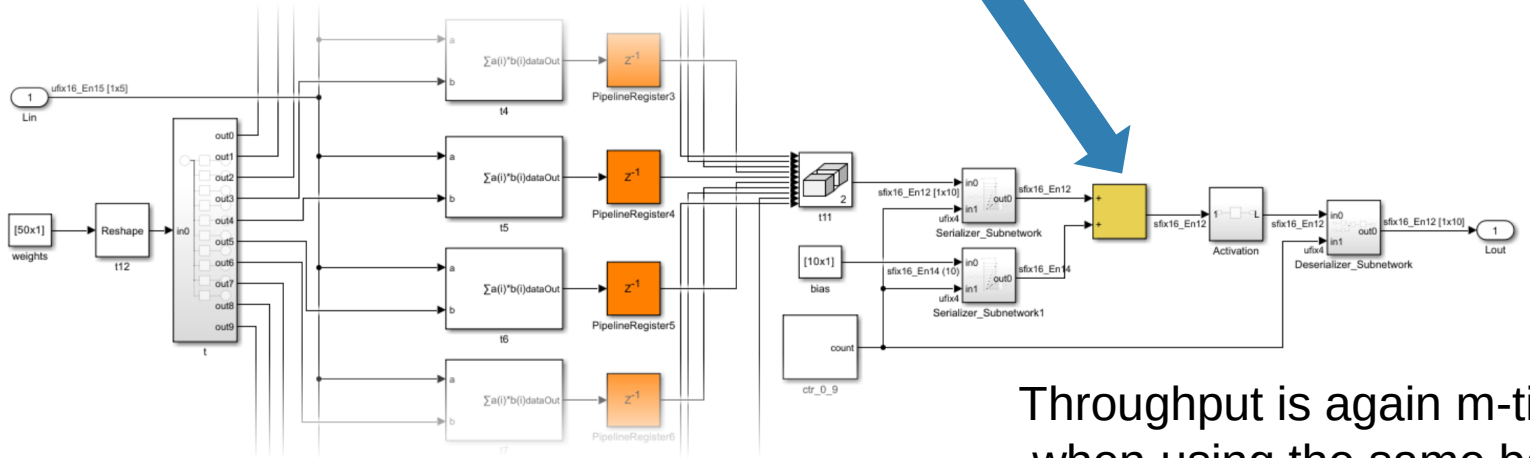
All this is carried out m-times and so the throughput is m-times lower when using the same base clock.

Multipliers	25
Adders/Subtractors	28
Registers	738
Total 1-Bit Registers	11721
RAMs	0
Multiplexers	63
I/O Bits	100
Static Shift operators	0
Dynamic Shift operators	0

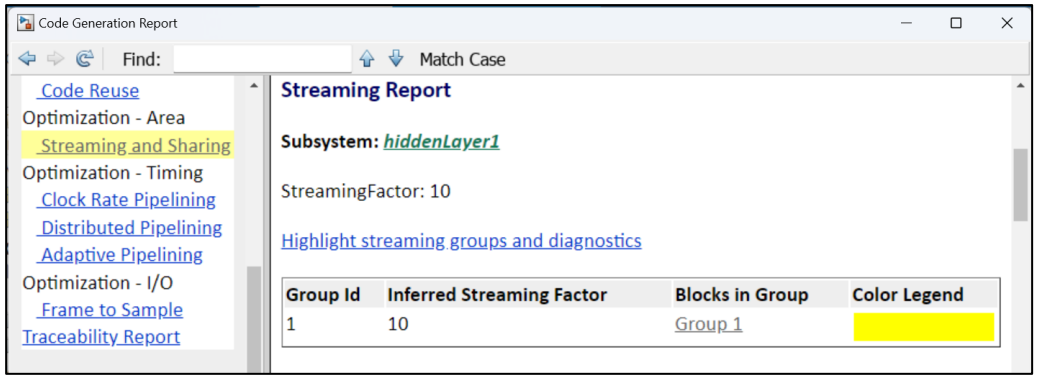
Vector-Matrix Multiplication with Bias – Architecture Setting

The Matrix-Multiply block has additional architecture settings that can be used together with Streaming.





Scalar-multiplication of a with the j-th column of B by using a Multiply-Accumulate block.



Throughput is again m-times lower when using the same base clock. Overall hardware costs are lower.

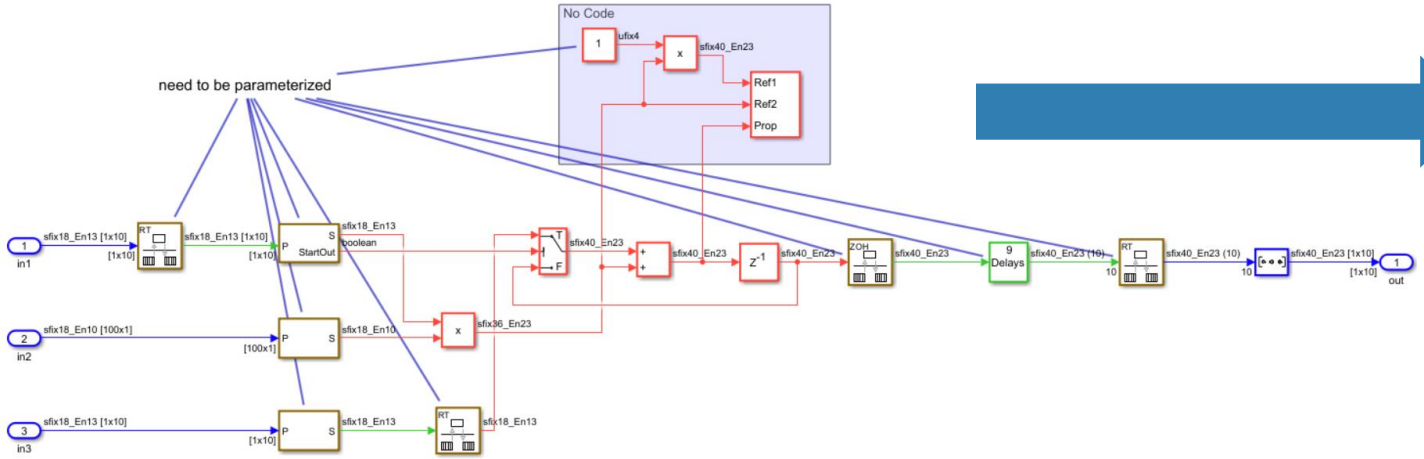
Note: The number of adders is higher, but most are small (3 or 4 bit).

