SYNOPSYS®
Silicon to Software™

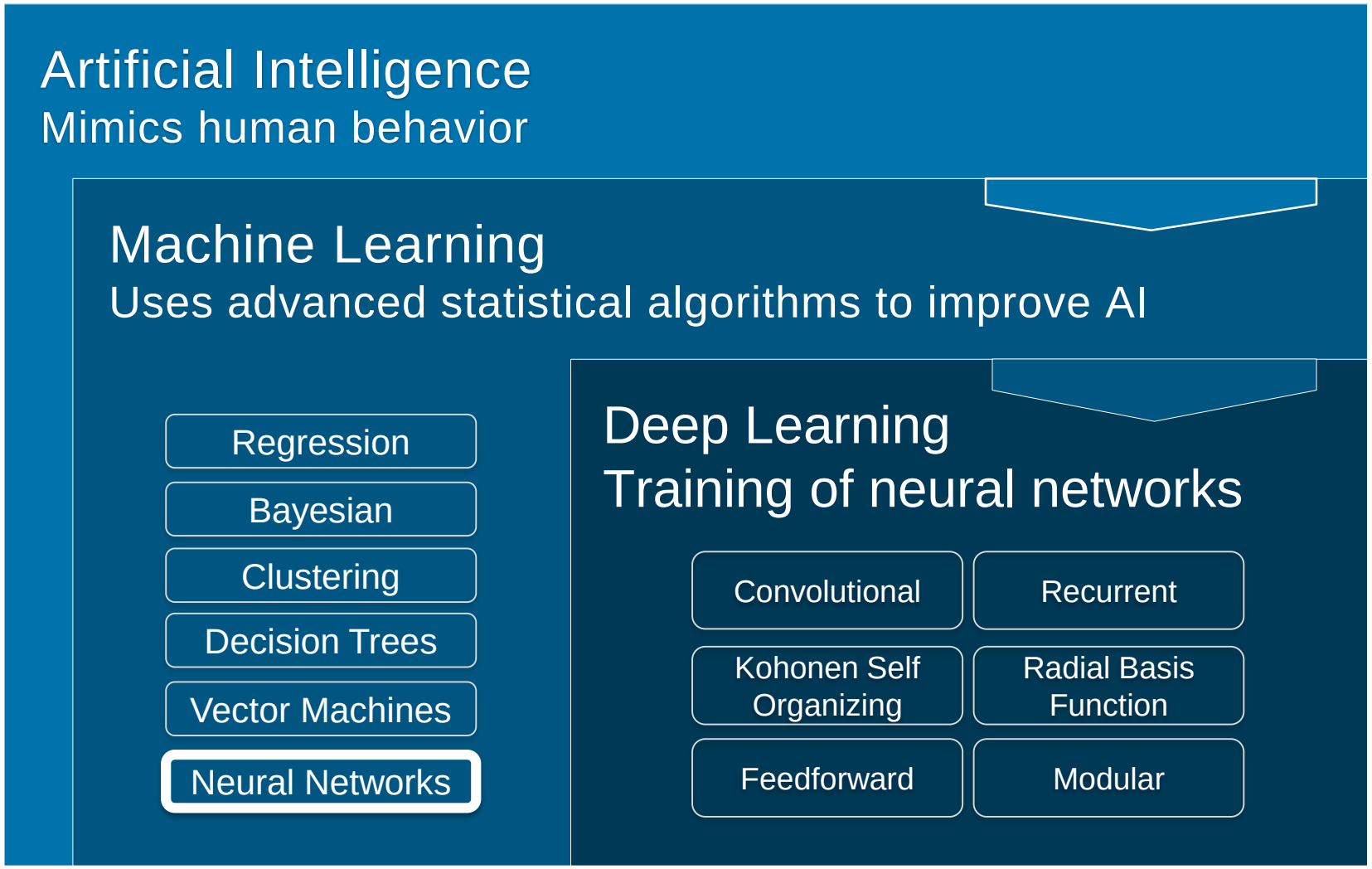
ARC Machine Learning Inference Software Library

April 2019

Agenda

- **Machine Learning for Always-on IoT**
- ARC Machine Learning Inference Software Library
- Example MLI Applications
- Low-power CNN Benchmark
- Conclusion

Defining Artificial Intelligence



- Artificial Intelligence mimics human behavior
- Machine learning uses advanced statistical models to find patterns & results
- Deep learning is a specialized subset of machine learning using neural networks data to recognize patterns

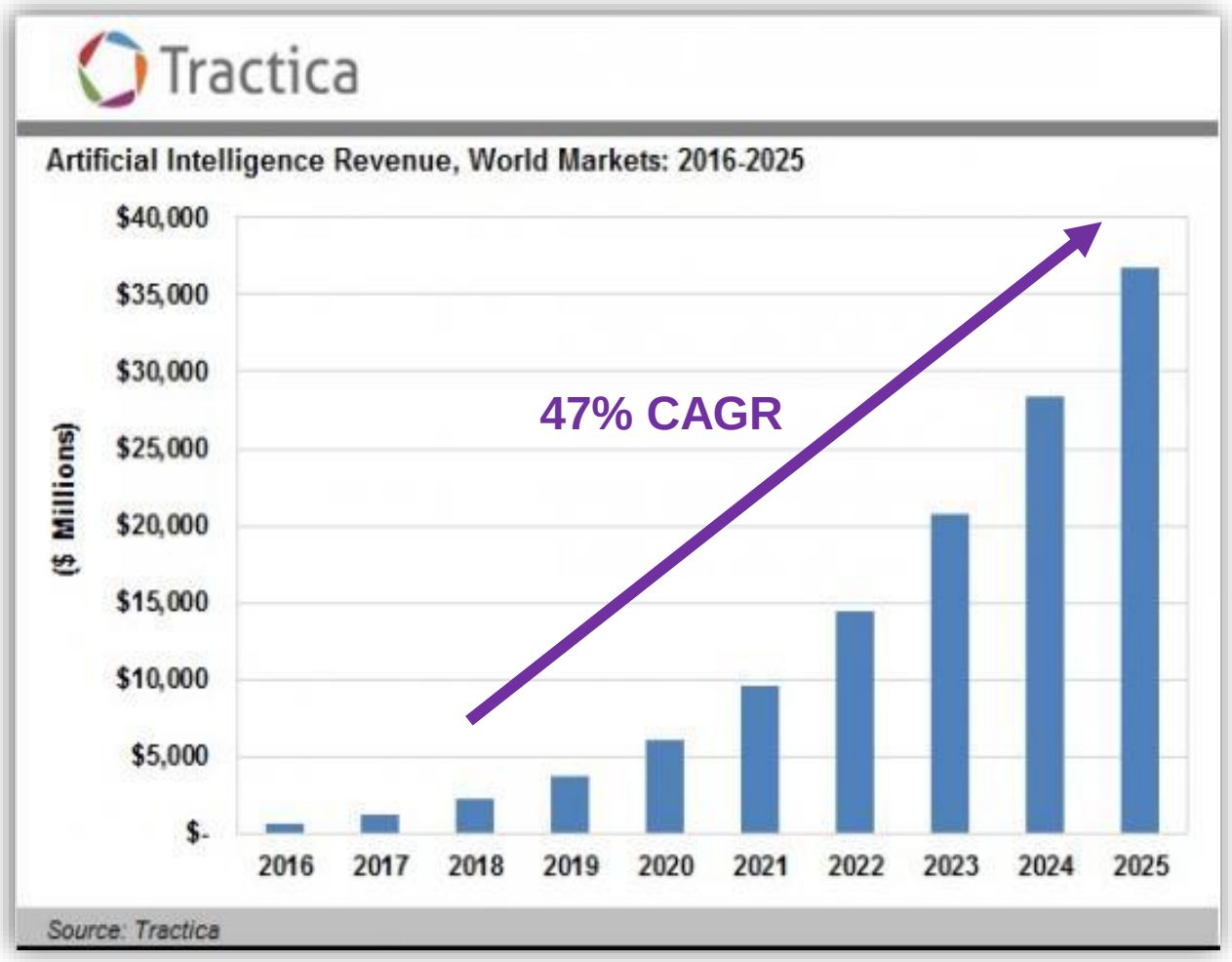
Artificial Intelligence Applications Exploding

Rapid Adoption of Human/Machine Interfaces (HMI)

