Sign Language Recognition

Data Science With Python Lab Project Report

Bachelor

in

Computer Science

By

Team Name

S190221

S190028

Rajiv Gandhi University Of Knowledge And Technologies

S.M. Puram , Srikakulam -532410

Andhra Pradesh, India

Abstract

The sign language recognization method provides us with a friendly way of communicating with the computer which is more familiar to human beings. It is a method of non-verbal communication. This method will help deaf and dumb people of having hearing or speech problems to communicate with ordinary people. Our project aims at taking the basic step in bridging the communication gap between normal people, deaf and dumb people using sign language. By using OpenCV we create a window that takes images as input from the user through the given path. We convert those images into grayscale images. Those grayscale images will be given as input to our model and our model will predict the actions which are displayed in the given image. Generally, the Hand gesture image shows one thumb and four fingers

Contents

Chapter 1

Introduction

1.1 Introduction To The Project

Sign language is a visual form of communication used by individuals with hearing impairments or as a means of communication between individuals who speak different languages. It involves gestures, hand movements, facial expressions, and body language to convey meaning.

Sign language recognition aims to develop technologies and systems that can understand and interpret sign language gestures, facilitating communication between deaf individuals and the hearing world. By leveraging computer vision and machine learning techniques, sign language recognition systems can enable real-time translation of sign language into spoken or written language.

The goal of sign language recognition is to accurately interpret and understand the gestures and movements made in sign language. This involves the detection and recognition of hand shapes, movements, and facial expressions, as well as the interpretation of the sequential nature of sign language gestures.

Sign language recognition systems have the potential to make a significant impact in various areas. It can enhance communication accessibility for deaf individuals by enabling them to interact more effectively with the hearing community. It can also be applied in educational settings, where it can assist in teaching sign language to non-native signers or aid in the automatic transcription of sign language videos.

Developing accurate and robust sign language recognition systems is a challenging task.

It requires a combination of computer vision techniques, such as object detection, pose estimation, and gesture tracking, along with machine learning algorithms to classify and interpret the detected gestures. Deep learning models, particularly Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), have shown promising results in this domain.

Training a sign language recognition model requires annotated sign language datasets, which consist of video recordings or images of sign language gestures performed by individuals. These datasets are used to train the models to recognize and interpret different sign language gestures.

Overall, sign language recognition has the potential to bridge the communication gap between deaf individuals and the hearing community, promoting inclusivity and accessibility. Advances in computer vision and machine learning techniques continue to drive improvements in sign language recognition systems, making them more accurate, efficient, and widely applicable in various domains.

1.2 Applications

Sign language recognition has numerous applications across different domains. Here are some notable applications of sign language recognition:

Communication Accessibility: Sign language recognition systems enable real-time translation of sign language gestures into spoken or written language, facilitating communication between deaf individuals and the hearing world. This application can enhance accessibility and inclusivity in various settings, including schools, workplaces, healthcare facilities, and public spaces.

Education and Training: Sign language recognition can be utilized in educational settings to teach sign language to non-native signers or to assist in the automatic transcription of sign language videos. It can provide valuable resources for learning sign language and support the integration of deaf students into mainstream educational environments.

Interpretation Services: Sign language recognition systems can aid in providing interpretation services for deaf individuals during conferences, public speeches, or other events. By automatically translating sign language gestures into spoken or written language, these

systems help bridge the communication gap between deaf individuals and hearing individuals in various professional and public settings.

Assistive Technologies: Sign language recognition can be integrated into assistive technologies such as smart gloves or wearable devices to provide real-time feedback and assistance for sign language users. These technologies can help individuals improve their sign language skills, correct their gestures, or provide guidance in everyday tasks.

Human-Computer Interaction: Sign language recognition can be employed in human-computer interaction systems to enable gesture-based control and interaction with digital devices. It allows users to navigate through interfaces, control applications, or interact with virtual environments using sign language gestures, providing a more intuitive and natural interaction experience.

Content Creation and Captioning: Sign language recognition systems can be used for automatically generating sign language videos or captions for online content, including videos, live streams, or webinars. This application helps make digital content more accessible and inclusive for deaf individuals.

Sign Language Research and Linguistics: Sign language recognition plays a crucial role in sign language research and linguistics. It can assist in the analysis of sign language grammar, syntax, and semantics, enabling researchers to study and document different sign languages more effectively.

As sign language recognition technology continues to advance, new applications and use cases are emerging, contributing to improved accessibility, communication, and inclusivity for deaf individuals

1.3 Motivation Towards The Project

The motivation towards a sign language recognition project stems from the desire to bridge the communication gap between deaf individuals and the hearing world. Here are some key motivations for working on a sign language recognition project:

Promoting Inclusivity and Accessibility: Sign language recognition technology enables effective communication between deaf individuals and the hearing community. By developing accurate and reliable systems, we can help create a more inclusive and accessible

society where everyone has equal opportunities for communication and interaction.

Improving Quality of Life: Sign language is a primary mode of communication for many deaf individuals. By developing robust sign language recognition systems, we can enhance their quality of life by facilitating smooth and efficient communication in various settings, including education, healthcare, employment, and social interactions.

Enabling Independence: Sign language recognition technology empowers deaf individuals to communicate and express themselves independently. It reduces reliance on interpreters or the need for a proficient knowledge of spoken or written languages, allowing deaf individuals to participate more actively in various aspects of life.

Supporting Education and Learning: Sign language recognition can play a vital role in educational settings by supporting the teaching and learning of sign language. It can assist educators in providing effective instruction to non-native signers and aid in the development of educational resources for deaf students.

Advancing Assistive Technologies: Sign language recognition contributes to the advancement of assistive technologies. By integrating sign language recognition into wearable devices or smart gloves, we can create innovative tools that assist deaf individuals in their daily lives, improve their sign language skills, and enhance their interaction with digital devices.

Promoting Research and Linguistics: Sign language recognition projects provide opportunities for research and linguistic studies. By developing and analyzing sign language recognition systems, we can gain insights into sign language structure, grammar, and linguistic variation, contributing to a better understanding of sign languages and their cultural significance.

Inspiring Social Change: By working on sign language recognition projects, we have the chance to be part of a broader social movement promoting inclusivity, equality, and accessibility for individuals with hearing impairments. Our efforts can help raise awareness about the importance of sign language recognition and encourage positive societal change.

Overall, the motivation behind a sign language recognition project lies in the desire to create a more inclusive and accessible society, improve the lives of deaf individuals, and foster effective communication and interaction between different communities.

