Text classification with an RNN

View on TensorFlow.org Run in Google Colab View source on GitHub Download notebook

This text classification tutorial trains a recurrent neural network on the IMDB large movie review dataset for sentiment analysis.

Setup

```
import numpy as np

import tensorflow_datasets as tfds
import tensorflow as tf

tfds.disable_progress_bar()
```

Import `matplotlib` and create a helper function to plot graphs:

```
import matplotlib.pyplot as plt


def plot_graphs(history, metric):
  plt.plot(history.history[metric])
  plt.plot(history.history['val_'+metric], '')
  plt.xlabel("Epochs")
  plt.ylabel(metric)
  plt.legend([metric, 'val_'+metric])
```

Setup input pipeline

The IMDB large movie review dataset is a *binary classification* dataset—all the reviews have either a *positive* or *negative* sentiment.

Download the dataset using TFDS. See the loading text tutorial for details on how to load this sort of data manually.

```
dataset, info = tfds.load('imdb_reviews', with_info=True,
                          as_supervised=True)
train_dataset, test_dataset = dataset['train'], dataset['test']

train_dataset.element_spec
```

```
Downloading and preparing dataset Unknown size (download: Unknown size, generated: Unknown size, total: Unknown size) to /ro
Dataset imdb_reviews downloaded and prepared to /root/tensorflow_datasets/imdb_reviews/plain_text/1.0.0. Subsequent calls wi
(TensorSpec(shape=(), dtype=tf.string, name=None),
 TensorSpec(shape=(), dtype=tf.int64, name=None))
```

Initially this returns a dataset of (text, label pairs):

```
for example, label in train_dataset.take(1):
  print('text: ', example.numpy())
  print('label: ', label.numpy())
```

```
text:  b"This was an absolutely terrible movie. Don't be lured in by Christopher Walken or Michael Ironside. Both are great
label:  0
```

Next shuffle the data for training and create batches of these `(text, label)` pairs:

```
BUFFER_SIZE = 10000
BATCH_SIZE = 64


train_dataset = train_dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
test_dataset = test_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)


for example, label in train_dataset.take(1):
  print('texts: ', example.numpy()[:3])
  print()
  print('labels: ', label.numpy()[:3])
```

```
texts:  [b'This movie is just not worth your time. Its reliance upon New-Age mysticism serves as its only semi-interesting d
 b'Dieter Bohlen, Germany\'s notorious composer and producer of slightly trashy pop hits like "You\'re my heart, you\'re my
 b'Working-class romantic drama from director Martin Ritt is as unbelievable as they come, yet there are moments of pleasure

labels:  [0 0 1]
```

Create the text encoder

The raw text loaded by `tfds` needs to be processed before it can be used in a model. The simplest way to process text for training is using the `TextVectorization` layer. This layer has many capabilities, but this tutorial sticks to the default behavior.

Create the layer, and pass the dataset's text to the layer's `.adapt` method:

```
VOCAB_SIZE = 1000
encoder = tf.keras.layers.TextVectorization(
    max_tokens=VOCAB_SIZE)
encoder.adapt(train_dataset.map(lambda text, label: text))
```

```
WARNING:tensorflow:From /usr/local/lib/python3.9/dist-packages/tensorflow/python/autograph/pyct/static_analysis/liveness.py:
Instructions for updating:
Lambda fuctions will be no more assumed to be used in the statement where they are used, or at least in the same block. http
```

The `.adapt` method sets the layer's vocabulary. Here are the first 20 tokens. After the padding and unknown tokens they're sorted by frequency:

```
vocab = np.array(encoder.get_vocabulary())
vocab[:20]
```

```
array(['', '[UNK]', 'the', 'and', 'a', 'of', 'to', 'is', 'in', 'it', 'i',
       'this', 'that', 'br', 'was', 'as', 'for', 'with', 'movie', 'but'],
      dtype='<U14')
```

Once the vocabulary is set, the layer can encode text into indices. The tensors of indices are 0-padded to the longest sequence in the batch (unless you set a fixed `output_sequence_length`):

```
encoded_example = encoder(example)[:3].numpy()
encoded_example
```

```
array([[ 11,  18,   7, ...,   0,   0,   0],
       [  1,   1,   1, ...,   0,   0,   0],
       [  1, 718, 477, ...,   0,   0,   0]])
```

With the default settings, the process is not completely reversible. There are three main reasons for that:

1. The default value for `preprocessing.TextVectorization`'s `standardize` argument is `"lower_and_strip_punctuation"`.
2. The limited vocabulary size and lack of character-based fallback results in some unknown tokens.

