**COS 484: Natural Language Processing** **(Due: 04/14/2026)**

# Assignment #4

*Instructors:* Tri Dao, Karthik Narasimhan

**Course Policy**: Read all the instructions below carefully before you start working on the assignment and before you make a submission. The course assignment policy is available at https://princeton-nlp.github.io/cos484/. When you're ready to submit, please follow the instructions found here: http://bit.ly/COS_NLP_Submission

- This assignment contains 2 parts, a theoretical and a programming part. The former consists of 2 problems, and the latter has 1, for a total of **3 problems**.
- We *highly* recommend that you typeset your submissions in LaTeX. Use the template provided on the website for your answers. If you have never used LaTeX, you can refer to the short guide here: http://bit.ly/WorkingWithLaTeX. Include your name and NetIDs with your submission. If you wish to submit hand-written answers, you can scan and upload the pdf.
- Assignments must be uploaded to Gradescope by **11:59pm Eastern** on the due date mentioned above.
- As per the late-day policy outlined on the course website, you have 4 late days that you can use any time during the semester, with at most 3 late days per assignment. Once you run out of late days, late submissions will incur a penalty of 10% for each day, up to a maximum of 3 days beyond which submissions will not be accepted.
- All programming problems in this class should be completed in Google Colab using Python. If you would like to get familiar with this environment, you may complete the problems in this introductory Colab notebook (This will not be graded). If you've never worked with Google Colab before, take a look through this introduction guide: Working With Colab. **The answers to the written questions proposed in the programming part should be answered in your Colab notebook.**
- **LLM usage policy**: You **may not** consult a Large Language Model (LLM) when working on the **Theoretical** part of this assignment. For the **Programming** Part only, you may use coding assistants (like GitHub Copilot, Cursor, etc.) for writing code. If you do use such assistants, include the prompts you used at the end of your Colab notebook.

**Theoretical Part**

**Submission Policy**: Submit a single PDF for the answers to all questions in this part.

**Problem 1: Cross-Entropy Loss – Forward and Backward** (30 points)

In this problem we will analyze the cross-entropy loss used in training large language models. The cross-entropy loss is the composition of a softmax and a negative log-likelihood, and is implemented in PyTorch as `torch.nn.CrossEntropyLoss` (https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html).

Let $\mathbf{x} \in \mathbb{R}^V$ be the input logits for a single token, where $V$ is the vocabulary size, and let $y \in \{1, \dots, V\}$ be the target class (the correct next token).

**(a) Forward pass derivation.** (5 points)

The cross-entropy loss for a single token is defined as:

$$\ell(\mathbf{x}, y) = -\log\left(\frac{\exp(x_y)}{\sum_{j=1}^{V}\exp(x_j)}\right)$$

Show that this can be rewritten as:

$$\ell(\mathbf{x}, y) = -x_y + \log\left(\sum_{j=1}^{V}\exp(x_j)\right)$$

In practice, for numerical stability, the computation uses the "log-sum-exp trick": defining $m = \max_j x_j$, the loss is computed as:

$$\ell(\mathbf{x}, y) = -x_y + m + \log\left(\sum_{j=1}^{V}\exp(x_j - m)\right)$$

Explain why this is equivalent and why it is more numerically stable.

**(b) Backward pass derivation.** (5 points)

Derive the gradient of the loss with respect to the logits, $\frac{\partial \ell}{\partial x_i}$, for all $i \in \{1, \dots, V\}$. Specifically, show that:

$$\frac{\partial \ell}{\partial x_i} = \text{softmax}(\mathbf{x})_i - \mathbb{1}[i = y] = \frac{\exp(x_i)}{\sum_{j=1}^{V}\exp(x_j)} - \mathbb{1}[i = y]$$

where $\mathbb{1}[i = y]$ is the indicator function that equals 1 when $i = y$ and 0 otherwise.

*Hint*: Differentiate the expression from part (a) with respect to $x_i$, considering the two cases $i = y$ and $i \neq y$.

**(c) Memory I/O, FLOPs, and arithmetic intensity.** (20 points)

Now consider a batch of $B$ tokens, where the logits are a matrix $\mathbf{X} \in \mathbb{R}^{B\times V}$ in `bfloat16` (2 bytes per element), the targets are $\mathbf{y} \in \{1, \dots, V\}^B$ stored as `int64` (8 bytes per element), and the output losses are $\boldsymbol{\ell} \in \mathbb{R}^B$ stored in `float32` (4 bytes per element). Assume `reduction='none'` (i.e., the loss is not aggregated via mean or sum — we output one loss value per token, so the output has $B$ elements). For the backward pass, the upstream gradient $\frac{\partial L}{\partial \boldsymbol{\ell}} \in \mathbb{R}^B$ is `float32` (4 bytes), and the output gradient $\frac{\partial L}{\partial \mathbf{X}} \in \mathbb{R}^{B\times V}$ is `bfloat16` (2 bytes). The backward kernel reloads the logits $\mathbf{X}$ and targets $\mathbf{y}$ to recompute the softmax.

