Deep Learning Lab
Assignment 3

Pietro Miotti

December 9, 2021

1 Language Modeling with RNNs

1.1 Text Data

1.1.1 Properties

Vocabulary Size: 107
Different Characters in the text: 177517
Total number of lines: 5033
Maximum line-lenght: 80
Minimum line-lenght: 1
Number of empty lines: 1868
Usage of capital letters: yes
Usage of paragraphs: yes

1.1.2 Preprocessing

The first thing I would do as a preprocessing would be normalize the text (convert all the words to lower case or either all upper case), such that there is no duality between ids of upper and lower case.
Another thing I would do would be remove long paragraphs and also convert the numbers from text to real numbers.

1.2 Batch construction

1.2.1

There is an if statement because maybe the given character is not yet present in the vocabulary and need to be added (if extend_vocab = True) or set as unknown.

1.2.2 What are the keys and values for each of the dictionaries

- id_to_string: the keys are the ids, items are the corresponding characters
- string_to_id: the keys are the characters, the ids are the corresponding ids.

1.2.3

By calling __len()__ (in LongTextData) we obtain the total number of characters present in the given book.

1.2.4

By calling __len()__ (in ChunkedTextData) we obtain the total number of batches for the given book (which is computed by $\frac{LongTextData.\_\_len()\_\_}{batch\_size * bptt\_len}$ considering that some sequences will also be padded.

1.2.5

- *padded = input_data.data.new_full((segment_len * bsz,), pad_id)* : this line creates a new tensor with lenght=segment_len * bsz with all elements initialized as pad_id which in our case is 0 (the full method makes sure that types and device of the new tensor are the same of the input_data tensor).
- *padded[:text_len] = input_data.data*: since padded is bigger than the original input_data, we now fill all values from 0 to text_len-1 with the content of input_data. In this way, at the end we will have a tensor that from 0 to text_len-1 is equal to our original text, and from text_len to segment_len * bsz will have padding characters

1.2.6

The line *batch = torch.cat([padded.new_full((1, bsz), pad_id), padded[i * bptt_len:(i + 1) * bptt_len]], dim=0)* appends a pad character at the beginning of all the sequences of the first batch, thus the first batch instead of being 64x32 will be 65x32.

1.2.7

The line *padded[i * bptt_len - 1:(i + 1) * bptt_len]* makes sure that each sequence has as first character the last of the previous one, thus each batch instead of being 64x32 will be 65x32.

1.3 Model and Training

1.3.1

Please check the implementation of the net in the *main.py* file, from line 224 to 269 (Net Class).

1.3.2

Please check the function *text_generator* implemented form line 271 to 330 of the *main.py* file submitted

1.3.3

Please check from line 323 to 325, the sampling choice is implemented within the aformentioned function *text_generator*

1.3.4

The training code is implemented from line 332 to line 406.
The stop criteria is based on the value of the perplexity which must be lower than the threshold 1.03 and also on the the max number of epochs.

1.3.5

Please consider the following results I obtained after the train with the parameters suggested in the assignment. In the plot Fig: 1 you can see the evolution of the mean perplexity per epoch. In the following table instead you can see different strings predicted with the initial text set as 'Dogs like best to '.

Epoch	Guessed Text
1	'Dogs like best to the said t'
10	'Dogs like best to the top of'
20	'Dogs like best to be willing'
23	'Dogs like best to eat it.\n \n'

1.3.6

In the following plot Fig: 2 it is possible to see the different behaviour of the model with different learning rates. In particular with learning rate equals to 0.01 the model does not manage to achieve the optimal perplexity of 1.03, in fact it remains stacked at 8.5. On the other hand, with learning rate=0.0001 the model managed to achieve the desired perplexity of 1.03 but it took much more epochs (80) than using lr=0.001 (23) as we did in our final model. We can conclude the lr = 0.001 is the optimal one.

In the plot Fig: 3 instead we can see what happens if we use different sequence-lenghts with the learning rate = 0.001.