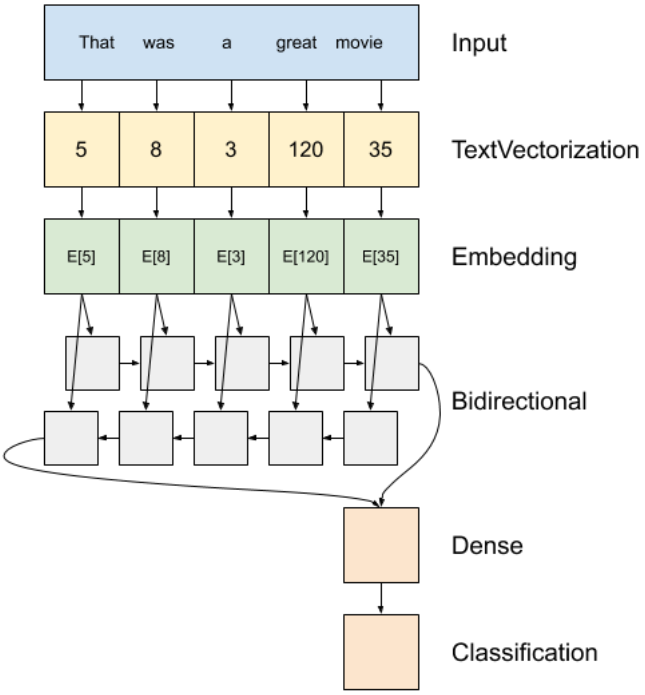
```
for n in range(3):
  print("Original: ", example[n].numpy())
  print("Round-trip: ", " ".join(vocab[encoded_example[n]]))
  print()
```

```
Original:  b'This movie is just not worth your time. Its reliance upon New-Age mysticism serves as its only semi-interesting
Round-trip:  this movie is just not worth your time its [UNK] upon [UNK] [UNK] [UNK] as its only [UNK] [UNK] the plot is one

Original:  b'Dieter Bohlen, Germany\'s notorious composer and producer of slightly trashy pop hits like "You\'re my heart, y
Round-trip:  [UNK] [UNK] [UNK] [UNK] [UNK] and [UNK] of [UNK] [UNK] [UNK] [UNK] like youre my heart youre my [UNK] felt the

Original:  b'Working-class romantic drama from director Martin Ritt is as unbelievable as they come, yet there are moments o
Round-trip:  [UNK] romantic drama from director [UNK] [UNK] is as [UNK] as they come yet there are moments of [UNK] due most
```

Create the model



Above is a diagram of the model.

1. This model can be build as a `tf.keras.Sequential`.
2. The first layer is the `encoder`, which converts the text to a sequence of token indices.
3. After the encoder is an embedding layer. An embedding layer stores one vector per word. When called, it converts the sequences of word indices to sequences of vectors. These vectors are trainable. After training (on enough data), words with similar meanings often have similar vectors.

   This index-lookup is much more efficient than the equivalent operation of passing a one-hot encoded vector through a `tf.keras.layers.Dense` layer.
4. A recurrent neural network (RNN) processes sequence input by iterating through the elements. RNNs pass the outputs from one timestep to their input on the next timestep.

   The `tf.keras.layers.Bidirectional` wrapper can also be used with an RNN layer. This propagates the input forward and backwards through the RNN layer and then concatenates the final output.

   - The main advantage of a bidirectional RNN is that the signal from the beginning of the input doesn't need to be processed all the way through every timestep to affect the output.
   - The main disadvantage of a bidirectional RNN is that you can't efficiently stream predictions as words are being added to the end.
5. After the RNN has converted the sequence to a single vector the two `layers.Dense` do some final processing, and convert from this vector representation to a single logit as the classification output.

The code to implement this is below:

```
model = tf.keras.Sequential([
    encoder,
    tf.keras.layers.Embedding(
        input_dim=len(encoder.get_vocabulary()),
        output_dim=64,
        # Use masking to handle the variable sequence lengths
        mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)
])
```

Please note that Keras sequential model is used here since all the layers in the model only have single input and produce single output. In case you want to use stateful RNN layer, you might want to build your model with Keras functional API or model subclassing so that you can retrieve and reuse the RNN layer states. Please check Keras RNN guide for more details.

The embedding layer uses masking to handle the varying sequence-lengths. All the layers after the `Embedding` support masking:

```
print([layer.supports_masking for layer in model.layers])
```

```
[False, True, True, True, True]
```

To confirm that this works as expected, evaluate a sentence twice. First, alone so there's no padding to mask:

```
# predict on a sample text without padding.

sample_text = ('The movie was cool. The animation and the graphics '
               'were out of this world. I would recommend this movie.')
predictions = model.predict(np.array([sample_text]))
print(predictions[0])
```

```
1/1 [==============================] - 6s 6s/step
[0.00053373]
```

Now, evaluate it again in a batch with a longer sentence. The result should be identical:

```
# predict on a sample text with padding

padding = "the " * 2000
predictions = model.predict(np.array([sample_text, padding]))
print(predictions[0])
```

```
1/1 [==============================] - 0s 66ms/step
[0.00053373]
```

Compile the Keras model to configure the training process:

```
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(1e-4),
              metrics=['accuracy'])
```

