Solving Text Imputation Using Recurrent Neural Networks

Arathi Mani
Department of Computer Science
Stanford University
Stanford, CA 94305
arathim@stanford.edu

Abstract

This paper investigates various solutions to the challenge of processing sentences with exactly one word missing and determining where in the sentence the word is missing and also figure out the correct word to insert in the predicted location. We train multiple flavors of Recurrent Neural Network Language Models (RNNLM) on a large corpus of English sentences and use the trained model to predict the likelihood of possible sentences with a new word inserted in the missing data corpus. We show that the bidirectional RNN has the best performance on a dataset that was limited to sentence that had between 15 and 20 words, reporting a Levenshtein distance of 5.315 on the test set and an overall perplexity score of 211.312.

1 Introduction

Text imputation is the process of inserting words into incomplete sentences and is a rather challenging task due to the dual nature of each solution: one must predict the location of the missing work and then pick the appropriate vocabulary word to be inserted into that location. Such a challenge has various applications such as auto-completion of sentences (or even search term fragments) or the completion of ancient texts where we may be missing parts of the text which have been lost over time.

This task was derived from a Kaggle challenge which provided a corpus of sentences each with one word missing; the original dataset is a collection of sentences using one billion words curated by Chelba et al.. Kaggle divided this dataset up into a training, development, and test set where for the development and test set, exactly one word was removed from a random location. The only restriction on the location was that the location was neither the first word nor the last word in the sentence.

Given this dataset, we would like to see how effective recurrent neural networks are at the combination of the two tasks. We train three RNNs for this paper and use the language model to predict the likelihood of predicted sentences, outputting the set of sentences that results in the minimum cross entropy error. We use two metrics, perplexity and Levenshtein distance, as measures of our success. Levenshtein distance is also the metric that the original Kaggle challenge uses which gives us a comparison of our success against some of the other solutions that may not have used deep learning.

However, one caveat that we had to consider for our work was that the Kaggle dataset turned out to be too huge of a corpus for us to run the RNN models. In fact, the predicted time to train the RNN on the entire dataset was two weeks and entire line of research was spawned from this challenge on scaling RNN implementations to utilize GPUs. To combat this issue, we pruned our dataset down to 10,000 sentences using an 80-10-10 training-development-test split. The result was that the Levenshtein distance could not be accurately compared to that of the baseline (5.55211), but we

could compare our models to each other to see which one had the highest success at this task and extrapolate to suggest future edits that might result in even higher accuracies.

2 Related Work

As the task for this paper was derived from a Kaggle challenge from the previous year, several approaches had been discussed in the forums with regards to the solution. One possible approach to lower the perplexity of the one billion word dataset was to train using an interpolated KN 5-gram model using 100 CPUs, resulting in a perplexity score of 243.2.

Another model that was submitted by one of the challenge participants involved using a simple n-gram model (of up to length 6) combined with the Stanford POS tagger to determine the most likely location for the missing word. This participant also used a linear regression to predict the edit distance using features such as the gap between the score of the highest predicted candidate sentence and the next highest predicted candidate sentence to guess a word of the approximate correct length.

Microsoft recently also had a sentence completion challenge where the company curated a list of 1040 sentences each of which has four words that are not supposed to be a part of the sentence, yet share a similar occurrence statistic to the actual word. The model must then choose one of five words to correctly replace the ”imposter” words. The baseline results using n-gram models resulted in 34 to 39 percent accuracy (the latter number included smoothing) while a human had 91 percent accuracy. A standard RNN has also been tested on the dataset resulting in a slightly higher 42 percent accuracy.

With regard to other applications of missing data prediction, others have experimented with the idea of using deep learning to tasks such as traffic data imputation achieving varying degrees of success. Duan et al. shows that their calculated error was less than 25 percent of the maximum reported error score for 90 percent of their varying missing rates.

3 Models

We will train three different neural network models for this project to determine which shows the greatest accuracy in predicting both the correct location of the missing word and its success in imputing the correct vocabulary word into the predicted location.

