Comparison of Bidirectional NLP Language Models: BERT & ElMo

Shehroz Bin Reza

Submitted for the Degree of Master of Science in

Data Science and Analytics

Department of Computer Science
Royal Holloway University of London
Egham, Surrey TW20 0EX, UK

August 20, 2019

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 10270

Student Name: **Shehroz Bin Reza**

Date of Submission: 03/09/2019

Signature:

Abstract

The main goal of this project is to compare two of the most powerful language models which are in existence today. The Google's BERT and Allen's ELMo. As both of them are bidirectional language model but still very different from each other. The ELMo follows more traditional design and uses LSTMs for its computation of word embeddings while state of the art BERT uses Transformers which involves in two approach for the language modelling.

The ELMo uses two layers each of them consists of forward and backward pass to compute the intermediate word vectors. It helps the ELMo to achieve high efficiency compared to other traditional language models.

On the other hand, the BERT uses the transformer for the same thing the ELMo does. But instead of having two separate forward and backward pass layer, it uses single layer and analyses the a given sentence from both sides using the unique masking technique. In the masking technique random words are masked in order to prevent the model to see the word indirectly to itself only.

The dataset IMDB Large Movie Review Dataset used to compare the two different models is a well-balanced dataset which is hosted by the Stanford University, US. It consists of variety of strings that help us to check performance of the language model in handling different context of the situation. The main comparison factor in it is the accuracy. Accuracy is the benchmark for this project.

Contents

1 Introduction 5

1.1 Origin 5

1.2 Application of NLP 6

1.3 Present and Future of NLP 7

1.4 Challenges 8

2 Background Research 9

2.1 LSTM 9

2.1.1 Forward Pass 9

2.1.2 Backpropagation 10

2.2 Transformers 11

2.3 BERT 14

2.4 ElMo 18

3 Implementation 20

3.1 Objective 20

3.2 Requirements 20

3.3 Steps 21

3.4 Outcome 27

3.5 Conclusion 27

4 Self-Assessment 29

5 Professional Ethics 31

6 References 33

7 Appendix .... Error! Bookmark not defined.

1 Introduction

According to M. Chandhana Surabhi [1] Natural Language Processing, generally known as NLP, is a branch of artificial intelligence that facilitates the interaction between computers and humans by using the same language we humans use, that is also known as natural language. Therefore, in simple sense NLP makes human to interact with the machine easily. There are many applications developed in past few decades in NLP. Most of these are very useful in everyday life for example a machine that takes instructions by voice like many different virtual assistants now a days are being used in our daily driver smartphones and so on.

Another researcher from Syracuse University, New York, Elizabeth D. Liddy [2] defines Natural Language Processing is a theoretically motivated range of computational techniques for analysing and representing naturally occurring texts at one or more levels of linguistic analysis for the purpose of achieving human-like language processing for a range of tasks or applications.
The NLP can be divided into three areas [1];

- Acoustic – Phonetic
- Morphological – Syntactic
- Semantic - Pragmatic

Acoustic - Phonetic: where acoustic knowledge studies rhythm and intonation of language; i.e. how to form phonemes, the smallest unit of sounds. Phonemes and phones are aggregated into word sounds. Phonetic knowledge relates sounds to the words we recognize.
Morphological - Syntactic: Morphology is lexical knowledge which studies sub words (morphemes) that would form a word. Syntactic knowledge studies the structural roles of words or collection of words to form correct sentences.

Semantic - Pragmatic: Semantic knowledge deals with the meaning of words and sentences, while pragmatic knowledge deals with deriving sentence meanings from the outside world or outside the content of the document.

1.1 Origin

The roots of Natural language Processing goes way back to 17th century, when two researchers Leibniz and Descartes proposed an idea that would relate words of different languages, but the realistic sense of idea only understood in 20th century during the world war time. Machine translation was or believed the first computer-based application for the natural language. Weaver and Booth started one of earliest MT projects in 1946, on computer translation based on expertise in breaking enemy codes during World War II. However, a general agreement was made that, Weaver's memorandum of 1949 has brought the idea of MT to general notice and had inspired many projects. Weaver proposed using ideas from information and cryptography theory for language translation for natural language translation [3]. According to Liddy [2] earliest works in MT followed the basic view, that the only difference between languages was vested in their vocabularies and the permitted word orders. So, the machines who were made using this idea basically used a lookup method, mainly the dictionary lookup for appropriate words for translation and reorganising of words after translation to match the target of word order of the target languages. Although this idea was quite new, this was done without considering the lexical ambiguity inherent in natural language which generated poor results. The inefficient performance of the system prompted other researchers to come up with something new and efficient. In 1957, a scientist come up with the proposal of syntactic structure which introduced the idea of generative grammar [4], which gave the linguistic a better understanding of how they could help the machine translation.

After 1960 there were several developments in in both the prototypes of system and theoretical issues. The system mainly focused on the problems of how to represent meaning and developing computationally tractable solutions that the existing systems or theories of grammar were not able to identify before 1960. Examples are: Chomsky's 1965 transformational model of linguistic [4], case grammar of Fillmore [5], semantic networks of Quillian [6] and conceptual dependency theory of Schank, which explained syntactic anomalies, and provided semantic representations.

Apart from theoretical development, many prototype systems have been developed. According to Liddy [2] these include: Weizenbaum's ELIZA [7] which was built to replicate the conversation between a psychologist and a patient; Winograd's SHRDLU simulation [8] of a robot that manipulated blocks on a tabletop which showed that natural language understanding was indeed possible for the computer [8] and PARRY 's a theory of paranoia [9] in a system which used groups of keywords instead of single keywords and used synonyms if keywords were not found.

By the 1970's a significant amount of work has been done in the field of natural language processing as for example McKeown's discourse planner TEXT [10] and McDonald's response generator MUMMBLE [11] used rhetorical predicates to create declarative descriptions in short texts form (that is paragraphs) and TEXT's which generated comprehensible responses online. However, by the early 1980s, there was an increasing awareness of the limitations of solutions to NLP problems and a general push towards applications that worked with language in a broad, real-world context. Since then to the present times, NLP has rapidly grown. This growth could be accredited to the advent of technologies such as: Internet; fast computers with increased memory; increased availability of large amounts of electronic text [2].

1.2 Application of NLP

Another major contributors in field of NLP, Church And Rau [12] states that, in recent years, the text interpretation and processing technologies have become more sophisticated and efficient. It is now possible to build very-targeted systems for specific purposes, for example, finding index terms in open text, and the ability to judge what level of syntax analysis is appropriate. These NLP technologies are helping us to create user-friendly decision-support systems for everyday non-expert users, especially in the field of language translation, knowledge acquisition and so on [12].

The rapid increase in the NLP technologies happened because some of the following reasons; the internet have a major role in this as it has provided researchers easy access of documents in electronic from at any given time period; the study pattern or the academic routine has replaced a new focus upon empirical approaches to language processing that rely more heavily upon corpus statistics than linguist theory and Modern networked machines are capable of processing millions of documents and performing the billions of calculations to build statistic profiles of large corpora[13].

Online information services are reaching mainstream computer users. With media attention reach time, hardly a day goes by without a new article on the national information infrastructure, digital libraries, networked services, digital convergence or intelligent agents. Massive quantities of text are becoming available in electronic form, ranging from published documents such as electronic dictionaries, encyclopaedias, libraries and archives for information retrieval services, private databases, personal email and so on [14].

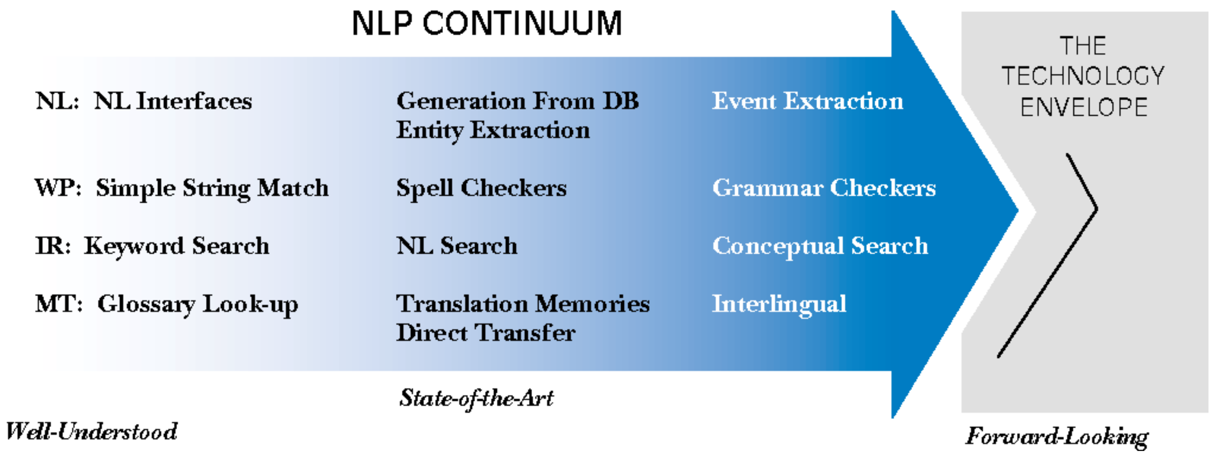


Fig. A

Figure A [12] shows several technologies ranging from well-understood technologies, such as string matching, to more forward-looking technologies such as grammar checkers, conceptual search, event extraction, interlingual and so on.

