American Sign Language Interpreter

Bonafide record of work done by

Abhinav	**21z203**
Anbukkumar P K	**21z208**
Arulpathi A	**21z209**
Kavin Dev	**21z224**
Rohith Sundharamurthy	**21z244**
Sharan S	**21z254**

19Z610 - MACHINE LEARNING LABORATORY

Dissertation submitted in partial fulfillment of the requirement for the award of degree of

BACHELOR OF ENGINEERING

Branch: COMPUTER SCIENCE AND ENGINEERING
Of Anna University

APRIL 2024

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PSG COLLEGE OF TECHNOLOGY

(Autonomous Institution)

COIMBATORE – 641 004

Table Of Contents

Problem Statement

The project aims to create a Sign Language Interpreter using machine learning. It will recognize hand gestures from video input, facilitating real-time translation of sign language into text or speech for enhanced communication accessibility. By leveraging advanced techniques, the system seeks to accurately interpret diverse sign language gestures, enabling seamless interaction between sign language users and non-signers. This initiative promises to bridge communication gaps, empowering individuals with hearing impairments to communicate effectively in various settings. Through continuous refinement and innovation, the Sign Language Interpreter endeavors to offer a reliable and intuitive solution for fostering inclusivity and accessibility in communication.

Dataset Description

The original MNIST image dataset of handwritten digits is a popular benchmark for image-based machine learning methods but researchers have renewed efforts to update it and develop drop-in replacements that are more challenging for computer vision and original for real-world applications. As noted in one recent replacement called the Fashion-MNIST dataset, the Zalando researchers quoted the startling claim that "Most pairs of MNIST digits (784 total pixels per sample) can be distinguished pretty well by just one pixel". To stimulate the community to develop more drop-in replacements, the Sign Language MNIST is presented here and follows the same CSV format with labels and pixel values in single rows. The American Sign Language letter database of hand gestures represent a multi-class problem with 24 classes of letters (excluding J and Z which require motion). The dataset format is patterned to match closely with the classic MNIST. Each training and test case represents a label (0-25) as a one-to-one map for each alphabetic letter A-Z (and no cases for 9=J or 25=Z because of gesture motions). The training data (27,455 cases) and test data (7172 cases) are approximately half the size of the standard MNIST but otherwise similar with a header row of label, pixel1,pixel2….pixel784 which represent a single 28x28 pixel image with grayscale values between 0-255. The original hand gesture image data represented multiple users repeating the gesture against different backgrounds. The Sign Language MNIST data came from greatly extending the small number (1704) of the color images included as not cropped around the hand region of interest. To create new data, an image pipeline was used based on ImageMagick and included cropping to hands-only, gray-scaling, resizing, and then creating at least 50+ variations to enlarge the quantity. The modification and expansion strategy was filters ('Mitchell', 'Robidoux', 'Catrom', 'Spline', 'Hermite'), along with 5% random pixelation, +/- 15% brightness/contrast, and finally 3 degrees rotation. Because of the tiny size of the images, these modifications effectively alter the resolution and class separation in interesting, controllable ways.

Dataset link: https://www.kaggle.com/datasets/datamunge/sign-language-mnist

Models Used

Multilayer Perceptron for Sign Language Gesture Recognition

This Multilayer Perceptron (MLP) model is designed for sign language gesture recognition, aiming to interpret hand gestures from video input and facilitate real-time translation into text or speech. It comprises four dense layers, each with 128 units and ReLU activation, allowing the network to learn complex patterns in the input data. The input layer has 784 neurons, representing flattened images of hand gestures.[4] The output layer consists of 26 units with softmax activation, corresponding to the 26 classes of sign language gestures being recognized. The model is compiled with stochastic gradient descent (SGD) optimizer with a learning rate of 0.001 and categorical cross-entropy loss function. Training is performed on a dataset consisting of input features (`train_X`) and corresponding target labels (`train_Y`) using a batch size of 32 over 100 epochs to optimize model parameters.[5]

```
model = Sequential()

model.add(Dense(units=128,activation="relu",input_shape=(784,)))

model.add(Dense(units=128,activation="relu"))

model.add(Dense(units=128,activation="relu"))

model.add(Dense(units=26,activation="softmax"))
```

