BERT Explained: A Complete Guide with Theory and Tutorial

Unless you have been out of touch with the Deep Learning world, chances are that you have heard about BERT — it has been the talk of the town for the last one year.

At the end of 2018 researchers at Google AI Language open-sourced a new technique for Natural Language Processing (NLP) called **BERT** (Bidirectional Encoder Representations from Transformers) — a major breakthrough which took the Deep Learning community by storm because of its incredible performance. Since BERT is likely to stay around for quite some time, in this blog post, we are going to understand it by attempting to answer these 5 questions:

1. ***Why was BERT needed?***
2. ***What is the core idea behind it?***
3. ***How does it work?***
4. ***When can we use it and how to fine-tune it?***
5. ***How can we use it? Using BERT for Text Classification — Tutorial***

In the first part of this post, we are going to go through the theoretical aspects of BERT, while in the second part we are going to get our hands dirty with a practical example.

Part I

1. Why was BERT needed?

One of the biggest challenges in NLP is the lack of enough training data. Overall there is enormous amount of text data available, but if we want to create task-specific datasets, we need to split that pile into the very many diverse fields. And when we do this, we end up with only a few thousand or a few hundred thousand human-labeled training examples. Unfortunately, in order to perform well, deep learning based NLP models require much larger amounts of data — they see major improvements when trained on millions, or billions, of annotated training examples. To help bridge this gap in data, researchers have developed various techniques for training general purpose language representation models using the enormous piles of unannotated text on the web (this is known as ***pre-training***). These general purpose pre-trained models can then be ***fine-tuned*** on smaller task-specific datasets, e.g., when working with problems like question answering and sentiment analysis. This approach results in great accuracy improvements compared to training on the smaller task-specific datasets from scratch. BERT is a recent addition to these techniques for NLP pre-training; it caused a stir in the deep learning community because it presented state-of-the-art results in a wide variety of NLP tasks, like question answering.

The best part about BERT is that it can be download and used for free — we can either use the BERT models to extract high quality language features from our text data, or we can fine-tune these models on a specific task, like sentiment analysis and question answering, with our own data to produce state-of-the-art predictions.

2. What is the core idea behind it?

What is language modeling really about? Which problem are language models trying to solve? Basically, their task is to "fill in the blank" based on context. For example, given

"The woman went to the store and bought a _____ of shoes."

a language model might complete this sentence by saying that the word "cart" would fill the blank 20% of the time and the word "pair" 80% of the time.

In the pre-BERT world, a language model would have looked at this text sequence during training from either left-to-right or combined left-to-right and right-to-left. This one-directional approach works well for generating sentences — we can predict the next word, append that to the sequence, then predict the next to next word until we have a complete sentence.

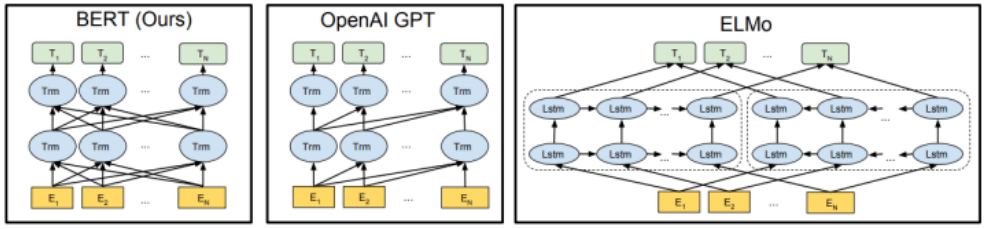
Now enters BERT, a language model which is **bidirectionally trained** (this is also its key technical innovation). This means we can now have a deeper sense of language context and flow compared to the single-direction language models.

Instead of predicting the next word in a sequence, BERT makes use of a novel technique called **Masked LM** (MLM): it randomly masks words in the sentence and then it tries to predict them. Masking means that the model looks in both directions and it uses the full context of the sentence, both left and right surroundings, in order to predict the masked word. Unlike the previous language models, it takes both the previous and next tokens into account at the **same time.** The existing combined left-to-right and right-to-left LSTM based models were missing this "same-time part". (It might be more accurate to say that BERT is non-directional though.)

But why is this non-directional approach so powerful?

