Make Deep Learning Great Again: Character-Level RNN Speech Generation in the Style of Donald Trump

Benjamin Sherman
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA 95064
bcsherma@ucsc.edu

Zayd Hammoudeh
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA 95064
zayd@ucsc.edu

Abstract

A character-level recurrent neural network (RNN) is a statistical language model capable of producing text that superficially resembles a training corpus. The efficacy of these networks in mimicking Shakespeare, `Linux` source code, and other forms of text have already been demonstrated. In this paper, we show that character-level RNNs are capable of very believably mimicking the language of President Donald J. Trump after training on a corpus of speech transcriptions. We believe our most significant contributions to the study of character-level statistical language models are in sampling methodologies; specifically, we propose that introducing dropout during the text-generation phase introduces randomness that leads to more believable text.

1 Introduction

The generation of natural language text entails not only the ability to produce speech, but also the ability to understand the relationships between words. [9] In addition, natural language takes many very different forms including colloquial, formal, legal, mathematical/scientific, etc. Speech patterns are uniquely individual and are influenced by one's background, motivations, and biases.

Natural language generation (NLG) is an open area of research drawing in corporate heavyweights such as Microsoft with Cortana, Amazon with its Alexa platforms, Apple via its Siri subsidiary, and Google through Android and its "Home" product line. All of these attempts at NLG have focused on creating products that speak in a generic manner with broad customer appeal. By itself, that is an immense task but to an extent sterilizes the speech to be as globally non-offensive as possible.

In this project, we simplify the NLG problem by trying to generate speech in the style of only one person – President Donald J. Trump; our selection of him as our focus is based on a very specific rationale. First, Mr. Trump is a particularly polarizing figure leading to the vast majority of people in the nation being very familiar with his style. In addition, President Trump has many clichéd refrains that are repeated often such "build the wall," "make America great again," "little rocketman," etc. We expect that this repetition will make it easier for a learner to emulate Mr. Trump's speech essence. Furthermore, the president tends to use "highly simplistic" words in a "grammatically awkward" fashion. [3] Therefore, if the generated speech has any defects in structure – grammatical or otherwise – or if it uses infantile language, we expect the audience may attribute these shortcomings to the president himself instead of our tool (especially in a place where Mr. Trump is nearly unanimously derided like Santa Cruz).

[5] demonstrated the surprising effectiveness of character-level recurrent neural networks at generating both natural language and structured text. His work was done in the Lua programming language.

In this project, we attempt to verify his findings with our Trump character-level RNN written in TensorFlow. We provide more background on character-level RNNs in the next section.

1.1 Character-Level RNNs

A character-level RNN is a statistical model of language in the sense that it views the behavior of language probabilistically. To better understand this, allow that my brain uses a statistical language model and that my job is to try to finish all of your sentences. You say "I am going to the grocery-". I would guess with high probability that you are about to say "store", though I have to accept that there is a non-zero chance you will say "outlet"– or, perhaps you will stub your toe and say "ouch!" in the middle of your sentence. What a character-level RNN does is really no different, except that the inferences it learns to make happen on the character level.

Let $\mathcal{C} : v^L \rightarrow \Pi$ be a character-level recurrent neural network, such that v is a vocabulary of approximately 100 characters, L is an arbitrary sequence length, and Π is the set of all probability distributions over v. More plainly, we can think of a character-level RNN as a function that takes a sequence of L characters and gives us a probability distribution; in particular, it gives the probability of the next character given the previous L characters. As an example of how the network will ideally behave, imagine that you let $p = \mathcal{C}(\texttt{(M,a,k,e, ,A,m,e,r,i,c)})$. If the network is well-trained, you should find that p predicts "`a`" as the next character with high probability.

In order to generate meaningful text with a character-level RNN, a seed needs to be provided. This represents the sentence or thought to be finished using the game described earlier. Prompted with the sequence "We will build a g", the RNN will produce a distribution over the vocabulary. We then sample from this distribution and add the resulting character to the sequence. If the sampled character was "`r`", as we would hope, then we now have the sequence "We will build a gr". We can continue this process for arbitrarily many characters. It is important to note that we can not make networks that accept arbitrarily long sequences as input, so at some point we have to start removing a character from the beginning of the sequence for every character we add to the end of the sequence.

