Ungraded Lab: Preparing Time Series Features and Labels

In this lab, you will prepare time series data into features and labels that you can use to train a model. This is mainly achieved by a *windowing* technique where in you group consecutive measurement values into one feature and the next measurement will be the label. For example, in hourly measurements, you can use values taken at hours 1 to 11 to predict the value at hour 12. The next sections will show how you can implement this in Tensorflow.

Let's begin!

Imports

Tensorflow will be your lone import in this module and you'll be using methods mainly from the tf.data API, particularly the tf.data.Dataset class. This contains many useful methods to arrange sequences of data and you'll see that shortly.

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

Create a Simple Dataset

For this exercise, you will just use a sequence of numbers as your dataset so you can clearly see the effect of each command. For example, the cell below uses the range() method to generate a dataset containing numbers 0 to 9.

In [2]:
```
# Generate a tf dataset with 10 elements (i.e. numbers 0 to 9)
dataset = tf.data.Dataset.range(10)

# Preview the result
for val in dataset:
   print(val.numpy())
```

```
0
1
2
3
4
5
6
7
8
9
```

You will see this command several times in the next sections.

Windowing the data

As mentioned earlier, you want to group consecutive elements of your data and use that to predict a future value. This is called windowing and you can use that with the window() method as shown below. Here, you will take 5 elements per window (i.e. `size` parameter) and you will move this window 1 element at a time (i.e. `shift` parameter). One caveat to using this method is that each window returned is a Dataset in itself. This is a Python iterable and it won't show the elements if you use the `print()` method on it. It will just show a description of the data structure (e.g. `<_VariantDataset shapes: (), types: tf.int64>`).

In [3]:
```
# Generate a tf dataset with 10 elements (i.e. numbers 0 to 9)
dataset = tf.data.Dataset.range(10)

# Window the data
dataset = dataset.window(size=5, shift=1)

# Print the result
for window_dataset in dataset:
  print(window_dataset)
```

```
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
```

If you want to see the elements, you will have to iterate over each iterable. This can be done by modifying the print statement above with a nested for-loop or list comprehension. The code below shows the list comprehension while in the lecture video, you saw the for-loop.

In [4]:
```python
# Print the result
for window_dataset in dataset:
  print([item.numpy() for item in window_dataset])
```

```
[0, 1, 2, 3, 4]
[1, 2, 3, 4, 5]
[2, 3, 4, 5, 6]
[3, 4, 5, 6, 7]
[4, 5, 6, 7, 8]
[5, 6, 7, 8, 9]
[6, 7, 8, 9]
[7, 8, 9]
[8, 9]
[9]
```

Now that you can see the elements of each window, you'll notice that the resulting sets are not sized evenly because there are no more elements after the number `9`. You can use the `drop_remainder` flag to make sure that only 5-element windows are retained.

In [5]:
```python
# Generate a tf dataset with 10 elements (i.e. numbers 0 to 9)
dataset = tf.data.Dataset.range(10)

# Window the data but only take those with the specified size
dataset = dataset.window(size=5, shift=1, drop_remainder=True)

# Print the result
for window_dataset in dataset:
  print([item.numpy() for item in window_dataset])
```

```
[0, 1, 2, 3, 4]
[1, 2, 3, 4, 5]
[2, 3, 4, 5, 6]
[3, 4, 5, 6, 7]
[4, 5, 6, 7, 8]
[5, 6, 7, 8, 9]
```

Flatten the Windows

In training the model later, you will want to prepare the windows to be tensors instead of the `Dataset` structure. You can do that by feeding a mapping function to the flat_map() method. This function will be applied to each window and the results will be flattened into a single dataset. To illustrate, the code below will put all elements of a window into a single batch then flatten the result.

NOTE: In the mapping function passed to `flat_map()`, it's important to specify the batch size to be the same size as the window (i.e. `5` in this case) so all elements will be in a single list. You can put the size manually or use the cardinality() method to detect the window size automatically. We're using the manual approach in the lectures and other exercises more often, but the other approach will also work. Try replacing `5` with `window.cardinality()` in the lambda function below to verify.

In [6]:
```python
# Generate a tf dataset with 10 elements (i.e. numbers 0 to 9)
dataset = tf.data.Dataset.range(10)

# Window the data but only take those with the specified size
dataset = dataset.window(5, shift=1, drop_remainder=True)

# Flatten the windows by putting its elements in a single batch
dataset = dataset.flat_map(lambda window: window.batch(5))

# Print the results
for window in dataset:
  print(window.numpy())
```

```
[0 1 2 3 4]
[1 2 3 4 5]
[2 3 4 5 6]
[3 4 5 6 7]
[4 5 6 7 8]
[5 6 7 8 9]
```

Group into features and labels

Next, you will want to mark the labels in each window. For this exercise, you will do that by splitting the last element of each window from the first four. This is done with the map() method containing a lambda function that defines the window slicing.

