Open in Colab

Ungraded Lab: Subword Tokenization with the IMDB Reviews Dataset

In this lab, you will look at a pre-tokenized dataset that is using subword text encoding. This is an alternative to word-based tokenization which you have been using in the previous labs. You will see how it works and its implications on preparing your data and training your model.

Let's begin!

Download the IMDB reviews plain text and tokenized datasets

First, you will download the IMDB Reviews dataset from Tensorflow Datasets. You will get two configurations:

- `plain_text` - this is the default and the one you used in Lab 1 of this week
- `subwords8k` - a pre-tokenized dataset (i.e. instead of sentences of type string, it will already give you the tokenized sequences). You will see how this looks in later sections.

```
import tensorflow_datasets as tfds

# Download the plain text default config
imdb_plaintext, info_plaintext = tfds.load("imdb_reviews", with_info=True, as_supervised=True

# Download the subword encoded pretokenized dataset
imdb_subwords, info_subwords = tfds.load("imdb_reviews/subwords8k", with_info=True, as_superv
```

```
oading and preparing dataset imdb_reviews/plain_text/1.0.0 (download: 80.23 MiB, generate
npleted...: 100%       1/1 [00:05<00:00, 5.61s/ url]

...: 100%       80/80 [00:05<00:00, 35.47 MiB/s]


ling and writing examples to /root/tensorflow_datasets/imdb_reviews/plain_text/1.0.0.inco
                                                    24999/25000 [00:00<00:00, 42933.80 examples/s]
ling and writing examples to /root/tensorflow_datasets/imdb_reviews/plain_text/1.0.0.inco
                                                    24999/25000 [00:00<00:00, 46313.97 examples/s]
ling and writing examples to /root/tensorflow_datasets/imdb_reviews/plain_text/1.0.0.inco
                                                    49999/50000 [00:00<00:00, 151216.30 examples/s]
NG:absl:Dataset is using deprecated text encoder API which will be removed soon. Please u
et imdb_reviews downloaded and prepared to /root/tensorflow_datasets/imdb_reviews/plain_t
NG:absl:TFDS datasets with text encoding are deprecated and will be removed in a future v
```

Compare the two datasets

As mentioned, the data types returned by the two datasets will be different. For the default, it will be strings as you also saw in Lab 1. Notice the description of the `text` key below and the sample sentences:

```
                                                    24999/25000 [00:00<00:00, 44443.90 examples/s]
```

```
# Print description of features
info_plaintext.features
```

```
FeaturesDict({
    'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=2),
    'text': Text(shape=(), dtype=tf.string),
})
```

```
# Take 2 training examples and print the text feature
for example in imdb_plaintext['train'].take(2):
  print(example[0].numpy())
```

```
s was an absolutely terrible movie. Don't be lured in by Christopher Walken or Michael I
ave been known to fall asleep during films, but this is usually due to a combination of
```

For `subwords8k`, the dataset is already tokenized so the data type will be integers. Notice that the `text` features also include an `encoder` field and has a `vocab_size` of around 8k, hence the name.

```
# Print description of features
info_subwords.features
```

```
FeaturesDict({
```

```
    'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=2),
    'text': Text(shape=(None,), dtype=tf.int64, encoder=<SubwordTextEncoder vocab_size=8
})
```