1.2 Impracticality of Word-Level RNNs

It is obvious that generating paragraphs of text one character at a time may yield suboptimal results. As part of the feedback to our project proposal, the grader specifically asked why we chose to use a character-level RNN instead of making decisions at the word level. Superficially, a word-level RNN has clear advantages including that it would not produce spelling errors and also that it makes decisions at a coarser granularity, which would be expected to yield superior results. However, upon a more detailed analysis, it is clear that word-level RNNs are impractical.

First, the current Oxford English dictionary has over 170,000 words. [2] A world-level learner would need a separate output node for each of these words (there are optimizations that reduce this output count such as using morphemes but that is no longer a true word-level RNN). Such a large output is likely to suffer from underflow and floating point errors that would severely degrade the quality of its predictions. In addition, training such a large network would be prohibitively long and would extend significantly past the short duration of this project. Even Google with its near limitless computational and human resources does not do word-level prediction.

One team in the CMPS242 class chose to use a word-level learner. To address the output-layer size issue mentioned previously, this team reduced the vocabulary to several hundred words. They also only generated phrases of approximately 10 words or less. Such extreme constraints yield results that significantly underachieve character-level RNNs.

2 Training

This section describes the techniques we employed to train our Trump character-level RNN. Specifically, it outlines the training dataset, the structure of our base neural network as well as implementation-level details including the optimizer, learning rate, and batch size.

2.1 Dataset

A user can provide any text they wish as the seed to a character-level RNN. While some words are substantially more common than others, it does not change the fact that a character-level RNN must be able to generate meaningful outputs from countlessly many input seeds. Therefore, to achieve acceptable performance, the training set needs to be large.

Although President Trump is credited as the principal author of over a dozen books [23, 20, 16, 24, 21, 15, 14, 25, 17, 22, 19, 13, 18], we deliberately excluded all of these texts from the training set for two primary reasons. First, many of the books featured co-authors or were entirely ghostwritten [7]. Consequently, it would be difficult to distinguish Trump's own style from that of his writing partners. Moreover, most of Trump's books were written between the late 1980's and early 2000's. Most of the students in this course had not even been born when these books were written. Hence, the generated text they may yield may not be meaningful to the class's relatively young audience, while Trump's contemporary rhetorical style is uniquely recognizable to almost anyone in the world.

Another possible source of training content are Trump's tweets, but it was better to exclude those from the training set for different reasons. First, Twitter limits tweets to only 144 characters. As such, tweeters deliberately prune content and commentary to fit within this strict size limit. This leads to extensive use of abbreviations, skipping of words, or hashtags (e.g., "#MAGA" for "Make America Great Again") many of which are exclusive to the Twitter platform. For example, one of President Trump's signature Twitter mannerisms is to end a tweet with "Sad!"; however, this is not done in everyday speech even by the president himself. In addition, similar to at least some of Mr. Trump's books, many of his tweets are ghostwritten. It has been reported that at least White House social media director Dan Scavino Jr. [11] and Trump lawyer, John Dowd [12], have authored tweets in Trump's name. These "imposter" tweets risk polluting the data set with non-Trump content.

Given the deficiencies associated with training on tweets or the president's books, we decided to exclusively limit the dataset to public speeches made by Mr. Trump since he announced his bid to run for president on June 16, 2015. Some of these public speeches had already been compiled in a repository [8]. Another database of Trump speeches have been collected in a separate repository [10]. Unlike the first set of speeches which was a static collection in text format, the second set used a web scraper to pull speeches from the University of California, Santa Barbara's presidency project campaign archive [1]. However, the web scraper had significant bugs that corrupted the speech output. We modified the script using Python's `BeautifulSoup4` package to properly extract the content. We then manually merged the two training sets and verified that there were no duplicates.

In total, the training set consisted of more than 115 speeches. There were more than 365,000 words and two million training sequences. We are confident that this training set is more than sufficient for a project of this scope.

