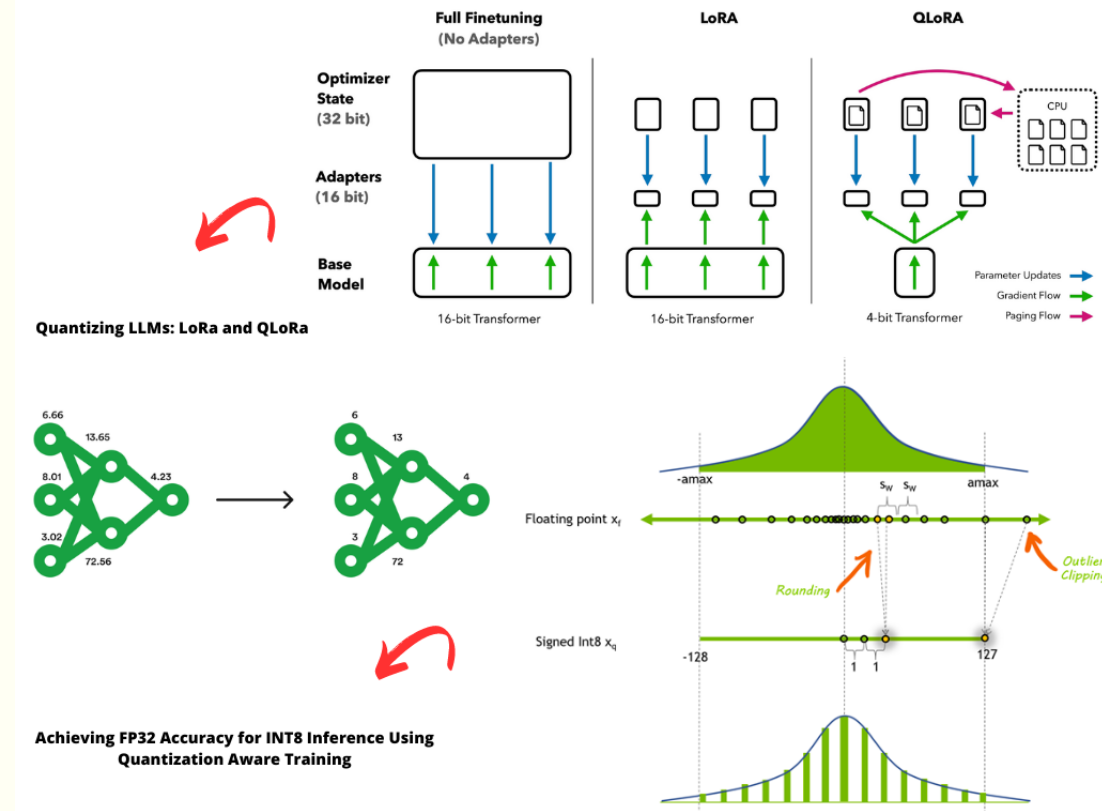


ZIPPING LLMS: QUANTIZATION & OFF-LOADING

Visit

[Murat Karakaya Akademi YouTube Channel](#)
for the tutorial video



Content

- The Need for Optimization: LLMs are huge!
- How to zip LLMs
- Quantization: Packing More Value in Less Space
- Off-loading: Sharing the Burden
- Demo: Run Mixtral-8x7B Model on Google Colab's Free Version



The Problem with LLMs: LLMs are huge!



Here are some Large Language Models (LLMs) and their VRAM requirements for inference without any quantization:

- GPT3: Requires 350 GB VRAM
- Bloom: Requires 352 GB VRAM
- Llama-2-70b: Requires 140 GB VRAM
- Falcon-40b: Requires 90 GB VRAM
- MPT-30b: Requires 60 GB VRAM
- bigcode/starcoder: Requires 31 GB VRAM
- Jurassic-1 Jumbo: Requires 150 GB VRAM
- PaLM-540B: Requires 320 GB VRAM

The Problem with LLMs: LLMs are huge!

Without Quantization:

- Falcon-40B: Approximately **90** GB of VRAM
- Falcon-180B: Approximately **640** GB of VRAM

With Quantization:

- Falcon-40B:
 - Q-4bit: **45** GB of VRAM
 - Q-8bit: **60** GB of VRAM
- Falcon-180B:
 - Q-4bit: **95** GB of VRAM
 - Q-8bit: **70** GB of VRAM



The Problem with LLMs: LLMs are huge!

With Quantization:

GPT3: For 8-bit quantization, a 13B parameter model requires more than **24GB** VRAM.

Bloom: For 8-bit quantization, a 176B parameter model requires more than **352GB** VRAM. For 4-bit quantization, it requires at least **176GB** VRAM.

Llama-2-70b: For 8-bit quantization, a 70B parameter model requires approximately **70-80GB** of VRAM. For 4-bit quantization, it requires at least **35GB** VRAM.

Falcon-40b: For 8-bit quantization, it requires more than **40GB** VRAM. For 4-bit quantization, it requires at least **35GB** VRAM.

MPT-30b: For 8-bit quantization, it requires less than **22GB** VRAM.

bigcode/starcoder: For 4-bit quantization, it requires approximately **11GB** VRAM.

Note that these are approximate values.

