Final project: Henshin

Object detection and speech recognition to select a birdsong using our fingers and control its volume using our voice.

Manuela Cleves Luna
Artificial Intelligence and Machine Learning

1. Abstract

This report presents a novel system enabling users to select one out of five British birdsongs by displaying the corresponding number of fingers and controlling the volume of the birdsong using speech commands. The system integrates two key components: object detection and speech recognition. Flexibility is ensured by accepting three types of inputs: still images, webcam videos, and live webcam streams. Initial project exploration involved multiple paths, with the final concept emerging after overcoming technical hurdles. The methodology section outlines the system's architecture, including model training, data preprocessing, and prediction processes. Experimentation focused on improving model accuracy through iterations on dataset adjustments, model architecture modifications, and hyperparameter tuning. The latest iteration utilizes a VGG16 model, achieving acceptable accuracy for real-life applications. Further development includes integrating sound playback and speech recognition functionalities, enabling seamless user interaction. Despite achieving promising results in finger counting and audio playback, challenges remain in adapting to diverse webcam conditions, leading to some inaccuracies. This report details the system's implementation, experimentation, and results, providing insights for future enhancements.

2. Introduction and Background

This system allows users to select one out of 5 British birdsongs by holding up the number of fingers that corresponds to each birdsong. Furthermore, users can control the volume of the birdsong that is playing with their speech by saying the words "up" or "down". This means that the ML system is composed of two key parts: 1) object detection and 2) speech recognition.

To give the user greater flexibility in how she or he uses the model, and to gain greater learning from exploring more potential outputs, it’s been designed to receive three possible inputs: 1) a still image taken with the webcam, 2) a video taken with the webcam, 3) a life webcam stream to be used in real-time.

Before reaching this project idea, several paths were explored and eventually pushed aside because of technical difficulties. The following diagram outlines major project ideas, for most of which substantial pieces of code were developed (*see Annex 1*).

	Details	Issues	Libraries explored
Bodypix Body Segmentation	Segment the body in images as well as in real-time to later overlay specific images over certain animal parts.	• Bodypix Body Segmentation package had become deprecated • Attempted to use older versions of Tensorflow and python but eventually ran into compatibility issues amongst libraries and packages	• TensorFlow.js (for running pre-trained BodyPix models) • BodyPix (JavaScript library for real-time person and body part segmentation)
React.JS Body Segmentation	Segment the body in images as well as in real-time to later overlay specific images over certain animal parts.	• Bodypix Body Segmentation package had become deprecated • Attempted to use older versions of Tensorflow and python but eventually ran into compatibility issues amongst libraries and packages	• TensorFlow.js (for running pre-trained BodyPix models) • React.js (JavaScript library for building user interfaces) • BodyPix (JavaScript library for real-time person and body part segmentation)
OpenCV Gesture Volume Control	Combine gesture control with object detection finger counting to select an audio to play (number of fingers) and then gesture control to increase/decrease the volume of that audio.	• Using fingers for both finger counting and volume control meant that one had to be done on the right hand and the other on the left hand. • The only dataset found that differentiated between left/right did not have realistic images and had very low accuracy.	• OpenCV (Computer vision library) • NumPy (Library for numerical computing with Python) • Mediapipe (Optional, for hand tracking and pose estimation)
Tensorflow Object Detection for Finger Counting	Segment the body in images as well as in real-time to later overlay specific images over certain animal parts.	• Had library incompatibility issues, had to switch to Google Colab to avoid issues with environment set up • Accuracy of the trained model was low	• TensorFlow (Deep learning library) • OpenCV (Computer vision library) • NumPy (Library for numerical computing with Python)
Mediapipe Finger Counting	Use a Media Pipe pretrained model to identify number of fingers and then train a complementary model for speech recognition	• Didn't have enough time to begin training a new type of model for speech recognition from scratch	• TensorFlow (Deep learning library) • MediaPipe (Framework for building cross-platform applied ML pipelines) • NumPy (Library for numerical computing with Python)
Pytorch (torch visión) to train a model to identify fingers in images	This final model is built on pytorch and uses torchvision for image transformation and model definition.	The greatest challenge was improving the model's accuracy. More detail on this is found in the experimentation section of this document.	• PyTorch (Deep learning library) • TorchVision (PyTorch's computer vision library) • NumPy (Library for numerical computing with Python)

