This notebook is an exercise in the Computer Vision course. You can reference the tutorial at this link.

Introduction

In these exercises, you'll explore what effect various random transformations have on an image, consider what kind of augmentation might be appropriate on a given dataset, and then use data augmentation with the *Car or Truck* dataset to train a custom network.

Run the cell below to set everything up!

In [1]:

```
# Setup feedback system
from learntools.core import binder
binder.bind(globals())
from learntools.computer_vision.ex6 import *

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.layers.experimental import preprocessing

# Imports
import os, warnings
import matplotlib.pyplot as plt
from matplotlib import gridspec

import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing import image_dataset_from_directory

# Reproducability
def set_seed(seed=31415):
    np.random.seed(seed)
    tf.random.set_seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    os.environ['TF_DETERMINISTIC_OPS'] = '1'
set_seed()

# Set Matplotlib defaults
plt.rc('figure', autolayout=True)
plt.rc('axes', labelweight='bold', labelsize='large',
       titleweight='bold', titlesize=18, titlepad=10)
plt.rc('image', cmap='magma')
warnings.filterwarnings("ignore") # to clean up output cells


# Load training and validation sets
ds_train_ = image_dataset_from_directory(
    '../input/car-or-truck/train',
    labels='inferred',
    label_mode='binary',
    image_size=[128, 128],
    interpolation='nearest',
    batch_size=64,
    shuffle=True,
```

```python
)
ds_valid_ = image_dataset_from_directory(
    '../input/car-or-truck/valid',
    labels='inferred',
    label_mode='binary',
    image_size=[128, 128],
    interpolation='nearest',
    batch_size=64,
    shuffle=False,
)

# Data Pipeline
def convert_to_float(image, label):
    image = tf.image.convert_image_dtype(image, dtype=tf.float32)
    return image, label

AUTOTUNE = tf.data.experimental.AUTOTUNE
ds_train = (
    ds_train_
    .map(convert_to_float)
    .cache()
    .prefetch(buffer_size=AUTOTUNE)
)
ds_valid = (
    ds_valid_
    .map(convert_to_float)
    .cache()
    .prefetch(buffer_size=AUTOTUNE)
)
```

```
Found 5117 files belonging to 2 classes.
2022-12-19 13:55:39.601696: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-19 13:55:39.713726: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-19 13:55:39.714643: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-19 13:55:39.722794: I tensorflow/core/platform/cpu_feature_guard.cc:142] This Te
nsorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the
following CPU instructions in performance-critical operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler fla
gs.
2022-12-19 13:55:39.723116: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-19 13:55:39.724067: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-19 13:55:39.724956: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-19 13:55:41.807768: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-19 13:55:41.808655: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
```

```
2022-12-19 13:55:41.809381: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2022-12-19 13:55:41.810010: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1510] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 15401 MB memory:  -> device: 0, name: Tesla P100-PCIE-16GB, pci bus id: 0000:00:04.0, compute capability: 6.0
Found 5051 files belonging to 2 classes.
```

