Deploying LLMs Module

Deploying LLMs

Goals: Familiarize students with efficient LLM deployment techniques, emphasizing quantization and pruning. Offer hands-on experiences with deployments on platforms like GCP and Intel® CPUs.

This module dives into deploying Large Language Models. Model **quantization and pruning** are central to these strategies, each serving as an effective tool to optimize model performance without compromising efficiency. With a foundation in research articles and real-world applications, this module also introduces participants to deployment on cloud platforms.

- **Challenges of LLM deployment**: The lesson covers challenges during LLM deployment such as the sheer size of models, associated costs, and potential latency issues. We also provide a survey on optimizations, rooted in a research article on Transformer inference and a deepened perspective on potential solutions in addressing these challenges.
- **Model Quantization**: This lesson centers on Quantization, highlighting its role in streamlining LLM deployments. We research the balance between model performance and efficiency by understanding its usefulness and various techniques.
- **Model Pruning**: This module discusses model pruning, showcasing its place in LLM optimization. We will introduce various pruning techniques backed by recent research.
- **Deploying an LLM on a Cloud CPU**: This module uncovers the advantages, considerations, and challenges of deploying large language models on cloud-based CPUs. This lesson needs a server instance equipped with an Intel® Xeon® processor.

By the end of this module, students have gained a robust understanding of the intricacies involved in LLM deployment. The exploration of model quantization, pruning, and practical deployment strategies has provided them with the tools necessary to navigate real-world challenges. Moving beyond foundational concepts, the next section offers a deep dive into the advanced topics and future directions in the realm of LLMs.

After navigating the diverse terrain of Transformers and LLMs, participants now deeply understand significant architectures like GPT and BERT. The sessions shed light on model evaluation metrics, advanced control techniques for optimal outputs, and the roles of pretraining and finetuning. The upcoming module dives into the complexities of deciding when to train an LLM from scratch, the operational necessities of LLMs, and the sequential steps crucial for the training process.

Challenges of LLM Deployment

Introduction

In this lesson, we will study the challenges of deploying Large Language Models, with a focus on the importance of latency and memory. We also explore optimization techniques with concepts like quantization and sparsity and how they can be applied using the Hugging Face Optimum and Intel® Neural Compressor libraries. We also discuss the role of Intel's® optimization technologies in efficiently running LLMs. This lesson will provide a deeper understanding of how to optimize LLMs for better performance and user experience.

Importance of Latency and Memory

Latency is the delay before a transfer of data begins following an instruction. It is a crucial factor in LLM applications. High latency in real-time or near-real-time applications can lead to a poor user experience.

For instance, in a conversational AI application, a delay in response can disrupt the natural flow of conversation, leading to user dissatisfaction. Therefore, reducing latency is a critical aspect of LLM deployment.

Considering an average human reading speed of ~250 words per minute (translated to ~312 tokens per minute) that is about 5 tokens per second, therefore a latency of 200ms per token. Usually, acceptable latency for near-real-time LLM applications is between 100ms and 200ms per token.

Transformers can be computationally intensive and memory-demanding, due to their complex architecture and large size. However, several optimization techniques can be employed to enhance their efficiency without significantly compromising their performance.

Quantization

Quantization is a technique used for compressing neural network models, including Transformers, by lowering the precision of model parameters and/or activations. This method can significantly reduce memory usage. It leverages low-bit precision arithmetic and decreases the size, latency, and energy consumption.

However, it's important to strike a balance between performance gains through reduced precision and maintaining model accuracy. Techniques such as mixed-precision quantization, which assign higher bit precision to more sensitive layers, can mitigate accuracy degradation.

We'll learn different quantization methods later in the course.

Sparsity

Sparsity, usually achieved by pruning, is another technique for reducing the computational cost of LLMs by eliminating redundant or less important weights and activations. This method can significantly decrease off-chip memory consumption, the corresponding memory traffic, energy consumption, and latency.

Pruning can be broadly divided into types: weight pruning and activation pruning.

- **Weight pruning** can be further categorized into unstructured pruning and structured pruning. Unstructured pruning allows any sparsity pattern, and structured pruning imposes an additional constraint on the sparsity pattern. While structured pruning can provide benefits in terms of memory, energy consumption, and latency without additional hardware support, it is known to achieve a lower compression rate than unstructured pruning.
- On the other hand, **activation pruning** prunes redundant activations during inference, which can be especially effective for Transformer models. However, this requires support to detect and zero out unimportant activations at run-time dynamically.

  We'll study different pruning methods later in the course.

Utilizing Optimum and Intel® Neural Compressor Libraries

The Hugging Face Optimum and the Intel® Neural Compressor libraries provide a suite of tools helpful in optimizing models for inference, especially for Intel® architectures.

- The Hugging Face Optimum library serves as an interface between the Hugging Face transformers and diffuser libraries and the various tools provided by Intel®.
- The Intel® Neural Compressor is an open-source library that facilitates the application of popular compression techniques such as quantization, pruning, and knowledge distillation. It supports automatic accuracy-driven tuning strategies, enabling users to generate quantized models easily. This library allows users to apply static, dynamic, and aware-training quantization approaches while maintaining predefined accuracy criteria. It also supports different weight pruning techniques, allowing for the creation of pruned models that meet a predefined sparsity target.

  These libraries provide a practical application of the quantization and sparsity techniques, and their usage will be of great use in optimizing the deployment of LLMs.

Intel® Optimization Technologies for LLMs

Intel's® optimization technologies play a significant role in running LLMs efficiently on CPUs. The 4th Gen Intel® Xeon® Scalable processors are equipped with AI-infused acceleration known as Intel® Advanced Matrix Extensions (Intel® AMX). These processors have built-in BF16 and INT8 GEMM (general matrix-matrix multiplication) accelerators in every core, which significantly accelerate deep learning training and inference workloads.

The Intel® Xeon® Proecssor Max Series offers up to 128GB of high-bandwidth memory, which is particularly beneficial for LLMs, as these models are often memory-bandwidth bound.

By (1) running model optimizations like quantization and pruning and (2) leveraging the Intel® hardware acceleration technologies, it's possible to achieve a good latency for LLMs too. Take a look at this page to see the performance improvements (better throughput, with less memory size) of several optimized models.

Conclusion

In this lesson, we have explored the challenges of deploying Large Language Models, with a particular focus on latency and memory.

We also discussed optimization techniques like quantization and sparsity, which can significantly reduce LLMs' computational cost and memory usage.

