Quantizing Model Parameters for OCR Models

Ethan Urmson and Amanda Jen

Abstract

Our goal was to produce a quantized VLM (Vision Language model) that retains accuracy and has a reduction in size. We used a hugging face model, Qwen2-VL-2B [1] that had 2 billion parameters and was BF16. We attempted an 8-bit and 2-bit PTQ (post training quantization) of the model. The results are mixed. Our 8-bit quantized model had a 69.5% accuracy, the same as the original Qwen2-VL-2B model. And when comparing, our 8-bit model to a 8-bit quantized model using BitsAndBytes both had a similar performance and slight difference in size. On the other hand, multiple 2-bit models were created, but the model size or accuracy were not optimal. We further explore our results in later sections.

1 Introduction

Optical character recognition (OCR) has been an important usage of machine learning especially on lower computation edge devices. With the dawn of LLMs (Large language models) and VLMs there are several of these VLMs that are fine-tuned to do OCR tasks. Using VLMs has benefits in more flexibility in the formatting of the output, but the downside of much larger model sizes and computational requirements. There has been research into quantizing models to reduce the overall size of the model in memory, and stored on the disk. Accuracy is known to be negatively affected by bit length reductions [2]. We chose to implement quantized models in pytorch and Triton, as both are python compatible and are useful for quantization. Triton is especially useful for implementing custom kernels.

2 Method

While we aimed to quantize the model ourselves we also wanted to see how it would compare to other quantization libraries available. For our project, we used BitsAndBytes, a popular python based library. Using BitsAndBytes we reduced the model to 4-bit and 8-bit versions. During the evaluation we tested our baseline model, the BitsAndBytes models, and our own implementations on handwriting recognition. Our original dataset size was 40,000 images but we reduced it to a smaller set of 1,000 photos due to time and GPU constraints. We found that running 40,000 vs. 1,000 photos didn’t make a large difference in accuracy. There was only a slight deviation if any which was within a 1-2% range.

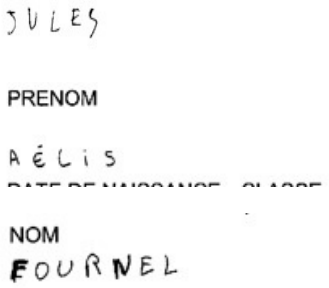

Figure 1: Samples from the handwriting dataset [4]

We recorded the model size on the GPU before and after quantization in addition to two tests to assess the OCR task. Our first one was exact character matching. If the output name matched the correctly labeled name then it would add to the correctness count and it would be averaged at the end. The second method of assessment was the Levenshtein distance test. The Levenshtein distance is the number of operations (like insertions and deletions) needed to make one string into another. We added up the Levenshtein distances and averaged it across the dataset. For our custom quantizations, we tried to utilize PyTorch and Triton in hopes of reducing model size while maintaining accuracy. For this experiment, we used multiple devices. A Windows PC with a virtual desktop as well as two laptops. In the beginning, a Mac laptop ran BitsAndBytes and had a M1 chip with 8.0GB, but after the initial setup it was not used. The second device was primarily used because it was compatible with CUDA. Its OS was Windows and the hardware was a NVIDIA GeForce RTX 3050 Laptop GPU with only 4.0GB of memory. The PC has a Nvidia GeForce RTX 4080 Super with 16GB of GPU memory, allowing for reduced runtimes and increased batch sizes. We attempted to quantize the model into 8-bit and 2-bit versions. The sections below go into more detail.

2.1 8-bit Quantization

For our 8-bit quantization, we tried two approaches. One was with PyTorch and the other used Triton. The PyTorch and Triton methods had the same underlying quantization scheme, but the biggest difference being the Triton custom kernel and matrix operations, while the PyTorch version used the standard matmul. To lower precision from BF16 to INT8 we used a basic quantization technique: linear symmetric quantization. A linear symmetric quantization maps a floating point value to a symmetric integer range centered around zero [5]. Because it is symmetric our Z (zero-point, aligns the mapping of numbers), can be set to 0. We scaled x (given weight tensor) by S, the scale factor that converts FP16 to INT8. This is quantization formula:

$$q = round(x/S + Z)$$

In this case, we chose to map it to range -127 to 127, a common range for 8-bit. While we do lose -128 in the signed 8-int range, it is known to reduce the number of addition operations, making it faster [5]. We used a row per channel quantization. Which is where an input tensor is quantized separately for each individual channel [5]. Compared to per tensor quantizations, per channel tends to be more accurate, but with some increase in memory [5]. Our general steps are as follows:

- Extract the FP16 weight matrix
- Compute per-channel scales

- Quantize to INT8 weights, while ignoring sensitive layers like lm_head, vision modules, activations, and embeddings

