Efficient and Edge AI: A Detailed Analysis of Large Language Model (LLM) Quantization Techniques and Tools

Abstract

Worldwide, institutions, individuals, and innovators meticulously use large language models (LLMs) and their purpose-specific derivations to swiftly and sagaciously fulfil their everyday assignments. Today, we have pre-trained and generic large language models for providing distinct use cases associated with natural language processing, understanding, and generation. They are being further finetuned to tackle domain-specific tasks with more accuracy. Vision language models (VLMs) are being unearthed and used extensively these days to tackle visual data. Multimodal large language models (MM-LLMs) are being designed and deployed to seamlessly cater to multimodal data types. Considering the proliferation of the Internet of Things (IoT) edge devices and to provide real-time edge data processing, knowledge discovery and actuation, small language models (SLMs) are being produced and deployed to facilitate on-device data processing and to gain real-time edge intelligence. Thus, the field of language modelling is getting faster and greater adoption. However, a few principal challenges are getting widely attached to language models. The computing, memory, storage, and inter-memory bandwidth resources are exponentially growing with the increasing number of parameters in language models. Furthermore, the inferencing latency is also on the higher side.

Researchers are working on vulnerable workarounds to surmount the above-mentioned constraints. Quantisation is being prescribed as the best-in-class solution approach to overcome the technical challenges. This paper explores quantisation techniques and tools to arrive at highly optimised and compressed language models that can run with reduced resources and fulfil the need for lowered latency.

Code: GitHub - Avinashhmavi/Quantization-using-different-methods
Models: Hugging Face – avinashhm/qwen3-1.7b-gguf-q4_0

The Surging Popularity of Language Models

The flourish of language models (LMs) in the recent past has unleashed a new life for the paradigm of artificial intelligence (AI), especially for the domain of natural language processing (NLP). There are many commercial-grade and open source large, small, vision and multimodal language models (LLMs, SLMs, VLMs and MM-LLMs) elegantly tackling a growing array of NLP use cases such as question-answer systems, conversational chatbots, expert and decision-support systems, content and code generation, reasoning engines, etc. Language models have become inseparable in our personal, professional and social lives. We are empowered by various information and knowledge services artistically provided by language models in our everyday assignments and obligations. With the astounding growth of data volumes, the ready availability of AI-centric processing units, the game-changing transformer architecture, and language model development and deployment are seeing many noteworthy improvements. With many real-world capabilities, language models are rapidly being adopted across industry verticals.

Demystifying the Technical Challenges of Language Models

However, the memory and computational requirements are higher for training, fine-tuning, and inferring language models. In the context of cloud AI, such models do not pose any problems as the cloud is blessed with large-scale compute, storage and network resources. However, running language models has several technical challenges in fulfilling edge AI's goals. Thus, the domain of efficient AI has gained the attention of many these days. With fewer computational resources, how to fine-tune and infer from language models has become the moot question. The real challenge for AI engineers and experts is running language models on resource-constrained Internet of Things (IoT) edge devices such as smartphones, robots, drones, cameras, industry automation and control systems, automotive, consumer and medical electronics devices. In other words, how to deploy and infer language models on commodity hardware, embedded devices and operational systems is being touted as the critical challenge to be surmounted to make use of language models pervasively. So building efficient AI models through a host of competent optimisation technologies and tools is the need of the hour. The challenges of running and using AI models across edge devices must be overcome technologically. The way forward is to bring compressed and optimised language models to use fewer resources. In this report, we are to discuss existing and emerging model compression techniques and tools in detail.

Elucidating the Significance of Efficient and Edge AI

AI models and agents are becoming ubiquitous these days. All business automation requirements are being meticulously accomplished through competent AI models and agents. The AI phenomenon has gotten a new life, especially with language models gaining widespread adoption and adaptation. With the disruptive transformer architecture gaining the imagination of many, the development of language models has been rapid. However, as pre-trained and generic language models have billions of parameters (weights and biases), fine-tuning and inference from pre-trained and generic language models require large-scale resources (memory, storage and processing). The inferencing latency is on the higher side.

Large language models (LLMs) typically comprise billions of parameters and are trained on trillions of words. Therefore, they require computational power during both training and deployment. With the leverage of compression methods, it is possible to reduce LLMs' size without substantially degrading performance. Thus, AI efficiency has drawn the attention of AI scientists and scholars. Quantisation is an important technique towards model size reduction. LLM quantisation assists in processing large amounts of language data. Quantisation converts a continuous data set to a discrete one, reducing the number of bits needed to display the signal. Quantisation takes a finite set of discrete values and uses them to approximate continuous values. Many discrete values should be translated to support a continuous signal. This quantisation power is essential for computer graphics and digital communication activities.

Further, running language models on edge devices and embedded systems is neither simple nor straightforward. To achieve the goal of running AI models, especially language models, on resource-constrained personal, social, and professional devices, LLM compression, especially quantisation, is undoubtedly the way forward. This paper provides a technical perspective on LLM quantisation. It also discusses various quantisation methods, their implications on model performance, and their practical applications across industry verticals. Sections depict the challenges and opportunities in LLM quantisation.

Why LLM Optimization is Important?

LLM optimisation is an essential process for enhancing the performance and efficiency of LLMs by achieving heightened accuracy, speed, and resource utilisation. The goal is also to reduce the environmental impact. There are promising techniques like prompt engineering and fine-tuning. The LLM optimisation is also enabled through LLM compression techniques and hardware acceleration (using advanced processing units (GPUs, TPUs, etc.)). Retrieval-augmented generation (RAG) is another approach for achieving LLM optimisation. At the infrastructure level, distributing LLM requests across different and distributed edge devices or compute nodes is recommended for LLM optimisation.

Improved resource utilisation by LLMs is another requirement for LLM optimisation. That is, while training and inferencing, an optimal amount of computational, memory and storage resources has to be used. The potential techniques like model compression, efficient attention mechanisms, batching strategies, key-value caching, and distributed computing are to optimise the resource usage.

Hardware Acceleration towards LLM Optimization - Experts say that hardware acceleration is an option for LLM optimisation. That is, the performance of LLMs can be substantially increased by running computationally intensive tasks on AI-centric processors such as GPUs and TPUs. Such advanced processing units offer parallel processing; hence, LLMs can exhibit efficiency and become faster in delivering results. There are options such as in-memory computing and leveraging prominent hardware accelerators such as ASICs and FPGAs towards LLM optimisation.

AI Model Compression Mechanisms - Research papers have indicated that a 7B model would need approximately 280 GB of memory. Thus, there is a clear call for viable methods to reduce models' size without compromising model performance. Researchers have indicated many ways and means of achieving this goal.

- **Pruning**
- **Knowledge Distillation**
- **Quantisation**

Pruning—Neural network (NN) models connect neurons with synapses/connections. Removing some neurons or connections to achieve NN models with reduced sizes is possible. The issue with this method is finding an efficient way to understand which neurons and synapses can be removed without compromising model performance. An LLM is typically made of billions of neurons and synapses.

Knowledge Distillation is a popular way to arrive at a smaller model. Knowledge distillation means transferring the knowledge of a bigger model to a smaller one. Transferring the knowledge from multiple models into a single, small model is also possible. Generally, the origin and bigger model are named teacher models, and the smaller model is termed student models.

We discussed the prominent techniques for accomplishing the ideals of LLM optimisation. In the subsequent sections, we will address the quantisation technique and why it is proclaimed the key model compression method for LLM optimisation.

A Recap on Quantization

Quantisation represents weights and activations using lower-precision data types, such as 8-bit integers (int8), instead of the standard 32-bit floating-point (float32) format. The most frequently used quantisation conversions are from float32 to either float16 or int8.

Model quantisation involves transforming the parameters of a neural network, such as weights and activations, from high-precision (e.g., 32-bit floating point) representations to lower-precision (e.g., 8-bit integer) formats. This reduces memory usage, facilitates quick inferencing, and increases energy preservation.

For instance, consider a parameter with a 32-bit precision value of 7.892345678. This value can be approximated as the integer 8 using 8-bit precision. This process significantly reduces the model size, enabling faster execution on devices with limited memory. The diagram below clearly explains the importance of quantisation. Here is a simple example illustrating how quantisation works.

