* **Overview**

The intuition for utilizing a pre-trained model is simple: A deep neural network that is trained on large corpus, say all the Wikipedia data, should have enough knowledge about the underlying relationships between different words and sentences. It should also be easily adapted to a different domain, such as medical, autos or financial domain, with better performance than training from scratch.

* **Architecture**

* **How fine-tuning works fine without pre-training, the whole humongous machine learning models on our customized datasets?**

In the field of computer vision, researchers have repeatedly shown the value of transfer learning — pre-training a neural network model on a known task, for instance ImageNet, and then performing fine-tuning — using the trained neural network as the basis of a new purpose-specific model. In recent years, researchers have been showing that a similar technique can be useful in many natural language tasks. So, here I embarked upon testing this statement for the NLP model, BERT. For NLP models the technique I have used is called as feature-based training.

In this approach, a pre-trained neural network produces word embeddings which are then used as features in NLP models.

* Before BERT came into picture, we had ELMo and transformer which brought in the concept of attention (

learns contextual relations between words (or sub-words) in a text) and encoder-decoder. BERT absorbed in the encoder and attention layers with multipurpose output and input layers making it flexible to use for various NLP tasks like sentence classification, QnA, paragraph summarisation etc giving out state-of-the-art results on public datasets.

* As opposed to directional models, which read the text input sequentially (left-to-right or right-to-left), the Transformer encoder reads the entire sequence of words at once. Therefore, it is considered bidirectional, though it would be more accurate to say that it’s non-directional. This characteristic allows the model to learn the context of a word based on all of its surroundings (left and right of the word).
* **MLM (Masked Language Model)**

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.

Embedding 
to vocab + 
softmax 
Classification Layer. Fully-connected layer + GELIJ + Norm 
01 
Embedding 
04 
Transformer encoder 
[MASKI 
05 

WORD MASKING EXPLAINED:

Training the language model in BERT is done by predicting 15% of the tokens in the input, that were randomly picked. These tokens are pre-processed as follows – 80% are replaced with a “[MASK]” token, 10% with a random word, and 10% use the original word. The intuition that led the authors to pick this approach is as follows (Thanks to Jacob Devlin from Google for the insight):

* If we used [MASK] 100% of the time the model wouldn’t necessarily produce good token representations for non-masked words. The non-masked tokens were still used for context, but the model was optimized for predicting masked words.
* If we used [MASK] 90% of the time and random words 10% of the time, this would teach the model that the observed word is *never* correct.
* If we used [MASK] 90% of the time and kept the same word 10% of the time, then the model could just trivially copy the non-contextual embedding.

No ablation was done on the ratios of this approach, and it may have worked better with different ratios. In addition, the model performance wasn’t tested with simply masking 100% of the selected tokens.

* Next Sentence Prediction (NSP)

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:

1. A [CLS] token is inserted at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
2. 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.
3. 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.

6u!ppøqLL13 
但 uO sod 
山 吅 
6u!ppøqtua 
00u0 as 
s6u'ppeqw3 
V'dul 

To predict if the second sentence is indeed connected to the first, the following steps are performed:

1. The entire input sequence goes through the Transformer model.
2. 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).
3. Calculating the probability of IsNextSequence with SoftMax.

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.

**Fine-tuning and using BERT as classifier:**

BERT can be used for a wide variety of language tasks, while only adding a small layer to the core model:

Classification tasks such as sentiment analysis are done similarly to Next Sentence classification, by adding a classification layer on top of the Transformer output for the [CLS] token.

I had data with two classes: 1(domain positive queries)

0(domain negative queries and non-domain queries)

**Implementation**

Rest of the article describes how I used Google's BERT to help classify Queries of certain domain as positive (domain positive) and negative (domain negative + other negative). The data format requirements and the form of output can be confusing so this is a *detailed* walk through my application.

It builds on Google-research(official) repository of BERT and bert\_paper\_classification repository of craic(master).

This describes how to set up and use BERT on a Linux machine with a GPU. My configuration is a Standard\_NC6\_Promo AzureVM with Ubuntu 18.04.2 LTS.

