ITCS 6101/8101 Homework 4: Implementation (100 + 40 points)

March 5, 2026

1 Overview

In this assignment, you are given a code base in Python that implements a multi-layer LSTM-based Language Model (LSTM-LM) using pre-trained GloVe embeddings. 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.

```
hw04/
   report.pdf
   code/
     lstmlm.py
     lm_model.pt
     traces.txt
   data/
     glove.6B/
         glove.6B.50d.txt + glove.6B.100d.txt
         glove.6B.200d.txt + glove.6B.300d.txt
     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 LSTM-LM architectures contains H LSTM layers, each with a hidden state of size N, and one softmax output layer that produces as output a probability distribution over all the tokens in the vocabulary. Pre-trained GloVe embeddings of various dimensions for a pre-defined set of 400K tokens are provided in the text files from the `data/glove.6B/` folder.

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 `-H` (the number of LSTM layers), `-N` (the size of the LSTM hidden state and cell state), `-E` (the number of training epochs), `-B` (the minibatch size, i.e., the number of stories in a minibatch), `-VSize` (the vocabulary size), `-a` (whether to use dot-product attention), `-greedy` (whether to use greedy vs. multinomial sampling), as well as paths for various types of input files.

To train, tune, and evaluate the LSTM-LM 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 the multi-layer LSTM in the function `LSTMLM.forward()`. For efficient parallel processing of the multiple stories in a mini-batch, it is strongly recommended that you transformed the padded tensor representation into a `torch.nn.utils.rnn.PackedSequence` representation of the minibatch tokens to feed as input to the `self.lstm` object. The output of LSTM inference can then be turned back into a padded tensor representation to do further processing, e.g., to compute and add attention. Relevant functions are `pad_sequence()`, `pack_padded_sequence()` and `pad_packed_sequence()` from the module `torch.nn.utils.rnn`.

   (a) Implement a simple **dot-product attention** mechanism over strictly previous LSTM hidden states. To keep the number of parameters comparable, when attention is enabled the LSTM hidden state and cell state sizes are reduced to N/2 (Section 2.1, mandatory for 8101, bonus for 6101).

2. **Create a minibatch of training examples** in the function `collate_stories()`. This will create a batch of `input` sequences of tokens and the corresponding batch of `target` labels (a label is the next word for the corresponding position in the input sequence). The input batch should pe padded with 0's up to the maximum length of input sequences in that batch, whereas the `target` batch should be padded with label -100 (which is ignored by the CE loss criterion). The PyTorch function `pad_sequence()` will come handy here. Also store the actual lengths in the `lengths` tensor.

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()`.

   (a) Add a new command line parameter `-fpenalty` specifying a constant $fp \in [-2.0, +2.0]$ to be used as a frequency penalty when computing the token probabilities at test time (Section 2.2, mandatory for 8101, bonus for 6101).

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 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.

2.1 Self Attention (*) 20 points

When the optional `-a` flag is provided during training, the LSTM LM model should also implement a simple dot-product attention mechanism over the previous hidden states. The design is as follows:

1. To keep the number of parameters comparable to the non-attention model, the LSTM hidden state and cell state sizes are reduced to N/2 (half of the specified N parameter).

2. At each time step t, after the LSTM produces its output h_t (dimension N/2), a causal dot-product attention is computed over strictly previous positions:

   - The query is $\mathbf{h}_t$.
   - The keys and values are all hidden states $\mathbf{h}_1, \mathbf{h}_2, ..., \mathbf{h}_{t-1}$ from the previous positions.
   - The attention score between positions i and j is computed as:

   $$e(i,j) = \frac{\mathbf{h}_i^T \mathbf{h}_j}{\sqrt{N/2}} \tag{1}$$

     where $N/2$ is the dimension of the hidden state, and the scaling factor $\sqrt{N/2}$ improves numerical stability.
   - It is highly recommended to compute attention score in parallel over all positions, by properly masking future tokens or padded tokens:
     - A causal mask can be applied so that positions cannot attend to current or future positions (scores for $j \geq i$ are set to $-\infty$).
     - A padding mask can also be applied so that padded positions are not attended to.
   - Softmax is applied to the scores to obtain attention weights. At position 0 (the first position, typically <bos>), since there are no previous positions to attend to, the attention context vector should be set to zero.
   - The attention context vector $\mathbf{c}_t$ is the weighted sum of the previous hidden states using the corresponding attention weights.

3. The attention context vector $\mathbf{c}_t$ (dimension N/2) is concatenated with the LSTM output $\mathbf{h}_t$ (dimension N/2) to form a combined vector of dimension N.

4. This combined vector of size N is fed to the output (softmax) layer, which has the same input dimension N as in the non-attention case.

2.2 Frequency Penalty (*) 20 points

Use a frequency penalty $fp \in [-2.0, 2.0]$, by default set at 0 (no penalty), to discourage repeating tokens in the autoregressive output from the LM. Let s be the sequence of tokens so far (starting with <bos>) at timestep $t-1$. Let $z[w]$ be the logit score computed by the LSTM LM for an arbitrary token $w \in V$, and let $c(s,w)$ be how many times the token

w already appeared in the input sequence s. Then the frequency penalty factor is used to modify the logit score for token w as shown below:

$$z[w] \longleftarrow z[w] - fp \times c(s, w) \tag{2}$$

This logit score update is done for all the tokens in the vocabulary, before computing the softmax probability distribution for timestep t. Thus, if $fp > 0$, tokens that appeared before will have their logit score decreased, hence a lower likelihood of being sampled at time t.

3 Bonus exercises

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

1. Update your FFLM implementation to accommodate the <bos> and <eos> tokens by providing their embeddings, as learned by the LSTM LM. Then compare the FFLM and the LSTM-LM in terms of their perplexity on the test data. To make comparisons fair, design them such that they have roughly the same number of parameters, and tune them accordingly on the validation data.

2. Tune the various hyper-parameters in order to optimize loss or perplexity on the validation data, e.g., the learning rate, the number of hidden layers, the number of neurons, the optimal embedding size, the number of K words to use in the context, the number of epochs. You may also want to try early stopping with inertia.

3. Currently, the embeddings are frozen. Try fine-tuning the embeddings while training the LSTM-LM parameters, perhaps using a separate lower learning rate.

4. Compare dot-product attention with other attention variants, e.g., the multiplicative (bilinear) attention and additive attention shown on slide 53 in the Attention lecture.

5. Run BPE on the training dataset and learn a vocabulary of up to 10,000 tokens. Randomly initialize their embeddings, and then train the LSTM-LM jointly with the token embeddings, using a larger version of the TinyStories dataset. Contact me if you need a tokenized version, otherwise you can get the raw version from from HuggingFace.

6. Compare the results obtained with the LSTM implementation vs. a vanilla RNN or a GRU implementation.

7. Investigate whether makign the LSTM biderectional helps.

8. Investigate whether adding positional embeddings (cosine and sine, as in the original Transformer paper) makes the LM better.

9. The model size is dominated by the softmax parameters. A widely used technique to reduce the number of parameters is to use the embedding matrix as the softmax parameters. This requires setting the last hidden layer to have the same size as the embeddings, and removing bias parameters. Implement and evaluate this approach – you would need to tie the softmax parameters and the embeddings and allow the pre-trained embeddings to be fine-tuned.

4 Submission

Electronically submit on Canvas a `hw04.zip` file that contains the hw03 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 hw04.zip hw04`.

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.