

DEPLOYING CUSTOM MODELS USING INTEL® NEURAL COMPUTE STICK 2 AND OPENVINO™ TOOLKIT

LEGAL DISCLAIMER

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at [intel.com](https://www.intel.com).

This sample source code is released under the Intel Sample Source Code License Agreement.

Intel, the Intel logo, Intel Atom, Intel Core, Movidius, Myriad, OpenVINO, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

AGENDA

Chapter Outcome: Deploy Custom Models on Intel® Neural Compute Stick 2 (Intel® NCS2) Using the OpenVINO™ Toolkit

- Why would you need a custom model?
- What makes a model “custom”?
- End-to-end training and inference of a custom model on Intel NCS2
- Additional profiling and fine tuning steps
- OpenVINO toolkit support for custom layers

WHY DO WE NEED A CUSTOM MODEL?

The OpenVINO™ toolkit model zoo downloads and compiles a number of pretrained deep neural networks such as GoogLeNet, AlexNet, SqueezeNet, MobileNet* and so on, trained on an ImageNet dataset, with over a thousand classes (also called categories) of images.

For a custom application like a door security camera, do we need all 1000 categories? NO.

Instead, you probably need just 15 to 20 classes, such as 'person', 'dog', 'mailman', 'person wearing hoody', and so on.

By reducing your dataset from a thousand classes down to 20, you are also reducing the number of features that need to be extracted.

WHAT MAKES A MODEL CUSTOM?

- Custom dataset
- Customized classification labels
- Custom layers within the model
