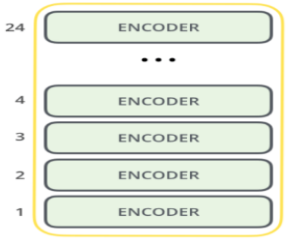
BIDIRECTIONAL ENCODER REPRESENTATIONS FROM TRANSFORMERS (BERT)

What is BERT?

BERT stands for “Bidirectional Encoder Representation with Transformers”. To put it in simple words BERT extracts patterns or representations from the data or word embeddings by passing it through an encoder. The encoder itself is a transformer architecture that is stacked together. It is a bidirectional transformer which means that during training it considers the context from both the left and right of the vocabulary to extract patterns or representations.



BERT uses two training paradigms: **Pre-training** and **Fine-tuning**.

During **pre-training,** the model is trained on a large dataset to extract patterns. This is generally an **unsupervised learning** task where the model is trained on an unlabelled dataset like the data from a big corpus like Wikipedia.

During **fine-tuning** the model is trained for downstream tasks like Classification, Text-Generation, Language Translation, Question-Answering, and so forth. Essentially, you can download a pre-trained model and then Transfer-learn the model on your data.

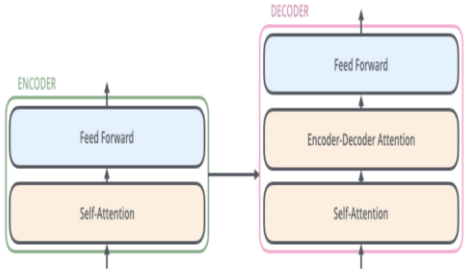
CORE COMPONENTS OF BERT

BERT borrows ideas from the previous release SOTA models. Let’s elaborate on that statement.

THE TRANSFORMERS

BERT’s main component is the transformer architecture. The transformers are made up of two components: **encoder** and **decoder**. The encoder itself contains two components: the **self-attention layer** and **feed-forward neural network**.

The self-attention layer takes an input and encodes each word into intermediate encoded representations which are then passed through the feed-forward neural network. The feed-forward network passes those representations to the decoder that itself is made up of three components: **self-attention layer, Encoder-Decoder Attention,** and **feed-forward neural network**.



The benefit of the transformer architecture is that it helps the model to retain infinitely long sequences that were not possible from the traditional RNNs, LSTMs, and GRU. But even from the fact that it can achieve long-term dependencies it still **lacks contextual understanding**.

ELMo

BERT borrows another idea from ELMo which stands for Embeddings from Language Model. ELMo was introduced by Peters et. al. in 2017 which dealt with the idea of contextual understanding. The way ELMo works is that it uses **bidirectional** LSTM to make sense of the context. Since it considers words from both directions, it can assign different word embedding to words that are spelled similarly but have different meanings.

So, ELMo assigns embeddings by considering the words from both the right and left directions as compared to the models that were developed previously which took into consideration words, only from the left. These models were unidirectional like RNNs, LSTMs et cetera.

This enables ELMo to capture contextual information from the sequences but since ELMo uses LTSM it does not have long-term dependency compared to transformers.

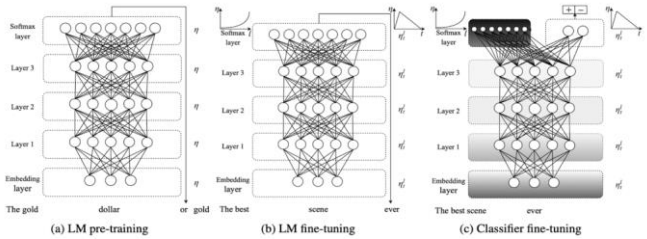
ULM-FiT

In 2018 Jeremy Howard and Sebastian Ruder released a paper called Universal Language Model Fine-tuning or ULM-FiT, where they argued that transfer learning can be used in NLP just like it is used in computer vision.

Previously we were using pre-trained models for word-embeddings that only targeted the first layer of the entire model, i.e. the embedding layers, and the whole model was trained from the scratch, this was time-consuming, and not a lot of success was found in this area. However, Howard and Ruder proposed 3 methods for the classification of text:

- The first step includes training the model on a larger dataset so that the model learns representations.
- The second step included fine-tuning the model with a task-specific dataset for classification, during which they introduced two more methods: Discriminative fine-tuning and Slanted triangular learning rates (STLR). The former method tries to fine-tune or optimize the parameters for each during the transfer layer in the network while the latter controls the learning rate in each of the optimization steps.

