University of Dhaka
Department of Computer Science and Engineering

Natural Language Processing
4th Year 2nd Semester
Session : 2019 -20

Topic: BERT

Submitted by:
Md Fahim (FH-26)
Md Sakib Khan (AE-45)

Submission Date: 09-02-2021

Submitted to:
Dr. Asif Hossain Khan,
Professor,
Department of Computer Science and Engineering,
University of Dhaka

Table of Contents

BERT

Today we are going to talk about **bert**.

No, no, we are not going to talk about cartoon BERT. We are going to talk about **Google BERT**.

BERT, What is BERT!

BERT (**B**idirectional **E**ncoder **R**epresentations from **T**ransformers) is awesome research in Natural Language Processing (NLP) published by researchers at Google AI Language in 2018. 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 Neural Machine Translation, Question Answering (SQuAD v1.1), Sentence Pair Classification task (MNLI), Sentiment Analysis, Text Summarization and others.**

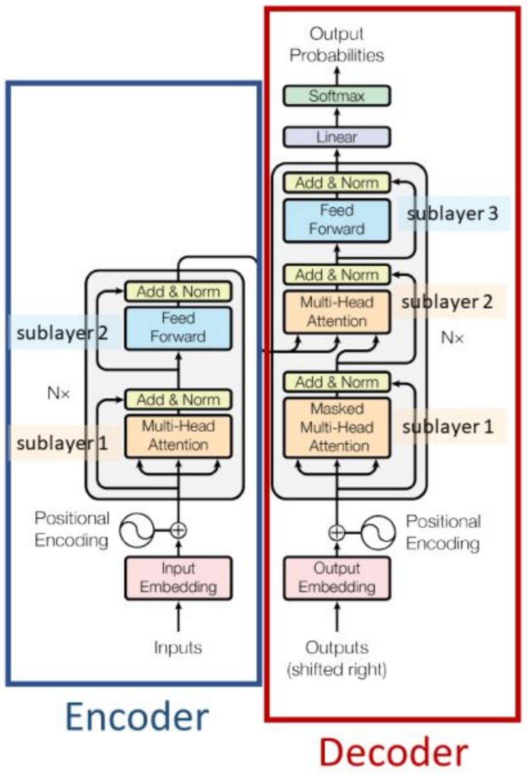
From the abbreviation of the **BERT**, we can figure out some kind of feature of it.

1. It is Bi-directional
2. It uses an Encoder Representation
3. It has a Transformer based architecture

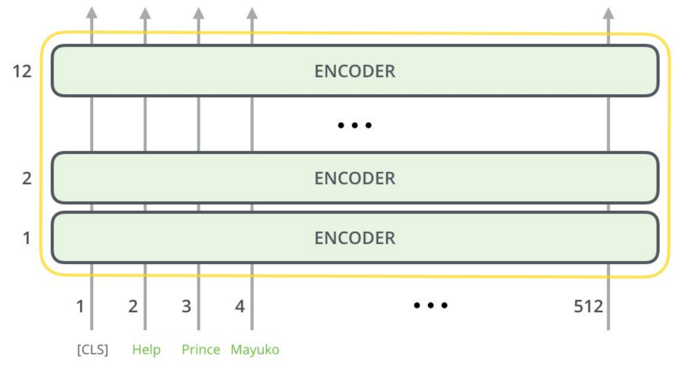
So basically, BERT's key technical innovation is applying the bidirectional training of Transformer, a popular attention model, to language modeling. This is in contrast to previous efforts which looked at a text sequence either from left to right or combined left-to-right and right-to-left training. The paper's results show that a language model that is bi-directionally 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 Language Modelling (MLM) which allows bidirectional training in models** in which it was previously impossible. We will walk through all the details later in this material.

High-Level Overview of BERT

Now, let's look at a high-level overview of the **BERT** architecture. It is a **Transformer** based model architecture, which opens a new era to work with NLP tasks. In the last session, we learned about the Transformer. A transformer is an Encoder-Decoder model architecture that also uses positional encoding, self-attention, multi-head attention, and also with Residual connection.



Now, BERT uses only the **Encoder** portion of the Transformer architecture canceling out the Decoder part. Like the Transformer, BERT also uses positional encoding, self-attention, multi head attention, and Residual Connection. It uses exactly the same architecture for encoders that are used in the Transformer. In short, BERT is basically, **Stacking of Encoders** and the encoder is from the Transformer.

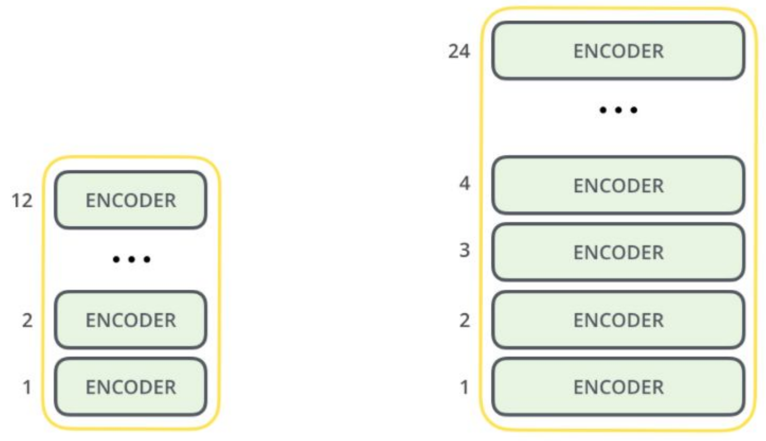


BERT

There are two types of BERT.

1. **BERT base**
2. **BERT large**

In bert_base, it is used twelve (12) encoder stacking. On the other hand, in bert_large twenty-four (24) encoder stacking is used. The types are also different from feedforward-networks (no. of hidden neurons). For the bert_base, 768 neurons are used in the feedforward network where the bert_large uses 1024 hidden units, and more attention heads (12 and 16 respectively) than the default configuration in the reference implementation of the Transformer in the initial paper (6 encoder layers, 512 hidden units, and 8 attention heads).



BERT$_{BASE}$ BERT$_{LARGE}$

How does BERT come and why has it become so popular?

Nowadays, BERT has been the first choice to work on solving different NLP tasks. Why has it become so popular in the NLP community? It is not just for the being State Of The Art (SOTA) model but also several reasons behind it. The reasons can be differentiated into three main reasons.

1. Limitations of Transformer
2. Transfer Learning in NLP
3. Fully replacement of LSTM

1. Limitations of Transformer

No doubt, the invention of the transformer in NLP is one of the most powerful and efficient research in the NLP community. It describes the NLP task in a different way for which the NLP model can be trained independently. As a result, **parallelization is possibl**e and so the training process becomes **much faster than LSTM/GRU-based models**. It has achieved LSTM/GRU models on different tasks such as machine translation started to make some in the field think of them as a replacement to LSTMs. This was compounded by the fact that Transformers deal with long-term dependencies better than LSTMs.

Ok, it is well designed for the encoder-decoder architecture which is able to solve some NLP tasks like **Machine Translation**. But there are many more tasks which do not use an encoder-decoder architecture just like **Text Classification, Sentence Pair task and so on**. On that task, how can we use transformers on those tasks? Here is the limitation of the transformer. For this limitation, we can't replace LSTM

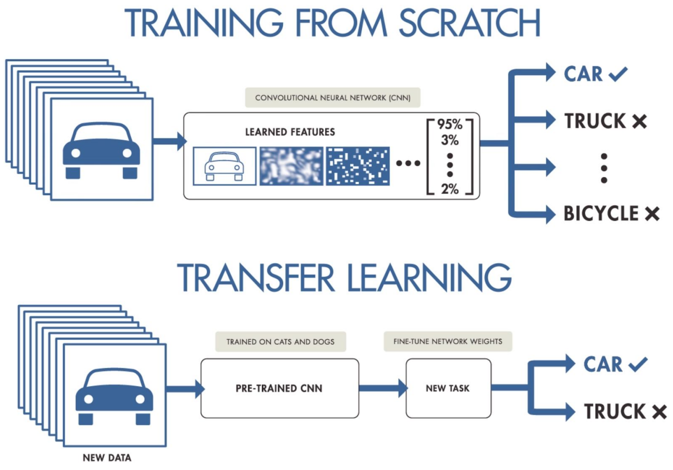
2. Transfer Learning in NLP

Transfer learning is the reuse of a pre-trained model on a new problem. It's currently very popular in deep learning because it can train deep neural networks with comparatively little data. This is very useful in the data science field since most real-world problems typically do not have millions of labeled data points to train such complex models.

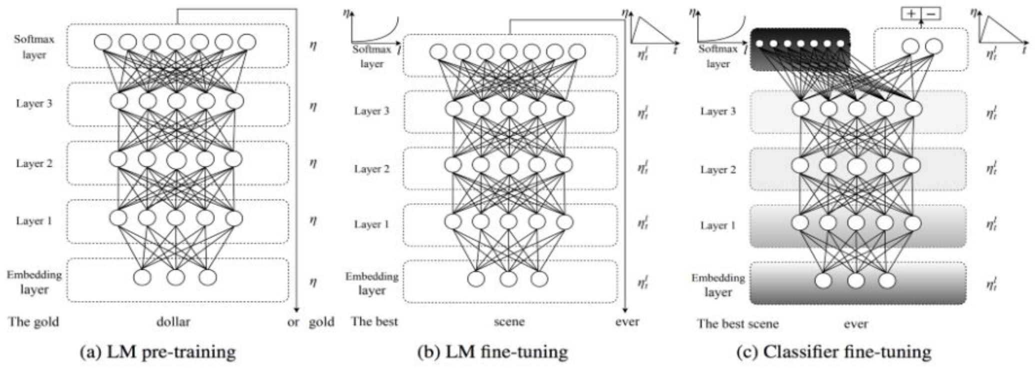
Transfer learning is used not only for the faster training process but also for building an efficient model. In computer vision tasks, transfer learning is widely used. A robust model is trained on a huge amount of image data and finds the optimal weights. Then, the model can be used for downstream computer vision tasks as a pre-trained model. So, by adding some NN layers and fine-tuning the pre-trained model, it is easy to build an efficient model on that downstream task. Some pre-trained models in computer vision are VGG-16, ResNet, ImageNet, LeNet, Efficient net, and many more.

For computer vision, we have a very good set of well-trained models on millions of data and they can be used easily to perform object recognition tasks. **We can build a robust and very**

accurate model with 20 lines of code. Just importing a pre-trained model and fine-tuning a few layers will give us the desired result.