Higher precision data types let us represent large or floating-point numbers with higher digits. For instance 16-bit floating points can represent floats with 3 to 4 digits. The quantisation technique is being pitched as the way forward to reduce the parameters' size. That is, the workaround is to quantise the parameters to fewer bits and use low-bit-precision matrix multiplication. Quantisation methods reduce memory use but degrade performance, requiring further tuning after training.

Delineating the Standard Steps to Perform Quantisation—Quantisation converts a range of continuous values into smaller discrete values. The following steps show this process.

It is to determine the range of values for the weights and activations. There are a few methods, such as min-max values or statistical methods, such as mean and standard deviation. Secondly, it is the scale and zero-point calculation. This is to calculate a scaling factor and zero point to map the continuous values to the discrete integer values. The scaling factor determines the step size between discrete values. On the other hand, the zero point aligns the scale with the original data distribution.

Finally, there is the quantisation step. It involves applying the scaling factor and zero point to convert the continuous values to discrete integer values. Quantisation involves converting a neural network (NN)'s parameters' continuous (floating-point) values to discrete (integer) values to reduce the model's computational and memory footprint.

The Quantisation Methods - Symmetric linear quantisation is a technique in which a range of floating-point values is mapped to a symmetric range around zero in a quantised space (using integers). This means the zero in the floating-point space maps to zero in the quantised space. Also, the positive and negative ranges have equal magnitude,

Asymmetric linear quantisation maps floating-point values to a lower-precision integer representation. This uses different scale factors for positive and negative values. This allows for mapping the zero point of the floating-point range to a specific value in the quantised range.

Key Parameters to Consider When Quantising a Model—Several parameters must be carefully managed when quantising a model.

- Precision level—Choosing the right precision level (e.g., 16-bit, 8-bit, and 4-bit) is important. Lower precision levels offer greater efficiency gains but can result in accuracy drops.
- Scaling factors—Identifying the scaling factor for weights and activations is important to maintaining the model's performance. Finding the right balance between range and precision is paramount for achieving the success of LLM quantisation.
- Quantisation scheme – There are multiple options for LLM quantisation; each scheme has pros and cons.
- Hardware compatibility—It is equally important to check whether the target hardware supports the chosen quantisation format. That is, the inference engine must use the quantised model.

In linear quantisation, scale and zero-point are the crucial parameters that define how floating-point values are mapped to integer values. These parameters ensure that the quantised representation can approximate the original values as closely as possible, within the constraints of reduced precision.

Blockwise quantisation is more accurate than linear quantisation for models with non-uniform weight distributions. It is a more sophisticated method that involves quantising weights in smaller blocks rather than across the entire range. Quantisation can be applied separately, allowing better handling of variations within different model parts.

Distribution-aware blocks—The quantisation process considers the relative frequency of the weights within each block. This creates blocks that are aware of the distribution of the weights, resulting in a more efficient mapping of values.

Weight Quantisation vs. Activation Quantisation - While weight quantisation is an important step for model optimisation, it is also important to consider that the activations of a model can also be quantised. Activation quantisation refers to reducing the precision of the intermediate outputs of each layer in the network. Unlike weights, which are static (constant) once the model is trained, activations are dynamic. This means that activations change with each input to the network. This makes their range harder to predict. Activation quantisation is more complex to implement than weight quantisation. It requires careful calibration to ensure that the dynamic range of activations is well captured.

Calibration Techniques - Some quantisation methods require a calibration step. For example, we must determine a model's original activation range before quantisation. General calibration usually involves running inference on a representative dataset to optimise the quantisation parameters and minimise quantisation error.

During this calibration process, the quantisation algorithm collects statistics about the distribution and range of the model's activations and weights. These statistics help determine the best quantisation parameters. When quantising the weights, computing the scale and the zero-point is also a calibration.

Illustrating the Prominent Quantization Approaches and Algorithms

The prominent quantisation methods include post-training quantisation (PTQ) and quantisation-aware training (QAT). Each method trades off model size, speed, and accuracy. There are three types of quantisation.

- **Fixed-point quantisation**: Here, values are divided into several intervals (or levels), and each interval is assigned a single value.
- **Floating-point quantisation** allows users to include numbers that use a sign or exponent, simplifying the addition of high-precision dynamic representations.
- **Binary quantisation** relies only on 1s and 0s to underlie digital communication systems. It supports swift processing but loses precision.

LLM quantisation involves displaying weights and activations of the large language model (LLM) using less precise formats. This speeds up the weights' processing (with reduced precision). Weights represent the model's parameters, and activations represent the intermediate outputs.

While both (weight and activation quantisation) aim to reduce the precision of these components to save memory and computation, they have distinct characteristics and challenges. While weights are static, activations are typically dynamic. This implies that we can access the full range of the weight values at quantisation time. This also means we must find a solution for quantising activations without having the full range at quantisation time.

Training vs. Inference Quantization - Quantization can be applied at two main stages:

Demystifying Post-training Quantization (PTQ)

After the LLM has been trained, quantisation is performed to convert the model's parameters to lower precision. PTQ can reduce a model's accuracy as some detailed information encoded in the original floating-point values might be lost when the model is quantised. Careful tuning and evaluation are needed to balance making the model smaller through quantisation and keeping the model's accuracy high. That is, it requires careful calibration.

Several libraries facilitate post-training quantisation, including PyTorch's Quantization APIs, OpenVINO's Neural Network Compression Framework(NNCF), and Intel Neural Compressor. These libraries convert floating-point numbers to lower precision integers. The PTQ method is being leveraged for applications that do not insist on higher accuracy. PTQ offers several sub-methods, as illustrated below.

Static quantisation converts a model's weights and activations to lower precision, such as INT8. The range for each activation is computed in advance at quantisation time, typically by passing representative data through the model and recording the activation values.

In a nutshell, PTQ is typically a uniform quantisation mapping a continuous range of floating-point values to a fixed set of discrete quantisation levels. The process divides the floating-point values into equal-sized bins and maps each value to the midpoint of its corresponding bin. The number of bins determines the quantisation level (e.g., 8-bit quantisation has 256 levels). The process is as follows.

1. The continuous range of floating-point values is divided into equal intervals.
2. A single quantisation level represents each interval.
3. Values within an interval are rounded to the nearest quantisation level.

Dynamic quantisation—Only weights are quantised here. The activation values remain in higher-precision during inference. The activation values are quantised dynamically based on their observed range during runtime. The range for each activation is computed on the fly at runtime. This helps to adapt quantisation parameters based on input statistics during inference. Unlike uniform quantisation, dynamic quantisation adjusts the quantisation range based on the actual values encountered during inference.

This groups weights into clusters and represents each cluster with a central value. The original weights are replaced with their corresponding cluster centers, reducing the number of unique weights in the model. This helps attain memory savings and potential computational efficiency gains.

Weight weights are easy since the actual range is known at quantisation time. But it is less clear for activations. Calibration data is used to determine the range of activation values. Calibration is the step during quantisation where the float32 ranges are computed. As per experts, the practical steps are

1. Observers are put on activations to record their values.
2. A certain number of forward passes on a calibration dataset is done (around 200 examples are enough).

3. The ranges for each computation are computed according to some *calibration technique*.

Full Integer Quantization converts all model calculations to integer-based operations. It is for deployment on integer-only hardware.

General Pre-Trained Transformer Quantization(GPTQ) is a post-training quantisation (PTQ) method designed to reduce model size and enable efficient deployment of LLMs on a single GPU. It uses layer-wise quantisation to minimise output error (reducing the mean squared error (MSE) between the original and quantised layers). The key features include

1. **Layer-wise quanti**sation: Weights are processed one layer at a time to preserve accuracy by minimising MSE during quantisation. **This** adjusts weights in batches and minimises the MSE between the original and quantised layers.
2. **Lazy batch updates**—Weights are updated in batches (e.g., 128 columns) to improve GPU utilisation and speed up the process.
3. **Mixed precision**—Weights are quantised to 4-bit integers (INT4), while activations remain in higher precision (FP16). During inference, weights are dequantized in real time for computations.

GPTQ works through a form of layer-wise quantisation:

- An approach that quantises a model layer at a time
- To discover the quantised weights that minimise output error (MSE) between the full-precision and quantised layers.

All the model's weights are converted into a matrix, which is worked through in batches of 128 columns at a time through lazy batch updating. This involves quantising the weights in batch, calculating the MSE, and updating the weights to values that diminish it. After processing the calibration batch, all the remaining weights in the matrix are updated following the MSE of the initial batch. Then all the individual layers are re-combined to produce a quantised model.

