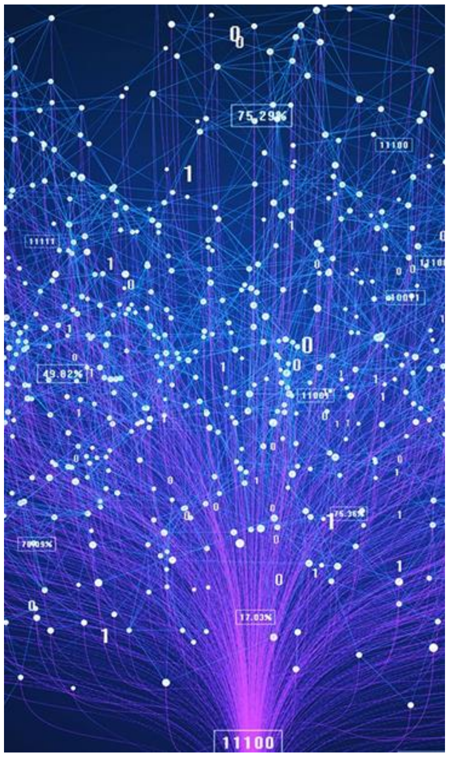
Optimization Guide

Intel® Deep Learning Boost
Intel® Xeon® Scalable Processor

Deep Learning with Intel® AVX512 and Intel® Deep Learning Boost Tuning Guide on 3rd Generation Intel® Xeon® Scalable Processors



Contents

Revision Record

Date	Rev.	Description
04/06/2021	1.0	Initial public release.

1. Overview

This user guide is intended to explain how the 3rd Gen Intel® Xeon® Scalable Processor platform ((codename Ice Lake/Whitley) is used for machine learning and deep learning-related tasks. Executing machine learning and deep learning workloads on the Intel® Xeon® Scalable Processor platform has the following advantages:

- The platform is very suitable for processing memory-intensive workloads and 3D-CNN topologies used in medical imaging, GAN, seismic analysis, genome sequencing, etc.
- The simple **numactl** command can be used for flexible core control; it is still very suitable for real-time inference even when the number of batches is small.
- It is supported by a powerful ecosystem and can be used for distributed training (such as for computations directly at the data source) directly on large-scale clusters. This avoids the additional costs for large storage capacity and expensive cache mechanisms that are usually required for the training of scaled architecture.
- Multiple types of workloads (HPC/BigData/AI) are supported on the same cluster to achieve better TCO.
- It satisfies the computing requirements in many real deep learning applications via SIMD acceleration.
- The same infrastructure can be used directly for training and inference.

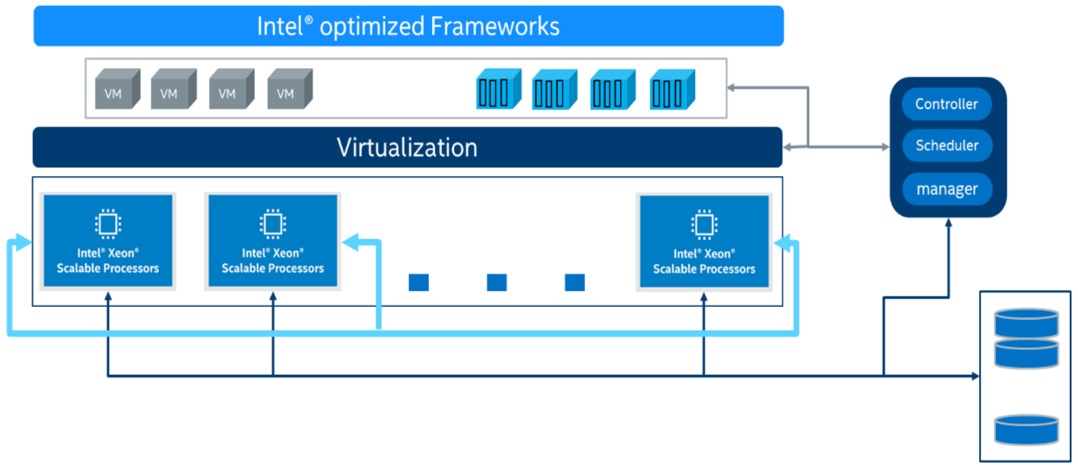
The development and deployment of typical deep learning applications involve the following stages:

Dataset preparation
Model training and tuning
Model optimization and migrated learning
Model deployment

These different stages require the allocation of following resources, and choosing the right resources can greatly accelerate the efficiency of your AI services:

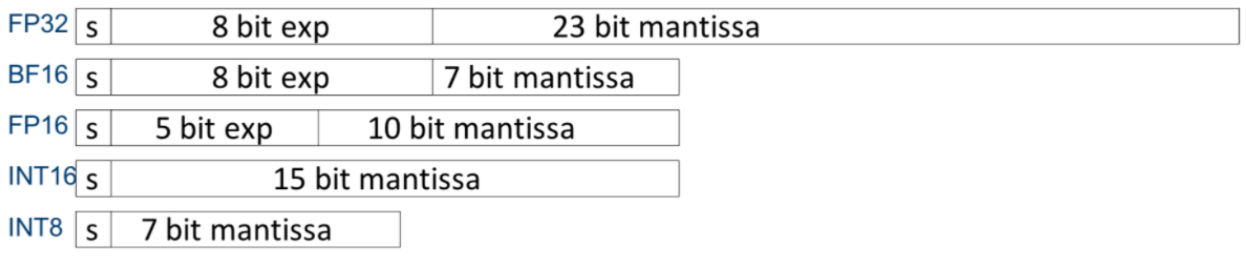
- Computational power
- Memory
- Storage for datasets
- Communication link between compute nodes
- Optimized software

All the processes including dataset preparation, model training, model optimization, and model deployment, can be done on an Intel® Xeon® Scalable Processor platform-based infrastructure which also supports machine learning/deep learning platforms for training and inference. The proposed infrastructure is shown in the figure below:

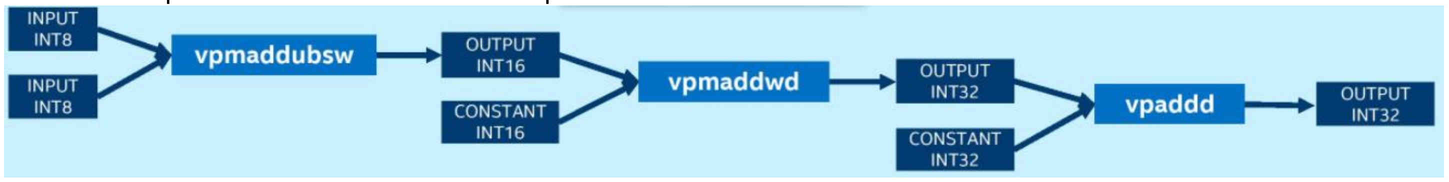

Intel® optimized Frameworks
VM
VM
VM
VM
Virtualization
Intel® Xeon® Scalable Processors
Intel® Xeon® Scalable Processors
Intel® Xeon® Scalable Processors
Controller
Scheduler
manager

2. Introducing Intel® AVX-512 and Intel® Deep Learning Boost

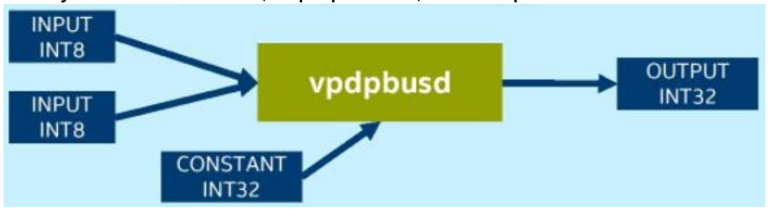
Intel® Advanced Vector Extensions 512 (Intel® AVX-512) is a “single instruction, multiple data” (SIMD) instruction set based on x86 processors. Compared to traditional “single instruction, single data” instructions, a SIMD instruction allows for executing multiple data operations with a single instruction. As the name implies, Intel® AVX-512 has a register width of 512 bytes, and it supports 16 32-byte single-precision floating-point numbers or 64 8-byte integers. Intel® Xeon® Scalable Processors support multiple types of workloads, including complex AI workloads, and improve AI computation performance with the use of Intel® Deep Learning Boost (Intel® DL Boost). Intel Deep Learning Boost includes Intel® AVX-512 VNNI (Vector Neural Network Instructions) which is an extension to the Intel® AVX-512 instruction set. It can combine three instructions into one for execution, which further unleashes the computing potential of next-generation Intel® Xeon® Scalable Processors and increases the inference performance of the INT8 model. Both 2nd-Generation and 3rd-Generation Intel® Xeon® Scalable Processors support VNNI.



