Twitter Bot

Tim Marco Mennicken
Master student computer engineering
University of Applied Sciences Cologne
tim_marco.mennicken@smail.th-koeln.de

Sina Behdadkia
Master student computer engineering
University of Applied Sciences Cologne
sina.behdadkia@smail.th-koeln.de

Robert Rose
Master student computer engineering
University of Applied Sciences Cologne
robert.rose@th-koeln.de

***Abstract*—The objective of this research project is to design and develop a neural network that is able to create and publish short text messages, further named tweets, on the social networking service Twitter. The content of these tweets will be based on the tweets of a public personality. Since there is a steady increase of machines and algorithms in our everyday life, researches are looking for ways to facilitate human machine interaction. The most natural way for humans to communicate with other beings is spoken and written language. Therefore, on the one side, it is essential to develop algorithms, which are able to extract information from human language. On the other side it is not less important that these algorithms can form sentences, which are understandable for human beings. In this project we propose a LSTM long short-term memory architecture, that is capable of extracting features with long term dependencies. The prediction will be based on a starting seed with a word-level approach, so that complete words will be predicted. The data will be scraped from Twitter profiles and can be of arbitrary language.**

***Index Terms*—LSTM, neural network, Twitter**

I. Introduction

The possibility of generating text based on deep learning algorithms exists since years. Also social media platforms established themselves in the last years as the real-time communication standard. Platforms like Twitter enable people to communicate at any time with people all over the world over a broad choice of topics. The effect of this is the democratization of political discussion. But with the possibility, the probability of misuse is also given. A study from March 2017 already proposed, that 9-17% of all Twitter users are bots.

Regarding the increase of bots used in social networks like Twitter, Facebook or Instagram, the risk of political manipulation is widely spread. Cases like Cambridge Analytica show, it is very likely to be victim of such a manipulation.

Social bots are accounts which take actions controlled by a software either to establish interaction or generate content algorithmically. They don't need to be necessarily political, there are also a lot of informative bots, which report on a specific genre of news. Unfortunately the number of bots, who contribute to the political disinformation, is growing rapidly. This project is dealing with the complexity to set up such a Twitter bot in an entertaining fashion. Already existing Twitter accounts shall be imitated with the goal to be as good as the real person.

The difficulty regarding this project consists of the semantic understanding for the person to imitate. Therefore the program needs to extract features of the language. But not only it needs to understand the general grammar or spelling, but also the specific language features of the specific person.

This could also include errors made by the person, which will be adapted by the algorithm. We use techniques of the Deep Learning area to make semantic analysis of the gathered tweets and investigate the human readability of the tweets with aspects like plausibility.

Tweets are sequential data of a determined length. For this reason, it is appropriate to use a RNN network structure as the main architecture. Further LSTMs proved themselves to be a well thought choice to include long-term memory capabilities in the learning process.

Another important part is the preprocessing of the gathered data. Users on Twitter can not only post tweets, there is also the possibility to repost other users. These tweets need to be filtered. Also emojis or hashtags need to be considered.

II. Related Papers

A related project found is a deep neural network used for bot detection. It also uses a LSTM approach but also additional meta data of the targeted user. It uses semantic analysis of the tweets to extract the contextual features. But it doesn't generate text, the semantic analysis only assists the classification.

III. Methods

A. Architecture

For the processing of sequential data, RNNs are a widely used approach. They are suited for a range of tasks in the field of Natural Language Processing. In comparison to traditional feedforward neural networks, they hold a hidden state which also holds information from previous inputs. This ability is added through looping the processed input to the next cell. This hidden state acts like the neural networks memory. This memory is passed onto the next cell, where it will be concatenated with the input and goes through a tanh activation. This activation implements non-linearity into the system and reduces the values to a range from -1 to 1. This cells can be stacked up onto each other, which can be unrolled for better representation.

