Classify Structured Data

Import TensorFlow and Other Libraries

```
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.

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

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

   ca        thal  target
0   0       fixed       0
1   3      normal       1
2   2  reversible       0
3   0      normal       0
4   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.

```
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.

```
# 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(buffer_size = len(dataframe))

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

    return ds

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.

```
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', 'oldpeak', 'slope', 'ca', 'thal']
A batch of ages: tf.Tensor([64 61 54 62 74], shape=(5,), dtype=int64)
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.

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

# 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.

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

demo(age)

[[55.]
 [67.]
 [51.]
 [40.]
 [61.]]
```

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.

```
# 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 = feature_column.bucketized_column(age,
boundaries=boundaries)

demo(age_buckets)
```

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

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

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

demo(thal_one_hot)
```

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

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

```
# 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 = feature_column.embedding_column(thal, dimension=8)


demo(thal_embedding)
```

```
[[-0.3816596  -0.1579989   0.5514784  -0.14075178  0.42594242 -
0.1745316
   0.39247578 -0.41802773]
 [-0.3816596  -0.1579989   0.5514784  -0.14075178  0.42594242 -
0.1745316
   0.39247578 -0.41802773]
 [-0.15146169 -0.04329393  0.09052955  0.08309027 -0.04808125
0.16416822
   0.01192441  0.36713275]
 [-0.3816596  -0.1579989   0.5514784  -0.14075178  0.42594242 -
0.1745316
   0.39247578 -0.41802773]
 [-0.15146169 -0.04329393  0.09052955  0.08309027 -0.04808125
0.16416822
   0.01192441  0.36713275]]
```

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.

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

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

```
[[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.

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

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

```
[[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.

```
dataframe.dtypes
```

```
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.

```
# 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 = 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]
boundaries = [18, 25, 30, 35, 40, 45, 50, 55, 60, 65]
age_buckets = feature_column.bucketized_column(age,
boundaries=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 = feature_column.categorical_column_with_vocabulary_list('thal',
['fixed', 'normal', 'reversible'])

# Create an indicator column out of the created thal categorical
column
thal_one_hot = 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 = feature_column.embedding_column(thal, dimension=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 = feature_column.crossed_column([age_buckets, thal],
hash_bucket_size=1000)

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

```
# 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.