amazon

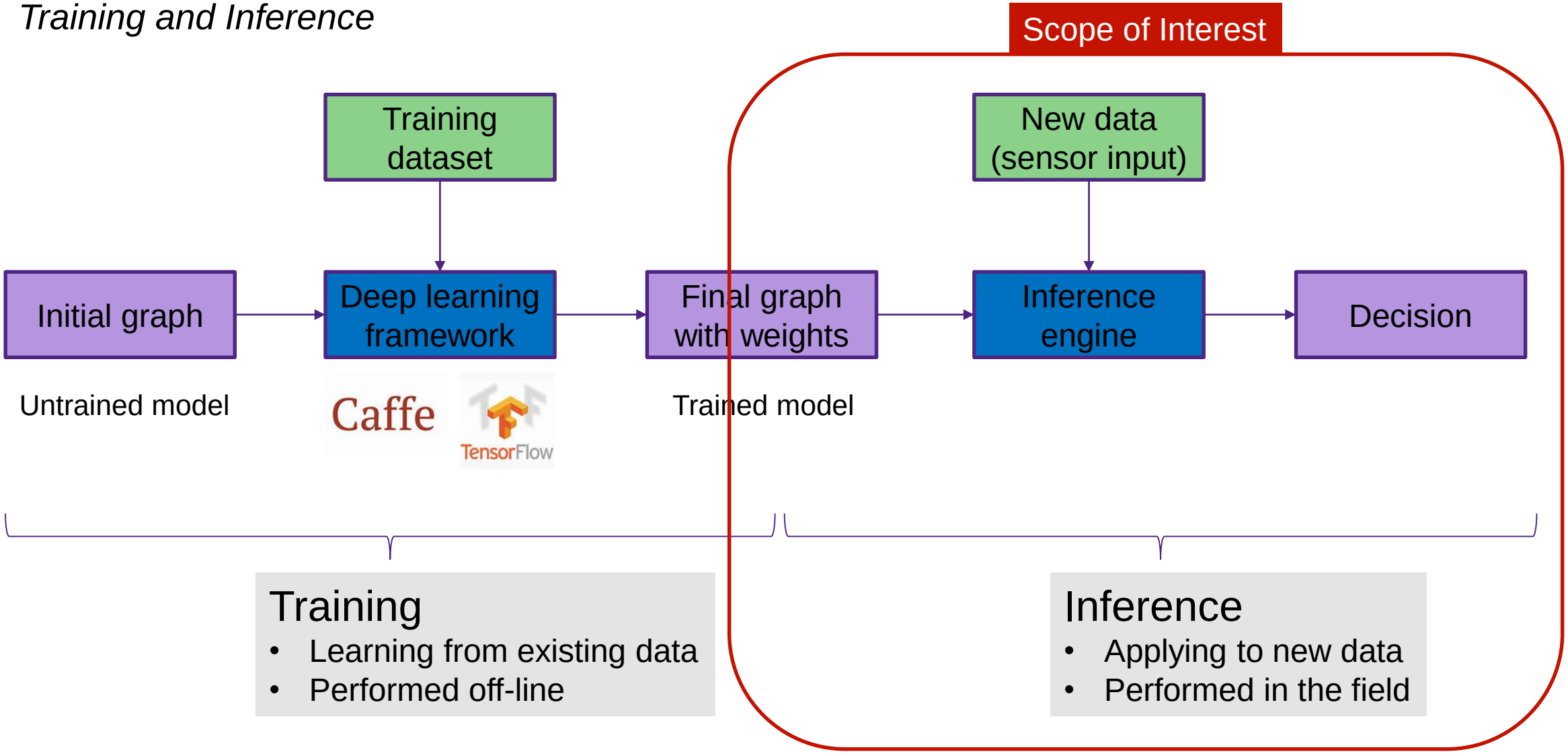
Google

Microsoft

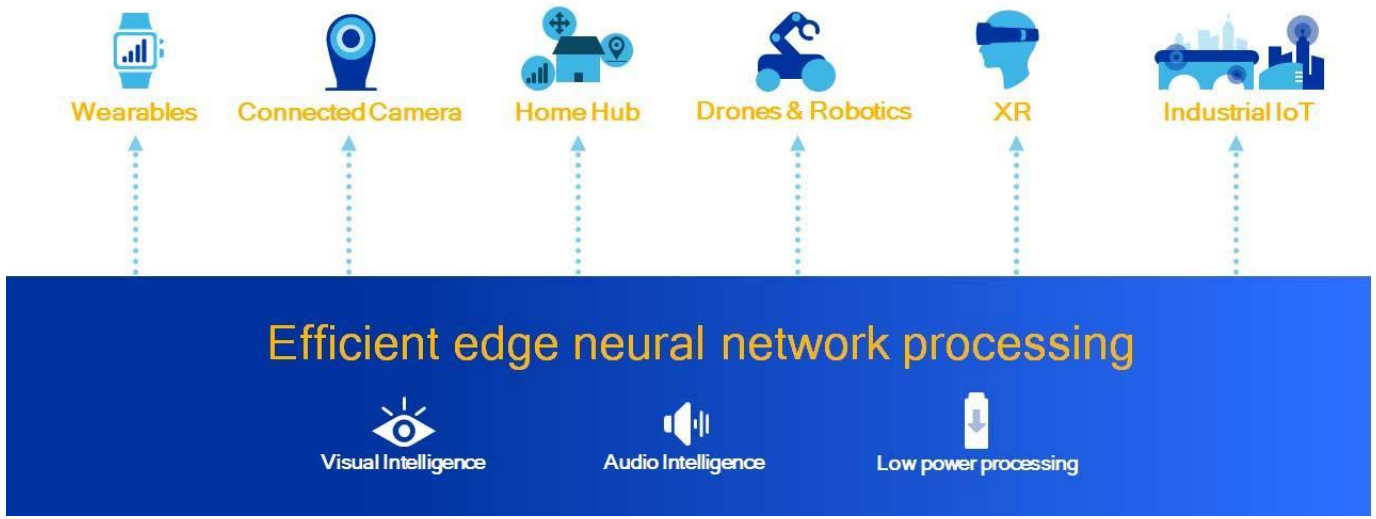


Machine Learning

Training and Inference



Machine Learning Use-cases for Edge Devices



ML Applications	Complexity (MOPS)
9D Sensor Hub	< 1
Voice Trigger	< 20
Face Trigger	<100
Human Activity Recognition	< 1.000
Natural HMI	< 2.000
Personal Fitness & Healthcare	< 2.000
Video Processing	> 10.000
Video Recognition	> 100.000

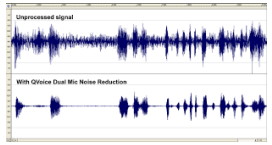
ARC EM / HS

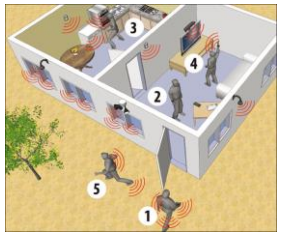
EV6x

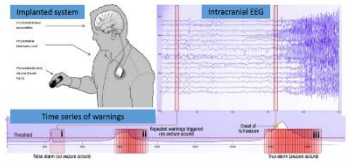
MOPS is mega operations per second.
Key operations (usually MACs) are counted

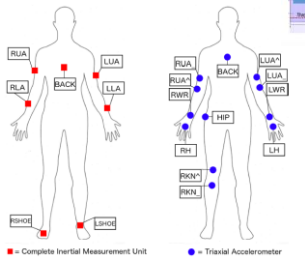
Target Market Segments

- **Natural HMI**
  - *noise reduction*
  - *speech recognition*
  - *acoustic awareness*
- **Personal Fitness & Healthcare**
  - *activity recognition*
  - *context awareness*
  - *early disease prediction*
  - *health monitoring*
- **Industrial IoT**
  - *multisensory data fusion*
  - *behavior prediction*
  - *acoustical fault detection*









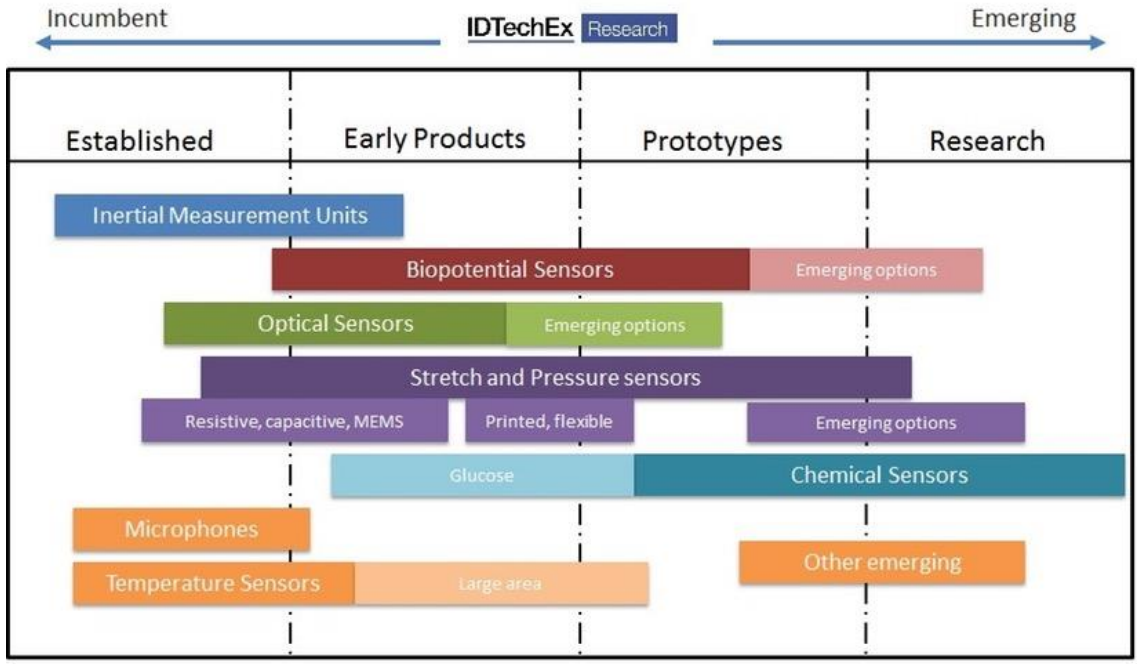
Always-on IoT

Typical Application Domain for ARC EM

- **Always-on IoT**
  - Continuous monitoring of sensor inputs
  - Microphones, cameras or other sensors
- **Example always-on functions**
  - Always-on voice command – “always listening”
    - *Smart speakers, smartphones, smart watches, headsets, automotive*
  - Visual object detection – “always watching”
    - *Face trigger, gesture recognition, …*
  - Health & fitness monitoring
    - *Monitor heart rate, activity patterns, …*
- **Low power consumption is key requirement**
  - Specifically for battery-operated devices
  - Low-power control & DSP for continuous monitoring

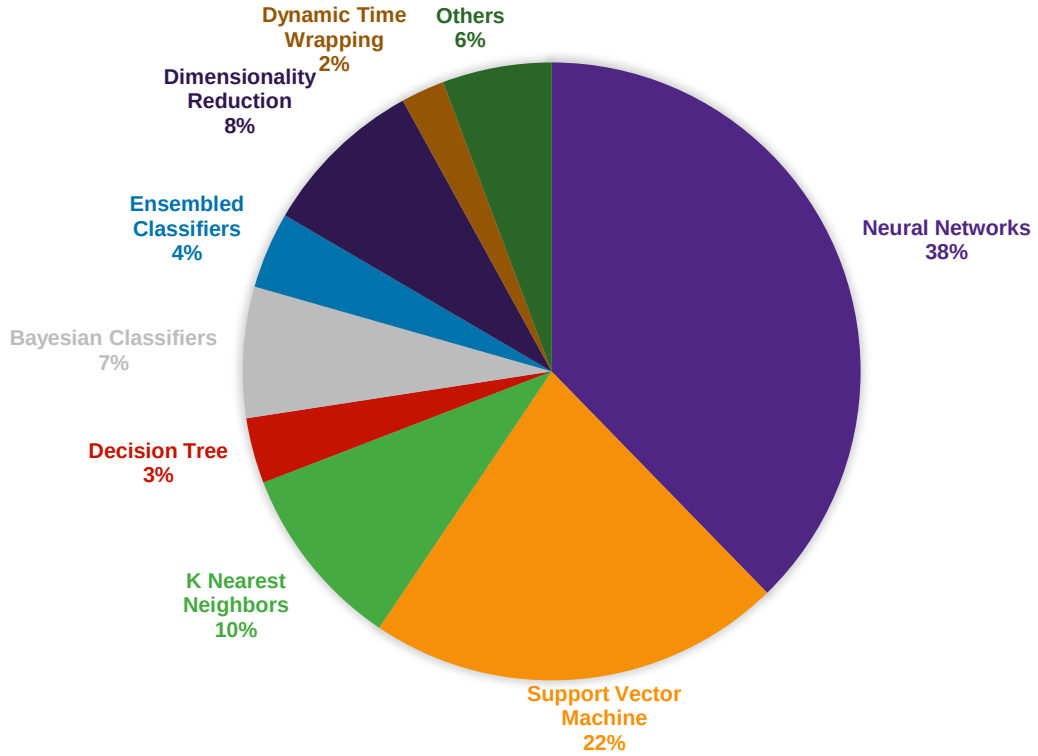
Machine Learning in IoT

Different Sensor Types Require Different Processing Methods



For more information, see the IDTechEx Research report:
Wearable Sensors 2015-2025: Market Forecasts, Technologies, Players
(www.IDTechEx.com/WTSensors)

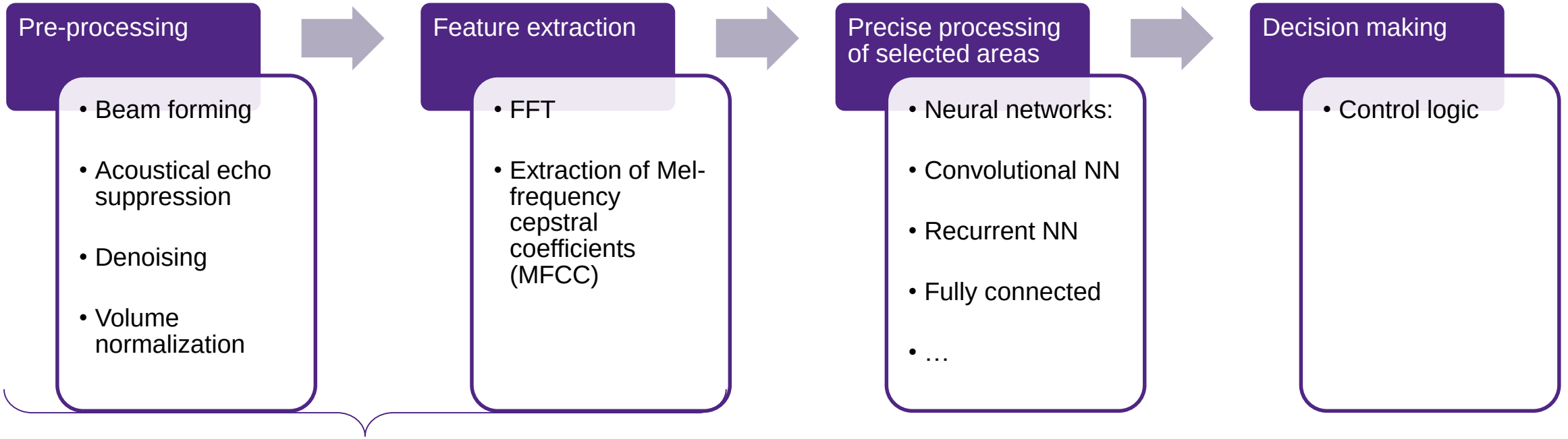
DISTRIBUTION OF MACHINE LEARNING ALGORITHMS FOR ANALYSIS OF WEARABLE SENSORS DATA



