Advanced Methods in Text Analytics
Exercise 4: Language Models - Part 2

Daniel Ruffinelli

FSS 2025

The main goal of this exercise is to implement the fundamental components of language models using PyTorch, so that when you use higher-level libraries, you have an intuition of what is happening underneath. To this end, we provide a Jupyter notebook with incomplete implementations of three types of language models, from n-grams to basic deep learning models. You will be asked to complete the missing code throughout the tasks, which were designed to make you think about how different components interact with each other, how the data is shaped/shifted/transformed throughout the pipeline, and how models are trained from data to make useful predictions and be used in applications. So, follow the instructions in each question plus the documentation in the code when implementing tasks in this exercise.

1 N-Gram Language Models

To implement this model, we make use of a simple tokenizer defined in the function *tokenize*. It first separates punctuation marks by whitespace, and then creates tokens by splitting the string with whitespaces.

(a) Complete the function *compute_ngrams*, which we will use to count n-grams. To make predictions with n-gram models, we compute the probability of a sequence of n words and compare it against the probability of the same sequence minus the last word, i.e. the $n-1$ words sequence. Thus, instead of representing n-grams as n-size tuples, we will represent them as tuples of the form *(context, next_word)*, where *context* is the first $n-1$ tokens in a given sequence, and *next_word* is the last token.

(b) We implement n-gram models with the *NGramModel* class. Following the format of n-grams defined above, this model keeps track of *(context, next_word)* pairs and an n-gram counter. Complete the functions *update* and *next_word_prob* by following the documentation in the code. In this class, we make use of defaultdicts in Python, but feel free to replace them with standard dictionaries if you prefer.

(c) Complete the function *predict_next_word* used to predict the next word given a prompt and an n-gram model implemented with the *NGramModel* class above. For sampling the next word, use the probabilities of each possible next word given by the model. Use the model's "context to next word" dictionary and its *next_word_prob* function to construct such a distribution.

(d) Let's test the code we worked with in the questions above. First, we load our Shakespeare dataset, which is nothing more than all of Shakespeare's work, obtained from Project Gutenberg. Then, we create a trigram model, count the n-grams in the tokenized

corpus, and then check the most common ngrams. Do the most common n-grams make sense? Why do they look like that?

(e) Let's generate text with our model. Complete the function *generate_text* and then test it with some prompts. Try different ones based on the n-grams counted in the corpus as prompts. Does the output make sense? Does it change a lot given the same input?

(f) Let's use stronger models now. Create a fourgram and fivegram model and, as before, count n-grams and then check for the most common ones in the corpus. Then, use some of those n-grams as prompts to generate text with these models. Are they better than the trigram model?

(g) Complete the function *compute_likelihood*, which we use to compute the likelihood and perplexity of the validation data. Note that we can compute the perplexity directly from the likelihood. For details, see here. Is the likelihood low? Is the perplexity low? How can you tell?

2 Language Models with Fully-Connected Neural Networks

In this task, we use PyTorch (official tutorials here) to implement neural LMs. Specifically, we implement a language model similar to the one proposed by Bengio et al. For simplicity, we will not implement the optional skip connection between input and output, but we will implement a more flexible model that can have multiple hidden layers of different sizes. This should allow us to test different architectures. But first, we need to load up PyTorch and preprocess our dataset. These models are different from ngram models, so we prepare our data for this, including the creation of our vocabulary.

(a) We implement our LMs with fully-connected neural networks using the *NeuralLM* class. Read the code and complete the *forward* function, which computes the forward pass of the network. In this function, do not use softmax! We will do that during training. **Hint:** use view to get the necessary embeddings.

(b) A dataloader is an important component of using PyTorch, as in deep learning, loading data can be expensive. The *dataloader* is in charge of that, but it needs two things: (i) a dataset, and (ii) a *collate* function, in charge of turning raw data into training examples, i.e. an (input, target) pair. Complete the class *SelfSupervisedTextDataset*, which is a custom PyTorch dataset, following the instructions in the code documentation. Then, complete the *collate_fn*, which takes a batch as input during training, and processes it to create the training examples and corresponding targets. This is where self-supervision takes place!