Figure 1. Ideas explored during the project's initial phase

The last two ideas described in Figure 1 were fully developed and are explained in greater detail in the coming sections.

3. Methodology

a. General Logic

In the briefest terms possible, the system trains off a dataset that includes color images of people displaying various numbers of fingers on both left and right hands. Some images depict the full person, while others show only from the wrist up, and there is a diversity of backgrounds. The model trained on these images can identify the number of fingers appearing in an image. These labels range from 0 to 4, corresponding to numbers 1 to 5 respectively. More details on the model's components can be found below in Figure 2.

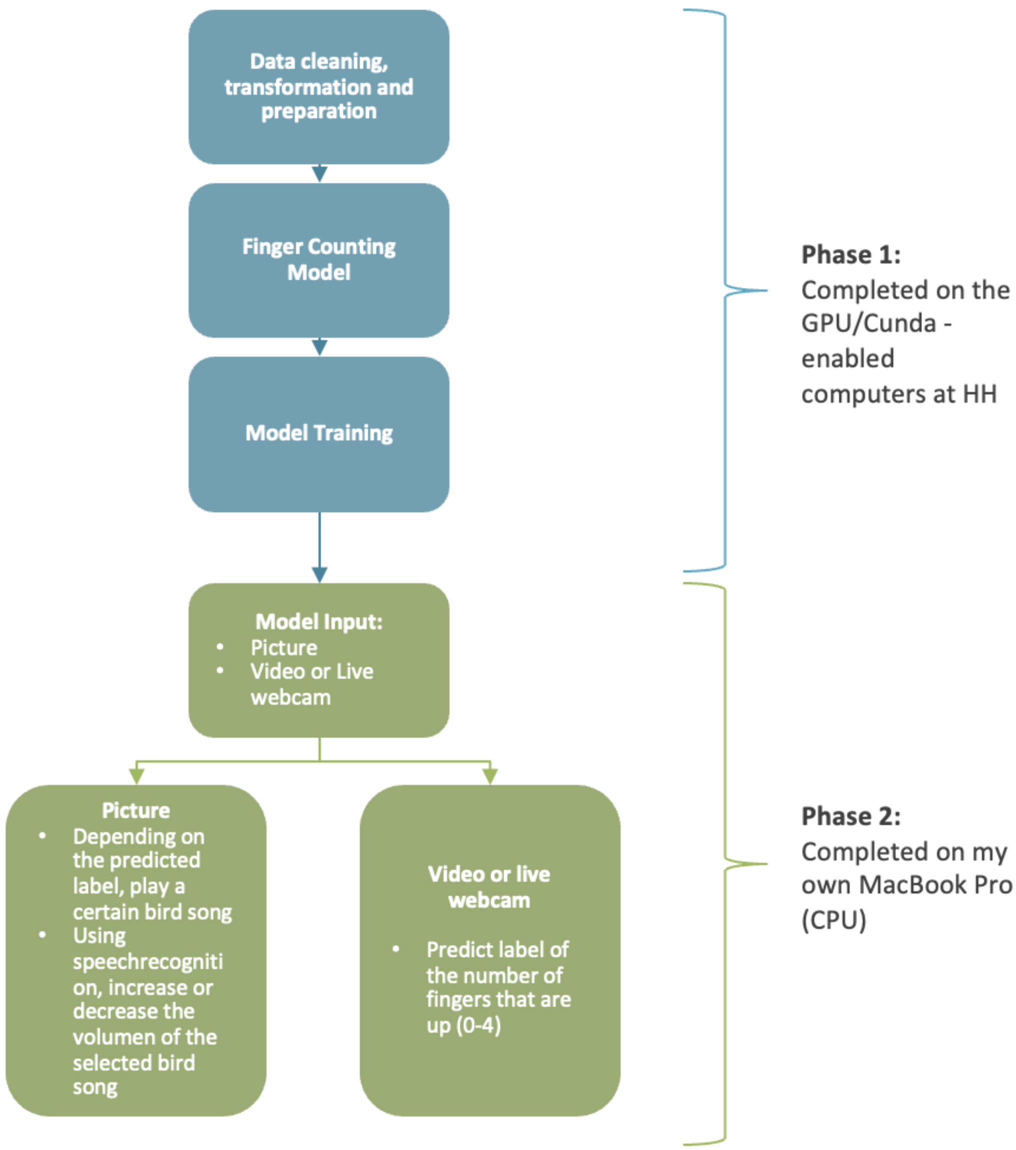


Figure 2. General process

b. System Components

The system has the following components:

- **Import Libraries:** This section imports all the necessary libraries and modules required for the rest of the code, including libraries for data manipulation, image processing, model building, and training.
- **Load Images:** In this part, images from the specified folders (train and test) are loaded into lists along with their corresponding labels. This process involves traversing through the folders, reading image files, and extracting labels from file names.
- **Transform Images and Prepare for Training:** Here, the loaded images are transformed using the Albumentations library to augment and preprocess them for training. Two sets of transformations, one for training and one for validation are defined to perform operations like flipping, cropping, resizing, and converting to tensors.
- **Create Model Classes:** This section defines custom model classes using PyTorch. Specifically, it implements an attention mechanism on top of a pre-trained VGG16 model. The attention blocks and the model architecture are defined, along with methods for resetting parameters and forward propagation.
- **Adjust Loss and Optimizer for Created Model:** In this part, the loss function and optimizer are defined for training the created model. The loss function is set to Cross Entropy Loss, and the optimizer is initialized with the Adam optimizer, targeting the parameters of the custom model.
