Classify Structured Data

Import TensorFlow and Other Libraries

In [1]:

```
import pandas as pd
import tensorflow as tf

from tensorflow.keras import layers
from tensorflow import feature_column

from os import getcwd
from sklearn.model_selection import train_test_split
```

Use Pandas to Create a Dataframe

Pandas is a Python library with many helpful utilities for loading and working with structured data. We will use Pandas to download the dataset and load it into a dataframe.

In [2]:

```
filePath = f"{getcwd()}/../tmp2/heart.csv"
dataframe = pd.read_csv(filePath)
dataframe.head()
```

Out[2]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	1	145	233	1	2	150	0	2.3	3	0	fixed	0
1	67	1	4	160	286	0	2	108	1	1.5	2	3	normal	1
2	67	1	4	120	229	0	2	129	1	2.6	2	2	reversible	0
3	37	1	3	130	250	0	0	187	0	3.5	3	0	normal	0
4	41	0	2	130	204	0	2	172	0	1.4	1	0	normal	0

Split the Dataframe Into Train, Validation, and Test Sets

The dataset we downloaded was a single CSV file. We will split this into train, validation, and test sets.

In [3]:

```
train, test = train_test_split(dataframe, test_size=0.2)
train, val = train_test_split(train, test_size=0.2)
print(len(train), 'train examples')
print(len(val), 'validation examples')
print(len(test), 'test examples')
```

```
193 train examples
49 validation examples
61 test examples
```

Create an Input Pipeline Using `tf.data`

Next, we will wrap the dataframes with tf.data. This will enable us to use feature columns as a bridge to map from the columns in the Pandas dataframe to features used to train the model. If we were working with a very large CSV file (so large that it does not fit into memory), we would use tf.data to read it from disk directly.

In [4]:

```
# EXERCISE: A utility method to create a tf.data dataset from a Pandas Dataframe.

def df_to_dataset(dataframe, shuffle=True, batch_size=32):
    dataframe = dataframe.copy()

    # Use Pandas dataframe's pop method to get the list of targets.
    labels = dataframe.pop('target')

    # Create a tf.data.Dataset from the dataframe and labels.
    ds = tf.data.Dataset.from_tensor_slices((dict(dataframe), labels.values))

    if shuffle:
        # Shuffle dataset.
        ds = ds.shuffle(1024)

    # Batch dataset with specified batch_size parameter.
    ds = ds.batch(batch_size)

    return ds
```

In [5]:

```
batch_size = 5 # A small batch sized is used for demonstration purposes
train_ds = df_to_dataset(train, batch_size=batch_size)
val_ds = df_to_dataset(val, shuffle=False, batch_size=batch_size)
test_ds = df_to_dataset(test, shuffle=False, batch_size=batch_size)
```

Understand the Input Pipeline

Now that we have created the input pipeline, let's call it to see the format of the data it returns. We have used a small batch size to keep the output readable.

In [6]:

```
for feature_batch, label_batch in train_ds.take(1):
    print('Every feature:', list(feature_batch.keys()))
    print('A batch of ages:', feature_batch['age'])
    print('A batch of targets:', label_batch )
```

```
Every feature: ['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', 'thalach', 'exang', 'old
peak', 'slope', 'ca', 'thal']
A batch of ages: tf.Tensor([41 58 59 43 63], shape=(5,), dtype=int32)
A batch of targets: tf.Tensor([0 1 0 0 0], shape=(5,), dtype=int64)
```

We can see that the dataset returns a dictionary of column names (from the dataframe) that map to column values from rows in the dataframe.

Create Several Types of Feature Columns

TensorFlow provides many types of feature columns. In this section, we will create several types of feature columns, and demonstrate how they transform a column from the dataframe.

In [7]:

```
# Try to demonstrate several types of feature columns by getting an example.
example_batch = next(iter(train_ds))[0]
```

In [8]:

```
# A utility method to create a feature column and to transform a batch of data.
def demo(feature_column):
    feature_layer = layers.DenseFeatures(feature_column, dtype='float64')
    print(feature_layer(example_batch).numpy())
```

Numeric Columns

The output of a feature column becomes the input to the model (using the demo function defined above, we will be able to see exactly how each column from the dataframe is transformed). A numeric column is the simplest type of column. It is used to represent real valued features.

