Text generation with LSTM

Nicola Dainese

I. INTRODUCTION

In this project I implemented in Pytorch a Recurrent Neural Network with Long Short Term Memory (LSTM) cells working at word-level. I then trained the network on some books of Jane Austen and finally generated some sample text providing different sequences as context seed.

II. WORD-LEVEL TEXT PREPROCESSING

The purpose of the text preprocessing in this case is to pass from a bunch of books in some format (e.g. .txt) to a dataset of sentences where the use of symbols is as restricted as possible and every word (or best said, every token) in a sentence is correct and meaningful.

First of all I used both the number of consecutive newline characters and the keywords "CHAPTER" and "Chapter" to split every book in chapters, throwing away the initial and the final parts added by project Gutenberg. Then I obtained all the sentences in the text and concatenated them again in a standardized way. For instance sometimes inside a paragraph there can be two phrases separated by a point but without the newline, whereas sometimes two paragraphs are separated by two newlines; after this stage of the preprocessing all the sentences end by a point followed by a new line character.

The next stage is to join again all the sequences and process the full text removing special symbols, numbers and more in general everything except the alphabetic characters, the point, the comma and the newline character. I kept the point and the comma to give a better expressive power to the network (they will be encoded as "words", thus the network will learn their position in a typical phrase), and the newline just for splitting purposes. I also substituted the accented vowels with the standard ones and kept everything lowercase.

At this point I extracted the set of all unique tokens (i.e. all the sets of characters between two spaces) and created two dictionaries for encoding (word2index) and decoding (index2word) each word to a numeric index and vice versa.

I finally created a list containing all the encoded (sometimes I will call them "numerical") sentences, that are python lists, whose elements are the indices associated to the words that compose the sentence. This list of numerical sentences is saved on file together with the two dictionaries, because the encoding depends on the order in which the element of the set are numbered and this is not unique in general (since sets aren't ordered).

III. WORD EMBEDDING IN PYTORCH

By the term "word embedding" we usually intend every dense vector representation of categorical words. Basically we first associate to every word a unique index, then represent it as a one hot encoded vector and feed it to a matrix (the embedding layer) so that we have an input dimension equal to the number of unique words in the dataset and on the output dimension we have what is called the embedding dimension, whose choice is somewhat arbitrary, but usually is of the order of 100. Pytorch provides an implementation of the embedding layer that can either be trained as a standard layer or can be used to load embedding parameters obtained through libraries like word2vec or GloVe. For simplicity and lack of time I just used the random initialization and trained the layer together with the rest of the network.

Of course to expand an index to a one hot encoded version that usually needs like 10.000 times more memory just for feeding it to a matrix that will compress it again is very inefficient, thus the Pytorch Embedding layer accepts directly the indices (it is effectively a sort of lookup table).

The other thing to consider is that to train the network efficiently (e.g. with multiple cores or GPUs) we usually want to use mini-batching, but the problem is that each sequence is of different length in general, thus we cannot form naturally tensors because, assuming that we have the two dimensions as (batch size, sequence length), the second axis won't match for different sample sequences. The solution is to use padding, that is to add a special value (e.g. the zero) at the end or the beginning of each sequence, in order to match the longest sequence. This introduces other two complications: the first is that we don't want the padding to change somehow the result and the second is that, being the sequence length highly variable, it's inefficient up to a factor 20 (at least in my case that is the ratio between the longest and the shortest sentences) to process.

The first issue is solved by Pytorch providing to the embedding layer a keyword "padding_idx" that maps to a vector of zeros all the indices that match the one used for padding. Hence it is sufficient to reserve in the vocabularies used for encoding and decoding the pair {'<PAD>':0} to obtain a null input to the network; still it's not completely clear if that is sufficient to have a null output or how to handle it without a proper masking of the final loss. However this is not a problem if we use padding together with packing: as the name suggests, a packed sequence is a compressed version of a padded one in which we register for each batch the non -zero

values and the effective sequence length for each sample in the batch. To do that one has to sort in ascending or descending order the original sequence before padding, otherwise the packing is not efficient. With packing we solve also the efficiency issue, since it is designed appositely to avoid computing the forward and backward feeds of the network.