Some of the application of the NLP are listed below:

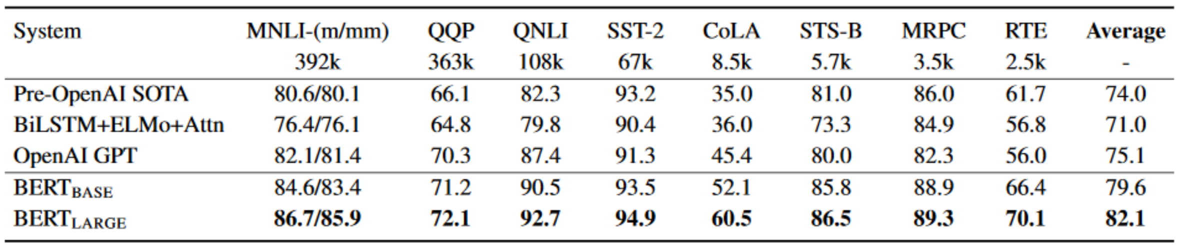
- Classify text into categories
- Automatic translation
- Speech understanding: Understand phone conversations
- Information extraction: Extract useful information from resumes
- Automatic summarization: Condense 1 book into 1 page
- Question answering
- Knowledge acquisition
- Text generations / dialogues

1.3 Present and Future of NLP

NLP technologies have come a long way since its real-world implementation back in 1930s. It still a futuristic technology which keeps getting better and better day by day, but in reality, it has entered into mainstream. Examples for NLP that are in use on high level are IBM's Watson for Cyber security, MIT's Laboratory for Social Machines which he analysis of social systems for positive change [15]. These are quite a few examples which shows NLP are being used in mainstream reality.

A multinational corporation in Europe is already chairing NLP (in combination with RPA) to digitize its sourcing ways for its long term spend, the long list of small purchases that altogether may account for only a few portion points of all the spending. It uses using Natural Language Processing to analyse free-form text and match the available order requirements to groups of suppliers, waiting the procurement robot to compare bids and make a purchase [16].

NLP technology will continue to gain momentum. One other example in is that, in china if you met a vehicle accident, soon, possibilities are you'll be able to pull out your smartphone device, take a photograph of the accident, and file an insurance claim with an Artificial Intelligence body. Person with injuries or disabilities that cannot write or having hard time in writing will benefit and use machine translation based on NLP as we can use other deep learning techniques for interpretation. Natural Language Processing has come to transform business and its impact will only going to increase day by day.



System	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	Average
	392k	363k	108k	67k	8.5k	5.7k	3.5k	2.5k	-
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
BERT$_{BASE}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
BERT$_{LARGE}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

Fig. B

Back in 2018, Google came up with an algorithm called as Bidirectional Encoder Representations from Transformers or BERT. It is a pre trained model based on bidirectional encoding of the words. Google claims that it already better than the other language models, even better than other bidirectional models such as ElMo and ULMFit. In some cases, it is also found to be more efficient than the humans itself. As we can see from the above results published google about the accuracy of their model [17].

Another example is the Media Lab's Cortico provides newsrooms, advocacy and non-profit organizations, and community influencers tools and programs to connect with their audiences on greater common ground. All these above examples point to how the advancements have made in the field of NLP has changed and is changing the concept of using NLP for our benefit. From Chomsky's 1965 Transformational Model of Linguistic [4] to Google's 2018 BERT [17], NLP technology have developed very rapidly.

1.4 Challenges

Even though we have come a long way in this field, a system that can almost have an efficiency like human is still a far cry. The sheer computing power generally required for NLP is itself a limitation of NLP. Artificial neural networks are far from matching the efficiency of the brain when it comes to process terabytes of data. As a consequence, deep learning-based NLP tools are reduced to analyse samples of the Big Text Data, which, in the case of email surveillance for example, is just not enough. Another one of major challenges the NLP faces is to train a system that can extract the context from conversation or discussion accurately.

2 Background Research

2.1 LSTM

Invented by Schmidhuber in 1997 [18], Long-short term memory network, generally known as LSTM too is one of most used neural network structures in deep learning field. It is widely used in many areas, mostly in machine learning application field, including speech recognition, natural language processing and other pattern recognition applications. It also avoids the famous vanishing gradient issue by adding three gated units: forget gate, input and output gates, through which the memory of past states can be efficiently controlled. Earlier methods for attacking several learning problems have either been tailored toward a specific problem or did not scale to long time dependences. LSTMs on the other hand are both general and effective at capturing long-term temporal dependences. They do not suffer from the optimization hurdles that plague simple recurrent networks (SRNs) [19] and have been used to advance the state of the art for many difficult problems.

The central idea behind the LSTM architecture is a memory cell, which can maintain its state over time, and nonlinear gating units, which regulate the information flow into and out of the cell. Most modern studies incorporate many improvements that have been made to the LSTM architecture since its original formulation [18]. However, LSTMs are now applied to many learning problems, which differ significantly in scale and nature from the problems that these improvements were initially tested on. Below is the fig that describes the basic architecture of a LSTM [20]:

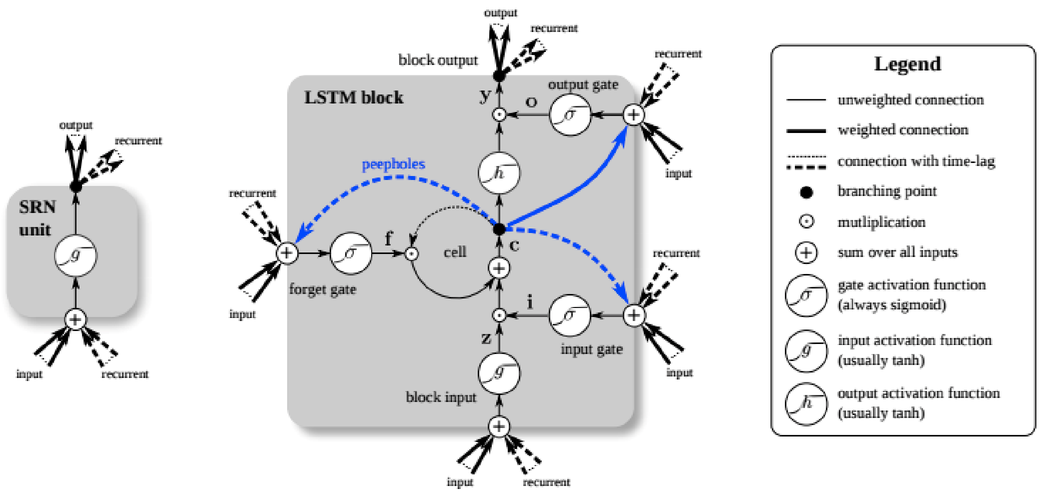


Figure 1. Detailed schematic of the Simple Recurrent Network (SRN) unit (left) and a Long Short-Term Memory block (right) as used in the hidden layers of a recurrent neural network.

Figure 1

The most commonly used LSTM setup was originally proposed by Graves and Schmidhuber [21]. Also known as vanilla LSTM. The vanilla LSTM incorporates changes by Gers et al. [22] and Gers and Schmidhuber [23] into the original LSTM [18] and uses full gradient training.

2.1.1 Forward Pass

Let $\mathbf{x}^t$ be the input vector at time t, N be the number of LSTM blocks, and M the number of inputs. Then, we get the following weights for an LSTM layer [24].

1. *Input Weights:* $\mathbf{W}_z, \mathbf{W}_s, \mathbf{W}_f, \mathbf{W}_o \in R^{N\text{x}M}$.
2. *Recurrent Weights:* $\mathbf{R}_z, \mathbf{R}_s, \mathbf{R}_f, \mathbf{R}_o \in R^{N\text{x}N}$.
3. *Peephole Weights:* $\mathbf{p}_s, \mathbf{p}_f, \mathbf{p}_o \in R^{N}$.
4. *Bias Weights:* $\mathbf{b}_z, \mathbf{b}_s, \mathbf{b}_f, \mathbf{b}_o \in R^{N}$.

Then the vector formulas for a vanilla LSTM layer forward pass can be written as [24]:

$$\begin{aligned}
\delta\mathbf{y}^t &= \Delta^t + \mathbf{R}_z^T \delta\mathbf{z}^{t+1} + \mathbf{R}_i^T \delta\mathbf{i}^{t+1} + \mathbf{R}_f^T \delta\mathbf{f}^{t+1} + \mathbf{R}_o^T \delta\mathbf{o}^{t+1} \\
\delta\mathbf{o}^t &= \delta\mathbf{y}^t \odot h(\mathbf{c}^t) \odot \sigma'(\bar{\mathbf{o}}^t) \\
\delta\mathbf{c}^t &= \delta\mathbf{y}^t \odot \mathbf{o}^t \odot h'(\mathbf{c}^t) + \mathbf{p}_o \odot \delta\mathbf{o}^t + \mathbf{p}_i \odot \delta\mathbf{i}^{t+1} \\
&\quad + \mathbf{p}_f \odot \delta\mathbf{f}^{t+1} + \delta\mathbf{c}^{t+1} \odot \mathbf{f}^{t+1} \\
\delta\mathbf{f}^t &= \delta\mathbf{c}^t \odot \mathbf{c}^{t-1} \odot \sigma'(\bar{\mathbf{f}}^t) \\
\delta\mathbf{i}^t &= \delta\mathbf{c}^t \odot \mathbf{z}^t \odot \sigma'(\bar{\mathbf{i}}^t) \\
\delta\mathbf{z}^t &= \delta\mathbf{c}^t \odot \mathbf{i}^t \odot g'(\bar{\mathbf{z}}^t).
\end{aligned}$$

where σ, g, and h are pointwise nonlinear activation functions. The logistic sigmoid (σ(x)=(1/1+e−x)) is used as the gate activation function and the hyperbolic tangent (g(x)=h(x)=tanh(x)) is usually used as the block input and output activation function. Pointwise multiplication of two vectors is denoted by $\odot$.

