LLM Deployment Optimization: Quantization, Pruning, and Speculative Decoding

Latency and memory usage are critical factors in the deployment and performance of large language models (LLMs). Latency refers to the time delay between input and output, which is particularly crucial in real-time or near-real-time applications. For instance, in conversational AI, excessive latency can disrupt the natural flow of interaction, leading to a poor user experience. The typical reading speed of an average person is approximately 250 words per minute (equivalent to around 312 tokens per minute), which translates to about five tokens every second, implying a latency of 200 milliseconds per token. The goal thus is to achieve a latency of 100 to 200 milliseconds per token to align with typical human reading speeds.

However, achieving low latency is challenging due to the immense size and complexity of LLMs. These models often contain billions of parameters, resulting in large weight files that must be loaded into memory for inference. The sheer size of these weight files creates a significant memory bottleneck, which directly impacts latency. The memory-intensive nature of LLMs stems from their architecture, primarily based on the transformer model. During inference, the model must load its weights from slower VRAM (Video RAM) into faster on-chip cache memory. This process of moving large amounts of data between different memory hierarchies is often the primary bottleneck in LLM inference rather than the actual computation itself. For this reason, the time taken to process a single token is often comparable to processing multiple tokens in a batch. This is because once the weights are loaded into memory, applying them to multiple inputs is relatively inexpensive in terms of additional time.

Reducing memory requirements or improving memory access patterns can have a significant impact on overall performance. This has driven the development of various optimization techniques, including model quantization, pruning, and novel inference strategies to address latency, like speculative decoding.

Model Quantization

An increase in the number of parameters in large language models results in substantial memory usage, subsequently increasing hosting and deployment costs. Techniques like quantization reduce the memory requirements of neural network models and help with cost reduction.

Quantization is a process that reduces the numerical precision of model parameters, including weights and biases. It involves reducing the precision of the model parameters and/or activations. This technique can significantly lessen memory usage by employing low-bit precision arithmetic, leading to reductions in the models' size, latency, and energy consumption, making it more feasible to deploy them on devices with limited resources, such as mobile phones, smartwatches, and other embedded systems.

Quantization is relatively simple to understand. Consider this example: When asked for the time, a person can either respond with the exact time, 10:58 p.m. or about 11 p.m. The second version reduces the response time, making it less precise but more accessible to explain and understand. Similarly, in deep learning, the precision of model parameters is reduced to make the model more efficient, but at the expense of some accuracy. In machine learning, the precision of model parameters is determined by the floating-point data types used. Higher precision types, such as Float32 or Float64, yield greater accuracy but increase memory usage. Conversely, lower precision types like Float16 or BFloat16 consume less memory but may slightly decrease accuracy.

Many open-source LLMs accessible via Cloud APIs or downloaded to run locally will be quantized versions. Inference providers typically use post-training quantization methods to compress open-source LLMs from 32-bit floating point (FP32) or FP16 to lower precision formats like 8-bit integers (INT8) or even 4-bit. Moving from FP32 to FP8 reduces the file size and memory needs by four times. The quality of quantized models can vary widely depending on the specific techniques used. Poor quantization choices can lead to significant accuracy drops, while state-of-the-art methods may preserve most of the original model performance. There's generally a tradeoff between model size/speed and accuracy. More aggressive quantization (e.g., 4-bit) allows for smaller models that can run on devices with limited resources, but risk larger accuracy drops. Different inference providers may use varying quantization approaches, leading to differences in model quality and performance. For example, some providers like

Together.ai or Groq specialize in fast inference for large models, but the exact quantization techniques they use are often not publicly disclosed. This can lead to variations in model performance when accessing models such as Llama 3.1 via different inference providers.

The memory needs for an AI model can be approximated with its parameter count. For instance, the Llama2 70B model uses Float16 precision, with each parameter taking up two bytes. The formula to determine the required memory in gigabytes (GB), where 1GB equals 1024^3 bytes, is:

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

The following sections explain various quantization techniques.

Scalar Quantization

Scalar quantization involves treating each dimension of a dataset separately. First, it calculates the maximum and minimum values for each dimension across the dataset. Next, the distance between these maximum and minimum values in each dimension is segmented into uniform-sized intervals or bins. Finally, every value in the dataset is assigned to one of these bins, thus quantizing the data.

