Deep Neural Networks 2020

Android-NLP Project

AM DS1190015

Apostolos Papatheodorou

This report is part of the final project of the course of Deep Neural Networks of the master in Data Science and Information Technologies ***(DSIT)*** and describes the development of NLP techniques for multiclass text categorization. More specifically, there are implemented two Recurrent Neural Networks ***(RNN)***, the former with Long short-term memory layer ***(LSTM)*** and the latter with Gated Recurrent Unit layer ***(GRU)***. These models are used by a ***Client-Server*** application in which a python server handles requests and delivers responses to android devices(clients). Code and more details for this project are available on ***my Github***.

Application

Introduction

The application follows the classical client-server structure in which a provider is responsible for delivering services to a customer. Below is analyzed the purpose and functionality of these components.

Client-side

The first part of the architecture is the client and its user interface which is written in android studio. The android interface prompts the user to type natural language text or to just give a URL address as input. After that, the device establishes a socket connection with a python server using TCP/IP protocol, pass the textual data to the server, and communicates with it *synchronously*. At the end of the communication, the client is able to assign the input text to one of four categories (business, medicine/health, science/technology, entertainment). It also displays the most significant sentences of the input text as the exact summary of the data. If there is no reply from the server the connection is terminated and the whole procedure is finished otherwise the whole process can be repeated for many users continuously.

Server-Side

The server host is responsible for manipulating and processing all the requests from the client. For this application, the server is delegated with two different tasks.

a) Assign each input into one of four predefined categories.
b) Detect the most essential sentences in the document.

After the accomplishment of the above targets, the server returns a proper message as a response to the client

a) Text Classification

Before starting to serve requests, the server loads a tokenizer ***tokenizer.pickle,*** which is responsible for the preprocessing of the input data, and sequentially, it loads one of the two pre-trained models, ***lstm_model*** and ***gru_model*** which are charged with the classification tasks. After that, the server opens and handles requests from users. More detail about classification is presented below in the NLP section.

b) Exact Text Summarization

This implementation of exact text summarization is based on the **term frequencies** of the input vocabulary.

This means that for each word corresponds to a weight that is estimated by dividing the frequency of each word with the frequency of the most common word. As a result, for each sentence in the document, it can be calculated a score that expresses

the importance of the period in the whole set. Finally, the server finds the 25% most significant sentences and returns them to the client as a summary. Sentences with many words (|words|>40) are discarded. For the implementation, it is used python's nltk library.

Natural Language Processing

The ML models are trained on *news aggregator dataset* dataset from *uci machine learning repository*. In the dataset there are 422,937 news stories collected by a web aggregator (2014). Each story could be relevant with business, science/tech, entertainment, or health. The target is to classify correctly text passages to one of the above classes.

Preprocessing

The news aggregator dataset contains raw texts that and therefore it is unsuitable for training. We want to transform the data into ***dense vectors*** i.e. (vectors of integers that represent real natural text) so as to be suitable for learning. This can be achieved with the use of keras ***tokenizer***. The procedure for creating dense vectors is the following:

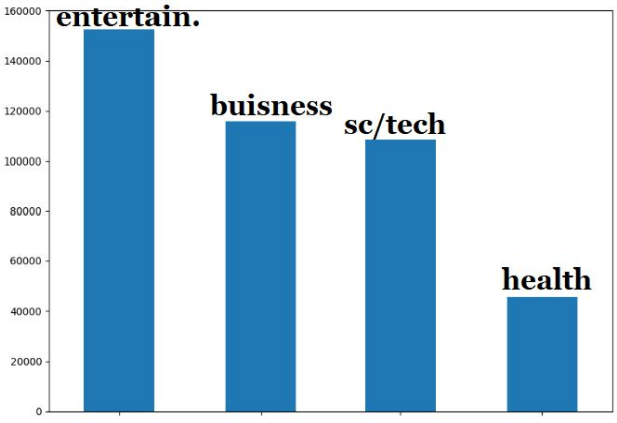
1) Find the number of unique words in the corpus.
2) Create an index in which each word corresponds to an integer and create a new training set based on this index.
3) Pad sequences so as to all the sentences to have the same length.

