View on TensorFlow.org Run in Google Colab View on GitHub Download notebook See TF Hub model

Solve GLUE tasks using BERT on TPU

BERT can be used to solve many problems in natural language processing. You will learn how to fine-tune BERT for many tasks from the GLUE benchmark:

1. CoLA (Corpus of Linguistic Acceptability): Is the sentence grammatically correct?
2. SST-2 (Stanford Sentiment Treebank): The task is to predict the sentiment of a given sentence.
3. MRPC (Microsoft Research Paraphrase Corpus): Determine whether a pair of sentences are semantically equivalent.
4. QQP (Quora Question Pairs2): Determine whether a pair of questions are semantically equivalent.
5. MNLI (Multi-Genre Natural Language Inference): Given a premise sentence and a hypothesis sentence, the task is to predict whether the premise entails the hypothesis (entailment), contradicts the hypothesis (contradiction), or neither (neutral).
6. QNLI(Question-answering Natural Language Inference): The task is to determine whether the context sentence contains the answer to the question.
7. RTE(Recognizing Textual Entailment): Determine if a sentence entails a given hypothesis or not.
8. WNLI(Winograd Natural Language Inference): The task is to predict if the sentence with the pronoun substituted is entailed by the original sentence.

This tutorial contains complete end-to-end code to train these models on a TPU. You can also run this notebook on a GPU, by changing one line (described below).

In this notebook, you will:

- Load a BERT model from TensorFlow Hub
- Choose one of GLUE tasks and download the dataset
- Preprocess the text
- Fine-tune BERT (examples are given for single-sentence and multi-sentence datasets)
- Save the trained model and use it

Key point: The model you develop will be end-to-end. The preprocessing logic will be included in the model itself, making it capable of accepting raw strings as input.

Note: This notebook should be run using a TPU. In Colab, choose **Runtime -> Change runtime type** and verify that a **TPU** is selected.

Setup

You will use a separate model to preprocess text before using it to fine-tune BERT. This model depends on tensorflow/text, which you will install below.

```
!pip install -q -U tensorflow-text
```

You will use the AdamW optimizer from tensorflow/models to fine-tune BERT, which you will install as well.

```
!pip install -q -U tf-models-official
```

```
!pip install -U tfds-nightly
```

```
import os
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds
import tensorflow_text as text  # A dependency of the preprocessing model
import tensorflow_addons as tfa
from official.nlp import optimization
import numpy as np

tf.get_logger().setLevel('ERROR')
```

Next, configure TFHub to read checkpoints directly from TFHub's Cloud Storage buckets. This is only recommended when running TFHub models on TPU.

Without this setting TFHub would download the compressed file and extract the checkpoint locally. Attempting to load from these local files will fail with the following error:

```
InvalidArgumentError: Unimplemented: File system scheme '[local]' not implemented
```

This is because the TPU can only read directly from Cloud Storage buckets.

Note: This setting is automatic in Colab.

```
os.environ["TFHUB_MODEL_LOAD_FORMAT"]="UNCOMPRESSED"
```

Connect to the TPU worker

The following code connects to the TPU worker and changes TensorFlow's default device to the CPU device on the TPU worker. It also defines a TPU distribution strategy that you will use to distribute model training onto the 8 separate TPU cores available on this one TPU worker. See TensorFlow's TPU guide for more information.

```
import os

if os.environ['COLAB_TPU_ADDR']:
  cluster_resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='')
  tf.config.experimental_connect_to_cluster(cluster_resolver)
  tf.tpu.experimental.initialize_tpu_system(cluster_resolver)
  strategy = tf.distribute.TPUStrategy(cluster_resolver)
  print('Using TPU')
elif tf.test.is_gpu_available():
  strategy = tf.distribute.MirroredStrategy()
  print('Using GPU')
else:
  raise ValueError('Running on CPU is not recommended.')
```

Loading models from TensorFlow Hub

Here you can choose which BERT model you will load from TensorFlow Hub and fine-tune. There are multiple BERT models available to choose from.

- BERT-Base, Uncased and seven more models with trained weights released by the original BERT authors.
- Small BERTs have the same general architecture but fewer and/or smaller Transformer blocks, which lets you explore tradeoffs between speed, size and quality.
- ALBERT: four different sizes of "A Lite BERT" that reduces model size (but not computation time) by sharing parameters between layers.
- BERT Experts: eight models that all have the BERT-base architecture but offer a choice between different pre-training domains, to align more closely with the target task.
- Electra has the same architecture as BERT (in three different sizes), but gets pre-trained as a discriminator in a set-up that resembles a Generative Adversarial Network (GAN).
- BERT with Talking-Heads Attention and Gated GELU [base, large] has two improvements to the core of the Transformer architecture.

