sound_dogs_mfcc_CNN

March 4, 2024

[ ]:
```
#purpose is the run a shorter model to see if it improves predicts and␣
 ↪ordination results
```

[1]:
```
#pip install --upgrade torch torchvision
```

[23]:
```
#pip install librosa
import torch                                          # imports the core PyTorch␣
 ↪library
import torch.nn as nn                                 # torch.nn library is a␣
 ↪high-level interface for building and training neural networks
import torch.optim as optim                           # Needed for optimizer
from torch.utils.data import DataLoader               # Needed for loading data
from torchvision import datasets, transforms          # Needed for making sure␣
 ↪data is properly formatted
import torch.nn.functional as F                       # Needed for functional␣
 ↪equation in forward loop
import numpy as np                                    # Needed for outputting␣
 ↪weights and biases
from torchvision.datasets import VisionDataset        # VisionDataset is is␣
 ↪designed to be a base class for datasets in computer vision tasks
from PIL import Image                                 # Needed for manipulating␣
 ↪png image files
import os                                             # Needed for operating␣
 ↪system for dealing with dir of training and test png files
from torchvision import transforms
from skimage import exposure
import matplotlib.pyplot as plt
from torch.utils.data import Subset
from sklearn.model_selection import train_test_split
from torchvision.datasets.vision import VisionDataset
from torch.utils.data import Subset
from sklearn.model_selection import train_test_split
from PIL import Image
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
#import librosa
#import librosa.display
```

```
import matplotlib.pyplot as plt
#from python_speech_features import mfcc
import numpy as np
from matplotlib.backends.backend_pdf import PdfPages
#import pygame
import random


# Set the device
device = "mps" if torch.backends.mps.is_available() else "cpu"
# Check PyTorch has access to MPS (Metal Performance Shader, Apple's GPU␣
 ↪architecture)
print(f"Is MPS (Metal Performance Shader) built? {torch.backends.mps.
 ↪is_built()}")
```

```
Is MPS (Metal Performance Shader) built? True
```

[47]:
```
class DogSounds(VisionDataset):
    def __init__(self, root, split='train', transform=None,␣
 ↪target_transform=None):
        super(DogSounds, self).__init__(root, transform=transform,␣
 ↪target_transform=target_transform)

        self.split = split
        self.data_folder = "training" if split == 'train' else "testing"
        self.images_folder = os.path.join(self.root, self.data_folder)

        self.image_paths = self._get_image_paths()
        self.mean = np.array([0.36062344, 0.36062344, 0.36062344])
        self.std = np.array([0.28482647, 0.28482647, 0.28482647])

        #Calculated mean: [0.33440762 0.11394685 0.3411391 ]
        #Calculated std: [0.27980838 0.12647624 0.18165884]
        #Calculated mean: [0.36062344 0.36062344 0.36062344]
        #Calculated std: [0.28482647 0.28482647 0.28482647]

         # Add a check to ensure transform is not None
        if transform is None:
            self.transform = transforms.Compose([
                transforms.ToTensor(),
                # Commented out normalization for now
                transforms.Normalize(mean=self.mean, std=self.std),
            ])
        else:
            self.transform = transform

    def _get_image_paths(self):
```

```
        image_paths = []
        #print("Getting image paths...")
        #print(f"Checking folder: {self.images_folder}")

        for digit_folder in os.listdir(self.images_folder):
            digit_folder_path = os.path.join(self.images_folder, digit_folder)

            # Check if it's a directory before listing its contents
            if os.path.isdir(digit_folder_path):
                for image_name in os.listdir(digit_folder_path):
                    image_path = os.path.join(digit_folder_path, image_name)

                    # Skip files with the '.DS_Store' extension
                    if not image_path.endswith('.DS_Store'):
                        image_paths.append((image_path, int(digit_folder)))

        return image_paths

    def _split_dataset(self, dataset, train_size=0.70, val_size=0.15,␣
↪test_size=0.15, random_state=None):
        # Extract images and labels from the dataset
        images, labels = zip(*dataset)

        # First, split into train and temp sets
        train_images, temp_images, train_labels, temp_labels = train_test_split(
            images, labels, test_size=(val_size + test_size),␣
↪random_state=random_state
        )

        # Second, split temp set into validation and test sets
        val_images, test_images, val_labels, test_labels = train_test_split(
            temp_images, temp_labels, test_size=test_size / (val_size +␣
↪test_size), random_state=random_state
        )

        train_dataset = list(zip(train_images, train_labels))
        val_dataset = list(zip(val_images, val_labels))
        test_dataset = list(zip(test_images, test_labels))

        return train_dataset, val_dataset, test_dataset

    def get_datasets(self):
        if self.split not in ['train', 'val', 'test']:
            raise ValueError("Invalid split. Use 'train', 'val', or 'test'.")

        dataset = self._get_image_paths()
        train_dataset, val_dataset, test_dataset = self._split_dataset(dataset)
```

```
        return train_dataset, val_dataset, test_dataset
    def __getitem__(self, index):
        if index < 0 or index >= len(self.image_paths):
            raise IndexError("Index out of range")

        image_path, label = self.image_paths[index]

    # Skip files with the '.DS_Store' extension
        while image_path.endswith('.DS_Store'):
            index += 1
            if index >= len(self.image_paths):
                raise IndexError("Index out of range")
            image_path, label = self.image_paths[index]

    # Open and read the image
        print(f"Opening image: {image_path}")

        with Image.open(image_path) as image:
        # Convert image to RGB, regardless of its mode
            image = image.convert('RGB')
            print(f"Image shape before transformation: {np.array(image).
↪shape}", flush=True)

        # Ensure the image has 3 channels
            if image.mode != 'RGB':
                image = image.convert('RGB')

            if self.transform is not None:
                image = self.transform(image)
            print(f"Image shape after transformation: {image.shape}",␣
↪flush=True)

        return image, label

    def __len__(self):
        return len(self.image_paths)

    def custom_collate(self, batch):
    # Unpack the batch
        image_paths, labels = zip(*batch)

    # Load images and apply transformations
        image_tensors = [self.transform(Image.open(img_path).convert('RGB'))␣
↪for img_path in image_paths]

    # Convert labels to PyTorch tensor
```

```
        label_tensor = torch.tensor(labels, dtype=torch.long)

        return torch.stack(image_tensors), label_tensor
```