Convolutional Neural Network (CNN) for Sign Language Gesture Recognition

This Convolutional Neural Network (CNN) architecture is tailored for the task of sign language gesture recognition, facilitating real-time translation of hand gestures into text or speech. The network begins with a convolutional layer comprising 8 filters of size 3x3, employing ReLU activation to capture spatial features in the input images of size 28x28x1. Subsequently, a max-pooling layer with a pool size of 2x2 is applied to downsample the feature maps. Another convolutional layer follows, with 16 filters of size 3x3, leveraging ReLU activation to extract higher-level features.[6] A dropout layer with a dropout rate of 0.5 is introduced to mitigate overfitting. Further, max-pooling with a larger pool size of 4x4 is performed to reduce spatial dimensions. The extracted features are then flattened and fed into a dense layer comprising 128 neurons with ReLU activation. Finally, a dense output layer with 26 units and softmax activation is employed to predict the probabilities of the 26 classes of sign language gestures. The model is compiled with stochastic gradient descent (SGD) optimizer, utilizing categorical cross-entropy loss to measure the difference between predicted and actual distributions, while accuracy is used as the evaluation metric.[4]

```
classifier = Sequential()

classifier.add(Conv2D(filters=8,
kernel_size=(3,3),strides=(1,1),padding='same',input_shape=(28,2
8,1),activation='relu', data_format='channels_last'))

classifier.add(MaxPooling2D(pool_size=(2,2)))

classifier.add(Conv2D(filters=16,
kernel_size=(3,3),strides=(1,1),padding='same',activation='relu'
))

classifier.add(Dropout(0.5))

classifier.add(MaxPooling2D(pool_size=(4,4)))

classifier.add(Dense(128, activation='relu'))

classifier.add(Flatten())

classifier.add(Dense(26, activation='softmax'))

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

Tools Used:

1. Python: Python is a popular programming language widely used in machine learning and deep learning projects due to its simplicity, readability, and extensive libraries such as TensorFlow, Keras, and OpenCV, which are commonly used for developing neural networks and computer vision applications.

2. TensorFlow / Keras: TensorFlow and Keras are widely used open-source deep learning frameworks in Python. They provide high-level APIs for building and training neural networks, making it easier and faster to develop complex models like convolutional neural networks (CNNs) and multilayer perceptrons (MLPs).

3. PyCharm: PyCharm is a powerful integrated development environment (IDE) specifically designed for Python development. It offers features such as code highlighting, debugging, version control integration, and intelligent code completion, which can streamline the development process.

4. Google Colab: Google Colab is a cloud-based platform provided by Google that allows users to write and execute Python code in a browser-based environment. It provides free access to GPU and TPU resources, making it ideal for training deep learning models on large datasets without requiring high-end hardware.

5. OpenCV: OpenCV (Open Source Computer Vision Library) is a popular open-source computer vision and machine learning software library. It provides a wide range of functions for image processing, feature detection, object recognition, and more, making it useful for preprocessing image data in sign language recognition projects.

Challenges Faced:

During the implementation of sign language recognition models, we often encounter several challenges. One major hurdle is the variability and complexity of sign language gestures, which can differ significantly based on factors such as region, culture, and individual signing styles. Ensuring robustness and accuracy across diverse gestures requires extensive data collection and annotation, as well as sophisticated preprocessing techniques to handle variations in lighting, background clutter, and hand orientation. Another challenge is the limited availability of annotated sign language datasets, which may not adequately represent the full range of gestures and expressions. This scarcity necessitates techniques such as data augmentation and transfer learning to leverage existing datasets and adapt models to new domains. Additionally, optimizing model performance while balancing computational resources is crucial, especially for real-time applications. This entails fine-tuning hyperparameters, selecting appropriate architectures, and optimizing inference speed without compromising accuracy. Lastly, ensuring accessibility and inclusivity in the design and deployment of sign language recognition systems requires collaboration with stakeholders from the deaf and hard-of-hearing communities to incorporate their feedback and address usability concerns effectively. Overcoming these challenges demands a multidisciplinary approach, combining expertise in machine learning, computer vision, human-computer interaction, and cultural sensitivity to develop robust and user-friendly solutions for sign language communication.

Contribution of Team Members

21Z203 - Abhinav	Dataset preparation, model literacy and Developing real time Application using models
21Z208 - Anbukkumar P K	Implementing models and Developing real time Application using models
21Z209 - Arulpathi A	Implementing models and Developing real time Application using models
21Z224 - Kavin Dev	Developing HandHistogram and Developing real time Application using models
21Z244 - Rohith Sundharamurthy	Developing real time Application using models
21Z254 - Sharan S	Developing real time Application using models

Annexure I: Code

main.py