 - The Intel® Deep Learning Deployment Toolkit (Intel® DLDT) of the OpenVINO™ toolkit supports standard layers for most of the popular Image Classification and Object Detection topologies.
 - Any layer that is unsupported by Intel DLDT is considered a custom layer. The OpenVINO toolkit needs explicit instructions to deploy a model with such layers. This will be covered later in the chapter.

BENEFITS OF CUSTOM NEURAL NETWORK MODELS

Customizing neural networks allows us to achieve the following:

1. Save time during network training because you have a reduced dataset.
 - This in turn saves money spent on keeping the training hardware up and running.
 - This also helps speed up development time, so you can get to market faster.
2. Reduce hardware bill of materials (BOM) cost by minimizing the memory footprint of your model.
3. The forward pass during inference would be faster because of the reduced complexity (that is, the edge device can process camera frames much faster).

SOME CONTEXT BEFORE WE BEGIN

Creating a custom dataset, customized labels, and custom layers are steps that are necessary during **model training**.

At the point of inference, the OpenVINO™ toolkit **refers** to the trained custom model, customized labels, and layers to deploy on the preferred hardware.

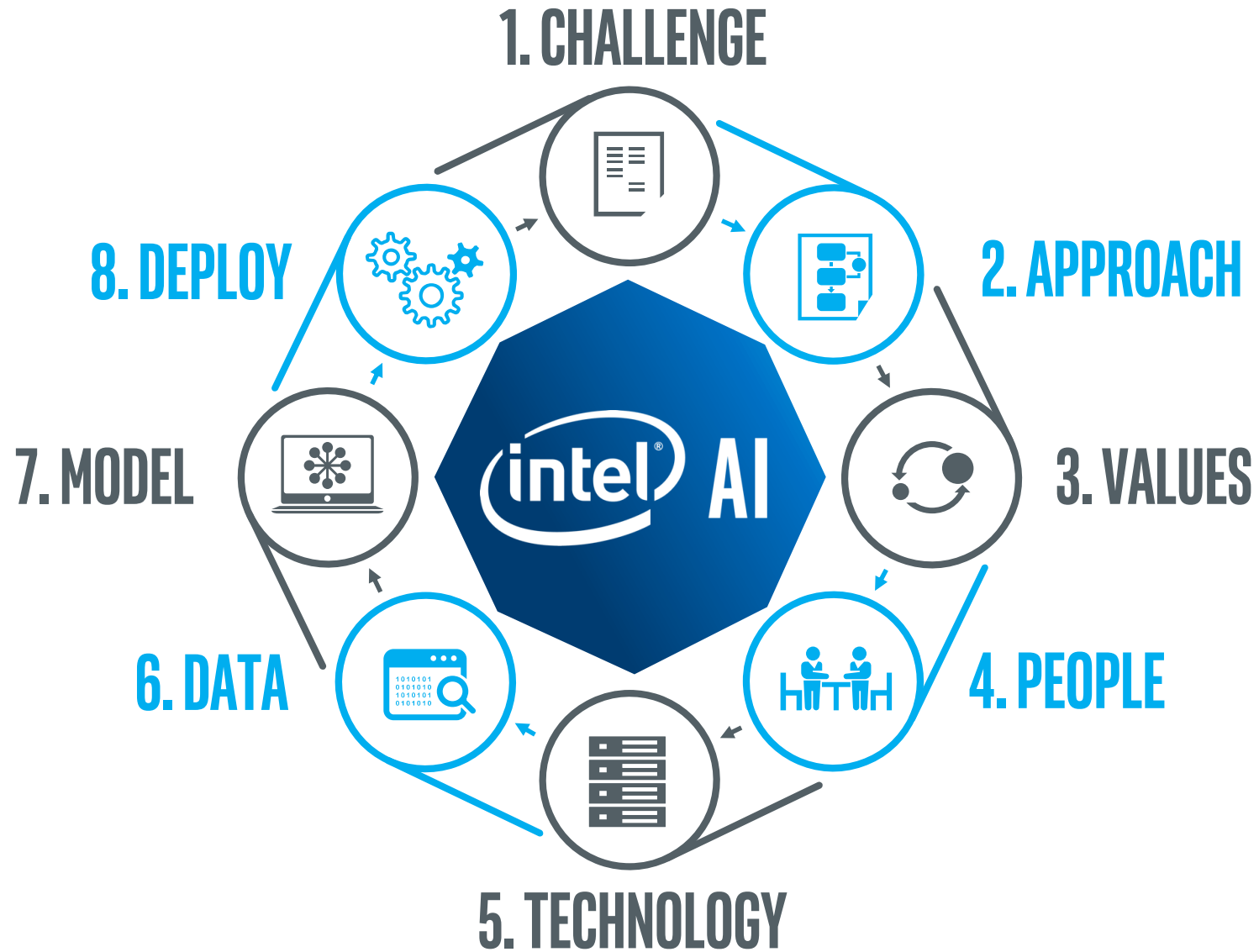
To illustrate this, we will start with a brief introduction of the end-to-end data science workflow, show steps needed for customization during training, and follow with deployment using the OpenVINO toolkit using two examples:

- Example 1: Image Classification model that uses a **custom dataset** and **customized classification labels**
- Example 2: Object Detection model that demonstrates the use of **custom layers** through the OpenVINO toolkit

Both examples will use TensorFlow*

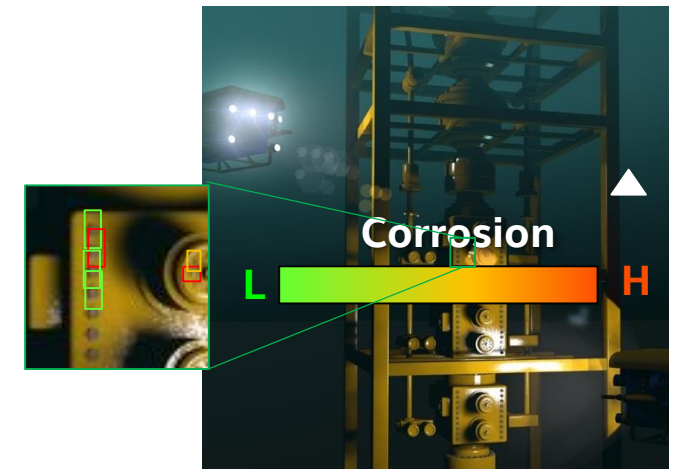
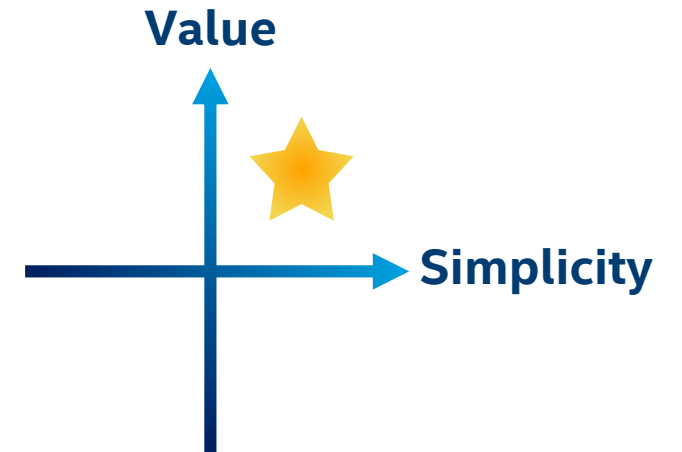
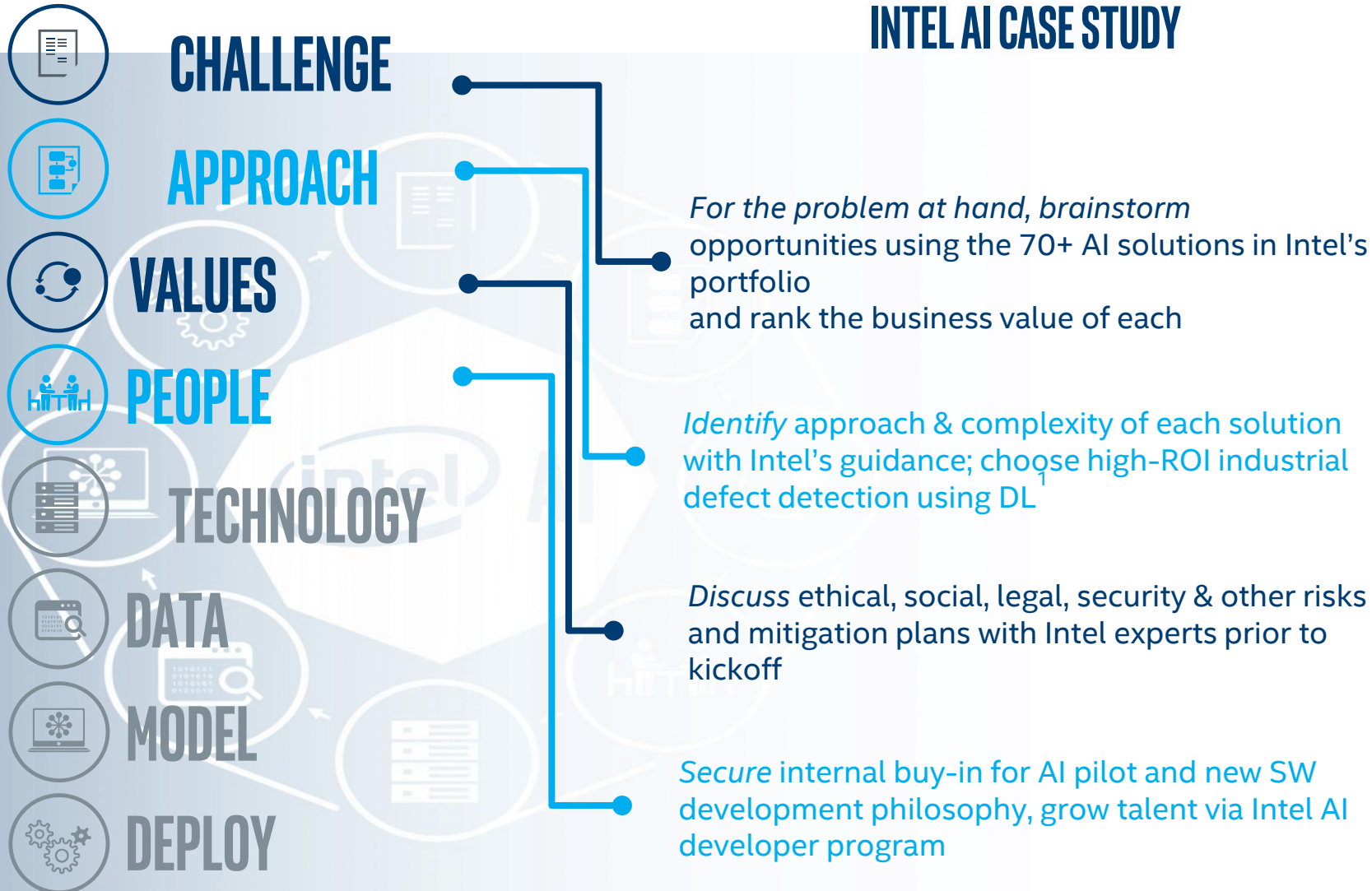
THE DATA SCIENCE PROCESS