- The third step was to fine-tune the classifier on the task-specific dataset for classification.



With the release of ULM-FiT NLP practitioners can now practice the transfer learning approach in their NLP problems. But the only problem with the ULM-FiT approach to transfer learning was that it included fine-tuning all the layers in the network which was a lot of work.

OpenAI GPT

Generative Pre-trained Transformer or GPT was introduced by OpenAI's team: Radford, Narasimhan, Salimans, and Sutskever. They presented a model that only uses decoders from the transformer instead of encoders in a unidirectional approach. As a result, it outperformed all the previous models in various tasks like:

- Classification
- Natural Language Inference
- Semantic similarity
- Question answering
- Multiple Choice.

Even though the GPT used only the decoder, it could still retain long-term dependencies. Furthermore, it reduced fine-tuning to a minimum compared to what we saw in ULM-FiT.

Why BERT?

BERT falls into a self-supervised model. That means, it can generate inputs and labels from the raw corpus without being explicitly programmed by humans. Remember the data it is trained on is unstructured.

BERT was pre-trained with two specific tasks: Masked Language Model and Next sentence prediction. The former uses masked input like "the man [MASK] to the store" instead of "the man went to the store". This restricts BERT to see the words next to it which allows it to learn bidirectional representations as much as possible making it much more flexible and reliable for several downstream tasks. The latter predicts whether the two sentences are contextually assigned to each other.

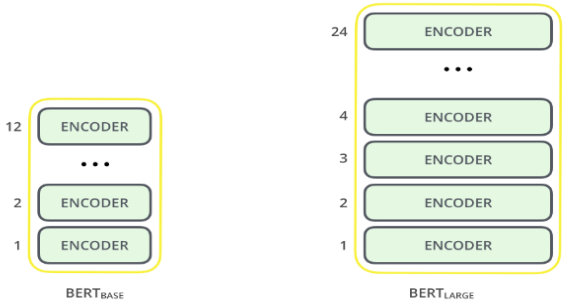
BERT can also be used for feature extraction because of the properties we discussed previously and feed these extractions to your existing model.

How Does BERT Work?

1. BERT's Architecture

The BERT architecture builds on top of Transformer. We currently have two variants available:

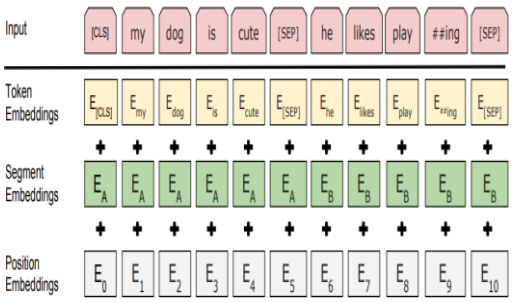
- BERT Base: 12 layers (transformer blocks), 12 attention heads, and 110 million parameters
- BERT Large: 24 layers (transformer blocks), 16 attention heads and, 340 million parameters



The BERT Base architecture has the same model size as OpenAI's GPT for comparison purposes. All of these Transformer layers are **Encoder**-only blocks.

Now that we know the overall architecture of BERT, let's see what kind of text processing steps are required before we get to the model building phase.

2. Text Preprocessing



The developers behind BERT have added a specific set of rules to represent the input text for the model. Many of these are creative design choices that make the model even better.

For starters, every input embedding is a combination of 3 embeddings:

1. **Position Embeddings**: BERT learns and uses positional embeddings to express the position of words in a sentence. These are added to overcome the limitation of Transformer which, unlike an RNN, is not able to capture “sequence” or “order” information
2. **Segment Embeddings**: BERT can also take sentence pairs as inputs for tasks (Question-Answering). That’s why it learns a unique embedding for the first and the second sentences to help the model distinguish between them. In the above example, all the tokens marked as EA belong to sentence A (and similarly for EB)
3. **Token Embeddings**: These are the embeddings learned for the specific token from the WordPiece token vocabulary

For a given token, its input representation is constructed by summing the corresponding token, segment, and position embeddings.

