```
# Import DS Python Libraries
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.preprocessing import image as image_utils
from tensorflow.keras.applications.imagenet_utils import preprocess_input

from google.colab import drive
drive.mount('/drive')
%cd "/drive/MyDrive/cnn_demo/"
```

```
Drive already mounted at /drive; to attempt to forcibly remount, call drive.mount("/drive", force_remount=True).
/drive/MyDrive/cnn_demo
```

CNN Applications

Convolutional neural networks (CNNs) are great NN models for computer vision tasks, where we make models that "see" images and make decisions on them, such as object detection or image classification. Here, we'll look at the application of a CNN to an image classification problem: making a doggy door that *only* lets in a *specific* dog!

To do this with high accuracy and less training time, we'll take advantage of the many pretrained CNN models available in Keras and use them with a technique called **transfer learning**. This is the technique where we use a pretrained model and add more to it to train those *new* layers on our specific data without training the pretrained layers. This cuts back on training time for us and allows us to specialize this model from its general "knowledge" without making it from scratch, as well as in some cases, it helps us not need many data observations to train on.

Note: It is worthwhile to investigate different pretrained models before you use them. For example, a pretrained model may have learned over many animal images, so it'd be great to classifiy cats, but perhaps not different cars or other vehicles.

In this problem, we will create a doggy door model that will identify when Bo, the U.S. First Dog of 2009-2017, is at the "door" so that we can let her in, while *not* opening the door for other dogs. The realistic problem of this that we don't have many pictures of Bo, so we have to work with only 30 pictures of her. Making a model from scratch with only this much data of interest will probably give us pretty bad results.

Model Modification/Creation

Prepare the Pretrained Model

Here, we'll be using an ImageNet model that has learned to classify many different animals through the animal features that it's learned on. To be able to add on layers that will learn on *our* specific dog data, we'll remove the last layer of the pretrained model, then **freeze** the model so that it doesn't get retrained on the pretrained layers. We only want our new layers to train!

```
# Download the Pretrained Model
pretrained_model = keras.applications.VGG16(
    weights = "imagenet",  # Load the pre-trained weights on ImageNet
    input_shape = (224, 224, 3),
    include_top = False)

# Freeze the model
pretrained_model.trainable = False

# View the model's architecture
pretrained_model.summary()
```

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792
 block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

=================================================================
Total params: 14714688 (56.13 MB)
Trainable params: 0 (0.00 Byte)
Non-trainable params: 14714688 (56.13 MB)
_________________________________________________________________
```

Keras is a bit lower-code (less under-the-hood than PyTorch), and does a great job of showing you what's going on overall. We see that this model has 14,714,688 non-trainable parameters since we froze the model (they won't get retrained).

Add our New Layers

We'll add 2 new model layers that we will train on our doggy door data:

- A **2D average pooling** layer
- A **Dense** layer that makes the decision if the dog is Bo or not

```
# Prepare our model to take in our data:
### RGB images (3 channels) of 224*224 pixels
inputs = keras.Input(shape = (224, 224, 3))

# Set training to False to not train our base model
x = pretrained_model(inputs, training = False)

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

# A Dense classifier with a single unit (binary classification)
outputs = keras.layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

# View the model's architecture
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, 224, 224, 3)]     0

 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 global_average_pooling2d (  (None, 512)               0
 GlobalAveragePooling2D)

 dense (Dense)               (None, 1)                 513

=================================================================
Total params: 14715201 (56.13 MB)
Trainable params: 513 (2.00 KB)
Non-trainable params: 14714688 (56.13 MB)
_________________________________________________________________
```

Here we see the separation of our pretrained (nontrainable) VGG16 model and our trainable layers, all with the correct input and output shapes (dimensions).

Compile the Model

Finally, we compile the model (different from PyTorch, which is usually not compiled this way) to get it ready to fit to our data.

