FoldersCreation.py

```
# Importing the Libraries Required
import os    interact with the operating system, such as creating directories, listing directory contents and so on.
import string   provides a collection of string constants and utility functions.
# Creating the directory Structure
if not os.path.exists("dataSet"):   checks if the directory named “dataSet” does not exist in the current directory
    os.makedirs("dataSet")         it creates a directory named “dataset” if it does not exist. The ‘os.makeirs()’ function creates a
                                   directory and also creates any intermediate directories in the path if they do not exist.

if not os.path.exists("dataSet/trainingData"):  checks if the directory named “trainingData” does not exist inside the “dataset” directory
    os.makedirs("dataSet/trainingData")  creates the “trainingData” directory inside the “dataset” directory if it does not exist.

if not os.path.exists("dataSet/testingData"):   checks if the directory named “testingData” does not exist inside the “dataset” directory
    os.makedirs("dataSet/testingData") creates the “testingData” directory inside the “dataset” directory if it does not exist
# Making folder  0 (i.e blank) in the training and testing data folders respectively
for i in range(0):  It starts a loop that iterates over a range of numbers. However since the range is from 0 to 0(exclusive) this loop will
                    not execute any iterations. It’s essentially a no-op
    if not os.path.exists("dataSet/trainingData/" + str(i)):   checks if a directory corresponding to the current value of i does not exist
                                                               inside the “trainingData” directory
        os.makedirs("dataSet/trainingData/" + str(i))   creates a directory named with the current value of i inside the “trainingData”
                                                        directory if it does not exist.

    if not os.path.exists("dataSet/testingData/" + str(i)):    checks if a directory corresponding to the current value of i does not exist
                                                               inside the “testingData” directory.
        os.makedirs("dataSet/testingData/" + str(i))     creates a directory named with the current value of i inside the “testingData”
                                                         directory if it does not exist.
# Making Folders from A to Z in the training and testing data folders respectively
for i in string.ascii_uppercase:   starts loop that iterates over each uppercase letter in the English alphabet
    if not os.path.exists("dataSet/trainingData/" + i):   checks the directory corresponding to the current uppercase letter ‘i’ does not
                                                          exist inside the “trainingData” directory
        os.makedirs("dataSet/trainingData/" + i)   creates a directory named with the current uppercase letter ‘i’ inside the “trainingData”
                                                   directory if it does not exist.

    if not os.path.exists("dataSet/testingData/" + i):   checks if a directory corresponding to the current uppercase letter ‘i’ does not
                                                         exist inside the “testingData” directory
        os.makedirs("dataSet/testingData/" + i)     creates a directory named with the current uppercase letter ‘i’ inside the “testingData”
                                                    directory if it does not exist.
```

The code essentially creates a directory structure for storing training and testing data for a classification task, with subdirectories for each class label (from “A” to “Z”) however the loop with range(0) does not actually create any directories because it doesn’t execute any iterations.

TestingDataCollection.py

```
# Importing the Libraries Required
import cv2
import numpy as np
import os

# Creating and Collecting Training Data
mode = 'testingData'  sets the mode which determine whether the data is being collected for training or testing
directory = 'dataSet/' + mode + '/'  constructs the directory path based on the mode variable. It specifies the directory where the images
                                     will be saved.
minValue = 35   threshold for binarization in the image processing step

capture = cv2.VideoCapture(0)  initializes the video capture object, specifying the camera index(0, represents the default camera)
interrupt = -1  used to detect keyboard interrupts later

while True:        starts an infinite loop to continuously capture frames from the camera.
    _, frame = capture.read()   reads a frame from the video capture object. The '_' is used to discard the return value, as it indicates
                                Whether the read operation was successful.

    # Simulating mirror Image
    frame = cv2.flip(frame, 1)  flips the frame horizontally simulating a mirror image.

    # Getting count of existing images
    count = {
                'zero': len(os.listdir(directory+"/0")),

                'a': len(os.listdir(directory+"/A")),
                'b': len(os.listdir(directory+"/B")),
                'c': len(os.listdir(directory+"/C")),
                'd': len(os.listdir(directory+"/D")),
                ………………….
                'x': len(os.listdir(directory+"/X")),
                'y': len(os.listdir(directory+"/Y")),
                'z': len(os.listdir(directory+"/Z")),
    }  creates a dictionary called 'count' which stores the number of existing images for each class label (from zero to Z)

    # Printing the count of each set on the screen
    cv2.putText(frame, "ZERO : " +str(count['zero']), (10, 60), cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
    cv2.putText(frame, "a : " +str(count['a']), (10, 70), cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
    cv2.putText(frame, "b : " +str(count['b']), (10, 80), cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
    cv2.putText(frame, "c : " +str(count['c']), (10, 90), cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
    ...............................................
    cv2.putText(frame, "x : " +str(count['x']), (10, 290), cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
    cv2.putText(frame, "y : " +str(count['y']), (10, 300), cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
    cv2.putText(frame, "z : " +str(count['z']), (10, 310), cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
      these lines display the count of images for each class label on the frame using ‘cv2.putText()’

    # Coordinates of the ROI
    x1 = int(0.5*frame.shape[1])
    y1 = 10
    x2 = frame.shape[1]-10
    y2 = int(0.5*frame.shape[1])
                                  defines the coordinates of the region of interest rectangle which is used to capture hand gestures

    # Drawing the ROI
    # The increment/decrement by 1 is to compensate for the bounding box
    cv2.rectangle(frame, (x1-1, y1-1), (x2+1, y2+1), (255,0,0) ,1)

    # Extracting the ROI

    roi = frame[y1:y2, x1:x2]   extracts ROI from the frame

    cv2.imshow("Frame", frame)

    # Image Processing
    gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)   converts the ROI to grayscale
    blur = cv2.GaussianBlur(gray,(5,5),2)  applies gaussian blur to the grayscale ROI to reduce noise

    th3 = cv2.adaptiveThreshold(blur,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRESH_BINARY_INV,11,2)  applies adaptive thresholding to the
                                                                                                     blurred image
    ret, test_image = cv2.threshold(th3, minValue, 255, cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU) applies binary thresholding to the
                                                                                                adaptive thresholder image

    # Output Image after the Image Processing that is used for data collection
    test_image = cv2.resize(test_image, (300,300))   resizes the processed image to a fixed size of 300x300 pixels
    cv2.imshow("test", test_image) displays the processed image in a separate window named “test”
    # Data Collection
    interrupt = cv2.waitKey(10)  waits for a key press for 10 milliseconds and stores the key code in the interrupt variable
    if interrupt & 0xFF == 27:
        # esc key
        break
  checks if the ESC key is pressed. If pressed it breaks out of the loop, terminating the program

    if interrupt & 0xFF == ord('0'):
        cv2.imwrite(directory+'0/'+str(count['zero'])+'.jpg', roi)
 these lines check if a specific key corresponding the a class label is pressed. If pressed it saves the ROI image to the corresponding
directory. This patter continues for all uppercase letters from ‘A’ to ‘Z’ with each letter associated with a specific key press event.

    if interrupt & 0xFF == ord('a'):
        cv2.imwrite(directory+'A/'+str(count['a'])+'.jpg', roi)

    if interrupt & 0xFF == ord('b'):
        cv2.imwrite(directory+'B/'+str(count['b'])+'.jpg', roi)

    if interrupt & 0xFF == ord('c'):
        cv2.imwrite(directory+'C/'+str(count['c'])+'.jpg', roi)

    ……………………………..

    if interrupt & 0xFF == ord('x'):
        cv2.imwrite(directory+'X/'+str(count['x'])+'.jpg', roi)

    if interrupt & 0xFF == ord('y'):
        cv2.imwrite(directory+'Y/'+str(count['y'])+'.jpg', roi)

    if interrupt & 0xFF == ord('z'):
        cv2.imwrite(directory+'Z/'+str(count['z'])+'.jpg', roi)

capture.release()
cv2.destroyAllWindows()   releases the video capture object and close all OpenCV windows effectively ending the program
```

TrainingDataCollection.py   similar to testing

Application.py

```
# Importing Libraries
import numpy as np

import cv2
import os, sys    functions for interacting with the operating system
import time       provides various time related functions
import operator   provides functions corresponding to the standard operators in Python

from string import ascii_uppercase     contains uppercase letters from A to Z

import tkinter as tk     creating GUI applications in Python
from PIL import Image, ImageTk   Python Imaging Library which is used for image processing tasks


from keras.models import model_from_json   used to load neural network models stored in JSON format

os.environ["THEANO_FLAGS"] = "device=cuda, assert_no_cpu_op=True"
sets environment variables for Theano, a numerical library used as a backend for Keras

'os.environ': this is a python module that provides access to the operating system's

'THEANO_FLAGS': this is the name of the environment variable being set. Theano uses environment variables to configure its behaviour

'device=cuda': this specifies that Theano should use the CUDA backend for GPU acceleration. CUDA is a parallel computing platform and
programming model developed by NVIDIA for general computing on GPUs(graphics processing units)

'assert_no_cpu_op=True': this option tells Theano to raise an error if it encounters an operation that cannot be performed on the CPU. This
can be useful for debugging or ensuring that your code runs exclusively on the GPU when using CUDA

So, overall this line of code is configuring Theano to use CUDA for GPU acceleration and to raise an error if any operation cannot be
performed on the CPU


#Application :

class Application:

    def __init__(self):
```