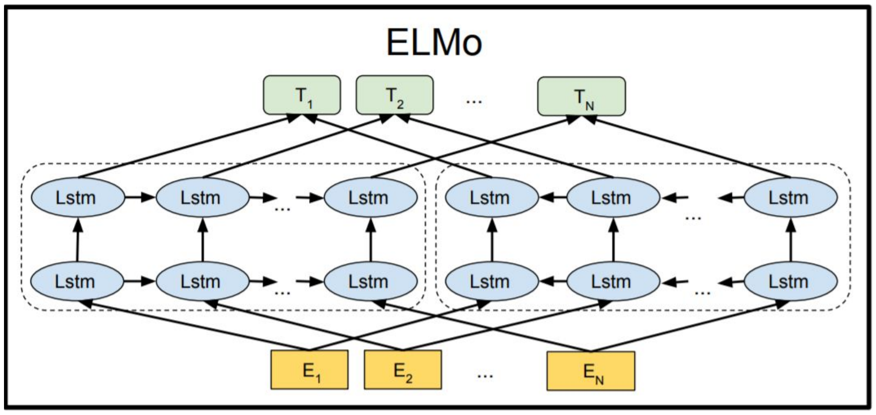


Though transfer learning is a much more popular and widely used model, in NLP we've seen very little use of this. For NLP, the process is more complicated, because in NLP different text data contains different contexts. So, to use transfer learning we need good contextual language modeling. Contextual language modeling means the context of the sentence (for example, ***abide by*** **and** ***abide in*** having fully different meanings though both use the main word *abide*). **Better contextual language modeling we can define, better models we can build using transfer learning.** After much research, some good contextual language models were built like **Elmo**, **ULMFiT** which had become popular. But both **Elmo**(**E**mbeddings from **L**anguage **Mo**dels) and **ULMFiT**(**U**niversal **L**anguage **M**odel **Fi**ne-**T**uning) were LSTM based. In Elmo, 2 layers of bi-directional LSTM were used and in ULMFit 3 layers of LSTM were used.



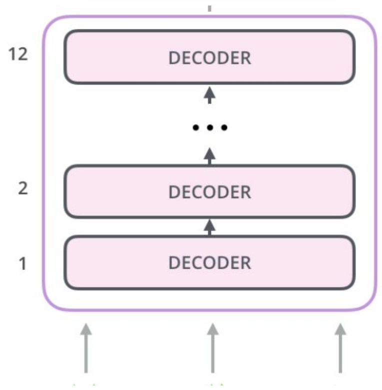
a) ULMFiT training for general language models (b) and then training for a specific language model © and then actually training for that work (Picture from ULMFiT paper)

Here E_1, E_2, ... E_N is the word embedding, and T_1, T_2, ... T_N is the output of the model



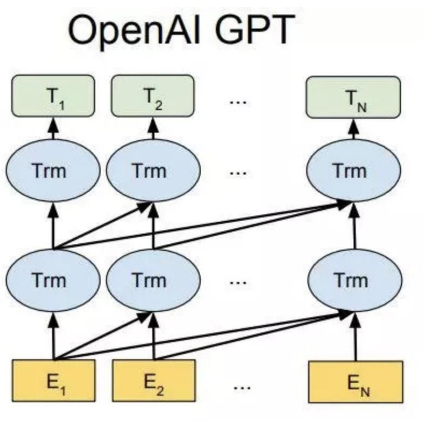
But, what if we can build a Language Modeling with a transformer in place of LSTM. As transformer is much faster than LSTM, so we can train much more data to train by which we can define a much better language model. Basically, a transformer-based Language Modeling is needed which also can be used as a pre-training model in transfer learning.

Before BERT, OpenAI introduces their GPT-2 (**G**enerative **P**re-**T**raining) model which can solve the problem discussed in the previous section. GPT-2 model is transformer based. If we look at the architecture of GPT-2 we will see, **GPT-2 uses a 12 layer stacking of the decoder** (unlike BERT, it uses only the decoder). The encoder of the transformer is canceled out in the model. As, the model uses only the decoders so it also cancels out the **Encoder-Decoder Attention** block from the decoder architecture. The rest remains the same.

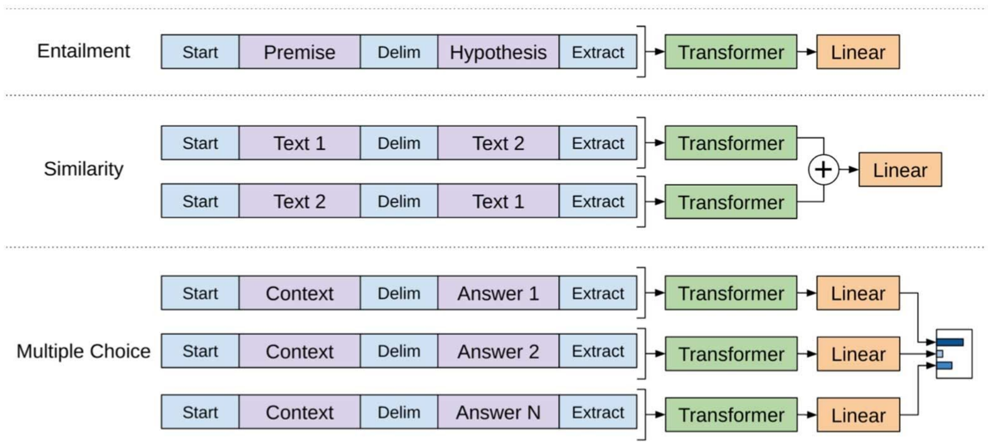


We already know that the decoder in the transformer uses **masked self-attention** to predict the next word for learning. That's why the decoder always gives shifted right words as a prediction. It works like the forward LSTM (from left to right). As GPT-2 is able to predict the next word for any sentence in an autoregressive way using a transformer, so making a language modeling is possible with a transformer.

Here E_1, E_2, ... E_N is the word embedding, T_1, T_2, ... T_N is the output of the model and *Trm* is Transformer Decoder.



GPT-2 is used as a pre-trained model in transfer learning using a transformer. It also becomes able to use the transformer in different NLP tasks. Here is the big picture of how to use GPT-2 (Transformer) in different NLP tasks. For more about GPT-2, you can explore **their blogs**



3. Fully Replacement of LSTM

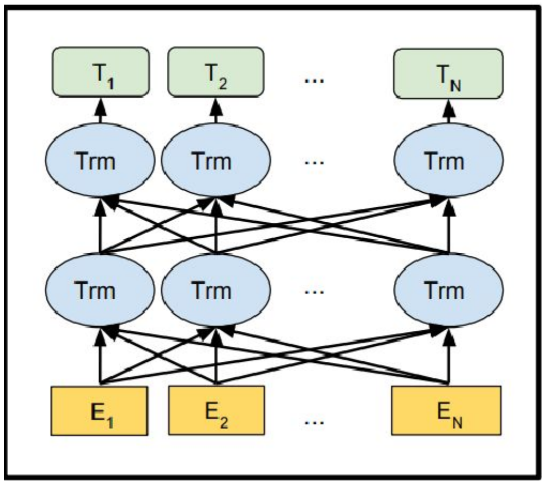
In the previous section, we've seen that OpenAI GPT-2 becomes able to replace LSTM in many NLP tasks using a transformer. Then, one important question arises, is GPT-2 able to replace LSTM totally? The answer is **NO**. Because, while we use LSTM in our task, we can build more robust and contextual language modeling using bi-directionality. **Bi-directional LSTM** understands the better context in language modeling. As GPT-2 uses only the forward direction (from left to right), in some cases **Bi-directional LSTM** works much better by understanding the better context of the language. So, we could not replace LSTM fully. Now, there arises one question: is it possible to build a model adding **bi-directionality** using the **transformer?** If it could be done, LSTM would fully be replaced by the transformer.

So, there required to build one model which

1. Is capable to use a transformer in all kinds of NLP task
2. Can be used as a pre-trained model for the downstream task while using transfer learning in NLP
3. Is bi-directional and understands the language from left to right and right to left
4. Understands deeper and better context of the language

Here comes BERT !

We've already seen the basic overview of **BERT**. Basically, BERT is the stacking of **Encoders**. Now, it is high time to explain why they use only the encoder? It is due to use BERT in all kinds of tasks. Another reason is that we've seen that OpenAI GPT-2 uses stacking of decoders. As a result, they would build their model in an autoregressive model (next word prediction). So, they were able to define unidirectional (left to right) transformer architecture. To build bi-directional transformer architecture BERT uses an encoder from the transformer instead of the decoders. So, BERT builds a model for not only left to right but also right to left. A simple overview of BERT is like:

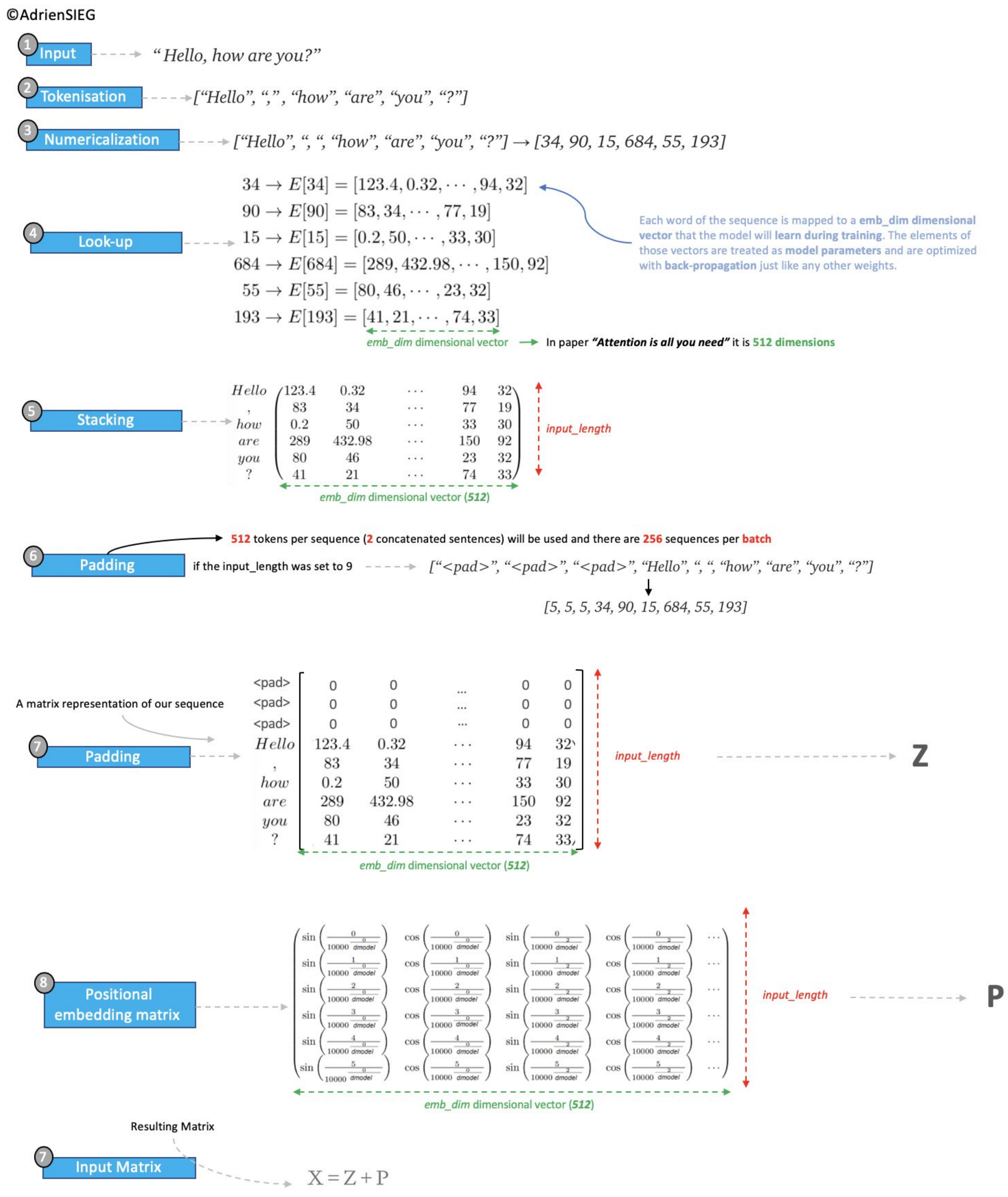


Now, what is inside the BERT? It is not just the stacking of encoders but also many more. BERT also introduces **masked language modeling and next sentence prediction**. There are four major parts of BERT.