Pre-trained language representations can either be ***context-free*** or ***context-based***. *Context-based* representations can then be ***unidirectional*** or ***bidirectional***. Context-free models like word2vec generate a single word embedding representation (a vector of numbers) for each word in the vocabulary. For example, the word "*bank*" would have the same

context-free representation in "*bank account*" and "*bank of the river*." On the other hand, context-based models generate a representation of each word that is based on the other words in the sentence. For example, in the sentence "*I accessed the bank account*," a unidirectional contextual model would represent "*bank*" based on "*I accessed the*" but not "*account*." However, BERT represents "*bank*" using both its previous and next context — "*I accessed the ... account*" — starting from the very bottom of a deep neural network, making it deeply bidirectional.



Moreover, BERT is based on the Transformer model architecture, instead of LSTMs. We will very soon see the model details of BERT, but in general:

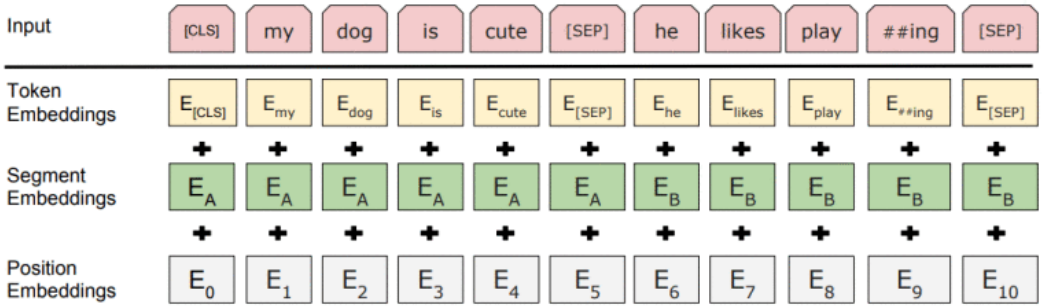
A Transformer works by performing a small, constant number of steps. In each step, it applies an attention mechanism to understand relationships between all words in a sentence, regardless of their respective position. For example, given the sentence, "I arrived at the bank after crossing the river", to determine that the word "bank" refers to the shore of a river and not a financial institution, the Transformer can learn to immediately pay attention to the word "river" and make this decision in just one step.

Now that we understand the key idea of BERT, let's dive into the details.

3. How does it work?

BERT relies on a Transformer (the attention mechanism that learns contextual relationships between words in a text). A basic Transformer consists of an encoder to read the text input and a decoder to produce a prediction for the task. Since BERT's goal is to generate a language representation model, it only needs the encoder part. The input to the encoder for BERT is a sequence of tokens, which are first converted into vectors and then processed in the neural network. But before processing can start, BERT needs the input to be massaged and decorated with some extra metadata:

1. **Token embeddings**: A [CLS] token is added to the input word tokens at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
2. **Segment embeddings**: A marker indicating Sentence A or Sentence B is added to each token. This allows the encoder to distinguish between sentences.
3. **Positional embeddings**: A positional embedding is added to each token to indicate its position in the sentence.



Essentially, the Transformer stacks a layer that maps sequences to sequences, so the output is also a sequence of vectors with a 1:1 correspondence between input and output tokens at the same index. And as we learnt earlier, BERT does not try to predict the next word in the sentence. Training makes use of the following two strategies:

1. Masked LM (MLM)

The idea here is "simple": Randomly mask out 15% of the words in the input — replacing them with a [MASK] token — run the entire sequence through the BERT attention based encoder and then predict only the masked words, based on the context provided by the other non-masked words in the sequence. However, there is a problem with this naive masking approach — the model only tries to predict when the [MASK] token is present in the input, while we want the model to try to predict the correct tokens regardless of what token is present in the input. To deal with this issue, out of the 15% of the tokens selected for masking:

- 80% of the tokens are actually replaced with the token [MASK].
- 10% of the time tokens are replaced with a random token.
- 10% of the time tokens are left unchanged.

While training the BERT loss function considers only the prediction of the masked tokens and ignores the prediction of the non-masked ones. This results in a model that converges much more slowly than left-to-right or right-to-left models.

2. Next Sentence Prediction (NSP)

In order to understand *relationship* between two sentences, BERT training process also uses next sentence prediction. A pre-trained model with this kind of understanding is relevant for tasks like question answering. During training the model gets as input pairs of sentences and it learns to predict if the second sentence is the next sentence in the original text as well.

