Deep Learning PhD course: Hand-in assignment 3

Antônio Horta Ribeiro

May 17, 2021

Due: 10th of June 2021, 23:59

Language model

For this task the goal is to implement a language model. A language model is a model that gives the probability of a setence or phrase to be used in a given language. More, specifically, you will build a *word-level* language model for the *English* language. For this type of model, the "words" are the minimal units considered by the model.

As an example, consider a sentence with 4 words represented by (w_1, w_2, w_3, w_4). To compute the probability of this sentence, it is possible to break down the probabilities in the following way:

$$P(w_1, w_2, w_3, w_4) = P(w_4|w_1, w_2, w_3)P(w_3|w_1, w_2)P(w_2|w_1)P(w_1) \tag{1}$$

You will treat this task as a sequence problem that, given the past words that have appeared, computes the probability distribution of the current word.

> REMARK: We recommend you to use a GPU acceleration for the tasks in this hand-in assignment. If you do not have direct access to a GPU, we would like to point that Google Colab offer GPUs and TPUs access for free. The computational power of the free resources offered there is enough to run all the experiments in this assignment. You can find more information about it in: https://colab.research.google.com/. The next tutorials show how to use TPUs and GPUs on it:
>
> 1. https://colab.research.google.com/notebooks/gpu.ipynb
> 2. https://colab.research.google.com/notebooks/tpu.ipynb
>
> They use Tensorflow, but the procedure for using PyTorch or other frameworks is the same.

> REMARK: PyTorch and Tensorflow both have examples of language model implementations easily available:
>
> 1. PyTorch has one official example of word-level language model (as in this exercise): https://github.com/pytorch/examples/tree/master/word_language_model
> 2. Tensorflow has one official example of char-level language model (where the basic units are the characteres): https://www.tensorflow.org/tutorials/text/text_generation
>
> Taking a look at these examples might be helpful to complete the tasks bellow.

The dataset

For this assignment we use the *PennTreebank (PTB) dataset* [1]. Which is a corpus consisting of collection of stories from the Wall Street Journal (WSJ).

Exercise 1 *In this exercise you will download and familiarize yourself with the dataset.*

1. ***Download the dataset.*** *The dataset can be found in multiple sources and here we ask you to download the PTB dataset already preprocessed for language modeling. You can download it from:*

   https://deepai.org/dataset/penn-treebank.

   After extracting the file you should obtain, among others, the files `ptb.train.txt`, `ptb.valid.txt`, `ptb.test.txt`, *which should be used respectively for training, validating (i.e. tuning the hyper parameters, early stopping and so on) and testing (assess the performance on an unseen dataset).*

2. ***Print a couple of example phrases*** *from this dataset. Include 5 of these sentences in your report.*

   You might noticed that the dataset has already been preprocessed:

   - *the text does contain the special term* `"<unk>"`*. This special term was added during the preprocessing and replaces rare words that do not occur frequently in the dataset.*
   - *there is no punctuation (exception in abbreviations)*
   - *words are in lower case*
   - *most numbers have been replaced by* `"N"`
   - *spaces have been included before some terms such as* `"'ll"`, `"'s"`, `"n't"`

3. ***Add*** `"<eos>"`*. Another common special term is the* `"<eos>"` *term. This term is commonly used for representing the end of the sentence. This is a step that has not been included in the preprocessing of the dataset. Hence, in your pipeline, you should add one step that adds this special character at each line break. (At the end of each sentence)*

4. ***Assessing the size of the splits.*** *What is the total number of words in each of the splits* `train`, `valid` *and* `test`*?*

   We consider every string contained in the file that is separated either by a space or by a line break to be a word. Consider the special characters `"<eos>"` *and* `"<unk>"` *in your count.*

5. ***Build a dictionary.*** *Consider a* ***dictionary*** *to be the set of all the* ***unique*** *words that appear in the text (in the three splits combined), how many words there are in this dictionary? Consider the special characters* `"<eos>"` *and* `"<unk>"` *in your count.*