SOURCE: over 100 conference publications at IEEEXplore in 2016

ML Design Challenges

On the Example of Voice-based Human / Machine Interface



DSP intensive processing, usually with simple data level parallelism

Matrix operations, more difficult data level parallelism

RISC intensive code

ML applications present unique combination of RISC and DSP processing requirements

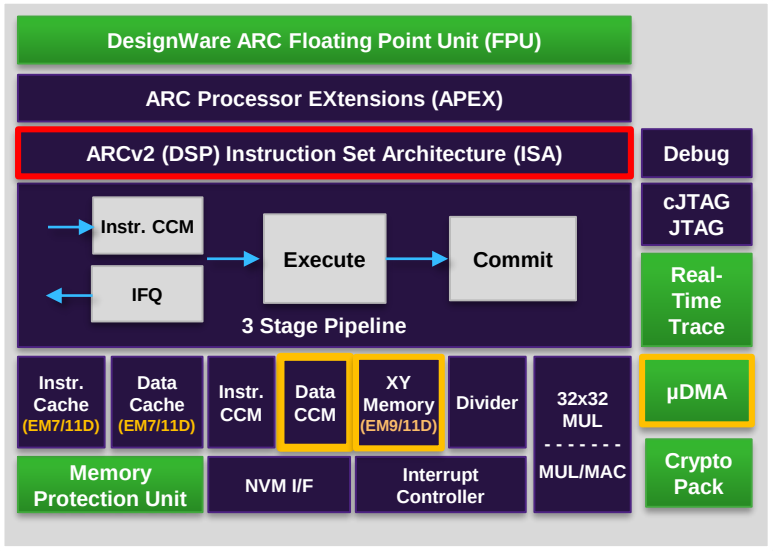
Agenda

- Machine Learning for Always-on IoT
- **ARC Machine Learning Inference Software Library**
- Example MLI Applications
- Low-power CNN Benchmark
- Conclusion

DesignWare ARC DSP cores for IoT

ARC EM/HS Cores Support Machine Learning Inference

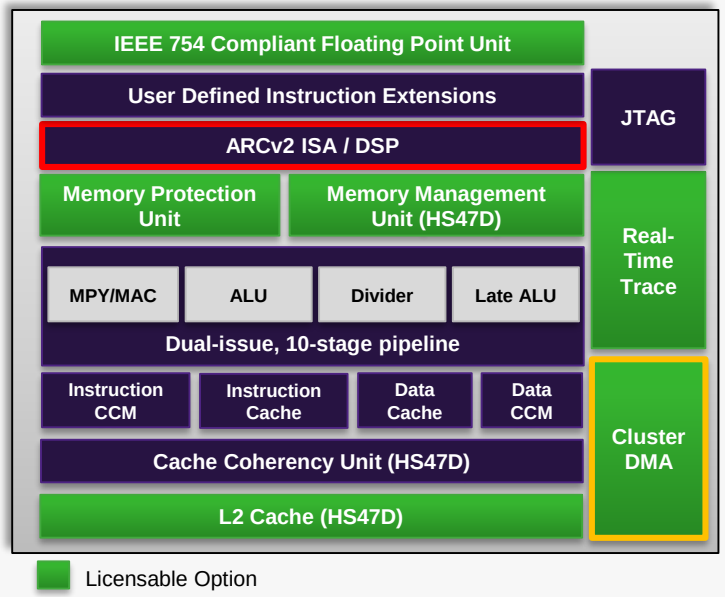
ARC EM DSP Processors



EM5D	Up to 2MB I & D CCMs, DSP
EM7D	Up to 2MB I & D CCMs, I & D Caches (up to 32K), DSP
EM9D	Up to 2MB I & D CCMs, DSP, XY Memory
EM11D	Up to 2MB I & D CCMs, I & D Caches (up to 32K), DSP, XY Memory

Ultra low-power

ARC HS DSP Processors



ARC HS45D	Dual-issue, 16M I & D CCMs, DSP
ARC HS47D	Dual issue, I & D CCMs, I & D caches (up to 64K), DSP

Higher performance

- Efficient control and DSP
- Dot-product style 8x16 / 16x16 dual MAC / quad MAC instructions
- Programmable Address Generation Units (AGUs)
- Zero latency XY memory architecture (EM DSP)
- Unaligned memory access support for efficient random memory access
- DMA controller for data block transfers
- Optimized NN software library for Machine Learning inference

embARC MLI S/W Library v.1.0

Optimized for ARC EMxD and HS4xD

Group	Functions	Short Description
Main operations	• 2D convolution • Depth wise 2D convolution • LSTM • Simple RNN • Fully connected	Convolve input features with a set of trained weights
Pooling	• Average pooling • Max pooling	Pool input features with a function
Transform / activation functions	• ReLU • SoftMax • Leaky ReLU • Sigmoid • TanH	Transform each element of input set according to a particular function
Data routing operations	• Padding • Transpose • Concatenation	Move input data by a specified pattern
Elementwise operations	• Vector arithmetic operations	Apply multi operand function elementwise to several inputs

- **Software library targeted to ML inference on ARC EM & HS DSP cores**
  - Library of kernel functions for effective inference of small to mid-sized machine learning models
- **Supports easy implementation of layered NN graph topologies**
  - Library of optimized for implementing multiple NN layer types
  - High efficiency & small footprint kernel implementations
  - C style APIs
- **Use model:**
  - Short term: user callable (manual graph mapping)
  - Longer term: automated graph mapping (Caffe, TensorFlow Lite, ..)
- **Open Source on embARC.org**

Comparison of embARC MLI to Arm CMSIS NN

- embARC MLI software library covers convolutional networks as Arm CMSIS NN including:
  - Convolutions, fully connected, poolings, depth wise separable convolutions layers
  - ReLU, sigmoid, tanh, softmax activations
  - Tensor padding, transpose, concatenate

- embARC MLI adds recurrent networks support, not available in Arm CMSIS NN including:
  - LSTM cells and simple RNN cells
  - More specific RNN architectures can be constructed using Data Manipulation and Elementwise operations.

- Recurrent networks are widely used for audio/voice processing tasks like
  - Voice recognition
  - Acoustical context awareness

- embARC MLI on ARC EM/HS DSP has better performance than Arm CMSIS NN on CM4/7
  - See benchmark below

embARC MLI: Code example

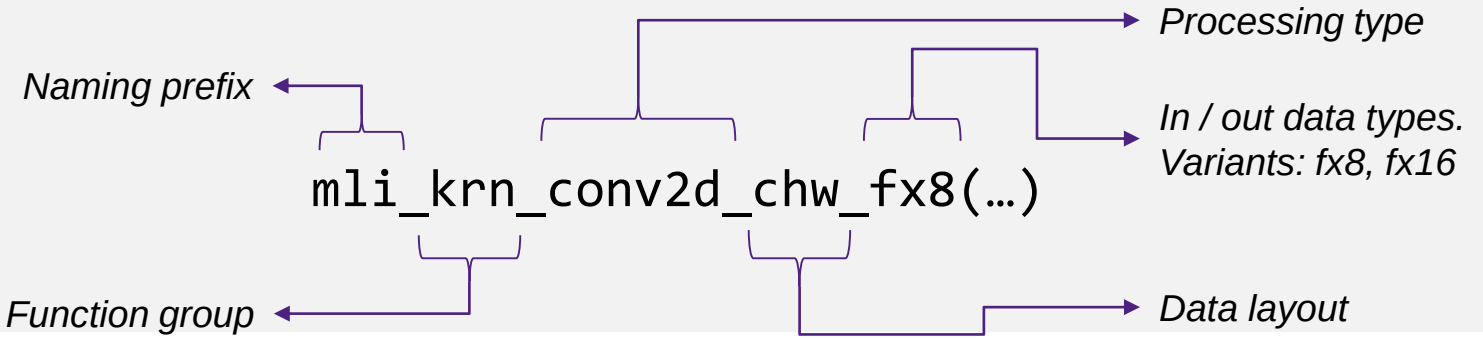
```
/* Data (tensors) initialization and layers parameters configuration */
user_init_tensors_and_params();

/* The first convolutional layer and pooling */
mli_krn_conv2d_chw_fx8(&in_tensor, &conv1_w_tensor, &conv1_b_tensor, &conv1_params, &aux_tensor_1);
mli_krn_relu_fx8(&aux_tensor_1, &relu_params, &aux_tensor_2);
mli_krn_maxpool_chw_fx8(&aux_tensor_2, &pool1_params, &aux_tensor_1);

/* The second convolutional layer and pooling */
mli_krn_conv2d_chw_fx8(&aux_tensor_1, &conv2_w_tensor, &conv2_b_tensor, &conv2_params, &aux_tensor_2);
mli_krn_avepool_chw_fx8(&aux_tensor_2, &pool2_params, &aux_tensor_1);

/* other intermediate layers */
...
/* Fully connected layer */
mli_krn_fully_connected_fx8(&aux_tensor_2, &fc_w_tensor, &fc_b_tensor, &fc_params, &aux_tensor_1);
mli_krn_softmax_fx8(&aux_tensor_1, &softmax_params, &aux_tensor_2);
```

Data and related parameters are passed as tensor structures.
This simplifies interface, splits initialization and runtime operations and improves code readability.