Train the model

```
history = model.fit(train_dataset, epochs=10,
                    validation_data=test_dataset,
                    validation_steps=30)
```

```
Epoch 1/10
391/391 [==============================] - 47s 95ms/step - loss: 0.6316 - accuracy: 0.5838 - val_loss: 0.4550 - val_accuracy
Epoch 2/10
391/391 [==============================] - 24s 62ms/step - loss: 0.3753 - accuracy: 0.8276 - val_loss: 0.3501 - val_accuracy
Epoch 3/10
391/391 [==============================] - 23s 59ms/step - loss: 0.3312 - accuracy: 0.8522 - val_loss: 0.3332 - val_accuracy
Epoch 4/10
391/391 [==============================] - 23s 58ms/step - loss: 0.3190 - accuracy: 0.8598 - val_loss: 0.3278 - val_accuracy
Epoch 5/10
391/391 [==============================] - 22s 56ms/step - loss: 0.3090 - accuracy: 0.8669 - val_loss: 0.3224 - val_accuracy
Epoch 6/10
```

```
391/391 [==============================] - 23s 57ms/step - loss: 0.3055 - accuracy: 0.8682 - val_loss: 0.3475 - val_accuracy
Epoch 7/10
391/391 [==============================] - 22s 55ms/step - loss: 0.3025 - accuracy: 0.8677 - val_loss: 0.3183 - val_accuracy
Epoch 8/10
391/391 [==============================] - 22s 57ms/step - loss: 0.3017 - accuracy: 0.8692 - val_loss: 0.3159 - val_accuracy
Epoch 9/10
391/391 [==============================] - 23s 58ms/step - loss: 0.2995 - accuracy: 0.8704 - val_loss: 0.3279 - val_accuracy
Epoch 10/10
391/391 [==============================] - 23s 59ms/step - loss: 0.2984 - accuracy: 0.8720 - val_loss: 0.3214 - val_accuracy
```

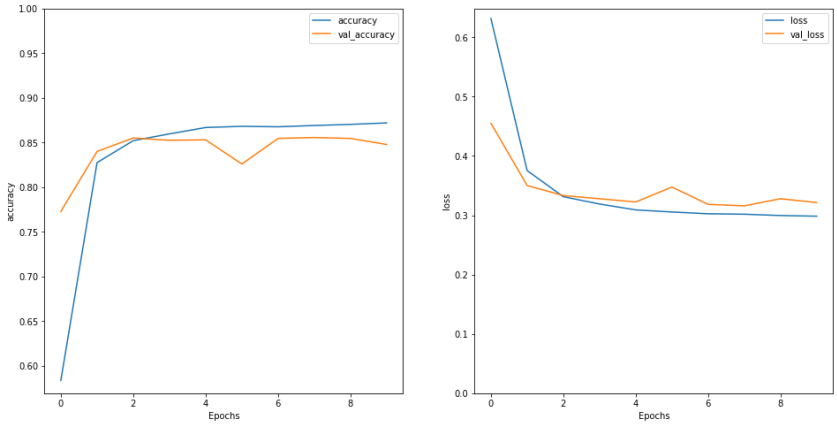
```
test_loss, test_acc = model.evaluate(test_dataset)

print('Test Loss:', test_loss)
print('Test Accuracy:', test_acc)
```

```
391/391 [==============================] - 10s 26ms/step - loss: 0.3172 - accuracy: 0.8518
Test Loss: 0.317198783159256
Test Accuracy: 0.8517600297927856
```

```
plt.figure(figsize=(16, 8))
plt.subplot(1, 2, 1)
plot_graphs(history, 'accuracy')
plt.ylim(None, 1)
plt.subplot(1, 2, 2)
plot_graphs(history, 'loss')
plt.ylim(0, None)
```

```
(0.0, 0.6482444569468498)
```



Run a prediction on a new sentence:

If the prediction is >= 0.0, it is positive else it is negative.

```
sample_text = ('The movie was cool. The animation and the graphics '
               'were out of this world. I would recommend this movie.')
predictions = model.predict(np.array([sample_text]))
```

```
1/1 [==============================] - 3s 3s/step
```

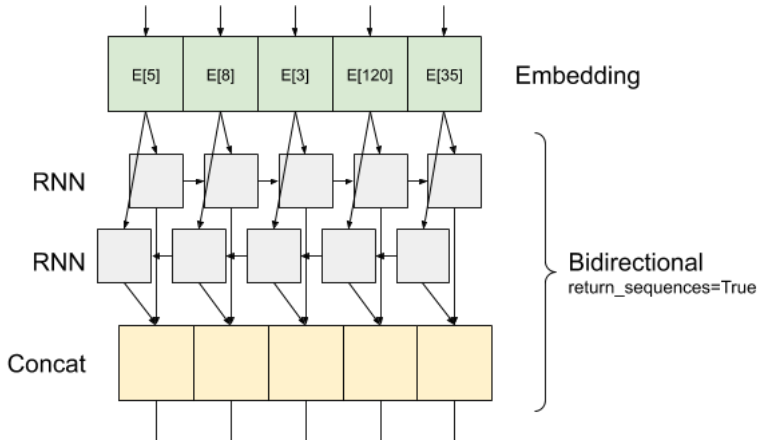
Stack two or more LSTM layers

Keras recurrent layers have two available modes that are controlled by the `return_sequences` constructor argument:

- If `False` it returns only the last output for each input sequence (a 2D tensor of shape (batch_size, output_features)). This is the default, used in the previous model.