Such a comprehensive embedding scheme contains a lot of useful information for the model.

These combinations of preprocessing steps make BERT so versatile. This implies that without making any major change in the model’s architecture, we can easily train it on multiple kinds of NLP tasks.

3. Pre-training Tasks

BERT is pre-trained on two NLP tasks:

- Masked Language Modeling
- Next Sentence Prediction

Let’s understand both of these tasks in a little more detail!

a. Masked Language Modeling (Bi-directionality)

Need for Bi-directionality

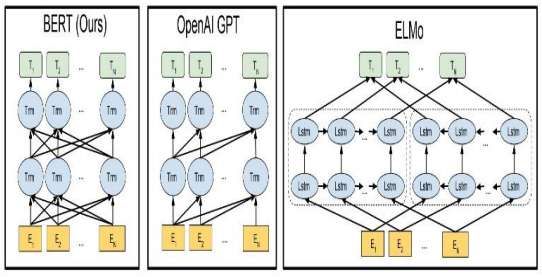
BERT is designed as a ***deeply bidirectional*** model. The network effectively captures information from both the right and left context of a token from the first layer itself and all the way through to the last layer.

Traditionally, we had language models either trained to predict the next word in a sentence (right-to-left context used in GPT) or language models that were trained on a left-to-right context. This made our models susceptible to errors due to loss in information.

ELMo tried to deal with this problem by training two LSTM language models on left-to-right and right-to-left contexts and shallowly concatenating them. Even though it greatly improved upon existing techniques, it wasn't enough.

"Intuitively, it is reasonable to believe that a deep bidirectional model is strictly more powerful than either a left-to-right model or the shallow concatenation of a left-to-right and a right-to-left model." – BERT

That's where BERT greatly improves upon both GPT and ELMo. Look at the below image:



The arrows indicate the information flow from one layer to the next. The green boxes at the top indicate the final contextualized representation of each input word.

It's evident from the above image: BERT is bi-directional, GPT is unidirectional (information flows only from left-to-right), and ELMO is shallowly bidirectional.

This is where the **Masked Language Model** comes into the picture.

About Masked Language Models

Let's say we have a sentence – "I love to read data science blogs on Analytics Vidhya". We want to train a bi-directional language model. Instead of trying to predict the next word in the sequence, we can build a model to predict a missing word from within the sequence itself.

Let's replace "Analytics" with "[MASK]". This is a token to denote that the token is missing. We'll then train the model in such a way that it should be able to predict "Analytics" as the missing token: "I love to read data science blogs on [MASK] Vidhya."

This is the crux of a Masked Language Model. The authors of BERT also include some caveats to further improve this technique:

- To prevent the model from focusing too much on a particular position or tokens that are masked, the researchers randomly masked 15% of the words
- The masked words were not always replaced by the masked tokens [MASK] because the [MASK] token would never appear during fine-tuning
    - So, the researchers used the below technique:
    - 80% of the time the words were replaced with the masked token [MASK]
    - 10% of the time the words were replaced with random words
    - 10% of the time the words were left unchanged

b. Next Sentence Prediction

Masked Language Models (MLMs) learn to understand the relationship between words. Additionally, **BERT is also trained on the task of Next Sentence Prediction for tasks that require an understanding of the relationship between sentences.**

A good example of such a task would be question answering systems.

The task is simple. Given two sentences – A and B, is B the actual next sentence that comes after A in the corpus, or just a random sentence?

Since it is a binary classification task, the data can be easily generated from any corpus by splitting it into sentence pairs. Just like MLMs, the authors have added some caveats here too. Let's take this with an example:

Consider that we have a text dataset of 100,000 sentences. So, there will be 50,000 training examples or pairs of sentences as the training data.

- For 50% of the pairs, the second sentence would actually be the next sentence to the first sentence
- For the remaining 50% of the pairs, the second sentence would be a random sentence from the corpus
- The labels for the first case would be ***'IsNext'*** and ***'NotNext'*** for the second case

And this is how BERT is able to become a true task-agnostic model. It combines both the Masked Language Model (MLM) and the Next Sentence Prediction (NSP) pre-training tasks.