Model Quantization

Introduction

As AI models, including large language models, grow more advanced, their increasing number of parameters leads to significant memory usage. This, in turn, increases the costs of hosting and deploying these tools.

In this lesson, we will learn about **quantization,** a process that can be employed to **diminish the memory requirements** of these models. We will explore the various types of quantization, such as scalar and product quantization. We will also learn how fine-tuning techniques like QLoRA use quantization.

Finally, we will examine applying these techniques to AI models using a CPU with methods implemented in the Intel® neural compressor library.

Overview of Quantization

In deep learning, quantization is a technique that **reduces the numerical precision** of model parameters, such as the weights and biases. This reduction helps decrease the model's memory footprint and computational requirements, enabling easier deployment on resource-constrained devices such as mobile phones, smartwatches, and other embedded systems.

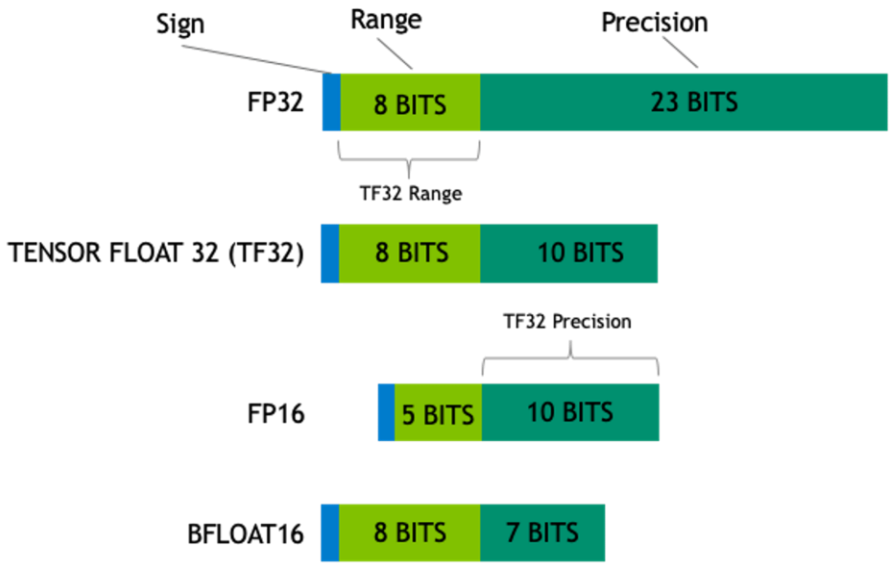
Everyday Example

To understand the concept of quantization, consider an everyday scenario. Imagine two friends, Jay and John. Jay asks John, "What's the time?" John can reply with the exact time, 10:58 p.m., or he can say it's around 11 p.m. In the latter response, John simplifies the time, making it less precise but easier to communicate and understand. This is a basic example of quantization, which is analogous to the process in deep learning, where the precision of model parameters is reduced to make the model more efficient, albeit at the cost of some accuracy.

Quantization in Machine Learning

In Machine Learning, different floating point data types can be used for model parameters, a characteristic also called precision. The precision of the data types affects the amount of memory required by the model. Defining the parameters in higher precision types, like Float32 or Float64, provides greater accuracy but

requires more memory, while lower precision types, like Float16 or BFloat16, use less memory but may result in a loss of accuracy.
In the figure below, you can see the main floating point data types.



From "A Gentle Introduction to 8-bit Matrix Multiplication for transformers at scale using Hugging Face Transformers, Accelerate and bitsandbytes" blog post.

We can estimate the memory required for an AI model with its number of parameters. For example, consider the **Llama2 70B** model that uses Float16 precision for its parameters. Each parameter requires **two bytes**. To calculate the memory required in gigabytes (GB), where 1GB = 1024^3 bytes, the calculation is as follows:

$$(70{,}000{,}000{,}000 * 2)/1024^3 = 130.385 GB$$

Now, let's explore the different basic quantization techniques.

Scalar Quantization

In scalar quantization, each dimension of the dataset is treated independently. The maximum and minimum values are calculated for each dimension across the dataset. The range between the maximum and minimum values in each dimension is then divided into equal-sized bins. Each value in the dataset is mapped to one of these bins, effectively quantizing the data.
For example, consider a dataset of 2000 vectors with 256 dimensions sampled from a Gaussian distribution. The goal is to perform scalar quantization on this dataset.

```
import numpy as np

dataset = np.random.normal(size=(2000, 256))

# Calculate and store minimum and maximum across each dimension

ranges = np.vstack((np.min(dataset, axis=0), np.max(dataset, axis=0)))
```

Now, calculate each dimension's start value and step size. The start value is the minimum value, and the step size is determined by the number of discrete bins in the integer type being used. This example uses 8-bit unsigned integers (`uint8`), providing 256 bins.

```
starts = ranges[0,:]

steps = (ranges[1,:] - ranges[0,:]) / 255
```

The quantized dataset is then calculated as follows:

```
scalar_quantized_dataset = np.uint8((dataset - starts) / steps)Copy
```

The overall scalar quantization process can be encapsulated in a function:

```
def scalar_quantisation(dataset):
    # Calculate and store minimum and maximum across each dimension
    ranges = np.vstack((np.min(dataset, axis=0), np.max(dataset, axis=0)))
    starts = ranges[0,:]
    steps = (ranges[1,:] - starts) / 255
    return np.uint8((dataset - starts) / steps)Copy
```

Product Quantization

In scalar quantization, the data distribution in each dimension should ideally be considered to avoid loss of information. Product quantization can preserve more information by dividing each vector into sub-vectors and quantizing each sub-vector independently.
For example, consider the following array:
Copy

```
array = [ [ 8.2, 10.3, 290.1, 278.1, 310.3, 299.9, 308.7, 289.7, 300.1],
                                        [ 0.1, 7.3, 8.9, 9.7, 6.9, 9.55, 8.1, 8.5, 8.99] ]Copy
```

Quantizing this array to a 4-bit integer using scalar quantization results in significant information loss:
Copy

```
quantized_array = [[ 0  0 14 13 15 14 14 14 14]
                                                                [ 0  0  0  0  0  0
0  0  0]]Copy
```

In contrast, **product quantization** involves the following steps:

1. Divide each vector in the dataset into *m* disjoint sub-vectors.
2. For each sub-vector, cluster the data into *k* centroids (using *k*-means, for example).
3. Replace each sub-vector with the index of the nearest centroid in the corresponding codebook.