- If `True` the full sequences of successive outputs for each timestep is returned (a 3D tensor of shape `(batch_size, timesteps, output_features)`).

Here is what the flow of information looks like with `return_sequences=True`:



The interesting thing about using an `RNN` with `return_sequences=True` is that the output still has 3-axes, like the input, so it can be passed to another RNN layer, like this:

```
model = tf.keras.Sequential([
    encoder,
    tf.keras.layers.Embedding(len(encoder.get_vocabulary()), 64, mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64,  return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1)
])
```

```
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(1e-4),
              metrics=['accuracy'])
```

```
history = model.fit(train_dataset, epochs=10,
                    validation_data=test_dataset,
                    validation_steps=30)
```

```
Epoch 1/10
391/391 [==============================] - 72s 142ms/step - loss: 0.6148 - accuracy: 0.6011 - val_loss: 0.4169 - val_accurac
Epoch 2/10
391/391 [==============================] - 43s 109ms/step - loss: 0.3808 - accuracy: 0.8314 - val_loss: 0.3597 - val_accurac
Epoch 3/10
391/391 [==============================] - 42s 108ms/step - loss: 0.3347 - accuracy: 0.8557 - val_loss: 0.3401 - val_accurac
Epoch 4/10
391/391 [==============================] - 42s 108ms/step - loss: 0.3190 - accuracy: 0.8631 - val_loss: 0.3241 - val_accurac
Epoch 5/10
391/391 [==============================] - 41s 105ms/step - loss: 0.3125 - accuracy: 0.8666 - val_loss: 0.3337 - val_accurac
Epoch 6/10
391/391 [==============================] - 42s 108ms/step - loss: 0.3088 - accuracy: 0.8670 - val_loss: 0.3183 - val_accurac
Epoch 7/10
391/391 [==============================] - 42s 107ms/step - loss: 0.3037 - accuracy: 0.8694 - val_loss: 0.3250 - val_accurac
Epoch 8/10
391/391 [==============================] - 42s 107ms/step - loss: 0.2989 - accuracy: 0.8698 - val_loss: 0.3168 - val_accurac
Epoch 9/10
391/391 [==============================] - 42s 107ms/step - loss: 0.3006 - accuracy: 0.8692 - val_loss: 0.3182 - val_accurac
Epoch 10/10
391/391 [==============================] - 44s 111ms/step - loss: 0.2957 - accuracy: 0.8709 - val_loss: 0.3170 - val_accurac
```

```
test_loss, test_acc = model.evaluate(test_dataset)

print('Test Loss:', test_loss)
print('Test Accuracy:', test_acc)
```

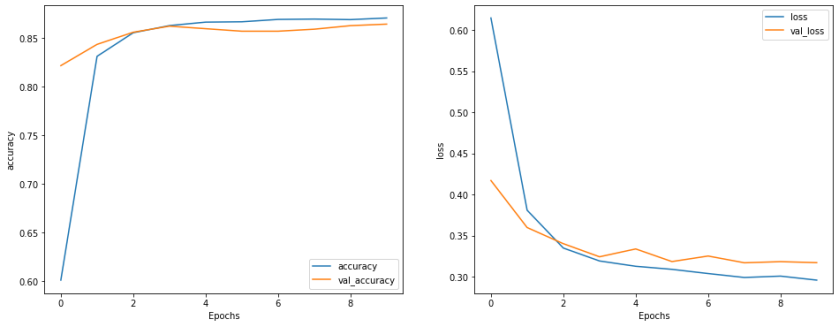
```
391/391 [==============================] - 17s 43ms/step - loss: 0.3183 - accuracy: 0.8639
Test Loss: 0.3183119595050812
Test Accuracy: 0.8638799786567688
```

```
# predict on a sample text without padding.

sample_text = ('The movie was not good. The animation and the graphics '
               'were terrible. I would not recommend this movie.')
predictions = model.predict(np.array([sample_text]))
print(predictions)
```

```
1/1 [==============================] - 5s 5s/step
[[-1.53878]]
```

```
plt.figure(figsize=(16, 6))
plt.subplot(1, 2, 1)
plot_graphs(history, 'accuracy')
plt.subplot(1, 2, 2)
plot_graphs(history, 'loss')
```



Check out other existing recurrent layers such as GRU layers.

If you're interested in building custom RNNs, see the Keras RNN Guide.

✓ 0s completed at 12:00 PM

Show code

Classify text with BERT

This tutorial contains complete code to fine-tune BERT to perform sentiment analysis on a dataset of plain-text IMDB movie reviews. In addition to training a model, you will learn how to preprocess text into an appropriate format.