1. From Word to Vectors
2. The Encoders from the transformer
3. Masked Language Modeling (MLM)
4. Next Sentence Prediction (NSP)

It is high time to give a brief description of those parts. First of all, we need to know that BERT takes two sentences as input at a time. Let say those are sentence A and sentence B.

1. From Word to Vectors


©AdrienSIEG
1 Input
"Hello, how are you?"
2 Tokenisation
["Hello", ",", "how", "are", "you", "?"]
3 Numericalization
["Hello", ", ", "how", "are", "you", "?"] → [34, 90, 15, 684, 55, 193]
4 Look-up
34 → E[34] = [123.4, 0.32, ··· , 94, 32]
90 → E[90] = [83, 34, ··· , 77, 19]
15 → E[15] = [0.2, 50, ··· , 33, 30]
684 → E[684] = [289, 432.98, ··· , 150, 92]
55 → E[55] = [80, 46, ··· , 23, 32]
193 → E[193] = [41, 21, ··· , 74, 33]
Each word of the sequence is mapped to a emb_dim dimensional vector that the model will learn during training. The elements of those vectors are treated as model parameters and are optimized with back-propagation just like any other weights.
emb_dim dimensional vector
In paper "Attention is all you need" it is 512 dimensions
5 Stacking
input_length
emb_dim dimensional vector (512)
6 Padding
512 tokens per sequence (2 concatenated sentences) will be used and there are 256 sequences per batch
if the input_length was set to 9
["<pad>", "<pad>", "<pad>", "Hello", ", ", "how", "are", "you", "?"]
[5, 5, 5, 34, 90, 15, 684, 55, 193]
A matrix representation of our sequence
7 Padding
input_length
Z
emb_dim dimensional vector (512)
8 Positional embedding matrix
input_length
P
emb_dim dimensional vector (512)
Resulting Matrix
7 Input Matrix
X = Z + P

is the input of the first encoder block and has dimensions *(input_length) x (emb_dim)*.

- **Tokenization** is the task of chopping it up into pieces, called tokens, perhaps at the same time throwing away certain characters, such as punctuation.

- **Using word pieces** (e.g. playing -> play + ##ing) instead of words. This is effective in reducing the size of the vocabulary and increases the amount of data that is available for each word.

- **Numericalization** aims at mapping each token to a unique integer in the corpus's vocabulary.

- **Token embedding** is the task of getting the embedding (i.e. a vector of real numbers) for each word in the sequence. Each word of the sequence is mapped to an emb_dim dimensional vector that the model will learn during training. You can think about it as a vector look-up for each token. The elements of those vectors are treated as model parameters and are optimized with back-propagation just like any other weights.

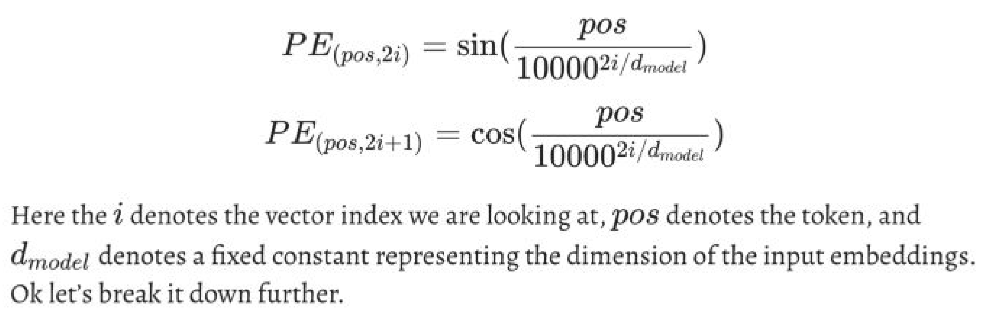
- **Padding** was used to make the input sequences in a batch have the same length. That is, we increase the length of some of the sequences by adding '' tokens.

- **Positional encoding** Recall that the positional encoding is designed to help the model learn some notion of sequences and relative positioning of tokens. This is crucial for language-based tasks especially here because we are not making use of any traditional recurrent units such as RNN, GRU or LSTM

  Intuitively, we aim to be able to modify the represented meaning of a specific word depending on its position. We don't want to change the full representation of the word but we want to modify it a little to encode its position by adding numbers between [-1,1] using predetermined (non-learned) sinusoidal functions to the token embeddings. For the rest of the Encoder, the word will be represented slightly differently depending on the position the word is in (even if it is the same word).

  The encoder must be able to use the fact that some words are in a given position while, in the same sequence, other words are in other specific positions. That is, we want the

network to be able to understand relative positions and not only absolute ones.
Now, the function used for the positional encoding is given below:



$$PE_{(pos,2i)} = \sin(\frac{pos}{10000^{2i/d_{model}}})$$
$$PE_{(pos,2i+1)} = \cos(\frac{pos}{10000^{2i/d_{model}}})$$

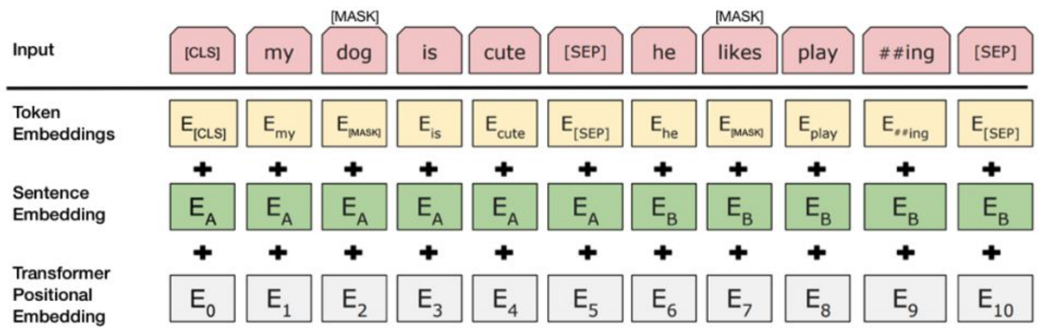
Here the i denotes the vector index we are looking at, pos denotes the token, and d_{model} denotes a fixed constant representing the dimension of the input embeddings.
Ok let's break it down further.

- Here, several questions arise like why the function uses sinusoidal functions, why the function does not use a single sin or cos function, and finally why this encoding is added up with the encoding instead of concatenation? First of all, to encode anything binary encoding is the first choice. But all we know is that all the weight, bias, and the others of a NN model ranges from [-1, 1]. So, if we want to use binary encoding for this then we need float continuous values to encode. Obviously, binary encoding of the floating values is much more complex and very much a waste of space. To solve this condition, they use the sinusoidal function.

  Besides, the function uses a combination of sin and cos, because it works equivalent to alternative bits when we use binary encoding. Single sin or single cos can not do this. For example, for **pos = 0** and for **index = 0,1** the function gives us **0,1** respectively which just like the alternative bits in binary encoding. Finally, this positional encoding is summed up with the embedding representation instead of concatenation just because providing a good source of features and storing a smaller dim vector. (if we concatenate, we need to store higher dimensionality vectors)

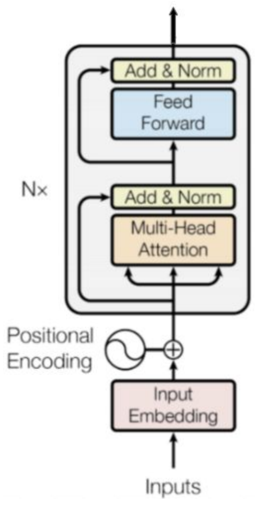
- **Sentence embedding** techniques represent entire sentences and their semantic information as vectors. This helps the machine in understanding the context, intention, and other nuances in the entire text. It simply notices which word belongs to which sentences. A marker indicating Sentence A or Sentence B is added to each token. This allows the model to distinguish between sentences.

After the first step, we did our embedding and encoding step just like


Input
[MASK]
[MASK]
[CLS]
my
dog
is
cute
[SEP]
he
likes
play
##ing
[SEP]
Token Embeddings
E[CLS] E my E [MASK] E is E cute E [SEP] E he E [MASK] E play E ##ing E [SEP]
Sentence Embedding
E A E A E A E A E A E A E B E B E B E B E B
Transformer Positional Embedding
E 0 E 1 E 2 E 3 E 4 E 5 E 6 E 7 E 8 E 9 E 10

2. The Encoders from the transformer

We already know about the transformer architecture. The transformer architecture is based on encoder-decoder form. The encoder consists of four parts (self attention, multi-head attention, residual connections and normalization and feed forward network). The encoder of the transformer architecture looks like



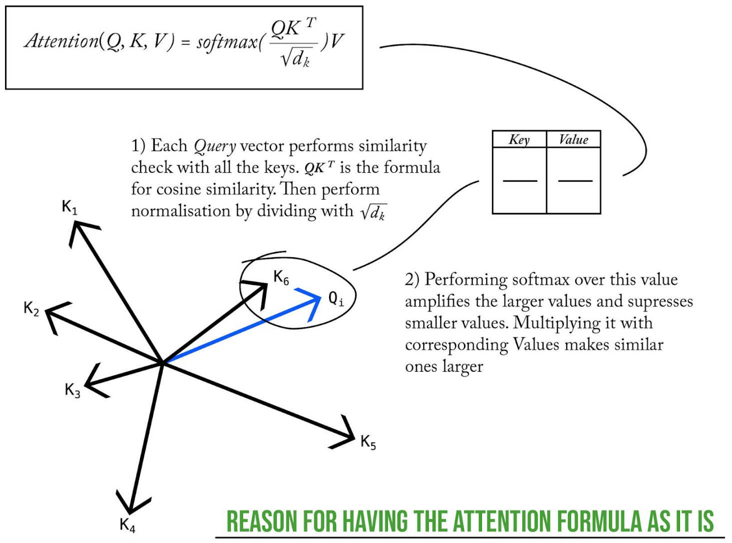
In the encoder architecture, we've seen that there is a four-measure part in the encoder part of the transformer. To understand the whole big picture of the encoder architecture of the transformer the following topics need to be understood

