This is a two part Notebook

1. Comprehensive BERT Tutorial
2. Implementation in Tensorflow 2.0

> **Note:** The main objective of this notebook is to provide a **baseline for this competition with some explanation about BERT**. I decided to wite such a notebook because I didn't find anything quite like this when I started out at NLP Competitions. I hope beginners can benefit from this notebook. Even if you're a non-beginner there might be some elements in this notebook you may be interested in.

If you like this approach please give this kernel an UPVOTE to show your appreciation

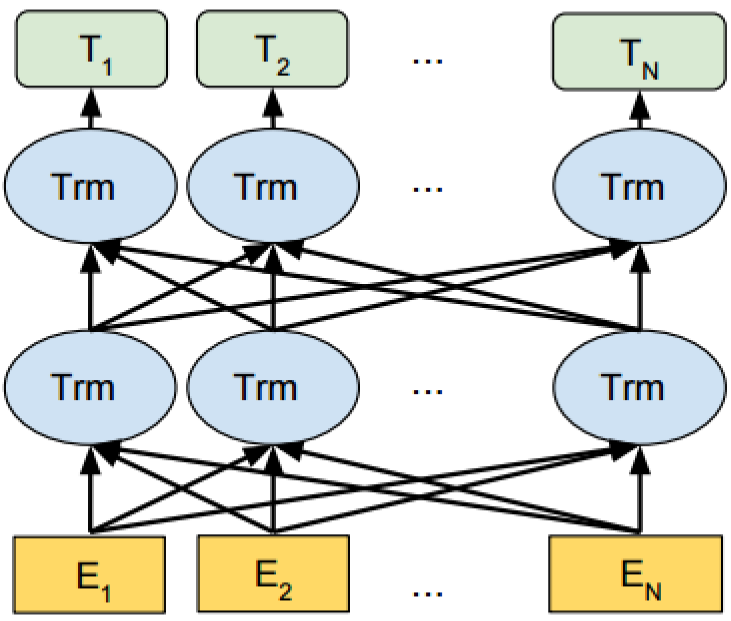
Comprehensive BERT Tutorial

Introduction

So if you're like me just starting out at NLP after spending a few months building Computer Vision models as a beginner then surely this kernel has something in store for you.

So if you're like me just starting out at NLP after spending a few months building Computer Vision models as a beginner then surely this kernel has something in store for you.

I had a hard time wrapping my head around this all new bleeding-edge, state-of-the-art NLP model BERT, I had to dig through a lot of articles to truly grasp what BERT is all about, I'll share my understanding of BERT in this notebook.



Contents

1. The BERT Landscape
2. What is BERT?
3. Why BERT Matters?
4. How BERT works?
    4.1 Architecture of BERT
    4.2 Preprocessing text for BERT
    4.3 Pre-training
5. Fine Tuning Techniques for BERT
    5.1 Sequence Classification Tasks
    5.2 Sentence Pair Classification Tasks
    5.3 Question Answering Tasks
    5.4 Single Sentence Tagging Tasks
    5.5 Hyperparameter Tuning
6. BERT Benchmarks on Question/Answering Tasks
7. Key Takeaways
8. Conclusion

1. The BERT Landscape

> BERT is a deep learning model that has given state-of-the-art results on a wide variety of natural language processing tasks. It stands for **Bidirectional Encoder Representations for Transformers**. It has been pre-trained on Wikipedia and BooksCorpus and requires (only) task-specific fine-tuning.

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 **Question Answering (SQuAD v1.1)**, **Natural Language Inference (MNLI)**, and others.

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.

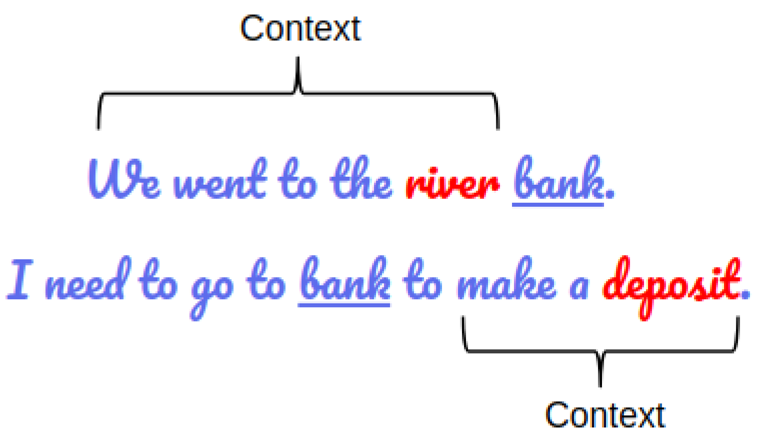
2. What is BERT?

It is basically a bunch of Transformer encoders stacked together (not the whole Transformer architecture but just the encoder). The concept of bidirectionality is the key differentiator between BERT and its predecessor, OpenAI GPT. BERT is bidirectional because its self-attention layer performs self-attention on both directions.

There are a few things I want to explain in this section.

- First, It’s easy to get that BERT stands for Bidirectional Encoder Representations from Transformers. Each word here has a meaning to it and we will encounter that one by one. 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 pretraining step is really important for 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.

This bidirectional understanding is crucial to take NLP models to the next level. Let's see an example to understand what it really means. There may be two sentences having the same word but their meaning may be completely different based on what comes before or after as we can see here below.


Context
We went to the river bank.
I need to go to bank to make a deposit.
Context

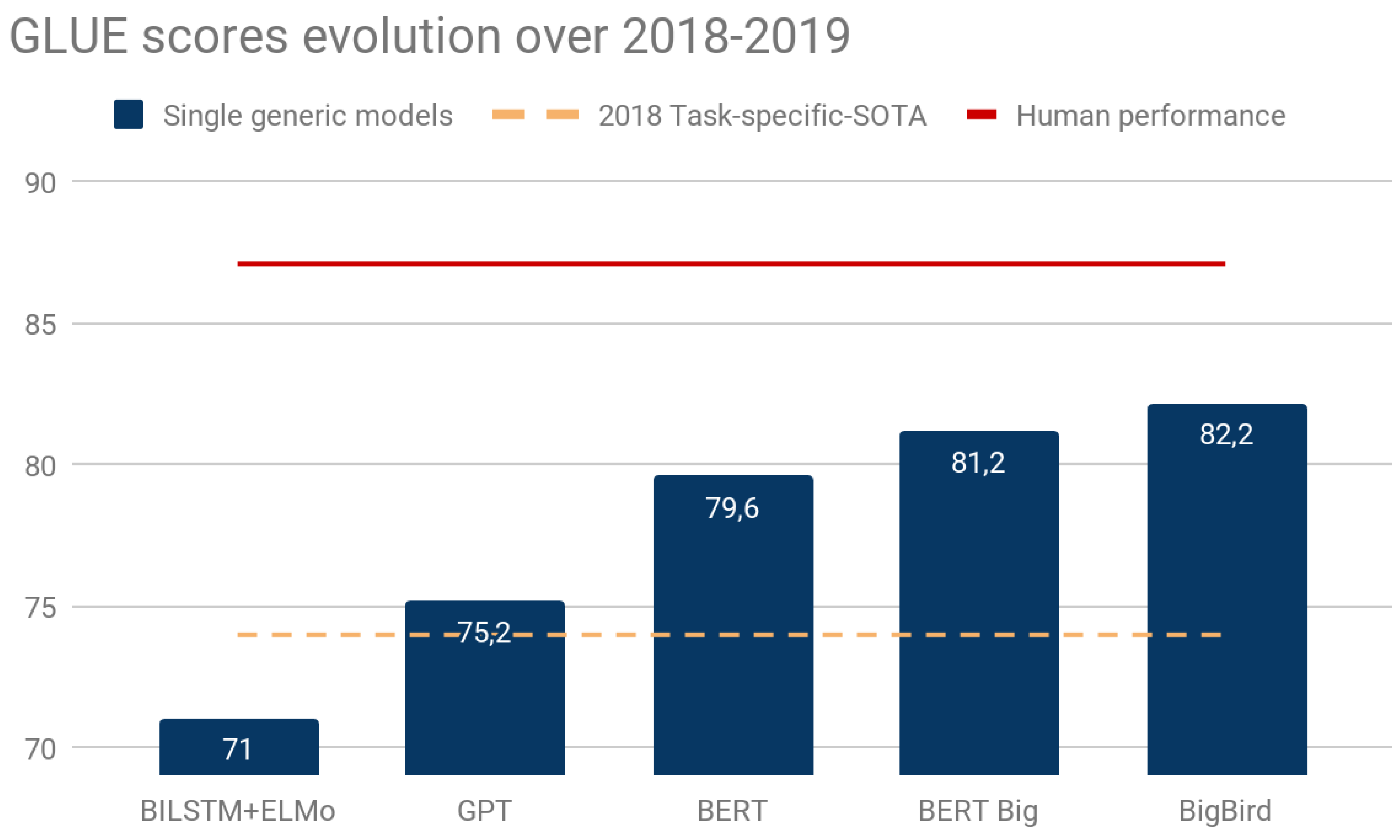
Without taking these contexts into consideration it's impossible for machines to truly understand meanings and it may throw out trashy responses time and time again which is not really a good thing.