```
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

```
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)
```

```
Epoch 1/100
WARNING:tensorflow:Layers in a Sequential model should only have a
single input tensor. Received: inputs={'age': <tf.Tensor
'IteratorGetNext:0' shape=(None,) dtype=int64>, 'sex': <tf.Tensor
'IteratorGetNext:8' shape=(None,) dtype=int64>, 'cp': <tf.Tensor
'IteratorGetNext:3' shape=(None,) dtype=int64>, 'trestbps': <tf.Tensor
'IteratorGetNext:12' shape=(None,) dtype=int64>, 'chol': <tf.Tensor
'IteratorGetNext:2' shape=(None,) dtype=int64>, 'fbs': <tf.Tensor
'IteratorGetNext:5' shape=(None,) dtype=int64>, 'restecg': <tf.Tensor
'IteratorGetNext:7' shape=(None,) dtype=int64>, 'thalach': <tf.Tensor
'IteratorGetNext:11' shape=(None,) dtype=int64>, 'exang': <tf.Tensor
'IteratorGetNext:4' shape=(None,) dtype=int64>, 'oldpeak': <tf.Tensor
'IteratorGetNext:6' shape=(None,) dtype=float64>, 'slope': <tf.Tensor
'IteratorGetNext:9' shape=(None,) dtype=int64>, 'ca': <tf.Tensor
'IteratorGetNext:1' shape=(None,) dtype=int64>, 'thal': <tf.Tensor
'IteratorGetNext:10' shape=(None,) dtype=string>}. Consider rewriting
this model with the Functional API.
WARNING:tensorflow:Layers in a Sequential model should only have a
single input tensor. Received: inputs={'age': <tf.Tensor
'IteratorGetNext:0' shape=(None,) dtype=int64>, 'sex': <tf.Tensor
'IteratorGetNext:8' shape=(None,) dtype=int64>, 'cp': <tf.Tensor
'IteratorGetNext:3' shape=(None,) dtype=int64>, 'trestbps': <tf.Tensor
'IteratorGetNext:12' shape=(None,) dtype=int64>, 'chol': <tf.Tensor
'IteratorGetNext:2' shape=(None,) dtype=int64>, 'fbs': <tf.Tensor
'IteratorGetNext:5' shape=(None,) dtype=int64>, 'restecg': <tf.Tensor
'IteratorGetNext:7' shape=(None,) dtype=int64>, 'thalach': <tf.Tensor
'IteratorGetNext:11' shape=(None,) dtype=int64>, 'exang': <tf.Tensor
'IteratorGetNext:4' shape=(None,) dtype=int64>, 'oldpeak': <tf.Tensor
'IteratorGetNext:6' shape=(None,) dtype=float64>, 'slope': <tf.Tensor
'IteratorGetNext:9' shape=(None,) dtype=int64>, 'ca': <tf.Tensor
'IteratorGetNext:1' shape=(None,) dtype=int64>, 'thal': <tf.Tensor
'IteratorGetNext:10' shape=(None,) dtype=string>}. Consider rewriting
this model with the Functional API.
```

```
7/7 [==============================] - ETA: 0s - loss: 3.6434 -
accuracy: 0.5751 WARNING:tensorflow:Layers in a Sequential model
should only have a single input tensor. Received: inputs={'age':
<tf.Tensor 'IteratorGetNext:0' shape=(None,) dtype=int64>, 'sex':
<tf.Tensor 'IteratorGetNext:8' shape=(None,) dtype=int64>, 'cp':
<tf.Tensor 'IteratorGetNext:3' shape=(None,) dtype=int64>, 'trestbps':
<tf.Tensor 'IteratorGetNext:12' shape=(None,) dtype=int64>, 'chol':
<tf.Tensor 'IteratorGetNext:2' shape=(None,) dtype=int64>, 'fbs':
<tf.Tensor 'IteratorGetNext:5' shape=(None,) dtype=int64>, 'restecg':
<tf.Tensor 'IteratorGetNext:7' shape=(None,) dtype=int64>, 'thalach':
<tf.Tensor 'IteratorGetNext:11' shape=(None,) dtype=int64>, 'exang':
<tf.Tensor 'IteratorGetNext:4' shape=(None,) dtype=int64>, 'oldpeak':
<tf.Tensor 'IteratorGetNext:6' shape=(None,) dtype=float64>, 'slope':
<tf.Tensor 'IteratorGetNext:9' shape=(None,) dtype=int64>, 'ca':
<tf.Tensor 'IteratorGetNext:1' shape=(None,) dtype=int64>, 'thal':
<tf.Tensor 'IteratorGetNext:10' shape=(None,) dtype=string>}. Consider
rewriting this model with the Functional API.
7/7 [==============================] - 3s 76ms/step - loss: 3.6434 -
accuracy: 0.5751 - val_loss: 1.5336 - val_accuracy: 0.3061
Epoch 2/100