2.1.2 Backpropagation

The deltas inside the LSTM block are then calculated as [24]:

$$\begin{aligned}
\delta\mathbf{y}^t &= \Delta^t + \mathbf{R}_z^T \delta\mathbf{z}^{t+1} + \mathbf{R}_i^T \delta\mathbf{i}^{t+1} + \mathbf{R}_f^T \delta\mathbf{f}^{t+1} + \mathbf{R}_o^T \delta\mathbf{o}^{t+1} \\
\delta\mathbf{o}^t &= \delta\mathbf{y}^t \odot h(\mathbf{c}^t) \odot \sigma'(\bar{\mathbf{o}}^t) \\
\delta\mathbf{c}^t &= \delta\mathbf{y}^t \odot \mathbf{o}^t \odot h'(\mathbf{c}^t) + \mathbf{p}_o \odot \delta\mathbf{o}^t + \mathbf{p}_i \odot \delta\mathbf{i}^{t+1} \\
&\quad + \mathbf{p}_f \odot \delta\mathbf{f}^{t+1} + \delta\mathbf{c}^{t+1} \odot \mathbf{f}^{t+1} \\
\delta\mathbf{f}^t &= \delta\mathbf{c}^t \odot \mathbf{c}^{t-1} \odot \sigma'(\bar{\mathbf{f}}^t) \\
\delta\mathbf{i}^t &= \delta\mathbf{c}^t \odot \mathbf{z}^t \odot \sigma'(\bar{\mathbf{i}}^t) \\
\delta\mathbf{z}^t &= \delta\mathbf{c}^t \odot \mathbf{i}^t \odot g'(\bar{\mathbf{z}}^t).
\end{aligned}$$

Here, Δt is the vector of the deltas passed down from the layer above. If E is the loss function, it formally corresponds to (∂E/∂yt), but not including the recurrent dependences. The deltas for the inputs are only needed if there is a layer below that needs training, and can be computed as follows [24]:

$$\delta\mathbf{x}^t = \mathbf{W}_z^T \delta\mathbf{z}^t + \mathbf{W}_i^T \delta\mathbf{i}^t + \mathbf{W}_f^T \delta\mathbf{f}^t + \mathbf{W}_o^T \delta\mathbf{o}^t.$$

Apart from the conventional vanilla LSTM, there are 4 major architecture of LSTM [25].

- Conventional LSTM
- Deep LSTM
- LSTMP - LSTM with Recurrent Projection Layer
- Deep LSTMP

Conventional LSTM contains special units called memory blocks in the recurrent hidden layer. The memory blocks contain memory cells with self-connections storing the temporal state of the network in addition to special multiplicative units called gates to control the flow of information. Each memory block in the original architecture contained an input gate and an output gate. The input gate controls the flow of input activations into the memory cell [25].

Deep LSTM are built by stacking multiple LSTM layers. Note that LSTM RNNs are already deep architectures in the sense that they can be considered as a feed-forward neural network unrolled in time where each layer shares the same model parameters. The deep LSTM are mostly used for speech recognition [26]. It has been argued that deep layers in RNNs allow the network to learn at different timescales over the input [27]. Deep LSTM RNNs offer another benefit over standard LSTM RNNs: They can make better use of parameters by distributing them over the space through multiple layers [25].

LSTMP - LSTM with Recurrent Projection Layer was proposed as an alternative to standard LSTM architecture. The standard LSTM RNN architecture has an input layer, a re-current LSTM layer and an output layer. The input layer is connected to the LSTM layer. The recurrent connections in the LSTM layer are directly from the cell output units to the cell input units, input gates, output gates and forget gate. It was proposed mainly to address the computational complexity of learning LSTM. This architecture has a separate linear projection layer after the LSTM layer. The recurrent connections now connect from this recurrent projection layer to the input of the LSTM layer. The network output units are connected to this recurrent layer [25].

Deep LSTMP are very much like the deep LSTM having architecture in which multiple LSTM layers each with a separate recurrent projection layer are stacked. LSTMP allows the memory of the model to be in-creased independently from the output layer and recurrent connections. As we know that DNNs generalize better to unseen examples with increasing depth. The depth makes the models harder to overfit to the training data since the inputs to the network need to go through many non-linear functions [25].

2.2 Transformers

The *Vaswani et al* [28] introduced a new architecture called transformer. It is based on the mechanism called attention. According to Vaswani et al attention can be defined as "a function that can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is calculated as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key". In simpler word, just like how we start reading something, we don't keep whole sentences in our mind instead we hold the important keywords of any text just to deduce its context. Similarly, encoder instead of writing down transformation of a sentence or a word in some X language which is known by both encoder and decoder for a translation task of 2 different languages, he Encoder notes down the important keywords that are important to the structure or semantics of the sentence too, and gives them to the Decoder in addition to the regular translation. These new keywords make the work of decoder much easier as these keywords provides important information about the context of the sentence.

Just like the LSTMs, the transformer is also a system which transform one sequence to another sequence with the help of encoder and decoder but if differs by aspect that it does not imply any Recurrent Networks. The RNN (Recurrent Neural Networks) were one of the best ways to gasp the time dependencies in the provided or given sequences. Then, a team in 2018 came up with a paper proved that a design with just attention-mechanism and on top that even without any RNN (Recurrent Neural Networks) can improve the outcome of translation task and other tasks. This new design which the google team came up with is known as the BERT.

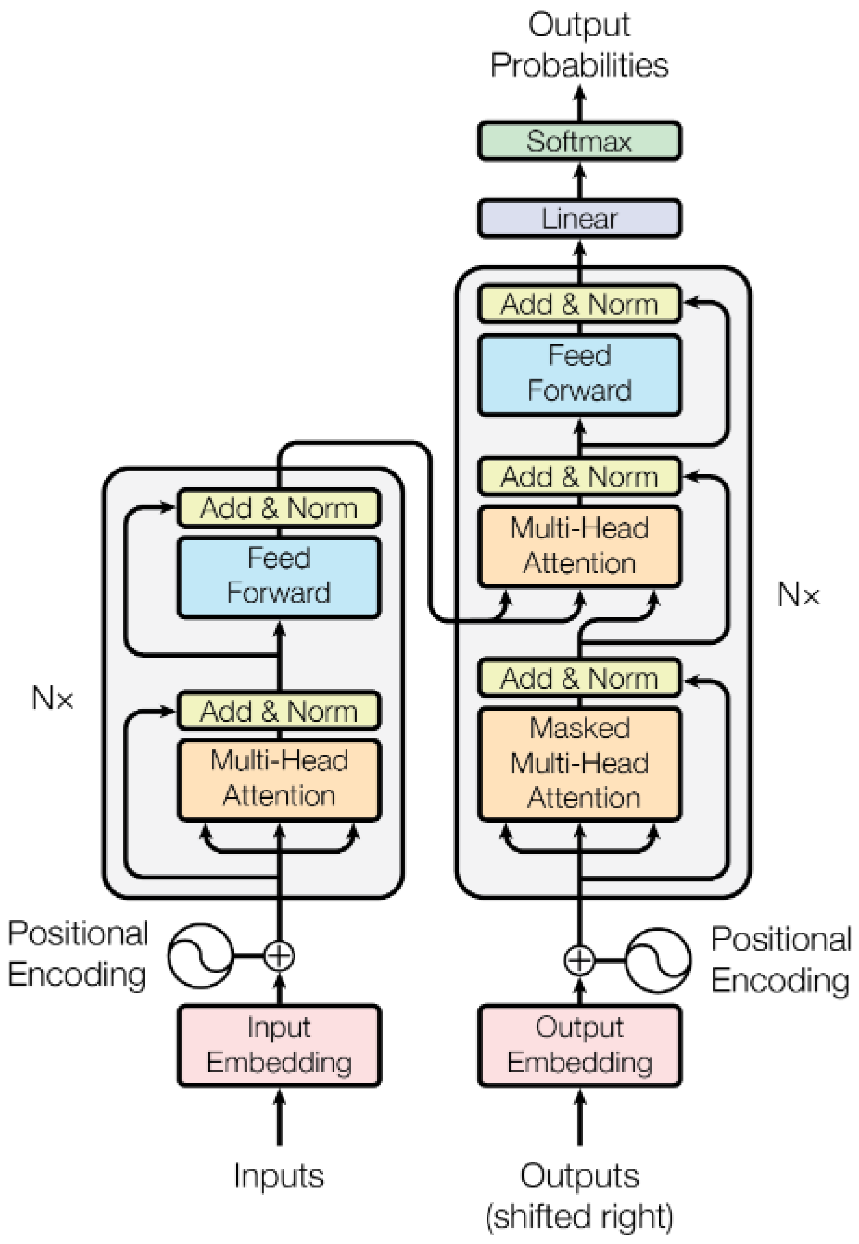


Figure 2: From 'Attention Is All You Need' by Vaswani et al [28].

As we can see from the above figure, it is consisting of both encoder and the decoder. They both are composed of modules that can be layered upon each other several times, which is given or defined by *Nx.*

As we know we don't have any recurrent networks that can keeps track that how sequences are supplied to a model, we need to give every word/part in a given sequence a corresponding location, as the sequence totally relies on the sequence of its elements. This where the multi headed attention comes in to play which use to provide the positional encoding of different words.

