

Name: **Krishna Gadam**

Div: **BE9-Q9**

Roll no: **43124**

Title: **Assignment 6: Object detection using Transfer Learning of CNN architectures**

In [1]: *#Importing all the necessary Libraries*

```
from IPython.core.interactiveshell import InteractiveShell
import seaborn as sns
# PyTorch
from torchvision import transforms, datasets, models
import torch
from torch import optim, cuda
from torch.utils.data import DataLoader, sampler
import torch.nn as nn

import warnings
warnings.filterwarnings('ignore', category=FutureWarning)

# Data science tools
import numpy as np
import pandas as pd
import os

# Image manipulations
from PIL import Image
# Useful for examining network
from torchsummary import summary
# Timing utility
from timeit import default_timer as timer

# Visualizations
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['font.size'] = 14

# Printing out all outputs
InteractiveShell.ast_node_interactivity = 'all'
```

```
In [2]: # Location of data
datadir = '/home/wjk68/'
traindir = datadir + 'train/'
validdir = datadir + 'valid/'
testdir = datadir + 'test/'

save_file_name = 'vgg16-transfer-4.pt'
checkpoint_path = 'vgg16-transfer-4.pth'

# Change to fit hardware
batch_size = 128

# Whether to train on a gpu
train_on_gpu = cuda.is_available()
print(f'Train on gpu: {train_on_gpu}')

# Number of gpus
if train_on_gpu:
    gpu_count = cuda.device_count()
    print(f'{gpu_count} gpus detected.')
    if gpu_count > 1:
        multi_gpu = True
    else:
        multi_gpu = False
```

```
Train on gpu: True
2 gpus detected.
```

In [3]: #Looking at the number of images in each category and the size of the images

```

# Empty lists
categories = []
img_categories = []
n_train = []
n_valid = []
n_test = []
hs = []
ws = []

# Iterate through each category
for d in os.listdir(traindir):
    categories.append(d)

    # Number of each image
    train_imgs = os.listdir(traindir + d)
    valid_imgs = os.listdir(validdir + d)
    test_imgs = os.listdir(testdir + d)
    n_train.append(len(train_imgs))
    n_valid.append(len(valid_imgs))
    n_test.append(len(test_imgs))

    # Find stats for train images
    for i in train_imgs:
        img_categories.append(d)
        img = Image.open(traindir + d + '/' + i)
        img_array = np.array(img)
        # Shape
        hs.append(img_array.shape[0])
        ws.append(img_array.shape[1])

# Dataframe of categories
cat_df = pd.DataFrame({'category': categories,
                       'n_train': n_train,
                       'n_valid': n_valid, 'n_test': n_test}).\
    sort_values('category')

# Dataframe of training images
image_df = pd.DataFrame({
    'category': img_categories,
    'height': hs,
    'width': ws
})

cat_df.sort_values('n_train', ascending=False, inplace=True)
cat_df.head()
cat_df.tail()

```

Out[3]:

	category	n_train	n_valid	n_test
4	airplanes	400	200	200
2	motorbikes	398	200	200
0	faces	274	138	109

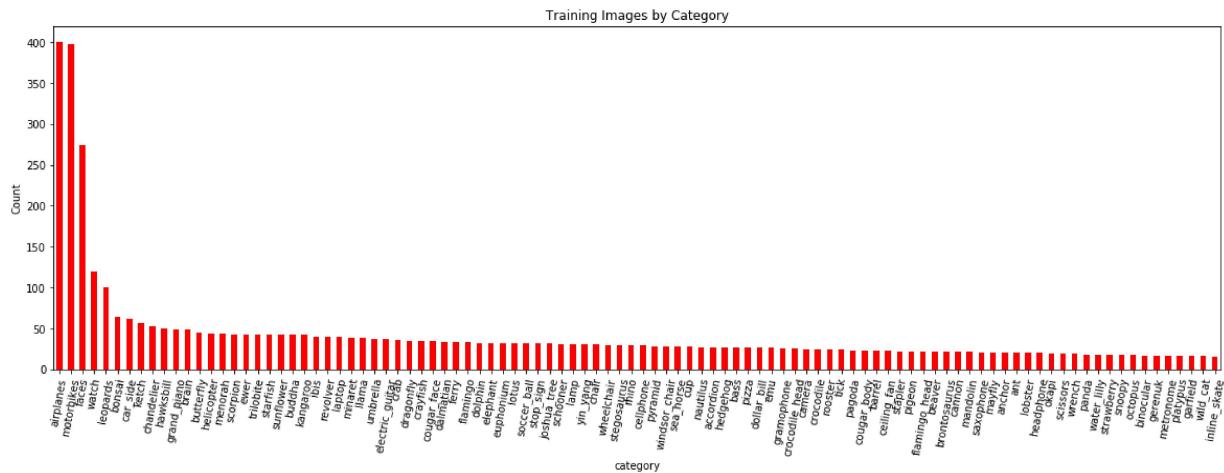
	category	n_train	n_valid	n_test
93	watch	119	60	60
1	leopards	100	50	50

Out[3]:

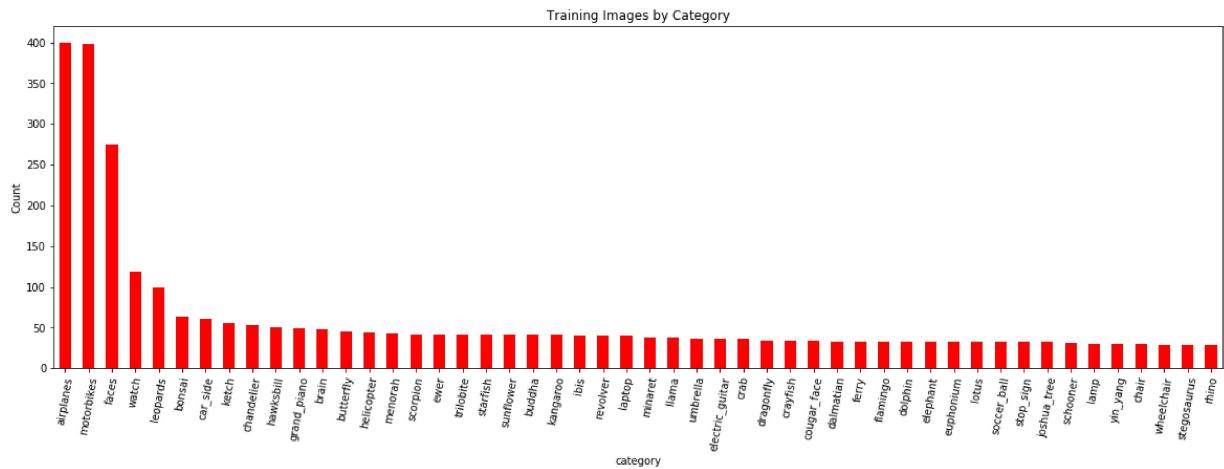
	category	n_train	n_valid	n_test
63	metronome	16	8	8
72	platypus	16	9	9
42	garfield	16	9	9
96	wild_cat	16	9	9
51	inline_skate	15	8	8