In [7]:
```
# Generate a tf dataset with 10 elements (i.e. numbers 0 to 9)
dataset = tf.data.Dataset.range(10)

# Window the data but only take those with the specified size
dataset = dataset.window(5, shift=1, drop_remainder=True)

# Flatten the windows by putting its elements in a single batch
dataset = dataset.flat_map(lambda window: window.batch(5))

# Create tuples with features (first four elements of the window) and labels (last element)
dataset = dataset.map(lambda window: (window[:-1], window[-1]))

# Print the results
for x,y in dataset:
  print("x = ", x.numpy())
  print("y = ", y.numpy())
  print()
```

```
x =  [0 1 2 3]
y =  4

x =  [1 2 3 4]
y =  5

x =  [2 3 4 5]
y =  6

x =  [3 4 5 6]
y =  7

x =  [4 5 6 7]
y =  8

x =  [5 6 7 8]
y =  9
```

Shuffle the data

It is good practice to shuffle your dataset to reduce *sequence bias* while training your model. This refers to the neural network overfitting to the order of inputs and consequently, it will not perform well when it does not see that particular order when testing. You don't want the sequence of training inputs to impact the network this way so it's good to shuffle them up.

You can simply use the shuffle() method to do this. The `buffer_size` parameter is required for that and as mentioned in the doc, you should put a number equal or greater than the total number of elements for better shuffling. We can see from the previous cells that the total number of windows in the dataset is `6` so we can choose this number or higher.

In [8]:
```
# Generate a tf dataset with 10 elements (i.e. numbers 0 to 9)
dataset = tf.data.Dataset.range(10)

# Window the data but only take those with the specified size
dataset = dataset.window(5, shift=1, drop_remainder=True)

# Flatten the windows by putting its elements in a single batch
dataset = dataset.flat_map(lambda window: window.batch(5))

# Create tuples with features (first four elements of the window) and labels (last element)
dataset = dataset.map(lambda window: (window[:-1], window[-1]))

# Shuffle the windows
dataset = dataset.shuffle(buffer_size=10)
```

```
# Print the results
for x,y in dataset:
  print("x = ", x.numpy())
  print("y = ", y.numpy())
  print()
```

```
x =  [5 6 7 8]
y =  9

x =  [2 3 4 5]
y =  6

x =  [3 4 5 6]
y =  7

x =  [0 1 2 3]
y =  4

x =  [4 5 6 7]
y =  8

x =  [1 2 3 4]
y =  5
```

Create batches for training

Lastly, you will want to group your windows into batches. You can do that with the batch() method as shown below. Simply specify the batch size and it will return a batched dataset with that number of windows. As a rule of thumb, it is also useful to add a cache() and prefetch() step. These optimize the execution time when the model is already training.

By specifying a prefetch `buffer_size` of `1` as shown below, Tensorflow will prepare the next one batch in advance (i.e. putting it in a buffer) while the current batch is being consumed by the model. You can read more about it here. If you've taken the first 3 courses of this Specialization, you'll know that you can also pass in a `tf.data.AUTOTUNE` here to let Tensorflow dynamically change this value at runtime.

In [9]:
```
# Generate a tf dataset with 10 elements (i.e. numbers 0 to 9)
dataset = tf.data.Dataset.range(10)

# Window the data but only take those with the specified size
dataset = dataset.window(5, shift=1, drop_remainder=True)

# Flatten the windows by putting its elements in a single batch
dataset = dataset.flat_map(lambda window: window.batch(5))

# Create tuples with features (first four elements of the window) and labels (last element)
dataset = dataset.map(lambda window: (window[:-1], window[-1]))

# Shuffle the windows
dataset = dataset.shuffle(buffer_size=10)

# Create batches of windows
dataset = dataset.batch(2)

# Optimize the dataset for training
dataset = dataset.cache().prefetch(1)

# Print the results
for x,y in dataset:
  print("x = ", x.numpy())
  print("y = ", y.numpy())
  print()
```

```
x =  [[0 1 2 3]
 [1 2 3 4]]
y =  [4 5]

x =  [[3 4 5 6]
 [5 6 7 8]]
y =  [7 9]

x =  [[2 3 4 5]
 [4 5 6 7]]
y =  [6 8]
```

Wrap Up

This short exercise showed you how to chain different methods of the `tf.data.Dataset` class to prepare a sequence into shuffled and batched window datasets. You will be using this same concept in the next exercises when you apply it to synthetic data and use the result to train a neural network. On to the next!