```
import cv2
import numpy as np
from keras.models import load_model
from skimage.transform import resize, pyramid_reduce
import PIL
from PIL import Image

model = load_model('CNNmodel.h5')


def prediction(pred):
    return(chr(pred+ 65))


def keras_predict(model, image):
    data = np.asarray( image, dtype="int32" )

    pred_probab = model.predict(data)[0]
    pred_class = list(pred_probab).index(max(pred_probab))
    return max(pred_probab), pred_class

def keras_process_image(img):

    image_x = 28
    image_y = 28
    img = cv2.resize(img, (1,28,28), interpolation = cv2.INTER_AREA)

    return img


def crop_image(image, x, y, width, height):
    return image[y:y + height, x:x + width]

def main():
    l = []

    while True:

        cam_capture = cv2.VideoCapture(0)
```

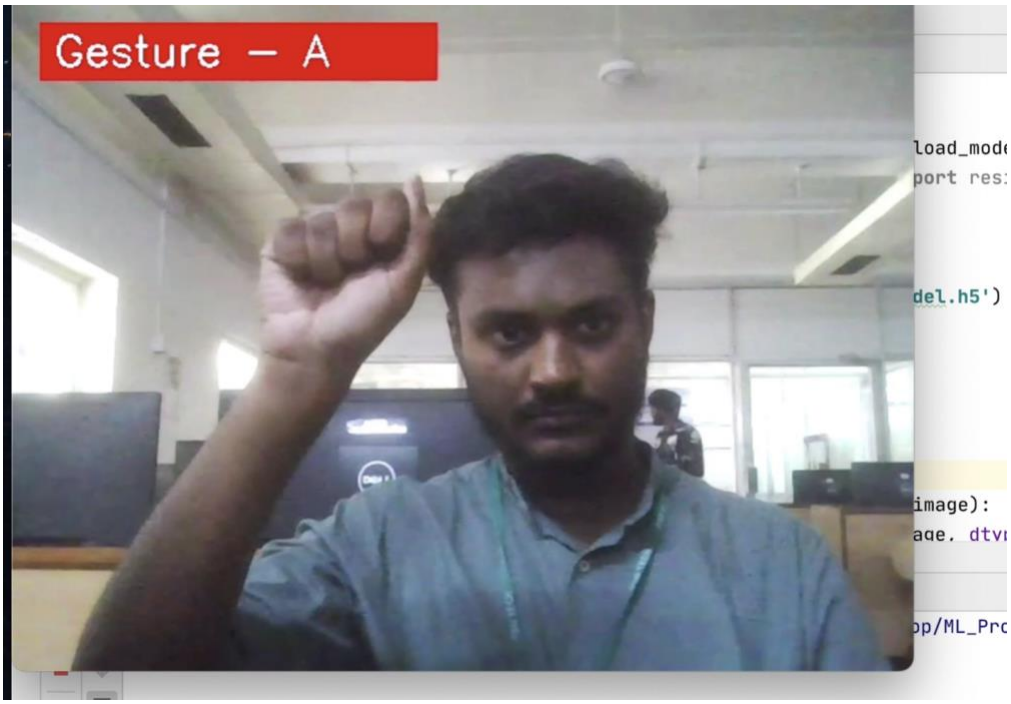
```
        _, image_frame = cam_capture.read()
    # Select ROI
        im2 = crop_image(image_frame, 300,300,300,300)
        image_grayscale = cv2.cvtColor(im2, cv2.COLOR_BGR2GRAY)

        image_grayscale_blurred = cv2.GaussianBlur(image_grayscale, (15,15), 0)
        im3 = cv2.resize(image_grayscale_blurred, (28,28), interpolation = cv2.INTER_AREA)



        im4 = np.resize(im3, (28, 28, 1))
        im5 = np.expand_dims(im4, axis=0)


        pred_probab, pred_class = keras_predict(model, im5)

        curr = prediction(pred_class)

        cv2.putText(image_frame, curr, (700, 300), cv2.FONT_HERSHEY_COMPLEX, 4.0, (255,
255, 255), lineType=cv2.LINE_AA)