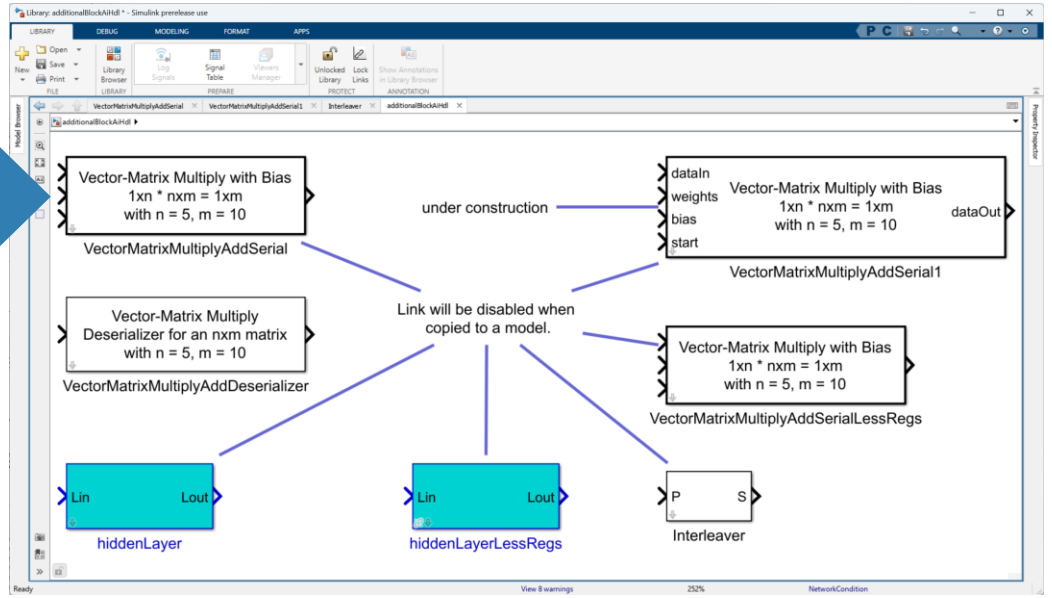
Multipliers	30
Adders/Subtractors	55
Registers	137
Total 1-Bit Registers	1575
RAMs	0
Multiplexers	135
I/O Bits	100
Static Shift operators	0
Dynamic Shift operators	0

Creating your own Blocks and Libraries

You don’t have to stick to existing blocks and their settings.

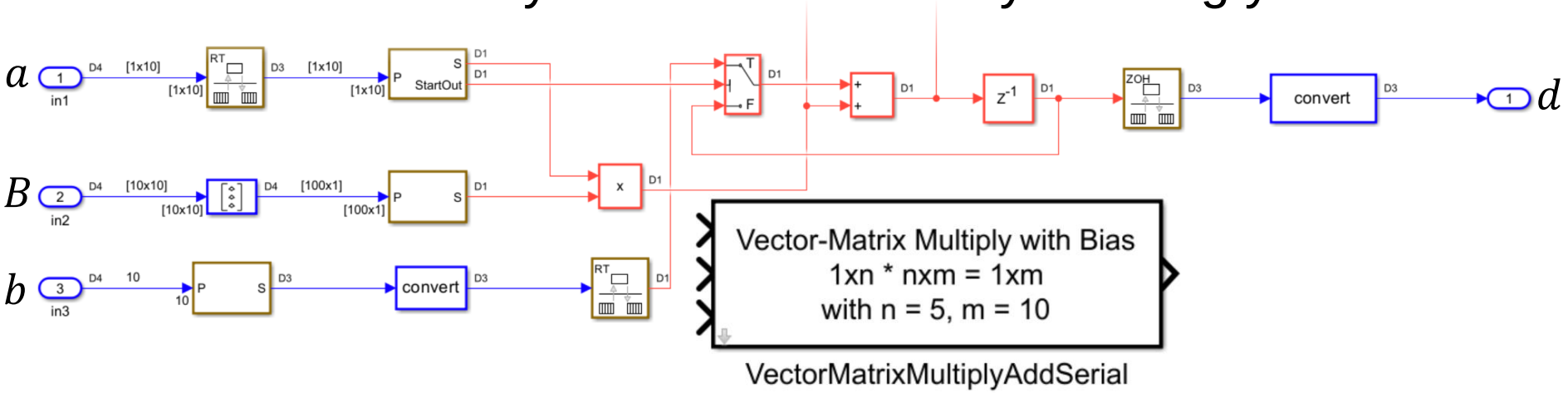
- Create your own models from basic library blocks (Add, Product, Delay, etc.)
- Test the model against existing blocks (Matrix-Multiply, Filter, etc.)
- Create a dialog for your model or subsystem for parameterization (masking)
- Add the new block together with a documentation to a library