[48]:
```
import numpy as np
from PIL import Image
import os

# Assuming you have a folder containing images
folder_path = '/Users/peternoble/Desktop/dog_sounds_mfcc/training/0'

# Get all image paths in the folder
image_paths = [os.path.join(folder_path, filename) for filename in os.
 ↪listdir(folder_path) if filename.endswith(('.png', '.jpg', '.jpeg'))]

# Calculate mean and std
mean = np.zeros(3)
std = np.zeros(3)

for img_path in image_paths:
    img = np.array(Image.open(img_path).convert('RGB')) / 255.0
    mean += np.mean(img, axis=(0, 1))
    std += np.std(img, axis=(0, 1))

mean /= len(image_paths)
std /= len(image_paths)

print("Calculated mean:", mean)
print("Calculated std:", std)
```

```
Calculated mean: [0.26379778 0.71238532 0.44071646]
Calculated std: [0.1195104  0.10945572 0.07368146]
```

[49]:
```
# Function to print class distribution
def print_class_distribution(loader, name):
    labels = [label for _, label in loader.dataset]
    print(f"{name} set size: {len(loader.dataset)}")
    for class_label in range(3):  # Assuming you have 4 classes
        count = labels.count(class_label)
        print(f"Class {class_label}: {count} examples in the {name} set")

# Create separate instances for training, validation, and test datasets
root_directory = '/Users/peternoble/Desktop/dog_sounds_mfcc'
DogSounds_instance = DogSounds(root_directory, split='train')
train_dataset, val_dataset, test_dataset = DogSounds_instance.get_datasets()

# Create data loaders
```

```
# Create data loaders
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32,␣
 ↪shuffle=True, collate_fn=DogSounds_instance.custom_collate)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=32,␣
 ↪shuffle=False, collate_fn=DogSounds_instance.custom_collate)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32,␣
 ↪shuffle=False, collate_fn=DogSounds_instance.custom_collate)

# Print dataset sizes
print_class_distribution(train_loader, "Train")
print_class_distribution(val_loader, "Validation")
print_class_distribution(test_loader, "Test")
```

```
Train set size: 72
Class 0: 47 examples in the Train set
Class 1: 11 examples in the Train set
Class 2: 14 examples in the Train set
Validation set size: 16
Class 0: 8 examples in the Validation set
Class 1: 4 examples in the Validation set
Class 2: 4 examples in the Validation set
Test set size: 16
Class 0: 10 examples in the Test set
Class 1: 2 examples in the Test set
Class 2: 4 examples in the Test set
```