GPTQ employs a mixed INT4/FP16 quantisation method in which a 4-bit integer quantises weights, and activations remain in a higher-precision float16 data type. Subsequently, during inference, the model's weights are dequantized in real time, so computations are carried out in float16.

GPTQ is a post-training quantisation (PTQ) technique designed for 4-bit quantisation. This involves compressing all weights to a 4-bit quantisation by minimising the MSE associated with each weight. During inference, the model dynamically de-quantises its weights to float16, aiming to enhance performance while maintaining low memory usage.

Generative Pretrained Transformer Vector Quantisation (GPTVQ) applies vector quantisation to LLMs by quantising groups of parameters together, rather than individually. This method preserves the accuracy of the models and reduces their size more effectively than traditional quantisation methods. For instance, experiments by Qualcomm's researchers have shown that GPTVQ could achieve near-original model accuracy with significantly reduced model size, making it a game-changer for deploying powerful AI models on edge devices.

Elucidating the Quantisation-aware Training (QAT) Approach

Quantum-aware training (QAT) integrates quantisation into the training process itself. The model is trained with quantisation simulated in the forward pass. This empowers the model to learn to adapt to the reduced precision. This often results in higher accuracy because the model can better compensate for the quantisation loss. Techniques include simulated quantisation, straight-through estimator (STE), and differentiable quantisation. However, QAT has to embark on extra steps during training so that the model can learn to perform well when it gets compressed. The additional steps lead to the need for additional computational resources. After training, the model needs thorough testing and fine-tuning to ensure the desired accuracy. Quantisation-aware training has 3 steps:

1. The intermediate state holds both quantised and unquantised versions of weights.
2. Forward pass using quantised version of the model
3. Backpropagation of unquantised version of the model weights.

For both post training static quantisation and quantisation-aware training, it is necessary to define calibration techniques, the most common are:

- Min-max - The computed range is [min observed value, max observed value], which works well with weights.
- Moving average min-max - The computed range is [moving average min observed value, moving average max observed value], which works well with activations.
- Histogram - Records a histogram of values along with min and max values, then chooses according to some criterion.
- Entropy - The range is computed to minimise the error between the full-precision and the quantised data.
- Mean Square Error (MSE) - The range is computed as the minimisation of the MSE between the full-precision and the quantised data.
- Percentile - The range is computed using a given percentile value p on the observed values. The idea is to have p% of the observed values in the computed range. While this is possible when doing affine quantisation, it is not always possible to exactly match that when doing symmetric quantisation.

To effectively quantise a model to int8, the steps to follow are

1. Choose which operators to quantise. Good operators to quantify dominant computation time (linear projections and matrix multiplications).
2. Try post-training dynamic quantisation. If it is fast enough, stop here; otherwise, continue to step 3.
3. Try post-training static quantisation. Apply observers to your models in places where you want to quantise.
4. Choose a calibration technique and perform it.
5. Convert the model to its quantised form: the observers are removed, and the float32 operators are converted to their int8 counterparts.
6. Evaluate the quantised model: Is the accuracy good enough? If yes, stop here. Otherwise, start again at step 3, with quantisation-aware training.

In summary, unlike PTQ, QAT integrates the weight conversion process during the training stage. It also integrates quantisation into the training process and simulates the effects of quantisation during training. Thus, QAT helps train the model to be robust to quantisation noise. A highly used QAT technique is the QLoRA.

Fine-tuning after quantisation—There are different viewpoints on this. Some say quantised models cannot be fine-tuned, while others insist that they can be fine-tuned to enhance performance, such as accuracy. Fine-tuning involves some retraining on task-specific data.

Weight sharing involves clustering similar weights and sharing the same quantised value. This reduces the number of unique weights, achieving more compression. By limiting the number of unique weights, weight sharing saves energy using large neural networks.

Hybrid quantisation combines different quantisation techniques within the same model. For example, weights may be quantised to 8-bit precision while activations remain at higher precision. Another option is that different layers can use different precisions chosen based on the layer's sensitivity to quantisation. This reduces models' size by applying quantisation to both the weights and the activations.

Per–tensor quantisation applies the same quantisation scale across an entire tensor (e.g., all weights in a layer). That is, quantisation parameters can be computed on a per-tensor basis. This means one pair of (S, Z) will be used per tensor.

Per-channel quantisation uses different quantisation scales for different channels within a tensor. This enables finer granularity in quantisation, guaranteeing better accuracy for convolutional neural networks (CNNs).

It is possible to store a pair of (S, Z) per element along one of the dimensions of a tensor. For example, for a tensor of shape [N, C, H, W], having per-channel quantisation parameters for the second dimension would result in having C pairs of (S, Z). While this can improve accuracy, it requires more memory.

Adaptive quantisation methods adjust the quantisation parameters dynamically based on the input data distribution. These can guarantee better accuracy by fitting the quantisation to the specific characteristics of the data.

The primary libraries used for QAT are PyTorch and TensorFlow. PyTorch offers QAT through its torch.ao library. TensorFlow utilises its Model Optimization Toolkit for QAT. Additionally, libraries like NNCF and AQT can be used for QAT, particularly with TensorFlow and JAX.

Demystifying the Promising Techniques and Tools for Quantization

QLoRA (Quantized Low-Rank Adaptation) is an advanced quantisation technique that builds upon the Low-Rank Adaptation (LoRA) method to enable more efficient fine-tuning of large language models (LLMs). QLoRA combines quantisation with parameter-efficient fine-tuning to significantly reduce memory requirements while maintaining model performance. The key features of QLoRA include:

1. **4-bit quantisation** - QLoRA quantises the base LLM weights to 4-bit precision, drastically reducing memory usage.
2. **LoRA integration** - This freezes the quantised base model's weights and only updates a small set of trainable parameters (Low-Rank Adapters) during fine-tuning.
3. **Dual data types** - QLoRA uses two data types. 4-bit NormalFloat (NF4) for storing quantised weights. 16-bit BrainFloat (BFloat16) for computations during forward and backward passes.
4. **Double quantisation (DQ)** - This technique further reduces memory usage by quantising the quantisation constants themselves.

Herein, weights are quantised in blocks of 64. 32-bit scaling factors for each block are compiled into blocks of 256. These blocks are then quantised to 8-bit, and DQ reduces the memory overhead from 0.5 bits to 0.127 bits per weight. QLoRA can fine-tune a 65B parameter model on a 48GB GPU while maintaining performance comparable to full 16-bit fine-tuning.

QLoRA's first approach to quantisation is 4-bit NormalFloat or NF4. It is a new information-theoretically optimal data type built on Quantile Quantization techniques. NF4 estimates the $2k + 1$ quantiles (where k is the number of bits) within a normalised range of [−1, 1]. This ensures that each quantisation bin contains an equal number of data points, optimising the representation of normally distributed values such as large language model weights. Unlike standard quantisation methods that divide data into equally spaced bins, NF4 adjusts bin sizes according to data density (equally-sized bins), keeping precision and dynamic range even within a limited 4-bit format.

Double quantisation in QLoRA further reduces memory usage by applying quantisation to the constants from the initial quantisation process. This two-step process starts with quantising the model weights to a 4-bit precision using NormalFloat (NF4). Next, the first step's quantisation constants (scales and zero points) are further quantised to a lower 8-bit precision. This is important because QLoRA uses Block-wise quantisation. Instead of quantising all weights, this method divides them into smaller blocks or chunks of weights, which are then quantised independently. While this block-wise approach improves accuracy, it also generates multiple quantisation constants, which can consume more memory. By applying a second quantisation level to these constants, QLoRA effectively reduces their storage requirements and optimises memory utilisation during training or fine-tuning.

Advantages of QLoRA Over Standard LoRA - QLoRA has some important benefits compared to standard LoRA, mainly because it uses quantisation.

- Improved Memory Efficiency: QLoRA reduces memory usage compared to standard LoRA. Quantising model weights to lower precision minimises the model's size, enabling fine-tuning on consumer-grade GPUs with limited memory.
- Faster Training: Reducing memory size through quantisation leads to faster training times. This helps in quicker experimentation and iteration, accelerating the development process.
- Comparable Performance: QLoRA achieves accuracy comparable to full fine-tuning and standard LoRA.
- Block-wise Quantization for Fine-Grained Control: QLoRA's use of block-wise quantisation ensures that each segment of model weights is optimised

independently, allowing flexibility and additional compression without affecting the model's performance.