But BERT fixes this. Yes it does. That was one of the game changing aspect of BERT.

- Fourth, finally the biggest advantage of BERT is it brought about the **ImageNet movement** with it and the most impressive aspect of BERT is that 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.

3. Why BERT matters?

Now I think it's pretty clear to you why but let's see proof, as we should always do.



While it’s not clear that all GLUE tasks are very meaningful, generic models based on an encoder named Transformer (Open-GPT, BERT and BigBird), closed the gap between task-dedicated models and human performance and within less than a year.

4. How BERT Works?

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.

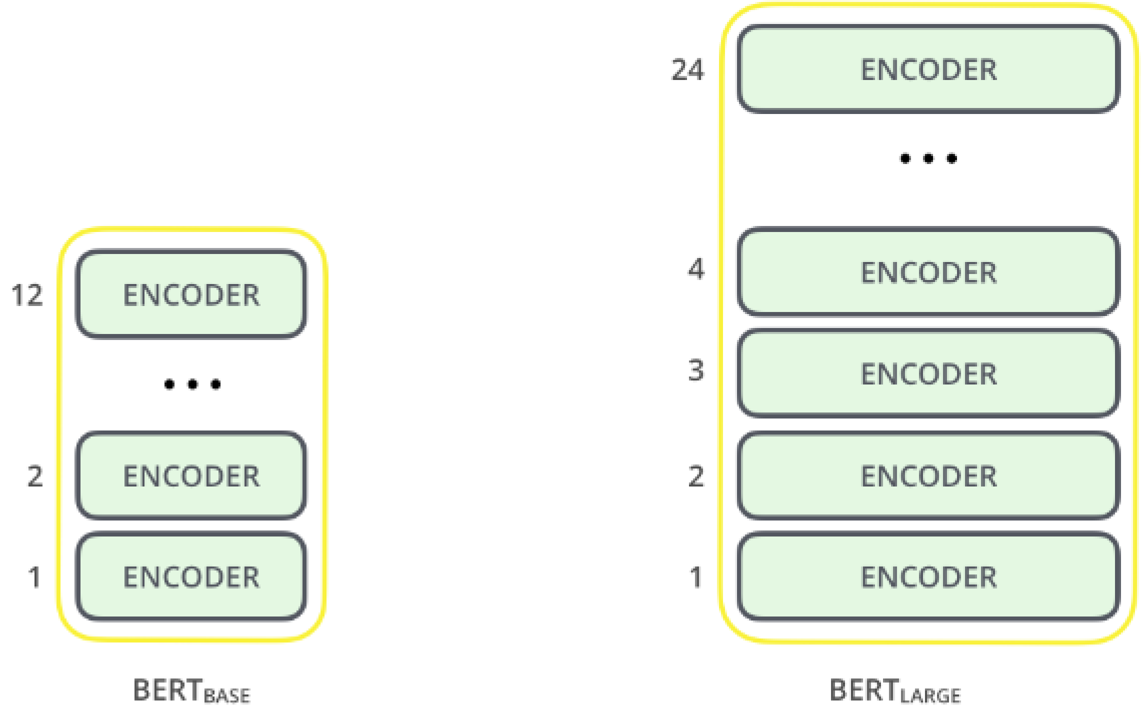
1. Architecture of BERT

BERT is a multi-layer bidirectional Transformer encoder. There are two models introduced in the paper.

- BERT base – 12 layers (transformer blocks), 12 attention heads, and 110 million parameters.
- BERT Large – 24 layers, 16 attention heads and, 340 million parameters.

For an in-depth understanding of the building blocks of BERT (aka Transformers), you should definitely check this awesome post (http://jalammar.github.io/illustrated-transformer/) – The Illustrated Transformers.

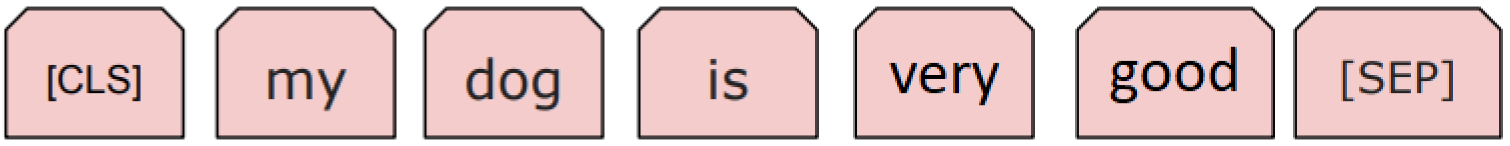
Here's a representation of BERT Architecture