In this notebook, you will:

- Load the IMDB dataset
- Load a BERT model from TensorFlow Hub
- Build your own model by combining BERT with a classifier
- Train your own model, fine-tuning BERT as part of that
- Save your model and use it to classify sentences

If you're new to working with the IMDB dataset, please see Basic text classification for more details.

About BERT

BERT and other Transformer encoder architectures have been wildly successful on a variety of tasks in NLP (natural language processing). They compute vector-space representations of natural language that are suitable for use in deep learning models. The BERT family of models uses the Transformer encoder architecture to process each token of input text in the full context of all tokens before and after, hence the name: Bidirectional Encoder Representations from Transformers.

BERT models are usually pre-trained on a large corpus of text, then fine-tuned for specific tasks.

Setup

```
# A dependency of the preprocessing for BERT inputs
!pip install -q -U "tensorflow-text==2.11.*"
```

You will use the AdamW optimizer from tensorflow/models.

```
!pip install -q tf-models-official==2.11.0
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.3/2.3 MB 35.0 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 38.2/38.2 MB 13.8 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 630.1/630.1 KB 20.5 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.1/1.1 MB 52.5 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 352.1/352.1 KB 28.9 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 38.1 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 118.9/118.9 KB 12.9 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 238.9/238.9 KB 18.8 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 43.6/43.6 KB 3.9 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
  Building wheel for seqeval (setup.py) ... done
```

```
import os
import shutil

import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_text as text
from official.nlp import optimization  # to create AdamW optimizer
```

```
import matplotlib.pyplot as plt

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

Sentiment analysis

This notebook trains a sentiment analysis model to classify movie reviews as *positive* or *negative*, based on the text of the review.

You'll use the Large Movie Review Dataset that contains the text of 50,000 movie reviews from the Internet Movie Database.

Download the IMDB dataset

Let's download and extract the dataset, then explore the directory structure.

```
url = 'https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz'

dataset = tf.keras.utils.get_file('aclImdb_v1.tar.gz', url,
                                  untar=True, cache_dir='.',
                                  cache_subdir='')

dataset_dir = os.path.join(os.path.dirname(dataset), 'aclImdb')

train_dir = os.path.join(dataset_dir, 'train')

# remove unused folders to make it easier to load the data
remove_dir = os.path.join(train_dir, 'unsup')
shutil.rmtree(remove_dir)
```

```
Downloading data from https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
84125825/84125825 [==============================] - 2s 0us/step
```

Next, you will use the `text_dataset_from_directory` utility to create a labeled `tf.data.Dataset`.

The IMDB dataset has already been divided into train and test, but it lacks a validation set. Let's create a validation set using an 80:20 split of the training data by using the `validation_split` argument below.

Note: When using the `validation_split` and `subset` arguments, make sure to either specify a random seed, or to pass `shuffle=False`, so that the validation and training splits have no overlap.

```
AUTOTUNE = tf.data.AUTOTUNE
batch_size = 32
seed = 42

raw_train_ds = tf.keras.utils.text_dataset_from_directory(
    'aclImdb/train',
    batch_size=batch_size,
    validation_split=0.2,
    subset='training',
    seed=seed)

class_names = raw_train_ds.class_names
train_ds = raw_train_ds.cache().prefetch(buffer_size=AUTOTUNE)

val_ds = tf.keras.utils.text_dataset_from_directory(
    'aclImdb/train',
    batch_size=batch_size,
    validation_split=0.2,
    subset='validation',
    seed=seed)

val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)

test_ds = tf.keras.utils.text_dataset_from_directory(
    'aclImdb/test',
    batch_size=batch_size)

test_ds = test_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

```
Found 25000 files belonging to 2 classes.
Using 20000 files for training.
Found 25000 files belonging to 2 classes.
```

```
Using 5000 files for validation.
Found 25000 files belonging to 2 classes.
```

Let's take a look at a few reviews.

```
for text_batch, label_batch in train_ds.take(1):
  for i in range(3):
    print(f'Review: {text_batch.numpy()[i]}')
    label = label_batch.numpy()[i]
    print(f'Label : {label} ({class_names[label]})')
```

```
Review: b'"Pandemonium" is a horror movie spoof that comes off more stupid than funny. Believe me when I tell you, I love co
Label : 0 (neg)
Review: b"David Mamet is a very interesting and a very un-equal director. His first movie 'House of Games' was the one I lik
Label : 0 (neg)
Review: b'Great documentary about the lives of NY firefighters during the worst terrorist attack of all time.. That reason a
Label : 1 (pos)
```

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.

- 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.

The model documentation on TensorFlow Hub has more details and references to the research literature. Follow the links above, or click on the `tfhub.dev` URL printed after the next cell execution.

The suggestion is to start with a Small BERT (with fewer parameters) since they are faster to fine-tune. If you like a small model but with higher accuracy, ALBERT might be your next option. If you want even better accuracy, choose one of the classic BERT sizes or their recent refinements like Electra, Talking Heads, or a BERT Expert.

