2016

A Deep Learning Approach to Text Simplification

Wayne Zhang
Bard College

A Deep Learning Approach to Text Simplification

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Wayne Zhang

Annandale-on-Hudson, New York
May, 2016

Abstract

Deep learning is a relatively new area in the field of machine learning, and its full potential has yet to be known. Nevertheless, systems employing deep learning have made significant strides in the computing world - beating professionals in Go, a game previously thought to be too difficult for current computers to solve, accurately describing images and identifying a speaker by voice. The goal of this project is to evaluate how well recurrent neural networks, a variant of deep learning, perform against humans in simplifying single words in sentences. Multiple language models were generated from the recurrent neural network, and the best models were used to rank lexical choices. Predictions made by the network were compared with human-generated lexical rankings.

Contents

List of Figures and Tables

Dedication

I dedicate this project to my family.

Acknowledgments

First, I'd like to extend my thanks to my mentor and friend, Prof. Sven Anderson, without whom this project would have never come to fruition. Thank you so very much.

I would also like to thank all the faculty members in the Computer Science and Japanese departments for making my time at Bard as fun and enlightening as possible. I'd also like thank all my friends, especially Kachun, and family for their unwavering support and encouragement.

Thank you all so much from the bottom of my heart!

1
Introduction

1.1 Background

Natural language processing (NLP) is an important and exciting field in the world of computer science. NLP gives us insight into ways in which languages can be understood by computers. One task that NLP can help with is text simplification.

Text simplification is the process of reducing the complexity of text, while still retaining the original meaning (see Table 1.1.1). This is a very broad field, and it encompasses many other tasks, such as lexical simplification and text summarization. Automating the task of simplifying natural language would bring about great changes in how information is processed and accessed. In addition, it carries many pedagogical benefits - language learners would be able to understand difficult text and teachers would have tools to help them reduce the difficulty of their lectures.

Table 1.1.1: Example of text simplification from [16]

Sentences	
We are 13.7 billion light-years from the edge of the observable universe, That's a good estimate, With well-defined error bars, And with the available information, I predict that I will always be with you. - Simon Singh	We are twelve billion light years from the edge, That's a guess, No-one can ever say it's true, But I know that I will always be with you - Katie Melua

There are many different ways to go about reducing the difficulty of text, targetting for instance [16]:

- Lexis
  - Replace difficult words with simpler synonyms
- Syntax
  - Split a sentence with multiple clauses into shorter ones
- Text Length
  - Get rid of peripheral information
  - Add redundancy to help retain information
- Discourse
  - Reorder information to make it more comprehensible
- Semantic
  - Simplify based on the reader's background knowledge

- Quality
  - Make the sentences more descriptive
  - Correct all misspelled words

Furthermore, many studies have shown that targeting these elements would result in higher literacy and better reading performances [15]. Therefore, automating these processes would undoubtedly increase the literacy rate around the world in a very profound way.

For this project, we want to know if a language model created with a recurrent neural network, after training on simplified sentences, can replace a complicated word with a ranking of words that is comparable to that of humans. If it does perform as well as a human, then it will be a very useful tool in text simplification.

1.2 Language Models in Natural Language Processing

Knowledge of the proper sequence (or pattern) of words is very useful in NLP. What that allows us to do is give estimates of certain sequences words or N-grams, and that is very useful in fields such as speech recognition, machine translation, and also text simplification.

A **language model** is a probability distribution over sequences of words. That is to say, it is the likelihood of a word appearing given the previous n-words seen. As such, the probability of a sentence of length m would be:

$$P(w_1, ..., w_m) = \prod_{\text{i=1}}^{m} P(w_i | w_i, ..., w_{i-1}) \tag{1.2.1}$$

To understand this more clearly, given the sentence *The color of turtles is*, it would be processed by the language model like this:

$$\underbrace{<s>}_{x_1}\ \underbrace{The}_{x_2}\ \underbrace{color}_{x_3}\ \underbrace{of}_{x_4}\ \underbrace{turtles}_{x_5}\ \underbrace{is}_{x_6}\ \underbrace{\quad}_{x_7} \tag{1.2.2}$$

The language model would then return a vector of words along with their respective probabilities for x_7, the highest of which would be the one that is most commonly seen after a sequence involving $x_{7-(n-1)}, ..., x_6$, where n is the N-gram.

1.2.1 N-Gram

Traditionally, language models have been built in N-grams. That means to obtain the probability of a word or a sequence of words we would need to count the relative frequency:

$$P(w_i|w_{i-(n-1)}, ..., w_{i-1}) = \frac{count(w_{i-(n-1)}, ..., w_{i-1}, w_i)}{count(w_{i-(n-1)}, ..., w_{i-1})} \tag{1.2.3}$$

So, for a unigram model, the probability of a word, w_i, would be $count(w_i)$ divided by $\sum_{k=0}^{n} count(w_k)$ (this is the total count of words in the model); for a bigram model, the probability of a bigram, w_i, w_{i-1}, is $count(w_i, w_{i-1})$ divided by $count(w_i)$; and so on for the higher n-grams.

Here we can see the unigram (see Table 1.2.1) and bigram (see Table 1.2.2) counts from the Berkeley Restaurant Project.[1] If a bigram language model was generated from the data, we can see that the language model would, for the most part, capture grammatically-correct syntax (for example, most *to*'s come after *want* and most *eat*'s come after *to*). However, in a unigram model, there is no context so a sentence such as $i\ i\ i$ would be considered by the language model as more likely to appear than $i\ want\ food$.

[1] http://www1.icsi.berkeley.edu/Speech/berp.html

Table 1.2.1: Unigram count from the Berkeley Restaurant Project [10]

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

Table 1.2.2: Bigram count from the Berkeley Restaurant Project [10]

		w_1							
		i	want	to	eat	chinese	food	lunch	spend
w_2	i	5	827	0	9	0	0	0	2
	want	2	0	608	1	6	6	5	1
	to	2	0	4	686	2	0	6	211
	eat	0	0	2	0	16	2	42	0
	chinese	1	0	0	0	0	82	1	0
	food	15	0	150	0	1	4	0	0
	lunch	2	0	0	0	0	1	0	0
	spend	1	0	1	0	0	0	0	0

As stated earlier, to obtain the bigram probability, we would need to divide the bigram counts by the unigram counts. For example, to calculate the bigram probability, $P(want|i)$, we would divide the count of $i, want$ by the count of i. This gives us $\frac{827}{2533}$, which is roughly 0.33. We can see the full bigram probability in Appendix A.