7/7 [==============================] - 0s 7ms/step - loss: 1.3516 -
accuracy: 0.6062 - val_loss: 0.8075 - val_accuracy: 0.6939
Epoch 3/100
7/7 [==============================] - 0s 7ms/step - loss: 0.7178 -
accuracy: 0.5907 - val_loss: 0.6794 - val_accuracy: 0.6939
Epoch 4/100
7/7 [==============================] - 0s 11ms/step - loss: 0.6716 -
accuracy: 0.7047 - val_loss: 0.8363 - val_accuracy: 0.5102
Epoch 5/100
7/7 [==============================] - 0s 10ms/step - loss: 0.7414 -
accuracy: 0.6321 - val_loss: 0.7103 - val_accuracy: 0.7143
Epoch 6/100
7/7 [==============================] - 0s 7ms/step - loss: 0.6801 -
accuracy: 0.6528 - val_loss: 0.6816 - val_accuracy: 0.7551
Epoch 7/100
7/7 [==============================] - 0s 7ms/step - loss: 0.6126 -
accuracy: 0.7254 - val_loss: 0.6044 - val_accuracy: 0.7143
Epoch 8/100
7/7 [==============================] - 0s 10ms/step - loss: 0.5222 -
accuracy: 0.7254 - val_loss: 0.6046 - val_accuracy: 0.7551
Epoch 9/100
7/7 [==============================] - 0s 11ms/step - loss: 0.5007 -
accuracy: 0.7513 - val_loss: 0.5708 - val_accuracy: 0.6939
Epoch 10/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4995 -
accuracy: 0.7461 - val_loss: 0.5375 - val_accuracy: 0.7755
Epoch 11/100
7/7 [==============================] - 0s 11ms/step - loss: 0.4846 -
accuracy: 0.7565 - val_loss: 0.5769 - val_accuracy: 0.7347
```

```
Epoch 12/100
7/7 [==============================] - 0s 10ms/step - loss: 0.4868 -
accuracy: 0.7668 - val_loss: 0.6905 - val_accuracy: 0.6122
Epoch 13/100
7/7 [==============================] - 0s 7ms/step - loss: 0.5660 -
accuracy: 0.7150 - val_loss: 0.4996 - val_accuracy: 0.7959
Epoch 14/100
7/7 [==============================] - 0s 10ms/step - loss: 0.5686 -
accuracy: 0.6891 - val_loss: 0.9021 - val_accuracy: 0.6939
Epoch 15/100
7/7 [==============================] - 0s 10ms/step - loss: 1.0099 -
accuracy: 0.7047 - val_loss: 0.6451 - val_accuracy: 0.6327
Epoch 16/100
7/7 [==============================] - 0s 8ms/step - loss: 0.5933 -
accuracy: 0.7047 - val_loss: 0.7670 - val_accuracy: 0.6939
Epoch 17/100
7/7 [==============================] - 0s 8ms/step - loss: 0.5370 -
accuracy: 0.7461 - val_loss: 0.8153 - val_accuracy: 0.5510
Epoch 18/100
7/7 [==============================] - 0s 11ms/step - loss: 0.6093 -
accuracy: 0.6580 - val_loss: 0.9758 - val_accuracy: 0.6939
Epoch 19/100
7/7 [==============================] - 0s 10ms/step - loss: 0.6643 -
accuracy: 0.7202 - val_loss: 0.8304 - val_accuracy: 0.6122
Epoch 20/100
7/7 [==============================] - 0s 8ms/step - loss: 0.7636 -
accuracy: 0.6218 - val_loss: 0.7815 - val_accuracy: 0.6939
Epoch 21/100
7/7 [==============================] - 0s 11ms/step - loss: 0.5065 -
accuracy: 0.7720 - val_loss: 0.5380 - val_accuracy: 0.7143
Epoch 22/100
7/7 [==============================] - 0s 10ms/step - loss: 0.4604 -
accuracy: 0.7668 - val_loss: 0.5791 - val_accuracy: 0.7551
Epoch 23/100
7/7 [==============================] - 0s 8ms/step - loss: 0.4147 -
accuracy: 0.7824 - val_loss: 0.5212 - val_accuracy: 0.6939
Epoch 24/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4329 -
accuracy: 0.7927 - val_loss: 0.6042 - val_accuracy: 0.7143
Epoch 25/100
7/7 [==============================] - 0s 11ms/step - loss: 0.4878 -
accuracy: 0.7772 - val_loss: 0.5005 - val_accuracy: 0.7347
Epoch 26/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3929 -
accuracy: 0.8083 - val_loss: 0.5763 - val_accuracy: 0.7347
Epoch 27/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4558 -
