Comparing deep learning spell corrction approaches for a keyboard inaccurate typing.

As a part of the accessible keyboard project

Danylo Kolinko, Ukrainian Catholic University

***Abstract*—Spell correction is one of the most successful and valuable domains of the text generation field of NLP. Its' applications are mostly aiming at two problems. First is the user experience improvement, and second is the preprocessing of textual data. By this project, I would like to tackle quite a different problem - accessibility. People who have cerebral palsy and other movement disorders often cannot properly use the current keyboard since usage requires hand movement, which can be challenging and trigger cramps. The partial solution is developing a more accessible keyboard layout with only three keyboard rows to restrain keyboard usage to only finger movements. This approach requires more control keys or rotation-on-press approach. Both techniques are much more time consuming, and there is some space for further automation. The first automated technique is spell correction, which can relieve users from manual correction, what they had to perform quite often due to cramps or just inaccuracies in the movements. So I chose to develop some of the spell correction techniques and compare their efficiency. The code with experiments and preprocessing is available on Github.**

***Index Terms*—spell correction, accessibility, keyboard layout, etc.**

I. Introduction

SPELL correction problem undergone huge transformation in recent years. Just some ten years before, the main approach was to learn the language model and the error model. Which can be briefly explained in the following list:

- Selection Mechanism: argmax
  We choose the candidate with the highest combined probability.
- Candidate Model: $c \in candidates$
  This tells us which candidate corrections, c, to consider.
- Language Model: $P(c)$
  The probability that c appears as a word of English text. For example, occurrences of "the" make up about 7 percent of English text, so we should have $P(the) = 0.07$.
- Error Model: $P(w|c)$
  The probability that w would be typed in a text when the author meant c. For example, $P(teh|the)$ is relatively high, but $P(theeexyz|the)$ would be very low.

This model was later improved by reformulating it as an n-gram problem to add the context to language modeling.

With a rise of DL was developed several new approaches to this problem. I would like to report results on several of them modified for keyboard-neighboring mistypes:

1) semi-character RNN
2) character CNN
3) seq2seq approach

II. Dataset and preprocessing

Reviewing related to spell correction datasets, I discovered that available datasets are focused on the unigram correction, which is not optimal for the usage of RNN. Among them are WikiEdit Corpus and Birkbeck Spelling Error Dataset. Additionally, they provide little information to train on.

Having that, I tested Peter Norvig's spell correction model, which is essentially the simple unsupervised model described in the Introduction.

I discovered that accuracy for such errors remained at 40-60% varying across datasets and was highly inefficient in terms of time. This test was an excellent sign to dug into DL approaches. Of course, there are some techniques to improve such an approach, but a significant improvement seemed doubtful.

So I chose to create my own dataset. For this purpose, I used large corpus that consists of concatenated files from the Gutenberg project - open-source book collection.
Then to create target data, I used the nlpaug library, which has a rich collection of augmenters. I decided to create target data with KeyboardAugmenter that simulates keyboard errors.

Other than that, general data preprocessing differs for each model, so this step will be additionally reviewed for each model.

III. Semi-character RNN model

This model was a recreation of the Sakaguchi et al. 2017 paper that introduced a new deep learning approach to the spell correction problem.

Researches suggest that since for humans, spell correction is usually effortless, there might be a more intuitive technique for this problem then building a language and error model with generating candidates as exhaustive search on the n-edit space.

They also suggest that cognitive studies show that spell correction is so easy for humans, particularly in the jumbled letter problem (middle letters are randomly interchanged while first and the last stay at a place). Such as this little piece:

Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy,it deosn't mttaer in waht oredr the ltteers in a wrodare, the olny iprmoetnt tihng is taht the frist and lsatltteer be at the rghit pclae. The rset can be a toatlmses and you can sitll raed it wouthit porbelm. Tihsis bcuseae the huamn mnid deos not raed ervey lteterby istlef, but the wrod as a wlohe.

So the suggestion is to use the following model (scRNN) with the jumbled word as input and clear one as input.

