Homework-3
11-664/763: Inference Algorithms for Language Modeling
Fall 2025

Your name (Andrew ID): Test Student (tests)

Instructors: Graham Neubig, Amanda Bertsch
Teaching Assistants: Clara Na, Vashisth Tiwari, Xinran Zhao

Due: November 25, 2025

Instructions

Please refer to the collaboration and AI use policy as specified in the course syllabus. Additionally, note that **no off-the-shelf inference servers can be used (e.g. vLLM, sglang, etc)**. Assume NVIDIA GPU hardware with CUDA throughout this assignment.

Shared Tasks

Throughout the semester, you will be working with data from three shared tasks. We host the data for each shared task on Hugging Face; you can access them at **[this link]**. We will generally ask for results on the "dev-test" split, which consists of 100 examples for each task, using the evaluation scripts provided. The remainder of the examples can be used for validation, tuning hyperparameters, or any other experimentation you would like to perform. The final shared task at the end of the semester will be evaluated on a hidden test set.

Algorithmic The task that the language model will tackle is N-best Path Prediction (Top-P Shortest Paths). Given a directed graph $G = (V, E)$ with $|V| = N$ nodes labeled $0, \dots, N-1$ and non-negative integer edge weights $w : E \to 1, \dots, W$, the task is to find the top-P distinct simple paths from source $s = 0$ to target $t = N - 1$ minimizing the additive cost

$$c(\pi) = \sum_{(u,v)\in\pi} w(u, v). \tag{1}$$

The output is a pair

$$\texttt{paths} = [\pi_1, \dots, \pi_P], \quad \texttt{weights} = [c(\pi_1), \dots, c(\pi_P)], \tag{2}$$

sorted by non-decreasing cost. The language model will be expected to use tool calls[1] to specify its answer.

[1] https://platform.openai.com/docs/guides/function-calling

Evaluation compares predicted pairs $(\pi, c(\pi))$ against the reference set with the score

$$\text{score} = \frac{|(\pi, c(\pi))\text{pred} \cap (\pi, c(\pi))\text{gold}|}{P}. \tag{3}$$

MMLU medicine We will use the two medicine-themed splits of MMLU: college_medicine and professional_medicine. Evaluation is on exact match with the correct multiple-choice answer (e.g. "A").

Infobench Infobench provides open-ended queries with detailed evaluation rubrics. Evaluation **requires calling gpt-5-nano**; we expect that the total cost for evaluation for this homework will be substantially less than \$5. See the paper for more information.

1 Optimization for available hardware [25 points]

GPUs and other accelerator hardware have been a significant factor in the (re)surge(nce) of interest in and progress in artificial intelligence and neural machine learning since the early 2010s. Relatively few people who regularly publish at top AI/ML/NLP venues would claim to have a deep understanding of the inner workings of a GPU, however. For the curious, these are two very nicely written blogs that go over the GPU math in detail and discuss the bottlenecks of inference.

- Transformer Inference Arithmetic by Kipply [1]
- Making Deep Learning Go Brrrr From First Principles by Horace He [2]

In general, there are enough well-maintained, open-source, high-level ML frameworks (e.g. PyTorch) and inference engines (e.g. vLLM) that an ML researcher or practitioner can usually get quite far in their career with just a handful of practical rules of thumb. Many of these practical rules of thumb revolve around GPU VRAM[2], and transfers between it and "regular" system RAM. We want to avoid both out-of-memory (OOM) errors and excessive swapping to system RAM.

1.1 Warm-up

During standard autoregressive generation with an LLM, the GPU's VRAM is typically loaded with model weights, KV cache, and a small amount of overhead for model code, intermediate calculations that are not part of the KV cache, and framework overhead.[3] Briefly describe the key differences between LLM generation and training in terms of what is typically loaded into VRAM. What needs to be known and tracked? Is there more or less uncertainty in one? Write 2-3 sentences total.