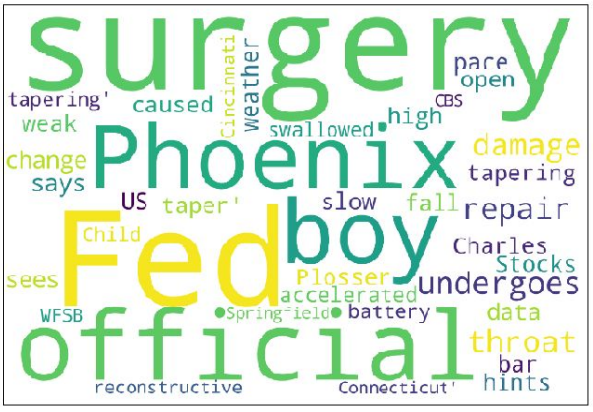
Up to this point the data corpus has been transformed into a new training set proper for training. However this representation still has two drawbacks.

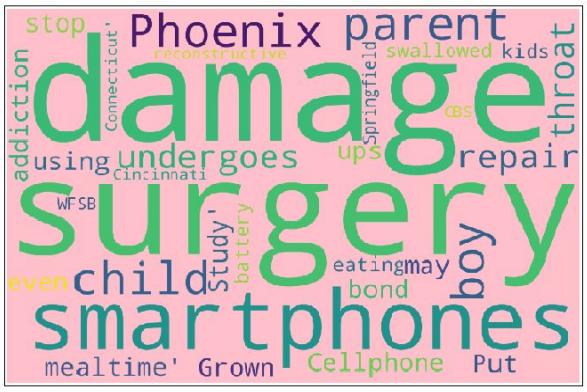
- It does not capture any relationship between words
- There is no relationship between the similarity of any two words and the similarity of their encodings

These problems will be addressed by introducing the ***embedding layer***.

(Left Image: number of texts/category)
(Right Image: WordCloud for all dataset)







(Left Image: WordCloud for health)
(Right Image: WordCloud for sc./tech.)

Training

Embeddings

The first step for the model is to define the embedding layer. To import embedding layer it is used from keras library the ***layers.Embedding***.

```
GRU MODEL
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 150, 128)          1024000
_________________________________________________________________
spatial_dropout1d (SpatialDr (None, 150, 128)          0
_________________________________________________________________
gru (GRU)                    (None, 64)                37248
_________________________________________________________________
dense (Dense)                (None, 4)                 260
=================================================================
Total params: 1,061,508
Trainable params: 1,061,508
Non-trainable params: 0
```

(Image: The summary structure of the model. The model contains an embedding layer, a spatial dropout layer to avoid over fitting, a RNN and a dense layer with 64 neurons. The 4 outputs are for the categs in the training set).

This takes as input a 2D tensor of integers, of shape (samples, sequence_length), and returns a 3D floating point tensor, of shape (samples, sequence-legth, embedding dimensionality).

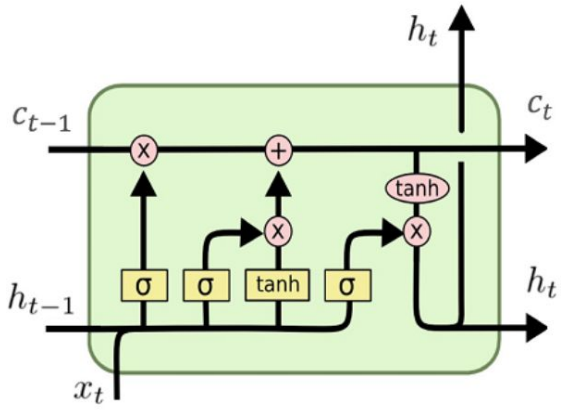
In essence, word embeddings are a dense vector of floating point values in which similar words have a similar encoding. They can be seen as lookup table that maps from integer indices (which stand for specific words) to dense vectors (their embeddings). This encoding does not need to be specified manually, but there are trainable parameters (weights learned by the model during training, in the same way a model learns weights for a dense layer). A higher dimensional embedding can capture more subtle relationships between words, but demands higher execution time.

RNN

Recurrent neural networks (RNN) are networks for processing long data sequences. RNNs differ from the classical ANN as they are algorithms that take into account the previous states of the sequence and not the entire input as whole. Despite the good performance of vanilla RNNs for short term sequences, it has been proven that they are incapable of handling and remembering long-term dependencies. To avoid this issue for this project it is used two different more resilient and powerful special types of RNNs.