[50]:
```
import os
from torchvision import transforms

# Set the path where you want to save the images. Useful for visualizing the␣
 ↪actual images used for validation
output_path = '/Users/peternoble/Desktop/results'
# Ensure the output directory exists
os.makedirs(output_path, exist_ok=True)
# Variables to store all images and labels
all_images = []
all_labels = []

# Iterate through the validation loader and concatenate images and labels
for batch_idx, (images, labels) in enumerate(val_loader):
    # Apply histogram equalization to each image in the batch
    # equalized_images = torch.stack([torch.tensor(exposure.equalize_hist(img.
 ↪numpy())) for img in images])
    # all_images.append(equalized_images)
    all_images.append(images)
    all_labels.append(labels)
```

```
# Concatenate the lists to get all images and labels
all_images = torch.cat(all_images, dim=0)
all_labels = torch.cat(all_labels, dim=0)

# Define a function to save images
def save_images(images, labels, output_path):
    for i in range(len(images)):
        image, label = images[i], labels[i]
        # Assuming the images are in the range [0, 1], and using torchvision to␣
↪convert to PIL
        image_pil = transforms.ToPILImage()(image)
        image_pil.save(os.path.join(output_path, f"image_{i}_label_{label}.
↪png"))

# Save all equalized images from the validation dataset
save_images(all_images, all_labels, output_path)
```

```
/Users/peternoble/miniforge3/envs/my_arm_environment/lib/python3.8/site-
packages/torchvision/transforms/functional.py:281: RuntimeWarning: invalid value
encountered in cast
  npimg = (npimg * 255).astype(np.uint8)
/Users/peternoble/miniforge3/envs/my_arm_environment/lib/python3.8/site-
packages/torchvision/transforms/functional.py:281: RuntimeWarning: invalid value
encountered in cast
  npimg = (npimg * 255).astype(np.uint8)
/Users/peternoble/miniforge3/envs/my_arm_environment/lib/python3.8/site-
packages/torchvision/transforms/functional.py:281: RuntimeWarning: invalid value
encountered in cast
  npimg = (npimg * 255).astype(np.uint8)
/Users/peternoble/miniforge3/envs/my_arm_environment/lib/python3.8/site-
packages/torchvision/transforms/functional.py:281: RuntimeWarning: invalid value
encountered in cast
  npimg = (npimg * 255).astype(np.uint8)
```