**Install BERT software model files**

You 'install' BERT by cloning the Git repo:

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

And get the missing files from the bert\_paper\_classification repository of craic(master).

There are 2 sizes of BERT Model - Base and Large - the Base model is the only one that will fit in GPU memory.

There are 2 versions of the Base model - Uncased and Cased - choose Cased if you think the case of your text is significant.

The links for these two are:

* [BERT Base Uncased](https://storage.googleapis.com/bert_models/2018_10_18/uncased_L-12_H-768_A-12.zip)
* [BERT Base Cased](https://storage.googleapis.com/bert_models/2018_10_18/cased_L-12_H-768_A-12.zip)

As the case isn't significant in queries, I have used BERT Base Uncased.

**Data Preparation**

The structure of input and output file is specified hence you need to deal with some quirks to get the data in and out.

Input Data

BERT requires 3 input files - Training Data, Validation Data and Test Data. These need to be in TAB delimited format with one query per line.

Training and Validation data have one format ...*but Test data has a slightly different format*...

* Training Data

The Training Data format has 4 columns, separated by a Tab (\t) character

* column 1 - a unique ID
* column 2 - an integer label - my dataset uses 0 for a negative query and 1 for a positive query
* column 3 - a dummy column where each line has the same letter (in this case 'a') - perhaps this is used in other NLP tasks
* column 4 - the text, which has had tabs and newlines stripped out of it.

Note: Shuffle the training data. ($ shuf -o train.tsv < train.tsv)

To change to the required format from a file with format:

|  |  |
| --- | --- |
| 1 | Honda cvr specs |
| 0 | My first bike |

The following line code transforms the above format data into required format:

|  |
| --- |
| $python convert\_to\_required\_format.py --file filename.tsv < filename.tsv |

Here is an example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1142218 | 0 | a | suvs for sale in dallas tx |
| 1684872 | 0 | a | blasey ford posted that someone should accuse gorsich |
| 2036574 | 0 | a | tesla model 3 charging cable 40amp |
| 2289428 | 1 | a | points for 1975 honda 125 xl |
| 1955936 | 1 | a | eqinozpricing29 |
| 477633 | 0 | a | chevy 3 5 p0300 |

* Evaluation Data

dev.tsv is the evaluation data with the same format as the training data(train.tsv)

* Testing Data

The test dataset, test.tsv, has a slightly different format. It gets a header line and the only two fields included are the ID and Text - no label column. That's why I created test\_original.tsv with same format as training data (keep this around as it is needed later).

convert\_tsv\_to\_test.py does this using the 'initial' test.tsv file:

|  |
| --- |
| $ python convert\_tsv\_to\_test.py --file test\_original.tsv > test.tsv |

The format looks like:

|  |
| --- |
| id text  0 only possesses inherent sovereignty  1 Mitsubishi electric phev car  2 mbrp 7 tip  3 roush wiple pdf |

**Run BERT Fine-tuning**

We need to run **run\_classifier.py** with suitable set of arguments.

This command assumes that your input files are in directory ./bert\_input and you want the output in ./bert\_output. It also assumes that the BERT software and model files are in the ./bert subdirectory of your current location.

|  |
| --- |
| $ export BERT\_BASE\_DIR = ./bert/uncased\_L-12\_H-768\_A-12 |

To avoid typos in a long command, I made a run\_bert.sh (install bash if not available) file with the following commands.

|  |
| --- |
| $ python bert/run\_classifier.py \  --task\_name=cola \  --do\_train=true \  --do\_eval=true \  --vocab\_file=$BERT\_BASE\_DIR/vocab.txt \  --bert\_config\_file=$BERT\_BASE\_DIR/bert\_config.json \  --init\_checkpoint=$BERT\_BASE\_DIR/bert\_model.ckpt \  --max\_seq\_length=128 \  --train\_batch\_size=32 \  --learning\_rate=2e-5 \  --num\_train\_epochs=3 \  --data\_dir=./bert\_input \  --output\_dir=./bert\_output |

What do these arguments mean?