2.2 Vocabulary

As with all character-level RNNs, the vocabulary is a set of individual characters that may be received as part of an input sequence or generated as an output. Specifically, the vocabulary consists of all letters – both capitalized and lowercase, digits (0-9), and punctuation (e.g., comma, space, newline, exclamation point, etc.).

The set of characters, v, that comprises the vocabulary is dictated by the training data. In this project, the size of the vocabulary, $|v|$, was 97.

2.3 Learner Architecture

The base neural network architecture we used is similar to the one proposed in homework #5 and is shown in Figure 1; we specifically refer to the training architecture as the "base" since it is a subset of our complete architecture, which is described in Section 3.2.

The training architecture consists of five primary components, namely: the one-hot vector input, embedding matrix, long-short term memory (LSTM) RNN, the feed-forward network, and finally the softmax-output layer.

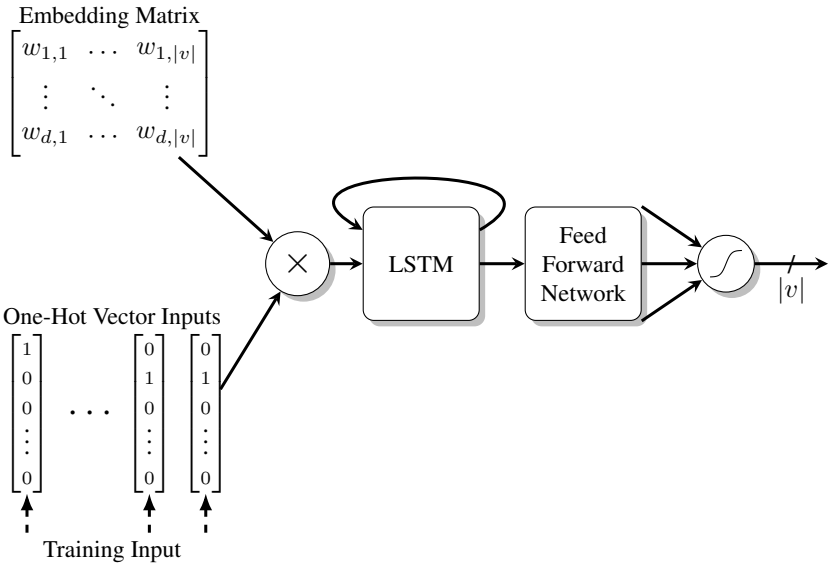

Figure 1: Trump Character-Level RNN Training Architecture

2.3.1 One-Hot Vector Input

As the name indicates, a "one-hot" vector is a zero vector with exactly one element in the vector set to one. Each element in the vector corresponds to a specific character in the vocabulary. Hence, the size of the vector is $|v|$, i.e., the number of elements in the vocabulary. Each training example consists of an ordered series of one-hot vectors and a single output one-hot vector. The neural network defines a maximum sequence length, m, which is the upper limit for the number of one-hot vectors that can correspond to a single output. For our network, m was 50.

2.3.2 Embedding Matrix

The embedding matrix is a learned object that maps the one-hot vectors of size $|v|$ to size d, i.e., the width of the LSTM block (see Section 2.3.3). An embedding matrix may seem superfluous for this type of project since the size of the vocabulary is comparatively small. However, we decided to include it in our design. We knew that the duration of the project was relatively short (about two weeks) and our team was smaller than others (only two members). Hence, we expected we may need to accelerate the training by reducing the number of neurons in the LSTM. In such a scenario, the embedding matrix would have served as a learned dimensionality-reduction technique.

2.3.3 Long Short-Term Memory

Long short-term memory (LSTM) cells are recurrent neural network components, each comprising several carefully arranged neurons and gates. If our recurrent network accepts sequences of length L, then the network must contain a sequence of L LSTM cells. The i^{th} character of the input sequence will feed into the i^{th} LSTM cell, and the i^{th} LSTM cell will also feed into the $i + 1^{\text{th}}$ LSTM cell.