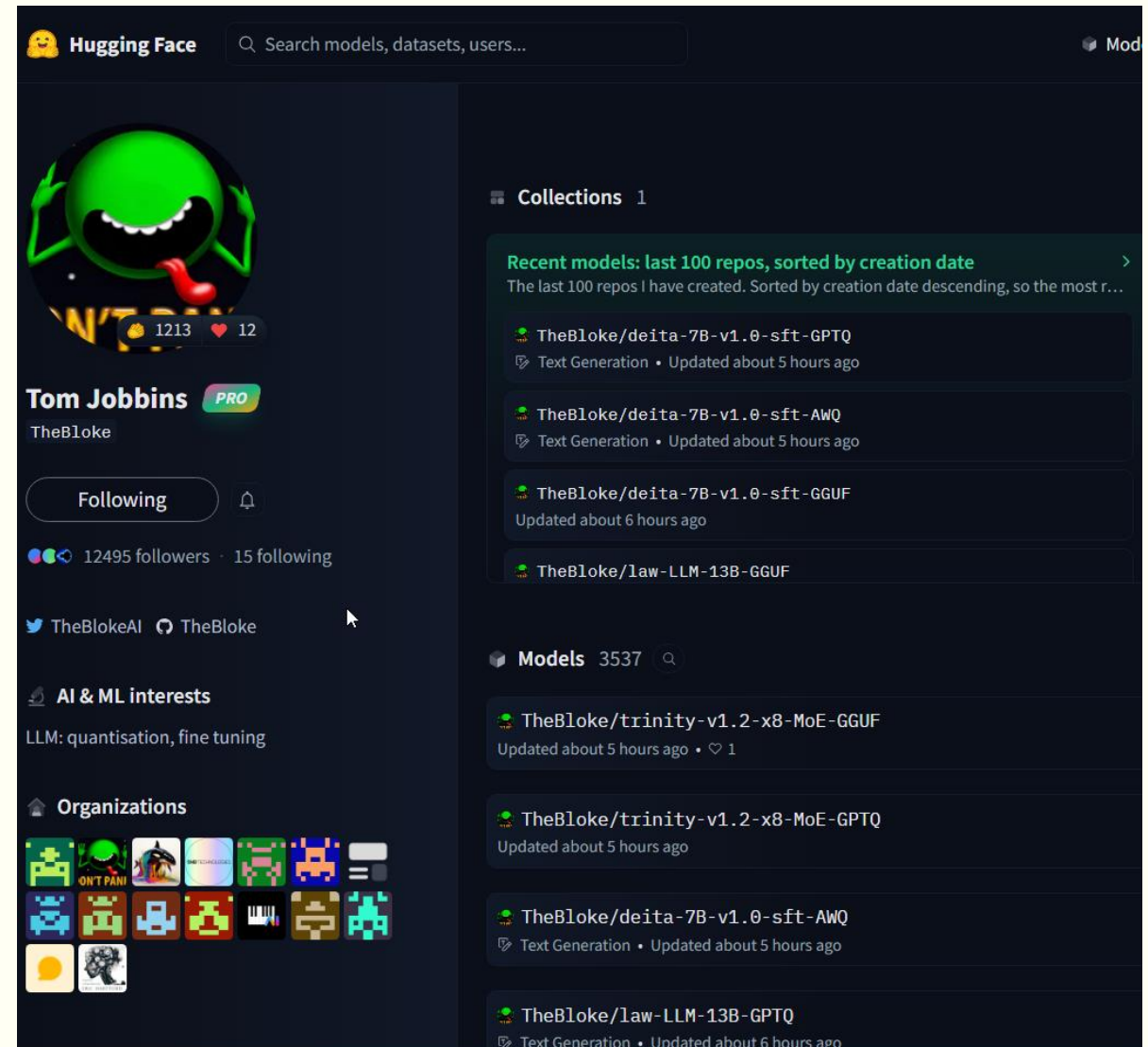


The Problem with LLMs: LLMs are huge!

You can access the quantized versions of the most of the LLMs on the HF Bloke's account:

<https://huggingface.co/TheBloke>

<https://huggingface.co/TheBloke/Mistral-7B-Instruct-v0.1-GPTQ>



Quantization & LLMs: Understanding LLM Internals

- **Embedding Layer:**

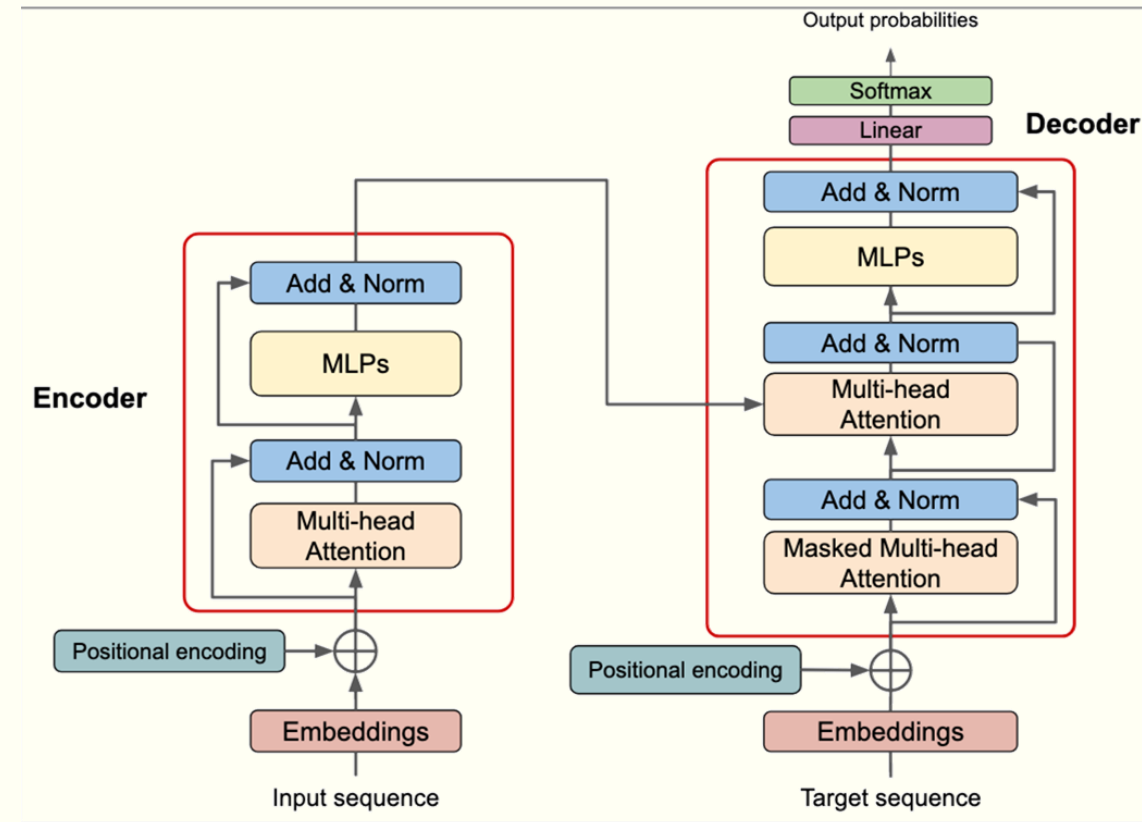
- Maps words from the vocabulary to dense vectors in a high-dimensional space.
- Captures semantic relationships between words and helps the model understand the meaning of text.

