Assignment 3. Language Identification. Sentence Classification

Alex Carrillo and Robert Tura
Spoken and Written Language Processing

April 15, 2020
Universitat Politècnica de Catalunya

1 Introduction

In previous assignments we worked in depth with the *word2vec* approach for handling several NLP tasks. The technique caused words that occurred in similar contexts to have similar embeddings. Now, in this report we study the specific task of classifying in which language are some pieces of text written.

However, this time instead of aggregating from words to vectors, we aggregate from characters to vectors. The reason for doing that comes from the fact that word vectors operate with a fixed vocabulary, usually lacking some rarely used words. Hence, if several words of a text do not belong to the vocabulary of the language model, the model will not be able to interpret these words, being that quite inefficient. This set of new rarely used words can be seen as words like own names, slang or typos which may not be present in language vocabulary.

As an example described by V. Chikin, a comment like "directed by Tarantino" in a movie comments dataset may appear written with errors or typos in surname in other comments. The ability to extract this "Tarantino" feature from new words like "Taratino" or "Torantino" would obviously improve model's accuracy [2].

2 LSTM and GRU

As we know, recurrent neural networks (RNNs) suffer from short-term memory in the sense that if a sequence is long enough, they will have a hard time carrying information from earlier time steps to later ones. So, if we are trying to process paragraphs of text to predict its language, RNNs may leave out important information from the beginning (even more when we are not analyzing them

at a word level but at character level).

Long short-term memory (LSTM) and *gated recurrent unit* (GRU) networks were created as the solution to short-term memory. Without going into details, the key of these architectures is they have internal mechanisms called *gates* that can regulate the flow of information of previous hidden states of a RNN, hence, dealing with the problem of vanishing/exploding gradients.

As GRUs are less known to us, we should briefly comment them a bit. They are the newer generation of RNNs and work pretty similar to an LSTM but with the nuance of not having the *cell state*, thus using the hidden state to transfer information. So, they have only two gates: a *reset gate* and an *update gate*. Broadly speaking, they are a little speedier to train than LSTMs, as they perform fewer tensor operations. In any case, there is not a clear answer to which one is better, so we will have to try both to determine which one works better for our case.

3 Hyperparameter optimization

3.1 Embedding size

The first hyper parametrization change is going to be issuing the embedding size: this change affects the encoding and proceeds to allow more or less data crammed into each encoding. We tried to use 3 different embedding sizes, the basic one, 64, and two of higher size, 128 and 256, respectively.

We perceived that increasing the embedding size we get a slight decrease in accuracy (both training and validation), even when the parameter is 2 times the basic value or 4 times. The only important change is in the execution time, an increase of 7.5% and 15% respectively. For that reason is why we are maintaining the basic value for the embedding size.

As a corollary of this behavior, and taking into account the information gathered in the previous assignments, we conclude that while smaller embeddings do not represent the semantics of the corpus well enough, longer ones may not add enough information. What matters more is the network architecture, the algorithms and the dataset size.

3.2 Batch size

The second hyper parameter we want to tweak is the batch size. This parameter affects directly to the amount of batches that the model is going to be using, and also the diversity of data samples inside the batches. Ideally, if these batches are big enough, this approach will help the training phase. So, for this specific cases we tried increasing this sizes by 2 and by 4 as previously, *i.e.* 512 and 1024, respectively.

As with the embedding size, this modification did nothing but decrease the accuracy in the model for both datasets, but unlike before here we get a decrease in execution time of 9% and 18%. This is due to the fact that as the batches get bigger and bigger there are less loops in which the model has to go through.

3.3 RNN hidden size

The last hyper parameter that we tweaked is the size of the hidden layers in the recurrent neural network. As a remark, it is clear that the higher the size of the hidden layer the higher the amount of parameters which are going to be trained and, therefore, that are going to help our model predict more precisely (according to what we expect). However, we also expect a huge increase in computation time.