Solution:

□

1.2 Some GPU math questions

1.2.1 Largest model that fits

What is the largest possible model size you could do a single forward pass[4], on a single 80GB GPU? Assume model weights are at half (16-bit floating point) precision, and report to the nearest billion parameters. State your assumptions and show your work.

Solution:

□

1.2.2 Largest sequence that fits

What is the largest possible sequence length you could generate to on a single L40S GPU, with a batch size of 8 and `Qwen/Qwen3-14B`? Assume the empty string prompt, and again assume half precision for model weights. Please show your work, state your assumptions, and report your answer to the nearest 10 tokens. Feel free to ignore overhead needed for torch and launch operations (i.e., consider only the VRAM size, model weights, and KV cache).

[2] the V stands for "video"; a GPU is after all a graphics processing unit

[3] Though machines with larger VRAM size exist, the most common GPUs used with or for LLMs as of 2025 are around 80GB or around half the size (e.g. 48GB for an A6000 or L40; there are also 40GB A100s).

[4] as we would do to evaluate a model-dataset combination for perplexity score – not for auto-regressive generation

Solution:

□

1.2.3 Estimating KV cache sizes

The size of the KV cache for a single sequence depends on model configuration, sequence length, and the number of bytes per parameter:

$$\text{KV Cache Size} = 2 \times S \times L \times n_{\text{kv}} \times d_{\text{head}} \times \text{bytes per parameter}$$

where S is sequence length, L is number of layers, n_{kv} is number of KV heads, and d_{model} is head dimension. Note: $d_{\text{model}} = n_{\text{kv}} \times d_{\text{head}}$ (or $n_{\text{q}} \times d_{\text{head}}$ for models with GQA/MQA).

Using the model configs for models in the Qwen3 family, calculate the size of the KV cache needed to generate a sequence length of 32k, for batch sizes of 1, 2, and 4. Assume a static KV cache, half precision model weights, and the empty string prompt.

Solution:

Model size (parameters)	Batch size 1	Batch size 2	Batch size 4
`Qwen/Qwen3-0.6B`			
`Qwen/Qwen3-1.7B`			
`Qwen/Qwen3-4B`			
`Qwen/Qwen3-8B`			
`Qwen/Qwen3-14B`			
`Qwen/Qwen3-32B`			

□

Please also show your work for one of your KV cache size calculations.

Solution:

□

How does the size of KV scale with batch size? How does it scale with total parameter size in this model family? Feel free to include a figure(s) as your answer.

Solution:

□

1.3 Basic benchmarking

In this question, you are asked to measure wall-clock time and token throughput. That is, input and output sequence lengths are random sequences intentionally constrained to specific values – e.g. `torch.randint(0, vocab_size), [1,64])` for `bs=1, input_len=64`, with enforced minimum = maximum output sequence lengths. Use 1 GPU and keep the type of GPU consistent for all parts of this question. An 80GB GPU is preferred, but 40GB+ is also acceptable if necessary. State which type of GPU hardware you are using.

Solution:

□

1.3.1 Varying input sequence lengths

Using `Qwen/Qwen3-8B` at half precision (use bfloat16 unless you must use older hardware that does not support it), a batch size of 8, and an output sequence length of 64 (new) tokens, sweep over input sequence lengths of $\{2^n : 0 \leq n \leq 15\}$.

Be sure to perform 2-5 warm-up iterations (fewer needed with longer sequences), and don't forget to run `torch.cuda.synchronize()` after your model finishes running. Measure and report wall-clock time (just use `time.time()`)[5] and token throughput in tokens / second – each number should be an average over 10 batches. Fill out the table provided (only for subset of your sweep, but **also note configurations that led to OOM errors**), and include a figure(s), plotting each metric against each input sequence length. Figures may be either multiple plots or a single combined plot overlaying all three metrics on scales that make sense.

Solution:

Input length (tokens)	Time (s)	Throughput (tokens/s)
1		
4		
16		
64		
256		
1024		
4096		
16384		
32768		

□

1.3.2 Varying output sequence lengths

Repeat the same as above, but with input size 64, and output sizes over $\{2^n : 0 \leq n \leq 8\}$.