1. Attention Score Measurement
2. Self Attention and its intuition
3. Multi-Head Attention/ Self Attention
4. Add & Norm
5. Feed Forward Network

Those topics are already covered. Let see a quick recap on those topics.

i. Attention Score Measurement

The attention mechanism is a part of a neural architecture that enables to dynamically highlight relevant features of the input data, which, in NLP, is typically a sequence of textual elements. It can be applied directly to the raw input or to its higher level representation. There are several functions used for measuring attention score (e.g additive attention score, dot product attention score). The most common is using dot product attention score by generating query, key , value vector. The function is used for measuring such attention score is given below



$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$
Key
Value
1) Each *Query* vector performs similarity check with all the keys. QK^T is the formula for cosine similarity. Then perform normalisation by dividing with $\sqrt{d_k}$
2) Performing softmax over this value amplifies the larger values and supresses smaller values. Multiplying it with corresponding Values makes similar ones larger
K_1
K_2
K_3
K_4
K_5
K_6
Q_i
REASON FOR HAVING THE ATTENTION FORMULA AS IT IS

For example, let's say we have a seq2seq type task. We use encoder-decoder architecture to solve the task. Now, let assume

At time step t we've encoder representation h_t = [4,5] and the previous time step decoder representation s_t-1 = [2,3]

So, our query vector, q = h_t = [4,5] and key vector, k = s_t-1 = [2,3]

The dot product will be = q X transpose(k) = [4,5] X transpose([2,3]) = 4*2 + 5*3 = 23

ii. Self Attention and its intuition

Self-**attention**, also known as intra-**attention**, is an **attention** mechanism relating different positions of a single sequence in order to compute a representation of the same sequence.

While doing self-attention in transformer, we follow few steps

- Create three vectors from each of the encoder's input vectors
- For each word, we create a Query vector, a Key vector, and a Value vector
- These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

We need to generate three vectors (Q,K and V) for each word embedding because we need to find the attention score for each word with all the words in the input sequence where the word belongs to.

We will generate three vectors (Q,K and V) for each word embedding by multiplying the word embedding with three weight metrics (W_Q,W_K,W_V). The metrics will be learned by model via backpropagation.

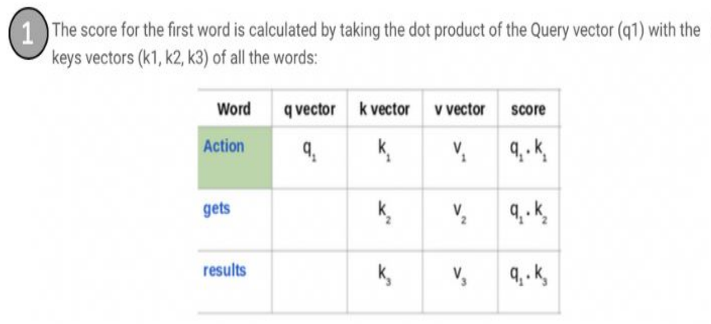
Notice that these new vectors are smaller in dimension than the embedding vector. Their dimensionality is 64, while the embedding and encoder input/output vectors have a dimensionality of 512.

Why is dimensionality 64? As we must have :

-> Output's dimension is [length of input sequences] x [dimension of embeddings — 512]

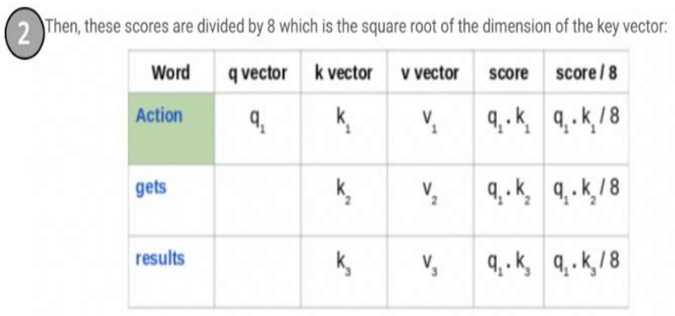
-> We use 8 heads during the Multi-head Self-Attention process. The output size of a given self-attention vector is [length of input sequences] x [64]. So the concatenated vector resulting from all Multi-head Self-Attention process would be [length of input sequences] x ([64] x [8]) = [length of input sequences] x ([512])

So, we will get a 64 dimension query, key, and value vector for each word.



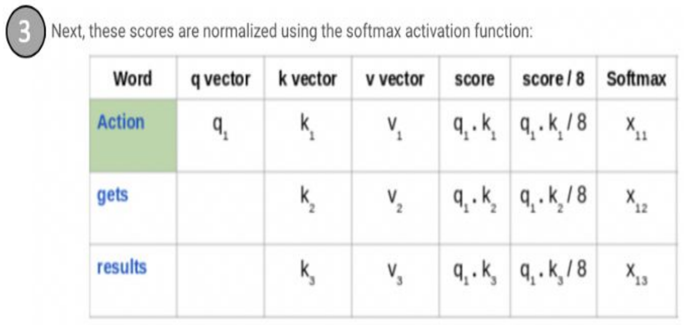
1 The score for the first word is calculated by taking the dot product of the Query vector (q1) with the keys vectors (k1, k2, k3) of all the words:

Word	q vector	k vector	v vector	score
Action	q_1	k_1	v_1	$q_1 \cdot k_1$
gets		k_2	v_2	$q_1 \cdot k_2$
results		k_3	v_3	$q_1 \cdot k_3$



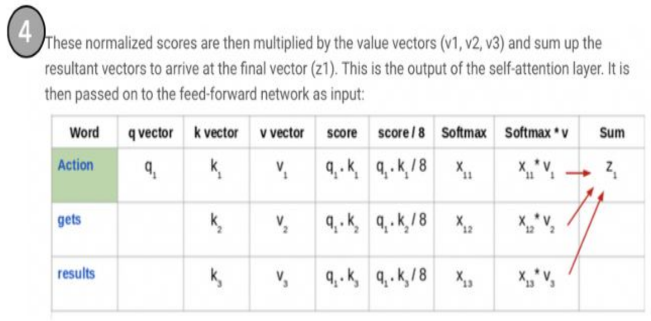
2 Then, these scores are divided by 8 which is the square root of the dimension of the key vector:

Word	q vector	k vector	v vector	score	score / 8
Action	q_1	k_1	v_1	$q_1 \cdot k_1$	$q_1 \cdot k_1 / 8$
gets		k_2	v_2	$q_1 \cdot k_2$	$q_1 \cdot k_2 / 8$
results		k_3	v_3	$q_1 \cdot k_3$	$q_1 \cdot k_3 / 8$



3 Next, these scores are normalized using the softmax activation function:

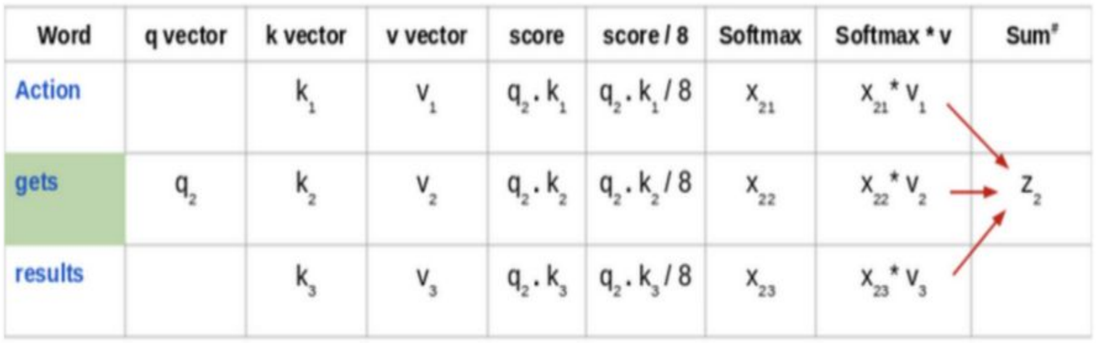
Word	q vector	k vector	v vector	score	score / 8	Softmax
Action	q_1	k_1	v_1	$q_1 \cdot k_1$	$q_1 \cdot k_1 / 8$	x_{11}
gets		k_2	v_2	$q_1 \cdot k_2$	$q_1 \cdot k_2 / 8$	x_{12}
results		k_3	v_3	$q_1 \cdot k_3$	$q_1 \cdot k_3 / 8$	x_{13}



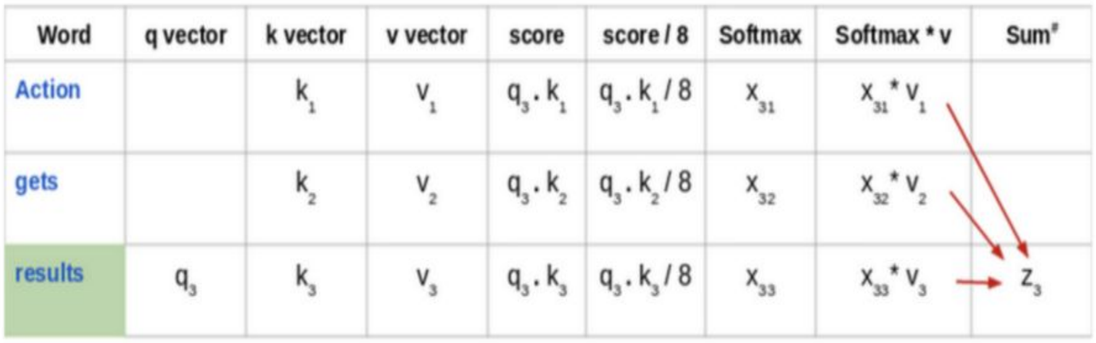
4 These normalized scores are then multiplied by the value vectors (v1, v2, v3) and sum up the resultant vectors to arrive at the final vector (z1). This is the output of the self-attention layer. It is then passed on to the feed-forward network as input:

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum
Action	q_1	k_1	v_1	$q_1 \cdot k_1$	$q_1 \cdot k_1 / 8$	x_{11}	$x_{11} * v_1$	z_1
gets		k_2	v_2	$q_1 \cdot k_2$	$q_1 \cdot k_2 / 8$	x_{12}	$x_{12} * v_2$	
results		k_3	v_3	$q_1 \cdot k_3$	$q_1 \cdot k_3 / 8$	x_{13}	$x_{13} * v_3$	