[51]:

```
class CustomCNN(nn.Module):
    def __init__(self, num_classes=3):
        super(CustomCNN, self).__init__()

        # Layer 1
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=2,␣
↪stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 2
        self.conv2 = nn.Conv2d(32, 16, kernel_size=4, padding=1)
```

```
        self.bn2 = nn.BatchNorm2d(16)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=4, stride=2)

        # Layer 3
        self.conv3 = nn.Conv2d(16, 4, kernel_size=2, padding=1)
        self.bn3 = nn.BatchNorm2d(4)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 4
        self.conv4 = nn.Conv2d(4, 16, kernel_size=2, padding=1)
        self.bn4 = nn.BatchNorm2d(16)
        self.relu4 = nn.ReLU()
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 5
        self.conv5 = nn.Conv2d(16, 32, kernel_size=2, padding=1)
        self.bn5 = nn.BatchNorm2d(32)
        self.relu5 = nn.ReLU()
        self.pool5 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 6
        self.conv6 = nn.Conv2d(32, 64, kernel_size=2, padding=1)
        self.bn6 = nn.BatchNorm2d(64)
        self.relu6 = nn.ReLU()
        self.pool6 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 7
        self.conv7 = nn.Conv2d(64, 128, kernel_size=2, padding=1)
        self.bn7 = nn.BatchNorm2d(128)
        self.relu7 = nn.ReLU()
        self.pool7 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 8
        self.conv8 = nn.Conv2d(128, 488, kernel_size=2, padding=1)
        self.bn8 = nn.BatchNorm2d(488)
        self.relu8 = nn.ReLU()
        self.pool8 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Adjusted fully connected layer
        #self.fc = nn.Linear(488 * 2 * 2, num_classes)  # Adjust the size based␣
↪on your model architecture
        self.fc = nn.Linear(2928, num_classes)  # Adjust the size based on your␣
↪model architecture

    def forward(self, x):
```

```
        x = self.pool1(self.relu1(self.bn1(self.conv1(x))))
        x = self.pool2(self.relu2(self.bn2(self.conv2(x))))
        x = self.pool3(self.relu3(self.bn3(self.conv3(x))))
        x = self.pool4(self.relu4(self.bn4(self.conv4(x))))
        x = self.pool5(self.relu5(self.bn5(self.conv5(x))))
        x = self.pool6(self.relu6(self.bn6(self.conv6(x))))
        x = self.pool7(self.relu7(self.bn7(self.conv7(x))))
        x = self.pool8(self.relu8(self.bn8(self.conv8(x))))

        # Flatten the output before passing it through the fully connected layer
        x = x.view(x.size(0), -1)

        # Fully connected layer
        x = self.fc(x)

        return x

# Create an instance of the CustomCNN model
model = CustomCNN()

# Print the model architecture
print(model)
```

```
CustomCNN(
  (conv1): Conv2d(3, 32, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu1): ReLU()
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv2d(32, 16, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu2): ReLU()
  (pool2): MaxPool2d(kernel_size=4, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv3): Conv2d(16, 4, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(4, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu3): ReLU()
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv4): Conv2d(4, 16, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu4): ReLU()
  (pool4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
```

```
  (conv5): Conv2d(16, 32, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn5): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu5): ReLU()
  (pool5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv6): Conv2d(32, 64, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn6): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu6): ReLU()
  (pool6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv7): Conv2d(64, 128, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu7): ReLU()
  (pool7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv8): Conv2d(128, 488, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn8): BatchNorm2d(488, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu8): ReLU()
  (pool8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (fc): Linear(in_features=2928, out_features=3, bias=True)
)
```

[52]:
```python
# Assuming sugarcane_loader is your DataLoader
for inputs, labels in val_loader:
    print("Input shape:", inputs.shape)
    break
```

```
Input shape: torch.Size([16, 3, 300, 600])
```

[ ]:

[ ]:
```python
#Instantiate your model, optimizer, and criterion
model = CustomCNN()  # Create an instance of your CustomModel
#optimizer = optim.Adam(model.parameters(), lr=0.001)
# Assuming you have a validation DataLoader named val_loader
# Specify the weight decay (regularization strength)
weight_decay = 0.01
#loaded_model = CustomCNN()
#loaded_model.load_state_dict(torch.load('complex_model_95_8.pth'))

# Define your optimizer and pass the weight_decay parameter L2 regular
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=weight_decay)
#optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
# Define RMSprop optimizer
#optimizer = optim.RMSprop(model.parameters(), lr=0.001, alpha=0.9)

criterion = nn.CrossEntropyLoss()

num_epochs = 100

class EarlyStopping:
    def __init__(self, patience=5, delta=0, verbose=False):
        self.patience = patience
        self.delta = delta
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False

    def __call__(self, val_loss, model):
        score = -val_loss

        if self.best_score is None:
            self.best_score = score
        elif score < self.best_score + self.delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.counter = 0

        if self.verbose:
            print(f'EarlyStopping counter: {self.counter} out of {self.
 ↪patience}')

        return self.early_stop

# Create an instance of EarlyStopping before the training loop
early_stopping = EarlyStopping(patience=25, verbose=True)

train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []

# Initialize variables for accuracy calculation
correct_train = 0
total_train = 0
correct_val = 0
```

```
total_val = 0

for epoch in range(num_epochs):
    model.train()
    running_train_loss = 0.0

    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item()

        _, predicted_train = outputs.max(1)
        total_train += labels.size(0)
        correct_train += predicted_train.eq(labels).sum().item()

    average_train_loss = running_train_loss / len(train_loader)
    train_losses.append(average_train_loss)
    train_accuracy = correct_train / total_train
    train_accuracies.append(train_accuracy)

    model.eval()
    running_val_loss = 0.0

    # Reset variables for accuracy calculation
    correct_val = 0
    total_val = 0

    with torch.no_grad():
        for val_inputs, val_labels in val_loader:
            val_outputs = model(val_inputs)
            val_loss = criterion(val_outputs, val_labels)
            running_val_loss += val_loss.item()

            _, predicted_val = val_outputs.max(1)
            total_val += val_labels.size(0)
            correct_val += predicted_val.eq(val_labels).sum().item()

    average_val_loss = running_val_loss / len(val_loader)
    val_losses.append(average_val_loss)
    val_accuracy = correct_val / total_val
    val_accuracies.append(val_accuracy)

    # Call early_stopping within the loop
    if early_stopping(average_val_loss, model):
```

```
        print("Early stopping")
        break

    print(f"Epoch {epoch + 1}/{num_epochs}, Training Loss: {average_train_loss:.
↪4f}, Training Accuracy: {train_accuracy:.4f}, Validation Loss:␣
↪{average_val_loss:.4f}, Validation Accuracy: {val_accuracy:.4f}")

# Plotting the accuracy curve
epochs = range(1, len(train_accuracies) + 1)
plt.plot(epochs, train_accuracies, label='Training Accuracy')
plt.plot(epochs, val_accuracies, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy Over Epochs')
plt.legend()
plt.show()

# Plotting the loss curve
epochs = range(1, len(train_losses) + 1)
plt.plot(epochs, train_losses, label='Training Loss')
plt.plot(epochs, val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss Over Epochs')
plt.legend()
plt.show()
```

```
EarlyStopping counter: 0 out of 25
Epoch 1/100, Training Loss: 2.0966, Training Accuracy: 0.3889, Validation Loss:
1.0791, Validation Accuracy: 0.5000
EarlyStopping counter: 0 out of 25
Epoch 2/100, Training Loss: 1.5739, Training Accuracy: 0.4375, Validation Loss:
1.0504, Validation Accuracy: 0.5000
EarlyStopping counter: 1 out of 25
Epoch 3/100, Training Loss: 1.5535, Training Accuracy: 0.4954, Validation Loss:
1.0764, Validation Accuracy: 0.5000
EarlyStopping counter: 2 out of 25
Epoch 4/100, Training Loss: 0.7801, Training Accuracy: 0.5625, Validation Loss:
1.0727, Validation Accuracy: 0.5000
EarlyStopping counter: 3 out of 25
Epoch 5/100, Training Loss: 0.7216, Training Accuracy: 0.5972, Validation Loss:
1.1025, Validation Accuracy: 0.5000
EarlyStopping counter: 4 out of 25
Epoch 6/100, Training Loss: 0.6672, Training Accuracy: 0.6157, Validation Loss:
1.2130, Validation Accuracy: 0.5000
EarlyStopping counter: 5 out of 25
Epoch 7/100, Training Loss: 0.5573, Training Accuracy: 0.6329, Validation Loss:
1.3331, Validation Accuracy: 0.5000
```

```
EarlyStopping counter: 6 out of 25
Epoch 8/100, Training Loss: 0.5649, Training Accuracy: 0.6458, Validation Loss:
1.3490, Validation Accuracy: 0.5000
EarlyStopping counter: 7 out of 25
Epoch 9/100, Training Loss: 0.5231, Training Accuracy: 0.6636, Validation Loss:
1.3169, Validation Accuracy: 0.3750
EarlyStopping counter: 8 out of 25
Epoch 10/100, Training Loss: 0.4804, Training Accuracy: 0.6736, Validation Loss:
1.3503, Validation Accuracy: 0.2500
EarlyStopping counter: 9 out of 25
Epoch 11/100, Training Loss: 0.3620, Training Accuracy: 0.6894, Validation Loss:
1.4352, Validation Accuracy: 0.3750
EarlyStopping counter: 10 out of 25
Epoch 12/100, Training Loss: 0.3776, Training Accuracy: 0.7002, Validation Loss:
1.2938, Validation Accuracy: 0.3750
EarlyStopping counter: 11 out of 25
Epoch 13/100, Training Loss: 0.2847, Training Accuracy: 0.7147, Validation Loss:
1.2220, Validation Accuracy: 0.3125
EarlyStopping counter: 12 out of 25
Epoch 14/100, Training Loss: 0.3515, Training Accuracy: 0.7232, Validation Loss:
1.1768, Validation Accuracy: 0.3125
EarlyStopping counter: 13 out of 25
Epoch 15/100, Training Loss: 0.4746, Training Accuracy: 0.7343, Validation Loss:
1.2736, Validation Accuracy: 0.3750
EarlyStopping counter: 14 out of 25
Epoch 16/100, Training Loss: 0.4218, Training Accuracy: 0.7396, Validation Loss:
1.3938, Validation Accuracy: 0.3125
EarlyStopping counter: 15 out of 25
Epoch 17/100, Training Loss: 0.3831, Training Accuracy: 0.7443, Validation Loss:
1.5216, Validation Accuracy: 0.5000
EarlyStopping counter: 16 out of 25
Epoch 18/100, Training Loss: 0.3774, Training Accuracy: 0.7500, Validation Loss:
1.1826, Validation Accuracy: 0.1875
EarlyStopping counter: 17 out of 25
Epoch 19/100, Training Loss: 0.4978, Training Accuracy: 0.7515, Validation Loss:
1.2198, Validation Accuracy: 0.1875
EarlyStopping counter: 18 out of 25
Epoch 20/100, Training Loss: 0.3098, Training Accuracy: 0.7583, Validation Loss:
1.8223, Validation Accuracy: 0.5625
EarlyStopping counter: 19 out of 25
Epoch 21/100, Training Loss: 0.3779, Training Accuracy: 0.7593, Validation Loss:
1.4411, Validation Accuracy: 0.3750
EarlyStopping counter: 20 out of 25
Epoch 22/100, Training Loss: 0.1818, Training Accuracy: 0.7677, Validation Loss:
1.0734, Validation Accuracy: 0.4375
EarlyStopping counter: 21 out of 25
Epoch 23/100, Training Loss: 0.1813, Training Accuracy: 0.7748, Validation Loss:
1.2872, Validation Accuracy: 0.5625
```

```
EarlyStopping counter: 22 out of 25
Epoch 24/100, Training Loss: 0.1874, Training Accuracy: 0.7818, Validation Loss:
1.4807, Validation Accuracy: 0.4375
```

