Correction of defective words

Sangrin Lee

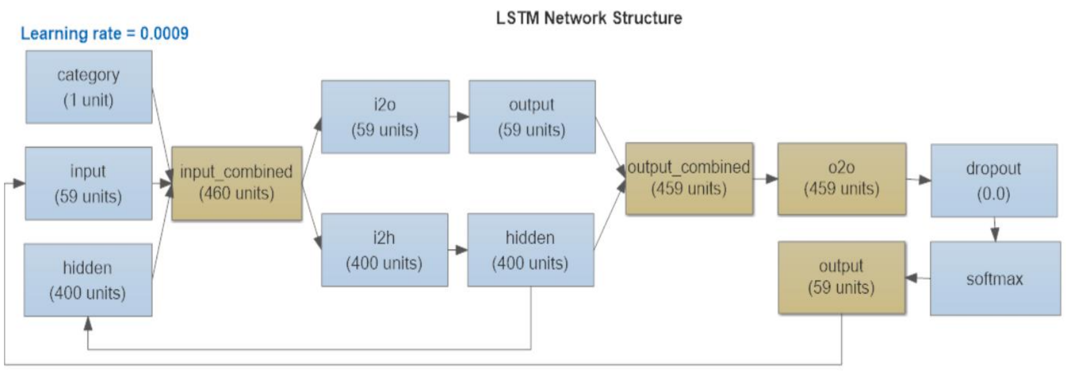
sangrinlee2018@u.northwestern.edu

EECS 349. Machine Learning, Northwestern University

My primary task is to predict the missing character(outputs) from the defective words(inputs) which has 'blackdot' character such as C*mputer(missing 'o'), Scie*ce(missing 'n'), M*chine(missing 'a'), and Le*rning(missing 'a'). This is quite important in a sense that we can fully understand the whole context from incomplete sentence by predicting the missing character and words. To do that, I built a language model by using LSTM(Long Short Term Memory) Networks.

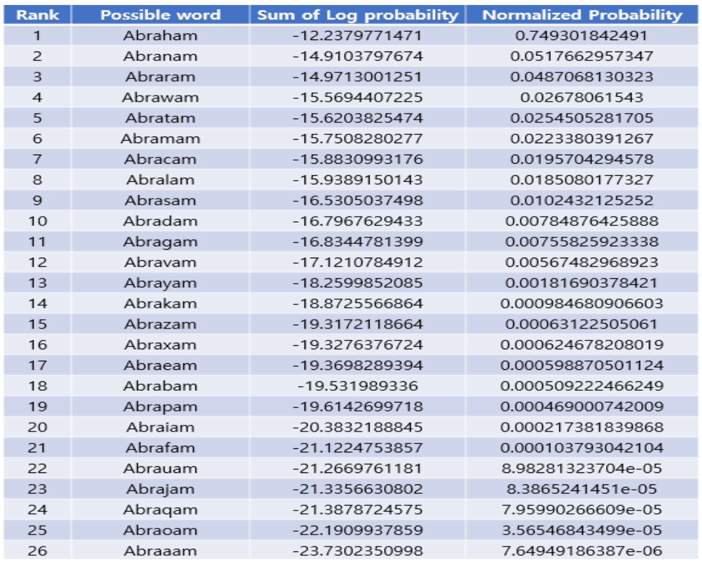
My approach is to build a language model to run inference and estimate the probability of substitutions of missing character using character-based LSTM networks. LSTM is very special kind of recurrent neural network capable of learning long-term dependencies. This language model recurrently predicts characters to form a certain word. With this model, I input a word with missing character, and then output the predicted word.

I collected the dataset from (https://download.pytorch.org/tutorial/data.zip). There are 3668 data which is comprised of large collection of English names such as 'Adam', 'Benjamin', 'Clark', and 'Daniel', and so on. I randomly split the whole dataset, and used 9/10 for training, and 1/10 for testing. For each name in the testing set, I randomly selected one character and then replaced it with asterisk. I used this modified name for testing to check if the model predicts the word correctly. First, I fed all training set into a character-based LSTM model. The LSTM model I built is comprised of four main layers – input combined layer(1 category states + 59 input states + 400 hidden states) with 460 states, output combined layer(400 i2h states + 59 i2o states) with 459 states, o2o layer with 59 states, and output layer with 59 states. 59 input states includes alphabet A to Z, a to z, special characters(.,;'-), and EOS marker. [Figure 1] shows the structure of LSTM Network.



[Figure 1] LSTM(Long Short Term Memory) Networks structure

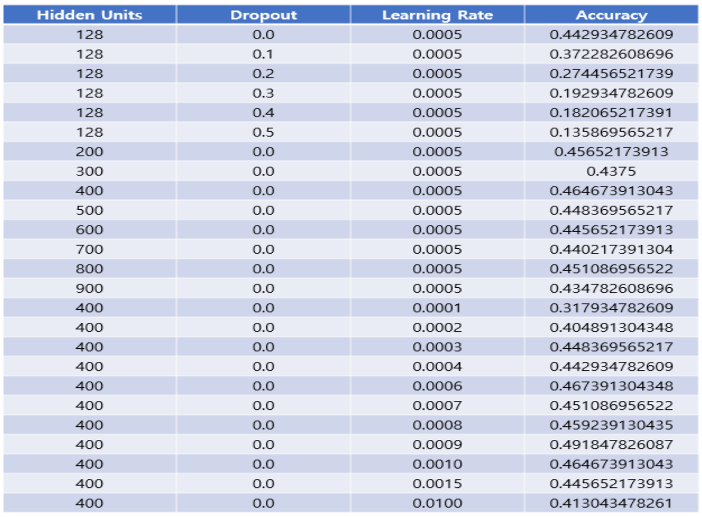
At test time, given a word with asterisk, there are 26 possible substitutions of the word. For example, given the input 'Abrah*m', there are 26 possible words such as 'Abraham', 'Abrahbm', 'Abrahcm', and so on. For each word, I got the probability estimates by summing log likelihood(the base of the logarithm is e) for each character along the sequence. And then, I exponentiated and re-normalized to estimate the probability of each substitution character. The result in the highest probability is the one that I predicted for the output. [Figure 2] is one of the task to predict the word given the input 'Abraham' using a character-based LSTM. In this case, the fifth character ('h') is randomly selected to be replaced with the asterisk. For each possible word, it calculates probability estimates. As seen in the figure below, 'Abraham' has the highest probability with 0.749301842491 among all other possible words.