In [9]:

```
# EXERCISE: Create a numeric feature column out of 'age' and demo it.
age = tf.feature_column.numeric_column('age')

demo(age)
```

```
[[54.]
 [62.]
 [57.]
 [63.]
 [45.]]
```

In the heart disease dataset, most columns from the dataframe are numeric.

Bucketized Columns

Often, you don't want to feed a number directly into the model, but instead split its value into different categories based on numerical ranges. Consider raw data that represents a person's age. Instead of representing age as a numeric column, we could split the age into several buckets using a bucketized column.

In [10]:

```
# EXERCISE: Create a bucketized feature column out of 'age' with
# the following boundaries and demo it.
boundaries = [18, 25, 30, 35, 40, 45, 50, 55, 60, 65]

age_buckets = tf.feature_column.bucketized_column(
    source_column = age,
    boundaries = boundaries)

demo(age_buckets)
```

```
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]]
```

Notice the one-hot values above describe which age range each row matches.

Categorical Columns

In this dataset, thal is represented as a string (e.g. 'fixed', 'normal', or 'reversible'). We cannot feed strings directly to a model. Instead, we must first map them to numeric values. The categorical vocabulary columns provide a way to represent strings as a one-hot vector (much like you have seen above with age buckets).

Note: You will probably see some warning messages when running some of the code cell below. These warnings have to do with software updates and should not cause any errors or prevent your code from running.

In [11]:

```
# EXERCISE: Create a categorical vocabulary column out of the
# above mentioned categories with the key specified as 'thal'.
thal = tf.feature_column.categorical_column_with_vocabulary_list('thal', ['fixed', 'normal', 'rever
sible'])

# EXERCISE: Create an indicator column out of the created categorical column.
thal_one_hot = tf.feature_column.indicator_column(thal)

demo(thal_one_hot)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow_core/python/feature_column/feature_column_v2.py:4276:
IndicatorColumn._variable_shape (from tensorflow.python.feature_column.feature_column_v2) is
deprecated and will be removed in a future version.
Instructions for updating:
The old _FeatureColumn APIs are being deprecated. Please use the new FeatureColumn APIs instead.
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow_core/python/feature_column/feature_column_v2.py:4331:
VocabularyListCategoricalColumn._num_buckets (from
tensorflow.python.feature_column.feature_column_v2) is deprecated and will be removed in a future
version.
Instructions for updating:
The old _FeatureColumn APIs are being deprecated. Please use the new FeatureColumn APIs instead.
[[0. 1. 0.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 1. 0.]
 [0. 1. 0.]]
```

The vocabulary can be passed as a list using categorical_column_with_vocabulary_list, or loaded from a file using categorical_column_with_vocabulary_file.

Embedding Columns

Suppose instead of having just a few possible strings, we have thousands (or more) values per category. For a number of reasons, as the number of categories grow large, it becomes infeasible to train a neural network using one-hot encodings. We can use an embedding column to overcome this limitation. Instead of representing the data as a one-hot vector of many dimensions, an embedding column represents that data as a lower-dimensional, dense vector in which each cell can contain any number, not just 0 or 1. You can tune the size of the embedding with the `dimension` parameter.

In [12]:

```
# EXERCISE: Create an embedding column out of the categorical
# vocabulary you just created (thal). Set the size of the
# embedding to 8, by using the dimension parameter.

thal_embedding = tf.feature_column.embedding_column(thal, 8)


demo(thal_embedding)
```

```
[[ 0.15709771 -0.31702602  0.09463281 -0.17384122 -0.1088668   0.20047447
  -0.46781024 -0.12508497]
 [-0.18542708 -0.69388235  0.15702565  0.01737775  0.0183248  -0.1570436
  -0.12817046 -0.05067151]
 [-0.18542708 -0.69388235  0.15702565  0.01737775  0.0183248  -0.1570436
  -0.12817046 -0.05067151]
 [ 0.15709771 -0.31702602  0.09463281 -0.17384122 -0.1088668   0.20047447
  -0.46781024 -0.12508497]
 [ 0.15709771 -0.31702602  0.09463281 -0.17384122 -0.1088668   0.20047447
  -0.46781024 -0.12508497]]
```

Hashed Feature Columns

Another way to represent a categorical column with a large number of values is to use a categorical_column_with_hash_bucket. This feature column calculates a hash value of the input, then selects one of the `hash_bucket_size` buckets to encode a string. When using this column, you do not need to provide the vocabulary, and you can choose to make the number of hash buckets significantly smaller than the number of actual categories to save space.