If you print the results, you will not see string sentences but a sequence of tokens:

```
# Take 2 training examples and print its contents
for example in imdb_subwords['train'].take(2):
  print(example)
```

```
(<tf.Tensor: shape=(163,), dtype=int64, numpy=
array([  62,   18,   41,  604,  927,   65,    3,  644, 7968,   21,   35,
       5096,   36,   11,   43, 2948, 5240,  102,   50,  681, 7862, 1244,
          3, 3266,   29,  122,  640,    2,   26,   14,  279,  438,   35,
         79,  349,  384,   11, 1991,    3,  492,   79,  122,  188,  117,
         33, 4047, 4531,   14,   65, 7968,    8, 1819, 3947,    3,   62,
         27,    9,   41,  577, 5044, 2629, 2552, 7193, 7961, 3642,    3,
         19,  107, 3903,  225,   85,  198,   72,    1, 1512,  738, 2347,
        102, 6245,    8,   85,  308,   79, 6936, 7961,   23, 4981, 8044,
          3, 6429, 7961, 1141, 1335, 1848, 4848,   55, 3601, 4217, 8050,
          2,    5,   59, 3831, 1484, 8040, 7974,  174, 5773,   22, 5240,
        102,   18,  247,   26,    4, 3903, 1612, 3902,  291,   11,    4,
         27,   13,   18, 4092, 4008, 7961,    6,  119,  213, 2774,    3,
         12,  258, 2306,   13,   91,   29,  171,   52,  229,    2, 1245,
       5790,  995, 7968,    8,   52, 2948, 5240, 8039, 7968,    8,   74,
       1249,    3,   12,  117, 2438, 1369,  192,   39, 7975])>, <tf.Tensor: shape=(), dt
(<tf.Tensor: shape=(142,), dtype=int64, numpy=
array([  12,   31,   93,  867,    7, 1256, 6585, 7961,  421,  365,    2,
         26,   14,    9,  988, 1089,    7,    4, 6728,    6,  276, 5760,
       2587,    2,   81, 6118, 8029,    2,  139, 1892, 7961,    5, 5402,
        246,   25,    1, 1771,  350,    5,  369,   56, 5397,  102,    4,
       2547,    3, 4001,   25,   14, 7822,  209,   12, 3531, 6585, 7961,
         99,    1,   32,   18, 4762,    3,   19,  184, 3223,   18, 5855,
       1045,    3, 4232, 3337,   64, 1347,    5, 1190,    3, 4459,    8,
        614,    7, 3129,    2,   26,   22,   84, 7020,    6,   71,   18,
       4924, 1160,  161,   50, 2265,    3,   12, 3983,    2,   12,  264,
         31, 2545,  261,    6,    1,   66,    2,   26,  131,  393,    1,
       5846,    6,   15,    5,  473,   56,  614,    7, 1470,    6,  116,
        285, 4755, 2088, 7961,  273,  119,  213, 3414, 7961,   23,  332,
       1019,    3,   12, 7667,  505,   14,   32,   44,  208, 7975])>, <tf.Tensor: shape=
```

You can get the `encoder` object included in the download and use it to decode the sequences above. You'll see that you will arrive at the same sentences provided in the `plain_text` config:

```
# Get the encoder
tokenizer_subwords = info_subwords.features['text'].encoder

# Take 2 training examples and decode the text feature
```

```
for example in imdb_subwords['train'].take(2):
  print(tokenizer_subwords.decode(example[0]))
```

```
This was an absolutely terrible movie. Don't be lured in by Christopher Walken or Michae
I have been known to fall asleep during films, but this is usually due to a combination
```

Note: The documentation for the encoder can be found here but don't worry if it's marked as deprecated. As mentioned, the objective of this exercise is just to show the characteristics of subword encoding.

Subword Text Encoding

From previous labs, the number of tokens in the sequence is the same as the number of words in the text (i.e. word tokenization). The following cells shows a review of this process.

```
# Get the train set
train_data = imdb_plaintext['train']

# Initialize sentences list
training_sentences = []

# Loop over all training examples and save to the list
for s,_ in train_data:
  training_sentences.append(s.numpy().decode('utf8'))
```

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

vocab_size = 10000
oov_tok = '<OOV>'

# Initialize the Tokenizer class
tokenizer_plaintext = Tokenizer(num_words = 10000, oov_token=oov_tok)

# Generate the word index dictionary for the training sentences
tokenizer_plaintext.fit_on_texts(training_sentences)

# Generate the training sequences
sequences = tokenizer_plaintext.texts_to_sequences(training_sentences)
```

The cell above uses a `vocab_size` of 10000 but you'll find that it's easy to find OOV tokens when decoding using the lookup dictionary it created. See the result below:

```
# Decode the first sequence using the Tokenizer class
tokenizer_plaintext.sequences_to_texts(sequences[0:1])
```

```
["this was an absolutely terrible movie don't be <OOV> in by christopher walken or micha
```

For binary classifiers, this might not have a big impact but you may have other applications that will benefit from avoiding OOV tokens when training the model (e.g. text generation). If you want the tokenizer above to not have OOVs, then the `vocab_size` will increase to more than 88k. This can slow down training and bloat the model size. The encoder also won't be robust when used on other datasets which may contain new words, thus resulting in OOVs again.

