Scientific and practical report on the topic

Tweets Natural Language Processing

Ihor Volokhovych, 1st grade master student, MMAI-1

Abstract. Natural Language Processing, or NLP for short, is broadly defined as the automatic manipulation of natural language, like speech and text, by software. The study of natural language processing has been around for more than 50 years and grew out of the field of linguistics with the rise of computers. In these papers, we discuss GRU, Self-attention, Keras and Bert methods for natural language processing, comparing them and finding out the most optimal way to solve our problem of classification of disastrous tweets.

1. Introduction

Natural language processing (NLP) is a branch of artificial intelligence that helps computers understand, interpret and manipulate human language. NLP draws from many disciplines, including computer science and computational linguistics, in its pursuit to fill the gap between human communication and computer understanding.

Evolution of NLP. While natural language processing isn't a new science, the technology is rapidly advancing thanks to an increased interest in human-to-machine communications, plus an availability of big data, powerful computing and enhanced algorithms.

As a human, you may speak and write in English, Spanish or Chinese. But a computer's native language – known as machine code or machine language – is largely incomprehensible to most people. At your device's lowest levels, communication occurs not with words but through millions of zeros and ones that produce logical actions.

Indeed, programmers used punch cards to communicate with the first computers 70 years ago. This manual and arduous process was understood by a relatively small number of people. Now you can say, "Alexa, I like this song," and a device playing music in your home will lower the volume and reply, "OK. Rating saved," in a humanlike voice. Then it adapts its algorithm to play that song – and others like it – the next time you listen to that music station.

Let's take a closer look at that interaction. Your device activated when it heard you speak, understood the unspoken intent in the comment, executed an action and provided feedback in a well-formed English sentence, all in the space of about five seconds. The complete interaction was made possible by NLP, along with other AI elements such as machine learning and deep learning.

Importance.

1. Large volumes of textual data

Natural language processing helps computers communicate with humans in their own language and scales other language-related tasks. For example, NLP makes it possible for computers to read text, hear speech, interpret it, measure sentiment and determine which parts are important.

Today's machines can analyze more language-based data than humans, without fatigue and in a consistent, unbiased way. Considering the staggering amount of unstructured data that's generated every day, from medical records to social media, automation will be critical to fully analyze text and speech data efficiently.

2. Structuring a highly unstructured data source

Human language is astoundingly complex and diverse. We express ourselves in infinite ways, both verbally and in writing. Not only are there hundreds of languages and dialects, but within each language is a unique set of grammar and syntax rules, terms and slang. When we write, we often misspell or abbreviate words, or omit punctuation. When we speak, we have regional accents, and we mumble, stutter and borrow terms from other languages.

While supervised and unsupervised learning, and specifically deep learning, are now widely used for modeling human language, there's also a need for syntactic and semantic understanding and domain expertise that are not necessarily present in these machine learning approaches. NLP is important because it helps resolve ambiguity in language and adds useful numeric structure to the data for many downstream applications, such as speech recognition or text analytics.

How does NLP work?. Natural language processing includes many different techniques for interpreting human language, ranging from statistical and machine learning methods to rules-based and algorithmic approaches. We need a broad array of approaches because the text- and voice-based data varies widely, as do the practical applications.

Basic NLP tasks include tokenization and parsing, lemmatization/stemming, part-of-speech tagging, language detection and identification of semantic relationships. If you ever diagrammed sentences in grade school, you've done these tasks manually before.

In general terms, NLP tasks break down language into shorter, elemental pieces, try to understand relationships between the pieces and explore how the pieces work together to create meaning.

These underlying tasks are often used in higher-level NLP capabilities, such as:

- **Content categorization.** A linguistic-based document summary, including search and indexing, content alerts and duplication detection.
- **Topic discovery and modeling**. Accurately capture the meaning and themes in text collections, and apply advanced analytics to text, like optimization and forecasting.

- **Contextual extraction.** Automatically pull structured information from text-based sources.
- **Sentiment analysis.** Identifying the mood or subjective opinions within large amounts of text, including average sentiment and opinion mining.
- **Speech-to-text and text-to-speech conversion.** Transforming voice commands into written text, and vice versa.
- **Document summarization.** Automatically generating synopses of large bodies of text.
- **Machine translation.** Automatic translation of text or speech from one language to another.

In all these cases, the overarching goal is to take raw language input and use linguistics and algorithms to transform or enrich the text in such a way that it delivers greater value.

NLP methods and applications. Natural language processing goes hand in hand with text analytics, which counts, groups and categorizes words to extract structure and meaning from large volumes of content. Text analytics is used to explore textual content and derive new variables from raw text that may be visualized, filtered, or used as inputs to predictive models or other statistical methods.

NLP and text analytics are used together for many applications, including:

- Investigative discovery. Identify patterns and clues in emails or written reports to help detect and solve crimes.
- Subject-matter expertise. Classify content into meaningful topics so you can take action and discover trends.
- Social media analytics. Track awareness and sentiment about specific topics and identify key influencers.

There are many common and practical applications of NLP in our everyday lives. Beyond conversing with virtual assistants like Alexa or Siri, here are a few more examples:

- Have you ever looked at the emails in your spam folder and noticed similarities in the subject lines? You're seeing Bayesian spam filtering, a statistical NLP technique that compares the words in spam to valid emails to identify junk mail.
- Have you ever missed a phone call and read the automatic transcript of the voicemail in your email inbox or smartphone app? That's speech-to-text conversion, an NLP capability.
- Have you ever navigated a website by using its built-in search bar, or by selecting suggested topic, entity or category tags? Then you've used NLP methods for search, topic modeling, entity extraction and content categorization.

A subfield of NLP called natural language understanding (NLU) has begun to rise in popularity because of its potential in cognitive and AI applications. NLU goes beyond the structural understanding of language to interpret intent, resolve context and word ambiguity, and

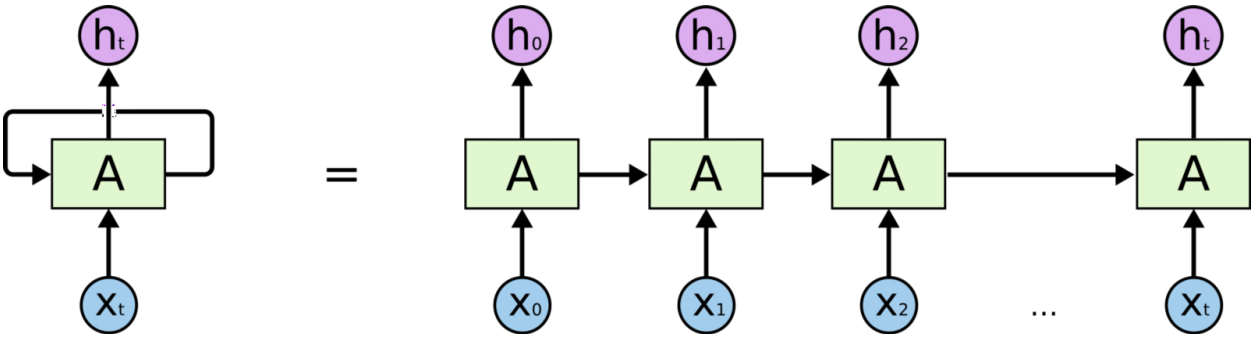
even generate well-formed human language on its own. NLU algorithms must tackle the extremely complex problem of semantic interpretation – that is, understanding the intended meaning of spoken or written language, with all the subtleties, context and inferences that we humans are able to comprehend.

The evolution of NLP toward NLU has a lot of important implications for businesses and consumers alike. Imagine the power of an algorithm that can understand the meaning and nuance of human language in many contexts, from medicine to law to the classroom. As the volumes of unstructured information continue to grow exponentially, we will benefit from computers' tireless ability to help us make sense of it all.

2. Recurrent Neural Networks

In a recurrent neural network we store the output activations from one or more of the layers of the network. Often these are hidden later activations. Then, the next time we feed an input example to the network, we include the previously-stored outputs as additional inputs. You can think of the additional inputs as being concatenated to the end of the "normal" inputs to the previous layer.

