Samuel Jordan
Patrick Woods
Gabriel Hillesheim
Mr. Sayedpedram Haeri Boroujeni
Project 3: Lightweight / On-Device LLMs & VLMs
November 6, 2025

Tasks 2 & 3 Summary

Task 2

This task explores several stages of quantization and efficiency analysis in Transformer-based language models. Each section builds on the previous one, starting from simple per-tensor quantization concepts, then scaling up to weight quantization, full-model runs, and static KV-cache quantization.

1. Quantization Fundamentals (Weights & Activations)

1.1. Goal

The goal of this section was to understand what quantization does and to test it on a small example. We built a simple function in PyTorch that converts a layer's weights from floating-point (normal precision) to INT8 (8-bit integers) and checked how this affects accuracy.

1.2. Method & Setup

Libraries used:

- torch, torch.nn, and torch.nn.functional for model definition and tensor math.
- torchvision and torchvision.datasets (MNIST) for a small test dataset.

1.3. Implementation

We created two helper functions:

- quantize_int8_per_tensor(): converts a float tensor to 8-bit integers using a single scale factor across the tensor.
- dequantize_int8_per_tensor(): restores float values from their quantized form (granted, imperfectly).

Then we built a QuantLinear layer that automatically quantizes its weights once and quantizes activations during each forward pass.

Finally, we compared the outputs of a normal nn.Linear layer and our quantized version to see how much difference quantization makes.

1.4. Evaluation

We measured:

- Numerical error: Mean-squared difference between float and quantized layer outputs.
- Accuracy test: Model accuracy on MNIST before and after replacing the last layer with QuantLinear (trained for 2 epochs).

1.5. Results

- The INT8 version produced almost identical outputs compared to the float layer.
- MNIST accuracy remained nearly unchanged, showing minimal performance loss.
- This shows that even simple per-tensor quantization can work effectively for smaller models.

2. Run the Base Model (SmolLM-135M)

2.1. Goal

The goal was to load and run SmolLM-135M to get baseline performance numbers (mainly latency and GPU memory usage) before applying quantization.

2.2. Method & Setup

Libraries used:

- transformers (for AutoModelForCausalLM and AutoTokenizer).
- torch (for GPU handling and CUDA memory tracking).
- time (for latency measurement).

Setup Details:

- Model: "HuggingFaceTB/SmolLM-135M" from Hugging Face.
- Device: CUDA GPU if available; CPU otherwise.
- Prompts: Several short text prompts to test responsiveness.
    - *"Explain quantization in LLMs in one paragraph."*
    - *"Write a short description of Scooby-Doo and the Mystery Inc. gang."*
    - *"Explain the process of photosynthesis in one paragraph."*

2.3. Implementation

The function run_prompt(prompt) was used to:

1. Tokenize and send the prompt to GPU (or CPU if GPU is unavailable).
2. Measure latency using time.perf_counter().
3. Record maximum GPU memory via torch.cuda.max_memory_allocated().
4. Decode and print generated text output.

All prompts were tested individually, and the results were averaged.

2.4. Evaluation

We evaluated:

- Latency (s): time to generate responses.
- Peak memory (MB): GPU memory used per run.
- Output quality: informal check that the generated text made sense.

2.5. Results

- SmolLM-135M generated mostly relevant and readable responses but showed some factual inaccuracies across topics.

- On CPU, average latency was around 20 seconds for a 250-token generation on a typical desktop processor, and around 12 seconds on higher-end CPUs.
- Although slow, these CPU-based metrics provide a consistent, reproducible baseline for later quantized and GPU runs.

2.6. Challenges & Limitations

- Only a few prompts were used, which does not constitute a full benchmark. More varied prompts to text realms like coding, math, text revision, etc. would better evaluate the model across a wider range of tasks, as the current prompts mainly test explanation.
- Runs were done sequentially, one prompt at a time, with no batching.
- Executions were limited to what could be done on CPU due to lack of a suitable GPU. This resulted in 0MB GPU usage.
- No formal text quality metrics were recorded. Evaluations in this respect were purely qualitative.

3. Implement Weight Quantization (8-bit vs 4-bit)

3.1. Goal

To test how 8-bit and 4-bit quantization affect SmolLM-135M's memory usage, speed, and output quality. We used the bitsandbytes library to load quantized versions of the model and compared them with the FP16 baseline.

