# Optimization Guide

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>OpenVINO™ Toolkit Workflow</td>
<td>2</td>
</tr>
<tr>
<td>OpenVINO™ Toolkit Components</td>
<td>4</td>
</tr>
<tr>
<td>Installation Guides</td>
<td>5</td>
</tr>
<tr>
<td>Get Started with OpenVINO™ Model Zoo</td>
<td>5</td>
</tr>
<tr>
<td>OpenVINO™ Model Optimizer</td>
<td>7</td>
</tr>
<tr>
<td>Practice Inference Engine API</td>
<td>10</td>
</tr>
<tr>
<td>Load Plugin</td>
<td>11</td>
</tr>
<tr>
<td>Read Model IR</td>
<td>11</td>
</tr>
<tr>
<td>Configure Input and Output</td>
<td>11</td>
</tr>
<tr>
<td>Load Model</td>
<td>12</td>
</tr>
<tr>
<td>Create Inference Request and Prepare Input</td>
<td>12</td>
</tr>
<tr>
<td>Inference Calls</td>
<td>13</td>
</tr>
<tr>
<td>Process the Output</td>
<td>13</td>
</tr>
<tr>
<td>Visualization of the Inference Results</td>
<td>15</td>
</tr>
<tr>
<td>Practice Post-Training Optimization Tool (POT)</td>
<td>15</td>
</tr>
<tr>
<td>Dataset Preparation</td>
<td>16</td>
</tr>
<tr>
<td>Quantize the Model</td>
<td>18</td>
</tr>
<tr>
<td>Compare FP32 and INT8 Model Performance</td>
<td>18</td>
</tr>
<tr>
<td>Conclusion</td>
<td>19</td>
</tr>
<tr>
<td>Additional Information</td>
<td>19</td>
</tr>
<tr>
<td>References</td>
<td>19</td>
</tr>
</tbody>
</table>
1. Introduction

This guide is for users who are already familiar with Intel® Distribution of OpenVINO™ toolkit. It provides hardware and software configurations that provide the best performance for most situations. However, please carefully consider these settings for their specific scenarios since Intel® Distribution of OpenVINO™ toolkit can be deployed in multiple ways.

OpenVINO™ toolkit is a comprehensive toolkit based on AI and deep learning inference for quickly developing applications to emulate human vision, automate speech recognition, process natural language, create recommendation systems, and more including:

- Enabling Convolutional Neural Networks (CNN) on edge devices for deep learning
- Supporting heterogeneous execution across an Intel® CPU, Intel® Integrated Graphics, Intel® Neural Compute Stick 2 and Intel® Vision Accelerator Design with Intel® Movidius™ VPUs
- Speeding time-to-market via an easy-to-use library of computer vision functions and pre-optimized kernels. OpenVino also includes optimized calls for computer vision standards, including OpenCV* and OpenCL®

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

OpenVINO™ Toolkit Workflow

The following diagram illustrates the typical OpenVINO™ workflow:
Figure 1. Typical OpenVINO™ workflow [1]
OpenVINO™ Toolkit Components

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

- **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.

- **Optimizing models post-training** - 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 Python* Tool
  - 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:
    - Deep Learning accuracy validation framework
    - Model Downloader and other automation tools
  - Overview of OpenVINO™ Toolkit Pre-Trained Models - Documentation for pretrained models that are available in the Open Model Zoo repository.

Intel® DL Streamer is a streaming analytics framework, based on GStreamer, for constructing graphs of media analytics components. Intel® DL Streamer can be installed by the Intel® Distribution of OpenVINO™ toolkit installer. Its open source version is available on Github. For more information see Intel® DL Streamer documentation.

**OpenCV** - OpenCV* community version compiled for Intel® hardware

**Intel® Media SDK** in Intel® Distribution of OpenVINO™ toolkit for Linux only
For building the Inference Engine from the source code, see the build instructions.

2. Installation Guides

Before installation review the Target System Platform requirements before installation. Install OpenVINO™ and configure the third-party dependencies based on your preference.

OS Based:

Install OpenVINO™Runtime on Linux, Windows, MacOS, or Raspbian OS

Install from Images or Repositories:

To install from Anaconda Cloud, APT, Docker, PyPI, Yocto, or the OpenVINO™Runtime using an installer see Install and Configure Intel® Distribution of OpenVINO™ toolkit for Linux

3. Get Started with OpenVINO™ Model Zoo

The Open Model Zoo in the Intel® Distribution of OpenVINO™ includes optimized deep learning models and a set of demos for developing high-performance deep learning inference applications. You can use free pre-trained models to speed-up development and deployment. To check the currently available models, use the Model Downloader. It is a set of python scripts for browsing and downloading these pre-trained models. Other automation tools are also available:

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

Use the following instructions to run downloader.py. This example runs on a Linux* machine with source installation.
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:
--help show this help message and exit
--name PAT[,PAT...] download only models whose names match at least one of specified patterns
--list FILE.LST download only models whose names match at least one of 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 for saving 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
Use the parameter --printall to see which pre-trained models are available for download in the current version of OpenVINO. The following example is a classic computer vision network used to detect a target picture. Download ssdmobilenetv1coco using Model Downloader.

```bash
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. For more information see the Model Optimizer Development Guide.

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

![Typical workflow for deploying a trained deep learning model](image.png)

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

Network topology files are described in .xml format. Weights and biases are stored in binary format in .bin files.

To be able to convert ssdmobilenetv1coco model into IR, some model specific parameters must be provided to the Model Optimizer. A .yml file with model specific information was downloaded when the model was downloaded from Open Model Zoo. Here is an example for ssdmobilenetv1coco:
The Model Downloader also contains another handy script 'converter.py' for accurately inputting the parameters of the downloaded model to the Model Optimizer (MO). Use this script for model conversion to reduce the workload.

```bash
python3 
/opt/intel/openvino_2021/deployment_tools/open_model_zoo/tools/downloader/convert
er.py 
--download_dir=. 
--output_dir=. 
--name=ssd_mobilenet_v1_coco 
--dry_run
```

You can choose to convert the model using "converter.py" or use the Model Optimizer execution parameters that were generated by the command above and add them as input parameters shown below:

```bash
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_support.json 
--tensorflow_object_detection_api_pipeline_config=public/ssd_mobilenet_v1_coco/ssd_mobilenet_v1_coco_2018_01_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
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 files using the Model Optimizer, use the Inference Engine to infer the result for a given input data. The Inference Engine is a C++ 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.

The Inference Engine is a software component that contains the complete implementation for inference on various Intel® hardware devices: CPU, GPU, VPU, FPGA, etc. Each of these plugins implements the unified API along with hardware-specific APIs. The following figure show the steps of the integration process:
Figure 3. **Integration process** [3]

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

```python
import xml
import numpy as np
from openvino.inference_engine import IECore

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

**Read Model IR**
Read a model IR created by the Model Optimizer.

```python
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 or (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"')
```

**Configure Input and Output**
The information about the input and output layers of the network is stored in the loaded neural network object net. Use the following two parameters to obtain the information about the ef layers and set the inference execution accuracy of the network:
• input_info
• outputs

```python
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'
```

**Load Model**

Load the model to the device using:

InferenceEngine::Core::LoadNetwork()

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

**Create Inference Request and Prepare Input**

To perform a neural network inference, read the image from disk and bind it to the input blob. After loading the image, determine the size of the image and format of the layout. For example, the default layout format of OpenCV is CHW, but the original layout of the image was HWC. Modify the layout format and add the Batch size N dimension, then organize the image format according to NCHW and resize the input image to the network input size.
Inference Calls
This example uses the synchronous API to demonstrate how to perform inference, calling:

- InferenceEngine::InferRequest::Infer()

To improve the inference performance, use the asynchronous API for inference as follows:

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

Process the Output
After the inference engine inputs the graph and performs inference, a result is generated. The result contains a list of classes (classid), 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 classid 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(' ', 1)[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!')
Visualization of the Inference Results

```python
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 (POT)

POT is designed to accelerate the inference of deep learning models by applying special methods without model retraining or fine-tuning, like 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. For details about the low-precision flow in OpenVINO™, see Optimizing models post-training.

- 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
- 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: ssdmobilenetv1int8.json
- model accuracy checker config: ssdmobilenetv1_coco.yml

**Dataset Preparation**

This example shows the dataset of Common Objects in Context (COCO). The model was trained with this dataset. Prepare the dataset according to Dataset Preparation Guide.

Download COCO dataset:

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

**Global Dataset Configuration**

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

- annotationfile: [PATH_TO_DATASET]/instancesval2017.json
- datasource: [PATH_TO_DATASET]/val2017
Prepare Model Quantization and Configuration

Create two config files. One to include model specific and another for the dataset specific configurations.

- `ssdmobilenetv1int8.json`
- `ssdmobilenetv1_coco.yml`

Follow these steps:

1. Create a new file and name it `ssdmobilenetv1int8.json`. This is the POT configuration file.

   ```json
   {
     "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 `ssdmobilenetv1_coco.yml`
Quantize the Model
Run the Accuracy checker tool and POT tool to create your quantized IR files.

```
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
```

Compare FP32 and INT8 Model Performance
This 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. For more information see OpenVINO™ Inference Engine Tools Benchmark Tool README

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

```
   -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 guide provides recommendations for tuning The Intel® Distribution of OpenVINO™ Toolkit using the power of vector neural network instructions (VNNI) and Intel® Advanced Vector Extensions (AVX512) with low precision inference workloads. You can use the steps and code to modify similar types of workloads so that you see a performance boost. Check out the this official benchmark result page for more examples.

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


Notices & Disclaimers

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.