As we have seen earlier, BERT separates sentences with a special [SEP] token. During training the model is fed with two input sentences at a time such that:

- 50% of the time the second sentence comes after the first one.
- 50% of the time it is a a random sentence from the full corpus.

BERT is then required to predict whether the second sentence is random or not, with the assumption that the random sentence will be disconnected from the first sentence:

$$\text{Input} = \texttt{[CLS] the man went to [MASK] store [SEP]}$$
$$\texttt{he bought a gallon [MASK] milk [SEP]}$$
$$\text{Label} = \texttt{IsNext}$$

$$\text{Input} = \texttt{[CLS] the man [MASK] to the store [SEP]}$$
$$\texttt{penguin [MASK] are flight ##less birds [SEP]}$$
$$\text{Label} = \texttt{NotNext}$$

To predict if the second sentence is connected to the first one or not, basically the complete input sequence goes through the Transformer based model, the output of the [CLS] token is transformed into a 2×1 shaped vector using a simple classification layer, and the IsNext-Label is assigned using softmax.

The model is trained with both Masked LM and Next Sentence Prediction together. This is to minimize the combined loss function of the two strategies — *"together is better"*.

Architecture

There are four types of pre-trained versions of BERT depending on the scale of the model architecture:

`BERT-Base` : 12-layer, 768-hidden-nodes, 12-attention-heads, 110M parameters
`BERT-Large` : 24-layer, 1024-hidden-nodes, 16-attention-heads, 340M parameters

Fun fact: BERT-Base was trained on 4 cloud TPUs for 4 days and BERT-Large was trained on 16 TPUs for 4 days!

For details on the hyperparameter and more on the architecture and results breakdown, I recommend you to go through the original paper.

4. When can we use it and how to fine-tune it?

BERT outperformed the state-of-the-art across a wide variety of tasks under general language understanding like natural language inference, sentiment analysis, question answering, paraphrase detection and linguistic acceptability.

Now, how can we fine-tune it for a specific task? BERT can be used for a wide variety of language tasks. If we want to fine-tune the original model based on our own dataset, we can do so by just adding a single layer on top of the core model.

For example, say we are creating **a question answering application**. In essence question answering is just a prediction task — on receiving a question as input, the goal of the application is to identify the right answer from some corpus. So, given a question and a context paragraph, the model predicts a start and an end token from the paragraph that most likely answers the question. This means that using BERT a model for our application can be trained by learning two extra vectors that mark the beginning and the end of the answer.

- Input Question:

```
Where do water droplets collide with ice
crystals to form precipitation?
```

- Input Paragraph:

```
...  Precipitation forms as smaller droplets
coalesce via collision with other rain drops
or ice crystals within a cloud.  ...
```

- Output Answer:

```
within a cloud
```

Just like sentence pair tasks, the question becomes the first sentence and paragraph the second sentence in the input sequence. However, this time there are two new parameters learned during fine-tuning: a **start vector** and an **end vector.**

In the fine-tuning training, most hyper-parameters stay the same as in BERT training; the paper gives specific guidance on the hyper-parameters that require tuning.

Note that in case we want to do fine-tuning, we need to transform our input into the specific format that was used for pre-training the core BERT models, e.g., we would need to add special tokens to mark the beginning ([CLS]) and separation/end of sentences ([SEP]) and segment IDs used to distinguish different sentences — convert the data into features that BERT uses.

Part II

5. How can we use it? Using BERT for Text Classification — Tutorial

Now that we know the underlying concepts of BERT, let's go through a practical example. For this guide, I am going to be using the **Yelp Reviews Polarity dataset** which you can find here. This is a simple binary text classification task — the goal is to classify short texts into *good* and *bad* reviews. Let's go through the full workflow for this:

1. Installation

Setting things up in your python tensorflow environment is pretty simple:

a. Clone the BERT Github repository onto your own machine. On your terminal, type

```
git clone https://github.com/google-research/bert.git
```

b. Download the pre-trained BERT model files from official BERT Github page here. These are the weights, hyperparameters and other necessary files with the information BERT learned in pre-training. Save this into the directory where you cloned the git repository and unzip it. Here are links to the files for English:

BERT-Base, Uncased *: 12-layers, 768-hidden, 12-attention-heads, 110M parameters*

BERT-Large, Uncased *: 24-layers, 1024-hidden, 16-attention-heads, 340M parameters*

BERT-Base, Cased *: 12-layers, 768-hidden, 12-attention-heads , 110M parameters*

BERT-Large, Cased *: 24-layers, 1024-hidden, 16-attention-heads, 340M parameters*

We need to choose which BERT pre-trained weights we want. For example, if we don't have access to a Google TPU, we'd rather stick with the Base models. And then the choice of "cased" vs "uncased" depends on whether we think letter casing will be helpful for the task at hand. I downloaded the BERT-Base-Cased model for this tutorial.

2. Preparing the data

In order to use BERT, we need to convert our data into the format expected by BERT — we have reviews in the form of csv files; BERT, however, wants data to be in a ***tsv*** file with a specific format as given below (four columns and no header row):

- **Column 0:** An ID for the row
- **Column 1:** The label for the row (should be an int — class labels: 0,1,2,3 etc)
- **Column 2:** A column of the same letter for all rows — this is a throw-away column that we need to include because BERT expects it.
- **Column 3:** The text examples we want to classify

So, create a folder in the directory where you cloned BERT for adding three separate files there, called `train.tsv` `dev.tsv` and `test.tsv` (tsv for tab separated values).
In `train.tsv` and `dev.tsv` we will have all the 4 columns while in `test.tsv` we will only keep 2 of the columns, i.e., id for the row and the text we want to classify.

The code below shows how we can read the Yelp reviews and set up everything to be BERT friendly:

Here *is the link to this code on git.*

3. Training Model using Pre-trained BERT model

Some checkpoints before proceeding further:

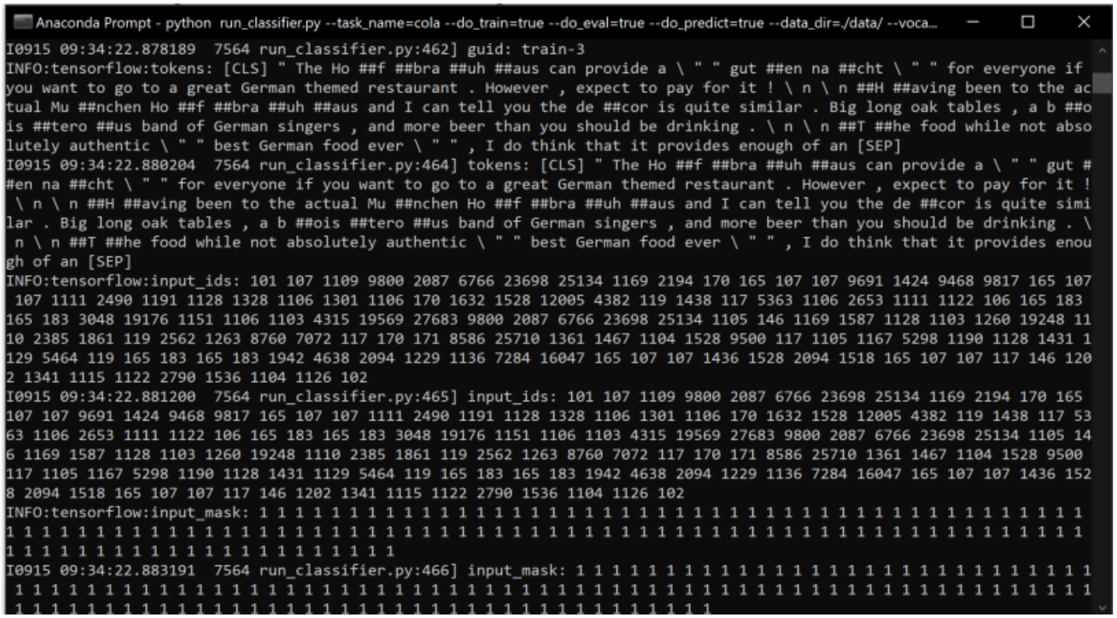
- All the *.tsv* files should be in a folder called ***"data"*** in the "BERT directory".
- We should have created a folder ***"bert_output"*** where the fine tuned model will be saved.
- The **pre-trained BERT model** should have been saved in the "BERT directory".
- The paths in the command are relative path, ***"./"***

