Assignment

AIL861 / ELL8299 / ELL881: Advanced LLMs

Due Date: October 24, 2025 (Friday), 11:59 pm

Total Points: 75

General Instructions

- You should submit all your code (including any pre/post-processing scripts written by you). Also, include a **single README file** with instructions on how to run your code for each part of the assignment.
- You should also submit a **report (pdf) file**, which includes a brief description for each part of the assignment explaining what you did. Include all your results, observations and analyses in this report.
- You should use Python as the programming language.
- Your code should have appropriate documentation for readability.
- This is an **individual assignment**. You are supposed to carry out all the implementations by yourself.
- There will be a penalty for AI-generated codes. We also plan to run Moss on your submissions. Any detected case of plagiarism will result in a zero on the assignment and possibly much stricter penalties (including a **fail grade** and/or a **DisCo**).
- You must submit your assignment via **Moodle** as a single `.zip` file named with your entry number (e.g., `2025XYZ123.zip`). This zip must include your report (pdf) and all relevant code files.
- Feel free to post your doubts on **Piazza**.
- Start early!

1 Implementing a Decoder-Only Transformer (35 points)

The goal of this assignment is to develop a decoder-only transformer language model from scratch. You will begin by implementing a minimal version of the transformer model, training it on a text dataset, and then progressively add more advanced techniques into your implementation. Please note that you have to write the entire code on your own. To ensure that you gain a deep understanding of the workings of transformer-based language models, you are restricted to using only the most basic functionalities of PyTorch (such as `nn.Linear`, `nn.Embedding`, `nn.Parameter`, simple non-linearities, and `nn.Dropout`) along with other common libraries (Numpy, Pandas, etc.)

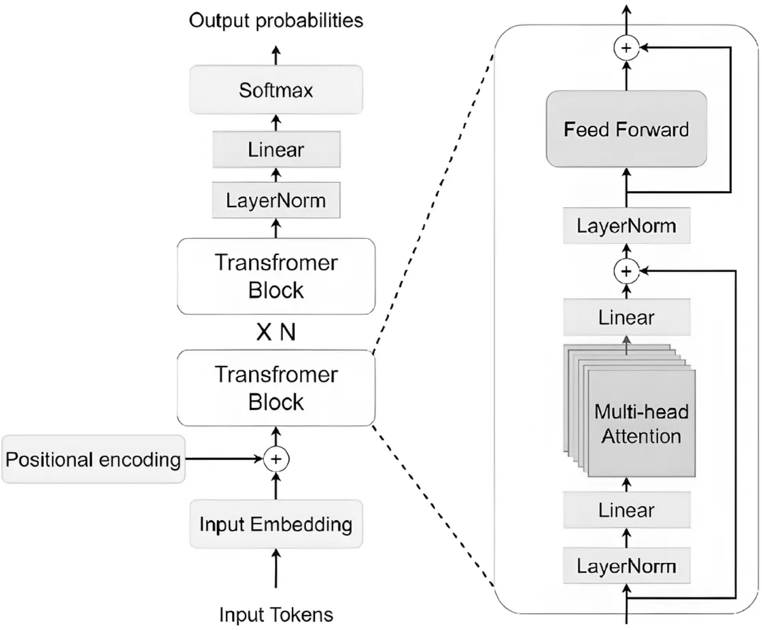
1.1 Pre-Training Dataset

In this assignment, you will train (or, rather 'pre-train'!) your transformer model using the TinyStories dataset, which is publicly available on Hugging Face. This dataset contains short narrative texts in English, which are well suited for toy language modeling tasks. You may preprocess the data in any way, that makes it suitable for training the model, but do mention your methods in the report. For all training runs, use ONLY the *train split* of the dataset. The *validation split* must be used ONLY for inference, evaluation of metrics, and reporting results. Don't end up training on the test set (the goal is to understand how pre-training works, not to 'hack' any benchmark :)).

Here is the link to the dataset: TinyStories.

1.2 Model Architecture

You will create a general blueprint for a decoder-only transformer, that should take as input key hyperparameters such as number of layers, number of attention heads in each multi-head attention block, hidden dimension, context size, etc. You have to implement everything from scratch, including LayerNorm. For the embedding layer, you should initialize your embeddings using *FastText* pre-trained vectors. You may choose to fine-tune the embeddings or keep them fixed depending upon your compute. You should use *word-level tokenization* to map text into FastText embeddings. Make sure your model correctly handles special tokens such as `<sos>` (start of sequence), `<eos>` (end of sequence), and `<pad>` (padding). For positional information, you should use the standard sinusoidal positional encoding as introduced in the original Transformer paper (*Attention is All You Need*, Vaswani et al., 2017). The overall structure of each decoder layer must follow the standard transformer design as shown below:



1.3 Training (20 points)

After setting up the transformer architecture and implementing the masked multi-head self-attention mechanism, you must train your model on the TinyStories dataset using **teacher forcing**. Your code should be written efficiently and must exploit **parallelization** (especially in the attention computations) to make training scalable. Keep in mind the importance of batching and vectorized tensor operations when implementing the forward pass.

During training, monitor both the training and validation performance of your model. Specifically, you are required to:

- Plot the **training loss curve** across steps/epochs.
- Plot the **validation loss curve** alongside the training curve.
- Plot the **perplexity curve** as training progresses.

