Documentation and Code Explanation of Live ASL Detection Project

Introduction

The American Sign Language (ASL) Detection project is a real-time system that detects hand gestures and converts them into corresponding ASL alphabets. This project is based on deep learning techniques, specifically using a convolutional neural network (CNN) for image recognition.

The goal of this project is to provide an accurate and efficient ASL detection system that can be used in a variety of applications. Some possible applications include improving communication between the hearing-impaired and non-hearing-impaired, assisting in sign language education, and enhancing accessibility in public spaces.

This documentation and code explanation will provide an overview of how the ASL detection system works, its architecture and algorithms, as well as instructions for installing and using the system. Additionally, it will explain the code structure, the functions of each file, and the variables and parameters used throughout the project.

Related Concepts

Here are some related concepts for the American Sign Language Detection project along with short descriptions:

1. **Convolutional Neural Network (CNN):** A type of neural network commonly used for image classification and recognition. CNNs consist of multiple layers of convolutional filters that can identify patterns and features within an image.

2. **Transfer learning**: A technique where a pre-trained model is used as a starting point for a new model. This allows the new model to benefit from the knowledge and experience gained by the pre-trained model and can result in faster training and better performance.

3. **Fine tuning**: A process where a pre-trained model is further trained on new data, often with a smaller learning rate, to improve its performance on a specific task.

4. **TensorFlow**: An open-source library for dataflow and differentiable programming across a range of tasks. TensorFlow is commonly used for building and training machine learning models.

5. **OpenCV**: An open-source computer vision library that provides tools for image and video processing, feature detection, and object recognition.

6. **Mobilenet model**: A type of neural network architecture designed specifically for mobile and embedded devices. Mobilenet models are typically smaller and faster than other CNN architectures, making them well-suited for real-time applications like the ASL detection system.

These concepts are all relevant to the development and implementation of the ASL detection system, and a deeper understanding of each can help in optimizing and improving the system.

Required Library

Here are short descriptions of some of the libraries used in the American Sign Language Detection project:

1. **NumPy**: A library for working with arrays and matrices in Python. NumPy provides fast mathematical operations on large sets of data, making it useful for data analysis and scientific computing.

2. **Matplotlib**: A data visualization library for Python. Matplotlib provides tools for creating charts, graphs, and other types of visualizations from data.

3. **OpenCV**: An open-source computer vision library that provides tools for image and video processing, feature detection, and object recognition. OpenCV is commonly used for real-time computer vision applications.

4. **CvZone**: A library that provides tools for computer vision and image processing, including hand tracking and pose estimation.

5. **TensorFlow**: An open-source library for dataflow and differentiable programming across a range of tasks. TensorFlow is commonly used for building and training machine learning models.

6. **Keras**: A high-level neural network API for Python. Keras provides a user-friendly interface for building and training neural networks, and is compatible with TensorFlow and other deep learning libraries.

7. **Mediapipe**: An open-source framework for building real-time, cross-platform computer vision pipelines. Mediapipe provides a wide range of customizable tools for processing and analyzing video and image data, including object detection, hand tracking, face detection, and pose estimation.

These libraries are all used in different parts of the ASL detection system, from processing and analyzing data to building and training neural networks. Understanding their capabilities and functionalities can help in optimizing and improving the system.

Working Procedure

0) **Install required library:**
A file named requirements.txt consists of required librarys.
You can install it by command:

```
pip install -r requirements.txt
```

1) **Data Collection:**
First step is to collect data for training model. First we created a folder “dataset_2” and inside that a folder “train”. Inside the train folder we created 26 folder labeled A to Z. We will keep every image to the corresponding folder. Every images size will be 300X300. We will use opencv to capture image from camera and then detect hand from image with cv_zone library to process and save that image to corresponding folder by pressing corresponding key.