For example, if a hidden layer had 10 regular input nodes and 128 hidden nodes in the layer, then it would actually have 138 total inputs (assuming you are feeding the layer's outputs into itself à la Elman) rather than into another layer). Of course, the very first time you try to compute the output of the network you'll need to fill in those extra 128 inputs with zeros or something



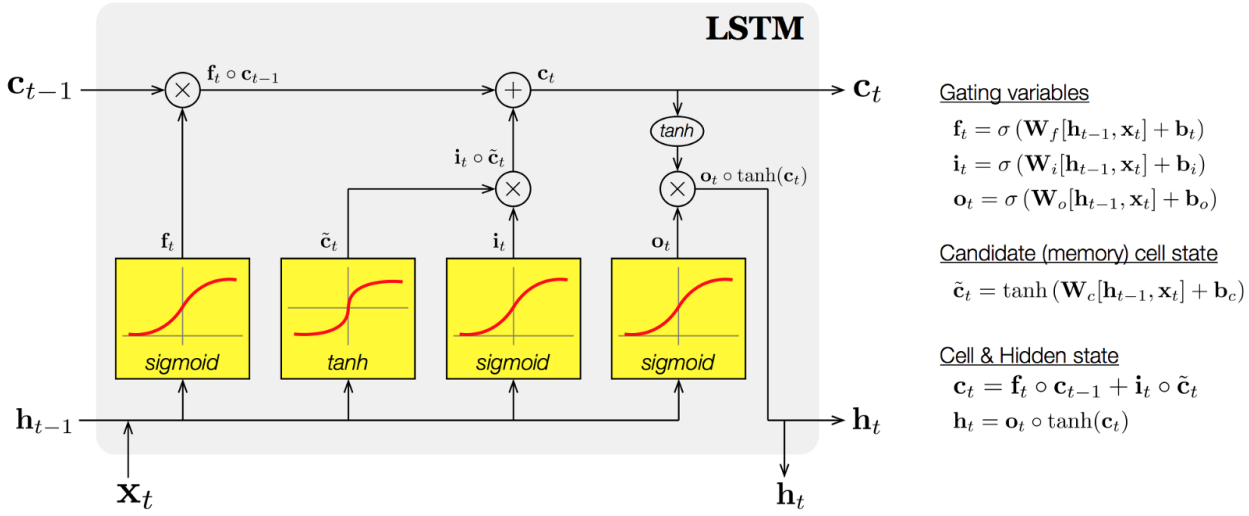
Now, even though RNNs are quite powerful, they suffer from a Vanishing gradient problem which hinders them from using long term information, like they are good for storing 3-4 instances of past iterations but larger numbers of instances don't provide good results so we don't just use regular RNNs. Instead, we use a better variation of RNNs: **Long Short Term Networks(LSTM)**.

2.1 Long Short Term Memory(LSTM)

Long short-term memory (LSTM) units (or blocks) are a building unit for layers of a recurrent neural network (RNN). A RNN composed of LSTM units is often called an LSTM

network. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell is responsible for "remembering" values over arbitrary time intervals; hence the word "memory" in LSTM. Each of the three gates can be thought of as a "conventional" artificial neuron, as in a multi-layer (or feedforward) neural network: that is, they compute an activation (using an activation function) of a weighted sum. Intuitively, they can be thought as regulators of the flow of values that goes through the connections of the LSTM; hence the denotation "gate". There are connections between these gates and the cell.

The expression long short-term refers to the fact that LSTM is a model for the short-term memory which can last for a long period of time. An LSTM is well-suited to classify, process and predict time series given time lags of unknown size and duration between important events. LSTMs were developed to deal with the exploding and vanishing gradient problem when training traditional RNNs.



Components of LSTMs:

- Forget Gate “f” (a neural network with sigmoid)
- Candidate layer “C"(a NN with Tanh)
- Input Gate “I” (a NN with sigmoid)
- Output Gate “O”(a NN with sigmoid)
- Hidden state “H” (a vector)
- Memory state “C” (a vector)
- Inputs to the LSTM cell at any step are Xt (current input) , Ht-1 (previous hidden state) and Ct-1 (previous memory state).
- Outputs from the LSTM cell are Ht (current hidden state) and Ct (current memory state)

Working of gates in LSTMs. First, the LSTM cell takes the previous memory state Ct-1 and does element wise multiplication with forget gate (f) to decide if present memory state Ct. If forget gate value is 0 then previous memory state is completely forgotten else if forget gate

value is 1 then previous memory state is completely passed to the cell (Remember f gate gives values between 0 and 1).

Ct = Ct-1 * ft

Calculating the new memory state:

Ct = Ct + (It * C`t)

Now, we calculate the output:

Ht = tanh(Ct)

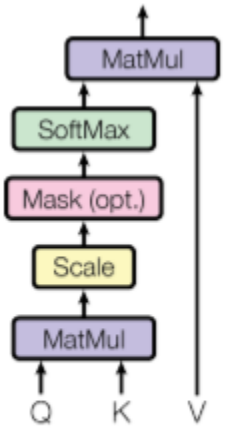
2.2 Gated Recurrent Units (GRU)

In simple words, the GRU unit does not have to use a memory unit to control the flow of information like the LSTM unit. It can directly make use of all hidden states without any control. GRUs have fewer parameters and thus may train a bit faster or need less data to generalize. But, with large data, the LSTMs with higher expressiveness may lead to better results.

They are almost similar to LSTMs except that they have two gates: reset gate and update gate. Reset gate determines how to combine new input to previous memory and update gate determines how much of the previous state to keep. Update gate in GRU is what input gate and forget gate were in LSTM. We don't have the second non linearity in GRU before calculating the output, nor do they have the output gate.

3. Self - attention

The attention mechanism allows output to focus attention on input while producing output while the self-attention model allows inputs to interact with each other (i.e calculate attention of all other inputs wrt one input.



- The first step is multiplying each of the encoder input vectors with three weights matrices (W(Q), W(K), W(V)) that we trained during the training process. This matrix multiplication will give us three vectors for each of the input vectors: the key vector, the query vector, and the value vector.

- The second step in calculating self-attention is to multiply the Query vector of the current input with the key vectors from other inputs.
- In the third step, we will divide the score by the square root of dimensions of the key vector (dk). In the paper the dimension of the key vector is 64, so that will be 8. The reason behind that is if the dot products become large, this causes some self-attention scores to be very small after we apply softmax function in the future.
- In the fourth step, we will apply the softmax function on all self-attention scores we calculated wrt the query word (here first word).
- In the fifth step, we multiply the value vector on the vector we calculated in the previous step.
- In the final step, we sum up the weighted value vectors that we got in the previous step, this will give us the self-attention output for the given word.

The above procedure is applied to all the input sequences. Mathematically, the self-attention matrix for input matrices (Q, K, V) is calculated as:

$$Attention\,(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Where Q, K, V are the concatenation of query, key, and value vectors.

4. BERT

BERT (Bidirectional Encoder Representations from Transformers) is a recent paper published by researchers at Google AI Language. It has caused a stir in the Machine Learning community by presenting state-of-the-art results in a wide variety of NLP tasks, including Question Answering (SQuAD v1.1), Natural Language Inference (MNLI), and others.

BERT's key technical innovation is applying the bidirectional training of Transformer, a popular attention model, to language modelling. This is in contrast to previous efforts which looked at a text sequence either from left to right or combined left-to-right and right-to-left training. The paper's results show that a language model which is bidirectionally trained can have a deeper sense of language context and flow than single-direction language models. In the paper, the researchers detail a novel technique named Masked LM (MLM) which allows bidirectional training in models in which it was previously impossible.