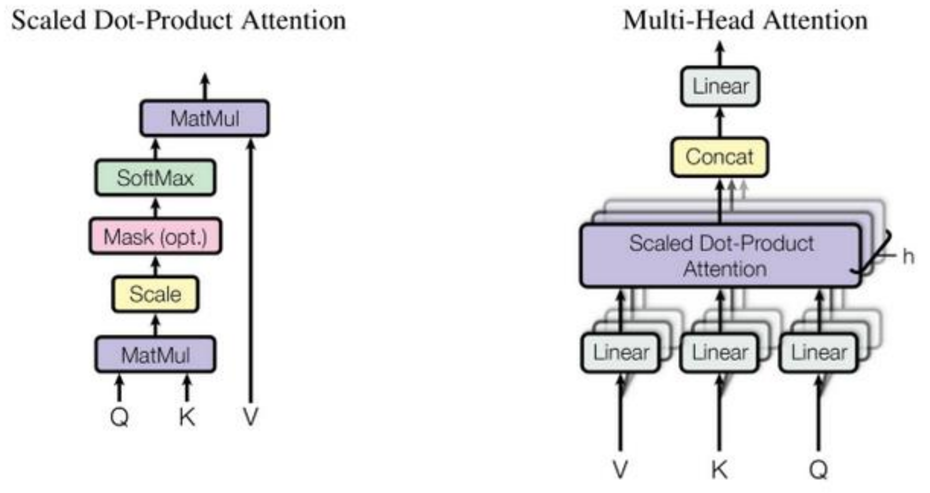


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Figure 3: From 'Attention Is All You Need' by Vaswani et al [28].

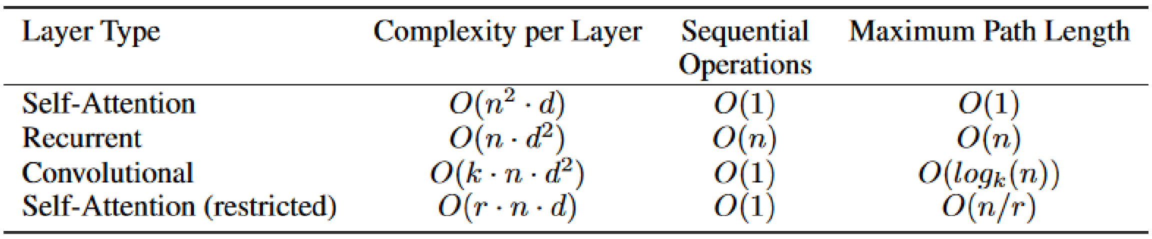
According to Vaswani et al the attention mechanism can be described by the below equation [28]**:**

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

Where the denominations of all the parts of the above equation is given by Vaswani et al [28] "Q is the matrix containing the vector representation of one word in the sequence or we can say queries, K are the keys which is again a vector representation of all words and the last V the values, same vector representation of all corresponding words. For the encoder and decoder, multi-head attention modules, V consists of the same word sequence than Q. However, for the attention module that is taking into account the encoder and the decoder sequences, V is different from the sequence represented by Q".

Coming to the Multi head attention pic in figure 3, it describes how attention-mechanism can be parallelized into multiple mechanisms that can be used side by side. The attention mechanism is repeated multiple times with linear projections of Q, K and V. This allows the system to learn from different representations of Q, K and V, which is beneficial to the model. These linear representations are done by multiplying Q, K and V by weight matrices W that are learned during the training [29].

These matrices Q, K and V are very much unlike for each position of the attention part in the structure depending if they are in the decoder, encoder or in the mid of both; the encoder and the decoder. Reason for this is we need to analyse context of just the part of decoder series or the whole input series. For example, if we have a task of language translation, from language X to Y. The input for encoder will be encoded X and input for decoder will be Y. However, we shift the input of decoder to the right by one position. This done to prevent model to just copy the output of decoder instead we want out model to learn from it. If the decoder sequence is not shifted, the model simply learns 'copy' the decoder input, since the target word for position *i* would be the word *i* in the decoder input [29]. Therefore, by shifting of the input of decoder by one position, the model needs to predict the target word for position *i* having only seen the word/characters *1, …, i-1* in the decoder sequence. Also, the complexity of this type of attention-based structures is much lower than the other architectures. Below is the table pointing out the complexities of the other architectures:



Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Figure 4: From 'Attention Is All You Need' by Vaswani et al [28]

To improve computational performance for tasks involving very long sequences, self-attention could be restricted to considering only a neighbourhood of size *r* in the input sequence cantered around the respective output position. This would increase the maximum path length to *O(n/r)* [28].

2.3 BERT (Bidirectional Encoder Representations from Transformers)

The term BERT stands for Bidirectional Encoder Representations from Transformers is new language representational model which just introduced back October 2018 by google brain team [17]. It was created to pre train deep bidirectional representations from different way or we can say unlabelled text. As it is claimed to be already a pre trained model, we only need to fine tune just one additional output layer for different variety of tasks.

The BERT (Bidirectional Encoder Representations from Transformers) is based on fine tuning approach which one of the two approaches used for pre trained models. The two approaches are mainly *fine tuning* and *feature based.* The models like ELMo [30] which is a feature-based technique model uses particular format that consists of pre-trained representations as additional features. On the other hand, the fine-tuning based technique like OpenAI GPT, uses small number of task related parameters and while training on a specific task only the fine tuning of those parameters is required only.

The implementation of BERT is based on two steps: *pre-training* and *fine-tuning* [17]. As the name suggest, during the pre-training phase the model is trained more like an unsupervised technique or we can say trained on unlabelled data. After that we need to initialize the model for fine-tuning. At first it is done with pre-trained parameters and then all of the parameters used is refined with the help of a supervised learning technique or in other words with the help of labelled data tasks. Every of these tasks have their own separate fine-tuned model, even if they are trained using the same initialized parameters for the pre training as it shown in the figure below.

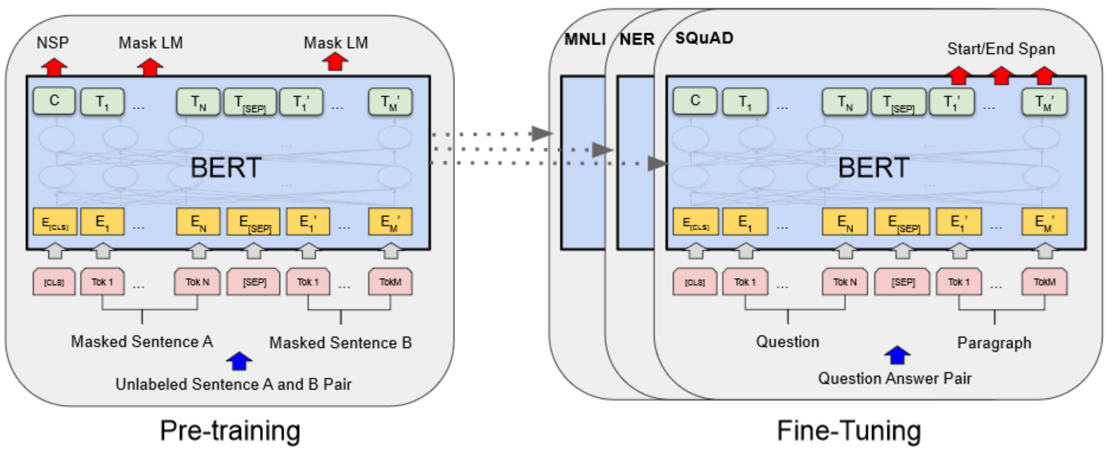


Figure 5: Pre-training and Fine-tuning of BERT [17]

As we can see from above pic the same pre-parameters are used to initialize every task for fine tuning of the model. As it is said in the official pdf paper release by google that "Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token (e.g. separating questions/answers)" [17].

BERT also have unified architectural design throughout different tasks. The pre-trained architecture and the final architecture have very minor differences.

Model Architecture

The architecture of BERT is made up of multi layered bidirectional transformer encoder which is similar to original design given by Vaswani et al [28]. The transformer blocks or the layers are denoted by the L, hidden layer size is denoted by H and the self-attention layer as A [17]. Primarily, there are two models that were used for benchmarking the BERT model; **BERT**$_{\text{BASE}}$ and **BERT**$_{\text{LARGE}}$. The models are differed by the size. **BERT**$_{\text{BASE}}$ architecture is consist of 110million parameters where L=12, H=768 and A=12, whereas **BERT**$_{\text{LARGE}}$ consist of 340million parameters where L=24, H=1024 and A=16.

Input/output Representations

The BERT can unambiguously characterise a pair of sentences and a single sentence. This is done to make BERT handle variety of tasks. It uses WordPiece embeddings [17] which consists of 30,000 token vocab. The sentences are changed into token format where each sequence start with special classification token [CLS]. Although the sentence pair are packed combined to single sequence. To identify different sentences the BERT uses two ways. A special token [SEP] just like [CLS] is added in between the packed tokens. Then a learned embedding is added to identify from where is belongs to; either sentence A or B. The input representation of a token is given by adding the corresponding segment, position embeddings and token [17]. This can be visualized from the figure given below:

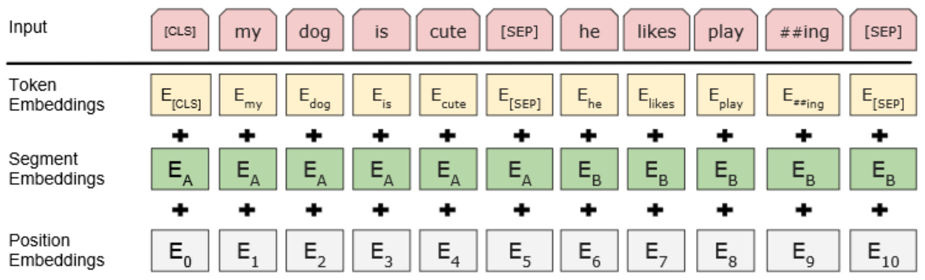

Input
[CLS] my dog is cute [SEP] he likes play ##ing [SEP]
Token Embeddings
$E_{[CLS]}$ E_{my} E_{dog} E_{is} E_{cute} $E_{[SEP]}$ E_{he} E_{likes} E_{play} $E_{\#\#ing}$ $E_{[SEP]}$
Segment Embeddings
E_A E_A E_A E_A E_A E_A E_B E_B E_B E_B E_B
Position Embeddings
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 E_{10}

Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

Figure 6:Input Representation of BERT [17]

2.3.1 Pre-training

BERT uses two unsupervised tasks for pre-training. Unlike other models it doesn't use left-to-right or right-to-left language models to pre train itself.