[ ]:
```python
# Save the model state_dict
torch.save(model.state_dict(), 'complex_model.pth')
```

[ ]:
```python
#change size of layers... let us see how this works
```

[ ]:
```python
def evaluate_model(loader, custom_labels, dataset_name):
    model.eval()  # Set the model to evaluation mode
    all_predictions = []
    all_labels = []

    with torch.no_grad():
        for inputs, labels in loader:
            try:
                #inputs = inputs.view(inputs.size(0), 4, 244, 244)  # Ensure␣
 ↪the input size is correct
                inputs = inputs.view(inputs.size(0), 3, 300, 600)  # Ensure the␣
 ↪input size is correct
                outputs = model(inputs)
                predictions = torch.argmax(outputs, dim=1)
                all_predictions.extend(predictions.tolist())
                all_labels.extend(labels.tolist())  # Populate the true labels
            except Exception as e:
                print(f"An error occurred: {e}")

    # Compute accuracy for the dataset
    if len(all_labels) > 0:
        accuracy = sum(p == l for p, l in zip(all_predictions, all_labels)) /␣
 ↪len(all_labels)
        print(f"{dataset_name} Accuracy: {accuracy:.1%}")
    else:
        print(f"No labels available for computing {dataset_name} accuracy.")

    # Analyze classification breakdown for the dataset
    for class_label, custom_label in enumerate(custom_labels):
        correct = sum(p == class_label and l == class_label for p, l in␣
 ↪zip(all_predictions, all_labels))
        total_in_set = all_labels.count(class_label)

        if total_in_set == 0:
            print(f"{custom_label}: No examples in the {dataset_name} set")
        else:
            percentage = correct / total_in_set if total_in_set != 0 else 0.0
```

```
            print(f"{custom_label}: Correctly classified {correct}/
 ↪{total_in_set} ({percentage:.1%})")

    # Return the lists of all_labels and all_predictions
    return all_labels, all_predictions


# Define custom labels
#custom_labels = ['Red', 'Blue', 'Green', 'Uncertain']
custom_labels = ['Red', 'Blue', 'Green']

# Evaluate on the validation set
all_labels_val, all_predictions_val = evaluate_model(val_loader, custom_labels,␣
 ↪"Validation")

# Evaluate on the test set
all_labels_test, all_predictions_test = evaluate_model(test_loader,␣
 ↪custom_labels, "\nTest")

# Compute confusion matrix for the test set
conf_matrix = confusion_matrix(all_labels_test, all_predictions_test)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',␣
 ↪xticklabels=custom_labels, yticklabels=custom_labels)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix - Test Set')
plt.show()

# Generate and print classification report
class_report = classification_report(all_labels_test, all_predictions_test,␣
 ↪target_names=custom_labels)
print("Classification Report - Test Set:\n", class_report)
```