Regarding our use case, we consider text as sequential data, which can be split up to feed it into the RNN. Before feeding them into the network, the data needs to be tokenized. This

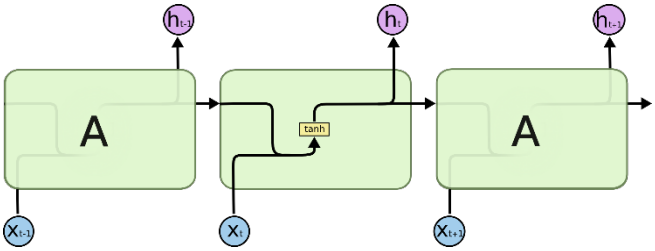

Fig. 1. Structure of an RNN cell.

means to transform the characters or words of the text into a form the network can understand. For this purpose we use a Tokenizer, which turns the text into a sequence of integers. The integers are part of a token dictionary with the different elements of the text. In our project we use the Tensorflow Tokenizer class.

The disadvantage of vanilla RNNs is the short-term memory, which is caused by the vanishing gradient problem. This problem is well known and does not only concern RNNs. It is caused by the nature of backpropagation. The gradient determines the affect of the corrections. But the gradient of each timestep is also dependent on the timestep before. So if the gradient in an earlier timestep was already small, it gets exponentially smaller with further backpropagation. So the corrections made by the network will shrink and lead to no impact in the learning process.

For this reason we used a modification of the RNN, the Long Short-Term Memory or LSTM in short.

The LSTM modifies the inner structure of a cell by adding different gates to the calculation of the output and inner state.

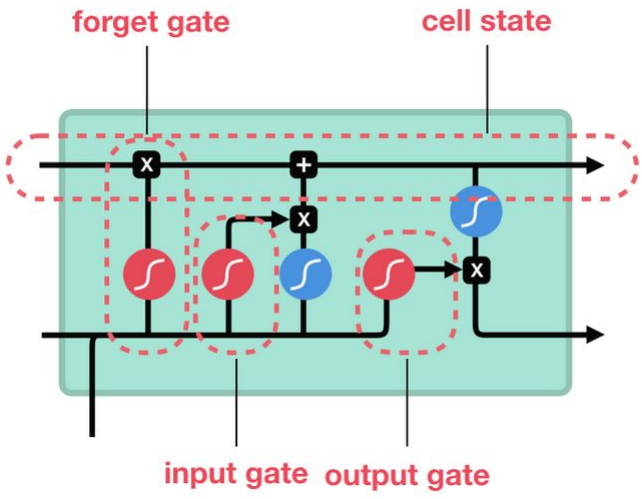

Fig. 2. Structure of an LSTM cell.

In figure 2 the different gates can be seen. The blue symbol marks a tanh activation and the red symbols a sigmoid activation. Converging lines mark a vector concatenation and the “x”/”+” marks a pointwise multiplication/addition. These three gates determine, which information shall be kept or thrown away.

1) Forget Gate: The forget gate determines whether information is relevant enough to be kept or should be thrown away. This is achieved by the sigmoid activation, which maps the values to a range of 0 – 1. Values near to 0 will be forgotten due to the low impact of the parameter.

2) Input Gate: The input gate determines, which values should be updated and kept in memory. Again the effect of the sigmoid activation determines which parameter will be kept and the tanh activation will keep the values in the range of -1 to 1.

With these two gates calculated, the new cell state can be updated, which is the top line of the diagram. Through the pointwise multiplication with the forget gate, the dropping of values will be applied.

3) Output Gate: The output gate is the last station of the data flow inside the LSTM cell. It calculates the hidden state, which will be forwarded to the next cell. Therefore the concatenated input and earlier hidden state will be activated by the sigmoid function. After this it will be pointwise multiplicated with the cell state.

To increase the performance of our architecture we decided to use a Bidirectional LSTM, which not only depends on the previous time steps but considers future timesteps. In our example this means, the network considers both directions of the sentence. This is achieved by splitting the state neurons into two parts. One is responsible for the positive time direction and one for the negative direction.

