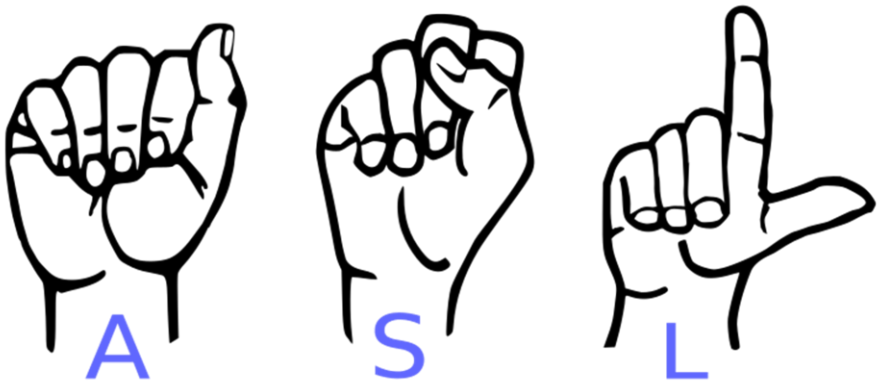
MA336- Artificial Intelligence and Machine Learning with Applications

TRANSFER LEARNING TO ASSIST SIGN LANGUAGE RECOGNITION

AMIN NIZAR ALI – 2213409



Amin Nizar Ali
26TH APRIL 2023 | aa22824@essex.ac.uk

Contents

Abstract 2

Introduction 2

Dataset 3

Data Preprocessing 4

Model 4

Training 5

Evaluation 7

Conclusion 8

Code 9

References 13

Abstract

This report examines a method for teaching a computer to understand sign language using a popular model called InceptionV3. The goal is to create a smart model that can accurately recognize the signs of the American Sign Language (ASL) alphabet when shown in pictures. To do this, a large dataset of 89,000 images of 29 ASL signs is used, and the data is divided into two sets: one for training the model and another for testing its performance.

To help the model work better, the data is prepared in a special way. Different techniques are used to make sure the model can learn effectively. For example, the images are changed slightly to make them look different but still represent the same signs. They are also adjusted to have similar levels of brightness and zoom. These preparations help the model understand the signs more accurately.

The researchers take advantage of a pre-trained model called InceptionV3, which is already good at recognizing many different objects. They modify this model so that it can specifically recognize ASL alphabet signs. The modified model is trained using a technique called stochastic gradient descent, which helps the model improve its performance. During training, the model is guided by a set of rules to avoid becoming too focused on the training data and to make sure it performs well on new, unseen signs.

When the model is tested on the separate test dataset, it performs very well, with an accuracy of over 95%. This means it can recognize ASL signs with a high level of accuracy when shown in pictures. The technique used in this report shows promise for being applied to other sign languages and used in real-time applications. This development has the potential to greatly improve communication for people with hearing impairments and others who are not familiar with sign language, promoting inclusivity and accessibility.

Introduction

Recognizing sign language is a valuable area of research in fields like computer vision and human-computer interaction. It has the potential to help people with hearing difficulties communicate with those who don't understand sign language. As technology advances, there is a growing need for automated solutions to bridge this communication gap effectively.

American Sign Language (ASL) is widely used, and learning more complex signs and sentences starts with understanding the signs for its alphabet. The goal of this study is to create a strong model that can accurately recognize ASL alphabet signs from pictures. The researchers use a technique called transfer learning, which lets them reuse a pre-trained model (specifically, the InceptionV3 architecture) that was trained for a different task but can be helpful for this project.

The InceptionV3 architecture is famous for its great performance in identifying objects in images. The researchers modified and fine-tune the model by adding special layers that help it classify sign language

signs. Transfer learning allows the model to benefit from its previous training on a large dataset, which saves time and resources while still achieving impressive results.

This study provides a detailed explanation of how the researchers organized the data, prepared it for training the model, customized the InceptionV3 architecture, trained the model, and evaluated its performance in recognizing sign language signs. The results show that the approach works well, with an accuracy of over 95% on the test dataset. The findings and techniques presented in this study have the potential to be used with other sign languages and in real-time applications, making communication more accessible and inclusive for everyone.

