Twitter Sentiment Analysis and Text Generation with LSTM

Abstract

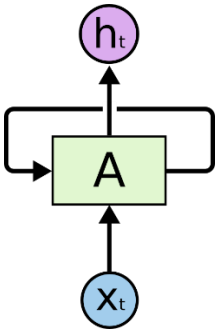
In this paper, twitter is scrapped for Artificial Intelligence and Data science related tweets and a csv file is generated and used as input. The scraping is done using Octoparse tool. The tweets are analysed for their sentiments. The tweets that positive or neutral are filtered and used for training our model. Based on the training, machine will automatically execute its own generated words. We explore the ability of Long short term memory (LSTM) recurrent neural network architecture as compared to normal Recurrent Neural Network. We find out that recurrent neural networks can be used as generative models. We discover how to create a generative model for text, word by word using Long Short Term recurrent neural networks with Keras and Tensorflow as backend.

Introduction

Sentiment Analysis also known as Opinion Mining is a field within Natural Language Processing (NLP) that builds systems that try to identify and extract opinions within text. Usually, besides identifying the opinion, these systems extract attributes of the expression e.g.:

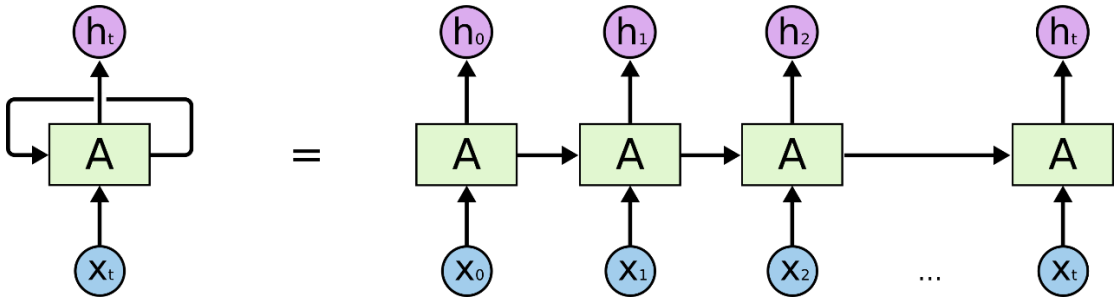
- Polarity: if the speaker express a positive or negative opinion,
- Subject: the thing that is being talked about,
- Opinion holder: the person, or entity that expresses the opinion.

Recurrent Neural Networks (RNNs) are popular models that are greatly used in many Natural Language Processing (NLP) tasks. A recurrent neural network (RNN) is a part of artificial neural network (ANN) where connections between units form a directed graph along a sequence. This allows it to exhibit dynamic temporal behavior for a time sequence. RNNs use their internal state to process the sequence of inputs. This is not possible in feedforward neural networks. They are neural networks which repeatedly make use of sequential information. The main assumption in a traditional artificial neural network is that all inputs and outputs are independent of each other and are not in sequence. But if we look at it, it is not such a great way to approach ANNs. Imagine a scenario where you have to predict the next song a user might want to listen. This will be impossible if the model won't know what songs have already been played. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being dependent on the previous computations.



In the above diagram, a chunk of neural network, AA, looks at some input xtxt and outputs a value htht. A loop allows information to be passed from one step of the network to the next. These loops make recurrent neural networks seem kind

of mysterious. However, if you think a bit more, it turns out that they aren't all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:

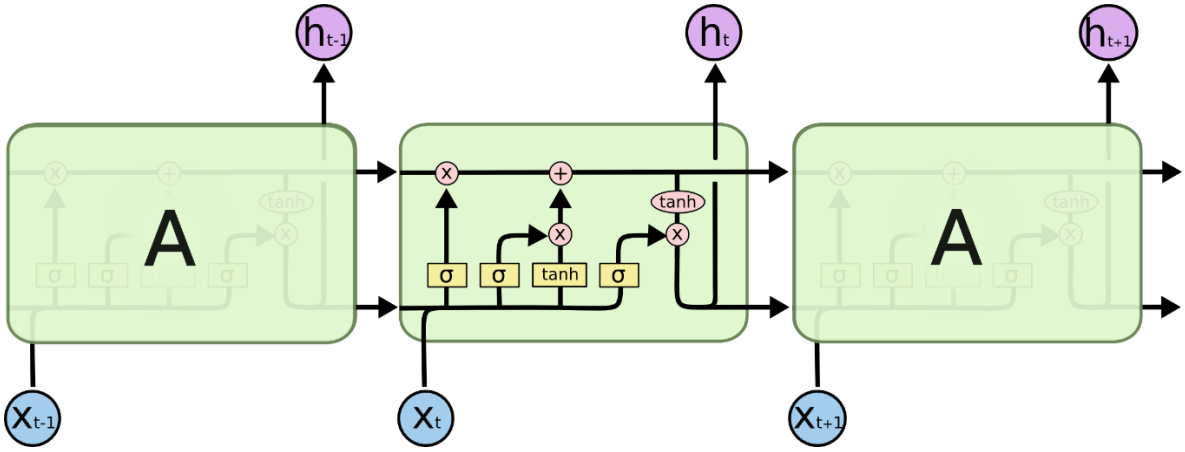


This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to use for such data.