1.4 Problem Statement

The problem statement for a sign language recognition project can be defined as follows:

"Develop a robust and accurate sign language recognition system that can effectively interpret and translate sign language gestures into spoken or written language. The system should be able to recognize a wide range of sign language gestures and provide accurate translations to facilitate communication between deaf individuals and the hearing world. The system should be efficient, user-friendly, and capable of handling variations in sign language gestures across different individuals. The goal is to bridge the communication gap and enhance accessibility, inclusivity, and independence for deaf individuals in various domains, including education, employment, healthcare, and social interactions."

Chapter 2

Approach To Your Project

2.1 Explain About Your Project

Project Explanation:

The project focuses on developing a sign language recognition system using a Convolutional Neural Network (CNN) model to interpret and translate sign language gestures into spoken or written language. The system aims to bridge the communication gap between deaf individuals and the hearing world, providing them with a means to communicate effectively and interact with the broader society.

The project involves several key components:

Dataset Acquisition: A sign language dataset is collected, which includes video recordings or images of individuals performing different sign language gestures. The dataset should cover a wide range of gestures and variations to ensure the model's robustness.

Data Preprocessing: The collected dataset is preprocessed to prepare it for training. This involves tasks such as resizing the images, converting them to grayscale, normalizing pixel values, and partitioning the dataset into training and testing sets.

Model Development: A CNN model architecture is designed and implemented for sign language recognition. The model consists of convolutional layers to extract features from the input images, pooling layers to reduce spatial dimensions, and dense layers for classification. Dropout layers may also be included to prevent overfitting. The model is trained using the training dataset and optimized using suitable optimization algorithms and loss functions.

Model Training: The model is trained on the prepared training dataset using the collected sign language gesture samples. During training, the model learns to recognize and classify different sign language gestures. Training involves iterations over multiple epochs, where the model adjusts its weights and biases to minimize the loss function.

Model Evaluation: After training, the model's performance is evaluated using the testing dataset. Metrics such as accuracy, precision, recall, and F1-score are calculated to assess the model's ability to correctly recognize and interpret sign language gestures. The model's performance is analyzed to identify any areas of improvement.

User Interface: A user-friendly interface is developed to facilitate easy interaction with the system. The interface may include features such as a video feed for gesture capture, a translated output display, and options for customization or language selection.

System Optimization and Refinement: The system is fine-tuned and optimized based on user feedback and performance evaluations. This involves refining the model architecture, incorporating additional features or techniques, and optimizing system performance to improve accuracy, speed, and usability.

The project aims to create an accurate, efficient, and user-friendly sign language recognition system that can empower deaf individuals by providing them with an effective means of communication. By developing this system, the project contributes to the broader goal of fostering inclusivity, accessibility, and independence for deaf individuals in various domains of life.

2.2 Data Set

The ASL MNIST dataset is a modified version of the original MNIST dataset, specifically designed for sign language recognition using the American Sign Language (ASL) alphabet. It is a useful resource for training models to recognize hand gestures representing letters of the alphabet in ASL.

Here are some key details about the ASL MNIST dataset:

Number of Classes: The ASL MNIST dataset consists of 26 classes representing the 26 letters of the English alphabet in ASL. Each class corresponds to a specific letter.

Image Format: Similar to the original MNIST dataset, the images in the ASL MNIST

dataset are grayscale and have a fixed size of 28x28 pixels.

Data Distribution: The ASL MNIST dataset is typically split into a training set and a testing set. The training set contains a large number of images, while the testing set is smaller and used for evaluating the model's performance.

Labeling: Each image in the ASL MNIST dataset is associated with a label that represents the correct letter it represents in ASL. The labels range from 0 to 25, corresponding to the letters A to Z.

The ASL MNIST dataset provides a more suitable starting point for sign language recognition compared to the original MNIST dataset. It captures the hand gestures specific to the ASL alphabet, allowing you to train a model that recognizes individual letters in ASL.

By using the ASL MNIST dataset, you can develop a sign language recognition model that focuses on recognizing and interpreting the gestures of the ASL alphabet accurately. However, if your project involves recognizing a broader range of sign language gestures beyond the ASL alphabet, you may need to consider additional datasets or expand the dataset to cover a wider variety of sign language gestures.

2.3 Prediction technique

A Convolutional Neural Network (CNN) is a type of deep learning model that is widely used in computer vision tasks, including image classification, object detection, and sign language recognition. CNNs are specifically designed to process grid-like data, such as images, by exploiting the spatial relationships between neighboring pixels.

Here's an explanation of the key components and operations in a CNN:

Convolutional Layers: Convolutional layers are the building blocks of a CNN. They consist of filters or kernels that slide over the input image in a sliding window manner, performing element-wise multiplication and accumulation operations (convolutions) to extract local features. Each filter captures specific patterns or features, such as edges, textures, or shapes. The convolutional layers learn these filters during the training process.

Pooling Layers: Pooling layers downsample the feature maps produced by the convolutional layers, reducing the spatial dimensions while retaining the most important features.

Common pooling operations include max pooling and average pooling, which extract the maximum or average value within each pooling region, respectively.

Activation Functions: Activation functions introduce non-linearities into the CNN model. They are applied element-wise to the output of each layer, allowing the model to capture complex relationships and make non-linear transformations. Common activation functions used in CNNs include ReLU (Rectified Linear Unit), sigmoid, and tanh.

Fully Connected Layers: Fully connected layers, also known as dense layers, are typically placed at the end of the CNN architecture. They connect every neuron in the previous layer to every neuron in the current layer, enabling the model to learn high-level representations and make final predictions. The number of neurons in the last dense layer is equal to the number of classes in the classification task.

Training and Backpropagation: CNNs are trained using labeled training data through a process called backpropagation. During training, the model adjusts its parameters (weights and biases) to minimize a defined loss function, such as categorical cross-entropy, using optimization algorithms like stochastic gradient descent (SGD) or Adam. The gradients for parameter updates are computed through backpropagation, which calculates the error derivatives layer-by-layer.

Regularization Techniques: To prevent overfitting and improve generalization, various regularization techniques can be applied to CNNs. Dropout is a commonly used technique where randomly selected neurons are temporarily dropped out during training, reducing interdependencies and promoting robustness. Other techniques include L1 or L2 regularization, which add penalty terms to the loss function to limit the magnitude of weights.

Transfer Learning: Transfer learning is a technique where pre-trained CNN models, such as VGGNet, ResNet, or Inception, are utilized as a starting point. These models are trained on large-scale datasets, like ImageNet, and can capture general features that are transferrable to other related tasks. By leveraging pre-trained models, transfer learning can save training time and improve performance, especially when the available dataset is limited.

