Technology Guide

Intel® Deep Learning Boost (Intel® DL Boost) - Improve Inference Performance of Hugging Face BERT Base Model in Google Cloud Platform (GCP)

Authors

David Lu

Shuangpeng Zhou

Jing Xu

Weizhuo Zhang

Abhijit Sinha

Heqing Zhu

Lulin Deng

1 Introduction

Emails and SMS messages are very popular communication tools, and many people rely on them every day. There are also cyber attackers who send massive phishing emails or SMS messages to steal private information. Despite various ways to prevent them, detecting these attacks can be very difficult due to the use of traffic engineering. With the development of deep learning technologies, it is proven to be the best way to prevent such advanced cyberattacks. Network security companies already used many different deep learning methods such as CNN, LSTM, GRU, and BERT in their security products. While the BERT model can provide the highest accuracy[1], it takes longer inference time. The inference latency is one of the challenges to adopting deep learning technology.

PyTorch* is one of the most widely used deep-learning frameworks. To boost the performance on Intel® hardware, Intel provides the open-source Intel® Extension for PyTorch* (IPEX) with the latest feature optimizations. These optimizations take advantage of Intel® AVX-512 Vector Neural Network Instructions (Intel® AVX-512 VNNI) and Intel® Advanced Matrix Extensions (Intel® AMX) on Intel® CPUs as well as Intel® X^e Matrix Extensions (Intel® XMX) AI engines on Intel discrete GPUs.

Google Cloud Platform* service (GCP) is a suite of cloud computing services offered by Google. It offers various cloud services to help customers build, deploy, and manage different kinds of applications and services. GCP facilitates more convenient performance evaluation. Deploying services on GCP conforms to the customers' scenario, making the evaluation results more reliable and authentic.

This guide illustrates how to use PyTorch and Intel IPEX tool to boost deep-learning inference performance on the Hugging Face BERT base model (cased). The evaluations were conducted on the GCP using three different hardware configurations. We will show a Gen-2-Gen performance comparison from 1st Gen Intel® Xeon® Scalable processor to 3rd Gen Intel® Xeon® Scalable processor.

This solution and its associated resources can serve as a reference for customers to replicate other workloads. This document is part of the Network Transformation Experience Kits.

[1] https://link.springer.com/article/10.1007/s11227-021-04169-6

Table of Contents

Figures

Tables

Document Revision History

Revision	Date	Description
001	April 2023	Initial release.

1.1 Terminology

Table 1. Terminology

Abbreviation	Description
AMX	Intel® Advanced Matrix Extensions
AVX	Advanced Vector Extensions
BERT	Bidirectional Encoder Representations from Transformers
CLS	Classification
CNN	Convolutional Neural Network
GCP	Google Cloud Platform
GRU	Gate Recurrent Unit
IPEX	Intel® Extension for PyTorch
LSTM	Long Short-Term Memory networks
oneDNN	Intel® oneAPI Deep Neural Network Library
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Networks

1.2 Reference Documentation

Table 2. Reference Documents

Reference	Source
Spam Detection Using BERT	https://arxiv.org/ftp/arxiv/papers/2206/2206.02443.pdf
Intel® Deep Learning Boost (Intel® DL Boost)	https://www.intel.com/content/www/us/en/artificial-intelligence/deep-learning-boost.html
Intel® oneAPI Deep Neural Network Library	https://github.com/oneapi-src/oneDNN
Intel® Extension for PyTorch	https://github.com/intel/intel-extension-for-pytorch
Bert Base model (cased)	https://huggingface.co/bert-base-cased

2 Technology Overview

2.1 Intel® Deep Learning Boost (Intel® DL Boost)

Intel Deep Learning Boost (Intel DL Boost) is a set of built-in accelerators to enhance the performance of common AI training and inferencing workloads. It was introduced in the 2nd Gen Intel® Xeon® Scalable processors and offers AI acceleration within the same CPU package that already offers exceptional performance, security features, and reliability for traditional workloads in data centers or the cloud.

The core of Intel DL Boost is Vector Neural Network Instructions (VNNI), which is a specialized instruction set that uses a single instruction for deep-learning computations that formerly required three separate instructions. VNNI is built to boost the performance of deep-learning workloads. Intel has invested resources in popular deep-learning frameworks like PyTorch, TensorFlow*, MXNet*, and Open Neural Network Exchange (ONNX*), helping customers to take advantage of Intel DL Boost to improve AI workload performance.