As before, we have tried 2 different layouts for this model: the sizes are 2 times bigger and 4 times bigger, *i.e.* 512 and 1024, respectively. On the one hand, the first one had a relatively good increase in accuracy, but the execution times went up by almost double the time. In the other hand, the second execution had a hidden size that was too high for the GPU and it was not able to run, as we foresaw.

Now, from this point forward we are going to try different changes in the network architecture in order to create a better model and then in the end we might try to improve it with more tweaking, for instance, in the hidden size.

4 Dropout, max-pool and mean-pool

The next things that we are going to try for the improvement of the model is going to be the use of different techniques and our personal approach to it, which becomes the most interesting part.

First of all, the basic model is using a max-pool layer to reduce the dimensionality of the network, but there exists other possible pooling layers which might perform better. We tried using the mean-pool layer, which is similar to the max-pool layer but instead of taking the maximum value we take the average. As an alternative, we could also take the median, for instance, but doing some research we found it is not very widespread and, in essence, the mean was better.

Following this line, the second approach of tweaking the pooling layer was adding the values from the max-pool and the mean-pool, and see what happened. Surprisingly, this strategy was a huge step forward because it gave us the best improvement we have had so far, and theoretically we did not expect that this would be a too significant approach. The code can be found in Appendix A.

Then, with the previous result in mind, we also tried concatenating both vectors: max and mean, but this time it did not turn into any good results compared with the models we got so far. The code of the network can be found in Appendix B.

Next, as another measure of improvement, we are going to add a dropout layer just before the final fully connected linear layer that gives out the output. As we know, dropout is a regularization technique to avoid overfitting and increase the validation accuracy. Thus, it could be very beneficial in our case and may help also increase the generalizing power of the network.

Lastly, it should be noted there are two aspects to take into account in what we did above. First of all, we have tried 4 different dropouts percentages (10%, 30%, 50%, and 75%) and basically we can define the behavior of the dropout layer as a kind of normal distribution. Above all, for the best results it came from the layer with 50% of dropout.

The second aspect of our own dropout layer is the definition of itself, the layer is just at the end of the model, what makes the output to be wrong X% of the times. Why is that? Because we are neglecting half of this outputs for the sake of training means. This method tries to lower the "low" values of the vector of probabilities. This method takes the probability vectors and shuts X% of the outputs to 0 turning this vector to the new output of the model, and therefore X% of the answers will surely be wrong, but that is not the case after several epochs, because the model learns to differentiate the correct answers from the others making those weights lower than 0. This shuts the internal training accuracy up to 100-X% but helps with the backpropagation.

5 Further: bigrams

The last approach for the model we experimented with is changing the process which is used in the baseline model in order to index the characters. This time, we came up with a method in which, instead of that, we make an indexation of two consecutive characters at the same time. For instance, instead of indexing 'HOLA' as 'H', 'O', 'L', 'A' tokens, we use the combinations of letters in pairs, *i.e.* 'HO', 'OL', 'LA'. This method is supposed to improve the confidence and the precision of the model because we are using the proximity between words and not just word appearance as the monogram model does. As stated in the introduction, the ability to extract this new features at that level, should indeed improve model's accuracy. The code of the bigram approach can be found in Appendix C and D.

However, as we did this on our own way and without using any clear criteria, there are flaws on the starts and ends of sentences, which might lead to mediocre results and not the best of performances. Leaving that aside, we have to say this approach works decently and gives us values of accuracy close to the best ones so far. It is clear that for future research related to this ngram-approach we might want to code a better method that ensures all possible pairs are being added to the model and study this type of architecture in more depth.