Platforms not using VNNI require the **vpmaddubsw, vpmaddwd** and **vpaddd** instructions to complete the multiply-accumulate operations in INT8 convolution operation:



Platforms using VNNI require only one instruction, “vpdpbusd”, to complete the INT8 convolution operation:



3. BIOS Settings and Hardware Configurations

3.1. BIOS Settings

The configuration items that can be optimized in BIOS and their recommended values are as follows:

Configuration item	Recommended value
Hyper-Threading	Enable
SNC (Sub NUMA)	Disable
Boot performance mode	Max Performance
Turbo Mode	Enable
Hardware P-State	Native Mode

3.2. Recommended Hardware Configurations

Machine learning workloads, and in particular deep learning workloads, are usually used for compute-intensive applications. Hence, they require a selection of suitable types of memory, CPU, hard drives, and other computing resources to achieve optimal performance. In summary, the following common configurations are recommended:

Memory configuration

The utilization of all memory channels is recommended so that the bandwidth of all memory channels can be utilized.

CPU configuration

FMA, the Intel AVX-512 acceleration module in Intel processors, is an important component in unleashing computational performance, and artificial intelligence-related workloads are usually part of compute-intensive applications. In order to achieve better computing performance, it is recommended to use the Intel Xeon® Scalable Processors Gold 6 series (or above) which have two Intel AVX512 computational modules per core.

Network configuration

If cross-node training clusters are required, then it is recommended to choose high-speed networking such as 25G/100G networks for better scalability.

Hard drive configuration

For high IO efficiency for workloads, SSDs and drives with higher read and write speeds are recommended.

4. Linux System Optimization

4.1. OpenMP Parameter Settings

The recommended configuration for the main parameters is as follows:

```
• OMP_NUM_THREADS = “number of cpu cores in container”
• KMP_BLOCKTIME = 1 or 0 (set according to actual type of model)
• KMP_AFFINITY=granularity=fine, verbose, compact,1,0
```

4.2. Number of CPU cores

The main impact of the number of CPU cores on inference performance is as follows:

• When **batchsize** is small (in online services for instance), the increase in inference throughput gradually weakens as the number of CPU cores increases; in practice, 8-16 CPU cores is recommended for service deployment depending on the model used.

• When **batchsize** is large (in offline services for instance), the inference throughput can increase linearly as the number of CPU cores increases; in practice, more than 20 CPU cores is recommended for service deployment.

```
# taskset -C xxx-xxx –p pid (limits the number of CPU cores used in service)
```

4.3. Impact of NUMA Configuration

For NUMA-based servers, there is usually a 5-10% increase in performance when configuring NUMA on the same node compared to using it on different nodes.

```
#numactl -N NUMA_NODE -l command args ...  (controls NUMA nodes running in service)
```

4.4. Configuration of Linux Performance Governor

Performance: As the name suggests, efficiency is the only consideration and the CPU frequency is set to its peak to achieve the best performance.

```
# cpupower frequency-set -g performance
```

4.5. CPU C-States Settings

CPU C-States: To reduce power consumption when the CPU is idle, the CPU can be placed in the low-power mode. There are several power modes available for each CPU which are collectively referred to as C-states or C-modes. Disabling C-States can increase performance.

```
#cpupower idle-set -d 2,3
```

5. Using Intel® Optimization for TensorFlow* Deep Learning Framework

TensorFlow* is one of the most popular deep learning frameworks used in large-scale machine learning (ML) and deep learning (DL) applications. Since 2016, Intel and Google* engineers have been working together to use Intel® oneAPI Deep Neural Network Library (Intel® oneDNN) to optimize TensorFlow* performance and accelerate its training and inference performance on the Intel® Xeon® Scalable Processor platform.

5.1. Deploying Intel® Optimization for TensorFlow* Deep Learning Framework

Reference: https://www.intel.com/content/www/us/en/develop/articles/intel-optimization-for-tensorflow-installation-guide.html
Step 1: Install a Python3.x environment. Here is an example to illustrate how to build Python* 3.6 with Anaconda*
Reference: https://www.anaconda.com/products/individual
Download and install the latest version of Anaconda

```
# wget https://repo.anaconda.com/archive/Anaconda3-2020.02-Linux-x86_64.sh
# sh Anaconda3-2020.02-Linux-x86_64.sh
# source /root/.bashrc
# conda install python=3.6  (create a Python3.6 environment)
#(base) [root@xx]# python -V
Python 3.6.10
```

Step 2: Install the Intel optimation for TensorFlow*: intel-tensorflow.
Install the latest version (2.x)

```
# pip install intel-tensorflow
```

If you need to install tensorflow1.x, we recommend installing the following version to take advantage of the performance acceleration on the 3rd Gen Intel® Xeon® Scalable Processor platform:

```
# pip install https://storage.googleapis.com/intel-optimized-tensorflow/intel_tensorflow-
1.15.0up2-cp36-cp36m-manylinux2010_x86_64.whl
```

Step 3: Set run-time optimization parameters.
Reference:
https://github.com/IntelAI/models/blob/master/docs/general/tensorflow/GeneralBestPractices.md
Usually, the following two methods are used for inference, which use different optimization settings

Batch inference: Batch Size >1, measures the number of input tensors that can be processed per second. Usually, all the physical cores in the same CPU socket can be used for batch inference to achieve the best performance.
On-line inference (also known as real-time inference): Batch Size=1, a measure of time needed to process one input tensor (when the batch size is 1). In real-time inference, multiple instances are run concurrently to achieve the optimal throughput.
1: Obtaining the number of physical cores in the system:
To confirm the current number of physical cores, we recommend using the following command:

```
# lscpu | grep "Core(s) per socket" | cut -d':' -f2 | xargs
```

In this example, we assume 8 physical cores.
2: Setting optimization parameters:
Optimization parameters are configured using the two following methods. Please choose the configuration method according to your needs.
Method 1: Configure environment parameters directly:

```
export OMP_NUM_THREADS=physical cores
export KMP_AFFINITY="granularity=fine,verbose,compact,1,0"
export KMP_BLOCKTIME=1
export KMP_SETTINGS=1
```

Method 2: Add environment variables in the Python code that is running:

```
import os
os.environ["KMP_BLOCKTIME"] = "1"
os.environ["KMP_SETTINGS"] = "1"
os.environ["KMP_AFFINITY"]= "granularity=fine,verbose,compact,1,0"
if FLAGS.num_intra_threads > 0:
  os.environ["OMP_NUM_THREADS"]= # <physical cores>
config = tf.ConfigProto()
config.intra_op_parallelism_threads = # <physical cores>
config.inter_op_parallelism_threads = 1
tf.Session(config=config)
```

5.2. Inferencing using Intel® Optimization for TensorFlow* DL Model with FP32/INT8 support

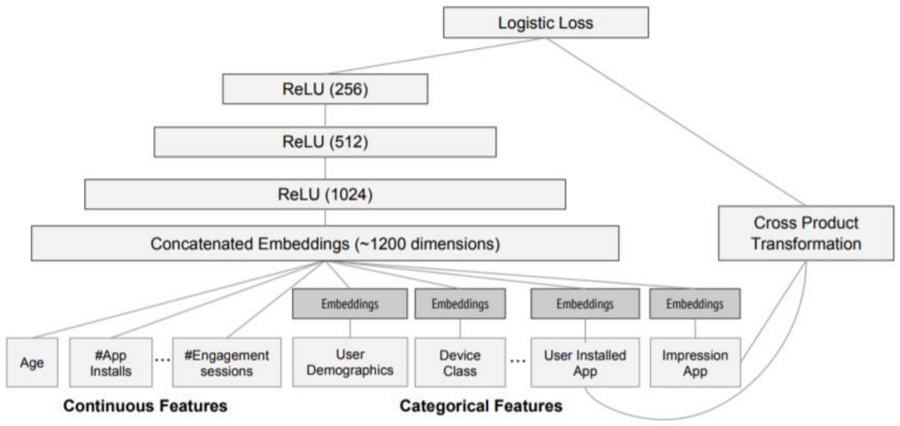
This section mainly explains how to run the inference benchmark on ResNet50. You can refer to the following reference to inference using your machine learning/deep learning model.
Reference: https://github.com/IntelAI/models/blob/master/docs/image_recognition/tensorflow/Tutorial.md
Taking inference benchmarking for ResNet50* as an example, FP32, BFloat16, and Int8 are supported for model inference.
Reference:
https://github.com/IntelAI/models/blob/master/benchmarks/image_recognition/tensorflow/resnet50v1_5/README.md
FP32-based model inference:
https://github.com/IntelAI/models/blob/master/benchmarks/image_recognition/tensorflow/resnet50v1_5/README.md#fp32-inference-instructions
INT8-based model inference:
https://github.com/IntelAI/models/blob/master/benchmarks/image_recognition/tensorflow/resnet50v1_5/README.md#int8-inference-instructions