STEPS INVOLVED



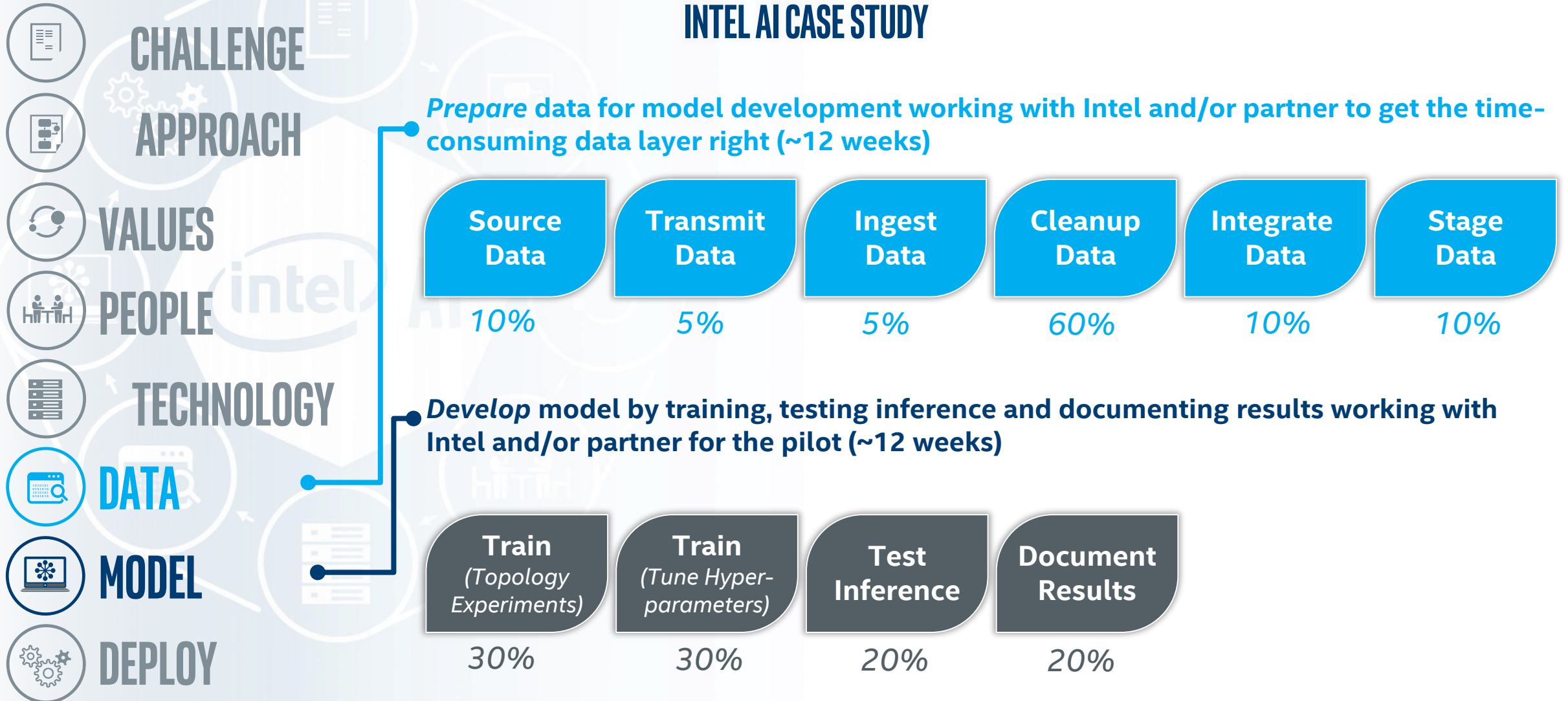
EXAMPLE END-TO-END DATA SCIENCE PROCESS

INTEL AI CASE STUDY



EXAMPLE END-TO-END DATA SCIENCE PROCESS

INTEL AI CASE STUDY



END-TO-END TRAINING—INFERENCE WORKFLOW USING THE OPENVINO™ TOOLKIT

END-TO-END TRAINING AND INFERENCE

TRAIN

Train a DL model.
Currently supports:

- Caffe*
- Apache MXNet*
- TensorFlow*



PREPARE OPTIMIZE

Model optimizer:

- Converting
- Optimizing
- Preparing for inference

(device agnostic,
generic optimization)

Run Model
Optimizer

IR
.xml
.bin

PROFILE

DL Workbench:

- Get throughput, latency, execution time per layer
- Visualize original IR and compare with MO optimized runtime graph

Run DL
WorkBench

IR
.xml
.bin

INFERENCE

Inference engine
lightweight API to use in
applications for inference

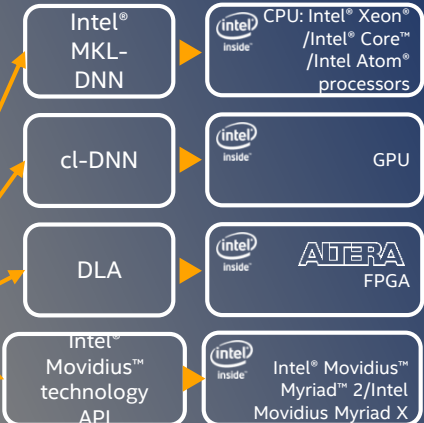
User Application

Inference Engine

OPTIMIZE/ HETEROGENEOUS

Inference engine supports
multiple devices for
heterogeneous flows

(device-level optimization)



STEPS INVOLVED IN TRAINING AND INFERENCE

STEP 1: Train a custom model—Challenge identification, dataset preparation, topology identification/fine tuning, and training

STEP 2: Fine tune—Optimize the neural network using the Model Optimizer to gain better execution time

STEP 3: Profile—Analyze the neural network for bandwidth, complexity, and execution time using the Deep Learning Workbench

STEP 4: Deploy—Deploy the customized neural network on an edge device powered by Intel® Movidius™ Neural Compute Stick

STEP 1: DATASET CUSTOMIZATION AND TOPOLOGY SPECIFIC TUNING

CREATE CUSTOM DATASET

What is the custom problem you are trying to solve?