The '__init__' is the constructor for the 'Application' class. It's the method that gets called automatically when you create a new instance of the class. Inside this method, you can initialize instance variables, set up initial states, and perform any necessary setup for your class.

```
        self.vs = cv2.VideoCapture(0)  initializes a VideoCapture object named 'vs' to capture video from the default camera (index 0)
```
cv2.VideoCapture: this is a class in the OpenCV library that provides methods to capture video from various sources, including cameras and video files
After executing 'self.vs' variable will hold a VideoCapture object that can be used to capture frames from the specified camera

```
        self.current_image = None
```
initializes the 'current_image' attribute of the 'Application' class to 'None'. This attribute is likely used to store the current image frame captured from the video stream.
Initializing it to 'None' allows you to check later in the code if the attribute has been assigned a value or not.
'self.current_image': this is an attribute of the 'Application' class
'None': this is Python built in constant representing the absence of a value. It indicates that the 'current_image' attribute does not currently hold any meaningful image data

```
        self.current_image2 = None
```
Similar to 'self.current_image' this attribute is likely used to store another image frame captured from the video stream

```
        self.json_file = open("Models\model_new.json", "r")  opening a JSON file named 'model_new.json' located in the 'Models' directory
        self.model_json = self.json_file.read() reads the content of the JSON file and assigns it to the variable 'self.mode_json'
        self.json_file.close()  closing the file after reading its content


        self.loaded_model = model_from_json(self.model_json)
```
The 'model_from_json' function from Keras is used to create a model from its JSON representation. 'self.model_json' contains the JSON string representation of the model architecture, which was read from the file 'Models/model_new.json'.By calling 'model_from_json(self.model_json)' , you're reconstructing the model architecture based on this JSON string
```
        self.loaded_model.load_weights("Models\model_new.h5")
```
'self.loaded_model': This is the keras model that we previously created and loaded from the JSON representation using 'model_from_json'
'.load_weights("Models\model_new.h5"): this method call loads the weights of the model from the specified HDF5 file. The argument passed to 'load_weights' is the path to the HDF5 file containing the model's weights.

```
        self.json_file_dru = open("Models\model-bw_dru.json" , "r")
```
this variable is assigned to the file object created by opening the file 'open("Models\model-bw_dru.json", "r")': this function call opens the JSON file named "model-bw_dru.json" in read mode. The first argument is the file path, which includes the directory ("Models" and the file name("model-bw_dru.json"). The second argument "r" indicates that the file is opened in read mode.
```
        self.model_json_dru = self.json_file_dru.read()
```

```
reads the content of the JSON file and assigns it to the variable ‘self.model_json_dru’
        self.json_file_dru.close() closing the file after reading its content


        self.loaded_model_dru = model_from_json(self.model_json_dru)
This line uses the ‘model_from_json’ function from Keras to create a model based on the architecture defined in the JSON format. It takes
the JSON content stored in ‘self.model_json_dru’ and constructs the model accordingly. This model will be stored in the
‘self.loaded_model_dru’ variable
        self.loaded_model_dru.load_weights("Models\model-bw_dru.h5")
After creating the model architecture, this line loads the weights of the model from the specified HDF5 file (‘model-bw_dru.h5’). the
weights are essential for the model to make accurate predictions. The weights are loaded into the ‘self.loaded_model_dru’ model
        self.json_file_tkdi = open("Models\model-bw_tkdi.json" , "r")
this line opens another JSON file named ‘model-bw_tkdi.json’ located in the ‘Models’ directory in read mode. It stores the file object in
the ‘self.json_file_tkdi’ variable
        self.model_json_tkdi = self.json_file_tkdi.read()
this line reads the content of the JSON file opened in the previous line and stores it in the ‘self.model_json_tkdi’ variable
        self.json_file_tkdi.close() closing the file after reading its content


        self.loaded_model_tkdi = model_from_json(self.model_json_tkdi)
        self.loaded_model_tkdi.load_weights("Models\model-bw_tkdi.h5")
        self.json_file_smn = open("Models\model-bw_smn.json" , "r")
        self.model_json_smn = self.json_file_smn.read()
        self.json_file_smn.close()

        self.loaded_model_smn = model_from_json(self.model_json_smn)
        self.loaded_model_smn.load_weights("Models\model-bw_smn.h5")

        self.ct = {}        Initializes an empty dictionary named ‘ct’. Dictionaries are mutable data structures that store key value pairs
        self.ct['blank'] = 0     assigns a value of ‘0’ to the key ‘blank’ which will be used to count occurrences of the symbol ‘blank’
        self.blank_flag = 0
initializes a variable ‘blank_flag’ and assigns it the value ‘0’. This flag will be used as a signal to track whether the current symbol
being processed is ‘blank’ or not. It is initially set to ‘0’

        for i in ascii_uppercase:
          self.ct[i] = 0
it iterates over each uppercase letter in the English alphabet (‘ascii_uppercase’) and assigns a value of ‘0’ to each letter in the
‘self.ct’ dictionary. It essentially initializes each letter’s count to zero in ‘self.ct’, which is used to keep track of the occurrences of
each symbol in the captured input
```

```
        print("Loaded model from disk")

        self.root = tk.Tk()           creates a Tkinter application window
        self.root.title("Sign Language To Text Conversion") sets the title of the application window
        self.root.protocol('WM_DELETE_WINDOW', self.destructor)
sets a protocol for the window. When the window is closed (using the window manager’s close button) it calls the ‘destructor’ method, which
handles the closing of the application
        self.root.geometry("800x600")  sets the initial size of the window to 800 pixels by 600 pixels. The format is width x height


        self.panel = tk.Label(self.root)    creates a label widget that will be placed inside the root window (‘self.root’)
        self.panel.place(x = 100, y = 10, width = 580, height = 580)
specifies the placement and size of the label widget within the root window. It sets the x-coordinates of the top-left cornet to 100, the y-
coordinate to 10, the width to 580 pixels, and the height to 580 pixels. The ‘place’ method is used to specify exact coordinates and
dimensions for widget placement.


        self.panel2 = tk.Label(self.root) # initialize image panel
        self.panel2.place(x = 400, y = 65, width = 275, height = 275)


        self.T = tk.Label(self.root)  creates a new label widget that will be placed inside the root window (‘self.root’)
        self.T.place(x = 100, y = 5)
specifies the position of the label widget within the root window. It sets the x-coordinate of the top-left corner to 100 and the y-
coordinate to 5
        self.T.config(text = "SignoSpeak: Bridging the Gap", font = ("Tilt Neon", 30, "bold"))
configures the label widget by setting its text and its font to Tilt Neon, with a size of 30 and bold weight. The ‘config’ method is used to
update the properties of the widget after it has been created.

        self.panel3 = tk.Label(self.root) # Current Symbol
        self.panel3.place(x = 500, y = 540)

        self.T1 = tk.Label(self.root)
        self.T1.place(x = 170, y = 540)
        self.T1.config(text = "Character :", font = ("Tilt Neon", 30, "bold"))

        self.bt1 = tk.Button(self.root, command = self.action1, height = 0, width = 0)
        self.bt1.place(x = 26, y = 745)
```

it creates a button widget (‘tk.Button’) and attaches a command (‘self.action1’) to it. When the button is clicked, the function ‘self.action1’ will be executed. The ‘height’ and ‘width’ parameters set the dimensions of the button, but since they are set to 0, the button will automatically adjust its size based on its content. Finally the ‘place’ method is used to specify the exact position of the button within the root window (‘self.root’) with (26, 745) being the coordinates of the button’s top left corner.

```
        self.bt2 = tk.Button(self.root, command = self.action2, height = 0, width = 0)
        self.bt2.place(x = 325, y = 745)

        self.bt3 = tk.Button(self.root, command = self.action3, height = 0, width = 0)
        self.bt3.place(x = 625, y = 745)

        self.str = "" initializes an empty string variable ‘str’
        self.word = " " initializes a string variable ‘word’ with a single space character.
        self.current_symbol = "Empty"
        self.photo = "Empty"
        self.video_loop()
```

Calls the ‘video_loop’ method to start the video capture and processing loop. This method continuously captures video frames from the camera, processes them and updates the user interface with the results.

```
    def video_loop(self):
        ok, frame = self.vs.read()
```

This function video_loop is designed to continuously capture video frames from the camera (self.vs) and process them.
Self.vs.read() it reads a frame from the video capture device (self.vs). it returns two values :ok, which indicates whether the frame was successfully read, and frame,, which contains the actual image data of the frame
ok, frame = self.vs.read(): it reads a frame from the video capture device (self.vs.read()) and unpacks the returned values into the variable ok and frame.ok will be True if the frame is successfully read, and False otherwise. frame will contain the image data of the frame.
The function continues to process this frame in subsequent lines to update the user interface and perform other tasks such as image processing, gesture recognition, and updating the display.

```
        if ok:
            cv2image = cv2.flip(frame, 1)
```

It checks if the variable ‘ok’ is ‘True’ indicating that a frame has been successfully read from the video capture device. If ‘ok’ is ‘True’ it proceeds to flip the frame horizontally using OpenCV’s ‘cv2.flip()’ function
‘cv2.flip(frame, 1)’ this function flips the input frame horizontally. The second argument ‘1’ specifies the flip direction, where ‘1’ indicates horizontal flipping. This operation is often performed to correct the orientation of the captured image for display purposes

These lines of code calculate the coordinates of a rectangle that will be drawn on the frame to create a region of interest (ROI). This ROI is typically used to isolate the area where hand gestures or signs are being made, allowing for better analysis and classification.

```
            x1 = int(0.5 * frame.shape[1])
```

it calculates the x-coordinate of the top left corner of the rectangle. It's set to half the width of the frame ('frame.shape[1]') to position the rectangle in the middle horizontally.

```
            y1 = 10
```

sets the y-coordinate of the top left corner of the rectangle it's set to '10', which means the rectangle will start '10' pixels down from the top of the frame.

```
            x2 = frame.shape[1] - 10
```

it calculates the x-coordinate of the bottom-right corner of the rectangle. Its set to the width of the frame ('frame.shape[1]') minus '10' pixels. This creates a margin of '10' pixels from the right edge of the frame

```
            y2 = int(0.5 * frame.shape[1])
```

it calculates the y-coordinate of the bottom-right corner of the rectangle. It's set to half the width of the frame ('frame.shape[1]') to maintain a square shape for the ROI

draw a rectangle on the frame and convert the frame from BGR (Blue, Green and Red) to RGBA (Red, Green, Blue and Alpha value) colour space.

```
            cv2.rectangle(frame, (x1 - 1, y1 - 1), (x2 + 1, y2 + 1), (255, 0, 0) ,1)
```

draws a rectangle on the 'frame' using OpenCV's 'rectangle' function. The rectangle is drawn with the following parameters:
(x1 – 1, y1 – 1): top left corner of the rectangle. Subtracts '1' from 'x1' and 'y1' to adjust the position slightly
(x2 + 1, y2 + 1): bottom right corner of the rectangle. Adds '1' to 'x2' and 'y2' to adjust the position slightly
(255, 0, 0): colour of the rectangle in RGB format. Here its set to blue
1: thickness of the rectangle border in pixels

```
            cv2image = cv2.cvtColor(cv2image, cv2.COLOR_BGR2RGBA)
```

converts the 'cv2image' from the BGR colour space to the RGBA colour space. This conversion is necessary because some image display functions in Tkinter require RGBA images

convert the OpenCV image ('cv2image') to a PIL (Python Imaging Library) image ('self.current_image') and then create a Tkinter-compatible image ('imgtk') from the PIL image

```
            self.current_image = Image.fromarray(cv2image)
```

converts the OpenCV image ('cv2image') to a PIL image ('self.current_image'). The 'Image.fromarray()' function from the PIL library is used for this conversion

```
            imgtk = ImageTk.PhotoImage(image = self.current_image)
```

creates a Tkinter-compatible image ('imgtk') from the PIL image ('self.current_image'). The 'ImageTK.PhotoImage()" function from the PIL library is used for this purpose. The 'image' parameter is set ot 'self.current_image'
after these lines, 'imgtk' can be used to display the current frame in a Tkinter GUI

updates the Tkinter Lable widget ('self.panel') with the new image ('imgtk')

```
            self.panel.imgtk = imgtk
```

assigns the new Tkinter-compatible image (‘imgtk’) to the ‘imgtk’ attribute of the Label widet(‘self.panel’) this attribute is used to keep a reference to the image so that it doesn’t get garbage collected

```
            self.panel.config(image = imgtk)
```

configures the label widget (‘self.panel’) to display the new image(‘imgtk’) the ‘image’ option of the ‘config()’ method is et to ‘imgtk’ which updates the image displayed by the Label widget to the new image

```
            cv2image = cv2image[y1 : y2, x1 : x2]
```

This line of code crops the region of interest (ROI) from the `cv2image` array. It selects a rectangular region defined by the coordinates `(x1, y1)` as the top-left corner and `(x2, y2)` as the bottom-right corner and extracts that portion of the image. The extracted region corresponds to the area inside the rectangle drawn previously on the frame.

```
            gray = cv2.cvtColor(cv2image, cv2.COLOR_BGR2GRAY)
```

This line of code converts the cropped image `cv2image` from BGR (Blue, Green, Red) color space to grayscale using the OpenCV function `cv2.cvtColor()`. Grayscale conversion is a common preprocessing step in computer vision tasks and simplifies image processing by reducing the dimensionality of the image from three channels (RGB) to one channel (intensity).

```
            blur = cv2.GaussianBlur(gray, (5, 5), 2)
```

This line applies Gaussian blur to the grayscale image `gray`. Gaussian blur is a widely used image filtering technique to reduce noise and detail in an image. It smooths out the image by averaging the pixel values in the vicinity of each pixel, with the degree of smoothing controlled by the standard deviation parameter (`2` in this case). The size of the kernel `(5, 5)` determines the area over which the averaging is performed. Smaller kernel sizes result in less blurring, while larger kernel sizes result in more blurring.

```
            th3 = cv2.adaptiveThreshold(blur, 255 ,cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 11, 2)
```

This line performs adaptive thresholding on the blurred grayscale image `blur`. Adaptive thresholding is a method used to binarize images by comparing each pixel's intensity to its local neighborhood. Here's what each parameter means:
- `blur`: The source image.
- `255`: The maximum intensity value that will be assigned to pixels exceeding the threshold.
- `cv2.ADAPTIVE_THRESH_GAUSSIAN_C`: Specifies the adaptive thresholding method, where the threshold value is the weighted sum of the local neighborhood values subtracted by a constant (`C`).
- `cv2.THRESH_BINARY_INV`: Specifies the type of thresholding, where pixels with intensities above the threshold are set to 0 (black) and below the threshold to 255 (white). The `_INV` suffix indicates that it's an inverted binary thresholding.
- `11`: The size of the local neighborhood (block size) used for calculating the threshold value. It must be an odd number.
- `2`: The constant `C` subtracted from the weighted sum of the local neighborhood values.