2. Preprocessing Text for BERT

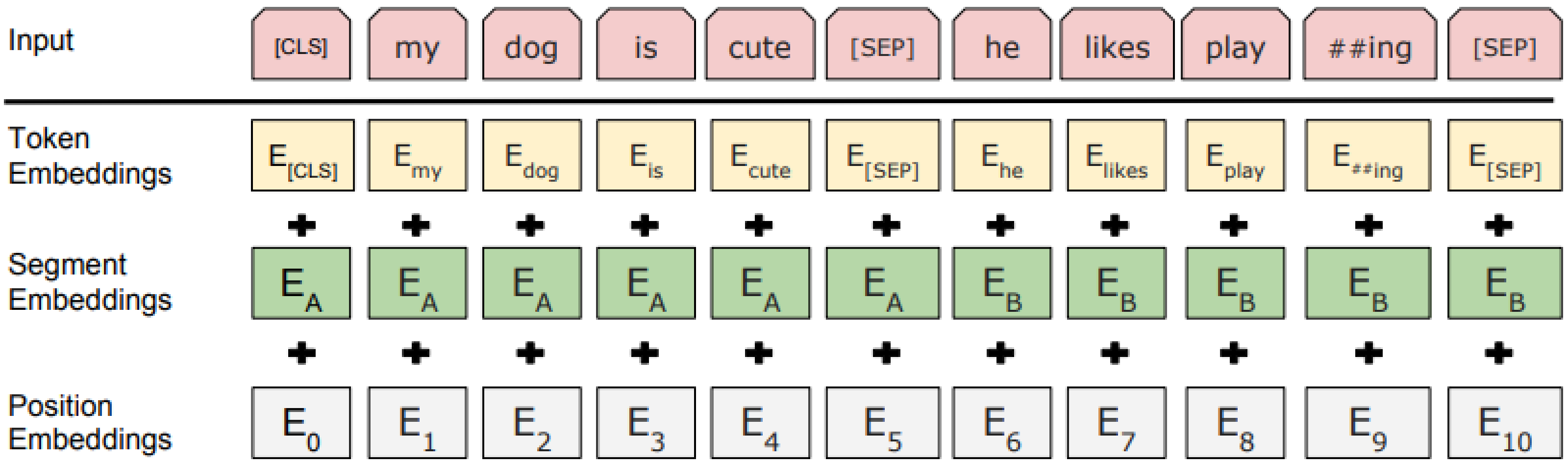
The input representation used by BERT is able to represent a single text sentence as well as a pair of sentences (eg., Question, Answering) in a single sequence of tokens.

- The first token of every input sequence is the special classification token – **[CLS]**. This token is used in classification tasks as an aggregate of the entire sequence representation. It is ignored in non-classification tasks.
- For single text sentence tasks, this **[CLS]** token is followed by the WordPiece tokens and the separator token – **[SEP]**.



- For sentence pair tasks, the WordPiece tokens of the two sentences are separated by another [SEP] token. This input sequence also ends with the **[SEP]** token.
- A sentence embedding indicating Sentence A or Sentence B is added to each token. Sentence embeddings are similar to token/word embeddings with a vocabulary of 2.
- A positional embedding is also added to each token to indicate its position in the sequence.

BERT developers have set a a specific set of rules to represent languages before feeding into the model.



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

- **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
- **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)
- **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.

Tokenization: BERT uses WordPiece tokenization. The vocabulary is initialized with all the individual characters in the language, and then the most frequent/likely combinations of the existing words in the vocabulary are iteratively added.

3. Pre Training

The model was trained in two tasks simultaneously:

1. Masked Language Model

2. Next Sentence Prediction.

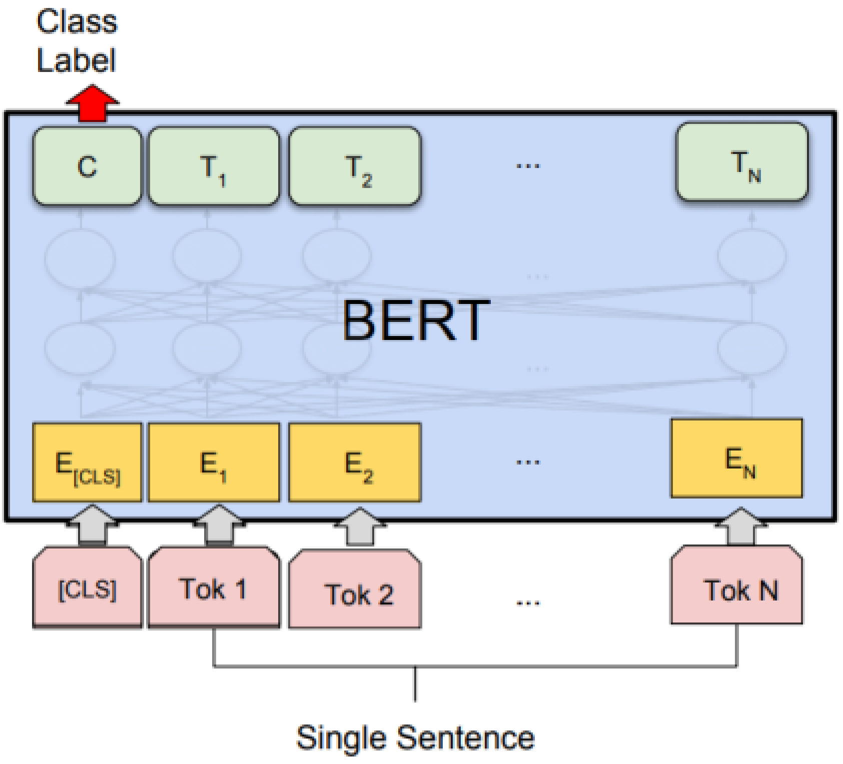
Note: I am not going to go over these two techniques in this notebook. I recommend online reading.

5. Fine Tuning Techniques for BERT

Using BERT for a specific task is relatively straightforward. BERT can be used for a wide variety of language tasks, while only adding a small layer to the core model

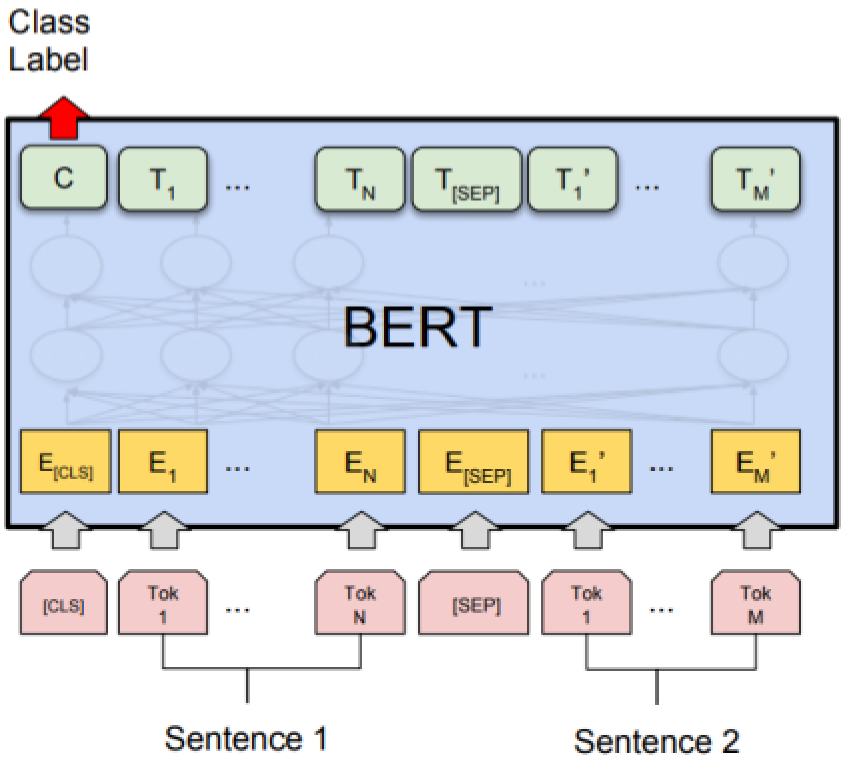
5.1 Sequence Classification Tasks

The final hidden state of the **[CLS]** token is taken as the fixed-dimensional pooled representation of the input sequence. This is fed to the classification layer. The classification layer is the only new parameter added and has a dimension of K x H, where K is the number of classifier labels and H is the size of the hidden state. The label probabilities are computed with a standard softmax.