B. Optimizations

1) Embedding: Embedding is a layer, which represents the words as dense vectors. The weights of the layer will be initialized randomly and updated through backpropagation. After the training it will approximately show similarities of words and can detect connections between the words.

2) Dropout: Because of the high probability to overfit from training data, we decided to add Dropout layers between the LSTM layers. This results in randomly deactivated neurons during the training. It enhances the capability of networks to develop different ways to learn the same concept. During the prediction phase the Dropout will be deactivated.

3) Word Level Approach: For the text generation there are two different approaches to build up a Tokenizer. The first one consists of characters. This results in a small Tokenizer, because there are limited amounts of characters. But it adds complexity to the learning process, because the network needs to learn to build words and in the next step bring them in a meaningful order.

The other approach creates a Tokenizer on word level. With growing datasets, the Tokenizer will be getting bigger and bigger. This increases the memory load and to a great amount of neurons in the fully connected layer at the end.

C. Final Network

Our proposed network uses the standard Tokenizer from Tensorflow to embed the words into indexes. The model starts

with an Embedding layer to detect similarities between words. Then two layer combination of Bidirectional LSTMs follow with a Dropout layer in between each of them. After this follows a Dense Layer and will be finished by the Fully Connected layer, which gives out the prediction for each possible word as a softmax output.

IV. Data Acquisition

In order to train the neural network it is necessary to fetch data sets of existing Tweets[1]. Therefore two different tools are tested and compared with regard to the resulting data sets. The tested tools are the Twitter developer platform [1] (IV-A) and GetOldTweets [2] (IV-B). Especially the amount of fetched text messages is a crucial factor, as the neural network is expected to adapt better on bigger data sets. A requirement of obtaining at least 5000 Tweets for a valid data set is made.

A. Twitter developer platform

Twitter provides an official developer platform which provides API products, tools and resources which enable to automate the whole functionality of Twitter. The APIs are separated in several classes, some require a special access requirement which is not purchased in this work. Subsequent a standard API endpoint is used. In order to use the API products, users need to register for a Twitter developer account. This requires an exact description of the use case of the respecting API. After obtaining a valid developer account, it was necessary to generate a personalized API key as well as user and application access tokens. Twitter offer various wrapper libraries in different programming languages for simplified usage of the API products. In this project Tweepy (IV-A1) is used. [3]

1) Tweepy: Tweepy is a Python wrapper for the Twitter API. It provides access to the entire Twitter RESTful API methods. After registration with the acquired API key and access token, Tweets can be fetched in form of Tweepy model class instances. Objects and method calls are named the same as the regarding instances of the Twitter API, i.e. one can use the terms of the official API reference.

2) Limitations: Each standard API endpoint underlies specific rate limits. That is, only a defined amount of requests are valid in a defined time window. Requests are separated in POST and GET endpoints. For data acquisition especially the GET (read) endpoints are of interest. The Twitter developer reference results in 900 possible status[2] requests per 15 minute window.
Furthermore the standard API endpoint has a limitation of only getting recent Tweets. Even when forcing the API to make requests in a loop, the oldest possible Tweets were published about a month ago with respect to the date of the request. Trying to get older Tweets results in blacklisting the API account, which prevents from passing the authentication stage get a response from the Twitter API.

3) Findings: The overall experience with the Twitter developer platform was not satisfying. Although it was possible to avoid the GET limitations with periodic calls in conjunction with pause times in between, it was not possible to evade the limitation of only getting recent Tweets. As purchasing a premium API is not an option in this project, a more convenient tool for getting a database is needed.

B. GetOldTweets

GetOldTweets [2] is a Python package to get old Tweets. It bypasses some limitations of the official Twitter API. Therefore it uses the how Twitter search through browsers work. The Twitter page scroll loader, which reveals a higher amount of Tweets depending on how far the user scrolls down, is used through calls to a JSON provider. Moreover there is no need to register with keys or access tokens.

