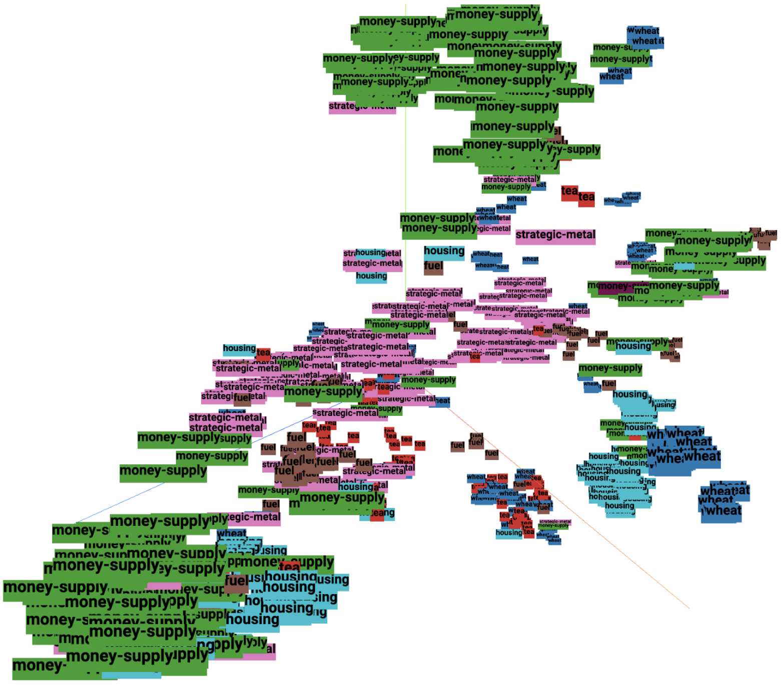
Building a Search Engine with BERT and TensorFlow

In this experiment, we use a pre-trained BERT model checkpoint to build a general-purpose text feature extractor, which we apply to the task of nearest neighbour search.

Denis Antyukhov Follow
Jun 28 · 7 min read ★



T-SNE decomposition of BERT text representations (Reuters-21578 benchmark, 6 classes)

Feature extractors based on deep Neural Probabilistic Language Models such as BERT may extract features that are relevant for a wide array of downstream NLP tasks. For that reason, they are sometimes referred to as Natural Language Understanding (NLU) modules.

These features may also be used for computing the similarity between text samples. To demonstrate this, we will build a nearest neighbour search engine for text, using BERT for feature extraction.

The plan for this experiment is:

1. getting the pre-trained BERT model checkpoint

2. extracting a sub-graph optimized for inference
3. creating a feature extractor with tf.Estimator
4. exploring vector space with T-SNE and Embedding Projector
5. implementing a nearest neighbour search engine
6. accelerating nearest neighbour queries with math
7. example: building a movie recommendation system

Questions and Answers

What is in this guide?

This guide contains implementations of two things: a BERT text feature extractor and a nearest neighbour search engine.

Who is this guide for?

This guide should be useful for researchers interested in using BERT for natural language understanding tasks. It may also serve as a worked example of interfacing with tf.Estimator API.

What does it take?

For a reader familiar with TensorFlow it should take around 30 minutes to finish this guide.

Show me the code.

The code for this experiment is available in Colab here. Also, check out the repository I set up for my BERT experiments: it contains bonus stuff.

Now, let’s start.

Step 1: getting the pre-trained model

We start with a pre-trained BERT checkpoint. For demonstration purposes, I will be using the uncased English model pre-trained by Google engineers.

For configuring and optimizing the graph for inference we will make use of the awesome bert-as-a-service repository. This repository allows for serving BERT models for remote clients over TCP.

Having a remote BERT-server is beneficial in multi-host environments. However, in this part of the experiment we will focus on creating a local (in-process) feature extractor. This is useful if one wishes to avoid additional latency and potential failure modes introduced by a client-server architecture.

Now, let us download the model and install the package.

```
!wget
https://storage.googleapis.com/bert_models/2019_05_30/wwm_un
cased_L-24_H-1024_A-16.zip
!unzip wwm_uncased_L-24_H-1024_A-16.zip
!pip install bert-serving-server --no-deps
```

Step 2: optimizing the inference graph

Normally, to modify the model graph we would have to do some low-level TensorFlow programming. However, thanks to bert-as-a-service, we can configure the inference graph using a simple CLI interface.

```
import os
import tensorflow as tf

from bert_serving.server.graph import optimize_graph
from bert_serving.server.helper import get_args_parser


MODEL_DIR = '/content/wwm_uncased_L-24_H-1024_A-16/' #
GRAPH_DIR = '/content/graph/' #@param {type:"string"}
GRAPH_OUT = 'extractor.pbtxt' #@param {type:"string"}
GPU_MFRAC = 0.2 #@param {type:"string"}

POOL_STRAT = 'REDUCE_MEAN' #@param {type:"string"}
POOL_LAYER = "-2" #@param {type:"string"}
SEQ_LEN = "64" #@param {type:"string"}

tf.gfile.MkDir(GRAPH_DIR)

parser = get_args_parser()
carg = parser.parse_args(args=['-model_dir', MODEL_DIR
                               "-graph_tmp_dir", GRAPH
                               '-max_seq_len', str(SEQ
                               '-pooling_layer', str(P
```

There are a couple of parameters there to look out for.

For each text sample, BERT encoding layers output a tensor of shape [*sequence_len*, *encoder_dim*] with one vector per token. We need to apply some sort of pooling if we are to obtain a fixed representation.

POOL_STRAT parameter defines the pooling strategy applied to the encoding layer number **POOL_LAYER**. The default value '*REDUCE_MEAN*' averages the vectors for all tokens in a sequence. This strategy works best for most sentence-level tasks when the model is not fine-tuned. Another option is *NONE*, in which case no pooling is applied at all. This is useful for word-level tasks such as Named Entity Recognition or POS tagging. For a detailed discussion of these options check out the Han Xiao's blog post.

SEQ_LEN affects the maximum length of sequences processed by the model. Smaller values will increase the model inference speed almost linearly.

Running the above command will put the model graph and weights into a GraphDef object which will be serialized to a *pbtxt* file at **GRAPH_OUT**. The file will often be smaller than the pre-trained model because the nodes and variables required for training will be removed. This results in a very portable solution: for example, the english model only takes 380 MB after serializing.

Step 3: creating a feature extractor

Now, we will use the serialized graph to build a feature extractor using the tf.Estimator API. We will need to define two things: **input_fn** and **model_fn**

input_fn manages getting the data into the model. That includes executing the whole text preprocessing pipeline and preparing a **feed_dict** for BERT.

First, each text sample is converted into a **tf.Example** instance containing the necessary features listed in INPUT_NAMES. The **bert_tokenizer** object contains the WordPiece vocabulary and performs the text preprocessing. After that, the examples are re-grouped by feature name in a **feed_dict**.

```python
INPUT_NAMES = ['input_ids', 'input_mask', 'input_type_
bert_tokenizer = FullTokenizer(VOCAB_PATH)

def build_feed_dict(texts):

    text_features = list(convert_lst_to_features(
        texts, SEQ_LEN, SEQ_LEN,
        bert_tokenizer, log, False, False))

    target_shape = (len(texts), -1)

    feed_dict = {}
    for iname in INPUT_NAMES:
```

tf.Estimators have a fun feature which makes them rebuild and reinitialize the whole computational graph at each call to the predict function. So, in order to avoid the overhead, we will pass a generator to the predict function, and the generator will yield the features to the model in a never-ending loop. Ha-ha.

```
def build_input_fn(container):

    def gen():
        while True:
          try:
            yield build_feed_dict(container.get())
          except StopIteration:
            yield build_feed_dict(container.get())

    def input_fn():
        return tf.data.Dataset.from_generator(
            gen,
            output_types={iname: tf.int32 for iname in
            output_shapes={iname: (None, None) for ina
    return input_fn

class DataContainer:
  def __init__(self):
```

. . .

model_fn contains the specification of the model. In our case, it is loaded from the *pbtxt* file we saved in the previous step. The features are mapped explicitly to the corresponding input nodes via **input_map**.

```
def model_fn(features, mode):
    with tf.gfile.GFile(GRAPH_PATH, 'rb') as f:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(f.read())

    output = tf.import_graph_def(graph_def,
                                 input_map={k + ':0':
                                            for k in I
                                 return_elements=['fir
```

Now we have almost everything we need to perform inference. Let's do this!

```
def batch(iterable, n=1):
    l = len(iterable)
    for ndx in range(0, l, n):
        yield iterable[ndx:min(ndx + n, l)]

def build_vectorizer(_estimator, _input_fn_builder, ba
  container = DataContainer()
  predict_fn = _estimator.predict(_input_fn_builder(co

  def vectorize(text, verbose=False):
    x = []
    bar = Progbar(len(text))
    for text_batch in batch(text, batch_size):
      container.set(text_batch)
      x.append(next(predict_fn)['output'])
```

A standalone version of the feature extractor described above can be found in the repository.

```
>>> bert_vectorizer = build_vectorizer(estimator,
build_input_fn)
>>> bert_vectorizer(64*['sample text']).shape
(64, 768)
```

Step 4: exploring vector space with Projector

Now it's time for a demonstration!

Using the vectorizer, we will generate embeddings for articles from the Reuters-21578 benchmark corpus.
To visualize and explore the embedding vector space in 3D, we will use a dimensionality reduction technique called T-SNE.

Let's get the article embeddings first.

```python
from nltk.corpus import reuters

nltk.download("reuters")
nltk.download("punkt")

max_samples = 256
categories = ['wheat', 'tea', 'strategic-metal',
              'housing', 'money-supply', 'fuel']

S, X, Y = [], [], []

for category in categories:
  print(category)

  sents = reuters.sents(categories=category)
```

The interactive visualization of generated embeddings is available on the Embedding Projector.

From the link you can run T-SNE yourself or load a checkpoint using the bookmark in lower-right corner (loading works only on Chrome).

Step 5: building a search engine

Now, let’s say we have a knowledge base of 50k text samples and we need to answer queries based on this data, fast. How do we retrieve the sample most similar to a query from a text database? The answer is nearest neighbour search.

Formally, the search problem we will be solving is defined as follows: given a set of points **S** in vector space **M**, and a query point **Q** ∈ ***M***, find the closest point in ***S*** to ***Q***. There are multiple ways to define 'closest' in vector space, we will use Euclidean distance.

So, to build an Search Engine for text we will follow these steps:

1. Vectorize all samples from the knowledge base — that gives **S**
2. Vectorize the query — that gives **Q**
3. Compute euclidean distances **D** between **Q** and **S**
4. Sort **D** in ascending order — providing indices of the most similar samples
5. Retrieve labels for said samples from the knowledge base

To make the simple matter of implementing this a bit more exciting, we will do it in pure TensorFlow.

First, we create the placeholders for **Q** and **S**

```
dim = 1024
graph = tf.Graph()
sess = tf.InteractiveSession(graph=graph)

Q = tf.placeholder("float", [dim])
```

Define euclidean distance computation

```
squared_distance = tf.reduce_sum(tf.pow(Q - S, 2), redu
distance = tf.sqrt(squared_distance)
```

Finally, get the most similar sample indices

```
top_k = 3

top_neg_dists, top_indices = tf.math.top_k(tf.negative(
top_dists = tf.negative(top_neg_dists)
```

Step 6: accelerating search with math

Now that we have a basic retrieval algorithm set up, the question is: can we make it run faster? With a tiny bit of math, we can.

For a pair of vectors **p** and **q,** the euclidean distance is defined as follows:

$$d(\mathbf{p},\mathbf{q}) = d(\mathbf{q},\mathbf{p}) = \sqrt{(q_1-p_1)^2 + (q_2-p_2)^2 + \cdots + (q_n-p_n)^2}$$
$$= \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}.$$

Which is exactly how we did compute it in Step 4.

However, since **p** and **q** are vectors, we can expand and rewrite this:

$$d(\mathbf{p},\mathbf{q})^2 = \langle \mathbf{p}-\mathbf{q}, \mathbf{p}-\mathbf{q} \rangle = \langle \mathbf{pp} \rangle - 2\langle \mathbf{pq} \rangle + \langle \mathbf{qq} \rangle$$

where $\langle\ldots\rangle$ denotes inner product.

In TensorFlow this can be written as follows:

```
Q = tf.placeholder("float", [dim])
S = tf.placeholder("float", [None, dim])

Qr = tf.reshape(Q, (1, -1))

PP = tf.keras.backend.batch_dot(S, S, axes=1)
QQ = tf.matmul(Qr, tf.transpose(Qr))
PQ = tf.matmul(S, tf.transpose(Qr))

```

Due to the fact that matrix multiplication op is highly optimized, this implementation works slightly faster than the previous one.

By the way, in the formula above **PP** and **QQ** are actually squared L2 norms of the respective vectors. If both vectors are L2 normalized, then **PP** = **QQ** = 1. That gives an interesting relation between inner product and euclidean distance:

$$d(\mathbf{p}, \mathbf{q})^2 = 2 * (1 - \langle \mathbf{pq} \rangle)$$

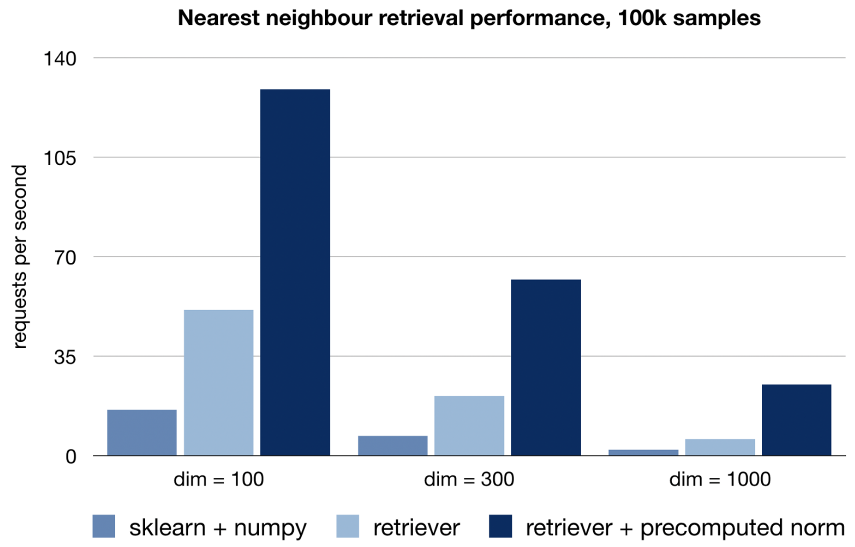
However, doing L2 normalization discards the information about vector magnitude which in many cases is undesirable .

Instead, we may notice that as long as the knowledge base does not change, **PP,** its squared vector norm, also remains constant. So, instead of recomputing it every time we can just do it once and then use the precomputed result further accelerating the distance computation.

Now let us put it all together.

```python
class L2Retriever:
    def __init__(self, dim, top_k=3, use_norm=False, u
        self.dim = dim
        self.top_k = top_k
        self.use_norm = use_norm
        config = tf.ConfigProto(
            device_count={'GPU': (1 if use_gpu else 0)
        )
        config.gpu_options.allow_growth = True
        self.session = tf.Session(config=config)

        self.norm = None
        self.query = tf.placeholder("float", [self.dim
        self.kbase = tf.placeholder("float", [None, se

        self.build_graph()

    def build_graph(self):

        if self.use_norm:
            self.norm = tf.placeholder("float", [None,

        distance = dot_l2_distances(self.kbase, self.q
        top_neg_dists, top_indices = tf.math.top_k(tf.
        top_dists = tf.negative(top_neg_dists)

        self.top_distances = top_dists
        self.top_indices = top_indices

    def predict(self, kbase, query, norm=None):
        query = np.squeeze(query)
        feed_dict = {self.query: query, self.kbase: kb
        if self.use_norm:
          feed_dict[self.norm] = norm

```

Naturally, you could use this implementation with any model, not just BERT. It is quite effective at nearest neighbour retrieval, being able to process dozens of requests per second on CPU.



Example: movie recommendation system

For this example we will use a dataset of movie summaries from IMDB. Using the NLU and Retriever modules, we will build a movie recommendation system that suggests movies with similar plot features.

First, let's download and prepare the IMDB dataset.

```python
import pandas as pd
import json

!wget http://www.cs.cmu.edu/~ark/personas/data/MovieSu
!tar -xvzf MovieSummaries.tar.gz

plots_df = pd.read_csv('MovieSummaries/plot_summaries.
meta_df = pd.read_csv('MovieSummaries/movie.metadata.t

plot = {}
metadata = {}
movie_data = {}

for movie_id, movie_plot in plots_df.values:
  plot[movie_id] = movie_plot

for movie_id, movie_name, movie_genre in meta_df[[0,2,
  genre = list(json.loads(movie_genre).values())
  if len(genre):
    metadata[movie_id] = {"name": movie_name,
                          "genre": genre}
```

Vectorize movie plots with the BERT NLU module:

```python
X_vect = bert_vectorizer(X, verbose=True)
```

Finally, using the L2Retriever, find movies with plot vectors most similar to the query movie, and return it to user.

```python
def buildMovieRecommender(movie_names, vectorized_plot
  retriever = L2Retriever(vectorized_plots.shape[1], u
  vectorized_norm = np.sum(vectorized_plots**2, axis=1

  def recommend(query):
    try:
      idx = retriever.predict(vectorized_plots,
                              vectorized_plots[movie_n
                              vectorized_norm)[0][1:]
      for i in idx:
        print(names[i])
    except ValueError:
```

Let’s check it out!

```
>>> recommend = buildMovieRecommender(names, X_vect)
>>> recommend("The Matrix")
Impostor
Immortel
Saturn 3
Terminator Salvation
The Terminator
Logan's Run
Genesis II
Tron: Legacy
Blade Runner
```

Even without supervision, the model performs adequately on several classification and retrieval tasks. While the performance can be further improved with supervised data, the described approach to text feature extraction provides a solid baseline for downstream NLP solutions.

This concludes the guide to building a search engine with BERT and TensorFlow.