Technology Guide

Intel® Deep Learning Boost - Improve Inference Performance of BERT Base Model from Hugging Face for Network Security

Authors

David Lu

Shuangpeng Zhou

Jing Xu

Weizhuo Zhang

Heqing Zhu

Feng Tian

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. There are many ways to prevent such cyberattacks. However, since scammers use traffic engineering, it is very difficult to detect phishing emails or SMS. With the development of deep learning technologies, it is proven to be the best way to prevent such advanced cyberattacks. Network security companies already use different deep-learning methods such as CNN, LSTM, and BERT in their security products. BERT model provides the best accuracy among these models. However, it takes a longer inference time when compared to other deep-learning models. The inference latency is one of the big challenges to adopt BERT model into their network security products such as SASE and NGFW.

PyTorch* is one of the widely used deep-learning frameworks. To boost the performance under Intel® hardware, Intel provides the open-source Intel® Extension for PyTorch* (IPEX) with 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.

This guide illustrates how to use PyTorch and Intel IPEX tool to boost deep-learning inference performance on Hugging Face BERT base model (cased). The guide also shows a gen-2-gen performance comparison between the 3rd Gen Intel® Xeon® Scalable processor (ICX) and the 4th Gen Intel® Xeon® Scalable processor (SPR).

Customers can use this solution and associated collateral as a reference to replicate other workloads as well.

This document is part of the [Network Transformation Experience Kits](#).

Table of Contents

Figures

Tables

Document Revision History

Revision	Date	Description
001	January 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	A classification token to represent the start of sequence
CNN	Convolutional Neural Network
IPEX	Intel® Extension for PyTorch*
LSTM	Long Short-Term Memory networks
NLP	Natural Language Processing
NGFW	Next Generation Firewall
oneDNN	Intel® oneAPI Deep Neural Network Library
RNN	Recurrent Neural Networks
SASE	Secure Access Service Edge
SEP	A separator token to represent a separate segment/sentence
SMS	Short Message Service
Intel XMX	Intel® X^e Matrix Extensions (Intel® XMX)

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 Technologies

- **Intel Deep Learning Boost (Intel DL boost):** Intel DL Boost was first introduced in the 2nd Gen Intel® Xeon® Scalable processors by adding Intel AVX512 VNNI, an extension to the Intel AVX-512 instruction set. It brings accelerated performance to demanding AI workloads without discrete add-on accelerators. It can significantly improve performance for common AI inferencing and training workload. Intel DL Boost includes some Intel AVX-512 instructions and Intel Advanced Matrix Extensions.
- **Intel® Advanced Matrix Extensions (Intel® AMX):** Intel AMX was introduced in the 4th Gen Intel® Xeon® Scalable processor. This built-in accelerator is dedicated to the matrix multiplication at the heart of deep learning workloads. Intel AMX combines a new instruction set that turns large matrices into a single operation with two-dimensional register files that store larger chunks of data for each core.
- **Intel Advanced Vector Extensions 512 (Intel AVX-512):** A 512-bit instruction set that can accelerate performance for demanding workloads and usages like AI inferencing. This specialized instruction set combines three operations into one Vector Neural Network Instructions (VNNI) set, thereby reducing the number of operations per clock cycle. Intel DL Boost also accelerates deep learning workloads using INT8 precision.

2.2 PyTorch

PyTorch is a widely used open-source machine learning framework based on the Torch library. As one of the most popular deep-learning frameworks, PyTorch is easy to learn and has been used in many applications, including natural language processing (NLP). With the latest release of PyTorch, the framework provides graph-based execution, distributed training, mobile deployment, and quantization.

