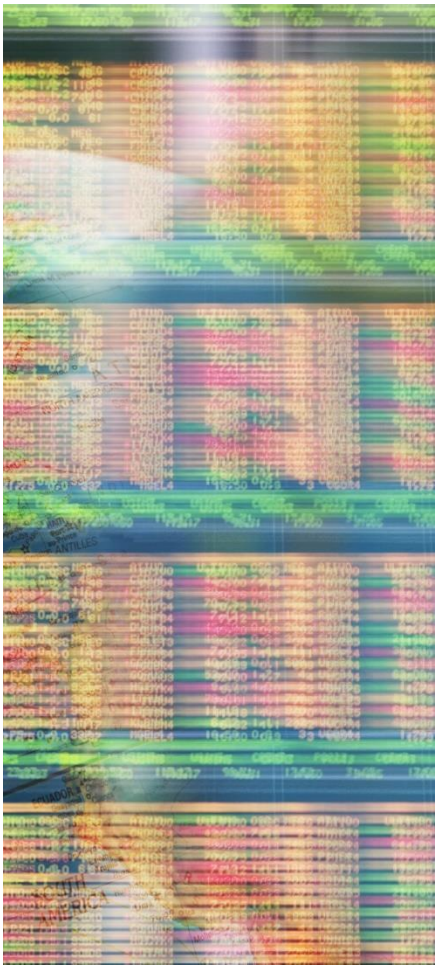
Optimization Guide

Intel® Xeon® Scalable Processor

Intel® Distribution of OpenVINO™ Toolkit Tuning Guide on 3rd Generation Intel® Xeon® Scalable Processors Based Platform



Revision Record

Date	Rev.	Description
08/15/21	1.0	Initial Release

1. Introduction

This guide is targeted towards users who are already familiar with Intel® Distribution of OpenVINO™ toolkit and provides pointers and system setting for hardware and software that will provide the best performance for most situations. However, please note that we rely on the users to carefully consider these settings for their specific scenarios, since Intel® Distribution of OpenVINO™ toolkit can be deployed in multiple ways and this is a reference to one such use-case.

OpenVINO™ toolkit is a comprehensive toolkit for quickly developing applications and solutions that solve a variety of tasks including emulation of human vision, automatic speech recognition, natural language processing, recommendation systems, and many others. Based on latest generations of artificial neural networks, including Convolutional Neural Networks (CNNs), recurrent and attention-based networks, the toolkit extends computer vision and non-vision workloads across Intel® hardware, maximizing performance. It accelerates applications with high-performance, AI and deep learning inference deployed from edge to cloud.

OpenVINO™ toolkit:

- Enables CNN-based deep learning inference on the edge
- Supports heterogeneous execution across an Intel® CPU, Intel® Integrated Graphics, Intel® Neural Compute Stick 2 and Intel® Vision Accelerator Design with Intel® Movidius™ VPUs
- Speeds time-to-market via an easy-to-use library of computer vision functions and pre-optimized kernels
- Includes optimized calls for computer vision standards, including OpenCV* and OpenCL™

3rd Gen Intel® Xeon® Scalable processors deliver industry-leading, workload-optimized platforms with built-in AI acceleration, providing a seamless performance foundation to help speed data's transformative impact, from the multi-cloud to the intelligent edge and back.

1.1. OpenVINO™ Toolkit Workflow

The following diagram illustrates the typical OpenVINO™ workflow:

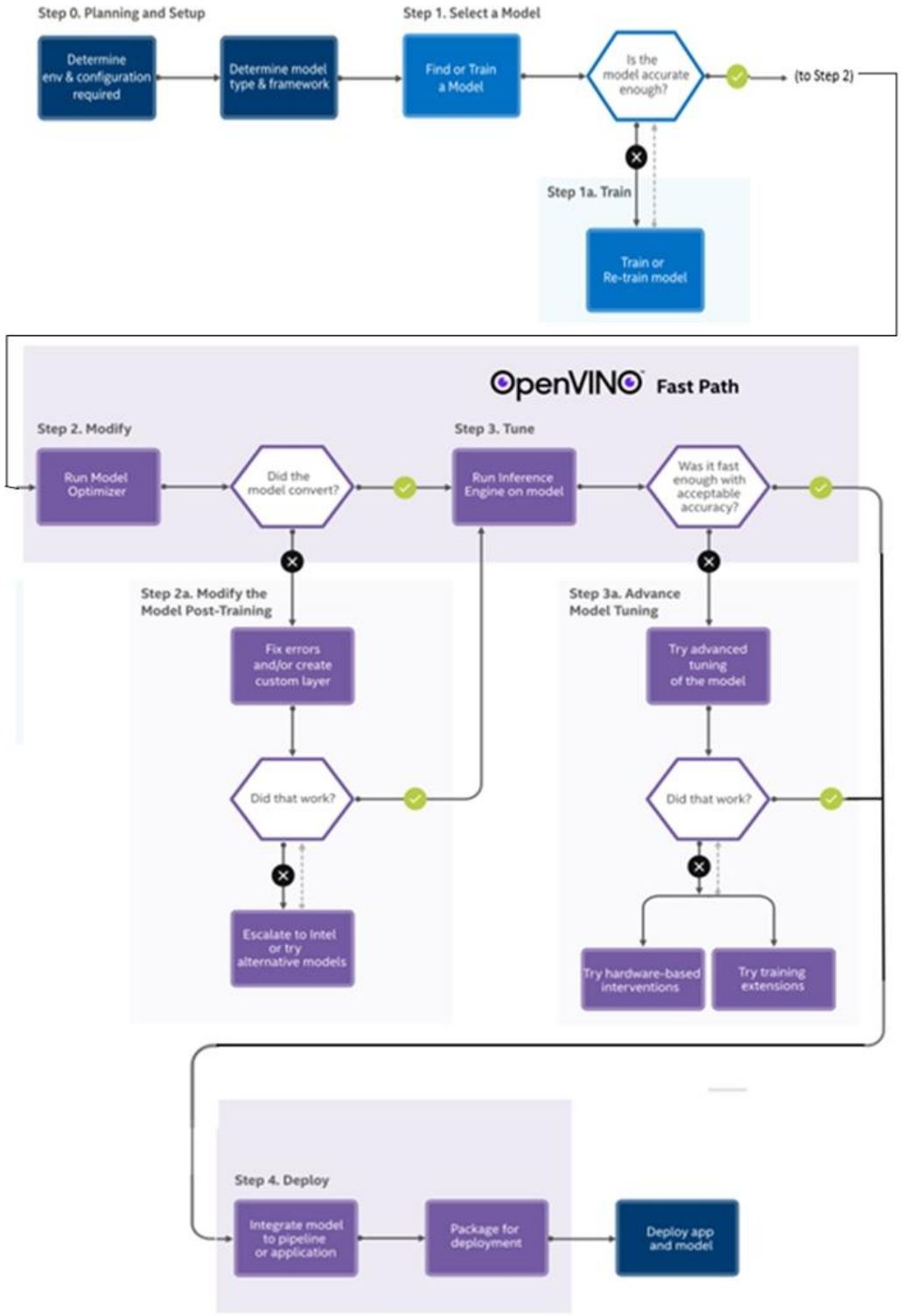


Figure 1: Typical OpenVINO™ workflow[1]

1.2. OpenVINO™ Toolkit Components

Intel® Distribution of OpenVINO™ toolkit includes the following components:

- Deep Learning Model Optimizer - A cross-platform command-line tool for importing models and preparing them for optimal execution with the Inference Engine. The Model Optimizer imports, converts, and optimizes models, which were trained in popular frameworks, such as Caffe*, TensorFlow*, MXNet*, Kaldi*, and ONNX*.
- Deep Learning Inference Engine - A unified API to allow high performance inference on many hardware types including Intel® CPU, Intel® Integrated Graphics, Intel® Neural Compute Stick 2, Intel® Vision Accelerator Design with Intel® Movidius™ vision processing unit (VPU).
- Inference Engine Samples - A set of simple console applications demonstrating how to use the Inference Engine in your applications.
- Deep Learning Workbench - A web-based graphical environment that allows you to easily use various sophisticated OpenVINO™ toolkit components.
- Post-Training Optimization tool - A tool to calibrate a model and then execute it in the INT8 precision.
- Additional Tools - A set of tools to work with your models including Benchmark App, Cross Check Tool, Compile tool.
- Open Model Zoo
    - Demos - Console applications that provide robust application templates to help you implement specific deep learning scenarios.
    - Additional Tools - A set of tools to work with your models including Accuracy Checker Utility and Model Downloader.
    - Documentation for Pretrained Models - Documentation for pretrained models that are available in the Open Model Zoo repository.
- Deep Learning Streamer (DL Streamer) – Streaming analytics framework, based on GStreamer, for constructing graphs of media analytics components. DL Streamer can be installed by the Intel® Distribution of OpenVINO™ toolkit installer. Its open source version is available on GitHub. For the DL Streamer documentation, see:
    - DL Streamer Samples
    - API Reference
    - Elements
    - Tutorial
- OpenCV - OpenCV* community version compiled for Intel® hardware
- Intel® Media SDK (in Intel® Distribution of OpenVINO™ toolkit for Linux only) (https://docs.openvinotoolkit.org/2021.1/index.html)

For building the Inference Engine from the source code, see the build instructions.