In summary, this line applies adaptive Gaussian thresholding to the blurred image `blur` to obtain a binary image `th3`, where the threshold value is determined locally for each pixel based on the intensities of its surrounding pixels.

```
            ret, res = cv2.threshold(th3, 70, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
```
This line applies thresholding to the image `th3` using both a manually set threshold value (`70`) and Otsu's thresholding method. Here's what each parameter means:
- `th3`: The source image.
- `70`: The manually set threshold value. Pixels with intensities below this value will be set to the maximum value (255) and those above will be set to 0. This thresholding is performed in conjunction with Otsu's thresholding.
- `255`: The maximum intensity value that will be assigned to pixels exceeding the threshold.
- `cv2.THRESH_BINARY_INV`: Specifies the type of thresholding, where pixels with intensities above the threshold are set to 0 (black) and below the threshold to 255 (white). The `_INV` suffix indicates that it's an inverted binary thresholding.
- `cv2.THRESH_OTSU`: Flag indicating the use of Otsu's thresholding method. Otsu's method automatically calculates the optimal threshold value based on the histogram of the image.
- `ret`: The calculated threshold value by Otsu's method (not used in this context).
- `res`: The resulting binary image after applying thresholding.
In summary, this line applies thresholding to the adaptive Gaussian thresholded image `th3` using both a manually set threshold value and Otsu's thresholding method, resulting in a binary image `res`.

This line of code calls the `predict` method with the parameter `res`.
```
            self.predict(res)
```
- `self.predict`: Calls the `predict` method of the current class instance (`self`).
- `(res)`: Passes the variable `res` as an argument to the `predict` method.
The `predict` method is presumably responsible for making predictions based on the input image `res`, which is a binary image obtained after thresholding. The predictions likely involve using a trained machine learning model to classify the input image into different classes or categories.

This line of code converts the NumPy array `res` into a PIL (Python Imaging Library) image object named `self.current_image2`.
```
            self.current_image2 = Image.fromarray(res)
```
- `Image.fromarray(res)`: Converts the NumPy array `res` into a PIL image object. `Image.fromarray()` is a method provided by the PIL library to create an image from a NumPy array.
- `self.current_image2 =`: Assigns the resulting PIL image object to the attribute `self.current_image2` of the current class instance. This attribute likely holds the processed image for display or further processing.

```
            imgtk = ImageTk.PhotoImage(image = self.current_image2)
```
This line of code creates a Tkinter-compatible image object (`imgtk`) from the PIL image object `self.current_image2`. Here's what each part does:

- `ImageTk.PhotoImage`: This is a method provided by the `ImageTk` module, which is part of the PIL library. It creates a Tkinter-compatible image object from a PIL image object.

- `image = self.current_image2`: This specifies the PIL image object (`self.current_image2`) that will be used to create the Tkinter-compatible image.
Once created, `imgtk` can be used to display the image in a Tkinter GUI.

These lines of code update the image displayed in `panel2` with the newly created Tkinter-compatible image `imgtk`.

```
            self.panel2.imgtk = imgtk
            self.panel2.config(image = imgtk)
```

- `self.panel2.imgtk = imgtk`: This assigns the `imgtk` object to the `imgtk` attribute of `panel2`. This is necessary to keep a reference to the `imgtk` object to prevent it from being garbage collected.
- `self.panel2.config(image = imgtk)`: This configures the `panel2` widget to display the image represented by the `imgtk` object. By setting the `image` option to `imgtk`, the panel will update its display to show the new image.
Together, these lines ensure that the `panel2` widget in the Tkinter GUI displays the updated image represented by the `imgtk` object.

This line of code updates the text displayed in `panel3` with the value of `self.current_symbol` using a specific font.

```
            self.panel3.config(text = self.current_symbol, font = ("Courier", 30))
```

- `self.panel3.config`: This configures the properties of the `panel3` widget.
- `text = self.current_symbol`: This sets the text displayed in the `panel3` widget to the value stored in the `self.current_symbol` variable.
- `font = ("Bebas Neue", 30)`: This specifies the font style and size for the text. In this case, it sets the font to "Bebas Neue" with a font size of 30.
Together, this line ensures that the text displayed in `panel3` is updated with the value stored in `self.current_symbol` and is styled according to the specified font.

```
            ret, predicts = self.vs.read()
```

In the line `ret, predicts = self.vs.read()`, `self.vs.read()` is a method call on the `self.vs` object, which is an instance of `cv2.VideoCapture`. This method reads a frame from the video stream captured by the camera.
- `ret`: This variable holds a boolean value indicating whether the frame was successfully read. If `ret` is `True`, it means a frame was successfully captured. If `ret` is `False`, it means there was an issue reading the frame.
- `predicts`: This variable holds the actual frame data that was read from the video stream.
However, the variable name `predicts` is a bit misleading here, as it suggests that it contains predictions or some sort of processed data. In reality, it holds the raw frame data read from the video stream. You might want to consider renaming it to something more descriptive, like `frame`.

```
            if(len(predicts) > 1):

                self.bt1.config(text = predicts[0], font = ("Courier", 20))
            else:

                self.bt1.config(text = "")

            if(len(predicts) > 2):

                self.bt2.config(text = predicts[1], font = ("Courier", 20))

            else:

                self.bt2.config(text = "")

            if(len(predicts) > 3):

                self.bt3.config(text = predicts[2], font = ("Courier", 20))

            else:

                self.bt3.config(text = "")
```

In this block of code, the application seems to be updating the text displayed on three buttons (`bt1`, `bt2`, and `bt3`) based on the number of elements in the `predicts` array.

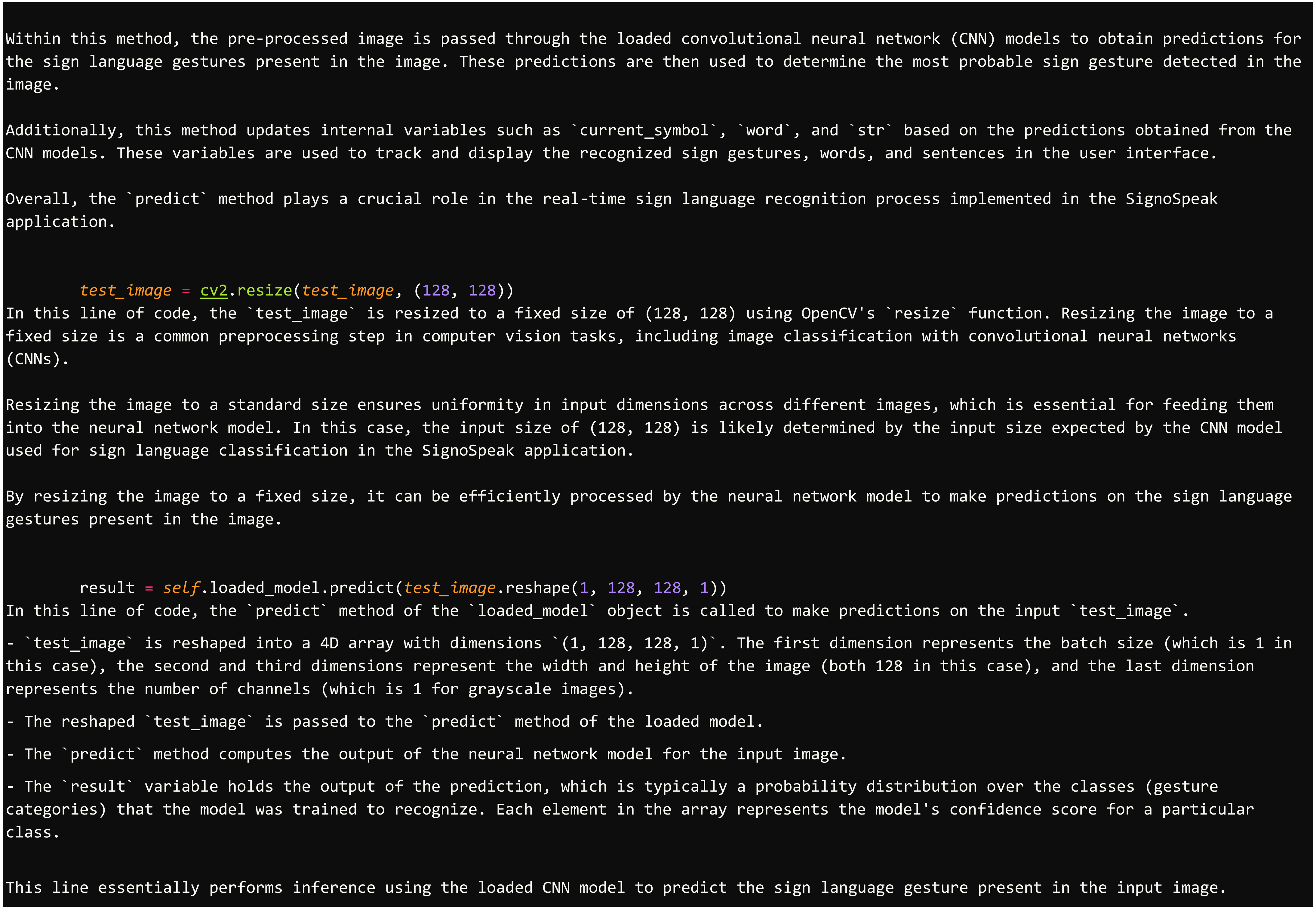
- `predicts` seems to be an array containing some data related to predictions or suggestions. The code checks its length to determine how many buttons should display data.
- If `predicts` has more than one element, the text of `bt1` is set to the first element, the text of `bt2` is set to the second element, and so on. The font used for the text is set to "Bebas Neue" with a size of 20.
- If `predicts` has fewer than two elements, the corresponding buttons' text is set to an empty string.
- This logic repeats for `bt2` and `bt3`, with adjustments made based on the length of the `predicts` array.

```
        self.root.after(5, self.video_loop)
```

This line schedules the `video_loop` function to be called again after 5 milliseconds. Essentially, it sets up a loop where the `video_loop` function is repeatedly called with a delay of 5 milliseconds between each call. This allows for the continuous capture and processing of video frames from the camera.

```
    def predict(self, test_image):
```

This method `predict` is responsible for making predictions on the input test image. It takes a single argument `test_image`, which represents the pre-processed image frame captured from the video feed.



Within this method, the pre-processed image is passed through the loaded convolutional neural network (CNN) models to obtain predictions for the sign language gestures present in the image. These predictions are then used to determine the most probable sign gesture detected in the image.