We also added support for a multi-LSTM recurrent layer. This means that each LSTM cell from the singular definition is replaced by an LSTM layer comprised of two or more LSTM cells. Doing this essentially gives the network more resources with which to memorize sequences. The connections between these LSTM layers are full, meaning that every cell of the i^{th} LSTM layer feeds into every cell of the $i + 1^{\text{th}}$ LSTM layer.

Dropout Dropout is a computationally inexpensive means to provide regularization during neural network training. It approximates the bagging technique of an ensemble classifier by training different *versions* of the learner through the temporary deletion of units within the network.[4] TensorFlow supports input, state, and output dropout, which may be used independently or in conjunction with one another. We experimented with different dropout modes and deletion probabilities. We did not

see a significant difference between the settings and settled on output dropout with a keep probability of 0.8.

2.3.4 Feed-Forward Network

In the same way that the embedding matrix maps the network's input to the LSTM's input, the feed-forward network maps the output of the LSTM to the output of network (i.e., the softmax layer). Most of the power of a character-level RNN comes from its LSTM; hence, to prevent extreme overfitting, we opted for a simple feed-forward network with only a single hidden layer of 256 fully-connected neurons. We also observed that using the rectified linear (relu) activation function for both the hidden and output layers achieved superior results. We believe this is because relu allows for greater differentiation between likely characters than the sigmoid activation function.

2.3.5 Softmax Layer

A softmax layer is a function, σ, that takes a real-valued vector, $\mathbf{x}$, of fixed sized, $|v|$, and returns an equal-sized vector, $\mathbf{p}$ whose elements are between 0 and 1 inclusive. Hence, $\sigma : \mathbb{R}^{|v|} \to [0, 1]^{|v|}$. For $k = 1, \ldots, m$, the softmax normalizes each element $x_k \in \mathbf{x}$ using the Softmax formula, which is defined as:

$$p_k = \frac{\exp(x_k)}{\sum_{j=1}^{|v|} \exp(x_j)}. \tag{1}$$

Therefore, the output vector $\mathbf{p}$ is necessarily a probability vector, the sum of whose elements adds to one. The softmax function has multiple advantages including that it creates a standardized output and that it enables the use of well-studied loss functions, such as cross-entropy which is described in the next section.

2.4 Loss Function

We use cross-entropy between the true and predicted distributions as our loss. For a sequence $\mathbf{z}$ preceding character c, we define the true distribution to be $y : y(c) = 1$. In the same example, the predicted distribution is the output of the network given the sequence, or $\mathcal{C}(\mathbf{z})$. For a particular example, if our sequence is $\mathbf{z}$ and the true distribution is $\mathbf{y}$, then the loss on this example is the cross entropy between $\mathbf{y}$ and $\mathcal{C}(\mathbf{z})$, or $H(\mathbf{y}, \mathcal{C}(\mathbf{z}))$. We take the total loss on a set of examples to be the sum of losses per example.

2.5 Batch Size, Learning Rate, and Optimizer

Three important factors that can have a significant impact on the training of a character-level RNN are batch size, learning rate, and optimizer. A smaller batch size often leads to a better learner. As such, we set our batch size to only 50 sequences, which meant that each training epoch required approximately 40,000 batches. The number of batches meant that training a single epoch on a modern high-end CPU took about one hour. With so many batches, a high learning rate would risk later batches wiping out the changes made in early ones. To avoid this, we set the learning rate very low (0.0005).

In homework #5, we observed that TensorFlow's `AdamOptimizer` (which implements adaptive moment estimation [6]) converged the fastest and generally produced the best results. `AdamOptimizer` also outperformed gradient descent in training our character-level RNN, and was used to train our best model.

2.6 Variable Sequence Length Training

Training time and output quality are competing concerns when selecting the sequence length of the learner. A shorter sequence limits the contextual information the learner can use when predicting an output, but longer sequences increase training time. Likewise, any sequences longer than 200-300 characters are not generally correlated with improved output quality.