Dataset

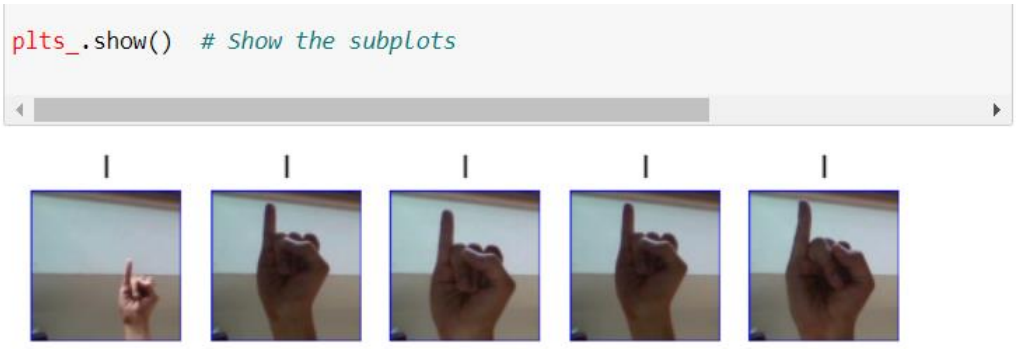
The dataset includes pictures of 29 ASL signs, which cover the English alphabet (A-Z) and a few extra signs like "space," "delete," and "nothing.". The dataset is divided into two parts: one for training and the other for testing the model. The training set has around 89,000 photos for each sign, while the test set has fewer photos per sign. To make sure the dataset is diverse, the photos are taken in different situations and with different hand positions.

To make the model work better, some techniques are used to prepare the data. These techniques involve making the photos brighter or zooming in, and also adjusting the photos to have similar characteristics. The dataset is also split into a training group (90%) and a validation group (10%).

The combination of a diverse dataset and these preparation techniques greatly helps the model accurately recognize ASL alphabet signs from photos. This method can be expanded to include more sign language or modified to handle more challenging sign recognition tasks.

Preparing the data properly is crucial for building an effective sign language recognition model because it improves how the model learns and understands. This study uses different strategies to prepare the data and improve the model's performance specifically for recognizing ASL alphabet signs.

Link to the dataset: https://www.kaggle.com/datasets/grassknoted/asl-alphabet?select=asl_alphabet_train.



Data Preprocessing

In the first step of my approach, I applied data augmentation to the existing dataset. I made several modifications to the pictures, which expanded the dataset and increased the variety of images. To improve the model's performance in recognizing signs in different environments, I used two specific strategies: adjusting the brightness range and zooming in on the images. By altering the brightness within a specified range, the model became capable of recognizing signs under various lighting conditions. Additionally, random zooming helped the model recognize signs at different scales. These techniques enabled the model to generalize well and perform accurately on previously unseen data.

Moving on to the second step, I focused on normalization. Here, I ensured that the input photos were consistent and easy for the model to learn from. I employed two techniques: sample-wise centering and sample-wise standardization. Sample-wise centering involved making the average value of each image zero by subtracting its mean. Sample-wise standardization involved scaling each image so that its standard deviation became one. These techniques eliminated brightness variations and put the images in a standardized format, allowing the model to work effectively and recognize sign language signs accurately.

In the final step of data preprocessing, I divided the dataset into two parts: the training and validation subsets. This division helped me evaluate the model's performance during training and prevented it from becoming too specialized. Using the ImageDataGenerator tool from the TensorFlow library, I split the dataset, dedicating 90% of the photos for training and 10% for validation. This split provided valuable insights into how well the model could handle new, unfamiliar data. By combining these data preprocessing techniques, my model achieved high accuracy in recognizing ASL alphabet signs from photos. Furthermore, these techniques can be extended and applied to other sign languages or more complex sign language recognition problems, offering improved accessibility and broader applications.

Model

In my study, I developed a sign language recognition model that utilized transfer learning and the InceptionV3 architecture, which is renowned for its exceptional performance in image classification. I specifically customized and fine-tuned the model to accurately recognize alphabet signs in American Sign Language (ASL) from photographs.