In [4]:

```
#Distribution of images
cat_df.set_index('category')[['n_train']].plot.bar(
    color='r', figsize=(20, 6))
plt.xticks(rotation=80)
plt.ylabel('Count')
plt.title('Training Images by Category')
```



```
In [5]: # Only top 50 categories
cat_df.set_index('category').iloc[:50]['n_train'].plot.bar(
    color='r', figsize=(20, 6))
plt.xticks(rotation=80)
plt.ylabel('Count')
plt.title('Training Images by Category')
```

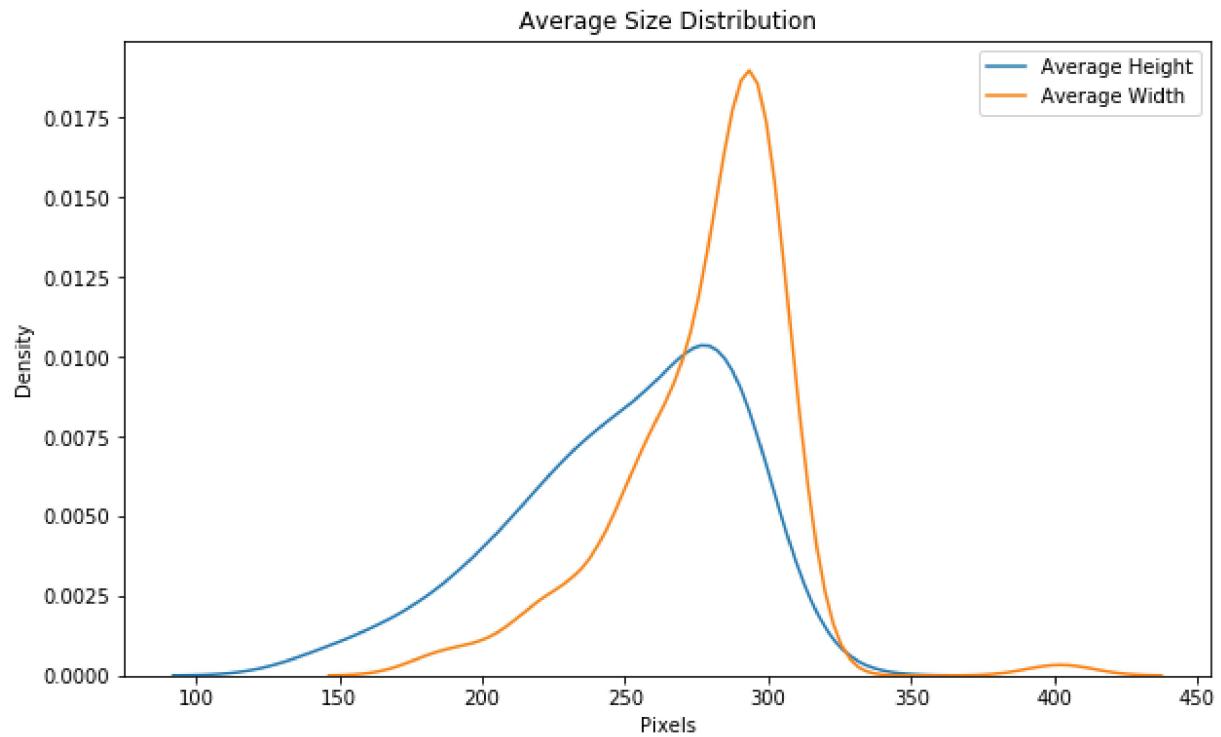


```
In [6]: #Distribution of Images sizes
img_dsc = image_df.groupby('category').describe()
img_dsc.head()
```

Out[6]:

category	height										
	count	mean	std	min	25%	50%	75%	max	count	mean	
accordion	27.0	263.851852	35.769243	199.0	233.00	265.0	300.00	300.0	27.0	280.333333	3
airplanes	400.0	158.455000	30.847397	101.0	141.00	154.0	170.25	494.0	400.0	402.137500	2
anchor	20.0	241.000000	38.608698	170.0	219.75	236.0	264.50	300.0	20.0	291.300000	2
ant	20.0	211.950000	47.137509	103.0	177.00	203.0	236.75	300.0	20.0	298.600000	2
barrel	23.0	284.086957	36.455344	188.0	300.00	300.0	300.00	300.0	23.0	241.869565	4

```
In [7]: plt.figure(figsize=(10, 6))
sns.kdeplot(
    img_dsc['height']['mean'], label='Average Height')
sns.kdeplot(
    img_dsc['width']['mean'], label='Average Width')
plt.xlabel('Pixels')
plt.ylabel('Density')
plt.title('Average Size Distribution')
```



```
In [ ]: #When we use the images in the pre-trained network, we'll have to reshape them to
```

```
In [8]: def imshow(image):
    """Display image"""
    plt.figure(figsize=(6, 6))
    plt.imshow(image)
    plt.axis('off')
    plt.show()

# Example image
x = Image.open(traindir + 'ewer/image_0002.jpg')
np.array(x).shape
imshow(x)
```

Out[8]: (300, 187, 3)



In [10]: # Data Augmentation and Image transformations

```
image_transforms = {
    # Train uses data augmentation
    'train':
        transforms.Compose([
            transforms.RandomResizedCrop(size=256, scale=(0.8, 1.0)),
            transforms.RandomRotation(degrees=15),
            transforms.ColorJitter(),
            transforms.RandomHorizontalFlip(),
            transforms.CenterCrop(size=224), # Image net standards
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406],
                               [0.229, 0.224, 0.225]) # Imagenet standards
        ]),
    # Validation does not use augmentation
    'val':
        transforms.Compose([
            transforms.Resize(size=256),
            transforms.CenterCrop(size=224),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
    # Test does not use augmentation
    'test':
        transforms.Compose([
            transforms.Resize(size=256),
            transforms.CenterCrop(size=224),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
}
}
```

In [11]: def imshow\_tensor(image, ax=None, title=None):

```
    """imshow for Tensor."""

    if ax is None:
        fig, ax = plt.subplots()

    # Set the color channel as the third dimension
    image = image.numpy().transpose((1, 2, 0))

    # Reverse the preprocessing steps
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    image = std * image + mean

    # Clip the image pixel values
    image = np.clip(image, 0, 1)

    ax.imshow(image)
    plt.axis('off')

    return ax, image
```

```
In [12]: ex_img = Image.open('/home/wjk68/train/elephant/image_0024.jpg')
imshow(ex_img)
```