So, z1 is the self-attention vector for the first word of the input sequence "Action gets results". We can get the vectors for the rest of the words in the input sequence in the same fashion:



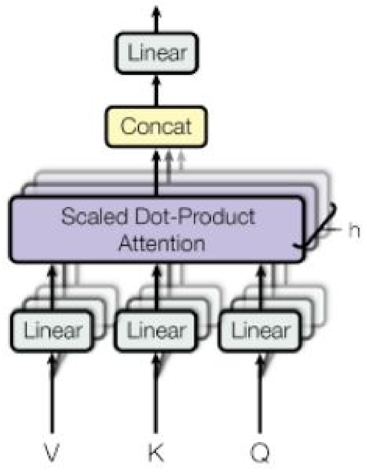
Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum*
Action		k_1	v_1	$q_2 \cdot k_1$	$q_2 \cdot k_1 / 8$	x_{21}	$x_{21} * v_1$	
gets	q_2	k_2	v_2	$q_2 \cdot k_2$	$q_2 \cdot k_2 / 8$	x_{22}	$x_{22} * v_2$	z_2
results		k_3	v_3	$q_2 \cdot k_3$	$q_2 \cdot k_3 / 8$	x_{23}	$x_{23} * v_3$	



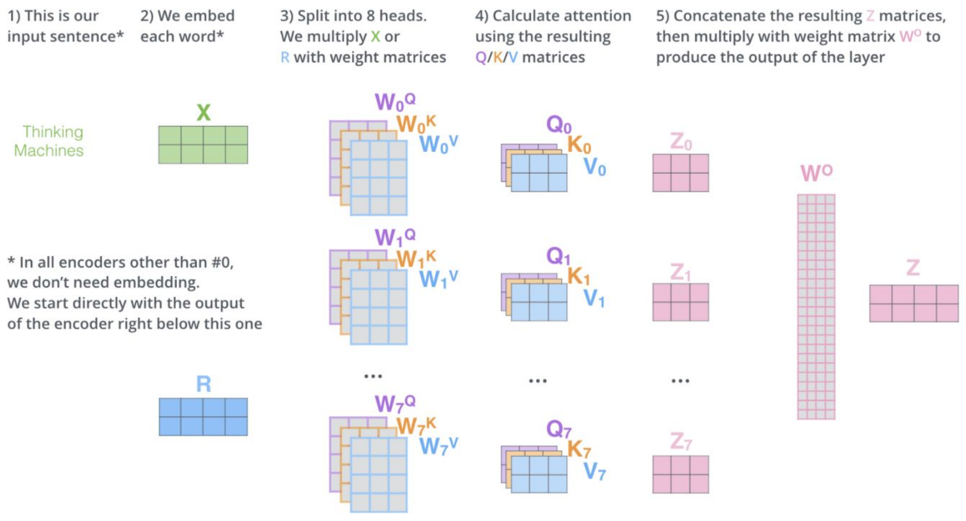
Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum*
Action		k_1	v_1	$q_3 \cdot k_1$	$q_3 \cdot k_1 / 8$	x_{31}	$x_{31} * v_1$	
gets		k_2	v_2	$q_3 \cdot k_2$	$q_3 \cdot k_2 / 8$	x_{32}	$x_{32} * v_2$	
results	q_3	k_3	v_3	$q_3 \cdot k_3$	$q_3 \cdot k_3 / 8$	x_{33}	$x_{33} * v_3$	z_3

iii. Multi-Head Attention

Multi-head attention means keeping attention to multiple words in a sequence for a single word. Multi-head attention is a module for attention mechanisms which runs through an attention mechanism several times in parallel. Intuitively, multiple attention heads allows for attending to parts of the sequence differently (e.g. longer term dependencies versus shorter-term dependencies).



Multi-Head Attention


1) This is our input sentence*
2) We embed each word*
3) Split into 8 heads. We multiply X or R with weight matrices
4) Calculate attention using the resulting Q/K/V matrices
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer
Thinking Machines
X
R
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one
W_0^Q W_0^K W_0^V
W_1^Q W_1^K W_1^V
W_7^Q W_7^K W_7^V
Q_0 K_0 V_0
Q_1 K_1 V_1
Q_7 K_7 V_7
Z_0
Z_1
Z_7
W^O
Z

Till now the whole picture is like that

7 Score

$$\frac{Q_i K_i^T}{\sqrt{d_k}} \quad \begin{array}{c|cccccc} & Hello & , & how & are & you & ? \\ \hline Hello & 78.49 & 43.29 & 1.2 & 41.74 & 91.43 & 74.47 \\ , & 95.84 & 28.78 & 57.13 & 68.20 & -60.94 & 26.85 \\ how & -95.69 & -52.16 & 17.00 & 45.71 & 48.49 & 64.35 \\ are & -69.92 & 85.16 & 94.94 & 91.04 & -92.83 & 77.49 \\ you & 65.85 & 55.85 & 62.54 & -97.46 & 76.38 & 13.20 \\ ? & -30.05 & -4.52 & 76.02 & 42.35 & 15.29 & 63.61 \end{array}$$

$$\begin{array}{c|cccccc} & Hello & , & how & are & you & ? \\ \hline Hello & 78.49 & 43.29 & 1.2 & 41.74 & 91.43 & 74.47 \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ & q1 \times k1 /8 & q1 \times k2 /8 & q1 \times k3 /8 & q1 \times k4 / 8 & q1 \times k5 /8 & q1 \times k6 /8 \\ ? & -30.05 & -4.52 & 76.02 & 42.35 & 15.29 & 63.61 \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ & q6 \times k1 /8 & q6 \times k2 /8 & q6 \times k3 /8 & q6 \times k4 / 8 & q6 \times k5 /8 & q6 \times k6 /8 \end{array}$$

8 Softmax

$$Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) \quad \begin{array}{c|cccccc|c} & Hello & , & how & are & you & ? & \\ \hline Hello & 72.40*10^{-06} & 1.23*10^{-21} & 6.51*10^{-40} & 2.62*10^{-22} & \boxed{9.99*10^{-01}} & 4.30*10^{-08} & =1 \\ , & \boxed{1.00*10^{+00}} & 7.51*10^{-30} & 1.54*10^{-17} & 9.91*10^{-13} & 8.15*10^{-69} & 1.09*10^{-30} & =1 \\ how & 3.12*10^{-70} & 2.51*10^{-51} & 2.72*10^{-21} & 8.03*10^{-09} & 1.29*10^{-07} & \boxed{9.99*10^{-01}} & =1 \\ are & 2.47*10^{-72} & 5.54*10^{-05} & \boxed{9.80*10^{-01}} & 1.98*10^{-02} & 2.77*10^{-82} & 2.58*10^{-08} & =1 \\ you & 2.67*10^{-05} & 1.21*10^{-09} & 9.75*10^{-07} & 3.17*10^{-76} & \boxed{9.99*10^{-01}} & 3.64*10^{-28} & =1 \\ ? & 8.59*10^{-47} & 1.05*10^{-35} & \boxed{9.99*10^{-01}} & 2.38*10^{-15} & 4.21*10^{-27} & 4.07*10^{-06} & =1 \end{array}$$

$$\Downarrow$$

The network will learn over training time which relationships are more useful and will relate tokens to each other based on these relationships

UNDERSTANDING

$$\begin{array}{c|cccccc|c} & Hello & , & how & are & you & ? & \\ \hline Hello & 0.1 & 0 & 0.06 & 0.1 & 0.6 & 0.14 & =1 \\ , & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & =1 \\ how & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & =1 \\ are & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & =1 \\ you & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & =1 \\ ? & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & =1 \end{array}$$

For the sake of understanding let's propose **a dummy simplification** by simplifying the previous matrix

9 Dot Value vector

emb_dim dimensional vector (***512***)

$$Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i \quad \begin{array}{c|cccccc} & Hello & , & how & are & you & ? \\ \hline Hello & 0.1 & 0 & 0.06 & 0.1 & 0.6 & 0.14 \\ , & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ how & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ are & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ you & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ ? & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{array} \begin{pmatrix} v_{Hello} \\ v_{,} \\ v_{how} \\ v_{are} \\ v_{you} \\ v_{?} \end{pmatrix}_{d_v} = \begin{array}{c|c} & \longleftarrow d_v \longrightarrow \\ \hline \boxed{Hello} & 0.1v_{Hello} + 0v_{,} + 0.06v_{how} + 0.1v_{are} + \boxed{0.6v_{you}} + 0.14v_{?} \\ , & \cdots\cdots\cdots\cdots \\ how & \cdots\cdots\cdots\cdots \\ are & \cdots\cdots\cdots\cdots \\ you & \cdots\cdots\cdots\cdots \\ ? & \cdots\cdots\cdots\cdots \end{array}$$

The resulting representation of "**Hello**" as a weighted combination (centroid) of the projected vectors through V_i of the input tokens.

Where **v_{token}** is the projection through V_i of the **token's representation**. In this case the word "**hello**" ends up with a representation based on the 4th token "**you**" of the input for that head.

a specific head captures a specific relationship between the input tokens

If we do that h times (a total of h heads) **each encoder block is capturing h different relationships between input tokens**.

First row of first head $\dashrightarrow$ $V_{Hello,1} = 0.1v_{Hello} + 0v_{,} + 0.06v_{how} + 0.1v_{are} + 0.6v_{you} + 0.14v_{?}$

First row of Multi-head head $\dashrightarrow$ $Concat(V_1, V_2, ..., V_h)W_0$

Size of Matrix of Multi-head head $\dashrightarrow$ $(input\_length)\ x\ (emb\_dim)$

64 x 8 = **512**

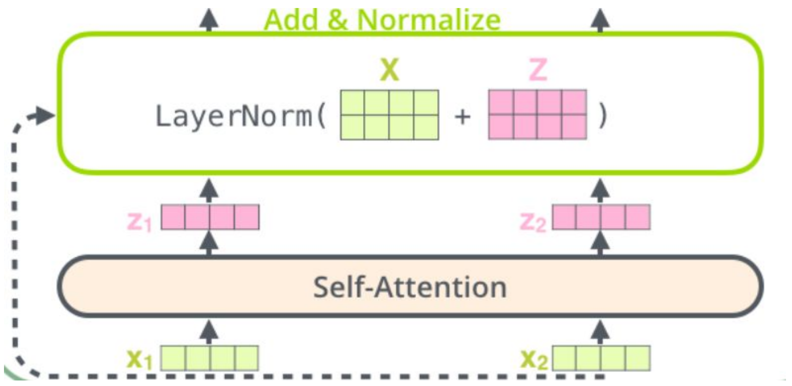
The representation of the token is the **concatenation of h weighted combinations of token representations (centroids) through the h different learned projections.**

iv. Add and Norm

Here Add means the *residual connection* and norm means the layer normalization. It also uses dropout in this layer to reduce overfitting.

