ITCS 6101/8101 Homework 5: Implementation (100 points)

March 20, 2026

1 Overview

In this assignment, you are given a code base in Python that implements a multi-layer Transformer decoder that is trained to do language modeling (TransformerLM). The model takes the embeddings of all tokens seen so far as input and computes the next word distribution. The skeleton code and data are linked from the course webpage. Organize your code in folders as shown in Table 1 below.

```
hw05/
   report.pdf
   code/
     TransformerLM.py
     lm_model.pt
     traces.txt
   data/
     tiny/
         tiny.train.tokens.4M + tiny.valid.tokens.200K
         tiny.test.tokens.4M
     wiki/
         wiki.train.tokens + wiki.valid.tokens
         wiki.test.tokens
```

Table 1: Folder structure.

The `main()` function is the main entry point in the program, and accepts command line parameters for 4 modes of use:

1. `-train`: Train the model on a tokenized training corpus, by calling the `train()` function.

2. `-use`: Use the trained model in interactive mode to predict the next token for a sequence of tokens from the user, by calling the `use()` function.

3. `-prompt`: Use the trained model to generate a complete sequence of tokens conditioned on a prompt from the user, by calling the `prompt()` function.

4. `-evaluate`: Evaluate the trained model on a tokenized test corpus, by calling the `evaluate()` function.

Furthermore, the `main()` function processes the command line to read a number of program parameters, including `-num_layers` (the number of Transformer decoder layers/blocks), `-d_model` (the dimensionality of the input and output contextual embeddings produced by the model in each block), `num_heads` (the number of attention heads), `dim_feedforward` (the dimension of the intermediate feedforward layer), `activation` (the activation function

used by the intermediate feedforward layer: 'relu' or 'gelu'), `dropout` (the dropout rate), `norm_first` (if true, layers perform LayerNorms before other attention and feedforward operations, otherwise after), `-E` (the number of training epochs), `-B` (the minibatch size, i.e., the number of stories in a minibatch), `-VSize` (the vocabulary size), `-greedy` (whether to use greedy vs. multinomial sampling), as well as paths for various types of input files.

To train, tune, and evaluate the TransformerLM model, you will use a portion of the TinyStories dataset, which contains short stories that only contain words that a typical 3 to 4-year-olds usually understand, generated by GPT-3.5 and GPT-4. The dataset is stored in `data/tiny/` and is partitioned into training, validation, and test sets.

2 Implementation Details

Complete the implementation by writing code in the sections marked with `YOUR CODE HERE`:

1. **Implement the forward pass** through one Transformer decoder layer in the function `TransformerDecoderLayer.forward()`. Apply dropout on the attention vector as well as on the output of the feed-forward network.

2. **Implement the forward pass** through the entire Transformer decoder in the function `TransformerLM.forward()`, by passing through each decoder layer. Make sure you use a causal mask to prevent attending to future tokens. Scale the input embeddings by `sqrt(d_model)`.

3. Given a model and a set of stories, **compute the average cross-entropy** loss in the function `compute_loss`. You may want to compute the loss over one batch of a time, to leverage parallelism. Keep in mind that `torch.nn.CrossEntropyLoss()` itself already averages the loss.

4. **Implement the gradient update step** in the function `train()`, by using the `optimizer`, the cross-entropy `criterion`, and the `backward()` function on the loss. Once the gradient is computed, mitigate gradient explosion by clipping the gradient to have a maximum norm of 1.0. For this, the PyTorch function `clip_grad_norm_()` may come handy.

5. Use the trained model to **compute the softmax probabilities** for predicting the next token, in the function `use()`.

6. **Generate the next token** using either greedy or multinomial sampling, in the function `generate_next_token()`.

7. **Compute perplexity on a test corpus**, in the function `evaluate()`.

Your submission should contain the trained model(s) saved in `code/lm_model.pt` (or similar), as well as a complete set of outputs from your model as printed during training and evaluation, saved into `code/traces.txt`. Your outputs should reflect the 4 main modes of use of the program:

8. `-train`: Train the model on `tiny/tiny.train.tokens.4M`.

9. `-use`: Use the trained model in interactive mode to predict the next token for a sequence of tokens from the user. At a minimum, show the predicted token for the following sequences: *'there was a girl '*, *'one day, a bird'*, *'there once was a boys and'*, *'one day , a'*, *'once upon a time , an elf '*, *'once upon a time'*.

10. `-prompt`: Use the trained model to generate a complete sequence of tokens conditioned on a prompt from the user. At a minimum, you should show the greedy completion of the prompt *once upon a time*, as well as 5 different sample continuations obtained with multinomial sampling.

    (a) Once you generate a continuation, use the entire text (prompt + continuation) as a new prompt and ask the LM to generate a new continuation. Do this repeatedly until either <eos> is generated, or the model enters a repeating pattern.

11. `-evaluate`: Evaluate the trained model on both `data/tiny/tiny.valid.tokens.200K` and `data/tiny/tiny.train.tokens.4M` and report the two perplexity numbers.

12. Write a **report** summarizing experimental results and insights. Describe all your work in sufficient detail, such that others can replciate yoru findings.

When you evaluate the top outputs, try to explain any unexpected results or mistakes, by referring to properties of the algorithm or the data. Whenever you formulate a hypothesis, it is important that it is supported empirically. As such, feel free to train and evaluate models that go beyond the setting described above. All the required results and analyses should be included in an appropriately edited homework report, using proper indentation, section titles, and formatting. If you include formulas, make sure that you use appropriate formatting of equations. The PDF of the report `report.pdf` should be submitted on Canvas, together with the code in the folder above. The assignment will be graded based on **both** the code and the quality of the report.

3 Bonus exercises

Any non-trivial and insightful extra work can be worth bonus points. For example:

1. Implement learned positional embeddings and compare their performance vs. the original since/cosine positional encodings.

2. Compare the performance of the *prenorm* vs. *postnorm* versions of the Transformer.

3. Tune the various hyper-parameters in order to optimize loss or perplexity on the validation data, e.g., the learning rate, the number of layers, the dimensionality `d_model`, the number of attention heads, the number of epochs. You may also want to try early stopping with inertia.

4. Compare the results obtained with your TransformerLM implementation vs. an implementation that uses the PyTorch Transformer class.

4 Submission

Electronically submit on Canvas a `hw05.zip` file that contains the hw05 folder in which your code is in the required files, as well as the `report.pdf` and `traces.txt` files.

On a Linux system, creating the archive can be done using the command:
`> zip -r hw05.zip hw05`.

Please observe the following when handing in homework:

1. Structure, indent, and format your code well.
2. Use adequate comments, both block and in-line to document your code.
3. Verify your code runs correctly when used in the directory structure shown above. We will not debug your code.
4. For your report, it is recommended to use an editor such as Overleaf for Latex, Word, or Jupyter-Notebook that allows editing and proper formatting of equations, plots, and tables with results.
5. Verify that your Canvas submission contains the correct files by downloading and unzipping it after posting on Canvas.