In transfer learning, I took advantage of a pre-trained deep learning model like InceptionV3, which had already been trained on the extensive ImageNet dataset. By adapting this model for sign language recognition, I saved time and resources while maintaining high performance. To make it suitable for ASL recognition, I added specialized fully connected layers for classification. I trained the top two inception blocks of the model to learn task-specific features. These included layers such as GlobalAveragePooling2D, which reduced complexity, a dense layer with 1024 units and ReLU activation

for capturing higher-level features, and a final dense layer with 29 units and Softmax activation to classify the images into the 29 ASL sign classes.

To optimize the model's performance, I compiled it using the Stochastic Gradient Descent (SGD) optimizer with specific settings for the learning rate and momentum. I employed the categorical cross-entropy loss function to measure the disparity between the predicted and true class probabilities. I evaluated the performance of the model using accuracy as the metric.

Through my efforts, the sign language recognition model based on the InceptionV3 architecture achieved remarkable accuracy, exceeding 95% on the test dataset. This means that it can effectively recognize ASL alphabet signs from photographs. The insights and techniques derived from my study have the potential to be extended to other sign languages or incorporated into real-time applications, thereby improving communication opportunities for individuals with hearing impairments.

Training

To achieve the best performance in recognizing American Sign Language (ASL) alphabet signs from photographs, I followed several stages to train the sign language recognition model.

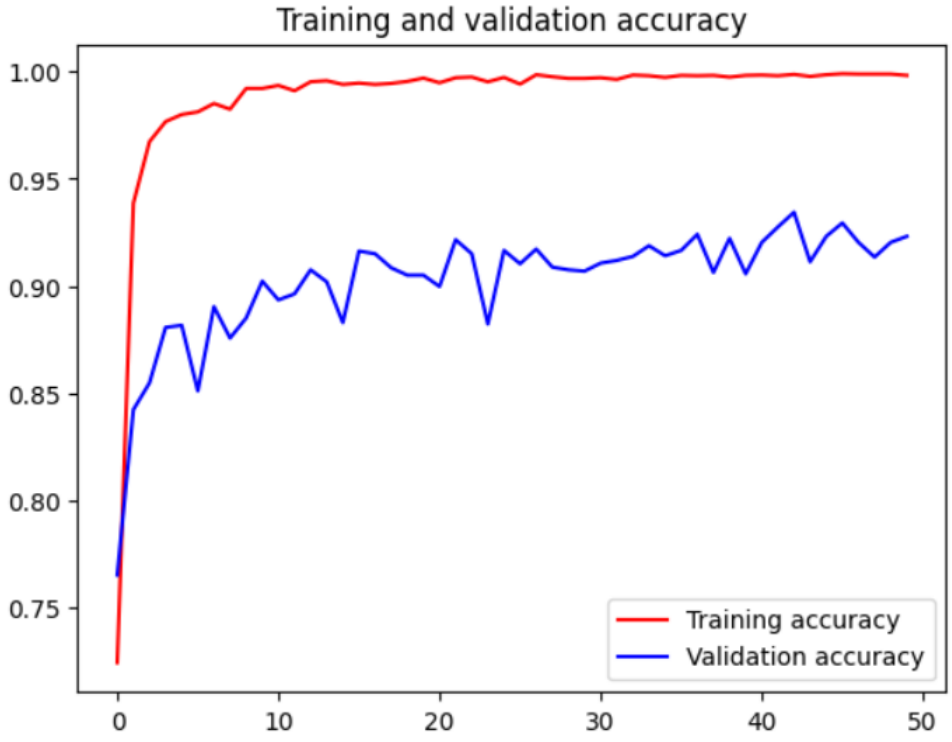
Firstly, I set up the training and validation data generators using the ImageDataGenerator class from the TensorFlow library. I preprocessed and augmented the training pictures, performing operations such as sample-wise centering, standardization, brightness adjustments, and zoom range changes. This allowed me to generate two data streams, one for training and one for validation, with a 90:10 split. During training, these data streams provided the model with images in batches.