- Create a custom data set to identify 10 most stolen cars in the US

What is the starter data set?

- Let's start with the VMMR dataset
- Contains 9000+ labels representing various models and types of cars

Create custom data set

- Contains labels for 10 most stolen cars
- Samples reduced from 290 K images to a few thousand images
- Apply data preprocessing techniques to accomplish customization

NETWORK CONFIGURATION PRE-DEPLOYMENT

Ensuring that deployment accuracy on NCS is equivalent to training accuracy requires configuring the following parameters:

- Mean subtraction
- Scale
- Color channel configuration
- Input image size

MEAN SUBTRACTION

- Mean subtraction on the input data to a convolutional neural network (CNN) is a common technique
- The mean is calculated on the data set
- For example, the mean on ImageNet is calculated on a per channel basis to be

104, 117, 123

These numbers are in BGR orientation.

Mean calculation in TensorFlow* is calculated for each topology differently. For more information refer to <https://tensorflow.org>.

SCALE

Typical 8-bit per pixel per channel images will have a scale of 0-255. Many CNN networks use the native scale, but some do not.

Example – TensorFlow* Inception v3: The `input_mean` and the `input_std` are listed below. This is a scaling factor.

```
input_mean = 128  
input_std = 128
```

You divide $255/128$, and it's about 2. In this case, the scale is 2, but the mean subtraction is 128. In the end, the scale is actually -1 to 1.

COLOR CHANNEL CONFIGURATION

- Different models may be trained with different color channel orientations (either RGB or BGR).
- Color channel configuration depends on the framework, topology, and image processing library used (for example, OpenCV, scikit-image).
- Typically, the Slim TensorFlow* models (at least Inception and MobileNet*) use RGB.

INPUT IMAGE SIZE FOR EXAMPLE TOPOLOGIES

Input image size configuration are topology specific. The following table shows the image size and color channel orientations for popular topologies:

TensorFlow*

Topologies	Color channel config	Image size
AlexNet	RGB	227 x 277
GoogLeNet	RGB	224 x 224
VGG-16	RGB	224 x 224
Inception V1	RGB	224 x 224
Inception V3	RGB	299 x 299
Inception V4	RGB	299 x 299
MobileNet*	RGB	224 x 224

Refer to the AI from the DataCenter to the Edge Course to understand training a custom Image Classification model using TensorFlow*/Keras*.

STEP 2: CUSTOMIZING FRAMEWORK/TOPOLOGY SPECIFIC PARAMETERS USING THE MODEL OPTIMIZER

MODEL OPTIMIZER GENERAL PARAMETERS

To maintain inference accuracy, maintaining consistency between topology specific parameters during training and inference is necessary.

The Model Optimizer provides general parameters to set the mean, scale, color channel configuration, and image size.

Model Optimizer flags:

- **-- input_shape:** [N,H,W,C] format to specify batch size, height of image, width of image, number of color channels
- **-- scale:** All input values coming from original network inputs will be divided by this value
- **-- reverse_input_channels:** Switch from BGR to RGB (only if input format is BGR). TensorFlow* uses RGB as the default format
- **-- mean_values:** Mean values to be used for the input image per channel, in [R,G,B] order
- **-- scale_values:** Scale values to be used for the input image per channel, in [R,G,B] order
- **-- data_type:** FP16 to deploy on the Intel® Movidius™ Neural Compute Stick

WHEN TO APPLY MEAN AND SCALE VALUES ON THE MO COMMAND LINE?

Input data is preprocessed in two ways:

The input preprocessing operations are a part of a topology. In this case, the application that uses the framework to infer the topology does not preprocess the input.

- Does not require mean and scale as inputs to MO

The input preprocessing operations are not a part of a topology and the preprocessing is performed within the application that feeds the model with an input data.

- Requires mean and scale as inputs to MO

The VMMR model we trained earlier in this chapter uses the first approach.

MODEL OPTIMIZER PARAMETER VALUES

Refer to the following table for default mean and scale values for different topologies in TensorFlow*

Topology	--mean_values	--scale
Inception v1	[127.5,127.5,127.5]	127.5
Inception v3	[127.5,127.5,127.5]	127.5
Inception V4	[127.5,127.5,127.5]	127.5
MobileNet* v1 224	[127.5,127.5,127.5]	127.5
VGG-16	[103.94,116.78,123.68]	1
VGG-19	[103.94,116.78,123.68]	1

For more information, refer to:

- [Model Optimizer general parameters](#)
- [TensorFlow/ topology specific values](#)

OUTPUT—INTERMEDIATE REPRESENTATION FILES

To generate the FP16 quantized intermediate representation (IR) files through the Model Optimizer for the VMMR model, run the following command:

- Go to the MO install directory
- Run the command:
 - `mo.py -input_model <path_to_pb_file> --model_name <name_of_IR> -
-output_dir <path_to_FP16_IR> --input_shape [1,299,299,3] -
data_type FP16`

FP16 quantized IR files:

- Bin file: Contains model weights
- Xml file: Contains model definition

STEP 3: PROFILING A NETWORK USING THE DEEP LEARNING WORKBENCH

WHAT IS THE DEEP LEARNING WORKBENCH?