2.2 PyTorch

PyTorch is a deep learning library that provides flexible and user-friendly interfaces for building and training neural networks. It is built with the help of Torch library and has been developed to support dynamic computational graphs, which allows easier and more flexible building of complex models.

PyTorch supports a wide range of neural network architectures, from simple feedforward networks to more complex models such as recurrent neural networks and convolutional neural network. It is widely used for a variety of applications including computer vision, natural language processing, and generative models.

2.3 Intel® Extension for PyTorch* (IPEX)

IPEX provides PyTorch users with up-to-date features and optimizations for Intel hardware, including Intel AVX-512 VNNI and Intel AMX. With IPEX, users can access simple Python APIs and tools to optimize performance through graph and operator optimization with only minor code modifications. Graph optimization includes the fusion of frequently used operator patterns, such as Conv2D+ReLU and Linear+ReLU. IPEX can optimize both eager mode and graph mode.

2.4 Bidirectional Encoder Representations from Transformers (BERT)

BERT is a pre-trained language representation model developed by Google AI Language researchers in 2018, which consists of transformer blocks with a variable number of encoder layers and a self-attention head. In contrast to traditional one-way language models or superficial combinations of two one-way language models, BERT prioritizes the use of a new Masked Language Model (MLM) to generate deep bidirectional representation. This allows BERT to better leverage the information from both left and right context in the input text, leading to more accurate language processing.

As shown in Figure 1, BERT takes the embeddings of each word in the sentence as the input and uses a special classification token ([CLS]) and a special separator token ([SEP]) to better understand the sequences. Each input embedding is corresponding to an output, for instance, C is the output of the last transformer of the classification token ([CLS]), T_i is the output of the last transformer of corresponding tokens. For some token level tasks, it can take advantage of each output of the token, and for the sequence level tasks, including the phishing email detection task, it can use the output of the classification token ([CLS]).

To detect phishing emails, the input email is first tokenized into chunks of words using the Hugging Face tokenizer, with a special CLS token added at the beginning. The tokens are then padded to the maximum BERT input size, which by default is 512. The total input tokens are converted to integer IDs and fed to the BERT model. A dense layer is added for email classification, which takes the last hidden state for the CLS token as input.