How it works. BERT makes use of Transformer, an attention mechanism that learns contextual relations between words (or sub-words) in a text. In its vanilla form, Transformer includes two separate mechanisms — an encoder that reads the text input and a decoder that produces a prediction for the task. Since BERT's goal is to generate a language model, only the encoder mechanism is necessary. The detailed workings of Transformer are described in a paper by Google.

As opposed to directional models, which read the text input sequentially (left-to-right or right-to-left), the Transformer encoder reads the entire sequence of words at once. Therefore it is considered bidirectional, though it would be more accurate to say that it's non-directional. This

characteristic allows the model to learn the context of a word based on all of its surroundings (left and right of the word).

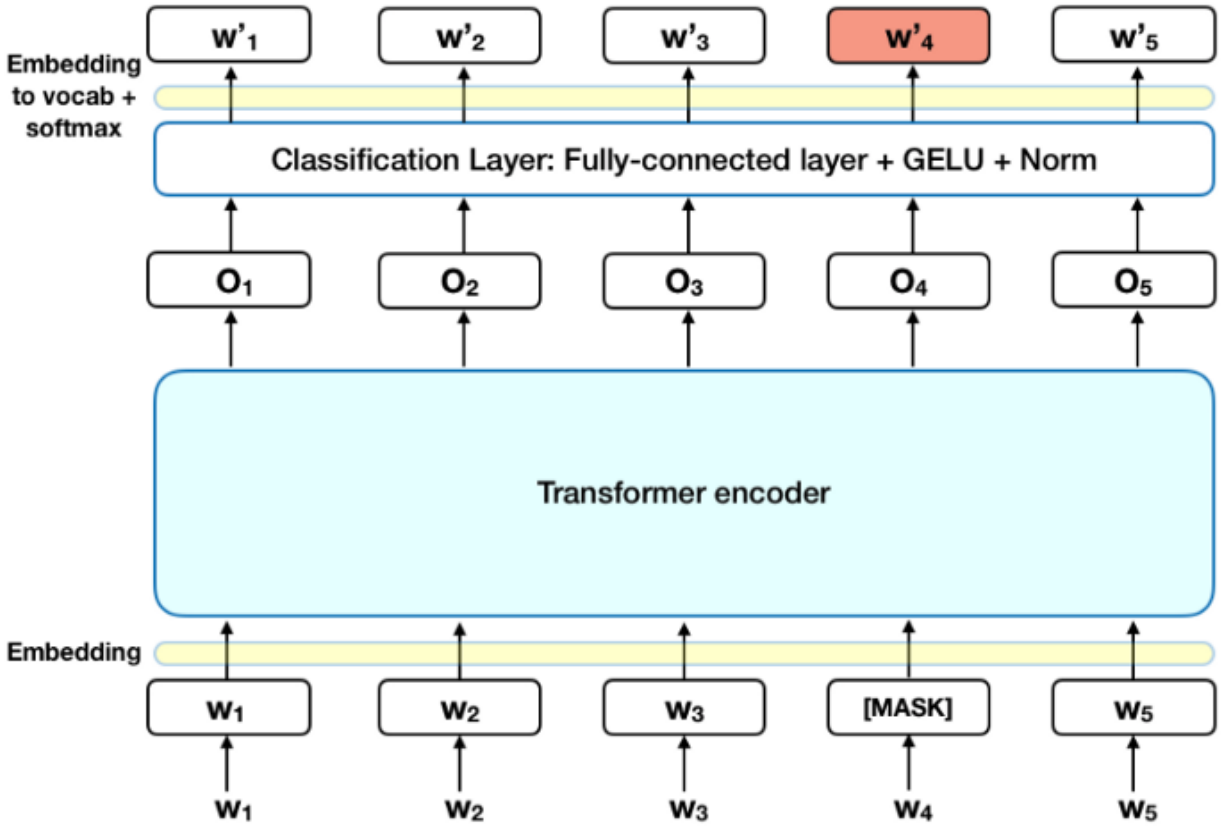
The chart below is a high-level description of the Transformer encoder. The input is a sequence of tokens, which are first embedded into vectors and then processed in the neural network. The output is a sequence of vectors of size H, in which each vector corresponds to an input token with the same index.

When training language models, there is a challenge of defining a prediction goal. Many models predict the next word in a sequence (e.g. “The child came home from ___”), a directional approach which inherently limits context learning. To overcome this challenge, BERT uses two training strategies:

1. Masked LM (MLM)

Before feeding word sequences into BERT, 15% of the words in each sequence are replaced with a [MASK] token. The model then attempts to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence. In technical terms, the prediction of the output words requires:

- Adding a classification layer on top of the encoder output.
- Multiplying the output vectors by the embedding matrix, transforming them into the vocabulary dimension.
- Calculating the probability of each word in the vocabulary with softmax.



The BERT loss function takes into consideration only the prediction of the masked values and ignores the prediction of the non-masked words. As a consequence, the model

converges slower than directional models, a characteristic which is offset by its increased context awareness (see Takeaways #3).

Note: In practice, the BERT implementation is slightly more elaborate and doesn't replace all of the 15% masked words. See Appendix A for additional information.

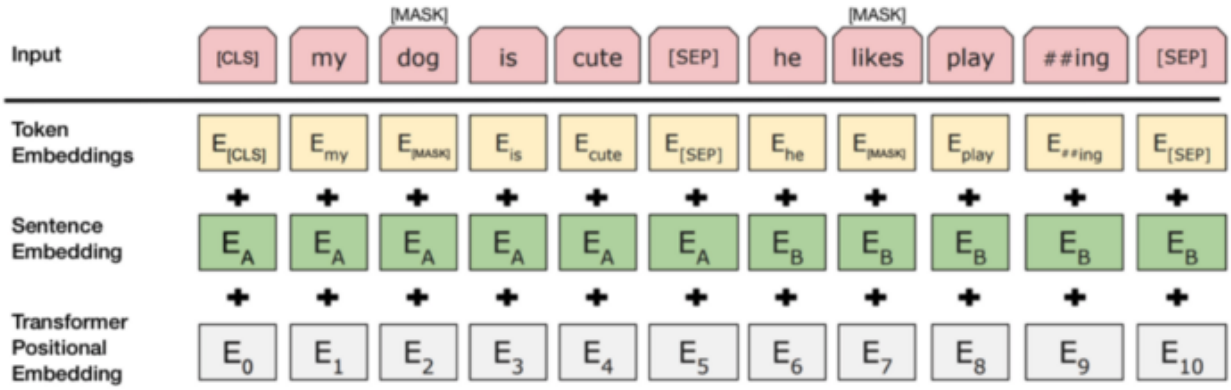
2. Next Sentence Prediction (NSP)

In the BERT training process, the model receives pairs of sentences as input and learns to predict if the second sentence in the pair is the subsequent sentence in the original document.

During training, 50% of the inputs are a pair in which the second sentence is the subsequent sentence in the original document, while in the other 50% a random sentence from the corpus is chosen as the second sentence. The assumption is that the random sentence will be disconnected from the first sentence.

To help the model distinguish between the two sentences in training, the input is processed in the following way before entering the model:

- A [CLS] token is inserted at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
- A sentence embedding indicating Sentence A or Sentence B is added to each token. Sentence embeddings are similar in concept to token embeddings with a vocabulary of
- A positional embedding is added to each token to indicate its position in the sequence. The concept and implementation of positional embedding are presented in the Transformer paper.

