bert-as-service Documentation

Release 1.6.1

Han Xiao

Jul 01, 2021

Contents

`bert-as-service` is a sentence encoding service for mapping a variable-length sentence to a fixed-length vector.

CHAPTER 1

Installation

The best way to install the `bert-as-service` is via pip. Note that the server and client can be installed separately or even on different machines:

```
pip install -U bert-serving-server bert-serving-client
```

Note: The server MUST be running on **Python >= 3.5** with **Tensorflow >= 1.10** (*one-point-ten*). Again, the server does not support Python 2!

Note: The client can be running on both Python 2 and 3.

CHAPTER 2

Table of Content

2.1 What is it

2.1.1 Preliminary

BERT is a NLP model developed by Google for pre-training language representations. It leverages an enormous amount of plain text data publicly available on the web and is trained in an unsupervised manner. Pre-training a BERT model is a fairly expensive yet one-time procedure for each language. Fortunately, Google released several pre-trained models where you can download from here.

Sentence Encoding/Embedding is a upstream task required in many NLP applications, e.g. sentiment analysis, text classification. The goal is to represent a variable length sentence into a fixed length vector, e.g. hello world to [0.1, 0.3, 0.9]. Each element of the vector should "encode" some semantics of the original sentence.

Finally, `bert-as-service` uses BERT as a sentence encoder and hosts it as a service via ZeroMQ, allowing you to map sentences into fixed-length representations in just two lines of code.

2.1.2 Highlights

- State-of-the-art: build on pretrained 12/24-layer BERT models released by Google AI, which is considered as a milestone in the NLP community.
- Easy-to-use: require only two lines of code to get sentence/token-level encodes.
- Fast: 900 sentences/s on a single Tesla M40 24GB. Low latency, optimized for speed. See benchmark.

- Scalable: scale nicely and smoothly on multiple GPUs and multiple clients without worrying about concurrency. See benchmark.

More features: asynchronous encoding, multicasting, mix GPU & CPU workloads, graph optimization, tf.data friendly, customized tokenizer, pooling strategy and layer, XLA support etc.

2.2 Getting Start

2.2.1 Installation

The best way to install the `bert-as-service` is via pip. Note that the server and client can be installed separately or even on different machines:

```
pip install -U bert-serving-server bert-serving-client
```

Warning: The server MUST be running on **Python >= 3.5** with **Tensorflow >= 1.10** (*one-point-ten*). Again, the server does not support Python 2!

Note: The client can be running on both Python 2 and 3.

2.2.2 Download a Pre-trained BERT Model

Download a model listed below, then uncompress the zip file into some folder, say `/tmp/english_L-12_H-768_A-12/`

List of pretrained BERT models released by Google AI:

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

Note: As an optional step, you can also fine-tune the model on your downstream task.

2.2.3 Start the BERT service

After installing the server, you should be able to use *bert-serving-start* CLI as follows:

```
bert-serving-start -model_dir /tmp/english_L-12_H-768_A-12/ -num_worker=4
```

This will start a service with four workers, meaning that it can handle up to four **concurrent** requests. More concurrent requests will be queued in a load balancer.

Below shows what the server looks like when starting correctly:

Start the Bert service in a docker container

Alternatively, one can start the BERT Service in a Docker Container:

```
docker build -t bert-as-service -f ./docker/Dockerfile .
NUM_WORKER=1
PATH_MODEL=/PATH_TO/_YOUR_MODEL/
docker run --runtime nvidia -dit -p 5555:5555 -p 5556:5556 -v $PATH_MODEL:/model -t␣
↪bert-as-service $NUM_WORKER
```

2.2.4 Use Client to Get Sentence Encodes

Now you can encode sentences simply as follows:

```
from bert_serving.client import BertClient
bc = BertClient()
bc.encode(['First do it', 'then do it right', 'then do it better'])
```

It will return a `ndarray`, in which each row is the fixed representation of a sentence. You can also let it return a pure python object with type `List[List[float]]`.

As a feature of BERT, you may get encodes of a pair of sentences by concatenating them with `|||`, e.g.

```
bc.encode(['First do it ||| then do it right'])
```

Below shows what the server looks like while encoding:

Use BERT Service Remotely

One may also start the service on one (GPU) machine and call it from another (CPU) machine as follows:

```
# on another CPU machine
from bert_serving.client import BertClient
bc = BertClient(ip='xx.xx.xx.xx')  # ip address of the GPU machine
bc.encode(['First do it', 'then do it right', 'then do it better'])
```

Note: You only need `pip install -U bert-serving-client` in this case, the server side is not required.

Want to learn more? Checkout our tutorials below:

2.3 Tutorials

The full list of examples can be found in here. You can run each via `python example/example-k.py`. Most of examples require you to start a BertServer first.

Note: Although `BertClient` works universally on both Python 2.x and 3.x, examples are only tested on Python 3.6.

2.3.1 Building a QA semantic search engine in 3 minutes

Note: The complete example can be found example8.py.

As the first example, we will implement a simple QA search engine using `bert-as-service` in just three minutes. No kidding! The goal is to find similar questions to user's input and return the corresponding answer. To start, we need a list of question-answer pairs. Fortunately, this README file already contains *a list of FAQ*, so I will just use that to make this example perfectly self-contained. Let's first load all questions and show some statistics.

```
prefix_q = '##### **Q:** '
with open('README.md') as fp:
    questions = [v.replace(prefix_q, '').strip() for v in fp if v.strip() and v.
↪startswith(prefix_q)]
    print('%d questions loaded, avg. len of %d' % (len(questions), np.mean([len(d.
↪split()) for d in questions])))
```

This gives `33 questions loaded, avg. len of 9`. So looks like we have enough questions. Now start a BertServer with `uncased_L-12_H-768_A-12` pretrained BERT model:

```
bert-serving-start -num_worker=1 -model_dir=/data/cips/data/lab/data/model/uncased_L-
↪12_H-768_A-12
```

Next, we need to encode our questions into vectors:

```
bc = BertClient(port=5555, port_out=5556)
doc_vecs = bc.encode(questions)
```

Finally, we are ready to receive new query and perform a simple "fuzzy" search against the existing questions. To do that, every time a new query is coming, we encode it as a vector and compute its dot product with `doc_vecs`; sort the result descendingly; and return the top-k similar questions as follows:

```
while True:
    query = input('your question: ')
    query_vec = bc.encode([query])[0]
    # compute normalized dot product as score
    score = np.sum(query_vec * doc_vecs, axis=1) / np.linalg.norm(doc_vecs, axis=1)
    topk_idx = np.argsort(score)[::-1][:topk]
    for idx in topk_idx:
        print('> %s\t%s' % (score[idx], questions[idx]))
```

That's it! Now run the code and type your query, see how this search engine handles fuzzy match:

2.3.2 Serving a fine-tuned BERT model

Pretrained BERT models often show quite "okayish" performance on many tasks. However, to release the true power of BERT a fine-tuning on the downstream task (or on domain-specific data) is necessary. In this example, I will show you how to serve a fine-tuned BERT model.

We follow the instruction in "Sentence (and sentence-pair) classification tasks" and use `run_classifier.py` to fine tune `uncased_L-12_H-768_A-12` model on MRPC task. The fine-tuned model is stored at `/tmp/mrpc_output/`, which can be changed by specifying `--output_dir` of `run_classifier.py`.

If you look into `/tmp/mrpc_output/`, it contains something like:

```
checkpoint                                        128
eval                                              4.0K
eval_results.txt                                  86
eval.tf_record                                    219K
events.out.tfevents.1545202214.TENCENT64.site     6.1M
events.out.tfevents.1545203242.TENCENT64.site     14M
graph.pbtxt                                       9.0M
model.ckpt-0.data-00000-of-00001                  1.3G
model.ckpt-0.index                                23K
model.ckpt-0.meta                                 3.9M
model.ckpt-343.data-00000-of-00001                1.3G
model.ckpt-343.index                              23K
model.ckpt-343.meta                               3.9M
train.tf_record                                   2.0M
```

Don't be afraid of those mysterious files, as the only important one to us is `model.ckpt-343.data-00000-of-00001` (looks like my training stops at the 343 step. One may get `model.ckpt-123.data-00000-of-00001` or `model.ckpt-9876.data-00000-of-00001` depending on the total training steps). Now we have collected all three pieces of information that are needed for serving this fine-tuned model:

- The pretrained model is downloaded to `/path/to/bert/uncased_L-12_H-768_A-12`
- Our fine-tuned model is stored at `/tmp/mrpc_output/`;
- Our fine-tuned model checkpoint is named as `model.ckpt-343` something something.

Now start a BertServer by putting three pieces together:

```
bert-serving-start -model_dir=/pretrained/uncased_L-12_H-768_A-12 -tuned_model_dir=/
↪tmp/mrpc_output/ -ckpt_name=model.ckpt-343
```

After the server started, you should find this line in the log:

```
I:GRAPHOPT:[gra:opt: 50]:checkpoint (override by fine-tuned model): /tmp/mrpc_output/
↪model.ckpt-343
```

Which means the BERT parameters is overrode and successfully loaded from our fine-tuned `/tmp/mrpc_output/model.ckpt-343`. Done!

In short, find your fine-tuned model path and checkpoint name, then feed them to `-tuned_model_dir` and `-ckpt_name`, respectively.

2.3.3 Getting ELMo-like contextual word embedding

Start the server with `pooling_strategy` set to NONE.

```
bert-serving-start -pooling_strategy NONE -model_dir /tmp/english_L-12_H-768_A-12/
```

To get the word embedding corresponds to every token, you can simply use slice index as follows:

```
# max_seq_len = 25
# pooling_strategy = NONE

bc = BertClient()
vec = bc.encode(['hey you', 'whats up?'])

vec  # [2, 25, 768]
vec[0]  # [1, 25, 768], sentence embeddings for `hey you`
vec[0][0]  # [1, 1, 768], word embedding for `[CLS]`
vec[0][1]  # [1, 1, 768], word embedding for `hey`
vec[0][2]  # [1, 1, 768], word embedding for `you`
vec[0][3]  # [1, 1, 768], word embedding for `[SEP]`
vec[0][4]  # [1, 1, 768], word embedding for padding symbol
vec[0][25]  # error, out of index!
```

Note that no matter how long your original sequence is, the service will always return a `[max_seq_len, 768]` matrix for every sequence. When using slice index to get the word embedding, beware of the special tokens padded to the sequence, i.e. `[CLS]`, `[SEP]`, `0_PAD`.