Solution:

Output length (tokens)	Time (s)	Throughput (tokens/s
1		
4		
16		
64		
256		

□

1.3.3 Varying the model

Now fix input=64 tokens, output=64, and repeat for three models: `Qwen/Qwen3-1.7B`, `Qwen/Qwen3-8B`, and `allenai/OLMo-7B-0724-hf`. No need for a figure for this one, but report all values in the table, and write 1-2 sentences of reflection and/or analysis.

[5] GPU execution time as measured with `torch.cuda.Event()` is often slightly less than the total CPU wall clock time, but in these particular settings, especially with single GPU inference with vanilla PyTorch/HF there is very little deviation expected

Solution:

Model	Time (s)	Throughput (tokens/s
Qwen/Qwen3-1.7B		
Qwen/Qwen3-8B		
allenai/OLMo-7B-0724-hf		

□

2 Implementation of KV caching [25 points]

One essential component of generation with auto-regressive transformers is **KV caching**, or caching of computed key and value tensors from previous generation steps such that only a partial computation is needed at each new generation step. In this section, we ask you to 1) empirically observe inference with and without KV caching for a standard LLM, and 2) implement your own KV cache for a minimal transformer model.

2.1 Benchmarking with vs. without KV caching

In general, generating without a KV cache is quite slow and almost never done in practice, and longer sequences (both input and output) require more time. In this section, you will perform benchmarking across a variety of settings and gain intuitions about factors that can lead to meaningful differences in *scaling patterns* seen as output sequence length grows. You will compare a short prompt with a long prompt, a small model with a larger model, and an older model with a newer model... We provide a starter notebook, `benchmark-kv-cache.ipynb` and a separate file containing the longer prompt (which gets tokenized to approximately 32k tokens, in `long_prompt.txt`). You do *not* need to submit the notebook.

2.1.1 Baseline figure

The "baseline" benchmark uses 'meta-llama/Llama-3.1-8B' (at half precision) and a short prompt, "Once upon a time,". See the notebook for more complete instructions. Show your figure comparing generation time with vs without a KV cache, and report the GPU hardware you are using as well. **For complete instructions, see `benchmark-kv-cache.ipynb`.**

Solution:
GPU:
Figure:
Description of trends:

□

2.1.2 Longer prompt

Follow the instructions in the notebook to build a cache-vs-no-cache figure for the much longer prompt provided.

Solution:
Figure:
Comparison and explanation: □

2.1.3 Smaller model

Follow the instructions in the notebook to build a cache-vs-no-cache figure for a smaller model, 'meta-llama/Llama-3.2-1B'.

Solution:
Figure:
Comparison and explanation: □

2.1.4 Older model

Follow the instructions in the notebook to build a cache-vs-no-cache figure for a smaller model, 'meta-llama/Llama-2-7b-hf'.

Solution:
Figure:
Comparison and explanation: □

2.2 Implementing a KV cache

Begin with the starter code provided in `kv-cache-implementation.py` and submit your code along with your other deliverables. Be sure to include your name and andrewid at the top of your file in a comment. **Please leave the "TODO" lines intact in your code,** as this will help us with grading.

3 Batching of requests [20 points]

In §1, you were asked to consider the sizes of the models and data at inference time alongside the physical constraints of common GPU hardware. In this section, we focus on batching concerns specifically.

3.1 Conceptual questions

In LLM pre-training, documents are commonly packed together such that all sequences in each batch are a uniform maximum length. This allows for relatively token throughput and relatively high GPU utilization, but this is clearly not universally appropriate or practical at inference time when use cases and deployment settings vary.

3.1.1 Batch or no batch?

In general, inference with a batch size of one tends to lead to poor utilization in a modern GPU. However, it is often assumed or performed in practice. Describe *two* examples of settings where inference with a batch size of 1 is appropriate or even necessary. Prioritize describing specific traits and characteristics of these settings, as opposed to specific instances of these settings (e.g. instead of just naming a specific dataset or model, describe the characteristics that justify its inclusion). Write 2-3 sentences for each example, including an explanation.