Agenda

- Machine Learning for Always-on IoT
- ARC Machine Learning Inference Software Library
- **Example MLI Applications**
- Low-power CNN Benchmark
- Conclusion

embARC MLI Example Applications

Code samples provided with version 1.0

CIFAR-10: Low Resolution Object Detection

Source: One of Caffe examples, open source

Face Detection

Source: CNN graph used in ARC always on IoT demo

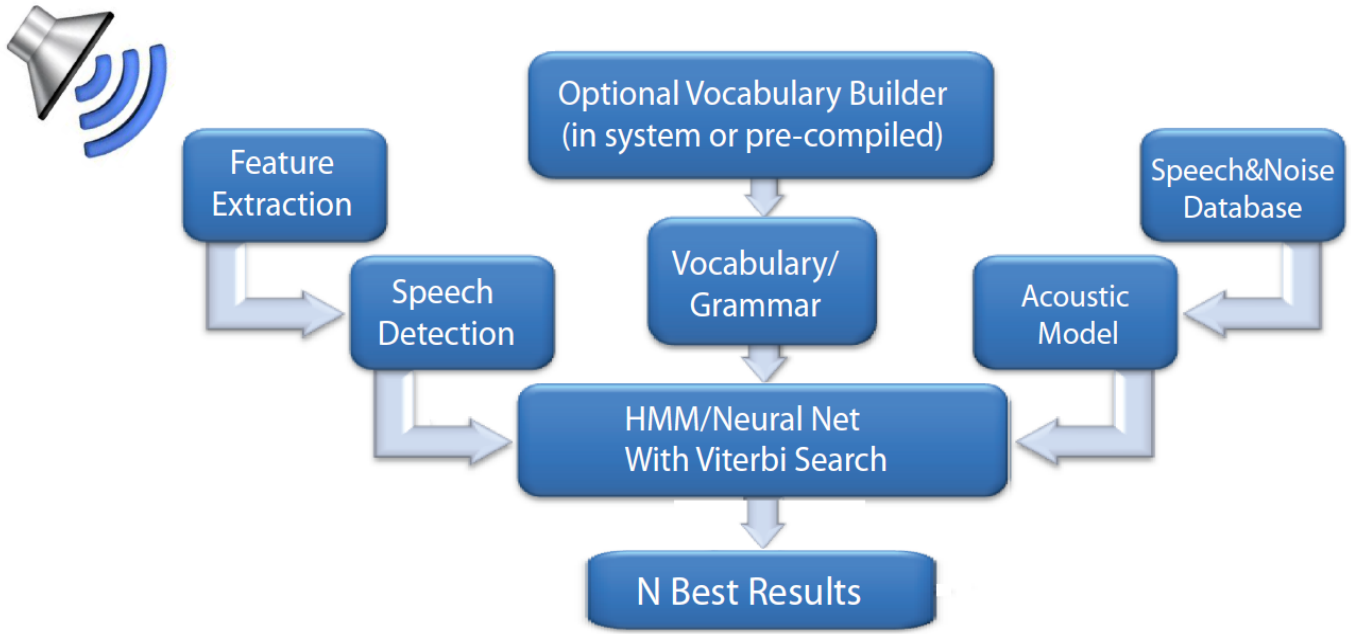
Key Phrase Detector

Source: TensorFlow example based on Google Speech Commands dataset, open source

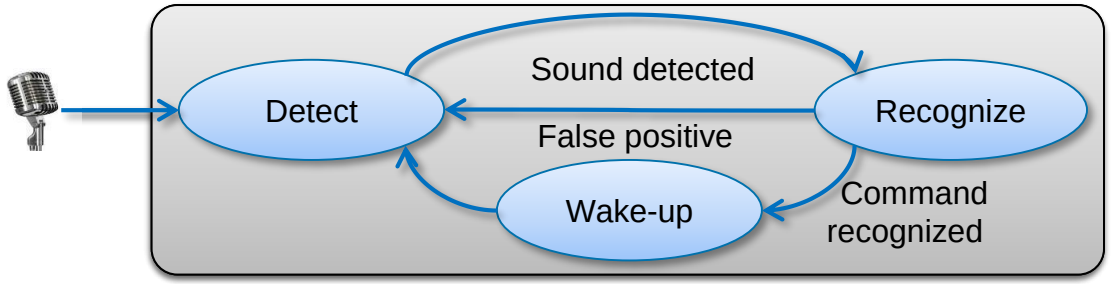
Human Activity Recognition

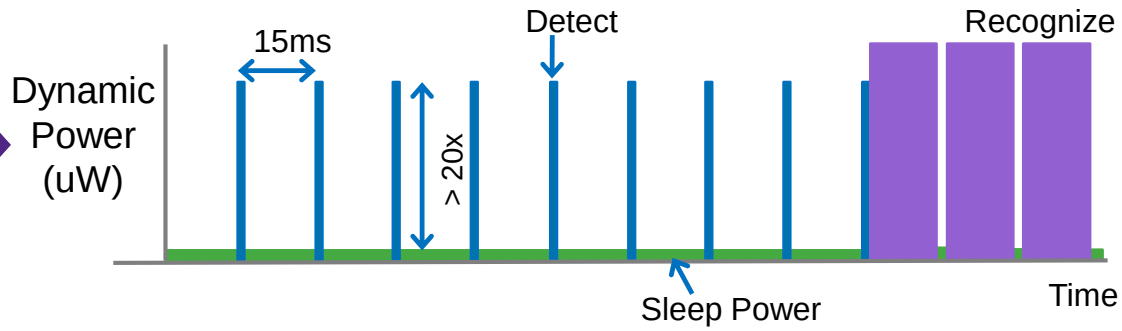
Source: Open source

Voice Activation / Voice Control



Dataset	Custom Audio Signals
Input data	16 kHz, mono audio
Output classes	12
Model size	25k coefficients
Computational complexity on ARC EM DSP	8 MCPS



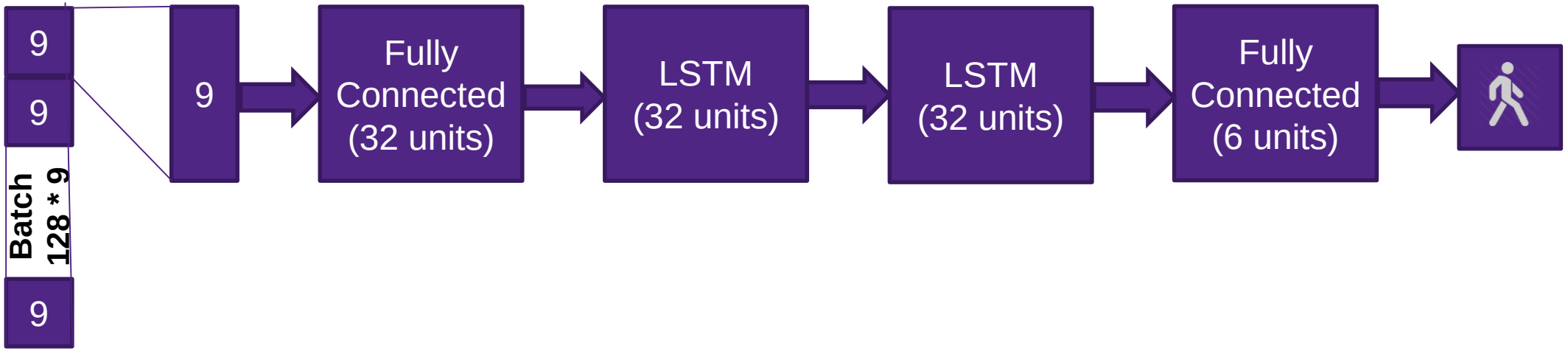


embARC MLI comes with open source version of keyword detection algorithm

9D Human Activity Recognition (HAR)

LSTM-based Classifier

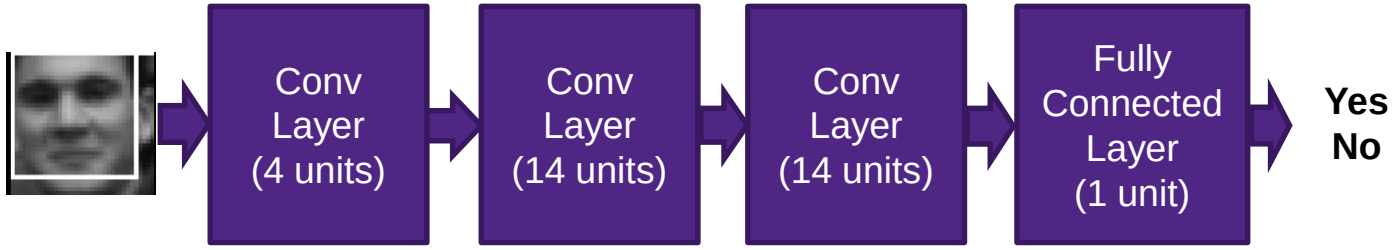
Dataset	UCI HAR Smartphone
Input data	9D (gyr+acc+mag) sensor data sampled at 50 Hz
Output classes	6
Model size	17k coefficients
Computational complexity on ARC EM DSP	6 MCPS



Low Power Face Detection Activation

CNN-based Classifier

Dataset	Custom set of faces
Input data	Grayscale images, 36x36 pixels
Image Pyramid Size	472 detections for 160x120 input frame, 2 fps
Number of Classes	2 (Face / Not Face)
Model Parameters	1k
Computational complexity on ARC EM DSP	38 MCPS

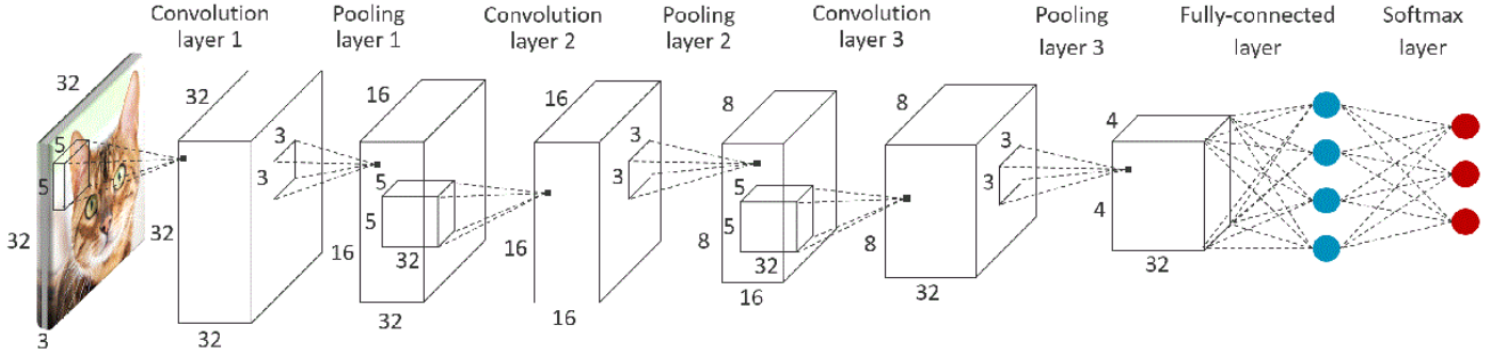


Agenda

- Machine Learning for Always-on IoT
- ARC Machine Learning Inference Software Library
- Example MLI Applications
- **Low-power CNN Benchmark**
- Conclusion

CIFAR-10 CNN Model on ARC EM

XY Based EM9D Provides 3-4x Improvement vs. Cortex-M4 & M7



Smaller Model	Layer 1: Convolution 1	Layer 3: Convolution 2	Layer 5: Convolution 3	Layer 7: Fully Connected	Pooling Layers	Total (Mcycles)
Arm CM4	7.87	6.37	1.80	0.56		***16.60***
ARC EM9D	1.74	1.77	0.53	0.30		***4.34***

Larger Model	Layer 1: Convolution 1	Layer 3: Convolution 2	Layer 5: Convolution 3	Layer 7: Fully Connected	Pooling Layers	Total (Mcycles)
Arm CM7	6.78	9.24	4.88	0.50		***21.40***
ARC EM9D	1.74	3.54	2.13	0.36		***7.77***

3-4x cycle improvement enables designs to run at much lower frequencies; resulting in significant power savings

CIFAR-10 Benchmark Model

- Low resolution object detection for dataset of 32x32 images
- Object classification for 10 classes of objects (cars, birds, cats, dogs, etc)
- 3 convolution layers; 1 fully-connected layer; 3 pooling layers

Benchmark results benefit from EM9D architectural features

- XY memory architecture with address generation units
- dual 16-bit MAC capability (achieves >1 MAC/cycle)

Benchmark results benefit from efficient software library

- embARC Machine Learning Inference Library (MLI) vs. Arm CMSIS NN

Agenda

- Machine Learning for Always-on IoT
- ARC Machine Learning Inference Software Library
- Example MLI Applications
- Low-power CNN Benchmark
- **Conclusion**

Why use ARC for MLI-based applications?