[ ]:
```
from matplotlib.backends.backend_pdf import PdfPages
import matplotlib.pyplot as plt
import random
from torchvision import transforms
from PIL import Image

# Assuming you have already defined the CustomCNN class and loaded the model
# Also, define the 'model' and 'transform' variables before this code snippet

# Create a PdfPages object to store the plots
pdf_pages = PdfPages('activation_plots.pdf')
```

```
# List of directories to investigate
directory_paths = [
    '/Users/peternoble/Desktop/dog_sounds/training/0',
    '/Users/peternoble/Desktop/dog_sounds/training/1',
    '/Users/peternoble/Desktop/dog_sounds/training/2',
   # '/Users/peternoble/Desktop/dog_sounds/training/3',
]

# Custom labels for each directory
#custom_labels = ['Red', 'Blue', 'Green', 'Uncertain']
custom_labels = ['Red', 'Blue', 'Green']

# Loop through each directory
for i, directory_path in enumerate(directory_paths):
    # Choose a random image file from the list
    random_image_file = random.choice(os.listdir(directory_path))

    # Construct the full path to the randomly chosen image file
    random_image_path = os.path.join(directory_path, random_image_file)

    # Open the image using PIL
    random_image = Image.open(random_image_path)

    # Define the transformation for the image
    transform = transforms.Compose([
        #transforms.Resize((244, 244)),
        transforms.ToTensor(),
    ])

    # Apply the transformation to the image
    input_image = transform(random_image).unsqueeze(0)  # Add batch dimension

    # Create a new figure for each plot
    fig, ax = plt.subplots()

    # Create a list to store the activation statistics
    activation_stats = []

    # Define a hook to store the activation statistics at each layer
    def hook_fn(module, input, output):
        mean_activation = output.mean().item()
        var_activation = output.var().item()
        activation_stats.append((mean_activation, var_activation))

    # Register the hook to each layer in your model
    hooks = []
```

```
    for layer in model.children():
        hook = layer.register_forward_hook(hook_fn)
        hooks.append(hook)

    # Forward pass to obtain activation statistics
    with torch.no_grad():
        model(input_image)

    # Remove the hooks
    for hook in hooks:
        hook.remove()

    # Visualize activation statistics
    layer_names = [f'Layer {i+1}' for i in range(len(activation_stats))]
    mean_activations, var_activations = zip(*activation_stats)

    # Create separate x-values for mean and variance
    x_mean = [i for i in range(len(mean_activations))]
    x_var = [i + 0.2 for i in range(len(var_activations))]  # Add a small␣
↪offset for variance

    # Plot the activation statistics
    ax.bar(x_mean, mean_activations, width=0.4, label='Mean Activation')
    ax.bar(x_var, var_activations, width=0.4, label='Variance Activation',␣
↪alpha=0.9)
    ax.set_xlabel('Layers')
    ax.set_ylabel('Values')
    ax.set_title(f'Activation Statistics - {custom_labels[i]}')
    ax.legend()

    # Save the current plot to the PDF
    pdf_pages.savefig(fig)

    # Show the current plot on the screen
    plt.show()

    # Close the current plot
    plt.close()

# Close the PdfPages object
pdf_pages.close()
```

