Demystifying BERT: A Comprehensive Guide to the Groundbreaking NLP Framework

Bidirectional Encoder Representations from Transformers

- Google's BERT has transformed the Natural Language Processing (NLP) landscape
- Learn what BERT is, how it works, the seismic impact it has made, among other things
- We'll also implement BERT in Python to give you a hands-on learning experience

Introduction to the World of BERT

Picture this – you're working on a really cool data science project and have applied the latest state-of-the-art library to get a pretty good result. And boom! A few days later, there's a new state-of-the-art framework in town that has the potential to further improve your model.

That is not a hypothetical scenario – it's the reality (and thrill) of working in the field of Natural Language Processing (NLP)! The last two years have been mind-blowing in terms of breakthroughs. I get to grips with one framework and another one, potentially even better, comes along.

Google's BERT is one such NLP framework. I'd stick my neck out and say it's perhaps the most influential one in recent times (and we'll see why pretty soon).

It's not an exaggeration to say that BERT has significantly altered the NLP landscape. Imagine using a single model that is trained on a large unlabelled dataset to achieve State-of-the-Art results on 11 individual NLP tasks. And all of this with little fine-tuning. That's BERT! It's a tectonic shift in how we design NLP models.

BERT has inspired many recent NLP architectures, training approaches and language models, such as Google's TransformerXL, OpenAI's GPT-2, XLNet, ERNIE2.0, RoBERTa, etc.

I aim to give you a comprehensive guide to not only BERT but also what impact it has had and how this is going to affect the future of NLP research. And yes, there's a lot of Python code to work on, too!

Note: In this article, we are going to talk a lot about Transformers. If you aren't familiar with it, feel free to read this article first – How do Transformers Work in NLP? A Guide to the Latest State-of-the-Art Models.

Table of Contents

What is BERT?

You've heard about BERT, you've read about how incredible it is, and how it's potentially changing the NLP landscape. But what is BERT in the first place?

Here's how the research team behind BERT describes the NLP framework:

> "BERT stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. It is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of NLP tasks."

That sounds way too complex as a starting point. But it does summarize what BERT does pretty well so let's break it down.

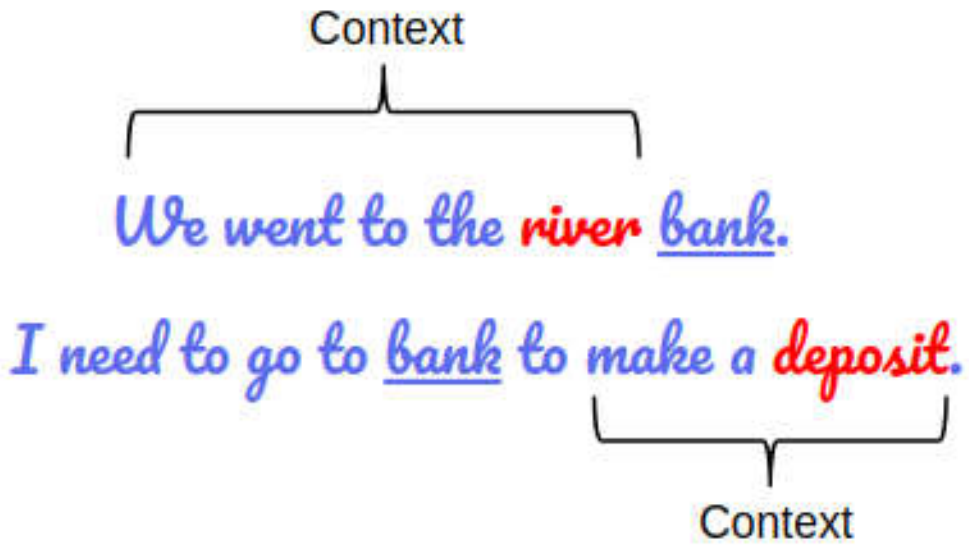
First, it's easy to get that BERT stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. Each word here has a meaning to it and we will encounter that one by one in this article. For now, the key takeaway from this line is – **BERT is based on the Transformer architecture.**

Second, BERT is pre-trained on a large corpus of unlabelled text including the entire Wikipedia(that's 2,500 million words!) and Book Corpus (800 million words).

> This **pre-training** step is half the magic behind BERT's success. This is because as we train a model on a large text corpus, our model starts to pick up the deeper and intimate understandings of how the language works. This knowledge is the **swiss army knife** that is useful for almost any NLP task.