5.3. Training using Intel® Optimization for TensorFlow* DL Model with FP32/ INT8 Support

This section mainly explains how to run a training benchmark on ResNet50. You can refer to the following reference to run your machine learning/deep learning model training.
FP32-based training:
https://github.com/IntelAI/models/blob/master/benchmarks/image_recognition/tensorflow/resnet50v1_5/README.md#fp32-training-instructions

5.4. Applications – Inferencing and Training Using Intel Optimized TensorFlow Wide & Deep Model

Among the many operations in the data center, it is a typical application to use recommendation systems to match users with content they are interested in. Recommendation system is a type of information filtering system that learns about users' interests according to their profiles and past behavior records and predict their ratings or preferences for a given item. It changes the way a business communicates with users and enhances the interaction between the business and its users.
When using deep learning, we find, from a large amount of complex raw data, the deep interactions between features that are difficult to be expressed with traditional machines using artificial feature engineering. Related study outcomes include Wide & Deep, DeepFM, FNN, DCN, and other models.
Using the Wide & Deep model as an example, the core idea is to take advantage of both the memorization capability of a linear model and the generalization capability of the DNN model and optimize the parameters in these models at the same time during training. This will result in better overall model prediction capabilities. Its structure is shown in the figure below:



Wide
"Wide" is a generalized linear model, and its inputs mainly include original and interactive features. We can use cross-product transformation to build the interactive features of K-group:

$$\phi_k(\mathbf{x}) = \prod_{i=1}^{d} x_i^{c_{ki}} \quad c_{ki} \in \{0,1\}$$

Deep
"Deep" is a DNN model, and the calculation for each layer is as follows:

$$a^{(l+1)} = f\left(W^{(l)}a^{(l)} + b^{(l)}\right)$$

Co-training
The Wide & Deep model uses co-training instead of integration. The difference is that co-training shares a loss function, then updates the parameters in either part of the model at the same time, while integration trains N models independently and fuses them together afterwards. Therefore, the output of the model is:

$$P(Y=1|\mathbf{x}) = \sigma\left(\mathbf{w}_{wide}^T[\mathbf{x}, \phi(\mathbf{x})] + \mathbf{w}_{deep}^T a^{(l_f)} + b\right)$$

The above is the background information on the Wide & Deep model. Next, we will describe how to run inference benchmarking for the Wide & Deep model.
Reference:
https://github.com/IntelAI/models/blob/master/docs/recommendation/tensorflow/Tutorial.md
Dataset preparation:
https://github.com/IntelAI/models/tree/master/benchmarks/recommendation/tensorflow/wide_deep_large_ds#Prepare-dataset
FP32-based model inference:
https://github.com/IntelAI/models/tree/master/benchmarks/recommendation/tensorflow/wide_deep_large_ds#fp32-inference-instructions
INT8-based model inference:

https://github.com/IntelAI/models/tree/master/benchmarks/recommendation/tensorflow/wide_deep_large_ds#int8-inference-instructions
FP32-based training:https://github.com/IntelAI/models/tree/master/benchmarks/recommendation/tensorflow/wide_deep_large_ds#fp32-training-instructions

5.5. Intel® Math Kernel Library (MKL) Threadpool-Based TensorFlow (Optional)

Starting with TensorFlow 2.3.0, a new feature has been added. You can choose Eigen Threadpool for TensorFlow multi-threading support instead of OpenMP, by using the compiling option `--config=mkl_threadpool` instead of --config=mkl, when compiling the Tensorflow source code.
If the user wants to try this feature with TensorFlow 1.15, they need to download the source code that has been ported and optimized by Intel and compile it (it should be particularly pointed out that **Bazel* 0.24.1** needs to be installed for the purpose):

```
# git clone https://github.com/Intel-tensorflow/tensorflow.git
# git checkout -b tf-1.15-maint remotes/origin/tf-1.15-maint
# bazel --output_user_root=$BUILD_DIR build --config=mkl_threadpool -c opt --copt=-O3
//tensorflow/tools/pip_package:build_pip_package
bazel-bin/tensorflow/tools/pip_package/build_pip_package $BUILD_DIR
```

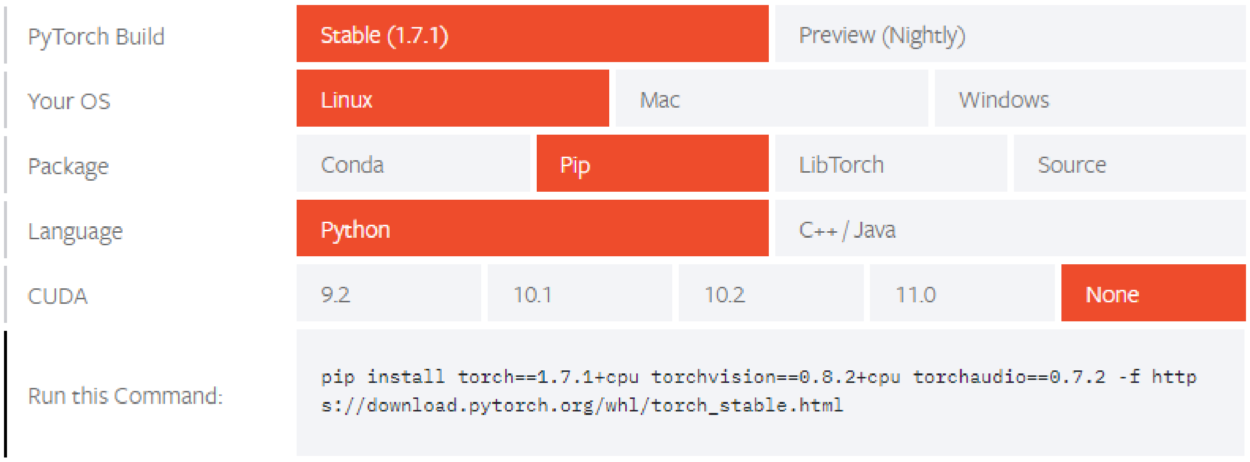
After successfully completing the steps above, the TensorFlow ***wheel*** file can be found under the ***$BUILD_DIR*** path. For example: ***tensorflow-1.15.0up2-cp36-cp36m-linux_x86_64.whl***. The installation steps are as follows:

```
# pip uninstall tensorflow
# pip install $BUILD_DIR/<filename>.whl --user
```

6. Using PyTorch*, a Deep Learning Framework

6.1. Deploying PyTorch

Reference:
https://software.intel.com/content/www/us/en/develop/articles/getting-started-with-intel-optimization-of-pytorch.html
Environment: Python3.6 or above
Step 1: Visit the official PyTorch website: https://pytorch.org/
Step 2: Select CPU
Currently, Intel oneDNN is integrated into the official version of PyTorch, so there is no need for additional installation to have accelerated performance on the Intel® Xeon® Scalable Processor platform. Select “None” for CUDA. See the figure below for details.



PyTorch Build	[✓ Stable (1.7.1)]	Preview (Nightly)			
Your OS	[✓ Linux]	Mac	Windows		
Package	Conda	[✓ Pip]	LibTorch	Source	
Language	[✓ Python]	C++ / Java			
CUDA	9.2	10.1	10.2	11.0	[✓ None]

Run this Command:

```
pip install torch==1.7.1+cpu torchvision==0.8.2+cpu torchaudio==0.7.2 -f http
s://download.pytorch.org/whl/torch_stable.html
```

Step 3: Installation

```
pip install torch==1.7.1+cpu torchvision==0.8.2+cpu torchaudio==0.7.2 -f
https://download.pytorch.org/whl/torch_stable.html
```