Aside from the models available below, there are multiple versions of the models that are larger and can yield even better accuracy, but they are too big to be fine-tuned on a single GPU. You will be able to do that on the Solve GLUE tasks using BERT on a TPU colab.

You'll see in the code below that switching the tfhub.dev URL is enough to try any of these models, because all the differences between them are encapsulated in the SavedModels from TF Hub.

Choose a BERT model to fine-tune

bert_model_name: small_bert/bert_en_uncased_L-4_H-512_A-8

Show code

```
BERT model selected           : https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-4_H-512_A-8/1
Preprocess model auto-selected: https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/3
```

The preprocessing model

Text inputs need to be transformed to numeric token ids and arranged in several Tensors before being input to BERT. TensorFlow Hub provides a matching preprocessing model for each of the BERT models discussed above, which implements this transformation using TF ops from the TF.text library. It is not necessary to run pure Python code outside your TensorFlow model to preprocess text.

The preprocessing model must be the one referenced by the documentation of the BERT model, which you can read at the URL printed above. For BERT models from the drop-down above, the preprocessing model is selected automatically.

Note: You will load the preprocessing model into a hub.KerasLayer to compose your fine-tuned model. This is the preferred API to load a TF2-style SavedModel from TF Hub into a Keras model.

```
bert_preprocess_model = hub.KerasLayer(tfhub_handle_preprocess)
```

Let's try the preprocessing model on some text and see the output:

```
text_test = ['this is such an amazing movie!']
text_preprocessed = bert_preprocess_model(text_test)

print(f'Keys       : {list(text_preprocessed.keys())}')
print(f'Shape      : {text_preprocessed["input_word_ids"].shape}')
print(f'Word Ids   : {text_preprocessed["input_word_ids"][0, :12]}')
print(f'Input Mask : {text_preprocessed["input_mask"][0, :12]}')
print(f'Type Ids   : {text_preprocessed["input_type_ids"][0, :12]}')
```

```
Keys       : ['input_word_ids', 'input_mask', 'input_type_ids']
Shape      : (1, 128)
Word Ids   : [ 101 2023 2003 2107 2019 6429 3185  999  102    0    0    0]
Input Mask : [1 1 1 1 1 1 1 1 1 0 0 0]
Type Ids   : [0 0 0 0 0 0 0 0 0 0 0 0]
```

As you can see, now you have the 3 outputs from the preprocessing that a BERT model would use (`input_words_id`, `input_mask` and `input_type_ids`).

Some other important points:

- The input is truncated to 128 tokens. The number of tokens can be customized, and you can see more details on the Solve GLUE tasks using BERT on a TPU colab.
- The `input_type_ids` only have one value (0) because this is a single sentence input. For a multiple sentence input, it would have one number for each input.

Since this text preprocessor is a TensorFlow model, It can be included in your model directly.

Using the BERT model

Before putting BERT into your own model, let's take a look at its outputs. You will load it from TF Hub and see the returned values.

```
bert_model = hub.KerasLayer(tfhub_handle_encoder)
```

```
bert_results = bert_model(text_preprocessed)

print(f'Loaded BERT: {tfhub_handle_encoder}')
print(f'Pooled Outputs Shape:{bert_results["pooled_output"].shape}')
print(f'Pooled Outputs Values:{bert_results["pooled_output"][0, :12]}')
print(f'Sequence Outputs Shape:{bert_results["sequence_output"].shape}')
print(f'Sequence Outputs Values:{bert_results["sequence_output"][0, :12]}')
```

```
Loaded BERT: https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-4_H-512_A-8/1
Pooled Outputs Shape:(1, 512)
Pooled Outputs Values:[ 0.7626287   0.99280983 -0.18611883  0.36673853  0.15233721  0.6550445
  0.9681154  -0.9486271   0.00216147 -0.98777324  0.06842694 -0.9763059 ]
Sequence Outputs Shape:(1, 128, 512)
Sequence Outputs Values:[[-0.289463    0.34321296  0.33231434 ...  0.2130085   0.7102072
  -0.05771152]
 [-0.28742045  0.3198105  -0.23018534 ...  0.5845504  -0.2132971
   0.72692066]
 [-0.6615697   0.6887678  -0.87433004 ...  0.10877264 -0.26173142
   0.4785532 ]
 ...
 [-0.22561082 -0.28925622 -0.07064454 ...  0.47566032  0.8327713
   0.40025327]
 [-0.29824215 -0.27473125 -0.0545053  ...  0.48849735  1.0955358
   0.18163429]
 [-0.44378236  0.00930765  0.07223701 ...  0.17290132  1.1833243
   0.0789801 ]]
```

The BERT models return a map with 3 important keys: `pooled_output`, `sequence_output`, `encoder_outputs`:

- `pooled_output` represents each input sequence as a whole. The shape is `[batch_size, H]`. You can think of this as an embedding for the entire movie review.
- `sequence_output` represents each input token in the context. The shape is `[batch_size, seq_length, H]`. You can think of this as a contextual embedding for every token in the movie review.