[33]:
```
# CustomCNN class definition goes here
from sklearn.decomposition import PCA

# Function to visualize PCA results with labels and explained variance
```

```
def visualize_pca(mean_activations, var_activations, layer_numbers,␣
 ↪directory_path):
    # Combine mean and variance into a single matrix
    activations_matrix = list(zip(mean_activations, var_activations))

    # Apply PCA
    pca = PCA(n_components=2)
    activations_pca = pca.fit_transform(activations_matrix)

    # Get the contribution of explained variances
    contribution_pc1 = pca.explained_variance_ratio_[0] * 100
    contribution_pc2 = pca.explained_variance_ratio_[1] * 100

    # Visualize PCA results with dots labeled by layer numbers
    plt.figure(figsize=(8, 8))

    # Annotate explained variance on axes
 #   plt.text(activations_pca[:, 0].max() + 0.1, 0, f'PC1 ({contribution_pc1:.
 ↪2f}%)', fontsize=10, ha='left', va='center')
 #   plt.text(0, activations_pca[:, 1].max() + 0.1, f'PC2 ({contribution_pc2:.
 ↪2f}%)', fontsize=10, ha='center', va='bottom')

    # Label dots with layer numbers
    for i, (x, y) in enumerate(activations_pca):
        plt.scatter(x, y, alpha=0)  # Make the dot invisible
        plt.text(x, y, str(layer_numbers[i]), fontsize=8, ha='center',␣
 ↪va='center')

    plt.title(f'PCA of Activation Statistics - {directory_path}')
    plt.xlabel('Principal Component 1 - Explained Variance: {:.2f}%'.
 ↪format(contribution_pc1))
    plt.ylabel('Principal Component 2 - Explained Variance: {:.2f}%'.
 ↪format(contribution_pc2))
    plt.show()

# List of directory paths
directory_paths = [
    '/Users/peternoble/Desktop/dog_sounds/training/0',
    '/Users/peternoble/Desktop/dog_sounds/training/1',
    '/Users/peternoble/Desktop/dog_sounds/training/2',
    #'/Users/peternoble/Desktop/dog_sounds/training/3',
]

# Initialize lists to accumulate activation statistics for all datasets
all_mean_activations = []
all_var_activations = []
```

```
# Initialize layer numbers
layer_numbers = []

for directory_path in directory_paths:
    model = CustomCNN()

    # ... (rest of the code remains unchanged)

    # Accumulate activation statistics for the current dataset
    all_mean_activations.extend(mean_activations)
    all_var_activations.extend(var_activations)

    # Add layer numbers for the current dataset
    layer_numbers.extend([i + 2 for i in range(len(mean_activations))])

    # Visualize PCA of activation statistics for each dataset
    visualize_pca(mean_activations, var_activations, layer_numbers,␣
 ↪directory_path)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[33], line 55
     50 model = CustomCNN()
     52 # … (rest of the code remains unchanged)
     53
     54 # Accumulate activation statistics for the current dataset
---> 55 all_mean_activations.extend(mean_activations)
     56 all_var_activations.extend(var_activations)
     58 # Add layer numbers for the current dataset

NameError: name 'mean_activations' is not defined
```

[ ]:
```
# Create a list to store the feature maps
feature_maps = []

# Define a hook to store the feature maps at each layer
def hook_fn(module, input, output):
    feature_maps.append(output)

# Register the hook to each layer in your model
hooks = []
for layer in model.children():
    hook = layer.register_forward_hook(hook_fn)
    hooks.append(hook)
```

```
# Forward pass to obtain feature maps
with torch.no_grad():
    model(input_image)

# Remove the hooks
for hook in hooks:
    hook.remove()

# Print the shape of the feature maps
for i, feature_map in enumerate(feature_maps):
    print(f'Layer {i + 1} - Shape: {feature_map.shape}')

# Visualize the feature maps from the 2nd layer
layer_to_visualize = 1  # 0-based index, corresponds to the 2nd layer
feature_map_normalized = feature_maps[layer_to_visualize].squeeze().numpy()

# Visualize the feature map
#plt.figure(figsize=(8, 8))
#plt.imshow(feature_map_normalized, cmap='viridis')
#plt.title(f'Layer {layer_to_visualize + 1} - Feature Map')
#plt.axis('off')
#plt.show()
```

[ ]:

[ ]: