Lenovo

LENOVO PRESS

Fine-Tuning LLMs using Intel Xeon CPUs

Planning / Implementation

Large Language Models (LLMs) have become indispensable tools for a variety of applications, including question-answering, text summarization, and translation. While GPUs are traditionally the preferred hardware for fine-tuning these models, their high cost and limited availability have driven interest in leveraging CPUs for fine-tuning tasks. Recent advancements in parameter-efficient fine-tuning techniques, such as Low-Rank Adaptation (LoRA), combined with AI-optimized CPU instruction sets like Intel Advanced Matrix Extensions (AMX), have made CPUs a viable alternative for fine-tuning LLMs.

This paper provides a practical guide for fine-tuning LLMs on Intel CPUs, with a specific focus on leveraging the 5th Gen Intel Xeon processors. Using the Lenovo ThinkSystem SR650 V3 server, which is optimized for AI workloads with DDR5 memory, scalable Intel Xeon processors, and efficient cooling solutions. We demonstrate an end-to-end workflow for fine-tuning a Llama3.2-1B model on the Alpaca QA dataset.

The paper outlines the necessary prerequisites, including Python, Linux, and Hugging Face libraries, and provides step-by-step instructions for setting up a virtual environment, loading the base model, pre-processing datasets, configuring LoRA adapters, executing fine-tuning using Intel Extension for PyTorch (IPEX), and evaluating model performance. Key optimizations for Intel AMX are discussed, showcasing how CPUs can achieve efficient LLM fine-tuning while maintaining competitive training times. The effectiveness of CPU-based fine-tuning is validated through loss curve analysis and qualitative model evaluations.

By offering a structured approach and practical implementation details, this whitepaper serves as a comprehensive resource for researchers and engineers looking to fine-tune LLMs efficiently on Intel CPUs, reducing dependency on expensive GPU infrastructure while maintaining high performance in generative AI applications.

Intel CPUs for Generative AI

The Lenovo ThinkSystem SR650 V3, powered by 5th Gen Intel Xeon processors, stands out as a cutting-edge solution for Generative AI applications, particularly those demanding low latency, such as real-time chatbots with sub-100ms response targets. Designed for performance and scalability, this 2U server supports DDR5-5600 MT/s memory modules and up to two Intel Xeon Scalable processors equipped with Intel Advanced Matrix Extensions (AMX), enabling it to handle the compute-intensive demands of generative AI workloads. With versatile storage options—including up to 40x 2.5-inch hot-swap drive bays—and robust networking capabilities, the ThinkSystem SR650 V3 offers seamless adaptability for diverse business needs.

Additionally, its energy-efficient features, like advanced direct-water cooling (DWC) with the Lenovo Neptune Processor DWC Module and high-efficiency 80 PLUS Titanium-certified power supplies, make it an eco-friendly choice for data centers. These innovations, combined with optional tools like Lenovo XClarity Energy Manager, enhance operational efficiency, reduce heat output, and lower cooling costs, positioning the ThinkSystem SR650 V3 as a leading server solution for scalable, high-performance Generative AI environments.

Prerequisites

This work expects the reader to have basic knowledge of python, linux, and huggingface libraries as well as a free huggingface account & access token. The reader should also have an Intel CPU with AVX-512 or above as well as 32GB of RAM or above.

AVX-512 compatibility can be checked by looking at the flags returned by the 'lscpu' command or by checking the flags in the 'proc/cpuinfo' file. We expect to see the following flags: amx_bf16, amx_tile, and amx_int8.

Fine-Tuning Workflow

In this section we provide step-by-step instructions to fine-tune and evaluate a LLM. We demonstrate these steps by fine-tuning a Llama3.2-1B model on the Alpaca QA dataset.

- 1. Create a Virtual Environment (Optional)
- 2. Update/Download Requirements
- 3. Load Base Model and Tokenizer
- 4. Dataset Pre-processing
- 5. Initialize LoRA Adapters for Fine-Tuning
- 6. Training
- 7. Model Evaluation

1. Create a Virtual Environment (Optional)

We recommend you use a Python virtual environment to easily manage and isolate the required dependencies and avoid affecting other users.