To recap, basically what needs to be done to use mini-batches is:

1. Extract a batch.
2. Sort the items by length and pad them (although I pad them before, in order to construct a tensor dataset and to use the DataLoader class for batching).
3. After the embedding layer (I am still wondering why it's not allowed to do this before the embedding), pack the sequence (this requires also to provide the original sequence lengths in a separate tensor).
4. Feed the packed input to the LSTM.
5. Unpack the LSTM output.
6. Use some linear layer to obtain a soft-max output of the same dimension of the input.
7. Compute the loss (e.g. cross-entropy) masking out the padded outputs.

To be as concrete as possible, this is the code I use for the forward pass in the neural network, already providing both the padded sequences x and their lengths:

```
def forward(self, x, seq_lengths, state=None):

# Embedding of x
x = self.embedding(x, padding_idx=0)

# packing for efficient processing
pack_input = pack_padded_sequence(x,
                seq_lengths,  batch_first=True)

# propagate through the LSTM
pack_out, state = self.rnn(pack_input, state)

# unpack for linear layers processing
output, _ = pad_packed_sequence(pack_out,
                            batch_first=True)

# Linear layer
output = F.leaky_relu(self.l1(output))

# Linear layer with log_softmax
output = F.log_softmax(self.out(output), dim=2)

return output, state
```

From the code above it is also clear the model that I used: an embedding layer, then an LSTM and two linear layers. To compute the loss I used a cross-entropy loss masking out the elements corresponding to the padding:

```
def masked_cross_ent_loss(y_true, y_pred,
                                pad_token=0):
"""
[y_true]=(batch_size, seq_len)
[y_pred] (batch_size, seq_len, vocab_size)
"""

y_t_flat = y_true.reshape(-1)
y_p_flat = y_pred.view(-1, y_pred.size()[-1])

# consider only the non-padded words
mask = (y_t_flat != pad_token).float()

# number of non-padded elements
num_tokens = int(torch.sum(mask).item())

# for each word choose the log of
# the prob predicted for the right label
indices = range(y_p_flat.shape[0])
y_p_masked = y_p_flat[indices, y_t_flat] * mask

# compute the mean cross entropy
cross_ent_loss = - torch.sum(y_p_masked) / num_tokens

return cross_ent_loss
```

IV. TRAINING AND GENERATION FOR A RNN AT WORD-LEVEL

In this section I discuss briefly the concepts behind the training of the RNN and the generation of text starting from a given context (seed).

When working with sequences one has to choose which is the objective of the training: usually we either want to predict the last (next) word of a sentence or we want to predict every word starting from the second one. I think that the second objective makes a better use of the data we have and also avoids a silly problem that I would have encountered otherwise, that is the fact that the last token in every sentence in my case is a point. Therefore it would be uninformative to have the same label for each sequence and the network would only learn that at the end of each phrase there is a point. One could argue that it would be sufficient to remove the points at the end of the sentences, but I wanted to shape the expressive power of the network so that in principle it could produce sensible paragraphs made of more sequences.

As a consequence I train the network in predicting for each word in a sentence the subsequent word and this is why in the loss function the labels y_true are of

dimension (batch_size, seq_len).

For text generation what I do is the following:

1. Pre-process the seed and encode it as a numerical sentence;
2. Use all words except the last to initialize the contextual hidden state of the LSTM (basically it's a feed-forward);
3. Do a feed-forward for the last word passing the contextual hidden state;
4. Loop as many times as the words that I want to generate giving as input the new contextual hidden state and the last word produced;
5. Decode them and join them in a single sentence.

An edited (pseudo) code of the generation part is the following (preprocessing is the custom module that contains most of the function for text processing that I have written):

```
def generate_sentence(seed, trained_net,
                      word2index, index2word,
                      len_generated_seq=10, T=1):
# import preprocessing as prep
seed = prep.seed(seed)
num_seq = prep.encoder(seed, word2index)

enc_seq = torch.LongTensor(num_seq).view(1,-1)
context = enc_seq[:,:-1]
last_word = enc_seq[:,-1].view(1,1)
# the network needs the length for packing
len_context = np.array([len(num_seq)-1])
len_context = torch.LongTensor(len_context)
len_context = len_context.view(1)
len_last = torch.LongTensor([1]).view(1)

with torch.no_grad():
net.eval() #disable dropout
# returns all the outputs "_" and the
# last hidden state
_, hidden_context = net(context, len_context)

gen_words = []
for i in range(len_generated_seq):
# the output contains the log
# of the predicted probabilities
last_word_ohe, hidden_context = net(last_word,
                len_last, state=hidden_context)
# T < 1 is less noisy
last_word_ohe = last_word_ohe.numpy().flatten()
prob_last_w = np.exp(last_word_ohe/T)
# normalize
prob_last_w = prob_last_w/ prob_last_w.sum()
# sample the last word
choices = np.arange(len(prob_last_w))
last_word_np = np.random.choice(choices,
                                p=prob_last_w)
gen_words.append(last_word_np)
last_word = torch.LongTensor([last_word_np])
last_word = last_word.view(1,1)
# end of the for loop
gen_words = np.array(gen_words).flatten()
decoded_sentence = prep.decoder(gen_words, index2word)
output_string = ' '.join(decoded_sentence)
return output_string
```

V. RESULTS AND DISCUSSION

In this section I present my results from a quantitative and qualitative point of view.