* --task\_name cola - this is the classification task
* --do\_train, --do\_eval - we want to train the model and evaluate it
* --vocab\_file, --bert\_config\_file, --init\_checkpoint - use these BERT model files - and in particular use this checkpoint file that represents all the wieghts from the pre-trained model that we want to fine tune
* --max\_seq\_length - limit the number of words in the text that we will use
* --train\_batch\_size - how many text records to use in each batch
* --learning\_rate, --num\_train\_epochs - use this learning rate for this number of epochs
* --data\_dir - the directory with your input data
* --output\_dir - the directory where your output data will be placed

Here is the end of a run as an example

|  |
| --- |
| INFO:tensorflow:Restoring parameters from ./bert\_output/model.ckpt-253107  INFO:tensorflow:Running local\_init\_op.  INFO:tensorflow:Done running local\_init\_op.  INFO:tensorflow:Finished evaluation at 2019-07-17-07:58:31  INFO:tensorflow:Saving dict for global step 253107: eval\_accuracy = 0.93616265, eval\_loss = 0.19613431, global\_step = 253107, loss = 0.19613406, precision = 1.0, recall = 0.93616265  INFO:tensorflow:Saving 'checkpoint\_path' summary for global step 253107: ./bert\_output/model.ckpt-253107  INFO:tensorflow:evaluation\_loop marked as finished  INFO:tensorflow:\*\*\*\*\* Eval results \*\*\*\*\*  INFO:tensorflow: eval\_accuracy = 0.93616265  INFO:tensorflow: eval\_loss = 0.19613431  INFO:tensorflow: global\_step = 253107  INFO:tensorflow: loss = 0.19613406  INFO:tensorflow: precision = 1.0  INFO:tensorflow: recall = 0.93616265 |

If you look in your output directory that you specified you will see a bunch of files:

|  |
| --- |
| (base) anjali@anjali:~/BERT/bert\_output$ ls -l  total 7609036  -rw-rw-r-- 1 anjali anjali 283 Jul 16 15:20 checkpoint  drwxr-xr-x 2 anjali anjali 4096 Jul 17 07:58 eval  -rw-rw-r-- 1 anjali anjali 483232083 Jul 17 05:08 eval.tf\_record  -rw-rw-r-- 1 anjali anjali 125 Jul 17 07:58 eval\_results.txt  -rw-rw-r-- 1 anjali anjali 14620356 Jul 16 15:20 events.out.tfevents.1563195835.anjali  -rw-rw-r-- 1 anjali anjali 9470934 Jul 15 13:04 graph.pbtxt  -rw-rw-r-- 1 anjali anjali 1313805344 Jul 16 15:01 model.ckpt-250000.data-00000-of-00001  -rw-rw-r-- 1 anjali anjali 22764 Jul 16 15:01 model.ckpt-250000.index  -rw-rw-r-- 1 anjali anjali 4012474 Jul 16 15:01 model.ckpt-250000.meta  -rw-rw-r-- 1 anjali anjali 1313805344 Jul 16 15:07 model.ckpt-251000.data-00000-of-00001  -rw-rw-r-- 1 anjali anjali 22764 Jul 16 15:07 model.ckpt-251000.index  -rw-rw-r-- 1 anjali anjali 4012474 Jul 16 15:07 model.ckpt-251000.meta  -rw-rw-r-- 1 anjali anjali 1313805344 Jul 16 15:13 model.ckpt-252000.data-00000-of-00001  -rw-rw-r-- 1 anjali anjali 22764 Jul 16 15:13 model.ckpt-252000.index  -rw-rw-r-- 1 anjali anjali 4012474 Jul 16 15:13 model.ckpt-252000.meta  -rw-rw-r-- 1 anjali anjali 1313805344 Jul 16 15:20 model.ckpt-253000.data-00000-of-00001  -rw-rw-r-- 1 anjali anjali 22764 Jul 16 15:20 model.ckpt-253000.index  -rw-rw-r-- 1 anjali anjali 4012474 Jul 16 15:20 model.ckpt-253000.meta  -rw-rw-r-- 1 anjali anjali 1313805344 Jul 16 15:20 model.ckpt-253107.data-00000-of-00001  -rw-rw-r-- 1 anjali anjali 22764 Jul 16 15:20 model.ckpt-253107.index  -rw-rw-r-- 1 anjali anjali 4012474 Jul 16 15:20 model.ckpt-253107.meta  -rw-rw-r-- 1 anjali anjali 154617259 Jul 16 16:08 predict.tf\_record  -rw-rw-r-- 1 anjali anjali 18294250 Jul 16 17:02 test\_results.tsv  -rw-rw-r-- 1 anjali anjali 522122645 Jul 15 13:03 train.tf\_record |