$$LayerNorm(x + Dropout(Sublayer(x)))$$

To have a clear look let consider the following example



Here, Self attention is the sublayer and X = [x1, x2] , Z = [z1, z2]

Now, the equation of the layer normalization

Layernorm changes input to have mean 0 and variance 1,
per layer and per training point (and adds two more parameters)

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2} \qquad h_i = f\left(\frac{g_i}{\sigma_i}\left(a_i - \mu_i\right) + b_i\right)$$

v. Feed Forward Network

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.This can be simplified by the equation

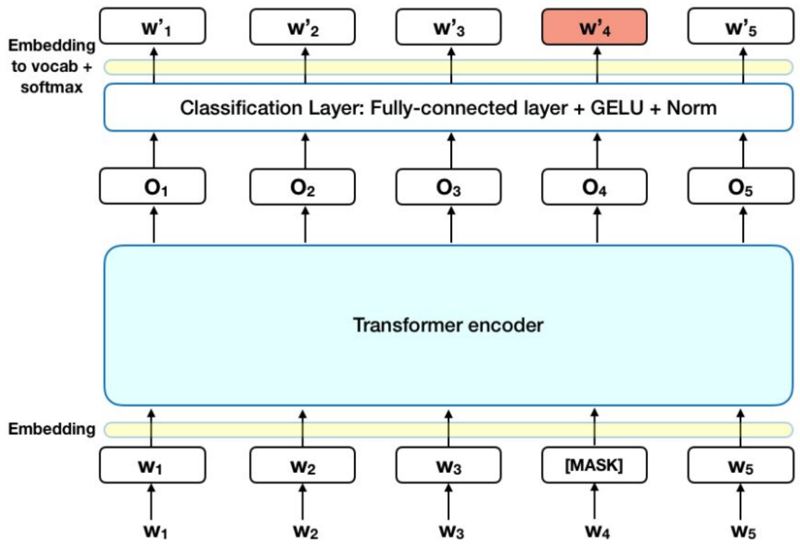
$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

3. Masked Language Modeling (MLM)

One of the greatest features of BERT is Masked Language Modeling (MLM). The masked language model randomly masks some of the tokens from the input, and the objective is to predict the original vocabulary id of the masked word based only on its context. Unlike left-to-right language model pre-training, the MLM objective allows the representation to fuse the left and the right context, which allows us to pre-train a deep bidirectional Transformer.

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:

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



W'1
W'2
W'3
W'4
W'5
Embedding to vocab + softmax
Classification Layer: Fully-connected layer + GELU + Norm
O1
O2
O3
O4
O5
Transformer encoder
Embedding
W1
W2
W3
[MASK]
W5
W1
W2
W3
W4
W5

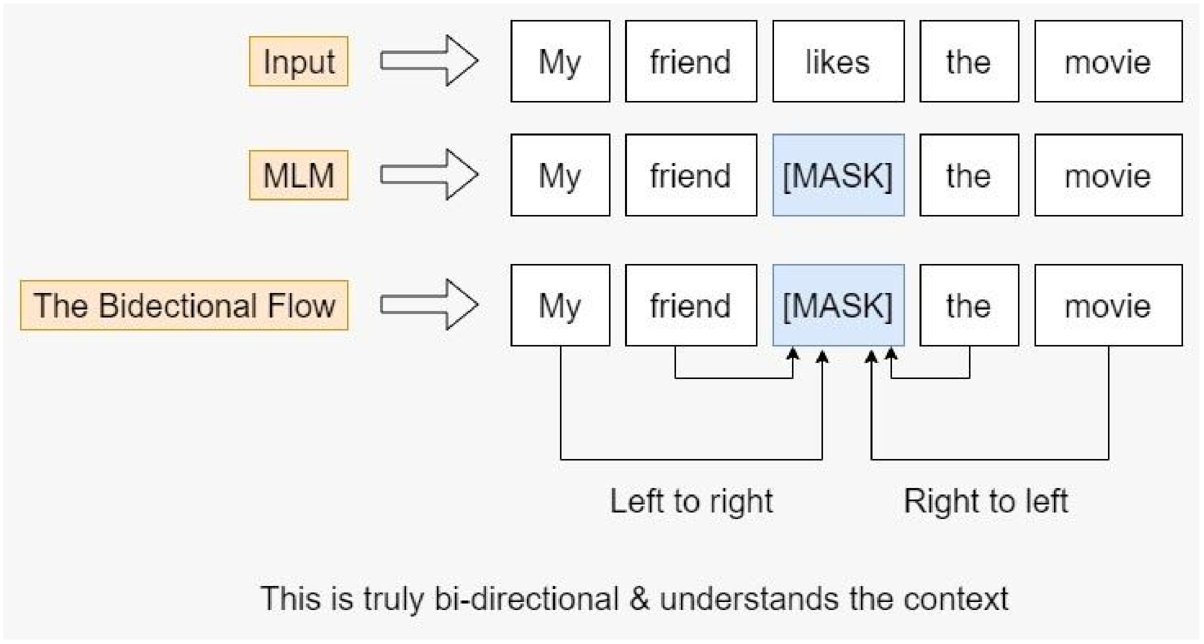
Here w_1, …, w_5 are the word embedding, and o_1, …, o_5 are the outputs of the BERT. BERT uses GELU (Gaussian Error Linear Unit) activation.

$$\mathrm{GELU}(x) := x\mathbb{P}(X \leq x) = x\Phi(x) = 0.5x\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

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.

Some important notes are the following

- **It is true bi-directional**
  Before BERT's MLM, bi-directional means just generating word representation going from left to right and right to left and then simply add or create the two directional representation. For example, if we use bi-directional LSTM, one LSTM goes from left to right generating the representation of the words, and the other goes from right to left. After that, it concatenates generated representations of the words. But it understands less context about the language. But in the case of MLM, it masks a word within the text so no need to train the model from both sides. Besides, the masked word is within the text, so BERT automatically follow that direction to the masked word from the other words



- **Semi-Supervised Training**
  As BERT randomly masks the word from the input text and learns from predicting the masked word, so there is no need to use supervised data with the label. So we can use

both unsupervised and supervised data for training the BERT. As semi supervised data can be used for training the BERT, so we get a huge amount of data for training. It will help us to build a more robust and contextual model.

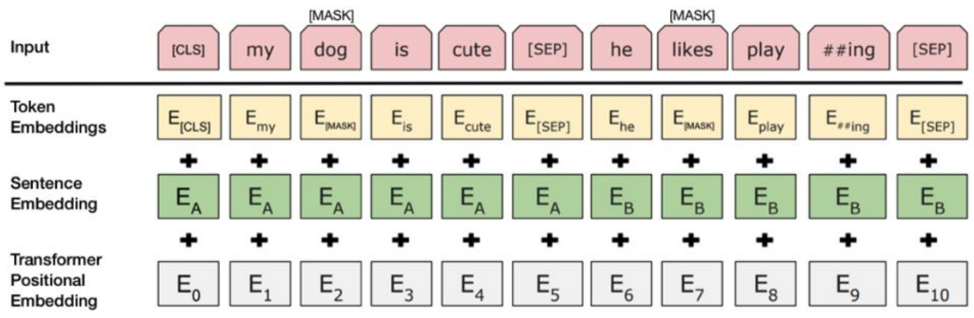
4. Next Sentence Prediction (NSP)

If we look back up at the input transformations the OpenAI transformer does to handle different tasks, you'll notice that some tasks require the model to say something intelligent about two sentences (e.g. are they simply paraphrased versions of each other? Given a Wikipedia entry as input, and a question regarding that entry as another input, can we answer that question?).

To make BERT better at handling relationships between multiple sentences, the pre-training process includes an additional task: Given two sentences (A and B), is B likely to be the sentence that follows A, or not?

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 2.
- 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.


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

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. (input vector size is 768 for each word in BERT base) (**why 768?** the answer is given below)
- Each position outputs a vector of size hidden_size (768 in BERT Base).
- 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 (IsNext and NotNext labels) with softmax.

For example,

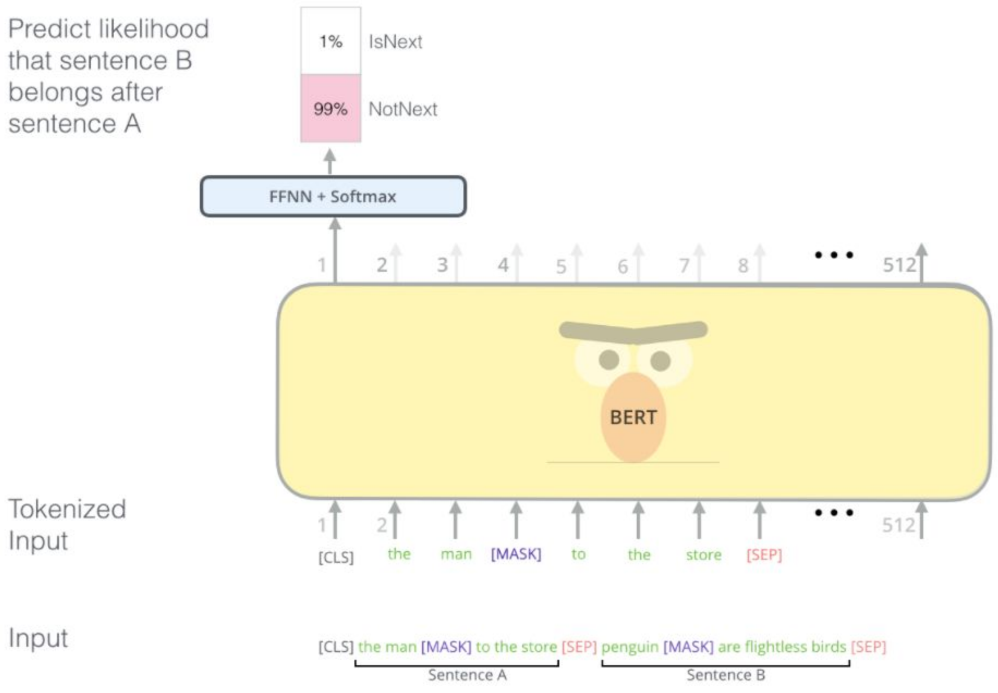
Input = [CLS] the man went to [MASK] store [SEP] he bought a gallon [MASK] milk [SEP]

Label = IsNext

Input = [CLS] the man [MASK] to the store [SEP] penguin [MASK] are flight ##less birds [SEP]

Label = NotNext

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. Now, the whole BERT model in this looks like


Predict likelihood that sentence B belongs after sentence A
1%
IsNext
99%
NotNext
FFNN + Softmax
1
2
3
4
5
6
7
8
...
512
BERT
Tokenized Input
[CLS]
the
man
[MASK]
to
the
store
[SEP]
Input
[CLS] the man [MASK] to the store [SEP] penguin [MASK] are flightless birds [SEP]
Sentence A
Sentence B