- **ARC DSP is extremely efficient for always-on applications**
  - Efficient control and DSP → single core for feature extraction and ML inference
  - Dot-product style dual/quad MAC → DSP ISA well-suited for Machine Learning inference
  - Lower 8-bit vector unpacking cost thanks to 8x16 MAC
  - Free 8-bit vector unpacking with AGU → vector load, unpack, sign extend, dual MAC in one cycle
  - Integrated μDMA for off-loaded concurrent data transfers

- **ARC EM9D beats competition**
  - Arm CM4 needs ~4x more cycles (CIFAR-10)
  - Arm CM7 needs ~3x more cycles (CIFAR-10)
  - Arm A55 + NEON NN needs ~2x more cycles (dot product kernel performance)

- **High efficiency of ARC EM9D reduces frequency requirements**
  - ARC EM9D can be synthesized for and run at 3-4x lower frequency than competition
  - Yielding significant power savings

- **embARC MLI is a library of highly optimized NN kernels**
  - Covers convolutional as well as recurrent networks
  - Freely available at embarc.org

SYNOPSYS®
Silicon to Software™

Thank You

Backup

- Quick Start Guide
- CIFAR-10 Benchmark Details
- Model Deployment Example

Tools Requirements / GNU Compatibility

Component	MWDT Toolchain	GNU Toolchain
embARC MLI Library	v.2018.12 or higher	Not compatible
embARC MLI Examples	v.2018.12 or higher	v.2018.09 or higher Prebuilt version of embARC MLI library is required

embARC MLI Documentation

- synopsys.com product landing page:
  https://www.synopsys.com/dw/ipdir.php?ds=machine-learning-inference-library
- embARC.org landing page:
  https://embarc.org/dl.html
- Readme and getting started manual:
  https://github.com/foss-for-synopsys-dwc-arc-processors/embarc_mli/
- embARC MLI full documentation:
  https://embarc.org/embarc_mli/

Getting and Building the Library

- embARC MLI Library is available from embARC: https://embarc.org/dl.html

- To get and build library you shall do following steps:
  - Open command line in your working directory
  - Clone repository *git clone https://github.com/foss-for-synopsys-dwc-arc-processors/embarc_mli.git*
  - change working directory to *./lib/make/*
  - start building *gmake TCF_FILE=../../hw/em9d.tcf*
  - Binary of library is created at *./bin*

Need Further MLI Support?

- Have active ARC core or MetaWare Development Toolkit maintenance contract?
  - Use SolvNet and file a CASE / STAR with you question
    - L1: DesignWare Cores ARC Tools
    - L2: embARC
    - L3: Machine Learning Inference Library

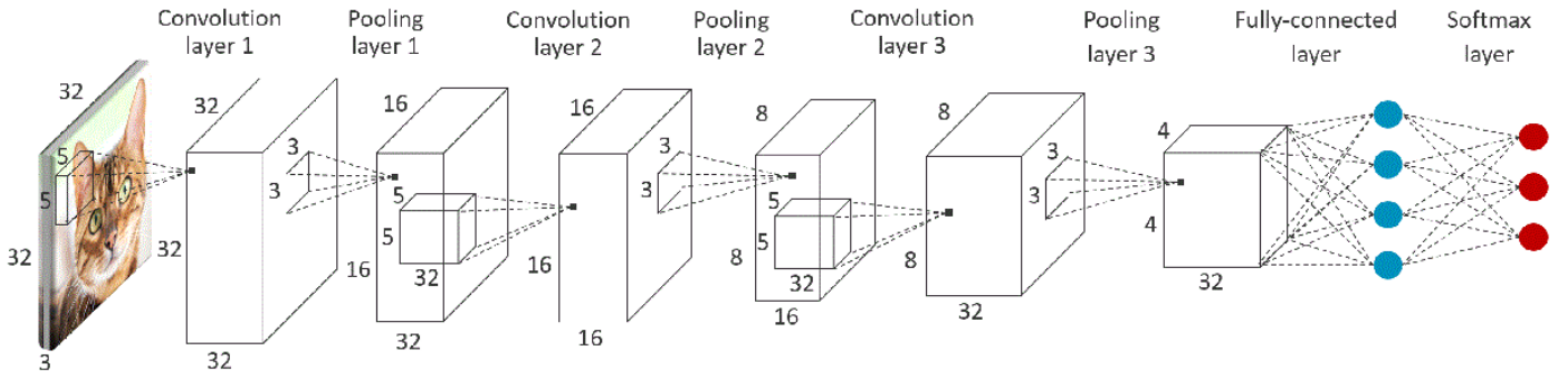
- Otherwise
  - Use MLI Github Issues system here
    https://github.com/foss-for-synopsys-dwc-arc-processors/embarc_mli/issues

Backup

- Quick Start Guide
- CIFAR-10 Benchmark Details
- Model Deployment Example

CIFAR-10 Model

Detailed Model for Arm CM4 Comparison



CIFAR-10 model

- (3) convolution layers
- (1) fully-connected layer
- (3) pooling layers

Layer	Filter shape	Output shape	MOPs
Layer 1: Conv1	5x5x3x32	32x32x32	2.4576 MMACs
Layer 2: pool	3x3	16x16x32	0.0737 MOPs
Layer 3: Conv2	5x5x32x16	16x16x16	3.2768 MMACs
Layer 4: pool	3x3	8x8x16	0.009 MOPs
Layer 5: Conv3	5x5x16x32	8x8x32	0.8192 MMACs
Layer 6: pool	3x3	4x4x32	0.0046 MOPs
Layer 7: FC	512x10	10	0.00512 MMACs

CIFAR-10 Benchmark Results for ARC EM9D

Details on Comparison to Arm Cortex-M4

- Efficient implementation of CIFAR-10 model on ARC EM9D
  - Benefits from XY memory architecture and address generation units (AGUs) of ARC EM9D
  - Benefits from dual 16-bit MAC capability; overall achieves >1 MAC/cycle
- Beats Arm CM4 by **~4x** in cycles
  - ARC EM9D can be synthesized for and run at much lower frequency → large power savings

Layer	Filter shape	Output shape	MOPs	ARC EM9D [Mcycles]	Arm CM4 [Mcycles]
Layer 1: Conv1	5x5x3x32	32x32x32	2.4576	1.74	7.87
Layer 3: Conv2	5x5x32x16	16x16x16	3.2768	1.77	6.37
Layer 5: Conv3	5x5x16x32	8x8x32	0.8192	0.53	1.80
Layer 7: FC	512x10	10	0.00512	0.30	0.56
Pooling layers			0.0876		
Total			6.6463	4.34	16.60

➡ **3.8x**

CIFAR-10 Benchmark Results for ARC EM9D

Details on Comparison to Arm Cortex-M7

- Benchmark results for larger CIFAR-10 model
  - Same graph structure with somewhat larger filter shapes and output shapes
- Used published results for Arm CM7 for CIFAR-10 benchmark on larger model
- Beats competitor by **~3x** in cycles

Layer	Filter shape	Output shape	MOPs	ARC EM9D [Mcycles]	Arm CM7 [Mcycles]
Layer 1: Conv1	5x5x3x32	32x32x32	2.4576	1.74	6.78
Layer 3: Conv2	5x5x32x32	16x16x32	6.5536	3.54	9.24
Layer 5: Conv3	5x5x32x64	8x8x64	3.2768	2.13	4.88
Layer 7: FC	1024x10	10	0.0102	0.36	0.02
Pooling layers			0.1014		0.48
Total			12.3996	7.77	21.40

➡ **2.8x**

Red parameter values in Filter shape and Output shape columns indicate differences with smaller model of CIFAR-10

Backup

- Quick Start Guide
- CIFAR-10 Benchmark Details
- Model Deployment Example

Background

Preferable skills

We assume familiarity with:

- **Neural Networks Basics** – words like convolutions, layers, tensors won't puzzle you
- **Python** – you can read and understand python code.
- **NumPy** – de-facto standard numerical library for python
- **C programming language** – MLI API is C level API
- **Caffe framework basics** – Considered example (CIFAR-10) is based on Caffe standard tutorial

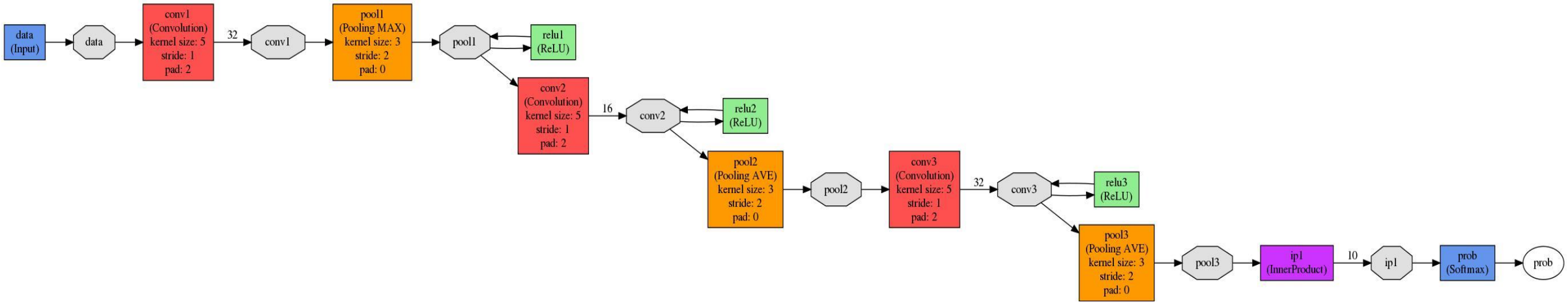
Manual Graph Mapping Process

Steps for get deployed model

1. Deploying Data
   - Get access to data in trained model
   - Define ranges of intermediate results
   - Define ranges of model coefficients
   - Quantization: Define data formats
   - Quantization: Extract Coefficients and prepare tensors
2. Deploying Operations
   - Translate operations into MLI Primitives

Caffe CIFAR-10 Example: Deploying Data

Starting point



- After we had trained the model, we have next main files required for deployment:
  - ***cifar10_small.prototxt*** – textual description of the graph
  - ***cifar10_small.caffemodel.h5*** – data of trained graph. Not necessary h5 format.
  - (Optional) ***mean file***. Not used in this example. The only mean value for all pixels in all channels was used: 128
- Easy way for getting access to model data (including intermediate results) is Caffe python API. Next modules will help us (should be installed in environment):

```
import caffe
import lmdb
import numpy as np
```

* Graph visualization is drawn by standard Caffe draw_net.py script (https://github.com/BVLC/caffe/blob/master/python/draw_net.py - requires dot module)

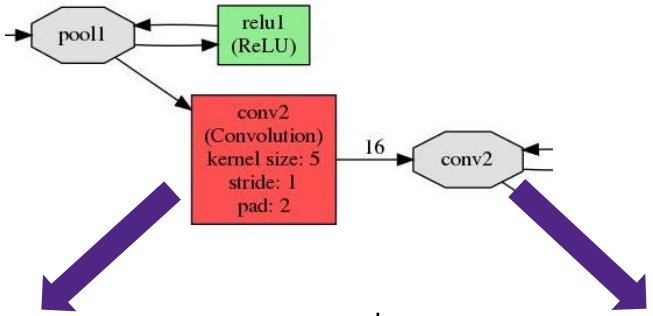
Caffe CIFAR-10 Example: Deploying Data

Access to Trained Model Data

Our goal is to define how to transform Caffe representation of data into numpy arrays we can easily work with. At first we need to load and initialize the model:

```
classifier = caffe.Net("cifar10_small.prototxt", "cifar10_small.caffemodel.h5", caffe.TEST)
out_key = classifier.outputs[0] # key to the first output of net (here it's a prob data)
```

We need to get two subsets of data



Model parameters

```
for layer_name, data_obj in classifier.params.items():
    weights_np_arr = data_obj[0].data # conv weights is blob #0
    bias_np_arr = data_obj[1].data    # conv bias is blob #1
    …
```

Intermediate results data

```
for blob_name, data_obj in classifier.blobs.items():
    data_np_arr = data_obj.data
    …
```