Each of the neural networks will essentially perform the same task. First they will train a language model using the training dataset. The hyperparameters will then be tuned for each model using a 1000 sentence development set before testing on the final curated test set.

For the prediction task, essentially the model will run through around O(2000N) sentences (where N is the number of words in the sentence and 2000 is the size of our pruned vocabulary) to determine which sentence results in the lowest cross entropy error. For each sentence in the training data, each of the possible words in the vocabulary will be inserted into the possibly locations in the sentences. For example, if we have the sentence ”What do you of her reaction ?” (taken from the test dataset), and for brevity, a vocabulary solely consisting of the word ”think”, the following sentences' probabilities will be evaluated:

”What think do you of her reaction ?”
”What do think you of her reaction ?”
”What do you think of her reaction ?”
”What do you of think her reaction ?”
”What do you of her think reaction ?”
”What do you of her reaction think ?”

We know that the word will not be the first token (space delimited) of the sentence nor the last token of the sentence, enforced as rule from the original Kaggle challenge. The sentence with the highest probability will be outputted as the most likely sentence containing the correct word in the correct location which will then be used to compute the average Levenshtein distance.

A random model was also implemented as a baseline measure where one random location was chosen and a vocabulary word was chosen based on the distribution of its frequency in the training corpus.

3.1 Vanilla Recurrent Neural Network

The first model that will be used to train the sentences will be a baseline Recurrent Neural Network. This model has just one hidden layer with size 100 (the word vectors as input to all the models also have a dimension size of 100). Each word in a sentence has its own 100 dimension word vector and the hidden and output layers are computed using the following equations respectively:

$$\begin{aligned} h_t &= \sigma(W \cdot h_{t-1} + Lx^{(t)}) \\ \hat{y}_t &= softmax(U \cdot h_t) \end{aligned} \tag{1}$$

The sigmoid function is represented by the sigma for the hidden layer and W and U are the corresponding weight matrices for each of the layers.

3.2 Deep (2-layer) Recurrent Neural Network

The second model is identical to the first model except for the fact that we introduce a second hidden layer. The motivation behind adding a second layer was to move farther away from the raw data to possibly capture unusual semantic patterns in the corpus that perhaps a single hidden layer cannot capture. We modify the equations previously used by adding two additional weight matrices and using only the second hidden layer's output to compute the final output. Again, the sigma represents the sigmoid function:

$$\begin{aligned} h_t^{(1)} &= \sigma(W^{(1)} \cdot h_{t-1}^{(1)} + Lx^{(t)}) \\ h_t^{(2)} &= \sigma(W^{(2)} \cdot h_{t-1}^{(2)} + V \cdot h_t^{(1)}) \\ \hat{y}_t &= softmax(U \cdot h_t^{(2)}) \end{aligned} \tag{2}$$

3.3 Bidirectional Recurrent Neural Network

The final model that was implemented and trained was the bidirectional recurrent neural network. The motivation behind using this model was the understanding that the missing word could appear in any location so the words succeeding the location of the missing word might have a significant impact on determining the correct word. For example, one could see that solely by looking forward, a sentence like "she into the house" could end up having a solution of "she ran into the house" using the vanilla RNN, but a more appropriate (and more likely) sentence like "she moved into the house" could be predicted by the bidirectional RNN which could understanding that moving is an associated verb with houses which appears only after the missing word location.