See the model documentation linked above for more details.

In this tutorial, you will start with BERT-base. You can use larger and more recent models for higher accuracy, or smaller models for faster training times. To change the model, you only need to switch a single line of code (shown below). All the differences are encapsulated in the SavedModel you will download from TensorFlow Hub.

Choose a BERT model to fine-tune

bert_model_name: bert_en_uncased_L-12_H-768_A-12

Preprocess the text

On the Classify text with BERT colab the preprocessing model is used directly embedded with the BERT encoder.

This tutorial demonstrates how to do preprocessing as part of your input pipeline for training, using Dataset.map, and then merge it into the model that gets exported for inference. That way, both training and inference can work from raw text inputs, although the TPU itself requires numeric inputs.

TPU requirements aside, it can help performance have preprocessing done asynchronously in an input pipeline (you can learn more in the tf.data performance guide).

This tutorial also demonstrates how to build multi-input models, and how to adjust the sequence length of the inputs to BERT.

Let's demonstrate the preprocessing model.

```
bert_preprocess = hub.load(tfhub_handle_preprocess)
tok = bert_preprocess.tokenize(tf.constant(['Hello TensorFlow!']))
print(tok)
```

Each preprocessing model also provides a method, `.bert_pack_inputs(tensors, seq_length)`, which takes a list of tokens (like `tok` above) and a sequence length argument. This packs the inputs to create a dictionary of tensors in the format expected by the BERT model.

```
text_preprocessed = bert_preprocess.bert_pack_inputs([tok, tok], tf.constant(20

print('Shape Word Ids : ', text_preprocessed['input_word_ids'].shape)
print('Word Ids       : ', text_preprocessed['input_word_ids'][0, :16])
print('Shape Mask     : ', text_preprocessed['input_mask'].shape)
print('Input Mask     : ', text_preprocessed['input_mask'][0, :16])
print('Shape Type Ids : ', text_preprocessed['input_type_ids'].shape)
print('Type Ids       : ', text_preprocessed['input_type_ids'][0, :16])
```

Here are some details to pay attention to:

- `input_mask` The mask allows the model to cleanly differentiate between the content and the padding. The mask has the same shape as the `input_word_ids`, and contains a 1 anywhere the `input_word_ids` is not padding.

- `input_type_ids` has the same shape as `input_mask`, but inside the non-padded region, contains a 0 or a 1 indicating which sentence the token is a part of.

Next, you will create a preprocessing model that encapsulates all this logic. Your model will take strings as input, and return appropriately formatted objects which can be passed to BERT.

Each BERT model has a specific preprocessing model, make sure to use the proper one described on the BERT's model documentation.

Note: BERT adds a "position embedding" to the token embedding of each input, and these come from a fixed-size lookup table. That imposes a max seq length of 512 (which is also a practical limit, due to the quadratic growth of attention computation). For this Colab 128 is good enough.

```
def make_bert_preprocess_model(sentence_features, seq_length=128):
  """Returns Model mapping string features to BERT inputs.

  Args:
    sentence_features: a list with the names of string-valued features.
    seq_length: an integer that defines the sequence length of BERT inputs.

  Returns:
    A Keras Model that can be called on a list or dict of string Tensors
    (with the order or names, resp., given by sentence_features) and
    returns a dict of tensors for input to BERT.
  """

  input_segments = [
      tf.keras.layers.Input(shape=(), dtype=tf.string, name=ft)
      for ft in sentence_features]

  # Tokenize the text to word pieces.
  bert_preprocess = hub.load(tfhub_handle_preprocess)
  tokenizer = hub.KerasLayer(bert_preprocess.tokenize, name='tokenizer')
  segments = [tokenizer(s) for s in input_segments]

  # Optional: Trim segments in a smart way to fit seq_length.
  # Simple cases (like this example) can skip this step and let
  # the next step apply a default truncation to approximately equal lengths.
  truncated_segments = segments

  # Pack inputs. The details (start/end token ids, dict of output tensors)
  # are model-dependent, so this gets loaded from the SavedModel.
  packer = hub.KerasLayer(bert_preprocess.bert_pack_inputs,
                          arguments=dict(seq_length=seq_length),
                          name='packer')
  model_inputs = packer(truncated_segments)
  return tf.keras.Model(input_segments, model_inputs)
```