3.2. Method & Setup

Libraries used:

- transformers (for loading the model and tokenizer).
  - BitsAndBytesConfig (for 8-bit and 4-bit quantization).
- torch (for CUDA memory and timing).
- time and gc (for measuring latency and clearing memory).

Setup details:

- Three model configurations were tested:
  - FP16 baseline HuggingFaceTB/SmolLM-135M
  - 8-bit quantized (BitsAndBytesConfig(load_in_8bit=True))
  - 4-bit quantized (BitsAndBytesConfig(load_in_4bit=True, bnb_4bit_quant_type='nf4'))
- Each model used the same prompt ("What is AI?"), generation settings, and GPU/CPU.

3.3. Implementation

- Loaded three separate instances of SmolLM-135M:
  - FP16 baseline (or FP32 if running on CPU).
  - 8-bit quantized model configured with load_in_8bit=True.

    - 4-bit quantized model configured with load_in_4bit=True and bnb_4bit_quant_type="nf4".
- For each version:
    - The prompt was tokenized and passed to the model for text generation.
    - Latency was measured using time.perf_counter() around the generation call.
    - Peak memory usage was recorded using torch.cuda.max_memory_allocated() (reporting 0 MB on CPU).
- After each run, the latency and memory statistics were printed for comparison across the three configurations.
- No qualitative or accuracy evaluation of generated text was included in this section.

3.4. Evaluation

Each configuration was compared based on:

- Latency: The time required to generate text for the same prompt.
- Memory usage: The peak amount of GPU memory allocated during generation (or 0 MB for CPU).

All other factors such as prompt, model parameters, and decoding settings were kept identical to ensure a fair comparison.

3.5. Results

- The base model and quantization were successfully implemented and tested.
- In this specific run, the quantized models did not reduce GPU memory usage or improve latency. The 8-bit and 4-bit models were shown to have a minor increase in memory usage, and a major increase in latency, with a 3.5x and ~1.5x increase respectively.

3.6. Challenges & Limitations

- Quantization results were limited by the small model size and FP16-capable GPU; thus, no memory or time savings were observed.
- No quality or accuracy metrics were recorded in this section.
- Results reflect post-training quantization only, with no fine-tuning or calibration.

4. Implement Static KV-Cache Quantization

4.1. Goal

The goal of this section was to implement and test static KV-cache quantization using 4-bit precision, measuring how it affects inference latency, memory usage, and KV-cache size.
The purpose was to demonstrate how quantizing the key-value cache can reduce memory requirements during long text generation without retraining the model.

4.2. Method & Setup

Libraries used:

- transformers (for loading and generating with the SmolLM-135M model).

- optimum-quanto (to enable 4-bit KV-cache quantization).
- torch (for CUDA operations and memory measurement).
- time, math, os, psutil (for timing, basic math operations, system access, and memory/process information).

4.3. Implementation

- Verified that Optimum Quanto was available using is_optimum_quanto_available(), and enforced GPU availability.
- Defined the helper function runOnce(prompt, max_new_tokens, kv_quant: bool), which:
    1. Tokenized the prompt with the model's tokenizer and moved tensors to the GPU.
    2. Reset CUDA memory tracking (torch.cuda.reset_peak_memory_stats()).
    3. If kv_quant=True, applied a 4-bit KV-cache quantization configuration.
    4. Ran text generation with model.generate() and measured elapsed time with time.perf_counter().
    5. Recorded latency, peak GPU memory, and the KV-cache difference.
- The function was executed for both short and long prompts, with and without quantization.
    - Short prompt: Constructed from a base paragraph string repeated enough times to simulate a brief context (approximately 128 tokens). The short prompt ended with: "Explain quantization in LLMs in one paragraph."
    - Long prompt: Built by repeating the same base paragraph many times to create a much longer input (roughly 512 tokens), followed by: "Continue with a concise summary"
- Each prompt was run twice (once without quantization and once with 4-bit quantization enabled) resulting in four total measurements:
    - Short prompt (baseline)
    - Short prompt (quantized)
    - Long prompt (baseline)
    - Long prompt (quantized)

4.4. Evaluation

The evaluation compared performance between baseline and 4-bit KV-quantized runs for both short and long prompts.

The following values were recorded for every run:

- Latency (s): total time taken for model.generate() to complete, measured with time.perf_counter()
- Peak Memory (MB): maximum GPU memory usage recorded via torch.cuda.max_memory_allocated() during text generation.