- **Train Model on GPU:** Finally, this section handles the training loop where the model is trained on the GPU. It iterates through the specified number of epochs, running batches of training data through the model, calculating losses, and optimizing the model parameters using backpropagation. Additionally, it evaluates the model's performance on the validation dataset, tracking metrics such as loss and accuracy.
- **Save Model:** This section saves the trained model's state dictionary to a file.
- **Load Model on CPU:** Here, the model is loaded from the saved state dictionary onto the CPU and set to evaluation mode.
- **Capture Image:** This part captures an image or a video using a webcam and saves it as a file.
- **Make Prediction for an Image:** It makes a prediction for the captured image using the trained model.
    - Play sound that corresponds to Image prediction: Plays a sound corresponding to the predicted class of the captured image.
    - Launch voice recognition for volume control: Initiates voice recognition for adjusting volume based on speech commands.
    - Capture a Video: This section captures a video using a webcam and saves it as a file.
- **Make prediction for a video:** Make predictions for each frame of the captured video using the trained model.

- **Start live webcam**
  - This part initiates the live webcam feed and makes predictions for each frame in real time.
  - Make a prediction for live webcam: Predicts the class of objects seen through the webcam and displays the prediction on the video feed.

4. Experimentation:

Throughout experimentation, the greatest challenge was achieving accuracy.
The very first model I trained was grossly inaccurate with image inputs other than train/test data. A potential hypothesis for this was that although the accuracy appeared acceptable (approx. 85%) after training, the images were not lifelike enough for this to transfer to real-life webcam photos (all images were floating hands with black backgrounds).