Let's execute scalar quantization on a dataset with 2000 vectors, each with 256 dimensions, originating from a Gaussian distribution:

```
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)))
```

Calculate the start and step values for each dimension. The start value is the lowest, and the step size is determined by the number of discrete bins in the integer type being used. This example employs 8-bit unsigned integers (uint8), yielding 256 bins.

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

The quantized dataset is calculated as follows.

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

The scalar quantization process can be encapsulated in the following 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)
```

Product Quantization

In scalar quantization, it is important to consider the data distribution within each dimension to minimize information loss. For example, consider the following vectors.

```
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]
]
```

Applying scalar quantization to convert these vectors to a 4-bit integer leads to a considerable loss of information:

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

Product quantization improves the process by dividing the original vector into sub-vectors and quantizing each one individually. In this method, each vector in the dataset is split into `m` sub-vectors. The data in each sub-vector is then grouped into `k` centroids using techniques like k-means clustering. Each sub-vector is replaced by the index of its nearest centroid from a corresponding codebook. Let's apply product quantization to the given array with `m` set to 3 sub-vectors and `k` set to 2 centroids.

```
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_array
```

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

Quantizing vectors and storing only the indices of the centroids leads to a notable reduction in memory footprint. This technique retains more information than scalar quantization, mainly when the data dimensions vary. Product quantization can decrease memory usage and expedite finding the nearest neighbor. However, this comes with a potential reduction in accuracy. The balance in product quantization hinges on the number of centroids and sub-vectors employed. More centroids improve accuracy but may not significantly reduce the memory footprint, and vice versa.

Quantizing Large Language Models

Scalar and product quantization methods may suffice for models with limited parameters, but they often result in a decline in accuracy when applied to larger models with billions of parameters. Large models encompass a larger amount of information within their parameters. These models can represent more complex functions because of increased neurons and layers. They also excel at capturing deeper and more nuanced relationships within the data. Consequently, the quantization process, which involves reducing the precision of these parameters, can lead to significant information loss. This often results in a notable decrease in model accuracy and overall performance.

Optimizing the quantization process and identifying a strategy that effectively reduces model size while maintaining accuracy becomes challenging with larger models due to their expansive parameter space.

More advanced quantization techniques have emerged to tackle the challenge of reducing the size of large language models while maintaining their accuracy. One such method is "LLM.int8()", which recognizes that activation outliers—values significantly different from the norm—can disrupt quantization. To address this, the method keeps these outliers in higher precision, preserving the model's performance.

GPTQ (a combination of the OPT model family name and post-training quantization, PTQ), from a 2023 paper, enhances text generation speed by quantizing each layer separately. It minimizes the difference between quantized and full-precision weights by using a mixed INT4-FP16 scheme: weights are quantized as INT4, while activations remain in Float16. During inference, weights are de-quantized, and computations are performed in Float16. This method requires running inferences on a calibration dataset to fine-tune the quantized weights.

Activation-aware Weight Quantization (AWQ) builds on this by assuming that not all model weights are equally important for performance. It identifies and preserves a small percentage (0.1%-1%) of these "salient" weights in FP16 format to reduce quantization loss. Unlike traditional methods that focus on weight distribution, AWQ selects important weights based on activation magnitude, improving quantized model performance. Although this mixed-precision approach can create hardware inefficiencies, researchers suggest quantizing all weights uniformly after scaling the critical ones to maintain efficiency without losing vital information.

Numerous open-source LLMs are accessible in a quantized format, offering the advantage of reduced memory requirements. Check the model's section on Hugging Face to find and use a quantized model. An example is the Mistral-7B-Instruct model, which is quantized using the GPTQ method.

Quantizing an LLM with a CPU

GPUs are typically used for inference due to their parallel processing power, which handles large models efficiently. However, quantization enables running models on CPUs as well, making inference feasible on lower-cost hardware. This is advantageous in scenarios where GPUs are unavailable, or power and budget constraints are important. While GPUs remain faster for high-performance tasks, CPUs can deliver acceptable performance for quantized models, offering a more accessible and energy-efficient alternative.

The Intel Neural Compressor Library helps with quantizing large language models by providing a variety of model quantization techniques so that the resulting quantized models can also be run on CPUs.

First, follow the step-by-step instructions in the neural compressor repository to ensure you have all the necessary components and expertise before proceeding. Next, install the `neural-compressor` and `lm-evaluation-harness` libraries using `pip`.

Inside the cloned neural compressor directory, proceed to the proper directory and install the essential packages with the following command:

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


pip install -r requirements.txt
```

As an experiment, you can quantize the opt-125m model with the GPTQ algorithm using 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. It is as simple as that!

