Text Generation on the Example of Trump Tweets

Ilay Koeksal, Ann-Katrin Thebille
7koeksal@informatik.uni-hamburg.de, 7thebill@informatik.uni-hamburg.de
Knowledge Processing in Intelligent Systems: Practical Seminar
Knowledge Technology, WTM, Department of Informatics, University of Hamburg

***Abstract*—The goal of this seminar paper was to create Trump-like tweets with an LSTM trained on real tweets. We took the raw tweet data from an archive and manually cleansed the data set. To improve our results, we embedded the raw tweet data with Word2Vec. The actual text generation is handled by an LSTM which predicts the next word vector based on an already existing tweet start. After training our LSTM network for 540 epochs with 24000 tweets, we generated tweets of mediocre quality. While the generated tweets seem to improve regarding correlated content, some words are still repeated or do not fit the grammatical structure of the sentence. The paper concludes with a discussion about some possible further improvements of our approach.**

I. Introduction

Text generation is a popular task among machine learning researchers. It is used in numerous natural language tasks like speech-to-text, conversational systems or text-summarization. It also gained attention through the media, thanks to natural language systems which generate jokes or pop-culture related content. Besides entertainment value, text generation applications are also highly profitable. Especially data-to-text systems and systems that generate textual summaries of databases become more and more common. Also, new fields like automated journalism, automated paraphrasing, and generative literature recently gained popularity thanks to the advances in automated text generation.

Our hypothesis is that we can recreate tweets from Donald Trump with a recurrent neural network (RNN). Even though Donald Trump is a controversial political figure nowadays, his Twitter account was famous among users before his political career started. His unique style of language and the sheer amount of tweets he publishes make him a good example for recreating a specific language style with a neural network. Trump's tweets frequently make headlines, therefore recognizing his tweet style should be quite easy for most people. This helps with the subjective analysis of the results of our tweet generator.

II. Related Work

A. Tweet mining

Good results require a large and extensive data set. Since most training data sets are not up-to-date, our initial idea was to mine our training data directly from Trump's Twitter page. To collect Twitter data, python libraries like Tweepy [1] can be used to access the Twitter API. Tweepy allows mining tweets from any public user's page in real-time. Therefore, creating a data set with Tweepy is very easy. However, the Twitter API has an in-built limit for the number of tweets to mine to avoid abuse. This limit is set at 3200 tweets (including retweets). Therefore, creating a large dataset of a single user is not possible.

B. Word2Vec

Word2Vec [2] is a method for embedding words into a space of low dimensionality. One-hot-encoded word vectors represent words by using vectors of the vocabulary size n which consist of $n-1$ zeros and 1 one. Thus, they require a lot of disk space and grow with the vocabulary size. Furthermore, which word is represented by which number in the vector is random, therefore no relationship between words can be indicated. To avoid memory problems and to get a better representation of the word relationships, words have to be embedded in a meaningful way. This can be achieved with Word2Vec, as is visualized in Figure 1.

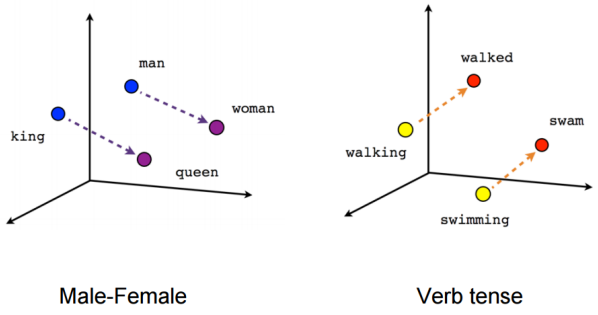

Figure 1. Visualization of word vectors learned by word2vec. Vectors created by word2vec incorporate semantic meaning of words and the relations between them. [3].

Word2Vec basically consists of two different model types: continuous bag-of-words (CBOW) and skip-gram. Since we used CBOW for our application, a short description follows. The CBOW model has the explicit goal of predicting the next one-hot-encoded word vector from one or more context words (depending on the window size). By training the model to achieve the best possible performance, the implicit goal of finding a good embedding is solved. This is caused by the fact that the weight matrix W of the hidden layer adapts to map words, which appear in the same context, closely together to minimize the error [4]. Thanks to the nature of

one-hot-encoded vectors, a word which is represented by a vector with a 1 at index i corresponds to column i of W. By adapting the row number of W, a representation with lower or higher dimensionality can be found.