2.3 IPEX

Intel® Extension for PyTorch*(IPEX) extends PyTorch with up-to-date feature optimizations for an extra performance boost on Intel hardware. Optimizations take advantage of Intel AVX-512 VNNI and Intel AMX on Intel CPUs as well as Intel XMX AI engines on Intel discrete GPUs. IPEX provides optimizations for both eager mode and graph mode. Compared to eager mode, graph mode in PyTorch yields better performance from optimization techniques, such as operation fusion. IPEX amplifies them with more comprehensive graph optimizations. Therefore, it is recommended that you take advantage of IPEX with TorchScript whenever your workload supports it. You could choose to run with `torch.jit.trace()` function or `torch.jit.script()` function, but based on our evaluation, `torch.jit.trace()` supports more workloads, which is recommended as the first choice.

2.4 Bidirectional Encoder Representations from Transformers (BERT)

BERT is an open-source machine learning technique for natural language processing developed by Google*. It can help computers understand the meaning of ambiguous language in text by using surrounding text to establish context. BERT is based on transformer model with a variable number of encoder layers and self-attention heads. The original BERT has two models BERT BASE with 12 encoders with 12 bidirectional self-attention heads and BERT LARGE with 24 encoders with 16 bidirectional self-attention heads. Both models are pre-trained and can be fine-tuned.

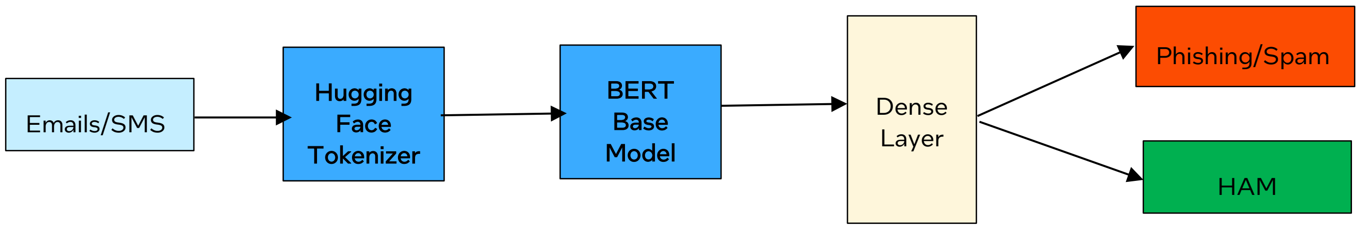


Figure 1. Workflow of BERT based phishing/SPAM emails detection

Based on the Hugging Face Tokenizer, BERT base model, a dense layer is added for phishing/spam emails/SMS detection, as shown in Figure 1. Here is the basic workflow:

1. The content of input email will be tokenized into chunks of tokens with Hugging Face tokenizer.
2. A special token CLS is added at the beginning and another special token SEP is added at the end.
3. The tokens will be padded to the maximum BERT input size (default is 512).
4. The total input tokens will be converted to input IDs, which will be fed to the BERT model.
5. The dense layer takes the last hidden state for CLS token as input, to decide whether the input email is phishing/spam or HAM.

3 Using Intel Deep Learning to Boost Performance

This technical solution shows step by step instruction on how to boost the performance based on Hugging Face BERT base model (cased)(https://huggingface.co/bert-base-cased) under 3rd Gen Intel Xeon Scalable processor and 4th Gen Intel Xeon Scalable processor.

3.1 Prepare the benchmark environment

1. Create a stand-alone Python virtual environment for benchmarking with the following commands:

```
# python3 -m venv bert
# source bert/bin/activate
# pip3 install --upgrade pip3
# pip3 install -r requirements.txt --extra-index-url https://download.pytorch.org/whl/cpu
```

The requirements.txt file enumerates dependent packages, with the following content:

```
# cat requirements.txt
torch==1.13.0
transformers
intel_extension_for_pytorch==1.13.0
```