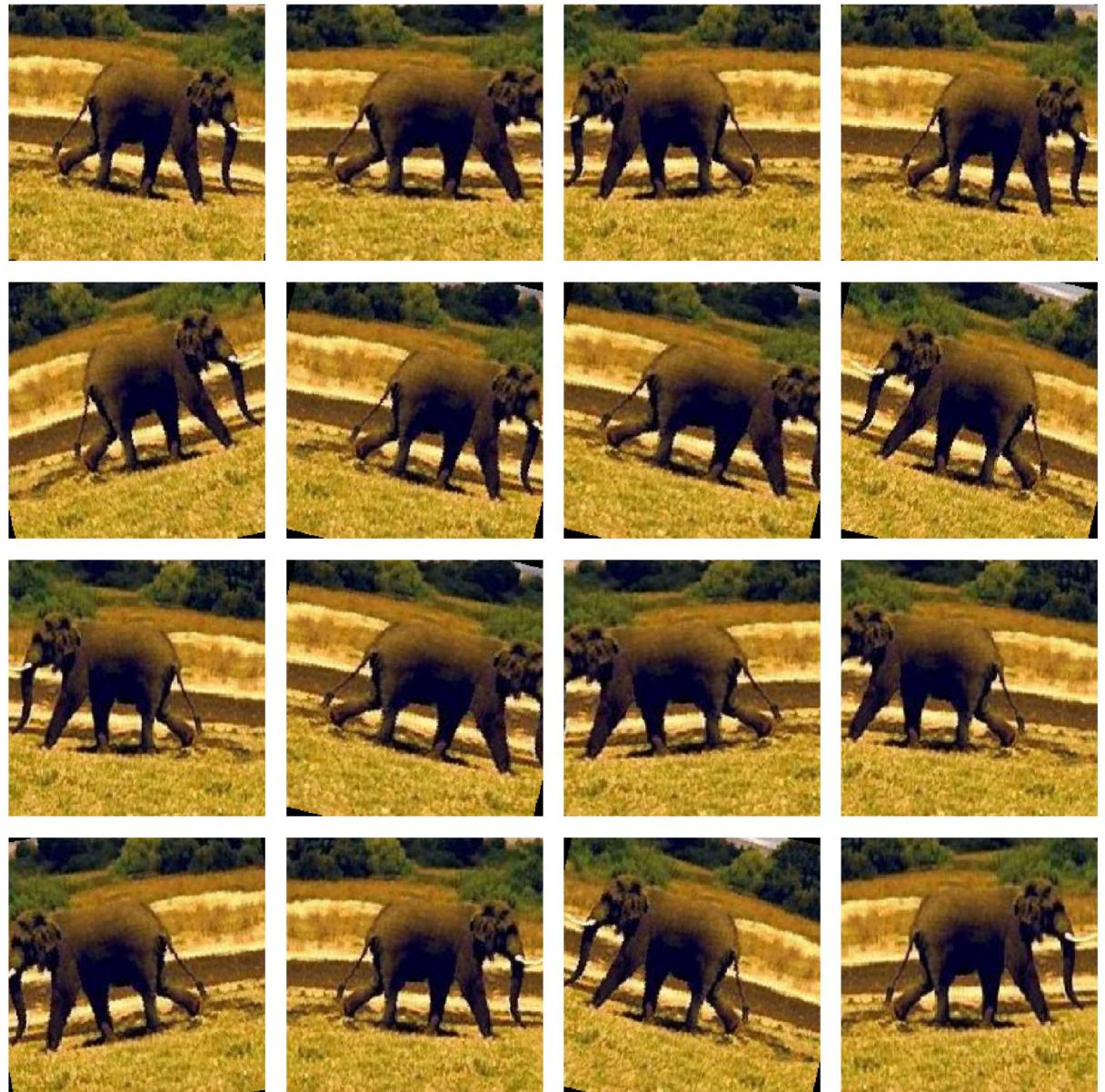


```
In [13]: t = image_transforms['train']
plt.figure(figsize=(24, 24))

for i in range(16):
    ax = plt.subplot(4, 4, i + 1)
    _ = imshow_tensor(t(ex_img), ax=ax)

plt.tight_layout()
```

Out[13]: <Figure size 1728x1728 with 0 Axes>



```
In [15]: #Using dataloaders for speedy iterations
'''This construction avoids the need to load all the data into memory and also wi

# Datasets from each folder
data = {
    'train':
        datasets.ImageFolder(root=traindir, transform=image_transforms['train']),
    'val':
        datasets.ImageFolder(root=validdir, transform=image_transforms['val']),
    'test':
        datasets.ImageFolder(root=testdir, transform=image_transforms['test'])
}

# Dataloader iterators
dataloaders = {
    'train': DataLoader(data['train'], batch_size=batch_size, shuffle=True),
    'val': DataLoader(data['val'], batch_size=batch_size, shuffle=True),
    'test': DataLoader(data['test'], batch_size=batch_size, shuffle=True)
}
```

```
In [16]: trainiter = iter(dataloaders['train'])
features, labels = next(trainiter)
features.shape, labels.shape
```

```
Out[16]: (torch.Size([128, 3, 224, 224]), torch.Size([128]))
```

```
In [17]: n_classes = len(cat_df)
print(f'There are {n_classes} different classes.')
len(data['train'].classes)
```

There are 100 different classes.

```
Out[17]: 100
```

In [19]: #using the pre-trained model -vgg for first few convolutional trained layers

```
model = models.vgg16(pretrained=True)
model
```

Out[19]: VGG(

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

```
In [ ]: #manual training to be done only on classifier layer. rest as it is
```

```
In [20]: # Freeze early layers
for param in model.parameters():
    param.requires_grad = False
```

```
In [21]: '''train a classifier consisting of the following layers

Fully connected with ReLU activation (n_inputs, 256)
Dropout with 40% chance of dropping
Fully connected with log softmax output (256, n_classes)'''

n_inputs = model.classifier[6].in_features

# Add on classifier
model.classifier[6] = nn.Sequential(
    nn.Linear(n_inputs, 256), nn.ReLU(), nn.Dropout(0.4),
    nn.Linear(256, n_classes), nn.LogSoftmax(dim=1))

model.classifier
```

```
Out[21]: Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Sequential(
        (0): Linear(in_features=4096, out_features=256, bias=True)
        (1): ReLU()
        (2): Dropout(p=0.4)
        (3): Linear(in_features=256, out_features=100, bias=True)
        (4): LogSoftmax()
    )
)
```

```
In [22]: total_params = sum(p.numel() for p in model.parameters())
print(f'{total_params:,} total parameters.')
total_trainable_params = sum(
    p.numel() for p in model.parameters() if p.requires_grad)
print(f'{total_trainable_params:,} training parameters.')
```

135,335,076 total parameters.  
1,074,532 training parameters.

```
In [23]: #moving to gpu
if train_on_gpu:
    model = model.to('cuda')

if multi_gpu:
    model = nn.DataParallel(model)
```

```
In [24]: #function to load in pretrained model
def get_pretrained_model(model_name):
    """Retrieve a pre-trained model from torchvision

    Params
    -----
        model_name (str): name of the model (currently only accepts vgg16 and resnet50)

    Return
    -----
        model (PyTorch model): cnn

    """
    if model_name == 'vgg16':
        model = models.vgg16(pretrained=True)

        # Freeze early layers
        for param in model.parameters():
            param.requires_grad = False
        n_inputs = model.classifier[6].in_features

        # Add on classifier
        model.classifier[6] = nn.Sequential(
            nn.Linear(n_inputs, 256), nn.ReLU(), nn.Dropout(0.2),
            nn.Linear(256, n_classes), nn.LogSoftmax(dim=1))

    elif model_name == 'resnet50':
        model = models.resnet50(pretrained=True)

        for param in model.parameters():
            param.requires_grad = False

        n_inputs = model.fc.in_features
        model.fc = nn.Sequential(
            nn.Linear(n_inputs, 256), nn.ReLU(), nn.Dropout(0.2),
            nn.Linear(256, n_classes), nn.LogSoftmax(dim=1))

    # Move to gpu and parallelize
    if train_on_gpu:
        model = model.to('cuda')

    if multi_gpu:
        model = nn.DataParallel(model)

    return model
```

```
In [25]: model = get_pretrained_model('vgg16')
if multi_gpu:
    summary(
        model.module,
        input_size=(3, 224, 224),
        batch_size=batch_size,
        device='cuda')
else:
    summary(
        model, input_size=(3, 224, 224), batch_size=batch_size, device='cuda')
```