Figure 4. Initial model's Accuracy

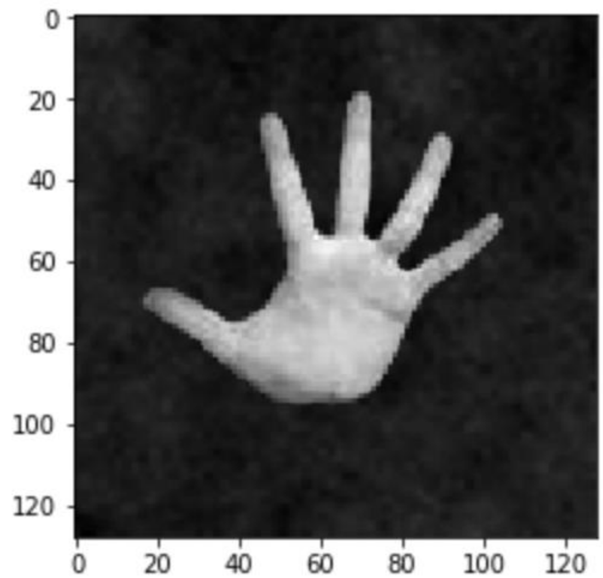


Figure 3. Example of data used to train the initial model

From that moment on, there were a series of iterations in an attempt to increase the model's real-life accuracy. These iterations revolved around adjustments to the data set, the model's architecture, the libraries used, hyperparameter tuning and the environment the model was running on. The following table describes each of these iterations.

Adjustment No.	Type of adjustment	Description	Accuracy
1	Model architecture	Tried adding complexity to the model by adding convolutional layers with increasing numbers of filters (64, 128, 256) to capture more complex patterns in the images. A fully Connected Layer with 512 neurons was also added to further increase the complexity of the model before the output layer.	84% (No change in real-time performance) (see Annex 1 for code)
2	Data set	Changed to new data set of more varied and realistic images with different backgrounds.	22% (see Annex 1 for code)
3	Model architecture & environment	Changed to a vgg16 model type on Google Colab to further increase models accuracy.	24% (see Annex 1 for code)
4	Hyperparameter tuning	Tried diverse adjustments in epoch, learning rate and batch size.	<=31%
5	Libraries used and environment	Create a new environment and replaced Tensor Flow with Image Classification (with Attention) using Pytorch. Increase model complexity using Pytorch's "Attention". Began running on Jupyter Lab.	88% (see *Results* section for greater detail and Annex 2 for code)

	precision	recall	f1-score	support
1	0.12	0.03	0.05	31
2	0.12	0.13	0.13	23
3	0.18	0.26	0.22	38
4	0.38	0.13	0.19	23
5	0.31	0.47	0.38	38
accuracy			0.23	153
macro avg	0.22	0.21	0.19	153
weighted avg	0.22	0.23	0.21	153

Figure 5. Accuracy after the second adjustment

Saved to photo.jpg

Saved to photo.jpg

Figure 6. Pictures depicting the inaccuracy after the 3rd adjustment (predicted labels appear at the bottom of each image)