The equations to compute the output follows where the arrow indicate the direction of the previous hidden layer node:

$$\begin{aligned} \overrightarrow{h_t} &= \sigma(\overrightarrow{W} \cdot \overrightarrow{h_{t-1}} + Lx^{(t)}) \\ \overleftarrow{h_t} &= \sigma(\overleftarrow{W} \cdot \overleftarrow{h_{t-1}} + Lx^{(t)}) \\ \hat{y}_t &= softmax(U \cdot [\overrightarrow{h_t}; \overleftarrow{h_t}]) \end{aligned} \tag{3}$$

4 Results

4.1 Evaluation Methodology

For this paper, the original dataset provided by Kaggle had to be vastly pruned as the computation time for training RNNs on around 5GB of data would take approximately two weeks. Instead, we used a corpus of around 10,000 sentences split such that the training set had 8,000 sentences and the development and test set each had 1,000 sentences. We also added an additional constraint to limit the number of word in each of the sentences so sentences with five to ten words were separated from sentences with ten to fifteen words which were separated from sentences with fifteen to twenty words. The motivation behind this was based off of a forum post on how the aggressiveness of the vocabulary prediction had to vary based on the length of the sentence; sentences with longer words

had to be more conservative in their guessing since the approximate word length was the same as shorter sentences but their scope was much more fine tuned that a word missing from a very short sentence. Each of the three models listed below were trained on the three different training sets separately and tested separately on the test sets.

The Kaggle competition uses *Levenshtein distance* to compute the accuracy of the prediction of the test sentences. This distance is computed by counting the minimum number of single character edits, which could be substitutions, deletions, or insertions, that is required to turn one string into another. For example, the words "rat" and "rod" have a Levenshtein distance of 2. This metric does an accurate job of reporting the difference between the two strings since it both penalizes the incorrect placement of the word and the difference in the choice of word without being skewed due to the offset. For example, if the model predicts the sentence "the car very drove fast" for the actual sentence "the car drove very fast", the entire second half of the sentence starting from the words succeeding "car" are not all penalized; only the deletion and insertion of the word "drove" will be reflected thus giving credit for choosing the correct missing word.

We will also use *perplexity* as a metric to determine how well each language model is working. Perplexity will act as an indicator to how well each language model is able to learn the training data and generate sentences in the prediction step. We use the cross entropy error to from forward propagation to compute this metric using the following equation:

$$PP^{(t)}(\hat{y}^{(t)}, y^{(t)}) = exp(J^{(t)}(\theta)) \tag{4}$$

4.2 Evaluation Results

Table 1: Results of sentences size 3 - 10

MODEL	AVERAGE LEVENSHTEIN	PERPLEXITY
Baseline	6.674	N/A
Random	5.554	N/A
Vanilla RNN	4.967	234.759
Deep RNN	4.892	230.591
Bidirectional RNN	4.513	227.247

Table 2: Results of sentences size 10 - 15

MODEL	AVERAGE LEVENSHTEIN	PERPLEXITY
Baseline	6.883	N/A
Random	5.661	N/A
Vanilla RNN	5.38	220.940
Deep RNN	5.442	219.19
Bidirectional RNN	5.262	213.444

Table 3: Results of sentences size 15 - 20

MODEL	AVERAGE LEVENSHTEIN	PERPLEXITY
Baseline	6.935	N/A
Random	5.547	N/A
Vanilla RNN	5.428	223.211
Deep RNN	5.399	219.488
Bidirectional RNN	5.315	211.312

4.3 Results Discussion

The vocabulary was created by a simple counting algorithm over the training data and determining the top 2000 most frequent words. We then modified the gold version of the development and test data so that any word that was removed, but not in the vocabulary, was replaced using the unique token "UUUNKKK." We did this so that we were not penalized for sentences for which the word was impossible to impute. The result of this process was that about 15 to 20 percent of the sentences had the word "UUUNKKK" as their correct missing word.

For the vanilla and bidirectional RNN, we used a learning rate of 0.01 while for the deep RNN, a learning rate of 0.1 was used. For the dataset with sentence size 5-10, we used 10 epochs, while for the other two datasets, only 5 epochs were used. We noticed that learning the structure of smaller sentences that were peppered with many "UUUNKKK" replacements was more difficult to train than a dataset with a more words (thus increasing the probability that the removed word was a common word and therefore found in our limited vocabulary) and so fewer "UUUNKKK" cases had to be trained.