The preprocessing step for this approach consists of tokenization and then embedding each word as a concatenation of 3 characteristic vectors. First and last vectors are one-hot embeddings of first and last characters in the word, while the second is the count of each character in the middle section (which disregards any order).

As model authors chose two stacked LSTMs with fully-connected layer on top, then Softmax is used and CrossEntropy as a loss function since the problem formulated as classification word to word.

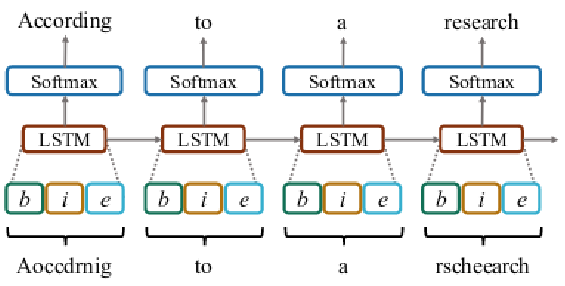

Figure 1. Semi-character RNN model structure.

LSTM is used to deal with two main problems of RNN which are vanishing gradient and long term blindness or inability to learn long term connection.

The LSTM cell has an ability to discard or keep previous information in its state. Technically,the LSTM architecture is given by the following equations:

$$i_n = \sigma(W_i[h_{n-1}, x_n] + b_i \tag{1}$$
$$f_n = \sigma(W_f[h_{n-1}, x_n] + b_f \tag{2}$$
$$o_n = \sigma(W_o[h_{n-1}, x_n] + b_o \tag{3}$$
$$g_n = \sigma(W_g[h_{n-1}, x_n] + b_g \tag{4}$$
$$c_n = f_n \odot c_{n-1} + i_n \odot g_n \tag{5}$$
$$h_n = o_n \odot \tanh(c_n) \tag{6}$$

Where σ is the (element-wise) sigmoid function and $\odot$ is the element-wise multiplication. While a standard input vector for RNN derives from either a word or a character, the input vector in scRNN consists of three sub-vectors embeddings.

There is another alternative to RNN, which is a somewhat similar Gated Recurrent Unit with a difference in two present gates rather than 4: input gate and forget gate. This approach leads to a smaller number of parameters per block comparing to LSTM.

I decided to enrich the experiments a bit. With generated data, I chose to try three different recurrent cells: RNN, LSTM, and GRU.

Also, the problem formulated as has one major drawback, since word dictionary can be extensive (40k words in my case) classification problem can be hard to learn.

So I thought of a different method, which is to use as a target, not the word itself but its embedding in the k dimensional space such as fasttext and reformulated classification problem as regression one. This approach can reduce the output vector substantially (from 40k to 128 in my case).

For the classification type of model, I used the hidden state size equal to 650, dictionary size $\sim$ 40k words and 93 characters subjected to encoding including alphanumeric and punctuation characters.

For the regression part, the words were subjected to fasttext embedding in 128-dimensional space trained on the same big.txt data. As loss function, I chose SmoothL1Loss because it is less sensitive to outliers than the mean square error loss and in some cases, prevents exploding gradients.

For both cases, 20 epochs are run. For the training process, the early stopping technique was utilized.

IV. CharCNN model

This model was a recreation of the Kim et al. 2015 paper.

The preprocessing step for this method consists of tokenization and then embedding each word as a concatenation of each character embedding (I decided to use one-hot encoding for this step). Also, all word embeddings are padded with zeros to match (maximum word length) x (character embedding size) shape.

As model authors chose the following architecture: convolution layer, then subjected to max-pooling layer, then processed with highway network. After this we have some context vector of the word which we can feed into two stacked LSTMs with fully-connected layer on top, then Softmax is used and CrossEntropy as a loss function since the problem formulated as classification word to word.