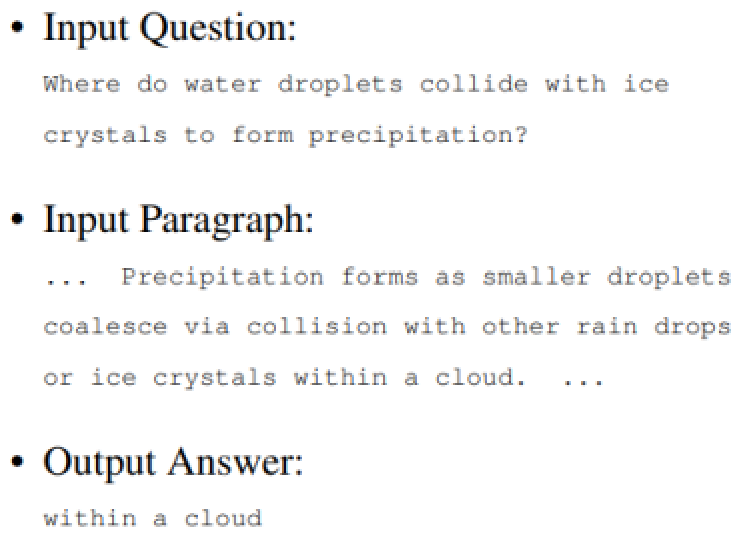
5.2 Sentence Pair Classification Tasks

This procedure is exactly similar to the single sequence classification task. The only difference is in the input representation where the two sentences are concatenated together.



5.3 Question-Answering Tasks (Goal of this competition)

Question answering is a prediction task. 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.



- Input Question:

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

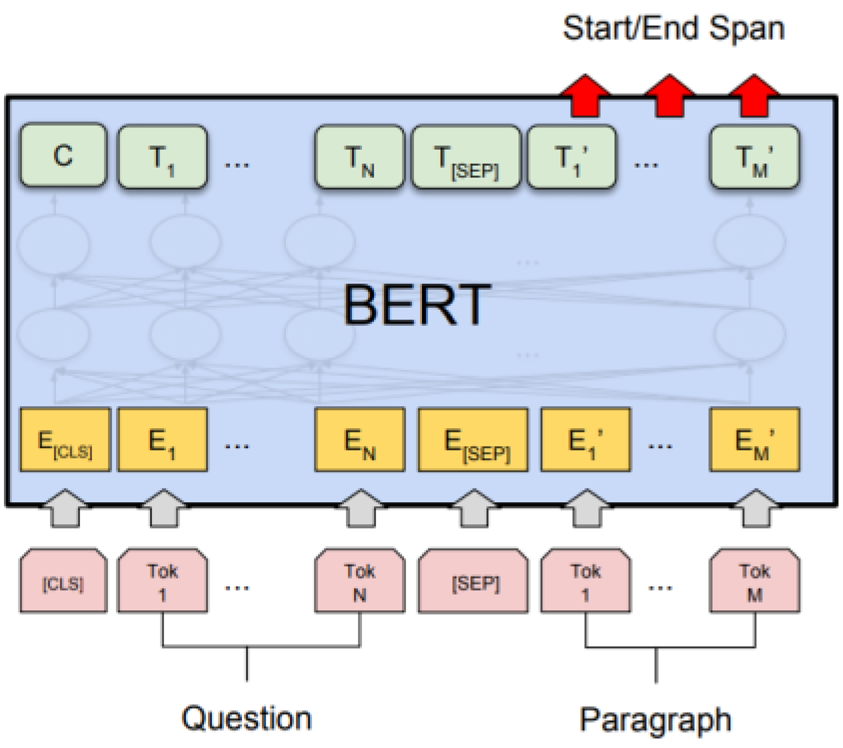
- Input Paragraph:

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

- Output Answer:

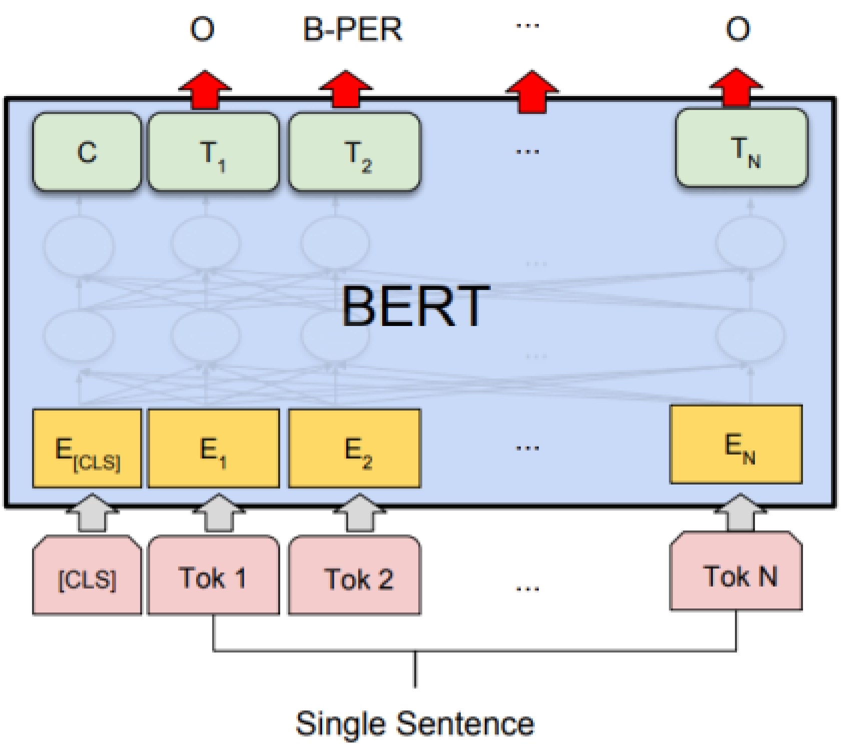
```
within a cloud
```

Just like sentence pair tasks, the question becomes the first sentence and paragraph the second sentence in the input sequence. There are only two new parameters learned during fine-tuning a start vector and an end vector with size equal to the hidden shape size. The probability of token i being the start of the answer span is computed as – softmax(S . K), where S is the start vector and K is the final transformer output of token i. The same applies to the end token.



5.4 Single Sentence Tagging Tasks

In single sentence tagging tasks such as named entity recognition, 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 WordPiece tokenizer breaks some words into sub-words, the prediction of only the first token of a word is considered.