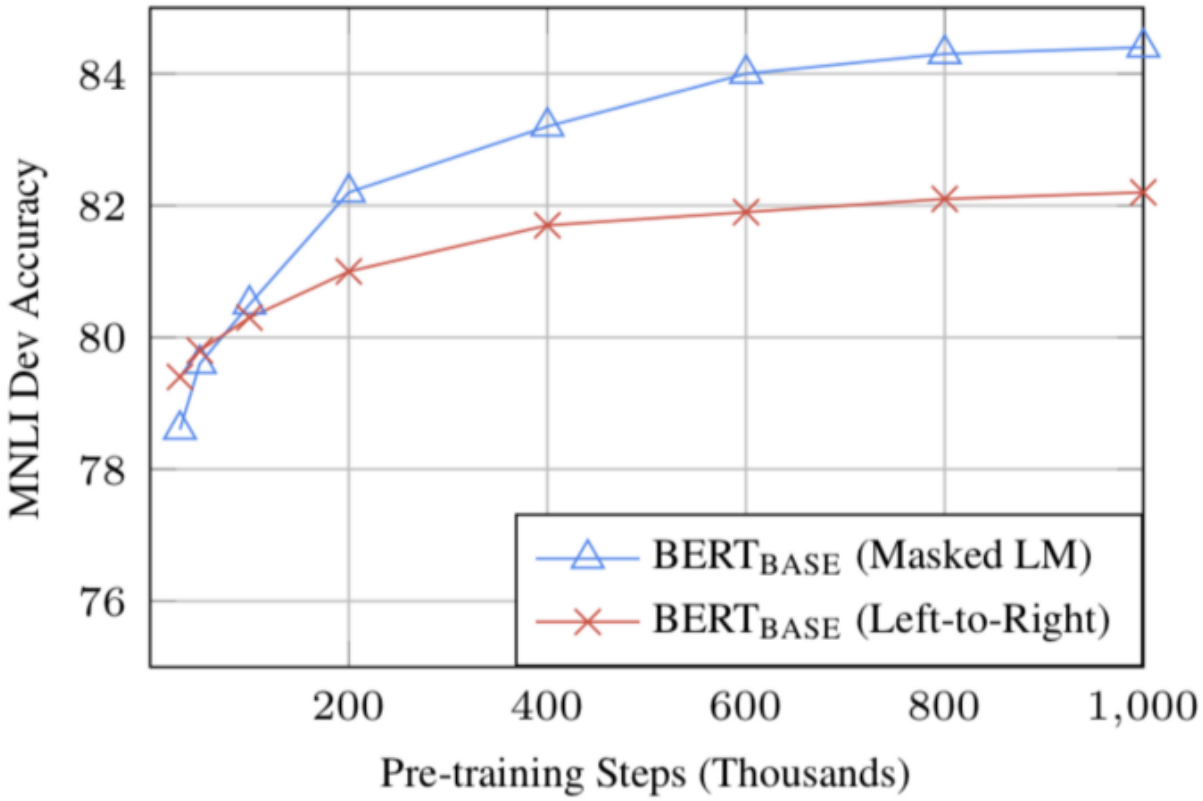


To predict if the second sentence is indeed connected to the first, the following steps are performed:

- The entire input sequence goes through the Transformer model.
- The output of the [CLS] token is transformed into a 2×1 shaped vector, using a simple classification layer (learned matrices of weights and biases).
- Calculating the probability of IsNextSequence with softmax.
- When training the BERT model, Masked LM and Next Sentence Prediction are trained together, with the goal of minimizing the combined loss function of the two strategies.

Takeaways.

1. Model size matters, even at a huge scale. BERT_large, with 345 million parameters, is the largest model of its kind. It is demonstrably superior on small-scale tasks to BERT_base, which uses the same architecture with “only” 110 million parameters.
2. With enough training data, more training steps == higher accuracy. For instance, on the MNLI task, the BERT_base accuracy improves by 1.0% when trained on 1M steps (128,000 words batch size) compared to 500K steps with the same batch size.
3. BERT’s bidirectional approach (MLM) converges slower than left-to-right approaches (because only 15% of words are predicted in each batch) but bidirectional training still outperforms left-to-right training after a small number of pre-training steps.



Compute considerations (training and applying).

	Training Compute + Time	Usage Compute
BERT$_{BASE}$	4 Cloud TPUs, 4 days	1 GPU
BERT$_{LARGE}$	16 Cloud TPUs, 4 days	1 TPU

5. Results

Some training models were created, trained and submitted. Training dataset was retrieved with a huge amount of tweets, split for train and test datasets. Main task: create and test clasificator for disaster tweets.

Dataset

Input data files are available in the read-only "../input/" directory For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory

Dataset was retrieved from Kaggle.

Files:

- train.csv - the training set
- test.csv - the test set
- sample_submission.csv - a sample submission file in the correct format.

Columns:

- id - a unique identifier for each tweet
- text - the text of the tweet
- location - the location the tweet was sent from (may be blank)
- keyword - a particular keyword from the tweet (may be blank)
- target - in train.csv only, this denotes whether a tweet is about a real disaster (1) or not (0)

```
[15] import os
     for dirname, _, filenames in os.walk('/content/drive/MyDrive/twitter/'):
         for filename in filenames:
             print(os.path.join(dirname, filename))

     pd.set_option('display.max_colwidth', None)
     main_path = '/content/drive/MyDrive/twitter/'
     train = pd.read_csv(main_path + 'train.csv', index_col = 'id', sep = ",")
     test =  pd.read_csv(main_path + 'test.csv', index_col = 'id', sep = ",")

     train.shape
     train.head(20)
```

id	keyword	location	text	target
1	NaN	NaN	Our Deeds are the Reason of this #earthquake May ALLAH Forgive us all	1
4	NaN	NaN	Forest fire near La Ronge Sask. Canada	1
5	NaN	NaN	All residents asked to 'shelter in place' are being notified by officers. No other evacuation or shelter in place orders are expected	1
6	NaN	NaN	13,000 people receive #wildfires evacuation orders in California	1
7	NaN	NaN	Just got sent this photo from Ruby #Alaska as smoke from #wildfires pours into a school	1
8	NaN	NaN	#RockyFire Update => California Hwy. 20 closed in both directions due to Lake County fire - #CAfire #wildfires	1
10	NaN	NaN	#flood #disaster Heavy rain causes flash flooding of streets in Manitou, Colorado Springs areas	1
13	NaN	NaN	I'm on top of the hill and I can see a fire in the woods...	1
14	NaN	NaN	There's an emergency evacuation happening now in the building across the street	1
15	NaN	NaN	I'm afraid that the tornado is coming to our area...	1
16	NaN	NaN	Three people died from the heat wave so far	1
17	NaN	NaN	Haha South Tampa is getting flooded hah- WAIT A SECOND I LIVE IN SOUTH TAMPA WHAT AM I GONNA DO WHAT AM I GONNA DO FVCK #flooding	1
18	NaN	NaN	#raining #flooding #Florida #TampaBay #Tampa 18 or 19 days. I've lost count	1
19	NaN	NaN	#Flood in Bago Myanmar #We arrived Bago	1
20	NaN	NaN	Damage to school bus on 80 in multi car crash #BREAKING	1
23	NaN	NaN	What's up man?	0
24	NaN	NaN	I love fruits	0
25	NaN	NaN	Summer is lovely	0
26	NaN	NaN	My car is so fast	0
28	NaN	NaN	What a goooooooaaaaaal!!!!!!	0

1. GRU-SelfAttention-Keras

Downloading embeddings

1. Download glove twitter embeddings
2. Takes about 5 minutes to execute, for 100-dim twitter vectors
3. Takes about 10+ minutes to execute, for 200-dim twitter vectors
4. glove_vectors_100 = gensim.downloader.load('glove-twitter-100')

```
[ ]  pprint(list(gensim.downloader.info()['models'].keys()))
```

```
['fasttext-wiki-news-subwords-300',
 'conceptnet-numberbatch-17-06-300',
 'word2vec-ruscorpora-300',
 'word2vec-google-news-300',
 'glove-wiki-gigaword-50',
 'glove-wiki-gigaword-100',
 'glove-wiki-gigaword-200',
 'glove-wiki-gigaword-300',
 'glove-twitter-25',
 'glove-twitter-50',
 'glove-twitter-100',
 'glove-twitter-200',
 '__testing_word2vec-matrix-synopsis']
```

```
[ ]  glove_vectors = gensim.downloader.load('glove-twitter-100')
     glove = glove_vectors
     embedding_length = 100
```

```
[==================================================] 100.0% 387.1/387.1MB downloaded
```