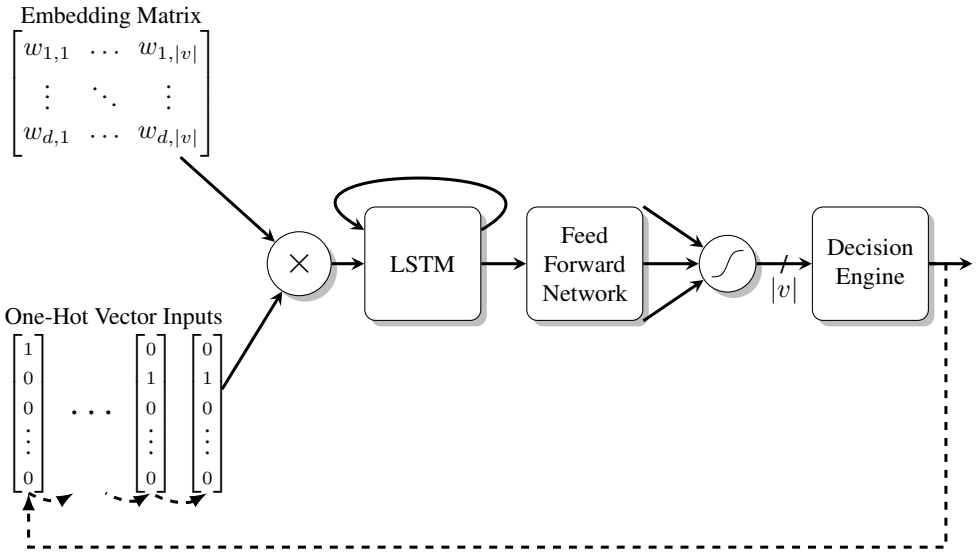

Figure 2: Trump Character-Level RNN Text Generation Architecture

In homework #5, each tweet had a fixed, predefined sequence length. In contrast, during speech generation, the learner is provided a user-created seed text that may be shorter or longer than the maximum sequence length, m. The case of a longer sequence is trivially handled by only considering the m most recent characters. In contrast, predicting the output from shorter sequences is ineffective, given that the learner is only trained with sequences of the unique, maximum length. To address this, we added *variable sequence length training*, meaning that the network is trained on sequences of varying length between the minimum allowed (e.g., 10 characters) and m. This is expected to improve our network's performance on shorter sequences.

3 Text Generation

To enable text generation, changes must be made to the training architecture described in Section 2. This section discusses these architectural modifications as well as how we seed the generator.

3.1 Seeding

Rather than generating text from random noise, we believe that users will have a much better experience if they can supply the initial *seed*, which the network will use to build its output. To ensure that the architecture has sufficient context upon which to begin generating text, we require that the seed string be at least 10 characters long. Text generation continues until the specified number of characters (e.g., 500) has been generated.

3.2 Text Generation Architecture

The neural network in Figure 1 is referred to as the "base architecture" since it is only a subset of the complete system used during text generation. Figure 2 shows the complete text-generation architecture, and there are two primary changes from the previously mentioned base. First, we add an entirely new block to the network, which we refer to as the "decision engine." Its role is to select a single output character from the softmax probability vector of length $|v|$; while this may seem like a trivial task, we explain in more detail in Section 4 the decision engine's unique challenges.

The second change is that each generated output character is fed back into the network to generate the next character. Similarly, the previous sequence of characters is shifted by one with the least recent character removed if the sequence length is longer than the maximum, m (e.g., 50). This approach allows the architecture to generate arbitrarily long text.

4 Decision Engine

We refer to the character selection algorithm as the *decision engine*. The name is apt because the problem of generation is: given a distribution for the "character probability," how do you decide what the next character is going to be? An obvious answer would be "take the most likely choice." However, what if the distribution you receive is nearly uniform? What if the distribution you receive is anomalous? In this section, we evaluate the advantages and disadvantages of greedy, randomized, and mixed decision engines.