5.5 Hyperparameter Tuning

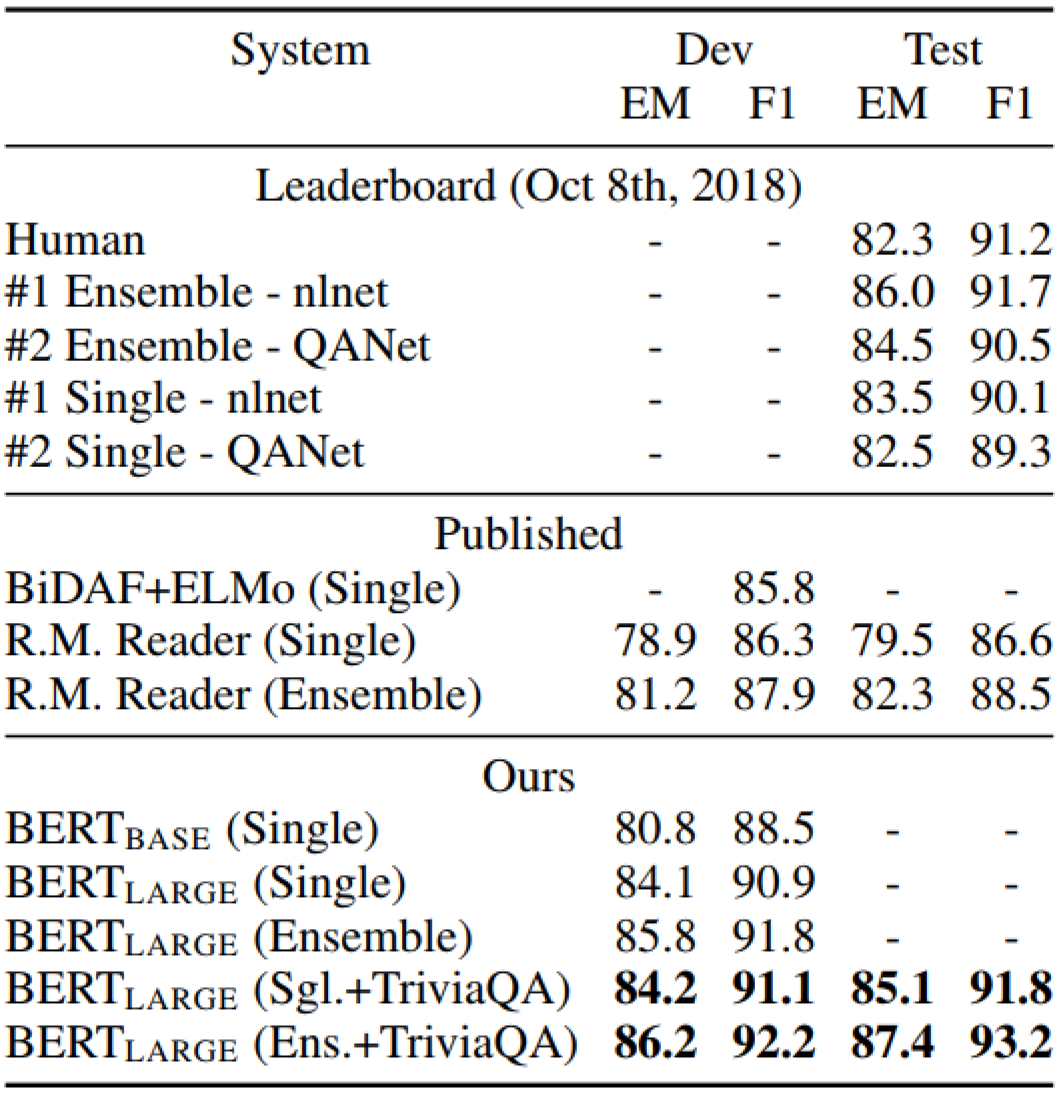
The optimal hyperparameter values are task-specific. But, the authors found that the following range of values works well across all tasks

- **Dropout** – 0.1
- **Batch Size** – 16, 32

6. BERT Benchmarks on Question Answering tasks

> The Standford Question Answering Dataset (SQuAD) is a collection of 100k crowdsourced question/answer pairs (Rajpurkar et al., 2016). Given a question and a paragraph from Wikipedia containing the answer, the task is to predict the answer text span in the paragraph.

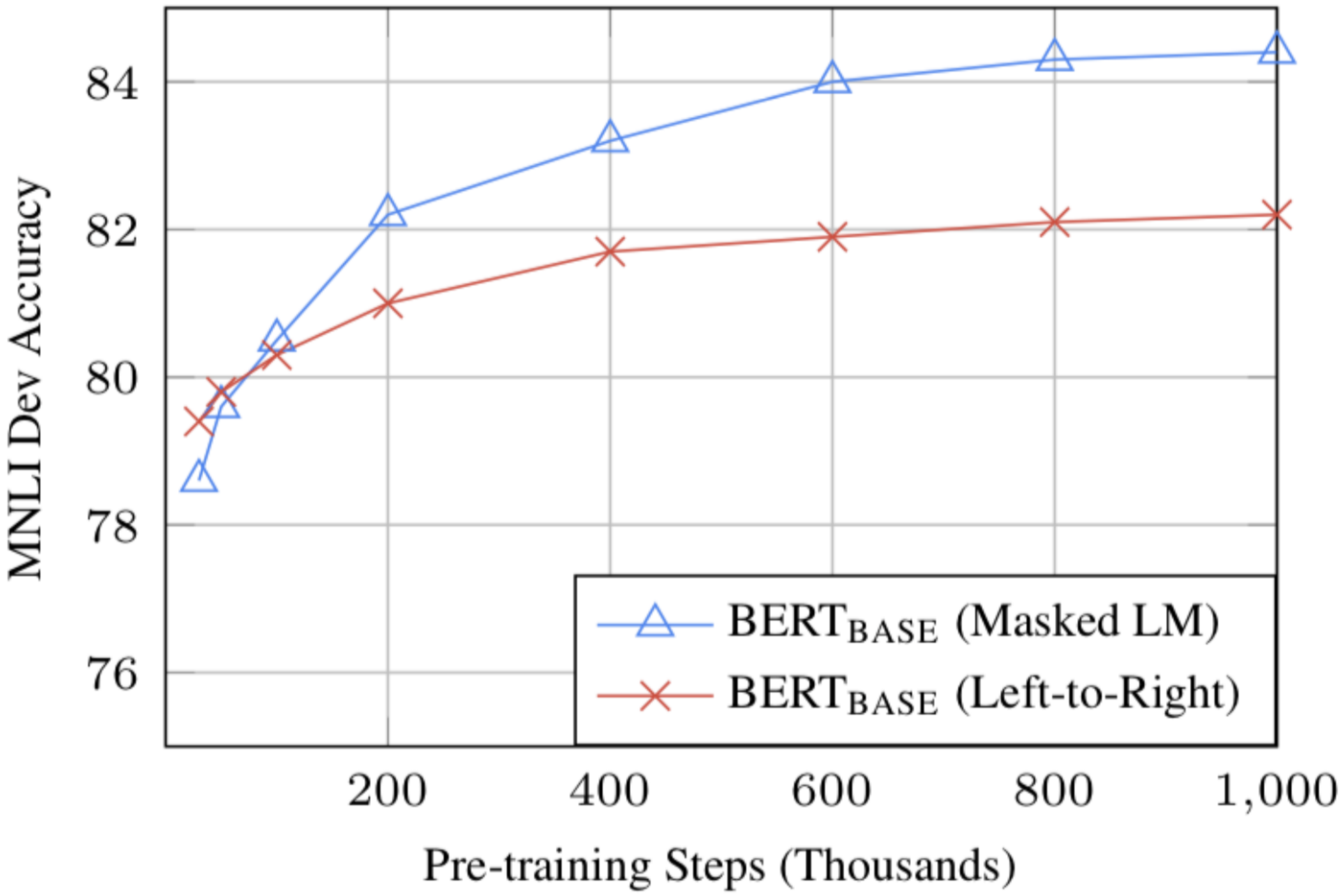
In SQUAD the big improvement in performance was achieved by BERT large. The model that achieved the highest score was an ensemble of BERT large models, augmenting the dataset with TriviaQA.