LSTM (1997)

In contrast with vanilla RNN, LSTMs are designed to handle the long-term dependency problem by introducing the memory unit. So it contains a cell vector which has the ability to encapsulate the notion of forgetting it's previously stored memory and to add new information as well.



LSTM is composed by three gates Input, forget and output, which optionally let information to flow through the network. All gates make use of a sigmoid function and pointwise multiplication operations. The sigmoid function determines if the information is vital for the network or not, in other words zeros means that the current information has no contribution to the learning process, while one means that information is important.
The three gates are:

Forget Gate Layer: It decides what info it is going to be thrown away from the cell state. It looks at the previous hidden state ht−1 and the current input xt, and outputs a number between 0 and 1. One represents remember while zero represents forget.

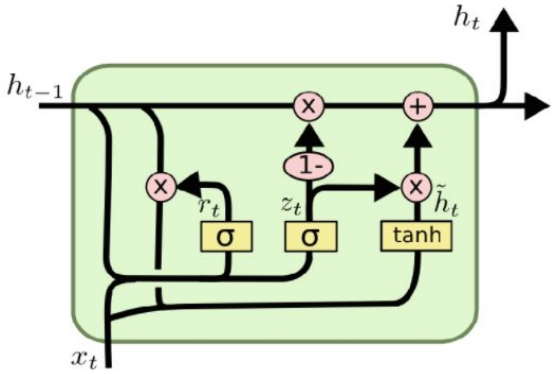
Input Gate Layer: The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, C't, that could be added to the state. In the next step, we'll combine these two to create an update to the state.

Output Gate Layer: This layer decides what the new output will be. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going

to output. Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the useful parts.

GRU (2014)

A different variation of the classical LSTM recurrent neural network is the Gated Recurrent Unit. GRU solves the vanishing gradient problem of a standard RNN, by using ***update gate*** and ***reset gate***. These are the vectors which decide what information should be passed to the output. Similar to the LSTM they can be trained to keep information for long period, without washing it through time. It also remove information which is irrelevant to the prediction. The resulting model is simpler than standard LSTM

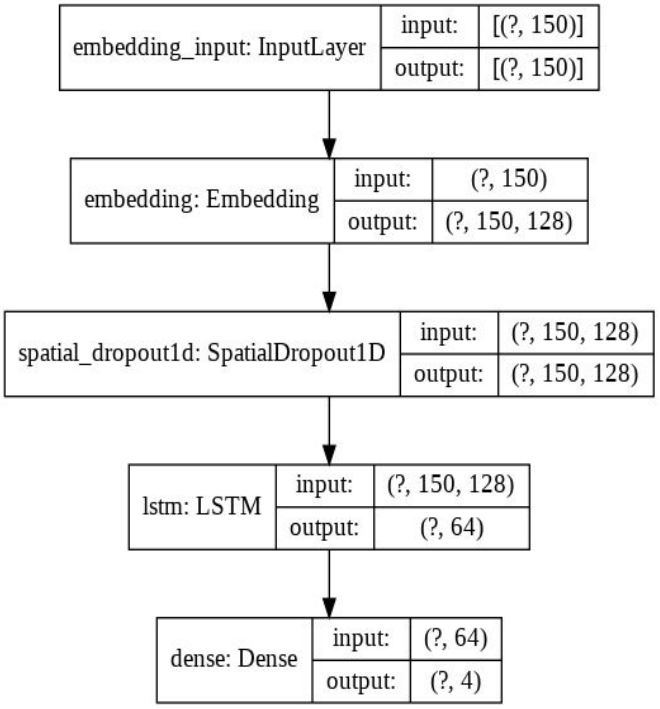


As it can be confirmed in the results below, while GRU demands less execution time, in general, LSTM is considered ***strictly stronger*** than the GRU.

Dense Layer

Because the output of an RNN is not a softmax function and because we aim to classify the input document to one of four categories (multi classification problem) it is added an dense layer at top of the LSTM. This layer is an ordinary ANN with 64 neurons which produces four probabilities each for each class.

On the Image below, shows the model graph as it can be extracted from ***keras.utils.plot_model***.