accuracy: 0.7824 - val_loss: 0.4775 - val_accuracy: 0.7755
Epoch 28/100
```

```
7/7 [==============================] - 0s 11ms/step - loss: 0.3781 -
accuracy: 0.8238 - val_loss: 0.6059 - val_accuracy: 0.7347
Epoch 29/100
7/7 [==============================] - 0s 11ms/step - loss: 0.4514 -
accuracy: 0.7513 - val_loss: 0.5020 - val_accuracy: 0.7347
Epoch 30/100
7/7 [==============================] - 0s 8ms/step - loss: 0.3803 -
accuracy: 0.8135 - val_loss: 0.6394 - val_accuracy: 0.7347
Epoch 31/100
7/7 [==============================] - 0s 8ms/step - loss: 0.4291 -
accuracy: 0.7565 - val_loss: 0.5786 - val_accuracy: 0.6939
Epoch 32/100
7/7 [==============================] - 0s 10ms/step - loss: 0.4371 -
accuracy: 0.7772 - val_loss: 0.5348 - val_accuracy: 0.7755
Epoch 33/100
7/7 [==============================] - 0s 10ms/step - loss: 0.4152 -
accuracy: 0.8135 - val_loss: 0.4792 - val_accuracy: 0.7959
Epoch 34/100
7/7 [==============================] - 0s 8ms/step - loss: 0.3810 -
accuracy: 0.8238 - val_loss: 0.5313 - val_accuracy: 0.7755
Epoch 35/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3940 -
accuracy: 0.7927 - val_loss: 0.6443 - val_accuracy: 0.7143
Epoch 36/100
7/7 [==============================] - 0s 11ms/step - loss: 0.5405 -
accuracy: 0.7461 - val_loss: 0.6884 - val_accuracy: 0.6735
Epoch 37/100
7/7 [==============================] - 0s 8ms/step - loss: 0.4769 -
accuracy: 0.7824 - val_loss: 0.4842 - val_accuracy: 0.7551
Epoch 38/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4915 -
accuracy: 0.7409 - val_loss: 0.4601 - val_accuracy: 0.7755
Epoch 39/100
7/7 [==============================] - 0s 10ms/step - loss: 0.4018 -
accuracy: 0.8031 - val_loss: 0.5887 - val_accuracy: 0.7347
Epoch 40/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4932 -
accuracy: 0.7409 - val_loss: 0.4821 - val_accuracy: 0.7755
Epoch 41/100
7/7 [==============================] - 0s 7ms/step - loss: 0.5071 -
accuracy: 0.7358 - val_loss: 0.5286 - val_accuracy: 0.7551
Epoch 42/100
7/7 [==============================] - 0s 10ms/step - loss: 0.7699 -
accuracy: 0.6166 - val_loss: 0.8345 - val_accuracy: 0.6939
Epoch 43/100
7/7 [==============================] - 0s 7ms/step - loss: 0.7239 -
accuracy: 0.7306 - val_loss: 0.4666 - val_accuracy: 0.7551
Epoch 44/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4549 -
```

```
accuracy: 0.7927 - val_loss: 0.4636 - val_accuracy: 0.7755
Epoch 45/100
7/7 [==============================] - 0s 10ms/step - loss: 0.4091 -
accuracy: 0.8290 - val_loss: 0.4262 - val_accuracy: 0.7959
Epoch 46/100
7/7 [==============================] - 0s 11ms/step - loss: 0.3917 -
accuracy: 0.8238 - val_loss: 0.4127 - val_accuracy: 0.8163
Epoch 47/100
7/7 [==============================] - 0s 7ms/step - loss: 0.3866 -
accuracy: 0.7927 - val_loss: 0.4256 - val_accuracy: 0.7755
Epoch 48/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3660 -
accuracy: 0.8290 - val_loss: 0.6118 - val_accuracy: 0.7143
Epoch 49/100
7/7 [==============================] - 0s 10ms/step - loss: 0.6628 -
accuracy: 0.7150 - val_loss: 0.4264 - val_accuracy: 0.7959
Epoch 50/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4139 -
accuracy: 0.8031 - val_loss: 0.4182 - val_accuracy: 0.7755
Epoch 51/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3723 -
accuracy: 0.8187 - val_loss: 0.4337 - val_accuracy: 0.7959
Epoch 52/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3704 -
accuracy: 0.8135 - val_loss: 0.4382 - val_accuracy: 0.7959
Epoch 53/100
7/7 [==============================] - 0s 7ms/step - loss: 0.3757 -