(c) We now need to write a training loop, a fundamental part of implementing deep learning models. This is the loop that takes place when you call functions such as *model.fit()* in higher-level libraries like *scikit-learn*. Complete the function *train* (ignore the *rnn* flag for now). To implement the training loop, you need to iterate over the epochs, and over batches given by the dataloader. For each batch, you need to compute the forward pass, the backward pass, and update the parameters of your model using the optimizer. We use the cross entropy loss, which expects logits (i.e. it handles the softmax computation), and the Adagrad optimizer. Also, and perhaps more specific to PyTorch, gradients are accumulated over different batches by default, so you need to zero them after each batch, using the *optimizer.zero_grad()* function.

(d) To compute perplexity on the validation data, we also need a loop for evaluation. Complete the function *evaluate* (ignore the *rnn* flag for now). Use the same likelihood as computed during training to then obtain the perplexity. The function should print the perplexity of the data at the end. As with the training loop, you need to iterate over the batches in the dataloader, but there are no epochs this time. In addition, in order to compute the perplexity over the entire data, we do not reduce the loss per examples but sum it. Thus, you also need to count the number of examples so we can then compute average likelihood and with it, perplexity. (Note that you should also do this during training if you want to see perplexity over the entire training set.) How is the perplexity on validation data? Can you compare it to the n-gram model from before?

(e) Try different architectures to see their impact on validation perplexity. You can change the number and size of hidden layers, but be careful with overfitting. Consider including different forms of regularization, such as dropout.

3 Language Models with RNNs

In this task, we focus on language models based on RNNs.

(a) We use the class *RNNLM* to implement an RNN-based language model. Read the code and understand what the function *init_hidden* does. We will use it later for training. Then, complete the *forward* function. Make sure you read PyTorch documentation on RNNs to understand the size of the input that the *rnn* object expects. *(input, target)* pairs are constructed differently from a given input sequence. To this end, complete the function *rnn_collate_fn* accordingly. Second, we need to modify the training loop from the neural LM used before so it can *also* train an RNN. You may use the *rnn* flag for this, but if you prefer, create an entirely new training loop for this RNN in a new cell. Again, read the RNN documentation to understand both the expected input to the RNN and the output it produces. Is it more expensive to train than the neural model from before? Why?

(b) As before, we need to modify our evaluation loop so it supports RNNs. Use the *rnn* flag and modify the *evaluation* function in the neural LM from before to allow evaluation of RNN-based LMs. How does the perplexity compare to the models we tested before?

(c) Let's now generate text with our RNN. For that, we first create a reverse dictionary that maps token IDs to the actual tokens. Complete the function *generate_text_with_rnn* by sampling the next word using the distribution over the vocabulary provided by the model. Then use this function to generate new text using any prompt you want, so long as it uses the vocabulary our model understands. How does the generated text compare to the output of the n-gram-based model from Task 1? Can you think of a reason that would explain the differences you observe in quality? Test the generation of text using different temperatures. Does it get better or worse? When? Why do you think that is?

(d) To see the impact of different sampling methods on the generated text, we generalize the function *generate_text_with_rnn* so it takes a *sampling_fn* parameter. Complete the function *topk_sampling* so it returns a single sample from a given set of logits and a value of k. Note that the new version of *generate_text_with_rnn* no longer computes softmax, because each *sampling_fn* is now responsible for that. For that reason, the function *topk_sampling* now takes the temperature parameter. Test the impact of different values of k on the generated text. Make sure to also try different values of k with different

temperatures. Did you find a combination of k and temperature that generates text similar to or better than the n-gram model from Task 1?

(e) Test the impact of using different architectures on validation perplexity and text generation. You may try different sizes of hidden layers, or even stack them. You may also implement an RNN with LSTMs, which would require minimal changes to your training and evaluation code (you need to manage the cell state in addition to the hidden state).