Task #1: Masked LM

Generally, a deep bidirectional model is believed to be more efficient than either a traditional left-to-right model or the combination of a left-to-right and a right-to-left model. The standard language models can only be trained left-to-right or right-to-left as bidirectional properties will allow each word to indirectly identify itself only [17]. However, to over come this BERT uses the masking technique. While training some percentage of the word input token are masked at random and then it predicts those masked word tokens. It is also known as Masked LM (MLM). In this process, the hidden vectors of respective word tokens are input into an output softmax on the vocabulary. In the BERT experiments performed by the team, they masked 15% of the tokens in each sequence at random [17]. This helps us to get a pre-trained bidirectional model, but it also creates a divergence between pre-training and fine-tuning as mask token [MASK] doesn't occur in fine-tuning. To remove this problem, the BERT not always replace masked words with the actual mask token [17]. For a case being say, if a x token is chosen then for the 80% of the time it is replaced by [MASK] token and 10% of the time is replaced by random token and the last 10% are left unchanged for the time. Then the token would be predicted using cross entropy loss [17].

Task #2: Next Sentence Prediction

Understanding the relationship between two given sentences is the most important for the given model. Many general tasks such as Question Answering and Natural Language Inference are based on this understanding of this relationship. To make a model understand this sentence relationship, it is pre-trained for a binarized NSP task which is obtained from any monolingual data corpus. Given two sentence X and Y for a example, while pre-training, half of the time Y is the actual sentence that comes after sentence X and rest of the half is just some random text from the dataset corpus. As we can from the figure below that C is used for prediction of the next sentence.

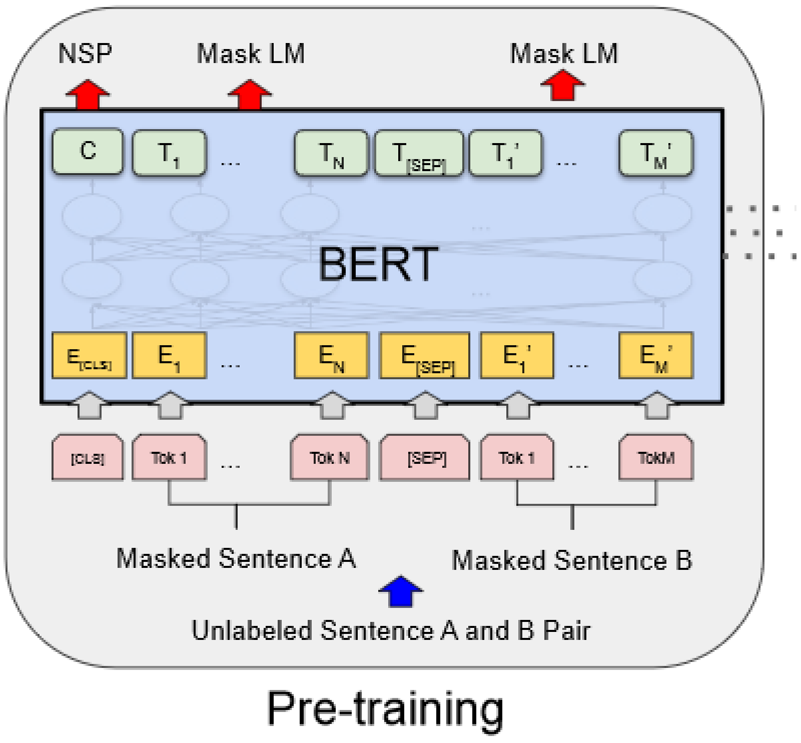


Figure 7: Next Sentence Prediction [17]

BERT transfers all the parameters that are used to initialise the task model parameters while the previous models only sentence embeddings were used to be transferred. For pre-training data, the BERT used the English Wikipedia corpus with 2,500 million words. Only the text passages were used in the pre-training the model and ignoring the lists, headers and labels.

2.3.2 Fine-Tuning

After the pre-training done, to make the model adapting to the work intended to be, we just need to fine tune the last layer of the model. In BERT the fine-tuning is the self-attention technique as it uses the transformers instead of the traditional LSTMs [17]. For the task having sentence pairs, we encode the sentence pairs before even applying the bidirectional cross attention. BERT instead of that uses the self-attention technique to merge the two. As specified in official BERT papers [17] "encoding of a combined sentence pair with self-attention technique involves bidirectional cross attention between the sentence pairs". As in BERT for any given task the input and output the task specific data and just make the model adaptive by fine-tuning the parameters.

The input data say the sentence X and Y are analogous other pairs of the sentences, question-pair passage in QA and a corrupt pair of sentences in the classification of the text. On the other hand, at the output part the token is sent into a separate layer for this token type tasks and then the classification token [CLS] is also sent to the output layer for the classification purpose. Fine-tuning is much less inexpensive than the pre-training as in pre-training whole parameters need to be initialised and trained while in fine tuning only the task specific parameters needs to by refined.

2.4 ELMo

In 2018 *Peters et al*.(2018a) [30] developed a new deep contextualized word representation model named as ELMo (Embeddings from Language Model) that works on both the syntax and semantics of words and its variation among different linguistic cases. It uses the word vector which are function of state of bidirectional model. This whole design can be added to any other working model as it works like a framework which further led to improvement in efficiency of the particular model. The word representation used by the ELMo is somewhat different from traditional word representation as each word in a given sentence have different vectors even if they are same words. ELMo takes the whole sentence into consideration while calculating the vectors of the words in the sentence. This helps the model to deal with the problem of Polysemy where a word could have multiple meaning [30].

The word representation used by ELMo is more like a function of internal layers of bidirectional language model. This combination of internals states where the states are stacked above each other for every task result in very detailed word representation. *Peters et al* claim that just adding the ELMo to any model can improve the efficiency of model to a significant level and reduce the error by 20% approx.

2.4.1 Bidirectional Language Model

The bidirectional models mainly consist of two LMs. One works as feed forward method and the other is feed backward method. The forward LM runs on any given sequence in forward direction while the backward LM similar to forward LM run in reverse on the given sequence. Suppose we have sequence of *N* given word tokens, forward LM calculates the probability the series by modelling probability of token t_k for the series ($t_1,\ldots,t_{k-1}$) [30]:

$$p(t_1, t_2, \ldots, t_N) = \prod_{k=1}^{N} p(t_k | t_1, t_2, \ldots, t_{k-1})$$

The language model generally calculates a token free of the given context and then passes it through the layers of forward LMs. The output of the LMs for the each given point generates a representation dependent on the context. Then the output of top layer of LSTM is used to predict the next token. Similarly, the a backward LM run on the given series in backward direction which predicts the backward token. This is where the actual bidirectional LM comes in to play. The model then combines the forward and backward LMs. Taking log of it and maximizing it with the softmax [30].

$$\sum_{k=1}^{N} \left(\log p(t_k \mid t_1, \ldots, t_{k-1}; \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s) \right. \\ \left. + \log p(t_k \mid t_{k+1}, \ldots, t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \right)$$

Then the parameters for are added to the word representation and the softmax layer in both the forward and backward side but the parameters for the LSTMs are kept separate. Although it is similar to previous approach of *Peters et al. (2017)* but instead of using independent parameters it shares weights within LM forward and backward direction.

2.4.2 ELMo Design

Similar to any bidirectional LMs, the ELMo also consist of two-layer bidirectional LM. These layers are stacked on top of each other together and each layer have 2 passes; the forward and backward pass [30].

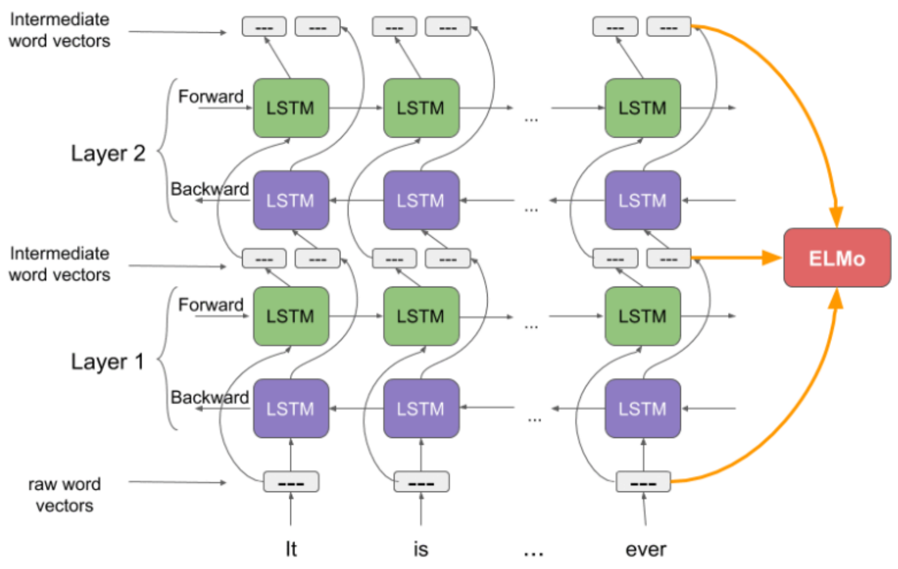


Figure 8: Picture form Analytics Vidhya [31]

ELMo uses CNN to represent the word tokens of a sentence into word vectors which are fed to the first layer of bidirectional LM. The forward pass holds the information about a word and context before the actual token of the word that is fed into it. Similarly, the backward pass holds the information about word and context after the word token. Then the combined information from backward and forward generates the mid word vectors which is then passed into next layer of the bidirectional LM. Then we get the final representation which is sum of weights of actual word vectors and mid word vectors. In ELMo, for each word token t_k, the bidirectional L layer calculates *2L+1* representation which is given by the equation below

$$\begin{aligned} R_k &= \{\mathbf{x}_k^{LM}, \overrightarrow{\mathbf{h}}_{k,j}^{LM}, \overleftarrow{\mathbf{h}}_{k,j}^{LM} \mid j = 1, \ldots, L\} \\ &= \{\mathbf{h}_{k,j}^{LM} \mid j = 0, \ldots, L\}, \end{aligned}$$

So, basically ELMo generates the word vectors are different from each other as every vector generated is generated by taking the whole sentence into the account. Therefore, the same word is benefit from having different word vector in different context. This helps the machine to differentiate between which word is being said in what context, in simpler words we can say that it helps a given system to understand and distinguish the context of a sentence in more effective way.