- **Transformer Blocks:**

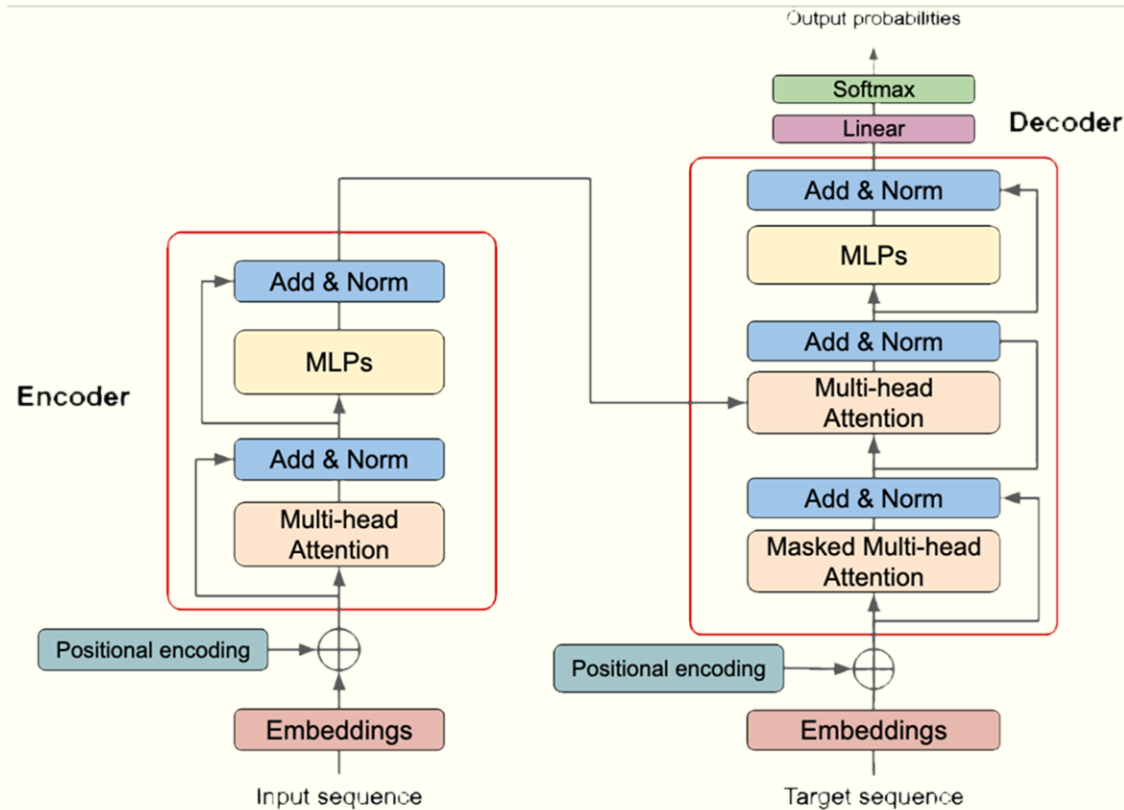
- Core building blocks of LLMs.
- Consist of self-attention layers and feed-forward layers.
- Handle long-range dependencies in text and capture semantic relationships.

- **Dense Layers:**

- Fully connected layers often used in LLM output stages.
- Combine information from previous layers to generate final predictions or classifications.



Quantization & LLMs: Understanding LLM Internals:

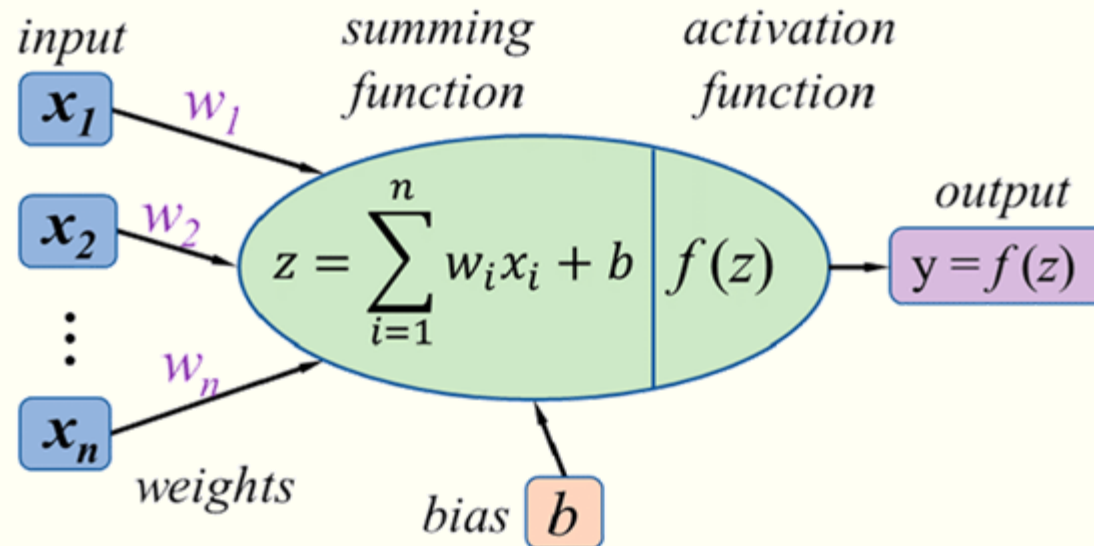


Each layer plays a crucial role in LLM architecture.

- **Embedding layers** represent words as vectors, enabling the model to grasp semantic relationships
- **Transformer blocks** excel at understanding long-range dependencies and semantic relationships.
- **Dense layers** integrate information for final predictions or classifications.

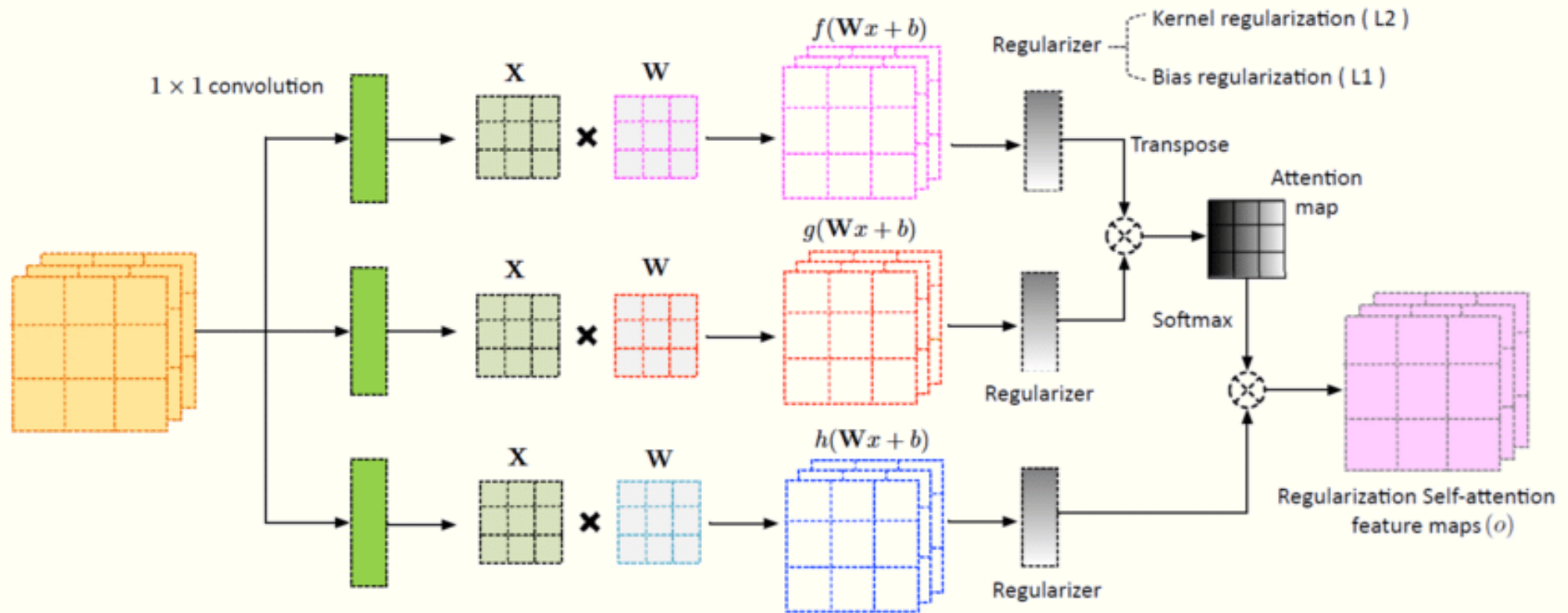
Quantization & LLMs: Understanding LLM Internals:

- All three layers - Transformer Blocks, Dense Layers, and Embedding Layers - heavily rely on ***weights and biases*** for their functionality and learning
- **Weights & Biases:**
 - Numerical values that determine the strength of connections between neurons, representing knowledge learned during training.
 - Numerical values that adjust overall activation levels, allowing for fine-tuning of model behavior.



Quantization & LLMs: Understanding LLM Internals:

Transformer Blocks: + Dense Layers \rightarrow Weights: & Biases \rightarrow Matrix Operations \rightarrow Store Numeric Values



Quantization & LLMs

Understanding **weights and biases** is crucial for **quantization** because:

- Quantization primarily targets these *numerical values* for reduction in precision.
- By strategically *reducing the number of bits* used to represent *weights and biases*, model size and computational costs can be significantly reduced.
- The challenge lies in finding the *optimal quantization levels* that maintain model *accuracy* while achieving desired *efficiency* gains.

Quantization & LLMs

Quantization's Aim:

- Reduce model size and computational complexity while minimizing performance impact.
- Achieved by reducing the ***precision of weights and activations*** in model layers.