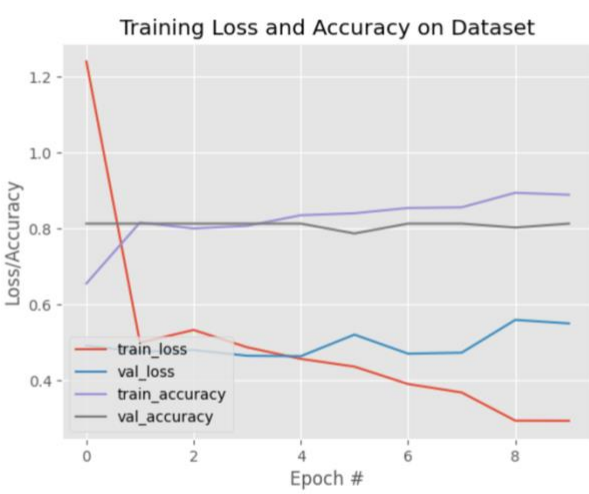
```
1/1 [==============================] - 0s 114ms/step
Predicted label: 4
```

```
1/1 [==============================] - 0s 131ms/step
Predicted label: 5
```

```python
# Initialize the Sequential model
model = Sequential()

model.add(Conv2D(input_shape=(IMG_SIZE,IMG_SIZE,3),filters=64,kernel_size=(3,3),padding="same", activation="relu"))
model.add(Conv2D(filters=64,kernel_size=(3,3),padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

# Flatten Layer
model.add(Flatten())

# Output Layer
model.add(Dense(6, activation='softmax'))  # Adjusted to the number of classes in your problem

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Figure 7 (above). Coded model after 3rd adjustment

Figure 8 (below). An example of accuracy reached while experimenting with hyperparameters



The latest version of the model leverages an existing VGG16 model. Full detail on this model, including its key code components, is found in Annex 2 of this document.

Once I reached the latest version of the model, I found the accuracy was acceptable enough and moved on to test it out on video, real-life webcam stream and later the audio generation and speech recognition part of the system. The next sections outline and explain how this part of the system works.

First, I imported the library for handling sound files and kept using the torch library for machine-learning tasks. By assigning each label to a sound filename stored in a dictionary The code then loads these sound files using pygame.mixer.Sound and store them in a dictionary. After predicting the class for a given image (detailed in the methodology), it triggers the playback of the corresponding sound using pygame.

Then, I defined a function for interacting with sound volume based on speech commands and making predictions while adjusting volume accordingly. The adjust_volume function initializes a speech recognizer from the speech_recognition library and listens for volume commands through the microphone. It adjusts the volume of the sound playback based on recognized commands such as "up" or "down". If the command is recognized, it adjusts the volume accordingly and prints a message indicating the change.

5. Results

Ultimately, my model was functioning for finger counting in images, videos, and a webcam live stream. When the input is an image, the system also successfully plays the birdsong that corresponds to each label and allows the user to activate a function for volume control with speech recognition.

The model still shows some inaccuracy when functioning, a potential hypothesis is that the training images (although they were cleaned, revised, and chosen to be realistic and diverse), did not train the model to adjust to my webcam's conditions. This is evidenced by the fact that the trained model has an accuracy of 88% on test data and yet proves to be accurate less than 50% of the time when it is tried out by a user. The images found on the next page outline the detailed results.

For the next stage, accuracy could continue improving (particularly for the video and live webcam functions). Also, the audio and speech recognition elements could be incorporated into the video and live webcam functions.

Result for a picture taken with a webcam (accurately predicts label 0 – 1 finger):

```
Image captured successfully!
Predicted Class: 0
Probabilities: [[1. 0. 0. 0. 0.]]
```

Figure 6. Results for a prediction with an image input

- The next function in the code accurately plays 'sound0.wav', birdsong associated with label “0”
- The next function accurately recognizes that the word “up” means the volume is to be increased while the word “down” means the volume should be decreased

Results for a video taken with a webcam (model cannot predict accurately):

