Assessment

Congratulations on going through today's course! Hopefully, you've learned some valuable skills along the way and had fun doing it. Now it's time to put those skills to the test. In this assessment, you will train a new model that is able to recognize fresh and rotten fruit. You will need to get the model to a validation accuracy of `92%` in order to pass the assessment, though we challenge you to do even better if you can. You will have the use the skills that you learned in the previous exercises. Specifically, we suggest using some combination of transfer learning, data augmentation, and fine tuning. Once you have trained the model to be at least 92% accurate on the validation dataset, save your model, and then assess its accuracy. Let's get started!

The Dataset

In this exercise, you will train a model to recognize fresh and rotten fruits. The dataset comes from [Kaggle (https://www.kaggle.com/sriramr/fruits-fresh-and-rotten-for-classification)](https://www.kaggle.com/sriramr/fruits-fresh-and-rotten-for-classification), a great place to go if you're interested in starting a project after this class. The dataset structure is in the `data/fruits` folder. There are 6 categories of fruits: fresh apples, fresh oranges, fresh bananas, rotten apples, rotten oranges, and rotten bananas. This will mean that your model will require an output layer of 6 neurons to do the categorization successfully. You'll also need to compile the model with `categorical_crossentropy`, as we have more than two categories.

Load ImageNet Base Model

We encourage you to start with a model pretrained on ImageNet. Load the model with the correct weights, set an input shape, and choose to remove the last layers of the model. Remember that images have three dimensions: a height, and width, and a number of channels. Because these pictures are in color, there will be three channels for red, green, and blue. We've filled in the input shape for you. This cannot be changed or the assessment will fail. If you need a reference for setting up the pretrained model, please take a look at notebook 05b (05b_presidential_doggy_door.ipynb) where we implemented transfer learning.

In [1]:

```
from tensorflow import keras

base_model = keras.applications.VGG16(
    weights='imagenet',
    input_shape=(224, 224, 3),
    include_top=False)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-appl
ications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58892288/58889256 [==============================] - 0s 0us/step
```

Freeze Base Model

Next, we suggest freezing the base model, as done in notebook 05b (05b_presidential_doggy_door.ipynb). This is done so that all the learning from the ImageNet dataset does not get destroyed in the initial training.

In [2]:

```
# Freeze base model
base_model.trainable = False
```

Now it's time to add layers to the pretrained model. Notebook 05b (05b_presidential_doggy_door.ipynb) can be used as a guide. Pay close attention to the last dense layer and make sure it has the correct number of neurons to classify the different types of fruit.

In [3]:

```
# Create inputs with correct shape
inputs = keras.Input(shape=(224, 224, 3))

x = base_model(inputs, training=False)

# Add pooling layer or flatten layer
x = keras.layers.GlobalAveragePooling2D()(x)

# Add final dense layer
outputs = keras.layers.Dense(6, activation = 'softmax')(x)

# Combine inputs and outputs to create model
model = keras.Model(inputs, outputs)
```

In [4]:

```
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0         
_________________________________________________________________
vgg16 (Model)                (None, 7, 7, 512)         14714688  
_________________________________________________________________
global_average_pooling2d (Gl (None, 512)               0         
_________________________________________________________________
dense (Dense)                (None, 6)                 3078      
=================================================================
Total params: 14,717,766
Trainable params: 3,078
Non-trainable params: 14,714,688
_________________________________________________________________
```

Compile Model

Now it's time to compile the model with loss and metrics options. Remember that we're training on a number of different categories, rather than a binary classification problem.

In [7]:

```
model.compile(loss =keras.losses.BinaryCrossentropy(from_logits=True ), metrics = [kera
s.metrics.BinaryAccuracy()])
```

Augment the Data

If you'd like, try to augment the data to improve the dataset. Feel free to look at [notebook 04a (04a_asl_augmentation.ipynb)](04a_asl_augmentation.ipynb) and [notebook 05b (05b_presidential_doggy_door.ipynb)](05b_presidential_doggy_door.ipynb) for augmentation examples. There is also documentation for the [Keras ImageDataGenerator class (https://keras.io/api/preprocessing/image/#imagedatagenerator-class)](https://keras.io/api/preprocessing/image/#imagedatagenerator-class). This step is optional, but it may be helpful to get to 92% accuracy.

In [8]:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(samplewise_center=True,  # set each sample mean to 0
        rotation_range=10,  # randomly rotate images in the range (degrees, 0 to 180)
        zoom_range = 0.1, # Randomly zoom image
        width_shift_range=0.1,  # randomly shift images horizontally (fraction of total
width)
        height_shift_range=0.1,  # randomly shift images vertically (fraction of total
 height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False) # we don't expect Bo to be upside-down so we will not flip
vertically
```

Load Dataset

Now it's time to load the train and validation datasets. Pick the right folders, as well as the right `target_size` of the images (it needs to match the height and width input of the model you've created). If you'd like a reference, you can check out [notebook 05b (05b_presidential_doggy_door.ipynb)](05b_presidential_doggy_door.ipynb).

In [12]:

```
# load and iterate training dataset
train_it = datagen.flow_from_directory('data/fruits/train/',
                                       target_size=(224, 224),
                                       color_mode='rgb',
                                       class_mode="categorical")
# load and iterate validation dataset
valid_it = datagen.flow_from_directory('data/fruits/valid/',
                                      target_size=(224, 224),
                                      color_mode='rgb',
                                      class_mode="categorical")
```

```
Found 1182 images belonging to 6 classes.
Found 330 images belonging to 6 classes.
```

Train the Model

Time to train the model! Pass the `train` and `valid` iterators into the `fit` function, as well as setting your desired number of epochs.

In [14]:

```
model.fit(train_it,
          validation_data=valid_it,
          steps_per_epoch=train_it.samples/train_it.batch_size,
          validation_steps=valid_it.samples/valid_it.batch_size,
          epochs=20)
```

```
Epoch 1/20
37/36 [==============================] - 27s 729ms/step - loss: 0.7263 - b
inary_accuracy: 0.8072 - val_loss: 0.7027 - val_binary_accuracy: 0.8525
Epoch 2/20
37/36 [==============================] - 22s 587ms/step - loss: 0.6770 - b
inary_accuracy: 0.9072 - val_loss: 0.6682 - val_binary_accuracy: 0.9232
Epoch 3/20
37/36 [==============================] - 20s 531ms/step - loss: 0.6582 - b
inary_accuracy: 0.9459 - val_loss: 0.6570 - val_binary_accuracy: 0.9480
Epoch 4/20
37/36 [==============================] - 20s 534ms/step - loss: 0.6494 - b
inary_accuracy: 0.9636 - val_loss: 0.6519 - val_binary_accuracy: 0.9601
Epoch 5/20
37/36 [==============================] - 20s 534ms/step - loss: 0.6442 - b
inary_accuracy: 0.9748 - val_loss: 0.6443 - val_binary_accuracy: 0.9778
Epoch 6/20
37/36 [==============================] - 20s 533ms/step - loss: 0.6414 - b
inary_accuracy: 0.9793 - val_loss: 0.6426 - val_binary_accuracy: 0.9773
Epoch 7/20
37/36 [==============================] - 20s 534ms/step - loss: 0.6392 - b
inary_accuracy: 0.9832 - val_loss: 0.6413 - val_binary_accuracy: 0.9813
Epoch 8/20
37/36 [==============================] - 20s 538ms/step - loss: 0.6362 - b
inary_accuracy: 0.9903 - val_loss: 0.6407 - val_binary_accuracy: 0.9813
Epoch 9/20
37/36 [==============================] - 20s 536ms/step - loss: 0.6363 - b
inary_accuracy: 0.9891 - val_loss: 0.6376 - val_binary_accuracy: 0.9869
Epoch 10/20
37/36 [==============================] - 20s 539ms/step - loss: 0.6352 - b
inary_accuracy: 0.9924 - val_loss: 0.6368 - val_binary_accuracy: 0.9879
Epoch 11/20
37/36 [==============================] - 19s 527ms/step - loss: 0.6347 - b
inary_accuracy: 0.9925 - val_loss: 0.6389 - val_binary_accuracy: 0.9823
Epoch 12/20
37/36 [==============================] - 20s 535ms/step - loss: 0.6345 - b
inary_accuracy: 0.9913 - val_loss: 0.6376 - val_binary_accuracy: 0.9843
Epoch 13/20
37/36 [==============================] - 20s 541ms/step - loss: 0.6335 - b
inary_accuracy: 0.9952 - val_loss: 0.6390 - val_binary_accuracy: 0.9823
Epoch 14/20
37/36 [==============================] - 20s 537ms/step - loss: 0.6335 - b
inary_accuracy: 0.9938 - val_loss: 0.6368 - val_binary_accuracy: 0.9874
Epoch 15/20
37/36 [==============================] - 20s 539ms/step - loss: 0.6334 - b
inary_accuracy: 0.9948 - val_loss: 0.6373 - val_binary_accuracy: 0.9854
Epoch 16/20
37/36 [==============================] - 20s 537ms/step - loss: 0.6324 - b
inary_accuracy: 0.9956 - val_loss: 0.6368 - val_binary_accuracy: 0.9859
Epoch 17/20
37/36 [==============================] - 20s 537ms/step - loss: 0.6330 - b
inary_accuracy: 0.9951 - val_loss: 0.6391 - val_binary_accuracy: 0.9813
Epoch 18/20
37/36 [==============================] - 20s 535ms/step - loss: 0.6324 - b
inary_accuracy: 0.9963 - val_loss: 0.6351 - val_binary_accuracy: 0.9904
Epoch 19/20
37/36 [==============================] - 20s 538ms/step - loss: 0.6323 - b
inary_accuracy: 0.9958 - val_loss: 0.6375 - val_binary_accuracy: 0.9864
Epoch 20/20
37/36 [==============================] - 20s 535ms/step - loss: 0.6320 - b
inary_accuracy: 0.9968 - val_loss: 0.6349 - val_binary_accuracy: 0.9909
```

Out[14]:

```
<tensorflow.python.keras.callbacks.History at 0x7f504851e9e8>
```

Unfreeze Model for Fine Tuning

If you have reached 92% validation accuracy already, this next step is optional. If not, we suggest fine tuning the model with a very low learning rate.

In [18]:

```
# Unfreeze the base model
base_model.trainable = True

# Compile the model with a low learning rate
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate = .00001),  # Very low l
earning rate
              loss=keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=[keras.metrics.BinaryAccuracy()])
```

In [19]:

```
model.fit(train_it,
          validation_data=valid_it,
          steps_per_epoch=train_it.samples/train_it.batch_size,
          validation_steps=valid_it.samples/valid_it.batch_size,
          epochs=10)
```

```
Epoch 1/10
37/36 [==============================] - 35s 934ms/step - loss: 0.6349 - b
inary_accuracy: 0.9911 - val_loss: 0.6345 - val_binary_accuracy: 0.9919
Epoch 2/10
37/36 [==============================] - 21s 574ms/step - loss: 0.6327 - b
inary_accuracy: 0.9942 - val_loss: 0.6363 - val_binary_accuracy: 0.9889
Epoch 3/10
37/36 [==============================] - 21s 559ms/step - loss: 0.6328 - b
inary_accuracy: 0.9938 - val_loss: 0.6343 - val_binary_accuracy: 0.9919
Epoch 4/10
37/36 [==============================] - 21s 570ms/step - loss: 0.6311 - b
inary_accuracy: 0.9980 - val_loss: 0.6343 - val_binary_accuracy: 0.9919
Epoch 5/10
37/36 [==============================] - 21s 558ms/step - loss: 0.6310 - b
inary_accuracy: 0.9977 - val_loss: 0.6346 - val_binary_accuracy: 0.9909
Epoch 6/10
37/36 [==============================] - 21s 571ms/step - loss: 0.6315 - b
inary_accuracy: 0.9966 - val_loss: 0.6351 - val_binary_accuracy: 0.9894
Epoch 7/10
37/36 [==============================] - 20s 548ms/step - loss: 0.6303 - b
inary_accuracy: 0.9994 - val_loss: 0.6345 - val_binary_accuracy: 0.9909
Epoch 8/10
37/36 [==============================] - 21s 568ms/step - loss: 0.6307 - b
inary_accuracy: 0.9983 - val_loss: 0.6332 - val_binary_accuracy: 0.9939
Epoch 9/10
37/36 [==============================] - 21s 562ms/step - loss: 0.6301 - b
inary_accuracy: 0.9997 - val_loss: 0.6334 - val_binary_accuracy: 0.9919
Epoch 10/10
37/36 [==============================] - 21s 564ms/step - loss: 0.6306 - b
inary_accuracy: 0.9986 - val_loss: 0.6321 - val_binary_accuracy: 0.9960
```

Out[19]:

```
<tensorflow.python.keras.callbacks.History at 0x7f505405f0b8>
```

Evaluate the Model

Hopefully, you now have a model that has a validation accuracy of 92% or higher. If not, you may want to go back and either run more epochs of training, or adjust your data augmentation.

Once you are satisfied with the validation accuracy, evaluate the model by executing the following cell. The evaluate function will return a tuple, where the first value is your loss, and the second value is your accuracy. To pass, the model will need have an accuracy value of `92% or higher` .

In [20]:

```
model.evaluate(valid_it, steps=valid_it.samples/valid_it.batch_size)
```

```
11/10 [================================] - 4s 344ms/step - loss: 0.6348 -
binary_accuracy: 0.9899
```

Out[20]:

```
[0.6347667574882507, 0.9898989796638489]
```

Run the Assessment

To assess your model run the following two cells.

NOTE: `run_assessment` assumes your model is named `model` and your validation data iterator is called `valid_it`. If for any reason you have modified these variable names, please update the names of the arguments passed to `run_assessment`.

In [21]:

```
from run_assessment import run_assessment
```

In [22]:

```
run_assessment(model, valid_it)
```

```
Evaluating model 5 times to obtain average accuracy...

11/10 [================================] - 4s 349ms/step - loss: 0.6335 -
binary_accuracy: 0.9929
11/10 [================================] - 4s 379ms/step - loss: 0.6340 -
binary_accuracy: 0.9909
11/10 [================================] - 4s 341ms/step - loss: 0.6342 -
binary_accuracy: 0.9919
11/10 [================================] - 4s 352ms/step - loss: 0.6337 -
binary_accuracy: 0.9919
11/10 [================================] - 4s 349ms/step - loss: 0.6341 -
binary_accuracy: 0.9919

Accuracy required to pass the assessment is 0.92 or greater.
Your average accuracy is 0.9919.

Congratulations! You passed the assessment!
See instructions below to generate a certificate.
```

Generate a Certificate

If you passed the assessment, please return to the course page (shown below) and click the "ASSESS TASK" button, which will generate your certificate for the course.