Count each addition, subtraction, multiplication, division, exponentiation, and logarithm as 1 FLOP. You may ignore the cost of the max operation.

(i) Count the total bytes loaded from and stored to GPU HBM for the **forward pass** (assuming a fused kernel — no intermediates materialized) and the **backward pass**. Express in terms of $B$ and $V$. (4 points)

(ii) Count the FLOPs per token for the **forward pass** and **backward pass**. Express in terms of $V$. (4 points)

*Hint (forward)*: Steps are: $x_j - m$ for all $j$, $\exp(\cdot)$ for all $j$, sum, log, final subtraction/addition.

(iii) Plug in $B = 4{,}096$ and $V = 128{,}256$. Compute the total bytes (in GB) for forward and backward. (2 points)

(iv) The *arithmetic intensity* (AI) is FLOPs / bytes transferred. Compute the AI for the forward and backward passes (the $B$ terms should cancel). (4 points)

(v) On an A100 SXM, peak bf16 compute is ∼312 TFLOPS and peak memory bandwidth is ∼2039 GB/s, giving a compute-to-bandwidth ratio of ∼153 FLOPs/byte. Based on your AI values, is cross-entropy memory-bound or compute-bound? What does this imply for how we should optimize the kernel? (6 points)

## Problem 2: Reward Modeling (30 points)

In this problem, we will analyze the reward modeling step used in Reinforcement Learning from Human Feedback (RLHF) (Ouyang et al., 2022). Recall that after supervised fine-tuning (SFT), we train a **reward model** $r_\theta(x, y)$ to score how good a response $y$ is for a given prompt $x$. This reward model is then used to optimize the language model policy via reinforcement learning (e.g., PPO) or direct preference optimization (e.g., DPO). This reward model is trained using **human preference data**. Specifically, given a prompt $x$ and two responses $y_w$ (winning) and $y_l$ (losing), the **Bradley-Terry model** assumes:

$$P(y_w \succ y_l \mid x) = \sigma\big(r_\theta(x, y_w) - r_\theta(x, y_l)\big)$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function.

### (a) Pairwise loss and its gradient. (12 points)

The reward model is trained by minimizing the negative log-likelihood of observed preferences over a dataset $\mathcal{D} = \{(x^{(n)}, y_w^{(n)}, y_l^{(n)})\}_{n=1}^N$:

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{n=1}^{N} \log \sigma\big(r_\theta(x^{(n)}, y_w^{(n)}) - r_\theta(x^{(n)}, y_l^{(n)})\big)$$

Consider a single training example and define the **reward margin** $\Delta = r_\theta(x, y_w) - r_\theta(x, y_l)$.

(i) (5 points) Compute $\frac{\partial \ell}{\partial\, r_\theta(x, y_w)}$ and $\frac{\partial \ell}{\partial\, r_\theta(x, y_l)}$, where $\ell = -\log \sigma(\Delta)$ is the loss for a single example. Express your answers in terms of $\sigma(\Delta)$.

*Hint:* Recall that $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

(ii) (4 points) Using your answer from (i), express the magnitude of the gradient $\left|\frac{\partial \ell}{\partial\, r_\theta(x, y_w)}\right|$ as a function of $\sigma(\Delta)$. For what value of $\sigma(\Delta)$ (and corresponding $\Delta$) is this magnitude largest? Explain intuitively why this makes sense.

(iii) (3 points) Describe qualitatively what happens to the gradient when $\Delta \gg 0$ (the model is already very confident that $y_w$ is better) and when $\Delta \ll 0$ (the model confidently assigns the wrong ranking). Compare this to the behavior of logistic regression.

### (b) From rankings to pairwise comparisons. (10 points)

We saw that in practice InstructGPT did not collect simple pairwise comparisons. Instead, for each prompt $x$, human labelers were shown $K$ responses $\{y_1, y_2, \ldots, y_K\}$ (sampled from the model) and asked to produce a **full ranking** $y_{\pi(1)} \succ y_{\pi(2)} \succ \cdots \succ y_{\pi(K)}$, where $\pi$ is a permutation. InstructGPT used $K$ between 4 and 9.

(i) (3 points) A ranking of $K$ items induces a set of pairwise preferences: for every pair $i, j$ with $\pi(i) < \pi(j)$ (i.e., $y_{\pi(i)}$ is ranked higher than $y_{\pi(j)}$), we get a preference $(x, y_w = y_{\pi(i)}, y_l = y_{\pi(j)})$. How many pairwise comparisons does a single ranking of $K$ items produce? Express your answer in terms of $K$.

