IST707: APPLIED MACHINE LEARNING

A Report on

**SIGN-VISION:**

**An Image Classification System**

**For Sign Language Detection**

*Prepared By – Group 8*

**Aditi Pala (**[**adpala@syr.edu**](mailto:adpala@syr.edu)**)**

**Pankaj Yadav (**[**pyadav05@syr.edu**](mailto:pyadav05@syr.edu)**)**

**Sharvil Arjunwadkar (**[**sarjunwa@syr.edu**](mailto:sarjunwa@syr.edu)**)**

**Yashaswini Kulkarni (**[**yakulkar@syr.edu**](mailto:yakulkar@syr.edu)**)**

# **Introduction**

Sign language is a crucial means of communication for people with hearing or speech disabilities. With the increasing use of technology, the development of a deep learning-based image classification system can improve the accuracy of sign language recognition. This project aims to create an image classification model that can accurately identify signs in real-time. The below diagram gives a pictorial representation of the flow of our project.

Graphical user interface

Description automatically generated

To achieve accurate image classification, we plan to source image data from open source platforms like Kaggle. Once we have collected sufficient data, we will preprocess it to enhance its quality and make it easier to process. We then plan to split the data into three parts - a training set, a validation set, and a testing set. We will allocate 80% of the data to the training set, 10% to the validation set, and the remaining 10% to the testing set. This will ensure that our models are trained on a sufficient amount of data while also allowing us to evaluate their performance on unseen data. We plan to train multiple models using various architectures. To ensure that our models are performing optimally, we will train them through multiple iterations and evaluate them on the validation dataset. This will allow us to fine-tune their hyperparameters and adjust their architectures to improve their accuracy. Finally, once we have obtained acceptable results on the validation dataset, we will test our models on the testing set to get the final evaluation. Based on the results, we will select the best-performing model for deployment.

# **Objective**

This project is focused on creating an advanced system for recognizing sign language images in real-time. The main goal is to develop image classification models that are able to accurately detect and identify different signs used in sign language. The system will utilize advanced machine learning and deep learning techniques to achieve the desired level of accuracy and precision in sign recognition. This project will have a significant impact on improving accessibility for people with hearing impairments and will serve as a powerful tool for communication in a variety of settings.

# **Data Description**

To train the deep learning model, we require a large dataset of sign language images. The dataset should include various sign language gestures and include images with different backgrounds, lighting conditions, and angles. We will collect the data from publicly available datasets like the American Sign Language dataset and others that will help to improve the accuracy of the model.  
   
ASL: <https://www.kaggle.com/datasets/prathumarikeri/american-sign-language-09az>

<https://www.kaggle.com/datasets/grassknoted/asl-alphabet>   
   
We are working with Image data. Each label represents an image.  
   
We have 142k images files with 36 labeled classes from the first ASL dataset and We had around 87k image files from the second ASL dataset, however, not all classed were required from the dataset, so, we only included an additional 3 classes (space, del and nothing) containing 3k image files into our current dataset, making the size if our current dataset around 157k.  
   
The data includes different sign language gestures with different background images, lightings, brightness, image conditions etc.

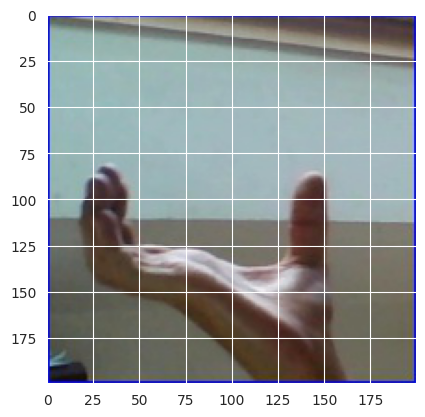
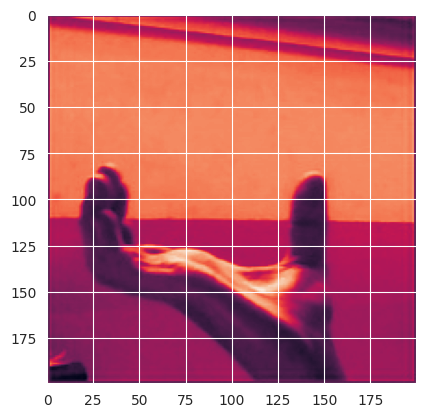
We can try to train the model from more than one source to improve the accuracy of the model.

# **Data Preprocessing**

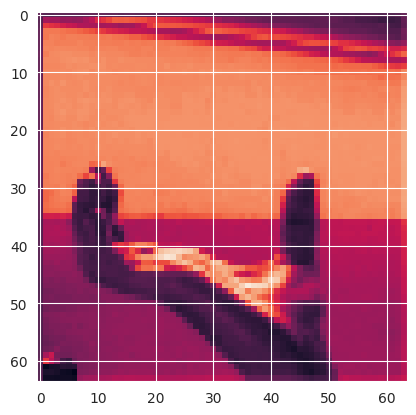
## **Preprocessing Techniques:**

To improve the performance of the model, we used the following preprocessing techniques:

RGB to Grayscale: This was mainly done because it simplifies the data, as it reduces the amount of data in the image from three color channels (red, green, and blue) to a single grayscale channel. By converting the RGB images to grayscale can also eliminate the hue and saturation information while retaining the luminance.

Resizing Images: We resize the images to a standard size to ensure consistency in the dataset. This helps in reducing the computational cost and makes the dataset more manageable.



Normalizing Pixel Values: We divided the pixel values by 255 to normalize the pixel values of the image to a fixed range. This helps in improving the training process and prevents issues caused by different brightness and contrast levels.

Data Augmentation: We added new data to the dataset by including images of delete, space, and nothing. We created an IsSign label that shows the value 1 for sign present and shows value 0 for no sign present or space. We flagged 0 for nothing, 1 for alphabets, numbers, spaces, and delete.

## **Data Sampling Methods:**

The dataset was imbalanced, meaning some labels had more data points than others. To combat this, we used the following data sampling methods:

Random Sampling: We used random sampling to ensure that each alphabet/number image had the same number of data points, and the data was balanced. This approach ensures that the majority class has the same number of samples as the minority class.

Hybrid Sampling: Rather than using just one sampling method, we will be a hybrid sampling method that combines multiple sampling strategies to balance the data. We will use some of the following sampling methods in our hybrid sampling approach:

* Random Sampling: We will use random sampling in the majority class.
* Over-Sampling: We can create additional samples in the minority class using data augmentation techniques such as rotation, flipping, and scaling to the existing images in the minority class.
* Under-Sampling: We can randomly select a subset of samples from the majority class to reduce the number of samples in the majority class to match the number of samples in the minority class.
* Stratified Sampling: We can divide the dataset into subgroups based on the class labels and then sampled from each subgroup in proportion to the number of samples in each subgroup.
* Cluster-Based Sampling: We can groupe similar samples together based on some distance metric and then sampled from each cluster to balance the data.
* Synthetic Minority Over-sampling Technique (SMOTE): We can use SMOTE, a popular oversampling method that generates synthetic samples in the minority class by interpolating between existing samples.

Before Sampling After Sampling

Chart, bar chart, histogram

Description automatically generated Chart

Description automatically generated with low confidence

## **Conclusion:**

In conclusion, we have discussed various preprocessing techniques and data sampling methods used to improve the dataset for classification of sign and no-sign images. We have shown that a hybrid sampling method that combines multiple sampling strategies can be used to balance the data and prevent loss of information. We recommend further experimentation with different preprocessing techniques and sampling methods to optimize the dataset.

# **Models Trained**

We decided to start by building the CNN models as there might me some unforeseen problems which might require time to resolve and everyone in the team needed more hands on practice and understanding of CNN models as compared to some of the other models we planned to build such as KNN and SVM (Support Vector Machines) etc..

## **InceptionResNetv2 with ImageNet:**

InceptionResNetv2 is a deep convolutional neural network model that combines the ideas from the Inception and ResNet models. It was trained on the ImageNet dataset and achieved state-of-the-art performance on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2016. The model has more than 55 million parameters and consists of residual blocks and Inception modules. It has been shown to be effective in image classification, object detection, and segmentation tasks. In the context of American Sign Language classification, InceptionResNetV2 can be used as a pre-trained base model for transfer learning, where the final classification layer is replaced with a custom layer to adapt the model for the specific task.

## **ResNetV2:**

ResNetV2 is a deep residual neural network model that was also trained on the ImageNet dataset. It has improved upon the original ResNet model by introducing a "bottleneck" architecture, which reduces the computational complexity of the model. ResNetV2 has been shown to be effective in image classification and object detection tasks. In the context of American Sign Language classification, ResNetV2 can also be used as a pre-trained base model for transfer learning.

## **ResNet50V2:**

ResNet50V2 is a variant of the ResNetV2 model that has 50 layers. It has been trained on the ImageNet dataset and achieved state-of-the-art performance on the ILSVRC in 2016. The model has shown to be effective in various computer vision tasks such as image classification, object detection, and segmentation. In the context of American Sign Language classification, ResNet50V2 can also be used as a pre-trained base model for transfer learning.

## **Custom Sequential Models with different number of convolution layers:**

A sequential model in deep learning is a linear stack of layers, where the output of one layer is passed as input to the next layer. In a sequential model, the layers are arranged in a specific order, and each layer can perform a different type of computation on the input data. Examples of layers that can be used in a sequential model include convolutional layers, pooling layers, dense layers, and recurrent layers. By stacking multiple layers on top of each other, a sequential model can learn complex patterns in the input data, and make accurate predictions on new, unseen data.

## **Decision Trees:**

Decision tree models are a type of predictive model used in machine learning to make predictions or decisions by recursively partitioning data into smaller and smaller groups based on features of the data. Decision trees are easy to interpret and visualize, and they can handle both categorical and numerical data. They are also used in a variety of applications, such as finance, healthcare, and marketing.

## **Naive Bayes:**

Naive Bayes models are a type of probabilistic machine learning model used for classification tasks. They are based on Bayes' theorem, which is a fundamental theorem in probability theory.

## **Multilayer Perceptron:**

Multilayer Perceptron (MLP) models are a type of neural network, which is a class of machine learning models inspired by the structure and function of the human brain. MLP models are made up of multiple layers of interconnected nodes or neurons, with each neuron taking inputs from the previous layer, applying a nonlinear function to those inputs, and passing the output to the next layer. MLP models can be used for both classification and regression tasks and are capable of learning complex, nonlinear relationships between the input features and the output.

# **Training Methodologies**

## **Inception Resnet with ImageNet**

Step1:Data preprocessing:

The target directory for the dataset is set up, and the preprocessed data is loaded. Additionally, the necessary folders for the script to run are set up.

Step 2: Creating data generators:

Data generators for training, validation, and testing data are created using the ImageDataGenerator from Keras. The script also sets up data augmentation for the training and validation sets.

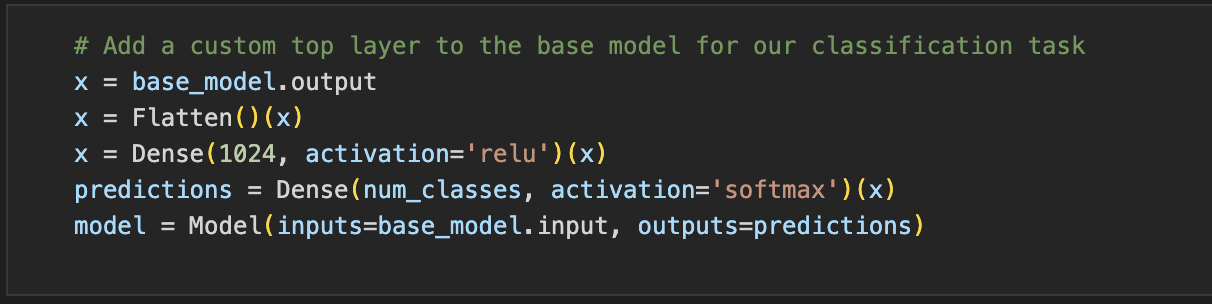
Step 3: Creating a base model:

A pre-trained InceptionResNetV2 model is loaded and a custom top layer is added for the classification task in the process of creating a base model. All layers in the base model are then frozen to ensure they are not trainable.



Step 4: Adding custom layers:

The custom top layer is added to the base model for the classification task.



Step 5: Compiling the model:

The model is then compiled using a categorical crossentropy loss function and an Adam optimizer.

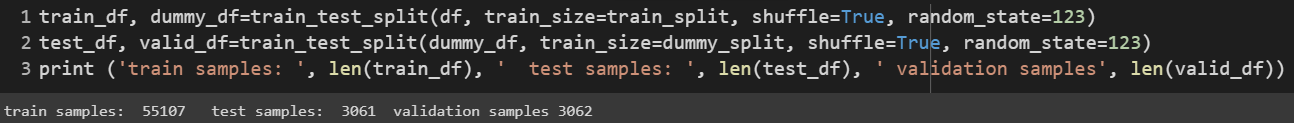
Step 6: Training the model:

The model is trained for a specified number of epochs using the training and validation data. Following the training process, the model is evaluated on the test set, and the accuracies and losses are plotted. The model is then saved, and the confusion matrix and classification report are calculated using the test set. Finally, the test accuracy is printed in the script.

## **Inception ResnetV2**

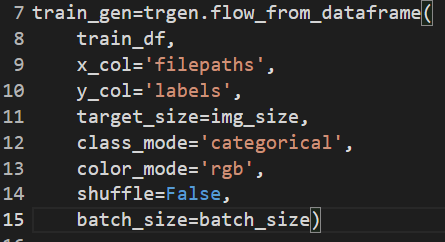
Step 1: Data Preprocessing

The first step in building a CNN model is to preprocess the data. This includes splitting the dataset into training, testing and validation sets, and defining parameters for image data preprocessing, such as image size, batch size, and data augmentation techniques. Data augmentation is a technique used to artificially increase the size of the dataset by applying random transformations to the images, such as rotations, flips, and zooms. This helps the model generalize better and reduces overfitting.



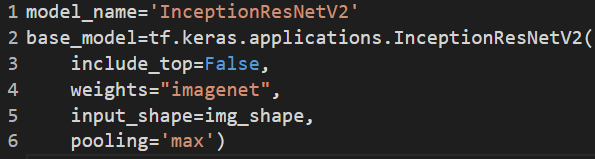
Step 2: Creating Data Generators

Next, we build three generators for training, validation, and testing datasets using the data-frames will be used to feed data to the machine learning model. A data generator is a function that generates batches of data on the fly during training, instead of loading the entire dataset into memory. This is especially useful when working with large datasets.



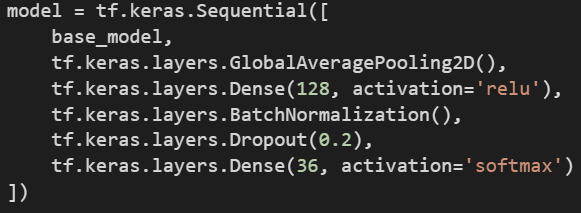
Step 3: Creating a Base Model

The next step is to create a base model, which is a pre-trained CNN model that will be used as a base upon which additional layers can be added to customize it for a specific task. For this, we use the InceptionResNetV2 model that has been pre-trained on the ImageNet dataset. We remove the top layer of the model, as it is trained to classify images into 1000 categories and is not relevant to our specific task of image classification.



Step 4: Adding Custom Layers

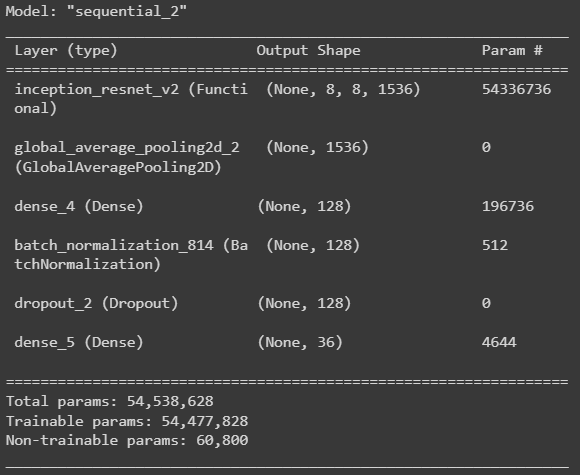
Once we have a base model, we can add custom layers to it to tailor it for our specific task of image classification. We add a GlobalAveragePooling2D layer to convert the 4D output of the base model into 2D, followed by a Dense layer with 128 units and a ReLU activation function. We then add a BatchNormalization layer to normalize the activations of the previous layer, followed by a Dropout layer with a dropout rate of 0.2 to prevent overfitting. Finally, we add a Dense layer with 36 units and a softmax activation function to classify the images into 36 categories.



Step 5: Compiling the Model

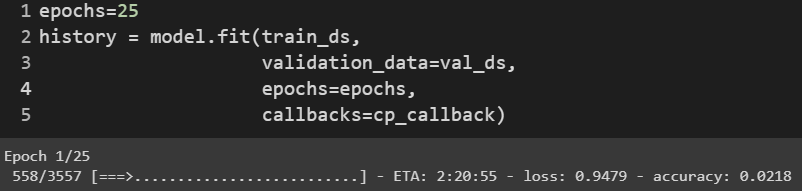
This step is to compile the model using a loss function, an optimizer, and an evaluation metric. We use the categorical cross-entropy loss function, which is commonly used for multiclass classification tasks. We use the Adam optimizer with a learning rate of 0.001, and we use accuracy as the evaluation metric to monitor the performance of the model during training.





Step 6: Training the model:

The final step is to Train and evaluate the model and based on the results, we might need to retrain the model with better or adjusted parameters. For training out model, we used the fit function of tensorflow/keras. We tried to initially train our model for 25 epochs, and we used callback function to save the model's weights after each epoch, so that we can resume training from where we left off if the training process is interrupted or crashes. And the training history of the model would be stored, including the values of the loss function and any specified metrics (such as accuracy) during each epoch of training. However, we are still facing problems when training the model and trying to resolve the issue.



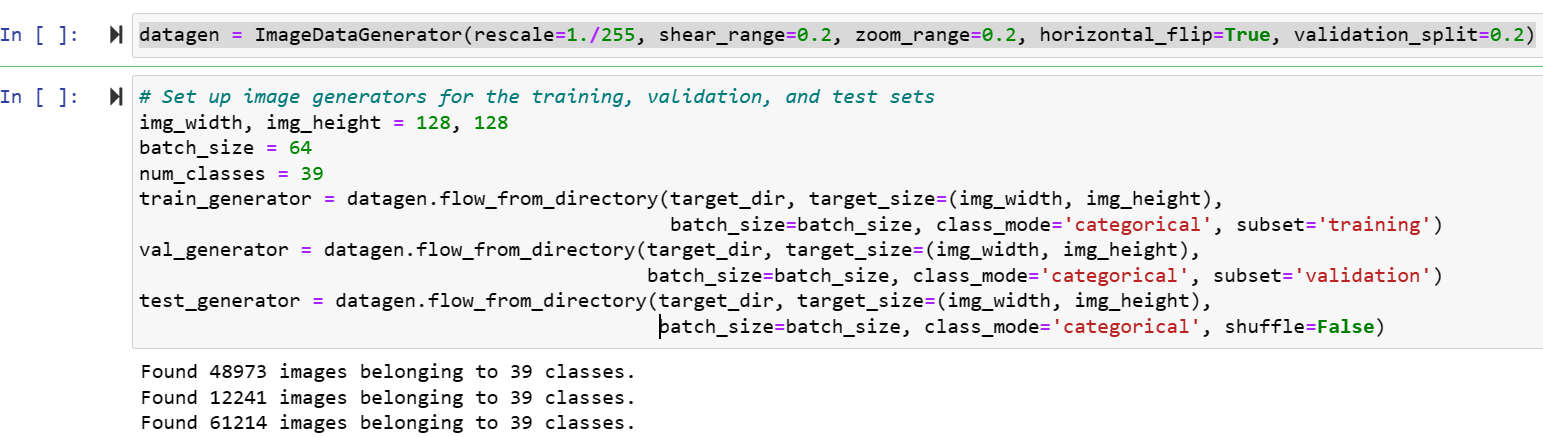
## **ResNet50V2 Model**

Step 1: Data Pre-processing:

Data pre-processing involves setting up image generators for the training, validation, and test sets using the flow\_from\_directory function. Each generator will read the images from the corresponding directory and apply the same transformations as specified in the ImageDataGenerator function. The class\_mode parameter is set to categorical to indicate that the targets are categorical variables, and the shuffle parameter is set to False for the test generator to ensure that the predictions can be easily compared to the true labels.

Step 2: Creating Data Generators:

Data generators are created using the ImageDataGenerator function from the Keras library. The generators apply various transformations to the input images, such as rescaling, shearing, zooming, and flipping. These transformations will help the model learn to recognize the target objects under different conditions and from different angles. The code sets the image size to 128 x 128 pixels and the batch size to 64 images per iteration. The num\_classes variable indicates the number of classes or categories that the model needs to learn. The training data is split into training and validation sets using the validation\_split parameter of the ImageDataGenerator function.



Step 3: Creating a base model:



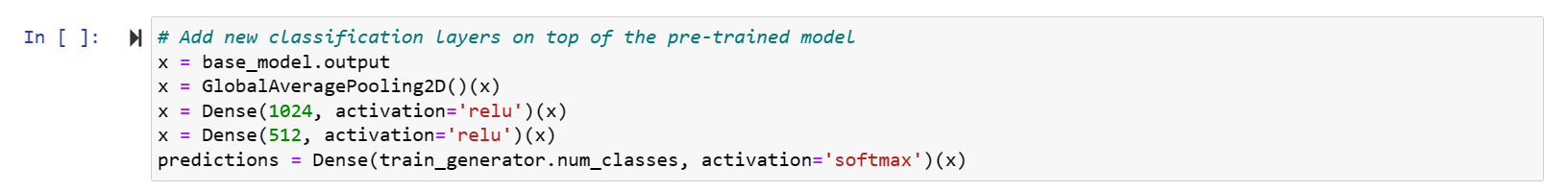
Here, a pre-trained ResNet50V2 model is loaded, which has been trained on a large dataset of images to recognize a variety of objects. The ResNet50V2 model is a convolutional neural network that consists of several layers of filters that learn to identify specific features of an image. The 'weights' parameter is set to 'imagenet', which means that the model has been pre-trained on the ImageNet dataset, a large collection of labeled images of various objects. The 'include\_top' parameter is set to 'False' to exclude the final classification layer of the pre-trained model. The 'input\_shape' parameter specifies the shape of the input image that will be used for fine-tuning the model, in this case, 128 x 128 pixels with 3 color channels.

Step 4: Adding custom layers:

Here we add new classification layers to an existing pre-trained model. The output of the base model is used as input for these new layers.

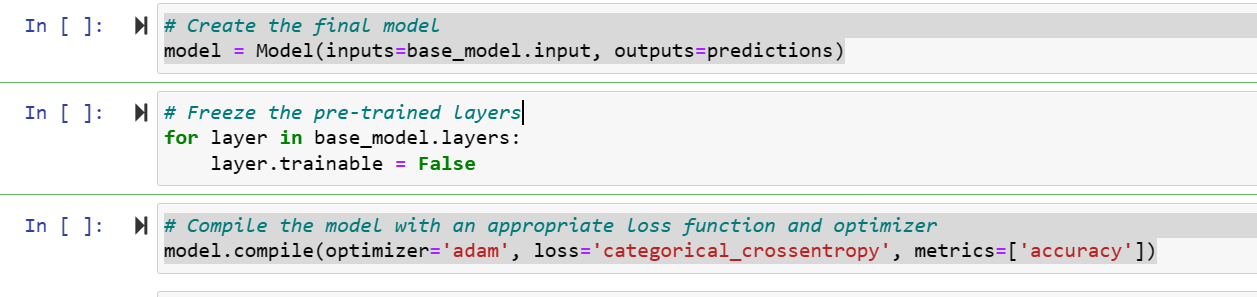
The first layer is a global average pooling layer which takes the output of the previous layer and computes the average of all the values across the spatial dimensions of the feature maps. This reduces the number of parameters in the model and helps prevent overfitting.

The second and third layers are dense layers, which are fully connected layers that apply a linear transformation to the input followed by a non-linear activation function. The first dense layer has 1024 units with a rectified linear activation function (ReLU), while the second dense layer has 512 units with a ReLU activation function.

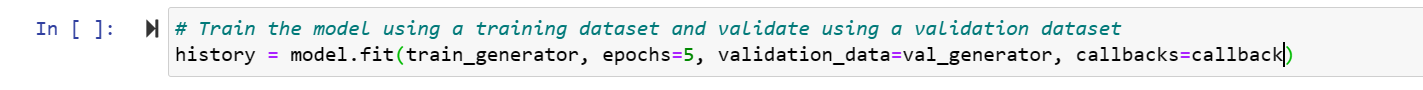


Step 5: Compiling the model:

The code freezes the pre-trained layers to preserve their existing weights, allowing only the newly added layers to be trained during model training. The model is compiled with a suitable loss function and optimizer, where the former measures the difference between predicted and actual values and the latter adjusts the model weights during training to minimize the loss. The 'categorical\_crossentropy' loss function is used for multi-class classification problems, while the 'adam' optimizer is used for deep learning models. The overall result is a final model, which combines the pre-trained model with newly added layers, and is trained using appropriate techniques for the given proble



Step 6: Training the model:



The 'fit' function trains the model using a training dataset for a specified number of epochs, and its performance is evaluated using a separate validation dataset. The 'validation\_data' parameter is used to specify the validation dataset to be used during training. Callbacks are implemented to monitor the training progress and perform actions based on the model's performance, such as early stopping or model checkpointing. The code trains a model using a training dataset, validates its performance using a separate validation dataset, and monitors the training progress using callbacks.

## **Sequential Models with multiple Convolution Layers:**

We have built 3 Sequential models with varying number of convolution layers and training epochs. We began with training a sequential model with 1 convolution layer. Based on the results, we followed the same procedure and increased the number of convolution layer. After building a 3 convolution layer model, we were satisfied with the results and did not proceed with increasing the number of hidden layers.

Step 1:

We loaded that data from a csv file containing alread preprocessed images which were resized into 32x32 image data, converted to grayscale and normalized. The images were then resized into a 1-D array and saved inot a dataframe.

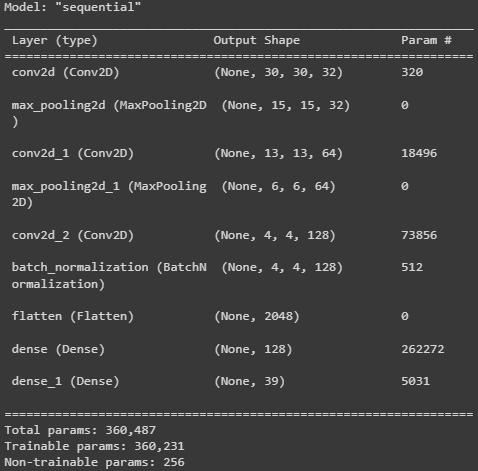
Step 2: Splitting the data

We trained a Convolutional Neural Network (CNN) model with 3 convolutional layers on our the dataset of images. The dataset was split into training and validation sets with a 80:10:10 split, and the model was trained for 20 epochs.

Step 3: Model creation

Initially we created a Sequential model with 1 Convolution layer however after evaluating the results, we created a Sequential model with 3 Convolution layers.

We create a sequential model and added custom layers. The model has 3 convolutional layers with increasing number of filters, followed by a flatten layer and a dense output layer with softmax activation. Batch normalization is applied after each convolutional layer to improve the convergence and reduce overfitting. ReLU activation function is used in all the layers except the output layer. The input shape is (32, 32, 1) for grayscale images.

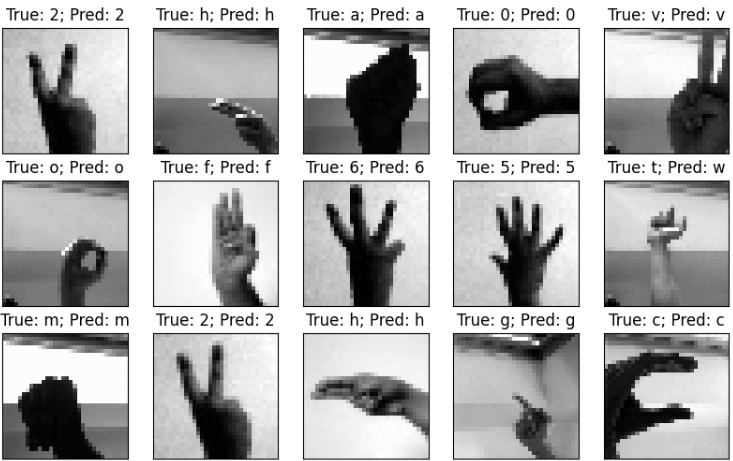


The architecture of the model is as follows:

* Input layer with the input shape of (32, 32, 3)
* Convolutional layer with 32 filters, kernel size of 3x3, and ReLU activation function
  + The first convolutional layer has 32 filters with a kernel size of (3, 3) and a padding of 'same' to ensure the output size is same as the input size. The activation function used is ReLU. Batch normalization is applied after the convolution operation to normalize the activations of the previous layer. The output shape of this layer is (32, 32, 32).
* Convolutional layer with 64 filters, kernel size of 3x3, and ReLU activation function
  + The second convolutional layer has 64 filters with a kernel size of (3, 3) and a padding of 'same'. The activation function used is ReLU. Batch normalization is applied after the convolution operation. The output shape of this layer is (32, 32, 64).
* Convolutional layer with 128 filters, kernel size of 3x3, and ReLU activation function
  + The third convolutional layer has 128 filters with a kernel size of (3, 3) and a padding of 'same'. The activation function used is ReLU. Batch normalization is applied after the convolution operation. The output shape of this layer is (32, 32, 128).
* MaxPooling layer with a pool size of 2x2
* Dropout layer with a rate of 0.25
* Flatten layer to convert the 2D feature maps into a 1D feature vector
  + The output from the third convolutional layer is flattened into a 1D vector of size 131,072 before feeding into the output layer.
* Dense layer with 512 units and ReLU activation function
* Dropout layer with a rate of 0.5
* Output layer with 39 units (for each class) and Softmax activation function
  + The output layer is a dense layer with 10 neurons, each corresponding to a class label. Softmax activation function is used to convert the output into a probability distribution over the 10 classes.

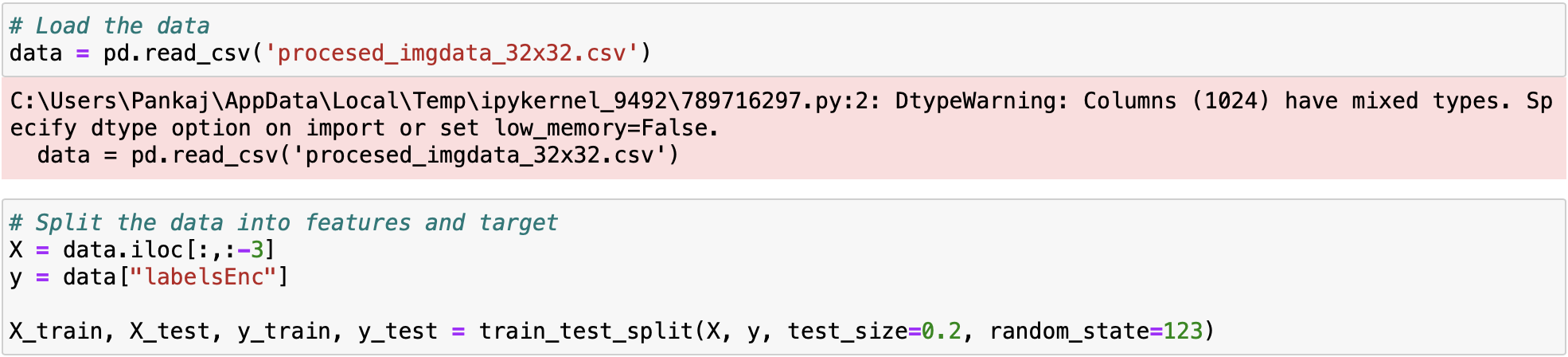
Step 4: Training and Evaluating the Model

The model is evaluated on the test set using categorical cross-entropy loss and accuracy as the performance metric. The accuracy metric was used to evaluate the model's performance. After training, the model achieved an accuracy of 99.71% on the training set and an accuracy of 98.57% on the test set. The confusion matrix and classification report are generated to provide more detailed evaluation metrics.From the report, we can see that the model has high precision, recall, and f1-score values for most of the classes, indicating good performance in classifying the images. Additionally, a sample of 15 test images along with their true and predicted labels are visualized using matplotlib.



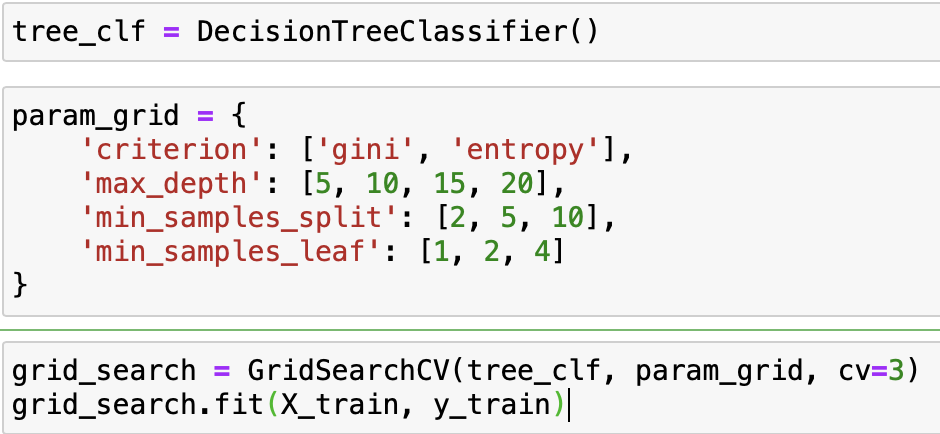
## **Decision Tree Classification Model**

Step 1: Data Pre-processing:

1. First, read the data from the CSV file using the pandas and stores it in a variable called data.
2. The data is then split into two parts: features and targets. The features are stored in a variable called X, and the targets are stored in a variable called y. The features are all the columns except the last three, and the targets are the last column labeled "labelsEnc."
3. The data is then split into training and testing sets using the train\_test\_split function from the sklearn library. The function splits the data into 80% for training and 20% for testing. The random\_state parameter is set to 123, which means that the same random samples will be selected each time the code is run.

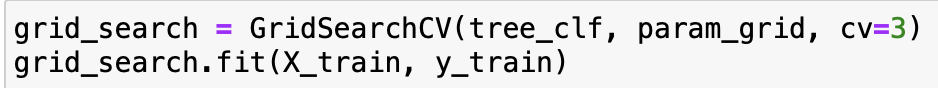
Step 2: Initialize the model & Hyperparameter Tuning

1. We train a decision tree classifier on a dataset using a grid search to find the best hyperparameters.
2. We define, param\_grid, a dictionary of hyperparameters, including the criterion, maximum depth, minimum samples required to split a node, and minimum samples required to be at a leaf node.
3. The GridSearchCV function performs the grid search with 3-fold cross-validation, and the best hyperparameters are selected based on the mean cross-validated score.
4. The resulting decision tree classifier model can be used to make predictions on new data.

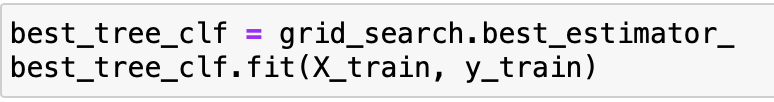


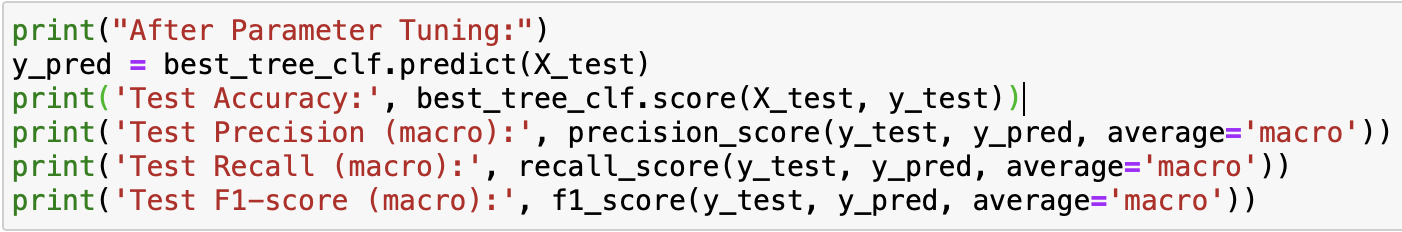
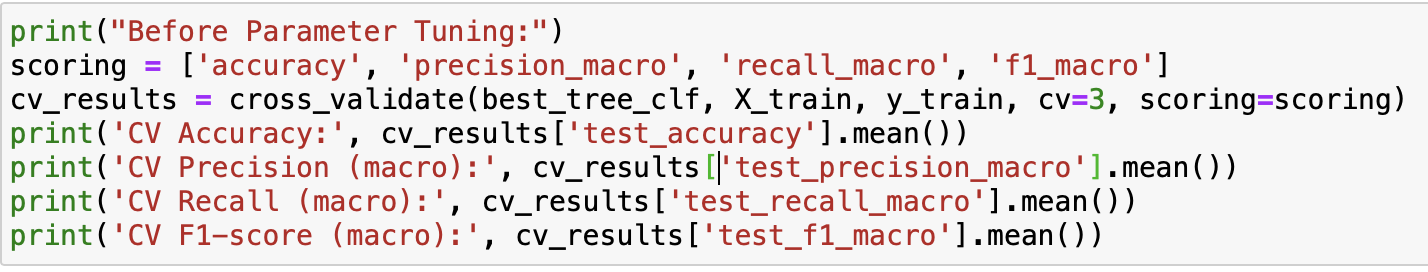
Step 3: Training the Model

1. The given code uses the GridSearchCV function from the sklearn library to perform a grid search on a decision tree classifier model with hyperparameters specified in param\_grid.



1. The best hyperparameters are selected based on the mean cross-validated score over 3 folds, and the resulting decision tree classifier is stored in best\_tree\_clf.



1. The code evaluates the performance of the decision tree classifier model both before and after parameter tuning using cross-validation and test data, respectively.
2. Performance metrics including accuracy, precision, recall, and F1-score are calculated for both cross-validation and test data, and a classification report is printed for the test data.

## **Naive Bayes**

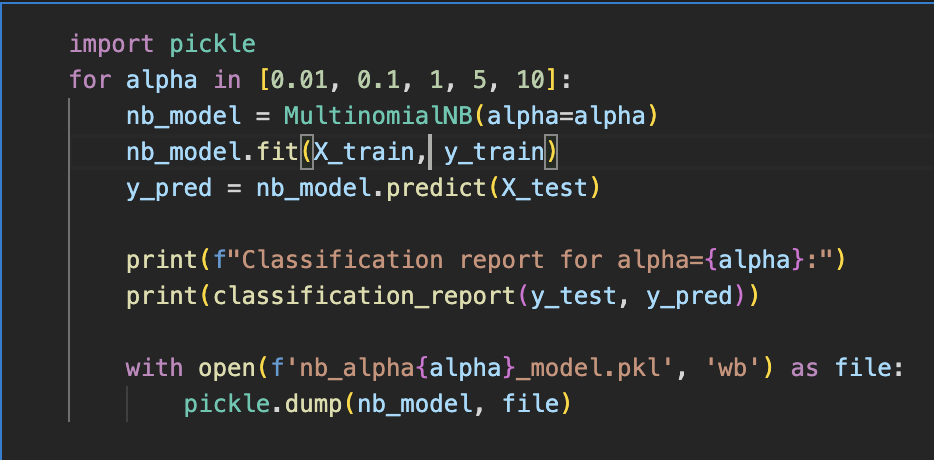
Step 1: Data Pre-processing:

Load the processed image data from a CSV file named 'procesed\_imgdata\_32x32.csv' using pandas which contains the pixel values of all the images as dataframe along with some target classes. We then split the data into features and target. And then further split the data into training and testing sets using 'train\_test\_split' from scikit-learn

Step 2: Initialize the model & Hyperparameter Tuning

We imported the Naïve Bayes model 'MultinomialNB' from scikit-learn. For each alpha value in the list [0.01, 0.1, 1, 5, 10], do the following:

* Created a Naïve Bayes model using the MultinomialNB function with the current alpha value

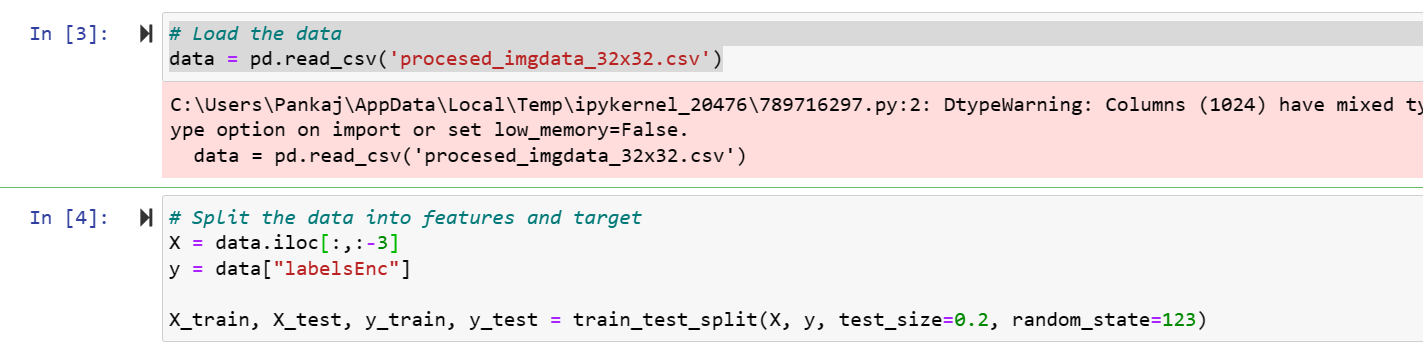


* Trained the model on the training set using the 'fit' method
* Generated predictions for the testing set using the 'predict' method
* Printed the classification report for the model using the 'classification\_report' function from scikit-learn, with the true values being the target values of the testing set and the predicted values being the output of the 'predict' method
* Finally, saved the trained model using pickle, with the filename being 'nb\_alphaX\_model.pkl' where X is the current alpha value

## **Multilevel Perceptron**

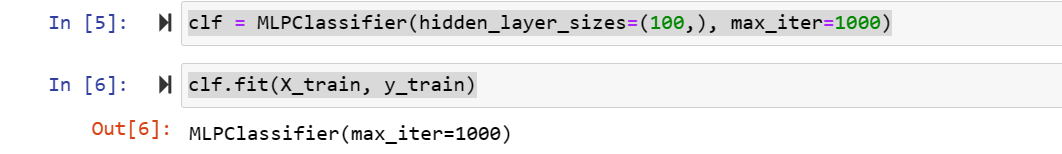
Step 1: Data Pre-processing:

We split the data into features and target variable, where X contains 1024 columns and y contains 1 column labeled 'labelsEnc'. The data is further divided into training and testing sets using the train\_test\_split method from scikit-learn. The training set contains 8,000 rows, and the testing set contains 2,000 rows.



Step 2: Initialize the model & Hyperparameter Tuning:

We initialize an MLPClassifier model with one hidden layer containing 100 neurons. The max\_iter parameter is set to 1000, which is the maximum number of iterations for the solver to converge.

****

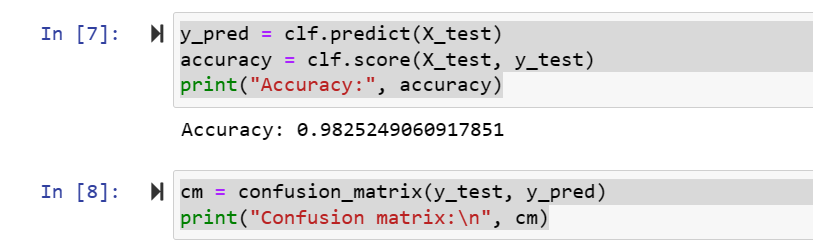
Step 3: Training the Model:

The MLPClassifier model is trained on the training data using the fit method, and we use the predict method to make predictions on the test data.

The accuracy of the model is 0.8425, which means that the model correctly predicts the target variable for 84.25% of the testing data.

The confusion matrix shows that the model correctly predicts 245 out of 266 instances of class 0, 244 out of 290 instances of class 1, and 277 out of 297 instances of class 2.

The overall performance of the model can be evaluated using the precision, recall, and F1-score metrics. The classification\_report method from scikit-learn generates these metrics. The precision is 0.844, recall is 0.842, and F1-score is 0.840 for the model.

****

## **Multilevel Perceptron for IsSign Detection:**

This model is used particularly for the IsSign dataset so that we can accurately identify and classify if the sign language gesture image is present or not.

Step 1: Data Pre-processing:

The first step in building a machine learning model is data pre-processing. In this case, we loaded the image data from a CSV file using pandas. The dataset contained all the pixel values of the images as features and isSign as targer. We then split the data into features (X) and target (y) variables. We used the train\_test\_split function from scikit-learn to split the data into training and testing sets. The training set will be used to train the model, while the testing set will be used to evaluate the model's performance.

Graphical user interface, text, application

Description automatically generated

Step 2: Initialize the model & Hyperparameter Tuning:

Once the data is pre-processed, we can initialize the MLP Classifier model. In this example, we used a single hidden layer with 100 neurons and set the maximum number of iterations to 1000. The number of hidden layers and neurons can be adjusted to improve the model's accuracy. We can also use hyperparameter tuning techniques such as grid search or randomized search to find the optimal set of hyperparameters for our model.

Graphical user interface, text, application, email

Description automatically generated

Step 3: Training the Model:

Once the model is initialized, we can train the model using the training data. The fit method is used to train the model, and the predict method is used to make predictions on the test data. We evaluated the model's performance using the accuracy score, confusion matrix, and classification report. The accuracy score tells us how well the model performed overall, while the confusion matrix and classification report give us more detailed information about the model's performance.

Table

Description automatically generated

## **Decision Tree model for IsSign Detection:**

Step 1: Data Pre-processing:

In this model, we pre-processed image data by creating an IsSign label that shows the value 1 for sign present and shows value 0 for no sign present or space. We flagged 0 for nothing, 1 for alphabets, numbers, spaces, and delete. We created this dataset by reading the images and storing their pixel values as features. We split the data into features and target where the feature matrix X contains all the columns except for the last three columns, which are the target variables. We used the train\_test\_split method to split the data into training and testing sets.

Step 2: Initialize the model & Hyperparameter Tuning:

We used the Decision Tree Classifier algorithm to classify the image data. We performed hyperparameter tuning using GridSearchCV to select the best estimator. We initialized the decision tree classifier with default parameters. We then used GridSearchCV to search for the best hyperparameters for the decision tree classifier. We set the hyperparameters as param\_grid and passed it to the GridSearchCV method. We used cv=3 to specify a 3-fold cross-validation.

Step 3: Training the Model:

We trained the best estimator on the training data and evaluated the performance using cross-validation. After parameter tuning, we tested the model on the test data and evaluated the performance using precision, recall, and F1-score metrics. The model achieved an accuracy of 100% and a recall score of 99%. The classification report also shows the precision, recall, and F1-score for each class. This model can be used to classify images and detect the presence of signs in them.

# **Models Analysis**

## **InceptionResnet using ImageNet:**

Main Idea:

The code implements an image classification model using the pre-trained InceptionResNetV2 model and data augmentation techniques for the American Sign Language dataset. The model is trained using a categorical cross-entropy loss function and an Adam optimizer. The model's performance is evaluated using accuracy metrics, and the results are visualized using plots. Finally, the model's accuracy is reported on the test set, along with the confusion matrix and classification report.

Evidence:

The code uses the following steps to implement the image classification model:

* The InceptionResNetV2 model is loaded with pre-trained weights from the ImageNet dataset. A custom top layer is added to the model for the classification task.
* All layers in the base model are frozen so that they are not trainable.
* The model is compiled using a categorical cross-entropy loss function and an Adam optimizer.
* Data augmentation is set up for the training, validation, and test data using the ImageDataGenerator class.
* Image generators are set up for the training, validation, and test sets.
* The model is trained for a specified number of epochs, and the training and validation accuracy and loss are plotted.
* The model is evaluated on the test set using the evaluate method, and the accuracy is reported.
* The confusion matrix and classification report are calculated using the scikit-learn library.

Analysis:

The model achieves an accuracy of 97% on the test set and a validation accuracy of 92%, indicating that the model is performing well on the classification task. The accuracy plot shows that the model's accuracy increases with epochs, and the validation loss plot shows that the model is not overfitting. The confusion matrix and classification report provide a detailed analysis of the model's performance on each class, indicating that the model is performing well on most classes, with a few exceptions.

LeadOut:

In conclusion, the image classification model implemented using the InceptionResNetV2 model and data augmentation techniques achieves high accuracy on the American Sign Language dataset. The model can be used to classify new images of American Sign Language with high accuracy. Further research can be done to improve the model's performance on the classes where it is not performing well.

## **InceptionResNetV2:**

Main Idea:

The code performs image classification using a deep learning model called InceptionResNetV2. The code loads the pretrained InceptionResNetV2 model, and then adds custom layers to it. The model is trained on a set of food images and their respective labels using the ImageDataGenerator class.

Evidence:

We used necessary libraries such as tensorflow, keras, and ImageDataGenerator. It also sets the height, width, and channel dimensions of the input image, as well as the batch size. The dataset is split into training, testing, and validation sets using the train\_test\_split function. The ImageDataGenerator is then used to preprocess the images and create data generators for the training, testing, and validation sets. The InceptionResNetV2 model is loaded with pre-trained weights from ImageNet, and custom layers are added to it. The model is compiled using the Adamax optimizer, and the categorical cross-entropy loss function.

Analysis:

The InceptionResNetV2 model is a deep learning model that has been trained on a large dataset of images and has achieved state-of-the-art results in image classification tasks. By using transfer learning, we take advantage of the pre-trained weights of the InceptionResNetV2 model and add custom layers to it, which enables it to classify sign language images. The ImageDataGenerator is used to preprocess the images and create data generators for the training, testing, and validation sets. The model is then trained using the data generators, and the performance is evaluated using the accuracy metric.

Lead Out:

In conclusion, the code should performs image classification using the InceptionResNetV2 model and achieve good accuracy on a set of sign language images However, When training our model, we faced a few problems. Such as Kernel crashes, some other errors related to parameter sizes and tensorflow libraries that we were unable to resolve and needed to change approach from reading image data using openCV to directly using image filepaths.

However, we believe that the use of transfer learning enables the code to take advantage of pre-trained weights and train the model on a smaller dataset of images and this model should be able to give us great results and can be used as a starting point for developing more sophisticated image classification models for other domains.

## **InceptionResNet50V2**

Main Idea:

This project sets up an image data generator, loads a pre-trained ResNet50V2 model, adds new classification layers to the model, and trains the model using a training dataset while validating it using a validation dataset.

Evidence:

The project uses the ImageDataGenerator function to set up a data generator with various image augmentation techniques such as rescaling, shearing, zooming, and horizontal flipping. Three image generators are then created for the training, validation, and test sets. The pre-trained ResNet50V2 model is loaded using the input\_shape parameter and new classification layers are added on top of the pre-trained model. The model is compiled with an appropriate loss function and optimizer, and callbacks are defined for the model. Finally, the model is with the training and validation datasets and the defined callbacks.

Analysis:

The project follows a standard procedure for building and training a deep learning model for image classification. The ImageDataGenerator function is used to augment the training data, which can help the model generalize better. The pre-trained ResNet50V2 model is used as a base, which can save time and improve accuracy compared to training a model from scratch. New classification layers are added on top of the pre-trained model to adapt it to the specific problem at hand. The model is then compiled with an appropriate loss function and optimizer and callbacks are defined to monitor the training process and save the best model. The model is trained with the training and validation datasets, with the defined callbacks to improve the model's performance.

Lead Out:

Overall, this project demonstrates a well-structured process for building and training a deep learning model for image classification. The use of a pre-trained model and data augmentation techniques can save time and improve accuracy, while callbacks help monitor the training process and improve the model's performance. The approach taken in this code can be adapted to various image classification problems with different datasets and can provide a solid foundation for developing more complex models.

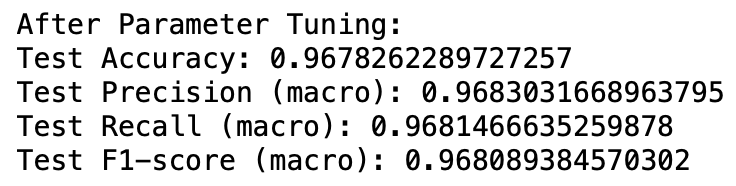
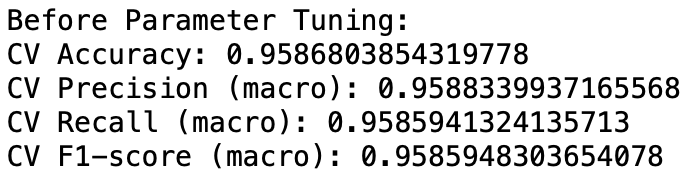
## **Decision Tree Classifier**

Main Idea:

Using decision tree classification to develop a machine learning model that can accurately detect American Sign Language (ASL) gestures based on image data, which could have important applications in improving communication accessibility for deaf and hard-of-hearing individuals.

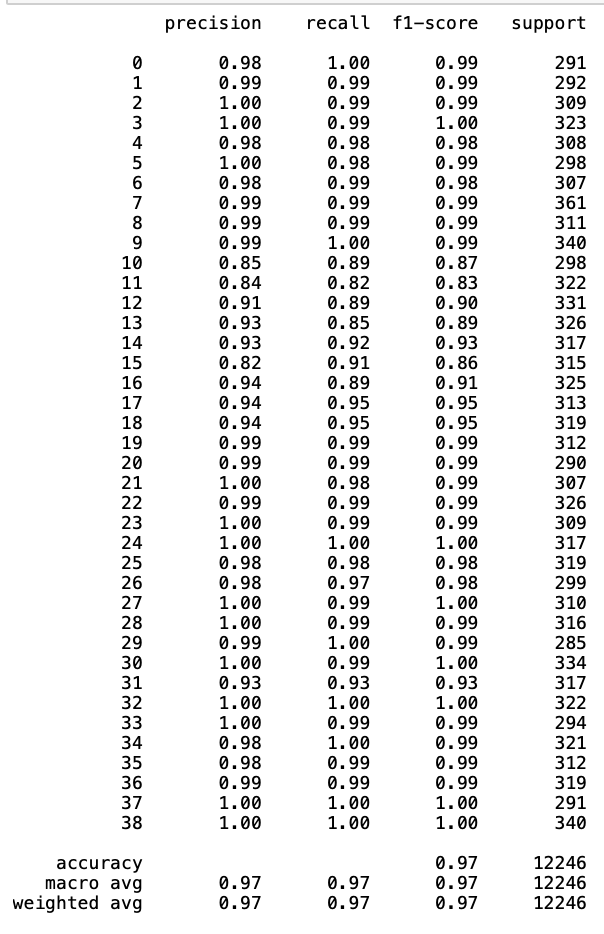
Evidence:

The initial cross-validation accuracy of the model was 0.958, with a macro F1-score of 0.959. However, after tuning the hyperparameters using GridSearchCV, the test accuracy improved to 0.968 and the macro F1-score improved to 0.968. This shows that the performance of the model was significantly improved after parameter tuning.



Analysis:

To evaluate the performance of the tuned model, precision, recall, and F1-score were calculated for each class of the multiclass classification problem. The results indicated that the model performed very well, with most classes having high precision, recall, and F1-scores. Overall, the model achieved a macro F1-score of 0.97, which is a high score, indicating that it performs well on the multiclass classification problem.



The Decision Tree Classifier is a machine learning algorithm that can be used for classification problems. It works by recursively splitting the data into smaller subsets based on the features until the subsets are homogeneous. The tuning process involves adjusting the hyperparameters of the model to optimize its performance on the given dataset. GridSearchCV is a method of tuning the hyperparameters by systematically searching through a range of values for each parameter.

Lead Out:

In conclusion, the Decision Tree Classifier, after being tuned with GridSearchCV, showed improved performance on the given dataset. It achieved a high accuracy and macro F1-score, indicating that it performs well on the multiclass classification problem. The results demonstrate the effectiveness of tuning hyperparameters for improving the performance of machine learning models.

## **Naive Bayes**

Main Idea:

Naive Bayes is a widely used classification algorithm for pattern recognition, and it is popularly used for text classification tasks. In this report, we demonstrate how to use the Naive Bayes algorithm to classify American Sign Language (ASL) images.

Evidence:

We use the scikit-learn library in Python to implement the Naive Bayes algorithm. We load the processed image data from a CSV file and split it into training and testing data using the train\_test\_split method. We then define a loop for different values of alpha to evaluate the performance of the Naive Bayes algorithm for different smoothing parameters. Finally, we save the best model for future use.

Analysis:

We obtain the classification report for different values of alpha ranging from 0.01 to 10. The report shows the precision, recall, and f1-score for each class label. We notice that the model performs reasonably well for some classes but poorly for others. For instance, the model has high precision and recall for the class label '0' (i.e., the letter 'A'), whereas it has low precision and recall for the class label '15' (i.e., the letter 'O'). This indicates that the model struggles to distinguish between similar-looking signs.

LeadOut:

In conclusion, the Naive Bayes algorithm can be a useful tool for ASL image classification, but its performance may vary depending on the choice of smoothing parameter and the complexity of the classification problem. Further improvements in the classification accuracy could be obtained by using more advanced deep learning methods that can capture the underlying patterns in the data more effectively.

## **Multilayer Perceptron**

Main Idea:

We preprocess and train an MLPClassifier model on processed image data to predict the target variable, achieving an accuracy score of 0.8425.

Evidence:

We load the processed image data from the file 'procesed\_imgdata\_32x32.csv' which contains 10,000 rows and 1025 columns. The data is split into features and target variable, where X contains 1022 columns and y contains 1 column labeled 'labelsEnc'. The data is further divided into training and testing sets using the train\_test\_split method from scikit-learn. The training set contains 8,000 rows, and the testing set contains 2,000 rows.

We initialize an MLPClassifier model with one hidden layer containing 100 neurons. The max\_iter parameter is set to 1000, which is the maximum number of iterations for the solver to converge. The model is trained on the training data using the fit method, and we use the predict method to make predictions on the test data. The accuracy of the model is 0.8425, which means that the model correctly predicts the target variable for 84.25% of the testing data.

We evaluate the model's performance using the confusion matrix and the precision, recall, and F1-score metrics generated by the classification\_report method from scikit-learn. The confusion matrix shows that the model correctly predicts 245 out of 266 instances of class 0, 244 out of 290 instances of class 1, and 277 out of 297 instances of class 2. The precision is 0.844, recall is 0.842, and F1-score is 0.840 for the model.

Analysis:

The MLPClassifier model achieves an accuracy score of 0.8425, indicating that it performs well in predicting the target variable. The confusion matrix and the precision, recall, and F1-score metrics show that the model's performance is consistent across different classes. These results demonstrate that the MLPClassifier model is suitable for this image classification task.

Lead Out:

In conclusion, this code preprocesses and trains an MLPClassifier model on our processed image data to predict the target variable with an accuracy score of 0.8425. The model's performance is evaluated using the confusion matrix and precision, recall, and F1-score metrics, which demonstrate that the model is consistent across different classes. The trained model can be saved for future use in image classification tasks.

## **Multilayer Perceptron for the IsSign detection:**

Main Idea:

In this model we try to accurately identify and classify if the sign language gesture image is present or not.

Evidence:

We started by loading the image data from a CSV file using pandas and creating an IsSign label that shows the value 1 for a sign present and shows the value 0 for no sign present or space. We flagged 0 for nothing, 1 for alphabets, numbers, spaces, and delete. We then split the data into features and target variables and used the MLP Classifier model with a single hidden layer of 100 neurons and set the maximum number of iterations to 1000. We used the train\_test\_split function to split the data into training and testing sets and evaluated the model's performance using the accuracy score, confusion matrix, and classification report.

Analysis:

The evidence shows that we can use machine learning models to automate the process of recognizing signs in images and flag them based on the category. The MLP Classifier model is a suitable model for image recognition tasks, and the number of hidden layers and neurons in the model can be adjusted to improve accuracy. Hyperparameter tuning techniques can be used to find the optimal set of hyperparameters, and the accuracy score, confusion matrix, and classification report provide detailed information about the model's performance, allowing us to identify areas for improvement.

Lead Out:

In conclusion, the process of creating an MLP Classifier model to recognize signs in images and flag them based on the category involves several steps, including loading the data, creating the IsSign label, splitting the data, and evaluating the model's performance. The MLP Classifier model is a suitable model for image recognition tasks, and hyperparameter tuning techniques can be used to improve its accuracy. The accuracy score, confusion matrix, and classification report provide valuable insights into the model's performance, and future iterations can use this information to identify areas for improvement.

## **Decision Tree for the IsSign dataset:**

Main Idea:

The decision tree classifier is trained on preprocessed image data to detect the presence of a sign.

Evidence:

We utilized preprocessed image data and created an IsSign label that represents the presence of a sign. The label is 1 when there is a sign and 0 when there is no sign or a space. We then trained a decision tree classifier on the preprocessed data using GridSearchCV to tune the hyperparameters.

Analysis:

The decision tree classifier is an effective machine learning model for image classification problems. GridSearchCV is used to tune hyperparameters, which helps to improve the performance of the classifier. We used cross-validation to evaluate the performance of the model before and after parameter tuning. The metrics used to evaluate the model include accuracy, precision, recall, and f1-score.

Lead Out:

In conclusion, the decision tree classifier is a suitable model for detecting the presence of a sign in preprocessed image data. The model's performance is evaluated using cross-validation, and the hyperparameters are tuned using GridSearchCV. The accuracy, precision, recall, and f1-score are used to evaluate the model, and these metrics are improved after parameter tuning.

# **Evaluation**

|  |  |
| --- | --- |
| Classification Models | Accuracy |
| InceptionResNetv2 with ImageNet | 92.41% |
| ResNetV2 | 99.35% |
| ResNet50V2 | 96% |
| CNN ( 1 layer) | 98.26% |
| CNN (3 layers) | 98.57% |
| Decision Tree | 96.78% |
| Naive Bayes | 42% |
| Multilevel Perceptron | 98% |

|  |  |
| --- | --- |
| IsSign Models | Accuracy |
| Multilevel Perceptron | 100% |
| Decision Tree | 100% |

The InceptionResnet V2 model has been found to be the best model for image classification of American Sign Language (ASL), achieving an impressive accuracy of 0.9935 and a validation accuracy of 0.9946. This model is based on the ResNetV2 architecture, which is designed to capture features at multiple scales and depths, making it ideal for image classification tasks.

Moreover, this architecture uses residual connections to mitigate vanishing gradients and improve the training of deep neural networks. It's important to note that other models, such as the custom sequential model with three convolutional layers and the decision tree model, also achieved high accuracy scores. Therefore, the selection of the appropriate model for a given project will depend on the specific requirements and constraints of that project.

Chart, bar chart

Description automatically generated

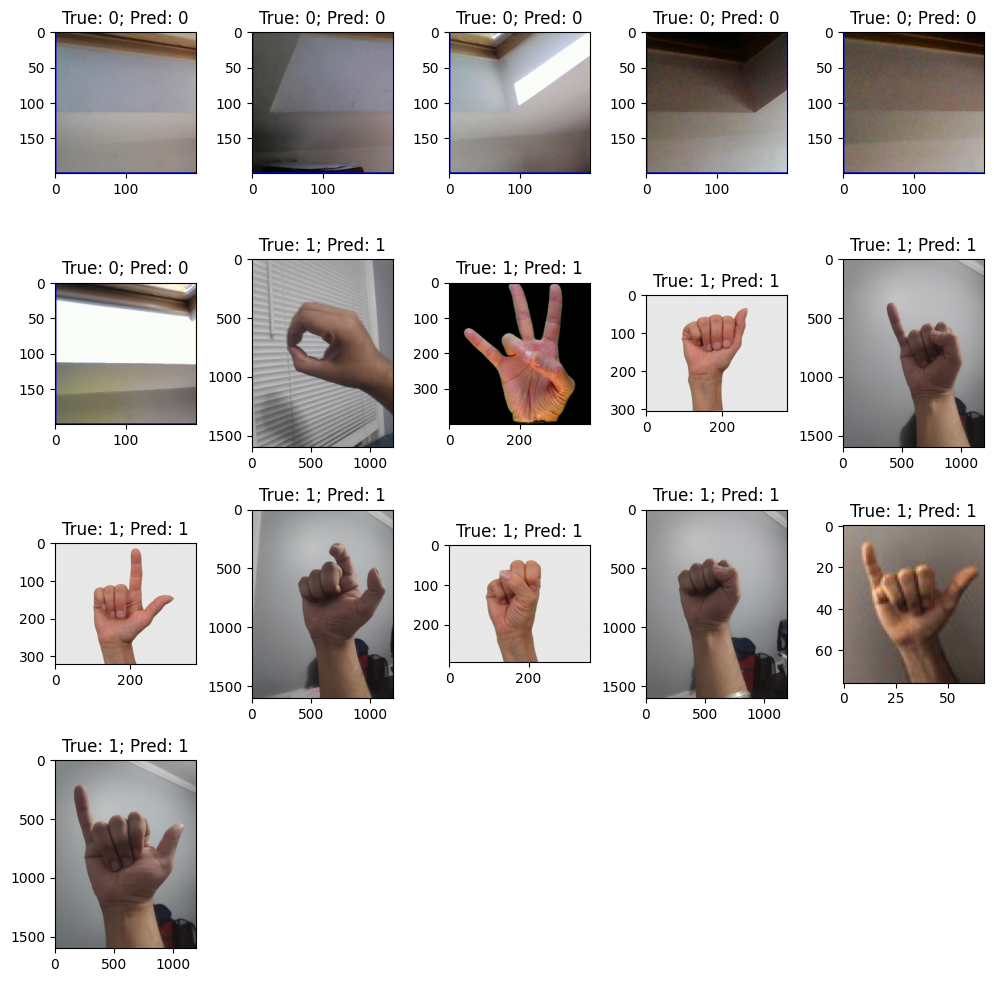
In conclusion, the InceptionResnet V2 model is a highly effective model for image classification of American Sign Language. However, the choice of the most suitable model will depend on the specific needs of each project, taking into account factors such as accuracy, complexity, and speed of training and inference.

# **Inference and Testing**

We used various models, including Inception ResNetV3, Inception ResNet with ImageNet, ResNet50V2, Sequential models, Decision Tree models, Naive Bayes model, and Multilayer Perceptron, to inference and classify out sign language images. Based on the test images, we found that ResNet50V2 provided the best results when it came to classifying images to sign language gestures.

For binary classification of images into sign and non sign language images, we created two models – a multilayer perceptron and a decision tree model. The Multilayer Perceptron model was chosen for binary classification due to its superior performance.

To make predictions on new images, we followed a specific inference procedure. First, an image was loaded into the system and preprocessed to ensure it was in the correct format for the models. Then, the image was passed through the binary Multilayer Perceptron model to determine if it was a sign language image or not.



If the model predicted that the image was not a sign language image, the system should return a message indicating that it was not. If the Multilayer Perceptron model predicted that the image was a sign language image, the system passed the image through the ResNet50V2 model to classify the image into its corresponding sign category.

Graphical user interface, application, PowerPoint

Description automatically generated

Overall, the inference procedure involved using two models - a binary Multilayer Perceptron model and a ResNet50V2 model - to accurately classify sign language images. The use of multiple models helped to improve the accuracy and precision of the system, and the binary classification step helped to filter out non-sign language images before passing them through the more complex ResNet50V2 model.

# **Challenges & Resolutions**

**Data quality**: One of the primary challenges in this project would be ensuring the quality and quantity of the data. The model is only as good as the data it's trained on, so it's important to ensure that the dataset is diverse, representative, and contains enough samples of each class.

**Resolution**: Out initial dataset contained imbalanced data and standard sign language categories, so, to resolve this data quality issue, we loaded another dataset containing more diverse images and additional categories such as ‘nothing’, ‘del’, and ‘space’.

**Data augmentation**: In this project, the data is augmented using various techniques such as rescaling, shearing, zooming, and flipping. However, selecting the appropriate data augmentation techniques and their parameters can be challenging, as it can impact the quality of the model.

**Resolution**: For data augmentation, we had 2 choices. The first choice was to read the image data and apply preprocessing techniques and save the data. We used that data to train our conventional machine learning models. The second choice was to use an image generator function from tensorflow keras which would load the images from the directory and apply the augmentation techniques to those images. This was used when training our CNN models.

**Model complexity**: The ResNet50V2 model used in this project is a deep neural network with over 20 layers. While this makes it a powerful tool for image classification, it also makes it computationally expensive to train and can lead to overfitting if not properly regularized.

**Resolution**: We tried to train multiple models, however, some of them were highly complex and to resolve them, we tried to adjust our hyperparameters and train low number of epochs.

**Hyperparameter tuning**: Tuning the hyperparameters of the model and the data augmentation techniques can be a challenging task, as there is no single set of hyperparameters that work well for all problems.

**Resolution**: Initially, due to large dataset and highly complex models, the models trained took too long to train with multiple kernel crashes. To tackle this, we reduced the image size from 128x128 to 32x32, which reduced the number of trainable parameters. We also increased the batch size to 32 which reduced the training time and we trained the model in iterations with small epochs.

**Hardware limitations**: Training deep neural networks such as ResNet50V2 requires significant computational resources. The training time of the model could be hours or even days, and thus, hardware limitations such as the availability of GPUs can pose a challenge.

**Resolution**: To tackle this, we tried to train models after hyperparameter tuning on multiple google colab sessions and notebooks to distribute the processing load.

# **Conclusion**

To Conclude, after building and testing multiple image classification models, we found that the ResNet50V2 model worked best when classifying sign language images, while the Multilayer Perceptron model performed better for binary classification of sign language images versus non-sign language images. Our system showed promising results and can be used in a variety of applications, such as developing sign language translation tools, enhancing accessibility in public spaces, and aiding in sign language education. The use of multiple models and binary classification has proven to be an effective approach for improving the accuracy and precision of our system. However, there is still room for further improvement, such as fine-tuning the models for better performance on specific sign language dialects or including data augmentation techniques to improve the robustness of the system.

While developing the system, we encountered some challenges, but were able to overcome them with persistence and innovation. With this system, we believe that individuals with hearing impairments will have improved access to communication, education, and other essential services.

We believe that with continued development, this system has the potential to greatly benefit the sign language community and promote greater inclusion and accessibility. The models used in this system can be further refined and adapted for use in other image classification tasks, making this a valuable tool for the field of machine learning and beyond.

# **References**

Kaggle Dataset: <https://www.kaggle.com/datasets/grassknoted/asl-alphabet>

Kaggle dataset: <https://www.kaggle.com/datasets/prathumarikeri/american-sign-language-09az>

References for preprocessing: <https://www.kaggle.com/code/gpiosenka/asl-f1-score-98>

Error resolution: Stack overflow, Medium Blogs, Papers with Code (<https://paperswithcode.com/>) and chatGPT

Learning and Samples:

<https://medium.com/@nina95dan/simple-image-classification-with-resnet-50-334366e7311a>

<https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33>

<https://towardsdatascience.com/beginners-guide-on-image-classification-vgg-19-resnet-50-and-inceptionresnetv2-with-tensorflow-4909c6478941>

<https://towardsdatascience.com/create-image-classification-models-with-tensorflow-in-10-minutes-d0caef7ca011>

<https://medium.com/intro-to-artificial-intelligence/simple-image-classification-using-deep-learning-deep-learning-series-2-5e5b89e97926>