```
Recording stopped!
Predicted Classes: []
Probabilities: []
```

Figure 7. Results for a prediction with a video input

Results for a live webcam stream (accurately predicts label 0 – 1 finger):

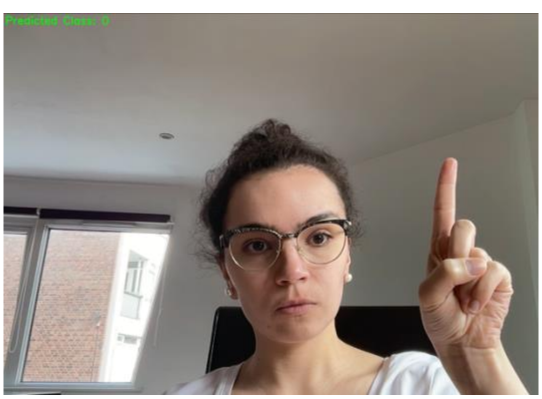


Figure 8. Results for a prediction with a live feed input

Prediction is found on the top right-hand corner of the screenshot.

6. References

- Brownlee, J. (n.d.). Difference Between a Batch and an Epoch in Neural Networks. Retrieved from https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/
- Paperspace Blog. (n.d.). Image Classification with Attention. Retrieved from https://blog.paperspace.com/image-classification-with-attention/
- Google. (n.d.). MediaPipe Hands. Retrieved from https://github.com/google/mediapipe/blob/master/docs/solutions/hands.md
- Stack Overflow. (n.d.). Loading All Images Using imread from a Given Folder. Retrieved from https://stackoverflow.com/questions/30230592/loading-all-images-using-imread-from-a-given-folder
- Chauhan, M. (n.d.). Hand Fingers Detection CNN Tensorflow Keras. Retrieved from https://github.com/chauhanmahavir/Hand-Fingers-Detection-CNN-Tensorflow-Keras/blob/master/Fingers_Detection_CNN_Tensorflow_Keras.ipynb
- PyTorch. (n.d.). Saving and Loading Models. Retrieved from https://pytorch.org/tutorials/beginner/saving_loading_models.html
- Paperspace Blog. (n.d.). Image Classification with Attention. Retrieved from https://blog.paperspace.com/image-classification-with-attention/
- Tatman, R. (n.d.). British Birdsong Dataset. Retrieved from https://www.kaggle.com/datasets/rtatman/british-birdsong-dataset
- Zakitaleb. (n.d.). Selfie Finger Counting Dataset. Retrieved from https://www.kaggle.com/datasets/zakitaleb/selfie-finger-counting?resource=download

Annex 1. How key components of the model definition changed while experimenting (shows an increase in complexity and adjustments in how images are read)

7. First model (developed on VS Code)

Define Model using tensorflow keras

```
Add layers to add complexity (copy chunks and add more convolusionar layers, aswell as activation) use vgg16 use colab

del = Sequential()

First Layer
del.add(   Conv2D(64,  (3, 3), input_shape = (IMG_SIZE, IMG_SIZE, 3))    )
del.add( Activation('relu') )
del.add( MaxPool2D(pool_size = (2,2)) )

Second Layer
del.add(   Conv2D(64,  (3, 3))    )
del.add( Activation('relu') )
del.add( MaxPool2D(pool_size = (2,2)) )

Third Layer
del.add(Flatten())
del.add(Dense(64))
del.add( Activation('relu') )

Output Layer
del.add(Dense(12))
del.add(Activation('sigmoid'))
```

8. Second model (developed on google colab, the same data preparation process was conducted on training data)

```
# Initialize the Sequential model
model = Sequential()

model.add(Conv2D(input_shape=(IMG_SIZE,IMG_SIZE,3),filters=64,kernel_size=(3,3),padding="same", activation="relu"))
model.add(Conv2D(filters=64,kernel_size=(3,3),padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

# Flatten Layer
model.add(Flatten())

# Output Layer
model.add(Dense(6, activation='softmax'))  # Adjusted to the number of classes in your problem

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Annex 2. Image preparation and model classes created for final model

```
# Initialize empty lists to store images and labels
TestFilesList = []
TestLabelsList = []

# Go through files in Test folder
for file_name in os.listdir(TestImagePaths):
    file_path = os.path.join(TestImagePaths, file_name)

    # Read the image file
    image = cv2.imread(file_path)

    # Check if the image is loaded successfully
    if image is not None:
        # Resize the image
        image = cv2.resize(image, (IMG_SIZE, IMG_SIZE))
#double size of dataset by taking an image, flipping it
        # Append the resized image and its label to the data and labels lists
        TestFilesList.append(image)
        TestLabelsList.append(file_path.split(os.path.sep)[-1][-5])
    else:
        print(f"Error loading image: {file_path}")

# Convert data and labels lists to NumPy arrays
TestData = np.array(TestFilesList, dtype="float") / 255.0
TestLabels = np.array(TestLabelsList)

# Encode the labels
le = LabelEncoder()
EncodeTestLabels = to_categorical(TestLabelsList, num_classes=6)

# Print the number of images and labels after resizing and encoding
print(f"Number of resized images: {len(TestData)}")
print(f"Number of encoded labels: {len(TestLabels)}")
```