3 Implementation

3.1 Objective

The main aim of this experiment is to compare two deep bidirectional language model. This experiment is mainly based on the accuracy of the model i.e. which model achieves high accuracy.

3.2 Requirements

3.2.1 Research

The main goal is to compare the functioning of two different algorithms as they both are somewhat similar to each other but still different from each other. Both algorithms are based on bidirectional language models. There were different possibilities to compare them as I went through the possibilities. I first started with BERT standalone and then found out ELMo would be most suitable for the comparison as it is the algorithm that BERT even claim to superior of. So, I decided to compare the performance of accuracy of the algorithms on the same data set. Comparing the two algorithms for a same given dataset would be perfect to analyse as the basic condition would be same for the both algorithms. I chose IMDB Large Movie Review Dataset data set which is suitable and more balanced for the two language models.

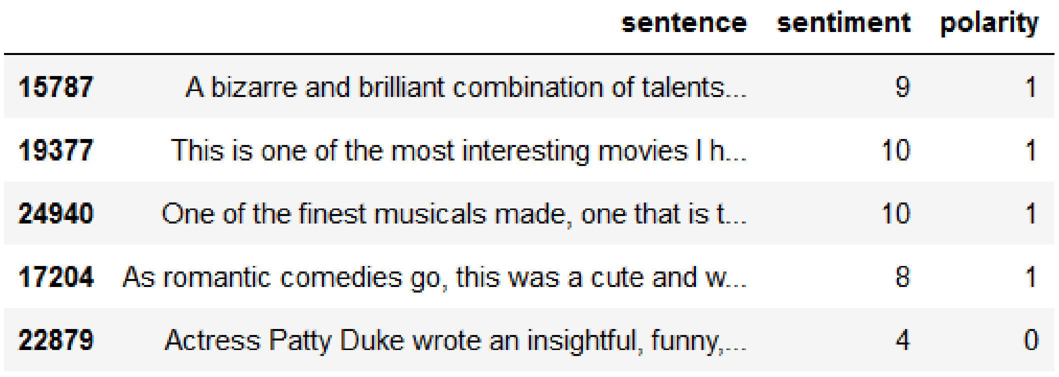
3.2.2 Information Gathering

After getting the research done next step was to get the details about the required modules. First thing I did was to read about what exactly is the Natural Language Processing as I had no idea about the NLP. I stared with NLP and its origin and its current state. My supervisor suggested me to read about the BERT first. As he directed me how the word vectors work in the field of Natural Language Processing which helped me lot as who have no experience with any sort of algorithm related to the NLP. Next thing I did was I read the official paper of BERT [17]. I had difficulties in getting hold of BERT as it was totally a new concept even in the field of NLP itself. So, next thing my supervisor did is directed me read about ELMo first then. While reading the ELMo papers [30] I was only then started to understand the word embeddings. Then I read about the working of LSTMs and how two LSTMs are combined to form a much more efficient bidirectional language model. After getting hold of the LSTMs, I moved to the transformers. Transformers are an amazing concept as it is employed by the BERT itself. Reading all this made me realise how far we have come in the field of NLP.

The language I chose for the implementation was Python. Python in one of the most flexible and easy language to learn. I had already some experience with python so it was much of an obvious choice for me. Python have vast range of libraries built for data science field and apart from that the combination of TensorFlow and Keras makes the python very useful and easy. I was able to find several code examples of NLP regarding through which I was able to code the required modules. Overall the flexibility of python makes it the most obvious choice over any given language.

3.2.3 Dataset Used

The dataset used here is the IMDB Large Movie Review Dataset [32] which is hosted by the Stanford University. The dataset consists of 25,0000 train samples and 25,000 test samples. The dataset is collection of movie reviews labelled positive or negative. Choosing this dataset was done after analysing different datasets. There were many datasets but most of them have one sided labelling done i.e. the no. of one label was much more than the other. The proportion was not up to the mark with other datasets. This dataset has very balanced labelling done in it and it was full of both long short sentences. This was one the ways to check how these two models perform this well balanced yet much versatile dataset. Below is the illustration how the dataset looks like:



	sentence	sentiment	polarity
15787	A bizarre and brilliant combination of talents...	9	1
19377	This is one of the most interesting movies I h...	10	1
24940	One of the finest musicals made, one that is t...	10	1
17204	As romantic comedies go, this was a cute and w...	8	1
22879	Actress Patty Duke wrote an insightful, funny,...	4	0

Figure 9: Screenshot taken after pre-processing the data in the code

As we can from the figure above the data frame is consist of three columns sentence, sentiment and polarity. The column 'polarity' is the one which I have used as the label for the both language model and the sentence as the given data for the two language models.

3.3 Steps

3.3.1 Loading Data to the Data frame

First, we need to download the dataset from the website "http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz" and paste the data set in your current working directory. One way to get your current working directory is using the "os" library. Below is the code for the os library:

```
import os
curr_dir = os.getcwd()
```

After done with the location of the data set, we will now read the data from the file and store it in a data frame having total of three columns.

```
def folder_data(folder_path):
    data = {}
    data["sentence"] = []
    data["sentiment"] = []
    for file_path in os.listdir(folder_path):
        with tf.gfile.GFile(os.path.join(folder_path, file_path), "r") as f:
            data["sentence"].append(f.read())
            data["sentiment"].append(re.match("\d+_(\d+)\.txt", file_path).group(1)
)
    return pd.DataFrame.from_dict(data)
```

As the downloaded dataset comes in distributed folder, we need to stack up all the positive and negative together and assign then a label 1 for the positive data and 0 for the negative data and then concatenate them together. You can see from the code below

```
def load_dataset(folder_path):

    pos_df = folder_data(os.path.join(folder_path, "pos"))
    neg_df = folder_data(os.path.join(folder_path, "neg"))
    pos_df["polarity"] = 1
    neg_df["polarity"] = 0
    return pd.concat([pos_df, neg_df]).sample(frac=1).reset_index(drop=True)
```

Once we have done with the code for the assembling of data, we need to create the train and test df from the dataset we have downloaded.

```
train_df = load_dataset(os.path.join(curr_dir, "Dataset", "aclImdb", "train"))
test_df = load_dataset(os.path.join(curr_dir,"Dataset", "aclImdb", "test"))
```

We need to provide the location of the dataset from where it is stored. One need to be careful while giving the path to the code as if fail to do so the dataset will not properly be loaded into the data frame.

3.3.2 Data Pre-Processing

3.3.2.1 Convert Data to BERT Format

Now we need to convert our data into a format that BERT understands. First, we create InputExample's using the constructor provided in the BERT library.

text_a is the text we want to classify, which in this case, is the Request field in our Data frame. **text_b** is used if we're training a model to understand the relationship between sentences (i.e. is text_b a translation of text_a? Is text_b an answer to the question asked by text_a?)

```
train_ex = train_df_sample.apply(lambda x: bert.run_classifier.InputExample(guid=No
ne, text_a = x['sentence'], text_b = None, label = x['polarity']), axis = 1)

test_ex = test_df_sample.apply(lambda x: bert.run_classifier.InputExample(guid=None
, text_a = x['sentence'], text_b = None, label = x['polarity']), axis = 1)
```

3.3.2.2 Load a Vocabulary file and Lowercasing Information from BERT

Next, we need to pre-process our data so that it matches the data BERT was trained on. For this, we'll need to do a couple of things (but don't worry--this is also included in the Python library):

- Lowercase our text (if we're using a BERT lowercase model)
- Tokenize it (i.e. "sally says hi" -> ["sally", "says", "hi"])
- Break words into WordPieces (i.e. "calling" -> ["call", "##ing"])
- Map our words to indexes using a vocab file that BERT provides

- Add special "CLS" and "SEP" tokens (see the readme)
- Append "index" and "segment" tokens to each input (see the BERT paper).

To start, we'll need to load a vocabulary file and lowercasing information directly from the BERT **tf hub** module:

```
def bert_tokenizer():
    with tf.Graph().as_default():
        URL = "https://tfhub.dev/google/bert_uncased_L-12_H-768_A-12/1"
        bert_module = hub.Module(URL)
        tokenization_info = bert_module(signature="tokenization_info", as_dict=True)
        with tf.Session() as ses:
            vocab, lower_case = ses.run([tokenization_info["vocab_file"],
                                         tokenization_info["do_lower_case"]])

    return bert.tokenization.FullTokenizer(vocab_file=vocab, do_lower_case=lower_case)
```

As we know the BERT model, we're using expects lowercase data (that's what stored in tokenization_info["do_lower_case"]) and we also loaded BERT's vocab file. We also created a tokenizer, which breaks words into word pieces.