System	Dev		Test	
	EM	F1	EM	F1
Leaderboard (Oct 8th, 2018)				
Human	-	-	82.3	91.2
#1 Ensemble - nlnet	-	-	86.0	91.7
#2 Ensemble - QANet	-	-	84.5	90.5
#1 Single - nlnet	-	-	83.5	90.1
#2 Single - QANet	-	-	82.5	89.3
Published				
BiDAF+ELMo (Single)	-	85.8	-	-
R.M. Reader (Single)	78.9	86.3	79.5	86.6
R.M. Reader (Ensemble)	81.2	87.9	82.3	88.5
Ours				
$\text{BERT}_{\text{BASE}}$ (Single)	80.8	88.5	-	-
$\text{BERT}_{\text{LARGE}}$ (Single)	84.1	90.9	-	-
$\text{BERT}_{\text{LARGE}}$ (Ensemble)	85.8	91.8	-	-
$\text{BERT}_{\text{LARGE}}$ (Sgl.+TriviaQA)	**84.2**	**91.1**	**85.1**	**91.8**
$\text{BERT}_{\text{LARGE}}$ (Ens.+TriviaQA)	**86.2**	**92.2**	**87.4**	**93.2**

7. Key Takeaways

> 1) Model size matters, even at huge scale. BERT_large, with 345 million parameters, is the largest model of its kind. It is demonstrably superior on small-scale tasks to BERT_base, which uses the same architecture with “only” 110 million parameters.
>
> 2) With enough training data, more training steps == higher accuracy. For instance, on the MNLI task, the BERT_base accuracy improves by 1.0% when trained on 1M steps (128,000 words batch size) compared to 500K steps with the same batch size.
>
> 3) BERT’s bidirectional approach (MLM) converges slower than left-to-right approaches (because only 15% of words are predicted in each batch) but bidirectional training still outperforms left-to-right training after a small number of pre-training steps.



8. Conclusion

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. For those wishing for a deeper dive, we highly recommend reading the full article and ancillary articles referenced in it.

> **Feel free to pass on any suggestion to improve this notebook in the comment section (if you have any)**

Please give this kernel an UPVOTE to show your appreciation, if you find it useful.

Code Implementation in Tensorflow 2.0

> **Note:** The code for this notebook is taken from the translated version (https://www.kaggle.com/dimitreoliveira/using-tf-2-0-w-bert-on-nq-translated-to-tf2-0) posted by Dimitre Oliviera (https://www.kaggle.com/dimitreoliveira)

This is a translated version of the baseline script (https://www.kaggle.com/philculliton/using-tensorflow-2-0-w-bert-on-nq) from the Tensorflow team

Oliviera translated the script to the Tensorflow 2.0 version, this way we can take part in the TF2 prizes and may use the version to improve the work.

A few notes:

- If you want to keep using **flags** and **logging** you will have to use the **absl** lib (this is recommended by the TF team).
- Since we won't use it with the kernels, he removed most of the **TPU** related stuff to reduce complexity.
- Tensorflow 2 don't let us use global variables **(tf.compat.v1.trainable_variables())**.
- If you have experience with Tensorflow 2 or have any correction/improvement, please let him know.

In this notebook, we'll be using the Bert baseline for Tensorflow to create predictions for the Natural Questions test set. Note that this uses a model that has already been pre-trained - we're only doing inference here. A GPU is required, and this should take between 1-2 hours to run.

The original script can be found here (https://github.com/google-research/language/blob/master/language/question_answering/bert_joint/run_nq.py). The supporting modules were drawn from the official Tensorflow model repository (https://github.com/tensorflow/models/tree/master/official). The bert-joint-baseline data is described here (https://github.com/google-research/language/tree/master/language/question_answering/bert_joint).

Note: This baseline uses code that was migrated from TF1.x. Be aware that it contains use of tf.compat.v1, which is not permitted to be eligible for TF2.0 prizes in this competition (https://www.kaggle.com/c/tensorflow2-question-answering/overview/prizes). It is intended to be used as a starting point, but we're excited to see how much better you can do using TF2.0!

In [1]:

```
import numpy as np
import pandas as pd
import tensorflow as tf
# import tf2_0_baseline_w_bert as tf2baseline # old script
import tf2_0_baseline_w_bert_translated_to_tf2_0 as tf2baseline # Oliviera's script
import bert_modeling as modeling
import bert_optimization as optimization
import bert_tokenization as tokenization
import json
import absl
import sys

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

```
/kaggle/input/tensorflow2-question-answering/simplified-nq-test.jsonl
/kaggle/input/tensorflow2-question-answering/simplified-nq-train.jsonl
/kaggle/input/tensorflow2-question-answering/sample_submission.csv
/kaggle/input/bertjointbaseline/vocab-nq.txt
/kaggle/input/bertjointbaseline/nq-train.tfrecords-00000-of-00001
/kaggle/input/bertjointbaseline/bert_joint.ckpt.data-00000-of-00001
/kaggle/input/bertjointbaseline/bert_config.json
/kaggle/input/bertjointbaseline/bert_joint.ckpt.index
```

Tensorflow flags are variables that can be passed around within the TF system. Every flag below has some context provided regarding what the flag is and how it's used.

Most of these can be changed as desired, with the exception of the Special Flags at the bottom, which *must* stay as-is to work with the Kaggle back end.

In [2]:

```python
def del_all_flags(FLAGS):
    flags_dict = FLAGS._flags()
    keys_list = [keys for keys in flags_dict]
    for keys in keys_list:
        FLAGS.__delattr__(keys)

del_all_flags(absl.flags.FLAGS)

flags = absl.flags

flags.DEFINE_string(
    "bert_config_file", "/kaggle/input/bertjointbaseline/bert_config.json",
    "The config json file corresponding to the pre-trained BERT model. "
    "This specifies the model architecture.")

flags.DEFINE_string("vocab_file", "/kaggle/input/bertjointbaseline/vocab-nq.txt",
                    "The vocabulary file that the BERT model was trained on.")

flags.DEFINE_string(
    "output_dir", "outdir",
    "The output directory where the model checkpoints will be written.")

flags.DEFINE_string("train_precomputed_file", None,
                    "Precomputed tf records for training.")

flags.DEFINE_integer("train_num_precomputed", None,
                     "Number of precomputed tf records for training.")

flags.DEFINE_string(
    "output_prediction_file", "predictions.json",
    "Where to print predictions in NQ prediction format, to be passed to"
    "natural_questions.nq_eval.")

flags.DEFINE_string(
    "init_checkpoint", "/kaggle/input/bertjointbaseline/bert_joint.ckpt",
    "Initial checkpoint (usually from a pre-trained BERT model).")

flags.DEFINE_bool(
    "do_lower_case", True,
    "Whether to lower case the input text. Should be True for uncased "
    "models and False for cased models.")

flags.DEFINE_integer(
    "max_seq_length", 384,
    "The maximum total input sequence length after WordPiece tokenization. "
    "Sequences longer than this will be truncated, and sequences shorter "
    "than this will be padded.")

flags.DEFINE_integer(
    "doc_stride", 128,
    "When splitting up a long document into chunks, how much stride to "
    "take between chunks.")

flags.DEFINE_integer(
    "max_query_length", 64,
    "The maximum number of tokens for the question. Questions longer than "
    "this will be truncated to this length.")

flags.DEFINE_bool("do_train", False, "Whether to run training.")