6.2. Optimization Recommendations for Training and Inferencing PyTorch-based Deep Learning Models

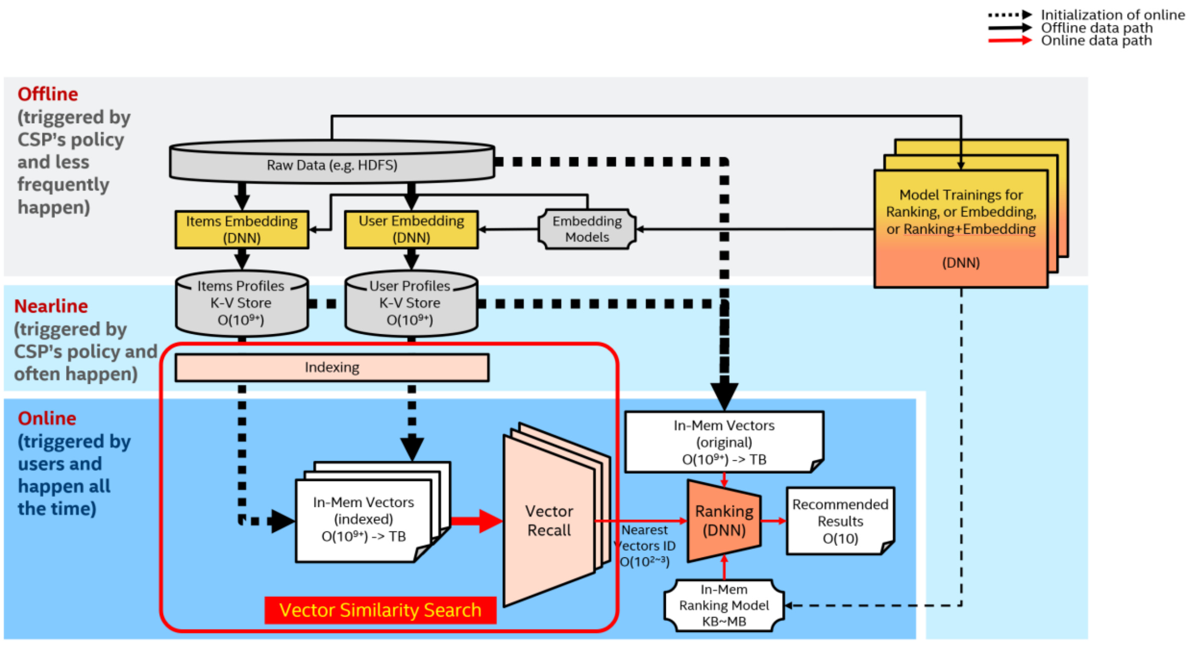
You may refer to the following website to learn more about optimization parameter settings for PyTorch* on the Intel® Xeon® Scalable Processor platform.
Reference: https://software.intel.com/content/www/us/en/develop/articles/how-to-get-better-performance-on-pytorchcaffe2-with-intel-acceleration.html

6.3. Introducing and Using Intel® Extension for PyTorch

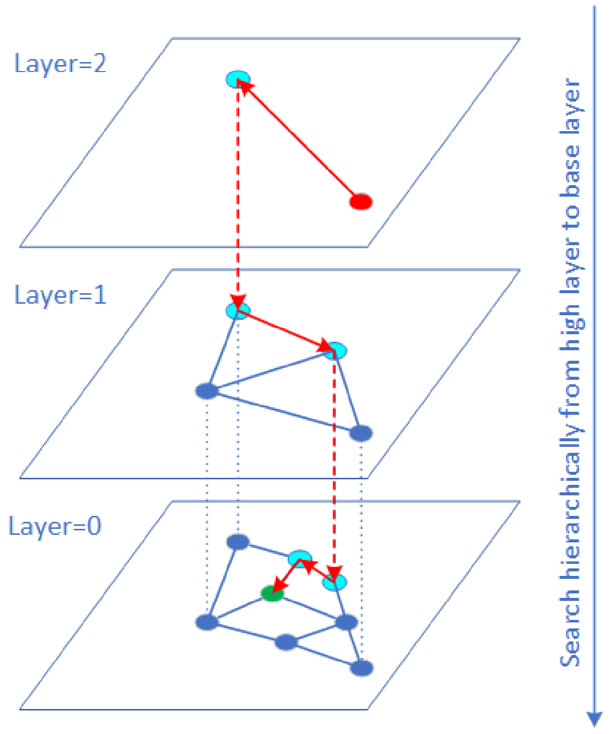
Intel® Extension for PyTorch is a Python extension of PyTorch that aims to improve the computational performance of PyTorch on Intel® Xeon® Processors. Not only does this extension includes additional functions, but it also provides performance optimizations for new Intel hardware.
The Github links to the Intel Extension for PyTorch are:
https://github.com/intel/intel-extension-for-pytorch
https://github.com/oneapi-src/oneAPI-samples/tree/master/AI-and-Analytics/Features-and-Functionality/IntelPyTorch_Extensions_AutoMixedPrecision

7. Accelerating Vector Recall in the Recommendation System with Intel® Deep Learning Boost VNNI

A problem that needs to be resolved in the recommendation system is how to generate a recommendation list with the length of K for a given user that matches their interests and needs as much as possible (high accuracy) and as fast as possible (low latency)? Conventional recommendation systems include two components: vector recall and ranking. “Vector recall” roughly filters out hundreds or thousands of items from a huge recommendation pool that will most likely interest the user, passes the results on to the ranking module for further sorting before the final recommendation results are obtained.


Initialization of online
Offline data path
Online data path
Offline
(triggered by CSP's policy and less frequently happen)
Raw Data (e.g. HDFS)
Items Embedding (DNN)
User Embedding (DNN)
Embedding Models
Model Trainings for Ranking, or Embedding, or Ranking+Embedding (DNN)
Nearline
(triggered by CSP's policy and often happen)
Items Profiles K-V Store O(10⁹⁺)
User Profiles K-V Store O(10⁹⁺)
Indexing
Online
(triggered by users and happen all the time)
In-Mem Vectors (indexed) O(10⁹⁺) -> TB
Vector Recall
Nearest Vectors ID O(10²⁻³)
In-Mem Vectors (original) O(10⁹⁺) -> TB
Ranking (DNN)
Recommended Results O(10)
In-Mem Ranking Model KB~MB
Vector Similarity Search

Vector recall can be converted into a high-dimensional vector similarity search problem.
The Hierarchical Navigable Small World (HNSW) algorithm is a type of Approximate Nearest Neighbor (ANN) vector similarity search algorithm based on graph structures. It is also one of the fastest and most precise algorithms.



Usually, the data type of the raw vector data is FP32. For many applications (such as image search), vector data can be expressed in INT8/INT6 and the impact of quantization error on the final search result is limited. The "VNNI intrinsic" instruction can be used for inner product calculations for INT8/INT6 vectors. Many experiments have shown that QPS Performance has greatly improved, and that recall rate remains virtually unchanged. The reason for the improvement in QPS performance is that the memory–bandwidth ratio for INT8/INT16 is smaller than for FP32, and VNNI instructions accelerate the distance calculations in addition.
Currently, optimized source code is implemented based on the HNSWLib[10] open source project. We have already ported it to the Faiss[9] framework, which is widely used in the industry.
To achieve the optimal performance, the following deployment steps are recommended:

1. Bind NUMA
2. Each physical CPU core executes a single query process

Reference command (using 1 socket and 24 cores as an example):

```
# numactl -C 0-23 <test_program>
```

When the dataset is large (in the range of 100 million to billions for example), the traditional approach is to slice the dataset into several smaller datasets to get the **topK** for each dataset separately before merging them back together at the end. Since the amount of communication between multiple machines has increased, latency also increases while the QPS performance decreases. Our experience with HNSW on large datasets show that it is better not to slice datasets if possible, but rather establish indices and execute searches on complete datasets to get the best performance. When a dataset is too large and there is not enough DDR space (e.g. local memory space), you can consider using PMem (Intel® Optane™ persistent memory)
By saving the **HNSW layer0** data on PMEM, the size of the dataset that can be supported has greatly increased (a single socket can support an INT8 database with up to 4 billion records @ **d=100**). The persistence feature allows you to skip the loading process for a large amount of data, which greatly reduces the time it takes to initialize.