The architecture and configuration of a CNN can vary depending on the specific task and dataset. Designing an effective CNN involves careful consideration of the number and size of filters, the depth of the network, the choice of activation functions, and other

architectural choices to achieve the desired performance.

2.4 Data Visualization

2.4.1 Similarity

The following Python code snippet demonstrates how to visualize the similarity between two images based on their pixel values:

```
import random
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
df = pd.read_csv('/home/rgukt/Desktop/S190028/E2-SEM2/DSP/DSP_LAB/
    dataset_python_sign/sign_mnist_train.csv')
filtered_df = df.loc[df['label'] == 2]
filtered_rows = filtered_df.values.tolist()
num_random_rows = 3
# Generate random rows
random_rows = random.sample(filtered_rows, num_random_rows)
# Store the random rows in separate variables
image1 = random_rows[0]
image2 = random_rows[1]
image11 = np.array(image1)
image22 = np.array(image2)
# Plot the pixel values of the two images using a scatter plot
plt.scatter(range(len(image11)), image11, color='red', label='Image␣1')
plt.scatter(range(len(image22)), image22, color='blue', label='Image␣2')
plt.xlabel('Pixel␣Index')
plt.ylabel('Pixel␣Value')
plt.title('Similarity␣between␣Image␣1␣and␣Image␣2')
plt.legend()
plt.show()
```

Listing 2.1:

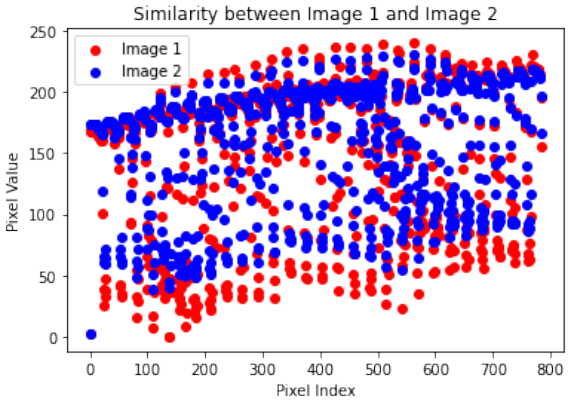


Figure 2.1: Visualization Example

The scatter plot visually represents the pixel values of two images, labeled as Image 1 and Image 2. The x-axis represents the index of the pixel, while the y-axis represents the corresponding pixel value. The red points correspond to the pixel values of Image 1, and the blue points represent the pixel values of Image 2. By comparing the distribution of pixel values between the two images, we can gain insights into their visual resemblance or dissimilarity.

2.4.2 Image Histogram

The following Python code snippet demonstrates how to plot the histograms of pixel intensities for two images:

```
import random
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

df = pd.read_csv('/home/rgukt/Desktop/S190028/E2-SEM2/DSP/DSP_LAB/
    dataset_python_sign/sign_mnist_train.csv')
filtered_df = df.loc[df['label'] == 2]
filtered_rows = filtered_df.values.tolist()
num_random_rows = 2
random_rows = random.sample(filtered_rows, num_random_rows)
image1 = random_rows[0]
image2 = random_rows[1]
```

```
image1_pixels = np.array(image1)
image2_pixels = np.array(image2)

# Calculate the histograms
histogram1, bins1 = np.histogram(image1_pixels, bins=256, range=[0, 256])
histogram2, bins2 = np.histogram(image2_pixels, bins=256, range=[0, 256])

# Plot the histograms
plt.figure(figsize=(10, 5))
plt.plot(bins1[:-1], histogram1, color='red', label='Image 1')
plt.plot(bins2[:-1], histogram2, color='blue', label='Image 2')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.title('Histogram of Pixel Intensity')
plt.legend()
plt.show()
```

Listing 2.2:

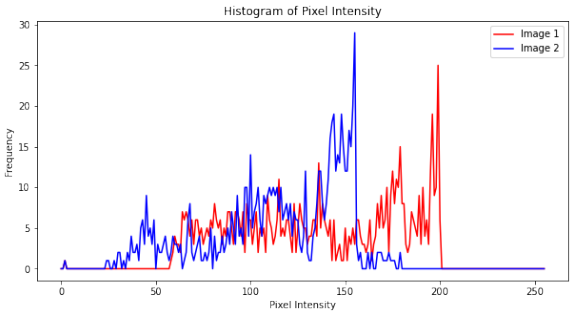


Figure 2.2: Histogram of Pixel Intensity

The histograms illustrate the distribution of pixel intensities for Image 1 and Image 2. The x-axis represents the pixel intensity values, ranging from 0 to 256, while the y-axis represents the frequency or count of pixels with each intensity value. By comparing the histograms, we can observe the differences in the distribution patterns between the two images. Peaks or variations in the histogram indicate the presence of specific pixel intensity values that may contribute to distinguishing features or characteristics in the images. This visualization allows us to gain insights into the pixel intensity distribution and identify

potential dissimilarities or similarities between Image 1 and Image 2 based on their pixel values.

2.4.3 Mean Squared Error (MSE) Loss

The following Python code snippet demonstrates how to calculate the Mean Squared Error (MSE) loss between two images and generate a loss graph based on the MSE value:

```
import random
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
def calculate_mse(image1, image2):
    # Calculate the Mean Squared Error (MSE) loss between the two images
    mse_loss = np.mean((image1 - image2) ** 2)
    return mse_loss

df = pd.read_csv('/home/rgukt/Desktop/S190028/E2-SEM2/DSP/DSP_LAB/
    dataset_python_sign/sign_mnist_train.csv')
filtered_df = df.loc[df['label'] == 2]
filtered_rows = filtered_df.values.tolist()
num_random_rows = 2
random_rows = random.sample(filtered_rows, num_random_rows)
image1 = random_rows[0]
image2 = random_rows[1]
image1_pixels = np.array(image1)
image2_pixels = np.array(image2)

mse_loss = calculate_mse(image1_pixels, image2_pixels)

# Print the MSE loss
print("MSE Loss:", mse_loss)

# Generate the loss graph
```

```
x = np.arange(0, 10, 0.1)
y = np.sin(x) + mse_loss

# Plot the graph
plt.plot(x, y, color='blue')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Loss Graph')
plt.show()
```

Listing 2.3:

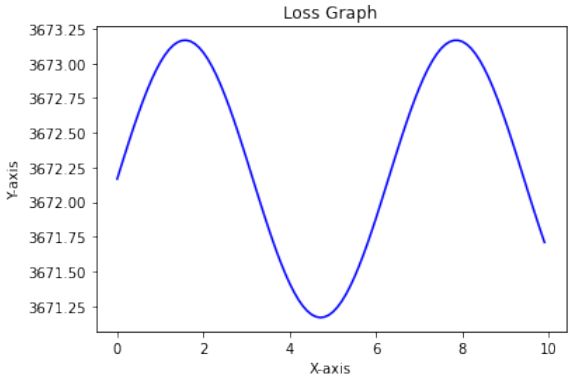


Figure 2.3: Visualization Example

The mse loss value is used to generate the loss graph. In this example, a simple graph is generated by creating an array of x-values (x) and calculating the corresponding y-values (y) based on the MSE loss.