2.3.4 Using your own tokenizer

Often you want to use your own tokenizer to segment sentences instead of the default one from BERT. Simply call `encode(is_tokenized=True)` on the client slide as follows:

```
texts = ['hello world!', 'good day']

# a naive whitespace tokenizer
texts2 = [s.split() for s in texts]

vecs = bc.encode(texts2, is_tokenized=True)
```

This gives `[2, 25, 768]` tensor where the first `[1, 25, 768]` corresponds to the token-level encoding of "hello world!". If you look into its values, you will find that only the first four elements, i.e. `[1, 0:3, 768]` have values, all the others are zeros. This is due to the fact that BERT considers "hello world!" as four tokens: `[CLS] hello world! [SEP]`, the rest are padding symbols and are masked out before output.

Note: There is no need to start a separate server for handling tokenized/untokenized sentences. The server can tell and handle both cases automatically.

Warning: The pretrained BERT Chinese from Google is character-based, i.e. its vocabulary is made of single Chinese characters. Therefore it makes no sense if you use word-level segmentation algorithm to pre-process the data and feed to such model.

2.3.5 Using BertClient with `tf.data` API

Note: The complete example can be found example4.py. There is also an example in Keras.

The `tf.data` API enables you to build complex input pipelines from simple, reusable pieces. One can also use `BertClient` to encode sentences on-the-fly and use the vectors in a downstream model. Here is an example:

```
batch_size = 256
num_parallel_calls = 4

# start a thead-safe client to support num_parallel_calls in tf.data API
bc = ConcurrentBertClient(num_parallel_calls)


def get_encodes(x):
    # x is `batch_size` of lines, each of which is a json object
    samples = [json.loads(l) for l in x]
    text = [s['raw_text'] for s in samples]  # List[List[str]]
    labels = [s['label'] for s in samples]  # List[str]
    features = bc.encode(text)
    return features, labels


ds = (tf.data.TextLineDataset(train_fp).batch(batch_size)
        .map(lambda x: tf.py_func(get_encodes, [x], [tf.float32, tf.string]),  num_
↪parallel_calls=num_parallel_calls)
        .map(lambda x, y: {'feature': x, 'label': y})
        .make_one_shot_iterator().get_next())
```

The trick here is to start a pool of `BertClient` and reuse them one by one. In this way, we can fully harness the power of `num_parallel_calls` in `Dataset.map()` API.

2.3.6 Training a text classifier using BERT features and `tf.estimator` API

Note: The complete example can be found example5.py, in which a simple MLP is built on BERT features for predicting the relevant articles according to the fact description in the law documents. The problem is a part of the

Chinese AI and Law Challenge Competition.

Following the last example, we can easily extend it to a full classifier using `tf.estimator` API. One only need minor change on the input function as follows:

```
estimator = DNNClassifier(
    hidden_units=[512],
    feature_columns=[tf.feature_column.numeric_column('feature', shape=(768,))],
    n_classes=len(laws),
    config=run_config,
    label_vocabulary=laws_str,
    dropout=0.1)

input_fn = lambda fp: (tf.data.TextLineDataset(fp)
                       .apply(tf.contrib.data.shuffle_and_repeat(buffer_size=10000))
                       .batch(batch_size)
                       .map(lambda x: tf.py_func(get_encodes, [x], [tf.float32, tf.
↪string]), num_parallel_calls=num_parallel_calls)
                       .map(lambda x, y: ({'feature': x}, y))
                       .prefetch(20))

train_spec = TrainSpec(input_fn=lambda: input_fn(train_fp))
eval_spec = EvalSpec(input_fn=lambda: input_fn(eval_fp), throttle_secs=0)
train_and_evaluate(estimator, train_spec, eval_spec)
```

2.3.7 Saving and loading with TFRecord data

Note: The complete example can be found example6.py.

The TFRecord file format is a simple record-oriented binary format that many TensorFlow applications use for training data. You can also pre-encode all your sequences and store their encodings to a TFRecord file, then later load it to build a `tf.Dataset`. For example, to write encoding into a TFRecord file:

```
bc = BertClient()
list_vec = bc.encode(lst_str)
list_label = [0 for _ in lst_str]  # a dummy list of all-zero labels

# write to tfrecord
with tf.python_io.TFRecordWriter('tmp.tfrecord') as writer:
    def create_float_feature(values):
        return tf.train.Feature(float_list=tf.train.FloatList(value=values))

    def create_int_feature(values):
        return tf.train.Feature(int64_list=tf.train.Int64List(value=list(values)))

    for (vec, label) in zip(list_vec, list_label):
        features = {'features': create_float_feature(vec), 'labels': create_int_
↪feature([label])}
        tf_example = tf.train.Example(features=tf.train.Features(feature=features))
        writer.write(tf_example.SerializeToString())
```

Now we can load from it and build a `tf.Dataset`:

```
def _decode_record(record):
    """Decodes a record to a TensorFlow example."""
    return tf.parse_single_example(record, {
        'features': tf.FixedLenFeature([768], tf.float32),
        'labels': tf.FixedLenFeature([], tf.int64),
    })

ds = (tf.data.TFRecordDataset('tmp.tfrecord').repeat().shuffle(buffer_size=100).apply(
    tf.contrib.data.map_and_batch(lambda record: _decode_record(record), batch_
↪size=64))
      .make_one_shot_iterator().get_next())
```

To save word/token-level embedding to TFRecord, one needs to first flatten `[max_seq_len, num_hidden]` tensor into an 1D array as follows:

```
def create_float_feature(values):
    return tf.train.Feature(float_list=tf.train.FloatList(value=values.reshape(-1)))
```

And later reconstruct the shape when loading it:

```
name_to_features = {
    "feature": tf.FixedLenFeature([max_seq_length * num_hidden], tf.float32),
    "label_ids": tf.FixedLenFeature([], tf.int64),
}

def _decode_record(record, name_to_features):
    """Decodes a record to a TensorFlow example."""
    example = tf.parse_single_example(record, name_to_features)
    example['feature'] = tf.reshape(example['feature'], [max_seq_length, -1])
    return example
```

Be careful, this will generate a huge TFRecord file.

2.3.8 Asynchronous encoding

Note: The complete example can be found example2.py.

`BertClient.encode()` offers a nice synchronous way to get sentence encodes. However, sometimes we want to do it in an asynchronous manner by feeding all textual data to the server first, fetching the encoded results later. This can be easily done by:

```
# an endless data stream, generating data in an extremely fast speed
def text_gen():
    while True:
        yield lst_str  # yield a batch of text lines

bc = BertClient()

# get encoded vectors
for j in bc.encode_async(text_gen(), max_num_batch=10):
    print('received %d x %d' % (j.shape[0], j.shape[1]))
```

2.3.9 Broadcasting to multiple clients

Note: The complete example can be found example3.py.

The encoded result is routed to the client according to its identity. If you have multiple clients with same identity, then they all receive the results! You can use this *multicast* feature to do some cool things, e.g. training multiple different models (some using `scikit-learn` some using `tensorflow`) in multiple separated processes while only call `BertServer` once. In the example below, `bc` and its two clones will all receive encoded vector.

```
# clone a client by reusing the identity
def client_clone(id, idx):
    bc = BertClient(identity=id)
    for j in bc.listen():
        print('clone-client-%d: received %d x %d' % (idx, j.shape[0], j.shape[1]))

bc = BertClient()
# start two cloned clients sharing the same identity as bc
for j in range(2):
    threading.Thread(target=client_clone, args=(bc.identity, j)).start()

for _ in range(3):
    bc.encode(lst_str)
```

2.3.10 Monitoring the service status in a dashboard

Note: The complete example can be found in plugin/dashboard/.

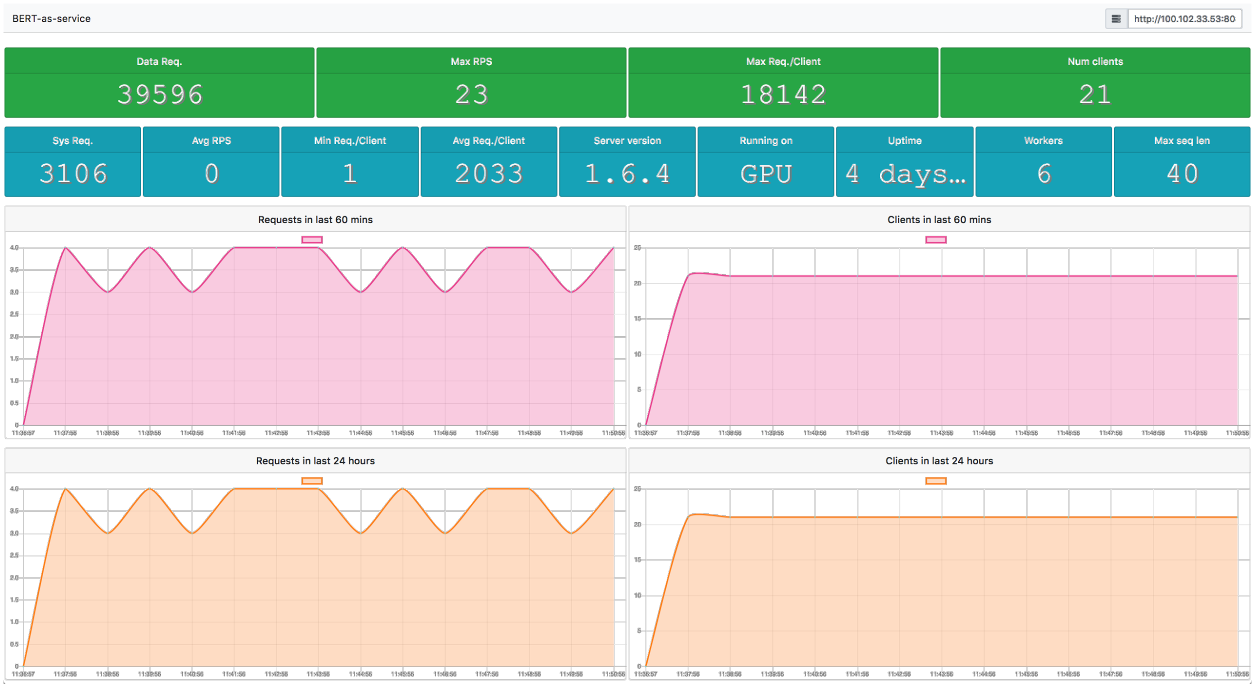
As a part of the infrastructure, one may also want to monitor the service status and show it in a dashboard. To do that, we can use:

```
bc = BertClient(ip='server_ip')

json.dumps(bc.server_status, ensure_ascii=False)
```

This gives the current status of the server including number of requests, number of clients, etc. in JSON format. The only thing remained is to start a HTTP server for returning this JSON to the frontend that renders it.