The baseline was computed using the raw missing data test corpus and determining the Levenshtein distance if there were no words predicted. The fact that the Levenshtein distance is around 7 for each of the datasets tells us that on average, the missing word has around 7 characters and that increasing the number of words in the sentence generally indicates that the word will have a higher number of characters.

We do not compute a perplexity score for the baseline or random models since the perplexity score is based on the idea that we have trained a model to give us a cost score that tells us how likely a sentence is to be generated by the trained model.

From the tables above, we can see a general trend that the Vanilla RNN provided significant gain over the Baseline and Random models. Also, the general trend for the Deep RNN was that it usually showed a slight improvement over the plain RNN, but usually not by much and the gain was significant for the longer sentences. Finally, the Bidirectional RNN yielded the best performance resulting in a lower perplexity score and a lower Levenshtein distance score for all three datasets.

Perhaps unsurprisingly, the sentences that had the best performance were ones where the missing word was actually "UUUNKKK". Although the word was not present in the training data, all three models for the two smaller sentence length datasets (the 15 to 20 word sentences frequently did not need to resort to using "UUUNKKK") were able to translate unfamiliar vocabulary from the training data into "UUUNKKK" and usually successfully complete the sentence. An example of a correct sentence that successfully was completed from the 3 - 10 test set was "players and staff are running out of UUUNKKK ." One clear pattern from analyzing the RNN and the deep RNN versus the bidirectional RNN was the former two's success in predicting the last word. Whenever the last word was the one missing from the gold set, we found that the model was good at predicting the concluding word. Other interesting examples that the model did a good job of predicting was any punctuation that appeared in the middle of the sentence. One example is the following: "no , ! ! ! ... why ? ." The model did make a mistake in that the placement of the "word" "..." was incorrect (the gold version was "no ! ! ! ... why ? ... .") but it was impressive to see the ellipses correctly identified among the 2000 words in the vocabulary only 18 of which were special characters and of which one was even the word ".....".

Common errors that were seen in the first two models were due to the limitation of numbers in the vocabulary. Many sentences had numbers which, while generalized to only the pattern using "DG" as a replacement for each digit, did not appear in the vocabulary so the gold version marked it as UUUNKKK and the model attempted to place an actual number there instead. This error could have had a significant impact on the Levenshtein distance as "DG" and "UUUNKKK" are vastly different in characters thus making it appear as though the model performed poorly with missing numbers.

Another common error appearing in all three models and especially seen among the larger sentences was the incorrect insertions of the comma "word." One possible explanation for this error could be the models understood that the larger 15 to 20 word sentences were frequently comprised of fragments that were delimited by commas and therefore a sentence with an additional comma inserted seemed like a plausible solution.

One interesting pattern that came out of the results was that the benefit of using a deep RNN versus a bidirectional RNN seemed to decrease as the length of the sentence increased. This seems to indicate that longer sentences probably have more complex structures that are being captured well by adding an additional hidden layer and that this additional information is more useful that understand words that succeed a particular point in the sentence. It would be an interesting experiment to see how well a deep bidirectional RNN would do since the combination of the two features would possibly lead to a strong gain in improvement.

The most successful part of the entire set of experiments was successfully predicting the location of the missing word. This can be seen from the table in that the perplexity scores for all three datasets were the scores that exhibited the greatest difference between the models compared to the Levenshtein distance which improved more subtly. Many sentences were obviously completely off-mark in predicting the correct word, such as the following sentence "rubbish burned in the closed , some cars had their windows broken , and police blocked access to roads ." where "closed" was actually supposed to be "street." However, it was a success at least in the aspect that the location of the missing word was correctly predicted. This observation might point us towards the fact that deep learning models may be effective at predicting *where* data is missing, but a more comprehensive and/or complex model might be needed for the prediction itself. The limited size of our vocabulary larger had a factor in these errors, but the fact that the example sentence came from the 15 to 20 word sentence dataset (and that there are several more like it) seems to indicate that the vocabulary is not only the problem.