8. AI Neural Network Model Quantization

8.1. AI neural network quantization process

Computations in neural networks are mainly concentrated in the convolutional layer and the fully connected layer. The computations on these two layers can be expressed as: **Y = X * Weights + Bias**. Therefore, it is natural to focus on matrix multiplication to optimize performance. The way to begin neural network model quantization is by trading-off precision (limited) for performance improvement. By replacing 32-bit floating-point numbers with low-precision integers for matrix operations, it not only speeds up calculations, but also compresses the model, thus saving memory bandwidth. There are three approaches to the quantization of neural network models:

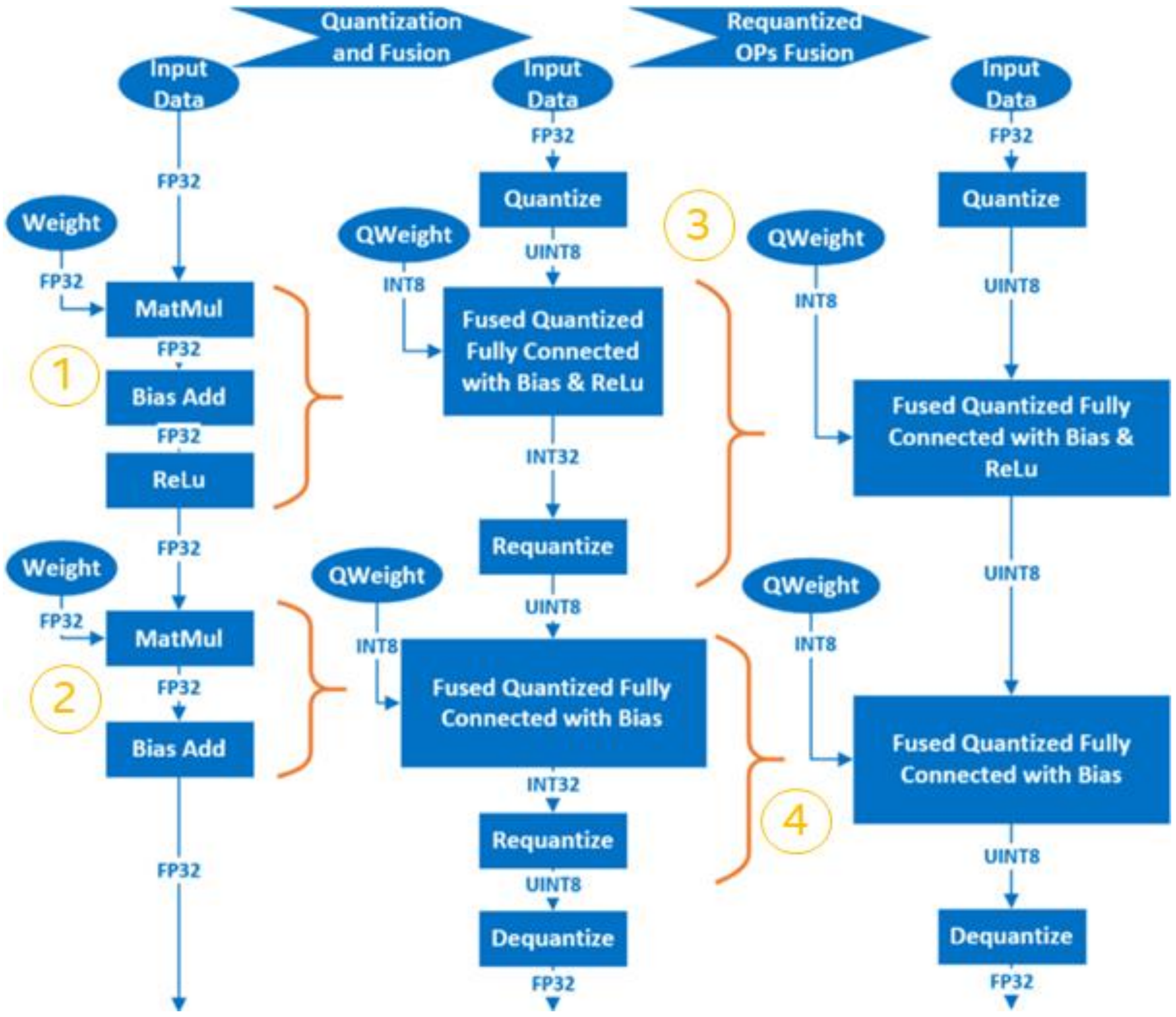
- **P**ost-**T**raining **Q**uantization, which is supported by most AI frameworks.
- **Q**uantization-**A**ware-**T**raining, which inserts the **FakeQuantization** node into the FP32 model when the training converges. It increases the quantization-induced noise. During the backpropagation stage of the training, the model weights fall into a finite interval which results in better quantization precision.
- **D**ynamic **Q**uantization is very similar to PTQ. They are both quantization methods used on post-trained models. The difference lies in that the quantization factor in the activation layer is dynamically decided by the data range used when the neural network model is run, while for PTQ samples from a small-scale pre-processed dataset are used to obtain data distribution and range information in the activation layer, then records it permanently in the newly generated quantization model. Of the Intel® AI Quantization Tools for TensorFlow which we will talk about later on, **onnxruntime** supports this method at the backend only.

The basic procedure for the post-training quantization of neural networks is as follows:

1. Fuse FP32 OP to INT8 OP. For example, *MatMul*, *BiasAdd* and *ReLU* can be fused into a single quantized OP at the fully connected layer, ***QuantizedMatMulWithBiasAndRelu***. Different neural network frameworks support different fuse-able OPs. For Intel® AI Quantization Tools for TensorFlow, which will be discussed later on, below we can see a list of fuse-able OPs supported by TensorFlow: https://github.com/intel/lpot/blob/master/lpot/adaptor/tensorflow.yaml#L190.

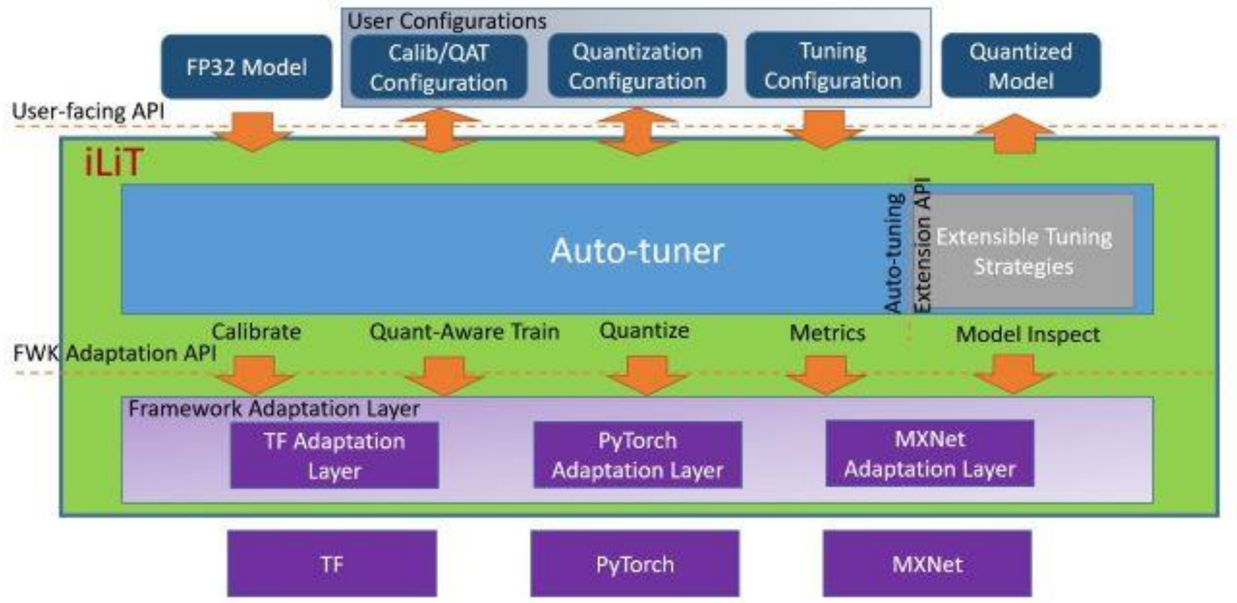
   For fuse-able OPs supported by pyTorch, please see : https://github.com/intel/lpot/blob/master/lpot/adaptor/pytorch_cpu.yaml#L124
2. Quantize weights and save them in the quantized model.
3. Quantize the input/activation layer by sampling the calibration dataset to acquire the distribution and range information of the data in the activation layer, which is then recorded in the newly generated quantized model.
4. The **Requantize** operation is fused into its corresponding INT8 OP to generate the final quantized model.