```
# Use the binary crossentropy loss function
# and binary accuracy since this is a binary classification problem
model.compile(loss = keras.losses.BinaryCrossentropy(from_logits = True),
              metrics = [keras.metrics.BinaryAccuracy()])
```

Prepare the Model Data

Data Augmentation

For our model to learn many different situations of dog images (angle adjustments, zooming in, facing different directions, etc.), and to generally combat the problem of having a small dataset to work with, we can **augment** our image data so that our model can learn different variations of the dogs we are training it on (in a sense has "new" images to learn on). Keras makes it very easy to augment this data with their ImageDataGenerator class, which does a variety of transformations to our data:

```
# Making the data generator for the training data
datagen_train = 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 do not flip vertically
)

# We don't augment our validation data since it's not being learned in our model
datagen_valid = ImageDataGenerator(samplewise_center = True)
```

Load in Data

Let's load in our data using Keras' `flow_from_directory` method, which also sizes our images to the input sizes we want:

```
# Load and iterate through our training data
train_it = datagen_train.flow_from_directory(
    "data/presidential_doggy_door/train/", # file directory of where our training data is
    target_size = (224, 224), # 224*224 image pixel sizes
    color_mode = "rgb", # images are in color, having 3 channels (red, green, blue)
    class_mode = "binary", # Bo or not Bo
    batch_size = 8 # batch sizes for training
)

# Load and iterate through our validation data
valid_it = datagen_valid.flow_from_directory(
    "data/presidential_doggy_door/valid/", # file directory of where our validation data is
    target_size = (224, 224),
    color_mode = "rgb",
    class_mode = "binary",
    batch_size = 8
)
```

```
Found 139 images belonging to 2 classes.
Found 30 images belonging to 2 classes.
```

Model Training

```
# Easy training with Keras
model.fit(train_it, # Training image iterator (augmented data)
          steps_per_epoch = 12, # Have to specify this due to using a data generator
          validation_data = valid_it, # Validation image iterator (non-augmented data)
          validation_steps = 4,
          epochs = 20)
```

```
Epoch 1/20
12/12 [==============================] - 89s 7s/step - loss: 0.6599 - binary_accuracy: 0.8022 - val_loss: 1.1193 - val_binary_accuracy:
Epoch 2/20
12/12 [==============================] - 73s 6s/step - loss: 0.4585 - binary_accuracy: 0.8750 - val_loss: 0.5873 - val_binary_accuracy:
Epoch 3/20
12/12 [==============================] - 69s 6s/step - loss: 0.2158 - binary_accuracy: 0.9121 - val_loss: 0.6512 - val_binary_accuracy:
Epoch 4/20
12/12 [==============================] - 73s 6s/step - loss: 0.2537 - binary_accuracy: 0.9341 - val_loss: 0.4018 - val_binary_accuracy:
Epoch 5/20
12/12 [==============================] - 69s 6s/step - loss: 0.1721 - binary_accuracy: 0.9341 - val_loss: 0.3934 - val_binary_accuracy:
Epoch 6/20
12/12 [==============================] - 70s 6s/step - loss: 0.0183 - binary_accuracy: 1.0000 - val_loss: 0.3114 - val_binary_accuracy:
Epoch 7/20
12/12 [==============================] - 76s 7s/step - loss: 0.0429 - binary_accuracy: 0.9688 - val_loss: 0.3114 - val_binary_accuracy:
Epoch 8/20
12/12 [==============================] - 75s 6s/step - loss: 0.0394 - binary_accuracy: 1.0000 - val_loss: 0.2403 - val_binary_accuracy:
Epoch 9/20
12/12 [==============================] - 71s 6s/step - loss: 0.0275 - binary_accuracy: 0.9890 - val_loss: 0.1944 - val_binary_accuracy:
Epoch 10/20
12/12 [==============================] - 69s 6s/step - loss: 0.0290 - binary_accuracy: 0.9792 - val_loss: 0.2202 - val_binary_accuracy:
Epoch 11/20
12/12 [==============================] - 73s 6s/step - loss: 0.0368 - binary_accuracy: 0.9890 - val_loss: 0.1755 - val_binary_accuracy:
Epoch 12/20
12/12 [==============================] - 73s 6s/step - loss: 0.0323 - binary_accuracy: 0.9792 - val_loss: 0.2378 - val_binary_accuracy:
Epoch 13/20
12/12 [==============================] - 68s 6s/step - loss: 0.0095 - binary_accuracy: 1.0000 - val_loss: 0.1936 - val_binary_accuracy:
Epoch 14/20
12/12 [==============================] - 71s 6s/step - loss: 0.0015 - binary_accuracy: 1.0000 - val_loss: 0.1783 - val_binary_accuracy:
Epoch 15/20
12/12 [==============================] - 71s 6s/step - loss: 0.0096 - binary_accuracy: 1.0000 - val_loss: 0.1261 - val_binary_accuracy:
Epoch 16/20
12/12 [==============================] - 73s 6s/step - loss: 0.0014 - binary_accuracy: 1.0000 - val_loss: 0.1218 - val_binary_accuracy:
Epoch 17/20
12/12 [==============================] - 67s 6s/step - loss: 0.0020 - binary_accuracy: 1.0000 - val_loss: 0.1301 - val_binary_accuracy:
Epoch 18/20
12/12 [==============================] - 76s 6s/step - loss: 0.0020 - binary_accuracy: 1.0000 - val_loss: 0.1927 - val_binary_accuracy:
Epoch 19/20
12/12 [==============================] - 70s 6s/step - loss: 0.0078 - binary_accuracy: 0.9890 - val_loss: 0.1526 - val_binary_accuracy:
Epoch 20/20
12/12 [==============================] - 73s 6s/step - loss: 0.0137 - binary_accuracy: 1.0000 - val_loss: 0.0650 - val_binary_accuracy:
<keras.src.callbacks.History at 0x7c7cad366200>
```

High accuracy in training and validation! But we can still do better on our validation data through fine-tuning the model.

Fine-Tuning our Model

This is different from hyperparameter tuning since we have an already trained complete model. To fine tune, we'll unfreeze all of the model and will "retrain" it with small steps by using a small learning rate to adjust/improve the model by a bit. We can do this since *all* model layers are already fully trained and won't update largely on our added layers (not destroying any pretrained features), we'll simply make small updates!