   Now, assign one integer for each word in the dictionary:

   $$\{..., \text{'years'}: 28, \text{'old'}: 29, \text{'will'}: 30, \text{'join'}: 31, \text{'the'}: 32, ...\}$$

   You should implement a function that convert a piece of PTB text into a sequence of integers. For instance, it attributes one number to each word:

   $$\overbrace{\textit{numerous}}^{2882}\ \overbrace{\textit{injuries}}^{5036}\ \overbrace{\textit{were}}^{113}\ \overbrace{\textit{reported}}^{83}\ \overbrace{<\textit{eos}>}^{24} \longrightarrow [2882, 5036, 113, 83, 24]$$

A simple language model

To start with, you will consider an Elman recurrent neural network (RNN) model. The probabilistic formulation from Eq. (1) can be approximated by a sequential model by considering the hidden state x_k as a compact representation of the previous words $w_{k-1}, w_{k-2}, ...$ The state is then propagated by the recurrent unit:

$$x_{k+1} = \sigma(W_{xx}x_k + W_{wx}w_k + b_x), \tag{2}$$

where σ is the activation function. And where w_k is the word embedding, obtained as in the following expression:

$$w_k = W_{iw} \text{ one-hot-encoding}(I_k). \tag{3}$$

Here I_k the integer representation of the word (as defined in the previous exercise) at position k. In this case, we use a one-hot-encode that converts I_k in one vector with zeros in all positions except in the position given by an integer and this vector is them multiplied by a matrix W_{iw}.

In turn, let P be a vector with the size of the dictionary described above containing the probability of each one of the corresponding words at position k of the sentence

$$P_k = \text{softmax}(W_{xp}x_k + b_p) \tag{4}$$

The matrices W_{xx}, W_{wx}, W_{iw} and W_{xp} and the vectors b_x and b_p are the parameters of this recurrent neural network model.

Exercise 2 *Train and evaluate the Elman recurrent neural network model.*

1. ***Implement the model above.*** *Using the deep-learning framework of your choice implement the model above. The model should be have three basic components: the embedding layer, described in Eq. 3, the recurrent layer (i.e., Eq 2) and the softmax layer (i.e., Eq 4). The dimension of the embedding vector w and the dimension of the hidden state x are design parameters that you should choose.*

REMARK: *Efficient implementations of embedding layers and recurrent layers are available in all major deep-learning frameworks. We recommend you to use those layers instead of implementing them from scratch. For instance, in Pytorch, we have all the layers mentioned above:*

- *Embeding layer:*
  https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html
- *Elman RNN:*
  https://pytorch.org/docs/stable/generated/torch.nn.RNN.html

2. ***Implement a data-loading routine.*** *Using the dictionary from Exercise 1 you should be able to obtain vectors of integers for training, validation and test data, respectively. For instance, let v_{train} be a vector of integers with dimension n_{train}.*

$$v_{train} = [\cdots, 2882, 5036, 113, 83, 24, \cdots] \tag{5}$$

As in other deep learning methods you do not feed all this data at once to your model. Let b be the batch size and l_{seq} the sequence length. Divide the entire sequence into smaller sequences of length l_{seq} and at each iteration of your training or evaluation routine you should load b sequences of this length.

3. ***Train the model and generate learning curves.*** *Train the model above on the training dataset included in the PTB dataset using stochastic gradient descent with the cross-entropy loss.*

   Also include the learning curves in your report. For each epoch, the learning curves should contain the average loss of the training and the validation set.