- `encoder_outputs` are the intermediate activations of the `L` Transformer blocks. `outputs["encoder_outputs"][i]` is a Tensor of shape `[batch_size, seq_length, 1024]` with the outputs of the i-th Transformer block, for `0 <= i < L`. The last value of the list is equal to `sequence_output`.

For the fine-tuning you are going to use the `pooled_output` array.

Define your model

You will create a very simple fine-tuned model, with the preprocessing model, the selected BERT model, one Dense and a Dropout layer.

Note: for more information about the base model's input and output you can follow the model's URL for documentation. Here specifically, you don't need to worry about it because the preprocessing model will take care of that for you.

```
def build_classifier_model():
  text_input = tf.keras.layers.Input(shape=(), dtype=tf.string, name='text')
  preprocessing_layer = hub.KerasLayer(tfhub_handle_preprocess, name='preprocessing')
  encoder_inputs = preprocessing_layer(text_input)
  encoder = hub.KerasLayer(tfhub_handle_encoder, trainable=True, name='BERT_encoder')
  outputs = encoder(encoder_inputs)
  net = outputs['pooled_output']
  net = tf.keras.layers.Dropout(0.1)(net)
  net = tf.keras.layers.Dense(1, activation=None, name='classifier')(net)
  return tf.keras.Model(text_input, net)
```

Let's check that the model runs with the output of the preprocessing model.

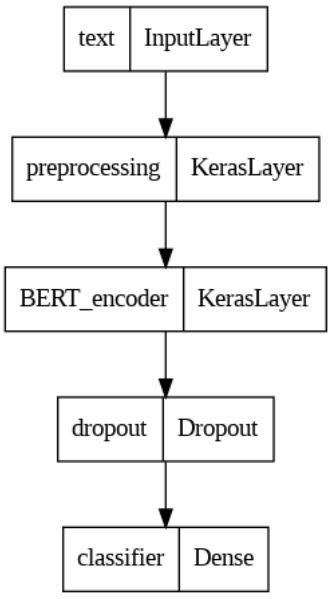
```
classifier_model = build_classifier_model()
bert_raw_result = classifier_model(tf.constant(text_test))
print(tf.sigmoid(bert_raw_result))
```

```
tf.Tensor([[0.386866]], shape=(1, 1), dtype=float32)
```

The output is meaningless, of course, because the model has not been trained yet.

Let's take a look at the model's structure.

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



Model training

You now have all the pieces to train a model, including the preprocessing module, BERT encoder, data, and classifier.

Loss function

Since this is a binary classification problem and the model outputs a probability (a single-unit layer), you'll use `losses.BinaryCrossentropy` loss function.

```
loss = tf.keras.losses.BinaryCrossentropy(from_logits=True)
metrics = tf.metrics.BinaryAccuracy()
```

Optimizer

For fine-tuning, let's use the same optimizer that BERT was originally trained with: the "Adaptive Moments" (Adam). This optimizer minimizes the prediction loss and does regularization by weight decay (not using moments), which is also known as AdamW.

For the learning rate (`init_lr`), you will use the same schedule as BERT pre-training: 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 = 1
steps_per_epoch = tf.data.experimental.cardinality(train_ds).numpy()
num_train_steps = steps_per_epoch * epochs
num_warmup_steps = int(0.1*num_train_steps)

init_lr = 3e-5
optimizer = optimization.create_optimizer(init_lr=init_lr,
                                          num_train_steps=num_train_steps,
                                          num_warmup_steps=num_warmup_steps,
                                          optimizer_type='adamw')
```

Loading the BERT model and training

Using the `classifier_model` you created earlier, you can compile the model with the loss, metric and optimizer.

```
classifier_model.compile(optimizer=optimizer,
                         loss=loss,
                         metrics=metrics)
```

Note: training time will vary depending on the complexity of the BERT model you have selected.

```
print(f'Training model with {tfhub_handle_encoder}')
history = classifier_model.fit(x=train_ds,
                               validation_data=val_ds,
                               epochs=epochs)
```

```
Training model with https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-4_H-512_A-8/1
625/625 [==============================] - 6395s 10s/step - loss: 0.3252 - binary_accuracy: 0.8539 - val_loss: 0.3620 - val_
```

Evaluate the model

Let's see how the model performs. Two values will be returned. Loss (a number which represents the error, lower values are better), and accuracy.