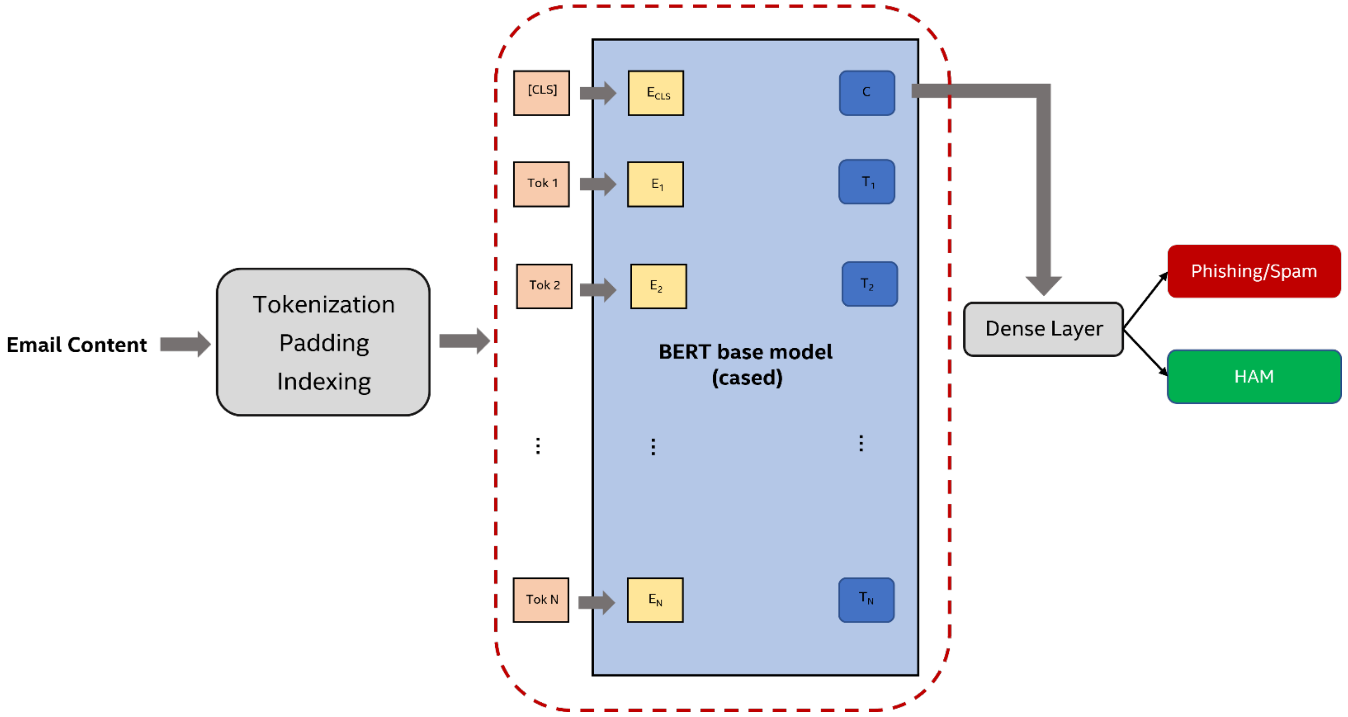


Figure 1. The workflow of BERT based phishing emails detection

3 Performance Optimization with Intel Extension for PyTorch (IPEX)

To improve the deep learning inference performance, one common technique is quantization. Quantization is the process of representing continuous values with a limited number of discrete values, so it is obvious that quantization can reduce storage requirements since it lowers the data precision. Furthermore, quantization can improve the efficiency of computations. By using fewer bits to represent values, quantization can reduce the computational load in the machine. There are two main types of quantization: static quantization and dynamic quantization. Static quantization involves the process of quantizing both the weights and activations of a model. To achieve optimal quantization parameters for activations, the quantization method requires a representative dataset to do the calibration. For dynamic quantization, the weights are quantized before the inference process, while the activations are quantized dynamically during the inference. Normally, static quantization is used when both memory bandwidth and compute savings are important. Dynamic quantization can introduce additional run-time overhead since it quantizes the activations during the inference. Therefore, dynamic quantization is better for those models in which the memory bandwidth is the bottleneck. In this chapter, we will illustrate how to improve the inference performance by using IPEX under Hugging Face BERT base model (cased).

3.1 Prepare the Benchmark Environment

First, create a clean virtual environment for benchmarking by using the following commands:

```
# python3 -m venv ipex-bert
# source ipex-bert/bin/activate
# pip install -U pip
# pip install -r requirements.txt
```

The requirements.txt file enumerates dependent packages, whose content is shown as below:

```
# cat requirements.txt
--extra-index-url https://download.pytorch.org/whl/cpu
torch==1.13.0
intel_extension_for_pytorch==1.13.0
accelerate
transformers
datasets
tqdm
```

3.2 Quantize BERT Model with Intel Extension for PyTorch (IPEX)

Apply IPEX static quantization method to BERT model by using the following code snippet:

```
# cat ipex_quantize.py
import torch
from tqdm import tqdm
from datasets import load_dataset
from transformers import BertTokenizer, BertForSequenceClassification

import intel_extension_for_pytorch as ipex
from intel_extension_for_pytorch.quantization import prepare, convert
from torch.ao.quantization import MinMaxObserver, PerChannelMinMaxObserver, QConfig

# Define variables
MAX_LENGTH = 512
MODEL_CHECKPOINT = 'bert-base-cased'

# Load model and tokenizer
tokenizer = BertTokenizer.from_pretrained(MODEL_CHECKPOINT)
model = BertForSequenceClassification.from_pretrained(MODEL_CHECKPOINT)
model.eval()

# Prepare calibration dataset
def preprocess(text):
    return tokenizer(text, max_length=MAX_LENGTH, padding='max_length', truncation=True, return_tensors='pt')
raw_dataset = load_dataset('SetFit/enron_spam', split='test')
calib_dataset = list(map(preprocess, raw_dataset['text'][:1000]))

# Execute ipex static quantization
example_inputs = tuple(calib_dataset[0].values())

qconfig = QConfig(activation=MinMaxObserver.with_args(qscheme=torch.per_tensor_affine, dtype=torch.quint8),
                  weight=PerChannelMinMaxObserver.with_args(dtype=torch.qint8,
qscheme=torch.per_channel_symmetric))
prepared_model = prepare(model, qconfig, example_inputs=example_inputs, inplace=False)

for encoding in tqdm(calib_dataset):
    prepared_model(**encoding)

converted_model = convert(prepared_model)
# Save quantized model
with torch.no_grad():
    traced_model = torch.jit.trace(converted_model, example_inputs, strict=False)
    traced_model = torch.jit.freeze(traced_model)
    traced_model.save(f'{MODEL_CHECKPOINT}_ipex-static-quan.pt')
```