REMARK: *Exploding gradients are quite a severe problem in recurrent neural networks and it is not uncommon to have big jumps in the loss function. Sometimes the values might be so big that you may observe 'NaN' or 'Inf' values. If you observe this in your model reduce the learning rate. It should be possible to find a sufficiently small learning rate for which this do not happen. Later we will see another solution that might help as well.*

4. ***Evaluate the model and report test performance.*** *Evaluate the best model obtained during training using the test set and report the perplexity in the test set.*

   *The **perplexity** is a common metric used for evaluating language models. It evaluates how well a probability distribution or probability model predicts a sample. In our case, the perplexity is just the exponential of the average cross entropy loss on the test data,*

$$\exp\left(\frac{1}{N_{\text{test}}}\sum_{i=1}^{N_{\text{test}}} l_i\right)$$

where l_i is the cross-entropy loss of prediction i and N_{test} is the size of the test-set.

REMARK: *Word-level language modeling in the PTB dataset is often used as benchmark for new sequence-to-sequence models. You can find the performance (also reported in terms of the model perplexity) of state-of-the-art models in:*
https://paperswithcode.com/sota/language-modelling-on-penn-treebank-word
Some of the models have been trained on additional datasets, but you can also find some models that are directly comparable to the ones you are training. Some traditional recurrent models are listed in the list. For instance, a well-tunned LSTM has a perplexity 78.93 and a well tunned GRU model has perplexity of 92.48, as reported in [2]. In [2], the authors also trained an Elman RNN with perplexity 114.5. For the model you trained now you should still expect a performance worse then that, but in exercise 3 we give suggestion in how to improve performance that might help you get a more competitive performance.

5. ***Generate sample phrases.*** *The phrase should be generated recursively: start with a prompt sentence sentence, say a three word sentence '$w_1 w_2 w_3$', and an initial hidden state x_0, which can be, for instance, picked randomly. Using these three words as input it is possible to propagate the state three times forward and compute x_4 and a probability vector for the 4-th word P_4. You can use this probability vector to pick the 4-th word that should continue the sentence: $\hat{w}_4$. You can them propagate the state and use it together with this word you pick to generate the probability vector for the 5-th word. And so on.*

   *There are multiple strategies for picking a word $\hat{w}_k$ from a probability distribution P_k. We ask you to implement **one** of the strategies bellow:*

(a) ***Gready**: Always pick the word with the highest probability (i.e. `argmax`).*

(b) ***Sampling**: Sample from the dictionary, with each word picked with the probability P_k given by the language model*

(c) ***Beam search**: The greedy approach does not always result in the final sequence with the highest overall probability. A beam search keeps track of n variants at each step to avoid being led astray by local maxima. (One lecture about it is: https://www.youtube.com/watch?v=RLWuzLLSIgw)*

Improving the recurrent model

There are some tricks that might improve your model. Here we give some directions

Exercise 3 *Improve and tune your model. For each of the items bellow you should include in your report: the set of hyperparameters used; the learning curves with the average loss in the training set and validation set. The perplexity performance on test; and a couple of generated phrases.*

1. ***Gradient clipping.** As mentioned in the Remark in Item 3 of the previous exercise, RNNs are often affected by exploding gradients, which in turn may cause big jumps in the cost function. One easy to implement workaround is to use gradient clipping. For instance, if you are using gradient descent, say, the update of a given parameter vector θ is:*

$$\theta_k = \theta_{k-1} - \gamma \nabla J(\theta_{k-1}) \tag{6}$$

for a learning rate γ and a cost J. Gradient clipping consist of defining an operation

$$clip_M(\theta) = \begin{cases} \theta & \text{if } \|\theta\| \leq M \\ M\frac{\theta}{\|\theta\|} & \text{if } \|\theta\| \geq M \end{cases} \tag{7}$$

And using the update rule:

$$\theta_k = \theta_{k-1} - \gamma clip_M\left(\nabla J(\theta_{k-1})\right) \tag{8}$$

Please include gradient clipping and try to retrain your model with it. Using larger learning rate values should become easier with this trick.