2. Installation Guides

Please follow the steps below to install OpenVINO™ and configure the third-party dependencies based on your

preference. Please look at the Target System Platform requirements before installation.

OS Based	Install from Images and Repositories
Download Page: https://software.intel.com/en-us/openvino-toolkit/choose-download • Linux • Windows • macOS Raspbian OS	• Docker • Docker with DL Workbench • APT • YUM • Anaconda Cloud • Yocto PyPI

3. Get Started with OpenVINO™ Model Zoo

The Open Model Zoo is part of Intel® Distribution of OpenVINO™ toolkit which includes optimized deep learning models and a set of demos to expedite development of high-performance deep learning inference applications. You can use these free pre-trained models instead of training your own models to speed-up the development and production deployment process.
To check the currently available models, you can use Model Downloader as a very handy tool. It is a set of python scripts that can help you browse and download these pre-trained models. Other automation tools also available to leverage:

- downloader.py (model downloader) downloads model files from online sources and, if necessary, patches them to make them more usable with Model Optimizer.
- converter.py (model converter) converts the models that are not in the Inference Engine IR format into that format using Model Optimizer.
- quantizer.py (model quantizer) quantizes full-precision models in the IR format into low-precision versions using Post-Training Optimization Toolkit.
- info_dumper.py (model information dumper) prints information about the models in a stable machine-readable format

You can run the downloader.py as shown below. Note that the following example is conducted on a Linux* machine with source installation. If you plan to use it on a different setting, please change the path of the tools accordingly.

```
python3 /opt/intel/openvino_2021/deployment_tools/open_model_zoo/tools/downloader/downloader.py --help
```

```
usage: downloader.py [-h] [--name PAT[,PAT...]] [--list FILE.LST] [--all]
                     [--print_all] [--precisions PREC[,PREC...]] [-o DIR]
                     [--cache_dir DIR] [--num_attempts N]
```

```
                    [--progress_format {text,json}] [-j N]


optional arguments:
  -h, --help            show this help message and exit
  --name PAT[,PAT...]   download only models whose names match at least one of
                        the specified patterns
  --list FILE.LST       download only models whose names match at least one of
                        the patterns in the specified file
  --all                 download all available models
  --print_all           print all available models
  --precisions PREC[,PREC...]
                        download only models with the specified precisions
                        (actual for DLDT networks)
  -o DIR, --output_dir DIR
                        path where to save models
  --cache_dir DIR       directory to use as a cache for downloaded files
  --num_attempts N      attempt each download up to N times
  --progress_format {text,json}
                        which format to use for progress reporting
  -j N, --jobs N        how many downloads to perform concurrently
```

You can use the parameter --print_all to see which pre-trained models are supported by the current version of OpenVINO for download. We will choose a classical computer vision network to detect the target picture. With the command below, we will download ssd_mobilenet_v1_coco using Model Downloader.

```
python3 /opt/intel/openvino_2021/deployment_tools/open_model_zoo/tools/downloader/downloader.py --name
ssd_mobilenet_v1_coco
```

4. OpenVINO™ Model Optimizer

Model Optimizer is a cross-platform command-line tool that facilitates the transition between the training and deployment environment, performs static model analysis, and adjusts deep learning models for optimal execution on end-point target devices. (https://docs.openvinotoolkit.org/2021.3/openvino_docs_MO_DG_Deep_Learning_Model_Optimizer_DevGuide.html)

Model Optimizer process assumes you have a network model trained using a supported deep learning framework. The scheme below illustrates the typical workflow for deploying a trained deep learning model:

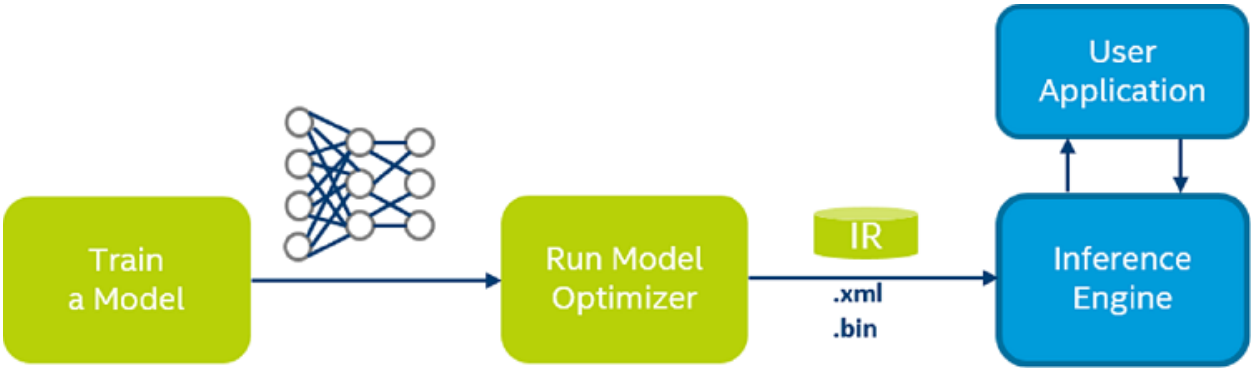


Figure 2: Typical workflow for deploying a trained deep learning model[2]

- .xml - Describes the network topology
- .bin - Contains the weights and biases binary data.

To be able to convert ssd_mobilenet_v1_coco model into IR, some model specific parameters are needed to be provided the Model Optimizer. Since we downloaded this model from Open Model Zoo, we also have created a yml file to provide model specific information in each file. Here is an example for ssd_mobilenet_v1_coco:

```
cat /opt/intel/openvino_2021/deployment_tools/open_model_zoo/models/public/ssd_mobilenet_v1_coco/model.yml
```

The Model Downloader also contains another handy script 'converter.py' that helps us to accurately input the parameters of the downloaded model to the Model Optimizer (MO). We can use this script directly for model conversion and reduce the workload considerably.

```
python3 /opt/intel/openvino_2021/deployment_tools/open_model_zoo/tools/downloader/converter.py \
    --download_dir=. \
    --output_dir=. \
    --name=ssd_mobilenet_v1_coco \
    --dry_run
```

We can either let “converter.py” convert the model directly or use the MO execution parameters that are generated by

the command above and use it when running MO.

```
python3 /opt/intel/openvino/deployment_tools/model_optimizer/mo.py \
    --framework=tf \
    --data_type=FP32 \
    --output_dir=public/ssd_mobilenet_v1_coco/FP32 \
    --model_name=ssd_mobilenet_v1_coco \
    --reverse_input_channels \
    --input_shape=[1,300,300,3] \
    --input=image_tensor \
    --output=detection_scores,detection_boxes,num_detections \
    --
transformations_config=/opt/intel/openvino/deployment_tools/model_optimizer/extensions/front/tf/ssd_v2_su
pport.json \
    --
tensorflow_object_detection_api_pipeline_config=public/ssd_mobilenet_v1_coco/ssd_mobilenet_v1_coco_2018_0
1_28/pipeline.config \
    --input_model=public/ssd_mobilenet_v1_coco/ssd_mobilenet_v1_coco_2018_01_28/frozen_inference_graph.pb
```

Model Optimizer arguments:

Common parameters:

```
 - Path to the Input Model:
/root/jupyter_root/public/ssd_mobilenet_v1_coco/ssd_mobilenet_v1_coco_2018_01_28/frozen_inference_graph.
pb
 - Path for generated IR:  /root/jupyter_root/public/ssd_mobilenet_v1_coco/FP32
 - IR output name:     ssd_mobilenet_v1_coco
 - Log level:     ERROR
 - Batch:     Not specified, inherited from the model
 - Input layers:  image_tensor
 - Output layers: detection_scores,detection_boxes,num_detections
 - Input shapes:  [1,300,300,3]
 - Mean values:   Not specified
 - Scale values:  Not specified
```

```
 - Scale factor:  Not specified

 - Precision of IR:   FP32

 - Enable fusing: True

 - Enable grouped convolutions fusing:  True

 - Move mean values to preprocess section:  None

 - Reverse input channels:     True

TensorFlow specific parameters:

 - Input model in text protobuf format:     False

 - Path to model dump for TensorBoard:  None

 - List of shared libraries with TensorFlow custom layers implementation: None

 - Update the configuration file with input/output node names:    None

 - Use configuration file used to generate the model with Object Detection API:
 /root/jupyter_root/public/ssd_mobilenet_v1_coco/ssd_mobilenet_v1_coco_2018_01_28/pipeline.config

 - Use the config file:    None

 - Inference Engine found in:  /opt/intel/openvino/python/python3.6/openvino

Inference Engine version:  2.1.2021.3.0-2787-60059f2c755-releases/2021/3

Model Optimizer version:      2021.3.0-2787-60059f2c755-releases/2021/3

The Preprocessor block has been removed. Only nodes performing mean value subtraction and scaling (if
applicable) are kept.

[ SUCCESS ] Generated IR version 10 model.

[ SUCCESS ] XML file: /root/jupyter_root/public/ssd_mobilenet_v1_coco/FP32/ssd_mobilenet_v1_coco.xml

[ SUCCESS ] BIN file: /root/jupyter_root/public/ssd_mobilenet_v1_coco/FP32/ssd_mobilenet_v1_coco.bin

[ SUCCESS ] Total execution time: 47.92 seconds.

[ SUCCESS ] Memory consumed: 455 MB.
```