- KV Δ (MB): estimated size of the key–value (KV) cache, computed as the difference between the memory before and after generation.
- Text Preview: a truncated string (first 200 characters) of the model's generated output, confirming that generation executed successfully.

4.5. Results

The following table was generated:

Type	Latency (s)	KV Δ (MB)	Peak MB	KV Quant?
Short Base	7.050 s	305.7 MB	562.4 MB	False
Short KVQ	9.314 s	276.1 MB	540.8 MB	True
Long Base	26.754 s	296.2 MB	561.0 MB	False
Long KVQ	35.667 s	274.6 MB	539.4 MB	True

This shows that using 4-bit KV-cache quantization (INT4) reduced the KV-cache footprint by approximately 21.6 MB, corresponding to a 7.29% memory reduction on the long-prompt test. While the quantized runs were about 32% slower, the memory reduction may be more valuable.

4.6. Challenges & Limitations

- The code enforces a CUDA GPU requirement (assert torch.cuda.is_available()), so it cannot run on CPU-only systems.
- Only two prompt lengths of repeating phrases were tested, at 128 and 512 token lengths. Perhaps a more varied setup and more genuine prompts (ones that vary in content throughout the prompt rather than just meeting some set length) would provide different results.
- Again, this does not evaluate whether 4-bit quantization affected text quality, token accuracy, or coherence.

Task 3

This week’s work introduced an adaptive, saliency-aware quantization approach. We studied how token saliency and entropy can guide dynamic precision, implemented real-time feature extraction during generation, and built a rule-based KV-cache controller that adjusts bit-widths based on uncertainty. Finally, we compared static vs. dynamic KV policies to evaluate accuracy, latency, and memory trade-offs.

1. Study: Token Saliency and Dynamic Precision

1.1. Goal

Understand what saliency means in language models and how uncertainty (entropy) can be used to guide dynamic precision.

1.2. Study

Token saliency refers to how strongly individual tokens contribute to a language model's intermediate internal activations and ultimate output predictions. By identifying which tokens the model "pays attention to" the most, we can better understand which parts of an input drive the model's decision-making. Saliency can be computed through gradient-based attribution (as seen in Integrated Gradients and Layer-wise Relevance Propagation for BERT) or through attention-weight analysis (such as studies of GPT-2 and T5 that visualize which words influence the next token prediction). Tools like Captum and PyTorch and scoring methods like BERTScore use similar attribution principles to explain influence on the token level. Understanding token saliency is useful not only for interpretability but also efficiency, because it reveals where higher numerical precision would be useful and influential in computation.

Another related concept is **entropy**, which measures a model's uncertainty in predicting the next token. When a model's probability distribution is sharply peaked and it has high confidence, entropy is low; when the distribution is flat and there is more uncertainty, entropy is higher. Entropy-based uncertainty estimation has been widely used in Active Learning for BERT and Bayesian Transformers, as well as in adaptive methods like Dynamic Evaluation that adjust model behavior in response to uncertainty during inference. When combined with saliency, entropy offers a way to locate highly influential regions that a model is uncertain about; this provides an opportunity to dynamically allocate additional precision or computational resources.

This leads to the idea of **dynamic precision**: adapting numerical precision in real time based on model confidence and token importance, only allocating more resources where they are most needed. Models like DeeBERT and FastBERT apply this principle by allowing early exits for confident predictions, allocating deeper computation only when uncertainty is high. Similarly, ElasticBERT dynamically adjusts layer usage according to token difficulty. Frameworks such as PyTorch Dynamic Quantization implement runtime precision scaling based on your activations statistics. Extending these ideas to LLMs suggests that token saliency and entropy could be combined to drive adaptive quantization; preserving high precision for salient, high entropy tokens while lowering it everywhere else to balance computational efficiency and output quality.

2. Implement Saliency Feature Extraction

2.1. Goal

2.1.1. The Goal of Part 2 was to analyze and record how SmolLM-135M allocates importance across input tokens when generating a response, and its confidence level in producing said token. We achieved this goal through the extraction of the following features:

- **Token Saliency** - Which tokens does the model view as "important" when generating and predicting
- **Entropy** - How confident is the model in its prediction
- **Token Rarity** - How unique/surprising the token is
- **Attention** - Observe how attention head patterns evolved across the sequence.