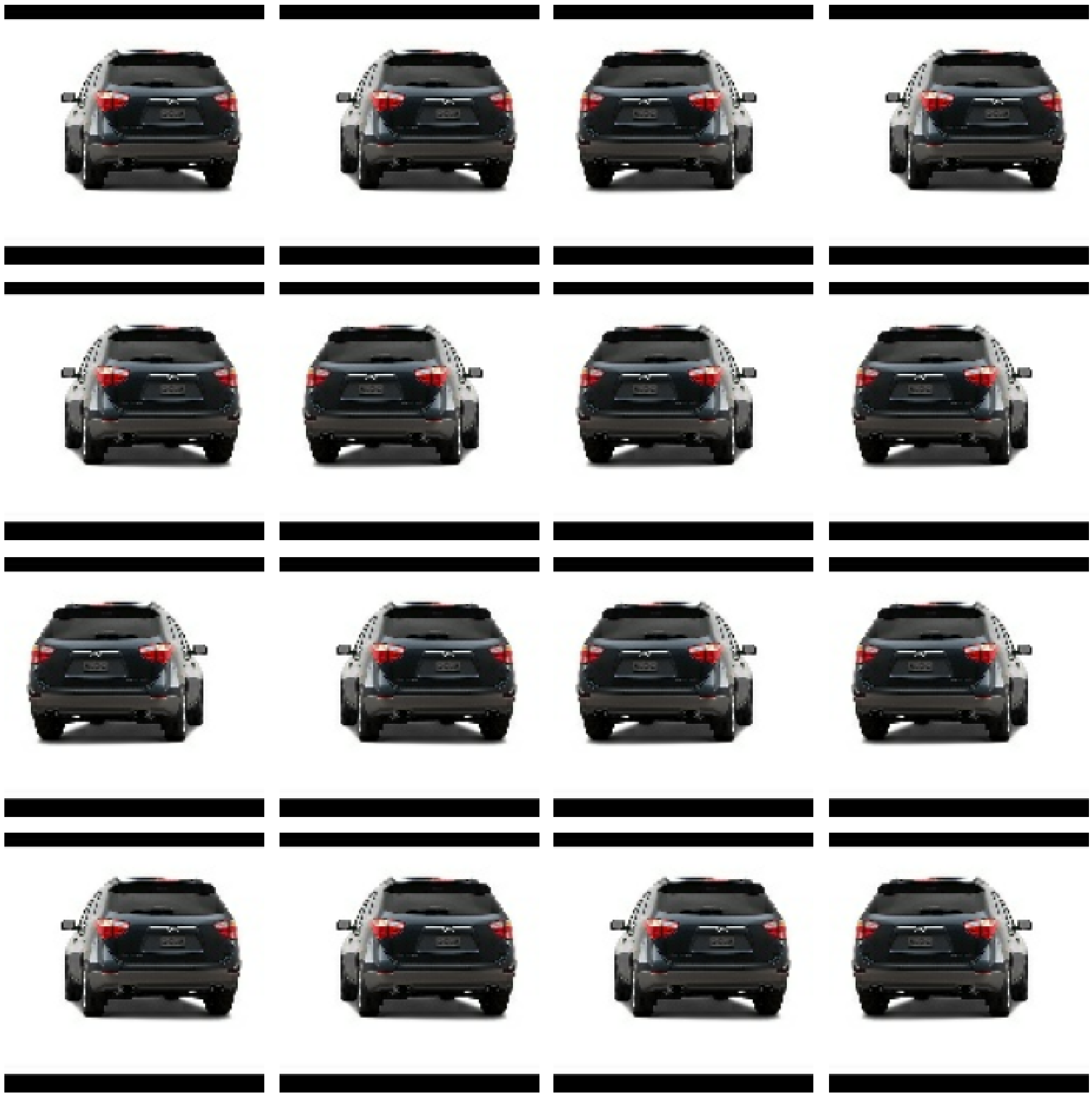
(Optional) Explore Augmentation

Uncomment a transformation and run the cell to see what it does. You can experiment with the parameter values too, if you like. (The `factor` parameters should be greater than 0 and, generally, less than 1.) Run the cell again if you'd like to get a new random image.

In [2]:

```
# all of the "factor" parameters indicate a percent-change
augment = keras.Sequential([
    # preprocessing.RandomContrast(factor=0.5),
    preprocessing.RandomFlip(mode='horizontal'), # meaning, left-to-right
    # preprocessing.RandomFlip(mode='vertical'), # meaning, top-to-bottom
    # preprocessing.RandomWidth(factor=0.15), # horizontal stretch
    # preprocessing.RandomRotation(factor=0.20),
    # preprocessing.RandomTranslation(height_factor=0.1, width_factor=0.1),
])


ex = next(iter(ds_train.unbatch().map(lambda x, y: x).batch(1)))

plt.figure(figsize=(10,10))
for i in range(16):
    image = augment(ex, training=True)
    plt.subplot(4, 4, i+1)
    plt.imshow(tf.squeeze(image))
    plt.axis('off')
plt.show()
```

```
2022-12-19 13:55:48.176031: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
2022-12-19 13:55:50.043178: W tensorflow/core/kernels/data/cache_dataset_ops.cc:768] The calling iterator did not fully read the dataset being cached. In order to avoid unexpected truncation of the dataset, the partially cached contents of the dataset  will be discarded. This can happen if you have an input pipeline similar to `dataset.cache().take(k).repeat()`. You should use `dataset.take(k).cache().repeat()` instead.
```