Rank	Possible word	Sum of Log probability	Normalized Probability
1	Abraham	-12.2379771471	0.749301842491
2	Abranam	-14.9103797674	0.0517662957347
3	Abraram	-14.9713001251	0.0487068130323
4	Abrawam	-15.5694407225	0.02678061543
5	Abratam	-15.6203825474	0.0254505281705
6	Abramam	-15.7508280277	0.0223380391267
7	Abracam	-15.8830993176	0.0195704294578
8	Abralam	-15.9389150143	0.0185080177327
9	Abrasam	-16.5305037498	0.0102432125252
10	Abradam	-16.7967629433	0.00784876425888
11	Abragam	-16.8344781399	0.00755825923338
12	Abravam	-17.1210784912	0.00567482968923
13	Abrayam	-18.2599852085	0.00181690378421
14	Abrakam	-18.8725566864	0.000984680906603
15	Abrazam	-19.3172118664	0.00063122505061
16	Abraxam	-19.3276376724	0.000624678208019
17	Abraeam	-19.3698289394	0.000598870501124
18	Abrabam	-19.531989336	0.000509222466249
19	Abrapam	-19.6142699718	0.000469000742009
20	Abraiam	-20.3832188845	0.000217381839868
21	Abrafam	-21.1224753857	0.000103793042104
22	Abrauam	-21.2669761181	8.98281323704e-05
23	Abrajam	-21.3356630802	8.3865241451e-05
24	Abraqam	-21.3878724575	7.95990266609e-05
25	Abraoam	-22.1909937859	3.56546843499e-05
26	Abraaam	-23.7302350998	7.64949186387e-06

[Figure 2] Probabilities of each possible words given the input 'Abraham',

There are a couple of methods that could determine the performance of the model. The one that I tried was hyperparameter optimization. Depending on the hyperparameters such as the number of hidden states, dropout, and learning rate, the performance of the model varies. [Figure 3] shows how the hyperparameters affect the accuracy of the model.



Hidden Units	Dropout	Learning Rate	Accuracy
128	0.0	0.0005	0.442934782609
128	0.1	0.0005	0.372282608696
128	0.2	0.0005	0.274456521739
128	0.3	0.0005	0.192934782609
128	0.4	0.0005	0.182065217391
128	0.5	0.0005	0.135869565217
200	0.0	0.0005	0.45652173913
300	0.0	0.0005	0.4375
400	0.0	0.0005	0.464673913043
500	0.0	0.0005	0.448369565217
600	0.0	0.0005	0.445652173913
700	0.0	0.0005	0.440217391304
800	0.0	0.0005	0.451086956522
900	0.0	0.0005	0.434782608696
400	0.0	0.0001	0.317934782609
400	0.0	0.0002	0.404891304348
400	0.0	0.0003	0.448369565217
400	0.0	0.0004	0.442934782609
400	0.0	0.0006	0.467391304348
400	0.0	0.0007	0.451086956522
400	0.0	0.0008	0.459239130435
400	0.0	0.0009	0.491847826087
400	0.0	0.0010	0.464673913043
400	0.0	0.0015	0.445652173913
400	0.0	0.0100	0.413043478261

[Figure 3] Accuracies depending on hyperparameters

Compared to other RNN network, LSTM network is explicitly designed to avoid the long-term dependency problem, and has four linear layers instead of having a single neural network layer. These layers interact in a very special way letting the information through.
The best hyperparameters depends on the problem at hand as well as the structure of the model. First, the dropout generally reduces overfitting in neural networks. However, it doesn't help in this model because the dataset is not large enough to drop out units. Regarding the number of hidden states, small number of hidden states might lead to high error as the predictive factors might be too complex for a small number of units to capture. On the other hand, large number of hidden states might overfit to the training data and not generalize well. By the accuracy testing above, the model performs best when the number of hidden states is 400. Lastly, Larger learning rate might overshoot the optimal point, on the other hand, small learning rate might focus on a particular local optimum. So, selecting the best learning rate affects the performance of the model. In this case, the model performs best when the learning is 0.0009.

Apart from tuning the hyperparameters, there are more ways to increase the performance of the model. The number of epochs that I haven't mentioned before might affect the performance of the model. Also, with more dataset, this language model will be able to perform well. For the future work, this language model might be used in a wide range of areas such as transcribing the old books, spelling checks, and word automation services.

* Machine Learning Software Packages
Machine learning software package I used is PyTorch, which provides tensor computation, and deep neural networks built on a tape-based autograd system. The specific component I utilized in PyTorch is torch(a tensor library like numpy, with strong GPU support), torch.autograd(a tape based automatic differentiation library that supports all differentiable tensor operations in torch), and torch.nn(a neural networks library deeply integrated with autograd designed for maximum flexibility).

* Main Task
- Process the dataset for training(9/10) and testing(1/10)
- Build a language model using LSTM(Long Short Term Dependency) Networks
- Evaluate the performance of the model
- Create the website