Pruned and Rank-Increasing Low-Rank Adaptation (PRILoRA)

PRILoRA is a fine-tuning technique recently proposed by researchers that aims to increase LoRA efficiency through the introduction of two additional mechanisms

- The linear distribution of ranks
- Ongoing importance-based A-weight pruning

Returning to the concept of low-rank decomposition, LoRA achieves fine-tuning by combining two matrices:

- W contains the entire model's weights
- AB represents all changes made to the model by training the additional weights, i.e., adapters.

The AB matrix can be decomposed into smaller matrices of lower rank – A and B – hence the term low-rank decomposition. While the low-rank r is the same across all the LLM layers in LoRA, PRILoRA linearly increases the rank for each layer. For example, the researchers who developed PRILoRA started with r = 4 and increased the rank until r = 12 for the final layer – producing an average rank of 8 across all layers.

Optimising LLMs with PRILoRA for Efficient Fine-Tuning - PRILoRA prunes the A matrix, eliminating the lowest, i.e., least significant weights every 40 steps throughout the fine-tuning process. The lowest weights are determined through an importance matrix, which stores both the temporary magnitude of weights and the collected statistics related to the input for each layer. Pruning the A matrix in this way reduces the number of weights that must be processed, reducing the time required to fine-tune an LLM and the memory requirements of the fine-tuned model. Although still a work in progress, PRILoRA showed encouraging results on benchmark tests conducted by researchers. This included outperforming full fine-tuning methods on 6 out of 8 evaluation datasets while achieving better results than LoRA on all datasets.

GGML/GGUF (Georgi Gerganov Machine Learning / GPT-generated unified format)

GGML quantises models to run efficiently on CPUs. GGUF is an updated format that extends GGML's capabilities to include non-Llama models and is more extensible.

- **k-Quant system:** This system divides model weights into blocks and quantises them using various bit-width methods depending on their importance (e.g., q2_k, q5_0, q8_0).
- **GGUF:** Extends GGML to support a broader range of models and is backward-compatible**.**

GGUF, the successor of GGML, is a quantisation method designed for LLMs. It allows users to run LLMs on a CPU while offloading some GPU layers, offering speed improvements. GGUF is particularly useful for those running models on CPUs or Apple devices. GGUF was introduced as a more efficient and flexible way of storing and using LLMs for inference. It was tailored to rapidly load and save models, with a user-friendly approach for handling model files. This is not less suited for GPU execution. This slows down inference speed compared to GPU-optimized methods. Although using the CPU is generally slower than using a GPU for inference, it is an incredible format for those running models on CPU or Apple devices.

Activation-Aware Weight Quantization (AWQ)

AWQ protects salient weights by observing activations rather than the weights themselves. AWQ achieves excellent quantisation performance, especially for instruction-tuned LMs and multi-modal LMs. It provides accurate quantisation and offers reasoning outputs. AWQ can easily reduce GPU memory for model serving and speed up token generation. It has been integrated into various platforms, including NVIDIA TensorRT-LLM, FastChat, vLLM, HuggingFace TGI, and LMDeploy.

- AWQ takes an activation-aware approach, by observing activations for weight quantisation.
- It excels in quantisation performance for instruction-tuned LMs and multi-modal LMs.
- AWQ provides a turn-key solution for efficient deployment on resource-constrained edge platforms.

AWQ is a novel quantisation method akin to GPTQ. While there are multiple distinctions between AWQ and GPTQ, a crucial divergence lies in AWQ's assumption that not all weights contribute equally to an LLM's performance. In essence, AWQ selectively skips a small fraction of weights during quantisation by mitigating quantisation loss. Consequently, their research indicates a noteworthy acceleration in speed compared to GPTQ, all by maintaining comparable and superior performance.

In summary, GPTQ, a one-shot weight quantisation method, harnesses approximate second-order information to achieve highly accurate and efficient quantisation. AWQ takes an activation-aware approach, by protecting salient weights by observing activations, and has showcased excellent quantisation performance, particularly for instruction-tuned LMs. GGUF, on the other hand, represents a new format designed for flexibility.

SpQR (sparse quantised representations)

SpQR combines sparsity with quantisation to optimise model size and performance.

- **Sparse representation**: Introduces sparsity by pruning non-essential weights, reducing the number of weights to be quantised.
- **Quantisation**: Applies quantisation to the sparse weights to further reduce memory and computation.

This reduces the number of weights and their precision, enabling potentially significant memory savings. However, if too many weights are pruned or quantised too aggressively, this may lead to reduced model performance.

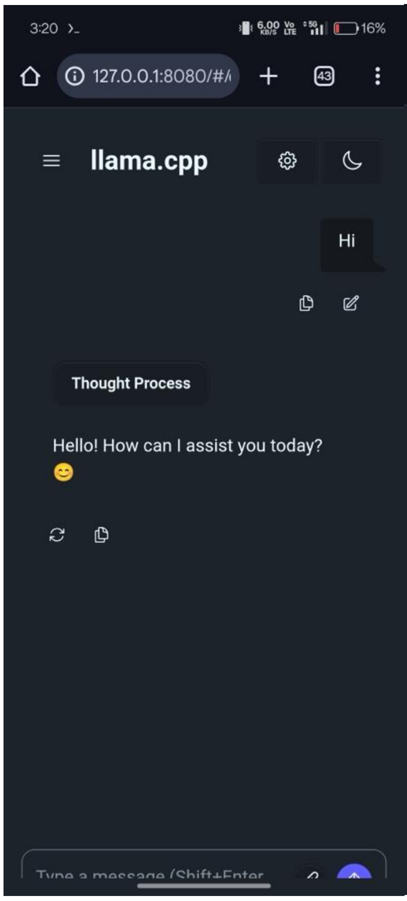
The Practical Experimentation Results and Findings

1. **AWQ Quantization of LLaMA-3.2-1B Model -** The quantization of the LLaMA-3.2-1B model using AWQ (Activation-aware Weight Quantization) to optimize memory usage and inference speed. The quantized model achieved significant

memory savings (approximately 60% reduction) while maintaining comparable response quality, though inference times were longer than the original model. The experiment was conducted on a GPU-enabled environment (CUDA) using the Hugging Face Transformers and AutoAWQ libraries. The results demonstrate that AWQ quantization is a viable approach for deploying large language models on resource-constrained environments, with some trade-offs in inference speed.

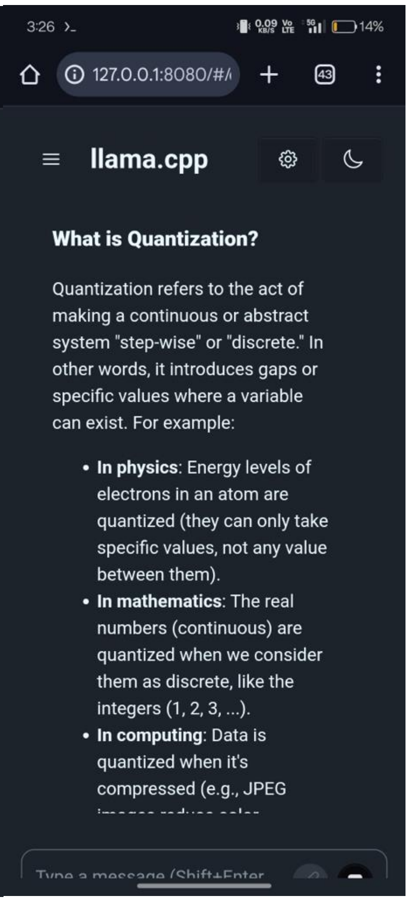
2. **GGUF Quantization Report for Llama-3.2-1B and Qwen3 Model -** The unsloth/Llama-3.2-1B and Qwen3 model from Hugging Face to the GGUF format, followed by quantization to Q4_K_M and Q8_0 formats using llama.cpp. The quantized models were tested for inference performance, memory usage, and output quality using a set of predefined prompts. The results demonstrate significant reductions in model size and inference time with quantization, though with some trade-offs in output quality.

   Running GGUF Model on Mobile using llama.cpp
   Deployment and interaction with a 4-bit quantized GGUF model (qwen3-1.7b-q4_0) on a mobile ARM64 processor using llama.cpp. The process involved compiling the server directly in Termux on Android and interacting with the model through a browser-based interface served locally at 127.0.0.1:8080.