accuracy: 0.8083 - val_loss: 0.4423 - val_accuracy: 0.8163
Epoch 54/100
7/7 [==============================] - 0s 11ms/step - loss: 0.4086 -
accuracy: 0.8238 - val_loss: 0.4891 - val_accuracy: 0.7143
Epoch 55/100
7/7 [==============================] - 0s 9ms/step - loss: 0.4529 -
accuracy: 0.7513 - val_loss: 0.4162 - val_accuracy: 0.7755
Epoch 56/100
7/7 [==============================] - 0s 7ms/step - loss: 0.3618 -
accuracy: 0.8238 - val_loss: 0.4692 - val_accuracy: 0.7143
Epoch 57/100
7/7 [==============================] - 0s 8ms/step - loss: 0.4688 -
accuracy: 0.7720 - val_loss: 0.4157 - val_accuracy: 0.7347
Epoch 58/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3702 -
accuracy: 0.8238 - val_loss: 0.4725 - val_accuracy: 0.7755
Epoch 59/100
7/7 [==============================] - 0s 8ms/step - loss: 0.5243 -
accuracy: 0.7668 - val_loss: 0.6391 - val_accuracy: 0.7347
Epoch 60/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4749 -
accuracy: 0.7617 - val_loss: 0.5452 - val_accuracy: 0.7347
```

```
Epoch 61/100
7/7 [==============================] - 0s 10ms/step - loss: 0.5438 -
accuracy: 0.7461 - val_loss: 0.4111 - val_accuracy: 0.7347
Epoch 62/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4963 -
accuracy: 0.7927 - val_loss: 0.4209 - val_accuracy: 0.7755
Epoch 63/100
7/7 [==============================] - 0s 8ms/step - loss: 0.4195 -
accuracy: 0.7927 - val_loss: 0.4096 - val_accuracy: 0.7755
Epoch 64/100
7/7 [==============================] - 0s 11ms/step - loss: 0.3899 -
accuracy: 0.8187 - val_loss: 0.4053 - val_accuracy: 0.7755
Epoch 65/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3727 -
accuracy: 0.8342 - val_loss: 0.4057 - val_accuracy: 0.7959
Epoch 66/100
7/7 [==============================] - 0s 7ms/step - loss: 0.3683 -
accuracy: 0.8290 - val_loss: 0.4173 - val_accuracy: 0.7959
Epoch 67/100
7/7 [==============================] - 0s 8ms/step - loss: 0.3797 -
accuracy: 0.8342 - val_loss: 0.5569 - val_accuracy: 0.7755
Epoch 68/100
7/7 [==============================] - 0s 10ms/step - loss: 0.6565 -
accuracy: 0.7098 - val_loss: 0.4489 - val_accuracy: 0.7959
Epoch 69/100
7/7 [==============================] - 0s 11ms/step - loss: 0.3711 -
accuracy: 0.8135 - val_loss: 0.3979 - val_accuracy: 0.7551
Epoch 70/100
7/7 [==============================] - 0s 7ms/step - loss: 0.3803 -
accuracy: 0.8238 - val_loss: 0.4578 - val_accuracy: 0.7551
Epoch 71/100
7/7 [==============================] - 0s 11ms/step - loss: 0.4690 -
accuracy: 0.7824 - val_loss: 0.4526 - val_accuracy: 0.7959
Epoch 72/100
7/7 [==============================] - 0s 15ms/step - loss: 0.4730 -
accuracy: 0.7565 - val_loss: 0.4807 - val_accuracy: 0.7143
Epoch 73/100
7/7 [==============================] - 0s 8ms/step - loss: 0.4346 -
accuracy: 0.7927 - val_loss: 0.4164 - val_accuracy: 0.7551
Epoch 74/100
7/7 [==============================] - 0s 8ms/step - loss: 0.3710 -
accuracy: 0.8238 - val_loss: 0.4654 - val_accuracy: 0.7551
Epoch 75/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3839 -
accuracy: 0.7979 - val_loss: 0.4307 - val_accuracy: 0.7959
Epoch 76/100
7/7 [==============================] - 0s 7ms/step - loss: 0.3531 -
accuracy: 0.8342 - val_loss: 0.4124 - val_accuracy: 0.7959
Epoch 77/100
```

```
7/7 [==============================] - 0s 8ms/step - loss: 0.3961 -
accuracy: 0.8342 - val_loss: 0.4071 - val_accuracy: 0.8163
Epoch 78/100
7/7 [==============================] - 0s 11ms/step - loss: 0.3724 -
accuracy: 0.8135 - val_loss: 0.4022 - val_accuracy: 0.8163