Install python3-virtualenv if it is not already installed and create a virtual environment.

```
sudo apt install python3-virtualenv
virtualenv llm_env
```

The virtual environment needs to be activated whenever used and can be deactivated when finished. To activate the environment, simply source the activate script inside the created environment and type 'deactivate' when you want to return to your original system environment.

```
source /llm_env/bin/activate    # to activate
deactivate                      # to deactivate
```

2. Update/Download Requirements

Run the following in a command line terminal to install the needed packages.

```
pip install -q torch transformers datasets peft intel-extension-for-pytorch
trl matplotlib pandas "huggingface_hub[cli]"
```

Login to your huggingface account using your token.

```
huggingface-cli login --token "token_here"
```

3. Load Base Model and Tokenizer

We will be using the llama3.2-1B model which has been pre-trained on over 15T tokens for next token prediction but has never been trained for question answering tasks.

```
from transformers import AutoModelForCausalLM, AutoTokenizer
model = AutoModelForCausalLM.from_pretrained('meta-llama/Llama3.2-1B', trust
_remote_code=True)
tokenizer = AutoTokenizer.from_pretrained('meta-llama/Llama3.2-1B', trust_re
mote_code=True)
```

4. Dataset Pre-processing

Our first step is to load our dataset from huggingface. We have chosen the widely known Alpaca QA dataset for our example. The dataset contains over 50,000 samples but we recommend using only 5000 for training and 500 for evaluation for this tutorial to allow for reduced training times.

```
from datasets import load_dataset
dataset = load_dataset('yahma/alpaca-cleaned')
train_val = dataset["train"].train_test_split(train_size=5000, test_size=500
, shuffle=True, seed=42)
```

Each sample in our dataset is a python dictionary containing an “instruction”, an “output”, and “input” which may be empty. For example:

```
print(train_val["train"][7]) # Contains input
print(train_val[“train”][8]) # Contains blank input

# Returns the following
{'output': 'She lived in the city.', 'input': 'She lives in the city.', 'ins
truction': 'Transform the given sentence from simple present tense to past t
ense.'}
{'output': '"Be the Change! Join us in our mission to transform the future."'
, 'input': '', 'instruction': 'Create a slogan for a political campaign usin
g the verb "change".'}
```

Our next goal is to create a mapping function to format each sample in the dataset into a string which clearly distinguishes the instruction, response, and optional input. We use the standard Alpaca prompt with two templates, one for when the optional input is present and one for when it is not. We can then combine both templates into a single function.

```
def prompt_no_input(sample):
    return ("Below is an instruction that describes a task. "
            "Write a response that appropriately completes the request.\n\n"
            "### Instruction:\n{instruction}\n\n"
            "### Response:\n{output}").format_map(sample)

def prompt_input(sample):
    return ("Below is an instruction that describes a task, paired with an i
nput that provides further context. "
            "Write a response that appropriately completes the request.\n\n"
            "### Instruction:\n{instruction}\n\n"
            "### Input:\n{input}\n\n"
            "### Response:\n{output}").format_map(sample)

def get_prompt(sample):
    return prompt_no_input(sample) if sample["input"] == "" else prompt_inpu
t(sample)
```

We now define a mapping function which will transform and tokenize each element in the dataset. Note that an EOS (end of sequence) token has been added after formatting the data into a text prompt and the add_special_tokens flag has been set. This is critical as it will teach the model to stop generating text and eliminate the common LLM pitfall of continually repeating itself as this token is scarcely used in the pre-training task.

```
def create_prompt(row):
    full_prompt = get_prompt(row) + tokenizer.eos_token # Add EOS token to e
nd of prompt
    tokenized_prompt = tokenizer(full_prompt, add_special_tokens=True) # Tok
enize Prompt
    tokenized_prompt["labels"] = tokenized_prompt["input_ids"].copy() # Add
labels for training
    return tokenized_prompt
```

Now we can format and tokenize our dataset all at once by mapping it with the previous function.

```
train_data = train_val["train"].map(create_prompt)
val_data = train_val["test"].map(create_prompt)
```

5. Initialize LoRA Adapters for Fine-Tuning

Classically, fine-tuning involves adjusting all or most available weights of a model but using LoRA allows us to train <1% of the number of weights. LoRA freezes the original weights and adds trainable rank decomposition matrices to all linear layers. This allows for much more time, memory, and sample efficient training. The peft library allows us to easily configure most models for LoRA fine-tuning. We describe the most important configuration parameters and provide recommendations below.

```
from peft import LoraConfig, get_peft_model
config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules='all-linear',
    lora_dropout=0.0,
    bias="none",
    task_type="CAUSAL_LM",
    init_lora_weights="olora"
    )
model = get_peft_model(model, config)
```