**Varying the Run Parameters**

There are only a few parameters that are available to the user from the command line:

* max\_seq\_length - the number of words in the document to include in training.
* train\_batch\_size
* learning\_rate
* num\_epochs

With the size of the Base model and the memory constraints of a GPU, you have little room for variation with these parameters. You can try increasing, or decreasing, max\_seq\_length by factors of 2 and doing the opposite with the train\_batch\_size.

**Predict the Results on the Test Dataset**

For this you run run\_classifier.py again but with a different set of arguments

|  |
| --- |
| $ export BERT\_BASE\_DIR=./bert/uncased\_L-12\_H-768\_A-12    $ export TRAINED\_CLASSIFIER=./bert\_output/model.ckpt-[highest checkpoint number you saw] |

I have made test\_bert.sh with the following commands:

|  |
| --- |
| $ python bert/run\_classifier.py \  --task\_name=cola \  --do\_predict=true \  --vocab\_file=$BERT\_BASE\_DIR/vocab.txt \  --bert\_config\_file=$BERT\_BASE\_DIR/bert\_config.json \  --init\_checkpoint=./bert\_output/model.ckpt-253107\  --max\_seq\_length=128 \  --data\_dir=./data \  --output\_dir=./bert\_output/ |

What do these arguments mean?

* --task\_name cola - this is the classification task
* --do\_predict - we want to use the model for predictions
* --vocab\_file, --bert\_config\_file - use these BERT model files
* --init\_checkpoint - but use the new, fine-tuned model checkpoint that we created
* --max\_seq\_length - this needs to be the same as you used in training
* --data\_dir - the directory with your input data
* --output\_dir - the directory where your output data will be placed

This command will take your test.tsv file and run it through the model to produce a file called test\_results.tsv in your output directory.

This will have, in my case, two probabilities for each line in the input file.

For example

|  |
| --- |
| 0.02100023        0.97899973 0.9973598        0.002640231 0.015744895        0.9842551 0.99622524        0.0037748201 |

The first column is the probability for the **negative** state and the second for the **positive** state.

So, lines 1 and 3 are predicted to be positive and lines 2 and 4 are predicted to be negative.

If you were doing a classification with multiple states, I would expect this would have multiple columns.

*But how do you see if the individual predictions are correct?*

The input **test.tsv** file did not have the original labels **AND** this output file does not include the record **ID** numbers that we are using. This is a problem with the current version of the BERT software - but we can solve that with another **custom script** and the original version of the Test data file that we had earlier on.

**evaluate\_test\_set\_predictions.py** takes the original format test data file and the results file and return the number of correctly classified queries and accuracy. It also returns **incorrect\_results.tsv** in your current directory which has the **ID** of wrongly classified queries.

I can then look up the queries by their ID in **test\_original.tsv** that it got wrong and see if I can rationalize the decision.

**Predict on New Data Records**

Now that I have my trained model, I want to use it in production whenever I have new batches of data. These have not been classified ahead of time and so I just give them the arbitrary label 0, run them through the model and make a list of those predicted to be positives.

**evaluate\_new\_data.py** is a variant of **evaluate\_test\_set\_predictions.py**

|  |
| --- |
| $ python evaluate\_new\_data.py --tsv test\_original.tsv --results test\_results.tsv  26004  30333  22286  21723  [...] |

**NOTE:**  Convert the raw data into TSV format.