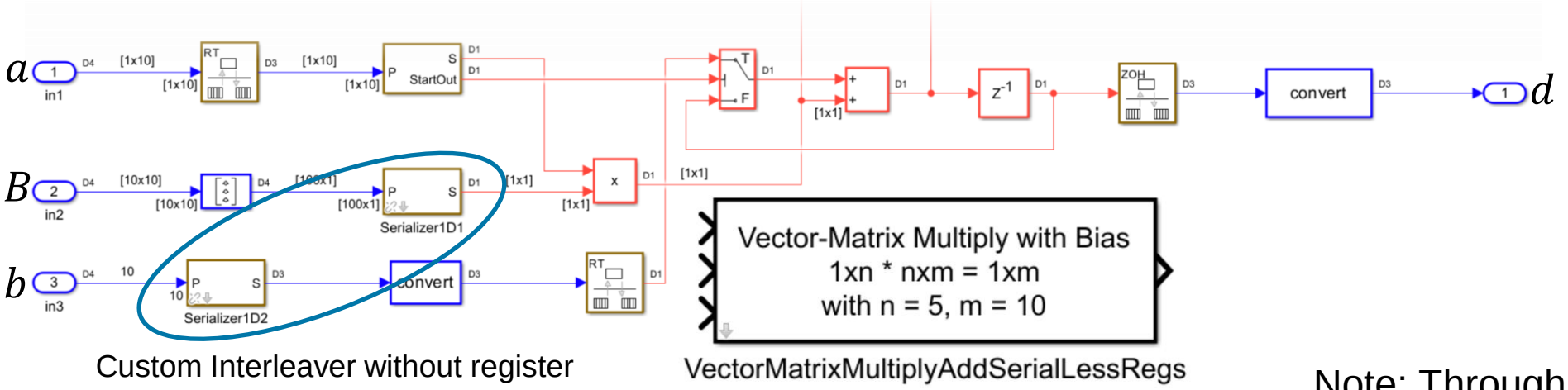


Vector-Matrix Multiplication with Bias – Custom Architectures

You can dramatically reduce the costs by creating your own library blocks.



Multipliers	3
Adders/Subtractors	13
Registers	267
Total 1-Bit Registers	3827
RAMs	0
Multiplexers	67
I/O Bits	94
Static Shift operators	0
Dynamic Shift operators	0