Do the transformations you chose seem reasonable for the *Car or Truck* dataset?

In this exercise, we'll look at a few datasets and think about what kind of augmentation might be appropriate. Your reasoning might be different that what we discuss in the solution. That's okay. The point of these problems is just to think about how a transformation might interact with a classification problem -- for better or worse.

The EuroSAT dataset consists of satellite images of the Earth classified by geographic feature. Below are a number of images from this dataset.

1) EuroSAT

What kinds of transformations might be appropriate for this dataset?

In [3]:

```
# View the solution (Run this code cell to receive credit!)
q_1.check()
```

Correct:

It seems to this author that flips and rotations would be worth trying first since there's no concept of orientation for pictures taken straight overhead. None of the transformations seem likely to confuse classes, however.

In [4]:

```
# Lines below will give you a hint
#q_1.solution()
```

The TensorFlow Flowers dataset consists of photographs of flowers of several species. Below is a sample.

2) TensorFlow Flowers

What kinds of transformations might be appropriate for the TensorFlow Flowers dataset?

In [5]:
```
# View the solution (Run this code cell to receive credit!)
q_2.check()
```

Correct:

It seems to this author that horizontal flips and moderate rotations would be worth trying first. Some augmentation libraries include transformations of hue (like red to blue). Since the color of a flower seems distinctive of its class, a change of hue might be less successful. On the other hand, there is suprising variety in cultivated flowers like roses, so, depending on the dataset, this might be an improvement after all!

In [6]:
```
# Lines below will give you a hint
#q_2.solution()
```

Now you'll use data augmentation with a custom convnet similar to the one you built in Exercise 5. Since data augmentation effectively increases the size of the dataset, we can increase the capacity of the model in turn without as much risk of overfitting.

3) Add Preprocessing Layers

Add these preprocessing layers to the given model.

```
preprocessing.RandomContrast(factor=0.10),
preprocessing.RandomFlip(mode='horizontal'),
preprocessing.RandomRotation(factor=0.10),
```

In [7]:

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.InputLayer(input_shape=[128, 128, 3]),

    # Data Augmentation
    preprocessing.RandomContrast(factor=0.10),
    preprocessing.RandomFlip(mode='horizontal'),
    preprocessing.RandomRotation(factor=0.10),

    # Block One
    layers.BatchNormalization(renorm=True),
    layers.Conv2D(filters=64, kernel_size=3, activation='relu', padding='same'),
    layers.MaxPool2D(),

    # Block Two
    layers.BatchNormalization(renorm=True),
    layers.Conv2D(filters=128, kernel_size=3, activation='relu', padding='same'),
    layers.MaxPool2D(),

    # Block Three
    layers.BatchNormalization(renorm=True),
    layers.Conv2D(filters=256, kernel_size=3, activation='relu', padding='same'),
    layers.Conv2D(filters=256, kernel_size=3, activation='relu', padding='same'),
    layers.MaxPool2D(),

    # Head
    layers.BatchNormalization(renorm=True),
    layers.Flatten(),
    layers.Dense(8, activation='relu'),
    layers.Dense(1, activation='sigmoid'),
])