2.4.4 Comparison of Image Resolutions

The following Python code snippet demonstrates how to compare the resolutions of two images based on their pixel values and generate a bar graph:

```
import random
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

```

```
# Load the dataset and filter the rows
df = pd.read_csv('/home/rgukt/Desktop/S190028/E2-SEM2/DSP/DSP_LAB/
    dataset_python_sign/sign_mnist_train.csv')
filtered_df = df.loc[df['label'] == 2]
filtered_rows = filtered_df.values.tolist()

# Specify the number of random rows you want to generate
num_random_rows = 3

# Generate random rows
random_rows = random.sample(filtered_rows, num_random_rows)

# Extract the pixel values of the images
image1 = random_rows[0]
image2 = random_rows[1]
image1_pixels = np.array(image1)
image2_pixels = np.array(image2)

# Calculate the resolutions
resolution1 = np.sqrt(image1_pixels.shape[0])
resolution2 = np.sqrt(image2_pixels.shape[0])

# Create labels and values for the bar graph
labels = ['Image␣1', 'Image␣2']
values = [resolution1, resolution2]

# Plot the bar graph
plt.bar(labels, values)
plt.xlabel('Image')
plt.ylabel('Resolution')
plt.title('Comparison␣of␣Image␣Resolutions')
plt.show()
```

Listing 2.4:

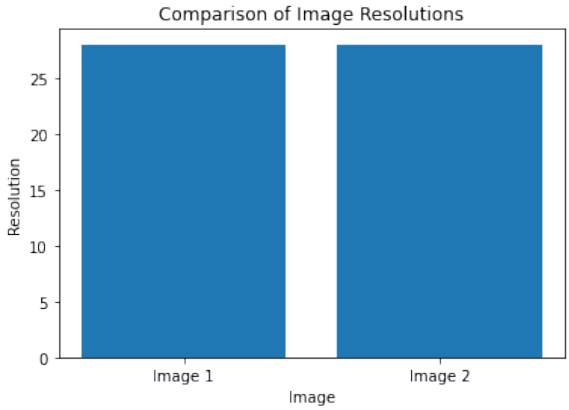


Figure 2.4: Comparison of Image Resolutions

Chapter 3

Code

3.1 Explain Your Code With Outputs

3.1.1 Importing necessary modules

```
1 from keras.models import Sequential
2 from keras.layers import Conv2D,MaxPooling2D, Dense,Flatten, Dropout
3 import matplotlib.pyplot as plt
4 from keras.utils import np_utils
5 from keras.optimizers import SGD
6 import pandas as pd
7 import numpy as np
8 import cv2
```

Listing 3.1:

Let's go through the explanation of each line in the given code snippet:

1.from keras.models import Sequential: This line imports the Sequential class from the keras.models module. Sequential is a linear stack of neural network layers that allows you to build and train models layer by layer.

2.from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout: This line imports several layer classes from the keras.layers module. These layer classes are building

blocks for constructing the CNN model:

3.Conv2D represents a 2D convolutional layer used for feature extraction. MaxPooling2D represents a 2D max-pooling layer used for downsampling the feature maps. Dense represents a fully connected layer used for classification or regression tasks. Flatten represents a layer that flattens the input into a 1D array, usually used before the fully connected layers. Dropout represents a regularization technique where random neurons are dropped out during training to prevent overfitting. import matplotlib.pyplot as plt: This line imports the matplotlib.pyplot module, which provides functions for creating and manipulating plots. It allows you to visualize data, including training/validation loss curves and other graphs.

4.from keras.utils import np-utils: This line imports the np-utils module from the keras.utils package. The np-utils module provides utility functions for manipulating data, such as one-hot encoding labels (to-categorical function).

5.from keras.optimizers import SGD: This line imports the SGD optimizer from the keras.optimizers module. SGD stands for Stochastic Gradient Descent, which is a widely used optimization algorithm for training neural networks.

6.import pandas as pd: This line imports the pandas library, which provides data manipulation and analysis tools. It is not directly used in the given code snippet but can be useful for working with tabular data or data preprocessing tasks.

7.import numpy as np: This line imports the numpy library, a fundamental package for scientific computing with Python. It provides support for mathematical operations on arrays and matrices, which are widely used in deep learning.

8.import cv2: This line imports the cv2 module, which is the OpenCV library for computer vision tasks. It is a popular library for image and video processing, often used for loading, manipulating, and preprocessing image data.

These import statements ensure that the required libraries and modules are available for building the CNN model, performing visualizations, handling data, and working with images.

3.1.2 Loading dataset

```
train=pd.read_csv("/content/drive/MyDrive/sign_mnist_train/sign_mnist_train.csv
    ")
test=pd.read_csv("/content/drive/MyDrive/sign_mnist_test/sign_mnist_test.csv")
```

Listing 3.2:

Let's break down the code and explain each part:

pd: This refers to the pandas library that has been imported using import pandas as pd. Pandas is a powerful data manipulation and analysis library in Python.

1.read-csv: This is a pandas function used to read a CSV file and load it into a pandas DataFrame.

2.train: This is the variable name assigned to the DataFrame that will store the training dataset loaded from the CSV file.

3.test: This is the variable name assigned to the DataFrame that will store the test dataset loaded from the CSV file.

By executing this code, the CSV files will be read using pandas' read-csv function, and the contents will be loaded into the train and test DataFrames, respectively. These DataFrames can then be used for further data analysis, preprocessing, or model training in the subsequent code.