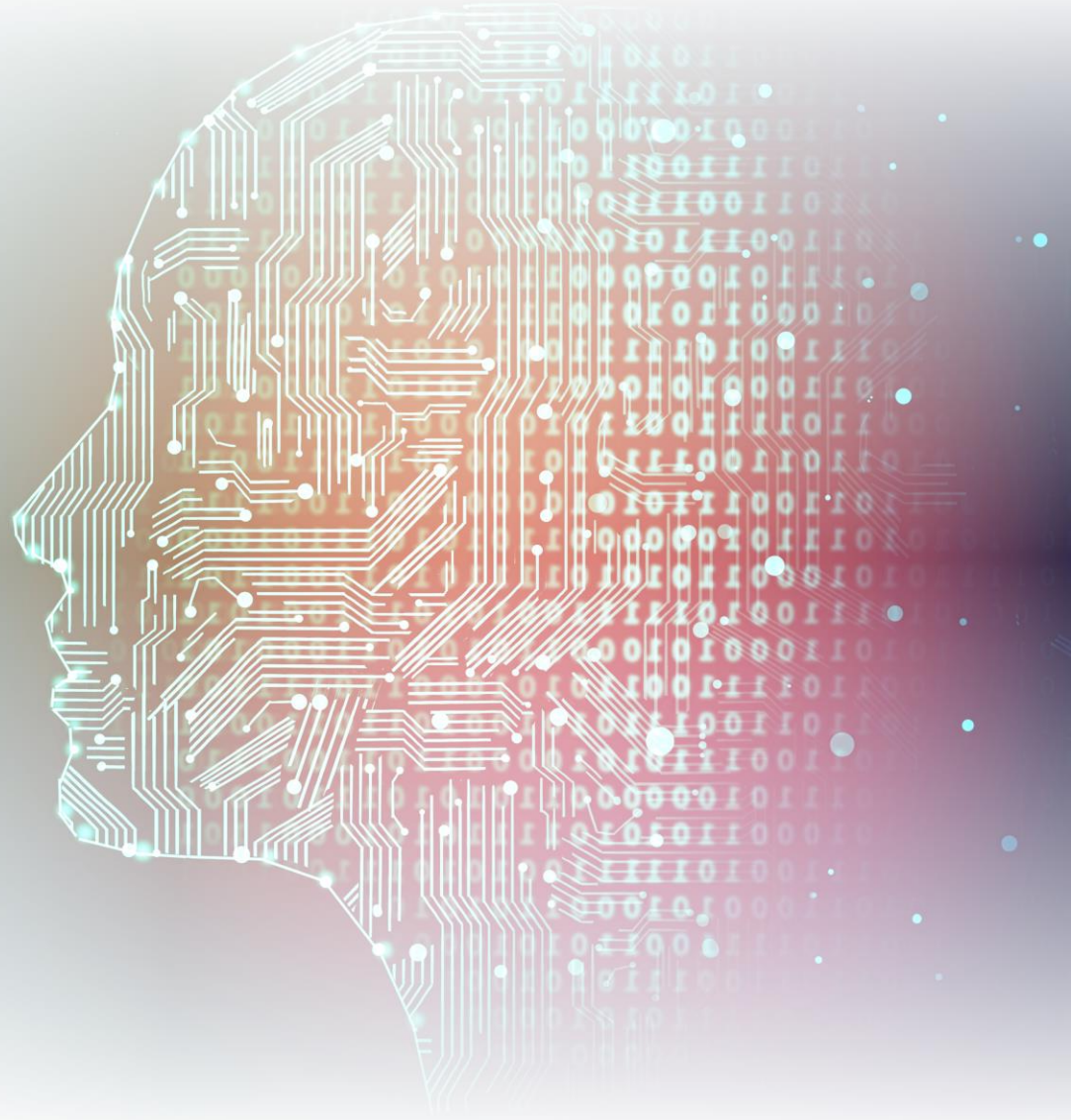


Multipliers	3
Adders/Subtractors	13
Registers	97
Total 1-Bit Registers	1316
RAMs	0
Multiplexers	63
I/O Bits	94
Static Shift operators	0
Dynamic Shift operators	0

Custom Interleaver without register usage because of constant input.
Small change – big impact.

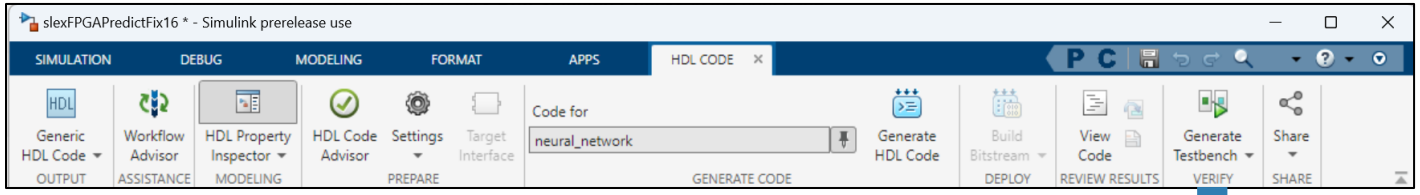
Note: Throughput is now (n*m)-times lower when using the same base clock.