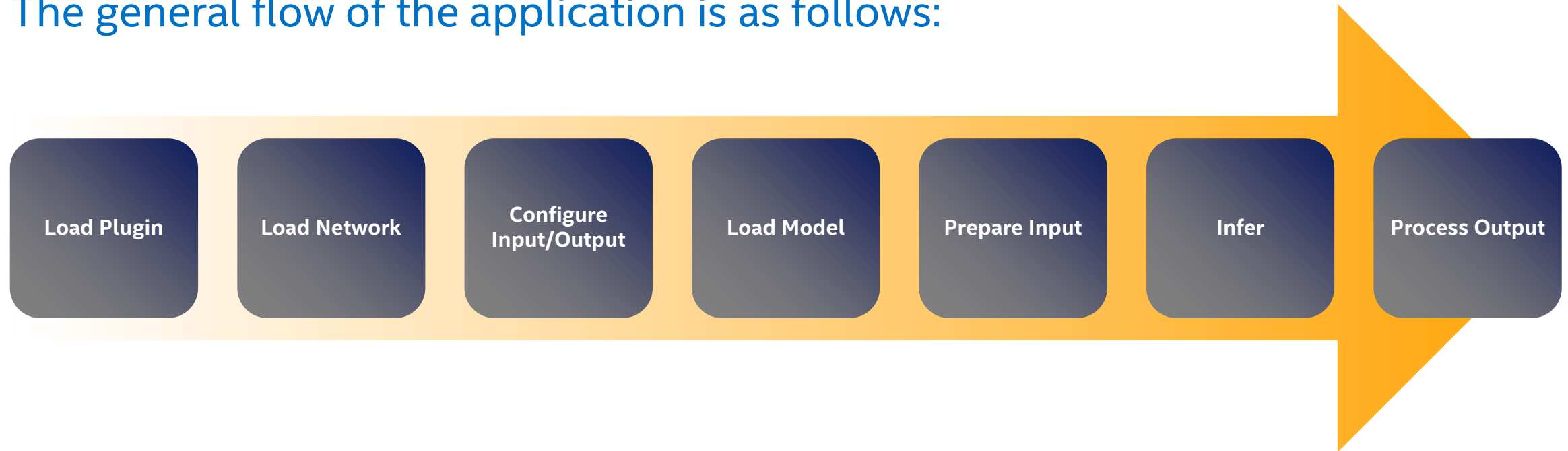
- A web-based graphical simulation environment to visualize deep learning models on Intel® architecture (CPU, processor graphics, VPU)
- Configure and measure accuracy of models
- Additionally, where possible, quantize a FP32 model to INT8 to fine-tune model performance
- **To profile your custom model using the DL Workbench, refer to Chapter 5**

STEP 4: DEPLOY A CUSTOM MODEL ON Intel® Neural Compute Stick 2

OPENVINO™ TOOLKIT—INFERENCE ENGINE WORKFLOW

Deploying a trained model on Intel® Neural Compute Stick 2 (Intel® NCS2) requires us to write an application that directs the model to be executed on the preferred hardware (CPU, GPU, or Intel NCS2).

The general flow of the application is as follows:



INFERENCE ENGINE FLAGS

The following flags are required to be passed to the Inference Engine:

- m: Model file - .xml file of the intermediate representation generated in Step 2
- i: Input image file path
- d: Device name – MYRIAD for deploying on Intel® Neural Compute Stick 2
- labels: Path to the labels file used by the model (optional)

You can test your model with the **classification_sample_async** application that was compiled during the OpenVINO™ toolkit installation in Chapter 2. Reference command line:

- `<path_to_sample>/classification_sample_async -m
<path_to_FP16model_xml> -i <path_to_image> -d MYRIAD`

