[Header
(https://www.nvidia.com/dli)](https://www.nvidia.com/dli)

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 to 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)](05b_presidential_doggy_door.ipynb) where we implemented transfer learning.

In [2]:

```
from tensorflow import keras

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

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 [3]:

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

Add Layers to Model

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 [4]:

```
# 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(1, activation = 'softmax')(x)

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

In [5]:

```
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, 1)                 513       
=================================================================
Total params: 14,715,201
Trainable params: 513
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 [8]:

```
model.compile(loss='categorical_crossentropy', metrics=['accuracy'])
```

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) and notebook 05b (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). This step is optional, but it may be helpful to get to 92% accuracy.

In [11]:

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

datagen_train = ImageDataGenerator(
    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 horizontally
    vertical_flip=False, # Don't randomly flip images vertically
)

datagen_valid = ImageDataGenerator(samplewise_center=True)
```

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). For a reference, check out notebook 05b (05b_presidential_doggy_door.ipynb).

In [13]:

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

```
Found 1182 images belonging to 6 classes.
Found 329 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 the desired number of epochs.

In [16]:

```
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
```

```
---------------------------------------------------------------------------
ResourceExhaustedError                    Traceback (most recent call last)
<ipython-input-16-4a228df73241> in <module>
      3           steps_per_epoch=(train_it.samples/train_it.batch_size),
      4           validation_steps=(valid_it.samples/valid_it.batch_size),
----> 5           epochs=20)

/usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/trainin
g.py in _method_wrapper(self, *args, **kwargs)
     64   def _method_wrapper(self, *args, **kwargs):
     65     if not self._in_multi_worker_mode():  # pylint: disable=protected
-access
---> 66       return method(self, *args, **kwargs)
     67 
     68     # Running inside `run_distribute_coordinator` already.

/usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/trainin
g.py in fit(self, x, y, batch_size, epochs, verbose, callbacks, validation_sp
lit, validation_data, shuffle, class_weight, sample_weight, initial_epoch, st
eps_per_epoch, validation_steps, validation_batch_size, validation_freq, max_
queue_size, workers, use_multiprocessing)
    846                 batch_size=batch_size):
    847               callbacks.on_train_batch_begin(step)
--> 848               tmp_logs = train_function(iterator)
    849               # Catch OutOfRangeError for Datasets of unknown size.
    850               # This blocks until the batch has finished executing.

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/def_function.p
y in __call__(self, *args, **kwds)
    578         xla_context.Exit()
    579     else:
--> 580       result = self._call(*args, **kwds)
    581 
    582     if tracing_count == self._get_tracing_count():

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/def_function.p
y in _call(self, *args, **kwds)
    609       # In this case we have created variables on the first call, so
 we run the
    610       # defunned version which is guaranteed to never create variable
s.
--> 611       return self._stateless_fn(*args, **kwds)  # pylint: disable=not
-callable
    612     elif self._stateful_fn is not None:
    613       # Release the lock early so that multiple threads can perform t
he call

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.py in
__call__(self, *args, **kwargs)
   2418     with self._lock:
   2419       graph_function, args, kwargs = self._maybe_define_function(args
, kwargs)
-> 2420     return graph_function._filtered_call(args, kwargs)  # pylint: dis
able=protected-access
   2421 
   2422   @property
```

```
/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.py in
_filtered_call(self, args, kwargs)
   1663         if isinstance(t, (ops.Tensor,
   1664                           resource_variable_ops.BaseResourceVariabl
e))),
-> 1665         self.captured_inputs)
   1666
   1667   def _call_flat(self, args, captured_inputs, cancellation_manager=No
ne):

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.py in
_call_flat(self, args, captured_inputs, cancellation_manager)
   1744       # No tape is watching; skip to running the function.
   1745       return self._build_call_outputs(self._inference_function.call(
-> 1746           ctx, args, cancellation_manager=cancellation_manager))
   1747     forward_backward = self._select_forward_and_backward_functions(
   1748         args,

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.py in
call(self, ctx, args, cancellation_manager)
    596               inputs=args,
    597               attrs=attrs,
--> 598               ctx=ctx)
    599         else:
    600           outputs = execute.execute_with_cancellation(

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/execute.py in
quick_execute(op_name, num_outputs, inputs, attrs, ctx, name)
     58     ctx.ensure_initialized()
     59     tensors = pywrap_tfe.TFE_Py_Execute(ctx._handle, device_name, op_
name,
---> 60                                         inputs, attrs, num_outputs)
     61   except core._NotOkStatusException as e:
     62     if name is not None:

ResourceExhaustedError:  OOM when allocating tensor with shape[32,64,224,224]
and type float on /job:localhost/replica:0/task:0/device:GPU:0 by allocator G
PU_0_bfc
         [[node model/vgg16/block1_conv1/Conv2D (defined at <ipython-input-14
-de257c3db5a5>:5) ]]
Hint: If you want to see a list of allocated tensors when OOM happens, add re
port_tensor_allocations_upon_oom to RunOptions for current allocation info.
 [Op:__inference_train_function_1370]

Function call stack:
train_function
```

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 [ ]:

```
# 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),
              loss='categorical_crossentropy' , metrics = ['accuracy'])
```

In [ ]:

```
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)
```

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 [ ]:

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

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 [ ]:

```
from run_assessment import run_assessment
```

In [ ]:

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

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.

Header
(https://www.nvidia.com/dli)