Solution: □

3.1.2 Static vs. dynamic vs. continuous batching

Naive batched inference pads sequences to the longest in each batch. In practice, these days it is almost always best to use an inference engine like vLLM or SGLang that does continuous batching, but not all inference settings are created equal, and not all inference optimizations are universally helpful. Describe *one* example of a setting where the benefits of continuous batching over naive batched inference might be relatively limited, and *one* example of a setting where continuous batching would be especially helpful in reducing some meaningful efficiency metric.

Solution:

□

3.2 Pseudo-code for continuous batching with disaggregated prefill and decode

Continuous batching handles not only variation in input sequence lengths (where requests might be bucketed into groups of similar input sequence lengths in order to reduce the amount of padding needed) but also variation in output lengths (where requests finish at different times and are "kicked out" of the GPU at different times). Although requests are sometimes sent with information about minimum or maximum generation length, dynamic serving settings are inherently associated with some uncertainty about output sequence length.

More recently, state of the art inference frameworks have begun supporting *disaggregated* prefill and decode, or disaggregated PD [3]. In this model, one spins up separate e.g. vLLM instances, each dedicated to just one of prefill or decode. KV caches calculated from prefill are saved and sent to the decode server. This has a number of advantages over traditional continuous batching: requests can have very similar input sizes but very different generation lengths, or vice versa; and prefill is compute-bound and highly parallelizeable, while decode requires sequential processing and tends to be a blocking process despite using less compute in a given moment.

Your task: For this problem, you are asked to write pseudo-code as parts of a basic implementation of continuous batching with PD disaggregation. You will design two concurrent loops (one for prefill and one for decode) that together serve all requests. We provide a pseudo-code REQUEST class for you that you may modify if needed. Your pseudo-code should be at an abstract enough level that it should not matter whether or not the underlying memory is continuous.[6]

Assumptions:

- Reasonable relevant variable such as `kv_cache_allocation`, `current_prefill_requests`, `curent_decode_requests`, and `finished_requests` have been pre-initialized.
- You may make up new auxiliary variables as needed.
- Requests have been already been added to a queue, `requests_remaining`
- This is an offline serving setting such that we can assume no further requests will be added as the initial ones are being served
- Basically, no failure modes: feel free to assume that KV cache storage, communication, and loading will always happen at a reasonable speed, that we never run out of disk space, and that we will never have occasion to requeue a request.

[6]In practice, paged attention [4] is typically used with continuous batching in order to alleviate the significant memory fragmentation and associated memory inefficiency that can occur with continuous batching.

We are not looking for any one specific answer. Instead, credit will be awarded for well-thought algorithms with justification. That being said, be sure to address:

1. Conditions for starting a queued request
2. Conditions for stopping generation for a request
3. State tracking and updating of requests
4. Batching requests appropriately for prefill, grouping by similar input lengths
5. Dynamic batch management in decode. You may optionally write an additional Batch class to help clarify your implementation
6. KV cache management (high-level: update, store, load, check sizes)

Magical function calls you may imagine you have at your disposal (unless, of course, you choose to implement them as your additional feature):

1. `does_fit()`, which takes in a request, a collection of requests already in memory, and a total KV cache size and checks (at negligible cost) whether the single request can fit in the GPU. Intended to work for prefill or decode, with the caveat that it will default to the maximum sequence length for the model if no output sequence length is set (and so `does_fit()` should be treated as a conservative bound).
2. `constrained_get()`, a PriorityQueue class method which takes a condition (described in pseudo-code or natural language :)) and returns the next request that fulfills the condition. Intended to work for either prefill and decode

For up to 3 bonus points on this assignment, incorporate *one* of the following additional features (as pseudo-code) into your pseudo-code:

1. A PriorityQueue class implementation that implements length bucketing logic
2. Explicit management of KV cache storage and communication/movement between servers
3. Want to do something else? Feel free to make a Piazza post and get pre-approval

Feel free to write your pseudo-code in a separate file and ask an LLM for help with formatting it into LaTeX :) **Please write your new lines of code and any modifications of provided code in this color**.