Layer (type)	Output Shape	Param #
Conv2d-1	[128, 64, 224, 224]	1,792
ReLU-2	[128, 64, 224, 224]	0
Conv2d-3	[128, 64, 224, 224]	36,928
ReLU-4	[128, 64, 224, 224]	0
MaxPool2d-5	[128, 64, 112, 112]	0
Conv2d-6	[128, 128, 112, 112]	73,856
ReLU-7	[128, 128, 112, 112]	0
Conv2d-8	[128, 128, 112, 112]	147,584
ReLU-9	[128, 128, 112, 112]	0
MaxPool2d-10	[128, 128, 56, 56]	0
Conv2d-11	[128, 256, 56, 56]	295,168
ReLU-12	[128, 256, 56, 56]	0
Conv2d-13	[128, 256, 56, 56]	590,080
ReLU-14	[128, 256, 56, 56]	0
Conv2d-15	[128, 256, 56, 56]	590,080
ReLU-16	[128, 256, 56, 56]	0
MaxPool2d-17	[128, 256, 28, 28]	0
Conv2d-18	[128, 512, 28, 28]	1,180,160
ReLU-19	[128, 512, 28, 28]	0
Conv2d-20	[128, 512, 28, 28]	2,359,808
ReLU-21	[128, 512, 28, 28]	0
Conv2d-22	[128, 512, 28, 28]	2,359,808
ReLU-23	[128, 512, 28, 28]	0
MaxPool2d-24	[128, 512, 14, 14]	0
Conv2d-25	[128, 512, 14, 14]	2,359,808
ReLU-26	[128, 512, 14, 14]	0
Conv2d-27	[128, 512, 14, 14]	2,359,808
ReLU-28	[128, 512, 14, 14]	0
Conv2d-29	[128, 512, 14, 14]	2,359,808
ReLU-30	[128, 512, 14, 14]	0
MaxPool2d-31	[128, 512, 7, 7]	0
Linear-32	[128, 4096]	102,764,544
ReLU-33	[128, 4096]	0
Dropout-34	[128, 4096]	0
Linear-35	[128, 4096]	16,781,312
ReLU-36	[128, 4096]	0
Dropout-37	[128, 4096]	0
Linear-38	[128, 256]	1,048,832
ReLU-39	[128, 256]	0
Dropout-40	[128, 256]	0
Linear-41	[128, 100]	25,700
LogSoftmax-42	[128, 100]	0

```
=====
Total params: 135,335,076
Trainable params: 1,074,532
Non-trainable params: 134,260,544
-----
Input size (MB): 73.50
Forward/backward pass size (MB): 27979.45
Params size (MB): 516.26
Estimated Total Size (MB): 28569.21
-----
```

```
In [26]: if multi_gpu:
    print(model.module.classifier[6])
else:
    print(model.classifier[6])
```

```
Sequential(
(0): Linear(in_features=4096, out_features=256, bias=True)
(1): ReLU()
(2): Dropout(p=0.2)
(3): Linear(in_features=256, out_features=100, bias=True)
(4): LogSoftmax()
)
```

```
In [27]: #mapping of classes to indexes
```

```
model.class_to_idx = data['train'].class_to_idx
model.idx_to_class = {
    idx: class_
    for class_, idx in model.class_to_idx.items()
}

list(model.idx_to_class.items())[:10]
```

```
Out[27]: [(0, 'accordion'),
(1, 'airplanes'),
(2, 'anchor'),
(3, 'ant'),
(4, 'barrel'),
(5, 'bass'),
(6, 'beaver'),
(7, 'binocular'),
(8, 'bonsai'),
(9, 'brain')]
```

```
In [28]: criterion = nn.NLLLoss() #keeps track of the loss itself and the gradients of the
optimizer = optim.Adam(model.parameters()) #updates the parameters (weights) with
```

```
In [29]: for p in optimizer.param_groups[0]['params']:
    if p.requires_grad:
        print(p.shape)
```

```
torch.Size([256, 4096])
torch.Size([256])
torch.Size([100, 256])
torch.Size([100])
```

In [30]: #Training

```
def train(model,
          criterion,
          optimizer,
          train_loader,
          valid_loader,
          save_file_name,
          max_epochs_stop=3,
          n_epochs=20,
          print_every=2):
    """Train a PyTorch Model

    Params
    -----
        model (PyTorch model): cnn to train
        criterion (PyTorch loss): objective to minimize
        optimizer (PyTorch optimizier): optimizer to compute gradients of model p
        train_loader (PyTorch dataloader): training dataloader to iterate through
        valid_loader (PyTorch dataloader): validation dataloader used for early s
        save_file_name (str ending in '.pt'): file path to save the model state d
        max_epochs_stop (int): maximum number of epochs with no improvement in va
        n_epochs (int): maximum number of training epochs
        print_every (int): frequency of epochs to print training stats

    Returns
    -----
        model (PyTorch model): trained cnn with best weights
        history (DataFrame): history of train and validation loss and accuracy
    """

    # Early stopping initialization
    epochs_no_improve = 0
    valid_loss_min = np.Inf

    valid_max_acc = 0
    history = []

    # Number of epochs already trained (if using Loaded in model weights)
    try:
        print(f'Model has been trained for: {model.epochs} epochs.\n')
    except:
        model.epochs = 0
        print(f'Starting Training from Scratch.\n')

    overall_start = timer()

    # Main Loop
    for epoch in range(n_epochs):

        # keep track of training and validation loss each epoch
        train_loss = 0.0
        valid_loss = 0.0

        train_acc = 0
        valid_acc = 0
```

```
# Set to training
model.train()
start = timer()

# Training Loop
for ii, (data, target) in enumerate(train_loader):
    # Tensors to gpu
    if train_on_gpu:
        data, target = data.cuda(), target.cuda()

    # Clear gradients
    optimizer.zero_grad()
    # Predicted outputs are log probabilities
    output = model(data)

    # Loss and backpropagation of gradients
    loss = criterion(output, target)
    loss.backward()

    # Update the parameters
    optimizer.step()

    # Track train Loss by multiplying average Loss by number of examples
    train_loss += loss.item() * data.size(0)

    # Calculate accuracy by finding max log probability
    _, pred = torch.max(output, dim=1)
    correct_tensor = pred.eq(target.data.view_as(pred))
    # Need to convert correct tensor from int to float to average
    accuracy = torch.mean(correct_tensor.type(torch.FloatTensor))
    # Multiply average accuracy times the number of examples in batch
    train_acc += accuracy.item() * data.size(0)

    # Track training progress
    print(f'Epoch: {epoch}\t{100 * (ii + 1) / len(train_loader):.2f}% complete')
    end='\r')

# After training loops ends, start validation
else:
    model.epochs += 1

    # Don't need to keep track of gradients
    with torch.no_grad():
        # Set to evaluation mode
        model.eval()

        # Validation Loop
        for data, target in valid_loader:
            # Tensors to gpu
            if train_on_gpu:
                data, target = data.cuda(), target.cuda()

            # Forward pass
            output = model(data)
```

```
# Validation loss
loss = criterion(output, target)
# Multiply average loss times the number of examples in batch
valid_loss += loss.item() * data.size(0)

# Calculate validation accuracy
_, pred = torch.max(output, dim=1)
correct_tensor = pred.eq(target.data.view_as(pred))
accuracy = torch.mean(
    correct_tensor.type(torch.FloatTensor))
# Multiply average accuracy times the number of examples
valid_acc += accuracy.item() * data.size(0)

# Calculate average losses
train_loss = train_loss / len(train_loader.dataset)
valid_loss = valid_loss / len(valid_loader.dataset)

# Calculate average accuracy
train_acc = train_acc / len(train_loader.dataset)
valid_acc = valid_acc / len(valid_loader.dataset)

history.append([train_loss, valid_loss, train_acc, valid_acc])

# Print training and validation results
if (epoch + 1) % print_every == 0:
    print(
        f'\nEpoch: {epoch} \tTraining Loss: {train_loss:.4f} \tValidation Loss: {valid_loss:.4f}')
    print(
        f'\t\tTraining Accuracy: {100 * train_acc:.2f}% \t Validation Accuracy: {100 * valid_acc:.2f}%')

# Save the model if validation loss decreases
if valid_loss < valid_loss_min:
    # Save model
    torch.save(model.state_dict(), save_file_name)
    # Track improvement
    epochs_no_improve = 0
    valid_loss_min = valid_loss
    valid_best_acc = valid_acc
    best_epoch = epoch

# Otherwise increment count of epochs with no improvement
else:
    epochs_no_improve += 1
    # Trigger early stopping
    if epochs_no_improve >= max_epochs_stop:
        print(
            f'\nEarly Stopping! Total epochs: {epoch}. Best epoch: {best_epoch}, Validation Accuracy: {100 * valid_best_acc:.2f}%')
        total_time = timer() - overall_start
        print(
            f'{total_time:.2f} total seconds elapsed. {total_time:.2f} minutes elapsed.')
        break

# Load the best state dict
model.load_state_dict(torch.load(save_file_name))
```

```
# Attach the optimizer
model.optimizer = optimizer

# Format history
history = pd.DataFrame(
    history,
    columns=[
        'train_loss', 'valid_loss', 'train_acc',
        'valid_acc'
    ])
return model, history

# Attach the optimizer
model.optimizer = optimizer
# Record overall time and print out stats
total_time = timer() - overall_start
print(
    f'\nBest epoch: {best_epoch} with loss: {valid_loss_min:.2f} and acc: {100 * train_acc:.2f}%')
print(
    f'{total_time:.2f} total seconds elapsed. {total_time / (epoch):.2f} seconds per epoch')
# Format history
history = pd.DataFrame(
    history,
    columns=['train_loss', 'valid_loss', 'train_acc', 'valid_acc'])
return model, history
```

```
In [31]: model, history = train(  
    model,  
    criterion,  
    optimizer,  
    dataloaders['train'],  
    dataloaders['val'],  
    save_file_name=save_file_name,  
    max_epochs_stop=5,  
    n_epochs=30,  
    print_every=2)
```