1) Limitations: The GetOldTweets package does not offer the possibility of publishing Tweets or getting a number of how often the regarding Tweet has been retweeted[3]. Basically it offers read only access without a lot of meta information.

2) Findings: As in this stage of the project it is most important to gather large data sets, GetOldTweets is the preferred tool. The handling is simpler than that of the official Twitter API and it allows to fetch more Tweets in shorter time. Besides it is able to extract Tweets independent of the datum it was published.

V. Data Preparation

In order to archive good results with the trained models, the fetched Tweets will be prepared before starting the training. This includes filtering unwanted content and manipulating punctuation of the Tweets.

A. Particular Content

Tweets can contain particular contents as weblinks, picture or video links, punctuation symbols, arbitrary special characters, retweets, hashtags[4] and references to Twitter usernames[5]. In order to make the newly-created Tweets as authentic as possible, some of these contents need to be filtered out before passing the data set to the neural network. Therefore the following cosiderations are made.

- Weblinks are connected to the content of the regarding Tweet. The generated Tweets will, in best case scenarios, newly-create a statement. An improper weblink would impair the authenticity of the generated Tweet.
- Picture and video links have the same connection to the content of the regarding Tweet as weblinks.
- Punctuation symbols are part of valid sentences. The neural network will not be taught the grammatical rules

[1]Tweets are short text messages limited to a specific count of characters. Initially only 140 characters were allowed, since 2017 the length of a Tweet was increased to 280 characters.

[2]Status in this context is equivalent to Tweet. Some naming of Twitter terms evolved over time without altering the API definition.

[3]A Retweet is a re-posting of a Tweet.

[4]A hashtag, written with a # symbol, is used to index keywords or topics on Twitter.

[5]A username appears in the respecting profile URL and is unique to the user.

of punctuation symbols, rather it will learn to use punctuation based on frequency of appearance. Hence only the most common punctuation symbols will be adapted.
- Special characters e.g. emoticons can be widely interpreted. It is hard to make an algorithm understand the regarding statement and when to use them.
- Retweets bear reference to other Tweets. As the trained model will have no or little connection to the Twitter universe, it has no point to include retweets.
- Hashtags are widely used in Tweets and are likely to be set multiple times in various Tweets of one user.
- Username references comply with the same considerations as hashtags.

With respect to the considerations only selected punctuation symbols[6], hashtags and references to usernames are fed into the neural network. The remaining particular contents will be filtered out before saving the data to the csv file. Therefore regular expressions are used to search all gathered Tweets for outstanding patterns and remove unwanted content.

B. Space characters

Before training, the neural network will separate the input data set at each space character to generate a list of tokens. Punctuation symbols that are directly behind words, will distort this process. For example would "house" and "house." be interpreted as two different words by the network. To prevent punctuation symbols from causing errors, a space needs to be added before each punctuation symbol. This will cause the network to notice each punctuation symbol as a unique word. Besides multiple consequent space characters, which possibly exist because of previous string manipulation, are collapsed to one space character.

C. Termination symbol

The trained model does typically not know when to stop generating new words. A simple approach to stop generation is to abort it after a specific amount of generated words. This has the poor effect of interrupting at an arbitrary point. The resulting output conveys a choppy feeling as it stops in the middle of generation.
A more advanced technique is to place a unique termination symbol on the end of each Tweet which will be interpreted as an independent word by the network. The trained models will generate the termination symbol on positions where stopping the model will look less artificial, e.g. the ending of a sentence. When a termination symbol is noticed in the text generation loop, the model will be stopped.

VI. Software Structure

The software for fetching data and training the model is written purely in python. The code is mostly inspired by a tutorial on machine learning [5]. It ranges over four different scripts, which all contain object oriented code. This facilitates tuning and experimenting with different models by interchangeable parameters of each method. Each script is described by an explanatory text and a simplified flowchart diagram. The flowchart diagrams only contain the most important points and do not completely display the whole content of the respecting script. Please note, that the flowcharts only show an advisable workflow and combine multiple method calls. The decision symbols are used to illustrate optional methods. The "Loader", "Trainer" and "Generator" classes are designed to be called coherent.