In [13]:

```
# EXERCISE: Create a hashed feature column with 'thal' as the key and
# 1000 hash buckets.
thal_hashed = tf.feature_column.categorical_column_with_hash_bucket('thal', 1000)

demo(feature_column.indicator_column(thal_hashed))
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
```

```
packages/tensorflow_core/python/feature_column/feature_column_v2.py:4331:
HashedCategoricalColumn._num_buckets (from tensorflow.python.feature_column.feature_column_v2) is
deprecated and will be removed in a future version.
Instructions for updating:
The old _FeatureColumn APIs are being deprecated. Please use the new FeatureColumn APIs instead.
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

Crossed Feature Columns

Combining features into a single feature, better known as [feature crosses](), enables a model to learn separate weights for each combination of features. Here, we will create a new feature that is the cross of age and thal. Note that `crossed_column` does not build the full table of all possible combinations (which could be very large). Instead, it is backed by a `hashed_column`, so you can choose how large the table is.

In [14]:

```
# EXERCISE: Create a crossed column using the bucketized column (age_buckets),
# the categorical vocabulary column (thal) previously created, and 1000 hash buckets.
crossed_feature = tf.feature_column.crossed_column([age_buckets, thal], 1000)

demo(feature_column.indicator_column(crossed_feature))
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow_core/python/feature_column/feature_column_v2.py:4331:
CrossedColumn._num_buckets (from tensorflow.python.feature_column.feature_column_v2) is deprecated
and will be removed in a future version.
Instructions for updating:
The old _FeatureColumn APIs are being deprecated. Please use the new FeatureColumn APIs instead.
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

Choose Which Columns to Use

We have seen how to use several types of feature columns. Now we will use them to train a model. The goal of this exercise is to show you the complete code needed to work with feature columns. We have selected a few columns to train our model below arbitrarily.

If your aim is to build an accurate model, try a larger dataset of your own, and think carefully about which features are the most meaningful to include, and how they should be represented.

In [15]:

```
dataframe.dtypes
```

Out[15]:

```
age           int64
sex           int64
cp            int64
trestbps      int64
chol          int64
fbs           int64
restecg       int64
thalach       int64
exang         int64
oldpeak     float64
slope         int64
ca            int64
thal         object
target        int64
dtype: object
```

You can use the above list of column datatypes to map the appropriate feature column to every column in the dataframe.

In [16]:

```
# EXERCISE: Fill in the missing code below
feature_columns = []

# Numeric Cols.
# Create a list of numeric columns. Use the following list of columns
# that have a numeric datatype: ['age', 'trestbps', 'chol', 'thalach', 'oldpeak', 'slope', 'ca'].
numeric_columns = ['age', 'trestbps', 'chol', 'thalach', 'oldpeak', 'slope', 'ca']

for header in numeric_columns:
    # Create a numeric feature column  out of the header.
    numeric_feature_column = tf.feature_column.numeric_column(header)

    feature_columns.append(numeric_feature_column)

# Bucketized Cols.
# Create a bucketized feature column out of the age column (numeric column)
# that you've already created. Use the following boundaries:
# [18, 25, 30, 35, 40, 45, 50, 55, 60, 65]
age_buckets = tf.feature_column.bucketized_column(feature_columns[0], boundaries)

feature_columns.append(age_buckets)

# Indicator Cols.
# Create a categorical vocabulary column out of the categories
# ['fixed', 'normal', 'reversible'] with the key specified as 'thal'.
thal = tf.feature_column.categorical_column_with_vocabulary_list('thal', ['fixed', 'normal', 'rever
sible'])

# Create an indicator column out of the created thal categorical column
thal_one_hot = tf.feature_column.indicator_column(thal)

feature_columns.append(thal_one_hot)

# Embedding Cols.
# Create an embedding column out of the categorical vocabulary you
# just created (thal). Set the size of the embedding to 8, by using
# the dimension parameter.
thal_embedding = tf.feature_column.embedding_column(thal, 8)

feature_columns.append(thal_embedding)

# Crossed Cols.
# Create a crossed column using the bucketized column (age_buckets),
# the categorical vocabulary column (thal) previously created, and 1000 hash buckets.
crossed_feature = tf.feature_column.crossed_column([age_buckets, thal], 1000)

# Create an indicator column out of the crossed column created above to one-hot encode it.
crossed_feature = tf.feature_column.indicator_column(crossed_feature)

feature_columns.append(crossed_feature)
```

Create a Feature Layer

Now that we have defined our feature columns, we will use a DenseFeatures layer to input them to our Keras model.