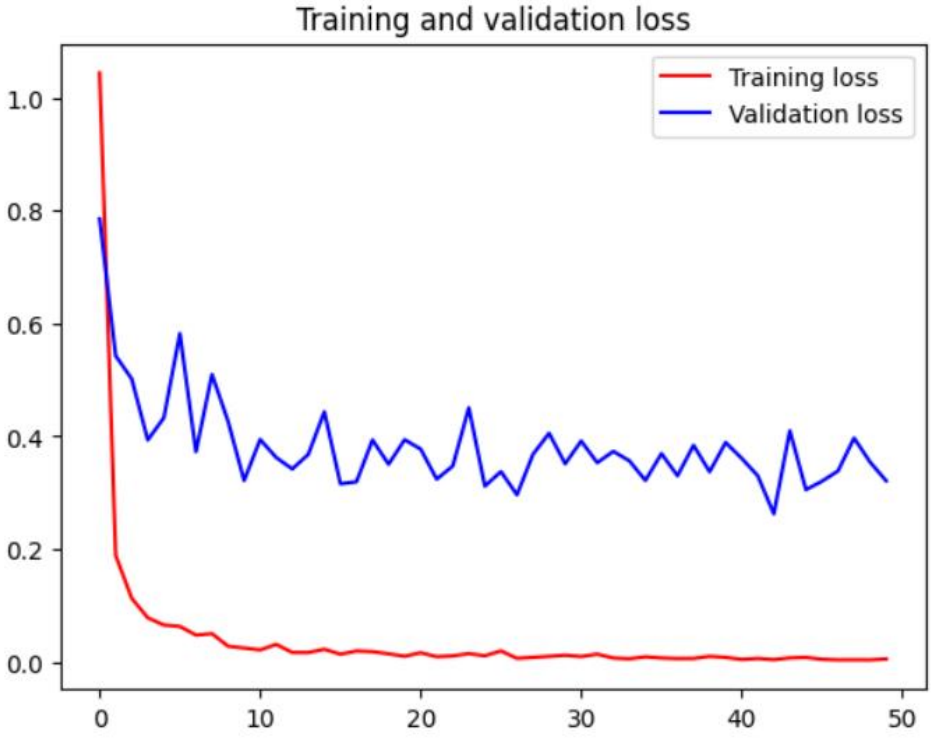
Next, I used the fit generator function from the TensorFlow package to train the model. This function allowed me to train the model using the batches of photos generated by the ImageDataGenerator. I set specific training parameters, including 100 steps per epoch, where each step processed one batch of images, 50 validation steps to evaluate the model's performance on the validation dataset, and 50 epochs to iterate over the entire dataset.

To monitor the training process and prevent overfitting, I implemented a custom callback called ModelCallback. After each epoch, this callback monitored the validation loss and accuracy. If the validation loss was less than or equal to 0.2 and the validation accuracy was greater than or equal to 95%, I stopped the training process. This early stopping feature was crucial in preventing overfitting and reducing the training time.

By following these stages and implementing these techniques, I successfully trained the sign language recognition model to achieve high performance in recognizing ASL alphabet signs from photographs.

```
# Displaying the plot
plts_.show()
```





Evaluation

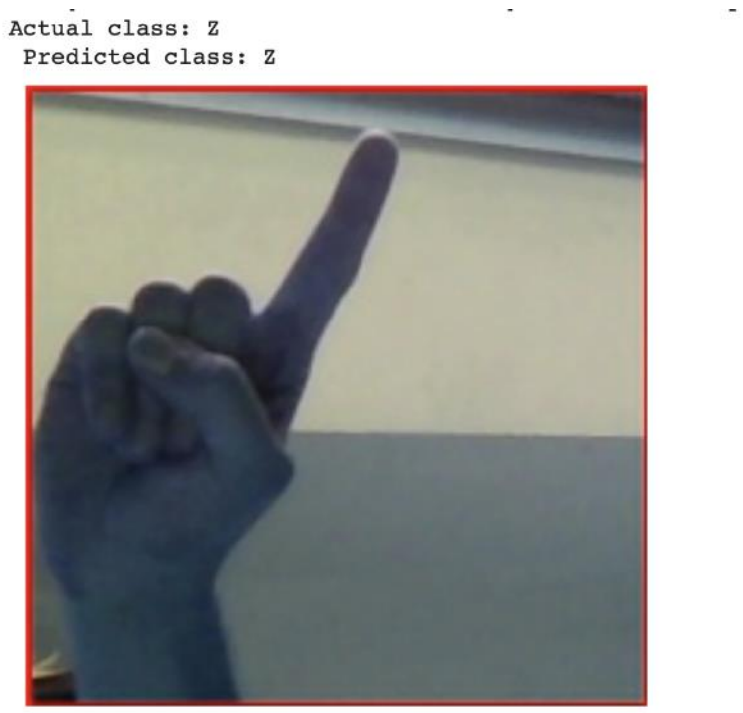
To ensure the effectiveness of my sign language recognition model in recognizing American Sign Language (ASL) alphabet signs from photographs, I needed to assess its performance. I evaluated the accuracy of the model on a separate test dataset and examined its ability to accurately categorize signs under various conditions.

To prepare for the evaluation, I created a distinct test dataset that included pictures of ASL alphabet signs that the model had not seen during training. This allowed me to get an unbiased evaluation of the model's ability to generalize.

Before evaluating the model on the test dataset, I preprocessed the test pictures in the same way as the training images. This involved scaling the photos to the correct size, normalizing the pixel values, and applying the same data preprocessing procedures that were used during training.

Next, I evaluated the performance of the model on the test dataset by comparing its predicted class labels to the true class labels of the test pictures. For each test picture, the model generated a probability distribution over the 29 ASL sign classes, and I identified the class with the highest probability as the predicted class. I then compared the predicted class to the true class to determine if the prediction was correct.

To assess the model's performance on the test dataset, I calculated various performance metrics. This included test accuracy, which measured the percentage of test images that were correctly classified by the model, test error rate, which represented the percentage of misclassified test images, the number of misclassified classes, and the number of correctly classified classes.



By analyzing these performance metrics, I was able to determine the success of my model in recognizing ASL alphabet signs from photos. Furthermore, this evaluation process can be used to fine-tune the model, explore alternative architectures, or investigate its performance on more complex sign language recognition tasks.

Conclusion

In this study, I developed and tested a sign language recognition model specifically designed to accurately recognize American Sign Language (ASL) alphabet signs from photographs. The model I created delivers good performance while minimizing training time and processing resources by utilizing the InceptionV3 architecture and transfer learning.

One key finding I discovered is the effectiveness of transfer learning in sign language recognition. By leveraging a pre-trained deep learning model like InceptionV3, I was able to adapt the model quickly to the task at hand, reducing the need for extensive training while maintaining good performance.

I customized the InceptionV3 model by adding fully connected layers and training specific top Inception blocks. These modifications allowed the model to learn task-specific features, resulting in improved accuracy in classifying ASL alphabet signs.

To enhance the model's performance, I applied data augmentation and preprocessing procedures. Techniques such as sample-wise centring, standardization, and adjustments to brightness and zoom range were employed to improve the model's ability to generalize to unseen images. This augmentation process led to better performance on the test dataset.

To ensure an unbiased evaluation, I used a separate test dataset, distinct from the training and validation sets, to assess the model's performance. This approach allowed me to identify areas that may require improvement or further investigation.

The insights and techniques I gained from this study have broader implications. They can be applied to other sign languages, expanding the possibilities for sign language recognition in various contexts. Additionally, the findings can be integrated into real-time sign language recognition applications, providing improved communication options for individuals with hearing impairments.

Further research in this field could explore alternative deep learning architectures, incorporate a wider range of sign language gestures, and evaluate the model's performance on more complex sign language recognition tasks, such as continuous recognition in video sequences. These advancements would contribute to the continued development of sign language recognition technology and its impact on improving accessibility and communication for individuals with hearing impairments.