Third, BERT is a **"deeply bidirectional"** model. Bidirectional means that BERT learns information from both the left and the right side of a token's context during the training phase.

The bidirectionality of a model is important for truly understanding the meaning of a language. Let's see an example to illustrate this. There are two sentences in this example and both of them involve the word "bank":



BERT captures both the left and right context

If we try to predict the nature of the word "bank" by only taking either the left or the right context, then we will be making an error in at least one of the two given examples.

One way to deal with this is to consider both the left and the right context before making a prediction. That's exactly what BERT does! We will see later in the article how this is achieved.

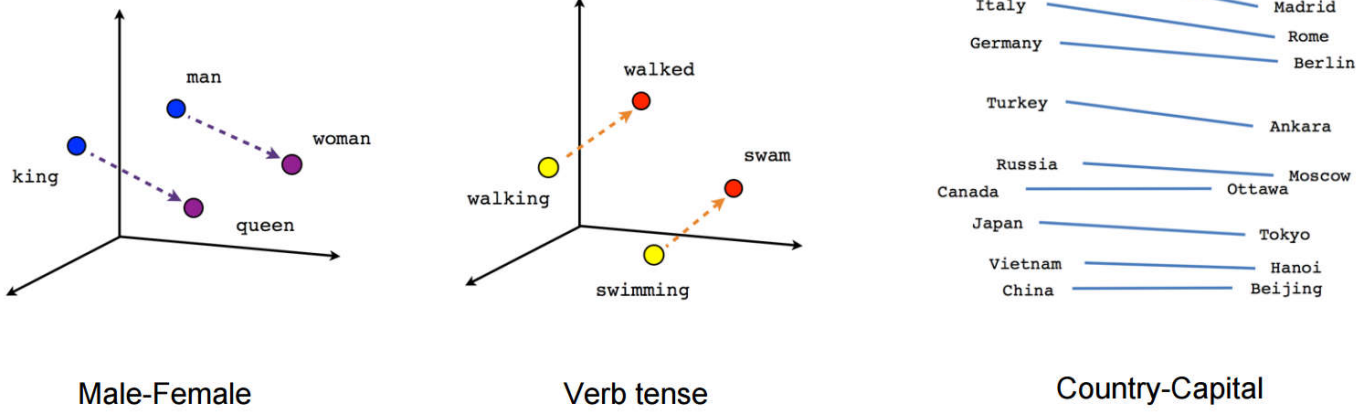
And finally, the most impressive aspect of BERT. We can fine-tune it by adding just a couple of additional output layers to create state-of-the-art models for a variety of NLP tasks.

From Word2Vec to BERT: NLP's Quest for Learning Language Representations

> "One of the biggest challenges in natural language processing is the shortage of training data. Because NLP is a diversified field with many distinct tasks, most task-specific datasets contain only a few thousand or a few hundred thousand human-labelled training examples." – Google AI

Word2Vec and GloVe

The quest for learning language representations by pre-training models on large unlabelled text data started from word embeddings like Word2Vec and GloVe. These embeddings changed the way we performed NLP tasks. We now had embeddings that could capture contextual relationships among words.