flags.DEFINE_bool("do_predict", True, "Whether to run eval on the dev set.")

flags.DEFINE_integer("train_batch_size", 32, "Total batch size for training.")

flags.DEFINE_integer("predict_batch_size", 8,
                     "Total batch size for predictions.")

flags.DEFINE_float("learning_rate", 5e-5, "The initial learning rate for Adam.")

flags.DEFINE_float("num_train_epochs", 3.0,
                   "Total number of training epochs to perform.")

flags.DEFINE_float(
    "warmup_proportion", 0.1,
    "Proportion of training to perform linear learning rate warmup for. "
    "E.g., 0.1 = 10% of training.")

flags.DEFINE_integer("save_checkpoints_steps", 1000,
                     "How often to save the model checkpoint.")

flags.DEFINE_integer("iterations_per_loop", 1000,
                     "How many steps to make in each estimator call.")

flags.DEFINE_integer(
    "n_best_size", 20,
    "The total number of n-best predictions to generate in the "
    "nbest_predictions.json output file.")

flags.DEFINE_integer(
    "verbosity", 1, "How verbose our error messages should be")
```

```
flags.DEFINE_integer(
    "max_answer_length", 30,
    "The maximum length of an answer that can be generated. This is needed "
    "because the start and end predictions are not conditioned on one another.")

flags.DEFINE_float(
    "include_unknowns", -1.0,
    "If positive, probability of including answers of type `UNKNOWN`.")

flags.DEFINE_bool("use_tpu", False, "Whether to use TPU or GPU/CPU.")
flags.DEFINE_bool("use_one_hot_embeddings", False, "Whether to use use_one_hot_embeddings")

absl.flags.DEFINE_string(
    "gcp_project", None,
    "[Optional] Project name for the Cloud TPU-enabled project. If not "
    "specified, we will attempt to automatically detect the GCE project from "
    "metadata.")

flags.DEFINE_bool(
    "verbose_logging", False,
    "If true, all of the warnings related to data processing will be printed. "
    "A number of warnings are expected for a normal NQ evaluation.")

flags.DEFINE_boolean(
    "skip_nested_contexts", True,
    "Completely ignore context that are not top level nodes in the page.")

flags.DEFINE_integer("task_id", 0,
                     "Train and dev shard to read from and write to.")

flags.DEFINE_integer("max_contexts", 48,
                     "Maximum number of contexts to output for an example.")

flags.DEFINE_integer(
    "max_position", 50,
    "Maximum context position for which to generate special tokens.")


## Special flags - do not change

flags.DEFINE_string(
    "predict_file", "/kaggle/input/tensorflow2-question-answering/simplified-nq-test.jsonl",
    "NQ json for predictions. E.g., dev-v1.1.jsonl.gz or test-v1.1.jsonl.gz")
flags.DEFINE_boolean("logtostderr", True, "Logs to stderr")
flags.DEFINE_boolean("undefok", True, "it's okay to be undefined")
flags.DEFINE_string('f', '', 'kernel')
flags.DEFINE_string('HistoryManager.hist_file', '', 'kernel')

FLAGS = flags.FLAGS
FLAGS(sys.argv) # Parse the flags
```

```
Out[2]: ['/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py']
```

Here, we:

1. Set up Bert
2. Read in the test set
3. Run it past the pre-built Bert model to create embeddings
4. Use those embeddings to make predictions
5. Write those predictions to `predictions.json`

Feel free to change the code below. Code for the `tf2baseline.*` functions is included in the `tf2_0_baseline_w_bert` utility script, and can be customized, whether by forking the utility script and updating it, or by creating your own non- `tf2baseline` versions in this kernel.

Note: the `tf2_0_baseline_w_bert` utility script contains code for training your own embeddings. Here that code is removed.

In [3]:

```
bert_config = modeling.BertConfig.from_json_file(FLAGS.bert_config_file)

tf2baseline.validate_flags_or_throw(bert_config)
tf.io.gfile.makedirs(FLAGS.output_dir)

tokenizer = tokenization.FullTokenizer(
    vocab_file=FLAGS.vocab_file, do_lower_case=FLAGS.do_lower_case)

run_config = tf.estimator.RunConfig(
    model_dir=FLAGS.output_dir,
    save_checkpoints_steps=FLAGS.save_checkpoints_steps)

num_train_steps = None
num_warmup_steps = None

model_fn = tf2baseline.model_fn_builder(
    bert_config=bert_config,
    init_checkpoint=FLAGS.init_checkpoint,
    learning_rate=FLAGS.learning_rate,
    num_train_steps=num_train_steps,
    num_warmup_steps=num_warmup_steps,
    use_tpu=FLAGS.use_tpu,
    use_one_hot_embeddings=FLAGS.use_one_hot_embeddings)

estimator = tf.estimator.Estimator(
    model_fn=model_fn,
    config=run_config,
    params={'batch_size':FLAGS.train_batch_size})


if FLAGS.do_predict:
  if not FLAGS.output_prediction_file:
    raise ValueError(
        "--output_prediction_file must be defined in predict mode.")

  eval_examples = tf2baseline.read_nq_examples(
      input_file=FLAGS.predict_file, is_training=False)

  print("FLAGS.predict_file", FLAGS.predict_file)

  eval_writer = tf2baseline.FeatureWriter(
      filename=os.path.join(FLAGS.output_dir, "eval.tf_record"),
      is_training=False)
  eval_features = []

  def append_feature(feature):
    eval_features.append(feature)
    eval_writer.process_feature(feature)

  num_spans_to_ids = tf2baseline.convert_examples_to_features(
      examples=eval_examples,
      tokenizer=tokenizer,
      is_training=False,
      output_fn=append_feature)
  eval_writer.close()
  eval_filename = eval_writer.filename

  print("***** Running predictions *****")
  print(f"  Num orig examples = %d" % len(eval_examples))
  print(f"  Num split examples = %d" % len(eval_features))
  print(f"  Batch size = %d" % FLAGS.predict_batch_size)
  for spans, ids in num_spans_to_ids.items():
    print(f"  Num split into %d = %d" % (spans, len(ids)))

  predict_input_fn = tf2baseline.input_fn_builder(
      input_file=eval_filename,
      seq_length=FLAGS.max_seq_length,
      is_training=False,
      drop_remainder=False)

  all_results = []

  for result in estimator.predict(
      predict_input_fn, yield_single_examples=True):
    if len(all_results) % 1000 == 0:
      print("Processing example: %d" % (len(all_results)))

    unique_id = int(result["unique_ids"])
    start_logits = [float(x) for x in result["start_logits"].flat]
    end_logits = [float(x) for x in result["end_logits"].flat]
    answer_type_logits = [float(x) for x in result["answer_type_logits"].flat]

    all_results.append(
        tf2baseline.RawResult(
            unique_id=unique_id,
            start_logits=start_logits,
            end_logits=end_logits,
            answer_type_logits=answer_type_logits))

  print ("Going to candidates file")
```

```
candidates_dict = tf2baseline.read_candidates(FLAGS.predict_file)

print ("setting up eval features")

raw_dataset = tf.data.TFRecordDataset(eval_filename)
eval_features = []
for raw_record in raw_dataset:
  eval_features.append(tf.train.Example.FromString(raw_record.numpy()))

print ("compute_pred_dict")

nq_pred_dict = tf2baseline.compute_pred_dict(candidates_dict, eval_features,
                                 [r._asdict() for r in all_results])
predictions_json = {"predictions": list(nq_pred_dict.values())}

print ("writing json")

with tf.io.gfile.GFile(FLAGS.output_prediction_file, "w") as f:
  json.dump(predictions_json, f, indent=4)
```