Text preprocessing

1. Word tokenization
2. Filtering out punctuation
3. Making words lowercase
4. Removing stopwords
5. Concatenating list of words

```
text_train = np.array(train['text'])
text_test = np.array(test['text'])
text = np.concatenate((text_train, text_test), axis = 0)
print(text.shape)

# word tokenization
for i in range(len(text)):
    text[i] = word_tokenize(text[i])
print('After tokenization:')
print(text[0])

# filter out punctuation
for i in range(len(text)):
    text[i] = [word for word in text[i] if word.isalpha()]
print('After filtering out punctuation:')
print(text[0])

# make words lowercase
for i in range(len(text)):
    text[i] = [word.lower() for word in text[i]]
print('After making lowercase:')
print(text[0])

# remove stopwords
for i in range(len(text)):
    text[i] = [word for word in text[i] if not word in stop_words]
print('After removing stopwords:')
print(text[0])

# concatenate list of words
for i in range(len(text)):
    text_concat = ''
    for word in text[i]:
        text_concat += word + ' '
    text[i] = text_concat
print('After concatenating words:')
text = np.array(text)
```

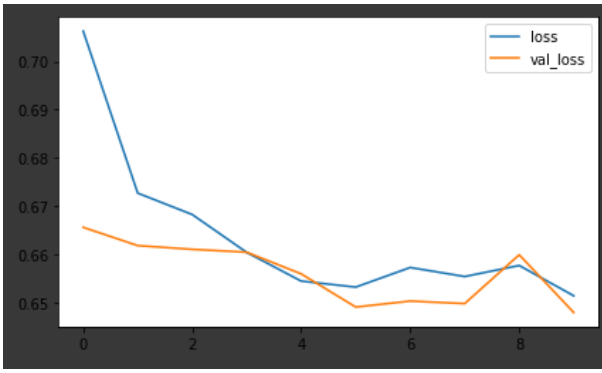
Creating Model

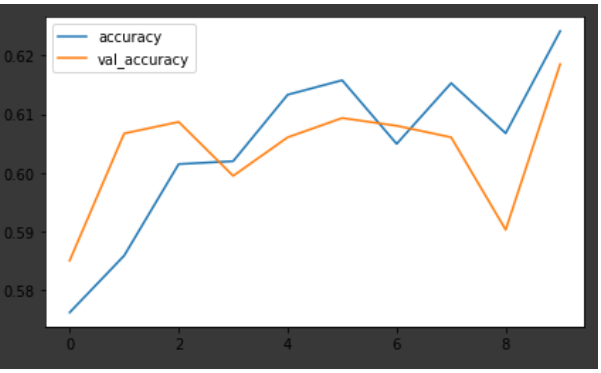
(Average the embeddings for each word of the tweet and learn to classify using a feedforward NN model)

```
act = 'tanh'
batch_len = 32
opt = 'adam'
epoch = 10
val_split = 0.2

keras.backend.clear_session()
inputs = keras.Input(shape = (X_train.shape[1]))
x = layers.Dense(32, activation = act)(inputs)
x = layers.Dense(32, activation = act)(x)
x = layers.Dense(16, activation = act)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs = inputs, outputs = outputs, name = 'Deep-Averaging-Network')
```

```
[ ] model.compile(optimizer = opt, loss = 'binary_crossentropy', metrics = ['accuracy'])
    fit_model_first = model.fit(X_train, y_train, epochs = epoch, batch_size = batch_len, verbose = 1, validation_split=val_split)
```

```
Epoch 1/10
191/191 [==============================] - 3s 6ms/step - loss: 0.7063 - accuracy: 0.5762 - val_loss: 0.6657 - val_accuracy: 0.5850
Epoch 2/10
191/191 [==============================] - 1s 5ms/step - loss: 0.6728 - accuracy: 0.5859 - val_loss: 0.6619 - val_accuracy: 0.6067
Epoch 3/10
191/191 [==============================] - 1s 5ms/step - loss: 0.6683 - accuracy: 0.6015 - val_loss: 0.6612 - val_accuracy: 0.6087
Epoch 4/10
191/191 [==============================] - 1s 5ms/step - loss: 0.6605 - accuracy: 0.6020 - val_loss: 0.6606 - val_accuracy: 0.5995
Epoch 5/10
191/191 [==============================] - 1s 5ms/step - loss: 0.6546 - accuracy: 0.6133 - val_loss: 0.6561 - val_accuracy: 0.6060
Epoch 6/10
191/191 [==============================] - 1s 5ms/step - loss: 0.6533 - accuracy: 0.6158 - val_loss: 0.6492 - val_accuracy: 0.6093
Epoch 7/10
191/191 [==============================] - 1s 5ms/step - loss: 0.6574 - accuracy: 0.6049 - val_loss: 0.6505 - val_accuracy: 0.6080
Epoch 8/10
191/191 [==============================] - 1s 5ms/step - loss: 0.6555 - accuracy: 0.6153 - val_loss: 0.6499 - val_accuracy: 0.6060
Epoch 9/10
191/191 [==============================] - 1s 5ms/step - loss: 0.6578 - accuracy: 0.6067 - val_loss: 0.6600 - val_accuracy: 0.5903
Epoch 10/10
191/191 [==============================] - 1s 4ms/step - loss: 0.6516 - accuracy: 0.6241 - val_loss: 0.6481 - val_accuracy: 0.6185
```



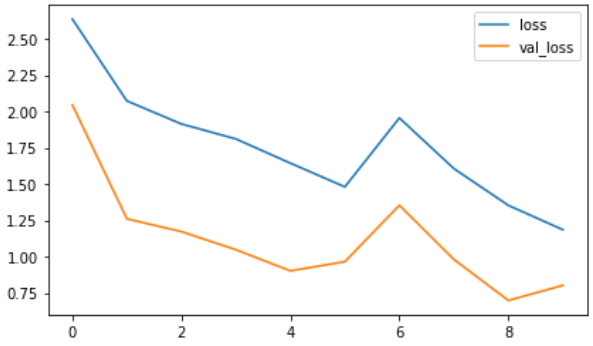


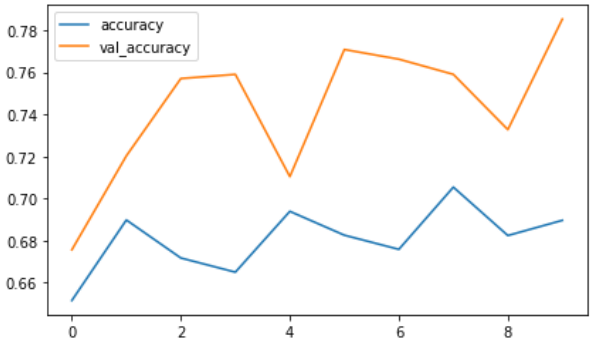
Creating Model (Concatenated Embeddings Model)

```
act = 'tanh'
batch_len = 32
opt = 'adam'
epoch = 10
val_split = 0.2

keras.backend.clear_session()
inputs = keras.Input(shape = (X_train.shape[1]))
x = layers.Dense(32, activation = act, input_dim = X_train.shape[1])(inputs)
x = layers.Dropout(0.3)(x)
x = layers.Dense(32, activation = act)(x)
x = layers.Dropout(0.3)(x)
x = layers.Dense(16, activation = act)(x)
x = layers.Dropout(0.2)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs = inputs, outputs = outputs, name = 'Glove-FFN-Concatenated')
```

```
[ ] model.compile(optimizer = opt, loss = 'binary_crossentropy', metrics = ['accuracy'])
    fit_model_second = model.fit(X_train, y_train, epochs = epoch, batch_size = batch_len, verbose = 1, validation_split=val_split)
```

```
Epoch 1/10
191/191 [==============================] - 2s 6ms/step - loss: 2.6357 - accuracy: 0.6516 - val_loss: 2.0443 - val_accuracy: 0.6756
Epoch 2/10
191/191 [==============================] - 1s 5ms/step - loss: 2.0722 - accuracy: 0.6898 - val_loss: 1.2602 - val_accuracy: 0.7203
Epoch 3/10
191/191 [==============================] - 1s 5ms/step - loss: 1.9137 - accuracy: 0.6718 - val_loss: 1.1720 - val_accuracy: 0.7571
Epoch 4/10
191/191 [==============================] - 1s 5ms/step - loss: 1.8103 - accuracy: 0.6650 - val_loss: 1.0473 - val_accuracy: 0.7590
Epoch 5/10
191/191 [==============================] - 1s 5ms/step - loss: 1.6436 - accuracy: 0.6939 - val_loss: 0.9016 - val_accuracy: 0.7104
Epoch 6/10
191/191 [==============================] - 1s 5ms/step - loss: 1.4804 - accuracy: 0.6826 - val_loss: 0.9653 - val_accuracy: 0.7708
Epoch 7/10
191/191 [==============================] - 1s 5ms/step - loss: 1.9550 - accuracy: 0.6759 - val_loss: 1.3534 - val_accuracy: 0.7663
Epoch 8/10
191/191 [==============================] - 1s 5ms/step - loss: 1.6062 - accuracy: 0.7054 - val_loss: 0.9802 - val_accuracy: 0.7590
Epoch 9/10
191/191 [==============================] - 1s 5ms/step - loss: 1.3528 - accuracy: 0.6824 - val_loss: 0.6978 - val_accuracy: 0.7328
Epoch 10/10
191/191 [==============================] - 1s 5ms/step - loss: 1.1851 - accuracy: 0.6897 - val_loss: 0.8020 - val_accuracy: 0.7853
```