Let's proceed with the Product Quantization of the given array with m=3 (number of sub-vectors) and k=2 (number of centroids)
Copy

```
from sklearn.cluster import KMeans
import numpy as np

# Given array
array = np.array([
    [8.2, 10.3, 290.1, 278.1, 310.3, 299.9, 308.7, 289.7, 300.1],
    [0.1, 7.3, 8.9, 9.7, 6.9, 9.55, 8.1, 8.5, 8.99]
])

# Number of subvectors and centroids
m, k = 3, 2
```

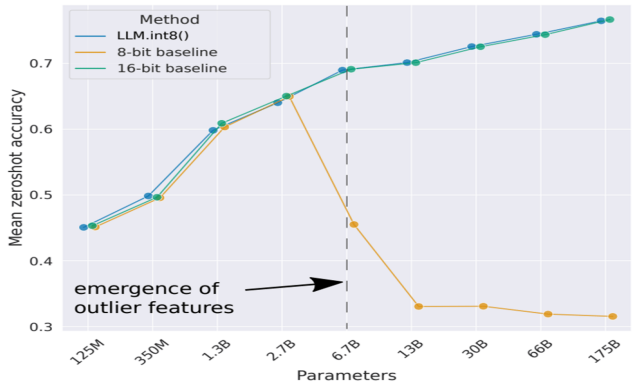
```
# Divide each vector into m disjoint sub-vectors
subvectors = array.reshape(-1, m)

# Perform k-means on each sub-vector independently
kmeans = KMeans(n_clusters=k, random_state=0).fit(subvectors)

# Replace each sub-vector with the index of the nearest centroid
labels = kmeans.labels_

# Reshape labels to match the shape of the original array
quantized_array = labels.reshape(array.shape[0], -1)

# Output the quantized array
quantized_arrayCopy
```

```
# Result
array([[0, 1, 1],
       [0, 0, 0]], dtype=int32)
```

By quantizing the vectors and storing only the indices of the centroids, the memory footprint is significantly reduced.
This method can help preserve more information than scalar quantization, especially when the distributions of different dimensions are diverse.
Product quantization can significantly reduce memory footprint and speed up the nearest neighbor search but at the cost of accuracy. The tradeoff in product quantization is based on the **number of centroids** and the number of sub-vectors we use. The more centroids we use, the better the accuracy, but the memory footprint would not decrease and vice versa.

Quantizing Large Models

We learned about two relatively basic quantization techniques that can be used with deep learning models. While these simple techniques can work well enough with models with few parameters, they usually lead to a drop in accuracy for larger models with billions of parameters.



From "LLM.int8(): 8bit Matrix Multiplication for Transformers at Scale" paper

Large models contain a **greater amount of information** in their parameters. With more neurons and layers, large models can represent more complex functions. They can capture deeper and more intricate relationships in the data, which smaller models might not be able to handle.
Thus, the quantization process, which reduces the precision of these parameters, can significantly lose this information, resulting in a substantial drop in model accuracy and performance.
Optimizing the quantization process for large models is also more difficult due to the larger parameter space. Finding the optimal quantization strategy that minimizes the loss of accuracy while reducing the model size is a more complex task for larger models.

Popular (Post-Training Quantization) Methods for LLMs

Fortunately, more sophisticated quantization techniques have been released to address these problems, aiming to maintain the accuracy of large models while effectively reducing their size.

LLM.int8()

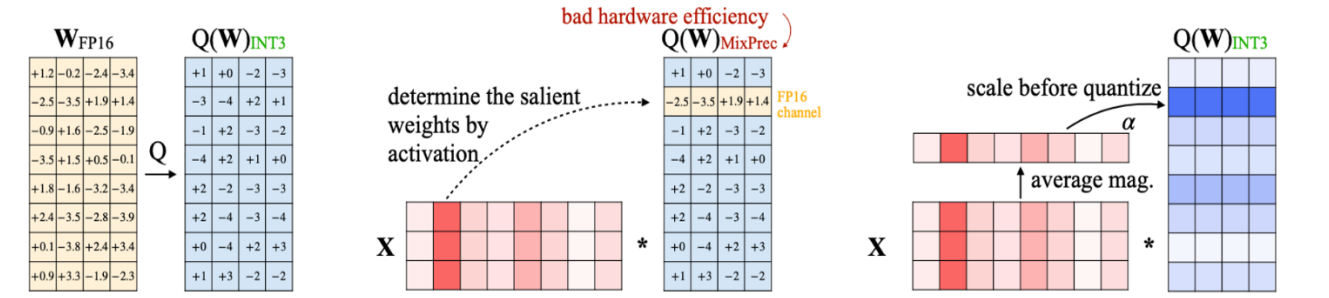
This research paper observes that activation outliers (activation values significantly different from the others) break the quantization of larger models and proposes keeping them in higher precision. By keeping doing that, the performance of the model is not negatively affected.

GPTQ

This technique allows for faster text generation. The quantization is done layer by layer, minimizing the mean squared error (MSE) between the quantized and full-precision weights when given an input.
The algorithm uses a mixed int4-fp16 quantization scheme where weights are quantized as int4 while activations remain in float16. During inference, weights are de-quantized on the fly, and the actual compute is performed in float16. This method makes use of a calibration dataset. The GPTQ algorithm requires calibrating the quantized weights of the model by making inferences on the quantized model.

AWQ

This method is grounded in the observation that not all weights contribute equally to Large Language Models performance. It identifies a small fraction (0.1%-1%) of 'important' or 'salient' weights, the quantization of which, if skipped, can substantially mitigate quantization loss.
Unlike traditional approaches that focus on weight distribution, the AWQ method selects these salient weights based on the magnitude of their activations. This approach leads to a notable enhancement in performance. By maintaining only 0.1%-1% of the weight channels, corresponding to larger activations, in the FP16 format, the method significantly boosts the performance of quantized models.



From "AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration" paper

The authors note that retaining certain weights in FP16 format can cause hardware inefficiency due to using mixed-precision data types. To address this, they propose a method where all weights, including the salient ones, are quantized to avoid mixed-precision data types. However, before the quantization process, the weights are scaled. This scaling step is crucial as it helps protect the outlier weight channels during quantization, ensuring that the important information they hold is not lost or significantly altered during the quantization process. This method aims to strike a balance, allowing the model to benefit from the quantization efficiency while preserving the essential information in the salient weights.