Solution:

Algorithm 1 Class Request

```
class Request:
    Input: request_id, input_tokens, output_length (optional)
    self.id ← request_id
    self.input_tokens ← input_tokens
    self.output_length ← output_length
    self.kv_cache ← None # Hint: update during prefill, and check + update during decode
    self.state ← "queued" # Hint: update this to "prefill" → "decode" → "done"
    self.generated_toks ← []
    # You may add additional initializations here

    Function has_prefill_requests(self):
        Return True if not any requests have not yet reached "decode" or "done"

    Function has_decode_requests(self):
        Return True if not any requests have not yet reached "done"

    Function add_token(self, token):
        Append token to self.generated_toks

    # You may add additional functions here

```

Algorithm 2 Prefill loop (runs simultaneously with decode loop)

```
while requests_remaining.has_prefill_requests() do
    # TODO: your pseudo-code implementation here
```

Algorithm 3 Decode loop (runs simultaneously with prefill loop)

```
while requests_remaining.has_decode_requests() do
    # TODO: your pseudo-code implementation here
    current_decode_requests.step() # generates one token per sequence currently in the batch
    # TODO: more pseudo-code
```

□

Use the space below to describe (in regular English) any assumptions you have made, along with a description of your algorithm and what it does. List and describe also any optional feature(s) you have included

Solution:

□

4 Speculative Decoding [30 Points]

4.1 Background

Large language models (LLMs) typically perform *autoregressive decoding*, generating one token at a time. This process is often *memory-bound*, meaning throughput is limited by the transfer of model weights rather than compute.

Speculative decoding accelerates generation by using a small, fast *draft model* to propose multiple tokens at once, which are then verified in parallel by a larger *target model*.

The method was independently introduced in *Fast Inference from Transformers via Speculative Decoding* [5] and *Accelerating Large Language Model Decoding with Speculative Sampling* [6].

We will use **Algorithm 2** from the second paper as our reference for the implementation. Reading Theorem 1 from paper 2 is optional but helpful for intuition about why rejection sampling preserves the target distribution.

4.2 Benchmarking Forward Pass on a Single GPU

Using model `Qwen/Qwen3-4B`:

- Measure the forward-pass time for batch sizes $B \in \{1, 2, 4, 8, 16\}$ at a fixed sequence length $S = 256$.
- Run each configuration for 5 trials after appropriate warm-up iterations to obtain the average time. Use `torch.cuda.Event`, `torch.cuda.time_record` for accurate GPU timing.
- **Plot results:** Batch Size vs. Wall-Clock Time (ms).
- **Briefly discuss:** How does the forward cost scale with B, and why is this relevant for speculative decoding?

Solution:

□

4.3 Implementation

Implement speculative decoding in the provided scaffold `specdec.py`. You only need to modify the `decode()` function; model loading and tokenization are handled for you.

You are given 20 test prompts (`prompts.jsonl`) and a benchmarking script (`benchmark.py`) that runs inference over them. You may edit it for finer timing analysis.

Solution:

□

4.4 Evaluation

In practice, speedups depend on both software and hardware factors. In this section, you will analyze how different target–draft pairs and lookahead values affect throughput under heterogeneous request lengths.

All experiments should be run on a single GPU. Each configuration should fit comfortably on GPUs with $\geq$48 GB VRAM.

Run your implementation with the following configurations:

- Target = Qwen-3-8B; Drafts = {Qwen-3-1.7B, Qwen-3-0.6B}
- Target = Llama-3.1-8B; Draft = Llama-3.2-1B

For each configuration, evaluate with speculative lookahead $\gamma \in \{2, 3, 5, 7\}$ and record:

- Empirical speedup = (Wall-clock time of AR baseline) / (Wall-clock time of speculative decoding)
- Empirical acceptance rate α

Deliverables

- Present results in a **table** and **plot the speedups vs. γ for each target–draft pair**.
- For your highest overall speedup configuration also comment on how the speedups vary per data source and why.