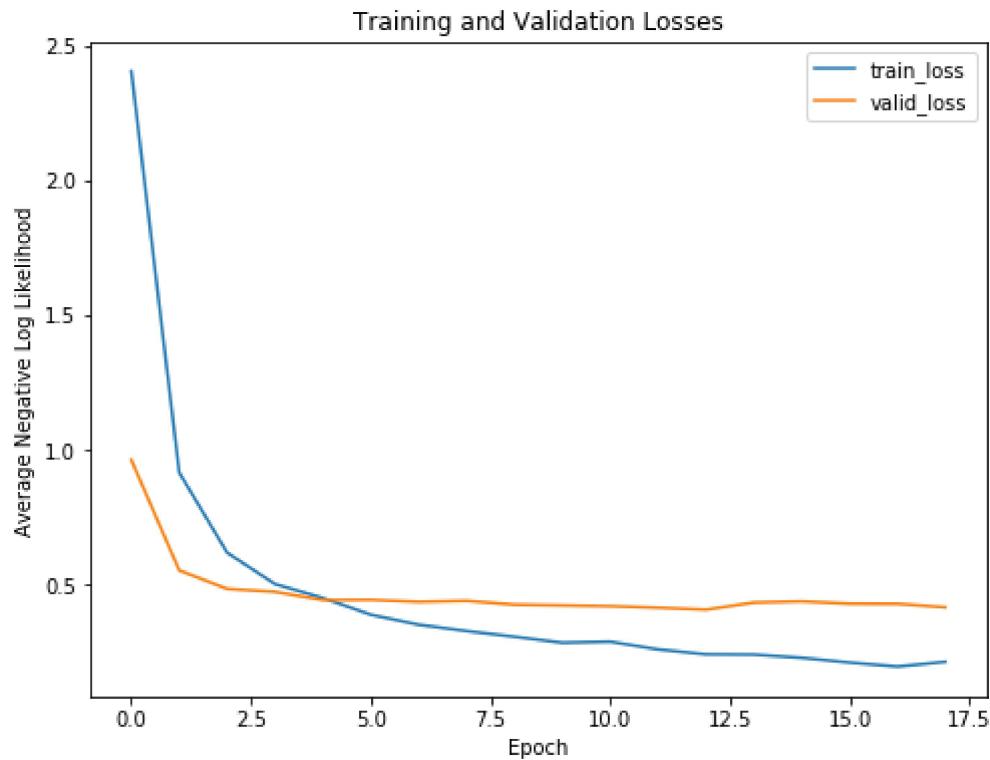
Starting Training from Scratch.

```
Epoch: 1      100.00% complete. 35.76 seconds elapsed in epoch.  
Epoch: 1      Training Loss: 0.9153   Validation Loss: 0.5520  
                  Training Accuracy: 76.76%       Validation Accuracy: 85.42%  
Epoch: 3      100.00% complete. 35.77 seconds elapsed in epoch.  
Epoch: 3      Training Loss: 0.5012   Validation Loss: 0.4724  
                  Training Accuracy: 86.06%       Validation Accuracy: 86.37%  
Epoch: 5      100.00% complete. 32.21 seconds elapsed in epoch.  
Epoch: 5      Training Loss: 0.3876   Validation Loss: 0.4425  
                  Training Accuracy: 88.92%       Validation Accuracy: 87.46%  
Epoch: 7      100.00% complete. 37.51 seconds elapsed in epoch.  
Epoch: 7      Training Loss: 0.3271   Validation Loss: 0.4389  
                  Training Accuracy: 90.04%       Validation Accuracy: 87.79%  
Epoch: 9      100.00% complete. 33.26 seconds elapsed in epoch.  
Epoch: 9      Training Loss: 0.2837   Validation Loss: 0.4220  
                  Training Accuracy: 91.61%       Validation Accuracy: 88.60%  
Epoch: 11     100.00% complete. 33.16 seconds elapsed in epoch.  
Epoch: 11     Training Loss: 0.2590   Validation Loss: 0.4135  
                  Training Accuracy: 92.17%       Validation Accuracy: 89.07%  
Epoch: 13     100.00% complete. 31.84 seconds elapsed in epoch.  
Epoch: 13     Training Loss: 0.2397   Validation Loss: 0.4326  
                  Training Accuracy: 92.51%       Validation Accuracy: 88.22%  
Epoch: 15     100.00% complete. 36.79 seconds elapsed in epoch.  
Epoch: 15     Training Loss: 0.2101   Validation Loss: 0.4284  
                  Training Accuracy: 93.33%       Validation Accuracy: 88.93%  
Epoch: 17     100.00% complete. 30.47 seconds elapsed in epoch.  
Epoch: 17     Training Loss: 0.2127   Validation Loss: 0.4152  
                  Training Accuracy: 93.36%       Validation Accuracy: 89.07%
```