Using Quantized models

Many open-source LLMs are available for download in a quantized format. As we learned in this lesson, these models will have reduced memory requirements.

You can look at the model's section on HuggingFace to find and use a quantized model. This platform hosts a variety of models. For instance, you can try the latest Mistral-7B-Instruct model, which has been quantized using the GPTQ method.

Quantizing Your Own LLM

You can use the Intel® Neural Compressor Library to quantize your own Large Language Model. This library offers various techniques for model quantization, some of which have been discussed in this module.

To get started, follow the step-by-step guide provided in the **repository**. This guide will walk you through quantizing a model, ensuring you have all the necessary components and knowledge to proceed.

Before beginning the quantization process, ensure you have installed the `neural-compressor` library and `lm-evaluation-harness`. Inside the cloned neural compressor directory, navigate to the appropriate directory and install the required packages by running the following commands:

```
cd examples/pytorch/nlp/huggingface_models/language-modeling/quantization/ptq_weight_only

pip install -r requirements.txtCopy
```

As an example, to quantize the `opt-125m` model with the GPTQ algorithm, use the following command:

```
python examples/pytorch/nlp/huggingface_models/language-
modeling/quantization/ptq_weight_only/run-gptq-llm.py \
    --model_name_or_path facebook/opt-125m \
    --weight_only_algo GPTQ \
    --dataset NeelNanda/pile-10k \
    --wbits 4 \
    --group_size 128 \
    --pad_max_length 2048 \
    --use_max_length \
    --seed 0 \
    --gpu
```

This command will quantize the `opt-125m` model using the specified parameters.

How Quantization is used in QLoRA

We saw in a previous lesson how fine-tuning can be achieved using fewer resources using QLoRa, a popular variant of LoRA that makes fine-tuning large language models even more accessible.

In the course, we saw that QLoRA involves backpropagating gradients through a frozen, 4-bit quantized pre-trained language model into Low-Rank Adapters. To accomplish this, QLoRA employs a novel data type, the **4-bit NormalFloat (NF4)**, which is theoretically optimal for normally distributed weights.

This optimality stems from quantile quantization, a technique particularly suited for normally distributed values. It ensures that each quantization bin holds **an equal number of values** from the input tensor, minimizing quantization error and providing a more uniform data representation.

Since pre-trained neural network weights typically exhibit a **zero-centered normal distribution** with a standard deviation (σ), QLoRA transforms all weights into a unified fixed distribution. This transformation is achieved by scaling σ to ensure the distribution aligns perfectly within the range of the NF4 data type, further enhancing the efficiency and accuracy of the quantization process.

This new fine-tuning technique shows no accuracy degradation in their experiments and matches BFloat16 performance.

Table 4: Mean 5-shot MMLU test accuracy for LLaMA 7-65B models finetuned with adapters on Alpaca and FLAN v2 for different data types. Overall, NF4 with double quantization (DQ) matches BFloat16 performance, while FP4 is consistently one percentage point behind both.

	Mean 5-shot MMLU Accuracy								
LLaMA Size	7B		13B		33B		65B		Mean
Dataset	Alpaca	FLAN v2	Alpaca	FLAN v2	Alpaca	FLAN v2	Alpaca	FLAN v2	
BFloat16	38.4	45.6	47.2	50.6	57.7	60.5	61.8	62.5	53.0
Float4	37.2	44.0	47.3	50.0	55.9	58.5	61.3	63.3	52.2
NFloat4 + DQ	39.0	44.5	47.5	50.7	57.3	59.2	61.8	63.9	53.1

From "QLoRA: Efficient Finetuning of Quantized LLMs" paper

Conclusion

In this lesson, we explored the concept of quantization, a technique that can reduce the memory requirements of large models and, in some cases, enhance the text generation speed for language models. We delved into some state-of-the-art quantization techniques suitable for models with billions of parameters, examining the unique contributions of each method.

We also learned how to quantize our own models using the Intel® Neural Compressor Library, which supports many popular quantization methods.

Lastly, we revisited QLoRA, understanding how it leverages quantization to make the fine-tuning of models more accessible to a broader audience.

Model Pruning

Introduction

Deep learning has revolutionized various fields, from computer vision to natural language processing. However, one drawback of deep neural networks is their large size and computational demands. These resource-intensive models can significantly hinder deployment, especially in resource-constrained environments like mobile devices and embedded systems. This is where model pruning comes into play as a powerful technique for reducing the size of neural networks without compromising their performance. In this blog post, we will explore what model pruning is, why it's useful, and various methods to achieve it.

What is Model Pruning?

Model pruning reduces the size of a deep neural network by removing certain neurons, connections, or even entire layers. The goal is to create a smaller and more efficient model while preserving its accuracy to the greatest extent possible. This reduction in model size leads to benefits such as faster inference times, lower memory footprint, and improved energy efficiency, making it ideal for deployment in resource-limited scenarios.

Pruned models are smaller and require fewer computational resources during inference. This is crucial for applications like mobile apps, IoT devices, and edge computing, where computational resources are limited. Moreover, pruned models typically execute faster and are more energy-efficient, enabling real-time applications and improving user experience.

Different Types of Model Pruning

There are several techniques and methodologies for model pruning, each with its own advantages and trade-offs. Some of the commonly used methods include:

Magnitude-based Pruning (or Unstructured Pruning)

In this approach, model weights or activations with small magnitudes are pruned. The intuition is that small weights contribute less to the model's performance and can be safely removed.

The paper titled "Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures" presented this approach to optimize deep neural networks by pruning unimportant neurons. This technique, known as network trimming, is based on the observation that a significant number of neurons in a large network produce zero outputs, regardless of the inputs received. These zero activation neurons are considered redundant and are removed without impacting the overall accuracy of the network. The process involves iterative pruning and retraining of the network, with the weights before pruning used as initialization. The authors demonstrate through experiments on computer vision neural netowrks that this approach can achieve a high compression ratio of parameters without compromising, and sometimes even improving, the accuracy of the original network.