Male-Female Verb tense Country-Capital

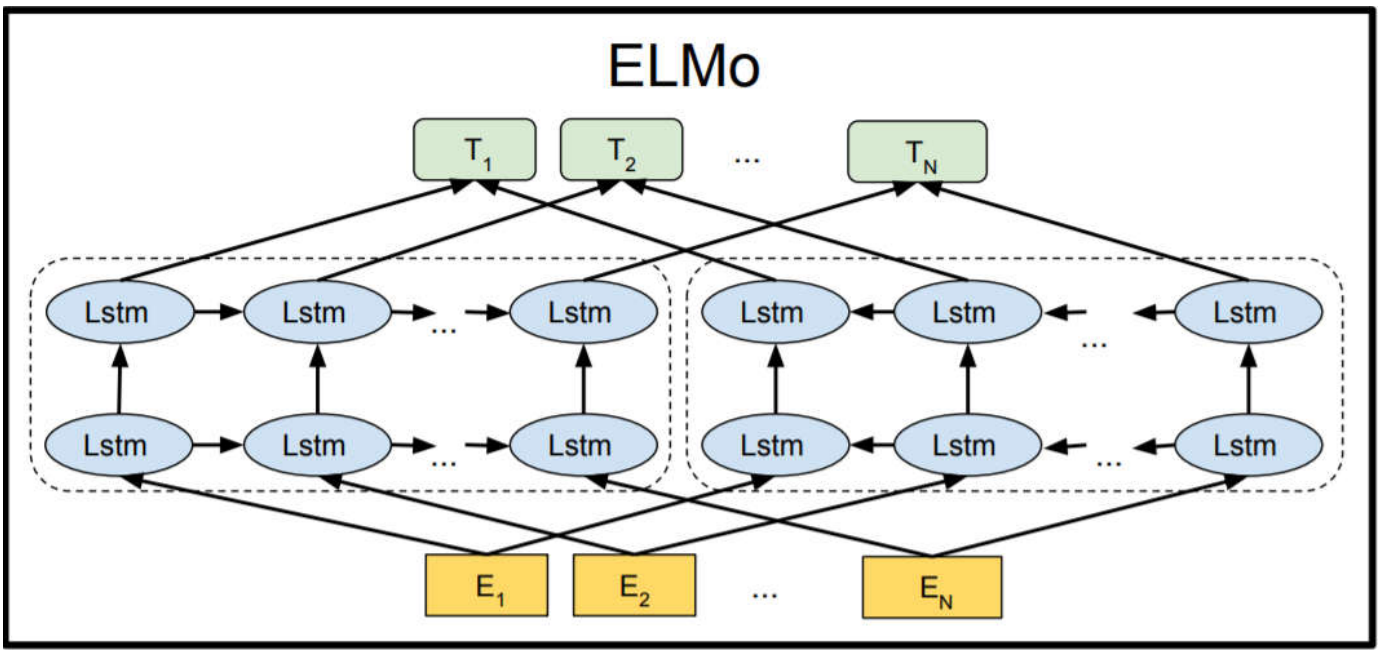
These embeddings were used to train models on downstream NLP tasks and make better predictions. This could be done even with less task-specific data by utilizing the additional information from the embeddings itself.

One limitation of these embeddings was the use of very shallow Language Models. This meant there was a limit to the amount of information they could capture and this motivated the use of deeper and more complex language models (layers of LSTMs and GRUs).

Another key limitation was that these models did not take the context of the word into account. Let's take the above "bank" example. The same word has different meanings in different contexts, right? However, an embedding like Word2Vec will give the same vector for "bank" in both the contexts.

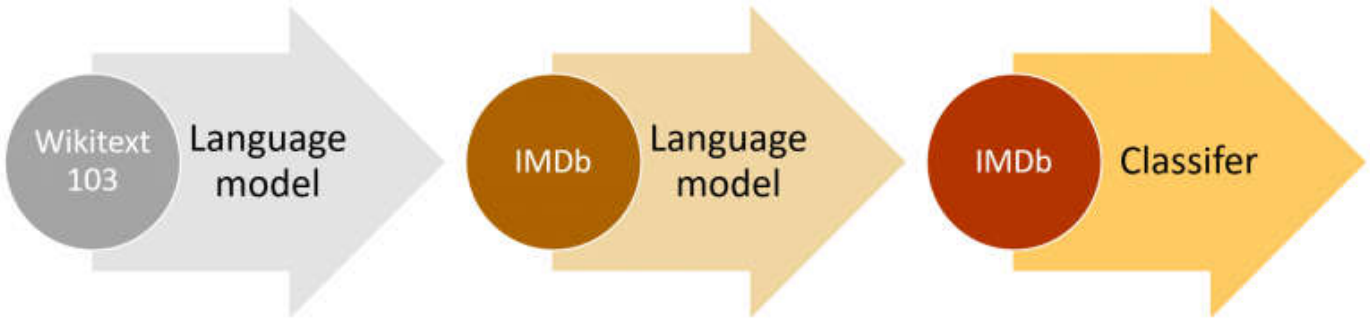
That's valuable information we are losing.

Enter ELMO and ULMFiT



ELMo was the NLP community's response to the problem of **Polysemy** – same words having different meanings based on their context. From training shallow feed-forward networks (Word2vec), we graduated to training word embeddings using layers of complex Bi-directional LSTM architectures. This meant that the same word can have multiple ELMO embeddings based on the context it is in.