# Check your answer
q_3.check()
```

Correct

In [8]:

```
# Lines below will give you a hint or solution code
#q_3.hint()
```

```
#q_3.solution()
```

Now we'll train the model. Run the next cell to compile it with a loss and accuracy metric and fit it to the training set.

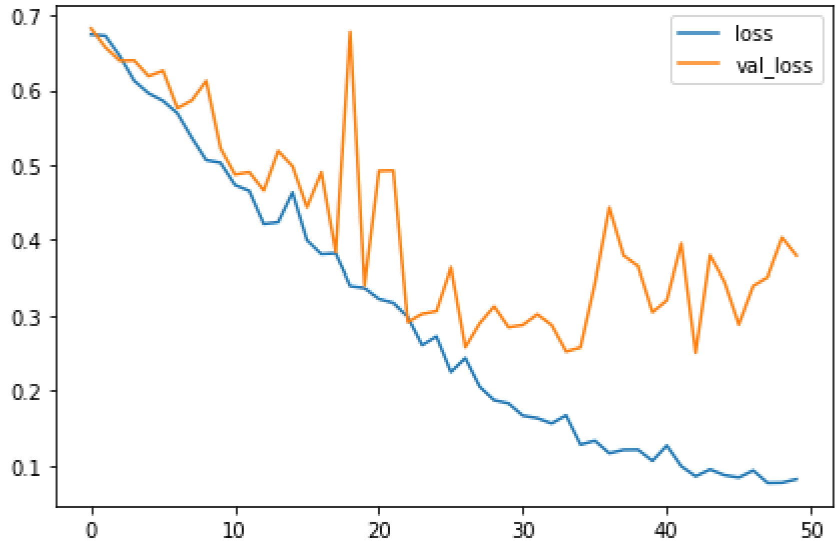
In [9]:

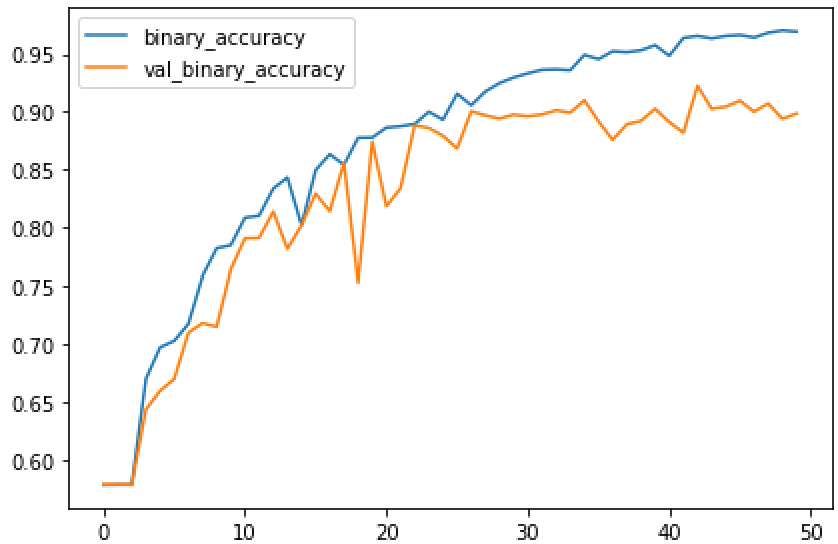
```
optimizer = tf.keras.optimizers.Adam(epsilon=0.01)
model.compile(
    optimizer=optimizer,
    loss='binary_crossentropy',
    metrics=['binary_accuracy'],
)

history = model.fit(
    ds_train,
    validation_data=ds_valid,
    epochs=50,
)

# Plot learning curves
import pandas as pd
history_frame = pd.DataFrame(history.history)
history_frame.loc[:, ['loss', 'val_loss']].plot()
history_frame.loc[:, ['binary_accuracy', 'val_binary_accuracy']].plot();
```