3.2 Benchmark the performance

1. Run the sample code provide below on the 3rd Gen Intel Xeon Scalable processor server under PyTorch and take the mean inference time as the baseline. For this testing purpose, these tests were run with different cores to demonstrate the scalability.
2. Use the Intel IPEX to perform post-training quantization. Like PyTorch post-training quantization, Intel IPEX also provides two post-training quantization methods: dynamic quantization and static quantization. For this testing purpose, the Intel IPEX post-training static quantization was used in the performance benchmark since it provides the best performance.

```
# cat test.py
import time
import torch
import numpy as np
import intel_extension_for_pytorch as ipex
from intel_extension_for_pytorch.quantization import prepare, convert
from torch.ao.quantization import MinMaxObserver, PerChannelMinMaxObserver, QConfig
from transformers import AutoConfig, AutoTokenizer, AutoModelForSequenceClassification

# define dummy input tensor to use for the model's forward call
# to record operations in the model for tracing
N, max_length = 1, 512
dummy_tensor = torch.ones((N, max_length), dtype=torch.long)

url = 'this is the test email'
tokenizer = AutoTokenizer.from_pretrained('bert-base-cased')
encoding = tokenizer.encode_plus(url, return_tensors='pt', padding='max_length',
truncation=True)

example_inputs = (dummy_tensor, dummy_tensor, dummy_tensor)

global_infer_time = []

def convert_to_traced_model(model):
    with torch.no_grad():
        traced_model = torch.jit.trace(model, example_inputs, strict=False)
        traced_model = torch.jit.freeze(traced_model)
    return traced_model

def benchmark(model, tag):
    with torch.no_grad():
        for _ in range(50):
            model(**encoding)

        infer_time = []
        for _ in range(500):
            start = time.time()
            model(**encoding)
            elapsed = 1000 * (time.time() - start)
            infer_time.append(elapsed)

    print(f'[{tag}] infer_time: mean={np.mean(infer_time)} min={np.min(infer_time)}
max={np.max(infer_time)} std={np.std(infer_time)}')

####################### Evaluate performance under PyTorch #############################
user_model = AutoModelForSequenceClassification.from_pretrained('bert-base-cased')
user_model.eval()
benchmark(user_model, 'Performance under PyTorch')

############# Evaluate performance with IPEX static quantization #################
user_model = AutoModelForSequenceClassification.from_pretrained('bert-base-cased')
user_model.eval()

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(user_model, qconfig, example_inputs=example_inputs, inplace=False)
prepared_model(**encoding)
converted_model = convert(prepared_model)

traced_model = convert_to_traced_model(converted_model)
benchmark(traced_model, 'Performance with IPEX static quantization')
```