Model Pruning

Despite the advancements in deep learning and quantization, deep neural networks often need help with their considerable size and computational requirements. Model pruning is an effective technique in addressing this issue, aiming to decrease the size of neural networks while maintaining their effectiveness.

Sparsity, often achieved by pruning, as discussed in the "Sparsity in Deep Learning" study in 2021, is a technique for reducing the computational demands of LLMs. It involves removing redundant or less essential weights and activations. This approach can substantially decrease off-chip memory consumption, corresponding memory traffic, energy use, and latency.

Model pruning aims to create a smaller and more efficient model while maintaining accuracy. The advantages of pruning include faster inference times, a smaller memory footprint, and enhanced energy efficiency. This makes pruned models ideal for applications with limited computational resources, such as mobile apps, IoT devices, and edge computing, where speed and efficiency are critical for real-time performance.

Pruning can be classified into two categories: weight pruning and activation pruning. Weight pruning is divided into unstructured and structured forms. Unstructured pruning allows for flexible sparsity patterns, while structured pruning imposes specific patterns that help improve memory efficiency, reduce energy consumption, and lower latency without requiring specialized hardware. However, structured pruning generally offers lower compression compared to unstructured approaches. Activation pruning focuses on eliminating redundant activations during inference, which can be particularly useful for transformer models. This method requires dynamically identifying and removing unnecessary activations during runtime.

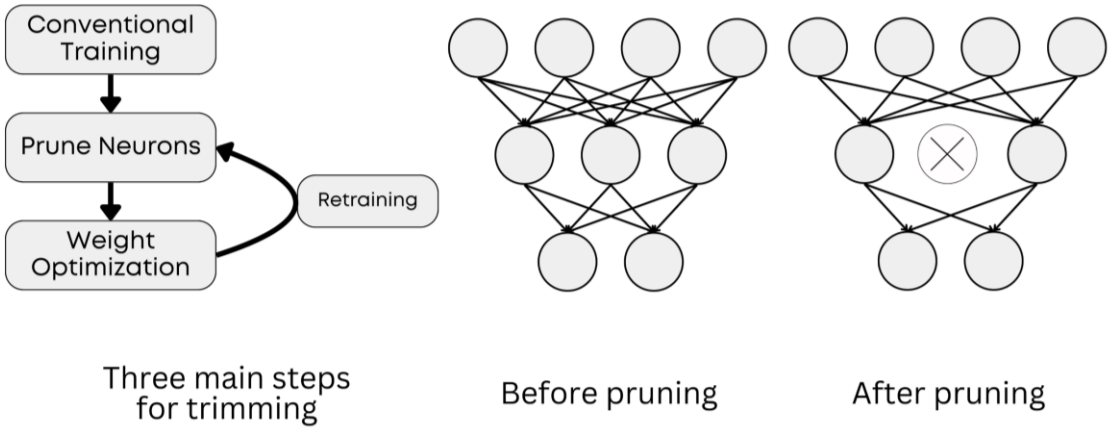
Several techniques and methodologies exist for model pruning, each with advantages and trade-offs. The following section investigates the three most relevant and interesting approaches.

Magnitude-based Pruning (or Unstructured Pruning)

Magnitude-based or unstructured pruning removes weights or activations of small magnitudes in a model. The underlying principle is that smaller weights have a lesser impact on the model's overall performance and can thus be eliminated without significant loss of functionality.

The concept was elaborately presented in the paper "Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures" in 2016. It introduced an optimization technique for deep neural networks by pruning non-essential neurons. Network Trimming is grounded in the observation that many neurons in a large network yield zero outputs, regardless of the input. Neurons that consistently produce zero activations are deemed redundant and can be pruned without adversely affecting the network's accuracy. The process entails a cycle of pruning and subsequent retraining, where the network's initial pre-pruning weights serve as the starting point. The

research demonstrates that this approach can significantly reduce the number of parameters in computer vision neural networks, maintaining or enhancing the accuracy compared to the original network.



Left: the usual pruning process, middle: an example neural network pre-pruning, and right: the same network after pruning the middle weight.

Another notable paper, "A Simple and Effective Pruning Approach for Large Language Models," published in 2023, presents a pruning method named Wanda (pruning by Weights and Activations) for LLMs. In this context, pruning selectively eliminates a portion of the network's weights to preserve performance while reducing the model's size. It targets weights based on the smallest magnitudes multiplied by their corresponding input activations, assessed on a per-output basis. This strategy draws inspiration from the emergent large-magnitude features in LLMs. One of the significant advantages of Wanda is it does not necessitate retraining or weight updates, so the pruned LLM can be employed immediately.