```
# Unfreeze the pretrained model
pretrained_model.trainable = True

# Recompile the model after we make changes to the `trainable` attribute
# of any inner layer, so that your changes are taken into account
model.compile(optimizer = keras.optimizers.RMSprop(learning_rate =
                                                   0.00001), # Very small learning rate
              loss = keras.losses.BinaryCrossentropy(from_logits = True),
              metrics = [keras.metrics.BinaryAccuracy()])


# Refit the model with unfrozen layers
model.fit(train_it,
          steps_per_epoch = 12,
          validation_data = valid_it,
          validation_steps=4,
          epochs = 10)
```

```
Epoch 1/10
12/12 [==============================] - 209s 17s/step - loss: 0.0864 - binary_accuracy: 0.9560 - val_loss: 0.0247 - val_binary_accurac
Epoch 2/10
12/12 [==============================] - 203s 17s/step - loss: 6.2584e-04 - binary_accuracy: 1.0000 - val_loss: 0.0193 - val_binary_acc
Epoch 3/10
12/12 [==============================] - 246s 21s/step - loss: 7.4603e-04 - binary_accuracy: 1.0000 - val_loss: 0.0135 - val_binary_acc
Epoch 4/10
12/12 [==============================] - 240s 20s/step - loss: 3.8194e-04 - binary_accuracy: 1.0000 - val_loss: 0.0126 - val_binary_acc
Epoch 5/10
12/12 [==============================] - 220s 19s/step - loss: 1.3165e-04 - binary_accuracy: 1.0000 - val_loss: 0.0132 - val_binary_acc
Epoch 6/10
12/12 [==============================] - 215s 18s/step - loss: 1.1867e-04 - binary_accuracy: 1.0000 - val_loss: 0.0144 - val_binary_acc
Epoch 7/10
12/12 [==============================] - 204s 17s/step - loss: 1.4127e-04 - binary_accuracy: 1.0000 - val_loss: 0.0096 - val_binary_acc
Epoch 8/10
12/12 [==============================] - 200s 17s/step - loss: 2.1009e-05 - binary_accuracy: 1.0000 - val_loss: 0.0107 - val_binary_acc
Epoch 9/10
12/12 [==============================] - 211s 18s/step - loss: 1.5392e-04 - binary_accuracy: 1.0000 - val_loss: 0.0079 - val_binary_acc
Epoch 10/10
12/12 [==============================] - 206s 17s/step - loss: 1.6987e-05 - binary_accuracy: 1.0000 - val_loss: 0.0091 - val_binary_acc
<keras.src.callbacks.History at 0x7c7c93cd2410>
```

Now our model is well trained!

Results

To make our predictions, we have to preprocess our data. We will use our validation holdout set as our testing data since we don't have more images.