`plugin/dashboard/index.html` shows a simple dashboard based on Bootstrap and Vue.js.



2.3.11 Using `bert-as-service` to serve HTTP requests in JSON

Besides calling `bert-as-service` from Python, one can also call it via HTTP request in JSON. It is quite useful especially when low transport layer is prohibited. Behind the scene, `bert-as-service` spawns a Flask server in a separate process and then reuse a `BertClient` instance as a proxy to communicate with the ventilator.

To enable this feature, we need to first install some Python dependencies:

```
pip install -U bert-serving-client flask flask-compress flask-cors flask-json
```

Then simply start the server with:

```
bert-serving-start -model_dir=/YOUR_MODEL -http_port 8125
```

Your server is now listening HTTP and TCP requests at port `8125` simultaneously!

To send a HTTP request, first package payload in JSON as following:

```
{
    "id": 123,
    "texts": ["hello world", "good day!"],
    "is_tokenized": false
}
```

, where `id` is a unique identifier helping you to synchronize the results; `is_tokenized` follows the meaning in `` `BertClient `` API‘_ and `false` by default.

Then simply call the server via HTTP POST request. You can use javascript or whatever, here is an example using `curl`:

```
curl -X POST http://xx.xx.xx.xx:8125/encode \
  -H 'content-type: application/json' \
  -d '{"id": 123,"texts": ["hello world"], "is_tokenized": false}'
```

, which returns a JSON:


```
{
    "id": 123,
    "results": [[768 float-list], [768 float-list]],
    "status": 200
}
```


To get the server's status and client's status, you can send GET requests at `/status/server` and `/status/client`, respectively.

Finally, one may also config CORS to restrict the public access of the server by specifying `-cors` when starting `bert-serving-start`. By default `-cors=*`, meaning the server is public accessible.

2.4 Using BertClient

2.4.1 Installation

The best way to install the client is via pip. Note that the client can be installed separately from BertServer or even on a different machine:

```
pip install bert-serving-client
```

Note: The client can be running on both Python 2 and 3.

2.4.2 Client-side API

class client.**BertClient**(*ip='localhost'*, *port=5555*, *port_out=5556*, *output_fmt='ndarray'*, *show_server_config=False*, *identity=None*, *check_version=True*, *check_length=True*, *check_token_info=True*, *ignore_all_checks=False*, *timeout=-1*)

Bases: `object`

A client object connected to a BertServer

Create a BertClient that connects to a BertServer. Note, server must be ready at the moment you are calling this function. If you are not sure whether the server is ready, then please set *ignore_all_checks=True*

You can also use it as a context manager:

```
with BertClient() as bc:
    bc.encode(...)

# bc is automatically closed out of the context
```

Parameters

- **ip** (*str*) – the ip address of the server
- **port** (*int*) – port for pushing data from client to server, must be consistent with the server side config
- **port_out** (*int*) – port for publishing results from server to client, must be consistent with the server side config
- **output_fmt** (*str*) – the output format of the sentence encodes, either in numpy array or python List[List[float]] (ndarray/list)
- **show_server_config** (*bool*) – whether to show server configs when first connected
- **identity** (*str*) – the UUID of this client
- **check_version** (*bool*) – check if server has the same version as client, raise AttributeError if not the same
- **check_length** (*bool*) – check if server *max_seq_len* is less than the sentence length before sent
- **check_token_info** (*bool*) – check if server can return tokenization
- **ignore_all_checks** (*bool*) – ignore all checks, set it to True if you are not sure whether the server is ready when constructing BertClient()
- **timeout** (*int*) – set the timeout (milliseconds) for receive operation on the client, -1 means no timeout and wait until result returns

close()

Gently close all connections of the client. If you are using BertClient as context manager, then this is not necessary.

encode(*texts*, *blocking=True*, *is_tokenized=False*, *show_tokens=False*)

Encode a list of strings to a list of vectors

texts should be a list of strings, each of which represents a sentence. If *is_tokenized* is set to True, then *texts* should be list[list[str]], outer list represents sentence and inner list represent tokens in the sentence. Note that if *blocking* is set to False, then you need to fetch the result manually afterwards.

```
with BertClient() as bc:
    # encode untokenized sentences
    bc.encode(['First do it',
              'then do it right',
              'then do it better'])

    # encode tokenized sentences
    bc.encode([['First', 'do', 'it'],
               ['then', 'do', 'it', 'right'],
               ['then', 'do', 'it', 'better']], is_tokenized=True)
```

Parameters

- **is_tokenized** (*bool*) – whether the input texts is already tokenized
- **show_tokens** (*bool*) – whether to include tokenization result from the server. If true, the return of the function will be a tuple
- **texts** (*list[str] or list[list[str]]*) – list of sentence to be encoded. Larger list for better efficiency.

- **blocking** (*bool*) – wait until the encoded result is returned from the server. If false, will immediately return.
- **timeout** (*bool*) – throw a timeout error when the encoding takes longer than the predefined timeout.

Returns encoded sentence/token-level embeddings, rows correspond to sentences

Return type numpy.ndarray or list[list[float]]

encode_async(*batch_generator*, *max_num_batch=None*, *delay=0.1*, ***kwargs*)

Async encode batches from a generator

Parameters

- **delay** – delay in seconds and then run fetcher
- **batch_generator** – a generator that yields list[str] or list[list[str]] (for *is_tokenized=True*) every time
- **max_num_batch** – stop after encoding this number of batches
- ****kwargs** – the rest parameters please refer to *encode()*

Returns a generator that yields encoded vectors in ndarray, where the request id can be used to determine the order

Return type Iterator[tuple(int, numpy.ndarray)]

fetch(*delay=0.0*)

Fetch the encoded vectors from server, use it with *encode(blocking=False)*

Use it after *encode(texts, blocking=False)*. If there is no pending requests, will return None. Note that *fetch()* does not preserve the order of the requests! Say you have two non-blocking requests, R1 and R2, where R1 with 256 samples, R2 with 1 samples. It could be that R2 returns first.

To fetch all results in the original sending order, please use *fetch_all(sort=True)*

Parameters **delay** (*float*) – delay in seconds and then run fetcher

Returns a generator that yields request id and encoded vector in a tuple, where the request id can be used to determine the order

Return type Iterator[tuple(int, numpy.ndarray)]

fetch_all(*sort=True*, *concat=False*)

Fetch all encoded vectors from server, use it with *encode(blocking=False)*

Use it *encode(texts, blocking=False)*. If there is no pending requests, it will return None.

Parameters

- **sort** (*bool*) – sort results by their request ids. It should be True if you want to preserve the sending order
- **concat** (*bool*) – concatenate all results into one ndarray

Returns encoded sentence/token-level embeddings in sending order

Return type numpy.ndarray or list[list[float]]

server_config

Get the current configuration of the server connected to this client

Returns a dictionary contains the current configuration of the server connected to this client

Return type dict[str, str]

server_status

Get the current status of the server connected to this client

Returns a dictionary contains the current status of the server connected to this client

Return type dict[str, str]

status

Get the status of this BertClient instance

Return type dict[str, str]

Returns a dictionary contains the status of this BertClient instance

class client.**ConcurrentBertClient**(*max_concurrency=10*, ***kwargs*)

Bases: *client.BertClient*

A thread-safe client object connected to a BertServer

Create a BertClient that connects to a BertServer. Note, server must be ready at the moment you are calling this function. If you are not sure whether the server is ready, then please set *check_version=False* and *check_length=False*

Parameters **max_concurrency** (*int*) – the maximum number of concurrent connections allowed

close()

Gently close all connections of the client. If you are using BertClient as context manager, then this is not necessary.

encode(***kwargs*)

Encode a list of strings to a list of vectors

texts should be a list of strings, each of which represents a sentence. If *is_tokenized* is set to True, then *texts* should be list[list[str]], outer list represents sentence and inner list represent tokens in the sentence. Note that if *blocking* is set to False, then you need to fetch the result manually afterwards.

```
with BertClient() as bc:
    # encode untokenized sentences
    bc.encode(['First do it',
               'then do it right',
               'then do it better'])

    # encode tokenized sentences
    bc.encode([['First', 'do', 'it'],
                ['then', 'do', 'it', 'right'],
                ['then', 'do', 'it', 'better']], is_tokenized=True)
```

Parameters

- **is_tokenized** (*bool*) – whether the input texts is already tokenized
- **show_tokens** (*bool*) – whether to include tokenization result from the server. If true, the return of the function will be a tuple
- **texts** (*list[str] or list[list[str]]*) – list of sentence to be encoded. Larger list for better efficiency.

- **blocking** (*bool*) – wait until the encoded result is returned from the server. If false, will immediately return.
- **timeout** (*bool*) – throw a timeout error when the encoding takes longer than the predefined timeout.

Returns encoded sentence/token-level embeddings, rows correspond to sentences

Return type numpy.ndarray or list[list[float]]

encode_async(***kwargs*)
Async encode batches from a generator

Parameters

- **delay** – delay in seconds and then run fetcher
- **batch_generator** – a generator that yields list[str] or list[list[str]] (for *is_tokenized=True*) every time
- **max_num_batch** – stop after encoding this number of batches
- ****kwargs** – the rest parameters please refer to *encode()*

Returns a generator that yields encoded vectors in ndarray, where the request id can be used to determine the order

Return type Iterator[tuple(int, numpy.ndarray)]

fetch(***kwargs*)
Fetch the encoded vectors from server, use it with *encode(blocking=False)*

Use it after *encode(texts, blocking=False)*. If there is no pending requests, will return None. Note that *fetch()* does not preserve the order of the requests! Say you have two non-blocking requests, R1 and R2, where R1 with 256 samples, R2 with 1 samples. It could be that R2 returns first.

To fetch all results in the original sending order, please use *fetch_all(sort=True)*

Parameters **delay** (*float*) – delay in seconds and then run fetcher

Returns a generator that yields request id and encoded vector in a tuple, where the request id can be used to determine the order

Return type Iterator[tuple(int, numpy.ndarray)]

fetch_all(***kwargs*)
Fetch all encoded vectors from server, use it with *encode(blocking=False)*

Use it *encode(texts, blocking=False)*. If there is no pending requests, it will return None.