Run the following command to quantize BERT model with IPEX. Then the quantized model will be saved as a .pt file.

```
# python3 ipex_quantize.py
```

3.3 Evaluate IPEX Quantization Performance

Evaluate the quantized BERT model's performance on the GCP n1-std-8 (1st Gen Intel Xeon Scalable processors), n2-std-8 (2nd Gen Intel Xeon Scalable processors), n2-std-8 (3rd Gen Intel Xeon Scalable processors) and take the result on n1-std-8 (1st Gen Intel Xeon Scalable processors) as the baseline. The evaluation code snippet is shown below:

```
# cat evaluate.py
import time
import torch
import numpy as np
from tqdm import tqdm
from datasets import load_dataset
import intel_extension_for_pytorch as ipex
from transformers import BertTokenizer, BertForSequenceClassification

# Define variables
MAX_LENGTH = 512
NUM_WARM_UP = 100
MODEL_CHECKPOINT = 'bert-base-cased'

# Load and tokenizer
tokenizer = BertTokenizer.from_pretrained(MODEL_CHECKPOINT)

# Prepare evaluation dataset
def preprocess(text):
    return tokenizer(text, max_length=MAX_LENGTH, padding='max_length', truncation=True, return_tensors='pt')
raw_dataset = load_dataset('SetFit/enron_spam', split='test')
eval_dataset = list(map(preprocess, raw_dataset['text'][:1000]))

# Define benchmark function
def benchmark(model, dataset):
    model.eval()

    latencies = []
    with torch.no_grad():
        for encoding in tqdm(dataset):
            start = time.time()
            model(**encoding)
            elapsed = 1000 * (time.time() - start)
            latencies.append(elapsed)

    latencies = latencies[NUM_WARM_UP:]
    mean_latency = np.mean(latencies)

    return mean_latency

# test performance with IPEX static quantization
q_model = torch.jit.load(f'{MODEL_CHECKPOINT}_ipex-static-quan.pt')
mean_latency = benchmark(q_model, eval_dataset)
print(f'mean latency with IPEX static quantization: {mean_latency} ms')
```

Run following commands on GCP with different virtual machine configurations to get the mean inference time with different cores.

```
# numactl -C 0 python3 evaluate.py                # for 1 core
# numactl -C 0-1 python3 evaluate.py              # for 2 cores
# numactl -C 0-3 python3 evaluate.py              # for 4 cores
```