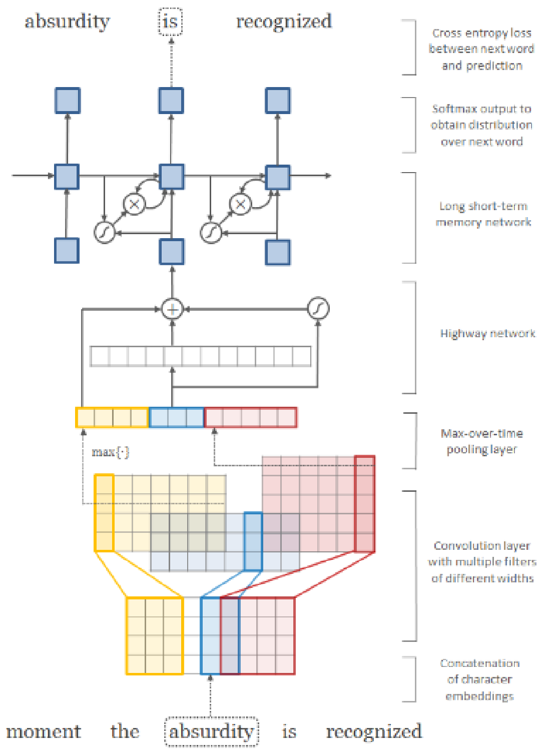
absurdity
is
recognized
Cross entropy loss between next word and prediction
Softmax output to obtain distribution over next word
Long short-term memory network
Highway network
Max-over-time pooling layer
max{·}
Convolution layer with multiple filters of different widths
Concatenation of character embeddings
moment
the
absurdity
is
recognized

Figure 2. Character CNN model structure.

Highway network, proposed by Srivastava et al.(2015) does the following:

$$z = t \odot g(W_H y + b_H) + (1 - t) \odot y \qquad (7)$$

Where g is a non-linearity, $t = \sigma(W_T y + b_T)$ is called the transform gate, and (1 - t) is called the carry gate. Similar to the memory cells in LSTM networks, highway layers allow for the training of deep networks by adaptively carrying some dimensions of the input directly to the output.

For this model, I used the hidden state size equal to 300, dictionary size $\sim$ 40k words and 93 characters subjected to encoding including alphanumeric and punctuation characters. Also, I decomposed 2d convolution into two 1d convolutions to improve training time, trick borrowed from Inception CNNs. In terms of training 20 epochs were run, the early stopping technique was utilized.

V. SEQ2SEQ MODEL

This model was a recreation of the Zhou et al. 2019 paper, which reformulated spell correction problem as machine translation one. This is a quite interesting approach, so I decided to give it a try.

The preprocessing step for this method consists of tokenization and then embedding each word as a byte-pair encoding of substrings. Also, all word embeddings are padded with zeroes to match maximum word length since it may be the longest substring representation.

This model consists of encoder and decoder in conjunction with an attention block. As target byte-pair encoded words used as well and masked cross-entropy loss used as the loss function.

Interestingly enough, this structure solves two significant problems of the previous architectures. First of all, the $\sim$ 40k classification problem produces too large a target matrix, which may be hard to optimize, this problem is partially solved by BPE, since we can use the fixed size of sub-string dictionary (I used 1k dictionary). The second problem is that output size may be different from input size, and this is perfectly reasonable for spell checking problems where concatenations of words are common. This problem is solved by autoencoder architecture.

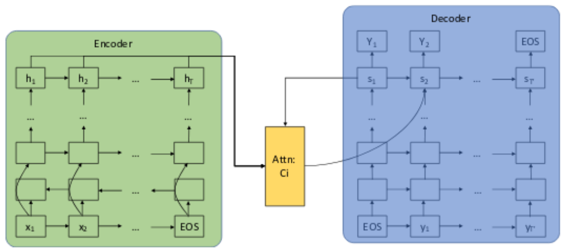

Figure 3. Sequence-to-sequence structure with attention block.