File: data-collection.py

Code Explanation:

```
# importing all necessary library
import cv2
import os
from cvzone.HandTrackingModule import HandDetector
import numpy as np
from math import ceil
```

This line imports the necessary libraries for the script to run, including OpenCV (cv2), operating system (os), the HandDetector class from the cvzone.HandTrackingModule, NumPy (numpy), and the ceil function from the math module.

```
# get video capture object (moreover the camera)
cap = cv2.VideoCapture(0)
```

This line initializes the video capture object 'cap' and assigns it the value returned by cv2.VideoCapture(0), which opens the default camera on the device.

```
# Initialize hand ditector, ditect maximum one hand
hand_ditector = HandDetector(maxHands=1)
```

This line initializes an instance of the HandDetector class with a maximum of one hand to detect.

```
# this offset is nothing but margin in bounding box
offset = 20
```

This line sets the variable 'offset' to 20, which is used to create a margin in the bounding box around the hand.

```
# directory of saved data
directory='dataset_2/train/'
```

This line sets the variable 'directory' to 'dataset_2/train/', which is the directory where the training data will be saved. This directory should already exist on the file system.

```
# A infinity loop to continiously capture image and process
while True:
    # capture image from camera returns two thing first one is boolean variable and second one is image
    success,img = cap.read()
```

An infinite loop to continuously capture images.
And below while-loop line captures an image from the camera using the video capture object created earlier and stores the result in two variables: 'success' (a boolean value indicating if the capture was successful or not) and 'img' (the captured image itself).

```
# count image on each directory
    count = {
             'a': len(os.listdir(directory+"/A")),
             .
             .
             .
             'z': len(os.listdir(directory+"/Z"))
             }
```

This section of code counts the number of images that have been saved in each subdirectory of the 'directory' variable, which is the location where the training data is being saved. It creates a dictionary called 'count' that maps each letter of the alphabet to the number of images that have been saved for that letter.

```
# track count of image by putting text on the frame
    cv2.putText(img, "a : "+str(count['a']), (10, 100), cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
    .
    .
    cv2.putText(img, "z : "+str(count['z']), (10, 340), cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
```

This code snippet is adding text on an image to display the count of various characters ('a' to 'z') stored in the dictionary variable 'count'. The text is added using the **cv2.putText()** function from the OpenCV library.
Each line of the code adds text to the image for a particular character along with its count. The function takes parameters in the following order:

- **img**: the image on which the text is to be added.
- The string to be displayed on the image, which includes the character and its count.
- The position where the text is to be placed on the image, which is specified as a tuple of (x,y) coordinates.
- The font used for the text.
- The size of the font.
- The color of the text.
- The thickness of the text.

```
 # Ditect hand from video
    hands = hand_ditector.findHands(img,draw= False)
    if hands: # if hand ditected
        hand = hands[0] # get first hand from hands list
        x,y,w,h = hand["bbox"] # get location (x,y) and width , height (w,h) of bounding box of hand
        # defining crop position
        py1,py2,px1,px2 = y-offset,y+h+offset,x-offset,x+w+offset
        # recalculate position in case of negative value
        py1 = 0 if py1<0 else py1
        px1 = 0 if px1<0 else px1
        # croped frame of ony hand
        croped_frame = img[py1:py2,px1:px2]
```

- The first line calls the **findHands()** function of the **hand_ditector** object to detect any hands in the **img** (presumably a single video frame) and returns a list of dictionaries, with each dictionary containing information about a detected hand.
- The **if hands:** statement checks if any hands were detected. If no hands were detected, the program skips the code block inside the **if** statement and proceeds to the next part of the program.

- If a hand was detected, the program selects the first hand in the list (since there may be multiple hands detected) and retrieves its bounding box coordinates **x**, **y**, **w**, and **h**. The bounding box is a rectangle that encloses the detected hand.
- The next few lines define the cropping position to crop out a rectangular region of the video frame that only contains the hand. The **offset** variable is presumably some distance in pixels to add to the bounding box coordinates to ensure that the cropped region extends slightly beyond the boundaries of the hand. The **py1**, **py2**, **px1**, and **px2** variables define the top, bottom, left, and right edges of the cropped region.
- The program checks if the cropped region extends beyond the boundaries of the original image and adjusts the cropping coordinates if necessary to ensure that they do not have negative values or exceed the dimensions of the original image.

```
    # Resize screen for appropriate image size
    croped_frame = cv2.resize(croped_frame,(244,244))
    croped_frame = cv2.cvtColor(croped_frame, cv2.COLOR_BGR2GRAY)
    # Merge the grayscale image with the original image's RGB channels
    croped_frame = cv2.cvtColor(croped_frame, cv2.COLOR_GRAY2BGR)
    cv2.imshow("Croped frame",croped_frame)
```

1. It resizes the **croped_frame** image to a size of 244x244 pixels using **cv2.resize()**.
2. It converts the **croped_frame** image from the BGR color space to grayscale using **cv2.cvtColor()** with the **cv2.COLOR_BGR2GRAY** flag. Grayscale images have only one color channel representing the intensity of each pixel.
3. Next, the code merges the grayscale image with the original image's RGB channels. To achieve this, it converts the **croped_frame** image back to the BGR color space using **cv2.cvtColor()** with the **cv2.COLOR_GRAY2BGR** flag. This step is necessary because the model you are using expects input images with three color channels (RGB) instead of just one (grayscale).
4. Finally, the code displays the resulting **croped_frame** image in a window titled "Croped frame" using **cv2.imshow()**.

The purpose of converting the hand sign portion of the image to grayscale and then back to BGR with grayscale colors is to provide a consistent input format for the model to recognize hand signs regardless of the skin tone variation. By converting the image to grayscale, the code removes the color information related to skin tone, focusing only on the intensity variations in the hand sign itself.

```
# wait 10ms to get key pressed event from keyboard
    interrupt = cv2.waitKey(10)
    # when a key pressed save current whiteframe(fixed size hand gesture image
    # ) to defined related path
    if interrupt & 0xFF == ord('a'):
      cv2.imwrite(directory+'A/'+str(count['a'])+'.png',croped_frame)
    .
    .
    .
    if interrupt & 0xFF == ord('z'):
      cv2.imwrite(directory+'Z/'+str(count['z'])+'.png',croped_frame)
```

This code block is waiting for a key pressed event from the keyboard for 10ms using the **cv2.waitKey(10)** method. Once a key is pressed, it checks whether the key is one of the letters **a** to **z**. If the pressed key is one of the letters, the current white frame (fixed size hand gesture image) is saved to a specific directory with the corresponding letter as the filename, and a count is incremented for that letter. For example, if the user pressed the letter 'a', the current white frame will be saved in a subdirectory named **A** in the defined directory, with the filename **count['a'].png**. The **count** dictionary keeps track of how many images have been saved for each letter, so that each image has a unique filename.

```
cv2.imshow("Hand Ditector",img)
```

This line show main image captured from camea. As this whole code is in an infinity while loop, so we will see this as video because it will capture and show many image in a second.

2) Data Processing and training:

In this section we will use Tensorflow library for processing and train data. We will use MobileNetV3Small Model as base model. And then we will modify it train with train data. Validation set will consists 20% of data from dataset. We will train this in Jupiter Notebook so you must add Jupyter extension to your vs code and install ipy-kernel to your python environment.

File: training.ipynb

Code Explanation:

```
import os
import shutil
import random
```

Importing necessary library for preparing training and validation dataset. Moreover for dividing dataset to 80% and 20% data for training and validation.

```
# creating folders of validation set
os.chdir("./dataset_gray_244/val")
for i in range(ord("A"),ord("Z")+1):
    os.mkdir(chr(i))
os.chdir("../")
```

This part of the code is creating folders for the validation set. But at first a folder named “val” must be created inside of “dataset_2” folder. The code first changes the working directory to the validation set directory using **os.chdir** function. Then it creates a loop that goes through all the letters of the English alphabet, from "A" to "Z", using **range(ord("A"),ord("Z")+1)** and creates a directory with that letter as its name using **os.mkdir(chr(i))**. Finally, the code changes the working directory back to the main dataset directory using **os.chdir("../")**.

By creating these folders, the code is preparing the dataset for the validation set, which is used to evaluate the performance of the trained machine learning model. The validation set is a separate subset of the dataset that is not used during training, but is used to check the performance of the model on unseen data. By creating these folders, the code is organizing the validation set data into separate folders, each containing images of a specific letter, making it easier to use the validation set for evaluation purposes.

```
# transfering 20% of image to validation set
for i in range(ord("A"),ord("Z")+1):
    # select random 5 image out of 50 image
    validation_set = random.sample(os.listdir(f"train/{chr(i)}"),5)
    for j in validation_set:
        shutil.move(f"train/{chr(i)}/{j}",f"val/{chr(i)}")
```

This block of code randomly selects 5 images from each class (A to Z) in the training set and transfers them to the corresponding class folder in the validation set. It does this by first looping through each class, selecting 5 images at random using the **random.sample()** function, and then moving these images from the **train** directory to the **val** directory using the **shutil.move()** function. The end result is that 20% of the images for each class are moved to the validation set for model evaluation.

1. **range(ord("A"),ord("Z")+1)**: This creates a range object that starts at the ASCII code for "A" and goes up to the ASCII code for "Z". The **ord()** function converts a character to its corresponding ASCII code.
2. **os.listdir(f"train/{chr(i)}")**: This returns a list of filenames in the directory "train/{chr(i)}". **chr()** function converts an ASCII code to its corresponding character.
3. **random.sample(os.listdir(f"train/{chr(i)}"),5)**: This returns a list of 5 random filenames from the list of filenames in "train/{chr(i)}".
4. **shutil.move(f"train/{chr(i)}/{j}",f"val/{chr(i)}")**: This moves the file with name **j** from directory **train/{chr(i)}** to directory **val/{chr(i)}**. This is done for each of the 5 randomly selected files for each letter from A to Z.

```
# come back to main directory
os.chdir("../")
```
Come back to main directory from dataset_2 directory

```
# defining train and validation dataset directory
train_path = "dataset_gray_244/train"
valid_path = "dataset_gray_244/val"
```
Defining train and validation path which will be needed for preparing and processing data for training purpose.

```
import tensorflow as tf
import numpy as np
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.mobilenet import preprocess_input,MobileNet,decode_predictions
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
```
Importing all necessary library and module for ASL ditection model training

```
train_batches =
ImageDataGenerator(preprocessing_function=preprocess_input,brightness_range=[0.8,1.5]).flow_from_directory(
    directory = train_path, target_size=(224,224), batch_size=10,class_mode="categorical"
)
valid_batches =
ImageDataGenerator(preprocessing_function=preprocess_input,brightness_range=[0.8,1.5]).flow_from_directory(
    directory = valid_path, target_size=(224,224), batch_size=10,class_mode="categorical"
)
```

This code is using the Keras **ImageDataGenerator** class to generate batches of images for training and validation. Here is a breakdown of the code:

1. **train_batches**:
    - An **ImageDataGenerator** object is created with two arguments:
        - **preprocessing_function=preprocess_input**: This specifies a function to be applied to each image before it is fed into the model. The **preprocess_input** function is likely a preprocessing function specific to the model being used. It could involve normalization or any other preprocessing steps required by the model.
        - **brightness_range=[0.8,1.5]**: This specifies the range of brightness values for the randomly adjusted brightness of the images. The brightness of each image in the training set will be randomly adjusted within this range.
    - The **flow_from_directory()** method is then called on the **ImageDataGenerator** object. This method generates batches of images and their labels by automatically scanning a directory structure. The following arguments are provided to the method:
        - **directory=train_path**: This specifies the directory path where the training images are located. The images are expected to be organized into subdirectories, where each subdirectory represents a different class.
        - **target_size=(224,224)**: This specifies the size to which the images will be resized during the loading process. All images will be resized to a width of 224 pixels and a height of 224 pixels.
        - **batch_size=10**: This specifies the number of images per batch.
        - **class_mode="categorical"**: This specifies the type of labels that will be generated. In this case, the labels will be encoded as categorical vectors.
2. **valid_batches**:
    - Similar to **train_batches**, an **ImageDataGenerator** object is created with the same arguments: **preprocessing_function** and **brightness_range**.
    - The **flow_from_directory()** method is called on this object with the following arguments:
        - **directory=valid_path**: This specifies the directory path where the validation images are located. Like the training images, the validation images should also be organized into subdirectories based on their classes.
        - **target_size=(224,224)**: This specifies the size to which the validation images will be resized.
        - **batch_size=10**: This specifies the number of images per batch.
        - **class_mode="categorical"**: This indicates that the labels will be encoded as categorical vectors, similar to the training set.

Overall, this code sets up generators (**train_batches** and **valid_batches**) that can provide batches of preprocessed images and their corresponding labels for training and validation, respectively. These generators make it easier to work with large datasets, as they load and preprocess the data on-the-fly during training, reducing the memory requirements and increasing efficiency.

.

```
# get mobile_net_v3_small model with input shape (300,300,3)
mobilenet_model = tf.keras.applications.MobileNetV2(input_shape=(244,244,3))

# view all layer of mobilenet model and its parameters
mobilenet_model.summary()
```

In these lines of code, we are using the MobileNetV3Small model from the TensorFlow Keras library for our image classification task.

Firstly, we import the MobileNetV3Small model from the Keras application and specify the input shape of our images to be (300,300,3) which means the input images will have a height and width of 300 pixels and 3 color channels (RGB).

Finally, we use the **summary()** method to print a summary of the model's layers, output shapes, and parameters. This helps us to see the structure of the model and the number of parameters that it contains.

```
x = mobilenet_model.layers[-2].output
output = Flatten()(x)
output = Dense(320, activation='relu')(output)
output = Dropout(0.2)(output)
output = Dense(26,activation="softmax")(output)

# Build a new model using transfer learing apporch from mobilenet model
model = Model(inputs=mobilenet_model.input, outputs=output)
```

This code involves building a new model using transfer learning with the MobileNet model as a base. Here is an explanation of each line:

1. **x = mobilenet_model.layers[-2].output**: This line retrieves the output of the second-to-last layer of the MobileNet model. It assigns this output to the variable **x**. By accessing this layer's output, we can use it as a starting point for our new model.

2. **output = Flatten()(x)**: Here, the **Flatten** layer is applied to the **x** output. This layer flattens the input, converting it from a multidimensional tensor to a single-dimensional tensor. This is necessary because the subsequent layers expect a one-dimensional input.

3. **output = Dense(320, activation='relu')(output)**: This line adds a fully connected **Dense** layer with 320 units to the model. The **relu** activation function is applied to the output of this layer. The **output** from the previous layer is used as the input to this layer.

4. **output = Dropout(0.2)(output)**: Here, a **Dropout** layer is added to the model. The purpose of the dropout layer is to randomly set a fraction of input units to 0 at each update during training, which helps prevent overfitting. In this case, a dropout rate of 0.2 (20%) is specified.

5. **output = Dense(26, activation="softmax")(output)**: This line adds another fully connected **Dense** layer with 26 units, corresponding to the number of classes in the desired classification task. The **softmax** activation function is used here, which normalizes the output into a probability distribution over the classes, allowing the model to make predictions for each class.

6. **model = Model(inputs=mobilenet_model.input, outputs=output)**: Finally, a new model is created using the **Model** class from Keras. It takes the **input** from the MobileNet model (the initial input layer) and the **output** tensor that we have constructed as arguments. This way, we create a new model that starts with the input of the MobileNet model and ends with our custom output layer. This is the model we will train for our specific task using transfer learning.

By using transfer learning, we leverage the pre-trained MobileNet model's knowledge and only train the new layers that we added on top. This approach can save training time and often results in good performance even with limited training data.

```
# Compiling the model
model.compile(loss='categorical_crossentropy',
          optimizer=Adam(lr=1e-4),
          metrics=['accuracy'])
# train the model
model.fit(x=train_batches ,validation_data=valid_batches,epochs=15)
```

In this code block, the **model** is compiled with **categorical_crossentropy** as the loss function, **Adam** optimizer with a learning rate of **1e-4**, and accuracy as the evaluation metric.