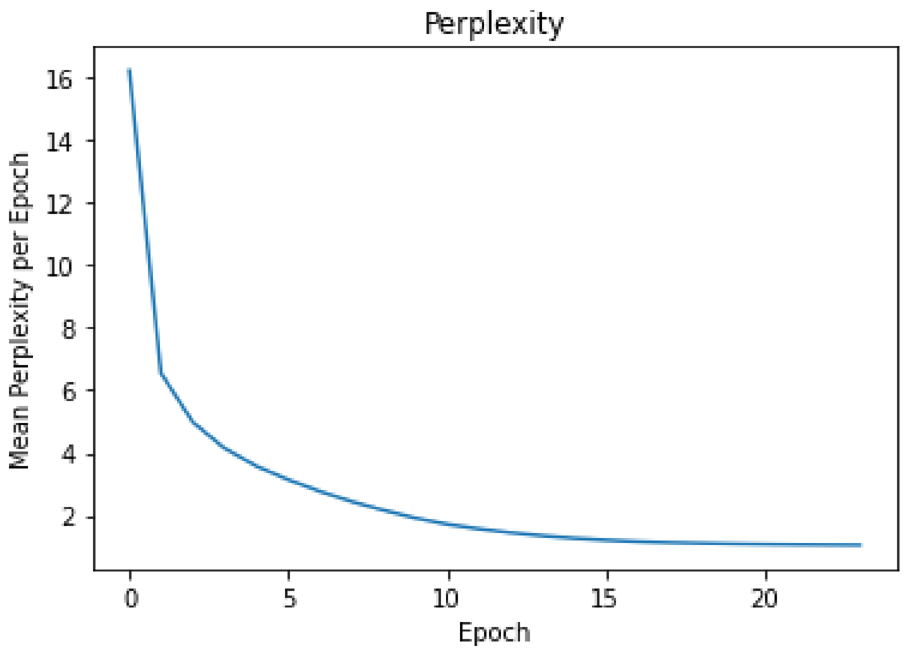


Figure 1: Evolution of the mean perplexity per epoch

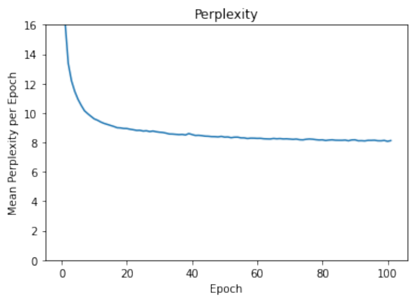
From the results we can conclude that the model with bptt = 16 didn't manage to achieve the desired threshold 1.03 even though it was very close. On the other hand with bptt=256 the model managed to achieve the desired perplexity threshold in ≈ 35 epochs.

1.3.7 Using Greedy choice

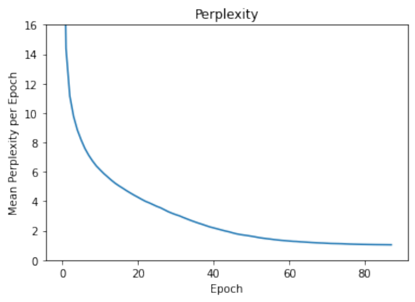
A)
Prompt Text: 'The Kid and the Wolf'
Result: 'The Kid and the Wolf heard the sound, and was caught by some hunters, w '

B)
Prompt Text: 'The Elephant and the Mouse'
Result: 'The Elephant and the Mouse would not enjoy it. When the Donkey plane the Bat'

C)
Prompt Text: 'the squirrel on the tree '
Result: 'the squirrel on the tree to sleep; and the Dog found a hollow in the trunk, '

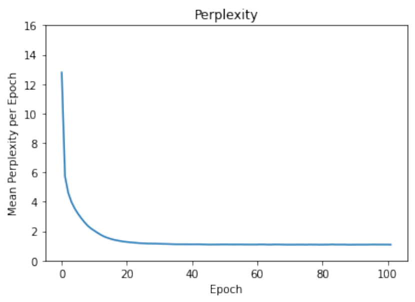


(a)

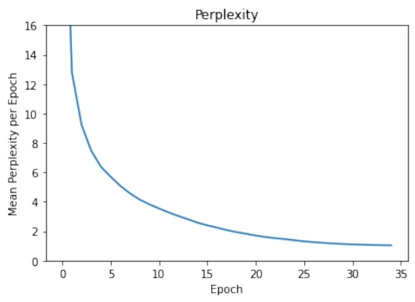


(b)

Figure 2: a) Mean Perplexity w/ lr=0.01 b) Mean Perplexity w/ lr=0.0001



(a)



(b)

Figure 3: a) bptt_len = 16 b) bptt_len = 256

D)
Prompt Text: 'Gandalf and Pippin '
Result: Gandalf and Pippin 195 The Moon and her Mother

1.3.8 Using Sampling choice

A)
Prompt Text: 'The Kid and the Wolf'
Result: 'The Kid and the Wolf 16 The Woodman and the Lamb '

B)
Prompt Text: 'The Elephant and the Mouse'
Result: 'The Elephant and the Mouse 201 The Goose with the Ash, t '

From the prediction I can see that the sampling choice managed to capture better the fact that both were titles (we can infer it from the fact that after the prompt given the lstm gives as output a number which somehow correspond to the page location); on the other hand the greedy choice prediction doesn't manage to capture this fact but anyway it produced some meaningfull text.

1.3.9

For this trial I tried to use the source code of the program (i.e the *main.py*) and I've obtained the following results:

Sampling choice
Prompt Text: 'print'
Result: 'print('\n \t— START TRAINING — \t \n') loss_fun '

Greedy choice
Prompt Text: 'print'
Result: 'print('\n \t— START TRAINING — \t \n') loss_fun '

Sampling choice
Prompt Text: 'import tensorflow as'
Result: 'import tensorflow as nolayers
self.id_to_string[]] = pad_token')

Greedy choice
Prompt Text: 'import tensorflow as'
Result: 'import tensorflow as of the text with open(filename, 'r') as text:'

Also in this case the sampling mode seems to be better in term of generalization (we can see that it managed to capture that after the as it requires an alias and not directly another command), maybe this is due to the fact that sampling from a probability provides a better 'exploration' of the words domain.

2 Questions

2.1 What is the perplexity of a language model that always predicts each character with equal probability of 1/V where V is the vocabulary size?

$$2^V$$

2.2

Others possible sequence prediction problems can be predict from time series data (i.e signal processing) or doing video processing (sequence of related images), for the latter I am thinking about using possible combinations between CNN and RNNs.

2.3 What is the vanishing/exploding gradient problem? And why does this affect the models ability to learn long-term dependencies?

The vanishing gradient refers to the fact that gradients become too small such that the updating step is insignificant; on the other hand exploding gradient refers to the situation in which gradient values become too large (grows exponentially) such that the steps taken by the optimizer are too large. Both situations lead the neural networks to bad performance. The task of learning Long-term dependencies is modelled using a long chain of Recurrent Neural Networks, which means that the chain rule applied to compute the gradient of the loss function is long as well and this very easly leads to vanishing gradient or exploding gradient problems. Longer is the chain of RNNs, longer is the chain rule, higher is the probability to encounter vanishing/exploding gradient problems.