Bahdanau et al. (2014) paper suggest that encoding the whole input string to a single fixed-length vector is not optimal, since it may not reserve all the information that is required for successful decoding. Therefore, authors introduce the attention mechanism from the mentioned paper into this model. Formally, the attention model calculates a context vector c_i from the encoding states $h_1, ..., h_T$ and decoder state s_{i-1} by

$$c_i = \sum_{j=1}^{(} T)\lambda_{ij} h_j \qquad (8)$$

$$\lambda_{ij} = \frac{exp(a_{ij})}{\sum_{k=1}^{(} T)exp(a_{ik})} \qquad (9)$$

$$a_{ij} = \tanh(W_s s_{i-1} + W_H h_j + b) \qquad (10)$$

For this model, I used the hidden state size equal to 300, dictionary size 1k sub-words. Masked CrossEntropy loss is masked in terms of not considering padding tokens. Also, I used bpemb package to byte-pair encode my input.

For this experiment, I decided to decompose the system and try it out with and without attention.

In terms of training ten epochs were run, early stopping technique was utilized.

VI. Results

The results of the experiments are the following:

Model	Modification	Test accuracy
scRNN	classification + RNN block	90.5%
scRNN	classification + LSTM block	91%
scRNN	classification + GRU block	92.7%
scRNN	regression + RNN block	46.1%
scRNN	regression + LSTM block	53%
scRNN	regression + GRU block	49.8%
CharCNN	-	72%
seq2seq	without attention	7%
seq2seq	with attention	97.6%

Table I
Test results for developed spell correction techniques

Models were tested by sentences, where third of the words were contaminated, and among those words, each for the character was interchanged, deleted, or inserted a new one with 40% probability, a sequence which can be considered as a good benchmark.

VII. Conclusions

The results suggest that for keyboard mistypes problem, seq2seq outperforms every other model tremendously. The interesting fact is that the key part is attention block since without it autoencoder performs just at 7% accuracy level.

I should add that seq2seq models require more time to train, and they are larger and therefore have a bit more computation to produce a result, thus longer reaction time. Never the less, the performance is impressive.

Regarding the CharCNN model, we can see quite mediocre results, without apparent advantages.

Now, the semi-character RNN as classification performs very well. The differences between the models are not substantial, which is quite interesting since RNN performs at a similar level to LSTM, which suggests that probably long term dependencies do not have a significant influence on this problem. Another finding is that proposed regression models perform poorly, probably due to the density of embeddings, which creates difficulties in coming back to word representation.

This project was super educational and fun!

References

[1] Danylo Kolinko, ”Github: Comparing spell correction approaches”, `https://github.com/Kolinko-Danylo/DL-approaches-for-spell-correction`

[2] Peter Norvig, ”How to Write a Spelling Corrector”, 2007 `https://norvig.com/spell-correct.html`

[3] Edward Ma, ”Github: nlpaug”, `https://github.com/makcedward/nlpaug`

[4] Joulin, Armand and Grave, Edouard and Bojanowski, Piotr and Mikolov, Tomas, ”Bag of Tricks for Efficient Text Classification”, 2016 `https://arxiv.org/pdf/1607.01759.pdf`

[5] Benjamin Heinzerling and Michael Strube, ”BPEmb: Tokenization-free Pre-trained Subword Embeddings in 275 Languages”, 2018

[6] Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua, ”Neural machine translation by jointly learning to align and translate”, 2014 `https://arxiv.org/pdf/1409.0473.pdf`

[7] Kim, Jernite, Sontag, Rush, ”Character-aware neural language models”, 2015 `https://arxiv.org/pdf/1508.06615.pdf`

[8] Sakaguchi, Duh, Post and Van Durm, ”Robsut Wrod Reocginiton via Semi-Character Recurrent Neural”, 2017 `https://arxiv.org/pdf/1608.02214v2.pdf`

[9] Srivastava, Greff and Schmidhuber, ”Training Very Deep Networks”, 2015 `https://arxiv.org/pdf/1507.06228.pdf`

[10] Zhou, Porwal and Konow, ”Spelling Correction as a Foreign Language”, 2019 `https://arxiv.org/pdf/1705.07371.pdf`