Additionally, this method updates internal variables such as `current_symbol`, `word`, and `str` based on the predictions obtained from the CNN models. These variables are used to track and display the recognized sign gestures, words, and sentences in the user interface.

Overall, the `predict` method plays a crucial role in the real-time sign language recognition process implemented in the SignoSpeak application.

```
        test_image = cv2.resize(test_image, (128, 128))
```

In this line of code, the `test_image` is resized to a fixed size of (128, 128) using OpenCV's `resize` function. Resizing the image to a fixed size is a common preprocessing step in computer vision tasks, including image classification with convolutional neural networks (CNNs).

Resizing the image to a standard size ensures uniformity in input dimensions across different images, which is essential for feeding them into the neural network model. In this case, the input size of (128, 128) is likely determined by the input size expected by the CNN model used for sign language classification in the SignoSpeak application.

By resizing the image to a fixed size, it can be efficiently processed by the neural network model to make predictions on the sign language gestures present in the image.

```
        result = self.loaded_model.predict(test_image.reshape(1, 128, 128, 1))
```

In this line of code, the `predict` method of the `loaded_model` object is called to make predictions on the input `test_image`.

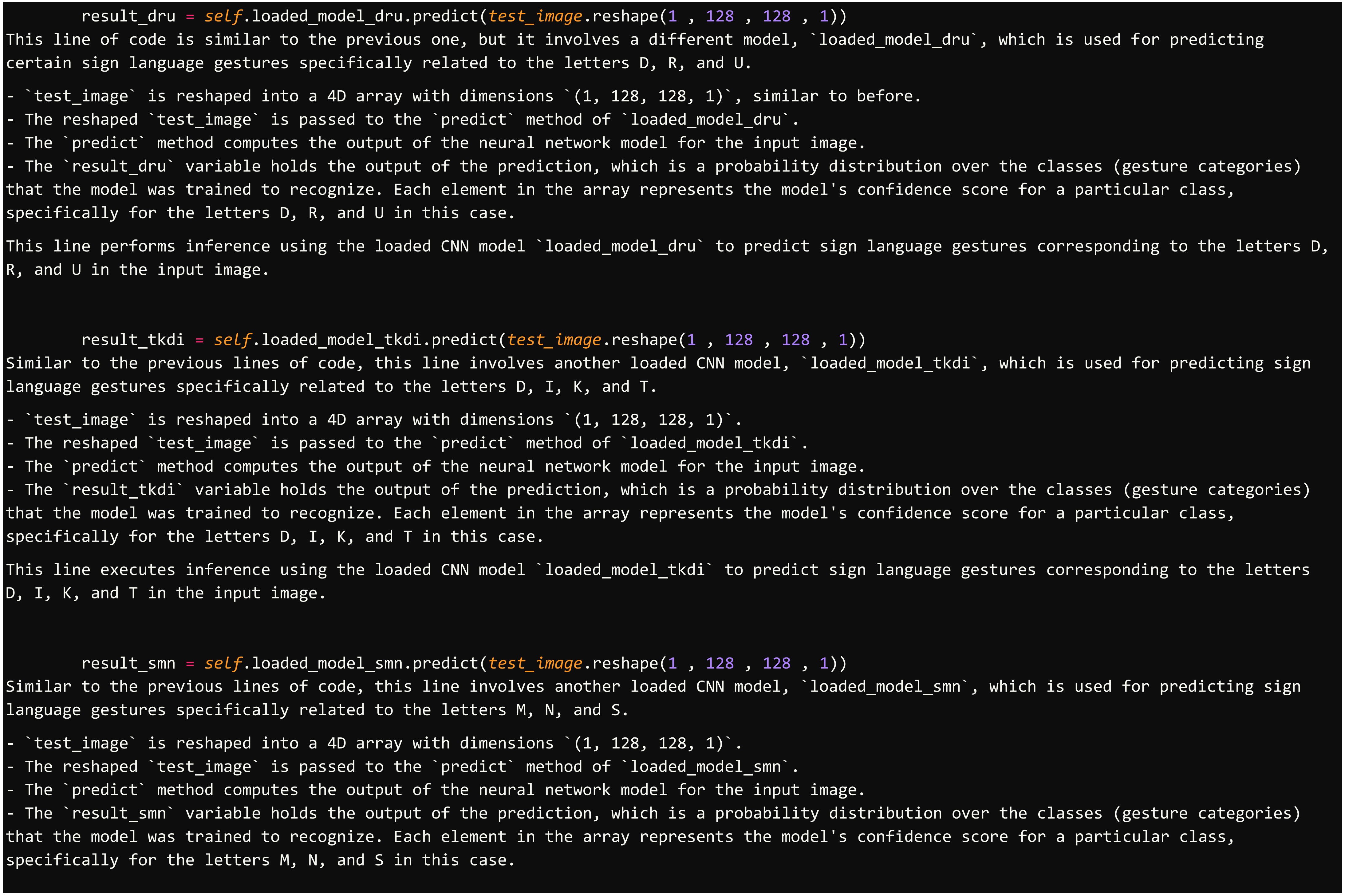
- `test_image` is reshaped into a 4D array with dimensions `(1, 128, 128, 1)`. The first dimension represents the batch size (which is 1 in this case), the second and third dimensions represent the width and height of the image (both 128 in this case), and the last dimension represents the number of channels (which is 1 for grayscale images).

- The reshaped `test_image` is passed to the `predict` method of the loaded model.

- The `predict` method computes the output of the neural network model for the input image.

- The `result` variable holds the output of the prediction, which is typically a probability distribution over the classes (gesture categories) that the model was trained to recognize. Each element in the array represents the model's confidence score for a particular class.

This line essentially performs inference using the loaded CNN model to predict the sign language gesture present in the input image.



```
        result_dru = self.loaded_model_dru.predict(test_image.reshape(1 , 128 , 128 , 1))
```

This line of code is similar to the previous one, but it involves a different model, `loaded_model_dru`, which is used for predicting certain sign language gestures specifically related to the letters D, R, and U.

- `test_image` is reshaped into a 4D array with dimensions `(1, 128, 128, 1)`, similar to before.
- The reshaped `test_image` is passed to the `predict` method of `loaded_model_dru`.
- The `predict` method computes the output of the neural network model for the input image.
- The `result_dru` variable holds the output of the prediction, which is a probability distribution over the classes (gesture categories) that the model was trained to recognize. Each element in the array represents the model's confidence score for a particular class, specifically for the letters D, R, and U in this case.

This line performs inference using the loaded CNN model `loaded_model_dru` to predict sign language gestures corresponding to the letters D, R, and U in the input image.

```
        result_tkdi = self.loaded_model_tkdi.predict(test_image.reshape(1 , 128 , 128 , 1))
```

Similar to the previous lines of code, this line involves another loaded CNN model, `loaded_model_tkdi`, which is used for predicting sign language gestures specifically related to the letters D, I, K, and T.

- `test_image` is reshaped into a 4D array with dimensions `(1, 128, 128, 1)`.
- The reshaped `test_image` is passed to the `predict` method of `loaded_model_tkdi`.
- The `predict` method computes the output of the neural network model for the input image.
- The `result_tkdi` variable holds the output of the prediction, which is a probability distribution over the classes (gesture categories) that the model was trained to recognize. Each element in the array represents the model's confidence score for a particular class, specifically for the letters D, I, K, and T in this case.

This line executes inference using the loaded CNN model `loaded_model_tkdi` to predict sign language gestures corresponding to the letters D, I, K, and T in the input image.

```
        result_smn = self.loaded_model_smn.predict(test_image.reshape(1 , 128 , 128 , 1))
```

Similar to the previous lines of code, this line involves another loaded CNN model, `loaded_model_smn`, which is used for predicting sign language gestures specifically related to the letters M, N, and S.

- `test_image` is reshaped into a 4D array with dimensions `(1, 128, 128, 1)`.
- The reshaped `test_image` is passed to the `predict` method of `loaded_model_smn`.
- The `predict` method computes the output of the neural network model for the input image.
- The `result_smn` variable holds the output of the prediction, which is a probability distribution over the classes (gesture categories) that the model was trained to recognize. Each element in the array represents the model's confidence score for a particular class, specifically for the letters M, N, and S in this case.

This line executes inference using the loaded CNN model `loaded_model_smn` to predict sign language gestures corresponding to the letters M, N, and S in the input image.

```
        prediction = {}
```
This line initializes an empty dictionary named `prediction`. This dictionary will be used to store the model's predictions for each class (gesture category) in the sign language alphabet.

```
        prediction['blank'] = result[0][0]
```
This line assigns the probability of the "blank" class (i.e., no gesture) predicted by the model to the corresponding key `'blank'` in the `prediction` dictionary. The probability value is extracted from the model's prediction result for the "blank" class, which is accessed using `result[0][0]`.

```
        inde = 1
```
This line initializes the variable `inde` to the value 1. It's likely used as an index counter in the following loop or computation.

```
        for i in ascii_uppercase:

            prediction[i] = result[0][inde]

            inde += 1
```
This loop iterates over each uppercase letter in the English alphabet (`ascii_uppercase`). Within the loop:

1. It assigns the prediction value for the current letter (`i`) to the corresponding index in the `prediction` dictionary.
2. It then increments the `inde` variable by 1 to move to the next prediction value in the `result` array. This ensures that each letter gets its corresponding prediction value from the `result` array.

This loop effectively populates the `prediction` dictionary with the predicted values for each letter.

```
        #LAYER 1

        prediction = sorted(prediction.items(), key = operator.itemgetter(1), reverse = True)
```
This line sorts the `prediction` dictionary items based on their values (the predicted probabilities), in descending order. The `operator.itemgetter(1)` function is used as the key argument to specify that the sorting should be based on the values of the dictionary items. The `reverse=True` parameter indicates that the sorting should be done in descending order, meaning that items with higher predicted probabilities will appear first in the sorted list.

```
        self.current_symbol = prediction[0][0]
```

This line assigns the most probable predicted symbol (the key of the first item in the sorted `prediction` list) to the `self.current_symbol` variable. In other words, it extracts the symbol with the highest predicted probability from the sorted list and stores it in `self.current_symbol`.

```
        #LAYER 2

        if(self.current_symbol == 'D' or self.current_symbol == 'R' or self.current_symbol == 'U'):
            prediction = {}
            prediction['D'] = result_dru[0][0]
            prediction['R'] = result_dru[0][1]
            prediction['U'] = result_dru[0][2]
            prediction = sorted(prediction.items(), key = operator.itemgetter(1), reverse = True)
            self.current_symbol = prediction[0][0]
```

This block of code is a refinement step in the prediction process based on certain conditions. If the current symbol is one of 'D', 'R', or 'U', it indicates that the initial prediction might need further refinement. In this case, it calculates new probabilities for these symbols using a separate model (`loaded_model_dru`) and assigns the symbol with the highest probability among 'D', 'R', and 'U' as the new `self.current_symbol`. This process helps to improve the accuracy of the prediction for specific symbols.

```
        if(self.current_symbol == 'D' or self.current_symbol == 'I' or self.current_symbol == 'K' or self.current_symbol == 'T'):
            prediction = {}
            prediction['D'] = result_tkdi[0][0]
            prediction['I'] = result_tkdi[0][1]
            prediction['K'] = result_tkdi[0][2]
            prediction['T'] = result_tkdi[0][3]
            prediction = sorted(prediction.items(), key = operator.itemgetter(1), reverse = True)
            self.current_symbol = prediction[0][0]
```