    # Display cropped image
        cv2.rectangle(image_frame, (300, 300), (600, 600), (255, 255, 00), 3)
        cv2.imshow("frame",image_frame)


    #cv2.imshow("Image4",resized_img)
        cv2.imshow("Image3",image_grayscale_blurred)

        if cv2.waitKey(25) & 0xFF == ord('q'):
                cv2.destroyAllWindows()
                break


if __name__ == '__main__':
    main()

cam_capture.release()
cv2.destroyAllWindows()
```

handhistogram.py

```
import cv2
import numpy as np
import pickle

def build_squares(img):
	x, y, w, h = 420, 140, 10, 10
	d = 10
	imgCrop = None
	crop = None
	for i in range(10):
		for j in range(5):
			if np.any(imgCrop == None):
				imgCrop = img[y:y+h, x:x+w]
			else:
				imgCrop = np.hstack((imgCrop, img[y:y+h, x:x+w]))
			#print(imgCrop.shape)
			cv2.rectangle(img, (x,y), (x+w, y+h), (0,255,0), 1)
			x+=w+d
		if np.any(crop == None):
			crop = imgCrop
		else:
			crop = np.vstack((crop, imgCrop))
		imgCrop = None
		x = 420
		y+=h+d
	return crop

def get_hand_hist():
	cam = cv2.VideoCapture(1)
	if cam.read()[0]==False:
		cam = cv2.VideoCapture(0)
	x, y, w, h = 300, 100, 300, 300
	flagPressedC, flagPressedS = False, False
	imgCrop = None
	while True:
		img = cam.read()[1]
		img = cv2.flip(img, 1)
		hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

		keypress = cv2.waitKey(1)
		if keypress == ord('c'):
			hsvCrop = cv2.cvtColor(imgCrop, cv2.COLOR_BGR2HSV)
			flagPressedC = True
```

```
                        hist = cv2.calcHist([hsvCrop], [0, 1], None, [180, 256], [0, 180, 0, 256])
                        cv2.normalize(hist, hist, 0, 255, cv2.NORM_MINMAX)
                elif keypress == ord('s'):
                        flagPressedS = True
                        break
                if flagPressedC:
                        dst = cv2.calcBackProject([hsv], [0, 1], hist, [0, 180, 0, 256], 1)
                        dst1 = dst.copy()
                        disc = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(10,10))
                        cv2.filter2D(dst,-1,disc,dst)
                        blur = cv2.GaussianBlur(dst, (11,11), 0)
                        blur = cv2.medianBlur(blur, 15)
                        ret,thresh =
cv2.threshold(blur,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
                        thresh = cv2.merge((thresh,thresh,thresh))
                        #cv2.imshow("res", res)
                        cv2.imshow("Thresh", thresh)
                if not flagPressedS:
                        imgCrop = build_squares(img)
                #cv2.rectangle(img, (x,y), (x+w, y+h), (0,255,0), 2)
                cv2.imshow("Set hand histogram", img)
        cam.release()
        cv2.destroyAllWindows()
        with open("hist", "wb") as f:
                pickle.dump(hist, f)


get_hand_hist()
```

Annexure II: Snapshots of the Output



References

[1] Characteristics and Practices of Sign Language Interpreters in Inclusive Education Programs, 1997

[2] The design of hand gestures for human–computer interaction: Lessons from sign language interpreters, 2014

[3] Sign language interpreting: Deconstructing the myth of neutrality, 2000

[4] An introduction to convolutional neural networks, 2015

[5] Understanding of a convolutional neural network, 2017

[6] Convolutional neural networks: an overview and application, 2018

[7] A brief introduction to OpenCV, 2012

[8] Learning OpenCV: Computer vision with the OpenCV library, 2018

[9] https://www.geeksforgeeks.org/introduction-convolution-neural-network/

[10] https://www.geeksforgeeks.org/convolutional-neural-network-cnn-in-machine-learning/