That's when we started seeing the advantage of ***pre-training*** as a training mechanism for NLP.



ULMFiT took this a step further. This framework could train language models that could be fine-tuned to provide excellent results even with fewer data (less than 100 examples) on a variety of document classification tasks. It is safe to say that ULMFiT cracked the code to transfer learning in NLP.

This is when we established the golden formula for transfer learning in NLP:

> Transfer Learning in NLP = Pre-Training and Fine-Tuning

Most of the NLP breakthroughs that followed ULMFIT tweaked components of the above equation and gained state-of-the-art benchmarks.

OpenAI's GPT

OpenAI's GPT extended the methods of pre-training and fine-tuning that were introduced by ULMFiT and ELMo. GPT essentially replaced the LSTM-based architecture for Language Modeling with a Transformer-based architecture.

The GPT model could be fine-tuned to multiple NLP tasks beyond document classification, such as common sense reasoning, semantic similarity, and reading comprehension.

GPT also emphasized the importance of the Transformer framework, which has a simpler architecture and can train faster than an LSTM-based model. It is also able to learn complex patterns in the data by using the Attention mechanism.

> OpenAI's GPT validated the robustness and usefulness of the Transformer architecture by achieving multiple State-of-the-Arts.

And this is how Transformer inspired BERT and all the following breakthroughs in NLP.

Now, there were some other crucial breakthroughs and research outcomes that we haven't mentioned yet, such as semi-supervised sequence learning. This is because they are slightly out of the scope of this article but feel free to read the linked paper to know more about it.

Moving onto BERT

So, the new approach to solving NLP tasks became a 2-step process:

1. Train a language model on a large unlabelled text corpus (unsupervised or semi-supervised)

2. Fine-tune this large model to specific NLP tasks to utilize the large repository of knowledge this model has gained (supervised)

With that context, let's understand how BERT takes over from here to build a model that will become a benchmark of excellence in NLP for a long time.

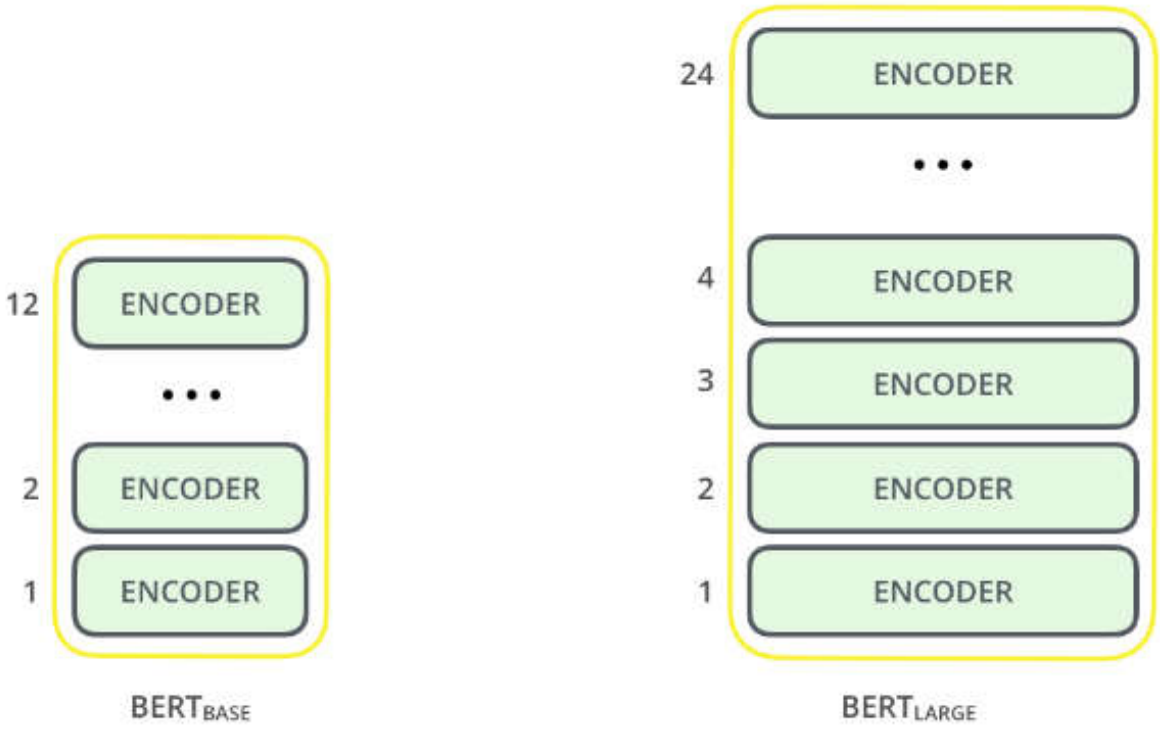
How Does BERT Work? A Look Under the Hood

Let's look a bit closely at BERT and understand why it is such an effective method to model language. We've already seen what BERT can do earlier – but how does it do it? We'll answer this pertinent question in this section.