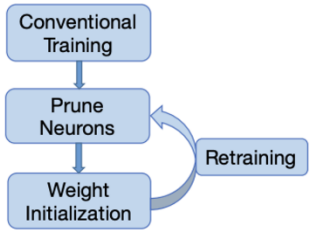


Figure 3: Three main steps for trimming

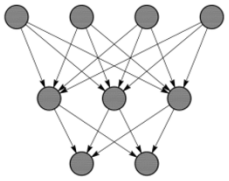


Figure 4: Before pruning

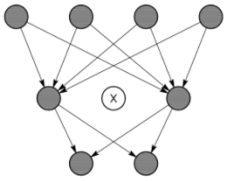
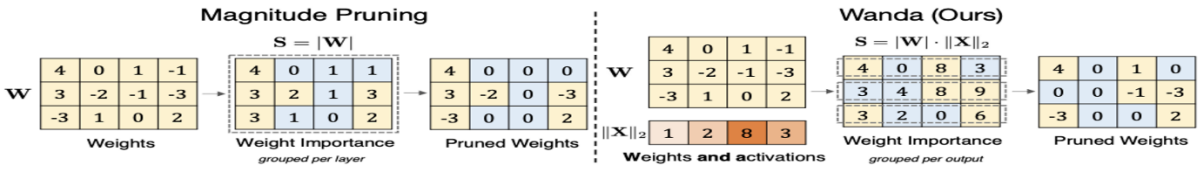


Figure 5: After pruning

The paper "Learning Efficient Convolutional Networks through Network Slimming" presented variations of the pruning scheme for deep convolutional neural networks aimed at reducing the model size, decreasing the run-time memory footprint, and lowering the number of computing operations without compromising accuracy.

The paper “A Simple and Effective Pruning Approach for Large Language Models” introduces a pruning method called Wanda (Pruning by Weights and activations) for pruning Large Language Models. Pruning is a technique that eliminates a subset of network weights to maintain performance while reducing the model's size. Wanda prunes weights based on the smallest magnitudes multiplied by the corresponding input activations, on a per-output basis. This method is inspired by the recent observation of emergent large magnitude features in LLMs. The key advantage of Wanda is that it does not require retraining or weight updates, and the pruned LLM can be used directly.



Structured Pruning

Structured pruning targets specific structures within the network, such as channels in convolutional layers or neurons in fully connected layers.

The paper "Structured Pruning of Deep Convolutional Neural Networks" introduces a new method of network pruning that incorporates structured sparsity at different scales, including channel-wise, kernel-wise, and intra-kernel strided sparsity. This approach is beneficial for computational resource savings. The method uses a particle filtering approach to determine the significance of network connections and paths, assigning importance based on the misclassification rate associated with each connectivity pattern. After pruning, the network is re-trained to compensate for any losses.

The Lottery Ticket Hypothesis

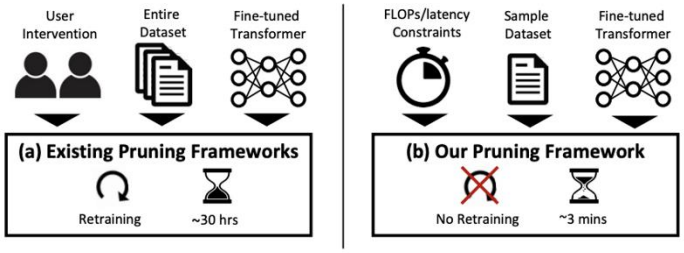
The paper "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks" presents an innovative perspective on neural network pruning, introducing the "Lottery Ticket Hypothesis". This hypothesis suggests that within dense, randomly-initialized, feed-forward networks, there exist smaller subnetworks ("winning tickets") that, when trained separately, can achieve test accuracy similar to the original network in a comparable number of iterations. These "winning tickets" are characterized by their initial weight configurations, which make them particularly effective for training.

The authors propose an algorithm to identify these "winning tickets" and present a series of experiments to support their hypothesis. They consistently discover "winning tickets" that are 10-20% the size of several fully-connected and convolutional feed-forward architectures for MNIST and CIFAR10 datasets. Interestingly, these subnetworks not only match the performance of the original network, but often surpass it, learning faster and achieving higher test accuracy.

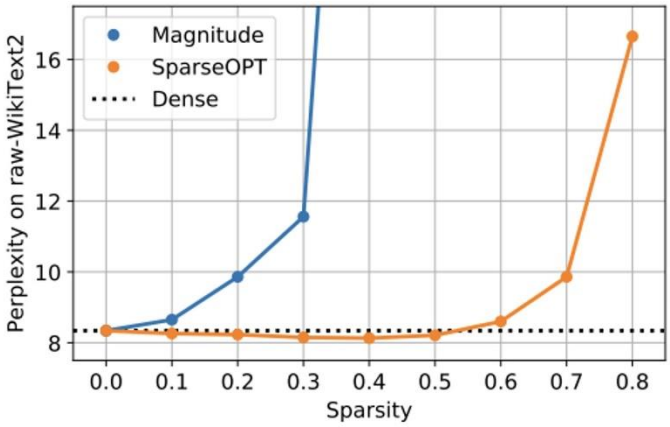
Intel® Neural Compressor Library

The Intel® Neural Compressor Library is valuable for leveraging already implemented model pruning techniques. Read this page to learn more about the pruning methods implemented. Here are a couple of pruning methods specifically for LLMs.

The paper "A Fast Post-Training Pruning Framework for Transformers" presents a fast post-training pruning framework for Transformer models, designed to reduce the high inference cost associated with these models. Unlike previous pruning methods that necessitate model retraining, this framework eliminates the need for retraining, thus reducing both the training cost and complexity of model deployment. The framework uses structured sparsity methods to automatically prune the Transformer model given a resource constraint and a sample dataset. To maintain high accuracy, the authors introduce three new techniques: a lightweight mask search algorithm, mask rearrangement, and mask tuning.



The paper titled "SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot" presents a pruning method, SparseGPT, that can reduce the size of large-scale generative pretrained transformer (GPT) models by at least 50% in a single step, without retraining and with minimal loss of accuracy. The authors demonstrate that SparseGPT can be applied to the very large models OPT-175B and BLOOM-176B, in less than 4.5 hours. The method can achieve 60% unstructured sparsity, meaning that over 100 billion weights can be disregarded during inference without a significant increase in perplexity.