Many quantization libraries have a parameter where you can exclude specific modules. BitsAndBytes version is called 'llm_int8_skip_modules'. We implemented something similar using a prefix to match with any module names that we didn't want to quantize. Our prefix matching was based on Govindaraj's [6] basic approach. After quantization, the model performed forward passes by dequantizing our weights back to FP16 to be used in a basic matrix multiplication. With this approach, we were able to see a reduced model size expected of an 8-bit model. Albeit, it was 21.6% higher in size (GB) than our BitsAndBytes 8-bit version. This was calculated using values from Table 1:

$$\frac{2.967-2.44}{2.44} \times 100 = 21.6\%$$

Furthermore, we experimented with Triton to try to improve the speed of inference as well as model size. Ultimately, this was unsuccessful. When run it produced nearly identical results to the PyTorch model. The only different evaluation metric was a levenshtein distance value of 0.637 for PyTorch and 0.636 for Triton. Poor utilization of Triton is probably the reason for these results. The GEMM kernel is too general for any substantial performance increases. It lacked packing, tensor core use, and other additions that could have optimized it further. With more time and a better understanding, we could properly utilize the hardware to improve the accuracy and reduce the model size closer to top used quantization libraries like BitsAndBytes, AWQ, GPTQ, etc.

2.2 2-bit Quantization

Based on some prior research into low bit width quantization by S. Bhatnagar et al.[3], we thought that it should be possible to get a 2-bit quantization of the weights working. We attempted this similarly to the 8-bit quantization, with a symmetric packed representation of the weights, with a scale factor for each block of weights. This allowed for tuning the group size for the amount of precision loss that is acceptable for the use case of the model. To perform calculations with the quantized model, we initially tried to use PyTorch to unpack the values then do a normal matrix multiply with the unpacked weights. This turned out to be extremely slow, as PyTorch would move the packed weights to the CPU to perform the unpacking step, then back to the GPU to do the matrix multiply, or continue doing the matrix on the CPU. This led to the necessity of a custom kernel to complete the unpacking in a reasonable amount of time. The triton kernel we tried to implement was fusing the unpacking and matrix multiply together. Initially, when trying to quantize all of the layers of the model, the model either crashed from invalid values on the GPU, or output gibberish. This led to aggressively limiting the layers that the model is allowed to quantize, which ended up removing most of the benefit of quantizing the model at all. Through experimentation, we found that the 'kqv' and 'proj' layers were able to be quantized while still providing reasonable outputs from the model, we also found that using any 'proj'

layers were still able to have the model output somewhat coherent if not correct results. It was also clear that increasing the group size also had a negative effect on the performance of the model, and quickly prevented the model from completing the task successfully. With more time we would be able to find a better selection of layers to quantize, as well as combining the 2-bit quantization with some less aggressive techniques, like BitsAndByte's 4 bit quantization.

3 Performance

We measured the size of the model as reported by get_memory_footprint method, which is built in for PyTorch models. This reports the size in bytes of the model on the GPU. Using this, we measured the size of the base model to be about 4.4GB. This allowed us to run multiple instances of inference off of the base model at the same time using the same weights. The size of the model decreased as we were expecting when using the BitsAndBytes library, with a 44.67% decrease in size using 8bit quantization and a 67.12% decrease in size using the 4 byte quantization. These were slightly less than the expected 50% and 75% decreases that would be the optimal decrease for going from the 16-bit original format of the model to the 8 and 4 bit versions. Our implementation of the 8-bit quantization was also able to greatly reduce the size of the model by 32.72% which was somewhat less than the bits and bytes version, but was still a significant reduction in the model size. And finally our attempts at 2-bit quantization were the least successful, only managing to reduce the model size between 1.59% and 7.48% while having the model remain even a little bit coherent.

Models	Size(GB)	Reduction
Qwen2-VL-2B	4.41	——
Bits&Bytes 4-bit	1.45	67.12%
Bits&Bytes 8-bit	2.44	44.67%
8-bit	2.967	32.72%
2-bit - 16 Group size - kqv + all proj	4.11	6.80%
2-bit - 8 Group size - qkv + all proj	4.08	7.48%
2-bit - 16 group size - qkv + simple proj	4.33	1.81%
2-bit - 8 group size - qkv + simple proj	4.34	1.59%

Table 1: Size of the model on the GPU in GB, and the reduction percentage in size compared to the Qwen2-VL-2B model

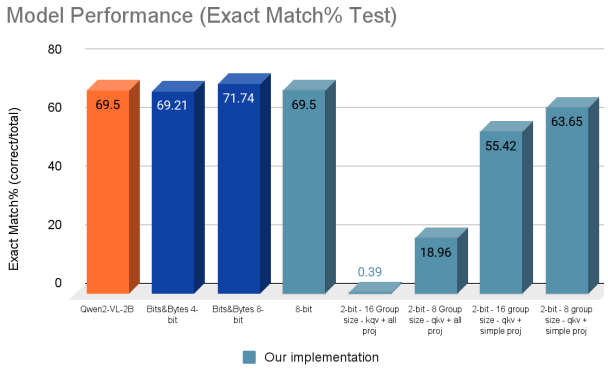

Figure 2: Exact Match Evaluation

Moving on to the accuracy of each of these models through the varying amounts of quantization. All of our testing was done using the same prompt, "The image contains a name, output the name, no spaces, and no other text. DO NOT output NOM or PRENOM or any codes", and each image was processed with the prompt to generate the output of the model. Our initial approach for measuring the model accuracy was to look for an exact match on the output of the model with the dataset labels, after

some light post processing (Stripping trailing and leading white space, and converting to capital letters). This gives the model very little leeway for mistakes and ends up with the base unquantized model getting the exact match with the label for the data ~70 percent of the time (figure 2). This did not change much with the “off the shelf” quantizations, and even our own 8-bit quantization had a very similar performance. We found it surprising as we had thought that there would be a larger drop off as the precision decreased. Moving on to the 2-bit quantizations there was an extreme drop off in the accuracy of the quantized model.

