ASL (American Sign Language) - Alphabet Image recognition

Introduction:

American Sign Language (ASL) is a rich, visual language used by the Deaf and hard-of-hearing communities in the United States and parts of Canada. It relies on hand shapes, movements, facial expressions, and body language to convey meaning. ASL has its own grammar and syntax, making it a distinct language with its own rules and structure.

Image recognition technology has been increasingly applied to ASL, particularly in recognizing the alphabet represented through handshapes. This involves training algorithms to interpret and distinguish different hand configurations that represent individual letters in the ASL alphabet. By using computer vision and machine learning, these systems aim to identify and translate the gestures accurately.

This technology holds immense potential for bridging communication barriers between the Deaf community and those who don't know sign language. It can facilitate real-time translation, enhance accessibility, and promote inclusivity in various domains, from education to communication platforms and assistive technologies. With advancements in image recognition algorithms, ASL recognition systems continue to evolve, offering promising opportunities for fostering better communication and understanding between diverse communities.

Technical Architecture:

www.smartbridge.com

Prerequisites:

Deep Learning Framework: Choose a framework like TensorFlow, PyTorch, or Keras for model development.

Convolutional Neural Network (CNN): Utilize CNN architectures, which are well-suited for image recognition tasks.

> **Transfer Learning (Optional): Implement pre-trained models like VGG, ResNet, or Inception and fine-tune them for your ASL alphabet recognition task. This approach often works well, especially if you have limited data.**

Project Objectives:

Understanding Convolutional Neural Networks (CNNs): Essential for image recognition tasks, CNNs specialize in processing visual data. Understanding their concepts and techniques is crucial for creating models that recognize ASL alphabet signs accurately.

Comprehensive Understanding of Image Data: ASL recognition heavily relies on image data. Understanding image formats, characteristics, and how to preprocess them for model input is key.

Data Preprocessing Techniques: Cleaning and preparing the data is vital. Techniques like normalization, resizing, noise reduction, and augmentation can significantly enhance model performance.

Constructing a Web Application using Flask: Developing a web application to showcase the ASL alphabet recognition system makes the project more accessible and usable. Flask is a great framework for creating web applications, allowing you to integrate your model and provide a user-friendly interface.

Project Flow:

Data Collection: Gather images of ASL alphabets. There are various datasets available online or create your dataset by capturing images.

Data Preprocessing:

Resize images to a uniform size.

Normalize pixel values (typically between 0 and 1).

Split the data into training and testing sets.

Model Building:

Import necessary libraries like TensorFlow, Keras.

Define the input shape of the image data (e.g., width, height, channels).

Build a CNN model:

Convolutional layers: Apply filters to detect features.

Pooling layers: Reduce spatial dimensions to capture important information.

Fully Connected layers: Flatten the output and add dense layers for classification.

Compile the model specifying optimizer (e.g., Adam), loss function (e.g., categorical cross-entropy), and metrics (e.g., accuracy).

Model Training:

Use the ImageDataGenerator class to augment images during training to increase the dataset size and generalizability.

Train the model using the training set, monitoring validation accuracy to prevent overfitting.

Model Evaluation:

Evaluate the model's performance on the testing set to determine accuracy and other relevant metrics (precision, recall, F1-score).

Adjust model parameters if necessary to improve performance.

Model Deployment:

Save the trained model for future use (serialization).

Deploy the model in real-world applications, potentially through Flask or other deployment frameworks.

Project Structure:

Create a Project folder which contains files as shown below

PROJECT DEVELOPMENT:

DATA COLLECTION:

The Dataset used in this project is collected from the following link: https://www.kaggle.com/datasets/grassknoted/asl-alphabet And it is used in our project with Kaggle API credentials in the following way-

Kaggle API credentials:-

Download and unzip dataset:

DATA PREPARATION:-

Importing the necessary libraries:

```
# Load Data
import os
import cv2
import numpy as np

# Data Visualisation
import matplotlib.pyplot as plt

# Model Training
from tensorflow.keras import utils
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D,
MaxPooling2D, BatchNormalization
from sklearn.model_selection import train_test_split
from tensorflow.keras.applications import VGG16

# Warning
import warnings
warnings.filterwarnings("ignore")

# Main
import os
import glob
import cv2
```

```
import numpy as np
import pandas as pd
import gc
import string
import time
import random
from PIL import Image
from tqdm import tqdm
tqdm.pandas()

# Visualization
import matplotlib
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE

# Model
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.preprocessing.image import load_img, img_to_array,
array_to_img
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.layers import Dense, Flatten, Dropout,
GlobalAveragePooling2D
from keras.models import load_model, Model
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint, EarlyStopping
from sklearn.metrics import classification_report
```