Let's demonstrate the preprocessing model. You will create a test with two sentences input (input1 and input2). The output is what a BERT model would expect as input: `input_word_ids`, `input_masks` and `input_type_ids`.

```
test_preprocess_model = make_bert_preprocess_model(['my_input1', 'my_input2'])
test_text = [np.array(['some random test sentence']),
             np.array(['another sentence'])]
text_preprocessed = test_preprocess_model(test_text)

print('Keys           : ', list(text_preprocessed.keys()))
print('Shape Word Ids : ', text_preprocessed['input_word_ids'].shape)
print('Word Ids       : ', text_preprocessed['input_word_ids'][0, :16])
print('Shape Mask     : ', text_preprocessed['input_mask'].shape)
print('Input Mask     : ', text_preprocessed['input_mask'][0, :16])
print('Shape Type Ids : ', text_preprocessed['input_type_ids'].shape)
print('Type Ids       : ', text_preprocessed['input_type_ids'][0, :16])
```

Let's take a look at the model's structure, paying attention to the two inputs you just defined.

```
tf.keras.utils.plot_model(test_preprocess_model)
```

To apply the preprocessing in all the inputs from the dataset, you will use the `map` function from the dataset. The result is then cached for performance.

```
AUTOTUNE = tf.data.AUTOTUNE


def load_dataset_from_tfds(in_memory_ds, info, split, batch_size,
                           bert_preprocess_model):
  is_training = split.startswith('train')
  dataset = tf.data.Dataset.from_tensor_slices(in_memory_ds[split])
  num_examples = info.splits[split].num_examples

  if is_training:
    dataset = dataset.shuffle(num_examples)
    dataset = dataset.repeat()
  dataset = dataset.batch(batch_size)
  dataset = dataset.map(lambda ex: (bert_preprocess_model(ex), ex['label']))
  dataset = dataset.cache().prefetch(buffer_size=AUTOTUNE)
  return dataset, num_examples
```

Define your model

You are now ready to define your model for sentence or sentence pair classification by feeding the preprocessed inputs through the BERT encoder and putting a linear classifier on top (or other arrangement of layers as you prefer), and using dropout for regularization.

Note: Here the model will be defined using the Keras functional API

```
def build_classifier_model(num_classes):
  inputs = dict(
      input_word_ids=tf.keras.layers.Input(shape=(None,), dtype=tf.int32),
      input_mask=tf.keras.layers.Input(shape=(None,), dtype=tf.int32),
      input_type_ids=tf.keras.layers.Input(shape=(None,), dtype=tf.int32),
  )

  encoder = hub.KerasLayer(tfhub_handle_encoder, trainable=True, name='encoder'
  net = encoder(inputs)['pooled_output']
  net = tf.keras.layers.Dropout(rate=0.1)(net)
  net = tf.keras.layers.Dense(num_classes, activation=None, name='classifier')(
  return tf.keras.Model(inputs, net, name='prediction')
```

Let's try running the model on some preprocessed inputs.

```
test_classifier_model = build_classifier_model(2)
bert_raw_result = test_classifier_model(text_preprocessed)
print(tf.sigmoid(bert_raw_result))
```

Let's take a look at the model's structure. You can see the three BERT expected inputs.

```
tf.keras.utils.plot_model(test_classifier_model)
```

Choose a task from GLUE

You are going to use a TensorFlow DataSet from the GLUE benchmark suite.

Colab lets you download these small datasets to the local filesystem, and the code below reads them entirely into memory, because the separate TPU worker host cannot access the local filesystem of the colab runtime.

For bigger datasets, you'll need to create your own Google Cloud Storage bucket and have the TPU worker read the data from there. You can learn more in the TPU guide.

It's recommended to start with the CoLa dataset (for single sentence) or MRPC (for multi sentence) since these are small and don't take long to fine tune.

tfds_name: glue/cola

The dataset also determines the problem type (classification or regression) and the appropriate loss function for training.

```
def get_configuration(glue_task):

  loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

  if glue_task == 'glue/cola':
    metrics = tfa.metrics.MatthewsCorrelationCoefficient(num_classes=2)
  else:
    metrics = tf.keras.metrics.SparseCategoricalAccuracy(
        'accuracy', dtype=tf.float32)

  return metrics, loss
```

Train your model

Finally, you can train the model end-to-end on the dataset you chose.

Distribution

Recall the set-up code at the top, which has connected the colab runtime to a TPU worker with multiple TPU devices. To distribute training onto them, you will create and compile your main Keras model within the scope of the TPU distribution strategy. (For details, see Distributed training with Keras.)

Preprocessing, on the other hand, runs on the CPU of the worker host, not the TPUs, so the Keras model for preprocessing as well as the training and validation datasets mapped with it are built outside the distribution strategy scope. The call to `Model.fit()` will take care of distributing the passed-in dataset to the model replicas.

Note: The single TPU worker host already has the resource objects (think: a lookup table) needed for tokenization. Scaling up to multiple workers requires use of `Strategy.experimental_distribute_datasets_from_function` with a function that loads the preprocessing model separately onto each worker.