Furthermore, if we increase the value of N in an N-gram language model, we would have a language model that more closely resembles the data it was trained on and would produce sentences that would appear to have come from the original dataset. This is the case because the selections for the next word would increasingly be more constrained as N increases, and the language model would predict that certain words are more likely to appear in the context of other words. Moreover, the possibility of certain combinations of words would be reduced as N increases. To more clearly see how increasing N affects the language model, here is a table of

sentences generated from four N-grams after training on Shakespeare's works:

Table 1.2.1: Sentences randomly generated from four N-grams computed from Shakespeare's works. Examples from [9]

N-Grams	**Sentences**
Unigram	– To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have
	– Every enter now severally so, let
	– Hill he late speaks; or! a more to leg less first you enter
	– Are where exeunt and sighs have rise excellency took of.. Sleep knave we. near; vile like
Bigram	– What means, sir. I confess she? then all sorts, he is trim, captain.
	– Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.
	– What we, hath got so she that I rest and sent to scold and nature bankrupt, nor the first gentleman?
Trigram	– Sweet prince, Falstaff shall die. Harry of Monmouth's grave.
	– This shall forbid it should be branded, if renown made it empty.
	– Indeed the duke; and had a very good friend.
	– Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.
Quadrigram	– King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in;
	– Will you not tell me who I am?
	– It cannot be but so.
	– Indeed the short and the long. Marry, 'tis a noble Lepidus.

Clearly, increasing N would produce a more accurate language model but the maximum N-gram count researchers would typically go to is five-gram[2]. The reason

[2]Google N-gram Viewer: http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

being the zero-frequency problem. As N increases the likelihood of seeing certain N-grams would be zero, and this affects the accuracy of our language model in a negative way. However, just because the model has not seen it, does not mean the N-gram does not exist. So, this begs the question of: how can we predict a sentence, if we have never seen it before?

1.2.2 Smoothing, Interpolation and Backoff

A language model is supposed to be a predictive model, but if it has never seen the word or N-gram before, it can not make an accurate prediction. A way to overcome this is through smoothing. Smoothing is adding a value to an unseen word or N-gram in order to redistribute some probability mass from observed items to those that do not occur. The simplest technique to solve this problem is by Laplace Smoothing.

Laplace Smoothing is adding one to every event (seen or unseen). This is very easy to implement, but the outcome is undesirable. Adding one to every event removes too much probability mass from the more likely events and as a result our language model is inaccurate.

A better technique is called **Good-Turing Smoothing**. Essentially, we assign the probability mass of all events that occur $n + 1$ times to events that occur n times. This discount means that words or N-grams which were never seen before would have the same probability of words or N-grams that were seen once, and this process is repeated $n - 1$ times. Even though this technique is much better than Laplace Smoothing, there are still problems: namely, there is not an event for every n count and the most frequent event might be discounted too much.

Kneser-Ney smoothing is a technique that most modern language models utilize because it outperforms all other smoothing techniques [3]. This technique utilizes two other methods that gives us more accurate information about the data, namely

interpolation and back-off. **Interpolation** is a linear function, and it weights each contribution for a probability. For example, the probability of a trigram using interpolation would be:

$$P(w_n|w_{n-2}, w_{n-1}) = \lambda_3 P(w_n|w_{n-2}, w_{n-1}) + \lambda_2 P(w_n|w_{n-1}) + \lambda_1 P(w_n) \tag{1.2.4}$$

where $\sum \lambda_i = 1$.

The other method, **Back-off**, is a non-linear function, and it predicts a N-gram to be normally distributed, but if there is not enough information, it is allowed to "back-off" to a shorter context. For instance, the probability of a trigram using back-off would be:

$$P(w_i|w_{i-2}, w_{i-1}) = \begin{cases} P(w_i|w_{i-2}, w_{i-1}), & \text{if } c(w_{i-2}, w_{i-1}, w_i) > 0; \\ \alpha_1 P(w_i|w_{i-1}), & \text{if } c(w_{i-1}, w_i) > 0; \\ \alpha_2 P(w_i), & \text{otherwise} \end{cases} \tag{1.2.5}$$

where α's are some fixed constant. We can see that if the count of a N-gram is zero, it would move to a shorter N-gram and multiply it by a certain constant.

Finally, to perform Kneser-Ney:

$$P_{KN}(w_i|w_{i-1}) = \frac{max(c(w_{i-1}, w_i) - \delta, 0)}{c(w_{i-1})} + \lambda \frac{|\{w_{i-1} : c(w_{i-1}, w_i) > 0\}|}{|\{w_{j-1} : c(w_{j-1}, w_j) > 0\}|} \tag{1.2.6}$$

$$\lambda(w_{i-1}) = \frac{\delta}{c(w_{i-1})} |\{w' : c(w_{i-1}, w') > 0\} \tag{1.2.7}$$

where λ is a normalizing constant and δ is a fixed discount value [6]. This technique basically asks questions about the lower-ordered N-grams. For example, *Francisco* might be a common word, but it only comes after *San*. However, if we used some of the other models, *Francisco* would be likely to appear after words that are not *San*, which would not make much sense. Kneser-Ney fixes this problem because it investigates in what context *Francisco* appears in.

1.2.3 Word Embedding

Word embedding is a technique used in language modeling to map words to vectors. This is a very important step in creating a language model because words provide little to no information for computers. Moreover, representing words as unique, discrete ids usually leads to data sparsity, and a large amount of data is needed to overcome that. Mapping words to vectors is a way to solve this problem, and there are many techniques to do this, the most simple being one-hot representation.