Parameters

- **sort** (*bool*) – sort results by their request ids. It should be True if you want to preserve the sending order
- **concat** (*bool*) – concatenate all results into one ndarray

Returns encoded sentence/token-level embeddings in sending order

Return type numpy.ndarray or list[list[float]]

server_config
Get the current configuration of the server connected to this client

Returns a dictionary contains the current configuration of the server connected to this client

Return type dict[str, str]

server_status
Get the current status of the server connected to this client

Returns a dictionary contains the current status of the server connected to this client

Return type dict[str, str]

status
Get the status of this BertClient instance

Return type dict[str, str]

Returns a dictionary contains the status of this BertClient instance

2.5 Using BertServer

2.5.1 Installation

The best way to install the server is via pip. Note that the server can be installed separately from BertClient or even on a different machine:

```
pip install bert-serving-server
```

Warning: The server MUST be running on **Python >= 3.5** with **Tensorflow >= 1.10** (*one-point-ten*). Again, the server does not support Python 2!

2.5.2 Command Line Interface

Once installed, you can use the command line interface to start a bert server:

```
bert-serving-server -model_dir /uncased_bert_model -num_worker 4
```

2.5.3 Server-side API

Server-side is a CLI `bert-serving-start`, you can get the latest usage via:

```
bert-serving-start --help
```

Start a BertServer for serving

```
usage: bert-serving-server [-h] -model_dir MODEL_DIR
                           [-tuned_model_dir TUNED_MODEL_DIR]
                           [-ckpt_name CKPT_NAME] [-config_name CONFIG_NAME]
                           [-graph_tmp_dir GRAPH_TMP_DIR]
                           [-max_seq_len MAX_SEQ_LEN] [-cased_tokenization]
                           [-pooling_layer POOLING_LAYER [POOLING_LAYER ...]]
                           [-pooling_strategy {NONE,REDUCE_MAX,REDUCE_MEAN,REDUCE_
↪MEAN_MAX,FIRST_TOKEN,LAST_TOKEN,CLS_POOLED,CLASSIFICATION,REGRESSION}]
                           [-mask_cls_sep] [-no_special_token]
                           [-show_tokens_to_client] [-no_position_embeddings]
                           [-num_labels NUM_LABELS] [-port PORT]
                           [-port_out PORT_OUT] [-http_port HTTP_PORT]
                           [-http_max_connect HTTP_MAX_CONNECT] [-cors CORS]
                           [-num_worker NUM_WORKER]
                           [-max_batch_size MAX_BATCH_SIZE]
                           [-priority_batch_size PRIORITY_BATCH_SIZE] [-cpu]
                           [-xla] [-fp16]
                           [-gpu_memory_fraction GPU_MEMORY_FRACTION]
                           [-device_map DEVICE_MAP [DEVICE_MAP ...]]
                           [-prefetch_size PREFETCH_SIZE]
                           [-fixed_embed_length] [-verbose] [-version]
```

Named Arguments

-verbose turn on tensorflow logging for debug

Default: False

-version show program's version number and exit

File Paths

config the path, checkpoint and filename of a pretrained/fine-tuned BERT model

-model_dir directory of a pretrained BERT model

-tuned_model_dir directory of a fine-tuned BERT model

-ckpt_name filename of the checkpoint file. By default it is "bert_model.ckpt", but for a fine-tuned model the name could be different.

Default: "bert_model.ckpt"

-config_name filename of the JSON config file for BERT model.

Default: "bert_config.json"

-graph_tmp_dir path to graph temp file

BERT Parameters

config how BERT model and pooling works

-max_seq_len maximum length of a sequence, longer sequence will be trimmed on the right side. set it to NONE for dynamically using the longest sequence in a (mini)batch.

Default: 25

-cased_tokenization Whether tokenizer should skip the default lowercasing and accent removal.Should be used for e.g. the multilingual cased pretrained BERT model.

Default: True

-pooling_layer the encoder layer(s) that receives pooling. Give a list in order to concatenate several layers into one

Default: [-2]

-pooling_strategy Possible choices: NONE, REDUCE_MAX, REDUCE_MEAN, REDUCE_MEAN_MAX, FIRST_TOKEN, LAST_TOKEN, CLS_POOLED, CLASSIFICATION, REGRESSION

the pooling strategy for generating encoding vectors

Default: REDUCE_MEAN

-mask_cls_sep masking the embedding on [CLS] and [SEP] with zero. When pooling_strategy is in {CLS_TOKEN, FIRST_TOKEN, SEP_TOKEN, LAST_TOKEN} then the embedding is preserved, otherwise the embedding is masked to zero before pooling

Default: False

-no_special_token add [CLS] and [SEP] in every sequence, put sequence to the model without [CLS] and [SEP] when True and is_tokenized=True in Client

Default: False

-show_tokens_to_client sending tokenization results to client

Default: False

-no_position_embeddings Whether to add position embeddings for the position of each token in the sequence.

Default: False

-num_labels Numbers of Label

Default: 2

Serving Configs

config how server utilizes GPU/CPU resources

-port, -port_in, -port_data server port for receiving data from client

Default: 5555

-port_out, -port_result server port for sending result to client

Default: 5556

-http_port server port for receiving HTTP requests

-http_max_connect maximum number of concurrent HTTP connections

Default: 10

-cors setting "Access-Control-Allow-Origin" for HTTP requests

Default: "*"

-num_worker number of server instances

Default: 1

-max_batch_size maximum number of sequences handled by each worker

Default: 256

-priority_batch_size batch smaller than this size will be labeled as high priority,and jumps forward in the job queue

Default: 16

-cpu running on CPU (default on GPU)

Default: False

-xla enable XLA compiler (experimental)

Default: False

-fp16 use float16 precision (experimental)

Default: False

-gpu_memory_fraction determine the fraction of the overall amount of memory that each visible GPU should be allocated per worker. Should be in range [0.0, 1.0]

Default: 0.5

-device_map specify the list of GPU device ids that will be used (id starts from 0). If num_worker > len(device_map), then device will be reused; if num_worker < len(device_map), then device_map[:num_worker] will be used

Default: []

-prefetch_size the number of batches to prefetch on each worker. When running on a CPU-only machine, this is set to 0 for comparability

Default: 10

-fixed_embed_length when "max_seq_len" is set to None, the server determines the "max_seq_len" according to the actual sequence lengths within each batch. When "pooling_strategy=NONE", this may cause two ".encode()" from the same client results in different sizes [B, T, D].Turn this on to fix the "T" in [B, T, D] to "max_position_embeddings" in bert json config.

Default: False

2.5.4 Server-side Benchmark

If you want to benchmark the speed, you may use:

```
bert-serving-benchmark --help
```

Benchmark BertServer locally

```
usage: bert-serving-benchmark [-h] -model_dir MODEL_DIR
                              [-tuned_model_dir TUNED_MODEL_DIR]
                              [-ckpt_name CKPT_NAME]
                              [-config_name CONFIG_NAME]
                              [-graph_tmp_dir GRAPH_TMP_DIR]
                              [-max_seq_len MAX_SEQ_LEN] [-cased_tokenization]
                              [-pooling_layer POOLING_LAYER [POOLING_LAYER ...]]
                              [-pooling_strategy {NONE,REDUCE_MAX,REDUCE_MEAN,REDUCE_
↪MEAN_MAX,FIRST_TOKEN,LAST_TOKEN,CLS_POOLED,CLASSIFICATION,REGRESSION}]
                              [-mask_cls_sep] [-no_special_token]
                              [-show_tokens_to_client]
                              [-no_position_embeddings]
                              [-num_labels NUM_LABELS] [-port PORT]
                              [-port_out PORT_OUT] [-http_port HTTP_PORT]
                              [-http_max_connect HTTP_MAX_CONNECT]
                              [-cors CORS] [-num_worker NUM_WORKER]
                              [-max_batch_size MAX_BATCH_SIZE]
                              [-priority_batch_size PRIORITY_BATCH_SIZE]
                              [-cpu] [-xla] [-fp16]
                              [-gpu_memory_fraction GPU_MEMORY_FRACTION]
                              [-device_map DEVICE_MAP [DEVICE_MAP ...]]
                              [-prefetch_size PREFETCH_SIZE]
                              [-fixed_embed_length] [-verbose] [-version]
                              [-test_client_batch_size [TEST_CLIENT_BATCH_SIZE [TEST_
↪CLIENT_BATCH_SIZE ...]]]
                              [-test_max_batch_size [TEST_MAX_BATCH_SIZE [TEST_MAX_
↪BATCH_SIZE ...]]]
                              [-test_max_seq_len [TEST_MAX_SEQ_LEN [TEST_MAX_SEQ_LEN .
↪..]]]
                              [-test_num_client [TEST_NUM_CLIENT [TEST_NUM_CLIENT ...
↪]]]
                              [-test_pooling_layer [TEST_POOLING_LAYER [TEST_POOLING_
↪LAYER ...]]]
                              [-wait_till_ready WAIT_TILL_READY]
                              [-client_vocab_file CLIENT_VOCAB_FILE]
                              [-num_repeat NUM_REPEAT]
```

Named Arguments

-verbose turn on tensorflow logging for debug

Default: False

-version show program's version number and exit

File Paths

config the path, checkpoint and filename of a pretrained/fine-tuned BERT model

-model_dir directory of a pretrained BERT model

-tuned_model_dir directory of a fine-tuned BERT model

-ckpt_name filename of the checkpoint file. By default it is "bert_model.ckpt", but for a fine-tuned model the name could be different.

Default: "bert_model.ckpt"

-config_name filename of the JSON config file for BERT model.

Default: "bert_config.json"

-graph_tmp_dir path to graph temp file

BERT Parameters

config how BERT model and pooling works

-max_seq_len maximum length of a sequence, longer sequence will be trimmed on the right side. set it to NONE for dynamically using the longest sequence in a (mini)batch.

Default: 25

-cased_tokenization Whether tokenizer should skip the default lowercasing and accent removal.Should be used for e.g. the multilingual cased pretrained BERT model.