Word2Vec is, for example, implemented in the python library Gensim [5]. One can specify the size of the resulting word vectors, as well as the minimum count of occurrences of words. Furthermore, the window size of the neighborhood can also be adapted. After training the model, words which have been part of the training data can easily be converted to the respective word vector. The word vectors can then be used as input for the text generator.

C. LSTM

Recurrent neural networks (RNNs) are a type of artificial neural network where the previous state of the network affects the outcome of the current state. This feature of RNNs makes them functional in applications which process sequential data. RNNs have been successfully used in, for example, speech recognition, language processing, and modeling, as well as translation studies.

Even though the architecture of an RNN allows it to remember information from the past, it is still limited in its context comprehension. In many cases, dependencies between input data fragments can be lost when there is a long gap between them. Those dependencies are referred to as long-term dependencies. To improve long-term dependencies, long short-term memory (LSTM) models have been invented [6]. In Figure 2, a text generation example with an unfolded LSTM model is shown. The LSTM uses the previous hidden state and the current input to predict the next output. This way, all previous outputs have an impact on the prediction.

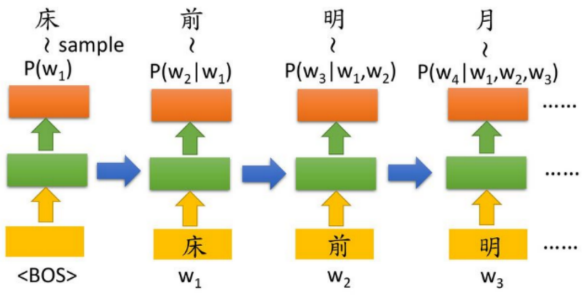

Figure 2. When given a word, an LSTM calculates the probability of the next word given the previous words. The expected output for a text generation LSTM is the input but shifted to the left by one word [7].

A simple LSTM unit consists of one memory cell and three gates. These three gates are called the input gate, forget gate and output gate. The gates control the flow of the input inside the unit. The input gate controls the input sequence and which parts of the input will be accepted by the unit. The forget gate controls which values inside the unit will remain in the unit. And lastly, the output gate decides which value will be used to calculate the activation of the unit. The connections of the gates and weights in these connections are learned during training and regulate the work of the gates. That is why long-term dependencies, as well as short-term dependencies, can be recognized by the LSTM and used for predicting the next word.

III. Our approach

A. Training data

Since 3200 tweets are not enough training data, we decided to create our own data set using the Trump Twitter Archive [8]. The Trump Twitter Archive collects and stores each tweet published by Donald Trump in real-time and enables downloading a specific subset of tweets. We created a dataset of more than 24000 tweets (excluding retweets).

We decided to use a word-level based approach since we wanted to avoid generating words, which do not exist in Trump's vocabulary [9]. Character-level based models, however, would have the benefit of requiring less training data since one tweet encompasses a lot more examples on the character-level than on the word-level. However, to bypass the problem of too little data, we prepared the training data to train line-by-line. To do so, each sentence of n words is split into n-1 examples. The first example x consists of n-2 padding words (in our case we chose *padchar*) and the first word of the tweet. The target of x is the second word of the tweet. The second example exists of n-3 padding words and the first two words of the tweet. The target is the third word of the tweet and so on. For the last example of a tweet, the target is the word *endchar*, which we added to indicate the end of the tweet. This is also used for the generation later on since no more words are generated after predicting *endchar*.

In total, we generated a training set of 388668 examples and a validation set of 43186 examples, which corresponds to a 90% - 10% split.

B. Word embedding

Using Word2Vec requires cleaning of the data set as well as tokenization. However, the already existing pre-processing functions of Gensim did not fulfill our requirements since they also remove hashtags or user tags. Besides, stop-words would be removed which would be problematic later on for the generation of new tweets. That is why we manually tokenized the tweets by replacing specific characters in the texts as well as removing hyperlinks and splitting the cleansed text at every space. We decided against using a pre-trained wordnet even though it would include more words since we only need a specific subset of all possible words for impersonating Trump's tweet-style. This way, the embedding is already adjusted to the application [10].