```
Epoch 1/50
2022-12-19 13:55:54.915690: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369] Loaded cu
DNN version 8005
80/80 [==============================] - 42s 409ms/step - loss: 0.6742 - binary_accurac
y: 0.5781 - val_loss: 0.6822 - val_binary_accuracy: 0.5785
Epoch 2/50
80/80 [==============================] - 8s 102ms/step - loss: 0.6724 - binary_accuracy:
0.5783 - val_loss: 0.6570 - val_binary_accuracy: 0.5785
Epoch 3/50
80/80 [==============================] - 8s 103ms/step - loss: 0.6455 - binary_accuracy:
0.5785 - val_loss: 0.6391 - val_binary_accuracy: 0.5781
Epoch 4/50
80/80 [==============================] - 8s 103ms/step - loss: 0.6125 - binary_accuracy:
0.6699 - val_loss: 0.6394 - val_binary_accuracy: 0.6434
Epoch 5/50
80/80 [==============================] - 8s 102ms/step - loss: 0.5957 - binary_accuracy:
0.6971 - val_loss: 0.6185 - val_binary_accuracy: 0.6595
Epoch 6/50
80/80 [==============================] - 8s 102ms/step - loss: 0.5858 - binary_accuracy:
0.7028 - val_loss: 0.6259 - val_binary_accuracy: 0.6698
Epoch 7/50
80/80 [==============================] - 8s 102ms/step - loss: 0.5691 - binary_accuracy:
0.7174 - val_loss: 0.5758 - val_binary_accuracy: 0.7098
Epoch 8/50
80/80 [==============================] - 8s 103ms/step - loss: 0.5366 - binary_accuracy:
0.7583 - val_loss: 0.5863 - val_binary_accuracy: 0.7181
Epoch 9/50
80/80 [==============================] - 8s 102ms/step - loss: 0.5066 - binary_accuracy:
0.7821 - val_loss: 0.6122 - val_binary_accuracy: 0.7149
Epoch 10/50
80/80 [==============================] - 8s 102ms/step - loss: 0.5030 - binary_accuracy:
0.7848 - val_loss: 0.5223 - val_binary_accuracy: 0.7640
Epoch 11/50
80/80 [==============================] - 8s 103ms/step - loss: 0.4734 - binary_accuracy:
```