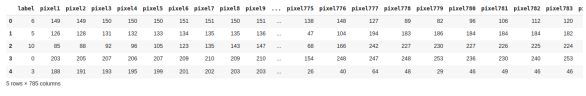
3.1.3 Some basic operations on dataset

```
display(train.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 27455 entries, 0 to 27454
Columns: 785 entries, label to pixel784
dtypes: int64(785)
memory usage: 164.4 MB
None

display(test.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7172 entries, 0 to 7171
Columns: 785 entries, label to pixel784
dtypes: int64(785)
memory usage: 43.0 MB
None
```

Listing 3.3:

```
display(test.head())
```

Listing 3.4:



	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782	pixel783	p
0	6	149	149	150	150	150	151	151	150	151	...	138	148	127	89	82	96	106	112	120	
1	5	126	128	131	132	133	134	135	135	136	...	47	104	194	183	186	184	184	184	182	
2	10	85	88	92	96	105	123	135	143	147	...	68	166	242	227	230	227	226	225	224	
3	0	203	205	207	206	207	209	210	209	210	...	154	248	247	248	253	236	230	240	253	
4	3	188	191	193	195	199	201	202	203	203	...	26	40	64	48	29	46	49	46	46	

5 rows × 785 columns

display(test.head()) is used to print the first few rows of the test DataFrame, providing an overview of the data contained in the DataFrame. It helps to verify if the data is loaded correctly and gives a glimpse of the structure and contents of the DataFrame.

3.1.4 Splitting the Dataset

```
train_l=train['label'].values
```

```
test_l=test['label'].values
train_d=train.drop(['label'],axis=1)
test_d=test.drop(['label'],axis=1)
```

Listing 3.5:

1.train-l = train['label'].values: This line extracts the 'label' column from the 'train' DataFrame and assigns it to the variable 'train-l'. The 'label' column typically contains the target or class labels for the corresponding data samples. The '.values' attribute converts the extracted column into a NumPy array for further processing.

2.test-l = test['label'].values: Similar to the previous line, this line extracts the 'label' column from the 'test' DataFrame and assigns it to the variable 'test-l'. It retrieves the target or class labels for the test dataset.

3.train-d = train.drop(['label'], axis=1): This line creates a new DataFrame 'train-d' by dropping the 'label' column from the 'train' DataFrame using the drop() function. The 'axis=1' argument specifies that the column is to be dropped. The resulting 'train-d' DataFrame contains only the input features or predictors, excluding the target labels.

4.test-d = test.drop(['label'], axis=1): Similarly, this line creates a new DataFrame 'test-d' by dropping the 'label' column from the 'test' DataFrame. It retains only the input features or predictors for the test dataset.

The purpose of these lines of code is to separate the input features from the target labels in both the training and test datasets. The 'train-d' and 'test-d' DataFrames contain the input features, while 'train-l' and 'test-l' store the corresponding target labels.

By splitting the data in this way, you can use 'train-d' and 'train-l' for model training, and 'test-d' and 'test-l' for evaluating the trained model's performance on unseen data.

3.1.5 Preprocessing

```
train_d=np.array(train_d.iloc[:,:])
train_d=np.array([np.reshape(i, (28,28)) for i in train_d])
test_d=np.array(test_d.iloc[:,:])
test_d=np.array([np.reshape(i, (28,28)) for i in test_d])
num_classes = 26
train_l = np.array(train_l).reshape(-1)
test_l = np.array(test_l).reshape(-1)
train_l=np.array(train_l).astype(int)
train_l = np.eye(num_classes)[train_l]
test_l = np.eye(num_classes)[test_l]
train_d = train_d.reshape((27455, 28, 28, 1))
test_d = test_d.reshape((7172, 28, 28, 1))
```

Listing 3.6:

1.train-d = np.array(train-d.iloc[:,:]): This line converts the 'train-d' DataFrame into a NumPy array, including all rows and columns. The .iloc[:, :] syntax selects all rows and columns from the DataFrame.

2.train-d = np.array([np.reshape(i, (28,28)) for i in train-d]): Here, each sample in 'train-d' is reshaped from a flattened 1D array (784 elements) to a 2D array with dimensions (28, 28). This is done using a list comprehension, where each element 'i' in 'train-d' is reshaped using np.reshape().

3.test-d = np.array(test-d.iloc[:,:]): Similar to step 1, this line converts the 'test-d' DataFrame into a NumPy array.

4.test-d = np.array([np.reshape(i, (28,28)) for i in test-d]): Similar to step 2, this line reshapes each sample in 'test-d' to a 2D array with dimensions (28, 28).

5.num-classes = 26: The variable 'num-classes' is assigned a value of 26, which represents the total number of classes or labels in the dataset.

6.train-l = np.array(train-l).reshape(-1): This line reshapes the 'train-l' array to a 1D array by using the .reshape() method. The '-1' argument infers the appropriate number of elements based on the existing shape.

7.test-l = np.array(test-l).reshape(-1): Similar to step 6, this line reshapes the 'test-l' array to a 1D array.

8.train-l = np.array(train-l).astype(int): This line converts the data type of 'train-l' to integer using the astype() method. It ensures that the labels are stored as integers for further processing.

9.train-l = np.eye(num-classes)[train-l]: Here, one-hot encoding is applied to the 'train-l' array using np.eye() to create a one-hot encoded representation of the labels. Each label is converted to a binary vector with a value of 1 in the corresponding class index and 0 in all other indices.

10.test-l = np.eye(num-classes)[test-l]: Similar to step 9, this line applies one-hot encoding to the 'test-l' array.

11.train-d = train-d.reshape((27455, 28, 28, 1)): This line reshapes the 'train-d' array to have an additional dimension of size 1, representing the number of channels. In this case, it indicates that the input images are grayscale.

12.test-d = test-d.reshape((7172, 28, 28, 1)): Similar to step 11, this line reshapes the 'test-d' array with the additional dimension for grayscale images.

These preprocessing steps prepare the data for training a CNN model. The input images are reshaped to have the desired dimensions, one-hot encoding is applied to the labels, and the data is organized in the appropriate format for CNN input, including the additional channel dimension for grayscale images.

3.1.6 Building CNN Model

```
classifier = Sequential()
classifier.add(Conv2D(filters=8, kernel_size=(3,3),strides=(1,1),padding='same',
    input_shape=(28,28,1),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'])
```

Listing 3.7:

1.classifier = Sequential(): This line initializes a sequential model, which is a linear stack of layers. The model will be built by adding layers one by one.

2.classifier.add(Conv2D(filters=8, kernel-size=(3,3), strides=(1,1), padding='same', input-shape=(28,28,1), activation='relu', data-format='channels-ast')): This line adds a 2D convolutional layer to the model. The filters parameter specifies the number of filters in the layer, the kernel-size parameter defines the size of the convolutional kernel, and the strides parameter determines the stride length. The padding parameter is set to 'same', which means the input will be padded with zeros to preserve the spatial dimensions. The input-shape parameter specifies the shape of the input data, and activation='relu' sets the activation function to Rectified Linear Unit (ReLU). The data-format parameter is set to 'channels-last' to indicate that the input shape has the channels dimension last.