4.1 Greedy Sampling

The greedy decision engine is in some sense the most obvious: just take the most likely character. A clear advantage is that the most confident choice is always made. This necessarily prevents any degree of spontaneity. A catastrophic issue with text generated using the greedy decision engine is that it tends to enter infinite loops. For example, when we give our best model the seed *"The media is so dishonest."* and generate text with the greedy decision engine, the output is *"They want to stop the people of the world. I want to stop the people of the world. I want to stop the people of the world. I want to stop the..."* etc.

4.2 Random First, Greedy Finish (RFGF)

To prevent infinite loops, we make the network stochastic. The obvious way to do this would to be to perform a weighted sample of the output of the softmax layer; in other words, the network inherently provides a weighted dice that we can roll to select the output character.

The problem with this scheme is the unconstrained randomness. The benefit of the greedy engine is that it always makes a confident choice, but the random engine provides no such guarantee. Every time we roll the die given the sequence "My name is Donald Trum," there is non-zero chance it will come up "Q", and this is a very serious flaw.

One solution is the random-first greedy-finish (RFGF) engine. The basic idea is: always start a new word with a random choice, but in the middle of a word make greedy choices. In practice, this means that the basic random engine is used if and only if the preceding character was whitespace; otherwise, the standard greedy engine is used. This provides an adequate balance of the previously mentioned concerns. It is random enough to avoid the infinite looping behavior, yet it does not mangle words with absurd characters.

4.3 Top-*k* First, Greedy Finish (TFGF)

Since the softmax function does not assign zero probability to any character, there is still a chance that absurd letter choices will be made including putting an exclamation point after a whitespace or inserting a newline mid-sentence. These simple examples illustrate that when selecting characters at random, the learner must assign zero probability to some choices. We achieve this by having the learner consider only the top-k characters with the highest probability (we used $k = 5$ in our experiments). The resulting sub-distribution is then renormalized so that the probabilities sum to one. Now you have a k-sided die that you can roll. We call this the top-k first, greedy-finish (TFGF) engine.

4.4 Randomization through LSTM Dropout

As explained in Section 2.3.3, dropout is traditionally a technique used exclusively during training. It is not generally used in a "live" environment producing real outputs. However, dropout is very computationally inexpensive and has been highly optimized in TensorFlow making it especially fast. Likewise, using dropout during text generation eliminates the need for random guessing that is associated with both RFGF and TFGF.

We have observed that the best form of randomization may in fact be dropout *during generation*. We are currently in the process of implementing and testing two new decision engines, namely Dropout First, Greedy Finish, which enables dropout only for the first character after a whitespace as well as Dropout And Greedy Always (DAGA) where dropout is permanently enabled and the greedy choice

is always made. Preliminary data indicates that the latter approach has execution time very close to that of the standard greedy sampling described in Section 4.1.

5 Conclusions

As demonstrated in class, we successfully implemented a character-level RNN that produces text in the style of Donald J. Trump. The generated output was so realistic that many in the class could not distinguish it from the real thing. It is important to note that not all text produced by our system is coherent; the classroom example we presented was specifically selected as it was significantly superior to the typical output. It is not uncommon that the generator produces incoherent sentences, which means that possible improvements to our system remain. The next section describes a new approach to character-level RNNs that we believe has the potential to significantly improve output quality.

5.1 Future Work

Our RFGF and TFGF engines provide clear advantages over a straight greedy strategy. However, at their core, RFGF and TFGF are both still greedy. They make point in time decisions about the best character to select using exclusively past information, and once a decision is made, it cannot be undone. Rather than selecting a character based solely on past data, we expect a far better decision could be made if we also consider the effect of the current decision on *future* decisions.

In Section 1.2, we explained that word-level RNNs are impractical. However, we believe that through character-level decisions it may be possible to achieve near word-level results. For example, at the start of a word, rather than immediately picking a character, the system could select the top-k characters and complete the resulting k words using our greedy-finish approach. The resulting k words could be examined and the best one selected. This approach reduces the number of possible outputs that must be considered at any given time from more than 170,000 words to just k. Likewise, rather than making word-level decisions, even better results may be achieved if n-gram decisions were made. For example, the k best words could be constructed at a given point in time. From there, each of their k-best descendants could also be constructed. This process would be repeated until a phrase of length n is built.