Conclusion

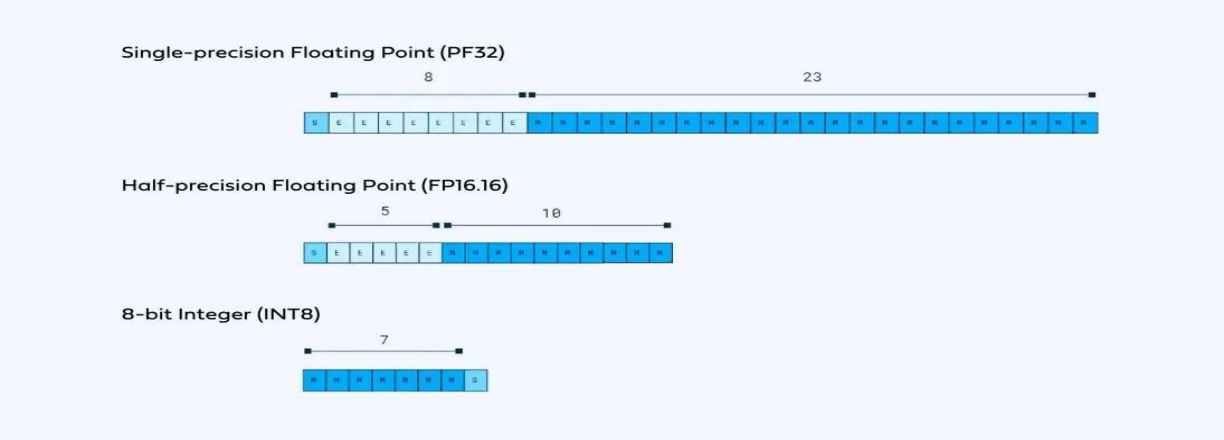
In conclusion, model pruning is a powerful technique for reducing the size of deep neural networks without significantly compromising their performance. It is a valuable tool for deploying models in resource-constrained environments, such as mobile devices and embedded systems. Various pruning methods exist, including magnitude-based pruning and structured pruning, each with its unique advantages and trade-offs. The Intel® Neural Compressor Library provides a practical implementation of these techniques, with specific methods designed for Large Language Models. By understanding and applying these pruning techniques, we can create smaller, faster, and more efficient models that maintain high accuracy, thereby improving the feasibility and user experience of deploying deep learning models in real-world applications.

Deploying an LLM on a Cloud CPU

Introduction

Training a language model can be costly, and expenses associated with deploying it can quickly accumulate over time. Utilizing optimization techniques that enhance the efficiency of the inference process is crucial for minimizing hosting expenses. In this lesson, we will discuss the utilization of the Intel® Neural Compressor library to implement quantization techniques. This approach aims to enhance the cost-effectiveness and speed of models when running on CPU instances (it supports also AMD CPU, ARM CPU, and NVidia GPU through ONNX Runtime, but with limited testing).

Various techniques can be employed for optimizing a network. Pruning involves trimming the parameter count by targeting less important weights, while knowledge distillation transfers insights from a larger model to a smaller one. Lastly, quantization decreases weight precision from 32 bits to 8 bits. It will significantly decrease the memory needed for loading models and generating responses with minimal accuracy loss.



Credit: Deci.ai

The primary focus of this lesson is the quantization technique. We will apply it to an LLM and demonstrate how to perform inference using the quantized model. Ultimately, we will execute several experiments to assess the resulting acceleration.

We'll begin by setting up the necessary libraries. Install the `optimum-intel` package directly from its GitHub repository.

```
pip install git+https://github.com/huggingface/optimum-intel.git@v1.11.0
pip install onnx===1.14.1 neural_compressor===2.2.1
```

The sample code.

Simple Quantization (using CLI)

You can utilize the `optimum-cli` command within the terminal to execute dynamic quantization. Dynamic quantization stands as the recommended approach for transformer-based neural networks. You have the choice to either specify the path to your custom model or select a model from the Huggingface Hub, which will be designated using the `--model` parameter. The `--output` parameter determines the name of the resulting model. We are conducting tests on Facebook's OPT model with 1.3 billion parameters.

```
optimum-cli inc quantize --model facebook/opt-1.3b --output opt1.3b-quantized
```

The script above will automatically load the model and handle the quantization process. I worth noting that if the script fails to recognize your model, you can employ the `--task` parameter. You might use `--task text-generation` for language models. Check the source code for a complete list of supported tasks.
The mentioned approach can be easily incorporated by simply executing a single command. However, for more complex applications where greater control over the process is needed, it might not provide the desired flexibility. In the following section we will use the same Intel Neural Compressor package to perform a more targeted quantization process.

Flexible Quantization (using Code)

As previously discussed, the library includes a constrained quantization method that allows you to specify a precise quantization target. In this approach, we must code and implement the function, which requires more steps while providing more control over the process. For example, you can employ an evaluation function to request quantization of the model while experiencing no more than a 1% decrease in accuracy.
To begin, let's install the necessary packages which also includes the Intel neural compressor from the previous section. This will allow us to load the model and carry out the quantization process.

```
pip install transformers===4.34.0 evaluate===0.4.0 datasets===2.14.5
```

The sample code.
The mentioned packages will help with loading the large language model (transformers), defining an evaluation metric to measure how close we are to the target (evaluate), and importing a dataset for the evaluation process (datasets). Now, we can load the model's weights and its accompanying tokenizer.

```
model_name "aman-mehra/opt-1.3b-finetune-squad-ep-0.4-lr-2e-05-wd-0.01"
tokenizer = AutoTokenizer.from_pretrained(model_name, cache_dir="./opt-1.3b")
model = AutoModelForQuestionAnswering.from_pretrained(model_name, cache_dir="./opt-1.3b")
```

We are initializing a model specifically tailored for the question-answering task. Keep in mind that the model you chose must be fine-tuned for question answering tasks before performing the quantization, we chose a fine-tuned version of the OPT-1.3 model.
Choosing a task is required to define an objective for our quantization target function and control the process. The task and evaluation metrics can vary widely, ranging from text generation with perplexity, summarization with ROUGE, translation with BLEU, to even classification based on simple accuracy. The next is to define the evaluation metric to assess the model's accuracy and the benchmark dataset that compliments it.

```
task_evaluator = evaluate.evaluator("question-answering")
eval_dataset = load_dataset("squad", split="validation", cache_dir="./squad-ds")
eval_dataset = eval_dataset.select(range(64)) # Ues a subset of dataset
```