Quantization & LLMs: Quantization Steps

1. Mapping to a Smaller Representation:

1. Rescale weights and activations from their original range (e.g., 32-bit floating-point) to a lower-precision format (e.g., 8-bit integers).
2. Example: Map a weight value of **0.456789** to the nearest **8-bit integer** value (0-255), potentially becoming **115**.

2. Calibration:

1. Find optimal mapping ranges to minimize information loss during quantization.
2. Crucial for preserving model accuracy.

Quantization & LLMs: Quantization Steps

4.Embedding Quantization:

1. Apply quantization to **word embeddings**, the numerical representations of words.
2. Reduces memory footprint and potentially improves efficiency of attention computations.

5.Transformer Block Quantization:

1. Quantize **weights and activations** within self-attention and feed-forward layers.
2. Techniques like mixed-precision quantization can preserve accuracy while reducing computational costs.

6.Dense Layer Quantization:

1. Apply quantization to **weights and activations** in dense layers.
2. 8-bit quantization often works well for dense layers without significant accuracy degradation.

Quantization & LLMs

An example for Dense Layer Quantization:

- Consider a hypothetical **Dense Layer** with:
 - Input: **1000** neurons
 - Output: **500** neurons
- **Weights:** 500,000 ($1000 * 500$) **floating-point** numbers, each typically occupying **32 bits (4 bytes)** of memory.
- **Biases:** 500 floating-point numbers, also using **32 bits each**.
- **NOTE: Each weight and bias** are represented with **4 bytes!**

Quantization & LLMs

Example 1 for Dense Layer Quantization:

- Quantizing to **8-bit integers**:
 - 1.Mapping:** Each weight and bias value is mapped to an **8-bit integer** (0-255), reducing memory usage per value to **1 byte**.
 - 2.Calibration:** An algorithm determines optimal mapping **ranges** to minimize accuracy loss.
 - 3.Storage:** Weights and biases now occupy only 500KB (500,000 * 1 byte) and 500 bytes, respectively, a **4x reduction** from 2MB (4 bytes per value).
 - 4.Computation:** Inference calculations use optimized **integer arithmetic** for 8-bit values, often faster than **floating-point** operations on modern hardware.

Quantization & LLMs

Example 2 for Dense Layer Quantization:

- Assume a hypothetical dense layer with **4** input neurons and **3** output neurons:

Original **32-bit** Floating-Point Weight Matrix:

```
[[0.54321, -0.23456, 0.98765],  
 [0.12345, 0.67890, -0.34567],  
 [-0.87654, 0.45678, 0.01234],  
 [0.56789, -0.90123, 0.23456]]
```

Quantization & LLMs

Example 2 Cont.:

- Quantizing to **8-bit Integers** with a range of 0 to 255:

1. Determine *Range and Scaling Factor*:

A. Find the minimum and maximum values in the matrix:

Min = -0.90123

Max = 0.98765

B. Calculate the scaling factor to map values to 0-255:

Scaling Factor = $255 / (\text{Max} - \text{Min}) = 255 / (0.98765 + 0.90123) \approx$
135.00063

Quantization & LLMs

Example 2 Cont.:

2. **Center Values:** Subtract the minimum value of the matrix from all elements before scaling.

[[0.54321, -0.23456, 0.98765],
[0.12345, 0.67890, -0.34567],
[-0.87654, 0.45678, 0.01234],
[0.56789, -0.90123, 0.23456]]



[[1.44444, 0.66667, 1.88888],
[1.02468, 1.58013, 0.55556],
[0.02469, 1.35801, 0.91357],
[1.46912, 0.00000, 1.13579]]

Quantization & LLMs

Example 2 Cont.:

3. Map Values to 8-bit Integers:

- **Multiply** each value by the **scaling** factor

[[1.44444, 0.66667, 1.88888],
[1.02468, 1.58013, 0.55556],
[0.02469, 1.35801, 0.91357],
[1.46912, 0.00000, 1.13579]]



[[194.9998, 89.9999, 254.9998],
[138.2918, 213.3176, 75.00003],
[3.3333, 183.3314, 123.3332],
[198.3316, 0.0000, 153.3331]]

Quantization & LLMs

Example 2 Cont.:

3. Map Values to 8-bit Integers:

- round to the nearest integer:

[[194.9998, 89.9999, 254.9998],
[138.2918, 213.3176, 75.0003],
[3.3333, 183.3314, 123.3332],
[198.3316, 0.0000, 153.3331]]



[[195, 90, 255],
[138, 213, 75],
[3, 183, 123],
[198, 0, 153]]

Quantization & LLMs

Original **32-bit** Floating-Point
Weight Matrix

[[0.54321, -0.23456, 0.98765],
[0.12345, 0.67890, -0.34567],
[-0.87654, 0.45678, 0.01234],
[0.56789, -0.90123, 0.23456]]



Quantized **8-bit** Integer
Weight Matrix

[[195, 90, 255],
[138, 213, 75],
[3, 183, 123],
[198, 0, 153]]

Quantization & LLMs

Benefits of Quantization:

- **Reduced Model Size:** Up to 4x smaller, enabling deployment on devices with limited memory.
- **Acceleration:** Potential speedup due to efficient integer arithmetic
- **Faster Inference:** Up to 2-3x faster processing, leading to quicker responses.
- **Lower Energy Consumption:** Reduced computational demands, making LLMs more energy-efficient.