This part of the code is similar to the previous one but is executed when the current symbol is one of 'D', 'I', 'K', or 'T'. It calculates new probabilities for these symbols using a different model (`loaded_model_tkdi`) and assigns the symbol with the highest probability among 'D', 'I', 'K', and 'T' as the new `self.current_symbol`. This conditional block refines the prediction further based on the specific symbols involved, improving the accuracy of the overall prediction.

```
        if(self.current_symbol == 'M' or self.current_symbol == 'N' or self.current_symbol == 'S'):
            prediction1 = {}
            prediction1['M'] = result_smn[0][0]
            prediction1['N'] = result_smn[0][1]
            prediction1['S'] = result_smn[0][2]
            prediction1 = sorted(prediction1.items(), key = operator.itemgetter(1), reverse = True)
            if(prediction1[0][0] == 'S'):
                self.current_symbol = prediction1[0][0]
            else:
                self.current_symbol = prediction[0][0]
```

This part of the code is similar to the previous ones, but it's executed when the current symbol is one of 'M', 'N', or 'S'. It calculates new probabilities for these symbols using a different model (`loaded_model_smn`) and assigns the symbol with the highest probability among 'M', 'N', and 'S' as the new `self.current_symbol`. Additionally, if the highest probability corresponds to 'S', it takes precedence and becomes the new `self.current_symbol`. Otherwise, it keeps the symbol assigned in the previous step. This conditional block further refines the prediction based on specific symbols, improving the accuracy of the overall prediction.

```
        if(self.current_symbol == 'blank'):
```

This conditional block checks if the current symbol predicted by the model is 'blank'. If it is, it resets the count of occurrences for each symbol (`self.ct`) and checks if there are any other symbols with high counts. If any other symbol has a count close to the count of 'blank', indicating ambiguity, it resets the counts and continues the prediction process. If no other symbol is close in count, it sets the `blank_flag` to 1, indicating the start of a new word. If there was a word being formed (`self.word` is not empty), it adds a space to the string (`self.str`) and appends the word to it. Finally, it resets `self.word` to an empty string. This logic handles the case when the model predicts a blank symbol and decides whether to start a new word or continue with the current word.

```
            for i in ascii_uppercase:
                self.ct[i] = 0
```

This loop iterates over all the uppercase letters in the English alphabet. For each letter, it initializes the count (`self.ct`) to zero. This step ensures that the count for each symbol is reset to zero, indicating that no symbols have been predicted yet. This is typically done to prepare the count dictionary for the next round of predictions.

```
        self.ct[self.current_symbol] += 1
```

This line increments the count of the current symbol (`self.current_symbol`) in the count dictionary (`self.ct`). It keeps track of how many times a particular symbol has been predicted continuously. This count is used later to determine when a gesture has been recognized consistently over a certain number of frames.

```
        if(self.ct[self.current_symbol] > 60):
```

This conditional statement checks if the count of the current symbol (`self.current_symbol`) in the count dictionary (`self.ct`) has exceeded a threshold of 60. This threshold determines how many consecutive frames must predict the same symbol before considering it as a valid prediction. If the count exceeds this threshold, further checks will be performed to confirm the prediction.

```
            for i in ascii_uppercase:
                if i == self.current_symbol:
```

```python
                    continue
```
This loop iterates over each uppercase letter in the English alphabet (`ascii_uppercase`). If the current letter matches the predicted symbol (`self.current_symbol`), the loop continues to the next iteration without executing the subsequent code block.

```python
                tmp = self.ct[self.current_symbol] - self.ct[i]
```
This line calculates the difference (`tmp`) between the count of occurrences of the current symbol (`self.current_symbol`) and the count of occurrences of the symbol represented by the current iteration of the loop (`i`).

```python
                if tmp < 0:
                    tmp *= -1
```
This conditional statement ensures that the difference `tmp` is always a positive value. If `tmp` is negative, it is multiplied by -1 to make it positive.

```python
                if tmp <= 20:
                    self.ct['blank'] = 0
```
This condition checks if the absolute difference `tmp` between the count of the current symbol and the count of other symbols is less than or equal to 20. If it is, it resets the count of the `'blank'` symbol to 0.

```python
                    for i in ascii_uppercase:
                        self.ct[i] = 0
                    return
```
This loop resets the count of each symbol to 0 in the dictionary `self.ct` and then returns from the function.

```
            self.ct['blank'] = 0  This line sets the count of the "blank" symbol to 0 in the dictionary `self.ct`.
```

```python
            for i in ascii_uppercase:
                self.ct[i] = 0
```
This loop iterates over each uppercase letter in the English alphabet and sets the count of each letter to 0 in the `self.ct` dictionary. This dictionary is used to keep track of the count of each detected symbol during the video processing loop.

```python
            if self.current_symbol == 'blank':
```
This condition checks if the current symbol detected is 'blank'. If the condition is true, it executes the code block inside the if statement.

```python
                if self.blank_flag == 0:
                    self.blank_flag = 1
```
This code block checks if the `blank_flag` variable is set to 0. If it is, the code sets `blank_flag` to 1. This flag likely helps keep track of whether a blank symbol has been detected, ensuring it's only counted once until another symbol is detected.

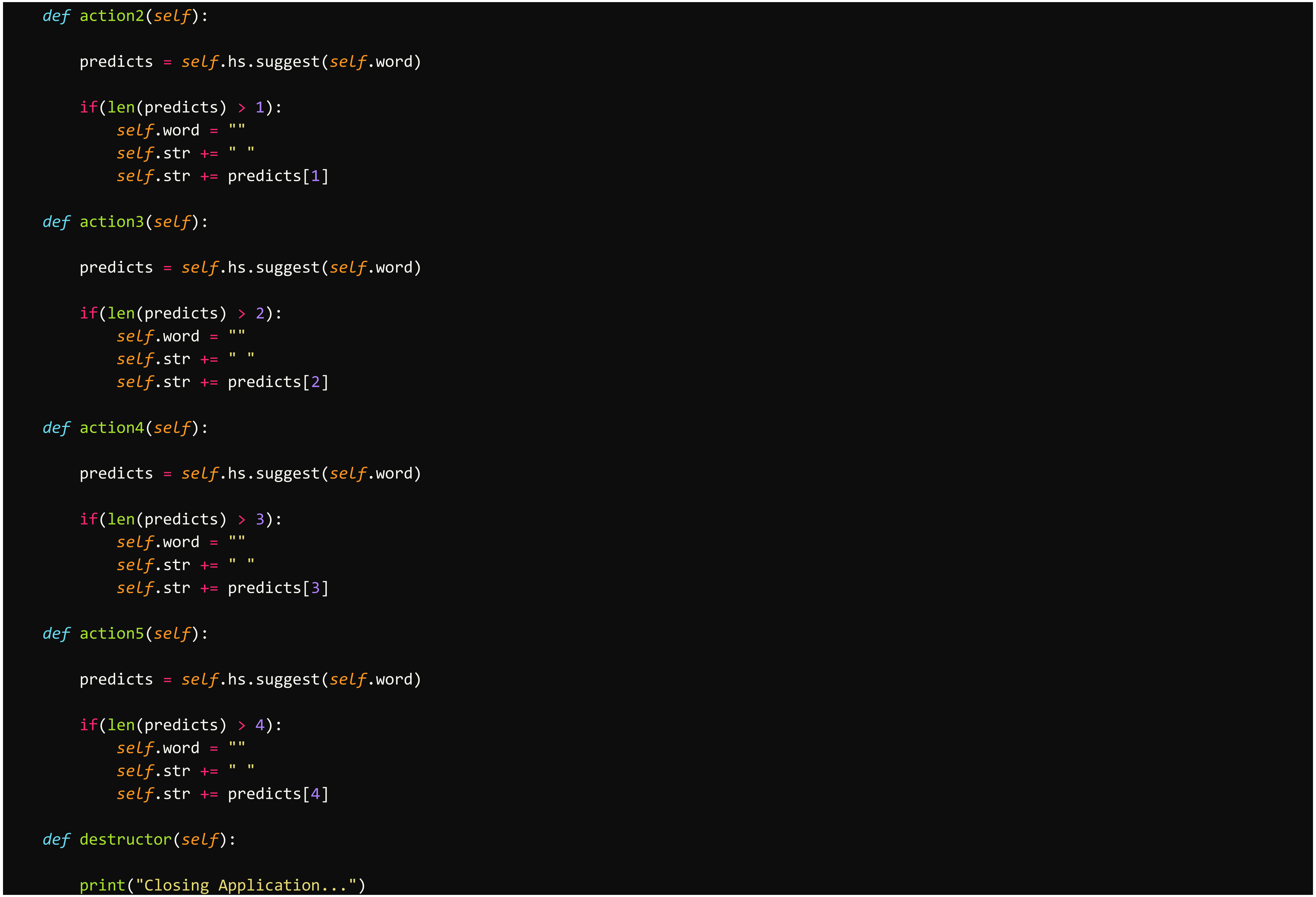
```python
                    if len(self.str) > 0:
                        self.str += " "
```
This code block checks if the length of the `str` variable is greater than 0. If it is, a space character is appended to the `str` variable. This likely ensures proper separation between recognized symbols in the text output.

```python
                    self.str += self.word
```
This line appends the content of the `word` variable to the `str` variable. It seems like `word` contains the recognized symbol or word from the sign language, and `str` is used to accumulate the recognized text.

```python
                    self.word = ""
```
This line clears the content of the `word` variable, presumably after appending its content to the `str` variable or after processing the recognized word. This ensures that the `word` variable is ready to store the next recognized word.

```python
            else:

                if(len(self.str) > 16):
                    self.str = ""

                self.blank_flag = 0

                self.word += self.current_symbol
```
This part of the code is executed when the current symbol is not 'blank'. It checks if the length of the accumulated string (`self.str`) is greater than 16. If it is, it resets the `self.str` variable to an empty string. Then, it resets the `blank_flag` to 0 and appends the current symbol to the `word` variable. This ensures that the `word` variable accumulates characters until it reaches a length of 16 or until a blank symbol is detected, at which point it resets `self.str` and starts accumulating characters for a new word.

```python
    def action1(self):

        predicts = self.hs.suggest(self.word)

        if(len(predicts) > 0):
            self.word = ""
            self.str += " "
            self.str += predicts[0]
```
This function `action1` seems to handle an action triggered by a button press (`bt1`). It suggests a word based on the input word (`self.word`) and updates the string (`self.str`) with the suggested word. If there are suggestions available, it clears the current word, adds a space to the string, and appends the first suggestion to the string.



```python
    def action2(self):

        predicts = self.hs.suggest(self.word)

        if(len(predicts) > 1):
            self.word = ""
            self.str += " "
            self.str += predicts[1]

    def action3(self):

        predicts = self.hs.suggest(self.word)

        if(len(predicts) > 2):
            self.word = ""
            self.str += " "
            self.str += predicts[2]

    def action4(self):

        predicts = self.hs.suggest(self.word)

        if(len(predicts) > 3):
            self.word = ""
            self.str += " "
            self.str += predicts[3]

    def action5(self):

        predicts = self.hs.suggest(self.word)

        if(len(predicts) > 4):
            self.word = ""
            self.str += " "
            self.str += predicts[4]

    def destructor(self):

        print("Closing Application...")
        self.root.destroy()
        self.vs.release()
        cv2.destroyAllWindows()
The `destructor` method is responsible for safely closing the application. It releases the video capture device (`self.vs.release()`) and
destroys all OpenCV windows (`cv2.destroyAllWindows()`). Finally, it destroys the tkinter root window (`self.root.destroy()`). Additionally,
it prints a message indicating that the application is closing.

print("Starting Application...")

(Application()).root.mainloop()
This code snippet initiates the application by creating an instance of the `Application` class and starting the tkinter main event loop
using `root.mainloop()`. Before starting the main loop, it prints "Starting Application..." to indicate that the application is beginning
its execution.
```

The provided code is an implementation of the "SignoSpeak: Bridging the Gap" project, which aims to facilitate sign language communication by converting sign language gestures into text. Let's break down the code and explain its functionality in detail.