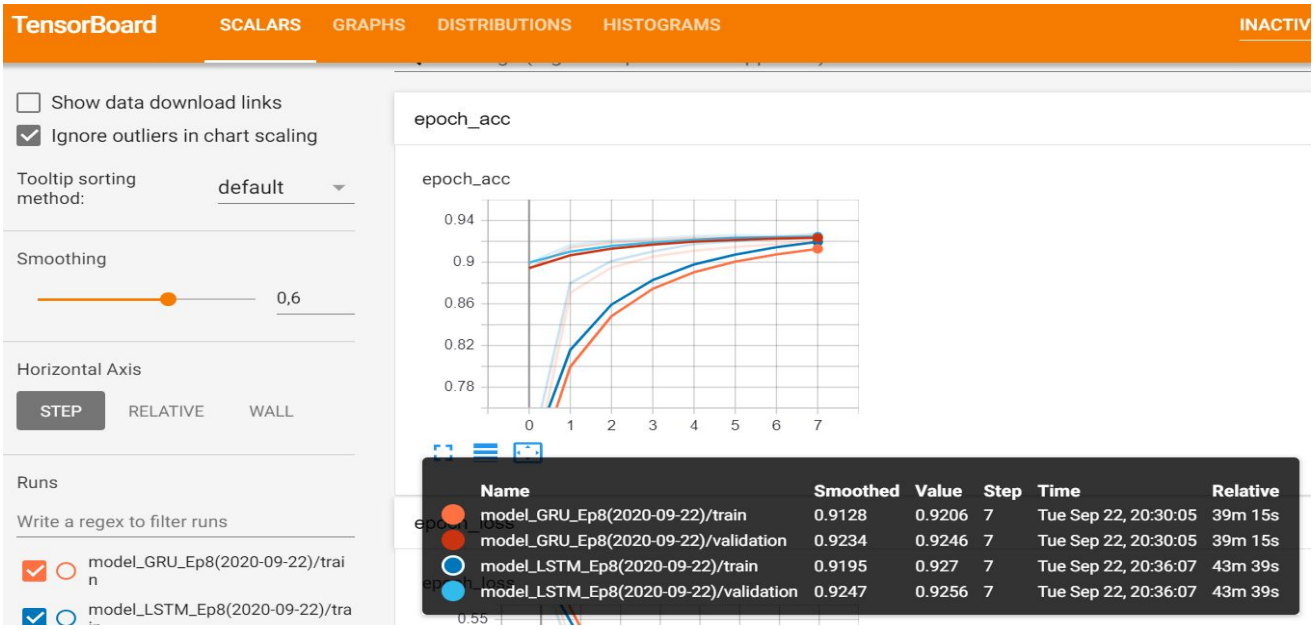


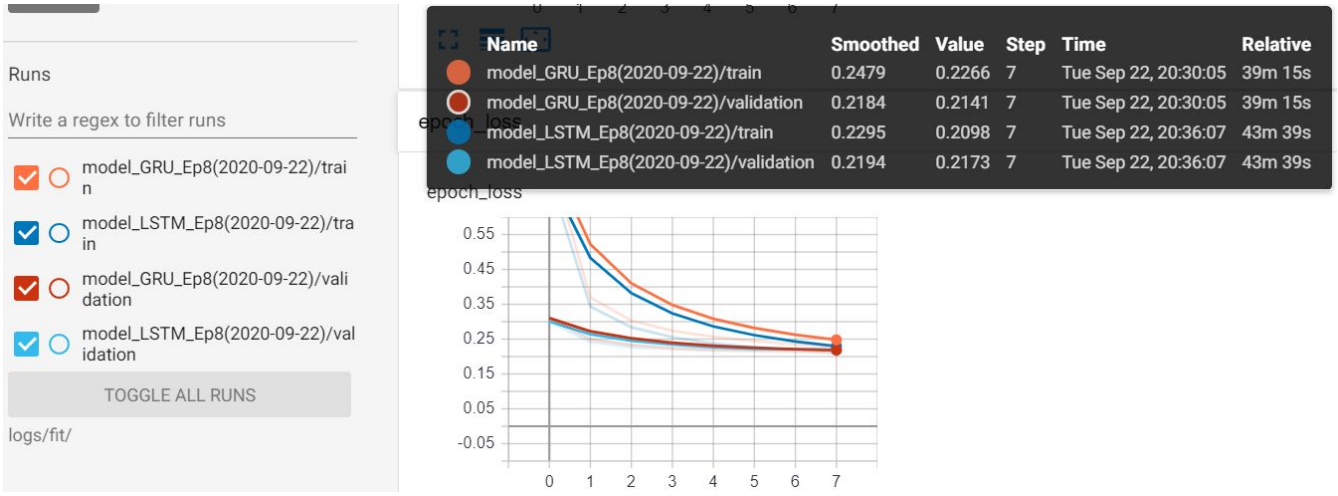
Evaluation & Results

In this sections there are displayed the performance of the models and their behaviour on some real-world inputs.