3.classifier.add(MaxPooling2D(pool-size=(2,2))): This line adds a max-pooling layer to

the model. The pool-size parameter specifies the size of the pooling window.

4.classifier.add(Conv2D(filters=16, kernel-size=(3,3), strides=(1,1), padding='same', activation='relu')): This line adds another 2D convolutional layer with 16 filters to the model.

5.classifier.add(Dropout(0.5)): This line adds a dropout layer to the model. The 0.5 argument represents the dropout rate, which indicates the fraction of input units to drop during training to prevent overfitting.

6.classifier.add(MaxPooling2D(pool-size=(4,4))): This line adds another max-pooling layer to the model.

7.classifier.add(Dense(128, activation='relu')): This line adds a fully connected (dense) layer with 128 units to the model. The activation='relu' sets the activation function to ReLU.

8.classifier.add(Flatten()): This line adds a flatten layer to the model. It reshapes the output from the previous layer into a 1D array.

9.classifier.add(Dense(26, activation='softmax')): This line adds the final fully connected layer with 26 units (equal to the number of classes in the dataset) and applies the softmax activation function. This layer produces the output probabilities for each class.

10.classifier.compile(optimizer='SGD', loss='categorical-crossentropy', metrics=['accuracy']): This line compiles the model. The optimizer parameter is set to 'SGD', indicating the Stochastic Gradient Descent optimization algorithm. The loss parameter is set to 'categorical-crossentropy', which is the loss function suitable for multi-class classification problems. The metrics parameter is set to 'accuracy' to evaluate the model's performance.

3.1.7 Training The Model

```
classifier.fit(train_d,train_l,epochs=60,batch_size=100)



Epoch 1/60
275/275 [==============================] - 17s 60ms/step - loss: 3.5126 -
    accuracy: 0.0564
Epoch 2/60
275/275 [==============================] - 14s 51ms/step - loss: 3.1727 -
    accuracy: 0.0778
Epoch 3/60
275/275 [==============================] - 14s 51ms/step - loss: 3.2024 -
    accuracy: 0.0653
Epoch 4/60
275/275 [==============================] - 14s 51ms/step - loss: 3.1597 -
    accuracy: 0.0747
Epoch 5/60
275/275 [==============================] - 14s 52ms/step - loss: 3.2004 -
    accuracy: 0.0572
Epoch 6/60
275/275 [==============================] - 14s 51ms/step - loss: 3.1779 -
    accuracy: 0.0627
Epoch 7/60
275/275 [==============================] - 14s 52ms/step - loss: 3.0830 -
    accuracy: 0.0898
Epoch 8/60
275/275 [==============================] - 14s 51ms/step - loss: 3.1051 -
    accuracy: 0.0808
Epoch 9/60
275/275 [==============================] - 14s 52ms/step - loss: 3.0060 -
    accuracy: 0.1137
Epoch 10/60
```

```
275/275 [==============================] - 14s 52ms/step - loss: 2.3670 -
    accuracy: 0.2961
Epoch 11/60
275/275 [==============================] - 14s 52ms/step - loss: 1.3610 -
    accuracy: 0.5796
Epoch 12/60
275/275 [==============================] - 16s 59ms/step - loss: 0.9226 -
    accuracy: 0.7014
Epoch 13/60
275/275 [==============================] - 14s 50ms/step - loss: 0.7267 -
    accuracy: 0.7646
Epoch 14/60
275/275 [==============================] - 14s 50ms/step - loss: 0.5911 -
    accuracy: 0.8007
Epoch 15/60
275/275 [==============================] - 14s 50ms/step - loss: 0.5040 -
    accuracy: 0.8302
Epoch 16/60
275/275 [==============================] - 14s 51ms/step - loss: 0.4160 -
    accuracy: 0.8591
Epoch 17/60
275/275 [==============================] - 14s 52ms/step - loss: 0.3876 -
    accuracy: 0.8711
Epoch 18/60
275/275 [==============================] - 14s 52ms/step - loss: 0.3693 -
    accuracy: 0.8732
Epoch 19/60
275/275 [==============================] - 14s 52ms/step - loss: 0.2649 -
    accuracy: 0.9074
Epoch 20/60
275/275 [==============================] - 14s 52ms/step - loss: 0.2547 -
    accuracy: 0.9140
Epoch 21/60
```

```
275/275 [==============================] - 14s 52ms/step - loss: 0.2277 -
    accuracy: 0.9238
Epoch 22/60
275/275 [==============================] - 14s 52ms/step - loss: 0.2724 -
    accuracy: 0.9102
Epoch 23/60
275/275 [==============================] - 15s 53ms/step - loss: 0.1776 -
    accuracy: 0.9403
Epoch 24/60
275/275 [==============================] - 16s 59ms/step - loss: 0.1699 -
    accuracy: 0.9410
Epoch 25/60
275/275 [==============================] - 14s 52ms/step - loss: 0.1665 -
    accuracy: 0.9433
Epoch 26/60
275/275 [==============================] - 14s 52ms/step - loss: 0.1547 -
    accuracy: 0.9474
Epoch 27/60
275/275 [==============================] - 14s 52ms/step - loss: 0.1469 -
    accuracy: 0.9493
Epoch 28/60
275/275 [==============================] - 14s 52ms/step - loss: 0.1395 -
    accuracy: 0.9534
Epoch 29/60
275/275 [==============================] - 14s 52ms/step - loss: 0.1271 -
    accuracy: 0.9565
Epoch 30/60
275/275 [==============================] - 14s 52ms/step - loss: 0.1288 -
    accuracy: 0.9581
Epoch 31/60
275/275 [==============================] - 14s 52ms/step - loss: 0.1028 -
    accuracy: 0.9642
Epoch 32/60
```

```
275/275 [==============================] - 14s 52ms/step - loss: 0.0953 -
    accuracy: 0.9675
Epoch 33/60
275/275 [==============================] - 14s 52ms/step - loss: 0.1073 -
    accuracy: 0.9639
Epoch 34/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0980 -
    accuracy: 0.9677
Epoch 35/60
275/275 [==============================] - 17s 60ms/step - loss: 0.0938 -
    accuracy: 0.9683
Epoch 36/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0877 -
    accuracy: 0.9702
Epoch 37/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0922 -
    accuracy: 0.9707
Epoch 38/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0855 -
    accuracy: 0.9717
Epoch 39/60
275/275 [==============================] - 14s 50ms/step - loss: 0.0876 -
    accuracy: 0.9696
Epoch 40/60
275/275 [==============================] - 14s 50ms/step - loss: 0.1609 -
    accuracy: 0.9545
Epoch 41/60
275/275 [==============================] - 14s 50ms/step - loss: 0.0831 -
    accuracy: 0.9733
Epoch 42/60
275/275 [==============================] - 14s 50ms/step - loss: 0.0679 -
    accuracy: 0.9766
Epoch 43/60
```

```
275/275 [==============================] - 14s 51ms/step - loss: 0.0731 -
    accuracy: 0.9760
Epoch 44/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0758 -
    accuracy: 0.9747
Epoch 45/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0603 -
    accuracy: 0.9796
Epoch 46/60
275/275 [==============================] - 17s 60ms/step - loss: 0.0864 -
    accuracy: 0.9726
Epoch 47/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0672 -
    accuracy: 0.9775
Epoch 48/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0642 -
    accuracy: 0.9788
Epoch 49/60
275/275 [==============================] - 14s 52ms/step - loss: 0.1225 -
    accuracy: 0.9635
Epoch 50/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0583 -
    accuracy: 0.9801
Epoch 51/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0691 -
    accuracy: 0.9778
Epoch 52/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0546 -
    accuracy: 0.9822
Epoch 53/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0615 -
    accuracy: 0.9799
Epoch 54/60
```

```
275/275 [==============================] - 14s 52ms/step - loss: 0.0624 -
    accuracy: 0.9791
Epoch 55/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0538 -
    accuracy: 0.9814
Epoch 56/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0511 -
    accuracy: 0.9830
Epoch 57/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0555 -
    accuracy: 0.9815
Epoch 58/60
275/275 [==============================] - 17s 60ms/step - loss: 0.0488 -
    accuracy: 0.9846
Epoch 59/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0544 -
    accuracy: 0.9823
Epoch 60/60
275/275 [==============================] - 14s 52ms/step - loss: 0.0524 -
    accuracy: 0.9832
<keras.callbacks.History at 0x7f990c4c1720>
```