A. Script tweet_dumper.py

This script can be run from the command-line and is responsible for fetching Tweets from Twitter. It imports the GetOldTweets package directly and accepts obligatory, as well as optional, arguments from the command-line to keep the usage as generic as possible. After checking the passed arguments, The script implements the class "TweetDumper" and will pass all command-line arguments to its constructor. Consequently the class starts to fetch Tweets from the passed person until a defined limit is reached. Afterwards it preprocesses the text of each Tweet by removing unwanted content as described in V-A. As a final step the fetched data gets saved in a csv[7] file. Please see figure **??** for a graphical representation.

B. Script loader.py

The "Loader" class is defined in this script. It can not be run directly and needs to be imported in an executable script. The Loader class implements static methods for loading a saved csv file, preparing the data for being passed to the neural network and saving the prepared data. Besides multiple space characters are collapsed and punctuation symbols are padded as described in V-B. Not all of the methods are obligatory, although it is advisable to call them in a specific order as seen in figure **??**.

C. Script trainer.py

The "Trainer" class cannot be run directly and is consecutive to the Loader class. It offers only static methods and can either open a saved file, call methods from the Loader class or be passed the cleaned and sequenced text directly. It creates the tokenizer which the neural network will use as well. Moreover it offers methods to separate the data into input and output as well as to build, compile and fit the model. Hence it forms the heart of this project. The tokenizer and the trained model can be saved afterwards in order to use them multiple times. It is strongly recommended to save model and tokenizer, as they have to be used together and the training is very time intensive. Please refer to figure **??** for further information.

D. Script generator.py

As a final step the "Generator" class should be invoked. It is defined in this script and cannot be run directly. Instead it implements static methods which load a saved model or a

[6]point, comma, exclamation mark, interrogation mark, parenthesis colon and hash

[7]CSV means comma-separated values. It is a delimited text file that uses a comma to separate values.

saved tokenizer, generate an output text based on a seed text and post-process the generated text to make it more human readable. The advised flowchart is displayed in figure **??**.

VII. Experiments

In this step we tried to find a general approach, how it should be down as well. After analysing some articles we've choosed RNN against CNN models, because of the nature of our problem, which is about sequences data. The relationship between characters in a single word is also an important issue. Schematically, a RNN layer uses a for loop to iterate over the timesteps of a sequence, while maintaining an internal state that encodes information about the timesteps it has seen so far. RNN models helps to find the most related character / word regarding to existed dataset.

A. Character Level

First of all we ran the character level model. However promising they might sound, character level models do run against intuition. Words have semantic meaning, characters don't and apriori it's not obvious we can expect a model to learn anything about the semantic contents of a piece of text by going over the characters.
However we built a single LSTM model at first to see how it will work. So that at the end of every epoch because of a for loop printed the results, while is predicting next character appropriate with the most recent character.
The results of this method was sometimes not understandable words, or even not a word at all, which bring us to that point to give a try to word level model.

B. Word Level

Word level neural language model can predict next word based on words already seen on the sequences.
Neural network models are a preferred method for developing statistical language models because they can use a distributed representation where different words with similar meanings have similar representation and because they can use a large context of recently observed words when making predictions. The language model is statistical and will predict the probability of each word given an input sequence of text. The predicted word will be fed in as input to in turn generate the next word. A key design decision is how long the input sequences should be. They need to be long enough to allow the model to learn the context for the words to predict. This input length will also define the length of seed text used to generate new sequences when we use the model.
There is no correct answer. With enough time and resources, we could explore the ability of the model to learn with differently sized input sequences.
Instead, we will pick a length of 50 words for the length of the input sequences, somewhat arbitrarily.
We could process the data so that the model only ever deals with self-contained sentences and pad or truncate the text to meet this requirement for each input sequence.
Regarding to the model design, we are transforming the raw text into sequences of 50 input words to 1 output word. Consequently we trained three types of RNN models.