```
# To display our images
def show_image(image_path):
    image = mpimg.imread(image_path)
    plt.imshow(image)

# To make predictions on an image
def make_predictions(image_path):
    show_image(image_path)
    image = image_utils.load_img(image_path, target_size = (224, 224))
    image = image_utils.img_to_array(image)
    image = image.reshape(1, 224, 224, 3) # reshape 1 image of size 224*224 with 3 color channels
    image = preprocess_input(image)
    preds = model.predict(image)
    return preds
```

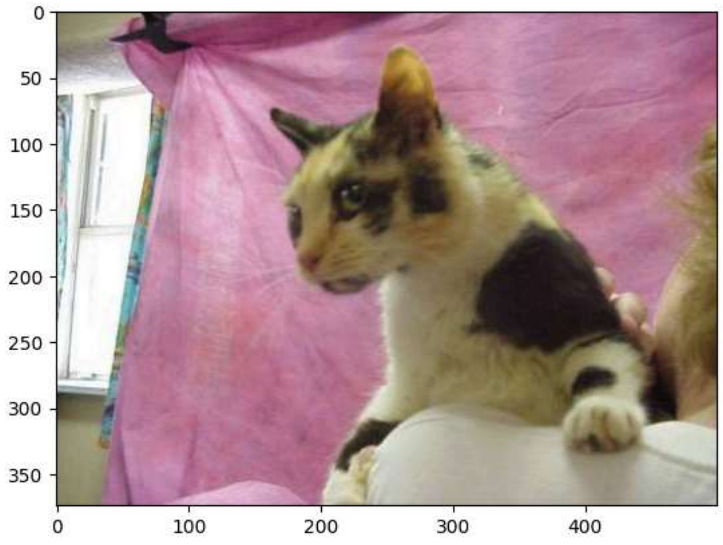
As examples, we can predict on 2 images where one is of Bo and not Bo:

```
make_predictions('data/presidential_doggy_door/valid/bo/bo_20.jpg')
```

```
1/1 [==============================] - 1s 806ms/step
array([[-17.981548]], dtype=float32)
```

```
make_predictions('data/presidential_doggy_door/valid/not_bo/121.jpg')
```

```
1/1 [==============================] - 1s 502ms/step
array([[22.399208]], dtype=float32)
```



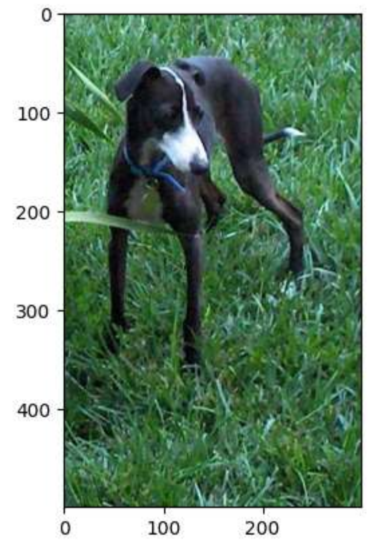
Make our Doggy Door

Code to decide if our given images are Bo or not

```
def presidential_doggy_door(image_path):
    preds = make_predictions(image_path)
    if preds[0] < 0:
        print("It's Bo! Let him in!")
    else:
        print("That's not Bo! Stay out!")
```

```
presidential_doggy_door('data/presidential_doggy_door/valid/not_bo/131.jpg')
```

```
1/1 [==============================] - 1s 502ms/step
That's not Bo! Stay out!
```



```
presidential_doggy_door('data/presidential_doggy_door/valid/bo/bo_29.jpg')
```

```
1/1 [==============================] - 0s 500ms/step
It's Bo! Let him in!
```

And that's how using CNNs with transfer learning can get you great results in image classification with less resources!