```
0.8085 - val_loss: 0.4873 - val_binary_accuracy: 0.7907
Epoch 12/50
80/80 [==============================] - 8s 102ms/step - loss: 0.4655 - binary_accuracy: 
0.8102 - val_loss: 0.4905 - val_binary_accuracy: 0.7909
Epoch 13/50
80/80 [==============================] - 8s 102ms/step - loss: 0.4214 - binary_accuracy: 
0.8337 - val_loss: 0.4661 - val_binary_accuracy: 0.8139
Epoch 14/50
80/80 [==============================] - 8s 103ms/step - loss: 0.4236 - binary_accuracy: 
0.8433 - val_loss: 0.5189 - val_binary_accuracy: 0.7816
Epoch 15/50
80/80 [==============================] - 8s 102ms/step - loss: 0.4634 - binary_accuracy: 
0.8020 - val_loss: 0.4984 - val_binary_accuracy: 0.8020
Epoch 16/50
80/80 [==============================] - 8s 102ms/step - loss: 0.3999 - binary_accuracy: 
0.8497 - val_loss: 0.4437 - val_binary_accuracy: 0.8293
Epoch 17/50
80/80 [==============================] - 8s 102ms/step - loss: 0.3812 - binary_accuracy: 
0.8634 - val_loss: 0.4906 - val_binary_accuracy: 0.8141
Epoch 18/50
80/80 [==============================] - 8s 103ms/step - loss: 0.3825 - binary_accuracy: 
0.8542 - val_loss: 0.3841 - val_binary_accuracy: 0.8561
Epoch 19/50
80/80 [==============================] - 8s 102ms/step - loss: 0.3397 - binary_accuracy: 
0.8777 - val_loss: 0.6777 - val_binary_accuracy: 0.7529
Epoch 20/50
80/80 [==============================] - 8s 103ms/step - loss: 0.3371 - binary_accuracy: 
0.8779 - val_loss: 0.3402 - val_binary_accuracy: 0.8743
Epoch 21/50
80/80 [==============================] - 8s 103ms/step - loss: 0.3225 - binary_accuracy: 
0.8865 - val_loss: 0.4921 - val_binary_accuracy: 0.8185
Epoch 22/50
80/80 [==============================] - 8s 102ms/step - loss: 0.3173 - binary_accuracy: 
0.8876 - val_loss: 0.4926 - val_binary_accuracy: 0.8339
Epoch 23/50
80/80 [==============================] - 8s 102ms/step - loss: 0.2986 - binary_accuracy: 
0.8896 - val_loss: 0.2915 - val_binary_accuracy: 0.8885
Epoch 24/50
80/80 [==============================] - 8s 102ms/step - loss: 0.2609 - binary_accuracy: 
0.9001 - val_loss: 0.3026 - val_binary_accuracy: 0.8862
Epoch 25/50
80/80 [==============================] - 8s 103ms/step - loss: 0.2726 - binary_accuracy: 
0.8933 - val_loss: 0.3063 - val_binary_accuracy: 0.8794
Epoch 26/50
80/80 [==============================] - 8s 102ms/step - loss: 0.2250 - binary_accuracy: 
0.9158 - val_loss: 0.3639 - val_binary_accuracy: 0.8683
Epoch 27/50
80/80 [==============================] - 8s 102ms/step - loss: 0.2438 - binary_accuracy: 
0.9058 - val_loss: 0.2582 - val_binary_accuracy: 0.9006
Epoch 28/50
80/80 [==============================] - 8s 103ms/step - loss: 0.2055 - binary_accuracy: 
0.9177 - val_loss: 0.2901 - val_binary_accuracy: 0.8971
Epoch 29/50
80/80 [==============================] - 8s 102ms/step - loss: 0.1875 - binary_accuracy: 
0.9250 - val_loss: 0.3125 - val_binary_accuracy: 0.8943
Epoch 30/50
80/80 [==============================] - 8s 102ms/step - loss: 0.1833 - binary_accuracy: 
0.9298 - val_loss: 0.2848 - val_binary_accuracy: 0.8976
Epoch 31/50
80/80 [==============================] - 8s 103ms/step - loss: 0.1669 - binary_accuracy: 
```