The Problem of Long-Term Dependencies

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task. Sometimes, we only need to look at recent information to perform the present task. In some cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information. But there are also cases where we need more context. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large. Unfortunately, as that gap grows, RNNs become unable to learn to connect the information. Thankfully, LSTMs don't have this problem!

LSTM Neural Networks, which stand for Long Short-Term Memory, are a particular type of recurrent neural networks. LSTM networks have some internal contextual state cells that act as long-term or short-term memory cells. The output of the LSTM network is modulated by the state of these cells. This is a very important property when we need the prediction of the neural network to depend on the historical context of inputs, rather than only on the very last input.



LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn. All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer. LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.

Methods

Firstly, we import all our dependencies. Then we get a csv file of scraped tweets. This csv file is imported and stored in data frame. We then remove all the empty records from the dataset. The dataset is then made consistent for analysis.

```
#DataProcessing
Data = pd.read_csv('AIDatascienceTwitter.csv', usecols =['Tweet'])
Data = Data.dropna()
modData =  pd.DataFrame(Data['Tweet'].apply(clean_tweet))
modData.head()
```

```
# Cleaning the tweets
def clean_tweet(tweet):
    tweet = ' '.join(re.sub("(@[A-Za-z0-9]+)|([^0-9A-Za-z \t])|(\w+:\/\/\S+)", " ", tweet).split())
    tweet = "".join(v for v in tweet if v not in string.punctuation).lower()
    tweet = tweet.encode("utf8").decode("ascii",'ignore')
    return tweet
```

For this we create a function to remove all the unwanted characters using regular expressions “re” library. The text is also converted to lower case so that the model doesn’t treat the same word as it might have started somewhere with a capital letter and somewhere it was in lower case. The encoding is converted t utf-8 to make it consistent.

```
#Getting tweet Sentiment
def get_tweet_sentiment(tweet):
        '''
        Utility function to classify sentiment of passed tweet
        using textblob's sentiment method
        '''
        # create TextBlob object of passed tweet text
        analysis = TextBlob(tweet)
        # set sentiment
        if analysis.sentiment.polarity > 0:
            return 'positive'
        elif analysis.sentiment.polarity == 0:
            return 'neutral'
        else:
            return 'negative'
```

The tweets in the data frame are then analyzed for their sentiment using the “textblob” library. The method polarity returns quantitative value. Now the tweets with negative polarity are considered having negative sentiments and tweets with polarity equating zero or positive are considered neutral or positive respectively. These are then stored in list of dictionaries with tweets and their sentiments as key value pairs. The listed is the further filtered for only positive and neutral sentiments and this is considered as final input for text generative procedure.

The tweets are then tokenized. Tokenization is the act of breaking up a sequence of strings into pieces such as words, keywords, phrases, symbols and other elements called tokens. Tokens can be individual words, phrases or even whole sentences.

```
def get_sequence_of_tokens(corpus):
    ## tokenization
    tokenizer.fit_on_texts(corpus)
    total_words = len(tokenizer.word_index) + 1

    ## convert data to sequence of tokens
    input_sequences = []
    for line in corpus:
        token_list = tokenizer.texts_to_sequences([line])[0]
        for i in range(1, len(token_list)):
            n_gram_sequence = token_list[:i+1]
            input_sequences.append(n_gram_sequence)
    return input_sequences, total_words
```

These tokens are formed into n-gram sequence.

Headline: i stand with the shedevils

Ngrams: | Sequence of Tokens

Ngram	Sequence of Tokens
i stand	[53, 12]
i stand with	[53, 12, 91]
i stand with th	[53, 12, 91, 139]
i stand with the shedevils	[53, 12, 91, 139, 239]

```
[[53, 12],
 [53, 12, 91],
 [53, 12, 91, 139],
 [53, 12, 91, 139, 239],
 [53, 12, 91, 139, 239, 130],
 [53, 12, 91, 139, 239, 130, 240],
 [53, 12, 91, 139, 239, 130, 240, 542],
 [53, 12, 91, 139, 239, 130, 240, 542, 943],
 [53, 12, 91, 139, 239, 130, 240, 542, 943, 543],
 [53, 12, 91, 139, 239, 130, 240, 542, 943, 543, 22]]
```