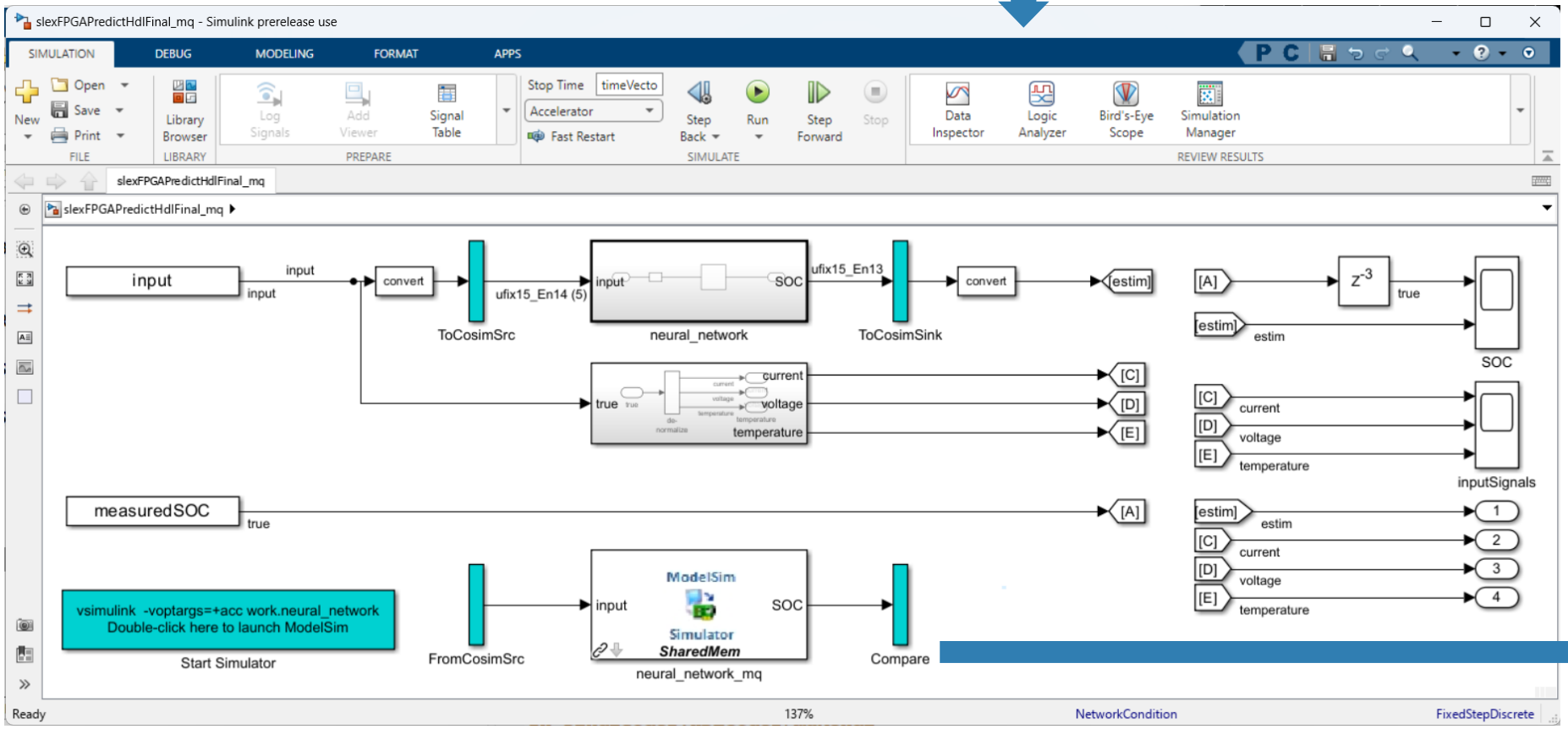
Functional Verification

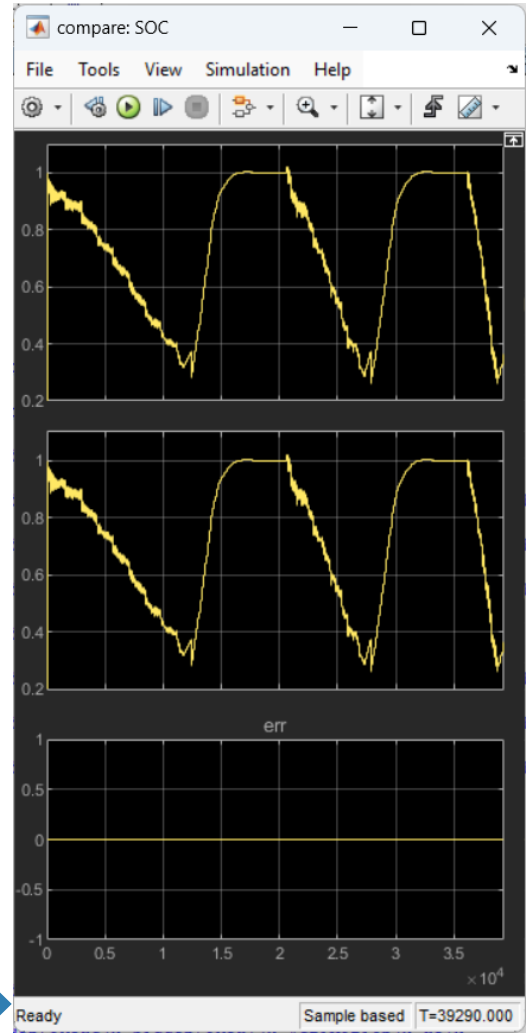


Demo – Battery SoC – RTL Verification

Once you generated code you can also generate a testbench model:



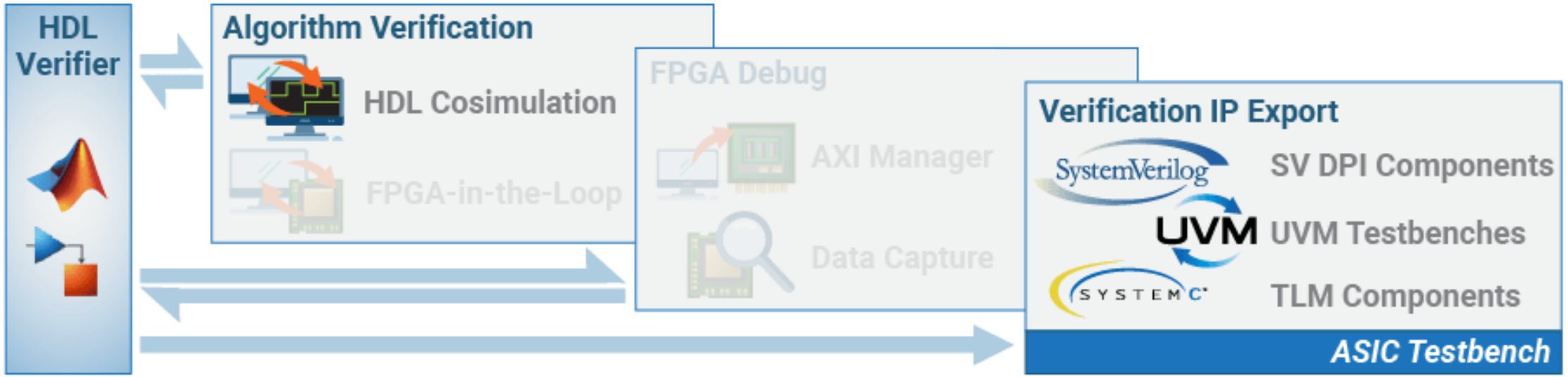




RTL Verification – Further Options

Co-simulation and other options can also be used to verify manually written code:

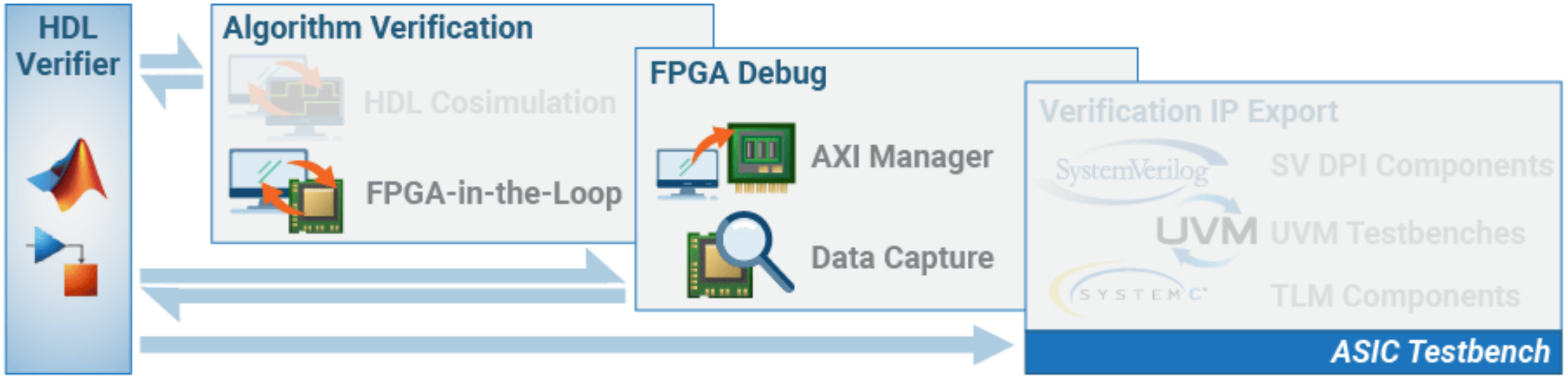
- SystemVerilog DPI-C component generation
- UVM component and testbench generation
- Option to generate a testbench to test individual SV, UVM components
- SystemC TLM component generation



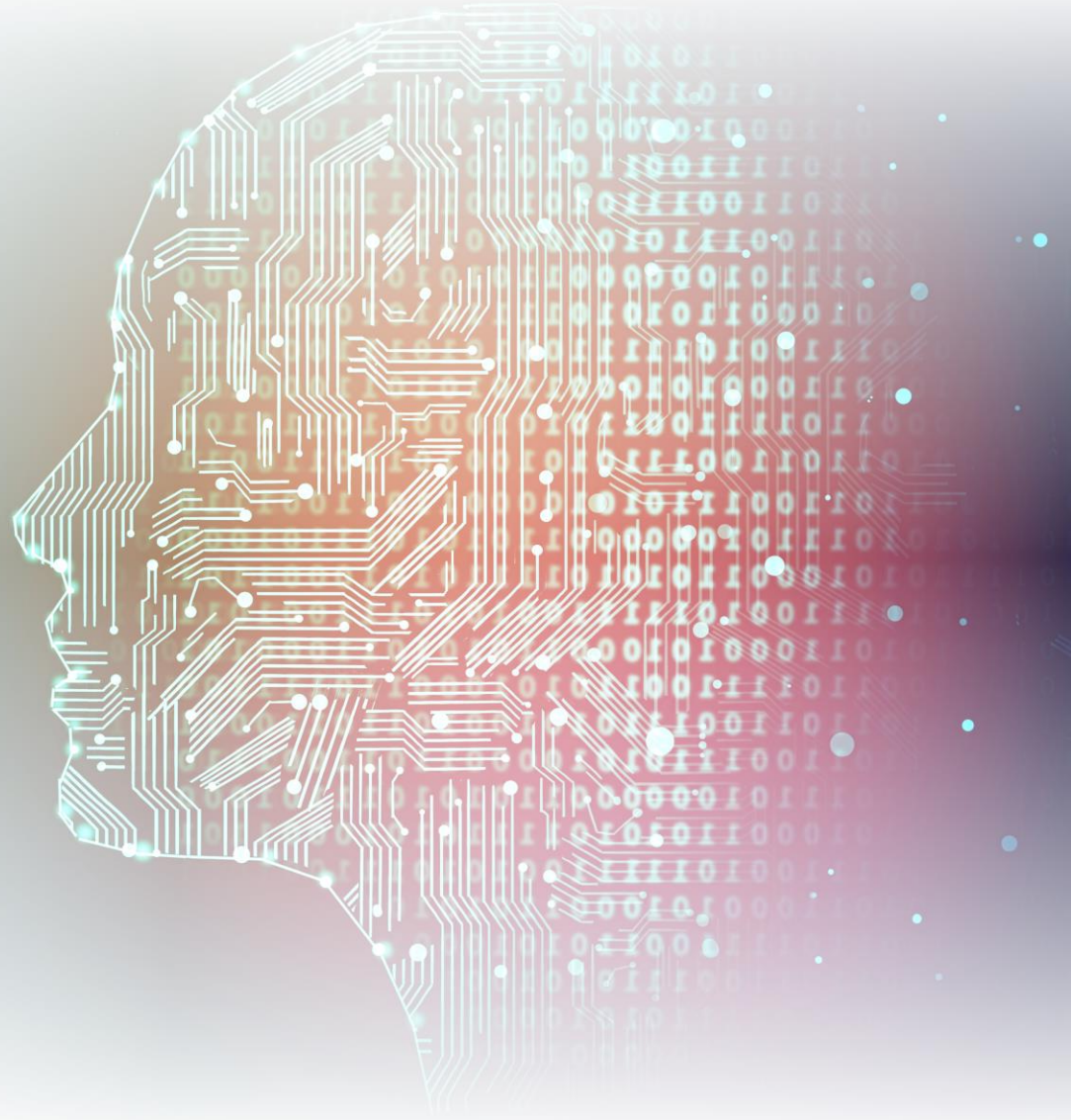
Test, Prototype and Debug on FPGA

Use FPGAs and SoCs for verification and prototyping:

- FPGA-in-the-Loop
- Prototyping and debugging
  - AXI Manager (synthesize, send, receive, and analyze data with MATLAB)
  - FPGA Data Capture (Capture data in real-time and analyze it then with MATLAB)

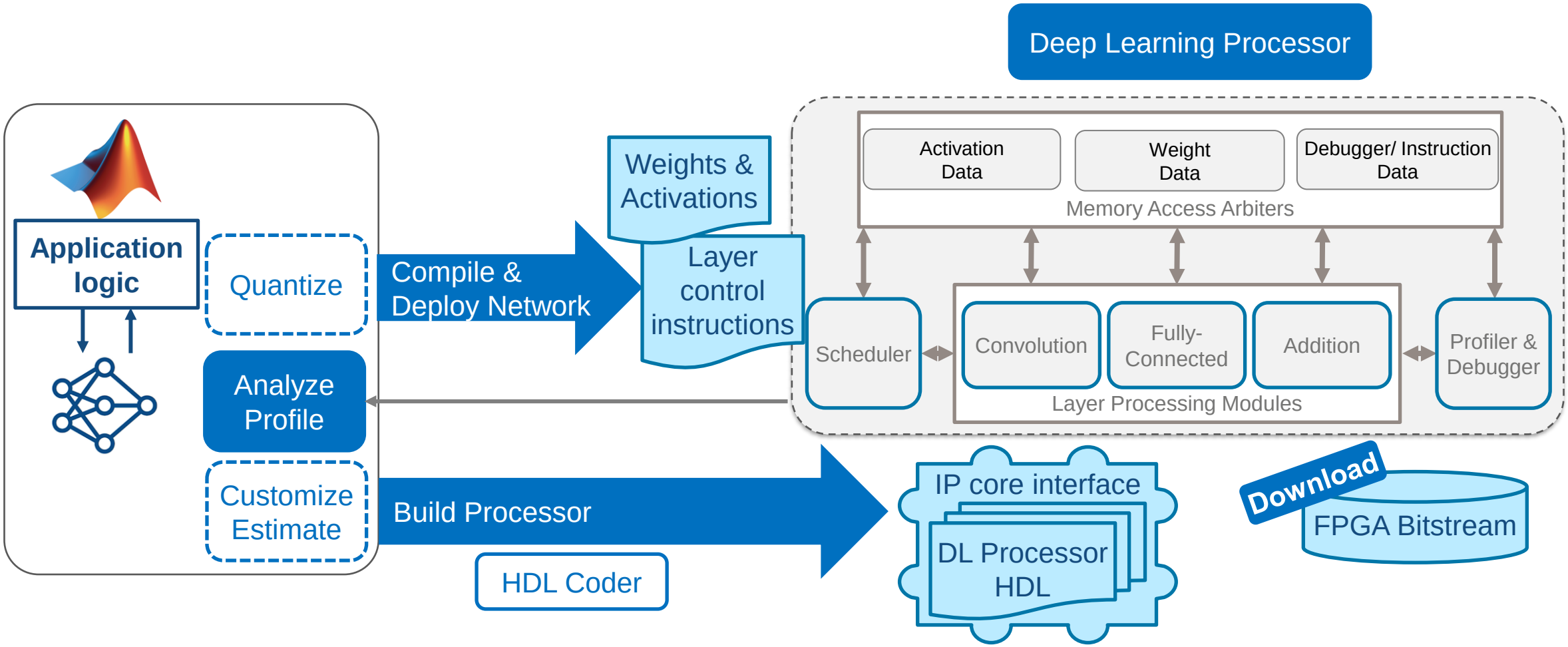


Deep Learning Network Implementation Options



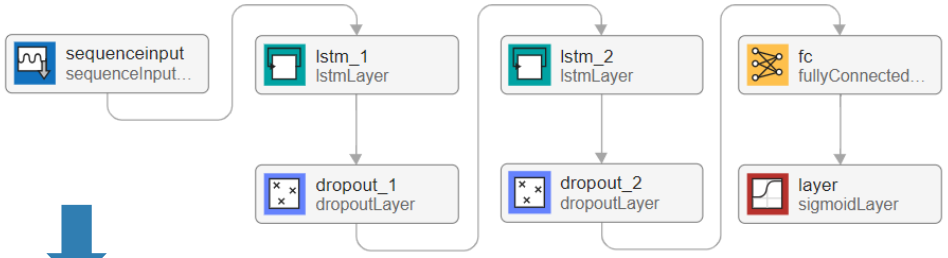
What about Deep Learning Networks?

They can also be implemented using Deep Learning HDL Toolbox.