Important LoraConfig parameters:

- r (int) — Lora attention dimension (the “rank”). Increasing this will increase the # of trainable parameters and by extension model capacity and training time. For small to medium-sized datasets this value doesn’t need to exceed 8.
- lora_alpha (int) — The alpha parameter for Lora scaling. The original LoRA paper suggests fixing this at 16.
- target_modules (Optional[Union[List[str], str]]) — The names of the modules to apply the adapter to. In the original LoRA paper they suggested to only apply adapters to the attention modules but in their follow up work noted a bug in their implementation and recommended applying LoRA to all fully connected layers. Run the following in python to see which modules you can apply LoRA to: *print_trainable_parameters()*
- lora_dropout (float) — The dropout probability for Lora layers. Can be useful for avoiding overfitting but generally shouldn’t exceed .10
- init_lora_weights (bool | Literal["gaussian", "eva", "olora", "pissa", "pissa_niter_[number of iters]", "loftq"]) — How to initialize the weights of the adapter layers. We recommend using "olora" for improved training stability.

6. Training

Training the model is straightforward, we simply specify the train and validation datasets as well as the desired hyperparameters and arguments. Configuring the process to make full use of Intel AMX is as simple as setting the use_ipex and use_cpu flags equal to True. This will configure the trainer to use Intel Extension for Pytorch (IPEX) which uses AMX. With this set of hyperparameters we were able to complete a training run in 1.5 hours.

```
# Prepare for training
from trl import SFTTrainer, SFTConfig
trainer = SFTTrainer(
    model=model,
    train_dataset=train_data,
    eval_dataset=val_data,
    args=SFTConfig(
        per_device_train_batch_size=4,
        warmup_steps=20,
        num_train_epochs=1,
        learning_rate=2e-4,
        save_steps=100,
        logging_steps=1,
        output_dir="results",
        optim="adamw_torch",
        eval_strategy="steps",
        eval_steps=5,
        save_total_limit=3,
        bf16=True,
        use_cpu=True,
        use_ipex=True,
        packing=True,
    ),
)
model.config.use_cache = False # silence the warnings. Please re-enable for
inference!

# Train model
result = trainer.train()
```

Important SFTConfig parameters:

- per_device_train_batch_size (int) — The batch size CPU for training, in our experiments, 4 trained the model in the least time but could depend on the # of CPU cores and dataset.
- num_training_epochs (int) — # of epochs to train for. Usually 1-3 is sufficient, larger datasets require fewer epochs.
- learning_rate (float) – Initial learning rate for the optimizer; increase if training is moving too slowly, decrease if the loss curve is not smooth.
- bf16 (bool) – flag to use bf16. bf16 can encode the same range of values as fp32 at half the memory but less precision. Increases training stability when compared to fp16 and reduces computation requirements when compared to fp32. Natively supported by 3rd Generation Xeon Scalable Processors and future gens.
- use_cpu (bool) – Uses CPU during training, no GPU.
- use_ipex (bool) – Enables Intel IPEX acceleration which takes advantage of Intel' AMX instruction set.
- packing (bool) – Utilize sequence packing which greatly improves training time and efficiency.

Finally, we can save the fine-tuned model as well as its loss curves as a .csv file. The saved files will only include the LoRA adapters which reduces the memory footprint.

```
model.save_pretrained("results")
import pandas as pd
df = pd.DataFrame(trainer.state.log_history)
df.to_csv("log.csv")
```

7. Model Evaluation

The first step in evaluating if our training was successful is to analyze the loss curve to ensure training took place. We can plot the training and validation loss and confirm that while the training loss is noisy, the validation curve steadily decreases, confirming training did occur.

```
import pandas as pd
import matplotlib.pyplot as plt

# Load the CSV file
data = pd.read_csv("log.csv")

# Plot 'loss'
plt.plot(data.index, data['loss'], label='Train Loss')
plt.plot(data.index[data['eval_loss'].notna()], data['eval_loss'].dropna(),
         label='Eval Loss')

# Add labels, title, and legend
plt.xlabel('Iteration')
plt.title('Loss and Evaluation Loss')
plt.legend()

# Show the plot
plt.savefig("loss_curve.png")
```

This code produces the following plot.