```
loss, accuracy = classifier_model.evaluate(test_ds)

print(f'Loss: {loss}')
print(f'Accuracy: {accuracy}')
```

```
782/782 [==============================] - 1978s 3s/step - loss: 0.3486 - binary_accuracy: 0.8489
Loss: 0.3486011028289795
Accuracy: 0.8489199876785278
```

Plot the accuracy and loss over time

Based on the `History` object returned by `model.fit()`. You can plot the training and validation loss for comparison, as well as the training and validation accuracy:

```
history_dict = history.history
print(history_dict.keys())
```

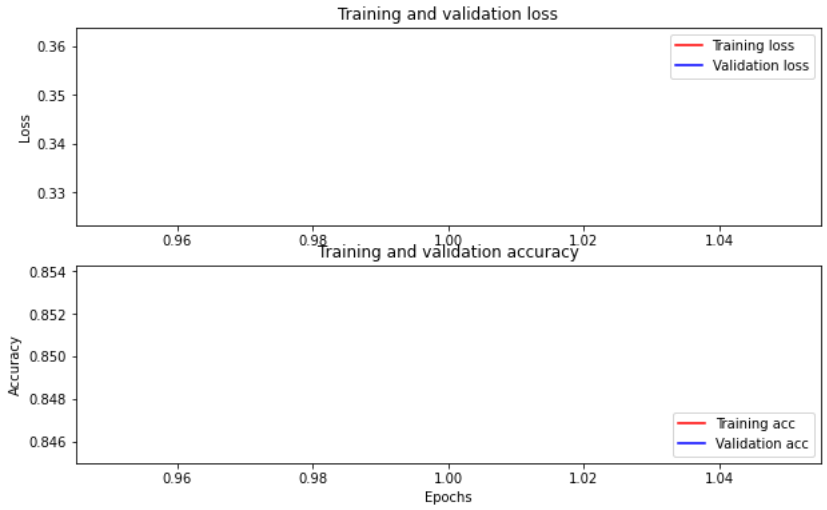
```
acc = history_dict['binary_accuracy']
val_acc = history_dict['val_binary_accuracy']
loss = history_dict['loss']
val_loss = history_dict['val_loss']

epochs = range(1, len(acc) + 1)
fig = plt.figure(figsize=(10, 6))
fig.tight_layout()

plt.subplot(2, 1, 1)
# r is for "solid red line"
plt.plot(epochs, loss, 'r', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
# plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(epochs, acc, 'r', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
```

```
dict_keys(['loss', 'binary_accuracy', 'val_loss', 'val_binary_accuracy'])
<matplotlib.legend.Legend at 0x7f67a07c5d00>
```



In this plot, the red lines represent the training loss and accuracy, and the blue lines are the validation loss and accuracy.

Export for inference

Now you just save your fine-tuned model for later use.

```
dataset_name = 'imdb'
saved_model_path = './{}_bert'.format(dataset_name.replace('/', '_'))

classifier_model.save(saved_model_path, include_optimizer=False)
```

```
WARNING:absl:Found untraced functions such as restored_function_body, restored_function_body, restored_function_body, restor
```

Let's reload the model, so you can try it side by side with the model that is still in memory.

```
reloaded_model = tf.saved_model.load(saved_model_path)
```

Here you can test your model on any sentence you want, just add to the examples variable below.

```
def print_my_examples(inputs, results):
  result_for_printing = \
    [f'input: {inputs[i]:<30} : score: {results[i][0]:.6f}'
                         for i in range(len(inputs))]
  print(*result_for_printing, sep='\n')
  print()


examples = [
    'this is such an amazing movie!',  # this is the same sentence tried earlier
    'The movie was great!',
    'The movie was meh.',
    'The movie was okish.',
    'The movie was terrible...'
]

reloaded_results = tf.sigmoid(reloaded_model(tf.constant(examples)))
original_results = tf.sigmoid(classifier_model(tf.constant(examples)))

print('Results from the saved model:')
print_my_examples(examples, reloaded_results)
print('Results from the model in memory:')
print_my_examples(examples, original_results)
```

```
Results from the saved model:
input: this is such an amazing movie! : score: 0.997409
input: The movie was great!           : score: 0.985886
input: The movie was meh.             : score: 0.880301
input: The movie was okish.           : score: 0.237019
input: The movie was terrible...      : score: 0.003578

Results from the model in memory:
input: this is such an amazing movie! : score: 0.997409
input: The movie was great!           : score: 0.985886
input: The movie was meh.             : score: 0.880301
input: The movie was okish.           : score: 0.237019
input: The movie was terrible...      : score: 0.003578
```

If you want to use your model on TF Serving, remember that it will call your SavedModel through one of its named signatures. In Python, you can test them as follows:

```
serving_results = reloaded_model \
            .signatures['serving_default'](tf.constant(examples))

serving_results = tf.sigmoid(serving_results['classifier'])

print_my_examples(examples, serving_results)
```

```
input: this is such an amazing movie! : score: 0.997409
input: The movie was great!           : score: 0.985886
input: The movie was meh.             : score: 0.880301
input: The movie was okish.           : score: 0.237019
input: The movie was terrible...      : score: 0.003578
```

Next steps

As a next step, you can try Solve GLUE tasks using BERT on a TPU tutorial, which runs on a TPU and shows you how to work with multiple inputs.

✓ 1s completed at 4:55 PM