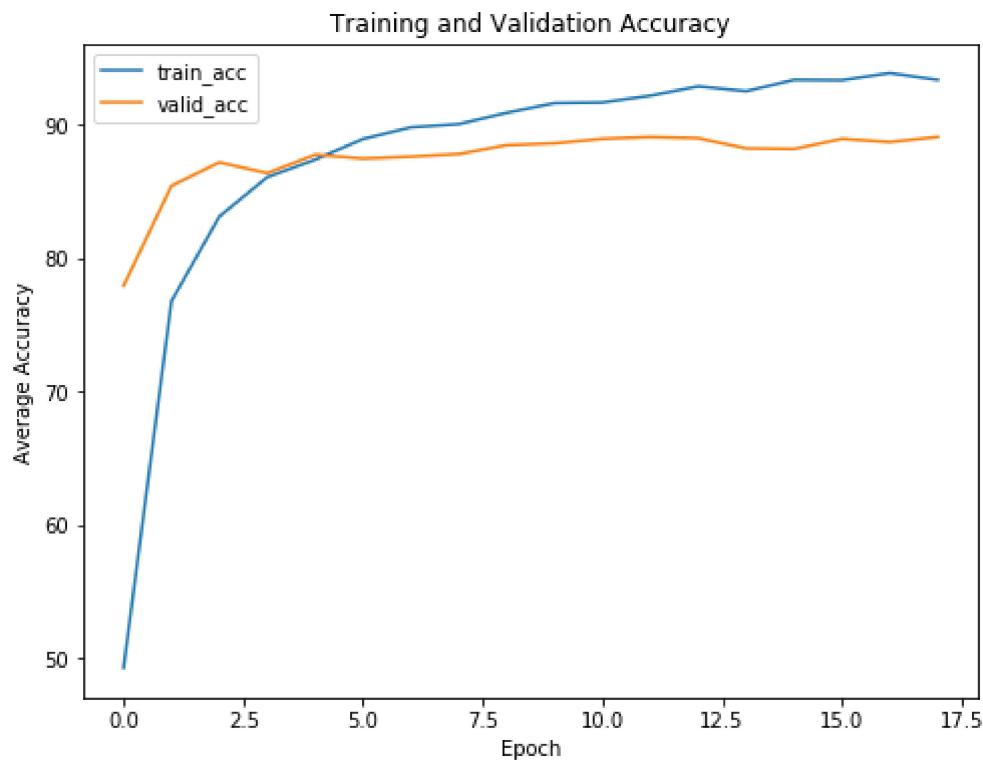
```
Early Stopping! Total epochs: 17. Best epoch: 12 with loss: 0.41 and acc: 89.07%  
931.73 total seconds elapsed. 51.76 seconds per epoch.
```

In [32]: #Training results shown graphically

```
plt.figure(figsize=(8, 6))
for c in ['train_loss', 'valid_loss']:
    plt.plot(
        history[c], label=c)
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Average Negative Log Likelihood')
plt.title('Training and Validation Losses')
```



```
In [33]: plt.figure(figsize=(8, 6))
for c in ['train_acc', 'valid_acc']:
    plt.plot(
        100 * history[c], label=c)
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Average Accuracy')
plt.title('Training and Validation Accuracy')
```



```
In [ ]: #The accuracy comes out to be >80%
```

```
In [40]: '''This function processes an image path into a PyTorch tensor for predictions. I

def process_image(image_path):
    """Process an image path into a PyTorch tensor"""

    image = Image.open(image_path)
    # Resize
    img = image.resize((256, 256))

    # Center crop
    width = 256
    height = 256
    new_width = 224
    new_height = 224

    left = (width - new_width) / 2
    top = (height - new_height) / 2
    right = (width + new_width) / 2
    bottom = (height + new_height) / 2
    img = img.crop((left, top, right, bottom))

    # Convert to numpy, transpose color dimension and normalize
    img = np.array(img).transpose((2, 0, 1)) / 256

    # Standardization
    means = np.array([0.485, 0.456, 0.406]).reshape((3, 1, 1))
    stds = np.array([0.229, 0.224, 0.225]).reshape((3, 1, 1))

    img = img - means
    img = img / stds

    img_tensor = torch.Tensor(img)

    return img_tensor
```

```
In [41]: x = process_image(testdir + 'dragonfly/image_0015.jpg')
x.shape
```

```
Out[41]: torch.Size([3, 224, 224])
```

```
In [42]: ax, image = imshow_tensor(x)
```



```
In [43]: ax, image = imshow_tensor(process_image(testdir + 'dalmatian/image_0053.jpg'))
```



In [44]: #Function to make predictions

```
def predict(image_path, model, topk=5):
    """Make a prediction for an image using a trained model

    Params
    -----
        image_path (str): filename of the image
        model (PyTorch model): trained model for inference
        topk (int): number of top predictions to return

    Returns
    """
    real_class = image_path.split('/')[-2]

    # Convert to pytorch tensor
    img_tensor = process_image(image_path)

    # Resize
    if train_on_gpu:
        img_tensor = img_tensor.view(1, 3, 224, 224).cuda()
    else:
        img_tensor = img_tensor.view(1, 3, 224, 224)

    # Set to evaluation
    with torch.no_grad():
        model.eval()
        # Model outputs log probabilities
        out = model(img_tensor)
        ps = torch.exp(out)

        # Find the topk predictions
        topk, topclass = ps.topk(topk, dim=1)

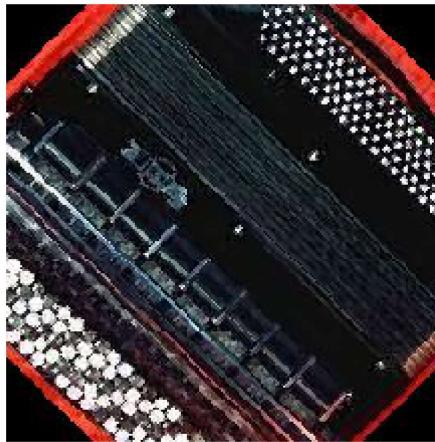
        # Extract the actual classes and probabilities
        top_classes = [
            model.idx_to_class[class_] for class_ in topclass.cpu().numpy()[0]
        ]
        top_p = topk.cpu().numpy()[0]

    return img_tensor.cpu().squeeze(), top_p, top_classes, real_class
```

```
In [45]: np.random.seed = 100
```

```
def random_test_image():
    """Pick a random test image from the test directory"""
    c = np.random.choice(cat_df['category'])
    root = testdir + c + '/'
    img_path = root + np.random.choice(os.listdir(root))
    return img_path

_ = imshow_tensor(process_image(random_test_image()))
```



```
In [46]: img, top_p, top_classes, real_class = predict(random_test_image(), model)
img.shape
```

```
Out[46]: torch.Size([3, 224, 224])
```

```
In [47]: top_p, top_classes, real_class
```

```
Out[47]: (array([0.615789 , 0.35459077, 0.01252878, 0.00679292, 0.00269399],
                  dtype=float32),
          ['ceiling_fan', 'gramophone', 'anchor', 'chair', 'octopus'],
          'ceiling_fan')
```