Load Images

[2]:
```
#Load images from each folder (test and train)
TrainImagePaths = "./V3Data/train"
TestImagePaths = "./V3Data/test"
```

[3]:
```
def load_Trainimages_from_folder(folder):
    TrainImages = []
    TrainLabelsList=[]
    for file in os.listdir(folder):
        TrainImages.append(os.path.join(folder,file))
        TrainLabelsList.append(int(file.split(os.path.sep)[-1][-5])-1)
    return TrainImages, TrainLabelsList
```

[4]:
```
def load_Testimages_from_folder(folder):
    TestImages = []
    TestLabelsList=[]
    for file in os.listdir(folder):
        TestImages.append(os.path.join(folder,file))
        TestLabelsList.append(int(file.split(os.path.sep)[-1][-5])-1)
    return TestImages, TestLabelsList
```

Transform Images and Prepare for Training

[2]:
```
train_transform = A.Compose(
    [
        A.HorizontalFlip(p=0.5),
        A.SmallestMaxSize(max_size=256),
        A.ShiftScaleRotate(shift_limit=0.05, scale_limit=0.05, rotate_limit=60, p=0.5),
        A.RandomCrop(height=224, width=224),
        A.RGBShift(r_shift_limit=5, g_shift_limit=5, b_shift_limit=5, p=0.5),
        A.RandomBrightnessContrast(p=0.5),
        ToTensorV2(),
    ]
)

val_transform = A.Compose(
    [
        A.SmallestMaxSize(max_size=256),
        A.CenterCrop(height=224, width=224),
        ToTensorV2(),
    ]
)
```

[ ]:
```
TrainImages, TrainLabels = load_Trainimages_from_folder(TrainImagePaths)
TestImages, TestLabels = load_Testimages_from_folder(TestImagePaths)
```

Create Model Classess

[3]:
```
class ImageDataset(Dataset):

    def __init__(self,data_paths,labels,transform=No
         self.data=data_paths
         self.labels=labels
         self.transform=transform
         self.mode=mode
    def __len__(self):
       return len(self.data)

    def __getitem__(self,idx):
        img_name = self.data[idx]
        img = cv2.imread(img_name)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        # img=Image.fromarray(img)
        # if self.transform is not None:
        img = self.transform(image=img)["image"]/255.
        img = img.cuda()

        labels = torch.tensor(self.labels[idx]).cuda()

        return img, labels
```

Name: V3Data
Path: Documents/ML&AI/Henshim/
V3Notebook&Model
Created: 3/13/24, 2:22 PM
Modified: 3/13/24, 2:22 PM
Writable: true

[ ]:
```
train_dataset=ImageDataset(data_paths=TrainImages,labels=TrainLabels,transform=train_transform)
val_dataset=ImageDataset(data_paths=TestImages,labels=TestLabels,transform=val_transform)

train_loader=DataLoader(train_dataset,batch_size=16,shuffle=True)
val_loader=DataLoader(val_dataset,batch_size=1,shuffle=False)
```