The `.evaluator()` method will load the essential functions required for evaluating the question-answering task. (Further information on various options is available in the Hugging Face documentation.) You can then employ the `load_dataset` function from the Hugging Face library to bring a dataset into memory. This function allows you to specify parameters such as the dataset name, which splits to download (train, test, or validation), and the location for storing the dataset. Using the mentioned variables to create the evaluation function is now possible.

```
qa_pipeline = pipeline("question-answering", model=model, tokenizer=tokenizer)
def eval_fn(model):
    qa_pipeline.model = model
```

```
    metrics    =    task_evaluator.compute(model_or_pipeline=qa_pipeline,    data=eval_dataset,
metric="squad")
    return metrics["f1"]
```

We need to create a pipeline that ties the model with the tokenizer to be able to calculate the model's performance by calling the task evaluator's `.compute()` function along with the mentioned pipeline and the evaluation dataset. The `eval_fn` function will compute the accuracy and return the percentage. The quantization process requires several configurations for guidance.

```
# Set the accepted accuracy loss to 1%
accuracy_criterion = AccuracyCriterion(tolerable_loss=0.01)


# Set the maximum number of trials to 10
tuning_criterion = TuningCriterion(max_trials=10)


quantization_config = PostTrainingQuantConfig(
    approach="dynamic",                                    accuracy_criterion=accuracy_criterion,
tuning_criterion=tuning_criterion)
```

The `PostTrainingQuantConfig` config variable will set the required parameters for the quantization process. We are employing the `dynamic` approach for quantization while accepting a maximum of 1% loss in accuracy, controlled by defining the `AccuracyCriterion` class. Use the `TuningCriterion` class to set the maximum number of trials before finishing the quantization process. Lastly, we will define a quantizer object using the `INCQuantizer` class, which accepts both the model and evaluation function. It could initiate the quantization process by calling the `.quantize()` method.

```
quantizer = INCQuantizer.from_pretrained(model, eval_fn=eval_fn)


quantizer.quantize(quantization_config=quantization_config,           save_directory="opt1.3b-
quantized")
```

Please note that it is impossible to execute the codes in this section on the Google Colab instance due to memory constraints. However, you can replace the model (`"facebook/opt-1.3b"`) with a smaller model like `"distilbert-base-cased-distilled-squad"`.

Inference

Now, the model is ready for inference purposes. In this section, we will focus on how to load these models and present the outcomes of our benchmark tests, highlighting the impact of quantization on the speed of the generation process. Prior to conducting the inference process, it's essential to load the pre-trained tokenizer using the `AutoTokenizer` class. As the quantization technique doesn't alter the model's vocabulary, we will employ the same tokenizer as the base model.

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-1.3b")
```

For loading the model, we utilize the `INCModelForCasualLM` class provided by the Optimum package. Additionally, it offers a range of loaders tailored for various tasks, including `INCModelForSequenceClassification` for classification and `INCModelForQuestionAnswering` for

tasks involving question answering. The `.from_pretrained()` method should be provided with the path to the quantized model from the previous section.

```
from optimum.intel import INCModelForCausalLM

model = INCModelForCausalLM.from_pretrained("./opt1.3b-quantized")
```

Finally, we can employ the identical `.generate` method from the Transformers library to input the prompt to the model and get the response.

```
inputs = tokenizer("<PROMPT>", return_tensors="pt")

generation_output = model.generate(**inputs,
                                   return_dict_in_generate=True,
                                   output_scores=True,
                                   min_length=512,
                                   max_length=512,
                                   num_beams=1,
                                   do_sample=True,
                                   repetition_penalty=1.5)
```

The last step is to convert the generated token ids from the model to the words. It is the same decoding process as we saw in previous lessons.