In [17]:

```
# EXERCISE: Create a Keras DenseFeatures layer and pass the feature_columns you just created.
feature_layer = tf.keras.layers.DenseFeatures(feature_columns)
```

Earlier, we used a small batch size to demonstrate how feature columns worked. We create a new input pipeline with a larger batch size.

In [18]:

```
batch_size = 32
train_ds = df_to_dataset(train, batch_size=batch_size)
val_ds = df_to_dataset(val, shuffle=False, batch_size=batch_size)
```

```
test_ds = df_to_dataset(test, shuffle=False, batch_size=batch_size)
```

Create, Compile, and Train the Model

In [19]:

```
model = tf.keras.Sequential([
        feature_layer,
        layers.Dense(128, activation='relu'),
        layers.Dense(128, activation='relu'),
        layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(train_ds,
          validation_data=val_ds,
          epochs=100)
```

```
WARNING:tensorflow:Layer sequential is casting an input tensor from dtype float64 to the layer's
dtype of float32, which is new behavior in TensorFlow 2.  The layer has dtype float32 because it's
dtype defaults to floatx.

If you intended to run this layer in float32, you can safely ignore this warning. If in doubt, thi
s warning is likely only an issue if you are porting a TensorFlow 1.X model to TensorFlow 2.

To change all layers to have dtype float64 by default, call
`tf.keras.backend.set_floatx('float64')`. To change just this layer, pass dtype='float64' to the l
ayer constructor. If you are the author of this layer, you can disable autocasting by passing auto
cast=False to the base Layer constructor.

Epoch 1/100
7/7 [==============================] - 5s 650ms/step - loss: 1.4806 - accuracy: 0.5959 - val_loss:
0.0000e+00 - val_accuracy: 0.0000e+00
Epoch 2/100
7/7 [==============================] - 0s 48ms/step - loss: 1.3091 - accuracy: 0.6684 - val_loss:
1.1278 - val_accuracy: 0.4082
Epoch 3/100
7/7 [==============================] - 0s 51ms/step - loss: 0.6802 - accuracy: 0.7409 - val_loss:
0.6787 - val_accuracy: 0.8163
Epoch 4/100
7/7 [==============================] - 0s 56ms/step - loss: 0.5294 - accuracy: 0.7409 - val_loss:
0.5340 - val_accuracy: 0.7755
Epoch 5/100
7/7 [==============================] - 0s 62ms/step - loss: 0.8012 - accuracy: 0.7668 - val_loss:
0.5739 - val_accuracy: 0.7347
Epoch 6/100
7/7 [==============================] - 0s 59ms/step - loss: 0.5323 - accuracy: 0.7098 - val_loss:
0.6249 - val_accuracy: 0.7959
Epoch 7/100
7/7 [==============================] - 0s 59ms/step - loss: 0.5861 - accuracy: 0.7565 - val_loss:
0.6645 - val_accuracy: 0.5714
Epoch 8/100
7/7 [==============================] - 0s 63ms/step - loss: 0.4408 - accuracy: 0.7720 - val_loss:
0.4989 - val_accuracy: 0.7755
Epoch 9/100
7/7 [==============================] - 0s 53ms/step - loss: 0.4290 - accuracy: 0.7927 - val_loss:
0.5865 - val_accuracy: 0.6735
Epoch 10/100
7/7 [==============================] - 0s 57ms/step - loss: 0.4313 - accuracy: 0.7927 - val_loss:
0.5739 - val_accuracy: 0.7959
Epoch 11/100
7/7 [==============================] - 0s 53ms/step - loss: 0.5054 - accuracy: 0.7668 - val_loss:
0.4917 - val_accuracy: 0.7755
Epoch 12/100
7/7 [==============================] - 0s 62ms/step - loss: 0.3925 - accuracy: 0.7824 - val_loss:
0.4975 - val_accuracy: 0.7755
Epoch 13/100