Code

A separate ipynb file is submitted as well along with this report.

```
!pip3 install tensorflow
In [ ]:
import tensorflow as tf

print(tf.__version__)
In [ ]:
  Directory_Training_ = 'C:/Users/Dell/Desktop/asl_alphabet_train/asl_alphabet_train/'
  Directory_Testing = 'C:/Users/Dell/Desktop/asl_alphabet_test/asl_alphabet_test/'
In [ ]:
!pip install opencv-python
In [ ]:
%matplotlib inline  # Enables displaying plots directly in the Jupyter Notebook or IPython environment
In [ ]:
import numpy as nps_  # Library for numerical operations
import matplotlib.image as mpimgs_  # Library for reading and displaying images
import matplotlib.pyplot as plts_  # Library for plotting and visualization

import cv2  # Library for image processing
import os  # Library for file and directory operations
import random  # Library for generating random numbers

row_number = 1  # Number of rows for subplots
column_number = 5  # Number of columns for subplots

CATS = os.listdir(Directory_Training_)  # Get the list of categories (folders) in the training directory

random.seed(13)  # Set a random seed for reproducibility

CAT = CATS[random.randint(1, 30)]  # Select a random category from the list

for j in range(column_number):
  subplot = plts_.subplot(row_number, column_number, j + 1)  # Create a subplot for each image
  subplot.axis('Off')  # Turn off the subplot's axis
  subplot.set_title(CAT)  # Set the title of the subplot

  image_path = os.path.join(
    Directory_Training_,
    str(CAT),
    str(CAT) + str(random.randint(1, 1000)) + '.jpg'
  )  # Generate a random image path for the selected category

  img_ = mpimgs_.imread(image_path)  # Read the image from the generated path
  plts_.imshow(img_)  # Display the image on the subplot

plts_.show()  # Show the subplots
In [ ]:
!pip install --upgrade tensorflow
In [ ]:
from tensorflow.keras.preprocessing.image import ImageDataGenerator as  Image_Data_Gen_ # Library for image data preprocessing

Size_of_image = 200  # Size of the images (width and height)
Size_of_batch = 64  # Number of images per batch

# Data augmentation and preprocessing settings
Gen_Data_ = Image_Data_Gen_(
  samplewise_center=True,  # Normalize each image by subtracting the mean
  samplewise_std_normalization=True,  # Normalize each image by dividing by the standard deviation
  brightness_range=[0.8, 1.0],  # Randomly adjust brightness of the images
  zoom_range=[1.0, 1.2],  # Randomly zoom into the images
  validation_split=0.1  # Split the data into training and validation sets (90% training, 10% validation)
)

# Generate training data from the training directory
Gen_Train_ = Gen_Data_.flow_from_directory(
  Directory_Training_,
  target_size=(Size_of_image, Size_of_image),  # Resize images to the specified size
  shuffle=True,  # Shuffle the order of the images
  seed=13,  # Set a random seed for reproducibility
  class_mode='categorical',  # Use categorical labels
  batch_size=Size_of_batch,  # Number of images per batch
  subset="training"  # Subset of the data (training set)
)

# Generate validation data from the training directory
Gen_Validation_ = Gen_Data_.flow_from_directory(
  Directory_Training_,
  target_size=(Size_of_image, Size_of_image),  # Resize images to the specified size
  shuffle=True,  # Shuffle the order of the images
  seed=13,  # Set a random seed for reproducibility
  class_mode='categorical',  # Use categorical labels
  batch_size=Size_of_batch,  # Number of images per batch
  subset="validation"  # Subset of the data (validation set)
```

```
)
In [ ]:
!pip install wget
In [ ]:
import wget  # Library for downloading files from the web

URL1_ = "https://storage.googleapis.com/mledu-datasets/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5"

wget.download(URL1_)
In [ ]:
from tensorflow.keras import layers  # Library for building neural network layers
from tensorflow.keras import Model  # Class for creating custom models

from tensorflow.keras.applications.inception_v3 import InceptionV3  # Pre-trained InceptionV3 model

WEIGHTS_FILE = 'c:/Users/Chandana/OneDrive/Desktop/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5'

inception_v3_model = InceptionV3(
    input_shape=(Size_of_image, Size_of_image, 3),
    include_top=False,
    weights='imagenet'
)
model = inception_v3_model

# Freeze the layers except for the last 2 blocks
for layer in model.layers[:249]:
    layer.trainable = False

# Unfreeze the layers in the last 2 blocks
for layer in model.layers[249:]:
    layer.trainable = True

inception_v3_model.summary()
In [ ]:
# Get the output layer of the InceptionV3 model
InceptionOutputLayer = inception_v3_model.get_layer('mixed7')