Default: True

-pooling_layer the encoder layer(s) that receives pooling. Give a list in order to concatenate several layers into one

Default: [-2]

-pooling_strategy Possible choices: NONE, REDUCE_MAX, REDUCE_MEAN, REDUCE_MEAN_MAX, FIRST_TOKEN, LAST_TOKEN, CLS_POOLED, CLASSIFICATION, REGRESSION

the pooling strategy for generating encoding vectors

Default: REDUCE_MEAN

-mask_cls_sep masking the embedding on [CLS] and [SEP] with zero. When pooling_strategy is in {CLS_TOKEN, FIRST_TOKEN, SEP_TOKEN, LAST_TOKEN} then the embedding is preserved, otherwise the embedding is masked to zero before pooling

Default: False

-no_special_token add [CLS] and [SEP] in every sequence, put sequence to the model without [CLS] and [SEP] when True and is_tokenized=True in Client

Default: False

-show_tokens_to_client sending tokenization results to client

Default: False

-no_position_embeddings Whether to add position embeddings for the position of each token in the sequence.

Default: False

-num_labels Numbers of Label

Default: 2

Serving Configs

config how server utilizes GPU/CPU resources

-port, -port_in, -port_data server port for receiving data from client

Default: 5555

-port_out, -port_result server port for sending result to client

Default: 5556

-http_port server port for receiving HTTP requests

-http_max_connect maximum number of concurrent HTTP connections

Default: 10

-cors setting "Access-Control-Allow-Origin" for HTTP requests

Default: "*"

-num_worker number of server instances

Default: 1

-max_batch_size maximum number of sequences handled by each worker

Default: 256

-priority_batch_size batch smaller than this size will be labeled as high priority,and jumps forward in the job queue

Default: 16

-cpu running on CPU (default on GPU)

Default: False

-xla enable XLA compiler (experimental)

Default: False

-fp16 use float16 precision (experimental)

Default: False

-gpu_memory_fraction determine the fraction of the overall amount of memory that each visible GPU should be allocated per worker. Should be in range [0.0, 1.0]

Default: 0.5

-device_map specify the list of GPU device ids that will be used (id starts from 0). If num_worker > len(device_map), then device will be reused; if num_worker < len(device_map), then device_map[:num_worker] will be used

Default: []

-prefetch_size the number of batches to prefetch on each worker. When running on a CPU-only machine, this is set to 0 for comparability

Default: 10

-fixed_embed_length when "max_seq_len" is set to None, the server determines the "max_seq_len" according to the actual sequence lengths within each batch. When "pooling_strategy=NONE", this may cause two ".encode()" from the same client results in different sizes [B, T, D].Turn this on to fix the "T" in [B, T, D] to "max_position_embeddings" in bert json config.

Default: False

Benchmark parameters

config the experiments of the benchmark

-test_client_batch_size Default: [1, 16, 256, 4096]

-test_max_batch_size Default: [8, 32, 128, 512]

-test_max_seq_len Default: [32, 64, 128, 256]

-test_num_client Default: [1, 4, 16, 64]

-test_pooling_layer Default: [[-1], [-2], [-3], [-4], [-5], [-6], [-7], [-8], [-9], [-10], [-11], [-12]]

-wait_till_ready seconds to wait until server is ready to serve

Default: 30

-client_vocab_file file path for building client vocabulary

Default: "README.md"

-num_repeat number of repeats per experiment (must >2), as the first two results are omitted for warm-up effect

Default: 10

2.6 Frequently Asked Questions

2.6.1 Where is the BERT code come from?

BERT code of this repo is forked from the original BERT repo with necessary modification, especially in extract_features.py.

2.6.2 How large is a sentence vector?

In general, each sentence is translated to a 768-dimensional vector. Depending on the pretrained BERT you are using, `pooling_strategy` and `pooling_layer` the dimensions of the output vector could be different.

2.6.3 How do you get the fixed representation? Did you do pooling or something?

Yes, pooling is required to get a fixed representation of a sentence. In the default strategy `REDUCE_MEAN`, I take the second-to-last hidden layer of all of the tokens in the sentence and do average pooling.

2.6.4 Are you suggesting using BERT without fine-tuning?

Yes and no. On the one hand, Google pretrained BERT on Wikipedia data, thus should encode enough prior knowledge of the language into the model. Having such feature is not a bad idea. On the other hand, these prior knowledge is not specific to any particular domain. It should be totally reasonable if the performance is not ideal if you are using it on, for example, classifying legal cases. Nonetheless, you can always first fine-tune your own BERT on the downstream task and then use `bert-as-service` to extract the feature vectors efficiently. Keep in mind that `bert-as-service` is just a feature extraction service based on BERT. Nothing stops you from using a fine-tuned BERT.

2.6.5 Can I get a concatenation of several layers instead of a single layer ?

Sure! Just use a list of the layer you want to concatenate when calling the server. Example:

```
bert-serving-start -pooling_layer -4 -3 -2 -1 -model_dir /tmp/english_L-12_H-768_A-12/
```

2.6.6 What are the available pooling strategies?

Here is a table summarizes all pooling strategies I implemented. Choose your favorite one by specifying `bert-serving-start -pooling_strategy`.

Strategy	Description
`NONE`	no pooling at all, useful when you want to use word embedding instead of sentence embedding. This will results in a `[max_seq_len, 768]` encode matrix for a sequence.
`REDUCE_MEAN`	take the average of the hidden state of encoding layer on the time axis
`REDUCE_MAX`	take the maximum of the hidden state of encoding layer on the time axis
`REDUCE_MEAN_MAX`	do `REDUCE_MEAN` and `REDUCE_MAX` separately and then concat them together on the last axis, resulting in 1536-dim sentence encodes
`CLS_TOKEN` or `FIRST_TOKEN`	get the hidden state corresponding to `[CLS]`, i.e. the first token
`SEP_TOKEN` or `LAST_TOKEN`	get the hidden state corresponding to `[SEP]`, i.e. the last token

2.6.7 Why not use the hidden state of the first token as default strategy, i.e. the `[CLS]`?

Because a pre-trained model is not fine-tuned on any downstream tasks yet. In this case, the hidden state of `[CLS]` is not a good sentence representation. If later you fine-tune the model, you may use `[CLS]` as well.

2.6.8 BERT has 12/24 layers, so which layer are you talking about?

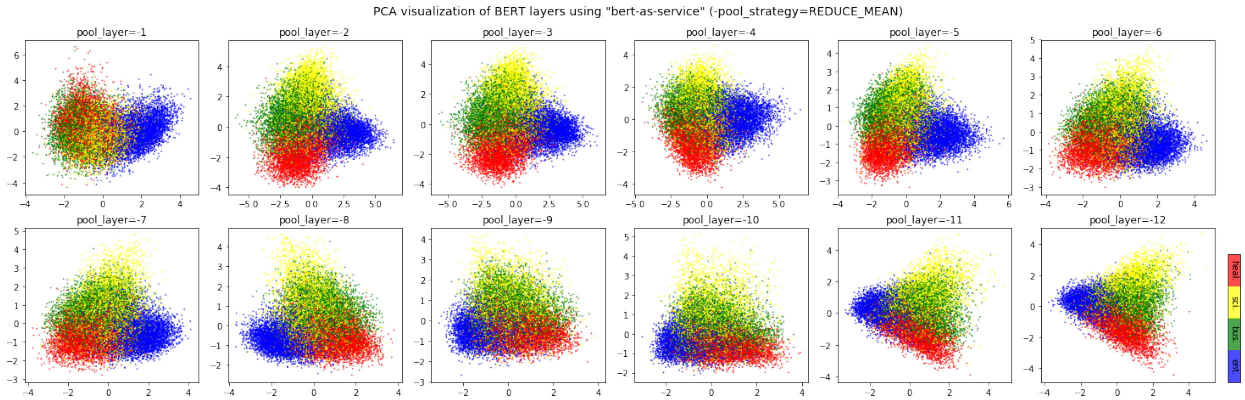
By default this service works on the second last layer, i.e. `pooling_layer=-2`. You can change it by setting `pooling_layer` to other negative values, e.g. -1 corresponds to the last layer.

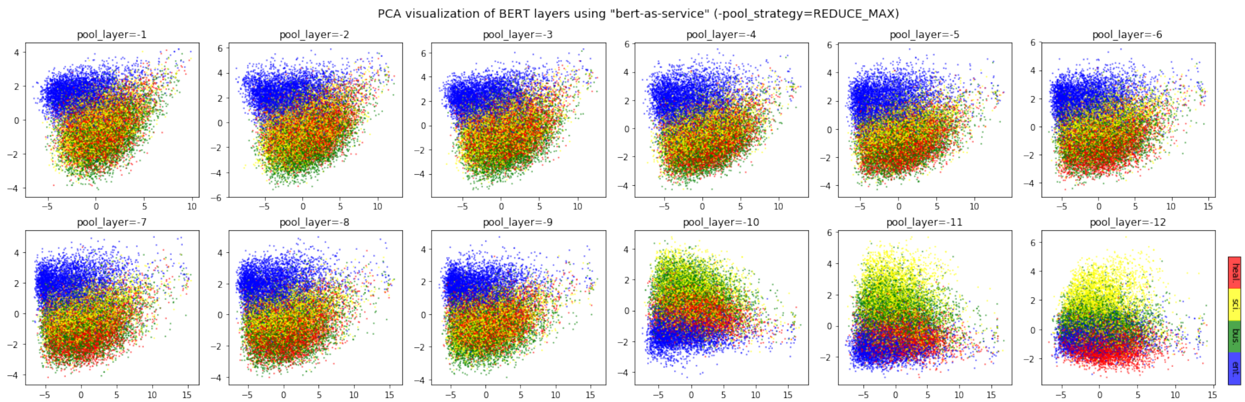
2.6.9 Why not the last hidden layer? Why second-to-last?

The last layer is too closed to the target functions (i.e. masked language model and next sentence prediction) during pre-training, therefore may be biased to those targets. If you question about this argument and want to use the last hidden layer anyway, please feel free to set `pooling_layer=-1`.

2.6.10 So which layer and which pooling strategy is the best?

It depends. Keep in mind that different BERT layers capture different information. To see that more clearly, here is a visualization on UCI-News Aggregator Dataset, where I randomly sample 20K news titles; get sentence encodes from different layers and with different pooling strategies, finally reduce it to 2D via PCA (one can of course do t-SNE as well, but that's not my point). There are only four classes of the data, illustrated in red, blue, yellow and green. To reproduce the result, please run example7.py.





Intuitively, `pooling_layer=-1` is close to the training output, so it may be biased to the training targets. If you don't fine tune the model, then this could lead to a bad representation. `pooling_layer=-12` is close to the word embedding, may preserve the very original word information (with no fancy self-attention etc.). On the other hand, you may achieve the very same performance by simply using a word-embedding only. That said, anything in-between [-1, -12] is then a trade-off.