Include these plots in your report. We also advice you to use `wandb` (or something similar) to monitor training runs, for your convenience.

Recommended baseline configuration. To help you get started, here is a simple configuration which you may use as a baseline:

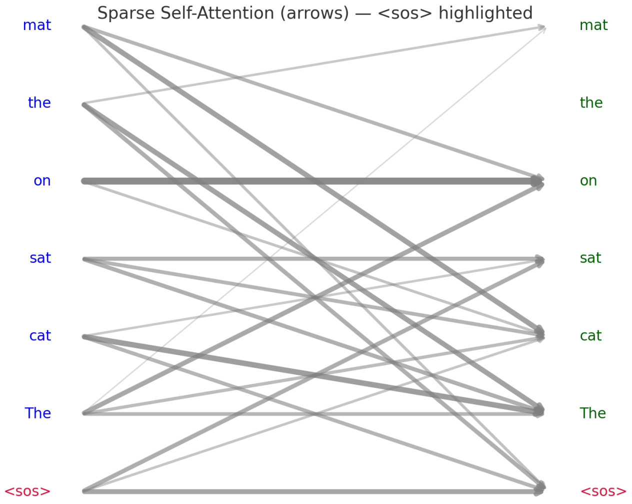
- Context length: 64
- Number of layers: 3
- Number of attention heads: 8
- Hidden (embedding) dimension: Same as the FastText embedding dimension you choose to use
- Optimizer: Adam; Learning rate 3×10^{-4}; No weight decay

Once everything is implemented and works fine, you can play around with different hyperparameters and report your observations.

1.4 Inference (15 points)

Once your model is trained, you must implement a `generate()` function that produces text auto-regressively from a given prompt. The function should take a text prompt as input, generate a continuation token by token, and return the full-generated sequence (until the `<eos>` token is generated, or a maximum sequence length is reached). Make sure the generated sequence is not deterministic in nature (i.e., include stochastic sampling strategies). Include some sample generations in your report, and also comment on the meaningfulness of the sentence generated. Take 50 different samples from the validation dataset. For each sample, use the first 5 tokens as a prompt and let your model generate the continuation. Compute and report:

- the average *perplexity per token* over all generated continuations.
- the average BLEU score (you can use Hugging Face `evaluate` library).
- Select at least 2–3 example sentences from the dataset and visualize the self-attention weight patterns for each attention head. Please refer to the figure below to get an idea. Alternatively, you can also use heatmap visualization for this purpose. Try to interpret what each head is learning and how they differ from other heads.



2 Training and Inference Enhancements (40 points)

In this part of the assignment, you will extend your implementation with more advanced techniques that are commonly used in large-scale language models. For each feature, you must implement the functionality from scratch, verify correctness, and report the effect on performance, efficiency, and generation quality. Wherever relevant, provide both plots and a short discussion in your report.
(**Note:** Each of the experiments are independent, i.e., if you are implementing KV caching, you are not required to use beam search for it and likewise.)

2.1 Beam Search Decoding (10 points)

Extend your `generate()` function to support beam search decoding. Experiment with beam widths $k = 5$ and $k = 10$, and compare the generated outputs with your previous decoding strategy. For both beam sizes, measure **runtime efficiency** (in terms of *tokens per second*) and the **BLEU score** of the generated continuations against the ground truth. Use 5 prompts from the validation data for this evaluation. Present the results in a table or plot, and briefly comment on the trade-off you observe between decoding speed and output quality.

2.2 KV Caching (10 points)

Extend your `generate()` function to support **Key–Value (KV) caching** in the self-attention mechanism, so that previously computed keys and values are reused during autoregressive generation instead of being recomputed at every step. Evaluate the benefit of KV caching by comparing it against your baseline implementation:

- **Runtime efficiency:** measure number of tokens generated per second for a batch of 20 samples, both with and without KV caching.

2.3 Gradient Accumulation (10 points)

Implement gradient accumulation to simulate training with larger effective batch sizes. Fix your mini-batch size (e.g., 16) and vary the number of gradient accumulation steps (2, 4, and 8), corresponding to effective batch sizes of 32, 64, and 128.

- Plot the **training loss curves** for all settings (baseline without accumulation, and accumulation steps 2, 4, 8) on a single graph for direct comparison.
- Report the **runtime per epoch** for each configuration in a table.

2.4 Gradient Checkpointing (10 points)

Implement manual gradient checkpointing for your transformer model. You may *not* use `torch.utils.checkpoint`; design your own mechanism.

Train your model with and without checkpointing, and compare:

- **Peak GPU memory usage** (measure with `torch.cuda.max_memory_allocated()`)
- **Training speed** (runtime per epoch)

Deliverables

Submission format: A single `.zip` file on Moodle, named with your entry number, containing both the code and report.

Your submission must include:

- **Code:** Complete and well-documented code (model, training, inference, and evaluation), along with a README file with instructions on how to run your code.
- **Model checkpoint(s):** Upload your trained model(s) as public repository on Hugging Face Hub, and share the path(s) at the top of your report.
- **Report (PDF):** A single report containing implementation details, training/validation plots, evaluation metrics (perplexity, BLEU), and sample generations. Include clear plots/tables for all experiments, along with a detailed explanation and your interpretations of the results in the report itself.