```
FLAGS.predict_file /kaggle/input/tensorflow2-question-answering/simplified-nq-test.jsonl
***** Running predictions *****
  Num orig examples = 346
  Num split examples = 9409
  Batch size = 8
  Num split into 3 = 8
  Num split into 19 = 9
  Num split into 50 = 5
  Num split into 2 = 6
  Num split into 34 = 6
  Num split into 54 = 1
  Num split into 40 = 7
  Num split into 42 = 3
  Num split into 22 = 7
  Num split into 11 = 12
  Num split into 29 = 8
  Num split into 102 = 1
  Num split into 60 = 3
  Num split into 10 = 12
  Num split into 21 = 6
  Num split into 41 = 4
  Num split into 6 = 7
  Num split into 35 = 8
  Num split into 23 = 4
  Num split into 32 = 7
  Num split into 17 = 10
  Num split into 85 = 1
  Num split into 30 = 6
  Num split into 9 = 8
  Num split into 1 = 7
  Num split into 57 = 3
  Num split into 5 = 9
  Num split into 28 = 5
  Num split into 31 = 7
  Num split into 18 = 6
  Num split into 47 = 5
  Num split into 4 = 12
  Num split into 67 = 1
  Num split into 45 = 4
  Num split into 27 = 7
  Num split into 8 = 9
  Num split into 63 = 1
  Num split into 43 = 5
  Num split into 13 = 9
  Num split into 12 = 6
  Num split into 16 = 6
  Num split into 24 = 2
  Num split into 14 = 4
  Num split into 53 = 4
  Num split into 20 = 5
  Num split into 15 = 6
  Num split into 7 = 6
  Num split into 44 = 2
  Num split into 112 = 1
  Num split into 37 = 5
  Num split into 46 = 3
  Num split into 39 = 5
  Num split into 87 = 1
  Num split into 48 = 1
  Num split into 33 = 2
  Num split into 66 = 1
  Num split into 49 = 3
  Num split into 62 = 2
  Num split into 125 = 1
  Num split into 36 = 6
  Num split into 26 = 8
  Num split into 76 = 2
  Num split into 121 = 1
  Num split into 38 = 6
  Num split into 55 = 1
  Num split into 25 = 7
  Num split into 56 = 3
  Num split into 82 = 1
  Num split into 58 = 1
  Num split into 98 = 1
  Num split into 52 = 1
  Num split into 89 = 1
  Num split into 73 = 1
  Num split into 187 = 1
Processing example: 0
Processing example: 1000
Processing example: 2000
Processing example: 3000
Processing example: 4000
Processing example: 5000
Processing example: 6000
Processing example: 7000
Processing example: 8000
Processing example: 9000
Going to candidates file
setting up eval features
```

```
compute_pred_dict
Examples processed: 100
Examples processed: 200
Examples processed: 300
writing json
```

Now, we turn `predictions.json` into a `submission.csv` file.

In [4]:
```
test_answers_df = pd.read_json("/kaggle/working/predictions.json")
```

The Bert model produces a `confidence` score, which the Kaggle metric does not use. You, however, can use that score to determine which answers get submitted. See the limits commented out in `create_short_answer` and `create_long_answer` below for an example.

Values for `confidence` will range between `1.0` and `2.0` .

In [5]:
```
def create_short_answer(entry):
    # if entry["short_answers_score"] < 1.5:
    #     return ""

    answer = []
    for short_answer in entry["short_answers"]:
        if short_answer["start_token"] > -1:
            answer.append(str(short_answer["start_token"]) + ":" + str(short_answer["end_token"]))
    if entry["yes_no_answer"] != "NONE":
        answer.append(entry["yes_no_answer"])
    return " ".join(answer)

def create_long_answer(entry):
   # if entry["long_answer_score"] < 1.5:
   # return ""

    answer = []
    if entry["long_answer"]["start_token"] > -1:
        answer.append(str(entry["long_answer"]["start_token"]) + ":" + str(entry["long_answer"]["end_token"
]))
    return " ".join(answer)
```

In [6]:
```
test_answers_df["long_answer_score"] = test_answers_df["predictions"].apply(lambda q: q["long_answer_score"])
test_answers_df["short_answer_score"] = test_answers_df["predictions"].apply(lambda q: q["short_answers_scor
e"])
```

In [7]:
```
test_answers_df["long_answer_score"].describe()
```

Out[7]:
```
count    346.000000
mean       1.243183
std        0.130974
min        0.930659
25%        1.151223
50%        1.237874
75%        1.315764
max        1.661695
Name: long_answer_score, dtype: float64
```

An example of what each sample's answers look like in `prediction.json` :

In [8]:
```
test_answers_df.predictions.values[0]
```

Out[8]:
```
{'example_id': '-1220107454853145579',
 'long_answer': {'start_token': -1,
  'end_token': -1,
  'start_byte': -1,
  'end_byte': -1},
 'long_answer_score': 1.204509004950523,
 'short_answers': [{'start_token': 117,
   'end_token': 148,
   'start_byte': -1,
   'end_byte': -1}],
 'short_answers_score': 1.204509004950523,
 'yes_no_answer': 'NONE'}
```

We re-format the JSON answers to match the requirements for submission.

In [9]:
```
test_answers_df["long_answer"] = test_answers_df["predictions"].apply(create_long_answer)
test_answers_df["short_answer"] = test_answers_df["predictions"].apply(create_short_answer)
test_answers_df["example_id"] = test_answers_df["predictions"].apply(lambda q: str(q["example_id"]))

long_answers = dict(zip(test_answers_df["example_id"], test_answers_df["long_answer"]))
short_answers = dict(zip(test_answers_df["example_id"], test_answers_df["short_answer"]))
```

Then we add them to our sample submission. Recall that each sample has both a `_long` and `_short` entry in the sample submission, one for each type of answer.

In [10]:

```
sample_submission = pd.read_csv("/kaggle/input/tensorflow2-question-answering/sample_submission.csv")

long_prediction_strings = sample_submission[sample_submission["example_id"].str.contains("_long")].apply(lambda q: long_answers[q["example_id"].replace("_long", "")], axis=1)
short_prediction_strings = sample_submission[sample_submission["example_id"].str.contains("_short")].apply(lambda q: short_answers[q["example_id"].replace("_short", "")], axis=1)

sample_submission.loc[sample_submission["example_id"].str.contains("_long"), "PredictionString"] = long_prediction_strings
sample_submission.loc[sample_submission["example_id"].str.contains("_short"), "PredictionString"] = short_prediction_strings
```

And finally, we write out our submission!

In [11]:

```
sample_submission.to_csv("submission.csv", index=False)
sample_submission.head()
```

Out[11]:

	example_id	PredictionString
0	-1011141123527297803_long	
1	-1011141123527297803_short	329:341
2	-1028916936938579349_long	42:321
3	-1028916936938579349_short	247:264
4	-1055197305756217938_long	221:335

Please give this kernel an UPVOTE to show your appreciation, if you find it useful.

Also don't forget to upvote Dimitre's kernel here (https://www.kaggle.com/dimitreoliveira/using-tf-2-0-w-bert-on-nq-translated-to-tf2-0)