2.6.11 Could I use other pooling techniques?

For sure. But if you introduce new `tf.variables` to the graph, then you need to train those variables before using the model. You may also want to check some pooling techniques I mentioned in my blog post.

2.6.12 Can I start multiple clients and send requests to one server simultaneously?

Yes! That's the purpose of this repo. In fact you can start as many clients as you want. One server can handle all of them (given enough time).

2.6.13 How many requests can one service handle concurrently?

The maximum number of concurrent requests is determined by `num_worker` in `bert-serving-start`. If you a sending more than `num_worker` requests concurrently, the new requests will be temporally stored in a queue until a free worker becomes available.

2.6.14 So one request means one sentence?

No. One request means a list of sentences sent from a client. Think the size of a request as the batch size. A request may contain 256, 512 or 1024 sentences. The optimal size of a request is often determined empirically. One large request can certainly improve the GPU utilization, yet it also increases the overhead of transmission. You may run `python example/example1.py` for a simple benchmark.

2.6.15 How about the speed? Is it fast enough for production?

It highly depends on the `max_seq_len` and the size of a request. On a single Tesla M40 24GB with `max_seq_len=40`, you should get about 470 samples per second using a 12-layer BERT. In general, I'd suggest smaller `max_seq_len` (25) and larger request size (512/1024).

2.6.16 Did you benchmark the efficiency?

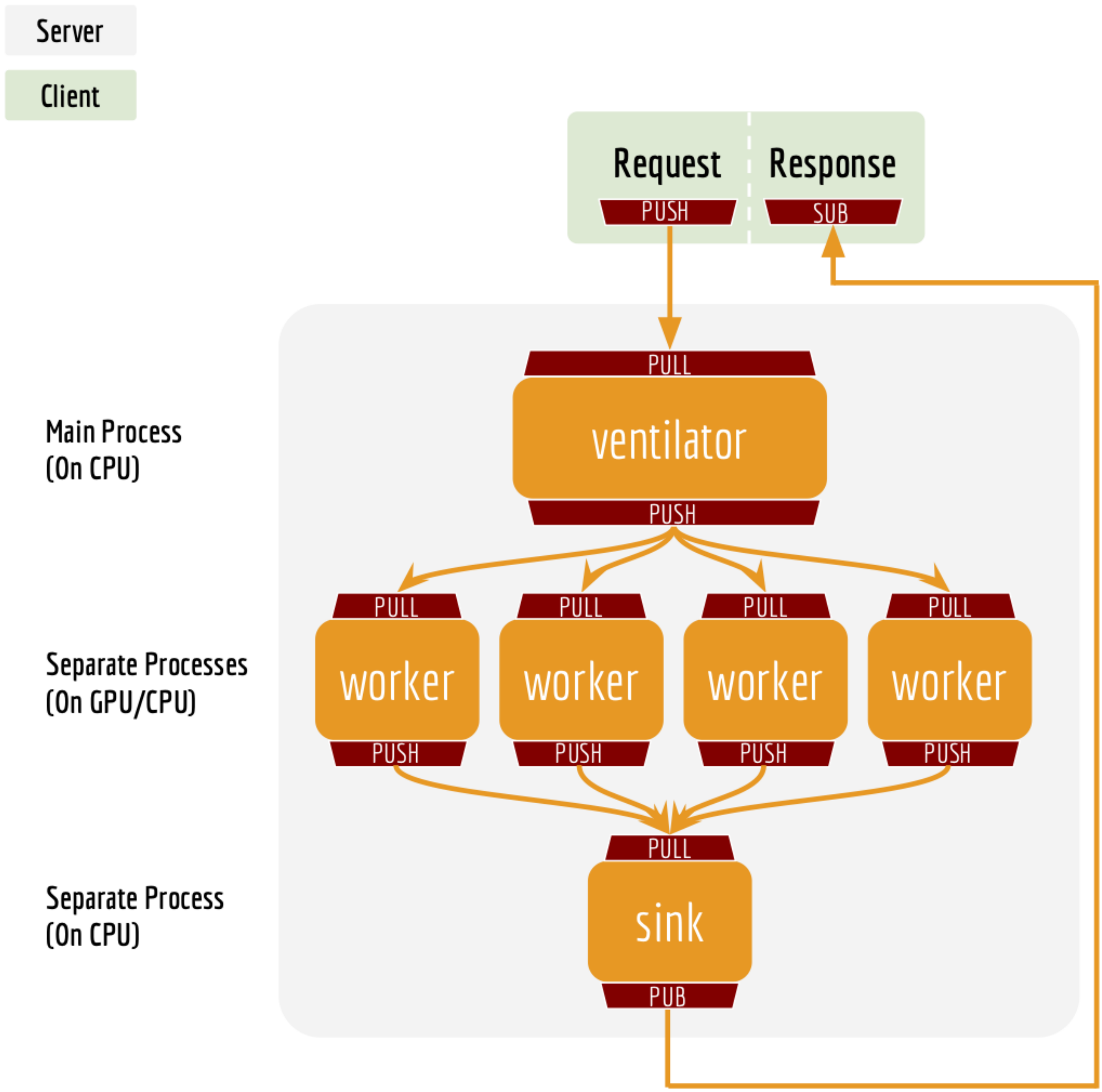
Yes. See *Benchmark*.

To reproduce the results, please run ‘‘bert-serving-benchmark –help‘‘.

2.6.17 What is backend based on?

ZeroMQ.

2.6.18 What is the parallel processing model behind the scene?



2.6.19 Why does the server need two ports?

One port is for pushing text data into the server, the other port is for publishing the encoded result to the client(s). In this way, we get rid of back-chatter, meaning that at every level recipients never talk back to senders. The overall message flow is strictly one-way, as depicted in the above figure. Killing back-chatter is essential to real scalability, allowing us to use `BertClient` in an asynchronous way.

2.6.20 Do I need Tensorflow on the client side?

No. Think of `BertClient` as a general feature extractor, whose output can be fed to *any* ML models, e.g. `scikit-learn`, `pytorch`, `tensorflow`.

2.6.21 Can I use multilingual BERT model provided by Google?

Yes.

2.6.22 Can I use my own fine-tuned BERT model?

Yes. In fact, this is suggested. Make sure you have the following three items in `model_dir`:

- A TensorFlow checkpoint (`bert_model.ckpt`) containing the pre-trained weights (which is actually 3 files).
- A vocab file (`vocab.txt`) to map WordPiece to word id.
- A config file (`bert_config.json`) which specifies the hyperparameters of the model.

2.6.23 Can I run it in python 2?

Server side no, client side yes. This is based on the consideration that python 2.x might still be a major piece in some tech stack. Migrating the whole downstream stack to python 3 for supporting `bert-as-service` can take quite some effort. On the other hand, setting up `BertServer` is just a one-time thing, which can be even run in a docker container. To ease the integration, we support python 2 on the client side so that you can directly use `BertClient` as a part of your python 2 project, whereas the server side should always be hosted with python 3.

2.6.24 Do I need to do segmentation for Chinese?

No, if you are using the pretrained Chinese BERT released by Google you don't need word segmentation. As this Chinese BERT is character-based model. It won't recognize word/phrase even if you intentionally add space in-between. To see that more clearly, this is what the BERT model actually receives after tokenization:

```
bc.encode(['hey you', 'whats up?', '', '  '])
```

```
tokens: [CLS] hey you [SEP]
input_ids: 101 13153 8357 102 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
input_mask: 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

tokens: [CLS] what ##s up ? [SEP]
input_ids: 101 9100 8118 8644 136 102 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
input_mask: 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

tokens: [CLS]     [SEP]
input_ids: 101 872 1962 720 8043 102 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
input_mask: 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

tokens: [CLS]     [SEP]
input_ids: 101 2769 6820 1377 809 102 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
input_mask: 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

That means the word embedding is actually the character embedding for Chinese-BERT.

2.6.25 Why my (English) word is tokenized to `##something`?

Because your word is out-of-vocabulary (OOV). The tokenizer from Google uses a greedy longest-match-first algorithm to perform tokenization using the given vocabulary.

For example:

```
input = "unaffable"
tokenizer_output = ["un", "##aff", "##able"]
```

2.6.26 Can I use my own tokenizer?

Yes. If you already tokenize the sentence on your own, simply send use `encode` with `List[List[Str]]` as input and turn on `is_tokenized`, i.e. `bc.encode(texts, is_tokenized=True)`.

2.6.27 I encounter `zmq.error.ZMQError: Operation cannot be accomplished in current state` when using `BertClient`, what should I do?

This is often due to the misuse of `BertClient` in multi-thread/process environment. Note that you can't reuse one `BertClient` among multiple threads/processes, you have to make a separate instance for each thread/process. For example, the following won't work at all:

This is often due to the misuse of `BertClient` in multi-thread/process environment. Note that you can't reuse one `BertClient` among multiple threads/processes, you have to make a separate instance for each thread/process. For example, the following won't work at all:

```
# BAD example
bc = BertClient()

# in Proc1/Thread1 scope:
bc.encode(lst_str)

# in Proc2/Thread2 scope:
bc.encode(lst_str)
```

Instead, please do:

```
# in Proc1/Thread1 scope:
bc1 = BertClient()
bc1.encode(lst_str)