7/7 [==============================] - 0s 60ms/step - loss: 0.4332 - accuracy: 0.7876 - val_loss:
0.7500 - val_accuracy: 0.7959
Epoch 14/100
7/7 [==============================] - 0s 61ms/step - loss: 0.7790 - accuracy: 0.7358 - val loss:
```

```
1.7389 - val_accuracy: 0.3265
Epoch 15/100
7/7 [==============================] - 0s 59ms/step - loss: 0.8513 - accuracy: 0.6269 - val_loss:
0.9605 - val_accuracy: 0.7959
Epoch 16/100
7/7 [==============================] - 0s 61ms/step - loss: 0.6638 - accuracy: 0.7617 - val_loss:
0.6660 - val_accuracy: 0.6735
Epoch 17/100
7/7 [==============================] - 0s 56ms/step - loss: 0.5584 - accuracy: 0.7876 - val_loss:
0.5007 - val_accuracy: 0.7551
Epoch 18/100
7/7 [==============================] - 0s 65ms/step - loss: 0.6262 - accuracy: 0.7461 - val_loss:
0.6250 - val_accuracy: 0.8163
Epoch 19/100
7/7 [==============================] - 0s 64ms/step - loss: 0.5371 - accuracy: 0.8031 - val_loss:
0.5695 - val_accuracy: 0.7347
Epoch 20/100
7/7 [==============================] - 0s 67ms/step - loss: 0.4306 - accuracy: 0.8031 - val_loss:
0.5193 - val_accuracy: 0.7755
Epoch 21/100
7/7 [==============================] - 0s 41ms/step - loss: 0.4156 - accuracy: 0.8031 - val_loss:
0.5105 - val_accuracy: 0.7755
Epoch 22/100
7/7 [==============================] - 0s 42ms/step - loss: 0.3920 - accuracy: 0.8135 - val_loss:
0.4987 - val_accuracy: 0.7755
Epoch 23/100
7/7 [==============================] - 0s 65ms/step - loss: 0.3661 - accuracy: 0.8187 - val_loss:
0.5002 - val_accuracy: 0.7755
Epoch 24/100
7/7 [==============================] - 0s 55ms/step - loss: 0.4021 - accuracy: 0.7876 - val_loss:
0.4974 - val_accuracy: 0.7959
Epoch 25/100
7/7 [==============================] - 0s 61ms/step - loss: 0.5009 - accuracy: 0.7720 - val_loss:
0.6091 - val_accuracy: 0.7143
Epoch 26/100
7/7 [==============================] - 0s 54ms/step - loss: 0.4163 - accuracy: 0.8083 - val_loss:
0.6035 - val_accuracy: 0.8163
Epoch 27/100
7/7 [==============================] - 0s 61ms/step - loss: 0.4778 - accuracy: 0.7720 - val_loss:
0.7771 - val_accuracy: 0.5918
Epoch 28/100
7/7 [==============================] - 0s 64ms/step - loss: 0.4358 - accuracy: 0.8083 - val_loss:
0.5428 - val_accuracy: 0.7143
Epoch 29/100
7/7 [==============================] - 0s 57ms/step - loss: 0.7016 - accuracy: 0.7150 - val_loss:
0.7792 - val_accuracy: 0.7959
Epoch 30/100
7/7 [==============================] - 0s 57ms/step - loss: 0.8856 - accuracy: 0.7461 - val_loss:
0.5753 - val_accuracy: 0.7347
Epoch 31/100
7/7 [==============================] - 0s 58ms/step - loss: 0.4491 - accuracy: 0.7927 - val_loss:
0.5275 - val_accuracy: 0.7551
Epoch 32/100
7/7 [==============================] - 0s 55ms/step - loss: 0.3746 - accuracy: 0.8238 - val_loss:
0.7307 - val_accuracy: 0.6327
Epoch 33/100
7/7 [==============================] - 0s 61ms/step - loss: 0.5831 - accuracy: 0.7409 - val_loss:
0.4796 - val_accuracy: 0.7959
Epoch 34/100
7/7 [==============================] - 0s 52ms/step - loss: 0.3421 - accuracy: 0.8394 - val_loss:
0.5451 - val_accuracy: 0.7347
Epoch 35/100
7/7 [==============================] - 0s 62ms/step - loss: 0.3619 - accuracy: 0.8083 - val_loss:
0.5084 - val_accuracy: 0.7755
Epoch 36/100
7/7 [==============================] - 0s 54ms/step - loss: 0.3426 - accuracy: 0.8238 - val_loss:
0.5018 - val_accuracy: 0.7755
Epoch 37/100
7/7 [==============================] - 0s 61ms/step - loss: 0.3367 - accuracy: 0.8290 - val_loss:
0.5451 - val_accuracy: 0.7143
Epoch 38/100
7/7 [==============================] - 0s 49ms/step - loss: 0.3523 - accuracy: 0.8290 - val_loss:
0.4843 - val_accuracy: 0.7959
Epoch 39/100
7/7 [==============================] - 0s 67ms/step - loss: 0.3583 - accuracy: 0.8031 - val_loss:
0.5013 - val_accuracy: 0.7347
Epoch 40/100
```

```
7/7 [==============================] - 0s 61ms/step - loss: 0.3317 - accuracy: 0.8342 - val_loss:
0.4902 - val_accuracy: 0.7347
Epoch 41/100
7/7 [==============================] - 0s 63ms/step - loss: 0.3356 - accuracy: 0.8497 - val_loss:
0.6064 - val_accuracy: 0.8163
Epoch 42/100
7/7 [==============================] - 0s 35ms/step - loss: 0.6860 - accuracy: 0.7461 - val_loss:
0.4905 - val_accuracy: 0.7347
Epoch 43/100
7/7 [==============================] - 0s 49ms/step - loss: 0.4008 - accuracy: 0.8342 - val_loss:
0.4950 - val_accuracy: 0.7959
Epoch 44/100
7/7 [==============================] - 0s 59ms/step - loss: 0.3964 - accuracy: 0.7979 - val_loss:
0.5144 - val_accuracy: 0.7347
Epoch 45/100
7/7 [==============================] - 0s 56ms/step - loss: 0.3693 - accuracy: 0.8342 - val_loss:
0.4916 - val_accuracy: 0.7959
Epoch 46/100
7/7 [==============================] - 0s 50ms/step - loss: 0.3289 - accuracy: 0.8394 - val_loss:
0.4923 - val_accuracy: 0.7347
Epoch 47/100
7/7 [==============================] - 0s 58ms/step - loss: 0.3243 - accuracy: 0.8394 - val_loss:
0.4953 - val_accuracy: 0.7347
Epoch 48/100
7/7 [==============================] - 0s 60ms/step - loss: 0.3159 - accuracy: 0.8497 - val_loss:
0.4894 - val_accuracy: 0.7347
Epoch 49/100
7/7 [==============================] - 0s 60ms/step - loss: 0.3160 - accuracy: 0.8549 - val_loss:
0.4925 - val_accuracy: 0.7551
Epoch 50/100
7/7 [==============================] - 0s 63ms/step - loss: 0.3173 - accuracy: 0.8394 - val_loss:
0.5021 - val_accuracy: 0.7551
Epoch 51/100
7/7 [==============================] - 0s 61ms/step - loss: 0.3141 - accuracy: 0.8446 - val_loss:
0.4954 - val_accuracy: 0.7551
Epoch 52/100
7/7 [==============================] - 0s 59ms/step - loss: 0.3106 - accuracy: 0.8290 - val_loss:
0.4845 - val_accuracy: 0.7347
Epoch 53/100
7/7 [==============================] - 0s 61ms/step - loss: 0.3103 - accuracy: 0.8549 - val_loss:
0.4777 - val_accuracy: 0.7551
Epoch 54/100
7/7 [==============================] - 0s 52ms/step - loss: 0.3242 - accuracy: 0.8497 - val_loss:
0.4627 - val_accuracy: 0.7551
Epoch 55/100
7/7 [==============================] - 0s 53ms/step - loss: 0.3237 - accuracy: 0.8342 - val_loss:
0.4723 - val_accuracy: 0.7959
Epoch 56/100
7/7 [==============================] - 0s 52ms/step - loss: 0.3130 - accuracy: 0.8290 - val_loss:
0.5342 - val_accuracy: 0.7143
Epoch 57/100
7/7 [==============================] - 0s 55ms/step - loss: 0.3209 - accuracy: 0.8549 - val_loss:
0.5221 - val_accuracy: 0.7959
Epoch 58/100
7/7 [==============================] - 0s 56ms/step - loss: 0.3224 - accuracy: 0.8238 - val_loss:
0.5319 - val_accuracy: 0.7347
Epoch 59/100
7/7 [==============================] - 0s 50ms/step - loss: 0.3100 - accuracy: 0.8446 - val_loss:
0.5048 - val_accuracy: 0.7551
Epoch 60/100
7/7 [==============================] - 0s 55ms/step - loss: 0.3236 - accuracy: 0.8446 - val_loss:
0.5132 - val_accuracy: 0.7551
Epoch 61/100
7/7 [==============================] - 0s 51ms/step - loss: 0.3173 - accuracy: 0.8342 - val_loss:
0.5244 - val_accuracy: 0.7755
Epoch 62/100
7/7 [==============================] - 0s 53ms/step - loss: 0.3500 - accuracy: 0.8290 - val_loss:
0.6028 - val_accuracy: 0.7143
Epoch 63/100
7/7 [==============================] - 0s 47ms/step - loss: 0.4683 - accuracy: 0.8238 - val_loss:
0.5869 - val_accuracy: 0.7755
Epoch 64/100
7/7 [==============================] - 0s 42ms/step - loss: 0.3397 - accuracy: 0.8290 - val_loss:
0.6688 - val_accuracy: 0.7143
Epoch 65/100
7/7 [==============================] - 1s 73ms/step - loss: 0.3699 - accuracy: 0.8497 - val_loss:
0.5786 - val_accuracy: 0.7959
```

```
Epoch 66/100
7/7 [==============================] - 0s 58ms/step - loss: 0.3137 - accuracy: 0.8342 - val_loss:
0.5544 - val_accuracy: 0.7551
Epoch 67/100
7/7 [==============================] - 0s 61ms/step - loss: 0.3070 - accuracy: 0.8446 - val_loss:
0.5708 - val_accuracy: 0.7755
Epoch 68/100
7/7 [==============================] - 0s 55ms/step - loss: 0.3042 - accuracy: 0.8290 - val_loss:
0.9608 - val_accuracy: 0.6122
Epoch 69/100
7/7 [==============================] - 0s 59ms/step - loss: 0.7181 - accuracy: 0.6632 - val_loss:
0.5154 - val_accuracy: 0.7959
Epoch 70/100
7/7 [==============================] - 0s 60ms/step - loss: 0.4485 - accuracy: 0.8083 - val_loss:
0.6433 - val_accuracy: 0.7143
Epoch 71/100
7/7 [==============================] - 0s 63ms/step - loss: 0.4334 - accuracy: 0.7927 - val_loss:
0.5612 - val_accuracy: 0.8163
Epoch 72/100
7/7 [==============================] - 0s 62ms/step - loss: 0.3939 - accuracy: 0.8187 - val_loss:
0.5269 - val_accuracy: 0.7143
Epoch 73/100
7/7 [==============================] - 0s 60ms/step - loss: 0.3372 - accuracy: 0.8031 - val_loss:
0.4861 - val_accuracy: 0.8163
Epoch 74/100
7/7 [==============================] - 0s 63ms/step - loss: 0.3348 - accuracy: 0.8549 - val_loss:
0.4904 - val_accuracy: 0.7551
Epoch 75/100
7/7 [==============================] - 0s 62ms/step - loss: 0.3512 - accuracy: 0.8238 - val_loss:
0.5431 - val_accuracy: 0.7347
Epoch 76/100
7/7 [==============================] - 0s 53ms/step - loss: 0.3422 - accuracy: 0.8601 - val_loss:
0.5299 - val_accuracy: 0.7755
Epoch 77/100
7/7 [==============================] - 0s 50ms/step - loss: 0.3600 - accuracy: 0.8394 - val_loss:
0.5746 - val_accuracy: 0.7347
Epoch 78/100
7/7 [==============================] - 0s 53ms/step - loss: 0.3659 - accuracy: 0.8601 - val_loss:
0.5528 - val_accuracy: 0.7959
Epoch 79/100
7/7 [==============================] - 0s 65ms/step - loss: 0.3314 - accuracy: 0.8290 - val_loss:
0.5539 - val_accuracy: 0.7143
Epoch 80/100
7/7 [==============================] - 0s 61ms/step - loss: 0.3024 - accuracy: 0.8601 - val_loss:
0.6396 - val_accuracy: 0.7959
Epoch 81/100
7/7 [==============================] - 0s 57ms/step - loss: 0.6219 - accuracy: 0.7513 - val_loss:
0.4953 - val_accuracy: 0.7143
Epoch 82/100
7/7 [==============================] - 0s 64ms/step - loss: 0.4008 - accuracy: 0.8135 - val_loss:
0.5106 - val_accuracy: 0.7959
Epoch 83/100