The code begins by importing necessary libraries such as numpy, OpenCV (cv2), operating system functions (os), time, operator, string, tkinter, and PIL (Python Imaging Library). These libraries are used for various tasks including image processing, user interface creation, file handling, and mathematical operations.

Next, the `Application` class is defined. In its constructor `__init__`, the code initializes the video capture device using OpenCV's `VideoCapture` class, loads pre-trained convolutional neural network (CNN) models from JSON files and their corresponding weights using Keras' `model_from_json` function, and sets up the Tkinter GUI window for displaying the webcam feed and processed images.

The `video_loop` method continuously captures frames from the webcam, preprocesses the images by converting them to grayscale, applying Gaussian blur, and thresholding to extract hand gestures. These processed images are then fed into the loaded CNN models to predict the corresponding sign language symbols. The predicted symbols are displayed on the GUI window along with suggested text predictions.

The `predict` method takes a preprocessed image as input, resizes it to the required input size of the CNN models, and performs inference using the loaded models. It then selects the most probable sign language symbol from the predictions.

The `action1`, `action2`, `action3`, `action4`, and `action5` methods are callback functions for buttons in the GUI window. They invoke a text prediction function (`suggest`) based on the current word being formed by the recognized sign language symbols and update the displayed text accordingly.

The `destructor` method is called when the application is closed. It releases the video capture device, closes the GUI window, and destroys all OpenCV windows.

Finally, the code starts the application by creating an instance of the `Application` class and entering the Tkinter main event loop.

In summary, this code implements a real-time sign language to text conversion system using pre-trained CNN models for gesture recognition and a Tkinter-based GUI for user interaction. It demonstrates the application of deep learning techniques in assisting individuals with hearing impairments in communicating effectively through sign language.

Model.ipynb

Importing the Libraries

```
import tensorflow as tf
from keras.preprocessing.image import ImageDataGenerator
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "1"
```

This code segment imports TensorFlow and the ImageDataGenerator class from Keras, which is a high-level neural networks API running on top of TensorFlow. It also sets the CUDA_VISIBLE_DEVICES environment variable to "1", which typically indicates that the GPU with index 1 will be used for CUDA computation if available.

1. `import tensorflow as tf`: This imports the TensorFlow library, which is an open-source machine learning framework developed by Google for building and training machine learning models, including deep learning models.

2. `from keras.preprocessing.image import ImageDataGenerator`: This imports the ImageDataGenerator class from the Keras library, which is used for real-time data augmentation and preprocessing of image data during model training.

3. `import os`: This imports the os module, which provides a way of using operating system dependent functionality, such as reading or writing to the file system.

4. `os.environ["CUDA_VISIBLE_DEVICES"] = "1"`: This sets the CUDA_VISIBLE_DEVICES environment variable to "1", which typically indicates that the GPU with index 1 will be used for CUDA computation if available. This is useful for specifying which GPU to use when multiple GPUs are available in the system.

Overall, this code segment sets up the environment for using TensorFlow and Keras for deep learning tasks, and specifies the GPU to be used for computation if available.

```
tf.__version__
```

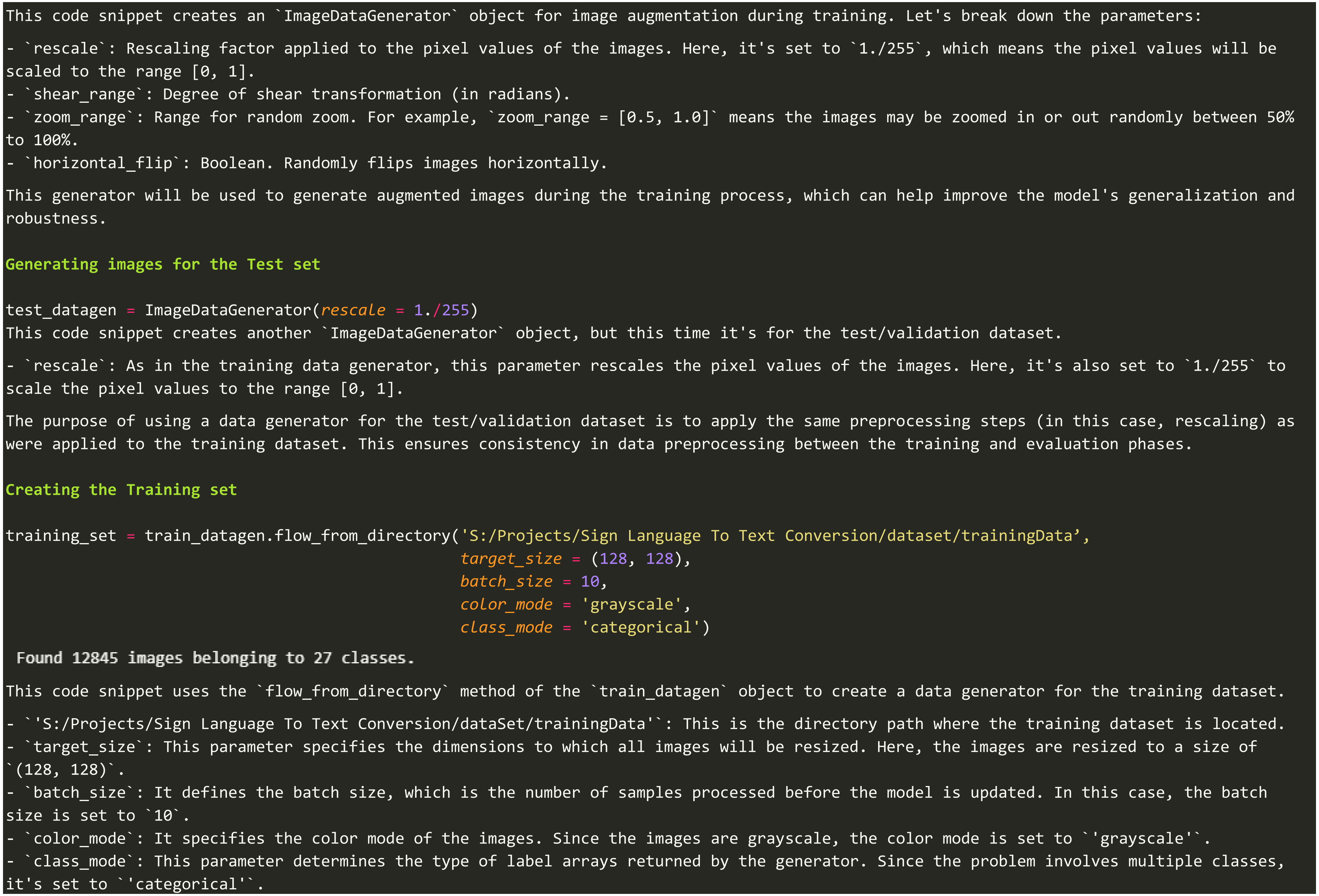
```
'2.3.1'
```

The code `tf.__version__` is used to check the version of TensorFlow installed in your environment. By running this code, you can determine the version of TensorFlow that is currently being used.

Part 1 - Data Pre-processing

Generating images for the Training set

```
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)
```



This code snippet creates an `ImageDataGenerator` object for image augmentation during training. Let's break down the parameters:

- `rescale`: Rescaling factor applied to the pixel values of the images. Here, it's set to `1./255`, which means the pixel values will be scaled to the range [0, 1].
- `shear_range`: Degree of shear transformation (in radians).
- `zoom_range`: Range for random zoom. For example, `zoom_range = [0.5, 1.0]` means the images may be zoomed in or out randomly between 50% to 100%.
- `horizontal_flip`: Boolean. Randomly flips images horizontally.

This generator will be used to generate augmented images during the training process, which can help improve the model's generalization and robustness.

Generating images for the Test set

```
test_datagen = ImageDataGenerator(rescale = 1./255)
```

This code snippet creates another `ImageDataGenerator` object, but this time it's for the test/validation dataset.

- `rescale`: As in the training data generator, this parameter rescales the pixel values of the images. Here, it's also set to `1./255` to scale the pixel values to the range [0, 1].

The purpose of using a data generator for the test/validation dataset is to apply the same preprocessing steps (in this case, rescaling) as were applied to the training dataset. This ensures consistency in data preprocessing between the training and evaluation phases.

Creating the Training set

```
training_set = train_datagen.flow_from_directory('S:/Projects/Sign Language To Text Conversion/dataset/trainingData’,
                                                 target_size = (128, 128),
                                                 batch_size = 10,
                                                 color_mode = 'grayscale',
                                                 class_mode = 'categorical')
```

```
 Found 12845 images belonging to 27 classes.
```

This code snippet uses the `flow_from_directory` method of the `train_datagen` object to create a data generator for the training dataset.

- `'S:/Projects/Sign Language To Text Conversion/dataSet/trainingData'`: This is the directory path where the training dataset is located.
- `target_size`: This parameter specifies the dimensions to which all images will be resized. Here, the images are resized to a size of `(128, 128)`.
- `batch_size`: It defines the batch size, which is the number of samples processed before the model is updated. In this case, the batch size is set to `10`.
- `color_mode`: It specifies the color mode of the images. Since the images are grayscale, the color mode is set to `'grayscale'`.
- `class_mode`: This parameter determines the type of label arrays returned by the generator. Since the problem involves multiple classes, it's set to `'categorical'`.

This data generator will generate batches of augmented images from the training dataset, which can be used to train a convolutional neural network (CNN) model for sign language classification.

```
test_set = test_datagen.flow_from_directory('S:/Projects/Sign Language To Text Conversion/dataSet/testingData',
                                            target_size = (128, 128),
                                            batch_size = 10,
                                            color_mode = 'grayscale',
                                            class_mode = 'categorical')
```

```
Found 4268 images belonging to 27 classes.
```

This code snippet is similar to the previous one but creates a data generator for the testing dataset.

- `'S:/Projects/Sign Language To Text Conversion/dataSet/testingData'`: This is the directory path where the testing dataset is located.
- `target_size`: This parameter specifies the dimensions to which all images will be resized. Here, the images are resized to a size of `(128, 128)`.
- `batch_size`: It defines the batch size, which is the number of samples processed before the model is updated. In this case, the batch size is set to `10`.
- `color_mode`: It specifies the color mode of the images. Since the images are grayscale, the color mode is set to `'grayscale'`.
- `class_mode`: This parameter determines the type of label arrays returned by the generator. Since the problem involves multiple classes, it's set to `'categorical'`.

Similar to the training data generator, this data generator will generate batches of augmented images from the testing dataset, which can be used to evaluate the performance of the trained CNN model.

Part 2 - Building the CNN

Initializing the CNN

```
classifier = tf.keras.models.Sequential()
```

This code initializes a sequential model using TensorFlow's Keras API. A sequential model is appropriate for building deep learning models layer by layer, where each layer has exactly one input tensor and one output tensor.

- `tf.keras.models.Sequential()`: This function initializes an empty sequential model. We can add layers to this model one by one using the `.add()` method or by passing a list of layers to the constructor.

This `classifier` object will serve as the container for all the layers of our neural network model. We'll add layers to it to define the architecture of our model.