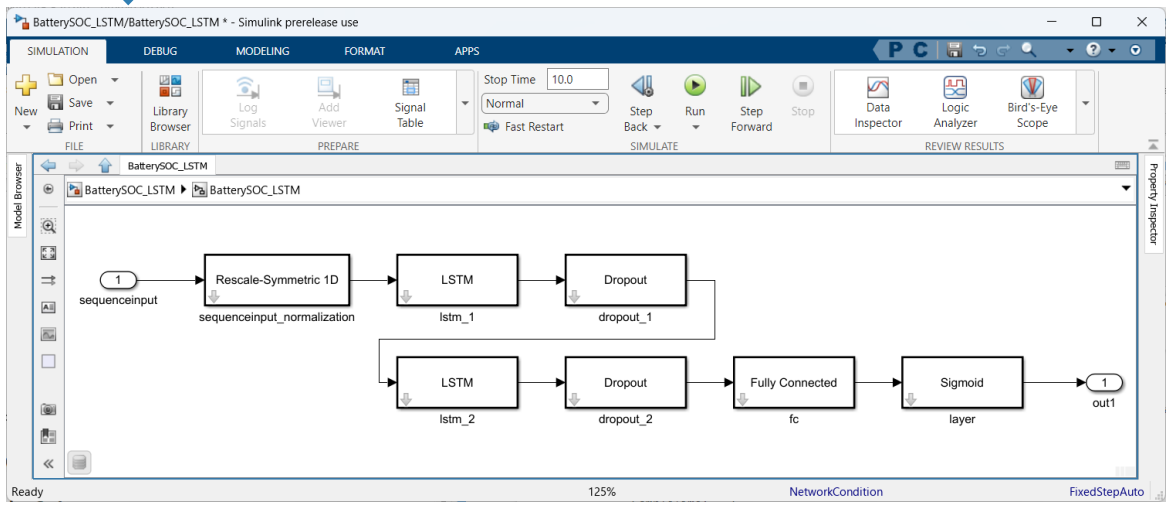


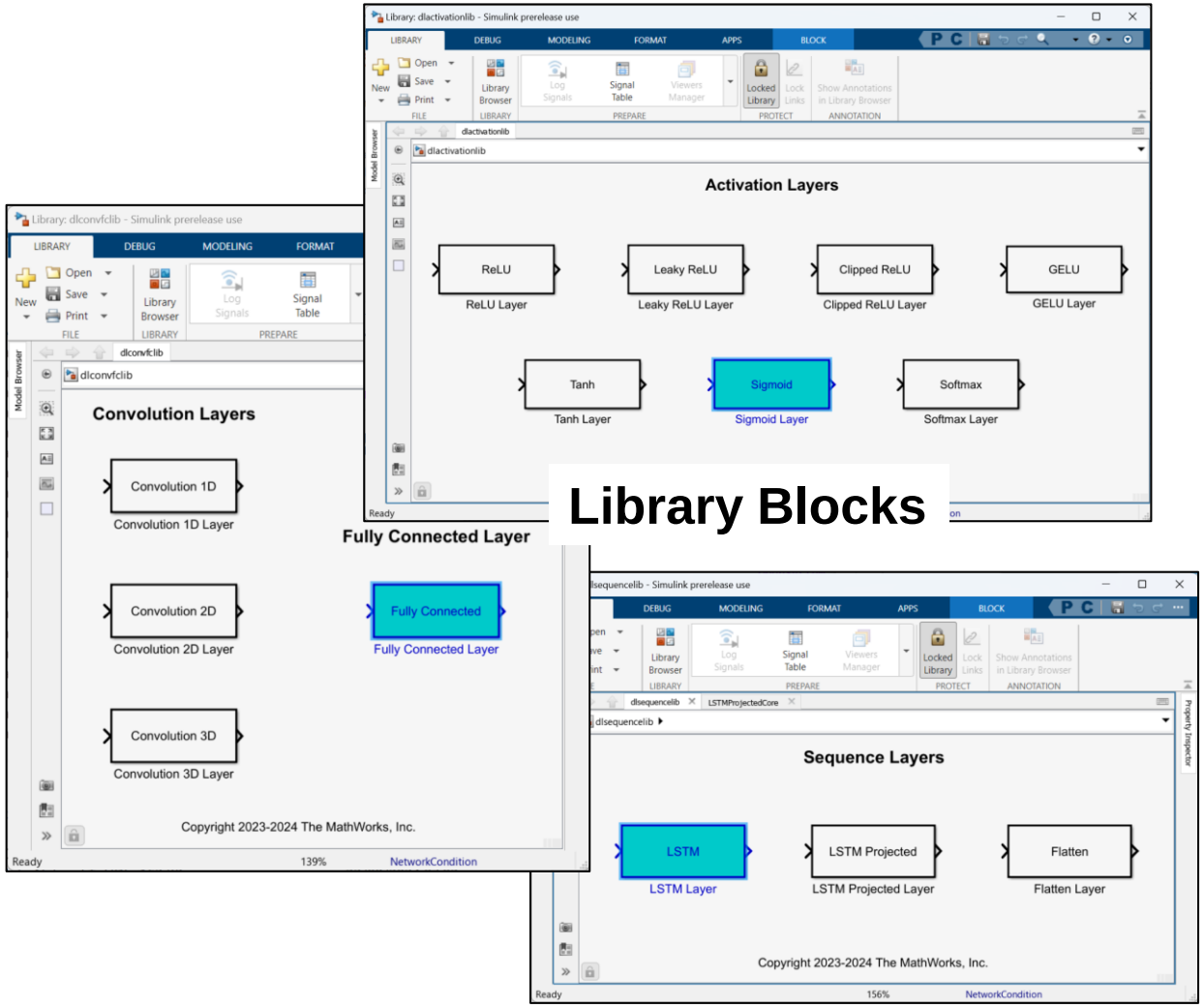
Deep Learning Layer Blocks R2024b

You now can export Deep Learning Networks (also shallow ones) to Simulink.




```
mdlInfo = exportNetworkToSimulink(net, ...
                "ModelName", "BatterySOC_LSTM", ...
                "SaveNetworkInModelWorkspace", true, ...
                "OpenSystem", true, ...
                "InputDataType", "double");
```





Questions