Implementing BERT for Text Classification in Python

Your mind must be whirling with the possibilities BERT has opened up. There are many ways we can take advantage of BERT's large repository of knowledge for our NLP applications.

One of the most potent ways would be fine-tuning it on your own task and task-specific data. We can then use the embeddings from BERT as embeddings for our text documents.

In this section, we will learn how to use BERT's embeddings for our NLP task. We'll take up the concept of fine-tuning an entire BERT model in one of the future articles.

Installing BERT-As-Service

BERT-As-Service works in a simple way. It creates a BERT server which we can access using the Python code in our notebook. Every time we send it a sentence as a list, it will send the embeddings for all the sentences.

We can install the server and client via pip. They can be installed separately or even on *different* machines:

```
pip install bert-serving-server  # server
pip install bert-serving-client  # client, independent of `bert-serving-server`
```

We'll download BERT Uncased and then decompress the zip file:

```
wget https://storage.googleapis.com/bert_models/2018_10_18/uncased_L-12_H-768_A-12.zip && unzip uncased_L-12_H-768_A-12.zip
```

Once we have all the files extracted in a folder, it's time to start the BERT service:

```
bert-serving-start -model_dir uncased_L-12_H-768_A-12/ -num_worker=2 -max_seq_len 50
```

You can now simply call the BERT-As-Service from your Python code (using the client library). Let's just jump into code!

Open a new Jupyter notebook and try to fetch embeddings for the sentence: "I love data science and analytics vidhya".

```
from bert_serving.client import BertClient


# make a connection with the BERT server using it's ip address; do not give any ip if same computer
bc = BertClient(ip="SERVER_IP_HERE")
# get the embedding
embedding = bc.encode(["I love data science and analytics vidhya."])
# check the shape of embedding, it should be 1x768
print(embedding.shape)
```

Here, the IP address is the IP of your server or cloud. *This field is not required if used on the same computer.*

The shape of the returned embedding would be (1,768) as there is only a single sentence which is represented by 768 hidden units in BERT's architecture.

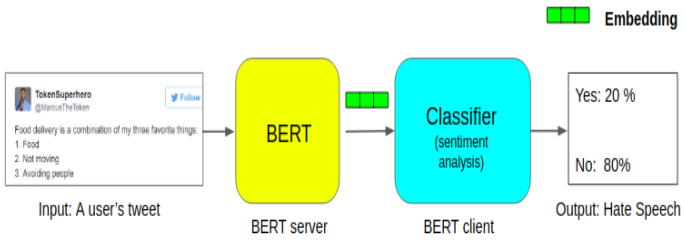
Problem Statement: Classifying Hate Speech on Twitter

Let's take up a real-world dataset and see how effective BERT is. We'll be working with a dataset consisting of a collection of tweets that are classified as being "hate speech" or not.

For the sake of simplicity, we say a tweet contains hate speech if it has a racist or sexist sentiment associated with it. **So, the task is to classify racist or sexist tweets from other tweets.**

We will use BERT to extract embeddings from each tweet in the dataset and then use these embeddings to train a text classification model.

Here is how the overall structure of the project looks like:



Splitting dataset into training and validation set:

```
from sklearn.model_selection import train_test_split

# split into training and validation sets
X_tr, X_val, y_tr, y_val = train_test_split(train.clean_text, train.label, test_size=0.25,
random_state=42)

print('X_tr shape:',X_tr.shape)
```

Let’s get the embeddings for all the tweets in the training and validation sets:

```
from bert_serving.client import BertClient

# make a connection with the BERT server using it's ip address
bc = BertClient(ip="YOUR_SERVER_IP")
# get the embedding for train and val sets
X_tr_bert = bc.encode(X_tr.tolist())
X_val_bert = bc.encode(X_val.tolist())
```

It’s model building time! Let’s train the classification model:

```
from sklearn.linear_model import LogisticRegression

# LR model
model_bert = LogisticRegression()
# train
model_bert = model_bert.fit(X_tr_bert, y_tr)
# predict
pred_bert = model_bert.predict(X_val_bert)
```

Check the classification accuracy:

```
from sklearn.metrics import accuracy_score

print(accuracy_score(y_val, pred_bert))
```

Even with such a small dataset, we easily get a classification accuracy of around 95%.