```
0.9334 - val_loss: 0.2879 - val_binary_accuracy: 0.8963
Epoch 32/50
80/80 [==============================] - 8s 102ms/step - loss: 0.1635 - binary_accuracy:
0.9365 - val_loss: 0.3020 - val_binary_accuracy: 0.8978
Epoch 33/50
80/80 [==============================] - 8s 102ms/step - loss: 0.1565 - binary_accuracy:
0.9369 - val_loss: 0.2877 - val_binary_accuracy: 0.9014
Epoch 34/50
80/80 [==============================] - 8s 102ms/step - loss: 0.1673 - binary_accuracy:
0.9361 - val_loss: 0.2523 - val_binary_accuracy: 0.8994
Epoch 35/50
80/80 [==============================] - 8s 103ms/step - loss: 0.1282 - binary_accuracy:
0.9494 - val_loss: 0.2578 - val_binary_accuracy: 0.9101
Epoch 36/50
80/80 [==============================] - 8s 102ms/step - loss: 0.1335 - binary_accuracy:
0.9457 - val_loss: 0.3427 - val_binary_accuracy: 0.8919
Epoch 37/50
80/80 [==============================] - 8s 102ms/step - loss: 0.1167 - binary_accuracy:
0.9525 - val_loss: 0.4436 - val_binary_accuracy: 0.8759
Epoch 38/50
80/80 [==============================] - 8s 103ms/step - loss: 0.1213 - binary_accuracy:
0.9517 - val_loss: 0.3794 - val_binary_accuracy: 0.8893
Epoch 39/50
80/80 [==============================] - 8s 102ms/step - loss: 0.1214 - binary_accuracy:
0.9533 - val_loss: 0.3652 - val_binary_accuracy: 0.8923
Epoch 40/50
80/80 [==============================] - 8s 102ms/step - loss: 0.1066 - binary_accuracy:
0.9578 - val_loss: 0.3048 - val_binary_accuracy: 0.9028
Epoch 41/50
80/80 [==============================] - 8s 103ms/step - loss: 0.1270 - binary_accuracy:
0.9486 - val_loss: 0.3206 - val_binary_accuracy: 0.8913
Epoch 42/50
80/80 [==============================] - 8s 102ms/step - loss: 0.0995 - binary_accuracy:
0.9640 - val_loss: 0.3957 - val_binary_accuracy: 0.8820
Epoch 43/50
80/80 [==============================] - 8s 102ms/step - loss: 0.0859 - binary_accuracy:
0.9658 - val_loss: 0.2507 - val_binary_accuracy: 0.9226
Epoch 44/50
80/80 [==============================] - 8s 102ms/step - loss: 0.0951 - binary_accuracy:
0.9637 - val_loss: 0.3799 - val_binary_accuracy: 0.9028
Epoch 45/50
80/80 [==============================] - 8s 103ms/step - loss: 0.0875 - binary_accuracy:
0.9658 - val_loss: 0.3453 - val_binary_accuracy: 0.9046
Epoch 46/50
80/80 [==============================] - 8s 102ms/step - loss: 0.0843 - binary_accuracy:
0.9666 - val_loss: 0.2881 - val_binary_accuracy: 0.9097
Epoch 47/50
80/80 [==============================] - 8s 102ms/step - loss: 0.0938 - binary_accuracy:
0.9644 - val_loss: 0.3398 - val_binary_accuracy: 0.9002
Epoch 48/50
80/80 [==============================] - 8s 103ms/step - loss: 0.0771 - binary_accuracy:
0.9685 - val_loss: 0.3514 - val_binary_accuracy: 0.9073
Epoch 49/50
80/80 [==============================] - 8s 102ms/step - loss: 0.0774 - binary_accuracy:
0.9705 - val_loss: 0.4036 - val_binary_accuracy: 0.8941
Epoch 50/50
80/80 [==============================] - 8s 102ms/step - loss: 0.0821 - binary_accuracy:
0.9695 - val_loss: 0.3795 - val_binary_accuracy: 0.8986
```





4) Train Model

Examine the training curves. What there any sign of overfitting? How does the performance of this model compare to other models you've trained in this course?

In [10]:
```
# View the solution (Run this code cell to receive credit!)
q_4.solution()
```

Solution: The learning curves in this model stayed close together for much longer than in previous models. This suggests that the augmentation helped prevent overfitting, allowing the model to continue improving.

And notice that this model achieved the highest accuracy of all the models in the course! This won't always be the case, but it shows that a well-designed custom convnet can sometimes perform as well or better than a much larger pretrained model. Depending on your application, having a smaller model (which requires fewer resources) could be a big advantage.

Conclusion

Data augmentation is a powerful and commonly-used tool to improve model training, not only for convolutional networks, but for many other kinds of neural network models as well. Whatever your problem, the principle remains the same: you can make up for an inadequacy in your data by adding in "fake" data to cover it over. Experimenting with augmentations is a great way to find out just how far your data can go!

The End

That's all for **Computer Vision** on Kaggle Learn! Are you ready to apply your knowledge? Check out our two bonus lessons! They'll walk you through preparing a submission for a competition while you learn how to train neural nets with TPUs, Kaggle's most advanced accelerator. At the end, you'll have a complete notebook ready to extend with ideas of your own.

- Create Your First Submission - Prepare a submission for our *Petals to the Metal* Getting Started competition. You'll train a neural net to recognize over 100 species of flowers.
- Cassava Leaf Disease - Rather compete for money and medals? Train a neural net to diagnose common diseases in the cassava plant, a staple security crop in Africa.

Have fun learning!

Have questions or comments? Visit the course discussion forum to chat with other learners.