Quantization & LLMs

Considerations and Challenges:

- **Accuracy Trade-offs:** Quantization can introduce slight ***accuracy*** loss. Careful ***calibration*** and technique selection are crucial.
- **Hardware Compatibility:** Ensure hardware supports efficient low-precision operations for optimal benefits.

An Overview of Quantization Methods

- **Post-Training Quantization (PTQ):**

- Applies after model training.
- Reduces precision of model weights, often from 32-bit to 8-bit, saving memory and speeding up inference.
- Examples: PTQ, 4-bit NormalFloat (NF4), QLoMP.

- **Quantization-Aware Training (QAT):**

- Incorporates quantization during training.
- Helps the model adapt to lower precision for better accuracy.
- Examples: QAT, LLM-QAT, QiloRA.

- **Efficient Fine-Tuning with Quantization:**

- Combines quantization with efficient fine-tuning techniques for faster adaptation and memory savings.
- Examples: LoRA, EfficientDM.

- **Other Quantization Techniques:**

- Use different strategies to reduce model size.
- Examples: Weight Clustering, OdysseyLLM, Eager Mode Quantization, FX Graph Mode Quantization.

Quantization & LLMs

LLM Name	Original VRAM (GB)	Quantized VRAM (Q-4bit) (GB)	VRAM Reduction (%)
GPT-3	140	35	75%
Jurassic-1 Jumbo	150	40	73%
Bloom	150	35	75%
Megatron-Turing NLG	300	75	75%
PaLM	320	80	75%
Chinchilla	75	20	73%
Falcon-40B	90	45	50%
Falcon-180B	450	95	78%

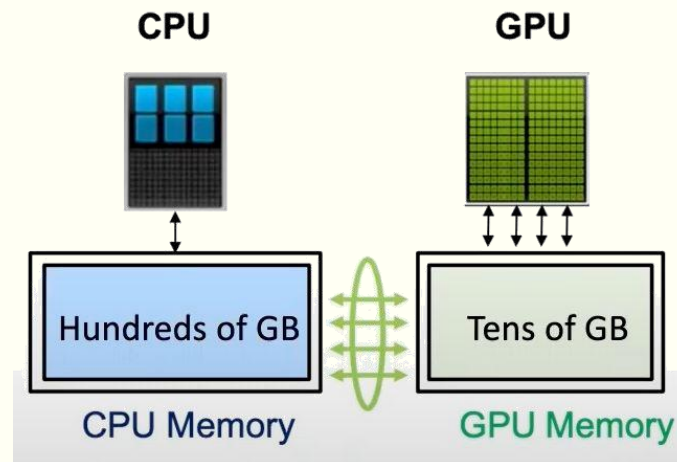
Content

- The Need for Optimization: LLMs are huge!
- How to zip LLMs
- Quantization: Packing More Value in Less Space
- **Off-loading: Sharing the Burden**
- Demo: Run Mixtral-8x7B Model on Google Colab's Free Version



Understanding Off-loading

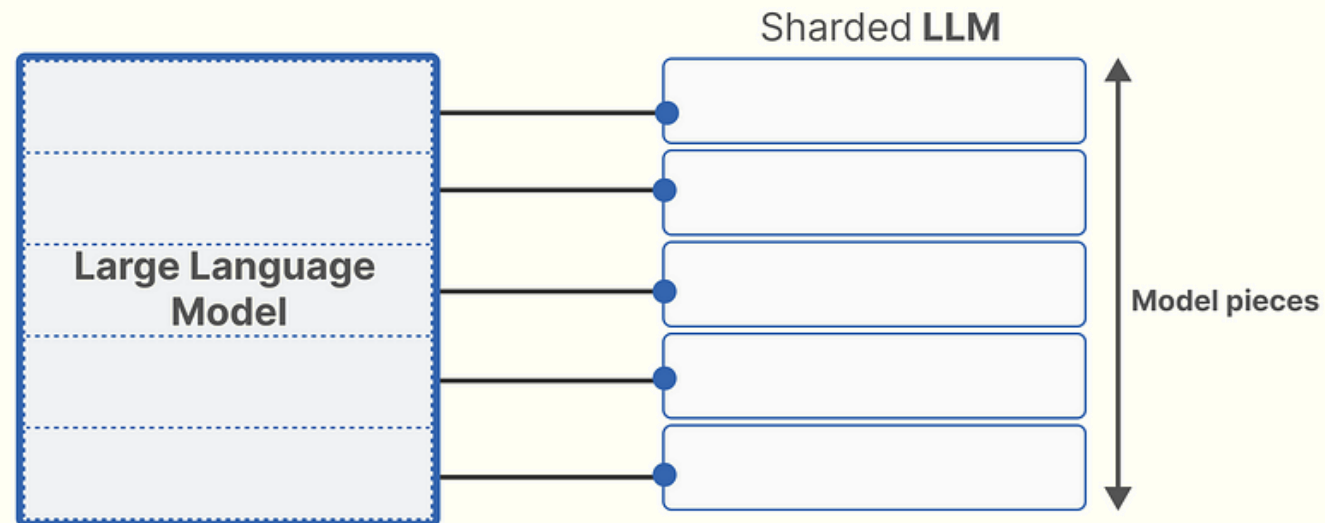
- Off-loading is a technique used to manage the **computational and memory** demands of large models like LLMs.
- It involves **moving** some of the computational tasks **from the GPU** (Graphics Processing Unit) **to the CPU** (Central Processing Unit), thereby freeing up GPU resources.
- **Example:** Consider a scenario where you're trying to run a large language model (LLM) like **Mixtral-7B** on a standard GPU. The model's parameters size exceeds the GPU's memory capacity. To manage this, we can use off-loading.



How does Off-loading work in LLMs?

Model Sharding:

- The parameters of the LLM are divided into smaller parts or “**shards**”. Each shard is small enough to fit into the GPU memory.
- This is similar to how large files are broken down into smaller parts for easier management and processing.

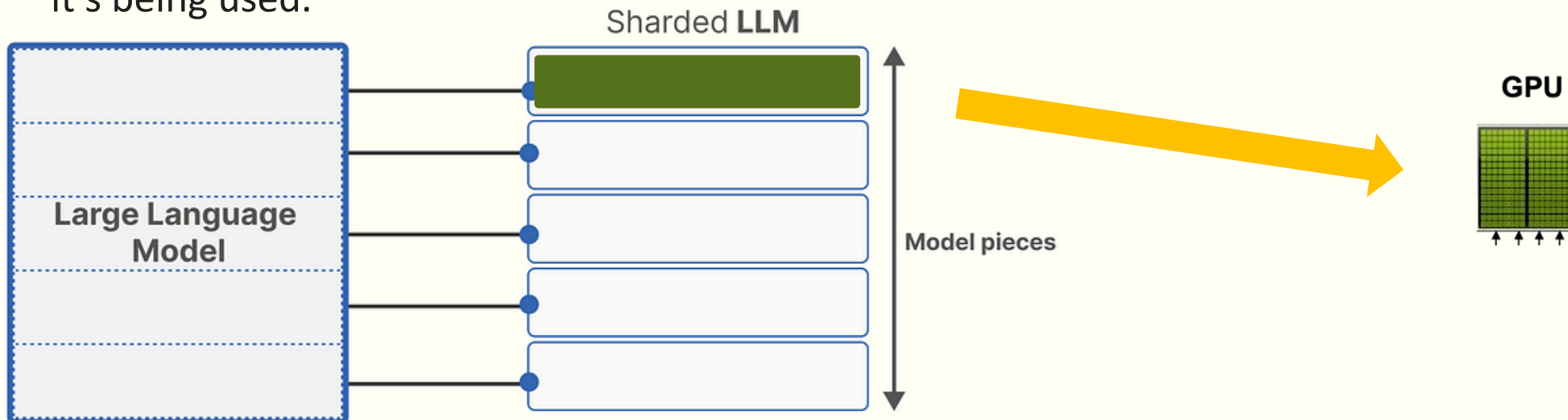


How does Off-loading work in LLMs?

Model Sharding:

Loading Shards:

- When the model needs to perform computations involving a particular set of parameters, the corresponding **shard** is **loaded** into the GPU memory.
- This is similar to how an application loads only the necessary parts of a file into memory when it's being used.



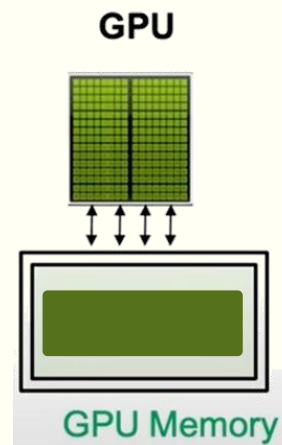
How does Off-loading work in LLMs?

Model Sharding:

Loading Shards:

Computation:

- The GPU performs the necessary computations using the ***loaded*** parameters.



How does Off-loading work in LLMs?

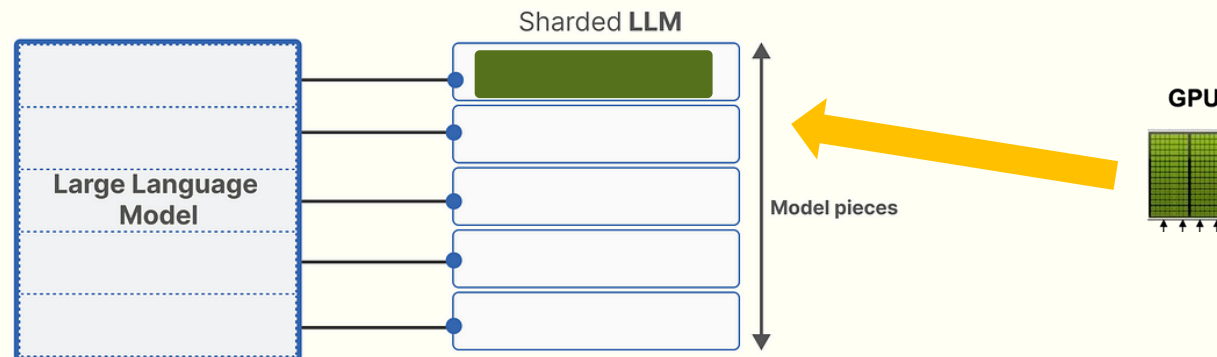
Model Sharding:

Loading Shards:

Computation:

Off-loading:

- Once the computations for a particular shard are completed, the shard is ***off-loaded*** from the GPU memory, making room for the next shard.
- This is similar to how an application frees up memory once it's done processing a part of a file.



How does Off-loading work in LLMs?

Model Sharding

Loading Shards

Computation

Off-loading



This process is **repeated** until all necessary computations for the entire model have been completed.

Why is Off-loading Important?

- The off-loading technique allows for the use of LLMs even on hardware with *limited GPU memory*.
- It also enables *faster loading*, usage, and *fine-tuning* of LLMs.
- By using off-loading, researchers and developers can run large models on standard hardware, making these powerful models more accessible.
- Furthermore, off-loading can improve the efficiency of applications by ensuring that resources are used optimally.

Off-loading Libraries

DeepSpeed: Developed by Microsoft Research, it's a comprehensive library for optimizing large model training and inference.

- Offers various off-loading techniques, including: Zero-Redundancy Optimizer (ZeRO) for memory optimization, DeepSpeed Sparse Attention for efficient handling of sparse models, Pipeline parallelism for distributing model across multiple GPUs
- It's highly optimized for PyTorch and is used in training large models like Megatron-Turing NLG.

Megatron-LM: An open-source library from NVIDIA specifically designed for training LLMs with billions of parameters.

- Provides model parallelism and pipeline parallelism for efficient distribution across multiple GPUs and nodes.
- Integrates with DeepSpeed for additional optimization features.

Off-loading Libraries

FairScale: A library from Facebook AI Research for scaling deep learning models.

- Focuses on simplifying model parallelism and pipeline parallelism for PyTorch models.
- Offers features for memory-efficient model training and inference, including off-loading techniques.

Hugging Face Transformers:

- While primarily a library for working with LLMs, it has built-in support for ***DeepSpeed*** and ***FairScale*** for off-loading.
- This allows you to easily leverage off-loading techniques within the familiar Transformers framework.

Off-loading Libraries

Mixture-of-Experts (MoE) Language Models with Offloading:

This approach uses sparse Mixture-of-Experts (MoE) - a type of model architecture where only a fraction of model layers are active for any given input.

This property allows MoE-based language models to generate tokens faster than their dense counterparts, but it also increases model size due to having multiple experts.

<https://arxiv.org/abs/2312.17238>

arXiv > cs > arXiv:2312.17238

Search...
Help | Advanced Search

Computer Science > Machine Learning

[Submitted on 28 Dec 2023]


Fast Inference of Mixture-of-Experts Language Models with Offloading

Artyom Eliseev, Denis Mazur

With the widespread adoption of Large Language Models (LLMs), many deep learning practitioners are looking for strategies of running these models more efficiently. One such strategy is to use sparse Mixture-of-Experts (MoE) - a type of model architectures where only a fraction of model layers are active for any given input. This property allows MoE-based language models to generate tokens faster than their dense counterparts, but it also increases model size due to having multiple experts. Unfortunately, this makes state-of-the-art MoE language models difficult to run without high-end GPUs. In this work, we study the problem of running large MoE language models on consumer hardware with limited accelerator memory. We build upon parameter offloading algorithms and propose a novel strategy that accelerates offloading by taking advantage of innate properties of MoE LLMs. Using this strategy, we build can run Mixtral-8x7B with mixed quantization on desktop hardware and free-tier Google Colab instances.

Comments: Technical report

Subjects: **Machine Learning (cs.LG)**; Artificial Intelligence (cs.AI); Distributed, Parallel, and Cluster Computing (cs.DC)

Cite as: [arXiv:2312.17238](https://arxiv.org/abs/2312.17238) [cs.LG]
(or [arXiv:2312.17238v1](https://arxiv.org/abs/2312.17238v1) [cs.LG] for this version)
<https://doi.org/10.48550/arXiv.2312.17238> 

Submission history
From: Denis Mazur [[view email](#)]
[v1] **Thu, 28 Dec 2023** 18:58:13 UTC (267 KB)

Off-loading Libraries

Mixture-of-Experts (MoE) Language Models with Offloading:

<https://arxiv.org/abs/2312.17238>

arXiv > cs > arXiv:2312.17238

Search... Help | Advanced Search

Computer Science > Machine Learning

[Submitted on 28 Dec 2023]

Fast Inference of Mixture-of-Experts Language Models with Offloading

Artyom Eliseev, Denis Mazur

With the widespread adoption of Large Language Models (LLMs), many deep learning practitioners are looking for strategies of running these models more efficiently. One such strategy is to use sparse Mixture-of-Experts (MoE) - a type of model architectures where only a fraction of model layers are active for any given input. This property allows MoE-based language models to generate tokens faster than their dense counterparts, but it also increases model size due to having multiple experts. Unfortunately, this makes state-of-the-art MoE language models difficult to run without high-end GPUs. In this work, we study the problem of running large MoE language models on consumer hardware with limited accelerator memory. We build upon parameter offloading algorithms and propose a novel strategy that accelerates offloading by taking advantage of innate properties of MoE LLMs. Using this strategy, we build can run Mixtral-8x7B with mixed quantization on desktop hardware and free-tier Google Colab instances.

Comments: Technical report

Subjects: **Machine Learning (cs.LG)**; Artificial Intelligence (cs.AI); Distributed, Parallel, and Cluster Computing (cs.DC)



Cite as: arXiv:2312.17238 [cs.LG]
(or arXiv:2312.17238v1 [cs.LG] for this version)
<https://doi.org/10.48550/arXiv.2312.17238>

Submission history
From: Denis Mazur [view email]
[v1] Thu, 28 Dec 2023 18:58:13 UTC (267 KB)



Hugging Face

Search models, datasets, users...

mistralai/**Mixtral-8x7B-Instruct-v0.1**   like 1.85k



Text Generation



Transformers



Safetensors



5 languages

mixtral



text-g



Model card



Files and versions



Community 86



Min hardware requirements #3

by narvind2003 - opened 29 days ago



Discussion



narvind2003 29 days ago

Could you please add the **minimum hardware requirements** to run this Instruct model?






10



7



kev216 29 days ago • edited 29 days ago

I have tried using a **4090 24G**, but it didn't work...    we need more RAM.



0-hero 29 days ago • edited 29 days ago

">130GB required"



2



Content

- The Need for Optimization: LLMs are huge!
- How to zip LLMs
- Quantization: Packing More Value in Less Space
- Off-loading: Sharing the Burden
- **Demo: Run Mixtral-8x7B Model on Google Colab's Free Version**