3.3.2.3 Convert Features to BERT Format

Using our tokenizer, we'll call run_classifier.convert_examples_to_features on our InputExamples to convert them into features BERT understands.

```
train_ex = train_df_sample.apply(lambda x: bert.run_classifier.InputExample(guid=None, text_a = x['sentence'], text_b = None, label = x['polarity']), axis = 1)

test_ex = test_df_sample.apply(lambda x: bert.run_classifier.InputExample(guid=None, text_a = x['sentence'], text_b = None, label = x['polarity']), axis = 1)
```

3.3.3 Model Creation

3.3.3.1 BERT

Now that we've prepared our data, let's focus on building a model. create_model does just this below. First, it loads the BERT tf hub module again (this time to extract the computation graph). Next, it creates a single new layer that will be trained to adapt BERT to our sentiment task (i.e. classifying whether a movie review is positive or negative). This strategy of using a mostly trained model is called fine-tuning [33].

```
def create_model(is_predicting, input_ids, input_mask, segment_ids, labels, num_labels):
    bert_module = hub.Module(URL, trainable=True)
    bert_inputs = dict(input_ids = input_ids,
                       input_mask = input_mask,
                       segment_ids = segment_ids)
    bert_outputs = bert_module(inputs = bert_inputs,
                               signature = "tokens",
                               as_dict = True)
```

We will use "pooled_output" for classification tasks on an entire sentence.

```
    output_layer = bert_outputs["pooled_output"]
    hidden_size = output_layer.shape[-1].value
```

Now we will create our own layer to tune for politeness data [33].

```
output_weights = tf.get_variable("output_weights",
                                     [num_labels, hidden_size],
                                     initializer=tf.truncated_normal_initializer(st
ddev=0.02))
    output_bias = tf.get_variable("output_bias", [num_labels], initializer=tf.zeros
_initializer())
```

Now as we know there may be the chance where the model can overfit. To prevent this, we can use dropout.

```
with tf.variable_scope("loss"):
        output_layer = tf.nn.dropout(output_layer, keep_prob=0.9)
        logit = tf.matmul(output_layer, output_weights, transpose_b=True)
        logits = tf.nn.bias_add(logit, output_bias)
        log_probs = tf.nn.log_softmax(logits, axis=-1)
```

Then we will convert the variables into one-hot notation [33].

```
        one_hot_labels = tf.one_hot(labels, depth=num_labels, dtype=tf.float32)
        predicted_labels = tf.squeeze(tf.argmax(log_probs, axis=-
1, output_type=tf.int32))
```

Then if we are predicting, we want predicted labels and the probabilities.

```
       if is_predicting:
            return (predicted_labels, log_probs)
        per_example_loss = -tf.reduce_sum(one_hot_labels * log_probs, axis=-1)
        loss = tf.reduce_mean(per_example_loss)
        return (loss, predicted_labels, log_probs)
```

Next we'll wrap our model function in a **model_fn_builder** function that adapts our model to work for training, evaluation, and prediction. This function ‘model_fn_builder’ actually creates our model function. The working of the code is described in the code below [33]:

```
       def model_fn_builder(num_labels, learning_rate, num_train_steps,num_warmup_s
teps):
  def model_fn(features, labels, mode, params):
    """The `model_fn` for TPUEstimator."""

    input_ids = features["input_ids"]
    input_mask = features["input_mask"]
    segment_ids = features["segment_ids"]
    label_ids = features["label_ids"]

    is_predicting = (mode == tf.estimator.ModeKeys.PREDICT)

    # TRAIN and EVAL
    if not is_predicting:
      (loss, predicted_labels, log_probs) = create_model(is_predicting, input_ids,
input_mask,
      segment_ids, label_ids, num_labels)
```

```
   train_op = bert.optimization.create_optimizer(loss, learning_rate, num_train_steps,
   num_warmup_steps, use_tpu=False)

   # Calculate evaluation metrics.
   def metric_fn(label_ids, predicted_labels):
    accuracy = tf.metrics.accuracy(label_ids, predicted_labels)
    f1_score = tf.contrib.metrics.f1_score(label_ids,predicted_labels)
    auc = tf.metrics.auc(label_ids,predicted_labels)
    return {
      "eval_accuracy": accuracy,
      "f1_score": f1_score,
      "auc": auc,
      }
   eval_metrics = metric_fn(label_ids, predicted_labels)

   if mode == tf.estimator.ModeKeys.TRAIN:
    return tf.estimator.EstimatorSpec(mode=mode,
     loss=loss,
     train_op=train_op)
   else:
     return tf.estimator.EstimatorSpec(mode=mode,
      loss=loss,
      eval_metric_ops=eval_metrics)
  else:
   (predicted_labels, log_probs) = create_model(is_predicting, input_ids, input_mask, segment_ids, label_ids, num_labels)
   predictions = {
     'probabilities': log_probs,
     'labels': predicted_labels
   }
   return tf.estimator.EstimatorSpec(mode, predictions=predictions)
 # Return the actual model function in the closure
 return model_fn
```

Now we will define all the necessary parameters for the model function [33].

```
    # Compute train and warmup steps from batch size
BATCH_SIZE = 32
LEARNING_RATE = 2e-5
NUM_TRAIN_EPOCHS = 3.0
# Warmup is a period of time where hte learning rate
# is small and gradually increases--usually helps training.
WARMUP_PROPORTION = 0.1

# Compute # train and warmup steps from batch size
num_train_steps = int(len(train_features) / BATCH_SIZE * NUM_TRAIN_EPOCHS)
num_warmup_steps = int(num_train_steps * WARMUP_PROPORTION)

# Specify outpit directory and number of checkpoint steps to save
test_config = tf.estimator.RunConfig(
  model_dir=OUTPUT_DIR,
  save_summary_steps=SAVE_SUMMARY_STEPS,
  save_checkpoints_steps=SAVE_CHECKPOINTS_STEPS)


model_fn = model_fn_builder(
 num_labels=len([0, 1]),
 learning_rate=LEARNING_RATE,
 num_train_steps=num_train_steps,
 num_warmup_steps=num_warmup_steps)

estimator = tf.estimator.Estimator(
 model_fn=model_fn,
 config=test_config,
    params={"batch_size": BATCH_SIZE})
```

3.3.3.2 ELMo

To define how we going to use our elmo embedding we need to define a function which takes a sentence then convert it to the tensor flow string and squeeze the input and put it into the embed layer.

```
url = "https://tfhub.dev/google/elmo/2"
embed = hub.Module(url)
def ELMoEmbd(x):
    return embed(tf.squeeze(tf.cast(x, tf.string)), signature="default", as_dict=True)["default"]
```

Now we will build our actual model. First, we define an input layer with the shape of 1 so that it will take one sentence at a time. Now then we will feed that layer into lambda layer which uses our ELMoEmbd function and define the output size of 1024 so that model knows that we size of ELMo vectors are of 1024. Then we take that layer and feed it into Dense layer with relu activation and then feed that Dense layer to prediction layer with softmax activation. Now we can define our functional model as having input as input_text and output as pred (prediction layer). Then at last we just need to compile the model with 'sparse_categorical_crossentropy' loss as we are categorizing and will give the default adam optimizer and will be calculating the accuracy of the model by metrics function.

```
input_text = Input(shape=(1,), dtype=tf.string)
embedding = Lambda(ELMoEmbd, output_shape=(1024, ))(input_text)
dense = Dense(256, activation='relu')(embedding)
pred = Dense(2, activation='softmax')(dense)
model = Model(inputs=[input_text], outputs=pred)
```

3.3.4 Model Training &Testing

Now as we have done with the creation of model now its time to train and test how they fair on a given dataset.

3.3.4.1 BERT

First, we will create an input function for training that will set up our model function with parameters required for the training.

```
#Train function
train_input_fn = bert.run_classifier.input_fn_builder(features=train_features, seq_length=128, is_training=True, drop_remainder=False)
```

Now then we will start with training of the model.

```
#Start training
estimator.train(input_fn=train_input_fn, max_steps=num_train_steps)
```

After the training is done, similarly we will proceed with the testing of the model too. First creating a test function for setting up of parameters and then with the validation process.

```
#Test the model
test_input_fn = run_classifier.input_fn_builder(features=test_features,seq_length=1
28,is_training=False,drop_remainder=False)

#Start the Validation
estimator.evaluate(input_fn=test_input_fn, steps=None)
```

3.3.4.2 ELMo

Similar to the above section we first start with the training of the ELMo model. First, we need to initialise some weights and specify batch size for the model.

```
with tf.Session() as session:
    K.set_session(session)
    session.run(tf.global_variables_initializer())
    session.run(tf.tables_initializer())
    history = model.fit(x_train, y_train, epochs=1, batch_size=16)
    model.save_weights('./elmo-model.h5')
```

Now we will set up another tensor flow model for the prediction in which we again initialize our model and then load our trained weights from the previous section to this session.

```
with tf.Session() as session:
    K.set_session(session)
    session.run(tf.global_variables_initializer())
    session.run(tf.tables_initializer())
    model.load_weights('./elmo-model.h5')
    predicts = model.predict(x_test, batch_size=16)
```

Now we will proceed with the validation of the model using our test set

```
# Validation of Model
accu  = metrics.accuracy(y_test, y_preds)
```

3.4 How to Use my Project

To use my project one need to install tensflow-bert package and execute the code line python file proved in the sequential order. The link for the dataset to use is "http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"

3.5 Outcome

Accuracy of the two-given model after the validation is given below:

Model/Accuracy	**Percentage (%)**
BERT	86.21
ELMo	84.93

3.6 Conclusion

As it was expected the BERT was slightly more accurate than the ELMo. Although the as claimed by the official Google, the difference between the accuracy is not as good as it is. The results are more like very close to each other. On the time side both the model almost took 4 hours to run in my laptop in batch size of 16.