These n-gram sequences are the padded so that they can converted into a matrix for consistent with a input matrix for the model to be trained. The labels and predictors and labels. In text generation modeling the labels are next possible words of the current. For example-:

Headline: they are learning data science

PREDICTORS	LABEL
they	are
they are	learning
they are learning	they are learning
they are learning data	science

These labels are then converted to a categorical feature as the model predicts the probability of the next word.

```
def generate_padded_sequences(input_sequences):
    max_sequence_len = max([len(x) for x in input_sequences])
    input_sequences = np.array(pad_sequences(input_sequences, maxlen=max_sequence_len, padding='pre'))

    predictors, label = input_sequences[:,:-1],input_sequences[:,-1]
    label = ku.to_categorical(label, num_classes=total_words)
    return predictors, label, max_sequence_len
```

We have modeled the architecture of the LSTM as follows. I have added total three layers in the model.

1. Input Layer: Takes the sequence of words as input
2. LSTM Layer: Computes the output using LSTM units. I have added 100 units in the layer, but this number can be fine-tuned later.

3. Dropout Layer: A regularization layer which randomly turns-off the activations of some neurons in the LSTM layer. It helps in preventing over fitting. (Optional Layer)
4. Output Layer: Computes the probability of the best possible next word as output

We will run this model for total 100 epochs, but it can be experimented further.

```
def create_model(max_sequence_len, total_words):
    input_len = max_sequence_len - 1
    model = Sequential()

    # Add Input Embedding Layer
    model.add(Embedding(total_words, 10, input_length=input_len))

    # Add Hidden Layer 1 - LSTM Layer
    model.add(LSTM(100))
    model.add(Dropout(0.1))

    # Add Output Layer
    model.add(Dense(total_words, activation='softmax'))

    model.compile(loss='categorical_crossentropy', optimizer='adam')

    return model
```

We fit the model and run numerous epochs with default batch size of 32. After extensively training the model we pick a random seed and generate characters in a way mentioned above. We append a single index value to the sequence and remove the extra index if it does not suit well.

Results

After parsing the input file, the result obtained is a machine generated text which is of length provided in the range method. The model.predict() function does the testing of the pattern which the machine learned. We will first tokenize the seed text, pad the sequences and pass into the trained model to get predicted word. The multiple predicted words can be appended together to get predicted sequence.

```
def generate_text(seed_text, next_words, model, max_sequence_len):
    for _ in range(next_words):
        token_list = tokenizer.texts_to_sequences([seed_text])[0]
        token_list = pad_sequences([token_list], maxlen=max_sequence_len-1, padding='pre')
        predicted = model.predict_classes(token_list, verbose=0)

        output_word = ""
        for word,index in tokenizer.word_index.items():
            if index == predicted:
                output_word = word
                break
        seed_text += " "+output_word
    return seed_text.title()
```

Discussion

Even after necessary alterations of the parameters, we will see that generally there are almost no spelling mistakes and the text looks more realistic. The machine has accurately showcased the sentence structure which is human readable and most of the sentences are making. As these are tweets and they are not always grammatically correct. For example, "`Market To Reach 34 16 Bn In`". Irregularities in the word structure. These are better results but there is still a lot of room for improvement.

```
print (generate_text("big data", 10, model, max_sequence_len))
print (generate_text("Neural Nets", 10, model, max_sequence_len))
print (generate_text("AI", 10, model, max_sequence_len))
print (generate_text("World", 10, model, max_sequence_len))
print (generate_text("Machine Learning", 10, model, max_sequence_len))
```

```
Big Data Analytics In Healthcare Market To Reach 34 16 Bn In
Neural Nets Learning To Service April 22 In Seattle New Medium Ai
Ai Tool Could Help Diagnose Alzheimer S Ai Ml Dl Machinelearning
World To Use Model What S Not Big Data And The
Machine Learning Algorithms In One Chart Via Ai Deeplearning Machinelearning Datascience Bigdatapic
```

We can further improve our results by doing the following things-:

1. The selection data can be based on their performance in social media ,i.e. that is cllecting tweets and posts which have high user acceptance.
2. We could increase the size of the dataset and scrap data across various social media platforms .
3. Fine Tuning the network architecture
4. Fine Tuning the network parameters

Licenses¶

References

http://colah.github.io/posts/2015-08-Understanding-LSTMs/
https://www.datacamp.com/community/tutorials/deep-learning-python
https://github.com/tensorflow/models
https://chunml.github.io/ChunML.github.io/project/Creating-Text-Generator-Using-Recurrent-Neural-Network/
https://en.wikipedia.org/wiki/Long_short-term_memory
https://deeplearning4j.org/lstm.html
https://keras.io/layers/core/