In [48]: `img, top_p, top_classes, real_class = predict(random_test_image(), model)`  
`top_p, top_classes, real_class`

Out[48]: `(array([9.9574465e-01, 9.7864203e-04, 9.5386576e-04, 6.3906156e-04,`  
`6.0763489e-04], dtype=float32),  
['pizza', 'brain', 'lobster', 'garfield', 'nautilus'],  
'pizza')`

In [49]: `#function to display predictions. some predicted classes are shown but title seen`

```
def display_prediction(image_path, model, topk):
    """Display image and predictions from model"""

    # Get predictions
    img, ps, classes, y_obs = predict(image_path, model, topk)
    # Convert results to dataframe for plotting
    result = pd.DataFrame({'p': ps}, index=classes)

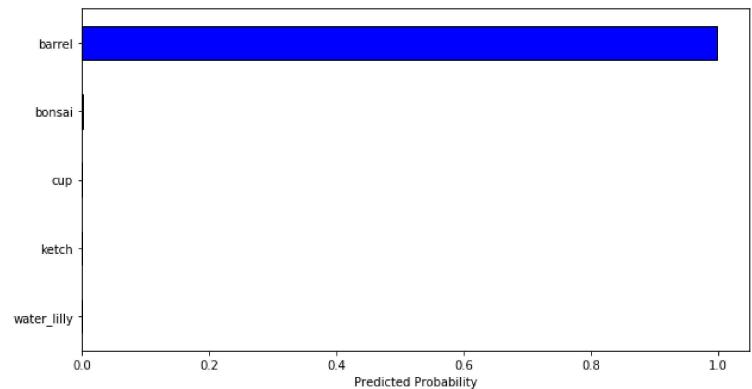
    # Show the image
    plt.figure(figsize=(16, 5))
    ax = plt.subplot(1, 2, 1)
    ax, img = imshow_tensor(img, ax=ax)

    # Set title to be the actual class
    ax.set_title(y_obs, size=20)

    ax = plt.subplot(1, 2, 2)
    # Plot a bar plot of predictions
    result.sort_values('p')['p'].plot.barh(color='blue', edgecolor='k', ax=ax)
    plt.xlabel('Predicted Probability')
    plt.tight_layout()
```

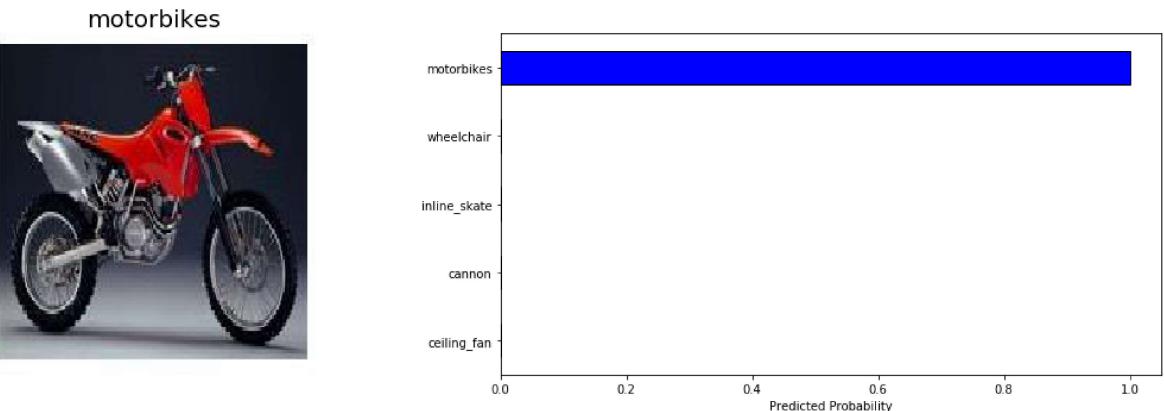
In [52]: `display_prediction(random_test_image(), model, topk=5)`

23 training images for barrel.



In [53]: `display_prediction(random_test_image(), model, topk=5)`

398 training images for motorbikes.



In [54]: `#Testing the accuracy`

```
def accuracy(output, target, topk=(1, )):
    """Compute the topk accuracy(s)"""
    if train_on_gpu:
        output = output.to('cuda')
        target = target.to('cuda')

    with torch.no_grad():
        maxk = max(topk)
        batch_size = target.size(0)

        # Find the predicted classes and transpose
        _, pred = output.topk(k=maxk, dim=1, largest=True, sorted=True)
        pred = pred.t()

        # Determine predictions equal to the targets
        correct = pred.eq(target.view(1, -1).expand_as(pred))

        res = []

        # For each k, find the percentage of correct
        for k in topk:
            correct_k = correct[:k].view(-1).float().sum(0, keepdim=True)
            res.append(correct_k.mul_(100.0 / batch_size).item())
    return res
```

```
In [55]: testiter = iter(dataloaders['test'])
# Get a batch of testing images and labels
features, targets = next(testiter)

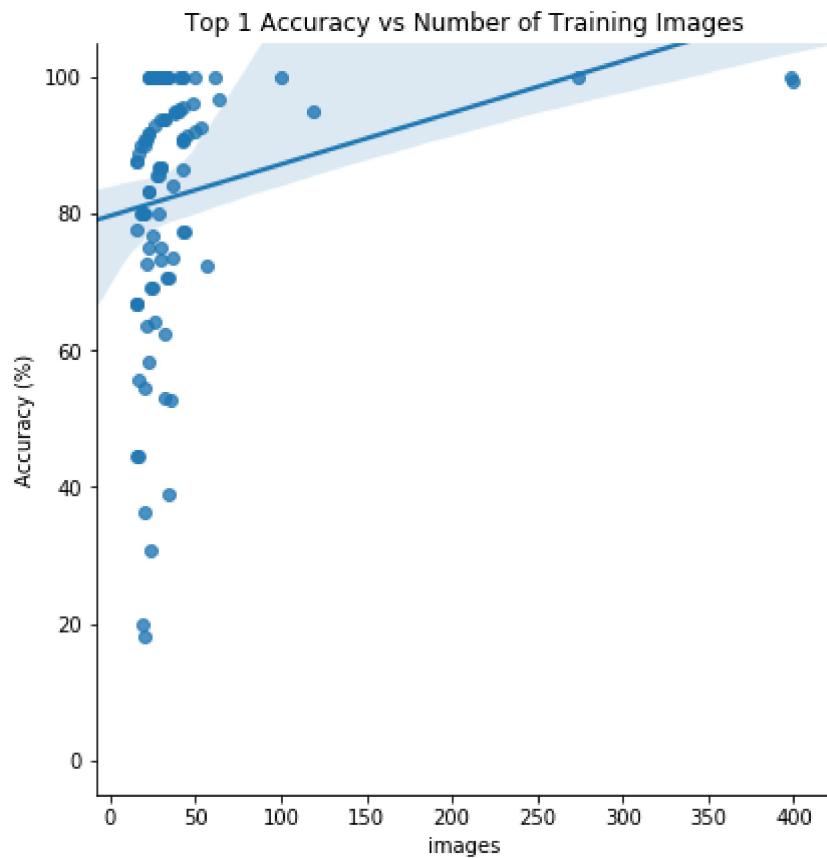
if train_on_gpu:
    accuracy(model(features.to('cuda')), targets, topk=(1, 5))
else:
    accuracy(model(features), targets, topk=(1, 5))
```

```
Out[55]: [89.84375, 99.21875]
```

```
In [58]: #visualising results

results = results.merge(cat_df, left_on='class', right_on='category').\
drop(columns=['category'])

# Plot using seaborn
sns.lmplot(
    y='top1', x='n_train', data=results, height=6)
plt.xlabel('images')
plt.ylabel('Accuracy (%)')
plt.title('Top 1 Accuracy vs Number of Training Images')
plt.ylim(-5, 105)
```



```
In [59]: print('Category with minimum accuracy.')
results.loc[results['top1'].idxmin]

print('Category with minimum images.')
results.loc[results['n_train'].idxmin]
```