[4]:

```python
# Implementing attention layer

class AttentionBlock(nn.Module):
    def __init__(self, in_features_l, in_features_g, attn_features, up_factor, normalize_attn=True):
        super(AttentionBlock, self).__init__()
        self.up_factor = up_factor
        self.normalize_attn = normalize_attn
        self.W_l = nn.Conv2d(in_channels=in_features_l, out_channels=attn_features, kernel_size=1, padding=
        self.W_g = nn.Conv2d(in_channels=in_features_g, out_channels=attn_features, kernel_size=1, padding=
        self.phi = nn.Conv2d(in_channels=attn_features, out_channels=1, kernel_size=1, padding=0, bias=True
    def forward(self, l, g):
        N, C, W, H = l.size()
        l_ = self.W_l(l)
        g_ = self.W_g(g)
        if self.up_factor > 1:
            g_ = F.interpolate(g_, scale_factor=self.up_factor, mode='bilinear', align_corners=False)
        c = self.phi(F.relu(l_ + g_)) # batch_sizex1xWxH

        # compute attn map
        if self.normalize_attn:
            a = F.softmax(c.view(N,1,-1), dim=2).view(N,1,W,H)
        else:
            a = torch.sigmoid(c)
        # re-weight the local feature
        f = torch.mul(a.expand_as(l), l) # batch_sizexCxWxH
        if self.normalize_attn:
            output = f.view(N,C,-1).sum(dim=2) # weighted sum
        else:
            output = F.adaptive_avg_pool2d(f, (1,1)).view(N,C) # global average pooling
        return a, output
```

```python
class AttnVGG(nn.Module):
    def __init__(self, num_classes, normalize_attn=False, dropout=None):
        super(AttnVGG, self).__init__()
        net = models.vgg16_bn(pretrained=True)
        self.conv_block1 = nn.Sequential(*list(net.features.children())[0:6])
        self.conv_block2 = nn.Sequential(*list(net.features.children())[7:13])
        self.conv_block3 = nn.Sequential(*list(net.features.children())[14:23])
        self.conv_block4 = nn.Sequential(*list(net.features.children())[24:33])
        self.conv_block5 = nn.Sequential(*list(net.features.children())[34:43])
        self.pool = nn.AvgPool2d(7, stride=1)
        self.dpt = None
        if dropout is not None:
            self.dpt = nn.Dropout(dropout)
        self.cls = nn.Linear(in_features=512+512+256, out_features=num_classes, bias=True)

       # initialize the attention blocks defined above
        self.attn1 = AttentionBlock(256, 512, 256, 4, normalize_attn=normalize_attn)
        self.attn2 = AttentionBlock(512, 512, 256, 2, normalize_attn=normalize_attn)


        self.reset_parameters(self.cls)
        self.reset_parameters(self.attn1)
        self.reset_parameters(self.attn2)
    def reset_parameters(self, module):
        for m in module.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_in', nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0.)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1.)
                nn.init.constant_(m.bias, 0.)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0., 0.01)
                nn.init.constant_(m.bias, 0.)
    def forward(self, x):
```

```python
                nn.init.constant_(m.bias, 0.)
    def forward(self, x):
        block1 = self.conv_block1(x)       # /1
        pool1 = F.max_pool2d(block1, 2, 2) # /2
        block2 = self.conv_block2(pool1)   # /2
        pool2 = F.max_pool2d(block2, 2, 2) # /4
        block3 = self.conv_block3(pool2)   # /4
        pool3 = F.max_pool2d(block3, 2, 2) # /8
        block4 = self.conv_block4(pool3)   # /8
        pool4 = F.max_pool2d(block4, 2, 2) # /16
        block5 = self.conv_block5(pool4)   # /16
        pool5 = F.max_pool2d(block5, 2, 2) # /32
        N, __, __, __ = pool5.size()

        g = self.pool(pool5).view(N,512)
        a1, g1 = self.attn1(pool3, pool5)
        a2, g2 = self.attn2(pool4, pool5)
        g_hat = torch.cat((g,g1,g2), dim=1) # batch_size x C
        if self.dpt is not None:
            g_hat = self.dpt(g_hat)
        out = self.cls(g_hat)

        return [out, a1, a2]
```

```python
model = AttnVGG(num_classes=5, normalize_attn=True)
```