5. Practice Inference Engine API

After creating Intermediate Representation (IR) files using the Model Optimizer, use the Inference Engine to infer the result for a given input data. The Inference Engine is a 2C++ library with a set of C++ classes to infer input data (images) and get a result. The C++ library provides an API to read the Intermediate Representation, set the input and output formats, and execute the model on devices.

Inference Engine uses a plugin architecture. Inference Engine plugin is a software component that contains complete implementation for inference on a certain Intel® hardware device: CPU, GPU, VPU, FPGA, etc. Each plugin implements

the unified API and provides additional hardware-specific APIs. Integration process consists of the following steps:

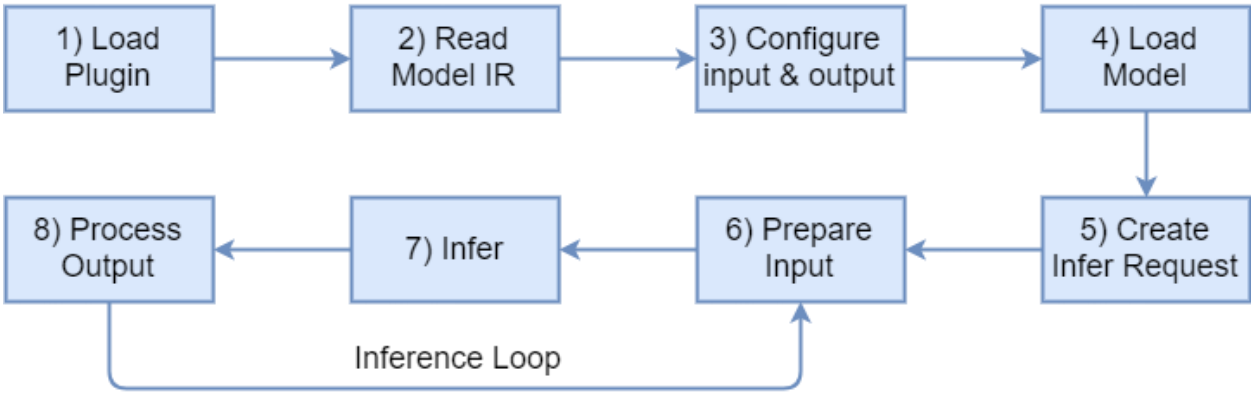


Figure 3: Integration process [3]

5.1. Load Plugin

Create Inference Engine Core to manage available devices and their plugins internally.

```
import cv2
import numpy as np
from openvino.inference_engine import IECore

print('Creating Inference Engine')
ie = IECore()
```

5.2. Read Model IR

Read a model IR created by the Model Optimizer.

```
model="/root/jupyter_root/public/ssd_mobilenet_v1_coco/FP32/ssd_mobilenet_v1_coco.xml"
print(f'Reading the network: {model}')
net = ie.read_network(model)

if len(net.input_info) != 1:
    print('The sample supports only single input topologies')

if len(net.outputs) != 1 and not ('boxes' in net.outputs or 'labels' in net.outputs):
    print('The sample supports models with 1 output or with 2 with the names "boxes" and
"labels"')
```

5.3. Configure Input & Output

The information about the input and output layers of the network is stored in the loaded neural network object **net**, and we need to obtain the information about the input and output layers and set the inference execution accuracy of the network by the following two parameters.