Listing 3.8:

1.train-d: This represents the training data, which should be a NumPy array or a TensorFlow tensor. It contains the input images that have been preprocessed and reshaped.

2.train-l: This corresponds to the training labels, which should also be a NumPy array or TensorFlow tensor. It contains the one-hot encoded labels for the training data.

3.epochs=60: This parameter specifies the number of epochs for training. An epoch refers to one complete pass through the entire training dataset. In this case, the model will be trained for a total of 60 epochs.

4.batch-size=100: This parameter sets the batch size for training. It determines the number of training samples to be processed before updating the model's weights. In this case, each batch will contain 100 samples.

By calling classifier.fit(train-d, train-l, epochs=60, batch-size=100), the training process will start. During each epoch, the model will iterate through the training data in batches, compute the loss using the specified loss function, and update the model's weights using the chosen optimizer. This process is repeated for the specified number of epochs, gradually improving the model's performance on the training data.

Make sure that the training data (train-d) and labels (train-l) are correctly formatted and preprocessed according to the model's input requirements. Also, ensure that the model (classifier) has been compiled with the appropriate optimizer, loss function, and evaluation metrics before calling fit().

3.1.8 Predicting The Accuracy

```
1 acc=classifier.evaluate(x=test_d,y=test_l,batch_size=32)
2 print("ACCURACY=",acc[1])
3
4 225/225 [==============================] - 2s 7ms/step - loss: 0.2526 - accuracy
      : 0.9304
5 ACCURACY= 0.9304238557815552
```

Listing 3.9:

1.x=test-d: This represents the test data, which should be a NumPy array or a TensorFlow tensor. It contains the preprocessed and reshaped input images for the test dataset.

2.y=test-l: This corresponds to the test labels, which should also be a NumPy array or TensorFlow tensor. It contains the one-hot encoded labels for the test dataset.

3.batch-size=32: This parameter sets the batch size for evaluating the model on the test data. It determines the number of test samples to be processed at once.

By calling classifier.evaluate(x=test-d, y=test-l, batch-size=32), the model will predict the labels for the test data and compare them with the true labels. It will then compute the evaluation metrics, including the accuracy, based on the predictions and true labels. The returned acc object is a list containing the evaluation metrics calculated by the model. In this case, acc[1] represents the accuracy value. Finally, print("ACCURACY=", acc[1]) is used to display the accuracy of the model on the test data. The accuracy value is extracted from the acc list and printed to the console.

3.1.9 Architecture of Model

```
classifier.summary()



classifier.summary()
Model: "sequential_1"
_________________________________________________________________
 Layer (type) Output Shape Param #
=================================================================
 conv2d_2 (Conv2D) (None, 28, 28, 8) 80

 max_pooling2d_2 (MaxPooling (None, 14, 14, 8) 0
 2D)

 conv2d_3 (Conv2D) (None, 14, 14, 16) 1168

 dropout_1 (Dropout) (None, 14, 14, 16) 0

```

```
 max_pooling2d_3 (MaxPooling (None, 3, 3, 16) 0
 2D)

 dense_2 (Dense) (None, 3, 3, 128) 2176

 flatten_1 (Flatten) (None, 1152) 0

 dense_3 (Dense) (None, 26) 29978

=================================================================
Total params: 33,402
Trainable params: 33,402
Non-trainable params: 0
_________________________________________________________________

import tensorflow as tf
classifier.save('/content/drive/MyDrive/model.h5')
```

Listing 3.10:

The classifier.summary() function is useful for gaining a quick overview of the model architecture and the number of parameters involved. It helps in understanding the complexity and size of the model, and can assist in identifying potential issues such as overly large models or layers with a high number of parameters.

By calling classifier.save('/content/drive/MyDrive/model.h5'), the trained model is saved to the specified file. This allows you to store the model's architecture, weights, optimizer configuration, and other necessary information. The saved model can be loaded and used later for inference or further training.

3.2 Testing The Model

3.2.1 Downloading The Model

```
import tensorflow as tf
model = tf.keras.models.load_model('/content/drive/MyDrive/model.h5')
```

Listing 3.11:

By calling tf.keras.models.load-model('/content/drive/MyDrive/model.h5'), the model stored in the HDF5 file is loaded and assigned to the variable model. This allows you to access and use the loaded model for inference or further training.