Using a simple model which includes two layers of **MatMul** as an example, we can observe the quantization process as follows:



8.2. Intel® AI Quantization Tools for TensorFlow

Intel® AI Quantization Tools for TensorFlow is an open source Python library which provides API access for low-precision quantization for cross-neural network development frameworks. It is intended to provide simple, easy-to-use and precision-driven auto tuning tools for the quantization of models for accelerating the inference performance of low-precision models on the 3rd Gen Intel® Xeon® Scalable Processor platform.
Reference: https://github.com/intel/lpot



Intel® AI Quantization Tools for TensorFlow currently support the following Intel optimized deep learning frameworks:

- Tensorflow*
- PyTorch*
- Apache* MXNet
- ONNX Runtime

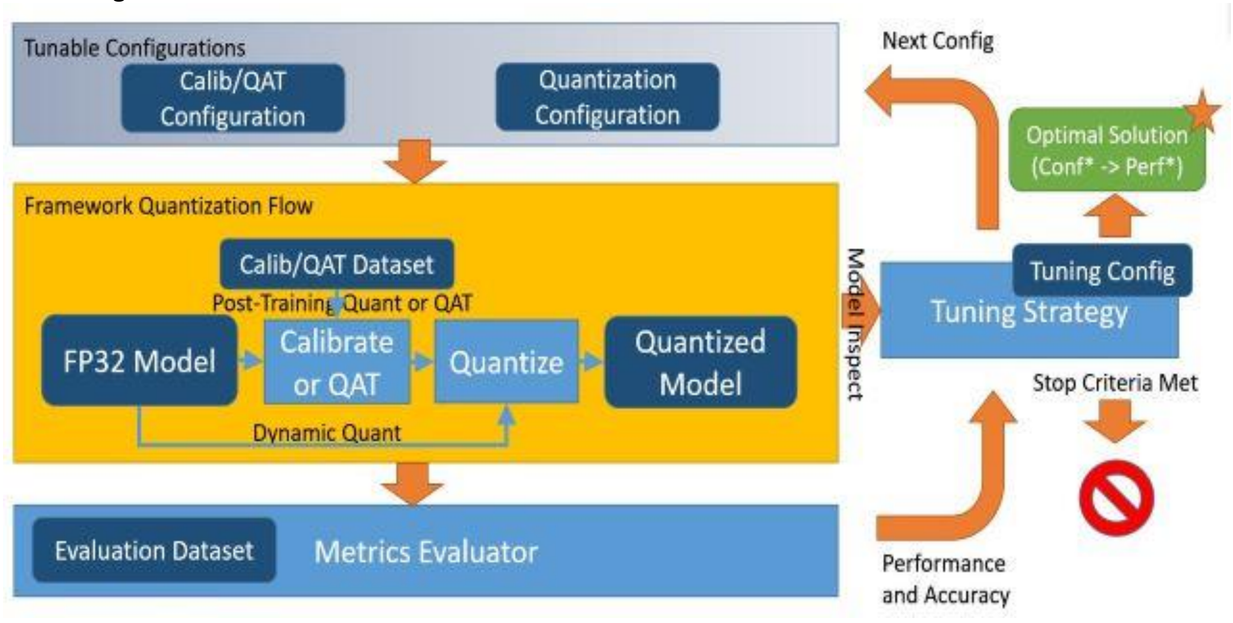
The frameworks and their versions that have already been verified are shown below:

OS	Python	Framework	Version
CentOS 7.8 Ubuntu 18.04	3.6 3.7	TensorFlow	2.2.0
			1.15.0 UP1
			1.15.0 UP2
			2.3.0
			2.1.0
			1.15.2
		PyTorch	1.5.0+cpu
		Apache* MXNet	1.7.0
			1.6.0
		ONNX Runtime	1.6.0

The tuning strategies supported by Intel® AI Quantization Tools for Tensorflow include:

- Basic
- Bayesian
- Exhaustive
- MSE
- Random
- TPE

The workflow for Intel® AI Quantization Tools for TensorFlow is shown below. The model quantization parameters matching the precision loss target are automatically selected according to the set tuning strategy, and the quantized model is generated:



8.3. Installing Intel® AI Quantization Tools for TensorFlow

For details on installation, refer to: https://github.com/intel/lpot/blob/master/README.md

Step 1: Use **Anaconda** to create a **Python3.x** virtual environment with the name of **lpot**. We are using **Python 3.7** here as an example:

```
# conda create -n lpot python=3.7
# conda activate lpot
```

Step 2: Install **lpot**; the two following installation methods are available:
Installing with the binary file:

```
# pip install lpot
```

Install from the source code

```
# git clone https://github.com/intel/lpot.git
# cd lpot
# pip install –r requirements.txt
# python setup.py install
```

8.4. Using Intel® AI Quantization Tools for TensorFlow

We are using **ResNet50 v1.0** as an example to explain how to use this tool for quantization.

8.4.1. Dataset preparation:

Step 1: Download and decompress the ImageNet validation dataset:

```
# mkdir –p img_raw/val && cd img_raw
# wget http://www.image-
net.org/challenges/LSVRC/2012/dd31405981ef5f776aa17412e1f0c112/ILSVRC2012_img_val.tar
# tar –xvf ILSVRC2012_img_val.tar -C val
```

Step 2: Move the image files to the child directories sorted by label:

```
# cd val
# wget -qO-
https://raw.githubusercontent.com/soumith/imagenetloader.torch/master/valprep.sh | bash
```

Step 3: Use the script, prepare_dataset.sh, to convert raw data to the **TFrecord** format:

```
# cd examples/tensorflow/image_recognition
# bash prepare_dataset.sh --output_dir=./data --raw_dir=/PATH/TO/img_raw/val/ --
subset=validation
```

Reference:
https://github.com/intel/lpot/tree/master/examples/tensorflow/image_recognition#2-prepare-dataset

8.4.2. Model preparation:

```
# wget https://storage.googleapis.com/intel-optimized-
tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb
```

8.4.3. Run Tuning:

Edit the file: examples/tensorflow/image_recognition/resnet50_v1.yaml, making sure the dataset path for **quantization\calibration**, **evaluation\accuracy** and **evaluation\performance** is the user's real local path. It should be where the **TFrecord** data generated previously during the data preparation stage, is located.

```
# cd examples/tensorflow/image_recognition
# bash run_tuning.sh --config=resnet50_v1.yaml \
--input_model=/PATH/TO/resnet50_fp32_pretrained_model.pb \
--output_model=./lpot_resnet50_v1.pb
```

Reference:
https://github.com/intel/lpot/tree/master/examples/tensorflow/image_recognition#1-resnet50-v10

8.4.4. Run Benchmark:

```
# bash run_benchmark.sh --input_model=./lpot_resnet50_v1.pb --config=resnet50_v1.yaml
```

The output is shown below. The performance data is for reference only:

```
accuracy mode benchmark result:
Accuracy is 0.739
Batch size = 32
Latency: 1.341 ms
Throughput: 745.631 images/sec

performance mode benchmark result:
Accuracy is 0.000
Batch size = 32
Latency: 1.300 ms
Throughput: 769.302 images/sec
```

9. Using Intel® Distribution of OpenVINO™ Toolkit for Inference Acceleration

9.1. Intel® Distribution of OpenVINO™ Toolkit

Intel® Distribution of OpenVINO^{TM} toolkit's official website and download websites:

```
https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html
```

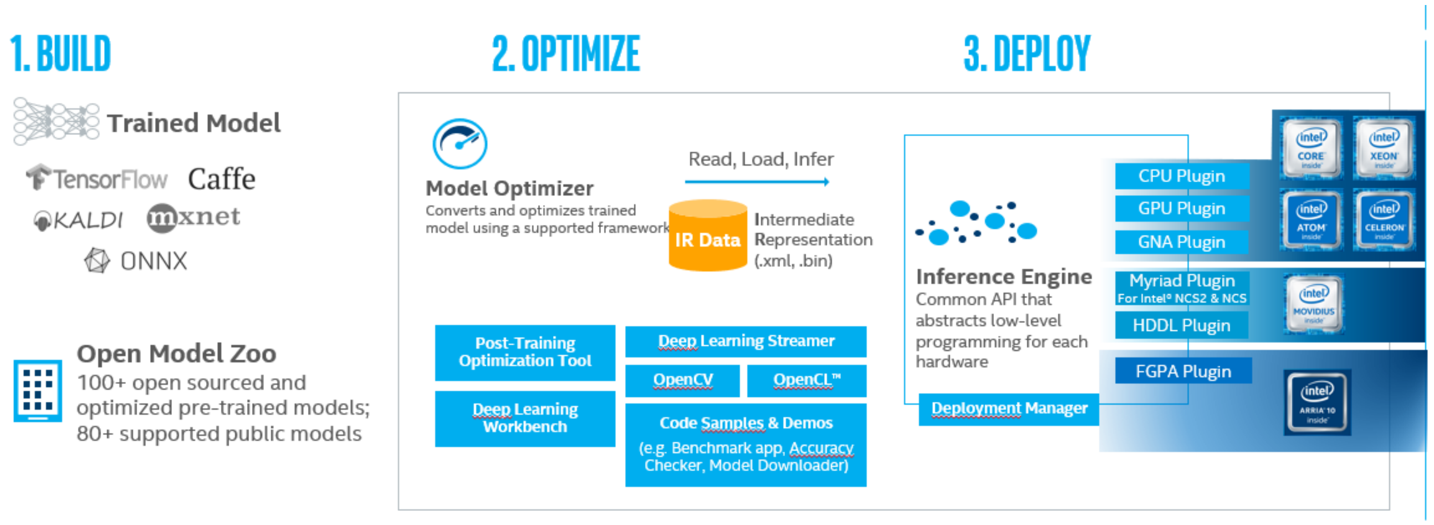
Online documentation:

```
https://docs.openvinotoolkit.org/latest/index.html
```

Online documentation in Simplified Chinese:

```
https://docs.openvinotoolkit.org/cn/index.html
```

The Intel® Distribution of OpenVINO^{TM} toolkit is used to accelerate the development of computer vision and deep learning applications. It supports deep learning applications with various accelerators, including CPUs, GPUs, FPGAs, and Intel® Movidius™ CPUs on the Intel® Xeon® Processor platform, and it also directly supports heterogenous execution.