One of the primary challenges of this word or n-gram-based approach is quantitatively deciding the "best" selection. Coherent speech is subjective, and given the simplistic and awkward nature of Trump's speech [3], we expect that many objective metrics of text coherence may perform poorly. One approach proposed by Manfred Warmuth was to select the one with the highest probability (normalized by word length). This approach appears plausible but requires further study.

We are still implementing this aspect of our learner. The implementation is not complete so we are unable to report the results in this report. However, we expect for this work to be completed in early 2018.

References

[1] *2016 presidential election speeches and remarks*. `http://www.presidency.ucsb.edu/2016_election_speeches.php?candidate=45&campaign=2016TRUMP&doctype=5000`.

[2] *How many words are there in the English language?* `https://en.oxforddictionaries.com/explore/how-many-words-are-there-in-the-english-language`. Online; accessed 15 December 2017.

[3] O. GOLDHILL, *Rhetoric scholars pinpoint why Trump's inarticulate speaking style is so persuasive*, Quartz.com, (2017).

[4] I. GOODFELLOW, Y. BENGIO, AND A. COURVILLE, *Deep Learning*, MIT Press, 2016. `http://www.deeplearningbook.org`.

[5] A. KARAPATHY, *The unreasonable effectiveness of recurrent neural networks*. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`, May 2015. Online; accessed 15 December 2017.

[6] D. P. KINGMA AND J. BA, *Adam: A method for stochastic optimization*, CoRR, (2014).

[7] J. MAYER, *Donald Trump's ghostwriter tells all*, The New Yorker, (2016).

[8] R. MCDERMOTT, *1MB archive of Donald Trump speeches*. `https://github.com/ryanmcdermott/trump-speeches`, 2017. Online; accessed 15 December 2017.

[9] R. MITKOV, *The Oxford Handbook of Computational Linguistics*, Oxford University Press, 2009.

[10] P. NAVID, *All of Trump's speeches from June 2015 to November 9, 2016*. `https://github.com/PedramNavid/trump_speeches`, 2017. Online; accessed 15 December 2017.

[11] A. OHLHEISER, *The (other) man behind the curtain of Trump's twitter account is revealed . . . again*, The Washington Post, (2017).

[12] K. PHILLIPS AND A. BLAKE, *Trump on Michael Flynn's guilty plea: It's a 'shame' because he had 'nothing to hide'*, The Washington Post, (2017).

[13] D. TRUMP, *Trump: The Best Golf Advice I Ever Received*, Crown Publishers, 2005.

[14] ——, *Time to Get Tough: Making America Great Again!*, Regnery Publishing, 2015.

[15] ——, *Crippled America: How to Make America Great Again*, Threshold Editions, 2016.

[16] D. TRUMP AND K. BOHNER, *Trump: The Art of the Comeback*, Times Books, 1997.

[17] ——, *Trump: The Art of the Comeback*, Times Books, 1997.

[18] D. TRUMP AND R. KIYOSAKI, *Midas Touch: Why Some Entrepreneurs Get Rich, and Why Most Don't*, Plata Publishing, 2012.

[19] D. TRUMP AND R. T. KIYOSAKI, *Why We Want You to be Rich: Two Men, One Message*, Plata Publishing, 2006.

[20] D. TRUMP AND C. LEERHSEN, *Trump: Surviving at the Top*, Random House, 1990.

[21] D. TRUMP AND M. MCIVER, *Trump: How to Get Rich*, Random House, 2004.

[22] ——, *Trump Never Give Up: How I Turned My Biggest Challenges Into Success*, John Wiley & Sons, 2008.

[23] D. TRUMP AND T. SCHWARTZ, *Trump: The Art of the Deal*, Random House, 1987.

[24] D. TRUMP AND D. SHIFLETT, *The America We Deserve*, Renaissance Books, 2000.

[25] D. TRUMP AND B. ZANKER, *Think Big and Kick Ass: Make it Happen in Business and Life*, Morrow Avon, 2007.