Configuring the parameters:

```
# Configuration
class CFG:
    # Set the batch size for training
    batch_size = 128
    # Set the height and width of input images
    img_height = 32
    img_width = 32
    epochs = 10
    num_classes = 29
    # Define the number of color channels in input images
    img_channels = 3
# Define a function to set random seeds for reproducibility
def seed_everything(seed: int):
```

```
    random.seed(seed)
    # Set the environment variable for Python hash seed
    os.environ["PYTHONHASHSEED"] = str(seed)
    np.random.seed(seed)
    tf.random.set_seed(seed)
```

Creating labels:

```

```

DATA PREPROCESSING:

Creating metadata:

```

```

Splitting the data into train and test sets:

```
# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    metadata['image_path'],
    metadata['label'],
    test_size=0.2,
    random_state=2253,
    shuffle=True,
    stratify=metadata['label']
)

# Create a DataFrame for the training set test set
data_train = pd.DataFrame({
```

```
    'image_path': X_train,
    'label': y_train
})

data_test = pd.DataFrame({
    'image_path': X_test,
    'label': y_test
})

# Split the training set into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(
    data_train['image_path'],
    data_train['label'],
    test_size=0.2/0.7,  # Assuming you want 20% for validation out of
the training set
    random_state=2253,
    shuffle=True,
    stratify=data_train['label']
)

# Create a DataFrame for the validation set
data_val = pd.DataFrame({
    'image_path': X_val,
    'label': y_val
})
```

DATA AUGMENTATION:

Applying data augmentation to train, test, validation data:

```
def data_augmentation():
    datagen = ImageDataGenerator(
        rescale=1/255.,
        # Add other augmentation parameters as needed
        rotation_range=20,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest'
    )
    train_generator = datagen.flow_from_dataframe(
        data_train,
        directory='./',
```

```
        x_col='image_path',
        y_col='label',
        class_mode='categorical',
        batch_size=CFG.batch_size,
        target_size=(CFG.img_height, CFG.img_width)
    )
    validation_generator = datagen.flow_from_dataframe(
        data_val,
        directory='./',
        x_col='image_path',
        y_col='label',
        class_mode='categorical',
        batch_size=CFG.batch_size,
        target_size=(CFG.img_height, CFG.img_width)
    )

    test_generator = datagen.flow_from_dataframe(
        data_test,  # Assuming you have a DataFrame for test data
        directory='./',
        x_col='image_path',
        y_col='label',
        class_mode='categorical',
        batch_size=CFG.batch_size,
        target_size=(CFG.img_height, CFG.img_width),
        shuffle=False  # Set to False for test data
    )

    return train_generator, validation_generator, test_generator

# Seed for reproducibility
seed_everything(2253)

# Get the generators
train_generator, validation_generator, test_generator =
data_augmentation()
```

MODEL BUILDING:

```
# Define input shape
input_shape = (32, 32, 3)
```

```
# Load the VGG16 model without the top (classification) layers
base_model = VGG16(weights='imagenet', include_top=False,
input_shape=input_shape)

# Add your custom classification layers on top of the base model
x = GlobalAveragePooling2D()(base_model.output)
x = Dense(128, activation='relu')(x)  # You can adjust the number of
units as needed
predictions = Dense(29, activation='softmax')(x)  # num_classes is the
number of classes in your dataset

# Create the final model
model = Model(inputs=base_model.input, outputs=predictions)

# Summarize the model architecture
model.summary()
```

Compiling and training the model:

MODEL EVALUATION:

Evaluating the model using accuracy, confusion matrix and giving classification report:

APPLICATION BUILDING:
Creating an HTML Page.
Adding styles to it using css.
Adding actions to it using javascript.
Creating App.py python script for web application that uses the model for image classification predictions.
Executing these files using Spyder IDE.

HTML CODE:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
```