Structured Pruning

Structured pruning focuses on specific structural elements within a neural network, such as channels in convolutional layers or neurons in fully connected layers. The paper "Structured Pruning of Deep Convolutional Neural Networks", published in 2015, introduced an innovative approach to network pruning that integrates structured sparsity at various levels, including channel-wise, kernel-wise, and intra-kernel strided sparsity, particularly effective in saving computational resources. The technique employs a particle filtering method to assess the importance of network connections and pathways and assigns significance based on the misclassification rate linked with each connectivity pattern. After the pruning process, the network undergoes retraining to recover any performance losses that may have occurred.

The Lottery Ticket Hypothesis

The paper "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks", published in 2019, introduced a novel viewpoint on neural network pruning with the "Lottery Ticket Hypothesis." This hypothesis proposes that within large, randomly initialized, feed-forward networks, there are smaller subnetworks ("winning tickets") that, if trained independently, can reach a level of test accuracy comparable to the original network in a similar number of iterations. These "winning tickets" are distinguished by their initial weight configurations, which are particularly helpful for practical training.

The algorithm developed to locate these "winning tickets" conducted a series of experiments to prove the hypothesis. They consistently found "winning tickets" that were only 10-20% the size of various full-sized, fully connected, and convolutional feed-forward architectures tested on the MNIST and CIFAR10 datasets. These smaller subnetworks did not just replicate the performance of the original network; they frequently outperformed it, exhibiting quicker learning and higher test accuracy.

Speculative Decoding

Speculative decoding is another optimization technique for improving the inference speed of large language models (LLMs) without compromising output quality. It leverages an unintuitive observation: forwarding an LLM on a single input token can take nearly as much time as forwarding it on K input tokens in a batch, where K is larger than one might expect. This is because LLM inference is heavily memory-bound, most of the work involves reading the transformer's weights from VRAM into the on-chip cache rather than computation. Once these weights are loaded, applying them to multiple input vectors is relatively cheap. However, we can't simply generate tokens in large batches due to the serial dependency in autoregressive generation, where each token depends on all previous tokens.

Speculative decoding cleverly circumvents this limitation by using a smaller, faster "draft" model to generate a candidate sequence of K tokens. These draft tokens are then fed together through the large target LLM in a batch, which is almost as fast as processing a single token. The target LLM then verifies these candidates from left to right. When a draft token is accepted, the system can immediately move to the next token, effectively skipping ahead. If a disagreement occurs, the remaining draft is discarded, incurring some wasted computation. This approach is effective because many tokens in a sequence are predictable based on context and can be easily handled by a smaller model. For instance, in the phrase "The cat is on the," a small model can confidently predict the next token as "mat" without needing the full power of a large LLM. These are the "easy" tokens—words or phrases that follow common patterns or predictable grammatical structures. By quickly generating these easy tokens, the system speeds through familiar parts of the text. The large model only steps in when the smaller model encounters a more ambiguous or complex token, such as in a less common phrase like "quantum entanglement is a," where it might disagree on the next word. In such cases, the draft is discarded, and the large model takes over to ensure accuracy. The system can leap through these easy parts quickly, only slowing down for the more challenging tokens where the large model disagrees with the draft.

Speculative decoding represents a clever way to partially overcome the sequential nature of autoregressive generation in LLMs, offering substantial performance improvements without compromising output quality.

The key factors affecting speculative decoding performance are the draft model's latency and its token acceptance rate (TAR). Interestingly, a draft model's accuracy on standard language modeling tasks doesn't strongly correlate with its TAR in speculative decoding. The draft model's latency is primarily determined by its depth (number of layers), while its width (size of layers) has less impact on speed. This insight suggests that wider, shallower draft models may be more efficient for speculative decoding than deeper models with the same parameter count. When designing draft models specifically for this purpose, it's crucial to balance latency reduction with maintaining a reasonable TAR. Recent experiments have shown that pruning larger models to create wide, shallow configurations can provide significant throughput improvements over existing approaches. As LLM architectures and hardware capabilities evolve, the optimal draft model design may change, but the principles of optimizing for both low latency and sufficient TAR are likely to remain fundamental to effective speculative decoding.