Hi

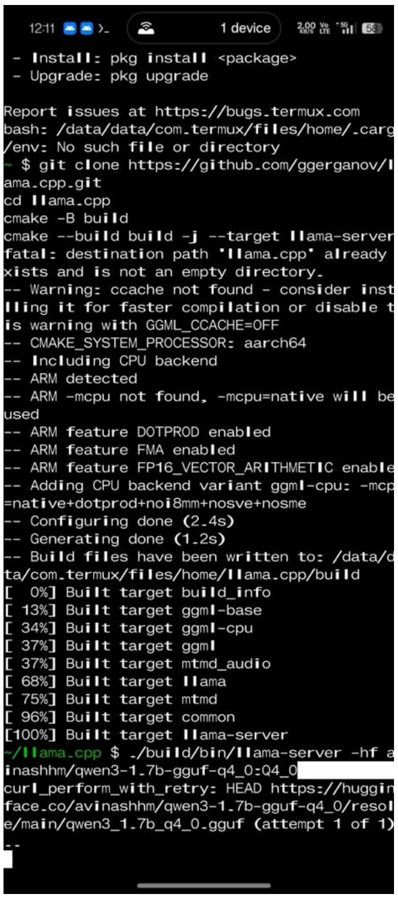
Hello! How can I assist you today? 😊



What is Quantization?

Quantization refers to the act of making a continuous or abstract system "step-wise" or "discrete." In other words, it introduces gaps or specific values where a variable can exist. For example:

- **In physics**: Energy levels of electrons in an atom are quantized (they can only take specific values, not any value between them).
- **In mathematics**: The real numbers (continuous) are quantized when we consider them as discrete, like the integers (1, 2, 3, ...).
- **In computing**: Data is quantized when it's compressed (e.g., JPEG



```
 - Install: pkg install <package>
 - Upgrade: pkg upgrade

Report issues at https://bugs.termux.com
bash: /data/data/com.termux/files/home/.carg
/env: No such file or directory
~ $ git clone https://github.com/ggerganov/l
ama.cpp.git
cd llama.cpp
cmake -B build
cmake --build build -j --target llama-server
fatal: destination path 'llama.cpp' already
xists and is not an empty directory.
-- Warning: ccache not found - consider inst
lling it for faster compilation or disable t
is warning with GGML_CCACHE=OFF
-- CMAKE_SYSTEM_PROCESSOR: aarch64
-- Including CPU backend
-- ARM detected
-- ARM -mcpu not found, -mcpu=native will be
used
-- ARM feature DOTPROD enabled
-- ARM feature FMA enabled
-- ARM feature FP16_VECTOR_ARITHMETIC enable
-- Adding CPU backend variant ggml-cpu: -mcp
=native+dotprod+noi8mm+nosve+nosme
-- Configuring done (2.4s)
-- Generating done (1.2s)
-- Build files have been written to: /data/d
ta/com.termux/files/home/llama.cpp/build
[  0%] Built target build_info
[ 13%] Built target ggml-base
[ 34%] Built target ggml-cpu
[ 37%] Built target ggml
[ 37%] Built target mtmd_audio
[ 68%] Built target llama
[ 75%] Built target mtmd
[ 96%] Built target common
[100%] Built target llama-server
~/llama.cpp $ ./build/bin/llama-server -hf a
inashhm/qwen3-1.7b-gguf-q4_0:Q4_0
curl_perform_with_retry: HEAD https://huggin
face.co/avinashhm/qwen3-1.7b-gguf-q4_0/resol
e/main/qwen3_1.7b_q4_0.gguf (attempt 1 of 1)
--
```

3. **LLaMA Model Quantization using GPTQ -** The quantization process and performance evaluation of the LLaMA-3.2-1B model, quantized to 4-bit precision using GPTQ. Conducted on a CUDA-enabled GPU, the process

achieved significant memory reduction while maintaining acceptable response quality. Key metrics include memory usage, inference time, and qualitative response accuracy compared to the original model. The quantized model shows substantial memory efficiency but has slower inference times and occasional response inaccuracies.

4. **Quantization for Qwen3-8B Model using BitsAndBytes Library -** The Qwen3-8B model was quantized to 4-bit precision using the BitsAndBytes library on a CUDA-enabled GPU environment. The quantized model achieved a memory usage of 13778.50 MB, which is a reduction compared to the unquantized model's expected memory footprint (estimated at ~30 GB for an 8B parameter model in FP16). Inference times averaged 26.7373 seconds per prompt, with a high perplexity of 103840.12, indicating potential degradation in language modelling capability. Response quality was generally acceptable for factual and computational prompts but showed limitations in creative tasks. The experiment demonstrates that 4-bit quantization with BitsAndBytes enables deployment on resource-constrained environments, though with trade-offs in inference speed and output quality.

Code Implementation

```
from transformers import AutoModelForCausalLM, AutoTokenizer,
BitsAndBytesConfig
import torch, time
model_id = "Qwen/Qwen3-8B"
quant_config = BitsAndBytesConfig(load_in_4bit=True,
bnb_4bit_compute_dtype=torch.float16)
model = AutoModelForCausalLM.from_pretrained(model_id,
quantization_config=quant_config, device_map="auto")
tokenizer = AutoTokenizer.from_pretrained(model_id)
prompt = "Explain why the sky is blue."
inputs = tokenizer(prompt, return_tensors='pt').to("cuda")
with torch.no_grad():
outputs = model.generate(inputs, max_new_tokens=100)
print(tokenizer.decode(outputs[0]))
```

Finetuning of Quantized Models for Improved Accuracy

We have finetuned an AWQ-quantized LLaMA model using LoRA (Low-Rank Adaptation) on the Alpaca English demo dataset. The idea is to evaluate its inference performance, memory usage, and response quality through interactive testing, and analyze the effectiveness of the fine-tuning process for deployment optimization.

The AWQ-quantized LLaMA model, stored at `/content/AutoAWQ/llama-AWQ`, was fine-tuned using LoRA on the Alpaca English demo dataset (`alpaca_en_demo.json`) with 1000 examples. The fine-tuning process, conducted on a CUDA-enabled GPU, took 19 minutes and 49.82 seconds, achieving a final training loss of 1.0431 and an average train loss of 1.1613 over 3 epochs. The fine-tuned model, saved to `llama-AWQ-finetuned`, maintained a memory footprint of 1012.50 MB allocated and 2922.00 MB reserved during inference. Interactive testing showed average inference times of ~35.5 seconds for detailed prompts, with responses varying in quality—accurate for factual queries but verbose and sometimes off-topic for open-ended tasks. The results demonstrate that fine-tuning an AWQ-quantized model with LoRA is effective for adapting to specific datasets while preserving memory efficiency, though response quality and inference speed require further optimization.

Model Details

Model: AWQ-quantized LLaMA
Quantization Method: AWQ (Activation-aware Weight Quantization, pre-applied)
Fine-Tuning Method: LoRA (Low-Rank Adaptation)
LoRA Configuration:

1. Rank (`r`): 16
2. LoRA Alpha: 32
3. Target Modules: `q_proj`, `v_proj`, `k_proj`, `o_proj`
4. LoRA Dropout: 0.05
5. Bias: None
6. Task Type: Causal Language Modeling.

Dataset: Alpaca English demo (`alpaca_en_demo.json`, 1000 examples)
Tokenizer: AutoTokenizer from Hugging Face, configured with padding and EOS token
Hardware: GPU (CUDA-enabled, specific GPU not specified)
Software:

- Python 3
- PyTorch
- Transformers
- AutoAWQ
- PEFT (Parameter-Efficient Fine-Tuning)
- Datasets

Generation Parameters

- Maximum New Tokens: 200 (300 for code generation)
- Number of Beams: 3 (beam search for higher quality)
- No-Repeat N-gram Size: 2
- Early Stopping: Enabled

Results

Memory Usage

- Allocated GPU Memory: 1012.50 MB
- Reserved GPU Memory: 2922.00 MB

The fine-tuned model maintained a low memory footprint, consistent across all inference tests, making it suitable for resource-constrained environments.