```
print( tokenizer.decode(generation_output.sequences[0]) )
```

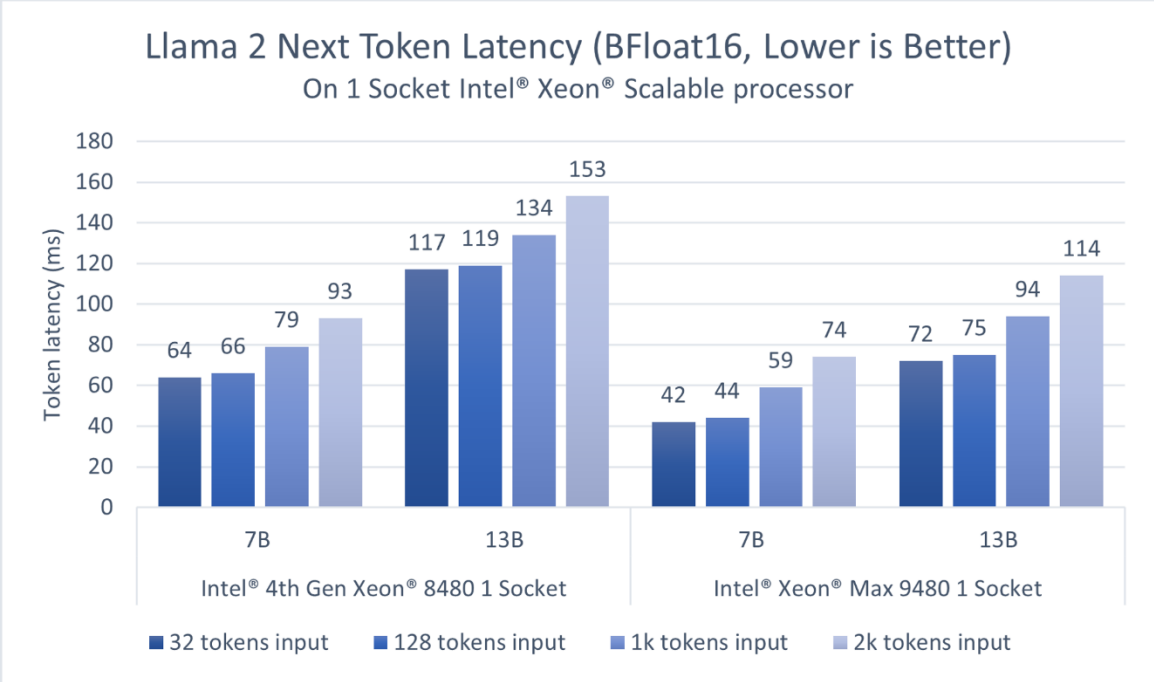
```
What does life mean? Describe in great details.\nI have no idea. I don't know how to describe
it. I don't know what I'm supposed to do with my life. I don't know what I want to do with my
life...
```

The output.
The chosen OPT model is not a instructed tuned model. so it tries to complete the sequence which is fed to it. As evident, it enters into a repetitive loop, reiterating the same words, since we have instructed it to produce exact 512 tokens. Even if the model desires to stop, it is unable to do so!
As mentioned, we force the model to produce 512 tokens by explicitly setting minimum and maximum length parameters. The rationale is maintaining a uniform token count between the standard model and the quantized version, facilitating a valid comparison of their generation times. We also experimented with the beam search decoding strategy.

Decoding Method	Vanilla *(seconds)*	Quantized *(seconds)*
Greedy	58.09	**26.847**
Beam Search (K=4)	144.77	**40.73**

The most significant enhancement involves implementing beam search with a batch size of 1, which led to a 3.5x acceleration in the inference process. All the mentioned experiments were conducted on a server instance equipped with the 4th Gen Intel® Xeon® Scalable processor with 8 vCPU (4 cores) and 64GB of memory. The utilized instance is the entry tier of the Intel CPUs, featuring the least number of cores among the Intel® Scalable Processor family. This highlights the feasibility of performing inference on CPU instances to mitigate costs and latency effectively.

On the other hand, the highest tier series of 4th Gen Intel Xeon Scalable processors, accessible on the Google Cloud Platform, offers 176 virtual CPUs with 88 cores. It's worth highlighting that the most powerful processor within the SPR family can have up to 112 physical cores. In a recent report released by Intel comparing the performance of their Intel 4th Gen Xeon to the latest Intel® Xeon® Max, the report highlights an impressive speed of 114ms for processing 2K tokens using the latest Max processors with the LLaMA 2 model, which has 13B parameters. Intel takes the lead in advancing and fine-tuning the CPU backend of `torch.compile`, a prominent feature in PyTorch 2.0. Additionally, Intel provides the Intel® Extension for PyTorch to implement advanced optimizations specifically designed for Intel CPUs before their integration into the official PyTorch distribution.



Llama 2 7B and 13B inference (BFloat16) performance on Intel® Xeon® Scalable Processors. (source: Intel Blog)

Deployment Frameworks

Deploying large language models into production is the final stage in harnessing their capabilities for a diverse array of applications. Creating an API is the most efficient and flexible approach among the various methods available. APIs allow developers to seamlessly integrate these models into their code, enabling real-time interactions with web or mobile applications. There are several ways to create such APIs, each with its advantages and trade-offs.

There exist specialized libraries, such as vLLM and TorchServe, designed for handling specific use cases. These libraries are capable of loading models from various sources and creating endpoints for convenient accessibility. In most cases, these libraries even offer optimization methods to enhance the speed of the inference process, batching income requests, and efficient memory management. On the other hand, there exist standard backend libraries such as FastAPI that facilitate the creation of any endpoints. While it may not be specifically designed for serving AI models, you can effortlessly integrate it into your development process to generate other APIs as needed.

Regardless of the chosen method, a well-designed API ensures that large language models can be deployed robustly, enabling organizations to leverage their capabilities in chatbots, content generation, language translation, and many other applications.

Deploying a model on CPU using Compute Engine with GCP

Follow these steps to deploy a language model on Intel® CPUs using Compute Engine with Google Cloud Platform (GCP):

1. **Google Cloud Setup**: Sign in to your Google Cloud account. If you don't have one, create it and set up a new project.
2. **Enable Compute Engine API**: Navigate to APIs & Services > Library. Search for "Compute Engine API" and enable it.
3. **Create a Compute Engine instance**: Go to the Compute Engine dashboard and click on “Create Instance”. Choose an CPU for your machine type. Here are several machine types that can be used in GCP and sporting Intel CPUs.

CPU processor	Processor SKU	Supported machine series and types	Base frequency (GHz)	All-core turbo frequency (GHz)	Single-core max turbo frequency (GHz)
Intel Xeon Scalable Processor (Sapphire Rapids) 4th generation	Intel® Xeon® Platinum 8481C Processor	• C3	1.9	3.0	3.3
Intel Xeon Scalable Processor (Ice Lake) 3rd Generation	Intel® Xeon® Platinum 8373C Processor	• N2*	2.6	3.4	3.5
		• M3	2.6	3.4	3.5
Intel Xeon Scalable Processor (Cascade Lake) 2nd Generation	Intel® Xeon® Gold 6268CL Processor	• N2*	2.8	3.4	3.9
	Intel® Xeon® Gold 6253CL Processor	• C2	3.1	3.8	3.9
	Intel® Xeon® Platinum 8280L Processor	• M2	2.5	3.4	4.0
	Intel® Xeon® Platinum 8273CL Processor	• A2 • G2	2.2	2.9	3.7
Intel Xeon Scalable Processor (Skylake) 1st Generation	Intel® Xeon® Scalable Platinum 8173M Processor	• E2 • **m1-megamem** memory-optimized machine types • N1	2.0	2.7	3.5
Intel Xeon E7 (Broadwell E7)	Intel® Xeon® E7-8880V4 Processor	• **m1-ultramem** memory-optimized machine types	2.2	2.6	3.3
Intel Xeon E5 v4 (Broadwell E5)	Intel® Xeon® E5-2696V4 Processor	• E2 • N1	2.2	2.8	3.7
Intel Xeon E5 v3 (Haswell)	Intel® Xeon® E5-2696V3 Processor	• N1	2.3	2.8	3.8
Intel Xeon E5 v2 (Ivy Bridge)	Intel® Xeon® E5-2696V2 Processor	• N1	2.5	3.1	3.5

Once the instance is up and running:

1. **Deploy the model**: SSH into your instance. Install the necessary libraries and dependencies and copy your server code (FastAPI, vLLM, etc) to the machine.
2. **Run the model**: Once the setup is complete, run your language model. If it's a web-based model, start your server.

   Remember, Google Cloud charges based on the resources used, so make sure to stop your instance when not in use.

   A similar process can be done for AWS too using EC2. You can find AWS machine types here.

Conclusion

In this lesson, we explored the potential of harnessing 4th Generation Intel Xeon Scalable Processors for the inference process and the array of optimization techniques available that make it a practical choice. Our

focus was on the quantization approach aimed at enhancing the speed of text generation while conserving resources. It is fairly straightforward to perform the optimization process courtesy of a series of libraries from Intel such as Intel Extension for PyTorch and Intel® Extension for Transformers.

The results demonstrate the advantages of applying this technique across various configurations. It is worth noting that there are additional techniques available to optimize the models further. The upcoming chapter will discuss advanced topics within language models, including aspects like multi-modality and emerging challenges.

>> Notebook.