Exercise 6

Deep Learning Lab

October 31, 2022

1 Language Modeling with RNNs

In this exercise, you will implement a character-level language model using recurrent neural networks (RNNs). As has been presented in the lecture (Sec. 3.2), an RNN language model is trained to predict texts from left to right one token (in our case one character) at a time. While the primary role of a language model is to evaluate the "correctness" of any sentence by computing its probability, such a model can be also used to *generate* texts recurrently by generating one character at a time according to its output distribution and feeding it back as an input to generate the next character.

1.1 Text data

Many public domain books are available in text format from Project Gutenberg. Download "Aesop's Fables (A Version for Young Readers)" from here. For simplicity, we'll assume that no pre-processing is needed.

1. Check important properties of the data: number of unique characters (vocabulary size), number of running characters in the file, number of lines in the file, and any other interesting observations about the structure of the text (e.g. usage of capitalized letters, contractions, paragraphs, line lengths, etc).

1.2 Batch construction

We consider a very long string representing the whole book (the text file you downloaded). Back-propagating gradients through an RNN which is enrolled as many time steps as there are characters in that long string (backpropagation through time; BPTT) will require too much memory. Instead, the string must be broken down into smaller text chunks. Chunking should be done such that the first token for the current chunk is the last token from the previous chunk. The first token of the chunk is the first input token to be fed to the language model, while the last token of the chunk is the last target token to be predicted by the model within a given chunk. We'll then train the model by *truncated backpropagation through time* (which was also covered in the lecture, Sec. 3.2), i.e.:

- We limit the span of backpropagation to be within one chunk. The length of the chunk is thus the BPTT span.

- We initialize the hidden state of the RNN at the beginning of the chunk by the last state from the previous chunk. We thus can not randomly shuffle chunks, since the RNN states need to be carried between consecutive text chunks.

To improve efficiency, we train on a batch of text chunks such that multiple chunks are processed in parallel. **The implementation for reading a raw text file and creating a batch generator which is compatible with the description above his available on iCorsi. No coding is thus needed here.** Read the provided code.

1.3 Model and Training

1. **Implement** an RNN language model with an input embedding layer, multiple RNN layers using nn.RNN, and a final softmax classification layer to predict the next character.

2. Our goal is to generate texts using the trained RNN language model. To be more specific, we will provide a *prompt* (i.e. a segment/beginning of a text) to the language model, and let the model *complete* the text (for a length that you specify). **Implement** a *greedy decoding algorithm* for such a completion, i.e., at each step, take the character with the highest probability according to the model, and use it as the input to the model in the next step. We recommend you to implement this **not** as a part of the `forward` function of the model, but as a separate function which calls the `forward` function. Remember: hints were presented in the lecture/helper code.

3. **Implement** the code for training. As has been described above, the hidden states of the recurrent layers must be passed from one batch to the next, while not allowing the error signal to propagate across batches during training. For that you can apply `detach` function to the hidden state tensors. Make sure that your code allows you to monitor the following measures for every n training steps (with a reasonable choice of n):

   - The *perplexity* of your model on the training data (on the batch level).
   - The model's text generation ability, by decoding always from the same prompt of your choice (e.g. "Dogs like best to" or the beginning of a story in the training set).

   The perplexity of a language model p for a text w_1^N (with a start token w_0) is defined as:

$$\text{Perplexity} = \left(\prod_{n=1}^{N} p(w_n|w_0^{n-1})\right)^{-\frac{1}{N}} = \exp\left(-\frac{1}{N}\sum_{n=1}^{N}\log p(w_n|w_0^{n-1})\right).$$

   From this equation, you should recognize that the perplexity can be directly computed by simply applying the exponential to the cross-entropy loss in PyTorch with the default parameters.

4. Train an RNN language model. We recommend the following hyper-parameters:

   - an input embedding layer with a dimension of 64
   - one RNN layer with a dimension of 2048
   - a BPTT span of 64 characters
   - the Adam optimizer with a learning rate of $5e^{-4}$.
   - with a gradient clipping with a clipping threshold of 1.0. This can be done by adding the following line between the gradient computation (typically `loss.backward()`) and the parameter update (typically `optimizer.step()`):
     `torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)` assuming that you defined your PyTorch model as `model`.

   We do not make use of any validation data: you can consider this model to be well trained when its training perplexity is below 1.8. You should be able to achieve this performance within less than 10 minutes using a GPU. Report the evolution of both perplexity and text generation quality.