Ungraded Lab: Generating Sequences and Padding

In this lab, you will look at converting input sentences into numeric sequences. Similar to images in the previous course, you need to prepare text data with uniform size before feeding it to your model. You will see how to do these in the next sections.

Text to Sequences

In the previous lab, you saw how to use the `TextVectorization` layer to build a vocabulary from your corpus. It generates a list where more frequent words have lower indices.

In [1]:
```
import tensorflow as tf

# Sample inputs
sentences = [
    'I love my dog',
    'I love my cat',
    'You love my dog!',
    'Do you think my dog is amazing?'
    ]

# Initialize the layer
vectorize_layer = tf.keras.layers.TextVectorization()

# Compute the vocabulary
vectorize_layer.adapt(sentences)

# Get the vocabulary
vocabulary = vectorize_layer.get_vocabulary()

# Print the token index
for index, word in enumerate(vocabulary):
  print(index, word)
```

```
0
1 [UNK]
2 my
3 love
4 dog
5 you
6 i
7 think
8 is
9 do
10 cat
11 amazing
```

You can then use the result to convert each of the input sentences into integer sequences. See how that's done below given a single input string.

In [2]:
```
# String input
sample_input = 'I love my dog'

# Convert the string input to an integer sequence
sequence = vectorize_layer(sample_input)

# Print the result
print(sequence)
```

```
tf.Tensor([6 3 2 4], shape=(4,), dtype=int64)
```

As shown, you simply pass in the string to the layer which already learned the vocabulary, and it will output the integer sequence as a `tf.Tensor`. In this case, the result is `[6 3 2 4]`. You can look at the token index printed above to verify that it matches the indices for each word in the input string.

For a given list of string inputs (such as the 4-item `sentences` list above), you will need to apply the layer to each input. There's more than one way to do this. Let's first use the `map()` method and see the results.

In [3]:
```
# Convert the list to tf.data.Dataset
sentences_dataset = tf.data.Dataset.from_tensor_slices(sentences)

# Define a mapping function to convert each sample input
sequences = sentences_dataset.map(vectorize_layer)

# Print the integer sequences
for sentence, sequence in zip(sentences, sequences):
  print(f'{sentence} ---> {sequence}')
```

```
I love my dog ---> [6 3 2 4]
I love my cat ---> [ 6  3  2 10]
You love my dog! ---> [5 3 2 4]
Do you think my dog is amazing? ---> [ 9  5  7  2  4  8 11]
```

As you can see, each sentence is successfully transformed into an integer sequence. The problem with this is they have varying lengths so it cannot be consumed by the model right away.

Padding

You can get a list of varying lengths to have a uniform size by padding or truncating tokens from the sequences. Padding is more common to preserve information.

Recall that your vocabulary reserves a special token index `0` for padding. It will add that token (called post padding) if you pass in a list of string inputs to the layer. See an example below. Notice that you have the same output as above but the integer sequences are already post-padded with `0` up to the length of the longest sequence.

In [4]:
```
# Apply the layer to the string input list
sequences_post = vectorize_layer(sentences)

# Print the results
print('INPUT:')
print(sentences)
print()

print('OUTPUT:')
print(sequences_post)
```

```
INPUT:
['I love my dog', 'I love my cat', 'You love my dog!', 'Do you think my dog is amazing?']

OUTPUT:
tf.Tensor(
[[ 6  3  2  4  0  0  0]
 [ 6  3  2 10  0  0  0]
 [ 5  3  2  4  0  0  0]
 [ 9  5  7  2  4  8 11]], shape=(4, 7), dtype=int64)
```

If you want pre-padding, you can use the pad_sequences() utility to prepend a padding token to the sequences. Notice that the `padding` argument is set to `pre` . This is just for clarity. The function already has this set as the default so you can opt to drop it.

In [5]:
```
# Pre-pad the sequences to a uniform length.
# You can remove the `padding` argument and get the same result.
sequences_pre = tf.keras.utils.pad_sequences(sequences, padding='pre')

# Print the results
print('INPUT:')
[print(sequence.numpy()) for sequence in sequences]
print()

print('OUTPUT:')
print(sequences_pre)
```

```
INPUT:
[6 3 2 4]
[ 6  3  2 10]
[5 3 2 4]
[ 9  5  7  2  4  8 11]

OUTPUT:
[[ 0  0  0  6  3  2  4]
 [ 0  0  0  6  3  2 10]
 [ 0  0  0  5  3  2  4]
 [ 9  5  7  2  4  8 11]]
```

If you switch the `padding` argument to `post` , you will arrive at the same result as applying the layer directly.

The function also has a `maxlen` argument that you can use to truncate tokens from the sequences. By default, it will drop tokens in front. If you want to drop tokens at the other end, you will have to set the `truncating` argument to `post` .

In [6]:
```
# Post-pad the sequences and limit the size to 5.
sequences_post_trunc = tf.keras.utils.pad_sequences(sequences, maxlen=5, padding='pre')

# Print the results
print('INPUT:')
[print(sequence.numpy()) for sequence in sequences]
print()

print('OUTPUT:')
print(sequences_post_trunc)
```

```
INPUT:
[6 3 2 4]
[ 6  3  2 10]
[5 3 2 4]
[ 9  5  7  2  4  8 11]

OUTPUT:
[[ 0  6  3  2  4]
 [ 0  6  3  2 10]
 [ 0  5  3  2  4]
 [ 7  2  4  8 11]]
```

Another way to prepare your sequences for prepadding is to set the TextVectorization to output a ragged tensor. This means the output will not be automatically post-padded. See the output sequences here.

In [7]:
```
# Set the layer to output a ragged tensor
vectorize_layer = tf.keras.layers.TextVectorization(ragged=True)

# Compute the vocabulary
vectorize_layer.adapt(sentences)

# Apply the layer to the sentences
ragged_sequences = vectorize_layer(sentences)

# Print the results
print(ragged_sequences)
```

```
<tf.RaggedTensor [[6, 3, 2, 4], [6, 3, 2, 10], [5, 3, 2, 4], [9, 5, 7, 2, 4, 8, 11]]>
```

With that, you can now pass it directly to the `pad_sequences()` utility.

In [8]:
```
# Pre-pad the sequences in the ragged tensor
sequences_pre = tf.keras.utils.pad_sequences(ragged_sequences.numpy())

# Print the results
print(sequences_pre)
```

```
[[ 0  0  0  6  3  2  4]
 [ 0  0  0  6  3  2 10]
 [ 0  0  0  5  3  2  4]
 [ 9  5  7  2  4  8 11]]
```

Out-of-vocabulary tokens

Lastly, you'll see what the other special token is for. The layer will use the token index `1` when you have input words that are not found in the vocabulary list. For example, you may decide to collect more text after your initial training and decide to not recompute the vocabulary. You will see this in action in the cell below. Notice that the token `1` is inserted for words that are not found in the list.

In [9]:
```
# Try with words that are not in the vocabulary
sentences_with_oov = [
    'i really love my dog',
    'my dog loves my manatee'
]

# Generate the sequences
sequences_with_oov = vectorize_layer(sentences_with_oov)

# Print the integer sequences
for sentence, sequence in zip(sentences_with_oov, sequences_with_oov):
  print(f'{sentence} ---> {sequence}')
```

```
i really love my dog ---> [6 1 3 2 4]
my dog loves my manatee ---> [2 4 1 2 1]
```

This concludes another introduction to text data preprocessing. So far, you've just been using dummy data. In the next exercise, you will be applying the same concepts to a real-world and much larger dataset.