Metrics

The metrics used for evaluation are: Tensorborad for accuracy and loss, confusion matrices, precision recall, f1 metrics





Confusion Matrix: LSTM

0.95	0.02	0.01	0.02
0.02	0.90	0.06	0.03
0.02	0.06	0.91	0.02
0.01	0.02	0.01	0.94

Labels: ['Entertainment', 'Business', 'Science/tech', 'Health']

Confusion Matrix: GRU

0.95	0.02	0.02	0.02
0.02	0.90	0.06	0.02
0.02	0.06	0.90	0.02
0.02	0.02	0.02	0.94

Labels: ['Entertainment', 'Business', 'Science/tech', 'Health']

Test Loss/Accuracy

```
1529/1529 [==============================] - 36s 24ms/step - loss: 0.2160 - acc: 0.9239
Test set
  GRU_Loss: 0.216
  GRU_Accuracy: 0.924

1529/1529 [==============================] - 42s 28ms/step - loss: 0.2178 - acc: 0.9248
Test set
  LSTM_Loss: 0.218
  LSTM_Accuracy: 0.925
```

LSTM

Precision Recall f1-score

LSTM	precision	recall	f1-score	support
Entertainment	0.95	0.95	0.95	12624
Bussiness	0.90	0.90	0.90	12582
Science/tech	0.91	0.90	0.91	12317
Health	0.94	0.95	0.94	11387
accuracy			0.92	48910
macro avg	0.92	0.93	0.93	48910
weighted avg	0.92	0.92	0.92	48910

GRU

Precision Recall f1-score

GRU	precision	recall	f1-score	support
Entertainment	0.95	0.95	0.95	12624
Bussiness	0.90	0.90	0.90	12582
Science/tech	0.90	0.91	0.91	12317
Health	0.94	0.94	0.94	11387
accuracy			0.92	48910
macro avg	0.92	0.92	0.92	48910
weighted avg	0.92	0.92	0.92	48910

*Testing Model in **Real data** (Wikipedia)*

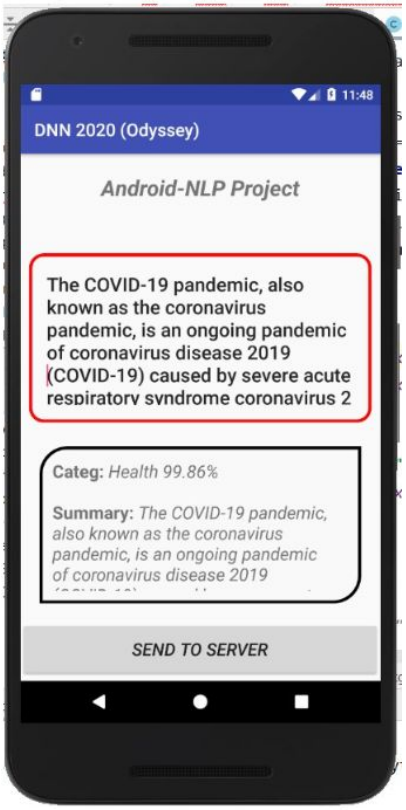
We test the performance of the model on real text passages which are extracted from wikipedia. In particular it was taken text for covid-19, Apple-inc, greek-dept-crisis and the movie 2001:A-Space-Odyssey. It is observed that for all the above examples the algorithm works satisfactory.