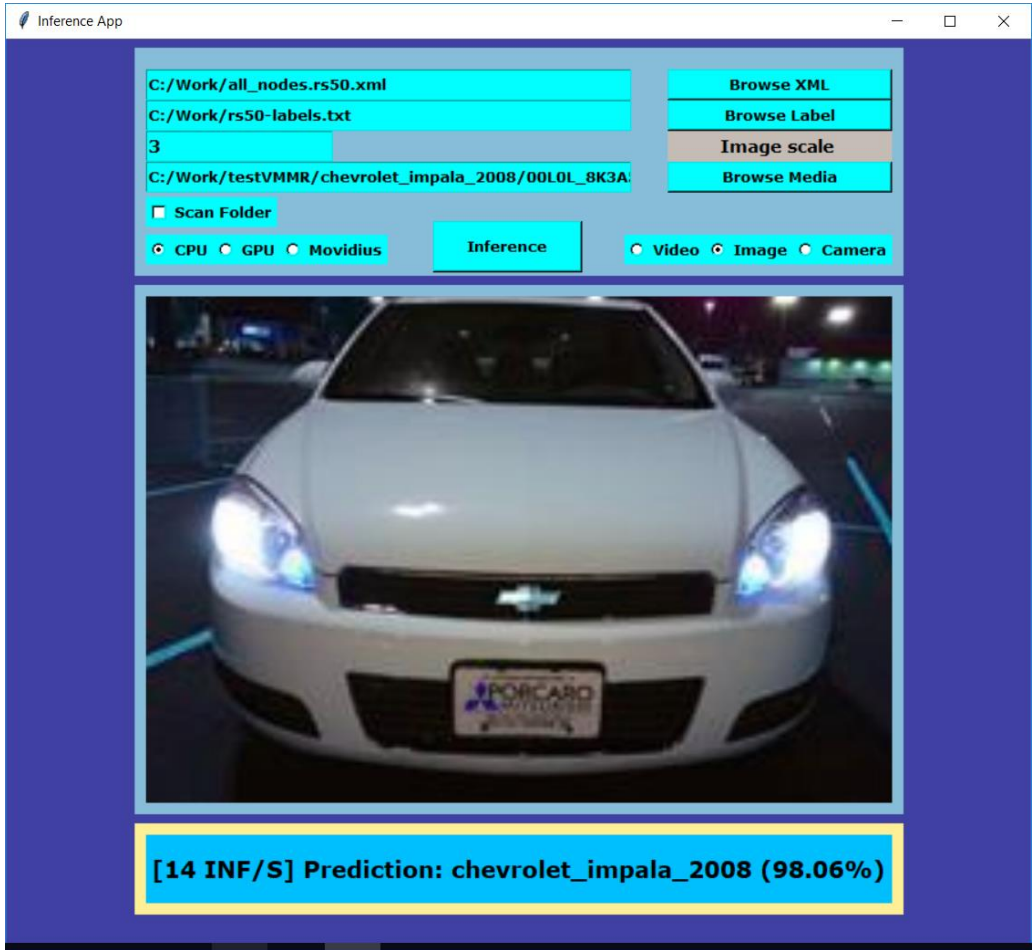
OUTPUT—PREDICTION OF CLASS LABEL WITH PERCENTAGE OF ACCURACY

- **Classification_sample_async**

Notice how the sample classifies the input image as GMC Sierra (label 8) with 100% accuracy

```
Top 10 results:
Image /home/meghana/openvino_models/VMMR_Model/FP16/SubsetVMMR_chevrolet_impala.jpg
classid probability
-----
8      1.0000000
4      0.0000000
7      0.0000000
9      0.0000000
3      0.0000000
5      0.0000000
1      0.0000000
0      0.0000000
2      0.0000000
6      0.0000000
[ INFO ] Execution successful
```

Custom application



USE OF CUSTOMIZED LAYERS IN DEEP LEARNING MODELS

STANDARD/CUSTOM TENSORFLOW LAYERS SUPPORTED BY OPENVINO™ TOOLKIT

- Most common operations included in popular deep learning topologies can be converted into intermediate representation layers through the Model Optimizer
- Some layers can be fused with others in order to optimize model performance
- List of ALL TensorFlow* operations that can be mapped to intermediate representation layers can be found here:
https://docs.openvino toolkit.org/latest/_docs_MO_DG_prepare_model_Supported_Frameworks_Layers.html
- Any operation that does not have such a direct mapping is considered a **Custom Layer** within the model

HOW DO I CONVERT A MODEL WITH CUSTOM LAYERS USING OPENVINO™ TOOLKIT?