6 Results

As a final recap we took all the information of each run of the models and we present it in the form of a table:

Arhitecture	Embed.	Hidden	Batch	Drop	Params.	Time	Test	Val
LSTM	64	256	256	-	1081835	0.72	98.7	92.9
	128	256	256	-	2217707	0.78	98.3	92.6
	256	256	256	-	3353579	0.83	98.6	92.7
	64	512	256	-	1996011	1.5	100	93.9
	64	1024	256	-	5397227	-	-	-
	64	256	512	-	1081835	0.67	97.3	92
	64	256	1024	-	1081835	0.61	97.1	92.6
LSTM with dropout	64	256	512	0.1	1996011	1.51	87.7	93.4
	64	256	512	0.3	1996011	1.51	70.4	94.4
	64	256	512	0.5	1996011	1.51	50.4	94.5
	64	256	512	0.75	1996011	1.51	25.3	94.5
sum of pool layers	64	256	512	0.5	1996011	1.9	50.5	94.8
conc. of pool layers	64	256	512	0.5	2116331	1.89	50.4	94.4
LSTM Bigram	64	256	512	0.5	21620459	1.92	50.3	94.5
GRU Monogram	64	256	512	0.5	1700075	1.6	50.3	93.9
GRU Bigram	64	256	512	0.5	21324523	1.66	50.3	94.0

Table 1: Summarized results of all our model architectures.

All this results are based on the output of the Kaggle notebook. Accuracy is based on the training set which, for the models that have dropout layer, they have a lower value due to the final dropout that is being done. Also, the Validation results come from the 15% of the training set that, this are not affected by the effect of the dropout layer (it only affects to the training) and the values are up to 100%. And finally the best model with the private part of the test set was the **LSTM with monogram, dropout layer (0.5), and sum of pooling layers**, with a total of 95.38% of accuracy.

References

[1] Jason Brownlee. *How to Develop a Character-Based Neural Language Model in Keras*. 2019. URL: https://machinelearningmastery.com/develop-character-based-neural-language-model-keras/.

[2] Vladimir Chikin. *Chars2vec: character-based language model for handling real world texts with spelling errors and...* 2019. URL: https://hackernoon.com/chars2vec-character-based-language-model-for-handling-real-world-texts-with-spelling-errors-and-a3e4053a147d.

[3] Edward Ma. *Besides Word Embedding, why you need to know Character Embedding?* 2018. URL: https://towardsdatascience.com/besides-word-embedding-why-you-need-to-know-character-embedding-6096a34a3b10.

[4] Michael Nguyen. *Illustrated Guide to LSTM's and GRU's: A step by step explanation*. 2018. URL: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21.

Appendix

A Sum of pool layers

```
class CharRNNClassifier(torch.nn.Module):

    def _init_(self, input_size, embedding_size, hidden_size, output_size, model="lstm",
               num_layers=1, bidirectional=False, pad_idx=0):
        super()._init_()
        self.model = model.lower()
        self.hidden_size = hidden_size
        self.embed = torch.nn.Embedding(input_size, embedding_size, padding_idx=pad_idx)
        if self.model == "gru":
            self.rnn = torch.nn.GRU(embedding_size, hidden_size, num_layers,
                                    bidirectional=bidirectional)
        elif self.model == "lstm":
            self.rnn = torch.nn.LSTM(embedding_size, hidden_size, num_layers,
                                     bidirectional=bidirectional)
        self.h2o = torch.nn.Linear(hidden_size, output_size)
        self.drop =  torch.nn.Dropout(p=0.5)

    def forward(self, input, input_lengths):
        # T x B
        encoded = self.embed(input)
        # T x B x E
        packed = torch.nn.utils.rnn.pack_padded_sequence(encoded, input_lengths)
        # Packed T x B x E
```

```
        output, _ = self.rnn(packed)
        # Packed T x B x H
        # Important: you may need to replace '-inf' with the default zero
        # padding for other pooling layers
        padded_m, _ = torch.nn.utils.rnn.pad_packed_sequence(output,
                                                             padding_value=float('-inf'))
        # T x B x H
        output_m, _ = padded_m.max(dim=0)

        padded_a, _ = torch.nn.utils.rnn.pad_packed_sequence(output, padding_value=0)
        output_a = padded_a.mean(dim=0)

        output = (output_m + output_a)
        # B x H
        output = self.drop(self.h2o(output))
        # B x O
        return output
```