Then the **fit** method is used to train the model using the **train_batches** and **valid_batches** data generators for 15 epochs. During training, the model will optimize the weights and biases of its layers to minimize the loss function and improve accuracy on both the training and validation data.

```
# save model
model.save("model_v2_gray2.h5")
```

This line of code used for saving our train model to a file named model_v2_gray2.h5 so that we can use this model for live ASL detection.

```
img = image.load_img("./Dataset/validation/V/1.png",target_size=(244,244))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)
predictions = model.predict(x)
np.argmax(predictions)
```

This code snippet loads an image **1.png** from the **./Dataset/validation/V/** directory and resizes it to the target size of **(300,300)** using **image.load_img** from Keras.

Next, **image.img_to_array** converts the image to a numpy array, which is then expanded to include an additional dimension using **np.expand_dims** so that it can be used as input for the model.

The **preprocess_input** function from MobileNetV3 is applied to preprocess the input.

model.predict(x) is used to get predictions for the input image **x**, which returns an array of probabilities for each class. **np.argmax(predictions[0],axis=0)** is used to find the index of the highest probability, which corresponds to the predicted class label for the input image.

```
import tensorflow as tf
# Get your model's predictions for the validation data
predictions = model.predict_generator(valid_batches)

# Convert the predictions to class labels
y_pred = np.argmax(predictions, axis=1)

# Get the true labels for the validation data
y_true = valid_batches.classes

# Calculate precision, recall, and F-measure using scikit-learn's classification_report function
# Compute precision
precision = tf.keras.metrics.Precision()
precision.update_state(y_true, y_pred)
precision_result = precision.result()

# Compute recall
recall = tf.keras.metrics.Recall()
recall.update_state(y_true, y_pred)
recall_result = recall.result()

# Compute F1 score (F-measure)
f1_score = 2 * (precision_result * recall_result) / (precision_result + recall_result)

# Print the results
print("Precision:", precision_result.numpy())
print("Recall:", recall_result.numpy())
print("F1 Score:", f1_score.numpy())
```

In this example, **y_true** represents the true labels, and **y_pred** represents the predicted labels for your classification task. You first create instances of **tf.keras.metrics.Precision()** and **tf.keras.metrics.Recall()** to compute precision and recall, respectively. Then, you use the **update_state()** method of each metric to update their internal state with the true and predicted labels. Finally, you can calculate the F1 score by using the computed precision and recall values.

Note that these metrics are calculated over the entire dataset or batches that you provide. If you want to compute these metrics for individual predictions or evaluate them during training, you need to update the metrics' states accordingly after each prediction.

Keep in mind that these metrics assume binary classification by default. If you are working with multi-class classification, you may need to specify additional parameters such as **average** in the metrics' constructors or use the **tf.keras.metrics.CategoricalAccuracy()** metric for accuracy calculations.

3) Live Detection

At this step we will use our trained model “model_2.h5” for live detection. Funny thing is , this steps code actually combination of previous two steps code.

File: live-ditection.py

Code Explanation:

In this few more line of code added and all the other code copied from **data-collector.py** . I only explain newly added code here other code are explained in **Data Collection** section.

```
from tensorflow.keras.models import load_model
from tensorflow.keras.applications.mobilenet import preprocess_input
```

On line 7-8: In this code block, two modules are imported from TensorFlow Keras: **load_model** and **preprocess_input**.

load_model is a function provided by Keras that allows you to load a pre-trained model saved as an H5 file. It loads the model architecture, weights, and optimizer state (if applicable) into memory.

preprocess_input is a function provided by the MobileNet model in Keras. It performs specific preprocessing on input images, which includes scaling and normalization. This function is necessary to prepare the input data in the same way as it was during the training of the MobileNet model, so that it can make accurate predictions.

```
# loading ASL dictection model
model = load_model("model_v2_gray2.h5")
# labels of ASL actually list of Alphabets
labels = [chr(i) for i in range(ord("A"),ord("Z")+1)]
```

line 17-20: The first line of code loads the saved model from the file named "model_2.h5". This is done using the **load_model** function from the **tensorflow.keras.models** module.