Now, navigate to the directory you cloned BERT into and type the following command:

```
python run_classifier.py
--task_name=cola
--do_train=true
--do_eval=true
```

```
    --do_predict=true
    --data_dir=./data/
    --vocab_file=./cased_L-12_H-768_A-12/vocab.txt
    --bert_config_file=./cased_L-12_H-768_A-12/bert_config.json
    --init_checkpoint=./cased_L-12_H-768_A-12/bert_model.ckpt
    --max_seq_length=128
    --train_batch_size=32
    --learning_rate=2e-5
    --num_train_epochs=3.0
    --output_dir=./bert_output/
    --do_lower_case=False
```

If we observe the output on the terminal, we can see the transformation of the input text with extra tokens, as we learned when talking about the various input tokens BERT expects to be fed with:



Training with BERT can cause out of memory errors. This is usually an indication that we need more powerful hardware — a GPU with more on-board RAM or a TPU. However, we can try some workarounds before looking into bumping up hardware. For example, we can try to reduce the training_batch_size; though the training will become slower by doing so — ***"no free lunch!"***

Training can take a veery long time. So you can run the command and pretty much forget about it, unless you have a very powerful machine. Oh, and it also slows down all the other processes — at least I wasn't able to really use my machine during training.

We can see the progress logs on the terminal. Once training completes, we get a report on how the model did in the `bert_output` directory; `test_results.tsv` is generated in the output directory as a result of predictions on test dataset, containing predicted probability value for the class labels.

4. Making predictions on new data

If we want to make predictions on new test data, `test.tsv`, then once model training is complete, we can go into the `bert_output` directory and note the number of the highest-number `model.ckpt` file in there. These checkpoint files contain the weights for the trained model. Once we have the highest checkpoint number, we can run the `run_classifier.py` again but this time *init_checkpoint* should be set to the highest model checkpoint, like so:

```
export TRAINED_MODEL_CKPT=./bert_output/model.ckpt-[highest checkpoint number]

python run_classifier.py
--task_name=cola
--do_predict=true
--data_dir=./data
--vocab_file=./cased_L-12_H-768_A-12/vocab.txt
--bert_config_file=/cased_L-12_H-768_A-12/bert_config.json
--init_checkpoint=$TRAINED_MODEL_CKPT
--max_seq_length=128
--output_dir=./bert_output
```

This should generate a file called `test_results.tsv`, with number of columns equal to the number of class labels.

(Note that we already had *–do_predict=true* parameter set during the training phase. That can be omitted and test results can be generated separately with the command above.)

5. Taking it a step further

We did our training using the out-of-the-box solution. However, we can also do custom **fine tuning** by **creating a single new layer trained to adapt BERT** to our sentiment task (or any other task). This blog post has already become very long, so I am not going to stretch it further by diving into creating a custom layer, but:

- Here is a tutorial for doing just that on this same Yelp reviews dataset in **PyTorch**.
- Alternatively, there is this great colab notebook created by Google researchers that shows in detail how to predict whether an IMDB movie review is positive or negative, with a new layer on top of the pre-trained BERT model in **Tensorflow**.

Final Thoughts

BERT is a really powerful language representation model that has been a big milestone in the field of NLP — it has greatly increased our capacity to do transfer learning in NLP; it comes with the great promise to solve a wide variety of NLP tasks. Here, I've tried to give a complete guide to getting started with BERT, with the hope that you will find it useful to do some NLP awesomeness.

If you want to learn more about BERT, the best resources are the original paper and the associated open sourced Github repo. There is also an implementation of BERT in PyTorch.

*This article was originally published on my **ML blog**.* Check out my other writings there, and *follow* to not miss out on the latest!

Also, help me reach out to the readers who can benefit from this by hitting the clap button. Thanks and Happy Learning! 🙃

P.S. I regularly post interesting AI related content on LinkedIn. If you want short weekly lessons from the AI world, you are welcome to follow me there!

References and Further Readings

- Original Paper
- Google-Research GitHub Repo
- Blog Post by Google AI
- Colab Notebook: Predicting Movie Review Sentiment with BERT on TF Hub
- Using BERT for Binary Text Classification in PyTorch