4 Self-Assessment

4.1 Strengths

Being familiar with the Python programming language was the biggest strength of this project. From the very start of the project my supervisor asked me how god I am with the python programming. Keeping that into the mind I was assigned this project as it was consist of both easy and difficult programming. There were data structures from where I have to take deep dive into it just to extract the required information for the model.

Next thing was the management as I started with the project, during first several weeks my supervise guided me through the very efficiently. Under his guidance I was able to cruise along slowly but smoothly on my project. I had already made a plan to follow which I follow almost thoroughly. Being able to be flexible help me out a lot.

Creating an account on google Colab platform was a life saver for me as even after having a powerful laptop the NLP tasks tends to use lot of resources. Use of google Colab platform help me save a lot of time as it provides a very descent and powerful cloud solution for the resource hungry projects. This help me to manage my time more efficiently than even.

4.2 Weakness

My biggest weakness was the almost zero experience in the NLP field. I had no knowledge of NLP technologies nor any experience apart from just being heard about the various NLP technologies. I told my supervisor that I am not familiar with any of the algorithms from the very start so he gave task of just compare two most efficient, new and powerful language models.

Apart from the inexperience resources fort the language model was very low as I have only the official research paper to study from. BERT was totally a new model which just released end of last year. There were very little resources to study from. The source code provided on the git repo was mainly Linux shell-based coding and I never used a Linux system before. Configuring my laptop to run Linux commands was totally a far cry for me.

Next major weakness is that I cannot keep myself concentrated enough for long time while studying. As this project required lots of studying and analysis, I was finding it difficult to keep up. I had to force my self to make sure that I give enough time to the project so that I can complete on the time.

4.3 Challenges

Major challenge was for me to get enough knowledge in the given span of time for something I never tried of ever before. Next major challenge came in the form of system hardware. Language modelling is a very resource hungry task. There was a time when my laptop used to run for 10 hours straight for just one session. Because of this I face so many resource management problems. I have even adjusted my code to make it run o my laptop in somewhat less time.

4.4 Opportunity

This project was my first project in the field of Deep Learning/Language Modelling. It gave me the opportunity to take deep dive into this field and explore it thoroughly. With my supervisor backing me up I slowly become more and more confident in this field. Even though I still have to learn a lot but it was totally an amazing and thrilling experience for me as I have many ups and down through this project.

The project also helps to gain the knowledge of Linux community as I learned how to relate between the Windows and Linux shell commands. Diving further into the world of deep learning I realised how vast this technology is and interesting too.

5 Professional Ethics

Having a title of data scientist brings a lot of responsibilities too. It us who have to make sure that the work we are doing for a given client have concrete result without any implications.

5.1 Ignoring the Basics

Ignoring the basic is one the major and most common type of problem we face in our daily professional life. People generally take things for granted during their work which led to massive loss in the future for any particular corporation. During my 2 year of work experience I have seen many people ignoring basic things that caused them lot of trouble afterwards. These mistakes are generally done because of over confidence. As it is truly said being confident is a good thing but being overconfident is not a good thing.

5.2 Proper Resource Management

One thing I learned from this project is to manage resources according to the project. Considering this project of mine as an example, I do have not done the proper resource management. Because of that I wasted a lot of time or I can say wasted several days. Only knowing the importance of resources, I analysed the situation and came up with the cloud solution Colab platform. As I said earlier my laptop was not good enough to handle this kind of powerful programs which in further resulted in the wastage of time for me. If only I had done a proper resource analysis, I would have saved a lot of time. So, analysis of anything is very important this, it must be done prior to any project. It show how professional we are although we are humans and mistake are bound to happen but still we can minimise the impact in the near future.

5.3 Improper Representation of Information and Data

Proper presentation of data and information is very important in a professional life. The whole schema of the project revolves around the data and information provided in this field. So, it's our responsibility to make sure person starting a project have proper information regarding that particular project. During my office working days we used to get projects with improper requirement document which didn't specified even the basic aspects of the project. This led to the delay in the project. Further it also led to a bad impression of a person who is responsible for this.

5.4 Proper Use of Technology

Deep learning is a very powerful technology field. It can used for good and bad also. As a data scientist its our responsibilities too make sure that the given tech is not used to harm someone in any possible way. One example of misuse of tech is that recently a android developer man an app that can make person pics nude. Although it was made just to use in funny way but people started it used in bad way and blackmailing others. Even though the application was pulled of it generated or sparked the idea of that app. After some time, there were several rip-off apps were made on the very save idea. So even the technology is helping us it can also harm us in many possible ways. To keep this minimal we need to maintain a professional ethics that tells to keep our self in limit.

6 References

[1] M. C. Surabhi, "Natural language processing future," in *IEEE*, 2013.

[2] E. D. Liddy, Natural Language Processing, New York: Syracuse University, 2001.

[3] J. Hutchins, "The history of machine translation in a nutshell," *The history of machine translation in a nutshell,* vol. 1, no. 1, p. 5, 2005.

[4] N. Chomsky, Syntactic Structures, Mouton & Co., 1957.

[5] C. J. Fillmore, The Case for Case, NewYork, 1968.

[6] R. Quillan, A notation for representing conceptual information: an application to semantics and mechanical English paraphrasing, Systems Development Corp., 1963.

[7] J. Weizenbaum, ELIZA-A Computer Program For the Study of Natural Language Communication Between Man And Machine, vol. 9, G. Salton, Ed., NewYork: ACM, 1966.

[8] T. Winograd, Procedures as a Representation for Data in a Computer Program for Understanding Natural Language, MIT-AI-TR-235, 1971.

[9] V. Cerf, 18 September 1972. [Online]. Available: https://tools.ietf.org/html/rfc439.

[10] K. R. McKeown, "Discourse Strategies for generating Natural Language Text," *Artificial Intelligence,* vol. 27, no. 1, 1985.

[11] R. Rubinoff, "Adapting mumble: experience with natural language generation," in *AAAI Press*, Philadelphia, 1986.

[12] K. W. Church and L. F. Rau, "Commercial applications of natural language processing," *Communications of the ACM,* vol. 38, no. 11, 1995.

[13] P. Jackson and I. Moulinier, Natural Language Processing for Online Applications, John Benjamins Publishing Company, 2007.

[14] R. Bose, "Natural Language Processing: Current state and future directions," Research Gate, 2004.

[15] P. Ghosh, *The Future of NLP in Data Science,* Dataversity, 2018.

[16] K. Jain and E. Woodcock, "McKinsey," 2018. [Online]. Available: https://www.mckinsey.com/business-functions/operations/our-insights/a-road-map-for-digitizing-source-to-pay.

[17] J. Devlin, K. Lee, M.-W. Chang and K. Toutanova, "arXiv," 11 10 2018. [Online]. Available: https://arxiv.org/abs/1810.04805.

[18] H. Sepp and J. Schmidhuber, "Long short-term memory," *Neural computation,* vol. 9, no. 8, pp. 1735-1781, 1997.

[19] S. a. B. Y. a. F. P. a. S. Hochreiter, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," in *A field guide to dynamical recurrent neural networks. IEEE Press*, 2001.

[20] C. Nicholson, "Skymind," 2019. [Online]. Available: https://skymind.ai/wiki/lstm.

[21] J. S. A. Graves, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural Netw,* vol. 18, no. 5, pp. 602-611, 2005.

[22] J. S. F. C. F. A. Gers, "Learning to forget: Continual prediction with LSTM," *Proc. 9th Int. Conf. Artif. Neural Netw. (ICANN),* vol. 2, pp. 850-856, 1999.

[23] J. S. F. A. Gers, "Recurrent nets that time and count," *Proc. IEEE-INNS-ENNS Int. Joint Conf. Neural Netw. (IJCNN),* vol. 3, pp. 189-195, 2000.

[24] R. K. Srivastava, J. Koutník, B. R. Steunebrink, J. Schmidhuber and K. Greff, "LSTM: A Search Space Odyssey," *IEEE Transactions on Neural Networks and Learning Systems,* vol. 28, no. 10, pp. 2222-2232, 2017.

[25] A. S. F. ,. B. Hasim Sak, "Long Short-Term Memory Recurrent Neural Network Architecturesfor Large Scale Acoustic Modeling," Google, 2015.

[26] M. W. B. S. a. A. G. . Eyben, "From speechto letters using a novel neural network architecture for graphemebased ASR," *Automatic Speech Recognition & Understanding,* pp. 376-380, 2009.

[27] M. H. a. B. Schrauwen, "Training and analysing deep re-current neural networks," *Advances in Neural Information Pro-cessing Systems,* pp. 190-198, 2013.

[28] N. S. N. P. J. U. L. J. A. N. G. L. K. I. P. Ashish Vaswani, "arXiv," 12 06 2017. [Online]. Available: https://arxiv.org/abs/1706.03762.

[29] M. Allard, "Medium," Medium, 4 1 2019. [Online]. Available: https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04.

[30] M. N. M. I. M. G. C. C. K. L. L. Z. Matthew E. Peters, "arXiv," 15 02 2018. [Online]. Available: https://arxiv.org/abs/1802.05365.

[31] P. Joshi, "Analytics Vidhya," 11 03 2019. [Online]. Available: https://www.analyticsvidhya.com/blog/2019/03/learn-to-use-elmo-to-extract-features-from-text/.

[32] A. L. a. D. R. E. a. P. P. T. a. H. D. a. N. A. Y. a. P. C. Maas, "Stanford University," June 2011. [Online]. Available: http://ai.stanford.edu/~amaas/data/sentiment/.

[33] M.-W. C. ,. K. L. K. T. Jacob Devlin, "Google colab," 10 2018. [Online]. Available: https://colab.research.google.com/github/google-research/bert/blob/master/predicting_movie_reviews_with_bert_on_tf_hub.ipynb.