3.4 Performance Boosting Results

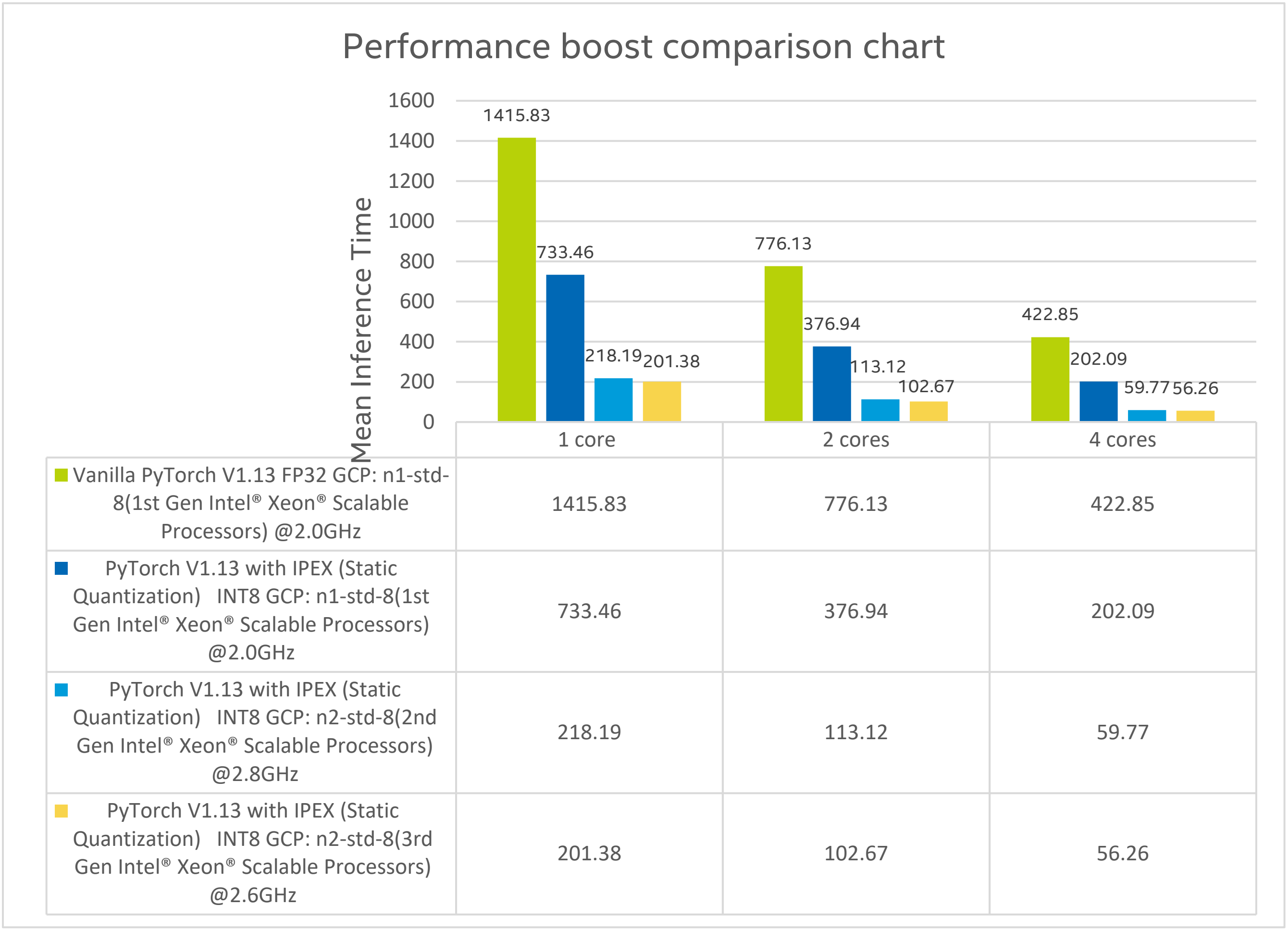
We evaluate the BERT performance on 1st to 3rd generations Intel Xeon Scalable Processors before and after applying IPEX static quantization, and with different number of cores. The details of mean inference time for each situation are shown in Table 3[2].

[2] Workloads and configurations. Results may vary.

Table 3. Results of Performance Boosting with IPEX

Mean inference time (ms) max_seq_length = 512 Batch size = 1	Vanilla PyTorch V1.13			PyTorch V1.13 with IPEX (Static Quantization) INT8			Performance Boost n2-std-8(3rd Gen Intel® Xeon® Scalable Processors) @2.6GHz vs *n1-std-8(1st Gen Intel® Xeon® Scalable Processors)* @2.0GHz under Vanilla PyTorch
	FP32			INT8			
	n1-std-8(1st Gen Intel® Xeon® Scalable Processors) @2.0GHz	*n2-std-8(2nd Gen Intel® Xeon® Scalable Processors)* @2.8GHz	*n2-std-8(3rd Gen Intel® Xeon® Scalable Processors)* @2.6GHz	*n1-std-8(1st Gen Intel® Xeon® Scalable Processors)* @2.0GHz	*n2-std-8(2nd Gen Intel® Xeon® Scalable Processors)* @2.8GHz	*n2-std-8(3rd Gen Intel® Xeon® Scalable Processors)* @2.6GHz	
1 core	1415.83	976.01	960.07	733.46	218.19	201.38	7.03 X
2 cores	776.13	531.93	512.76	376.94	113.12	102.67	7.56 X
4 cores	422.85	300.54	281.94	202.09	59.77	56.26	7.52 X

The comparison chart is shown in Figure 2, which uses a more intuitive way to display the acceleration effect from IPEX[3].