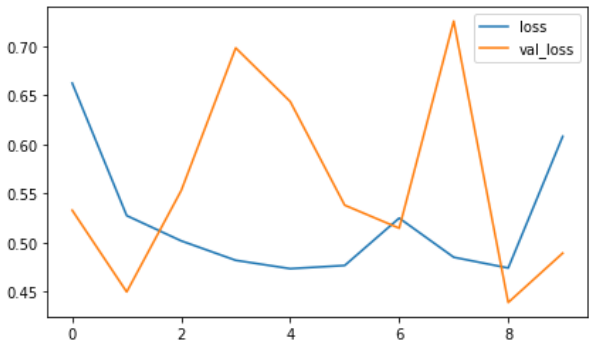
Creating and training LSTM model

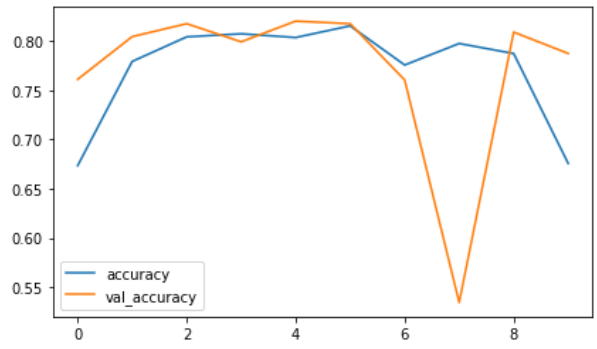
(Using only the text feature)

```
[ ] act = 'tanh'
    batch_len = 32
    opt = 'adam'
    epoch = 10
    val_split = 0.2

    keras.backend.clear_session()
    inputs = keras.Input(shape = (22, embedding_length))
    x = layers.LSTM(64)(inputs)
    x = layers.Dense(32, activation = act)(x)
    x = layers.Dense(16, activation = act)(x)
    outputs = layers.Dense(1)(x)
    model = keras.Model(inputs = inputs, outputs = outputs, name = 'Glove-LSTM')
```

```
[ ] model.compile(optimizer = opt, loss = 'binary_crossentropy', metrics = ['accuracy'])
    fit_model_third = model.fit(X_train, y_train, epochs = epoch, batch_size = batch_len, verbose = 1, validation_split=val_split)
```

```
Epoch 1/10
191/191 [==============================] - 6s 13ms/step - loss: 0.6622 - accuracy: 0.6734 - val_loss: 0.5328 - val_accuracy: 0.7610
Epoch 2/10
191/191 [==============================] - 2s 10ms/step - loss: 0.5274 - accuracy: 0.7791 - val_loss: 0.4497 - val_accuracy: 0.8043
Epoch 3/10
191/191 [==============================] - 2s 10ms/step - loss: 0.5017 - accuracy: 0.8041 - val_loss: 0.5529 - val_accuracy: 0.8175
Epoch 4/10
191/191 [==============================] - 2s 10ms/step - loss: 0.4819 - accuracy: 0.8072 - val_loss: 0.6981 - val_accuracy: 0.7991
Epoch 5/10
191/191 [==============================] - 2s 10ms/step - loss: 0.4734 - accuracy: 0.8034 - val_loss: 0.6435 - val_accuracy: 0.8201
Epoch 6/10
191/191 [==============================] - 2s 10ms/step - loss: 0.4766 - accuracy: 0.8153 - val_loss: 0.5380 - val_accuracy: 0.8175
Epoch 7/10
191/191 [==============================] - 2s 9ms/step - loss: 0.5249 - accuracy: 0.7755 - val_loss: 0.5146 - val_accuracy: 0.7603
Epoch 8/10
191/191 [==============================] - 2s 10ms/step - loss: 0.4850 - accuracy: 0.7974 - val_loss: 0.7254 - val_accuracy: 0.5345
Epoch 9/10
191/191 [==============================] - 2s 9ms/step - loss: 0.4741 - accuracy: 0.7872 - val_loss: 0.4390 - val_accuracy: 0.8089
Epoch 10/10
191/191 [==============================] - 2s 10ms/step - loss: 0.6080 - accuracy: 0.6757 - val_loss: 0.4892 - val_accuracy: 0.7873
```