C. Consisting Embedding Layer

The model we trained is a neural language model. It has a few unique characteristics:

- It uses a distributed representation for words so that different words with similar meanings will have a similar representation.
- It learns the representation at the same time as learning the model.
- It learns to predict the probability for the next word using the context of the last 100 words.

Specifically, we will use an Embedding Layer to learn the representation of words, and a Long Short-Term Memory (LSTM) recurrent neural network to learn to predict words based on their context.
We've used two LSTM hidden layers with 100 memory cells each. More memory cells and a deeper network may achieve better results.
A dense fully connected layer with 100 neurons connects to the LSTM hidden layers to interpret the features extracted from the sequence. The output layer predicts the next word as a single vector the size of the vocabulary with a probability for each word in the vocabulary. A softmax activation function is used to ensure the outputs have the characteristics of normalized probabilities.
Technically, the model is learning a multi-class classification and categorical cross_entropy is the suitable loss function for this type of problem. The efficient Adam implementation to mini-batch gradient descent is used and accuracy is evaluated of the model.
Finally, the model is fit on the data for 100 training epochs with a modest batch size of 128 to speed things up.

1) Result: ...

D. Without Embedding Layer

We specify the kind of model we want to make (a sequential one), and then add our first layer. We'll do dropout to prevent overfitting, followed by another layer or two. Then we'll add the final layer, a densely connected layer that will output a probability about what the next character in the sequence will be.

1) Result: ...

E. Bidirectional LSTM

In this step we have to create the training data for our LSTM. We create two lists:

- **sequences**: This list contains the sequences of words (i.e. a list of words) used to train the model.
- **next_words**: This list contains the next words for each sequences of the sequences list.

To create the first sequence of words, we take the 30th first words in the **wordlist** list. The word number 31 is the next word of this first sequence, and is added to the **next_words** list.

1) Result: ...

VIII. Post Processing

The generated Tweets will be modified slightly in order to increase authenticity and readability. Therefore space characters before punctuation symbols will be removed, as those were only needed in the training process. Furthermore all words are written lowercase, which is not desirable. The first words of all generated sentences are made uppercase with help of the natural language toolkit [4].

IX. Conclusion

We can probably have better results by increasing the size of the RNN, tuning the model to limit variance, etc.
Applying these modifications could increase the capacity for the neural network to generate good phrases, less fuzzy.
So, yes, LSTM can be use to generate not-so-bad text, without great instruction, bu we cannot say that it is a real text, with a global meaning. Regarding the sense of a paragraph (even with a network trained longer than what we did), we are far behind something readable. After few words, sentences does not mean anything, as a whole.

References

[1] Twitter, "Twitter Developers" [Online]. Available: https://developer.twitter.com/en.html. [Accessed: 21- Jan- 2020].

[2] Jefferson Henrique, "GetOldTweets-python - A project written in Python to get old tweets" [Online]. Available: https://github.com/Jefferson-Henrique/GetOldTweets-python. [Accessed: 21- Jan- 2020].

[3] Aaron Hill, Joshua Rosslein and Harmon, "Tweepy - Twitter for Python" [Online]. Available: https://github.com/tweepy/tweepy. [Accessed: 21- Jan- 2020].

[4] NLTK Project, "Natural Language Toolkit 3.4.5" [Online]. Available: https://www.nltk.org/. [Accessed: 21- Jan- 2020].

[5] Jason Brownlee, "How to Develop a Word-Level Neural Language Model and Use it to Generate Text" [Online]. Available: https://machinelearningmastery.com/how-to-develop-a-word-level-neural-language-model-in-keras/. [Accessed: 22- Jan- 2020].