There are three ways to address custom layers in trained models:

Register custom layers as extensions to the Model Optimizer (MO) (covered in this chapter)

- The MO generates a valid and optimized intermediate representation (IR)

Sub-graph replacement in the MO

- Especially useful if you have sub-graphs that should not be expressed with the analogous sub-graph in the IR, but another sub-graph should appear in the model

Registering definite sub-graphs of the model as those that should be offloaded to TensorFlow* during inference

- MO generates IR that can only be inferred on CPU
- Each sub-graph is mapped to a single custom layer in IR

Read More:

https://docs.openvino toolkit.org/latest/_docs_MO_DG_prepare_model_customize_model_optimizer_Customize_Model_Optimizer.html

MO FLAGS THAT SUPPORT CUSTOM LAYERS

- `--tensorflow_use_custom_operations_config`: Use the configuration file with custom operation description
- `--tensorflow_custom_operations_config_update`: Update the configuration file with node name patterns with input/output nodes information

Read More:

https://docs.openvinotoolkit.org/latest/_docs_MO_DG_prepare_model_convert_model_Convert_Model_From_TensorFlow.html

CUSTOM LAYER EXTENSIONS SUPPORT IN OPENVINO TOOLKIT

For the convenience of developers, the OpenVINO™ toolkit comes with an extension library that provides the definition of custom layers.

You can find them in the path:

<OPENVINO_INSTALL_DIR>/deployment_tools/model_optimizer/extensions/front/tf

These libraries are compiled by default when the OpenVINO toolkit samples are built as described in Chapter 2.

If this step was not done, the libraries will need to be compiled and your application will have to use the AddExtension method to load and use these extensions.

Read more:

https://docs.openvinotoolkit.org/2019_R1/_inference_engine_src_extension_README.html

EXAMPLE—YOLO V3 CUSTOM LAYERS

Why does YOLO* v3 need custom layers in TensorFlow*?

- YOLO v3 feature extractor (Darknet-53) includes three branches at the end for making predictions at three different scales. These branches must end with a Region layer.
- The Region layer is not implemented as a single layer in TensorFlow.
- Every public YOLO v3 model has implemented the Region layer using simpler layers leading to non-uniformity and reduced performance.
- To generate IR for YOLO v3 using MO, these variant Region layers are cut off from the original model and a custom Region layer definition is added to the MO as a custom extension.

YOLO* V3.JSON CUSTOM EXTENSION

The yolo_v3.json custom extension implements the Region layer as shown:

```
[
{
  "id": "TFYOLOV3",
  "match_kind": "general",
  "custom_attributes": {
    "classes": 80,
    "coords": 4,
    "num": 9,
    "mask": [0, 1, 2],
    "entry_points": ["detector/yolo-v3/Reshape", "detector/yolo-v3/Reshape_4", "detector/yolo-v3/Reshape_8"]
  }
}
```

- The “entry_points” custom attribute defines the Region layer
- The rest of the custom attributes can be taken from yolov3.cfg (if Darknet official shared weights are used for training)

GENERATING YOLO V3 IR USING CUSTOM EXTENSIONS THROUGH MO

Using the `--tensorflow_use_custom_operations_config` flag as an input to the MO, use the following command line to generate IR for YOLO* v3

```
python3 <MO_INSTALL_DIR>/mo_tf.py --input_model frozen_darknet_yolov3_model.pb --batch 1 --tensorflow_use_custom_operations_config /opt/intel/openvino/deployment_tools/model_optimizer/extensions/front/tf/yolo_v3.json -o FP16 --data_type FP16
```

- `--input_model`: Frozen graph (for steps on how to train YOLO v3, refer to Chapter 4)
- `--batch`: Batch size
- `--tensorflow_use_custom_operations_config`: Uses yolo_v3.json extension
- `-o`: Output path where the .bin and .xml files are saved
- `--data_type`: Required precision for the hardware you choose to deploy on. Since we will be deploying on the Intel® Neural Compute Stick 2, we select

DEPLOY THE IR FILES WITH SAMPLE APPS

The OpenVINO™ toolkit installation comes with many sample applications that integrate the Inference Engine capabilities for demonstration

- Build these samples using the OpenVINO toolkit installation instructions
- To test the IR we just created, we will use the `object_detection_demo_yolov3_async` demo

Instructions

- The Python* demos will be found in the `<OPENVINO_INSTALL_DIR>/deployment_tools/open_model_zoo/demos/python_demos` directory
- Download a sample video from <https://github.com/intel-iot-devkit/sample-videos>
- Run the following command to execute the sample using a sample video and infer on the Intel®

```
<OPENVINO_INSTALL_DIR>/deployment_tools/open_model_zoo/demos/python_demos/object_detection_demo_yolov3_async$python3 object_detection_demo_yolov3_async.py -i <VIDEO_PATH>/person-bicycle-car-detection.mp4 -m <PATH_to_FP16_model_xml_file> -d MYRIAD
```

OUTPUT—YOLO* V3 BOUNDING BOX WITH LABEL INDEX AND ACCURACY

Output:



Demonstrates:

- A bounding box around objects detected
- The label of the object detected (Example: Person is 0, car is 672. To understand this, check the coco.names labels file we downloaded in Step 1. The labels are 0 indexed)
- % accuracy