One-hot representation of a word is essentially a vector filled with 0s, except for a single 1, which corresponds to one position in the dictionary. The main drawback of this technique is that it does not capture semantics. So words like *hotel* and *motel* would be embedded as different entities, even though they have many similarities. The other more sophisticated methods to embed words are Word2Vec[3] and GloVe[4].

Word2Vec was created by Mikolov et al. [19] and there are two ways in which this model can embed words. The first algorithm is the skip-gram model, and this model predicts the context of a word, with a window of size n, given that word. The other model, CBOW[5], does the exact opposite. It predicts the word, based on its context. Typically, these two models output similar vectors (CBOW being slightly more accurate), but the skip-gram model works better with less data, whereas the CBOW model would need more data. This is the case, because of how it is structured - CBOW is conditioned on context and context is larger than words. Furthermore, the skip-gram model would take longer to train than the CBOW model, because learning context takes longer than learning words.

[3] https://code.google.com/archive/p/word2vec/
[4] http://nlp.stanford.edu/projects/glove/
[5] Continuous bad of words

GloVe was created by Pennington, Socher, and Manning [14], and works similarly to Word2Vec using a different approach. In GloVe, a co-occurrence matrix is built for the entire corpus, and then factorized to yield multiple matrices for word vectors and context vectors. Both of these techniques are comparable and the trade-off between GloVe and Word2Vec is more memory versus more time to train [19].

1.3 Recurrent Neural Network

A **Recurrent neural network** (RNN) (Figure 1.3.1) is a special type of neural network. In traditional neural networks (feed forward), the information is sequential, in that the signals travel only one way (from input to output) [1]. This is a limitation which recurrent neural networks overcome. Recurrent neural network introduces limited feedback to the network - this means computations derived from an earlier input are fed back into the network and are considered for future computations. This is clearly very useful in language modeling as there are sometimes long-range dependencies of words in sentences.

The formulas that govern a RNN are as follow:

- X is the sequence of words, and X_t correspond to the one-hot representation of a word.
- The output, O, is a vector of probabilities across our vocabulary for the next word of X_t. The output is the softmax of V_{s_t}

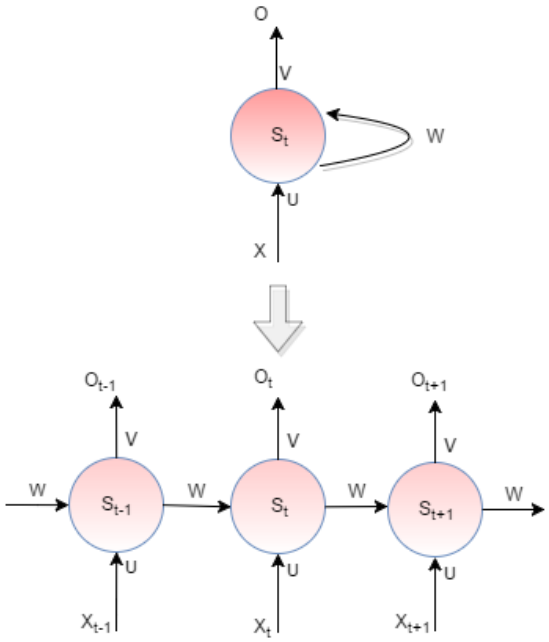


Figure 1.3.1: A model of a recurrent neural network after "unrolling"

- S_t is the hidden layer. It essentially acts as the memory of our model. S_t is calculated by the previous hidden state and the input is: $S_t = f(U_{X_t} + W_{S_{t-1}})$, where f is the activation function.

- U, V, W are our parameters. These variables are how the RNN learns. It will initially be randomly instantiated from $[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$

The **softmax function** is a logistic function, and it essentially assigns numbers between [0,1] to the elements within a vector, and the numbers sum up to 1.

$$h_\theta(x) = \frac{1}{1 + exp(-\theta^T x)} \tag{1.3.1}$$

The **activation function** is a linear associator. This means that if the input is above a certain threshold, it activate, but if it is below, it will not. There are a few activation functions, but for this project *tanh* was used.

$$f(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{1.3.2}$$

1.3.1 Training

Gradient descent is a technique that neural networks utilize to minimize the cost (activation) function. There are many different ways to perform gradient descent and each one has its own advantages and disadvantages. For this project, I implemented mini-batch gradient descent for my neural network. With **mini-batch gradient descent**, our model is updated based on small groups of training samples each of

size k; whereas with stochastic gradient descent (SGD), a variant of gradient descent, only one sample is trained on to update, and with standard gradient descent, all of the training samples are used to update. Mini-batch gradient descent is a compromise between standard gradient descent and SGD, as it allows the network to update and process training examples at a reasonable pace. **Backpropagation** is an algorithm that is used in conjunction with gradient descent. Essentially, as the neural network is training on the data, parameters are updated in a direction that reduces the error. To reduce that error, gradients are fed into a loss function, where it tries to minimize loss. These directions are governed by $\frac{\partial l}{\partial U}$, $\frac{\partial l}{\partial V}$ and $\frac{\partial L}{\partial W}$, where L is the loss function. Usually for a standard RNN, Backpropagation Through Time (BPTT) is used instead of standard backpropagation. Keep in mind that since the parameters are shared by all the timesteps, the network needs to calculate the gradient for all previous timesteps. However, we can use standard backpropagation to avoid long-term dependency problems.

1.3.2 Overfitting, Cross-validation, Regularization

While training, the neural network will undoubtedly encounter a lot of random noise (outliers). This noise is detrimental to our model because it will make our model try to fit a specific sample instead of generalizing for the population. This is the problem of **overfitting**. In Figure **??**, we can see multiple scatter plots, each model with different degrees. The first one shows us a model with a linear line (one-degree polynomial). This is a case of underfitting, and it is just as bad as overfitting, because now our model cannot predict accurately for the training sample. We can see that it leaves out many of the training set's data points.