Include the GPU name and memory capacity in your table. Use `torch.manual_seed(42)` to ensure reproducibility.

Solution:

Target Model	Draft Model	γ	α	Speedup
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 1: Acceptance rate and speedup (3 model pairs * 4 gammas = 12 runs total).

□

4.5 Hardware Analysis

Speculative decoding assumes inference is primarily memory-bound rather than compute-bound. This balance can shift across GPU architectures.

Select the best- and worst-performing configurations (in terms of speedup) and rerun them on a different GPU architecture. A *configuration* refers to a (Target, Draft, γ) combination.

1. Create a table including: VRAM capacity, memory bandwidth (GB/s), compute capability, and tensor core specifications for both GPUs (for bf16).

   Solution:

$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

   Table 2: configs for 2 gpus

   □

2. Report speedup on both GPUs and compare results.

Solution:

⋮	⋮	⋮	⋮	⋮

Table 3: 4 runs (2 that you have from §4.4, same configs on the new gpu)

□

3. Briefly explain the observed differences in speedup, referencing hardware factors such as memory bandwidth and TFLOPS.

Solution:

□

4.6 Analysis

4.6.1 Background

Let T_T and T_D denote the time for one forward pass of the target and draft models respectively, and T_V the verification time for γ tokens by the target.

The total speculative decoding time is:

$$T^{SD}_{\text{Total}} = \gamma \cdot T_D(B, S) + T_V(B, S, \gamma).$$

Given draft token acceptance rate $\alpha \in [0, 1]$ and lookahead γ, the expected number of tokens generated in one verification step is [5]:

$$\Omega(\gamma, \alpha) = \frac{1 - \alpha^{\gamma+1}}{1 - \alpha}. \tag{4}$$

Thus, the expected average latency per token is:

$$T^{SD}_{\text{Avg}} = \frac{T^{SD}_{\text{Total}}}{\Omega(\gamma, \alpha)},$$

and the relative latency (normalized by the target model's cost) is [7]:

$$\frac{T^{SD}_{\text{Avg}}}{T_T} = \frac{1}{\Omega(\gamma, \alpha)} \left(\gamma \cdot \frac{T_D}{T_T} + \frac{T_V(\gamma)}{T_T} \right). \tag{5}$$

From Eq. 5, the speedup depends on three key factors: (i) acceptance rate and expected generated tokens, (ii) the draft-to-target cost ratio, and (iii) verification overhead.

4.6.2 Questions

Combine your empirical results in the light of the the factors discussed above (and other relevant factors) to answer the following questions.

1. How do speedup and acceptance rate vary with γ? How do these trends differ across draft sizes and model families and why?

Solution:

□

2. **Optimal γ:** As you saw from your experiments, the speedup is a function of the γ value.

   How would you determine the optimal γ for your given system? Which factors should be considered?

   Solution:

   □

3. **Batched Inference:** What challenges arise for speculative decoding with batch size > 1? Sketch pseudo-code for a batched version and discuss what makes the implementation challenging.

   You might want to think about this in detail, as it can be very useful for the final project.

   Solution:

   □

References

[1] Kipply. Transformer inference arithmetic. https://kipp.ly/transformer-inference-arithmetic/, 2023. Accessed: 2025-11-02.

[2] Horace He. Making deep learning go brrrr: From first principles. https://horace.io/brrr_intro.html, 2020. Accessed: 2025-11-02.

[3] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.

[4] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Haotong Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[5] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding, 2023.

[6] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling, 2023.

[7] Ranajoy Sadhukhan, Jian Chen, Zhuoming Chen, Vashisth Tiwari, Ruihang Lai, Jinyuan Shi, Ian En-Hsu Yen, Avner May, Tianqi Chen, and Beidi Chen. Magicdec: Breaking the latency-throughput tradeoff for long context generation with speculative decoding, 2025.