- input_info
- outputs

```
print('Configuring input and output blobs')
# Get name of input blob
input_blob = next(iter(net.input_info))

# Set input and output precision manually
net.input_info[input_blob].precision = 'U8'

if len(net.outputs) == 1:
    output_blob = next(iter(net.outputs))
    net.outputs[output_blob].precision = 'FP32'
else:
    net.outputs['boxes'].precision = 'FP32'
    net.outputs['labels'].precision = 'U16'
```

5.4. Load Model

Load the model to the device using

- InferenceEngine::Core::LoadNetwork()

```
print('Loading the model to the plugin')
exec_net = ie.load_network(network=net, device_name="CPU")
```

5.5. Create Inference Request and Prepare Input

To perform a neural network inference, we need to read the image from disk and bind it to the input blob. After loading the image, we need to determine the image size and layout format. For example, the default layout format of OpenCV is **CHW**, but the original layout of the image is **HWC**, so we need to modify the layout format and add the Batch size **N dimension**, then organize the image format according to NCHW for inferencing and resize the input image to the network input size.

```
# load_network() method of the IECore class with a specified number of requests (default 1)
returns an ExecutableNetwork
# instance which stores infer requests. So you already created Infer requests in the previous
step.

image_input = "pics/horse1.bmp"
original_image = cv2.imread(image_input)
image = original_image.copy()
_, _, net_h, net_w = net.input_info[input_blob].input_data.shape

if image.shape[:-1] != (net_h, net_w):
    print(f'Image {image_input} is resized from {image.shape[:-1]} to {(net_h, net_w)}')
    image = cv2.resize(image, (net_w, net_h))

# Change data layout from HWC to CHW
image = image.transpose((2, 0, 1))
# Add N dimension to transform to NCHW
image = np.expand_dims(image, axis=0)
```

5.6. Inference Calls

In this tutorial, here we use the synchronous API to demonstrate how to perform inference, calling:

- `InferenceEngine::InferRequest::Infer()`

If we want to improve the inference performance, we can also use the asynchronous API for inference as follows:

- `InferenceEngine::InferRequest::StartAsync()`
- `InferenceEngine::InferRequest::Wait()`

```
print('Starting inference in synchronous mode...')
res = exec_net.infer(inputs={input_blob: image})
print('Finished inference!')
```

5.7. Process the Output

After the inference engine inputs the graph and performs inference, a result is generated. The result contains a list of classes (class_id), confidence and bounding boxes. For each bounding box, the coordinates are given relative to the upper left and lower right corners of the original image. The correspondence between class_id and the labels file allow us to parse the text corresponding to the class, which is used to facilitate human reading comprehension.

```
# Generate a label list

labels="coco_91cl_bkgr.txt"
with open(labels, 'r') as f:
    labels = [line.split(',')[0].strip() for line in f]

output_image = original_image.copy()
h, w, _ = output_image.shape

if len(net.outputs) == 1:
    res = res[output_blob]
    # Change a shape of a numpy.ndarray with results ([1, 1, N, 7]) to get another
one ([N, 7]),
    # where N is the number of detected bounding boxes
    detections = res.reshape(-1, 7)
else:
    detections = res['boxes']
    labels = res['labels']
    # Redefine scale coefficients
    w, h = w / net_w, h / net_h

for i, detection in enumerate(detections):
    if len(net.outputs) == 1:
        _, class_id, confidence, xmin, ymin, xmax, ymax = detection
    else:
        class_id = labels[i]
        xmin, ymin, xmax, ymax, confidence = detection

    if confidence > 0.33:
        label = labels[int(class_id)]

        xmin = int(xmin * w)
        ymin = int(ymin * h)
        xmax = int(xmax * w)
        ymax = int(ymax * h)

        print(f'Found: label = {label}, confidence = {confidence:.2f}, '
              f'coords = ({xmin}, {ymin}), ({xmax}, {ymax})')
        # print(f'Found: class_id = {class_id}, confidence = {confidence:.2f}, '
        #       f'coords = ({xmin}, {ymin}), ({xmax}, {ymax})')
        # Draw a bounding box on a output image
        cv2.rectangle(output_image, (xmin, ymin), (xmax, ymax), (0, 255, 0), 2)

cv2.imwrite('out.bmp', output_image)
print('Image out.bmp created!')
```

5.8. Visualization of the Inference Results

```
from matplotlib import pyplot as plt
img = cv2.imread("./out.bmp")
img_cvt=cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.imshow(img_cvt)
plt.show()
```