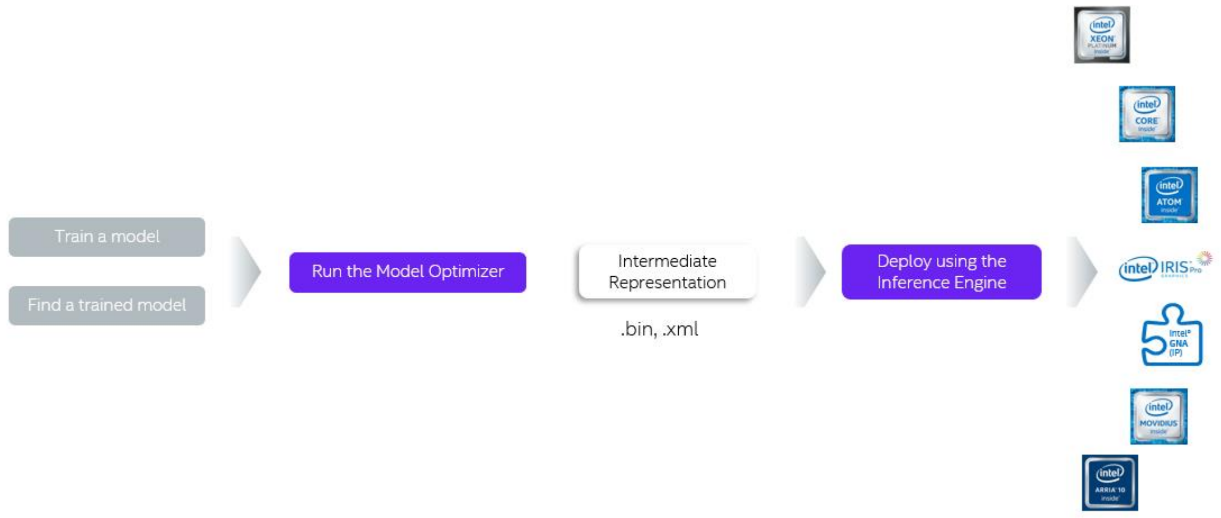


The Intel® Distribution of OpenVINO^{TM} toolkit is designed to improve the performance and reduce the development time of computer vision processing and deep learning inference solutions. It includes two components: computer vision and deep learning development kits.
The Deep Learning Deployment Toolkit (DLDT) is a cross-platform tool for accelerating deep learning inference performance, and includes the following components:

- **Model Optimizer**: converts models trained with Caffe*, TensorFlow, Mxnet, and other frameworks into Intermediate Representations (IR).
- **Inference Engine**: executes the IR on CPU, GPU, FPGA, VPU, and other hardware. It automatically calls the

hardware acceleration kit to accelerate inference performance.

The Intel® Distribution of OpenVINO™ toolkit Workflow:



9.2. Deploying the Intel® Distribution of OpenVINO™ Toolkit

You can refer to the installation documentation in Simplified Chinese:

Installing the Intel® Distribution of OpenVINO™ toolkit for Linux*:
https://docs.openvinotoolkit.org/downloads/cn/I03030-5-Install Intel_ Distribution of OpenVINO_ toolkit for Linux – OpenVINO_ Toolkit.pdf

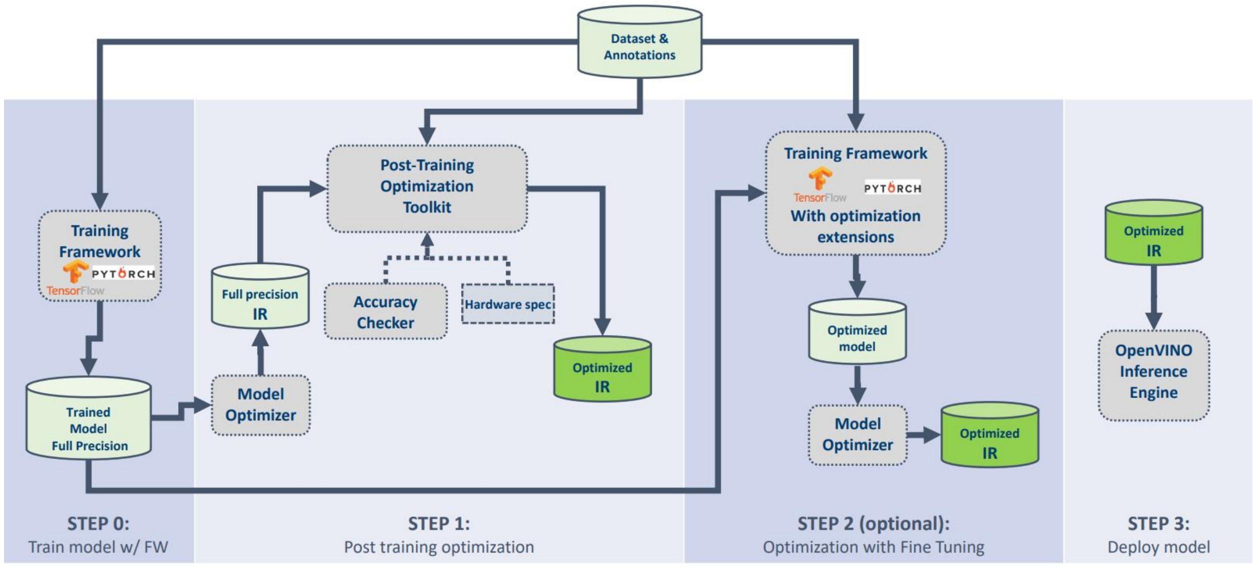
9.3. Using Deep Learning Deployment Toolkit (DLDT) of the Intel® Distribution of OpenVINO Toolkit

Introduction to the Intel® Deep Learning Deployment toolkit:
https://docs.openvinotoolkit.org/downloads/cn/I03030-9-Introduction to Intel_ Deep Learning Deployment Toolkit – OpenVINO_ Toolkit.pdf
Image Classification C++ Sample (Async):
https://docs.openvinotoolkit.org/downloads/cn/I03030-10-Image Classification Cpp Sample Async – OpenVINO_ Toolkit.pdf
Object Detection C++ Sample (SSD):
https://docs.openvinotoolkit.org/downloads/cn/I03030-11-Object Detection Cpp Sample SSD - OpenVINO_ Toolkit.pdf
Automatic Speech Recognition C++ Sample:
https://docs.openvinotoolkit.org/downloads/cn/I03030-12-Automatic Speech Recognition Cpp_Sample - OpenVINO_ Toolkit.pdf
Action Recognition Python* Demo:
https://docs.openvinotoolkit.org/downloads/cn/I03030-13-Action Recognition Python Demo - OpenVINO_ Toolkit.pdf
Crossroad Camera C++ Demo:
https://docs.openvinotoolkit.org/downloads/cn/I03030-14-Crossroad Camera Cpp_Demo - OpenVINO_ Toolkit.pdf
Human Pose Estimation C++ Demo:
https://docs.openvinotoolkit.org/downloads/cn/I03030-15-Human Pose Estimation Cpp Demo - OpenVINO_ Toolkit.pdf
Interactive Face Detection C++ Demo:
https://docs.openvinotoolkit.org/downloads/cn/I03030-16-Interactive Face Detection Cpp_Demo - OpenVINO_ Toolkit.pdf

9.4. Using the Intel® Distribution of OpenVINO™ Toolkit for INT8 Inference Acceleration

By inferencing on an INT8-based model and using Intel DL Boost on the Intel® Xeon® Scalable Processor platform for acceleration, you can greatly increase inference efficiency. At the same time, it saves computing resources and reduces

power consumption. The 2020 version and later versions of OpenVINO™ all provide INT8 quantization tools which support the quantization of FP32-based models.
The INT8-based model quantization tool provided by OpenVINO is a **P**ost-training **O**ptimization **T**oolkit **(POT)** is used to optimize and quantize trained models. There is no need to re-train or fine-tune models or to modify model structures. The figure below shows the process of how OpenVINO is used to optimize new models.
Step 0: Acquire the trained model,
Step 1: POT generation and optimization,
Step 2: Optional operation (Whether to fine-tune the model will be determined according to the actual situation for better accuracy), and
Step 3: Use OpenVINO IE for model inference.