```
# Total number of words in the word index dictionary
len(tokenizer_plaintext.word_index)
```

```
88583
```

Subword text encoding gets around this problem by using parts of the word to compose whole words. This makes it more flexible when it encounters uncommon words. See how these subwords look like for this particular encoder:

```
# Print the subwords
print(tokenizer_subwords.subwords)
```

```
['the_', ', ', '. ', 'a_', 'and_', 'of_', 'to_', 's_', 'is_', 'br', 'in_', 'I_', 'that_
```

If you use it on the previous plain text sentence, you'll see that it won't have any OOVs even if it has a smaller vocab size (only 8k compared to 10k above):

```
# Encode the first plaintext sentence using the subword text encoder
tokenized_string = tokenizer_subwords.encode(training_sentences[0])
print(tokenized_string)

# Decode the sequence
original_string = tokenizer_subwords.decode(tokenized_string)

# Print the result
print (original_string)
```

```
927, 65, 3, 644, 7968, 21, 35, 5096, 36, 11, 43, 2948, 5240, 102, 50, 681, 7862, 1244, 3,
tely terrible movie. Don't be lured in by Christopher Walken or Michael Ironside. Both ar
```

Subword encoding can even perform well on words that are not commonly found on movie reviews. See first the result when using the plain text tokenizer. As expected, it will show many OOVs:

```
# Define sample sentence
sample_string = 'TensorFlow, from basics to mastery'

# Encode using the plain text tokenizer
tokenized_string = tokenizer_plaintext.texts_to_sequences([sample_string])
print ('Tokenized string is {}'.format(tokenized_string))

# Decode and print the result
original_string = tokenizer_plaintext.sequences_to_texts(tokenized_string)
print ('The original string: {}'.format(original_string))
```

```
    Tokenized string is [[1, 37, 1, 6, 1]]
    The original string: ['<OOV> from <OOV> to <OOV>']
```

Then compare to the subword text encoder:

```
# Encode using the subword text encoder
tokenized_string = tokenizer_subwords.encode(sample_string)
print ('Tokenized string is {}'.format(tokenized_string))

# Decode and print the results
original_string = tokenizer_subwords.decode(tokenized_string)
print ('The original string: {}'.format(original_string))
```

```
    Tokenized string is [6307, 2327, 4043, 2120, 2, 48, 4249, 4429, 7, 2652, 8050]
    The original string: TensorFlow, from basics to mastery
```

As you may notice, the sentence is correctly decoded. The downside is the token sequence is much longer. Instead of only 5 when using word-encoding, you ended up with 11 tokens instead. The mapping for this sentence is shown below:

```
# Show token to subword mapping:
for ts in tokenized_string:
  print ('{} ----> {}'.format(ts, tokenizer_subwords.decode([ts])))
```

```
    6307 ----> Ten
    2327 ----> sor
    4043 ----> Fl
    2120 ----> ow
    2 ----> ,
```

```
48 ----> from
4249 ----> basi
4429 ----> cs
7 ----> to
2652 ----> master
8050 ----> y
```

Training the model

You will now train your model using this pre-tokenized dataset. Since these are already saved as sequences, you can jump straight to making uniform sized arrays for the train and test sets. These are also saved as `tf.data.Dataset` type so you can use the `padded_batch()` method to create batches and pad the arrays into a uniform size for training.

```
BUFFER_SIZE = 10000
BATCH_SIZE = 64

# Get the train and test splits
train_data, test_data = imdb_subwords['train'], imdb_subwords['test'],

# Shuffle the training data
train_dataset = train_data.shuffle(BUFFER_SIZE)

# Batch and pad the datasets to the maximum length of the sequences
train_dataset = train_dataset.padded_batch(BATCH_SIZE)
test_dataset = test_data.padded_batch(BATCH_SIZE)
```

Next, you will build the model. You can just use the architecture from the previous lab.

```
import tensorflow as tf

# Define dimensionality of the embedding
embedding_dim = 64

# Build the model
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(tokenizer_subwords.vocab_size, embedding_dim),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Print the model summary
model.summary()
```

Similarly, you can use the same parameters for training. In Colab, it will take around 20 seconds per epoch (without an accelerator) and you will reach around 94% training accuracy and 88% validation accuracy.