	1 core	2 cores	4 cores
🟩 Vanilla PyTorch V1.13 FP32 GCP: n1-std-8(1st Gen Intel® Xeon® Scalable Processors) @2.0GHz	1415.83	776.13	422.85
🟦 PyTorch V1.13 with IPEX (Static Quantization) INT8 GCP: n1-std-8(1st Gen Intel® Xeon® Scalable Processors) @2.0GHz	733.46	376.94	202.09
🟦 PyTorch V1.13 with IPEX (Static Quantization) INT8 GCP: n2-std-8(2nd Gen Intel® Xeon® Scalable Processors) @2.8GHz	218.19	113.12	59.77
🟨 PyTorch V1.13 with IPEX (Static Quantization) INT8 GCP: n2-std-8(3rd Gen Intel® Xeon® Scalable Processors) @2.6GHz	201.38	102.67	56.26

Figure 2. The comparison chart for performance boost

[3] Workloads and configurations. Results may vary.

From the performance comparison of BERT base model shown above, we have the following conclusions:

- Applying IPEX static post-training quantization will always boost the performance of the Hugging Face BERT base model, irrespective of which generation of Intel® Xeon® Scalable processor is used.
- With the help of the 3rd Gen Intel Xeon Scalable processor and IPEX static post-training quantization, the performance of BERT base model is 7.52 X faster than 1st Gen Intel Xeon Scalable processor under Vanilla PyTorch without quantization.[4]

4 Summary

This guide illustrates the improvement of the IPEX static post-training quantization for BERT base model performance. With the help of both the 3rd Gen Intel® Xeon® Scalable processor and IPEX static post-training quantization, the BERT base model can achieve up to 7.03x ~ 7.56x performance improvement compared to the 1st Gen Intel® Xeon® Scalable processor. In addition, it should be noticed that all the evaluation processes are completed in the GCP considering its flexibility and scalability, which also provide a more believable and feasible result to the customer since the GCP environment is closer to the real scenarios. Furthermore, the white paper provides the details of how to apply the IPEX optimization to the model, which is straightforward.

Appendix A Platform Configuration

Table 4. Platform Configuration

Name	GCP: n1-std-8 SKX	GCP: n2-std-8 CLX	GCP: n2-std-8 ICX
CPU Model	Intel® Xeon® CPU @ 2.00GHz	Intel® Xeon® CPU @ 2.80GHz	Intel® Xeon® CPU @ 2.60GHz
Stepping	3	7	6
Sockets	1	1	1
Cores per Socket	8	8	8
Hyperthreading	Enabled	Enabled	Enabled
CPUs	16	16	16
Intel Turbo Boost	Disabled	Disabled	Disabled
Base Frequency	2.0GHz	2.8GHz	2.6GHz
Maximum Frequency	2000 MHz	2800 MHz	2600 MHz
NUMA Nodes	1	1	1
Installed Memory	16GB (1x16GB RAM)	16GB (1x16GB RAM)	64GB (4x16GB RAM)
NIC	1x device	1x device	1x device
Disk	1x 100G PersistentDisk	1x 100G PersistentDisk	1x 100G PersistentDisk
BIOS	Google	Google	Google
OS	Ubuntu 20.04.5 LTS	Ubuntu 20.04.5 LTS	Ubuntu 20.04.5 LTS
Kernel	5.15.0-1021-gcp	5.15.0-1021-gcp	5.15.0-1021-gcp

[4] Workloads and configurations. Results may vary.

Appendix B Software Configuration

Table 5. Software Configuration

Software Configuration	Config 1 (Vanilla PyTorch for baseline)	Config 2 (PyTorch with IPEX)
Framework /Toolkit incl version	PyTorch 1.13.0	PyTorch 1.13.0 IPEX 1.13.0
Framework URL	https://pytorch.org/	
Topology or ML algorithm (include link)	https://huggingface.co/bert-base-cased	
Libraries (incl version) e.g., MKL DNN, or DAAL	oneDNN v2.7.3	
Dataset (size, shape)	https://huggingface.co/datasets/wikipedia, max_seq_length=512	
Precision (FP32, INT8., BF16)	FP32	INT8
NUMACTL	numactl -c	
OMP_NUM_THREADS	N/A	N/A
COMMAND LINE USED	python3 evaluate.py	

0423/DN/WIT/PDF 775815-001US