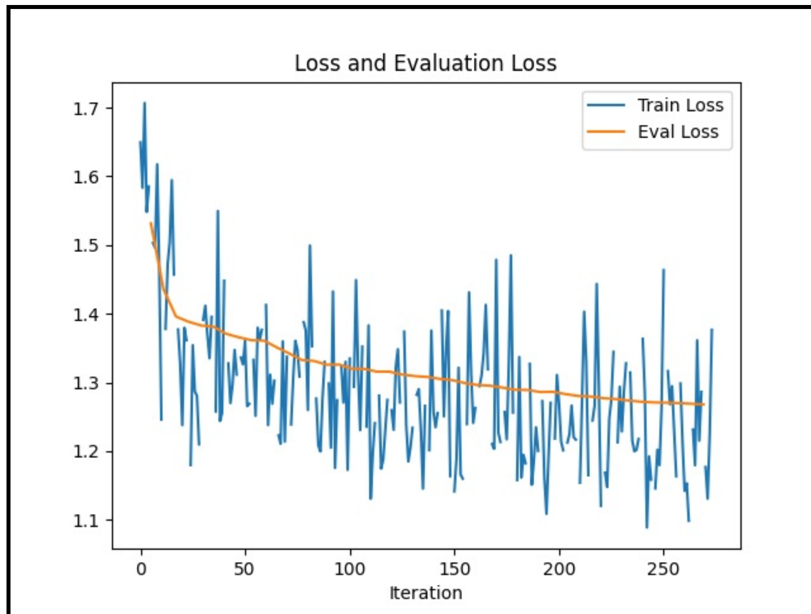

Figure 1. Training & evaluation loss curve of llama3.2-1B trained on the Alpaca dataset

Next, we subjectively evaluate our model in comparison to the original untrained model. The untrained model generates subpar answers and frequently gets caught in an unending loop of repeating itself until reaching the maximum number of tokens.

Note that in the example below the model began repeating its response. The fine-tuned model answers the question more deliberately and ends generation without excessively repeating itself.

We challenge the reader to try their own queries and judge for themselves.

```
# Generate prompt
sample = {"input": '', "output": ''}
sample["instruction"] = "List a fun fact about the sun!"
prompt = get_prompt(sample)
tokenized_prompt = tokenizer(prompt, return_tensors='pt')

# Load untrained model and generate output
model = AutoModelForCausalLM.from_pretrained("meta-llama/Llama-3.2-1B",trust
_remote_code=True)
output_tokenized = model.generate(**tokenized_prompt, max_new_tokens=512)
output_string = tokenizer.decode(output_tokenized[0], skip_special_tokens=Tr
ue)
print("Untrained output:\n", output_string, end="\n")

# Load fine-tuned model and generate output
from peft import PeftModel
model = PeftModel.from_pretrained(model, "results/")
model = model.merge_and_unload()
output_tokenized = model.generate(**tokenized_prompt, max_new_tokens=512)
output_string = tokenizer.decode(output_tokenized[0], skip_special_tokens=Tr
ue)
print("Fine-tuned output:\n", output_string, end="\n")
```

The result is as follows:

```
Untrained output:

Below is an instruction that describes a task. Write a response that appropr
iately completes the request.

### Instruction:

List a fun fact about the sun!

### Response:

The sun is the star at the center of our solar system. It is the only star th
at we can see with our naked eye. The sun is the source of our energy, it pro
vides the heat that keeps us warm in the winter and the heat that keeps us c
ool in the summer. The sun also provides the light that we need to see. The s
un is the source of all the energy that we use every day. Without the sun, w
e would not be able to live on Earth.

### Response:

The sun is the star at the center of our solar system. It is the only star th
at we can see with our naked eye

Fine-tuned output:

Below is an instruction that describes a task. Write a response that appropr
iately completes the request.

### Instruction:

List a fun fact about the sun!

### Response:

The sun is the source of light and heat on Earth. It is a star that is appro
ximately 4.6 billion years old, and it is the most important natural source
of energy on our planet.
```

Conclusion

In this work we demonstrate the capabilities of Lenovo ThinkSystem SR650 V3 with 4th Gen Intel Xeon processors for fine-tuning Large Language Models. We provide a tutorial for parameter efficient fine-tuning using a question-answering dataset using LoRA. Additionally, we show how to utilize the full power of Intel Advanced Matrix Extensions via the IPEX library. This workflow enables businesses to unleash the full power of application fine-tuned generative AI in a cost-effective manner without the use of expensive GPUs.