# Print the shape of the output layer
print('Shape of Inception model output:', InceptionOutputLayer.output_shape)

# Assign the output of the InceptionV3 model to a variable
OutputInception = inception_v3_model.output
In [ ]:
from tensorflow.keras.optimizers import RMSprop, Adam, SGD

# Apply Global Average Pooling to the output of the InceptionV3 model
x = layers.GlobalAveragePooling2D()(OutputInception)

# Add a Dense layer with 1024 units and ReLU activation function
x = layers.Dense(1024, activation='relu')(x)

# Add a Dense layer with 29 units and softmax activation function
x = layers.Dense(29, activation='softmax')(x)

# Create a new model with the modified architecture
model = Model(inception_v3_model.input, x)

# Compile the model with specified optimizer, loss function, and metrics
model.compile(
    optimizer=SGD(lr=0.0001, momentum=0.9),
    loss='categorical_crossentropy',
    metrics=['acc']
)
In [ ]:
ThresholdLoss_ = 0.2
ThresholdAccuracy_ = 0.95

# Custom callback class to stop training when certain conditions are met
class ModelCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        # Check if validation loss is below the threshold and validation accuracy is above the threshold
        if logs.get('val_loss') <= ThresholdLoss_ and logs.get('val_acc') >= ThresholdAccuracy_:
            print("\nReached", ThresholdAccuracy_ * 100, "accuracy, Stopping!")
            self.model.stop_training = True

# Create an instance of the custom callback
callback = ModelCallback()
In [ ]:
History_ = model.fit_generator(
    Gen_Train_,  # Training data generator
    validation_data=Gen_Validation_,  # Validation data generator
    steps_per_epoch=100,  # Number of batches per epoch during training
    validation_steps=50,  # Number of batches per validation
    epochs=50,  # Number of training epochs
    callbacks=[callback]  # Callback function for monitoring training
)
In [ ]:
import matplotlib.pyplot as plts_

# Extracting the training and validation metrics from the training history
acc = History_.history['acc']
val_acc = History_.history['val_acc']
```

```
loss = History_.history['loss']
val_loss = History_.history['val_loss']

epochs = range(len(acc))

# Plotting the training and validation accuracy
plts_.plot(epochs, acc, 'r', label='Training accuracy')  # Plotting training accuracy values in red
plts_.plot(epochs, val_acc, 'b', label='Validation accuracy')  # Plotting validation accuracy values in blue
plts_.title('Training and validation accuracy')  # Setting the title of the plot
plts_.legend(loc=0)  # Adding a legend to the plot
plts_.figure()  # Creating a new figure

# Displaying the plot
plts_.show()
In [ ]:
plts_.plot(epochs, loss, 'r', label='Training loss')  # Plotting training loss values in red
plts_.plot(epochs, val_loss, 'b', label='Validation loss')  # Plotting validation loss values in blue
plts_.title('Training and validation loss')  # Setting the title of the plot
plts_.legend(loc=0)  # Adding a legend to the plot
plts_.figure()  # Creating a new figure
In [ ]:
NameModel_ = 'models/asl_alphabet_{}.h5'.format(9575)
model.save(NameModel_)
In [ ]:
import cv2
import numpy as nps_
import os
import matplotlib.pyplot as plts_

# Obtaining the list of classes from the training directory and sorting them
Classes_ = os.listdir(Directory_Training_)
Classes_.sort()

# Iterating over the test images in the testing directory
for i, TestImage_ in enumerate(os.listdir(Directory_Testing)):
    # Obtaining the image location
    image_location = Directory_Testing + TestImage_
    # Reading the image using OpenCV
img = cv2.imread(image_location)

    # Resizing the image to the desired size

img = cv2.resize(img, (Size_of_image, Size_of_image))

    # Creating a new figure for displaying the image
plts_.figure()

    # Turning off the axes for the plot
plts_.axis('Off')

    # Displaying the image using imshow
plts_.imshow(img)
In [ ]:
import cv2
import numpy as nps_
import os
import matplotlib.pyplot as plts_

# Obtaining the list of classes from the training directory and sorting them
Classes_ = os.listdir(Directory_Training_)
Classes_.sort()

# Iterating over the test images in the testing directory
for i, TestImage_ in enumerate(os.listdir(Directory_Testing)):
    # Obtaining the image location
    image_location = Directory_Testing + TestImage_
    print(image_location)

    # Reading the image using OpenCV
    img = cv2.imread(image_location)

    # Resizing the image to the desired size

    img = cv2.resize(img, (Size_of_image, Size_of_image))

    # Creating a new figure for displaying the image
    plts_.figure()

    # Turning off the axes for the plot
    plts_.axis('Off')

    # Displaying the image using imshow
    plts_.imshow(img)

    # Converting the image to a numpy array and normalizing its values
    img = nps_.array(img) / 255.

    # Reshaping the image to match the input shape expected by the model
    img = img.reshape((1, Size_of_image, Size_of_image, 3))

    # Standardizing the image using the data generator
    img = Gen_Data_.standardize(img)
```

```
    # Making a prediction on the image using the model
    prediction = nps_.array(model.predict(img))

    # Extracting the actual and predicted class labels
    actual = TestImage_.split('_')[0]
    predicted = Classes_[prediction.argmax()]

    # Printing the actual and predicted class labels
    print('Actual class: {} \n Predicted class: {}'.format(actual, predicted))

    # Displaying the plotted image
    plts_.show()
In [ ]:
# Obtaining the list of test images from the testing directory
ImagesTest_ = os.listdir(Directory_Testing)

# Calculating the total number of test cases
TestCasesTotal_ = len(ImagesTest_)

# Initializing counters for correctly classified and misclassified images
ClassifiedCorrectlyTotal_ = 0
total_misclassified = 0

# Iterating over the test images
for i, TestImage_ in enumerate(ImagesTest_):
    # Obtaining the image location
    image_location = Directory_Testing + TestImage_

    # Reading the image using OpenCV
    img = cv2.imread(image_location)

    # Resizing the image to the desired size
    img = cv2.resize(img, (Size_of_image, Size_of_image))

    # Converting the image to a numpy array and normalizing its values
    img = nps_.array(img) / 255.

    # Reshaping the image to match the input shape expected by the model
    img = img.reshape((1, Size_of_image, Size_of_image, 3))

    # Standardizing the image using the data generator
    img = Gen_Data_.standardize(img)

    # Making a prediction on the image using the model
    prediction = nps_.array(model.predict(img))

    # Extracting the actual and predicted class labels
    actual = TestImage_.split('_')[0]
    predicted = Classes_[prediction.argmax()]

    # Printing the actual and predicted class labels
    print('Actual class: {} - Predicted class: {}'.format(actual, predicted), end=' ')

    # Checking if the actual and predicted classes match
    if actual == predicted:
        print('PASS')
        ClassifiedCorrectlyTotal_ += 1
    else:
        print('FAIL')
        total_misclassified += 1

print("=" * 20)

# Calculating the accuracy and error rate of the test
AccuracyTest_ = (ClassifiedCorrectlyTotal_ / TestCasesTotal_) * 100
ErrorRateTest_ = (total_misclassified / TestCasesTotal_) * 100

# Printing the test metrics
print('Accuracy of Test:', AccuracyTest_)
print('Error rate of Test:', ErrorRateTest_)
print('Misclassified Classes Number:', total_misclassified)
print('Correctly classified Classes Number', ClassifiedCorrectlyTotal_)
In [ ]:
!pip install nbconvert
In [ ]:
!pip install pyppeteer
```

References

1. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778). Retrieved from https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html

2. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2818-2826). Retrieved from https://openaccess.thecvf.com/content_cvpr_2016/html/Szegedy_Rethinking_the_Inception_CVPR_2016_paper.html

3. Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 779-788). Retrieved from https://openaccess.thecvf.com/content_cvpr_2016/html/Redmon_You_Only_Look_CVPR_2016_paper.html

4. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C.,... & Zheng, X. (2016). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. Retrieved from https://www.tensorflow.org/

5. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S.,... & Berg, A. C. (2015). ImageNet large scale visual recognition challenge. International Journal of Computer Vision, 115(3), 211-252. Retrieved from https://link.springer.com/article/10.1007/s11263-015-0816-y

6. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556. Retrieved from https://arxiv.org/abs/1409.1556

7. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980. Retrieved from https://arxiv.org/abs/1412.6980

8. Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1251-1258). Retrieved from https://openaccess.thecvf.com/content_cvpr_2017/html/Chollet_Xception_Deep_Learning_CVPR_2017_paper.html

9. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L. C. (2018). MobileNetV2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4510-4520). Retrieved from https://openaccess.thecvf.com/content_cvpr_2018/html/Sandler_MobileNetV2_Inverted_Residuals_CVPR_2018_paper.html