The second line of code creates a list called **labels**, which contains all the capital letters of the English alphabet, A to Z. These labels correspond to the different classes of the ASL.

```
# processing image for model input
        xs = np.expand_dims(croped_frame, axis=0)
        xs = preprocess_input(xs)
        # prediction from image
        predictions = model.predict(xs)
        # get alphabet from predictions
        prediction = labels[np.argmax(predictions)]
        # styling bounding of prediction
        cv2.rectangle(img,(x-offset,y-offset),(x+w,y+h),(0,225,0),4)
        cv2.rectangle(img,(x-offset-2,y-offset-40),(x+50,y-offset),(0,225,0),cv2.FILLED)
        cv2.putText(img,prediction,(x,y-offset-10),cv2.FONT_HERSHEY_PLAIN,2,(255,255,255),2)
```

line 51-61: The code is processing an image to detect American Sign Language (ASL) alphabet from a user's hand using a pre-trained ASL detection model. Here's what each line is doing:

1. **xs = np.expand_dims(croped_frame, axis=0)**: The code is expanding the dimensions of an image for model input. It is converting the **croped_frame** image (a NumPy array that represents the image captured by the camera) to a four-dimensional NumPy array with an additional dimension at the beginning. This is done to match the input dimensions required by the ASL detection model.
2. **xs = preprocess_input(xs)**: This line preprocesses the image **xs** before feeding it to the ASL detection model. The **preprocess_input** function scales the pixel values of the image between -1 and 1 and applies other preprocessing steps specific to the model architecture (MobileNetV3Small).
3. **predictions = model.predict(xs)**: The ASL detection model predicts the class probabilities for the input image.
4. **prediction = labels[np.argmax(predictions[0],axis=0)]**: The predicted class with the highest probability is determined using **argmax**, and the corresponding ASL alphabet label is obtained from the **labels** list.
5. **cv2.rectangle(img,(x-offset,y-offset),(x+w,y+h),(0,225,0),4)**: This code draws a green bounding box around the user's hand. **x**, **y**, **w**, and **h** represent the coordinates of the bounding box, and **offset** is used to make the box slightly larger than the hand.

6. **cv2.rectangle(img,(x-offset-2,y-offset-40),(x+50,y-offset),(0,225,0),cv2.FILLED)**: This code draws a filled green rectangle over the top of the bounding box to create a background for the predicted alphabet text.
7. **cv2.putText(img,prediction,(x,y-offset-10),cv2.FONT_HERSHEY_PLAIN,2,(255,255,255),2)**: This line adds the predicted alphabet text to the image. The text is positioned slightly above the green rectangle, and is styled in white text with a black outline.

```
interrupt = cv2.waitKey(10)
# if press q from keyboard then exit the loop (stops program)
if interrupt and 0xFF==ord("q"):
    break
```

line 64-67: In this code block, **cv2.waitKey()** function is called to check if any key has been pressed by the user. If a key has been pressed, the ASCII value of that key is returned, otherwise, **0** is returned.

The **ord("q")** returns the ASCII value of the **q** key. So, if the user presses the **q** key, the program will exit the loop and stop running. This is done to provide a way to manually exit the program if needed.

To Live Detect run **live-ditect.py** from VS code or command in terminal **python live-ditect.py.**

Conclusion

In this project, a model was trained to recognize American Sign Language alphabets using transfer learning from the MobileNetV3Small model. The model was trained on a dataset of 26 alphabets and achieved an accuracy of around 99% on the validation set. The trained model was then loaded to recognize ASL alphabets in real-time using a webcam. The user can hold up their hand to the camera, and the program will predict the alphabet being signed. The model's predictions were displayed on the screen with a bounding box around the user's hand and the predicted alphabet. The project demonstrates the application of computer vision and deep learning in recognizing sign language gestures and can be extended to other sign languages and gestures with appropriate data.