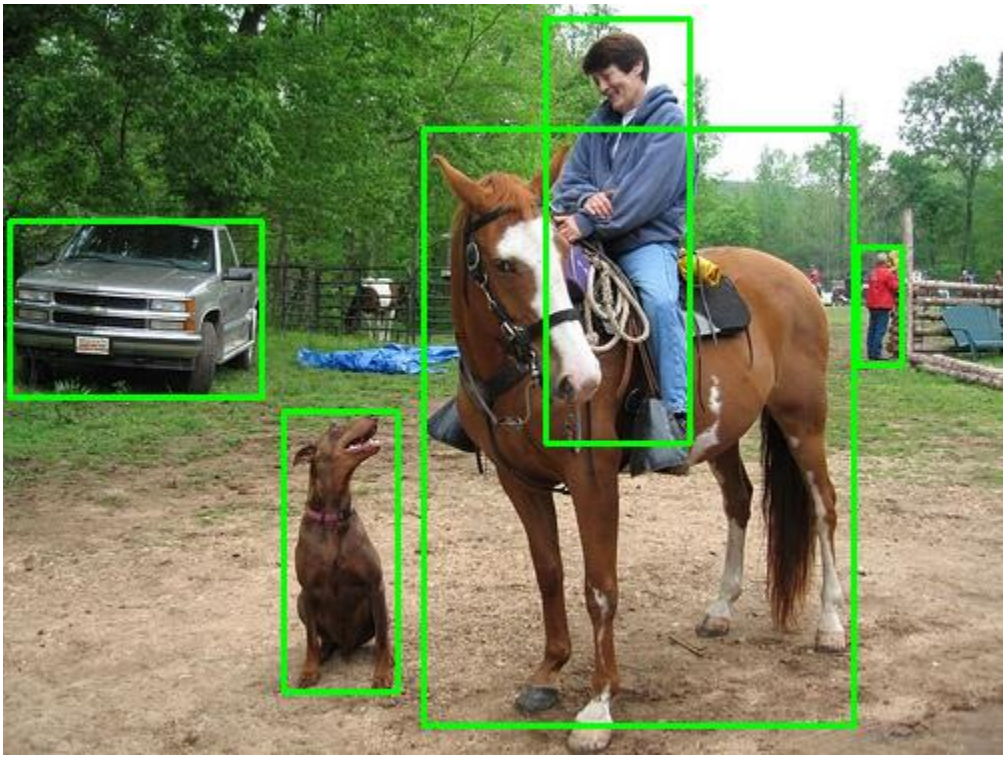


Figure 4: Image example

6. Practice Post-Training Optimization Tool

Post-training Optimization Tool (POT) is designed to accelerate the inference of deep learning models by applying special methods without model retraining or fine-tuning, liked post-training quantization. Therefore, the tool does not require a training dataset or a pipeline. To apply post-training algorithms from the POT, you need:

- A full precision model, FP32 or FP16, converted into the OpenVINO™ Intermediate Representation (IR) format
- A representative calibration dataset of data samples representing a use case scenario, for example, 300 images

The tool is aimed to fully automate the model transformation process without changing the model structure. The POT is available only in the Intel® Distribution of OpenVINO™ toolkit and is not open sourced. For details about the low-precision flow in OpenVINO™, see the Low Precision Optimization Guide.

Post-training Optimization Tool includes a standalone command-line tool and a Python* API that provide the following key features:

- Two post-training 8-bit quantization algorithms: fast DefaultQuantization and precise AccuracyAwareQuantization.
- Global optimization of post-training quantization parameters using the Tree-Structured Parzen Estimator.
- Symmetric and asymmetric quantization schemes. For details, see the Quantization section.

- Compression for different hardware targets such as CPU and GPU.
- Per-channel quantization for Convolutional and Fully-Connected layers.
- Multiple domains: Computer Vision, Recommendation Systems.
- Ability to implement a custom optimization pipeline via the supported API.

Before we start using the POT tool, we will need to prepare some config files:

- dataset files
- dataset definitions file: dataset_definitions.yml
- model json config for POT: ssd_mobilenetv1_int8.json
- model accuracy checker config: ssd_mobilenet_v1_coco.yml

6.1. Dataset Preparation

In this tutorial, we are using the dataset of Common Objects in Context (COCO) which the model was trained with this dataset. Please prepare the dataset according to Dataset Preparation Guide.

To download COCO dataset, you need to follow the steps below:

- Download 2017 Val images and 2017 Train/Val annotations
- Unpack archives

6.2. Global Dataset Configuration

If you want use definitions file in quantization via Post Training Optimization Toolkit (POT), you need to input the correct file path in these fields in the global dataset configuration file:

- annotation_file: [PATH_TO_DATASET]/instances_val2017.json
- data_source: [PATH_TO_DATASET]/val2017

6.3. Prepare Model Quantization and Configuration

We will need to create two config files to include model specific and dataset specific configurations to POT tool.