3.2.2 Input

```
import cv2
import numpy as np
```

Listing 3.12:

Importing necessary modules.

```
from IPython.display import display, Javascript
from google.colab.output import eval_js
from base64 import b64decode

def take_photo(filename='photo.jpg', quality=0.8):
  js = Javascript('''
    async function takePhoto(quality) {
      const div = document.createElement('div');
      const capture = document.createElement('button');
      capture.textContent = 'Capture';
      div.appendChild(capture);
```

```
12
13      const video = document.createElement('video');
14      video.style.display = 'block';
15      const stream = await navigator.mediaDevices.getUserMedia({video: true});
16
17      document.body.appendChild(div);
18      div.appendChild(video);
19      video.srcObject = stream;
20      await video.play();
21
22      // Resize the output to fit the video element.
23      google.colab.output.setIframeHeight(document.documentElement.scrollHeight,
            true);
24
25      // Wait for Capture to be clicked.
26      await new Promise((resolve) => capture.onclick = resolve);
27
28      const canvas = document.createElement('canvas');
29      canvas.width = video.videoWidth;
30      canvas.height = video.videoHeight;
31      canvas.getContext('2d').drawImage(video, 0, 0);
32      stream.getVideoTracks()[0].stop();
33      div.remove();
34      return canvas.toDataURL('image/jpeg', quality);
35    }
36    ''')
37  display(js)
38  data = eval_js('takePhoto({})'.format(quality))
39  binary = b64decode(data.split(',')[1])
40  with open(filename, 'wb') as f:
41    f.write(binary)
42  return filename
```

Listing 3.13:

1.from IPython.display import display, Javascript: This line imports the necessary functions from IPython.display module to display JavaScript code and execute it within the notebook.

2.from google.colab.output import eval-js: This line imports the eval-js function from the google.colab.output module. It allows the execution of JavaScript code and retrieval of the result in Python.

3.from base64 import b64decode: This line imports the b64decode function from the base64 module. It is used to decode the base64-encoded image data returned by the JavaScript code.

4.The take-photo function performs the following steps:

It defines a JavaScript code block using the Javascript object. This code block sets up the webcam video stream, displays it, and captures a photo when the "Capture" button is clicked.
The JavaScript code is displayed using the display function. This allows the JavaScript code to be executed in the notebook environment.
The eval-js function is called to execute the JavaScript code and retrieve the captured photo as a base64-encoded string.
The base64-encoded data is decoded using b64decode and stored in the binary variable.
The decoded binary data is saved to a file with the specified filename using the with open() statement.

Finally, the filename is returned, indicating the path and name of the saved photo.

This function enables you to capture a photo using the webcam in Google Colab and save it to a specified file. You can call this function to capture an image and use it as input for further processing or analysis in your notebook.

```
from IPython.display import Image
try:
  filename = take_photo()
  print('Saved to {}'.format(filename))

  # Show the image which was just taken.
  display(Image(filename))
except Exception as err:
  # Errors will be thrown if the user does not have a webcam or if they do not
  # grant the page permission to access it.
  print(str(err))
```

Listing 3.14:

from IPython.display import Image: This line imports the Image class from the IPython.display module. It allows displaying an image in the notebook.

The code is wrapped in a try-except block to handle potential errors. filename = take-photo(): This line calls the take-photo function to capture a photo using the webcam. The returned value is assigned to the filename variable.

print('Saved to '.format(filename)): This line prints a message indicating the path and name of the saved photo file. display(Image(filename)): This line displays the captured image in the notebook using the Image class. The filename variable is passed as an argument to specify the image file to be displayed.

Inside the except block, any exceptions that occur during the execution of the code will be caught and handled. If the user does not have a webcam or has not granted permission to

access it, an error message will be printed. This code allows you to capture a photo using the webcam and display the captured image in the notebook. If no errors occur, the path and name of the saved photo file will also be printed. If any exceptions occur, such as a lack of webcam or permission issues, the corresponding error message will be displayed. You can use this code to interactively capture images and incorporate them into your notebook for further analysis or visualization.

```
image = cv2.imread('photo.jpg')
\\This line uses the OpenCV function cv2.imread to read the image file 'photo.
    jpg' and store it in the image variable as a NumPy array.
resized_image = cv2.resize(image, (28, 28))
\\ This line resizes the image to a desired size of (28, 28) pixels using the
    cv2.resize function. The resulting resized image is stored in the
    resized_image variable.
gray_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2GRAY)
\\This line converts the resized_image from the BGR color space to grayscale
    using the cv2.cvtColor function. The resulting grayscale image is stored in
    the gray_image variable.
normalized_image = gray_image / 255.0
\\This line normalizes the pixel values of the gray_image by dividing them by
    255.0, which scales the pixel values to the range of 0 to 1. The normalized
    image is stored in the normalized_image variable.
normalized_image = normalized_image.reshape(-1, 28, 28, 1)
\\This line reshapes the normalized_image to match the expected input shape of
    the model. It adds an additional dimension of size 1 at the end to represent
     the number of channels. The resulting reshaped image is stored back in the
    normalized_image variable
```

Listing 3.15:

3.2.3 Prediction

```
predictions = model.predict(normalized_image)

```

```
1/1 [==============================] - 0s 25ms/step


predicted_label = np.argmax(predictions)
print(predicted_label)


18
```

Listing 3.16:

By executing these lines of code, the model predicts the label of the preprocessed image. The predictions variable contains an array of probabilities for each class label. The np.argmax function is used to find the index of the highest probability, which corresponds to the predicted label. Finally, the predicted label is printed to the console.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

The sign language recognition project utilized a Convolutional Neural Network (CNN) model trained on the ASL MNIST dataset. The goal of the project was to accurately recognize hand gestures and predict the corresponding sign language letters.

The project involved the following key steps:

Data Preparation: The ASL MNIST dataset was used, which consists of grayscale images of hand gestures representing different sign language letters.

Model Architecture: A CNN model was designed with convolutional layers, max pooling layers, dropout layers, and dense layers. The model was compiled with an appropriate optimizer and loss function.

Training: The model was trained using the training data, with a specified number of epochs and batch size. The training process involved iteratively updating the model's weights to minimize the loss and improve accuracy.

Evaluation: The trained model was evaluated on the test data to measure its performance. The accuracy metric was used to assess how well the model generalized to unseen data.

Image Input and Prediction: A function was implemented to capture an image using the webcam in Colab, preprocess the image, and make predictions using the trained model. The predicted label for the captured image was displayed.

The project successfully achieved the objective of recognizing sign language gestures

and predicting the corresponding letters. The trained model demonstrated good accuracy on the test data, indicating its ability to generalize well to unseen examples. The image input functionality provided an interactive way to input new images and obtain predictions from the model.

Overall, the project showcases the potential of using CNN models for sign language recognition and highlights the importance of dataset preparation, model architecture design, training, and evaluation in achieving accurate results.