Header

# 5b. Transfer Learning

So far, we have trained accurate models on large datasets, and also downloaded a pre-trained model that we used with no training necessary. But what if we cannot find a pre-trained model that does exactly what you need, and what if we do not have a sufficiently large dataset to train a model from scratch? In this case, there is a very helpful technique we can use called transfer learning.

With transfer learning, we take a pre-trained model and retrain it on a task that has some overlap with the original training task. A good analogy for this is an artist who is skilled in one medium, such as painting, who wants to learn to practice in another medium, such as charcoal drawing. We can imagine that the skills they learned while painting would be very valuable in learning how to draw with charcoal.

As an example in deep learning, say we have a pre-trained model that is very good at recognizing different types of cars, and we want to train a model to recognize types of motorcycles. A lot of the learnings of the car model would likely be very useful, for instance the ability to recognize headlights and wheels.

Transfer learning is especially powerful when we do not have a large and varied dataset. In this case, a model trained from scratch would likely memorize the training data quickly, but not be able to generalize well to new data. With transfer learning, you can increase your chances of training an accurate and robust model on a small dataset.

## 5b.1 Objectives

- Prepare a pretrained model for transfer learning
- Perform transfer learning with your own small dataset on a pretrained model
- Further fine tune the model for even better performance

In [5]:

```
import torch
import torch.nn as nn
from torch.optim import Adam
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms.v2 as transforms
import torchvision.io as tv_io

#which is used to search for files and directories using wildcard patterns (like *.jpg, *.csv, data/*.txt, etc.).
import glob

import json
from PIL import Image

import utils

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.cuda.is_available()
```

```
Out[5]: True
```

## 5b.2 A Personalized Doggy Door

In our last exercise, we used a pre-trained ImageNet model to let in all dogs, but keep out other animals. In this exercise, we would like to create a doggy door that only lets in a particular dog. In this case, we will make an automatic doggy door for a dog named Bo, the United States First Dog between 2009 and 2017. There are more pictures of Bo in the `data/presidential_doggy_door` folder.

No description has been provided for this image

The challenge is that the pre-trained model was not trained to recognize this specific dog, and, we only have 30 pictures of Bo. If we tried to train a model from scratch using those 30 pictures we would experience overfitting and poor generalization. However, if we start with a pre-trained model that is adept at detecting dogs, we can leverage that learning to gain a generalized understanding of Bo using our smaller dataset. We can use transfer learning to solve this challenge.

### 5b.2.1 Downloading the Pretrained Model

The ImageNet torchvision.models are often good choices for computer vision transfer learning, as they have learned to classify various different types of images. In doing this, they have learned to detect many different types of features that could be valuable in image recognition. Because ImageNet models have learned to detect animals, including dogs, it is especially well suited for this transfer learning task of detecting Bo.

Let us start by downloading the pre-trained model.

In [6]:
```
from torchvision.models import vgg16
from torchvision.models import VGG16_Weights

# load the VGG16 network *pre-trained* on the ImageNet dataset
weights = VGG16_Weights.DEFAULT
vgg_model = vgg16(weights=weights)
```

As we are downloading, there is going to be an important difference. The last layer of an ImageNet model is a dense layer of 1000 units, representing the 1000 possible classes in the dataset. In our case, we want it to make a different classification: is this Bo or not? We will add new layers to specifically recognize Bo.