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



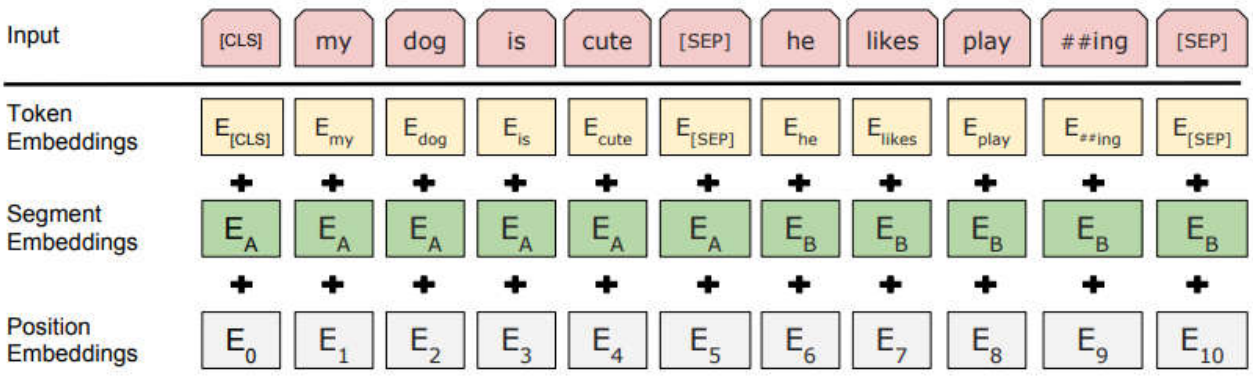
Source

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.

If your understanding of the underlying architecture of the Transformer is hazy, I will recommend that you read about it here.

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.

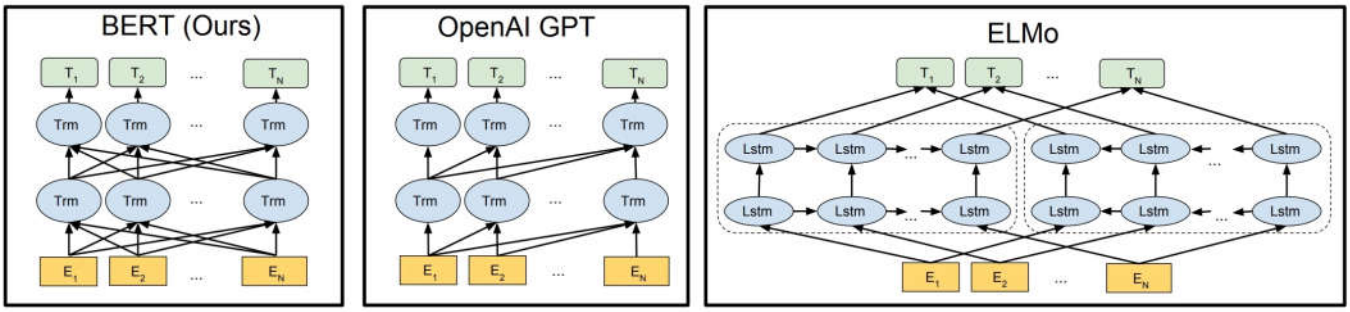
Traditionally, we had language models either trained to predict the next word in a sentence (right-to-left context used in GPT) or language models that were trained on a left-to-right context. This made our models susceptible to errors due to loss in information.

Predicting the word in a sequence

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

I have shown how to implement a Masked Language Model in Python in one of my previous articles here:

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.