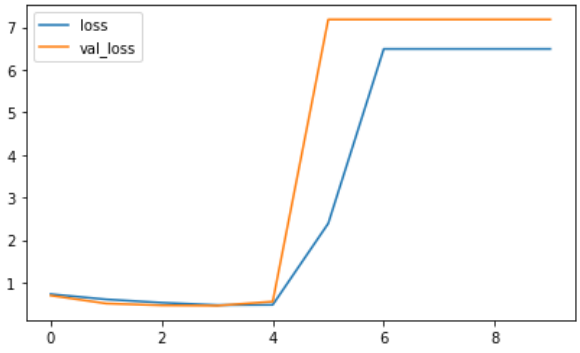


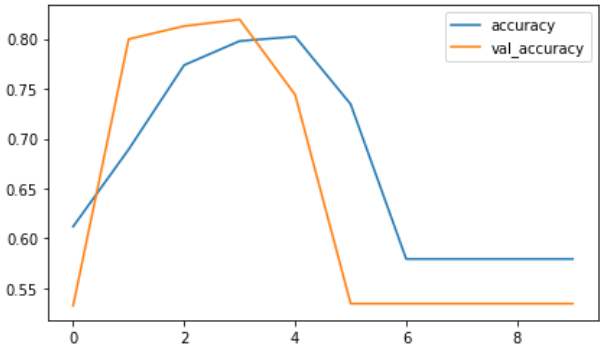
Creating and training GRU model

```
act = 'tanh'
batch_len = 16
opt = 'adam'
epoch = 10
val_split = 0.2

keras.backend.clear_session()
inputs = keras.Input(shape = (22, embedding_length))
x = layers.GRU(64)(inputs)
x = layers.Dense(32, activation = act)(x)
x = layers.Dense(16, activation = act)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs = inputs, outputs = outputs, name = 'Glove-GRU')
```

```
[ ] model.compile(optimizer = opt, loss = 'binary_crossentropy', metrics = ['accuracy'])
    fit_model_fourth = model.fit(X_train, y_train, epochs = epoch, batch_size = batch_len, verbose = 1, validation_split=val_split)
```

```
Epoch 1/10
381/381 [==============================] - 6s 10ms/step - loss: 0.7254 - accuracy: 0.6118 - val_loss: 0.6934 - val_accuracy: 0.5325
Epoch 2/10
381/381 [==============================] - 3s 9ms/step - loss: 0.6026 - accuracy: 0.6892 - val_loss: 0.5054 - val_accuracy: 0.7997
Epoch 3/10
381/381 [==============================] - 3s 9ms/step - loss: 0.5246 - accuracy: 0.7737 - val_loss: 0.4622 - val_accuracy: 0.8129
Epoch 4/10
381/381 [==============================] - 4s 10ms/step - loss: 0.4684 - accuracy: 0.7979 - val_loss: 0.4550 - val_accuracy: 0.8194
Epoch 5/10
381/381 [==============================] - 3s 9ms/step - loss: 0.4742 - accuracy: 0.8023 - val_loss: 0.5515 - val_accuracy: 0.7439
Epoch 6/10
381/381 [==============================] - 4s 10ms/step - loss: 2.3870 - accuracy: 0.7345 - val_loss: 7.1808 - val_accuracy: 0.5345
Epoch 7/10
381/381 [==============================] - 3s 9ms/step - loss: 6.4891 - accuracy: 0.5793 - val_loss: 7.1808 - val_accuracy: 0.5345
Epoch 8/10
381/381 [==============================] - 3s 9ms/step - loss: 6.4891 - accuracy: 0.5793 - val_loss: 7.1808 - val_accuracy: 0.5345
Epoch 9/10
381/381 [==============================] - 4s 10ms/step - loss: 6.4891 - accuracy: 0.5793 - val_loss: 7.1808 - val_accuracy: 0.5345
Epoch 10/10
381/381 [==============================] - 4s 10ms/step - loss: 6.4891 - accuracy: 0.5793 - val_loss: 7.1808 - val_accuracy: 0.5345
```



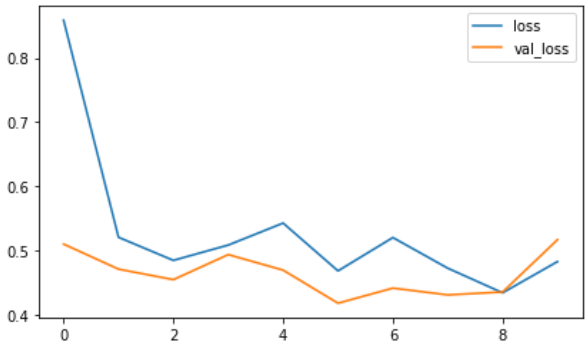


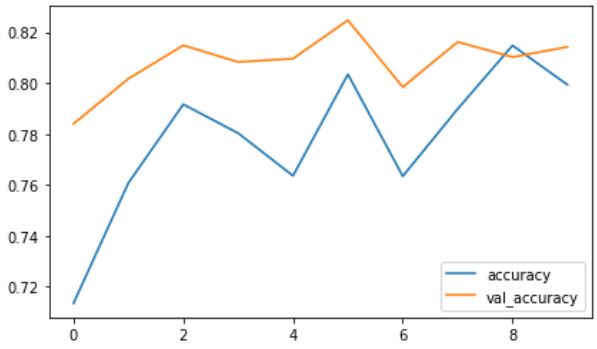
LSTM model with self-attention

```
act = 'tanh'
batch_len = 16
opt = 'adam'
epoch = 10
val_split = 0.2

keras.backend.clear_session()
inputs = keras.Input(shape = (22, embedding_length))
x = layers.GRU(64, return_sequences = True)(inputs)
x = SeqSelfAttention(attention_activation = 'tanh')(x)
x = layers.GlobalMaxPool1D()(x)
x = layers.Dense(32, activation = act)(x)
x = layers.Dense(16, activation = act)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs = inputs, outputs = outputs, name = 'GRU-self-attention')
```

```
[ ] model.compile(optimizer = opt, loss = 'binary_crossentropy', metrics = ['accuracy'])
    fit_model_fifth = model.fit(X_train, y_train, epochs = epoch, batch_size = batch_len, verbose = 1, validation_split=val_split)
```

```
Epoch 1/10
381/381 [==============================] - 8s 13ms/step - loss: 0.8586 - accuracy: 0.7135 - val_loss: 0.5097 - val_accuracy: 0.7840
Epoch 2/10
381/381 [==============================] - 4s 11ms/step - loss: 0.5203 - accuracy: 0.7608 - val_loss: 0.4707 - val_accuracy: 0.8017
Epoch 3/10
381/381 [==============================] - 4s 11ms/step - loss: 0.4844 - accuracy: 0.7916 - val_loss: 0.4543 - val_accuracy: 0.8148
Epoch 4/10
381/381 [==============================] - 4s 11ms/step - loss: 0.5082 - accuracy: 0.7803 - val_loss: 0.4933 - val_accuracy: 0.8083
Epoch 5/10
381/381 [==============================] - 4s 11ms/step - loss: 0.5425 - accuracy: 0.7635 - val_loss: 0.4691 - val_accuracy: 0.8096
Epoch 6/10
381/381 [==============================] - 4s 11ms/step - loss: 0.4680 - accuracy: 0.8034 - val_loss: 0.4176 - val_accuracy: 0.8247
Epoch 7/10
381/381 [==============================] - 4s 11ms/step - loss: 0.5199 - accuracy: 0.7634 - val_loss: 0.4411 - val_accuracy: 0.7984
Epoch 8/10
381/381 [==============================] - 4s 11ms/step - loss: 0.4722 - accuracy: 0.7898 - val_loss: 0.4306 - val_accuracy: 0.8162
Epoch 9/10
381/381 [==============================] - 4s 11ms/step - loss: 0.4338 - accuracy: 0.8148 - val_loss: 0.4351 - val_accuracy: 0.8102
Epoch 10/10
381/381 [==============================] - 4s 11ms/step - loss: 0.4824 - accuracy: 0.7993 - val_loss: 0.5165 - val_accuracy: 0.8142
```