Training Performance

- Total Training Time: 19 minutes and 49.82 seconds (1189.82 seconds)
- Average Train Loss: 1.1613
- Final Training Loss: 1.0431 (at step 750)

- Train Samples per Second: 2.521
- Train Steps per Second: 0.63

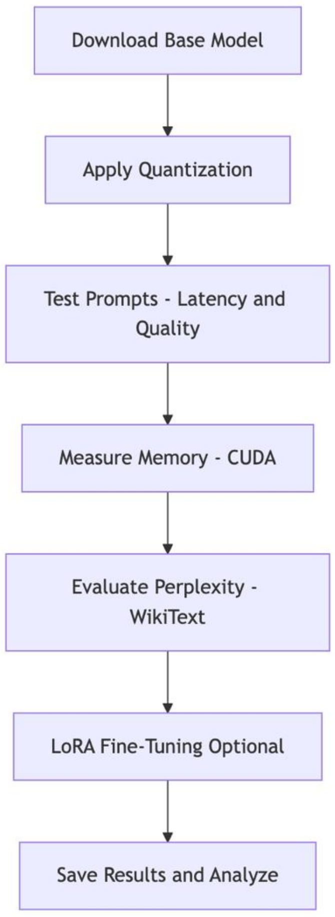
Training loss decreased steadily over 750 steps, from 1.8546 (step 5) to 1.0431 (step 750), indicating effective learning on the Alpaca dataset. The low percentage of trainable parameters (1.2805%) ensured efficiency during fine-tuning.

Inference Time

1. Prompt 1 ("hi"): 4.2212 seconds
2. Prompt 2 ("what is quant?"): 34.9074 seconds
3. Prompt 3 ("Transform the following sentence..."): 36.8615 seconds
4. Prompt 4 ("Design a project..."): 36.4094 seconds
5. Prompt 5 ("Suggest three ways..."): 36.3539 seconds
6. Average Inference Time (Prompts 2-5): ~35.5 seconds.

Inference time was significantly lower for the simple greeting prompt (4.2212 seconds) but averaged ~35.5 seconds for more complex prompts, indicating a potential bottleneck in processing longer or more detailed responses.

It is well-known that quantization, while supporting democratization and accessibility, degrades LLMs' performance. Therefore, the workaround is to finetune the quantized model to guarantee better response quality (accuracy and relevance). Thus, quantization and finetuning collectively make a good choice to derive high-quality quantized LLMs.


Download Base Model
Apply Quantization
Test Prompts - Latency and Quality
Measure Memory - CUDA
Evaluate Perplexity - WikiText
LoRA Fine-Tuning Optional
Save Results and Analyze

LLM Quantization at the Edge

These days, the realization of intelligent and real-time services and applications is indispensable for setting up and sustaining real-time organizations that can boost customer experience substantially. That is, besides business workloads, information and communication systems have to guarantee real-time capabilities in their deliveries to be right and relevant to their users. Considering this unique yet challenging requirement, there is a slow yet steady transition to leverage IoT devices to collect, cleanse and crunch local data to emit actionable insights in time. Such a data collection and proximate processing using one or more edge devices leads to the realization of real-time knowledge discovery and dissemination. In other words, on-device data processing using highly competent AI models and agents leads to real-time edge intelligence.

Edge AI is being termed as the game-changing technology at the intersection of edge computing and AI. That is, running AI models on edge devices is to open a variety of innovations and disruptions for business behemoths and start-ups. Edge devices are destined to be intelligent in their operations, outputs and offerings through the incorporation of high-end AI models into them methodically.

Enter Edge AI 2.0 – Without an iota of doubt, the aspect of generative AI is very flourishing everywhere these days with the rise of many language models (large, small, vision and multimodal). Natural language processing, understanding and generation (NLP/U/G) use cases are being elegantly realized and used widely. Due to the large-scale computational and memory requirements, language models are being trained and run in cloud environments. But for producing real-world applications, language models have to be deployed in edge devices to collect and process edge environment data to fulfil the long-pending goal of real-time inferencing. However, there are a few critical technical challenges in running language models on edge devices such as smartphones, robots, drones, consumer electronics, medical instruments, industry machinery, etc.

For installing and using language models on edge devices, which are typically resource-constrained, experts and exponents advocate for large language model (LLM) optimization and compression techniques and tools. In the subsequent sections, we are to focus on the ways and means of LLM quantization at the edge.

As explained above, quantization sharply lessens the computational and memory requirements for running and inferring from LLMs by using lower bits for representing a LLM's weights. Such a reduction eventually leads to faster inference and reduced power usage. Especially LLMs through quantization can be made to run on edge devices.

Low-bit Quantization on Resource-constrained Edge Devices – Training large language models (LLMs) turns out to be a resource-intensive activity. Therefore, the viable approach for reducing the resource consumption is to use lower bits to represent LLM weights and activations. This process brings down the memory footprint and boosts the efficiency. Such low-bit LLMs needs the capability of mixed-precision matrix multiplication (mpGEMM) that involves the multiplication of lower-precision weights with higher-precision activations. There is a recent advance in the form of mixed-precision matrix multiplication (mpGEMM). This mpGEMM element allows data of the same or different formats to be multiplied.

As known, many hardware supports only symmetric computations but with mpGEMM in place, this hardware limitation can be overcome easily. That is, to fully gain the distinct advantages of applying low-bit quantization on resource-constrained edge devices, hardware has to natively extend support for mpGEMM. There are three approaches articulated by the mpGEMM development team. For more information, readers can check this paper (LUT Tensor Core: A Software-Hardware Co-Design for LUT-Based Low-Bit LLM Inference). This paper talks building a special compiler to support different low-precision data types by converting unsupported types into hardware-compatible data types. Furthermore, there is a library to implement GEMM using a lookup table (LUT) approach. Finally, the team has introduced an innovative design for next-generation AI hardware that enables low-bit quantization.

As large models systematically adopt low-bit quantization, the LUT method could become the standard for system and hardware design. That is, LUT is to advance a series of hardware innovations. The paper (LUT Tensor Core: A Software-Hardware Co-Design for LUT-Based Low-Bit LLM Inference by Zhiwen Mo and his team) talks about a new proposal of LUT Tensor Core, a software hardware co-design for low-bit LLM inference.

mpGEMM facilitates lower-precision weights getting multiplied with higher-precision activations. However, GPUs and TPUs, the most prominent hardware for LLM training and inferencing, do not support mpGEMM. Hence a group of researchers have introduced MixPE. This is a specialized mixed-precision processing element designed for efficient low-bit quantization in LLM inference. The paper titled as MixPE: Quantization and Hardware Co-design for Efficient LLM

Inference) throws more light on this. The research team lead by Yu Zhang artistically leverage two key innovations to minimize dequantization overhead. Firstly, the researchers perform dequantization after per-group mpGEMM, to significantly reduce dequantization overhead. Second, MixPE, instead of relying on conventional multipliers, utilizes efficient shift & add operations for multiplication,

Quantized Transformer-based Language Models for Edge Devices – Bidirectional Encoder Representations from Transformers (BERT) models are widely used for NLP use cases. Due to the big size, running these models on resource-constrained devices is a tough thing and the inference latency is on the higher side. As a workaround, a group of researchers have articulated a solution approach of converting such big models into an optimized FlatBuffer format, which is tailored for deployment on resource-constrained edge devices. In this paper “Quantized Transformer Language Model Implementations on Edge Devices“talks about the performance of such FlatBuffer transformed MobileBERT models on different edge devices. The evaluation has shown that there is a memory footprint reduction with a small drop in accuracy,

DILEMMA – This is a novel framework for jointly optimizing layer placement and layer quantization in edge devices. This framework formulates an Integer Linear Programming problem to minimize total inference delay while ensuring acceptable LLM performance levels. This leverages layer-wise quantization and knowledge distillation for LLM performance control. There is more on this framework in the paper

“DILEMMA: Joint LLM Quantization and Distributed LLM Inference Over Edge Computing Systems” authored by Minoo Hosseinzadeh and Hana Khamfroush

MobileQuant – As inscribed above, reducing the number of bits being used, quantization methods have acquired the attention of many these days as the usage of LLMs grows fast. There are many definite successes being obtained through the process of quantization. That is, quantizing LLMs to lower bit widths (4-bit weights) gives a good result. But quantizing activations beyond 16 bits results in computational overheads as there is a poor on-device quantization support. 8-bit activations are found good for on-device deployment. This is because 8-bit activations enable LLMs to fully exploit mobile-friendly hardware (Neural Processing Units (NPUs)). The research team has attempted to deploy LLMs on edge devices using integer-only quantization. The team has brought in a simple post-training quantization (PTQ) method (MobileQuant) that extends previous weight equivalent transformation works by jointly optimizing the weight transformation and activation range parameters in an end-to-end manner. The paper titled as “MobileQuant: Mobile-friendly Quantization for On-device Language Models” published by Fuwen Tan and his team, has more relevant information.