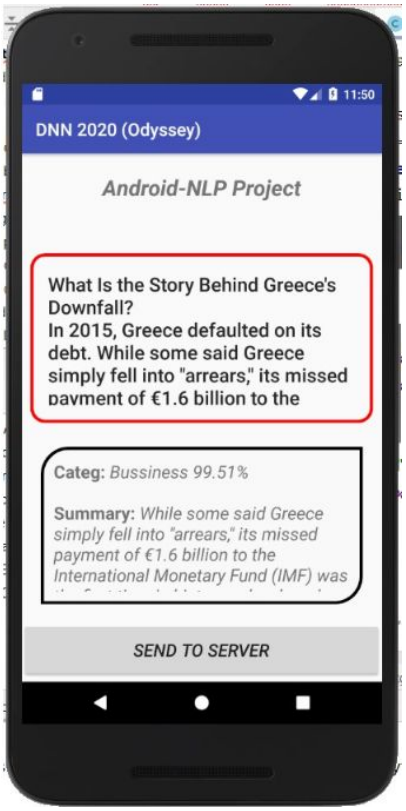
```
Categ: Covid-19 (wiki) => Health 99.48%
        All_predictions: [[0.003 0.002 0.    0.995]]

Categ: Apple inc (wiki) => Science/tech 99.27%
        All_predictions: [[0.    0.007 0.993 0.   ]]

Categ: Greek government-debt crisis (wiki) => Bussiness 99.67%
        All_predictions: [[0.    0.997 0.003 0.   ]]

Categ: 2001: A Space Odyssey (wiki) => Entertainment 99.01%
        All_predictions: [[0.99  0.001 0.008 0.001]]

Labels ['Entertainment', 'Bussiness', 'Science/tech', 'Health']
```

Conclusion

We observe that the model behaves pretty well when the passages belongs explicitly to one of the target classes. However when two or more classes overlapped then the model is confused. For example in the movie: 2001: A Space odyssey, despite the text belongs explicitly to movie (entertainment class) the model can be confused by the intense use of scientific vocabulary text and hence to make improper classification.

```
Categ: 2001: A Space Odyssey (wiki) => Entertainment 99.01%
        All_predictions: [[0.99  0.001 0.008 0.001]]

Categ: 2001: A Space Odyssey (wiki) => Science/tech 74.81%
        All_predictions: [[0.237 0.002 0.748 0.013]]
```





Server

```python
def recv():
    counter=0 # how many request the server has received
    Number_of_Requests=10 # Total numver of clients that will be served.

    try:
        client.bind((host, port))
    finally:
        pass
    client.listen(10) # how many connections can it receive at one time
    print ("Start Listening...\n")

    while counter<Number_of_Requests:
        conn, addr = client.accept()
        print ("client with address: ", addr, " is connected."); print()
        data = conn.recv(1024).decode('utf-8');
        print ("Decoded Data: ", data);print();
        #conn.sendall(bytes('Take your reply Text_Sum', 'utf-8'))

        # Summary of the text
        print('Text Summary')
        Text_Sum, Scores=TxtSummary(data)
        Text_Summary=DataProcess(Text_Sum, Scores); print('\n')

        # Class prediction
        print('Text Classification')
        categ, prob=Text_Categ(data)
        prediction=categ+" \n"+prob+' '; print()

        # Reply to the client's request
        reply = prediction +'\n'+ Text_Summary
        conn.sendall(bytes(reply, 'utf-8'))
        conn.close()
        print("Reply:", reply)
        print(); counter=counter+1
        print ("------------------------------------------------------------
```

Sources:

My Code (Github)
https://github.com/PapApostolos/NLP-Android.git

Dataset

https://www.kaggle.com/uciml/news-aggregator-dataset

http://archive.ics.uci.edu/ml/datasets/News+Aggregator

Server/Android
https://en.wikipedia.org/wiki/Client%E2%80%93server_model

https://en.wikipedia.org/wiki/Network_socket

https://developer.android.com/reference/java/net/Socket

https://developer.android.com/reference/android/widget/package-summary

Embeddings:
https://en.wikipedia.org/wiki/Word2vec

https://en.wikipedia.org/wiki/Word_embedding

https://www.tensorflow.org/tutorials/text/word_embeddings

https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526

LSTM/GRU
https://en.wikipedia.org/wiki/Long_short-term_memory

https://en.wikipedia.org/wiki/Gated_recurrent_unit

https://en.wikipedia.org/wiki/Recurrent_neural_network

https://ayearofai.com/rohan-lenny-3-recurrent-neural-networks-10300100899b

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

https://theaisummer.com/understanding-lstm/#lstm-long-short-term-memory-cells