As a result, we generated a Word2Vec model which contains 25537 words. Every word is mapped to a vector of the length 100. We decided on a minimum word count of 1 to avoid mapping all the words unknown to the word2vec model

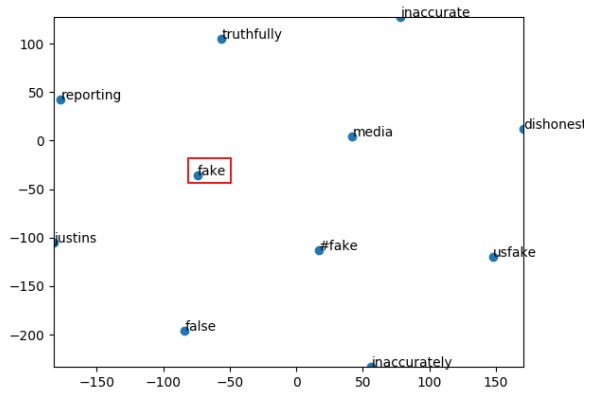

Figure 3. Most similar words to *fake* in our model. The graph visualizes the distance of the word vectors to *fake* in 2 dimensions using t-SNE [11]

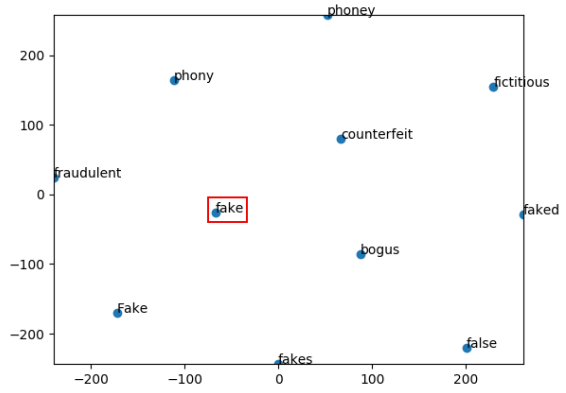

Figure 4. Most similar words to *fake* in the Google Word2Vec model. The graph visualizes the distance of the word vectors to *fake* in 2 dimensions using t-SNE [11]

to a zero-vector. To get a more context-based Word2Vec model, we set the window size to 10.

In Figure 3 and 4, the word vectors which are closest to one example word are shown. These examples show how influential the training data of word2vec is on the resulting word vectors. For example, the word2vec model trained on the Google news data set does not map *media* close to *fake* like our model trained on Trump's tweets.

C. Text generator

To decrease the size of our LSTM, we used the embedded words as input and output. The training data is handed to the LSTM in an already encoded fashion. We used an LSTM with two layers, the adam optimizer [12] and the mean squared error for the loss. Since we had a very big dataset and limited time, we decided on a large batch size (1000) to improve training time. The detailed structure of the LSTM is shown in Figure 5.

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 60, 100)	80400
lstm_1 (LSTM)	(None, 100)	80400
dense (Dense)	(None, 100)	10100

Total params: 170,900
Trainable params: 170,900
Non-trainable params: 0

Figure 5. Network structure of our LSTM

The output of the network is a vector of the size 100, which is then mapped to the closest word in the vocabulary of the word2vec model. To incorporate some variance into the generated text, selecting one of the most similar words at random can be used. This way different text is generated even if the tweet starts the same way.

Since the amount of training data is too large to fit it into one array, we needed to use a training data generator. The generator provides one batch of training/validation data at a time to the model during training. This way, no memory problems occur. Besides, it also handles generating the word vectors. If a word does not occur in the vocabulary, a vector of zeros is used (e.g. for *endchar*).

We trained our network for 540 epochs. One epoch took approximately 4 minutes if trained on a GPU or 15 minutes if trained on a CPU (36 to 135 hours).

IV. Results/Analysis

After training our network, the validation error dropped significantly as can be seen in Figure 6. However, the validation loss did not start increasing again, which could indicate that further training without overfitting is possible. Due to our time constraints, no further training has been performed yet.

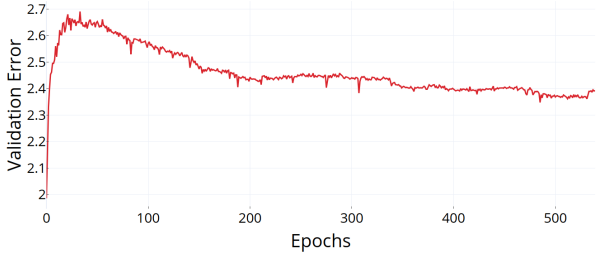

Figure 6. Validation loss of the network over 540 epochs

Some of the tweets which have been generated by the network during training can be found in Table I. The table shows the tweet and the respective epoch in which it was generated. To improve readability, repetitions of words have been deleted. Some sentences are shorter than others since the network predicted *endchar*, which ends the tweet. The tweets have been selected to demonstrate the improvement of our network and to show some variance.