FlexQuant – This is a novel framework that can generate an ensemble of quantized Models. The ensemble model guarantees 15x granularity improvement and 10x storage reduction compared to the state-of-the-art models. FlexQuant brings great performance and flexibility to the edge deployment of LLMs. Yuji Chai and his team meticulously produced this futuristic framework (FlexQuant: Elastic Quantization Framework for Locally Hosted LLM on Edge Devices)

LR-QAT – As reported earlier, there are a few promising and potential quantization methods.
Quantization-aware training (QAT) methods are widely recognized as the ones that produce the best quantized performance. One prominent issue with QAT methods is the high cost associated with long training hours and huge memory usage. There are workarounds in the form of parameter-effic8ent fine-tuning and low-rank adaptation (LoRA). The development team led by Yelysei Bondarenko, has come out with Low-Rank Quantization-Aware Training for LLMs (LR-QAT), a lightweight and memory-efficient QAT algorithm for LLMs. LR-QAT employs several components to save memory without sacrificing predictive performance.

Edge Intelligence Optimization - Here is an inference model for transformer decoder-based LLMs. This solution is to increase the inference throughput via batch scheduling and joint allocation of communication and computation resources. The batching technique and model quantization play a vital role here. Xinyuan Zhang and his team has published an informative and inspiring paper titled as “Edge Intelligence Optimization for Large Language Model Inference with Batching and Quantization”. Readers can get more details from the paper.

Agile-Quant – There are 8-bit or lower weight quantization methods that guarantee end-to-end task performance while the activation is not quantized. Xuan Shen and his team has come out with a novel framework (Agile-Quant). This is an activation-guided quantization framework for LLMs. The author team also has implemented an end-to-end accelerator on multiple edge devices for faster inference. They have introduced a basic activation quantization strategy to close the gap between task performance and real inference speed. Further on, they leverage the activation-aware token pruning technique to reduce the outliers and the adverse impact on attentivity. More intuitive details are given in the paper "Agile-Quant: Activation-Guided Quantization for Faster Inference of LLMs on the Edge"

EdgeQAT – As indicated in the beginning of the paper, post-training quantization (PTQ) methods contribute to minimizing the size of LLMs. However, PTQ methods dramatically degrade in quality when quantizing weights, activations, and KV cache together to below 8 bits. On the other hand, quantization-aware training (QAT) methods quantize model weights. But activations are not subjected to any quantization. That is, the full potential of quantization is not exploited to reduce inference latency. For edge devices to emit knowledge quickly, the requirement of accelerated inference must be realized. Xuan Shen and his team has developed EdgeQAT, which is the entropy and distribution guided QAT method. This method empowers lightweight LLMs to guarantee accelerated inference. The insightful details are found in the paper titled "EdgeQAT: Entropy and Distribution Guided Quantization-Aware Training for the Acceleration of Lightweight LLMs on the Edge"

We have leveraged llama.cpp package to run large language models in any smartphone**.**

Comparison of Quantisation Techniques

We have implemented various quantization methods to gain a deeper understanding of each of them. The table below tells the unique properties of each of them. The model size reduction ratios for each of the methods are etched in the table. Also what is the hardware support needed for those methods is also indicated in the table. The full source code for each of the implementations is given in the GitHub portal.

Comparative Summary of Quantization Techniques

Quantization Method	**Inference Platform**	**Inference Speed**	**Inference Quality**	**Model Size Reduction**	**Compatibility / Notes**
AWQ	GPU	Moderate (slower than FP16)	High	Significant (~60%)	Requires GPU with support for INT4 weights and FP16 activations.
GGUF (Q4_K_M)	CPU	High (3–7× faster than FP16)	Moderate	Significant (~50%)	CPU-only; ideal for mobile or offline deployment.
GGUF (Q8_0)	CPU	Moderate (faster than FP16)	Moderate	Moderate (~30%)	CPU-only; better quality than Q4_K_M but less compression.
GPTQ	GPU	Low (slower	Moderate	Significant (~45–61%)	Well-suited for transformer models; widely

		than FP16 baseline)			supported across frameworks.
BitsAndBytes (4-bit)	GPU	Moderate (slower than expected)	Moderate to High (task-dependent)	Significant (~54%)	Requires GPU with 4-bit compute support; supports many Hugging Face models directly.

The summary highlights the trade-offs of quantisation for LLMs:

- AWQ and GPTQ are ideal for GPU environments, offering up to 60% memory savings (as seen with LLaMA-3.2-1B and GPT-2). However, they may increase inference time (e.g., AWQ slowed down LLaMA-3.2-1B, and GPTQ increased GPT-2's inference time to 2.77s).

- GGUF Q4_K_M excels on CPUs, achieving 3-7X faster inference (e.g., LLaMA-3.2-1B inference time reduced to 15-35s), making it efficient for CPU-based deployments.

- BitsAndBytes reduces memory usage for Qwen3-8B but has high perplexity, indicating a trade-off in language modelling capability.

Real-World Applications

As mentioned above, the ideals of efficient and edge AI must be fulfilled. Model quantisation is pronounced as the way forward. Several real-world applications yearn for quantised AI models that enable high performance, resource efficiency, and model performance.

Smartphone applications—Quantised AI models are increasingly deployed in smartphones, handheld devices, and digital assistants. Mobile devices are used to accomplish real-world tasks such as image recognition, speech recognition, text translation, and augmented reality.

Autonomous vehicles – Self-driving cars are gaining much mind and market share. Quantised AI models can take faster decisions from data emanating from multiple sensors. That is, the real-time inferences can become the reality with quantisation techniques. Quantised AI models can process sensor and actuator data in real time to identify obstacles, read traffic signs, and make decisions. Shrunken AI models can be comfortably deployed and run in any reasonably resource-intensive edge devices such as car infotainment systems.

Edge devices – Today, most important locations are being stuffed with IoT devices to monitor, measure and manage everything happening in the locations. We have all kinds of sensors and actuators, medical instruments, defence equipment, information appliances, manufacturing machinery, kitchen utensils, consumer and automotive electronics, robots, drones, wares, digital assistants, and embedded systems in our everyday environments. As most edge devices are resource-constrained, applying a suitable quantisation technique is essential to arrive at shrunken AI models that can instantaneously run and infer from local data. Edge computing gains momentum to realise and release many real-world and real-time services and capabilities. With dynamically setting up and sustaining edge device clusters/clouds, edge data analytics gains speed and sagacity. With AI emerging as the most advanced and automated data analytics technology, edge AI is flourishing with proper nourishment from technology giants. Environment monitoring and surveillance, pinpointing any deviation from everyday happenings (anomaly or outlier detection), real-time data capture, processing, decision-making, action towards the production of real-time services and applications, federated learning for continuous learning, adaptation and improvement, etc., are some of the renowned edge-native applications being facilitated by quantisation.

Healthcare—As mentioned above, multi-speciality hospitals have several kinds of medical equipment and electronics. Medical diagnosis, prognosis, and treatment are simplified and sped up through various medical devices and instruments such as scanners. Multifaceted AI-powered robots and portable devices in healthcare environments tackle different tasks and assist doctors, surgeons, and other specialists in their assignments and obligations. Quantised AI models deployed in these real-world devices go a long way in streamlining medical activities.

Voice assistants—Today, we have several digital assistants, especially those that can understand and process voice commands perfectly and precisely. Quantisation is handy in empowering personal and professional devices to guarantee smooth, scintillating human-device interactions. This accelerates natural language understanding across various business domains.

Recommendation systems—We have a variety of online business-to-business (B2B) and business-to-consumer (B2C) platforms. Amazon, Netflix, and other platforms methodically use quantised models to offer real-time recommendations. These models process large amounts of user preference and feedback data to suggest products, movies, or videos, and quantisation enables AI models to perform faster data processing.

Thus, quantised AI models are the most sought-after method for affordable, real-time processing, planning, and execution. Quantisation is a method used to optimise the performance of AI models by reducing their computational and memory requirements. The lower memory footprint of quantised models makes them scalable; organisations can easily expand their IT infrastructure. Quantised AI models also reduce power energy consumption. On the other side, the quantisation may lead to degraded model performance. It is generally not recommended to train further or fine-tune quantised models. While larger quantised models (70B, 405B) show negligible performance degradation, relatively more minor models (8B) may experience slight variability in output quality.