Epoch 79/100
7/7 [==============================] - 0s 11ms/step - loss: 0.3816 -
accuracy: 0.8135 - val_loss: 0.4050 - val_accuracy: 0.7959
Epoch 80/100
7/7 [==============================] - 0s 7ms/step - loss: 0.3531 -
accuracy: 0.8238 - val_loss: 0.4086 - val_accuracy: 0.8163
Epoch 81/100
7/7 [==============================] - 0s 11ms/step - loss: 0.3550 -
accuracy: 0.8342 - val_loss: 0.4098 - val_accuracy: 0.8163
Epoch 82/100
7/7 [==============================] - 0s 11ms/step - loss: 0.4033 -
accuracy: 0.8290 - val_loss: 0.5627 - val_accuracy: 0.6939
Epoch 83/100
7/7 [==============================] - 0s 11ms/step - loss: 0.7076 -
accuracy: 0.7150 - val_loss: 0.4100 - val_accuracy: 0.8163
Epoch 84/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4917 -
accuracy: 0.7668 - val_loss: 0.4036 - val_accuracy: 0.7755
Epoch 85/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3469 -
accuracy: 0.8238 - val_loss: 0.4190 - val_accuracy: 0.7959
Epoch 86/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3565 -
accuracy: 0.8135 - val_loss: 0.4323 - val_accuracy: 0.7959
Epoch 87/100
7/7 [==============================] - 0s 8ms/step - loss: 0.3721 -
accuracy: 0.8290 - val_loss: 0.4248 - val_accuracy: 0.8163
Epoch 88/100
7/7 [==============================] - 0s 7ms/step - loss: 0.5072 -
accuracy: 0.7668 - val_loss: 0.4472 - val_accuracy: 0.7959
Epoch 89/100
7/7 [==============================] - 0s 11ms/step - loss: 0.3671 -
accuracy: 0.8187 - val_loss: 0.4276 - val_accuracy: 0.7755
Epoch 90/100
7/7 [==============================] - 0s 7ms/step - loss: 0.3461 -
accuracy: 0.8083 - val_loss: 0.4649 - val_accuracy: 0.7755
Epoch 91/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4643 -
accuracy: 0.7668 - val_loss: 0.3846 - val_accuracy: 0.8163
Epoch 92/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3531 -
accuracy: 0.8342 - val_loss: 0.3848 - val_accuracy: 0.8163
Epoch 93/100
7/7 [==============================] - 0s 11ms/step - loss: 0.3476 -
```

```
accuracy: 0.8342 - val_loss: 0.3894 - val_accuracy: 0.8163
Epoch 94/100
7/7 [==============================] - 0s 7ms/step - loss: 0.3335 -
accuracy: 0.8394 - val_loss: 0.4205 - val_accuracy: 0.8163
Epoch 95/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3413 -
accuracy: 0.8394 - val_loss: 0.4416 - val_accuracy: 0.7755
Epoch 96/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3605 -
accuracy: 0.8394 - val_loss: 0.3903 - val_accuracy: 0.8163
Epoch 97/100
7/7 [==============================] - 0s 8ms/step - loss: 0.3352 -
accuracy: 0.8394 - val_loss: 0.4082 - val_accuracy: 0.8163
Epoch 98/100
7/7 [==============================] - 0s 11ms/step - loss: 0.3430 -
accuracy: 0.8342 - val_loss: 0.4018 - val_accuracy: 0.8163
Epoch 99/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3454 -
accuracy: 0.8342 - val_loss: 0.4252 - val_accuracy: 0.7959
Epoch 100/100
7/7 [==============================] - 0s 8ms/step - loss: 0.3315 -
accuracy: 0.8342 - val_loss: 0.4336 - val_accuracy: 0.7959

<keras.callbacks.History at 0x7f6df1ea76d0>
```

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

```
2/2 [==============================] - 0s 5ms/step - loss: 0.2609 -
accuracy: 0.8689
Accuracy 0.868852436542511
```

Submission Instructions

```
# 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.

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

```
<IPython.core.display.Javascript object>
```

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

```
<IPython.core.display.Javascript object>
```