2.2. Method & Setup

Libraries Used:

- Transformers (To load SmolLM-135M, tokenizer, and enable generation)
- Torch (Tensor computation, gradient backpropagation, and CPU/GPU execution.)
- Collections Counter (Computing rarity)

Setup Details:

- Tests were performed on SmolLM
- A device check was recycled to automatically choose
  - CUDA (FP16)
  - CPU (FP32)
- Logging was performed every 64 tokens with features printed.

2.3. Implementation

- Logits were set to FP64 to avoid instability
- Saliency was computed using the gradient from the predicted next token with respect to its input embeddings
- Entropy was calculated from the softmax probability distribution. Low entropy is an indicator of confidence, while high entropy is a signal of uncertainty.
- Token Rarity was approximated through computing -log prob estimating the probability of a token appearing in the text.
- The features were then logged every 64 tokens to observe how they evolved over the course of generation.

2.4. Evaluation

We ran multiple prompts and compared feature scoring.

- Baseline Narrative: “Tell me a story about a tortoise that wins a race against a hare.”
- Rarity tests included stress words like “Petrichor”
- Metrics were evaluated at tokens 64, 128, 192, and 256 to see changes over time.

2.5. Results

- Entropy was moderate (0.5 - 1.8)
- Saliency was reliable in highlighting prominent structure words in prompts. For Example, in our prompt “Tell me a story about a tortoise that wins a race against a hare” words such as “Tell”, “Story”, “Tortoise”, “Wins”, “Race”, “Hare” were highlighted in the saliency scoring.
- Rarity spiked on uncommon tokens when they were put randomly within a prompt. The word we chose was “petrichor”. When the word was input into the sentence the rarity spiked from 1.667 to 5.1402 @ token 64
- Attention variance existed, however remained small, most likely due to the small size of SmolLM

2.6. Challenges & Limitations

- The main issue we came across was a small model collapse, leading to repetition loops and affecting entropy and saliency values in the later iterations.

3. Build a Rule-Based Dynamic KV Policy

3.1. Goal

3.1.1. The Goal was to adjust the KV cache precision during inference based on how uncertain the model was.

3.2. Method & Setup

3.2.1. We used a simple statement that changes the bits based on entropy. Takes in the entropy and returns the adjusted KV bit width

3.3. Implementation

3.3.1. Using the function to map the KV cache, we computing entropy before each step, to dynamically select the bit width

3.4. Evaluation

3.4.1. Took a simple prompt- “The quick brown fox” and tracked the entropy, bit width, and average bits per token

3.5. Results

3.5.1. The system uses more bits for high uncertainty, and more certain uses less bits all while trying to stay under the average bit width

3.6. Challenges & Limitations

3.6.1. I think the biggest challenge was the conceptual implementation of this, we need to go more in depth for actual quantization results, but does a good job at summarizing the concepts.

4. Evaluate Static vs Dynamic Policies

4.1. Goal

4.1.1. Compare static vs dynamic policies of the accuracy, latency and memory

4.2. Method & Setup

4.2.1. Static 4-Bit kv (baseline)

4.2.2. Dynamic KV (rule based, entropy driven)

4.2.3. Dynamic KV and weight quantization (simulated)

4.2.4. Tested on 200 samples on hellaswag

4.3. Implementation`

4.3.1. Evaluation functions to measure the following

4.3.1.1. Accuracy- log-likelihood on multiple choice tasks

4.3.1.2. Latency- milliseconds per token generated

4.3.1.3. Memory- Peak GPU memory usage

4.4. Evaluation

4.4.1. Run all implementations on the same dataset to compare the performance

4.5. Results

4.5.1. Static 4-bit: 36.5% accuracy, 18.87 ms/token, 1163.9 MB

4.5.2. Dynamic KV: 36.5% accuracy, 20.09 ms/token, 1163.9 MB

4.5.3. Dynamic KV + Quant: 35.8% accuracy, 17.00 ms/token, 814.7 MB

4.5.3.1. The dynamic policy maintained accuracy with similar latency. Adding quantization reduced the memory, with a slight accuracy drop.

4.6. Challenges & Limitations

4.6.1. We didn't get to see large gains in dynamic policy. Once again most likely due to the smaller model being run on a high end gpu, if we used a larger model we could see much larger differences here.