Benefits of LLM Quantization

The size, scope and sagacity of large language models (LLMs) vary substantially. Many pioneering applications across business domains need LLM capabilities to be correct and relevant to users. However, the real challenge arises due to the astronomical model's size and complexity. An increase in the model size means exponential growth in model parameters. For instance, GPT-3.5 boasts around 175 billion parameters, while GPT-4 exceeds 1 trillion parameters. The deployment, adoption and adaptation of such complex LLMs present technical challenges as pre-trained and generic LLMs inherently require more memory and high-end hardware with many expensive GPUs. Herein, researchers strive hard and stretch further to bring forth competent and cognitive solutions. Quantisation is recognised as the best-in-class solution for surmounting the above-mentioned limitations.

Smaller LLMs like Llamav3-8B phi-3 still achieve excellent results and can run on low-power edge devices. However, they require substantial memory and computational resources, which is a problem for deployment on low-power edge devices with less working memory and processing power than cloud-based systems.

LLM quantisation helps reduce the cost of deployment and running LLMs. The inferencing latency of LLMs decreases sharply, enhancing users' experience with them. Quantised LLMs can be made to run on IoT edge devices. The primary benefit is the sharp reduction in model size. The need for large-scale memory and storage capacities is largely negated, whereas the computational capability needed lies on the lower side.

By leveraging reduced precision formats, model initialisation speeds up, and inferencing gains the much-needed speed. The emerging concept of edge AI 2.0 (which deals with running generative AI models on edge devices) will be neatly realised through quantisation. All kinds of language models can be made to run on edge devices, thereby fully fulfilling the long-pending demand for ambient intelligence. Real-time and context-aware services can be built and delivered with the real-time and accurate prediction power of quantised LLMs running on edge devices. Quantisation helps in four main areas:

- Efficiency - Quantisation reduces the amount of memory needed to store the model weights
- Speed - Quantisation enables lower precision computations and hence inferencing happens fast
- Energy consumption - Quantised models consume less power energy
- Affordability - Reduced computational and memory requirements can lower the cost of deploying and running models on public and edge clouds.
- Scalability and accessibility - Quantisation enables language models to be adaptively scalable. That is, they can run anywhere without any hitch or hurdle.
- Storage - Quantised models occupy less disk space
- Latency - Faster arithmetic operations reduce the inference latency. Therefore, real-time responses can be delivered.
- Throughput - Enhanced computational efficiency allows for processing more data in the same amount of time, increasing the model’s throughput

Impact of Quantisation on Model Performance - Quantisation inevitably introduces some performance degradation. However, the extent of this degradation depends on several factors.

- Model Architecture - Deeper and wider models are expected to be more resilient to quantisation.
- Dataset Size and Complexity - Larger and complex datasets can mitigate performance loss.
- Quantization Bitwidth - Lower bitwidths may result in larger performance drops.
- Quantization Method - The choice of quantisation methods also decides the model performance.

Evaluation Metrics - To assess the impact of quantisation, practitioners recommend various metrics.

- Accuracy measures the model's performance on a given task, such as classification.
- Model size – This indicates the reduction achieved in model size.
- Inference Speed – This indicates the inference speed achieved through quantisation.
- Energy Consumption – This gives the value of the reduction in energy consumption.

Challenges and Considerations for LLM Quantization

LLM quantisation is neither simple nor straightforward. As per the experts, there are challenges in performing model quantisation. The deployment and inference from quantised models are faster, and the resource consumption levels will decrease sharply. Therefore, quantised AI models can run in edge devices. Amid all these critical advantages, there is an issue of accuracy reduction. Researchers recommend leveraging proven techniques such as post-quantisation training (PQT) and quantisation-aware training (QAT) to manage this challenge. There are adaptive and hybrid quantisation methods to offset the accuracy loss. Finetuning of quantised models is another approach widely recommended. Mixing different precision levels instead of sticking to the identical precision level is another option to overcome the accuracy drop caused by quantisation.

Furthermore, not all hardware and software platforms uniformly support quantisation. Experts recommend using standardised frameworks such as TensorFlow or PyTorch. These frameworks greatly simplify adding additional libraries and bring competent workarounds to address incompatibility. The future research should focus on

- Bringing forth sophisticated quantisation techniques with minimal accuracy drop.
- Investigating hardware-software co-design for optimised quantisation.
- Understanding the impact of quantisation on different LLM architectures.
- Calculating the environmental benefits of LLM quantisation towards sustainability

Larger Quantised Model vs Smaller non-Quantised – This is a moot question. Arriving at the correct choice depends on diverse factors. Researchers across the world are vehemently focusing on this question. There are a few results on this, too.

For example, Meta researchers have demonstrated that in some cases, the quantised model demonstrates superior performance and allows reduced latency and enhanced throughput. The same trend can be observed when comparing an 8-bit 13B model with a 16-bit 7B model. The larger quantised models can outperform their smaller and non-quantised counterparts when comparing models with similar inference costs. This advantage becomes even more pronounced with larger networks, as they exhibit a minor quality loss when quantised. More focused research will be done to settle the question once and for all.

Conclusion

As inscribed initially, language models have become important and insightful for personal, social and professional use cases. Large language models are indispensable for processing textual data. LLMs can understand human languages and interact with humans naturally. Small language models comprising a few billion parameters are being formulated and used successfully to take language models to resource-constrained embedded devices. Further on, vision language models (VLMs) are emerging and evolving fast to tackle visual data. Multimodal language models have the power to process different modalities such as text, images, audio, etc. With a greater number of parameters, language models become better and brighter. However, the unbridled increase in the parameter count throws technical challenges such as the need for more memory, storage and computation. That is, this increased requirement of resources dampens inferencing.

LLM quantisation techniques and tools are the best-in-class solution for achieving LLM optimisation. This paper has detailed the need for LLM optimisation and all the key LLM quantisation mechanisms. We have implemented each of them and compared their performance methodically. We have also indicated which method is superior for the use cases and scenarios. We have especially articulated how LLM quantisation methods come in handy in fulfilling the vision of efficient and edge AI.

References

1. Quantization for Large Language Models (LLMs): Reduce AI Model Sizes Efficiently, https://www.datacamp.com/tutorial/quantization-for-large-language-models

2. A Guide to Quantization in LLMs, https://symbl.ai/developers/blog/a-guide-to-quantization-in-llms/

3. What are Quantized LLMs? https://www.tensorops.ai/post/what-are-quantized-llms

4. Exploring quantization in Large Language Models (LLMs): Concepts and techniques, https://medium.com/data-science-at-microsoft/exploring-quantization-in-large-language-models-llms-concepts-and-techniques-4e513ebf50ee

5. A Visual Guide to Quantization - Demystifying the Compression of Large Language Models.https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization

6. Exploiting LLM Quantization, https://arxiv.org/abs/2405.18137

7. What Makes Quantization for Large Language Models Hard? An Empirical Study from the Lens of Perturbation, https://arxiv.org/html/2403.06408v1

8. Model Quantization. https://developer.synaptics.com/docs/synap/model_quantization

9. The Ultimate Handbook for LLM Quantization, https://towardsdatascience.com/the-ultimate-handbook-for-llm-quantization-88bb7cb0d9d7/

10. LLM Quantization: Quantize Model with GPTQ, AWQ, and Bitsandbytes https://towardsai.net/p/artificial-intelligence/llm-quantization-quantize-model-with-gptq-awq-and-bitsandbytes

11. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models, https://arxiv.org/abs/2211.10438

12. LLM Quantization, https://www.linkedin.com/pulse/llm-quantization-dinesh-sonsale-lfm0f/

13. Large Language Models as Optimizers, https://arxiv.org/abs/2309.03409

14. Mastering LLM Optimization: Key Strategies for Enhanced Performance and Efficiency https://www.ideas2it.com/blogs/llm-optimization

15. When Large Language Model Meets Optimization, https://arxiv.org/html/2405.10098v1

16. Mastering LLM Techniques: Inference Optimization https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/

17. LLM Model Optimization Techniques and Frameworks, https://medium.com/@yugank.aman/llm-model-optimization-techniques-and-frameworks-e21d57744ca1

Authors

1. Pethuru Raj
2. Avinash HM