7/7 [==============================] - 0s 55ms/step - loss: 0.3840 - accuracy: 0.7927 - val_loss:
0.4832 - val_accuracy: 0.7347
Epoch 84/100
7/7 [==============================] - 0s 53ms/step - loss: 0.3519 - accuracy: 0.8446 - val_loss:
0.4922 - val_accuracy: 0.7347
Epoch 85/100
7/7 [==============================] - 0s 34ms/step - loss: 0.3345 - accuracy: 0.8549 - val_loss:
0.6140 - val_accuracy: 0.7959
Epoch 86/100
7/7 [==============================] - 0s 59ms/step - loss: 0.8600 - accuracy: 0.7461 - val_loss:
0.5551 - val_accuracy: 0.8163
Epoch 87/100
7/7 [==============================] - 0s 56ms/step - loss: 0.3561 - accuracy: 0.8187 - val_loss:
0.4684 - val_accuracy: 0.7755
Epoch 88/100
7/7 [==============================] - 0s 51ms/step - loss: 0.6659 - accuracy: 0.7617 - val_loss:
0.6284 - val_accuracy: 0.7959
Epoch 89/100
7/7 [==============================] - 0s 54ms/step - loss: 0.3923 - accuracy: 0.8187 - val_loss:
0.6648 - val_accuracy: 0.7143
Epoch 90/100
7/7 [==============================] - 0s 60ms/step - loss: 0.3721 - accuracy: 0.7927 - val_loss:
0.4683 - val_accuracy: 0.7755
Epoch 91/100
7/7 [==============================] - 0s 57ms/step - loss: 0.4315 - accuracy: 0.8187 - val_loss:
```

```
0.4793 - val_accuracy: 0.7755
Epoch 92/100
7/7 [==============================] - 0s 61ms/step - loss: 0.4476 - accuracy: 0.7772 - val_loss:
0.5359 - val_accuracy: 0.7551
Epoch 93/100
7/7 [==============================] - 0s 63ms/step - loss: 0.3291 - accuracy: 0.8187 - val_loss:
0.6177 - val_accuracy: 0.6939
Epoch 94/100
7/7 [==============================] - 0s 58ms/step - loss: 0.3116 - accuracy: 0.8549 - val_loss:
0.5311 - val_accuracy: 0.7551
Epoch 95/100
7/7 [==============================] - 0s 65ms/step - loss: 0.3378 - accuracy: 0.8446 - val_loss:
0.5197 - val_accuracy: 0.7347
Epoch 96/100
7/7 [==============================] - 0s 60ms/step - loss: 0.3084 - accuracy: 0.8549 - val_loss:
0.5014 - val_accuracy: 0.7347
Epoch 97/100
7/7 [==============================] - 0s 58ms/step - loss: 0.3570 - accuracy: 0.8135 - val_loss:
0.5144 - val_accuracy: 0.7551
Epoch 98/100
7/7 [==============================] - 0s 67ms/step - loss: 0.2983 - accuracy: 0.8342 - val_loss:
0.5536 - val_accuracy: 0.7143
Epoch 99/100
7/7 [==============================] - 0s 62ms/step - loss: 0.2985 - accuracy: 0.8549 - val_loss:
0.5148 - val_accuracy: 0.7347
Epoch 100/100
7/7 [==============================] - 0s 59ms/step - loss: 0.2999 - accuracy: 0.8446 - val_loss:
0.5053 - val_accuracy: 0.7755
```

Out[19]:

```
<tensorflow.python.keras.callbacks.History at 0x7fd951923eb8>
```

In [20]:

```
loss, accuracy = model.evaluate(test_ds)
print("Accuracy", accuracy)
```

```
2/2 [==============================] - 0s 42ms/step - loss: 0.5113 - accuracy: 0.7377
Accuracy 0.73770493
```

Submission Instructions

In [ ]:

```
# Now click the 'Submit Assignment' button above.
```

When you're done or would like to take a break, please run the two cells below to save your work and close the Notebook. This frees up resources for your fellow learners.

In [ ]:

```
%%javascript
<!-- Save the notebook -->
IPython.notebook.save_checkpoint();
```

In [ ]:

```
%%javascript
<!-- Shutdown and close the notebook -->
window.onbeforeunload = null
window.close();
IPython.notebook.session.delete();
```