- ssd_mobilenetv1_int8.json
- ssd_mobilenet_v1_coco.yml

1. Create a new file and name it ssd_mobilenetv1_int8.json. This is the POT configuration file.

```
{
    "model": {
        "model_name": "ssd_mobilenet_v1_coco",
        "model": "ssd_mobilenet_v1_coco.xml",
        "weights": "ssd_mobilenet_v1_coco.bin"
    },
    "engine": {
        "config": "ssd_mobilenet_v1_coco.yml"
    },
    "compression": {
        "algorithms": [
            {
                "name": "DefaultQuantization",
                "params": {
                    "preset": "performance",
                    "stat_subset_size": 300
                }
            }
        ]
    }
}
```

2. Create a dataset config file and name it ssd_mobilenet_v1_coco.yml.

```
models:
  - name:  ssd_mobilenet_v1_coco
    launchers:
      - framework: dlsdk
        adapter: ssd

    datasets:
      - name: ms_coco_detection_91_classes
        preprocessing:
          - type: resize
            size: 300
        postprocessing:
          - type: resize_prediction_boxes
        metrics:
          - type: coco_precision
```

6.4. Quantize the Model

Now run the Accuracy checker tool and POT tool to create your quantized IR files.

```
accuracy_check -c ./calibration/ssd_mobilenet_v1_coco.yml

pot --config ./calibration/ssd_mobilenet_v1_coco.yml --output_dir ./INT8 --evaluate --log-
level INFO
```

6.5. Compare FP32 and INT8 Model Performance

This topic demonstrates how to run the Benchmark Python* Tool, which performs inference using convolutional networks. Performance can be measured for two inference modes: synchronous (latency-oriented) and asynchronous (throughput-oriented).

Upon start-up, the application reads command-line parameters and loads a network and images/binary files to the Inference Engine plugin, which is chosen depending on a specified device. The number of infer requests and execution approach depend on the mode defined with the `-api` command-line parameter.
(https://docs.openvinotoolkit.org/2020.4/openvino_inference_engine_tools_benchmark_tool_README.html)

Please run both of your FP32 and INT8 models on Benchmark Python* Tool and compare your results.

```
python3 /opt/intel/openvino_2021/deployment_tools/tools/benchmark_tool/benchmark_app.py \
    -m <Directory for FP32 and INT8
models>/ssd_mobilenet_v1_coco/INT8/ssd_mobilenet_v1_coco_int8.xml
```

Now that you have run both your FP32 and INT8 IRs, you can make a comparison of the performance gain you are achieving with INT8 IR files. See the official benchmark results for Intel® Distribution of OpenVINO™ Toolkit on various Intel® hardware settings.

7. Conclusion

This article describes an overview of Intel® Distribution of OpenVINO™ Toolkit with how to get started guides and using the power of vector neural network instructions (VNNI) and Intel® Advanced Vector Extensions (AVX512) with low precision inference workloads. The steps and codes shown are aimed at modifying the similar type of workloads to help you leverage these tutorials. You can also clearly see the performance boost of using these methodologies on this official benchmark result page.

8. Additional Information

- Jupyter* Notebook Tutorials - sample application Jupyter* Notebook tutorials
- Intel® Distribution of OpenVINO™ toolkit Main Page - learn more about the tools and use of the Intel® Distribution of OpenVINO™ toolkit for implementing inference on the edge

9. References

1. Typical OpenVINO™ workflow from https://docs.openvinotoolkit.org/latest/index.html#openvino_toolkit_components on 8/4/21
2. Typical workflow for deploying a trained deep learning model from https://docs.openvinotoolkit.org/2021.3/openvino_docs_MO_DG_Deep_Learning_Model_Optimizer_DevGuide.html on 8/5/21
3. *Integration process* from https://docs.openvinotoolkit.org/latest/openvino_docs_IE_DG_Deep_Learning_Inference_Engine_DevGuide.html on 8/5/21

10. Feedback

We value your feedback. If you have comments (positive or negative) on this guide or are seeking something that is not part of this guide, please reach out and let us know what you think.

License, Notices & Disclaimers

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that code included in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD.

Intel technologies may require enabled hardware, software or service activation.
No product or component can be absolutely secure.
Your costs and results may vary.
Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks.
The products described 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.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.