(ii) (4 points) Write the reward model loss for a single prompt $x$ with a full ranking of $K$ responses, including the appropriate normalization. That is, write $\mathcal{L}_{\text{rank}}(\theta)$ as a sum over all valid pairs $(y_w, y_l)$ implied by the ranking.

(iii) (3 points) Explain the advantage of collecting full rankings rather than independent pairwise comparisons in terms of **annotation efficiency**. Specifically: if you have a budget of $N$ labeler interactions (each interaction = one task the annotator performs given a prompt and the set of responses), how many pairwise training signals do you get with each strategy? Use $K = 9$ for a concrete comparison.

**(c) Shift invariance of the reward model. (8 points)**

(i) (4 points) Prove that the Bradley-Terry loss $\mathcal{L}(\theta)$ is **shift-invariant**, i.e. adding any constant $c \in \mathbb{R}$ to all reward scores does not change the loss. Then, if we define $r'_\theta(x, y) = r_\theta(x, y) + f(x)$ for an arbitrary function $f$ that depends only on the prompt $x$ (not on the response $y$), show that $\mathcal{L}$ is identical under $r_\theta$ and $r'_\theta$.

(ii) (4 points) This invariance means the reward model can only learn **relative** differences in quality, not absolute scores. Explain why this is problematic if we want to use the reward model to compare responses across *different* prompts (e.g., "Is response $y_1$ to prompt $x_1$ better than response $y_2$ to prompt $x_2$?"). Then, suggest one practical way to mitigate this issue.

## Programming Part

**Submission Policy**: The answers to the written questions for this part should be answered in the Colab notebook. **Do not include the answers to this part in the same PDF as the theory part.**

## Problem 1: Cross-Entropy Loss on GPU (40 points)

In this problem you will benchmark the cross-entropy loss in PyTorch and measure its efficiency relative to the theoretical memory bandwidth of the GPU. You will predict performance *before* measuring, then explain any discrepancies — this is the core skill of systems performance analysis. To get started, open this Colab notebook.

**Setup**: Use the following parameters throughout:

- Batch size: $B = 4{,}096$
- Vocabulary size: $V = 128{,}256$
- Logits dtype: `torch.bfloat16`
- Reduction: `'none'`
- Device: GPU (use a Colab GPU runtime, e.g., T4 or A100)
- **Report your PyTorch version** (e.g., `torch.__version__`) and GPU model in your notebook. Results can vary significantly across versions.

Generate random input logits and targets:

```
import torch

B, V = 4096, 128256
logits = torch.randn(B, V, dtype=torch.bfloat16, device='cuda',
                     requires_grad=True)
targets = torch.randint(0, V, (B,), device='cuda')
```

**(a) Predict before you measure.** (8 points)

*Before writing any benchmarking code*, use your answers from Theory Problem 1 to predict the performance of an ideal (perfectly memory-bound) cross-entropy kernel on your GPU.

(i) Look up the peak memory bandwidth of your Colab GPU. For this assignment, if you are using an A100, assume the **A100 SXM** bandwidth of ∼2039 GB/s; if you are using a T4, use ∼300 GB/s. Using the total bytes from Theory Problem 1(c), compute the **minimum possible time** (in ms) for the forward pass and backward pass, assuming the kernel runs at 100% of peak memory bandwidth. (4 points)

(ii) Based on your arithmetic intensity analysis from Theory Problem 1(c), explain why memory bandwidth (not TFLOPS) is the right performance limiter and the right metric to evaluate the efficiency of cross-entropy. (4 points)

**(b) Benchmark PyTorch eager mode.** (12 points)

(i) Write a benchmarking function that measures the wall-clock time (in milliseconds) for the cross-entropy **forward pass** and **backward pass** separately. Use `torch.cuda.Event` with `enable_timing=True` for accurate GPU timing. After recording the end event, make sure to synchronize the GPU (e.g., with `torch.cuda.synchronize()` or `end_event.synchronize()`) before reading the elapsed time. Run each measurement multiple times (e.g., 100 iterations) and report the **median** time. Make sure to do a few warmup iterations before timing. (5 points)

(ii) Compute the **achieved memory bandwidth** (in GB/s) for the forward and backward passes:

$$\text{Achieved BW (GB/s)} = \frac{\text{Total bytes (GB)}}{\text{Time (s)}}$$

How does the measured time compare to your prediction from part (a)? If there is a significant gap, **explain why** PyTorch eager mode is slower than the ideal. What extra memory traffic or overhead might the eager implementation incur? (7 points)

*Hint*: Think about whether the eager implementation fuses all the operations into a single kernel, or whether it materializes intermediate tensors (e.g., the full softmax output) to HBM.