In [7]:
```
vgg_model.to(device)
```

```
Out[7]: VGG(
          (features): Sequential(
            (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): ReLU(inplace=True)
            (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (3): ReLU(inplace=True)
            (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
            (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (6): ReLU(inplace=True)
            (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (8): ReLU(inplace=True)
            (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
            (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (11): ReLU(inplace=True)
            (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (13): ReLU(inplace=True)
            (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (15): ReLU(inplace=True)
            (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
            (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (18): ReLU(inplace=True)
            (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (20): ReLU(inplace=True)
            (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (22): ReLU(inplace=True)
            (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
            (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (25): ReLU(inplace=True)
            (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (27): ReLU(inplace=True)
            (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (29): ReLU(inplace=True)
            (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
          )
          (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
          (classifier): Sequential(
            (0): Linear(in_features=25088, out_features=4096, bias=True)
            (1): ReLU(inplace=True)
            (2): Dropout(p=0.5, inplace=False)
            (3): Linear(in_features=4096, out_features=4096, bias=True)
```

```
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

### 5b.2.2 Freezing the Base Model

Before we add our new layers onto the pre-trained model, let's take an important step: freezing the model's pre-trained layers. This means that when we train, we will not update the base layers from the pre-trained model. Instead we will only update the new layers that we add on the end for our new classification. We freeze the initial layers because we want to retain the learning achieved from training on the ImageNet dataset. If they were unfrozen at this stage, we would likely destroy this valuable information. There will be an option to unfreeze and train these layers later, in a process called fine-tuning.

Freezing the base layers is as simple as setting requires_grad_ on the model to `False`.

In [8]:
```
vgg_model.requires_grad_(False)
print("VGG16 Frozen")
```

```
VGG16 Frozen
```

### 5b.2.3 Adding New Layers

We can now add the new trainable layers to the pre-trained model. They will take the features from the pre-trained layers and turn them into predictions on the new dataset. We will add two layers to the model. In a previous lesson, we created our own custom module. A transfer learning module works in the exact same way. We can use is a layer in a Sequential Model.

Then, we'll add a `Linear` layer connecting all `1000` of VGG16's outputs to `1` neuron.

In [9]:
```
N_CLASSES = 1

my_model = nn.Sequential(
    vgg_model,
    nn.Linear(1000, N_CLASSES)
)
```

```
my_model.to(device)
```

Out[9]:

```
Sequential(
  (0): VGG(
    (features): Sequential(
      (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace=True)
      (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): ReLU(inplace=True)
      (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (6): ReLU(inplace=True)
      (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (8): ReLU(inplace=True)
      (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (11): ReLU(inplace=True)
      (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (13): ReLU(inplace=True)
      (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (15): ReLU(inplace=True)
      (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (18): ReLU(inplace=True)
      (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (20): ReLU(inplace=True)
      (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (22): ReLU(inplace=True)
      (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (25): ReLU(inplace=True)
      (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (27): ReLU(inplace=True)
      (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (29): ReLU(inplace=True)
      (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
    (classifier): Sequential(
      (0): Linear(in_features=25088, out_features=4096, bias=True)
      (1): ReLU(inplace=True)
      (2): Dropout(p=0.5, inplace=False)
      (3): Linear(in_features=4096, out_features=4096, bias=True)
      (4): ReLU(inplace=True)
      (5): Dropout(p=0.5, inplace=False)
      (6): Linear(in_features=4096, out_features=1000, bias=True)
    )
  )
  (1): Linear(in_features=1000, out_features=1, bias=True)
)
```

If we want to verify that the VGG layers are frozen, we can loop through the model parameters.

In [10]:

```
for idx, param in enumerate(my_model.parameters()):
    print(idx, param.requires_grad)
```

```
0 False
1 False
2 False
3 False
4 False
5 False
6 False
7 False
8 False
9 False
10 False
11 False
12 False
13 False
14 False
15 False
16 False
17 False
18 False
19 False
20 False
21 False
22 False
23 False
24 False
25 False
26 False
27 False
28 False
29 False
30 False
31 False
32 True
33 True
```

If we did want to make the VGG layers trainable, we could take `vgg_model` and set `requires_grad_` to `True`.

In [11]:

```
vgg_model.requires_grad_(True)
print("VGG16 Unfrozen")
```

```
for idx, param in enumerate(my_model.parameters()):
    print(idx, param.requires_grad)
```

```
VGG16 Unfrozen
0 True
1 True
2 True
3 True
4 True
5 True
6 True
7 True
8 True
9 True
10 True
11 True
12 True
13 True
14 True
15 True
16 True
17 True
18 True
19 True
20 True
21 True
22 True
23 True
24 True
25 True
26 True
27 True
28 True
29 True
30 True
31 True
32 True
33 True
```

But for now, we'd only like to train our new layers, so we will turn training off.

In [12]:

```
vgg_model.requires_grad_(False)
print("VGG16 Frozen")
```

```
VGG16 Frozen
```

### 5.2.4 Compiling the Model

As with our previous exercises, we need to compile the model with loss and metrics options. We have to make some different choices here. In previous cases we had many categories in our classification problem. As a result, we picked categorical crossentropy for the calculation of our loss. In this case we only have a binary classification problem (Bo or not Bo), and so we will use binary crossentropy. Further detail about the differences between the two can found here. We will also use binary accuracy instead of traditional accuracy.

By setting `from_logits=True` we inform the loss function that the output values are not normalized (e.g. with softmax).

In [13]:

```
loss_function = nn.BCEWithLogitsLoss()
optimizer = Adam(my_model.parameters())
my_model = my_model.to(device)
```

## 5b.3 Data Augmentation

Just like in the previous lessons, we'll create a custom Dataset to read in picutes of Bo (and not Bo). First, we'll grab the list of preprocessing transforms from the VGG `weights`.

In [14]:

```
pre_trans = weights.transforms()
```

### 5b.3.1 The Dataset

Rather than read from a DataFrame like in previous lessons, we will read image files directly and infer the `label` based on the filepath.

In [15]:

```
DATA_LABELS = ["bo", "not_bo"]

class MyDataset(Dataset):
    def __init__(self, data_dir):
        self.imgs = []
        self.labels = []

        for l_idx, label in enumerate(DATA_LABELS):
            data_paths = glob.glob(data_dir + label + '/*.jpg', recursive=True)
            for path in data_paths:
                img = Image.open(path)
                self.imgs.append(pre_trans(img).to(device))
                self.labels.append(torch.tensor(l_idx).to(device).float())


    def __getitem__(self, idx):
        img = self.imgs[idx]
        label = self.labels[idx]
        return img, label

    def __len__(self):
        return len(self.imgs)
```

## 5b.3.2 The DataLoaders

Now that we have our custom Dataset class, let's create our DataLoaders.

In [16]:

```
n = 32

train_path = "data/presidential_doggy_door/train/"
train_data = MyDataset(train_path)
train_loader = DataLoader(train_data, batch_size=n, shuffle=True)
train_N = len(train_loader.dataset)

valid_path = "data/presidential_doggy_door/valid/"
valid_data = MyDataset(valid_path)
valid_loader = DataLoader(valid_data, batch_size=n)
valid_N = len(valid_loader.dataset)
```

## 5b.3.3 Data Augmentation

Let's apply some data augmentation so the model can have a better chance at recognizing Bo. This time, we have color images, so we can use ColorJitter to full effect.

In [17]:

```
IMG_WIDTH, IMG_HEIGHT = (224, 224)

random_trans = transforms.Compose([
    transforms.RandomRotation(25),
    transforms.RandomResizedCrop((IMG_WIDTH, IMG_HEIGHT), scale=(.8, 1), ratio=(1, 1)),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=.2, contrast=.2, saturation=.2, hue=.2)
])
```

## 5b.4 The Training Loop

We will use most of the same training loop as before but with a few slight differences. First, our `get_batch_accuracy` function will be different because of using Binary Cross Entropy as our loss function. We could run the output through the sigmoid function, but we can be more efficient by being mathematically observant.

When our model `output` is greater than `0`, running it through the sigmoid function would be closer to `1`. When the model `output` is less than `0`, running it through the sigmoid function would be closer to `0`. Therefore, we only need to check if the model output is greater than (gt) `0` to see which class our prediction leans towards.

In [18]:

```
def get_batch_accuracy(output, y, N):
    zero_tensor = torch.tensor([0]).to(device)
    pred = torch.gt(output, zero_tensor)
    correct = pred.eq(y.view_as(pred)).sum().item()
    return correct / N
```

We also section to print the last set of gradients to show that only our newly added layers are learning.

In [19]:

```
def train(model, check_grad=False):
    loss = 0
    accuracy = 0

    model.train()
    for x, y in train_loader:
        output = torch.squeeze(model(random_trans(x)))
        optimizer.zero_grad()
        batch_loss = loss_function(output, y)
        batch_loss.backward()
        optimizer.step()

        loss += batch_loss.item()
        accuracy += get_batch_accuracy(output, y, train_N)
    if check_grad:
        print('Last Gradient:')
        for param in model.parameters():
            print(param.grad)
    print('Train - Loss: {:.4f} Accuracy: {:.4f}'.format(loss, accuracy))
```

Uncomment the below to see a sample of the model's gradients. Because VGG16 ends in 1000 neurons, there are 1000 weights connected to the single neuron in the next layer. Many numbers will be printed!

In [ ]:

```
#train(my_model, check_grad=True)
```

The `validate` function mostly remains the same:

In [20]:

```
def validate(model):
    loss = 0
    accuracy = 0

    model.eval()
    with torch.no_grad():
        for x, y in valid_loader:
            output = torch.squeeze(model(x))

            loss += loss_function(output, y.float()).item()
            accuracy += get_batch_accuracy(output, y, valid_N)
    print('Valid - Loss: {:.4f} Accuracy: {:.4f}'.format(loss, accuracy))
```

Moment of truth: can the model learn to recognize Bo?

In [21]:

```
epochs = 10

for epoch in range(epochs):
    print('Epoch: {}'.format(epoch))
    train(my_model, check_grad=False)
    validate(my_model)
```

```
Epoch: 0
Train - Loss: 2.3739 Accuracy: 0.7842
Valid - Loss: 0.8479 Accuracy: 0.6667
Epoch: 1
Train - Loss: 1.7771 Accuracy: 0.8633
Valid - Loss: 0.6187 Accuracy: 0.6333
Epoch: 2
Train - Loss: 1.3828 Accuracy: 0.8633
Valid - Loss: 0.3885 Accuracy: 0.8000
Epoch: 3
Train - Loss: 1.5046 Accuracy: 0.8633
Valid - Loss: 0.2355 Accuracy: 0.9000
Epoch: 4
Train - Loss: 1.2691 Accuracy: 0.8633
Valid - Loss: 0.1879 Accuracy: 0.9000
Epoch: 5
Train - Loss: 1.0705 Accuracy: 0.9065
Valid - Loss: 0.1651 Accuracy: 0.9667
Epoch: 6
Train - Loss: 1.0787 Accuracy: 0.8993
Valid - Loss: 0.1451 Accuracy: 0.9667
Epoch: 7
Train - Loss: 0.8949 Accuracy: 0.9353
Valid - Loss: 0.1223 Accuracy: 0.9667
Epoch: 8
Train - Loss: 1.1167 Accuracy: 0.8993
Valid - Loss: 0.1198 Accuracy: 0.9667
Epoch: 9
Train - Loss: 0.9635 Accuracy: 0.9209
Valid - Loss: 0.1239 Accuracy: 0.9667
```

# Discussion of Results

Both the training and validation accuracy should be quite high. This is a pretty awesome result! We were able to train on a small dataset, but because of the knowledge transferred from the ImageNet model, it was able to achieve high accuracy and generalize well. This means it has a very good sense of Bo and pets who are not Bo.

If you saw some fluctuation in the validation accuracy, that is okay too. We have a technique for improving our model in the next section.

## Fine-Tuning the Model

Now that the new layers of the model are trained, we have the option to apply a final trick to improve the model, called fine-tuning. To do this we unfreeze the entire model, and train it again with a very small learning rate. This will cause the base pre-trained layers to take very small steps and adjust slightly, improving the model by a small amount. VGG16 is a relatively large model,so the small learning rate will also prevent overfitting.

Note that it is important to only do this step after the model with frozen layers has been fully trained. The untrained linear layer that we added to the model earlier was randomly initialized. This means it needed to be updated quite a lot to correctly classify the images. Through the process of backpropagation, large initial updates in the last layers would have caused potentially large updates in the pre-trained layers as well. These updates would have destroyed those important pre-trained features. However, now that those final layers are trained and have converged, any updates to the model as a whole will be much smaller (especially with a very small learning rate) and will not destroy the features of the earlier layers.

Let's try unfreezing the pre-trained layers, and then fine tuning the model:

In [22]:

```
# Unfreeze the base model
vgg_model.requires_grad_(True)
optimizer = Adam(my_model.parameters(), lr=.000001)
```

In [23]:

```
epochs = 2

for epoch in range(epochs):
    print('Epoch: {}'.format(epoch))
    train(my_model, check_grad=False)
    validate(my_model)
```

```
Epoch: 0
Train - Loss: 0.9352 Accuracy: 0.9137
Valid - Loss: 0.1201 Accuracy: 0.9667
Epoch: 1
Train - Loss: 0.5911 Accuracy: 0.9640
Valid - Loss: 0.1151 Accuracy: 0.9667
```

In this case, we'll only train for a few `epochs`. Because VGG16 is such a large model, it can overfit when it trains for too long on this dataset.

## Examining the Predictions

Now that we have a well-trained model, it is time to create our doggy door for Bo! We can start by looking at the predictions that come from the model. We will preprocess the image in the same way we did for our last doggy door.

In [24]:

```python
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

def show_image(image_path):
    image = mpimg.imread(image_path)
    plt.imshow(image)
```

In [25]:

```python
def make_prediction(file_path):
    show_image(file_path)
    image = Image.open(file_path)
    image = pre_trans(image).to(device)
    image = image.unsqueeze(0)
    output = my_model(image)
    prediction = output.item()
    return prediction
```

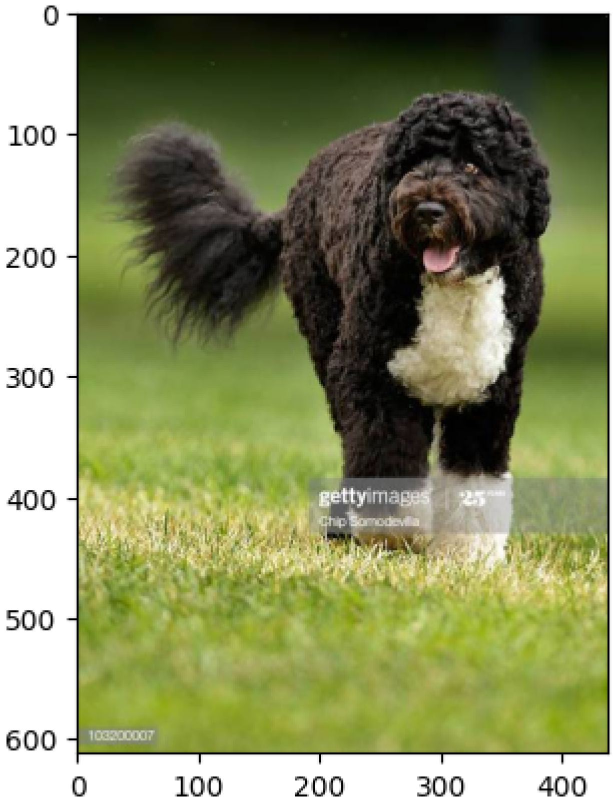
Try this out on a couple images to see the predictions:

In [26]:

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

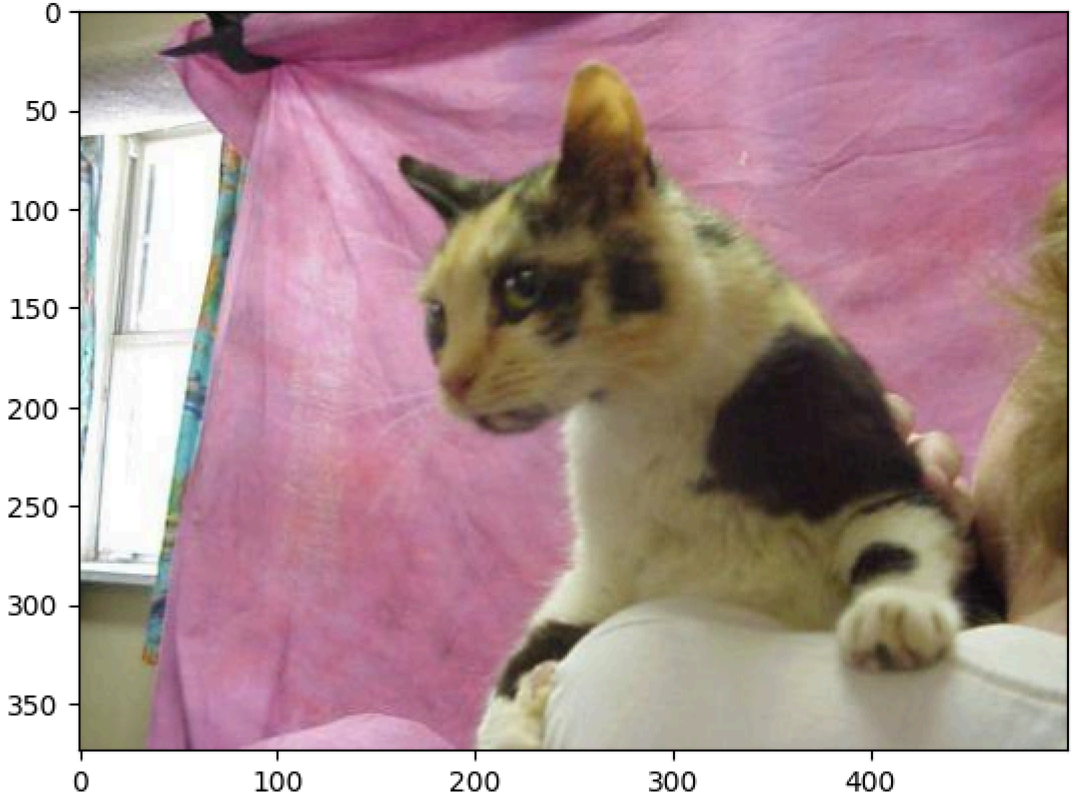
Out[26]:

```
-3.801337957382202
```



```
In [27]: make_prediction('data/presidential_doggy_door/valid/not_bo/121.jpg')
```

```
Out[27]: 9.247227668762207
```



It looks like a negative number prediction means that it is Bo and a positive number prediction means it is something else. We can use this information to have our doggy door only let Bo in!

## Exercise: Bo's Doggy Door

Fill in the following code to implement Bo's doggy door:

```
In [29]: def presidential_doggy_door(image_path):
             pred = make_prediction(image_path)
             if pred<0:
                 print("It's Bo! Let him in!")
```

```
    else:
        print("That's not Bo! Stay out!")
```

## Solution

Click on the '...' below to see the solution.

In [ ]:

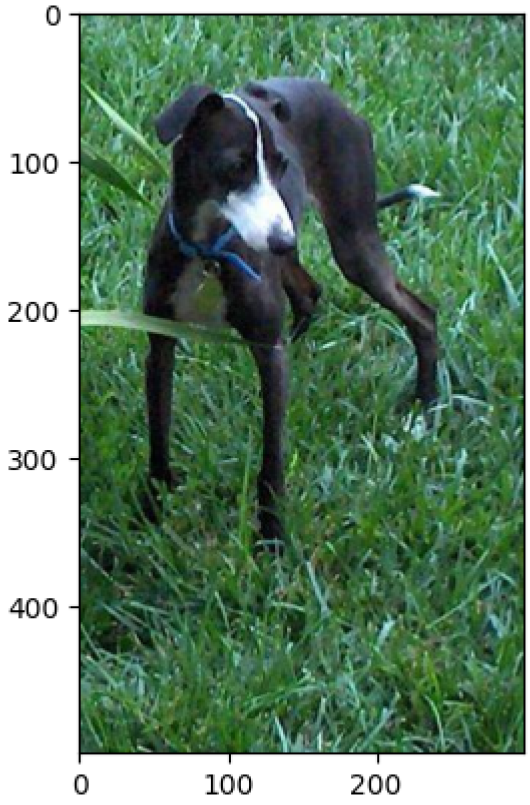
```
# SOLUTION
def presidential_doggy_door(image_path):
    pred = make_prediction(image_path)
    if pred < 0:
        print("It's Bo! Let him in!")
    else:
        print("That's not Bo! Stay out!")
```

Let's try it out!

In [30]:

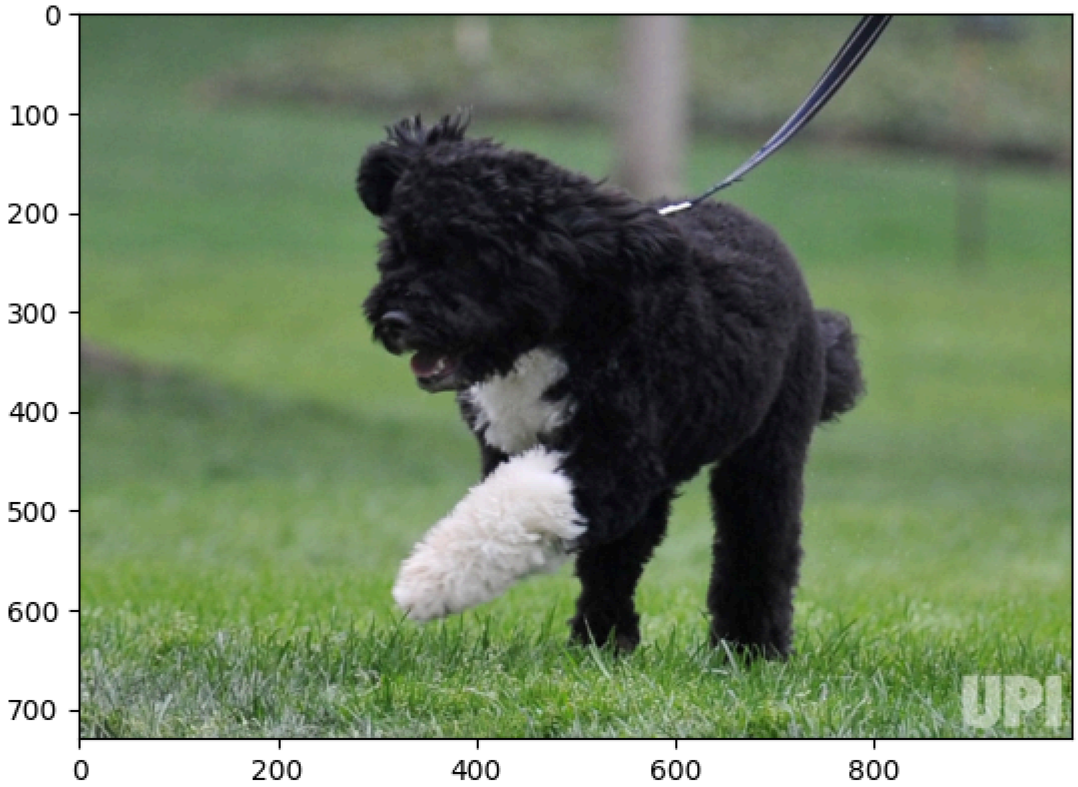
```
presidential_doggy_door('data/presidential_doggy_door/valid/not_bo/131.jpg')
```

```
That's not Bo! Stay out!
```



In [31]:
```
presidential_doggy_door('data/presidential_doggy_door/valid/bo/bo_29.jpg')
```

```
It's Bo! Let him in!
```



## Summary

Great work! With transfer learning, you have built a highly accurate model using a very small dataset. This can be an extremely powerful technique, and be the difference between a successful project and one that cannot get off the ground. We hope these techniques can help you out in similar situations in the future!

There is a wealth of helpful resources for transfer learning in the NVIDIA TAO Toolkit.

### Clear the Memory

Before moving on, please execute the following cell to clear up the GPU memory.

In [32]:

```python
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

Out[32]:

```
{'status': 'ok', 'restart': True}
```

## Next

So far, the focus of this workshop has primarily been on image classification. In the next section, in service of giving you a more well-rounded introduction to deep learning, we are going to switch gears and address working with sequential data, which requires a different approach.

Header