# in Proc2/Thread2 scope:
bc2 = BertClient()
bc2.encode(lst_str)
```

2.6.28 After running the server, I have several garbage `tmpXXXX` folders. How can I change this behavior ?

These folders are used by ZeroMQ to store sockets. You can choose a different location by setting the environment variable `ZEROMQ_SOCK_TMP_DIR` : `export ZEROMQ_SOCK_TMP_DIR=/tmp/`

2.6.29 The cosine similarity of two sentence vectors is unreasonably high (e.g. always > 0.8), what's wrong?

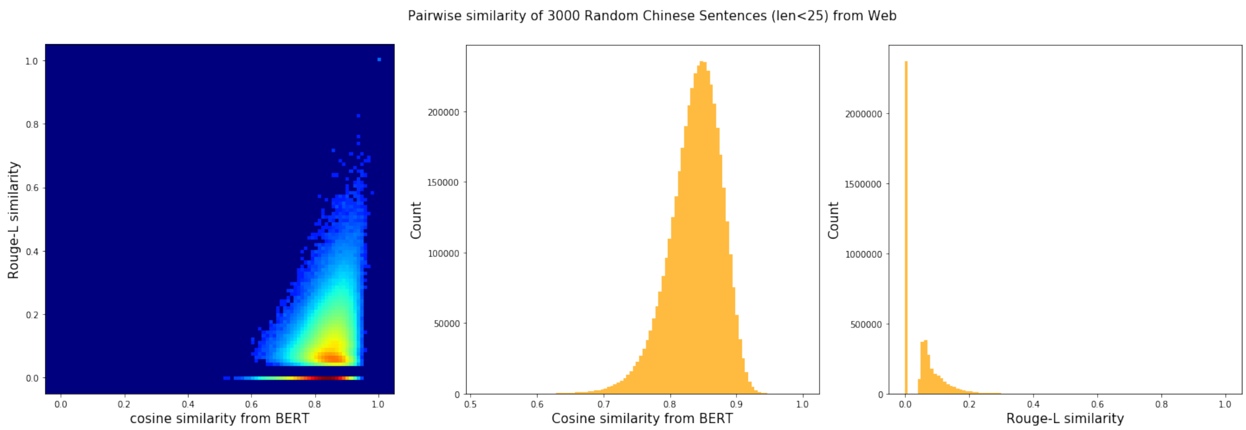
A decent representation for a downstream task doesn't mean that it will be meaningful in terms of cosine distance. Since cosine distance is a linear space where all dimensions are weighted equally. if you want to use cosine distance anyway, then please focus on the rank not the absolute value. Namely, do not use:

```
if cosine(A, B) > 0.9, then A and B are similar
```

Please consider the following instead:

```
if cosine(A, B) > cosine(A, C), then A is more similar to B than C.
```

The graph below illustrates the pairwise similarity of 3000 Chinese sentences randomly sampled from web (char. length < 25). We compute cosine similarity based on the sentence vectors and Rouge-L based on the raw text. The diagonal (self-correlation) is removed for the sake of clarity. As one can see, there is some positive correlation between these two metrics.



2.6.30 I'm getting bad performance, what should I do?

This often suggests that the pretrained BERT could not generate a descent representation of your downstream task. Thus, you can fine-tune the model on the downstream task and then use `bert-as-service` to serve the fine-tuned BERT. Note that, `bert-as-service` is just a feature extraction service based on BERT. Nothing stops you from using a fine-tuned BERT.

2.6.31 Can I run the server side on CPU-only machine?

Yes, please run `bert-serving-start -cpu -max_batch_size 16`. Note that, CPU does not scale as good as GPU on large batches, therefore the `max_batch_size` on the server side needs to be smaller, e.g. 16 or 32.

2.6.32 How can I choose `num_worker`?

Generally, the number of workers should be less than or equal to the number of GPU/CPU you have. Otherwise, multiple workers will be allocated to one GPU/CPU, which may not scale well (and may cause out-of-memory on GPU).

2.6.33 Can I specify which GPU to use?

Yes, you can specifying -device_map as follows:

```
bert-serving-start -device_map 0 1 4 -num_worker 4 -model_dir ...
```

This will start four workers and allocate them to GPU0, GPU1, GPU4 and again GPU0, respectively. In general, if num_worker > device_map, then devices will be reused and shared by the workers (may scale suboptimally or cause OOM); if num_worker < device_map, only device_map[:num_worker] will be used.

Note, device_map is ignored when running on CPU.

2.7 Benchmark

The primary goal of benchmarking is to test the scalability and the speed of this service, which is crucial for using it in a dev/prod environment. Benchmark was done on Tesla M40 24GB, experiments were repeated 10 times and the average value is reported.

To reproduce the results, please run

```
bert-serving-benchmark --help
```

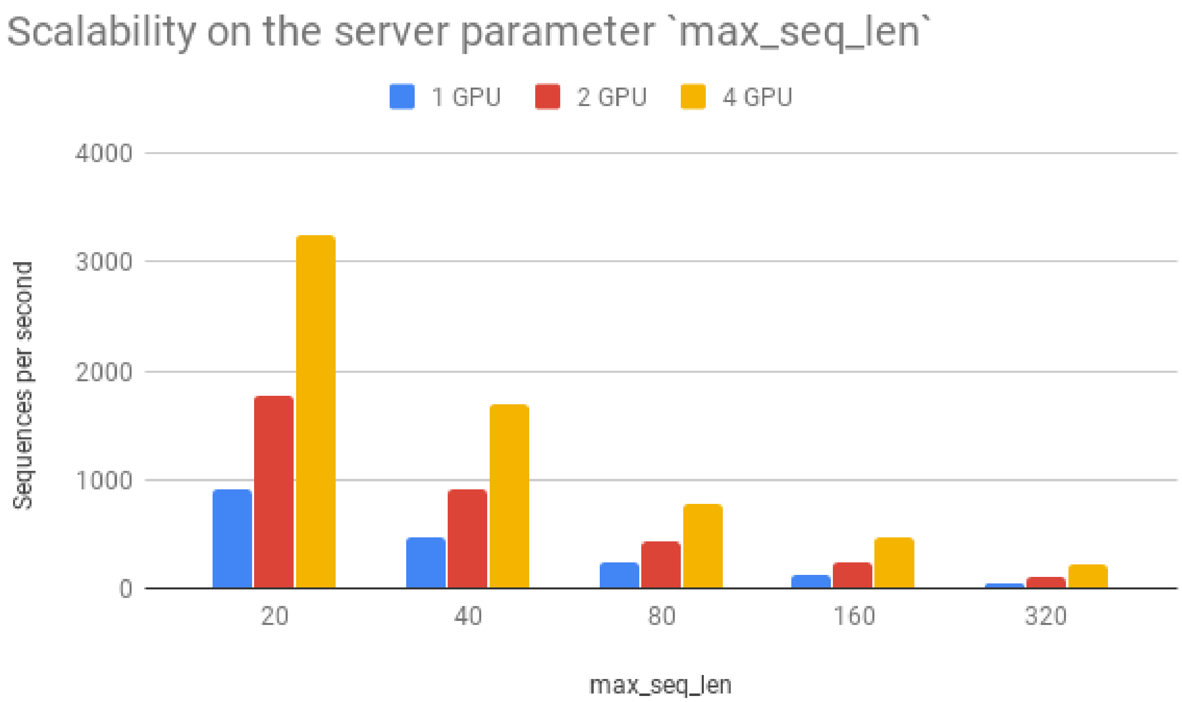
Common arguments across all experiments are:

Parameter	Value
num_worker	1,2,4
max_seq_len	40
client_batch_size	2048
max_batch_size	256
num_client	1

2.7.1 Speed wrt. max_seq_len

max_seq_len is a parameter on the server side, which controls the maximum length of a sequence that a BERT model can handle. Sequences larger than max_seq_len will be truncated on the left side. Thus, if your client want to send long sequences to the model, please make sure the server can handle them correctly.

Performance-wise, longer sequences means slower speed and more chance of OOM, as the multi-head self-attention (the core unit of BERT) needs to do dot products and matrix multiplications between every two symbols in the sequence.



max_seq_len	1 GPU	2 GPU	4 GPU
20	903	1774	3254
40	473	919	1687
80	231	435	768
160	119	237	464
320	54	108	212

2.7.2 Speed wrt. `client_batch_size`

`client_batch_size` is the number of sequences from a client when invoking `encode()`. For performance reason, please consider encoding sequences in batch rather than encoding them one by one.

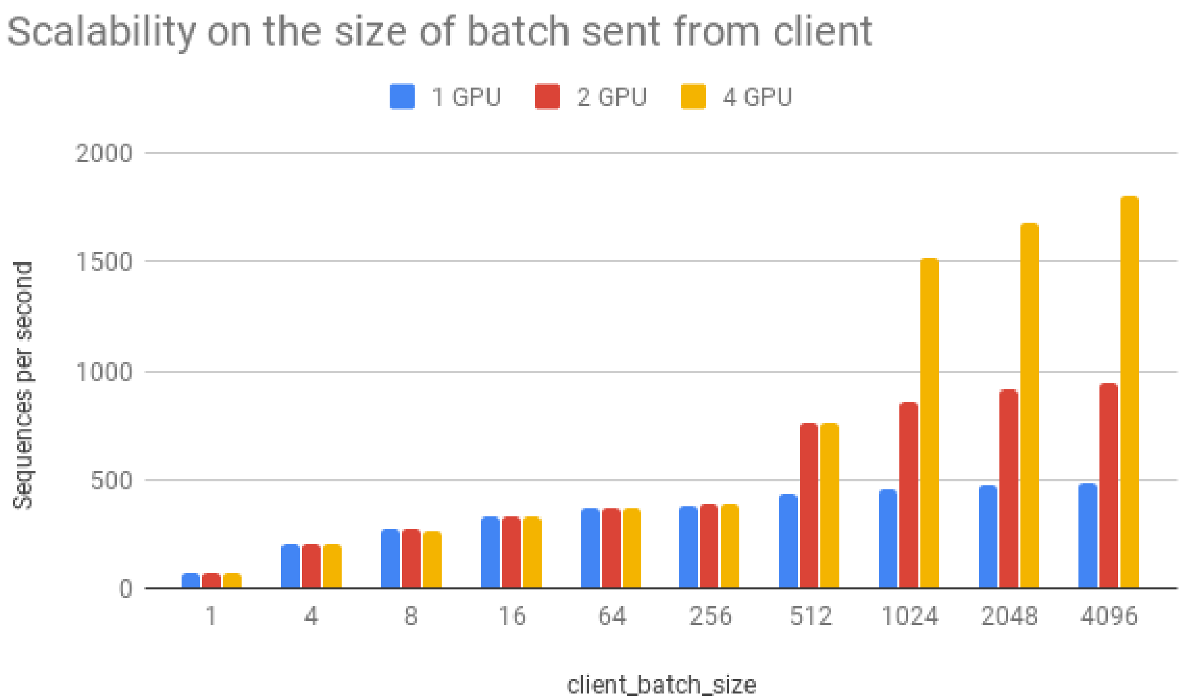
For example, do:

```
# prepare your sent in advance
bc = BertClient()
my_sentences = [s for s in my_corpus.iter()]
# doing encoding in one-shot
vec = bc.encode(my_sentences)
```

DON'T:

```
bc = BertClient()
vec = []
for s in my_corpus.iter():
    vec.append(bc.encode(s))
```

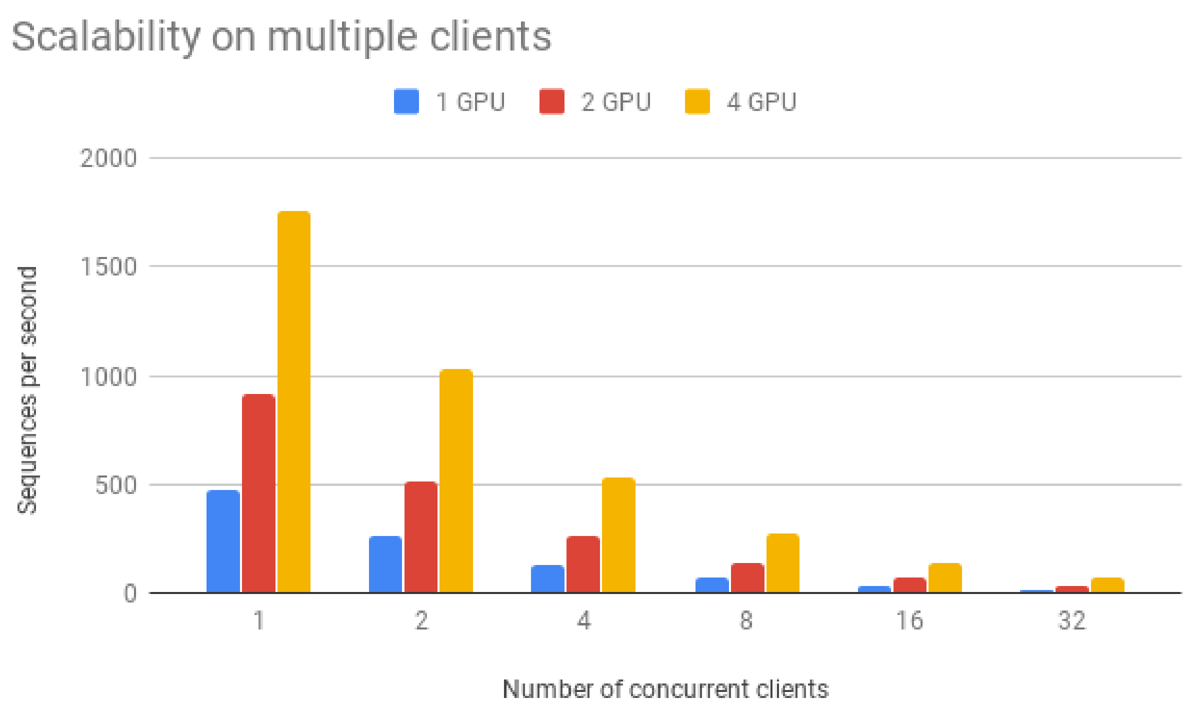
It's even worse if you put `BertClient()` inside the loop. Don't do that.



client_batch_size	1 GPU	2 GPU	4 GPU
1	75	74	72
4	206	205	201
8	274	270	267
16	332	329	330
64	365	365	365
256	382	383	383
512	432	766	762
1024	459	862	1517
2048	473	917	1681
4096	481	943	1809

2.7.3 Speed wrt. `num_client`

`num_client` represents the number of concurrent clients connected to the server at the same time.

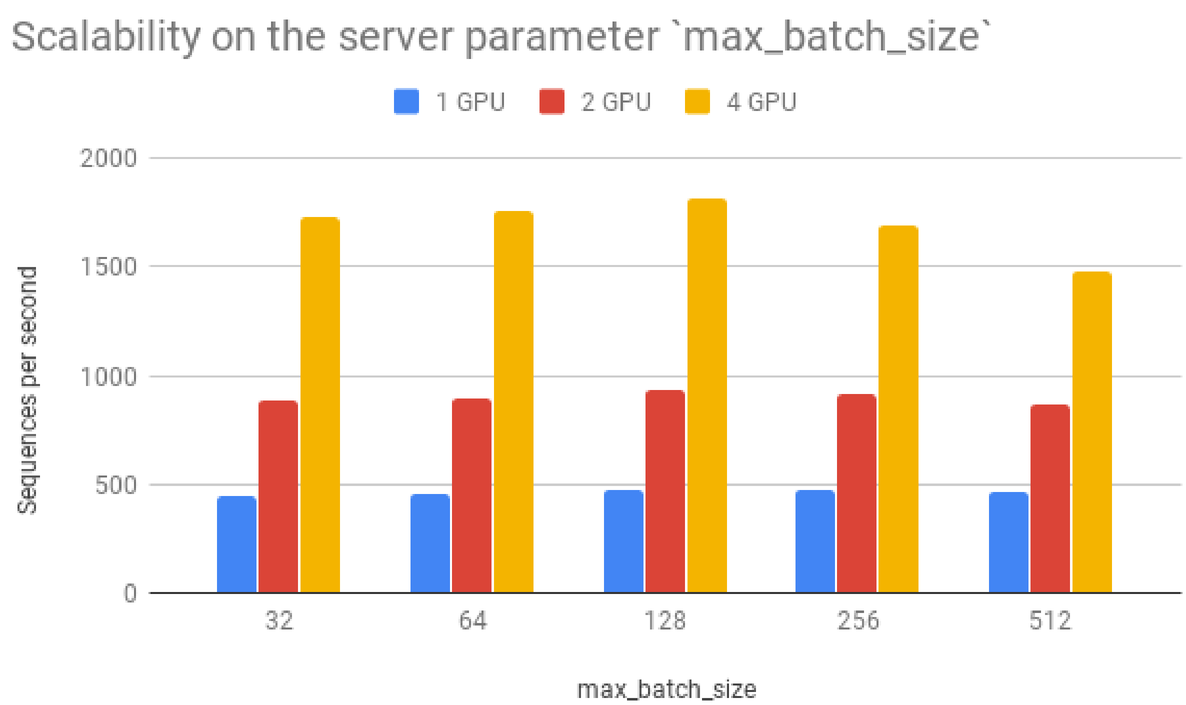


num_client	1 GPU	2 GPU	4 GPU
1	473	919	1759
2	261	512	1028
4	133	267	533
8	67	136	270
16	34	68	136
32	17	34	68

As one can observe, 1 clients 1 GPU = 381 seqs/s, 2 clients 2 GPU 402 seqs/s, 4 clients 4 GPU 413 seqs/s. This shows the efficiency of our parallel pipeline and job scheduling, as the service can leverage the GPU time more exhaustively as concurrent requests increase.

2.7.4 Speed wrt. `max_batch_size`

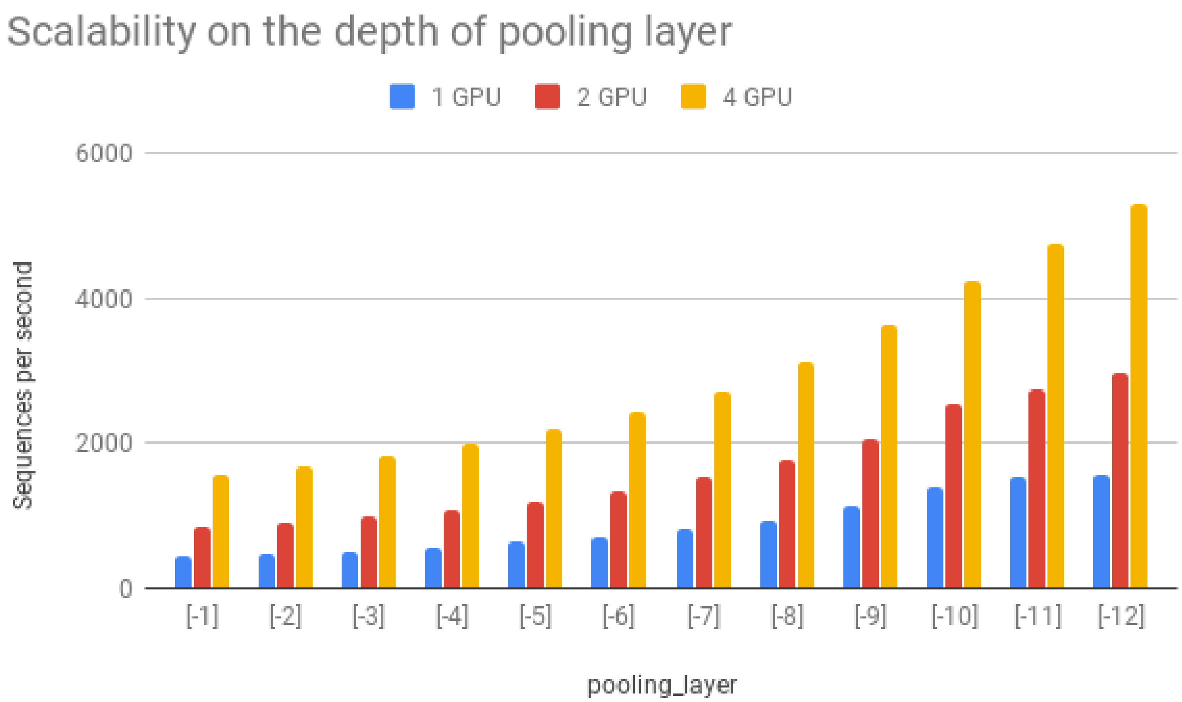
`max_batch_size` is a parameter on the server side, which controls the maximum number of samples per batch per worker. If a incoming batch from client is larger than `max_batch_size`, the server will split it into small batches so that each of them is less or equal than `max_batch_size` before sending it to workers.



max_batch_size	1 GPU	2 GPU	4 GPU
32	450	887	1726
64	459	897	1759
128	473	931	1816
256	473	919	1688
512	464	866	1483

2.7.5 Speed wrt. `pooling_layer`

`pooling_layer` determines the encoding layer that pooling operates on. For example, in a 12-layer BERT model, `-1` represents the layer closed to the output, `-12` represents the layer closed to the embedding layer. As one can observe below, the depth of the pooling layer affects the speed.



pooling_layer	1 GPU	2 GPU	4 GPU
[-1]	438	844	1568
[-2]	475	916	1686
[-3]	516	995	1823
[-4]	569	1076	1986
[-5]	633	1193	2184
[-6]	711	1340	2430
[-7]	820	1528	2729
[-8]	945	1772	3104
[-9]	1128	2047	3622
[-10]	1392	2542	4241
[-11]	1523	2737	4752
[-12]	1568	2985	5303

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

c

Index

B

C

E

F

S