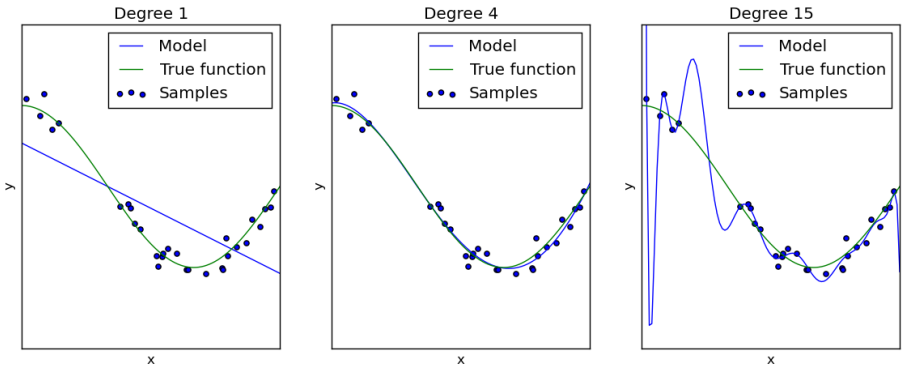


Figure 1.3.2: Example of underftting/overfitting from [17]

The third one captures most of the data-points but the curve is clearly too volatile. It learns too much from the noise. This means that our model will do exceptionally within our sample set, but probably not as well for a novel dataset.

The middle one fits perfectly and matches the true function exactly. The way to obtain that function is to go through a process known as regularization. **Regularization** is a technique that is commonly used to solve the overfitting problem - essentially it tunes our model's complexity to give us better predictions. It is an additional parameter to our model and it is usually used in conjunction with cross-validation to determine the regularization value. **Cross-validation** is a validation technique and is used to access how well our model can generalize. To perform it, we must divide our dataset, and then train it on one set, while validating it on the other. If the error value of the validation set does not decrease or stays the same with the parameters obtained from the training set, then the model will stop training. This is a good technique to prevent our model from over or underfitting.

1.3.3 Evaluation

To evaluate how well a neural network performs, we measure its perplexity. **Perplexity** is how well a probability distribution (language model) can correctly predict

a sample. It is essentially the average branching factor - the number of possibilities. For example, if the language model returns the different possibilities for: *The next color is*, and it returns five different possibilities, each with the same probability, then the perplexity is five. Also since this is a test on itself, this is an intrinsic measure, so it is possible for a model with a higher perplexity to perform better on unseen data than one with a lower score.

Now, to get the perplexity of a language model we use this formula:

$$2^{L(p,q)} = 2^{-\frac{1}{N}\sum_{n\in N} p_n log(q_n)} \tag{1.3.3}$$

$L(p, q)$ refers to the loss function, where in our case is the cross-entropy function. The variable p is the correct label and q is the output given by our network. This is just the sum of how far off our network's predictions are against the real predictions over all the training examples.

1.3.4 Problems

The vanishing gradient, and the equally bad, exploding gradient problems are two problems one encounters when training deep neural networks. The **vanishing gradient problem** occurs when the gradient ceases to have any magnitude (because it is too small). Backpropagation computes gradient by the chain rule, and as the hidden layers increase, the gradient can approach zero.

Similarly, if the gradient gets too large in the earlier layers, the gradient will continue to increase, and this turns into the **exploding gradient problem**. These are not unsolvable problems and variants of RNN have been built to circumvent this, namely long short term memory networks [7] and gated recurrent unit neural networks [4].

1.4 Past Research

Mikolov et al. [13] created a language model from a simple recurrent neural network, and the language model they created performed really well. Their model was trained on Wall Street Journals' dataset and when compared to the state of the art back-off model, it was able to reduce the perplexity by 50%. Furthermore, when tested on NLP tasks such as speech recognition, the model had a 18% reduction of word error rate. Finally, it was able to perform better than the back-off model, even when the back-off model was trained with more data than the recurrent neural network.

Horn et al. [8] conducted an text simplification experiment in which they tested how well their model performs against humans. In their experiment, they built a support vector machine (SVM) that replaces a word with a ranking of the most likely words. Their SVM uses the following features: candidate probability, frequency, language models and context frequency. To evaluate their SVM, they randomly selected 500 sentences from a sentence-aligned Wikipedia corpus. Then, 50 Amamzon's Mechanical Turk (MTurk)[6] workers were assigned per sentence, which gave them a total of 25,000 annotations. When their SVM was compared with the workers from MTurk, the support vector machine achieved a precision of 76% with changes in 86% of the examples.

Xu et al. [18] discussed a problem present in text simplification, namely there has been only one dataset which has been used in text simplification. Simple English Wikipedia has dominated simplfication research and in the paper they argued that Simple English Wikipedia is not really simple, and they present quantitative analysis to compare the NewSela and Simple English Wikipedia datasets. They concluded

[6]https://www.mturk.com/

that the NewSela dataset provides a significant improvement over Simple English Wikipedia in simplification tasks.

2
Datasets

For this project, two different datasets were used to build our language model. They are Simple English Wikipedia[1] and NewSela[2]. These two datasets are made up of simplified sentences, so after generating language models from them, they might perform text simplification as well as humans. A third dataset, the Google Web 1T 5-gram[3], was used to compare how well the language models generated by our neural networks would fare against a unigram model.

2.1 Simple English Wikipedia

The Simple English Wikipedia dataset was obtained using an open-source application known as XOWA[4]. With XOWA, one can download the most recent dump of any version of Wikipedia, and it allows us to view it offline. The most recent dump[5] of Simple English Wikipedia was downloaded, and a slightly-modified Perl

[1] https://simple.wikipedia.org/
[2] https://newsela.com/
[3] https://catalog.ldc.upenn.edu/LDC2006T13
[4] https://gnosygnu.github.io/xowa/
[5] https://dumps.wikimedia.org/simplewiki/20160407/

script[6] was run through the dataset to: strip extraneous text (such as XML tags, HTML tags, URLs, images links, etc.), change all characters to lowercase letters and spell out the digits. See Table 2.2.1 for the basic statistics of the Simple English Wikipedia data.

In terms of numbers of articles, characters, words and unique words, Simple English Wikipedia is a lot larger than NewSela. However, when we calculate the number of words per article it comes out to 296.25, whereas for NewSela, it has 756.83 words/article. Thus, there is more information about the same topic in the NewSela dataset than in Simple English Wikipedia.

2.2 NewSela

The NewSela dataset was provided to me by Professor Sven Anderson who obtained it under agreement from the NewSela team. The dataset had a mixture of Spanish and English articles, both with varying difficulty - 0 being the original (the most difficult), and 5 being the most simplified. A Bash script was used to separate the two languages, and then English articles were separated based on their level of difficulty. See Table 2.2.1 for the basic statistics of the NewSela data.

Within NewSela, the statistics are quite interesting. Looking at the total number of words, it gets progressively smaller as the articles become more simplified, but surprisingly level 1 and level 2 have the same number of words, but different number of unique words. As for the number of characters and unique words, it gets progressively smaller as the articles become simpler. However, for the number of sentences, we can see that it follows a different progression. From the original to level 1, the

[6] http://mattmahoney.net/dc/textdata

number of sentences gets lower but then it increases. The reason that is the case is probably due to the authors splitting sentences to make it simpler.

2.3 Google Web 1T

The Google Web 1T corpus was used as a control to test how significant our neural language models would perform. The Google 1T tracks frequency of N-grams from one to five, but I only used the unigram count for this project. The data was extracted from 1 trillion words of English Web text, and it has 11,052,329 unique words [5].

Table 2.2.1: Statistics of the Datasets

	Datasets							
	NewSela							Wikipedia
	Original	Simp-1	Simp-2	Simp-3	Simp-4	Simp-5	Combined	Simple English Wikipedai
Total # of articles	1,911	1,910	1,910	1,910	1,882	42	9,565	117,213
Total # of words	1,884,854	1,627,084	1,627,084	1,263,034	974,846	14,739	7,239,052	34,724,097
Total # of characters	11,461,124	9,829,735	8,935,539	7,369,650	5,602,857	81,593	43,149,878	197,057,546
Total # of sentences	102,935	98,317	105,920	106,994	99,769	1,870	513,473	
Total # of unique words	127,145	111,540	98,921	79,133	61,303	3,459	166,380	415,680

3
Methods

In this chapter, I will explain how to prepare the data for the neural network, build the recurrent neural network, and train the network. After, we have obtained the language model, we will compare Kauchak's lexical rankings with that of the language models.

3.1 Training Data and Preprocessing

To generate a language model, we first have to modify our datasets for input to the neural network. We first tokenize our sentences. This means all the words and punctuation will be split into individual tokens. To perform this task, we simply use built-in functions of NLTK[1], namely $word\_tokenize$ and $sent\_tokenize$.

Next, the words were ranked by frequency, and the top 8,000 words were placed into a dictionary. All words which were not in the dictionary became $< unk >$. It would have been possible to not change an infrequent word but it is: (1) Not practical - it would take a lot longer to train; (2) The word was seen only a couple

[1] http://www.nltk.org/

of times so the neural network would not be able properly learn it anyway. After that, we want our data to be in a format that can be properly parsed. Sentence start and end tokens were added to provide context.

Finally, we want to create a mapping between words and indices, because this is how our neural network ultimately learns. We want to match an x, a training example, with a y, the corresponding label. For example, x might look like [0,3,7,8,10], where the numbers are the one-hot representation of the words and so dict[0] is the start token, and y would be [3,7,8,10,1], where dict[1] is the end token. The purpose of this language model is to predict the next word, so y is just the x vector shifted by one position. So, the neural network would know that dict[7] comes after dict[3].

3.2 Building a Recurrent Neural Network

Next, we want to build a recurrent neural network model like this:

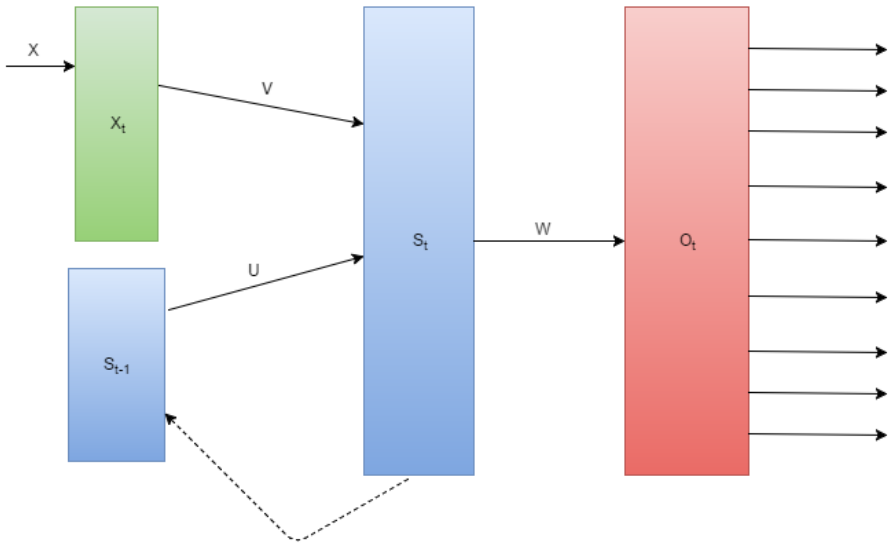


Figure 3.2.1: A model of a simple recurrent neural network

The input X will be a sequence of words, and each X_t will be a single word. Also as stated before, each word is represented as a one-hot vector of the vocabulary's

size. Therefore, a vector at index 5 in the dictionary would be represented as a vector of the vocabulary size, with all 0's and a 1 at position 5.

However, there is a slight difference between the model depicted in the Introduction (Figure 1.3.1), a standard recurrent neural network, and a simple one, the one we are going to create, and that is that a standard one "unrolls". This just means that a standard one BPTT, whereas a simple one only backpropagates.

3.2.1 Backpropagation & Gradient Descent

First, we want to predict the word probabilities, and then see how off we are. So we propagate through the network, by multiplying it with the weights. This returns the network's output and the hidden states which can be used later. Next, we want to calculate the gradients to determine which direction best reduces the loss function.

The loss function of our graph would be the cross-entropy error function:

$$L(y, o) = -\frac{1}{N} \sum_{n \in N} y_n log(o_n) \tag{3.2.1}$$

Essentially, this gives the network an idea of how far off it is.