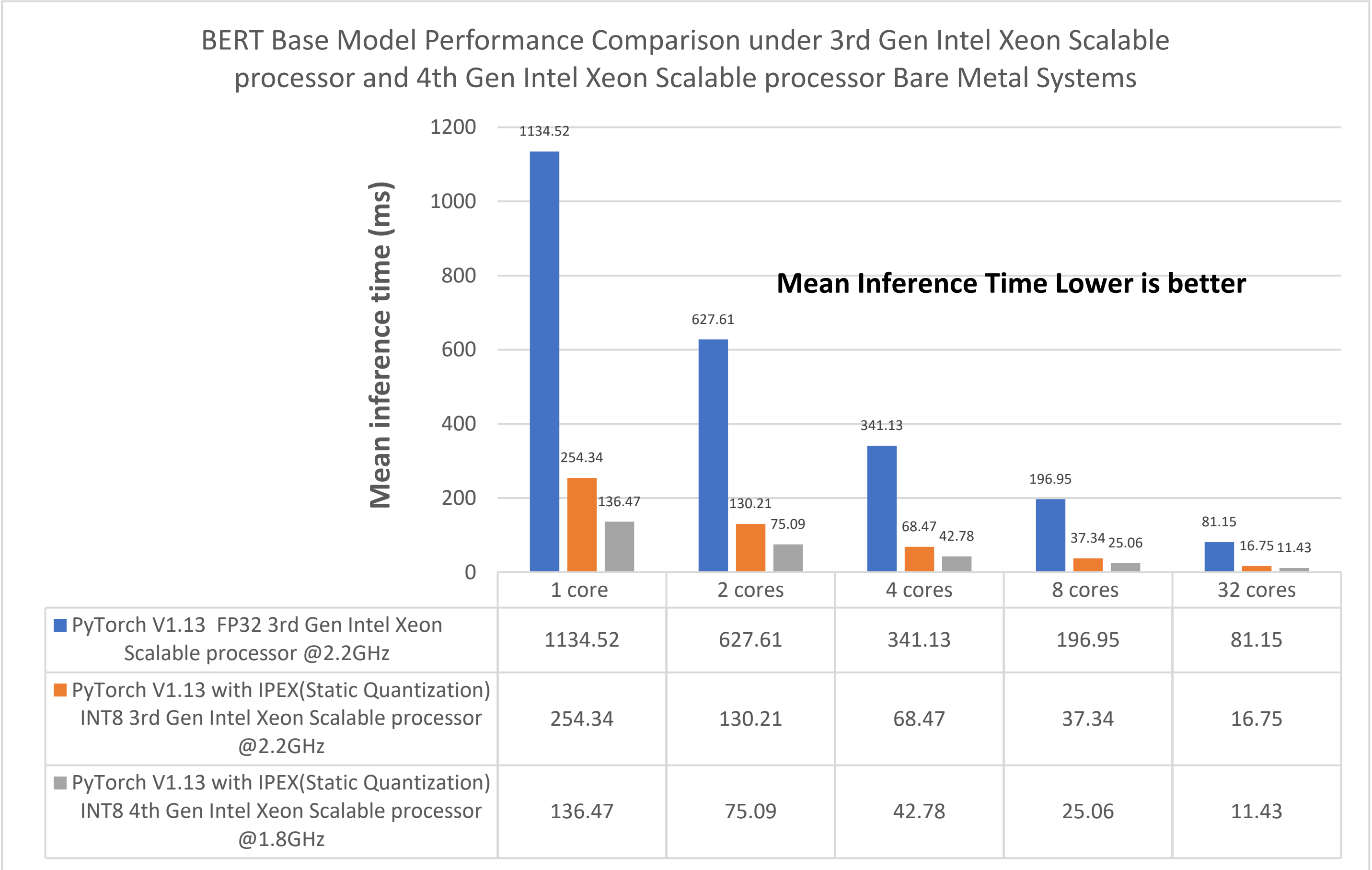
3. Run the following commands under 3rd Gen Intel Xeon Scalable processor and 4th Gen Intel Xeon Scalable processor servers to get the mean inference time for different cores.

```
# numactl -C core_range python test.py
```

4. You should get similar results as shown in Table 3.

Table 3. Result of performance testing based on Hugging Face BERT base model under 3rd Gen Intel Xeon Scalable processor and 4th Gen Intel Xeon Scalable processor (See Appendix for configuration details)

Mean inference time (ms) max_seq_length = 512 Batch size = 1	PyTorch V1.13 FP32	PyTorch V1.13 with IPEX (Static Quantization) INT8		Performance Boost 4th Gen Intel Xeon Scalable processor @1.8GHz vs 3rd Gen Intel Xeon Scalable processor @2.2GHz under PyTorch
	3rd Gen Intel Xeon Scalable processor @2.2GHz	3rd Gen Intel Xeon Scalable processor @2.2GHz	4th Gen Intel Xeon Scalable processor @1.8GHz	
1 core	1134.52	254.34	136.47	8.31 X
2 cores	627.61	130.21	75.09	8.36 X
4 cores	341.13	68.47	42.78	7.97 X
8 cores	196.95	37.34	25.06	7.86 X
32 cores	81.15	16.75	11.43	7.10 X



	1 core	2 cores	4 cores	8 cores	32 cores
🟦 PyTorch V1.13 FP32 3rd Gen Intel Xeon Scalable processor @2.2GHz	1134.52	627.61	341.13	196.95	81.15
🟧 PyTorch V1.13 with IPEX(Static Quantization) INT8 3rd Gen Intel Xeon Scalable processor @2.2GHz	254.34	130.21	68.47	37.34	16.75
⬜ PyTorch V1.13 with IPEX(Static Quantization) INT8 4th Gen Intel Xeon Scalable processor @1.8GHz	136.47	75.09	42.78	25.06	11.43