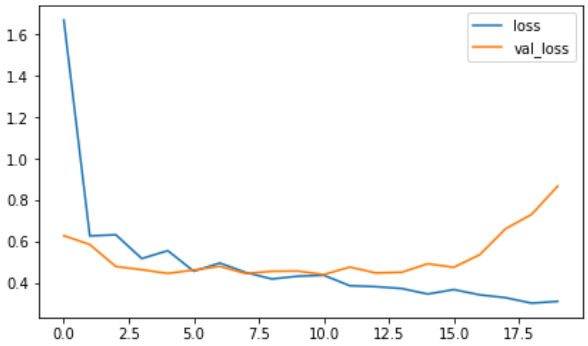


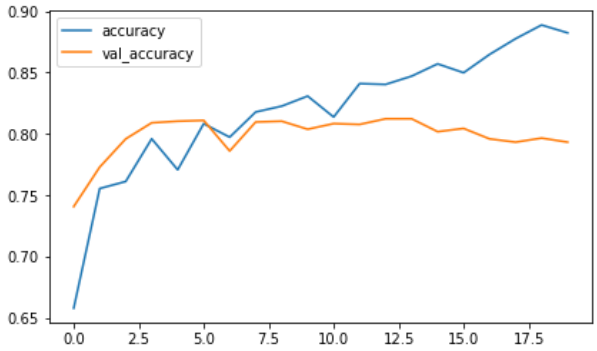


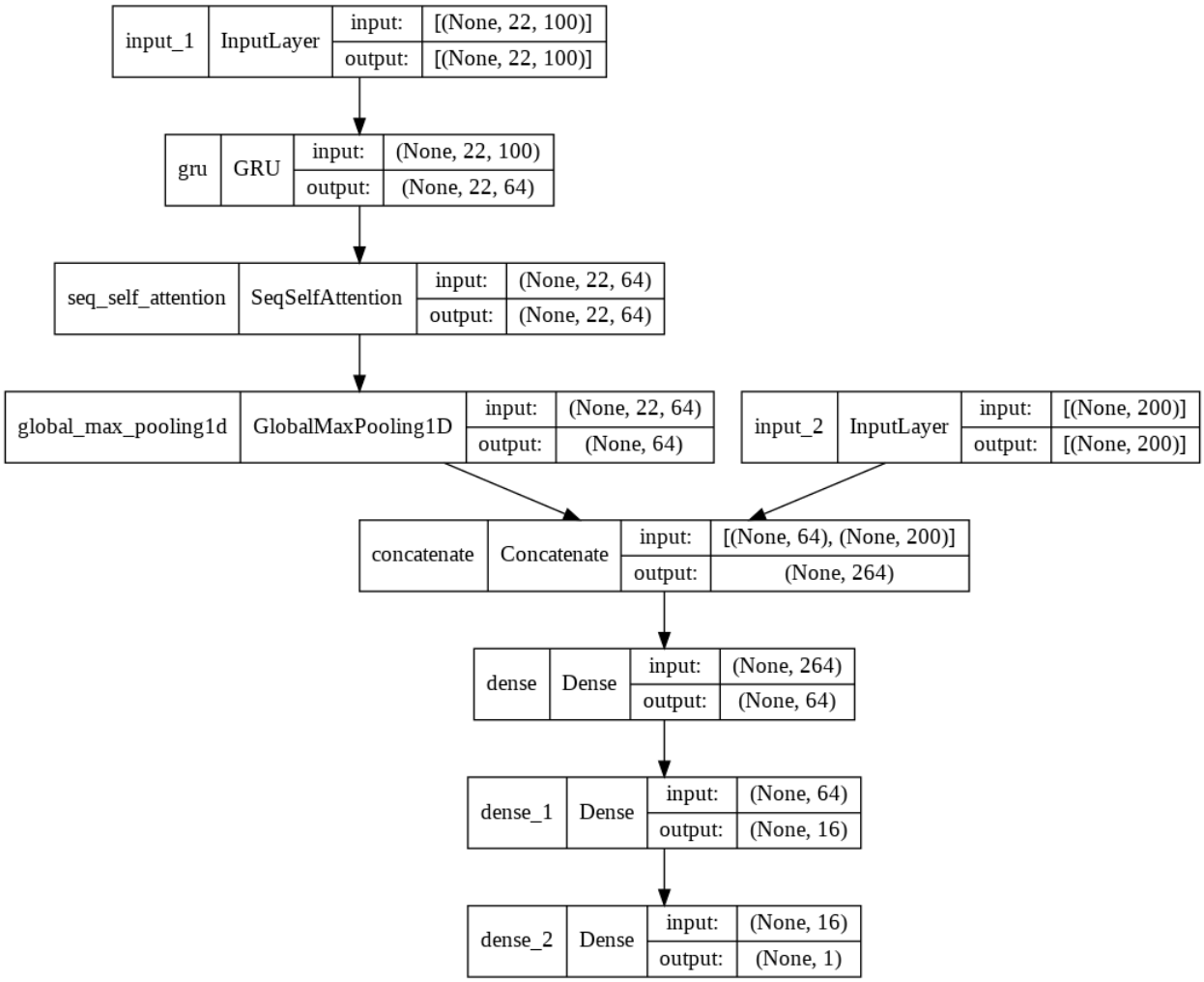
Non-linear LSTM model (with both seq and non-seq inputs)

```
act = 'tanh'
batch_len = 16
opt = 'adam'
epoch = 20
val_split = 0.2

keras.backend.clear_session()
seq_input = keras.Input(shape = (22, embedding_length))
non_seq_input = keras.Input(shape = (2 * embedding_length))
x = layers.GRU(64, return_sequences = True)(seq_input)
x = SeqSelfAttention(attention_activation = 'tanh')(x)
x = layers.GlobalMaxPool1D()(x)
x = layers.concatenate([x, non_seq_input])
x = layers.Dense(64, activation = act)(x)
x = layers.Dense(16, activation = act)(x)
output = layers.Dense(1)(x)
model = keras.Model(inputs = [seq_input, non_seq_input], outputs = output, name = 'complete_model')

model.compile(optimizer = opt, loss = 'binary_crossentropy', metrics = ['accuracy'])
fit_model_sixth = model.fit([X_train, non_sequential_train], y_train, epochs = epoch, batch_size = batch_len, verbose = 1, validation_
```

```
Epoch 1/20
381/381 [==============================] - 8s 13ms/step - loss: 1.6705 - accuracy: 0.6578 - val_loss: 0.6268 - val_accuracy: 0.7406
Epoch 2/20
381/381 [==============================] - 4s 11ms/step - loss: 0.6256 - accuracy: 0.7553 - val_loss: 0.5841 - val_accuracy: 0.7728
Epoch 3/20
381/381 [==============================] - 4s 12ms/step - loss: 0.6315 - accuracy: 0.7611 - val_loss: 0.4779 - val_accuracy: 0.7958
Epoch 4/20
381/381 [==============================] - 4s 11ms/step - loss: 0.5161 - accuracy: 0.7959 - val_loss: 0.4623 - val_accuracy: 0.8089
Epoch 5/20
381/381 [==============================] - 4s 12ms/step - loss: 0.5545 - accuracy: 0.7706 - val_loss: 0.4444 - val_accuracy: 0.8102
Epoch 6/20
381/381 [==============================] - 4s 11ms/step - loss: 0.4548 - accuracy: 0.8082 - val_loss: 0.4608 - val_accuracy: 0.8109
Epoch 7/20
381/381 [==============================] - 4s 11ms/step - loss: 0.4946 - accuracy: 0.7972 - val_loss: 0.4786 - val_accuracy: 0.7859
Epoch 8/20
381/381 [==============================] - 4s 11ms/step - loss: 0.4488 - accuracy: 0.8177 - val_loss: 0.4437 - val_accuracy: 0.8096
Epoch 9/20
381/381 [==============================] - 4s 11ms/step - loss: 0.4174 - accuracy: 0.8225 - val_loss: 0.4548 - val_accuracy: 0.8102
Epoch 10/20
```






input_1 InputLayer input: [(None, 22, 100)] output: [(None, 22, 100)]
gru GRU input: (None, 22, 100) output: (None, 22, 64)
seq_self_attention SeqSelfAttention input: (None, 22, 64) output: (None, 22, 64)
global_max_pooling1d GlobalMaxPooling1D input: (None, 22, 64) output: (None, 64)
input_2 InputLayer input: [(None, 200)] output: [(None, 200)]
concatenate Concatenate input: [(None, 64), (None, 200)] output: (None, 264)
dense Dense input: (None, 264) output: (None, 64)
dense_1 Dense input: (None, 64) output: (None, 16)
dense_2 Dense input: (None, 16) output: (None, 1)

Results.

```
null accuracy: [0.547194]
Accuracy: 84.09%
              precision    recall  f1-score   support

          -1       0.00      0.00      0.00         1
           0       1.00      0.80      0.89      2541
           1       0.58      1.00      0.74       721

    accuracy                           0.84      3263
   macro avg       0.53      0.60      0.54      3263
weighted avg       0.91      0.84      0.85      3263

/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classificatio
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classificatio
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classificatio
  _warn_prf(average, modifier, msg_start, len(result))
```

	0	1	2
0	0	1	0
1	0	2023	518
2	0	0	721

2. BERT

1. Create features for BERT
2. Build the model in Keras
3. Model tuning and cross validation
4. Make prediction for test set

```
null accuracy: 0.5700525394045535
Accuracy: 81.09%
roc auc: 0.7975603880603551
              precision    recall  f1-score   support

           0       0.80      0.89      0.84       651
           1       0.83      0.70      0.76       491

    accuracy                           0.81      1142
   macro avg       0.82      0.80      0.80      1142
weighted avg       0.81      0.81      0.81      1142
```

	0	1
0	581	70
1	146	345