Step 1 – Convolution

```
classifier.add(tf.keras.layers.Conv2D(filters=32,
                                      kernel_size=3,
                                      padding="same",
                                      activation="relu",
                                      input_shape=[128, 128, 1]))
```

This code snippet adds a convolutional layer to the neural network model.
- `tf.keras.layers.Conv2D`: This function creates a convolutional layer for 2D inputs. It is typically used for image data.
- `filters=32`: This parameter specifies the number of filters (or kernels) that the convolutional layer will learn. Each filter detects different features in the input image. Here, we have 32 filters.
- `kernel_size=3`: This parameter sets the size of the convolutional kernels. In this case, the kernel size is 3x3 pixels.
- `padding="same"`: This parameter determines the padding strategy. "same" padding means that the input size is padded in such a way that the output has the same spatial dimensions as the input.
- `activation="relu"`: This parameter specifies the activation function to be applied element-wise after performing the convolution operation. "relu" stands for Rectified Linear Unit, which introduces non-linearity into the network by outputting the input directly if it is positive, otherwise, it outputs zero.
- `input_shape=[128, 128, 1]`: This parameter specifies the shape of the input data. For grayscale images with size 128x128 pixels, the input shape is [128, 128, 1], where 1 represents the number of channels (1 for grayscale, 3 for RGB).

Overall, this layer will perform convolution operations on the input images, applying 32 filters with a kernel size of 3x3 pixels, using the ReLU activation function. The input shape for this layer is [128, 128, 1].

Step 2 – Pooling

```
classifier.add(tf.keras.layers.MaxPool2D(pool_size=2,
                                         strides=2,
                                         padding='valid'))
```

This code adds a MaxPooling layer to the neural network model.
- `tf.keras.layers.MaxPool2D`: This function creates a MaxPooling layer, which performs max pooling operations on the input data.
- `pool_size=2`: This parameter specifies the size of the pooling window, which determines the spatial extent over which each pooling operation is performed. Here, the pooling window size is 2x2 pixels.
- `strides=2`: This parameter specifies the stride of the pooling operation, which determines the step size taken by the pooling window as it moves across the input. Here, the stride size is 2 in both the horizontal and vertical directions.
- `padding='valid'`: This parameter determines the padding strategy. 'valid' padding means that no padding is applied to the input data before performing pooling operations.

Overall, this MaxPooling layer will reduce the spatial dimensions of the input data by half in both the horizontal and vertical directions, effectively downsampling the feature maps produced by the preceding convolutional layer.

Adding a second convolutional layer

```
classifier.add(tf.keras.layers.Conv2D(filters=32,
                                      kernel_size=3,
                                      padding="same",
                                      activation="relu"))

classifier.add(tf.keras.layers.MaxPool2D(pool_size=2,
                                         strides=2,
                                         padding='valid'))
```

These lines of code add another pair of convolutional and max-pooling layers to the neural network model.

1. Convolutional Layer:
- `tf.keras.layers.Conv2D`: This function creates a 2D convolutional layer.
- `filters=32`: This parameter specifies the number of filters (or kernels) used in the convolutional layer. Here, 32 filters are used.
- `kernel_size=3`: This parameter specifies the size of the convolutional kernel, which determines the spatial extent over which each convolutional operation is performed. Here, the kernel size is 3x3 pixels.
- `padding="same"`: This parameter determines the padding strategy. 'same' padding means that the input data is padded with zeros so that the output has the same spatial dimensions as the input.
- `activation="relu"`: This parameter specifies the activation function used in the layer. Here, the Rectified Linear Unit (ReLU) activation function is used.

2. MaxPooling Layer:
- `tf.keras.layers.MaxPool2D`: This function creates a MaxPooling layer.
- `pool_size=2`: This parameter specifies the size of the pooling window, which determines the spatial extent over which each pooling operation is performed. Here, the pooling window size is 2x2 pixels.
- `strides=2`: This parameter specifies the stride of the pooling operation, which determines the step size taken by the pooling window as it moves across the input. Here, the stride size is 2 in both the horizontal and vertical directions.
- `padding='valid'`: This parameter determines the padding strategy. 'valid' padding means that no padding is applied to the input data before performing pooling operations.

These layers are commonly used in convolutional neural networks (CNNs) for feature extraction and dimensionality reduction, helping to capture and preserve important spatial information in the input images.

Step 3 – Flattening

```
classifier.add(tf.keras.layers.Flatten())
```

This line adds a flatten layer to the neural network model.

- `tf.keras.layers.Flatten`: This function creates a flatten layer.

The Flatten layer serves as a bridge between the convolutional layers and the fully connected layers in the neural network. It takes the output of the preceding convolutional and pooling layers, which are typically 3D arrays (tensors) representing spatial features, and flattens them into a 1D array.

For example, if the output of the previous layer is a tensor with dimensions (batch_size, height, width, num_filters), the Flatten layer will reshape it into a 1D tensor with dimensions (batch_size, height * width * num_filters), where each entry corresponds to a feature.

Flattening the output is necessary before passing it to fully connected layers (dense layers) because these layers expect input data to be in the form of a vector rather than a multidimensional tensor.

Step 4 – Full Connection

```
classifier.add(tf.keras.layers.Dense(units=128,
                                     activation='relu'))
classifier.add(tf.keras.layers.Dropout(0.40))
classifier.add(tf.keras.layers.Dense(units=96, activation='relu'))
classifier.add(tf.keras.layers.Dropout(0.40))
classifier.add(tf.keras.layers.Dense(units=64, activation='relu'))
classifier.add(tf.keras.layers.Dense(units=27, activation='softmax')) # softmax for more than 2
```

These lines add dense layers to the neural network model, along with dropout layers for regularization. Let's break down each component:

- `tf.keras.layers.Dense`: This function creates a fully connected (dense) layer.

- `units`: Specifies the number of neurons in the dense layer.

- `activation`: Specifies the activation function to be applied to the output of the dense layer. In this case, `'relu'` (Rectified Linear Unit) activation function is used, which introduces non-linearity to the network.

- `tf.keras.layers.Dropout`: This function creates a dropout layer, which helps prevent overfitting by randomly dropping a fraction of input units during training.

- `Dropout(0.40)`: Specifies the dropout rate, which is the fraction of input units to drop. Here, 0.40 means dropping 40% of the input units.

- `units=27`: In the last dense layer, there are 27 units, corresponding to the number of classes (letters) in the dataset. The activation function used is `'softmax'`, which converts the raw scores (logits) into probabilities for each class. Softmax ensures that the output probabilities sum up to 1, making it suitable for multi-class classification problems.

Overall, these layers create a neural network architecture with multiple dense layers for learning complex patterns in the data, along with dropout layers to prevent overfitting. The final dense layer with softmax activation produces probability distributions over the 27 classes, enabling the model to make predictions.

Part 3 - Training the CNN

Compiling the CNN

```
classifier.compile(optimizer = 'adam',
                   loss = 'categorical_crossentropy',
                   metrics = ['accuracy'])
```

This line compiles the neural network model, specifying the optimizer, loss function, and evaluation metric.

- `optimizer='adam'`: Adam optimizer is used to minimize the loss function during training. Adam is an adaptive learning rate optimization algorithm that is well-suited for training deep neural networks.

- `loss='categorical_crossentropy'`: Categorical crossentropy is the loss function used for multi-class classification problems. It measures the difference between the true distribution and the predicted distribution of the classes. Since the problem involves predicting one out of 27 classes (letters), categorical crossentropy is an appropriate choice.

- `metrics=['accuracy']`: Accuracy is the evaluation metric used to assess the performance of the model during training and testing. It measures the proportion of correctly classified samples out of the total number of samples. In this case, accuracy provides insight into how well the model is performing in terms of correctly predicting the sign language letters.

By compiling the model with these settings, you prepare it for training using the specified optimizer, loss function, and evaluation metric.

Training the CNN on the Training set and evaluating it on the Test set

```
classifier.summary()
```

The `summary()` method provides a summary of the architecture of the neural network model, including details about each layer, the number of parameters, and the output shape of each layer. This summary helps in understanding the overall structure of the model and the flow of information through it during the forward pass.

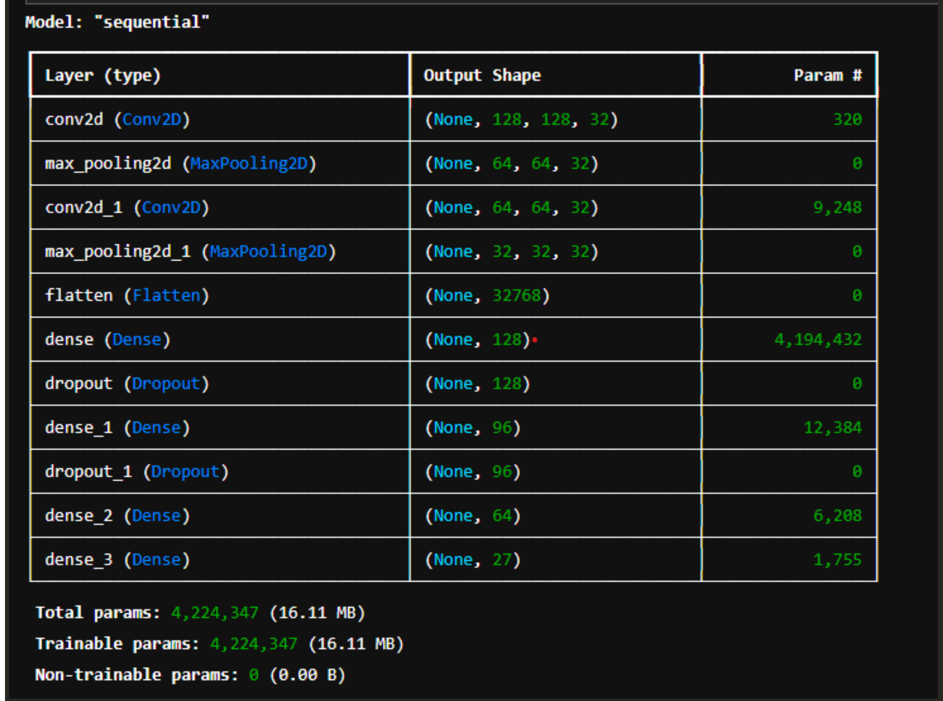
- Layer (type): This column indicates the type of layer in the neural network architecture, such as Conv2D, MaxPooling2D, Flatten, Dense, etc.

- Output Shape: This column shows the shape of the output produced by each layer. For convolutional layers (Conv2D) and pooling layers (MaxPooling2D), it represents the spatial dimensions (height, width, channels) of the output feature maps. For fully connected layers (Dense), it shows the number of neurons in the layer.

- Param #: This column displays the number of parameters (weights and biases) associated with each layer. It indicates the total number of trainable parameters in the model.

- Trainable: This column specifies whether the parameters of a layer are trainable or not. If set to True, the parameters will be updated during training based on the computed loss. If set to False, the parameters will remain fixed and not be updated during training.