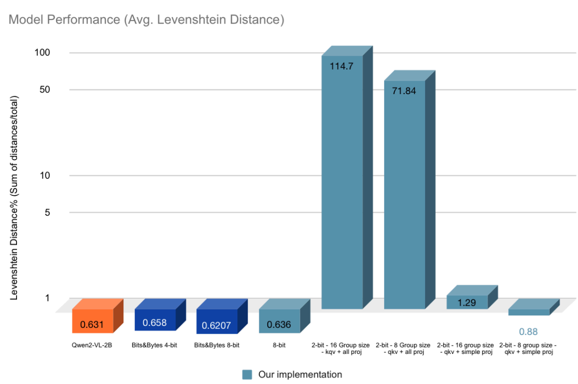

Figure 3: Levenshtein Distance Test

After testing with the exact match metric, we wanted to get a somewhat finer grain metric for how well the models were actually performing. Maybe even though the models were getting the right answer a similar percent of the time, some of them might be “closer” when they are wrong. We decided to try measuring the average Levenshtein distance between the expected output, and what the model output, again with the same simple postprocessing. This measures how many insertions, deletions, and replacements are needed to get from the expected output to the model output. Looking at the original model and the 8 and 4 bit quantizations of it, we once again see that there is not a large difference between any of them, with all of them on average being less than one character away from the correct answer. We can also see that when moving to the 2-bit quantizations, the less accurate ones increased their output size dramatically compared to the base model with distances near 100. The model as it lost accuracy often “forgot” what the instructions were, created excuses for not performing the task, or was unable to produce an end token and would repeat the name several times in the output, or sometimes just produced straight up gibberish. Table 2 includes some of this faulty output that led to a high Levenshtein distance score.

Expected Value	Model Output
DELCOURT	DELCOURT DELCOURT … … DELCOURT
AVRILLIER	A. THE IMAGE IS BLURRY, AND THE TEXT IS NOT CLEAR. THE TEXT IS NOT CLEAR, AS IT IS THE FIRST TO THE RIGHT. DISSERTATION: A. THE IMAGE IS BLURRY, AND THE TEXT IS NOT CLEAR. THE ANSWER IS AS FOLLOWS: THE ANSWER IS AS FOLLOWS: THE IMAGE IS NOT CLEAR
BENIZA	A. B. C. D. E. F. G. H. I. J. K. L. M. N. O. M. N. O. M. N. O. M. N. O. M. N. O. M. N. O. M. N
FROMENT	电路原理电路原理电路原理电路原理电路原理电路原理电路原理电路

Table 2: Sample responses from 2-bit quantization of different failure modes with group size of 16 and all proj layers quantized.

4 Conclusion

Our custom 8-bit model performed well when compared to the baseline and BytesAndBits version. While the reduced size was 2.9GB compared to 2.44GB (BitsAndBytes), we proved that we were able to quantize several layers while retaining accuracy and avoiding sensitive layers and activations. Using a similar strategy we could also probably do a 4-bit quantization. Our 2-bit models had some issues but with further improvements on the layers we selected for quantization and using a less aggressive quantization for some of the other layers, using more time we believe it is possible to do. We were able to affirm that quantizations of VLMs are possible and can run on lower end devices.

References

[1] P. Wang et al., "QWEN2-VL: Enhancing Vision-language model's perception of the world at any resolution," arXiv.org, ttps://arxiv.org/abs/2409.12191 (accessed Nov. 2025).

[2] J. Lang, Z. Guo, and S. Huang, "A Comprehensive Study on Quantization Techniques for Large Language Models," A comprehensive study on quantization techniques for large language models, https://arxiv.org/html/2411.02530v1#bib (accessed Nov. 2025).

[3] S. Bhatnagar, A. Xu, K.-H. Tan, and N. Ahuja, "LUQ: Layerwise Ultra-Low Bit Quantization for Multimodal Large Language Models," Luq: Layerwise Ultra-Low Bit Quantization for multimodal large language models, https://arxiv.org/html/2509.23729v2 (accessed Nov. 2025).

[4] Handwriting Recognition," Dataset, Kaggle, uploaded by landlord, 2020. (Online). Available: https://www.kaggle.com/datasets/landlord/handwriting-recognition. (accessed Nov. 2025)

[5] Quantization, https://huggingface.co/docs/optimum/en/concept_guides/quantization (accessed Nov. 2025).

[6] P. Govindaraj, "Build-an-Efficient-Quantizer-from-Scratch," GitHub, https://github.com/priyanthan07/Build-an-Efficient-Quantizer-from-Scratch?tab=MIT-1-ov-file (accessed Nov. 2025).