B Concatenation of pool layers

```
class CharRNNClassifier(torch.nn.Module):

    def _init_(self, input_size, embedding_size, hidden_size, output_size, model="lstm",
               num_layers=1, bidirectional=False, pad_idx=0):
        super()._init_()
        self.model = model.lower()
        self.hidden_size = hidden_size
        self.embed = torch.nn.Embedding(input_size, embedding_size, padding_idx=pad_idx)
        if self.model == "gru":
            self.rnn = torch.nn.GRU(embedding_size, hidden_size, num_layers,
                                    bidirectional=bidirectional)
        elif self.model == "lstm":
            self.rnn = torch.nn.LSTM(embedding_size, hidden_size, num_layers,
                                     bidirectional=bidirectional)
        self.h2o = torch.nn.Linear(hidden_size*2, output_size)
        self.drop =  torch.nn.Dropout(p=0.5)

    def forward(self, input, input_lengths):
        # T x B
        encoded = self.embed(input)
        # T x B x E
        packed = torch.nn.utils.rnn.pack_padded_sequence(encoded, input_lengths)
        # Packed T x B x E
        output, _ = self.rnn(packed)
        # Packed T x B x H
        # Important: you may need to replace '-inf' with the default zero
        # padding for other pooling layers
        padded_m, _ = torch.nn.utils.rnn.pad_packed_sequence(output,
```

```
                                                                padding_value=float('-inf'))
        # T x B x H
        output_m, _ = padded_m.max(dim=0)

        padded_a, _ = torch.nn.utils.rnn.pad_packed_sequence(output, padding_value=0)
        output_a = padded_a.mean(dim=0)

        output = torch.cat((output_m, output_a), dim=1)
        # B x H
        output = self.drop(self.h2o(output))
        # B x O
        return output
```

C Bigram approach part 1

```
#char_vocab = Dictionary()
pair_vocab = Dictionary()
pad_token = '<pad>' # reserve index 0 for padding
unk_token = '<unk>' # reserve index 1 for unknown token
#pad_index = char_vocab.add_token(pad_token)
#unk_index = char_vocab.add_token(unk_token)
pad_index = pair_vocab.add_token(pad_token)
unk_index = pair_vocab.add_token(unk_token)

# join all the training sentences in a single string
# and obtain the list of different characters with set

chars = list(''.join(x_train_full))
#chars = set(''.join(x_train_full))
#for char in sorted(chars):
#    char_vocab.add_token(char)
#print("Vocabulary:", len(char_vocab), "UTF characters")
i=0
k=0
for char in chars:
    if k==0:
        c1 = char
        #print(c1)
        k=1
        if i>0:
            pair_vocab.add_token(c2+c1)
    else:
        c2=char
        #print(c2)
        pair_vocab.add_token(c1+c2)
        k=0
    i=i+1
    #input()
```

```
print("Vocabulary:", len(pair_vocab), "UTF pairs")


lang_vocab = Dictionary()
# use python set to obtain the list of languages without repetitions
languages = set(y_train_full)
for lang in sorted(languages):
    lang_vocab.add_token(lang)
print("Labels:", len(lang_vocab), "languages")
```

D Bigram approach part 2

```
#From token or label to index
print('a ->', pair_vocab.token2idx['An'])
#print('a ->', char_vocab.token2idx['a'])
print('cat ->', lang_vocab.token2idx['cat'])
print(y_train_full[0], x_train_full[0][:10])
x_train_idx = []
t = 0
for line in x_train_full:
    t=t+1
    i = 0
    k = 0
    tmp = []
    for c in line:
        if k==0:
            c1 = c
            k=1
            if i>0:
                tmp.append(pair_vocab.token2idx[c2+c1])
            i=i+1
        else:
            c2 = c
            tmp.append(pair_vocab.token2idx[c1+c2])
            k=0
    x_train_idx.append(np.array(tmp))

#x_train_idx = [np.array([char_vocab.token2idx[c] for c in line]) for line in x_train_full]
y_train_idx = np.array([lang_vocab.token2idx[lang] for lang in y_train_full])
#print(y_train_idx[0], x_train_idx[0][:10])
```