**(c) Benchmark `torch.compile` mode.** (10 points)

(i) Wrap the cross-entropy computation with `torch.compile` and repeat the benchmarking from part (b). Note that the first call triggers compilation — make sure your warmup accounts for this. Report the achieved memory bandwidth. (4 points)

(ii) How does `torch.compile` compare to eager mode and to your ideal prediction? Explain what `torch.compile` is likely doing differently (e.g., kernel fusion) that accounts for the performance difference. (6 points)

**(d) Inspecting the compiled kernel.** (10 points)

When `torch.compile` compiles your function, it generates Triton kernel code under the hood.

(i) Use the environment variable `TORCH_LOGS=output_code` or `torch._dynamo.config.log_level` to dump the generated Triton kernel code for the cross-entropy forward pass. Include the generated code (or relevant excerpts) in your notebook. (3 points)

(ii) Annotate the generated kernel: identify which lines correspond to (1) computing the max for numerical stability, (2) the exp and sum (softmax denominator), (3) the log-sum-exp, and (4) the final loss computation. Relate these back to the formulas you derived in Theory Problem 1(a). (7 points)

**Problem 2: Bonus — Cross-Entropy Speed Competition** (up to 20 bonus points)

Write the fastest possible cross-entropy implementation (forward + backward) and submit it to our benchmarking server. Your implementation will be timed on an **NVIDIA A100 (80GB)** GPU with $B = 4{,}096$, `bfloat16` logits, `reduction='none'`, across three vocabulary sizes: $V \in \{32{,}000,\ 50{,}264,\ 128{,}256\}$.

**Rules**:

- Your submission must be a single Python file that defines two functions:

```
def cross_entropy_forward(logits, targets):
    """
    Args:
        logits: (B, V) tensor, bfloat16
        targets: (B,) tensor, int64
    Returns:
        losses: (B,) tensor, float32
    """
    ...

def cross_entropy_backward(logits, targets, grad_output):
    """
    Args:
        logits: (B, V) tensor, bfloat16
        targets: (B,) tensor, int64
        grad_output: (B,) tensor, float32
    Returns:
        grad_logits: (B, V) tensor, bfloat16
    """
    ...
```

- Your outputs must be numerically correct (we will check against a reference implementation with a tolerance of `atol=1e-3`, `rtol=1e-2`).
- **Allowed packages**: `torch` and `triton` only (and their submodules, e.g., `torch.nn.functional`, `triton.language`). No other external packages.
- **Environment**: Submissions will be evaluated with `PyTorch 2.11.0` and `Triton 3.6.0` on CUDA 12.x.
- You may write custom Triton kernels, use `torch.compile`, or any combination thereof.
- For each vocab size, the forward + backward time is measured using `torch.cuda.Event` (median of 100 runs after warmup). Your **competition score** is the **geometric mean of the speedup** (baseline time / your time) for the combined fwd+bwd across all three vocab sizes. The baseline is PyTorch eager mode. The test script reports fwd, bwd, and fwd+bwd times separately for your analysis, but only the fwd+bwd speedup is used for ranking.

**Local testing**: We provide a test and benchmarking script (`test_cross_entropy.py`) and a baseline submission (`baseline_submission.py`), available at: https://drive.google.com/drive/folders/1k3WSruZ2ahcO18jesg5cDyDu0usp=sharing. You can test your implementation locally before submitting:

```
python test_cross_entropy.py your_submission.py
```

This will verify correctness against the reference implementation and report timing and achieved memory bandwidth. The baseline uses PyTorch eager mode — try to beat it!

**Submission**: The benchmarking server and precise submission instructions will be announced on Ed. The server will be open from **Tuesday, April 7** to **Thursday, April 16**. You may submit multiple times; only your fastest correct submission will be scored. Note: the bonus competition is separate from the rest of the assignment — submitting to the server after the April 14 due date does **not** count as a late day. Late days do not apply to the competition; the server simply closes on April 16 at 11:59pm Eastern.

**Optimization journal** (required for bonus credit): Along with your submission, include a short write-up (in your Colab notebook or a separate PDF) documenting your optimization process:

- What approaches did you try? (e.g., `torch.compile` tricks, custom Triton kernels, fused forward+backward, memory layout changes, block size tuning, etc.)
- For each approach, report the timing and explain *why* it was faster or slower than your previous best.
- What is the achieved memory bandwidth of your final submission, and what fraction of peak A100 bandwidth (2039 GB/s) does it reach? What do you think is preventing you from reaching 100%?

The journal is worth 5 of the bonus points — even if your implementation is not in the top 20, a thoughtful journal demonstrating genuine experimentation and understanding can earn partial bonus credit.

**Leaderboard scoring**: The top 20 entries will receive bonus points:

- 1st–3rd place: 15 points
- 4th–5th place: 10 points
- 6th–10th place: 7 points
- 11th–15th place: 4 points
- 16th–20th place: 2 points