```
num_epochs = 10

# Set the training parameters
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

# Start training
history = model.fit(train_dataset, epochs=num_epochs, validation_data=test_dataset)
```

```
Epoch 1/10
391/391 [==============================] - 18s 43ms/step - loss: 0.6841 - accuracy: 0.56
Epoch 2/10
391/391 [==============================] - 17s 42ms/step - loss: 0.5389 - accuracy: 0.78
Epoch 3/10
391/391 [==============================] - 17s 42ms/step - loss: 0.3572 - accuracy: 0.87
Epoch 4/10
391/391 [==============================] - 17s 42ms/step - loss: 0.2882 - accuracy: 0.89
Epoch 5/10
391/391 [==============================] - 17s 43ms/step - loss: 0.2520 - accuracy: 0.90
Epoch 6/10
391/391 [==============================] - 17s 43ms/step - loss: 0.2261 - accuracy: 0.91
Epoch 7/10
391/391 [==============================] - 17s 43ms/step - loss: 0.2073 - accuracy: 0.92
Epoch 8/10
391/391 [==============================] - 17s 43ms/step - loss: 0.1909 - accuracy: 0.93
Epoch 9/10
391/391 [==============================] - 17s 43ms/step - loss: 0.1783 - accuracy: 0.93
Epoch 10/10
391/391 [==============================] - 17s 43ms/step - loss: 0.1654 - accuracy: 0.94
```

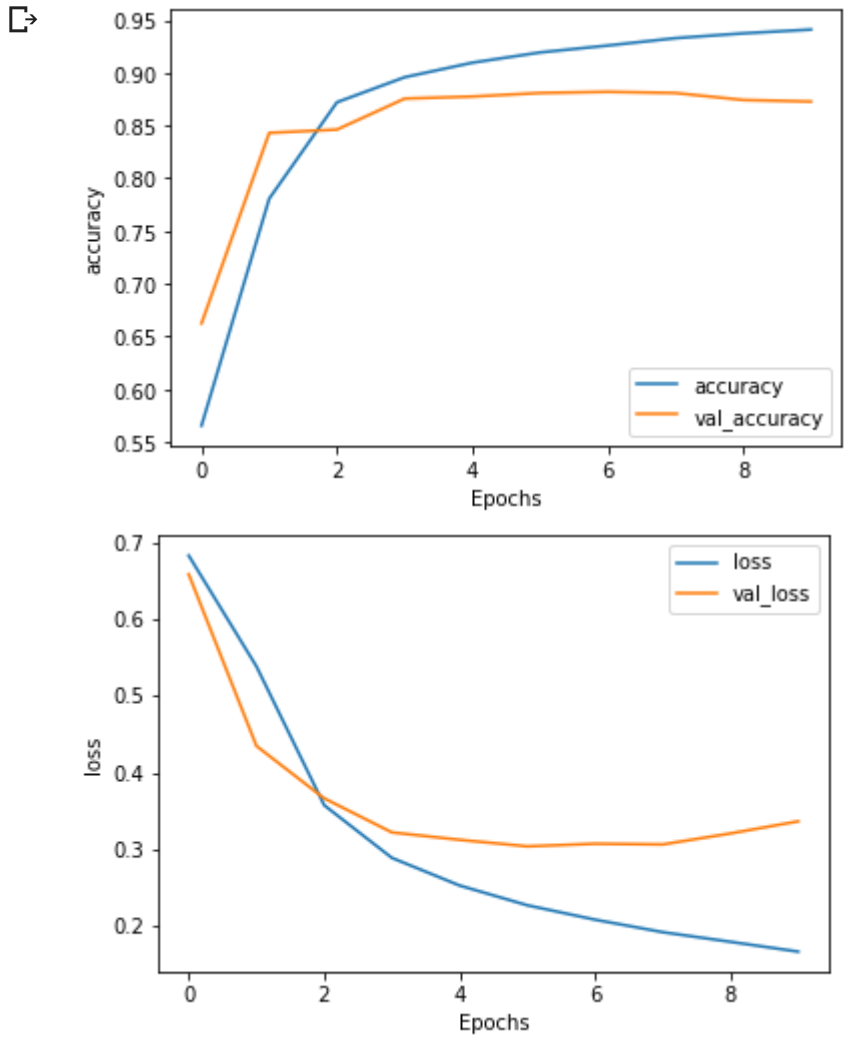
Visualize the results

You can use the cell below to plot the training results. See if you can improve it by tweaking the parameters such as the size of the embedding and number of epochs.

```
import matplotlib.pyplot as plt

# Plot utility
def plot_graphs(history, string):
  plt.plot(history.history[string])
  plt.plot(history.history['val_'+string])
  plt.xlabel("Epochs")
  plt.ylabel(string)
  plt.legend([string, 'val_'+string])
  plt.show()
```

```
# Plot the accuracy and results
plot_graphs(history, "accuracy")
plot_graphs(history, "loss")
```



Wrap Up

In this lab, you saw how subword text encoding can be a robust technique to avoid out-of-vocabulary tokens. It can decode uncommon words it hasn't seen before even with a relatively small vocab size. Consequently, it results in longer token sequences when compared to full word tokenization. Next week, you will look at other architectures that you can use when building your classifier. These will be recurrent neural networks and convolutional neural networks.

✓ 0s completed at 9:45 PM