Caffe CIFAR-10 Example: Deploying Data

Define Ranges of Intermediate results

Quantization implies not only **weights quantization**, but also defining **ranges of all intermediate data** (layers input/output). For this purpose we need to run model on some representative data subset and gather statistic for all intermediate data (It's better to use the full dataset)

```
# Open dataset and get cursor
lmdb_env = lmdb.open("cifar10_train_lmdb")
lmdb_txn = lmdb_env.begin()
lmdb_cursor = lmdb_txn.cursor()

# Init data parser and dictionary for min/max statistic
datum = caffe.proto.caffe_pb2.Datum()
ir_ranges = dict()

for key, value in lmdb_cursor:
    datum.ParseFromString(value)
    data_raw = caffe.io.datum_to_array(datum)

    # Don't forget about pre-processing if you need it (Mean and scale)
    test_data = np.asarray( [(data_raw - 128.0)/128.0] )
    test_label = datum.label

    # Model Inference on loaded data
    pred = classifier.forward_all(data=test_data)[out_key]

    # Update ranges (Note: dictionary requires proper initialization in first pass)
    for blob_name, v in classifier.blobs.items():
        ir_ranges[blob_name][0] = max(ir_ranges[key][0], v.data.max())
        ir_ranges[blob_name][1] = min(ir_ranges[key][1], v.data.min())
 …
```

Caffe CIFAR-10 Example: Deploying Data

Define Ranges of Weights

As weights are fixed after training and don't change in inference time we may just transform data to numpy arrays, which provides *min()* and *max()* methods for easy range definition. It also keeps the shape of data we need for MLI tensor definition later

```
weights_dict = dict()
bias_dict = dict()
for layer_name, data_obj in classifier.params.items():
    weights_np_dict[layer_name] = data_obj[0].data
    bias_np_dict [layer_name] = data_obj[1].data
    …
```

The more tricky thing starts when batch normalization and scale layers with convolution are being used. Parameters of these layers can be integrated into weights and biases, and it's highly recommended to do so

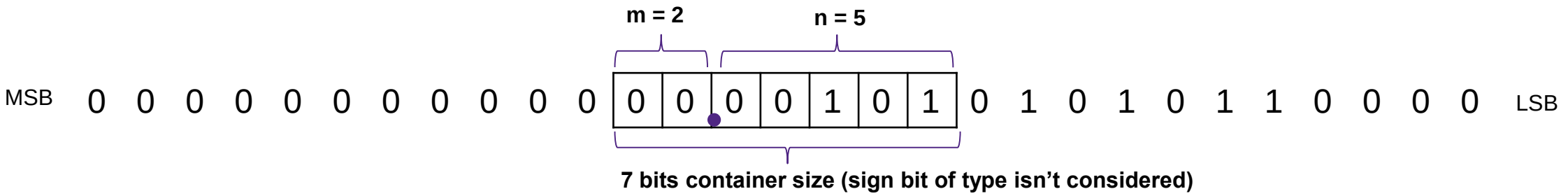
Caffe CIFAR-10 Example: Deploying Data

Quantization: Short Explanations on Data Format

MLI supports fixed point format regarding to Qm.n notation, where:

m – number of integer bits (or number of bits from the beginning of container to the fixed point)

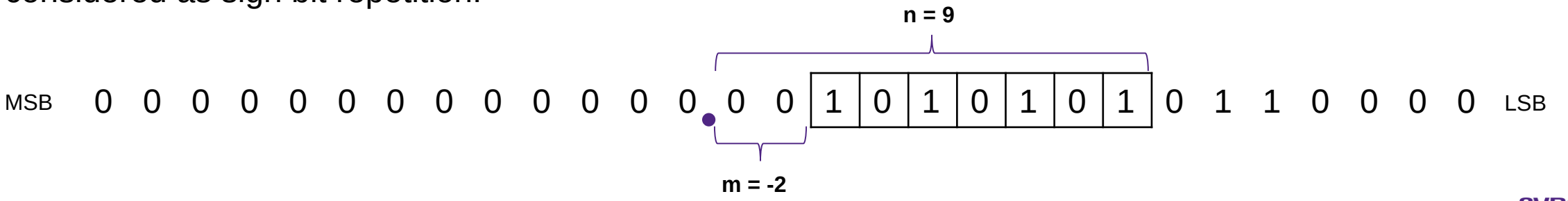
n – number of fractional bits (or number of bits from the fixed point to the end of container)



With known container size, its enough to specify the number of fractional bits. Integer bits can be derived from it:

$$\mathrm{m} = contsize - n$$

Number of fractional bits may exceed total number of bits in container. In this case all un-stored bits are considered as sign bit repetition.



Caffe CIFAR-10 Example: Deploying Data

Quantization: Define Data Format

We need to find the appropriate Q format of input, output and coefficients data for each layer to correctly represent float values. This format will be fixed at inference time. We will define number of integer bits because fractional bits can be easily derived from it.

CIFAR10-small	Maximum abs values of tensors				Minimum Integer bits requirements			
	Layer input Max ABS value	Layer weights Max ABS value	Layer Bias Max ABS val	Layer out Max ABS val	Layer input Integer bits	Layer weights Integer bits	Layer Bias Integer bits	Layer out Integer bits
Layer 1_conv	0.99	0.49	0.73	7.03	0	-1	0	3
Layer 2_conv	7.03	0.35	0.39	21.88	3	-1	-1	5
Layer 3_conv	17.89	0.29	0.18	27.22	5	-1	-2	5
Layer 4_fc	22.14	0.41	0.2	20.798	5	-1	-2	5

```
max_abs_val = max(abs(val_max), abs(val_min))
```

```
int_bits = int(np.ceil(np.log2(max_abs_val)))
```

While for 8bit depth data it's enough, for 16bit depth additional step is required (see tips and tricks).

Caffe CIFAR-10 Example: Deploying Data

Quantization: Extract Coefficients and Prepare Tensors

Now we have coefficients in numpy array objects, and defined Qm.n format for data.

Next we need to populate MLI structures for kernels and export the quantized data.

Define coefficients for the first convolution

a) You may make preprocessor quantize data for you at compile-time (in case there are not so many coefficients):

```
#define QMN(type, fraq, val)   \
     (type)(val * (1u << (fraq)) + ((val >= 0)? 0.5f: -0.5f))
#define L1_WQ(val)   QMN(int8_t,  8, val)
#define L1_BQ(val)   QMN(int8_t,  7, val)

const  int8_t L1_conv_wt_buf[] = {\
     L1_WQ( 0.096343018),L1_WQ( 0.148116693),L1_WQ( 0.023189211), … \
     L1_WQ(-0.123411559),L1_WQ(-0.047247209),L1_WQ( 0.091348067), … \
     …
};
const int8_t  L1_conv_bias_buf[] = {\
     L1_BQ( 0.058115590),L1_BQ(-0.098249219),L1_BQ( 0.456347317), … \
     L1_BQ(-0.135683402),L1_BQ(-0.039959636),L1_BQ( 0.527986348), … \
     …
};
```

b) Or you may quantize data externally in the same way and just put them into code:

```
const int8_t L1_conv_wt_buf[] = {25, 38, 6, -12, -7, …}
const int8_t L1_conv_bt_buf[] = {7, -12, 58, -1, -25, …}
```

Declare Tensors structures

```
// Conv 1 Layer weights and biases tensors
static const mli_tensor L1_conv_wt = {
     .data = (void *)L1_conv_wt_buf,
     .capacity = sizeof(L1_conv_wt_buf),
     .shape =  {32, 3, 5, 5},           // Get Shape from the NP Array
     .rank =  4,
     .el_type = MLI_EL_FX_8,
     .el_params.fx.frac_bits = 8,
 };

static const mli_tensor L1_conv_bias = {
     .data = (void *)L1_conv_bias_buf,
     .capacity = sizeof(L1_conv_bias_buf),
     .shape =  {32},
     .rank =  1,
     .el_type = MLI_EL_FX_8,
     .el_params.fx.frac_bits = 7,
 };

// Next value will be passed with output tensor structure
#define CONV1_OUT_FRAQ_BITS (4)
…
```

Caffe CIFAR-10 Example: Deploying Data

CHW and HWC Data Layouts

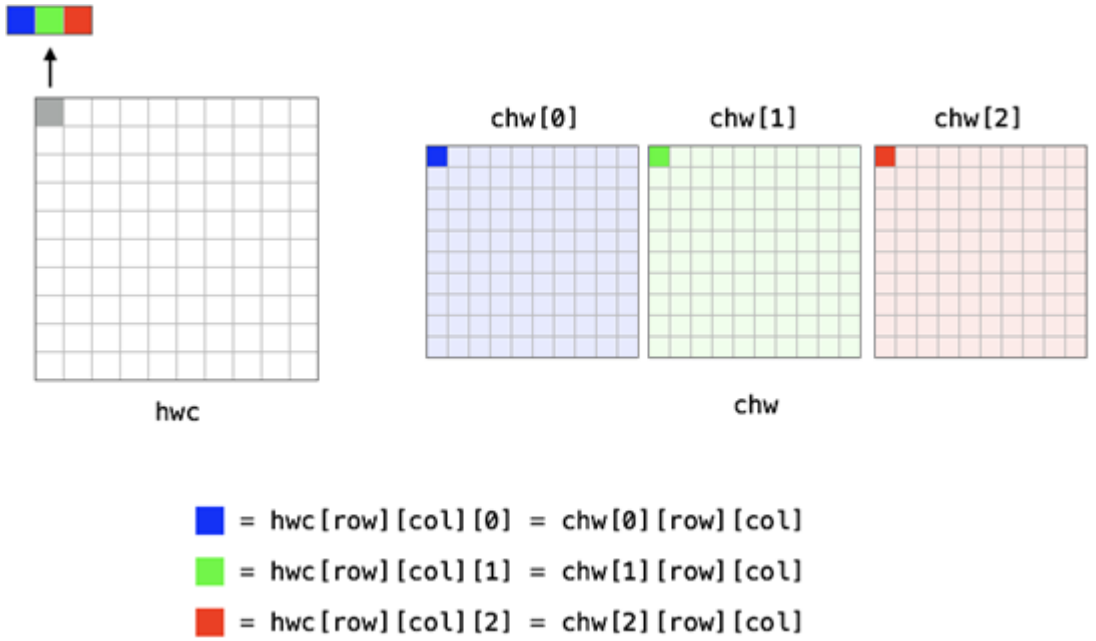
Convolution and pooling operations deal with multi-dimensional feature maps which might be considered as images.

In general, these maps have three dimensions with following names: **H**eight, **W**idth and **C**hannels (also called depth).

Order of elements in memory depends on the order of dimensions (data layout).