2. ***Improving the flow of information.** In Exercise 2, a dataloading procedure with a fixed a sequence length was described. There are different choices how to choose the first hidden state x_0 for each of these sequences. A naive implementation would just start with x_0 as zero (or other arbitrary value). But there is a better option: evaluating the sequences consecutively and, at the end of the sequence, passing the last hidden state and use it as the start for the next sequence. Implement this method and retrain and re-evaluate your model.*

3. ***Gated units.** There are some popular alternatives to the Elman recurrent neural network. The LSTM unit is one of the most popular. Replace the Elman RNN unit by the LSTM.*

4. ***Tune the parameters and experiment different design choices.** Explain and present at least two additional modifications. Examples include using a different learning rate scheduler, changing the optimization algorithm, changing the sizes, using a bi-directional recurrent model, use an alternative Gated Unit, use a recurrent model with more then one layer, and so on.*

Using a Transfomer (Optional)

*Here we leave the implementation of the transformer as an interesting possibility to be used in exercise 3 item 4. The implementation of the transformer might be more time consuming. Hence, it is **entirely optional** and is up to you whether you want to implement it, or choose a different modification.*

The Transformer, originally proposed in [3] in the context of language translation, has became one of the most popular choices for language modeling. For language translation two components are usually used: a transformer encoder which encodes the input sentence in one language and a decoder that decodes it to the target language. In the context of language modeling, we only use the transformer encoder.

The transformer encoder, as it was proposed, is a purely feedforward model (i.e. there is no hidden state propagated by recursion). Hence, given an input sequence of vectors (w_1, w_2, w_3, w_4) the transformer encoder outputs another sequence of vectors (y_1, y_2, y_3, y_4). It consists of layers that alternate self-attention layers and fully connected layers.

There are two important components that are important to keep in mind when implementing the transformer. The first is the **positional encoder** which is an embedding used to store the information about the positions. In this sense, the input vectors w_k are usually summed with additional entries that contain information about the position. This is needed because, unlike RNNs that inherently take the order of word into account, the transformer doesn't have any sense of position/order for each word, hence the positional encoder is used to fix this.

A second element is the **mask**. In a transformer, if you try to predict the probability of the word and use the entire phrase as input, the model does have access to all the words in the sentence, including the one that would be predicted in the first place. Hence, a mask in the input is needed.

> REMARK: There are plenty of resources online about the transformer layer. One of our personal favorites are these two blog posts:
>
> - http://jalammar.github.io/illustrated-transformer/
> - http://jalammar.github.io/illustrated-bert/
>
> Tensorflow does have one example where the transformer encoder and decoder are implemented from scratch: (for a different task than we are considering here!)
>
> - https://www.tensorflow.org/tutorials/text/transformer
>
> And, PyTorch (since version 1.2) does have a native implementation of the transformer. For language modeling, we usually use only the Transformer "encoder". This layer is, for instance, available in:
>
> - https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoderLayer.html
> - https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoder.html
>
> The PyTorch language model example does include the usage of a transformer and might also be useful: https://github.com/pytorch/examples/tree/master/word_language_model.

As a possibility of modification for exercise 3, item 4: **Train and evaluate a transformer encoder** for word-level language modeling in the PTB dataset. As in the other items, include in your report: the set of hyperparameters used; the learning curves with the average loss in the training set and validation set; the perplexity performance on the test set; and a couple of generated phrases.

The submission

Your submission is supposed to be a proper free-standing report which one can read without the support of these assignment instructions. Hence, the report should not be just the "solutions" to the task description. It should include (briefly) the problem formulation and how you have chosen to solve and implement it, for example with text supported with the relevant equations and/or pseudo code. Do make sure that all exercises are addressed and that all requested plots are included and have proper figure captions, legends and axis labels. You should submit two files, one pdf-file with the report and one zip-file with the code. Do not include the pdf in the zip-file.

References

[1] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[2] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling. *arXiv:1803.01271*, 2018.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems 30*, pages 5998–6008, 2017.