Now to calculate the gradient we must backpropagate. Essentially, it is a recursive application of chain rule, and it is performed to calculate $\frac{\partial L}{\partial U}, \frac{\partial L}{\partial V}, \frac{\partial L}{\partial W}$. These partial derivatives tell us how sensitive the parameters are on the loss function.

Finally, the gradient is to be updated by mini-batch gradient descent. As discussed in the introduction, With mini-batch gradient descent, it iterates over all our training examples in batches, and after each batch, we nudge the parameter in a direction that reduces the error.

3.2.2 Gradient Checking

Our network is pretty much done now. However, we want to be sure of the correctness of our model. This is accomplished through gradient checking:

$$\frac{\partial L}{\partial \theta} \approx \lim_{h \to 0} \frac{J(\theta + h) - J(\theta - h)}{2h} \tag{3.2.2}$$

The general idea behind gradient checking is that the derivative of a parameter is equal to the slope at a point, which is approximated by slightly changing the parameter. We then compare the gradient obtained to the one from backpropagation.

3.3 Compare Rankings

After training, we should have a fully functional language model. With the language model, we want the probabilities of the words the MTurk workers chose. So for each sentence, we have a subset of words, $w_1, \ldots, w_n$, where n is the number of unique words and w_1 is the most frequent simplification choice and w_n is the least frequent simplification choice.

Next, we want to input the sentence up to the word that was chosen to simplify. So for example, a sentence in Kauchak's dataset is *Photosynthesis is vital for life on Earth.*, and the word that was chosen to simplify is *vital*. We would input *Photosynthesis is.* to our language model.

Finally, we should get a vector of the probabilities of all of the words in our dictionary. We want to search for $w_1, \ldots, w_n$ in the vector and sort it by how likely they will appear. We then replace $w_1, \ldots, w_n$ and the sorted probabilities by $1, \ldots, n$. This can then be used for the Spearman's rank correlation coefficient.

4
Results

Now that we have our language models. We want to choose the one with the lowest perplexity - that is the one with the least branching factor.

4.0.1 Perplexity Scores

Table 4.0.1: Perplexity Scores

Hidden Units	NewSela	Simple Wiki
20	507.523565037	326.523565037
50	507.524534034	325.822353252
100	506.825565068	326.523650372
150	507.235534033	326.321231231
200	505.634742388	313.343223212

After seeing the perplexity scores, the language models that had 200 hidden units in their hidden layer had the lowest perplexity scores.

4.0.2 Spearman's Correlation

Next, we will see how well our language models and the frequency of words that the workers chose fare against the rankings of Kauchak's. The **Spearman's rank cor-**

relation coefficient will be used to compare the neural language models' orderings with Kauchak's. This kind of correlation was used because it is for ordinal numbers, and rankings are ordinal. Furthermore, the purpose of Spearman's rank correlation coefficient is to know if there is a relationship between the two variables.

Here (see Table 4.0.2) we can see the average Spearman's Correlation after comparing all 500 sentences:

Table 4.0.2: Average Spearman's Correlations

Datasets	Google 1T	NewSela	Simple Wiki
Rho	0.468938391081	0.790206143125	0.644574489422

Here are their respective scatter plots:

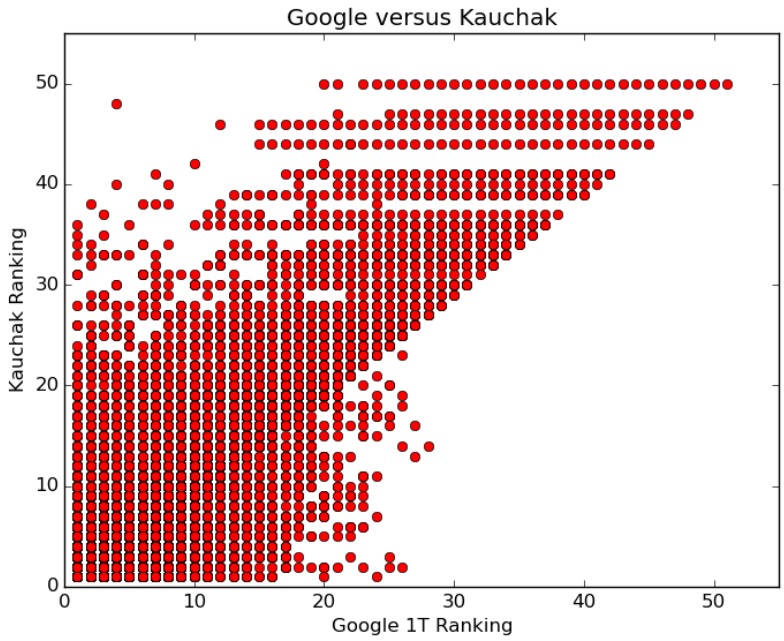


Figure 4.0.3: Google versus Kauchak Correlation

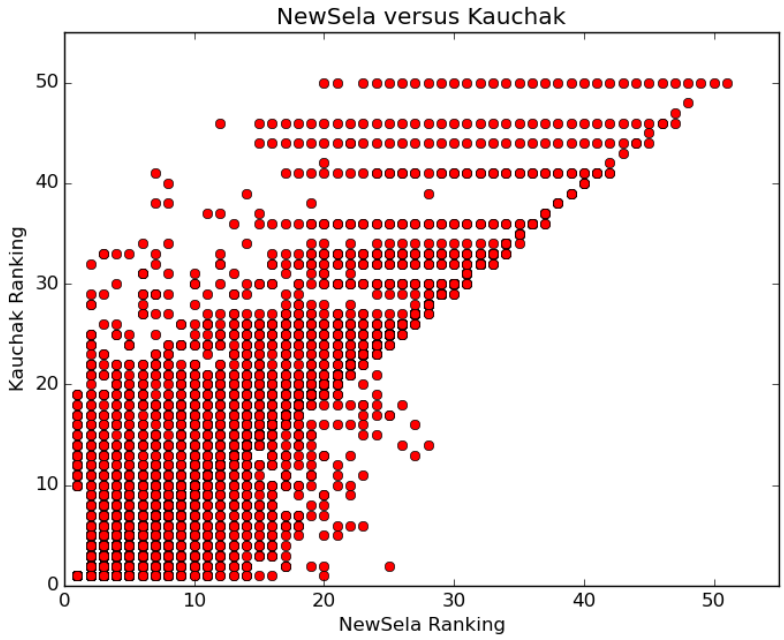


Figure 4.0.4: NewSela versus Kauchak Correlation

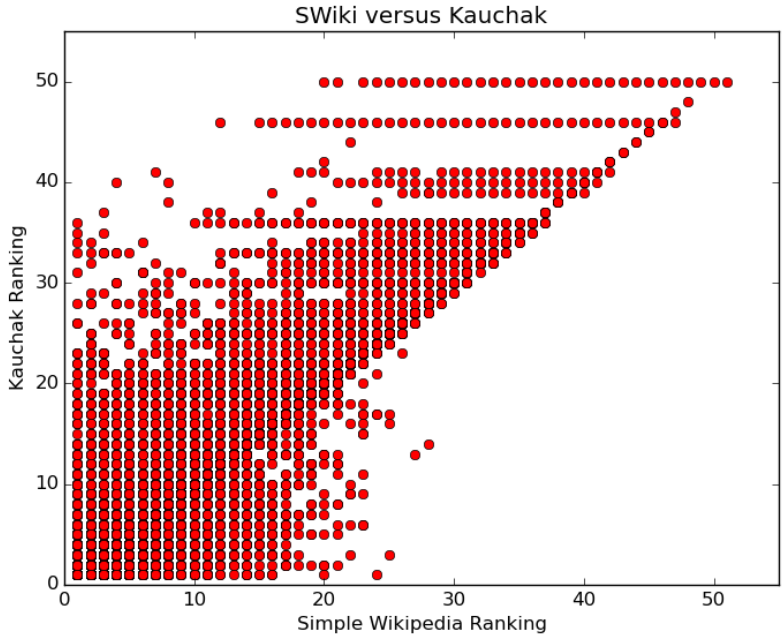


Figure 4.0.5: Simple English Wikipedia versus Kauchak Correlation

Finally, we want to calculate the confidence intervals. A **confidence interval** is an indicator of our measurement's precision and stability. Essentially, it is used to demonstrate the correctness of our results.

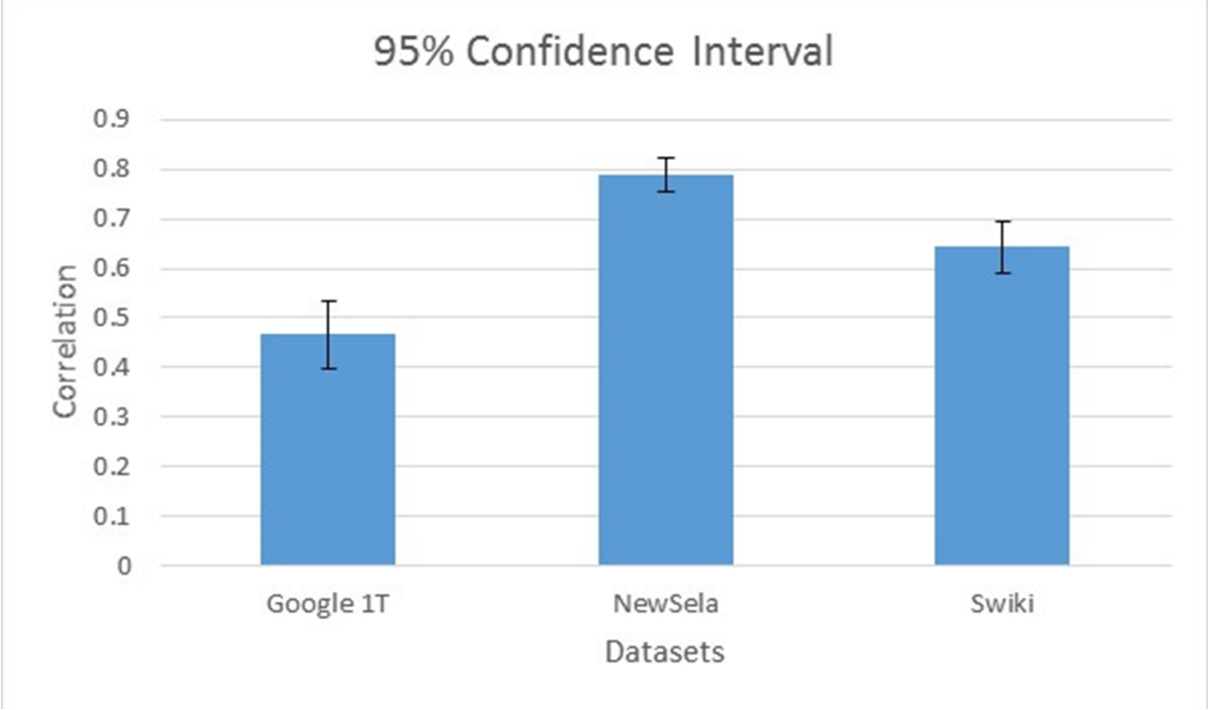
To calculate the 95% confidence interval, we use the formula:

$$[tanh(tanh^{-1}(r) - \delta), tanh(tanh^{-1}(r) + \delta)] \tag{4.0.1}$$

$$\delta = 1.96 * \frac{1}{\sqrt{n-3}} \tag{4.0.2}$$

where r is the correlation, and n is the sample size.

Table 4.0.6: 95% Confidence Intervals



Datasets	Google 1T	NewSela	Simple Wiki
Confidence Intervals	[0.397596,0.534645]	[0.754819,0.821007]	[0.590245,0.693090]

Table 4.0.7: 95% Confidence Intervals

5
Conclusion

After looking at the 95% confidence intervals (Table 4.0.6), we can say that a language model trained from the NewSela data performed statistically better than the control, Google Web 1T, and the other simplfied dataset, Simple English Wikipedia. Also the Simple English Wikipedia language model performed better than Google Web 1T. Thus, we can say that a language model trained with simplified sentences has a strong correlation with human judgement.