Figure 2. BERT base model performance comparison under 3rd Gen Intel Xeon Scalable processor and 4th Gen Intel Xeon Scalable processor (See Appendix for configuration details)

From the above results, the following conclusions are made:

- Intel IPEX static quantization can dramatically improve the performance of BERT base model (cased) on both 3rd Gen Intel Xeon Scalable processor (See Intel® Xeon® Gold 6338N CPU in Appendix A) and 4th Gen Intel Xeon Scalable processor (See Intel® Xeon® Gold 6428N CPU in Appendix A) platforms.
- With the help of Intel AVX512 VNNI instruction set, the inference performance of the BERT base model (cased) under the 3rd Gen Intel Xeon Scalable processor (See Intel® Xeon® Gold 6338N CPU in Appendix A) with IPEX static post-training quantization can be 4.46 X to 5.27 X faster than under PyTorch without quantization.
- With the help of Intel AMX instruction set, the inference performance of the BERT base model (cased) under the 4th Gen Intel Xeon Scalable processor (See Intel® Xeon® Gold 6428N CPU in Appendix A) with IPEX static post-training quantization can be 7.09 X to 8.36 X to faster than the 3rd Gen Intel Xeon Scalable processor (See Intel® Xeon® Gold 6338N CPU in Appendix A) under PyTorch without quantization.

4 Summary

From all above benchmark tests, it is concluded that the performance of BERT base model can significantly improve under PyTorch by applying IPEX static post-training quantization with minimal code changes. With the IPEX optimization, the 4th Gen Intel Xeon Scalable processor (See Intel® Xeon® Gold 6428N CPU in Appendix A) can get 7.09 X ~ 8.36 X performance improvement comparing the 3rd Gen Intel Xeon Scalable processor (See Intel® Xeon® Gold 6338N CPU in Appendix A). This solution also shows that it is very easy and straightforward to boost deep learning models with the help of Intel IPEX library. Moreover, Intel DL Boost including Intel AVX512 VNNI and Intel AMX instructions, as the contributor to the performance improvement, is a standard and universally available feature in the 3rd and 4th Gen Intel Xeon Scalable processors, which means there is no need to attach any auxiliary hardware accelerators.

Appendix A Platform Configuration

New: Gold 6428N: 1-node, 1x Intel Xeon Gold 6428N processor on Eagle Stream with 256 GB (8 slots/ 32GB/ 4800) total DDR5 memory, ucode 0x2b000111, HT on, Turbo off, Ubuntu 22.10, 5.19.0-23-generic, 1x SSDSC2KB240G8, huggingface.co/bert-base-cased, AI Framework PyTorch 1.13.0 and IPEX 1.13.0, run_type: AI inference mean time, Test by Intel as of 11/29/22.

Baseline: Gold 6338N: 1-node, 1x Intel Xeon Gold 6338N processor on Coyote Pass with 128 GB (8 slots/ 16GB/ 3200) total DDR4 memory, ucode 0xd000375, 0x2b000111, HT on, Turbo off, Ubuntu 22.10, 5.19.0-23-generic, 1x SSDSC2KB240G8 huggingface.co/bert-base-cased, AI Framework PyTorch 1.13.0 and IPEX 1.13.0, run_type: AI inference mean time, Test by Intel as of 11/29/22.

Appendix B Software Configuration

Software Configuration	Config 1 (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/	https://pytorch.org/
Topology or ML algorithm (include link)	https://huggingface.co/bert-base-cased	https://huggingface.co/bert-base-cased
Precision (FP32, INT8., BF16)	FP32	INT8
NUMACTL	NUMACTL –c	NUMACTL –c
Command Line Used	python test.py	python test.py

0123/DN/WIT/PDF 764388-001US