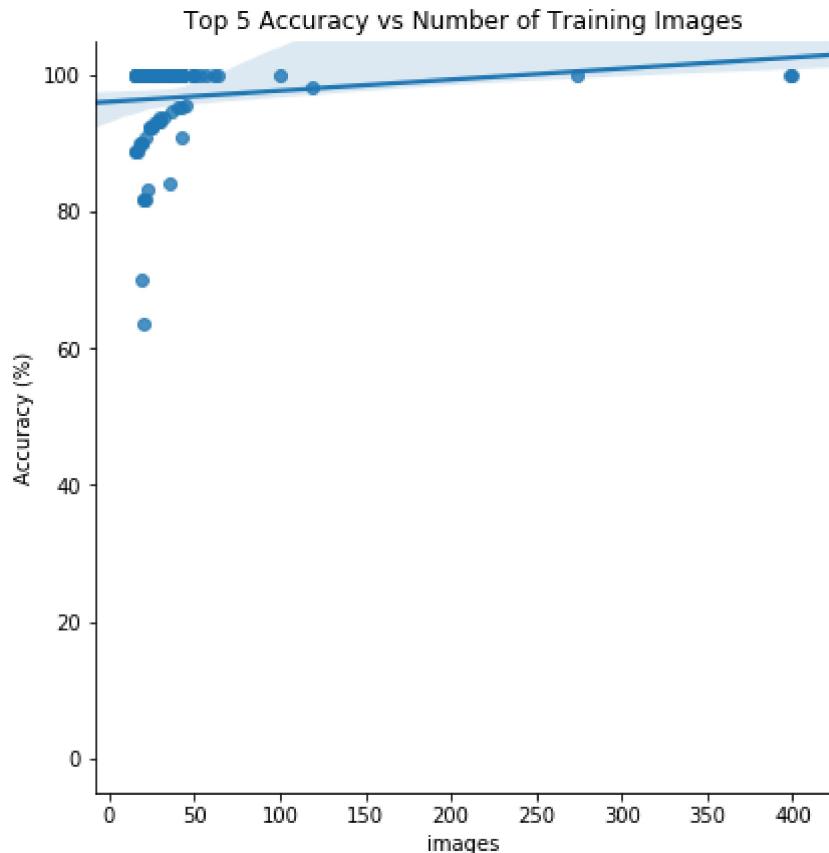
Category with minimum accuracy.

```
Out[59]: class      anchor
          top1      18.1818
          top5      63.6364
          loss      2.89586
          n_train     20
          n_valid     11
          n_test      11
Name: 2, dtype: object
```

Category with minimum images.

```
Out[59]: class      inline_skate
          top1      87.5
          top5      100
          loss      0.221262
          n_train     15
          n_valid     8
          n_test      8
Name: 49, dtype: object
```

```
In [60]: sns.lmplot(
    y='top5', x='n_train', data=results, height=6)
plt.xlabel('images')
plt.ylabel('Accuracy (%)')
plt.title('Top 5 Accuracy vs Number of Training Images')
plt.ylim(-5, 105)
```



```
In [61]: # Weighted column of test images
results['weighted'] = results['n_test'] / results['n_test'].sum()

# Create weighted accuracies
for i in (1, 5):
    results[f'weighted_top{i}'] = results['weighted'] * results[f'top{i}']

# Find final accuracy accounting for frequencies
top1_weighted = results['weighted_top1'].sum()
top5_weighted = results['weighted_top5'].sum()
loss_weighted = (results['weighted'] * results['loss']).sum()

print(f'Final test cross entropy per image = {loss_weighted:.4f}.')
print(f'Final test top 1 weighted accuracy = {top1_weighted:.2f}%')
print(f'Final test top 5 weighted accuracy = {top5_weighted:.2f}%')
```

Final test cross entropy per image = 0.3772.

Final test top 1 weighted accuracy = 88.65%

Final test top 5 weighted accuracy = 98.00%

In [70]: *#fucntion to display the predictions for an image*

```
def display_category(model, category, n=4):
    """Display predictions for a category
    """
    category_results = results.loc[results['class'] == category]
    print(category_results.iloc[:, :6], '/n')

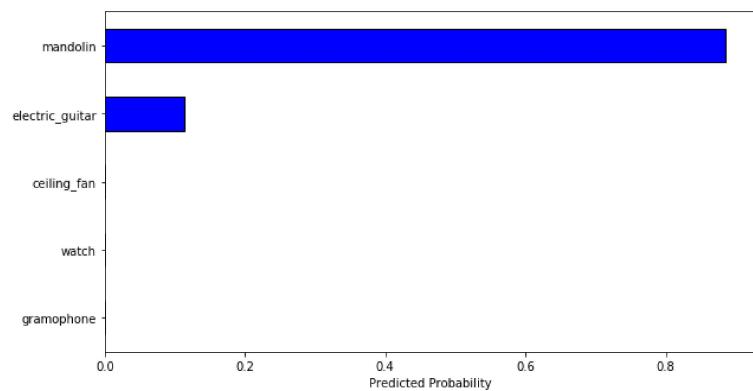
    images = np.random.choice(
        os.listdir(testdir + category + '/'), size=4, replace=False)

    for img in images:
        display_prediction(testdir + category + '/' + img, model, 5)
```

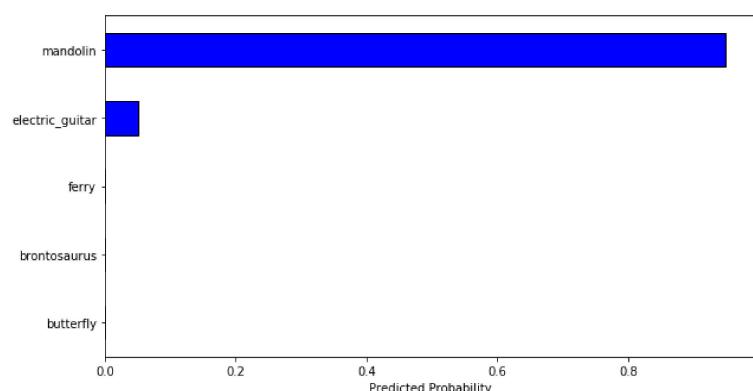
```
In [73]: display_category(model, 'mandolin')
```

class	top1	top5	loss	n_train	n_valid
59 mandolin	90.909091	100.0	0.420903	21	11 /n
21 training images for mandolin.					
21 training images for mandolin.					
21 training images for mandolin.					
21 training images for mandolin.					

mandolin



mandolin



mandolin