The summary provides a concise overview of the model's architecture, making it easier to understand its structure and the number of parameters involved in the learning process. This information is crucial for debugging, optimizing, and fine-tuning the model for better performance.



```
Model: "sequential"
┌───────────────────────────────┬──────────────────────────┬─────────────┐
│ Layer (type)                  │ Output Shape             │     Param # │
├───────────────────────────────┼──────────────────────────┼─────────────┤
│ conv2d (Conv2D)               │ (None, 128, 128, 32)     │         320 │
├───────────────────────────────┼──────────────────────────┼─────────────┤
│ max_pooling2d (MaxPooling2D)  │ (None, 64, 64, 32)       │           0 │
├───────────────────────────────┼──────────────────────────┼─────────────┤
│ conv2d_1 (Conv2D)             │ (None, 64, 64, 32)       │       9,248 │
├───────────────────────────────┼──────────────────────────┼─────────────┤
│ max_pooling2d_1 (MaxPooling2D)│ (None, 32, 32, 32)       │           0 │
├───────────────────────────────┼──────────────────────────┼─────────────┤
│ flatten (Flatten)             │ (None, 32768)            │           0 │
├───────────────────────────────┼──────────────────────────┼─────────────┤
│ dense (Dense)                 │ (None, 128)              │   4,194,432 │
├───────────────────────────────┼──────────────────────────┼─────────────┤
│ dropout (Dropout)             │ (None, 128)              │           0 │
├───────────────────────────────┼──────────────────────────┼─────────────┤
│ dense_1 (Dense)               │ (None, 96)               │      12,384 │
├───────────────────────────────┼──────────────────────────┼─────────────┤
│ dropout_1 (Dropout)           │ (None, 96)               │           0 │
├───────────────────────────────┼──────────────────────────┼─────────────┤
│ dense_2 (Dense)               │ (None, 64)               │       6,208 │
├───────────────────────────────┼──────────────────────────┼─────────────┤
│ dense_3 (Dense)               │ (None, 27)               │       1,755 │
└───────────────────────────────┴──────────────────────────┴─────────────┘
 Total params: 4,224,347 (16.11 MB)
 Trainable params: 4,224,347 (16.11 MB)
 Non-trainable params: 0 (0.00 B)
```

```
classifier.fit(training_set,
               epochs = 5,
               validation_data = test_set)
```

The `fit()` method is used to train the neural network model on the training data. It takes the following arguments:

- training_set: The training data generator, which generates batches of training samples and labels.

- epochs: The number of epochs (iterations over the entire training dataset) to train the model.

- validation_data: Optional. If provided, it specifies the validation data generator, which generates batches of validation samples and labels. The model performance on this data is evaluated after each epoch.

Here's how the `fit()` method works:

1. Training Loop: The method iterates over the specified number of epochs. In each epoch, it iterates over the batches of training data generated by the `training_set`.

2. Forward Pass: For each batch of training data, the method performs a forward pass through the neural network, computing the predicted outputs.

3. Loss Calculation: It computes the loss (error) between the predicted outputs and the actual labels for the training data.

4. Backpropagation: Using the calculated loss, the method performs backpropagation to update the weights of the neural network, minimizing the loss function.

5. Validation: If `validation_data` is provided, the method evaluates the model's performance on the validation data after each epoch. This allows monitoring of the model's generalization performance and helps detect overfitting.

6. Epoch Completion: After completing each epoch, the method prints the training and validation metrics (loss and accuracy) to track the model's performance.

By calling the `fit()` method with appropriate arguments, you can train the neural network model on the given training data, allowing it to learn from the input-output pairs and improve its performance over successive epochs.

```
Epoch 1/5
1285/1285 [==============================] - 190s 148ms/step - loss: 0.0691 - accuracy: 0.9811 - val_loss: 0.0046 - val_accuracy: 0.9986
Epoch 2/5
1285/1285 [==============================] - 167s 130ms/step - loss: 0.0539 - accuracy: 0.9853 - val_loss: 0.0069 - val_accuracy: 0.9977
Epoch 3/5
1285/1285 [==============================] - 164s 128ms/step - loss: 0.0433 - accuracy: 0.9889 - val_loss: 0.0207 - val_accuracy: 0.9946
Epoch 4/5
1285/1285 [==============================] - 164s 127ms/step - loss: 0.0503 - accuracy: 0.9863 - val_loss: 0.0059 - val_accuracy: 0.9988
Epoch 5/5
1285/1285 [==============================] - 168s 131ms/step - loss: 0.0370 - accuracy: 0.9900 - val_loss: 0.0018 - val_accuracy: 0.9993

<tensorflow.python.keras.callbacks.History at 0x21004d31730>
```

Saving the Model

```python
model_json = classifier.to_json()
with open("model_new.json", "w") as json_file:
    json_file.write(model_json)
print('Model Saved')
classifier.save_weights('model_new.h5')
print('Weights saved')
```

In the provided code snippet, the trained neural network model is being saved to disk. Here's what each part does:

1. Convert Model to JSON: The `to_json()` method of the `classifier` model is used to serialize the model architecture to JSON format. This JSON representation includes information about the model's layers, their configurations, and the connections between them.
model_json = classifier.to_json()

2. Save Model Architecture to File: The serialized model architecture (in JSON format) is then written to a file named `"model_new.json"`.

```
with open("model_new.json", "w") as json_file:
    json_file.write(model_json)
```

3. Save Model Weights: The `save_weights()` method is used to save the trained weights of the model to an HDF5 file format. These weights represent the learned parameters of the neural network, which capture the knowledge acquired during training.
classifier.save_weights('model_new.h5')

4. Print Confirmation Messages: Finally, confirmation messages are printed to indicate that the model architecture and weights have been successfully saved to disk.

```
print('Model Saved')
print('Weights saved')
```

After running this code snippet, you should have two files saved in your current directory: `"model_new.json"` containing the model architecture in JSON format and `"model_new.h5"` containing the trained weights of the model. These files can be later loaded to reconstruct the trained model for inference or further training.

```
Model Saved
Weights saved
```

model_new.json

```
{"class_name": "Sequential", "config": {"name": "sequential", "layers": [{"class_name": "InputLayer", "config": {"batch_input_shape": [null,
128, 128, 1], "dtype": "float32", "sparse": false, "ragged": false, "name": "conv2d_input"}}, {"class_name": "Conv2D", "config": {"name":
"conv2d", "trainable": true, "batch_input_shape": [null, 128, 128, 1], "dtype": "float32", "filters": 32, "kernel_size": [3, 3], "strides":
[1, 1], "padding": "same", "data_format": "channels_last", "dilation_rate": [1, 1], "groups": 1, "activation": "relu", "use_bias": true,
"kernel_initializer": {"class_name": "GlorotUniform", "config": {"seed": null}}, "bias_initializer": {"class_name": "Zeros", "config": {}},
"kernel_regularizer": null, "bias_regularizer": null, "activity_regularizer": null, "kernel_constraint": null, "bias_constraint": null}},
{"class_name": "MaxPooling2D", "config": {"name": "max_pooling2d", "trainable": true, "dtype": "float32", "pool_size": [2, 2], "padding":
"valid", "strides": [2, 2], "data_format": "channels_last"}}, {"class_name": "Conv2D", "config": {"name": "conv2d_1", "trainable": true,
"dtype": "float32", "filters": 32, "kernel_size": [3, 3], "strides": [1, 1], "padding": "same", "data_format": "channels_last",
"dilation_rate": [1, 1], "groups": 1, "activation": "relu", "use_bias": true, "kernel_initializer": {"class_name": "GlorotUniform",
"config": {"seed": null}}, "bias_initializer": {"class_name": "Zeros", "config": {}}, "kernel_regularizer": null, "bias_regularizer": null,
"activity_regularizer": null, "kernel_constraint": null, "bias_constraint": null}}, {"class_name": "MaxPooling2D", "config": {"name":
"max_pooling2d_1", "trainable": true, "dtype": "float32", "pool_size": [2, 2], "padding": "valid", "strides": [2, 2], "data_format":
"channels_last"}}, {"class_name": "Flatten", "config": {"name": "flatten", "trainable": true, "dtype": "float32", "data_format":
"channels_last"}}, {"class_name": "Dense", "config": {"name": "dense", "trainable": true, "dtype": "float32", "units": 128, "activation":
"relu", "use_bias": true, "kernel_initializer": {"class_name": "GlorotUniform", "config": {"seed": null}}, "bias_initializer":
{"class_name": "Zeros", "config": {}}, "kernel_regularizer": null, "bias_regularizer": null, "activity_regularizer": null,
"kernel_constraint": null, "bias_constraint": null}}, {"class_name": "Dense", "config": {"name": "dense_1", "trainable": true, "dtype":
"float32", "units": 128, "activation": "relu", "use_bias": true, "kernel_initializer": {"class_name": "GlorotUniform", "config": {"seed":
null}}, "bias_initializer": {"class_name": "Zeros", "config": {}}, "kernel_regularizer": null, "bias_regularizer": null,
"activity_regularizer": null, "kernel_constraint": null, "bias_constraint": null}}, {"class_name": "Dropout", "config": {"name": "dropout",
"trainable": true, "dtype": "float32", "rate": 0.4, "noise_shape": null, "seed": null}}, {"class_name": "Dense", "config": {"name":
"dense_2", "trainable": true, "dtype": "float32", "units": 96, "activation": "relu", "use_bias": true, "kernel_initializer": {"class_name":
"GlorotUniform", "config": {"seed": null}}, "bias_initializer": {"class_name": "Zeros", "config": {}}, "kernel_regularizer": null,
"bias_regularizer": null, "activity_regularizer": null, "kernel_constraint": null, "bias_constraint": null}}, {"class_name": "Dropout",
"config": {"name": "dropout_1", "trainable": true, "dtype": "float32", "rate": 0.4, "noise_shape": null, "seed": null}}, {"class_name":
"Dense", "config": {"name": "dense_3", "trainable": true, "dtype": "float32", "units": 64, "activation": "relu", "use_bias": true,
"kernel_initializer": {"class_name": "GlorotUniform", "config": {"seed": null}}, "bias_initializer": {"class_name": "Zeros", "config": {}},
"kernel_regularizer": null, "bias_regularizer": null, "activity_regularizer": null, "kernel_constraint": null, "bias_constraint": null}},
{"class_name": "Dense", "config": {"name": "dense_4", "trainable": true, "dtype": "float32", "units": 27, "activation": "softmax",
"use_bias": true, "kernel_initializer": {"class_name": "GlorotUniform", "config": {"seed": null}}, "bias_initializer": {"class_name":
"Zeros", "config": {}}, "kernel_regularizer": null, "bias_regularizer": null, "activity_regularizer": null, "kernel_constraint": null,
"bias_constraint": null}}]}, "keras_version": "2.4.0", "backend": "tensorflow"}
```

This JSON snippet represents the architecture of a neural network model. Let's break down its structure:

1. Sequential Model: The model is defined as a Sequential model, which means that layers are added sequentially one after another.

2. Input Layer: The first layer is an InputLayer, which specifies the shape of the input data. In this case, the input shape is `[null, 128, 128, 1]`, where `null` represents the batch size (it will be determined dynamically), `128x128` represents the dimensions of each input image, and `1` indicates that the images are grayscale.

3. Convolutional Layers: The model contains two Conv2D layers, each followed by a MaxPooling2D layer. These layers are responsible for learning features from the input images through convolution and downsampling.

4. Flatten Layer: After the convolutional layers, there is a Flatten layer, which flattens the output of the previous layer into a one-dimensional vector. This prepares the data for input into the dense (fully connected) layers.

5. Dense Layers: The model has three Dense layers with ReLU activation functions. These layers are fully connected and learn complex patterns from the flattened input data.

6. Dropout Layers: Two Dropout layers are added after the dense layers to prevent overfitting by randomly dropping a fraction of neurons during training.

7. Output Layer: The final Dense layer has 27 units and uses a softmax activation function, indicating that the model is performing multiclass classification with 27 classes.

8. Compilation Parameters: The model is compiled with the Adam optimizer and categorical cross-entropy loss function, along with accuracy as the evaluation metric.

This JSON representation captures the architecture of the model, including layer types, configurations, and parameters. It can be used to reconstruct the exact model in TensorFlow/Keras.