Optimizer

Fine-tuning follows the optimizer set-up from BERT pre-training (as in Classify text with BERT): It uses the AdamW optimizer with a linear decay of a notional initial learning rate, prefixed with a linear warm-up phase over the first 10% of training steps (`num_warmup_steps`). In line with the BERT paper, the initial learning rate is smaller for fine-tuning (best of 5e-5, 3e-5, 2e-5).

```
epochs = 3
batch_size = 32
init_lr = 2e-5

print(f'Fine tuning {tfhub_handle_encoder} model')
bert_preprocess_model = make_bert_preprocess_model(sentence_features)

with strategy.scope():

  # metric have to be created inside the strategy scope
  metrics, loss = get_configuration(tfds_name)

  train_dataset, train_data_size = load_dataset_from_tfds(
      in_memory_ds, tfds_info, train_split, batch_size, bert_preprocess_model)
  steps_per_epoch = train_data_size // batch_size
  num_train_steps = steps_per_epoch * epochs
  num_warmup_steps = num_train_steps // 10

  validation_dataset, validation_data_size = load_dataset_from_tfds(
      in_memory_ds, tfds_info, validation_split, batch_size,
      bert_preprocess_model)
  validation_steps = validation_data_size // batch_size

  classifier_model = build_classifier_model(num_classes)

  optimizer = optimization.create_optimizer(
      init_lr=init_lr,
      num_train_steps=num_train_steps,
      num_warmup_steps=num_warmup_steps,
      optimizer_type='adamw')

  classifier_model.compile(optimizer=optimizer, loss=loss, metrics=[metrics])

  classifier_model.fit(
      x=train_dataset,
      validation_data=validation_dataset,
      steps_per_epoch=steps_per_epoch,
      epochs=epochs,
      validation_steps=validation_steps)
```

Export for inference

You will create a final model that has the preprocessing part and the fine-tuned BERT we've just created.

At inference time, preprocessing needs to be part of the model (because there is no longer a separate input queue as for training data that does it). Preprocessing is not just computation; it has its own resources (the vocab table) that must be attached to the Keras Model that is saved for export. This final assembly is what will be saved.

You are going to save the model on colab and later you can download to keep it for the future (**View -> Table of contents -> Files**).

```
main_save_path = './my_models'
bert_type = tfhub_handle_encoder.split('/')[-2]
saved_model_name = f'{tfds_name.replace("/", "_")}_{bert_type}'

saved_model_path = os.path.join(main_save_path, saved_model_name)

preprocess_inputs = bert_preprocess_model.inputs
bert_encoder_inputs = bert_preprocess_model(preprocess_inputs)
bert_outputs = classifier_model(bert_encoder_inputs)
model_for_export = tf.keras.Model(preprocess_inputs, bert_outputs)

print('Saving', saved_model_path)
```

```
# Save everything on the Colab host (even the variables from TPU memory)
save_options = tf.saved_model.SaveOptions(experimental_io_device='/job:localhos
model_for_export.save(saved_model_path, include_optimizer=False,
                      options=save_options)
```

Test the model

The final step is testing the results of your exported model.

Just to make some comparison, let's reload the model and test it using some inputs from the test split from the dataset.

Note: The test is done on the colab host, not the TPU worker that it has connected to, so it appears below with explicit device placements. You can omit those when loading the SavedModel elsewhere.

```
with tf.device('/job:localhost'):
  reloaded_model = tf.saved_model.load(saved_model_path)
```

Utility methods

Test

```
with tf.device('/job:localhost'):
  test_dataset = tf.data.Dataset.from_tensor_slices(in_memory_ds[test_split])
  for test_row in test_dataset.shuffle(1000).map(prepare).take(5):
    if len(sentence_features) == 1:
      result = reloaded_model(test_row[0])
    else:
      result = reloaded_model(list(test_row))

    print_bert_results(test_row, result, tfds_name)
```

If you want to use your model on TF Serving, remember that it will call your SavedModel through one of its named signatures. Notice there are some small differences in the input. In Python, you can test them as follows:

```
with tf.device('/job:localhost'):
  serving_model = reloaded_model.signatures['serving_default']
  for test_row in test_dataset.shuffle(1000).map(prepare_serving).take(5):
    result = serving_model(**test_row)
    # The 'prediction' key is the classifier's defined model name.
    print_bert_results(list(test_row.values()), result['prediction'], tfds_name
```

You did it! Your saved model could be used for serving or simple inference in a process, with a simpler api with less code and easier to maintain.

Next Steps

Now that you've tried one of the base BERT models, you can try other ones to achieve more accuracy or maybe with smaller model versions.

You can also try in other datasets.