POT provides an independent command line tool and Python API and it mainly supports the following features:

- Two types of post-training INT8 quantization algorithms: fast DefaultQuantization and precise AccuracyAwareQuantization.
- Uses the **T**ree-structured **Parzen Estimator** for global optimization of post-training quantization parameters
- Supports both symmetrical and asymmetrical quantization
- Supports compression for multiple hardware platforms (CPU, GPU)
- Quantizes all channels at the convolutional layer and full connection layer
- Supports multiple applications: computer vision, recommendation system
- Provides customized optimization methods through provided API

Please refer to the following websites for instructions of operations and use:
Introduction to the Post-Training Optimization Toolkit
https://docs.openvinotoolkit.org/latest/pot_README.html
Low Precision Optimization Guide:
https://docs.openvinotoolkit.org/latest/pot_docs_LowPrecisionOptimizationGuide.html
Post-training Optimization Toolkit Best Practices
https://docs.openvinotoolkit.org/latest/pot_docs_BestPractices.html
Post-training Optimization Toolkit Frequently Asked Questions
https://docs.openvinotoolkit.org/latest/pot_docs_FrequentlyAskedQuestions.html
INT8 quantization and optimization using DL Workbench's web interface
https://docs.openvinotoolkit.org/latest/workbench_docs_Workbench_DG_Int_8_Quantization.html

10. Using Intel® DAAL for Accelerated Machine Learning

Intel® Data Analytics Acceleration Library (Intel® DAAL)

As a branch of artificial intelligence, machine learning is currently attracting a huge amount of attention. Machine learning-based analytics is also getting increasingly popular. The reason is that, when compared to other analytics, machine learning can help IT staff, data scientists, and various business teams and their organizations to quickly unleash the strengths of AI. Furthermore, machine learning offers many new commercial and open-source solutions, providing a vast ecosystem for developers. In addition, developers can choose from a variety of open-source machine learning libraries such as **Scikit-learn**, **Cloudera*** and **Spark* MLlib**.

10.1. Intel® Distribution for Python*

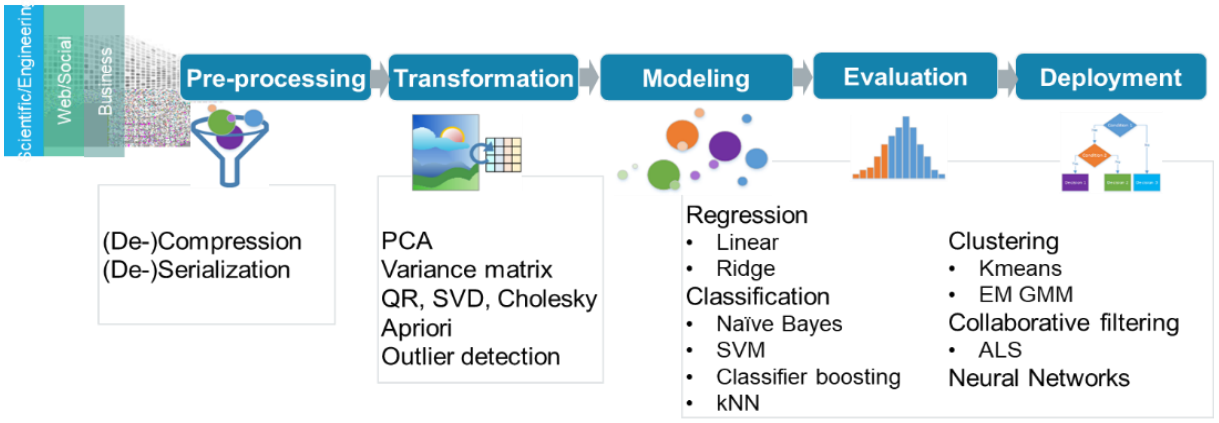
Intel® Distribution for Python* is a Python development toolkit for artificial intelligence software developers. It can be used to accelerate computational speed of Python on the Intel® Xeon® Scalable Processor platform. It is available at **Anaconda***, and it can also be installed and used with **Conda*, PIP*, APT GET, YUM, Docker***, among others. Reference and download site: https://software.intel.com/content/www/us/en/develop/tools/distribution-for-python.html
Intel® Distribution for Python* features:

- ✓ Out-of-the-box: no or little change to source code required to achieve faster Python application performance.
- ✓ The Integrated Intel® performance libraries: Intel® Math Kernel Library (MKL) and Intel® Data Analytics Acceleration Library (Intel® DAAL), for example, can be used to accelerate NumPy, SciPy, and scikit-learn*
- ✓ Latest vector and multithread instructions: Numba* and Cython can be combined to improve concurrency and vectorization efficiency.

10.2. Intel® DAAL

Intel® Data Analytics Acceleration Library (DAAL) is designed for data scientists to accelerate data analytics and prediction efficiency. In particular, it can take full advantage of vectorization and multithreading for applications with huge amount of data, as well as utilize other technologies to increase the overall performance of machine learning on the Intel® Xeon® Scalable Processor platform.

Intel® DAAL is a complete end-to-end software solution designed to help data scientists and analysts quickly build



everything from data pre-processing, to data feature engineering, data modeling and deployment. It provides various data analytics needed to develop machine learning and analytics as well as high-performance building blocks required by algorithms. It currently supports linear regression, logic regression, LASSO, AdaBoost, Bayesian classifiers, support vector machines, k-nearest neighbors, k-means clustering, DBSCAN clustering, various types of decision trees, random forest, gradient boosting, and other classic machine learning algorithms. These algorithms are highly optimized to achieve high performance on Intel® processors. For example, a leading big data analytics technology and service provider has used these resources to improve the performance of data mining algorithms by several times.

To make it easier for developers to use Intel® DAAL in machine learning applications in Intel-based environments, Intel has open-sourced the entire project (https://github.com/intel/daal), and provides full-memory, streaming and distributed algorithm support for different big data scenarios. For example, DAAL Kmeans can be combined with Spark to perform multi-node clustering on a Spark cluster. In addition, DAAL provides interfaces for C++, Java*, and Python.

DAAL4py

In order to provide better support for Scikitlearn, which is the most widely used with Python, Intel® DAAL provides a very simple Python interface, DAAL4py (please see the open source website for more details: https://github.com/IntelPython/daal4py). It can be used seamlessly with Scikitlearn and provides acceleration for machine learning algorithms at the underlying layer.
Developers do not need to modify the Scikitlearn source code to benefit from the advantages of automatic vectorization and multithreading. DAAL4py currently supports the following algorithms in Scikitlearn:

- Sklearn linear regression, Sklearn ridge regression and logic regression
- PCA
- KMeans
- pairwise_distance
- SVC (SVM classification)

10.3. Installing Intel® Distribution for Python & Intel® DAAL

Download and install Intel® Distribution for Python* (Intel® DAAL already included):
FYI: https://software.intel.com/en-us/distribution-for-python
Installing Intel® DAAL separately:
FYI: https://software.intel.com/en-us/daal.
Intel® DAAL Developer Guide:
FYI：https://software.intel.com/en-us/daal-programming-guide

10.4. Using Intel® DAAL

There are two ways to use Intel® DAAL to accelerate scikit-learn:
Method 1: Using the command line

```
# python -m daal4py <your-scikit-learn-script>
```

Method 2: Adding it to source code

```
import daal4py.sklearn
daal4py.sklearn.patch_sklearn('kmeans')
```

11. References

[1] Intel® AVX-512 info: https://colfaxresearch.com/skl-avx512/
[2] Intel® Optimized AI Frameworks: https://software.intel.com/en-us/frameworks
[3] Intel® Distribution of OpenVINO™ toolkit: https://docs.openvinotoolkit.org/
[4] Intel® Analytics Zoo: https://github.com/intel-analytics/analytics-zoo
[5] Hands-on IDP and Intel® DAAL : https://software.intel.com/en-us/videos/get-your-hands-dirty-with-intel-distribution-for-python
[6] IDP benchmarks: https://software.intel.com/en-us/distribution-for-python/benchmarks
[7] Intel® DL Boost : https://www.intel.ai/increasing-ai-performance-intel-dlboost/#gs.3cxhiw
[8] Intel® DL Boost: https://www.intel.com/content/dam/www/public/us/en/documents/product-overviews/dl-boost-product-overview.pdf
https://software.intel.com/content/www/us/en/develop/articles/lower-numerical-precision-deep-learning-inference-and-training.html
[9] Open source of Faiss project: https://github.com/facebookresearch/faiss
[10] Open source of HNSWLib project: https://github.com/nmslib/hnswlib