5 Conclusions

In this paper, we have shown that the bidirectional RNN yields the best Levenshetein and perplexity scores out of the three models tested for our missing data problem where we try to impute a single word into a sentence that is missing exactly one word from an unknown location. While an accurate comparison cannot be made against other work due to our limited data, we were successful at lowering the perplxity score of the dataset compared to the 243.2 reported by Chelba et al. which used a simple 5-gram model and we were able to beat the baseline Levenshtein distance of 5.55211 as reported by Kaggle. We observed that the deep learning models were very good at predicting the location of the missing word while the small size of the vocabulary and perhaps the training data size itself, added a limitation to the success of predicting the missing word itself. The partial success of these experiments points to the conclusion that a deep learning model may be part of the answer to a model that solves text imputation in general and adding a better word (or to generalize, data) prediction model could possibly yield a very accurate and powerful text imputation solver.

5.1 Future Work

One obvious extension of this work would be to train other models, such as varieties of recursive neural networks or LSTMs (long short-term memory), to determine whether they perform better or worse than RNNs. We could also examine the effect of adding smoothing, such as 5-gram KneserNey smoothing, on how well the model is able to predict the location of the missing word in particular.

One of the most prohibitive aspects of this works was simply the sheer size of the original corpus of sentences which contained a vocabulary of one billion words. One could imagine that being able to utilize the entire corpus could possibly lead to significant improvements in the prediction task, but additional aggressive pruning techniques would be required. One possible solution could be to do a two pass prediction task where we first create one model to first predict the location of the missing word and use a second model to predict which word from the vocabulary would be most likely to fit in that location. Using a two pass method means that we could combine various models, such as using a recursive neural network for the missing location prediction and a bidirectional RNN for the word prediction, to see which pair works best.

Finally, it would be interesting to see how well such a model could be applied to corpuses of data that are not limited to English sentences. Text imputation can be generalized to be a "missing data" task and it would be interesting to see how well deep learning models would perform on other types of data like social networks graphs or any survey-based data.

References

[1] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson, One billion word benchmark for measuring progress in statistical language modeling, Tech. Rep, Google, 2013.

[2] Will Williams, Niranjani Prasad, David Mrva, Tom Ash, Tony Robinson, Scaling Recurrent Neural Network Language Models, http://arxiv.org/abs/1502.00512

[3] O. İrsoy and C. Cardie, Opinion mining with deep recurrent neural networks, in Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2014, pp. 720728.

[4] Duan, Yanjie, L. V. Yisheng, Wenwen Kang, and Yifei Zhao. "A deep learning based approach for traffic data imputation." In Intelligent Transportation Systems (ITSC), 2014 IEEE 17th International Conference on, pp. 912-917. IEEE, 2014.

[5] Schuster, Mike, and Kuldip K. Paliwal. "Bidirectional recurrent neural networks." Signal Processing, IEEE Transactions on 45, no. 11 (1997): 2673-2681.

[6] Mikolov, Tom, Stefan Kombrink, Luk Burget, Jan Honza ernock, and Sanjeev Khudanpur. "Extensions of recurrent neural network language model." In Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on, pp. 5528-5531. IEEE, 2011.

[7] Mikolov, Tom, Stefan Kombrink, Luk Burget, Jan Honza ernock, and Sanjeev Khudanpur. "Extensions of recurrent neural network language model." In Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on, pp. 5528-5531. IEEE, 2011.

[8] Socher, Richard, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. "Recursive deep models for semantic compositionality over a sentiment treebank." In Proceedings of the conference on empirical methods in natural language processing (EMNLP), vol. 1631, p. 1642. 2013.

[9] Zweig, Geoffrey, and Christopher JC Burges. The Microsoft Research sentence completion challenge. Technical Report MSR-TR-2011-129, Microsoft, 2011.

[10] Zweig, Geoffrey, John C. Platt, Christopher Meek, Christopher JC Burges, Ainur Yessenalina, and Qiang Liu. "Computational approaches to sentence completion." In Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1, pp. 601-610. Association for Computational Linguistics, 2012.