Table I
GENERATED TWEETS (WITHOUT REPETITIONS)

Generated tweet	Epoch
Join me in weeks month level blue $200m	0
I will be interviewed on @taylorswift13 put... creative bliss	27
I will be some with hitting total player loyal $$ for lots	54
I am orange protect money100% dealers precious	77
Join me working pennsylvania see lure	139
Will be proud always that world learn why cannot funding rebuilding hike	276
Congratulations to the entire florida lou evening you @troy_balderson vegaswill diminish taxes spending	494
Congratulations to the entire florida rosendale evening reince businesswoman lines approved republicans	530
I am start paying some positive things positive deal year long year evidenced papers canada blunder canada mets decision nuke canada nuke president nuke today	537

To generate tweets, different seeds have to be used. Seeds are a combination of words which start the tweet so that the LSTM can generate text based on the previous input. During training, we generated tweets by using the most common start words (combination of two words) of Trump's tweets as seeds. Those seeds were the following: *Thank you*, *I will*, *Congratulations to*, *I am*, *Will be* and *Join me*. The tweets in Table I were generated by handing the whole sentence back to the LSTM until 30 words were generated or till the network predicted *endchar*. During training, we only produced sentences by selecting the closest words without any random sampling.

V. DISCUSSION

Especially in the early epochs, our model often got stuck in loops before finishing the tweets. For example, after the first epoch, one of the tweets looped the word *pm* 22 times. This has improved greatly, but occasionally some words are still repeated twice. Since we did not perform any stemming, the grammar of our tweets is sometimes questionable. However, the words contained in one tweet became more coherent and related over time.

One of the challenges we faced was the amount and the quality of the data. Preparing and cleaning data properly is very important in probabilistic language generation models such as ours, therefore, using raw data like tweets results in a challenging task. In our case, to get the style of the person, we kept the typos and grammatical mistakes in our data, which also made generating meaningful text harder. For example, our pool of words contained *Hillary* and *Hilary*, which both mean the same person in Trump's tweets but are seen as two different words by our network. To solve this problem, we would need to incorporate a grammar-checker into the pipeline before training the network. However, this would stray our results further from the source. Besides, our tweets currently do not contain any punctuation, which could further alter the perceived meaning of some sentences.

Another solution could be using a different training set. For example, Brad Hayes used the transcripts of speeches of Donald Trump to train a deep neural network for generating tweets [13]. An example tweet of his network can be seen in Figure 7.

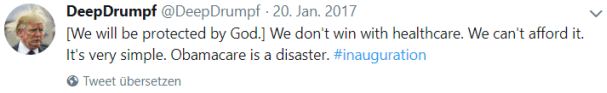

Figure 7. Tweet generated by an AI based on speech transcripts. The words in the brackets are the seed of the generation.

VI. CONCLUSION

As we observed, training our network requires a powerful GPU, as well as a large amount of memory. Furthermore, the network size and parameters are crucial for achieving good results. Since the training process takes a significant amount of time, it was not possible for us to try out a multitude of network parameters and/or network structures. Even though we got relatively good results, we believe that this model can be improved by experimenting with different parameters and adding more training time.

Despite the fact that text generation is a common task among researchers, it is still far from perfect. We believe that, with the improvements we mentioned previously and with new developments in the field, tweet generation can be achieved in a more coherent and natural way.

REFERENCES

[1] J. Roesslein, "Tweepy," 2009.
[2] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.
[3] Tensorflow, "Vector representations of words," 2018.
[4] X. Rong, "word2vec parameter learning explained," *CoRR*, vol. abs/1411.2738, 2014.
[5] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, (Valletta, Malta), pp. 45–50, ELRA, May 2010. http://is.muni.cz/publication/884893/en.
[6] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[7] H.-Y. Lee, "Conditional generation by rnn and attention.," 2018.
[8] B. Brown, "Trump twitter archive," 2017.
[9] E. A., "Word-level lstm text generator. creating automatic song lyrics with neural networks," 2018.
[10] P. Megret, "Gensim word2vec tutorial."
[11] G. Hinton and Y. Bengio, "Visualizing data using t-sne," in *Cost-sensitive Machine Learning for Information Retrieval 33*.
[12] D. Kingma and J. Ba 12 2014.
[13] B. Hayes, "Deep drumpf," 2016.