Furthermore, Xu et al. [18] were correct that the NewSela dataset would be better in text simplification than Simple English Wikipedia. Table 4.0.6 shows us that NewSela has a statistically better relationship with human judgements than Simple English Wikipedia.

5.1 Language Models

The perplexity scores for my language models were somewhat surprising. When Mikolov et al. (2010) [13] trained their recurrent neural network with similarly sized datasets, their perplexity scores were a lot lower. However, it was reassuring when

the language models' perplexity scores decrease when the size of the dataset or the number of hidden units were increased.

Moreover, the gradient checking algorithm determine that our neural network was learning correctly. Since the algorithm obtained gradients that were similar to those obtained through backpropagation, we know backpropagation was implemented properly.

5.2 Correlation

The correlations of our three datasets were unexpected, as we thought the language models would have a moderate to strong relationship with Kauchak's rankings. However, there seems to be a strong to very strong correlation between NewSela's rankings of words and Kauchak's rankings. Furthermore, for Simple English Wikipedia, there was a moderate to strong correlation with Kauchak's ranking. Finally, for Google Web 1T, there was a weak to moderate correlation, which was the lowest correlation. Clearly, ranking words by its frequency is not the best way to simplify text.

5.3 Future Work

Many improvements to this project are possible. Word2Vec or GloVe could have been used to embed word. Doing so would mean that our neural network would better understand the words, and as a result might rank the words differently.

Also, improvements to the neural network could have been made as well. In Kauchak's dataset, the word that we wanted to simplify was usually in between words. This means that our network would have been able to make better predictions had the network went the other direction in the sentence. For example, one of Kauchak's sentences is: *Photosynthesis is vital for life on Earth.*, and the word

that was chosen to simplify was *vital*. If the network was trained in the opposite direction, it would have had much more context to give a more accurate prediction.

Another improvement to the neural network would be to implement BPTT. This allows the neural network to factor all previous words in a sentence. There are complications such as vanishing and exploding gradients, but recent architectures of RNN have been created to circumvent those problems. A recurrent network which has LSTM or GRU would be able to learn long-range dependencies with no problems.

Lastly, a SVM, like the one Kauchak et al. (2014) created [8], could have been used in conjunction with our language models for even better predictions. In addition to a language model, they used candidate probability, frequency, context frequency.

Bibliography

[1] Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. "A Neural Probabilistic Language Model". In: *The Journal of Machine Learning Research* 3 (2003), pp. 1137–1155.

[2] Britz, D. "Recurrent Neural Networks Tutorial". Accessed: 2016-05-02. URL: `http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/`.

[3] Chen, S. F. and Goodman, J. "An Empirical Study of Smoothing Techniques for Language Modeling". In: *Comput. Speech Lang.* 13.4 (Oct. 1999), pp. 359–394.

[4] Cho, K., Merrienboer, B. van, Gülçehre, Ç., Bougares, F., Schwenk, H., and Bengio, Y. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* (2014).

[5] Evert, S. "Google Web 1T 5-grams Made Easy (but Not for the Computer)". In: *Proceedings of the NAACL HLT 2010 Sixth Web As Corpus Workshop*. WAC-6 '10. Association for Computational Linguistics, 2010, pp. 32–40.

[6] Gauthier, J. "Kneser-Ney smoothing explained". Accessed: 2016-05-02. URL: `http://www.foldl.me/2014/kneser-ney-smoothing/`.

[7] Hochreiter, S. and Schmidhuber, J. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[8] Horn, C., Manduca, C., and Kauchak, D. "Learning a Lexical Simplifier Using Wikipedia". In: *Association for Computational Linguistics*. 2014, pp. 458–463.

[9] Jurafsky, D. and Martin, J. H. *Speech and Language Processing (2nd Edition)*. 2009.

[10] Kalita, J. "N-Grams and Corpus Linguistics". Accessed: 2016-04-26. URL: http://www.cs.uccs.edu/~jkalita/work/cs589/2010/4NGrams.pdf.

[11] Manning, C. D. and Schütze, H. *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.

[12] Mikolov, T., Kombrink, S., Burget, L., Černocký, J., and Khudanpur, S. "Extensions of recurrent neural network language model". In: *The International Conference on Acoustics, Speech and Signal Processing*. 2011, pp. 5528–5531.

[13] Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., and Khudanpur, S. "Recurrent neural network based language model". In: *INTERSPEECH*. 2010, pp. 1045–1048.

[14] Pennington, J., Socher, R., and Manning, C. D. "GloVe: Global Vectors for Word Representation". In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: http://www.aclweb.org/anthology/D14-1162.

[15] Siddharthan, A. "A survey of research on text simplification". In: *ITL - International Journal of Applied Linguistics* 165.2 (2014), pp. 259–298.

[16] Siddharthan, A. "Automatic Text Simplification and Linguistic Complexity Measurements". Accessed: 2016-04-26. URL: https://www.uclouvain.be/cps/ucl/doc/cecl/documents/Siddharthan.pdf.

[17] "Underfitting vs. Overfitting". Accessed: 2016-04-27. URL: http://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html.

[18] Xu, W., Callison-Burch, C., and Napoles, C. "Problems in Current Text Simplification Research: New Data Can Help". In: *Transactions of the Association for Computational Linguistics* 3 (2015), pp. 283–297.

[19] Řehůřek, R. *Making sense of word2vec*. Accessed: 2016-04-26. URL: http://rare-technologies.com/making-sense-of-word2vec/.

Appendices

Appendix A

Bigram probability from the Berkeley Restaurant Project

		w_1							
		i	want	to	eat	chinese	food	lunch	spend
w_2	i	0.002	0.33	0	0.0036	0	0	0	0.00079
	want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
	to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
	eat	0	0	0.0027	0	0.021	0.0027	0.056	0
	chinese	0.0063	0	0	0	0	0.52	0.0063	0
	food	0.014	0	0.014	0	0.00092	0.0037	0	0
	lunch	0.0059	0	0	0	0	0.0029	0	0
	spend	0.0036	0	0.0036	0	0	0	0	0