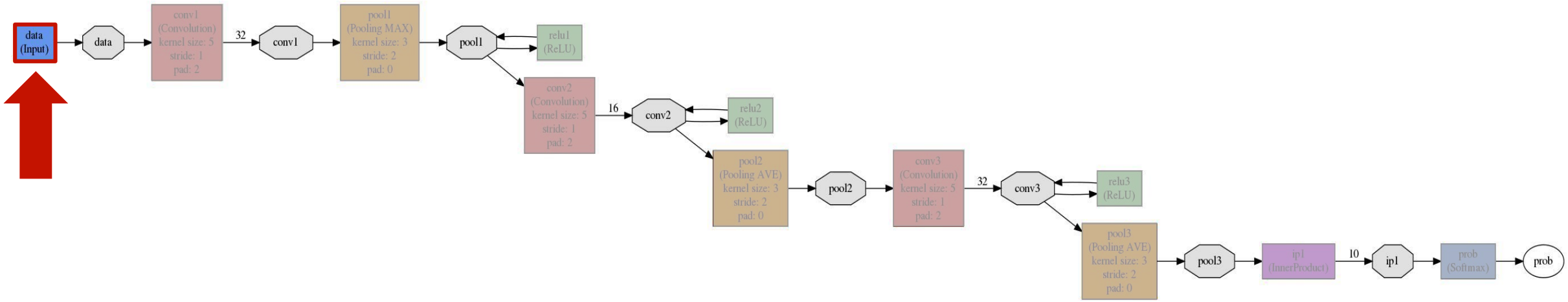
Two data layouts are traditionally used:

- HWC (**H**eight, **W**idth **C**hannel)
- CHW (**C**hannel **H**eight, **W**idth)



Caffe CIFAR-10 Example: Deploying Operations

Permute (Transpose) Input Data



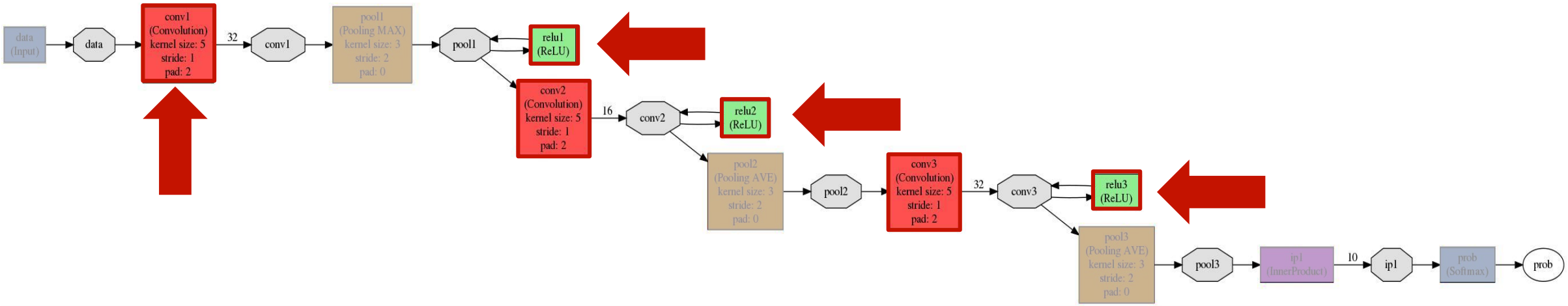
ProtoText description	MLI Function	MLI Function Config

layer {
  name: "data"
  type: "Input"
  top: "data"
  input_param {
  shape: { dim: 1 dim: 3 dim: 32 dim: 32
} }}
	mli_status mli_krn_permute_fx8(
	  const mli_tensor * in, // Input tensor
	  const mli_permute_cfg * cfg, // Permute configuration
	  mli_tensor * out // Output tensor
	);
		mli_permute_cfg permute_hwc2chw_cfg = {
		  .perm_dim =
		  {FMAP_C_DIM_HWC, // 2
		  FMAP_H_DIM_HWC, // 0
		  FMAP_W_DIM_HWC} // 1
		};

We assume input is RGB image (HWC Layout), while MLI mostly targets CHW layout.
We need to transpose data by permute layer.

Caffe CIFAR-10 Example: Deploying Operations

Convolution + ReLU Operations



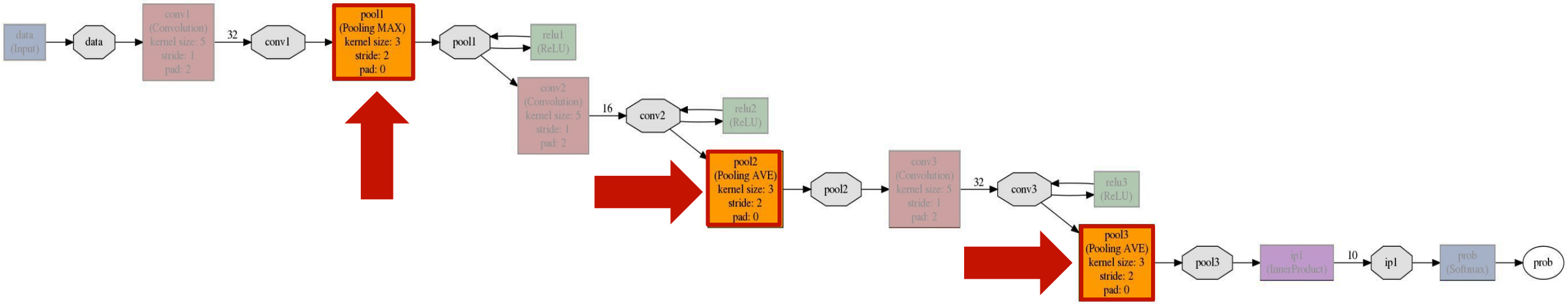
ProtoText description	MLI Function	MLI Function Config

layer {
name: "conv2“
type: "Convolution“
bottom: “pool1“
top: "conv2“
convolution_param {
num_output: 32
pad: 2 kernel_size: 5 stride: 1 }}
layer {
name: "relu2"
type: "ReLU"
bottom: "conv2"
top: "conv2"}
	mli_status mli_krn_conv2d_chw_fx8_k5x5_str1_krnpad(
	const mli_tensor * in, // Input tensor
	const mli_tensor * weights, // Weights tensor
	const mli_tensor * bias, // Biases tensor
	const mli_conv2d_cfg * cfg, // Convolution config
	mli_tensor * out // Output tensor
	);
		mli_conv2d_cfg shared_conv_cfg = {
		.stride_height = 1, .stride_width = 1,
		.padding_bottom = 2, .padding_top = 2,
		.padding_left = 2, .padding_right = 2,
		.relu.type = MLI_RELU_GEN
		};

- Use specialized functions for convolution kernel
- Use integrated ReLU functionality
- Order of max pooling and ReLU operations can be changed.

Caffe CIFAR-10 Example: Deploying Operations

Pooling Operations



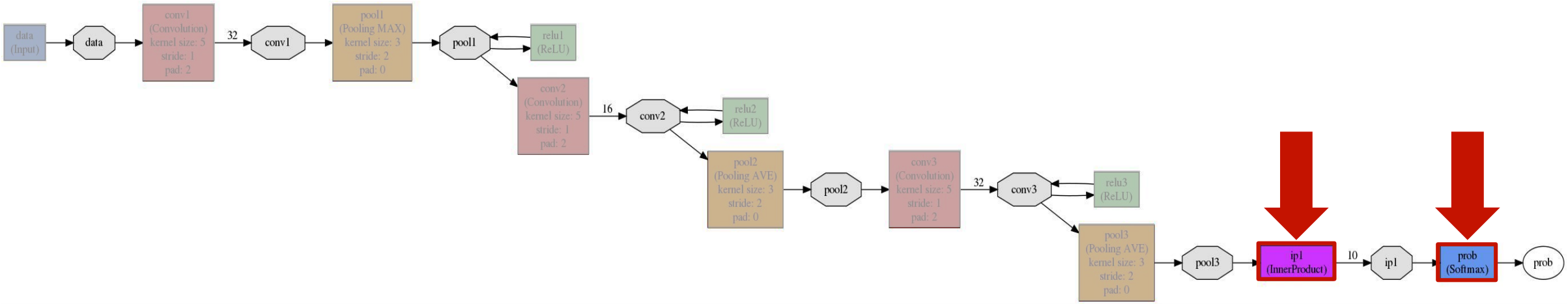
ProtoText description	MLI Function	MLI Function Config

layer {
name: "pool1"
type: "Pooling"
bottom: "conv1"
top: "pool1"
pooling_param {
pool: MAX kernel_size: 3 stride: 2
}}
	`mli_status mli_krn_maxpool_chw_fx8_k3x3(`
	`const mli_tensor * in, // Input tensor`
	`const mli_pool_cfg * cfg, // Pooling configuration`
	`mli_tensor * out // Output tensor`
	`);`
		`mli_pool_cfg shared_pool_cfg = {`
		`.kernel_height = 3, .kernel_width = 3,`
		`.stride_height = 2, .stride_width = 2,`
		`.padding_bottom = 1, .padding_top = 0,`
		`.padding_left = 0, .padding_right = 1`
		`};`

layer {
name: "pool2"
type: "Pooling"
bottom: "conv2"
top: "pool2"
pooling_param {
pool: AVE kernel_size: 3 stride: 2
}}
	`mli_status mli_krn_avepool_chw_fx8_k3x3(`
	`const mli_tensor * in, // Input tensor`
	`const mli_pool_cfg * cfg, // Pooling configuration`
	`mli_tensor * out // Output tensor`
	`);`
		We should indicate padding clearly because it is meant in the Caffe.

Caffe CIFAR-10 Example: Deploying Operations

Fully Connected and SoftMax Operations



ProtoText description	MLI Function	MLI Function Config

layer {
name: "ip1"
type: "InnerProduct“
bottom: "pool3“
top: "ip1"
inner_product_param {
num_output: 10
}}
	`mli_status mli_krn_fully_connected_fx8(`
	`const mli_tensor * in, // Input tensor`
	`const mli_tensor * weights, // Weights tensor`
	`const mli_tensor * bias, // Bias tensor`
	`mli_tensor * out // Output tensor`
	`);`
		No configuration is required (tensors provide all necessary information)

layer {
name: "prob"
type: "Softmax"
bottom: "ip1"
top: "prob"}
	`mli_status mli_krn_softmax_fx8(`
	`const mli_tensor * in, // Input tensor`
	`mli_tensor * out // Output tensor`
	`);`
		No configuration is required (tensors provide all necessary information)

Caffe CIFAR-10 Example: Deploying Operations

Final Sequence of Operations

When data extracted properly (wrapped into tensors and configuration structures), and functions for inference are defined, execution sequence in terms of MLI calls will be like this:

```
// LAYER 0: Change RGB Image layout
mli_krn_permute_fx16(&input, &permute_hwc2chw_cfg, &ir_tensor_Y);

// LAYER 1
ir_tensor_X.el_params.fx.frac_bits = CONV1_OUT_FRAQ;
mli_krn_conv2d_chw_fx8_k5x5_str1_krnpad(&ir_tensor_Y, &L1_conv_wt, &L1_conv_bias, &shared_conv_cfg, &ir_tensor_X);
mli_krn_maxpool_chw_fx16_k3x3(&ir_tensor_X, &shared_pool_cfg, &ir_tensor_Y);

// LAYER 2
ir_tensor_X.el_params.fx.frac_bits = CONV2_OUT_FRAQ;
mli_krn_conv2d_chw_fx8_k5x5_str1_krnpad(&ir_tensor_Y, &L2_conv_wt, &L2_conv_bias, &shared_conv_cfg, &ir_tensor_X);
mli_krn_avepool_chw_fx16_k3x3_krnpad(&ir_tensor_X, &shared_pool_cfg, &ir_tensor_Y);

// LAYER 3
ir_tensor_X.el_params.fx.frac_bits = CONV3_OUT_FRAQ;
mli_krn_conv2d_chw_fx8_k5x5_str1_krnpad(&ir_tensor_Y, &L3_conv_wt, &L3_conv_bias, &shared_conv_cfg, &ir_tensor_X);
mli_krn_avepool_chw_fx16_k3x3_krnpad(&ir_tensor_X, &shared_pool_cfg, &ir_tensor_Y);

// LAYER 4
ir_tensor_X.el_params.fx.frac_bits = FC4_OUT_FRAQ;
mli_krn_fully_connected_fx16(&ir_tensor_Y, &L4_fc_wt, &L4_fc_bias, &ir_tensor_X);
mli_krn_softmax_fx16(&ir_tensor_X, &output);
```

ir_tensor_X and *ir_tensor_X* are tensors for storing intermediate results (layers input/output).
They must keep valid pointer to buffer of sufficient capacity + number of fractional bits for MAC based layers.

Tips and Tricks

Bits Definition for fx16 Data

- MLI uses 40bit accumulator which provides 9 extra bits for processing up to 512 MAC operations in a row on 16x16 operands without saturation. For longer MAC series you should keep some bits in the operands unused to guarantee that the result won't saturate in accumulation.