A. Training results

The training was done with the following parameters for the network and the optimizer:

Hyper-parameters	
HP	Value
Embedding dimension	100
LSTM units	128
LSTM layers	2
Dropout	0.2
Number of epochs	20
Optimizer	Adamax
Learning rate	0.1
Type of penalty	L2
Penalty weight	10^{-4}

With those values and 20 epochs the model was converging without overfitting and took around 15 hours to train. The training curve can be seen in Figure 1.

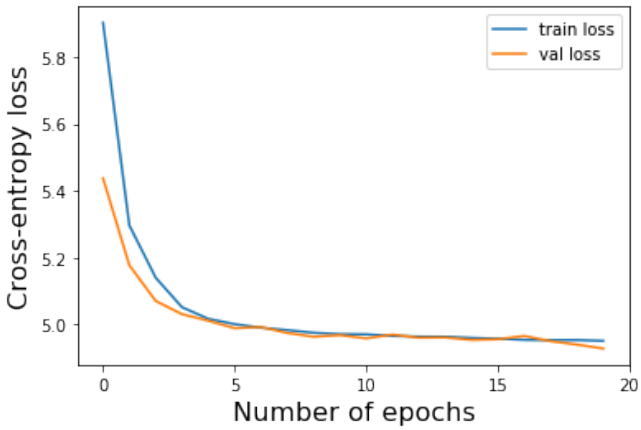

FIG. 1: Training converged in 20 epochs.

My choice was to prioritize fast convergence over all, hence I took what I thought to be the most stable optimizer (Adamax) from my previous assignment, used a large learning rate and batches of 32 samples each. I mainly relied over the dropout and the high learning rate to avoid overfitting.

B. Text generation results

I now report some paragraphs generated by the trained network with the same seed and different noise levels T. The seed was the incipit of Pride and Prejudice, "It is a truth universally acknowledged", and these are some of the results:

1. $T = 1.0$: "by us for you have been thorough in getting at the park . connexions or caution , to please before in dinner for him in his very voice with the time and visitors was taken up striking , french , till lady their william was well claim . in making her aunt , till this had her interest , because mr allen remained simile they had mr crawford , he spoke her from the girls said exercise to expect in her way , i shall set out of her memory , every awkward reproved in that countenance , mr dashwood made to comprehend from the subject , where renewed such years and beauty , how can i know the winter and from the same light . her duty was too far the smallest opportunity , and as a complacency to be made to do i have been all the touch of penetration with saying . . meanwhile but you thought what you and yourself they are shocked . , to forgive , nothing cannot be indeed taking our notice of our power . "

2. $T = 0.5$: ", and this will not be to call , and i have been more than . . . . . and that is so much in the subject . , to be engaged , and i am not more than to have the smallest reason to be the same little young man , in a while of what i shall have been the same young man . to get up , as ever as i have been aware of the young man . . . , who had been at the same time , and she has been to be able to be in the room , and this was to be . , and she was in the whole day , and ought to know , and the letter , and the attention of the business . , and taking the same of the end of the other day . and she was much in the same , and she had been the same of the evening . "

3. $T = 0.25$: ", and i am sure , i have been in the room . . , and i am sure , i am not sure , and i am sure , i am sure , and i am sure , and i am sure , i am sure , and you are not in the same time ."

First of all we can observe that, even though the text never makes sense, usually there is a good grammatical structure, with subjects, objects, verbs, pronouns and adjectives used properly.

Secondly, as the network hasn't learned the semantic of the text, it hasn't learned how to use punctuation either: there isn't a clear distinction between the use of points and commas, so much so that many times they are used together. This to me is the most strange thing, because it's probably the simplest error to avoid.

Regarding the temperature parameter, we can see that decreasing it we loose the natural richness of vocabularies of Austen's style and only the most common phrases like "and I am sure," appear.