Here is a small question: Why does the **input and the output of a single word hold a 764 dimension vector in the BERT_base model?**

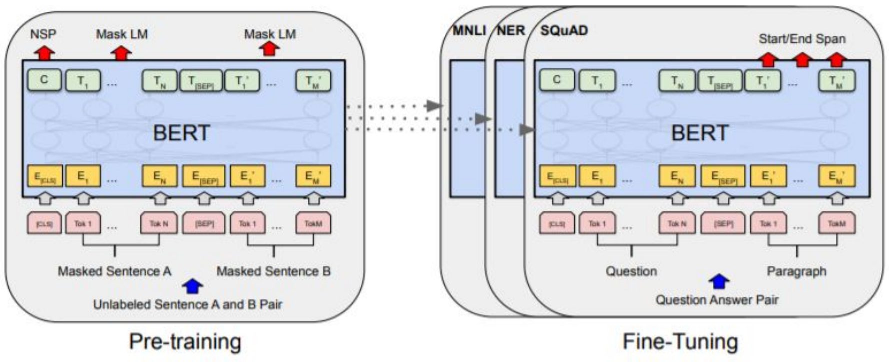
In the BERT_base model, it uses 12 attention heads for multi-head attention. So, while we create multi-head attention, we are using 12 heads for a single word. Each head contains 64 dimension key, query, and value vectors by which we get 64 dimension vectors with a self-attention score for a single word.

```
So, the input and output vector of a single token/word will be

             12 X 64 = 768  dimension embedding
```

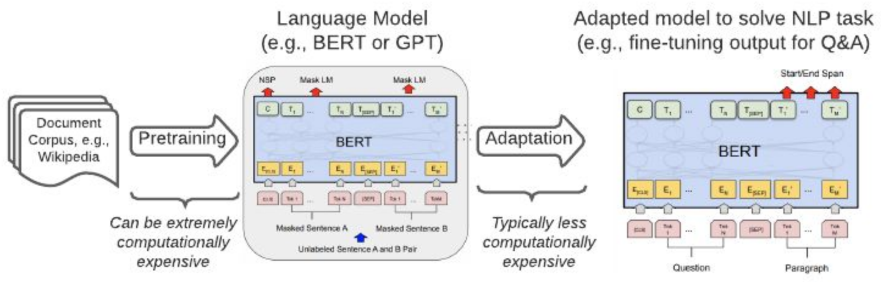
BERT as Transfer Learning in NLP

As BERT takes unlabeled sentence pairs and MLM understands the context of the language better so BERT can be trained on a huge dataset which can be used as a pre-trained model in the downstream tasks. The pre-training procedure largely follows the existing literature on language model pre-training. For the pre-training corpus, BERT uses the BooksCorpus (800M words) and English Wikipedia (2,500M words). For Wikipedia authors extract only the text passages and ignore lists, tables, and headers. It is critical to use a document-level corpus rather than a shuffled sentence-level corpus such as the Billion Word Benchmark in order to extract long contiguous sequences.

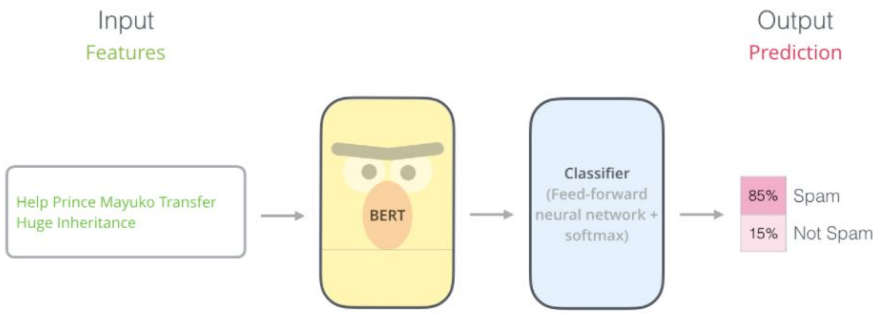


Overall pre-training and fine-tuning procedures for BERT given in the above picture. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, 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). Though BERT is trained on unsupervised data, BERT can be fine-tuned on the supervised task.

The whole picture if we use BERT as a pre-trained model in a downstream supervised task (let say question answering task) will look like this:


Language Model
(e.g., BERT or GPT)
Adapted model to solve NLP task
(e.g., fine-tuning output for Q&A)
Document Corpus, e.g., Wikipedia
Pretraining
Adaptation
BERT
Can be extremely computationally expensive
Typically less computationally expensive
Masked Sentence A
Masked Sentence B
Unlabeled Sentence A and B Pair
Start/End Span
Question
Paragraph

If we want to use BERT for text classification the transfer learning process for BERT will be like

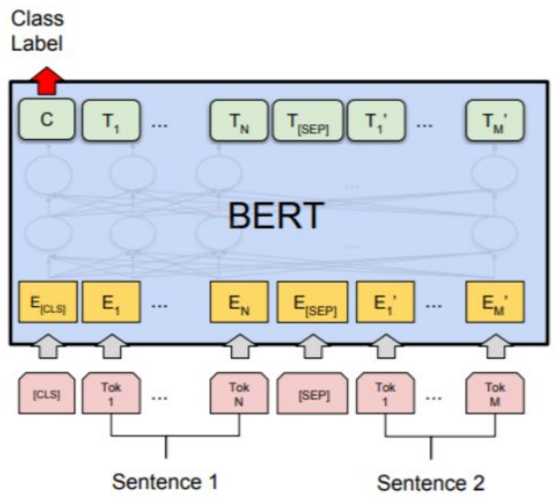

Input
Features
Output
Prediction
Help Prince Mayuko Transfer Huge Inheritance
BERT
Classifier (Feed-forward neural network + softmax)
85% Spam
15% Not Spam

BERT for Different NLP tasks

We've already told that transformers directly can not be used for different tasks but BERT can. How can we use BERT for different NLP tasks? Some of them described below

A. Sentence Pair Classification task
In sentence-pair classification, each example in a dataset has two sentences along with the appropriate target variable. E.g. Sentence similarity, entailment, etc. Sentence pairs are supported in all classification subtasks. There are many datasets like MNLI, QQP, SWAG, etc on the sentence pair classification task. We can fine-tune BERT in this task
To solve this task, BERT takes input of two sentences (Sentence 1 and Sentence 2). Both sentences are separated by a [SEP] token and before those sentences, a [CLS] token is inserted. Now, if we pass those sentences to the pre-trained BERT, the [CLS] token gives a

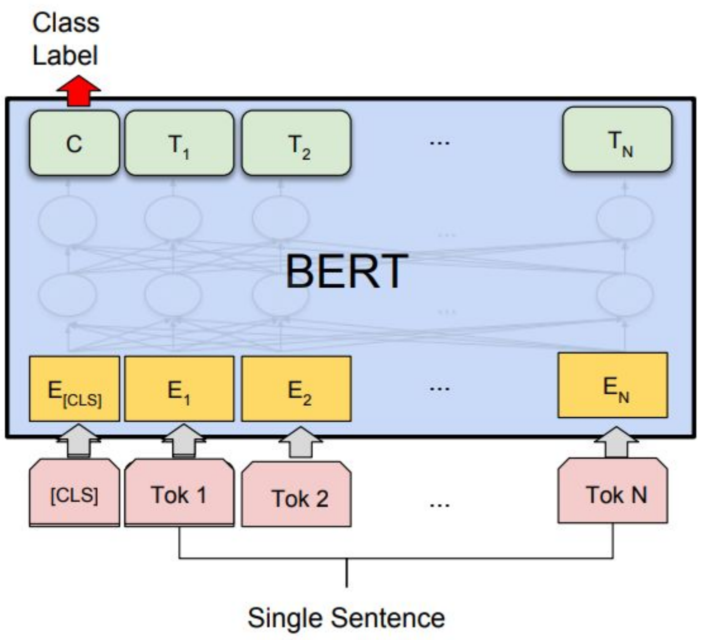
probability by classifying those sentences. As we are given the label output for the input, determining the loss by the label output we can train our sentence pair classification model using BERT.



B. Single Sentence Classification task

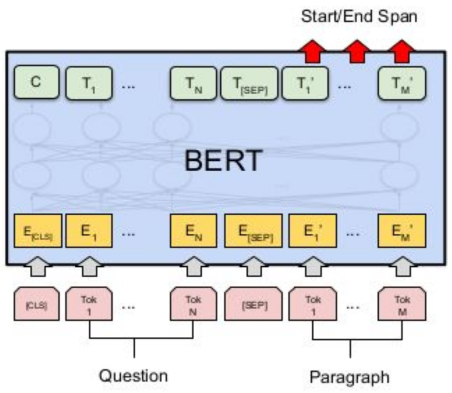
In the single sentence classification task, we are given a sentence and asked to classify the sentence. For example, if we are given the body text of an EMail, then we need to predict whether the EMail is spam or not. SST-2, CoLA are some datasets for this type of task.

Now, to use BERT in this type of task, we pass a single sentence to the BERT input. In the beginning of input (before the sentence a [CLS] is inserted. Now, if we pass the sentence to the pre-trained BERT, the [CLS] token gives a probability by classifying the sentences. As we are given the label output for the input, determining the loss by the label output we can train our single sentence classification model using BERT.