	Integer bits requirements (fx8 operands)			Accumulator restrictions			Integer bits requirements (fx16 operands)		
	Layer input Integer bits	Layer weights Integer bits	Layer out Integer bits	Macs per output value	Required extra bits for accumulation	Not enough bits in accumulator	Layer input Integer bits (updated)	Layer weights Integer bits (updated)	Layer out Integer bits (updated)
Layer X conv	5	-1	5	32*5*5+1=801	10	10 – 9 = 1	**5 + 1 = 6**	-1	**6 (next layer in)**
Layer X+1 FC	5	-1	5	64*16+1=1025	11	11 – 9 = 2	**5 + 1 = 6**	**-1 + 1 = 0**	5

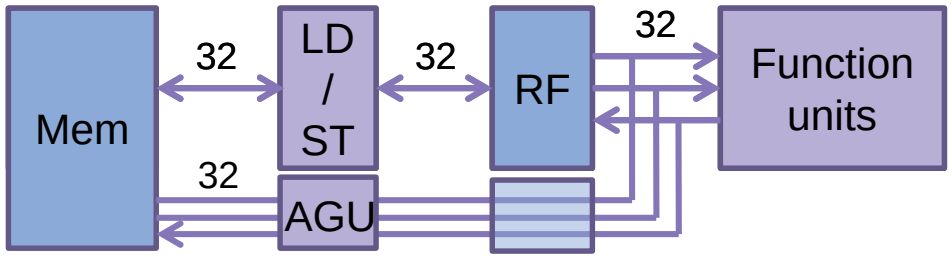
- For 8bit operands this isn't the case until your MAC series is more than 131072 operations.

Tips and Tricks

XY memory with AGUs for higher DSP performance

C source code

```
q31_t foo(__xy q31_t *b, __xy q31_t *c) {
   q31_t s = 0;
   for (i = 0; i <  N; i++)
      s += b[i] * c[i]
   return s;
}
```



	Non-XY	XY
Performance [MAC/cycle]	0.3	1.0
I-power [B/MAC]	12	4

Key benefits of XY:

- Higher DSP performance
- Lower I-memory power
- Smaller code size

Non-XY assembly

```
          // loop, unrolled 2x
          LP lpend
          LD %r2, [%r0, 4]
          LD %r3, [%r1, 4]
          LD %r4, [%r0, 4]
          LD %r5, [%r1, 4]
          MAC 0, %r2, %r3
          MAC 0, %r4, %r5
lpend: // epilogue for odd sized loops
          …
```

3x performance

XY assembly

```
          // prologue, set-up address gen
          SR ...
          SR ...
          LP lpend
          MAC 0, %agu_u0, %agu_u1
lpend: // no epilogue
          …
```


Tips and Tricks

XY Motivating Code Example

```
for (i=0; i<128;++i)
    acc += a[i] * c[i];
```

EM5D: MAC	EM9D (single-instance XY) bad memory allocation	EM9D (single-instance XY) good memory allocation

LP .lpend
  LD.AB %r0, %r3[4]
  LD.AB %r1, %r4[4]
  MAC 0, %r0, %r1
.lpend:
	SR %agu_mod0, INCR(4)
	SR %ap0, %r4
	…
	LP .lpend
	  MAC 0, %agu_u0, %agu_u1
	.lpend:
		SR %agu_mod0, INCR(4)
		SR %ap0, %r4
		…
		LP .lpend
		  MAC 0, %agu_u0, %agu_u1
		.lpend:

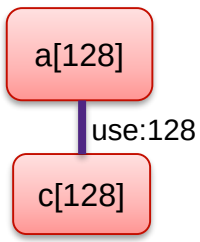
384 cycles	**256** cycles	**128** cycles (3.x from EM5D)

AGU setup

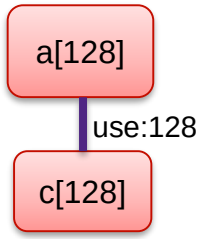
32-bit instruction with special operands can read data from CCM

1.5x speedup

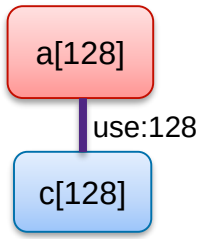
2.x speedup



DCCM



Both arrays in DCCM/X



Arrays in different logical memories

DCCM/X memory

YCCM memory

Legend: colors mean CCM memory bank

Tips and Tricks

Data Flow Using 2 Memories for Efficient AGU utilization

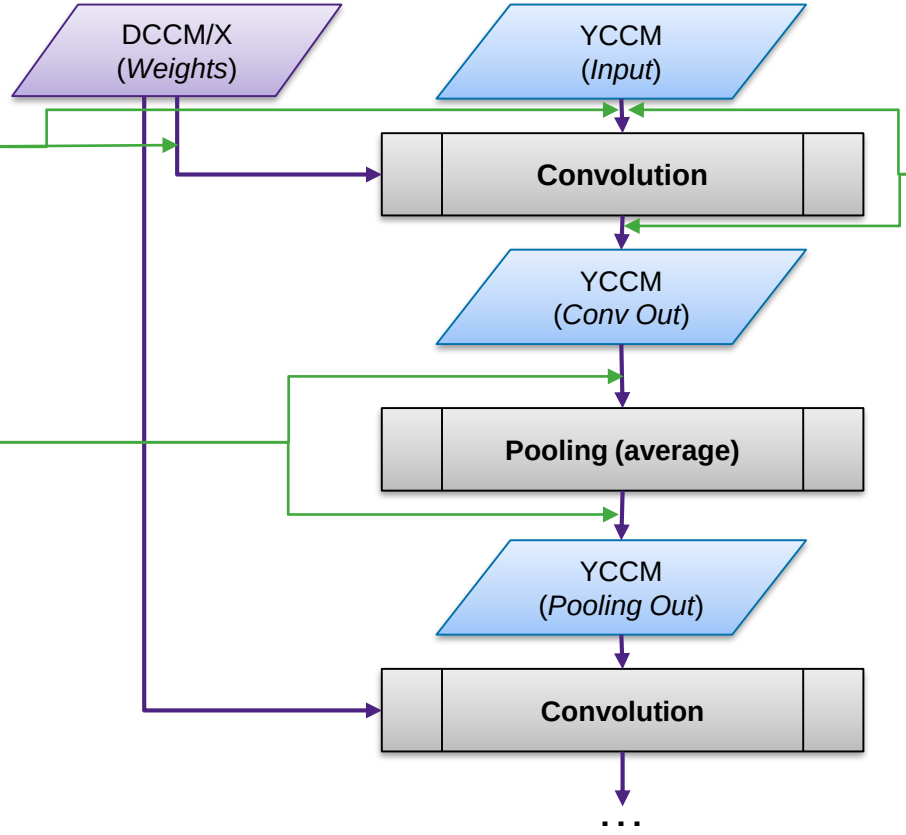
`vmac2hnfr 0,%agu_u0,%agu_u1`

2 Loads + DMAC in 1 cycle w/o stalls

AGU registers address different memories in MAC operation.

`add.p %r0,%r0,%agu_u0`

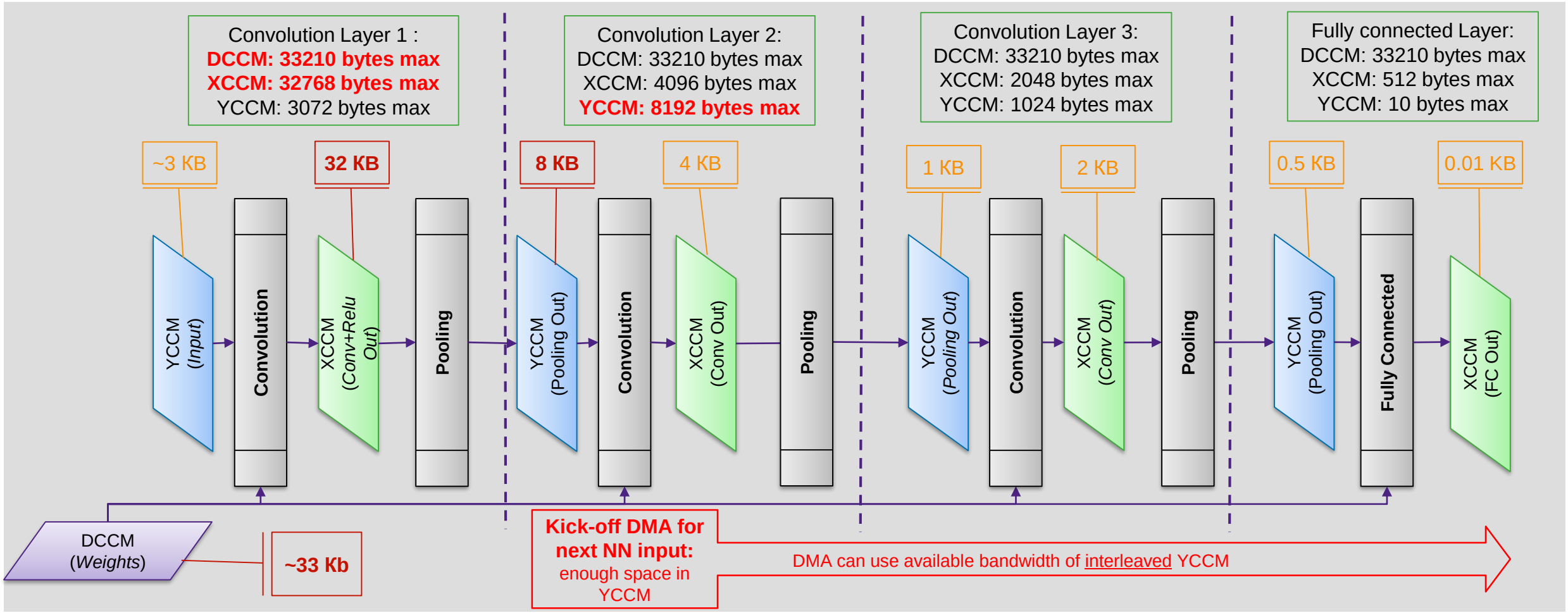
Load + ADD in 1 cycle w/o stalls



Loads and stores separated in time: output data stored after accumulation. The same logical memory can be used.

Tips and Tricks

CIFAR-10 with DCCM+X+Y memories



Final Requirements :

DCCM > 33 KB **XCCM 32 KB** **YCCM 8 KB**

Note: Here 2 memories instead of 3 may be used w/o affect on XY performance

SYNOPSYS®

Tips and Tricks

Batch normalization for Caffe

- Using Batch normalization may significantly improve training process. But for inference separate batch normalization is a heavy extra operation which may be avoided by propagating it into coefficients of the previous layer. For Caffe next transformation of weights can be applied.

```
# Extract learnde BN and Scale parameters
bn_varinace = classifier.params["bn_layer"][0]
bn_mean = classifier.params["bn_layer"][1]
bn_ma_factor = classifier.params["bn_layer"][2]
scale_gamma = classifier.params["sc_layer"][0]
scale_beta = classifier.params["sc_layer"][1]

# take into account MA factor of caffe Batch normalization
bn_varinace = bn_varinace / bn_ma_factor
bn_mean = bn_mean / bn_ma_factor

# 0.00001 is the default value of Caffe Batch Normalization layer (can be
changed)
scale_vals = np.reciprocal(np.sqrt(bn_varinace + 0.00001)) * scale_gamma

# Integrate bn and scale parameters into weights
conv_w = np.multiply(conv_w, scale_vals.reshape((-1, 1, 1, 1)))
conv_b = ((conv_b - bn_mean) * scale_vals) + scale_beta
```


SYNOPSYS®
Silicon to Software™