Diacritics Restoration using Neural Networks

Asad Idrees Razak, Ács Judit
Budapest University of Technology and Economics

In the past and still at present, people often replace characters with diacritics with their ASCII counterparts. Even though the resulting text is usually easy to understand for humans, it is much harder for further computational processing. When writing emails, tweets or texts in certain languages, people for various reasons sometimes write without diacritics. When using Latin script, they replace characters with diacritics (e.g., c with acute or caron) by the underlying basic character without diacritics. Practically speaking, they write in ASCII.
https://github.com/idreesshaikh/Diacritics-Restoration

1. Introduction

Almost half of the words in Hungarian Language Contains diacritics. So, restoring diacritics became an important task in particulars with Hungarian Language.

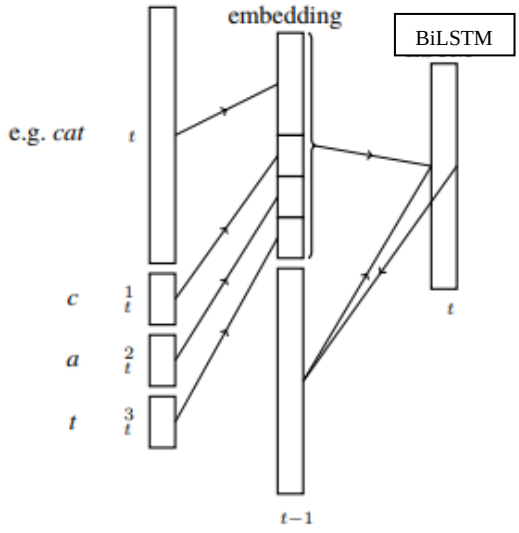
We make use of the preprocessed data of Hungarian Language into three train, dev, and test splits. Each one with and without their corresponding diacritics separated by a tab. That helps us read data eventually without the use of any other external library. In this very project we are trying to map our input sequence on the output sequence with a 'character level embedding' implementation. a Character-Word Long Short-Term Memory Language Model which both reduces the perplexity with respect to a baseline word-level language model and reduces the number of parameters of the model. Character information can reveal structural (dis)similarities between words and can even be used when a word is out-of-vocabulary, thus improving the modeling of infrequent and unknown words. We will use a parallel dataset of the following form where each tuple represents a pair of (non_diacritized, diacritized) input and output sequences.

('koszonom szepen', 'köszönöm szépen')

2. Model Architecture

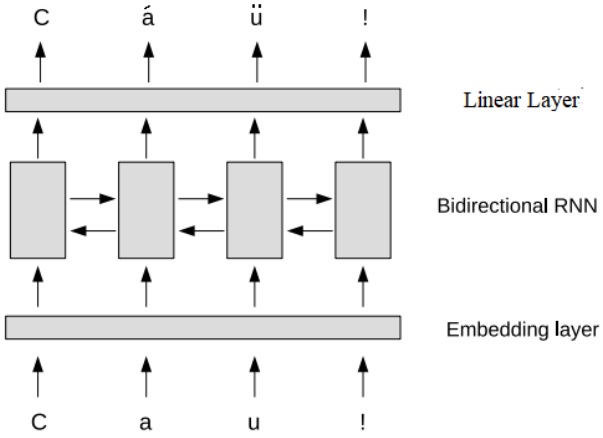
Neural networks are essentially number crunching machines and have no idea how to handle our non-diacritic input. We will somehow have to convert our strings into numbers for the network to make sense of them.

The idea is to represent every character in the alphabet with its own **D** dimensional embedding vector, where **D** is usually called the embedding dimension. If we decide to use an embed_dim of 10. This basically means that each of the 193 characters (Unicode) will have their own embedding vector of length 5. Often, these vectors are stored together as **V×D** dimensional embedding matrix, **E**.



After which these lists are converted into a pair of tensors, which is what the function read () returns of the class

getDataSet. A tensor is essentially a multidimensional matrix which supports optimized implementations of common operations.



General idea is to feed in inputs to an LSTM to get the predictions. Next, we pass on the predictions along with the targets to the loss function to calculate the loss. Finally, we backpropagate through the loss to update our model's parameters.

Diving into our model class BiRNN, we have initialized **torch.nn.Embedding**. The two required parameters in our case, these are character_len **V** and embed_dim **D,** respectively.

Next, we need to initialize an LSTM. We do this in a similar fashion by creating an instance of **torch.nn.LSTM**. The required parameters as **input_size**: the number of expected features in the input and **hidden_size**: the number of features in the hidden state. **torch.nn.LSTM** expects the input to be a 3D input tensor of size (seq_len, batch, embed_dim), and returns an output tensor of the size (seq_len, batch, hidden_dim). We will only feed in one sequence at a time, so batch is always seq_len.

BiLSTM reads the input in standard order (forward RNN) and the other in reverse order (backward RNN). The output is then a sum of forward and backward RNN outputs. This way, bidirectional RNN is processing information from both preceding and following context. The output of the (possibly multilayer) bidirectional RNN is at each time step reduced by an identical fully connected layer to an O-dimensional vector, where O is the size of the output alphabet. Generally, the **LSTM** is expected to run over the input sequence character by character to emit a probability distribution over all the letters in the vocabulary. So, for every input character, we expect a **V** dimensional output tensor where **V** is **193**. The most probable letter is then chosen as the output at every timestep.

BiLSTM produces an output tensor O of size **seq_len x batch x hidden_dim**. This essentially gives us an output tensor **O** of size **seq_len x hidden_dim**. Now if we multiply this output tensor with another tensor **W** of size **hidden_dim x embed_dim**, the resultant tensor **R=O×W** has a size of **seq_len x embed_dim.** Linear layer concatenates and transforms the input tensors.

Next, we pass on the predictions along with the targets to the loss function to calculate the loss.

LSTM is essentially performing multi-class classification at every time step by choosing one character out of the 193 characters of the alphabets. A common choice in such a case is to use the **cross-entropy loss** function **torch.nn.CrossEntropyLoss.**

Finally, we backpropagate through the loss to update our model's parameters.
Therefore, the popular choice is **Adam Optimizer.**

Now we just let our Recurrent Neural Network Train.

3. **Accuracy of the Neural Network**

The accuracy of this project lies around 89.88%, which means there is 0.8988 probability that it will restore the diacritics in particular to Hungarian Language.