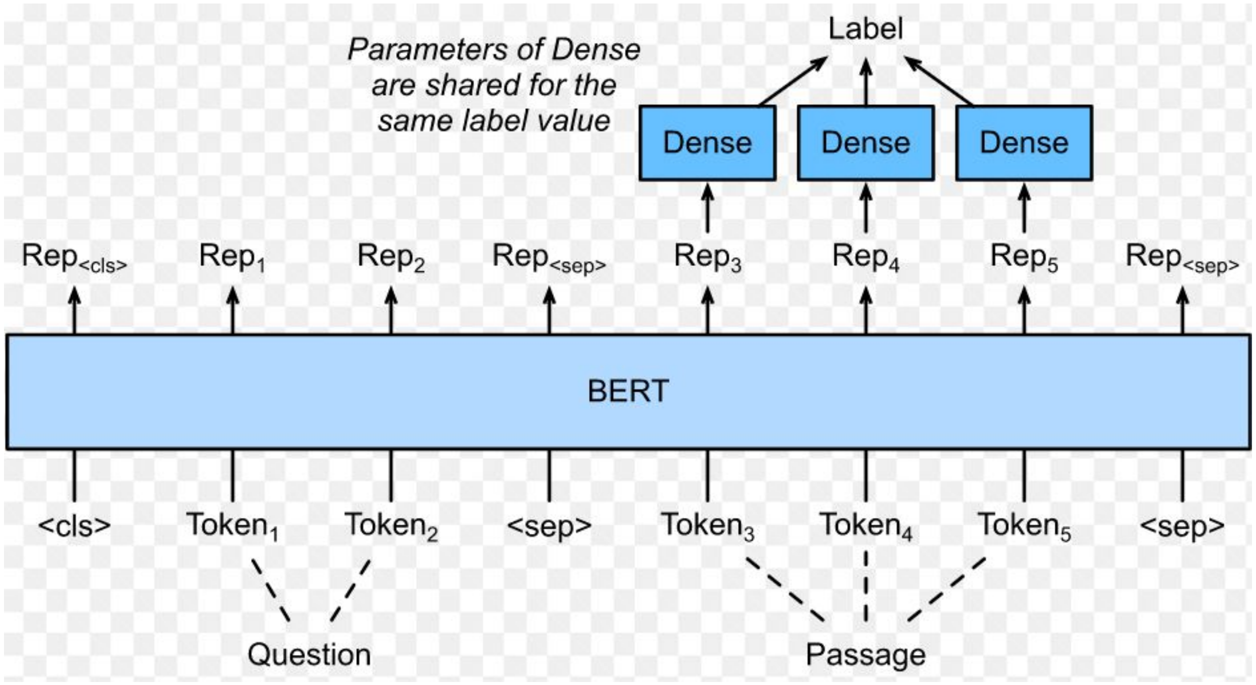


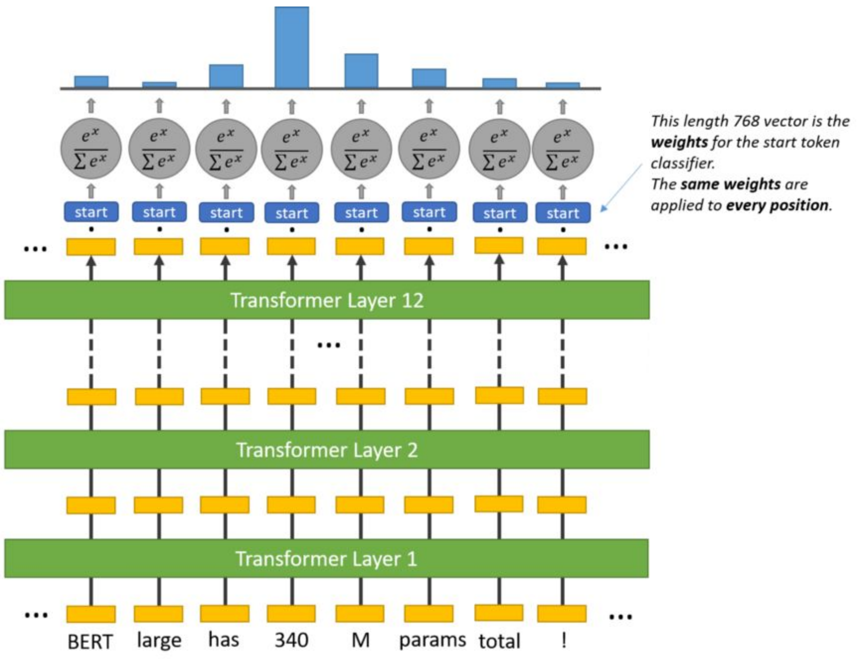
C. Question Answering Task

For the Question Answering task, BERT takes the input question and passage as a single packed sequence. The input embeddings are the sum of the token embeddings and the segment embeddings.



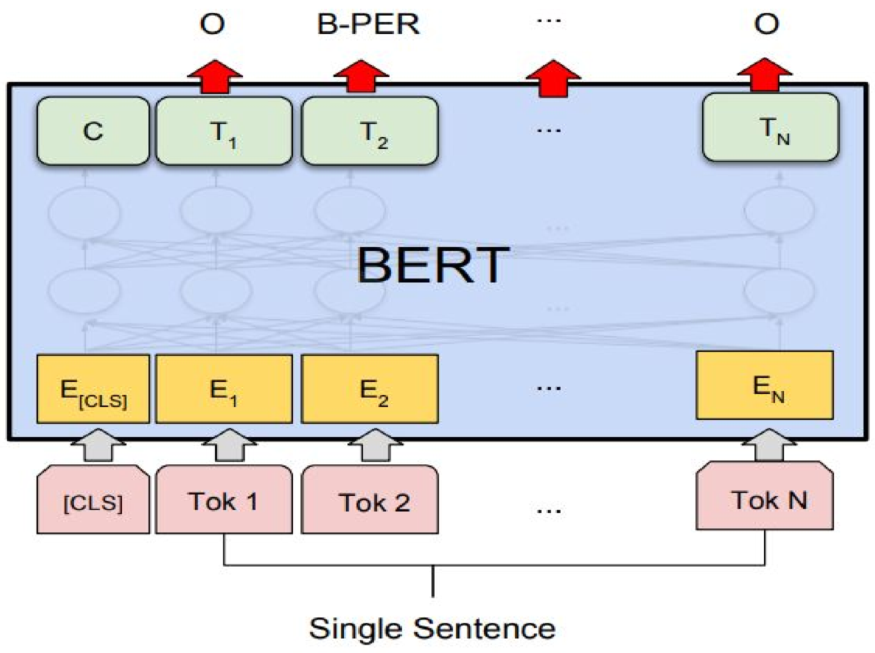
To fine-tune BERT for a Question-Answering system, it introduces a start vector and an end vector. The probability of each word being the start-word is calculated by taking a dot product between the final embedding of the word and the start vector, followed by a softmax over all the words. The word with the highest probability value is considered. A similar process is followed to find the end-word.

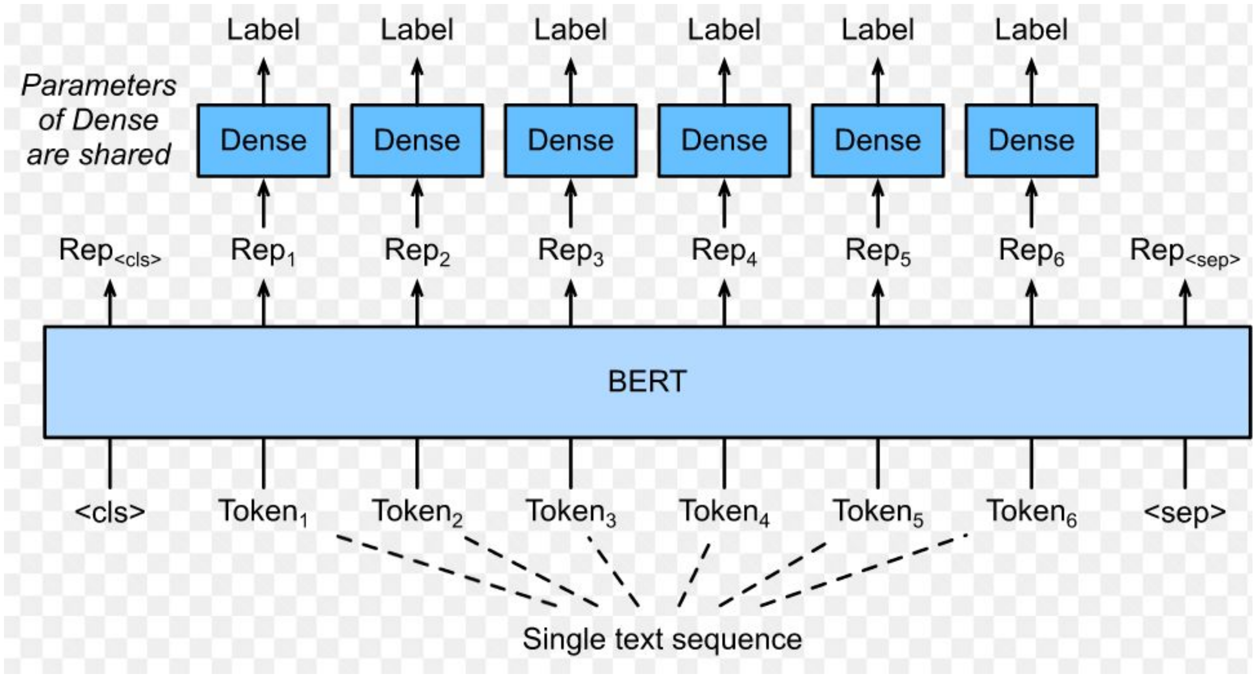



This length 768 vector is the weights for the start token classifier.
The same weights are applied to every position.
start
Transformer Layer 12
Transformer Layer 2
Transformer Layer 1
BERT
large
has
340
M
params
total
!

D. Single Sentence Tagging Task

In single sentence tagging tasks such as named entity recognition (where we are given a sentence and we want to find the name of any person/anything from the sentence), a tag must be predicted for every word in the input. The final hidden states (the transformer output) of every input token is fed to the classification layer to get a prediction for every token. Since WordPieces tokenizer breaks some words into sub-words, the prediction of only the first token of a word is considered.





Applications

We have seen the magic of BERT. This is so robust a model that till now it is the most favorite model and also first choice model to solve any NLP task.BERT is undoubtedly a breakthrough in the use of Machine Learning for Natural Language Processing. The fact that it’s approachable and allows fast fine-tuning will likely allow a wide range of practical applications in the future. In this summary, we attempted to describe the main ideas of the paper while not drowning in excessive technical details.

A different variation of BERT is now using in different real-life projects. Models trained on domain/application-specific corpus are Pre-trained models. Training on domain-specific corpus has shown to yield better performance when fine-tuning them on downstream NLP tasks like NER etc. for those domains, in comparison to fine tuning BERT. Some of the variations are listed below that are using different real-world NLP problem

- *RoBERta* (robustly optimized BERT for solving different tasks)
- *BioBERT* (use for biomedical text)
- *SciBERT* (use for scientific publications)
- *ClinicalBERT* (use for clinical notes)
- *G-BERT* (use for medical/diagnostic code representation and recommendation)
- *M-BERT from 104 languages for zero-shot cross-lingual model transfer* (task-specific annotations in one language is used to fine-tune a model for evaluation in another language)
- *ERNIE* (knowledge graph) + ERNIE (2) incorporates knowledge into pre-training but by masking entities and phrases using KG.
- *TransBERT* — unsupervised, followed by two supervised steps, for a story ending prediction task
- *videoBERT* (a model that jointly learns video and language representation learning) by representing video frames as special descriptor tokens along with text for pretraining. This is used for video captioning.
- *DocBERT* (use for Document classification)
- *PatentBERT* (Patent classification)

References

- BERT_Paper - https://arxiv.org/pdf/1810.04805.pdf
- OpenAI GPT2 - https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf
- ULMFit paper - https://arxiv.org/abs/1801.06146
- ELMo paper - https://arxiv.org/abs/1802.05365
- Transformer - https://arxiv.org/abs/1706.03762
- Bidirectional RNN/LSTM - https://ieeexplore.ieee.org/document/650093
- WordPieces Embedding - https://arxiv.org/abs/1609.08144