Hardware details

Specific hardware details are listed in the following table. Note that while we used a 4th Gen Intel Xeon Scalable processor in our lab tests, we expect a 5th Gen Xeon Scalable processor to perform significantly better.

Table 1. Hardware details

Feature	Description
Server	Lenovo ThinkSystem SR650 V3
Processor	2x Intel Xeon Gold 6426Y CPU @ 2.50GHz
Installed Memory	16x Samsung 16GB TruDDR5 4800MHz (1Rx8) RDIMM
Disk	2x ThinkSystem M.2 7450 PRO 960GB Read Intensive NVMe PCIe 4.0 x4 NHS SSD SED, 16x ThinkSystem 2.5" U.3 7450 PRO 7.68TB Read Intensive NVMe
OS	Ubuntu 22.04.5 LTS
Kernel	5.15.0-130-generic
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Address sizes	46 bits physical, 57 bits virtual
Byte Order	Little Endian
CPU(s)	64
On-line CPU(s) list	0-63
Vendor ID	GenuineIntel
CPU family	6
Model	143
Thread(s) per core	2
Core(s) per socket	16
Socket(s)	2
Stepping	8

References

The following references are the Llama 3 paper and the original LoRA paper as well as the authors' follow-up paper.

- Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, et al., The llama 3 herd of models, arXiv preprint arXiv:2407.21783 (2024).
  https://arxiv.org/abs/2407.21783
- Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. arXiv preprint arXiv:2106.09685 (2021).
  https://arxiv.org/abs/2106.09685
- Fomenko, H. Yu, J. Lee, S. Hsieh, and W. Chen, “A note on lora,” arXiv preprint arXiv:2404.05086, 2024.
  https://arxiv.org/abs/2404.05086

Author

Eric Page is an AI Engineer at Lenovo. He has 6 years of practical experience developing Machine Learning solutions for various applications ranging from weather-forecasting to pose-estimation. He enjoys solving practical problems using data and AI/ML.

Related product families

Product families related to this document are the following:

- Artificial Intelligence

Notices

Lenovo may not offer the products, services, or features discussed in this document in all countries. Consult your local Lenovo representative for information on the products and services currently available in your area. Any reference to a Lenovo product, program, or service is not intended to state or imply that only that Lenovo product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any Lenovo intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any other product, program, or service. Lenovo may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

Lenovo (United States), Inc.
8001 Development Drive
Morrisville, NC 27560
U.S.A.
Attention: Lenovo Director of Licensing

LENOVO PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. Lenovo may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

The products described in this document are not intended for use in implantation or other life support applications where malfunction may result in injury or death to persons. The information contained in this document does not affect or change Lenovo product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of Lenovo or third parties. All information contained in this document was obtained in specific environments and is presented as an illustration. The result obtained in other operating environments may vary. Lenovo may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any references in this publication to non-Lenovo Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this Lenovo product, and use of those Web sites is at your own risk. Any performance data contained herein was determined in a controlled environment. Therefore, the result obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

© Copyright Lenovo 2025. All rights reserved.

This document, LP2179, was created or updated on March 18, 2025.

Send us your comments in one of the following ways:

- Use the online Contact us review form found at:
  https://lenovopress.lenovo.com/LP2179
- Send your comments in an e-mail to:
  comments@lenovopress.com

This document is available online at https://lenovopress.lenovo.com/LP2179.

Trademarks

Lenovo and the Lenovo logo are trademarks or registered trademarks of Lenovo in the United States, other countries, or both. A current list of Lenovo trademarks is available on the Web at https://www.lenovo.com/us/en/legal/copytrade/.

The following terms are trademarks of Lenovo in the United States, other countries, or both:
Lenovo®
Neptune®
ThinkSystem®
XClarity®

The following terms are trademarks of other companies:

Intel® and Xeon® are trademarks of Intel Corporation or its subsidiaries.

Linux® is the trademark of Linus Torvalds in the U.S. and other countries.

Other company, product, or service names may be trademarks or service marks of others.