For extracting embeddings from BERT, we will use a really useful open source project called Bert-as-Service:

Running BERT can be a painstaking process since it requires a lot of code and installing multiple packages. That's why this open-source project is so helpful because it lets us **use BERT to extract encodings for each sentence in just two lines of code.**

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-ser
```

Note that the server MUST be running on **Python >= 3.5** with **TensorFlow >= 1.10** (*one-point-ten*).

Also, since running BERT is a GPU intensive task, I'd suggest installing the bert-serving-server on a cloud-based GPU or some other machine that has high compute capacity.

Now, go back to your terminal and download a model listed below. Then, uncompress the zip file into some folder, say */tmp/english_L-12_H-768_A-12/*.

Here's a list of the released pre-trained BERT models:

BERT-Base, Uncased	12-layer, 768-hidden, 12-heads, 110M parameters
BERT-Large, Uncased	24-layer, 1024-hidden, 16-heads, 340M parameters
BERT-Base, Cased	12-layer, 768-hidden, 12-heads, 110M parameters
BERT-Large, Cased	24-layer, 1024-hidden, 16-heads, 340M parameters
BERT-Base, Multilingual Cased (New)	104 languages, 12-layer, 768-hidden, 12-heads, 110M parameters
BERT-Base, Multilingual Cased (Old)	102 languages, 12-layer, 768-hidden, 12-heads, 110M parameters
BERT-Base, Chinese	Chinese Simplified and Traditional, 12-layer, 768-hidden, 12-heads, 110M parameters

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

```
wget https://storage.googleapis.com/bert_models/2018_10_18/uncased
```

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
```

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".

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

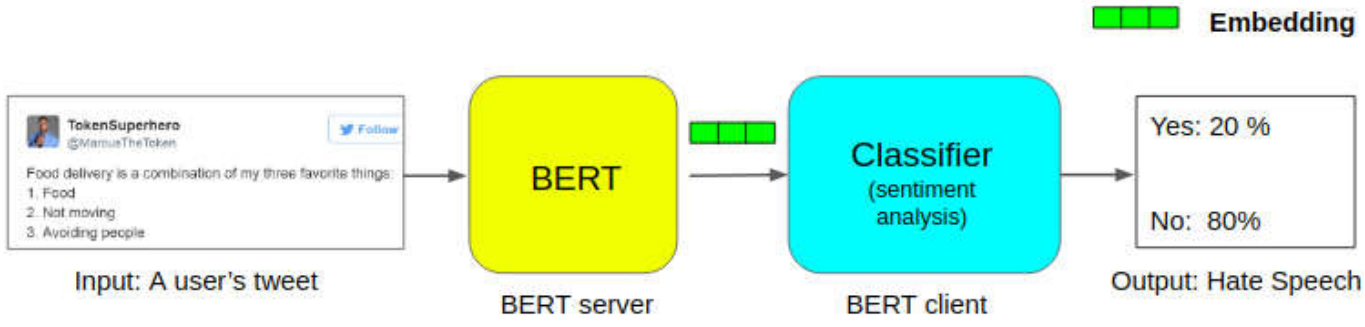
Let's take up a real-world dataset and see how effective BERT is. We'll be working with a dataset consisting of a collection of tweets that are classified as being "hate speech" or not.

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.**

You can download the dataset and read more about the problem statement on the DataHack platform.

> 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:



Let's look at the code now:

You'll be familiar with how most people tweet. There are many random symbols and numbers (aka chat language!). Our dataset is no different. We need to preprocess it before passing it through BERT:

Now that the dataset is clean, it's time to split it into training and validation set:

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

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

Check the classification accuracy:

Even with such a small dataset, we easily get a classification accuracy of around 95%. That's damn impressive.

I encourage you to go ahead and try BERT's embeddings on different problems and share your results in the comments below.

In the next article, I plan to take a BERT model and fine-tune it fully on a new dataset and compare its performance.

Beyond BERT: Current State-of-the-Art in NLP

BERT has inspired great interest in the field of NLP, especially the application of the Transformer for NLP tasks. This has led to a spurt in the number of research labs and organizations that started experimenting with different aspects of pre-training, transformers and fine-tuning.

Many of these projects outperformed BERT on multiple NLP tasks. Some of the most interesting developments were RoBERTa, which was Facebook AI's

improvement over BERT and DistilBERT, which is a compact and faster version of BERT.

You can read more about these amazing developments regarding State-of-the-Art NLP in this article.

Related Articles

---

https://www.analyticsvidhya.com/blog/2019/09/demystifying-bert-groundbreaking-nlp-framework/