BUS ADM 745

Artificial Intelligence for Business

Final Project

American Sign Language Detection

Group 1
Bryan Jensen
Gayatri Shinde
Shubham Kokate
Tejashree Rajagopalan

Introduction

Sign language serves as a crucial means of communication within communities facing challenges in speech and hearing. It empowers individuals who are deaf or hard of hearing to interact and communicate effectively. Despite its significance, sign language remains unfamiliar to many, hindering smooth communication for this community.

In this context, our project endeavors to bridge this communication gap by leveraging technology to detect and interpret American Sign Language (ASL) gestures. The project addresses two fundamental aspects of sign language communication: static hand gestures using Convolutional Neural Networks (CNN) and hand motion recognition employing Random Forest algorithms.

Business Value

Sign language encompasses diverse components beyond static hand gestures; it integrates facial expressions and body movements to convey meaning and context along with hand motions. Recognizing the importance of these multifaceted elements, our project focuses on detecting static hand gestures via images, acknowledging their role as static hand acts devoid of motion information.

Additionally, while motion detection in sign language is pivotal, it poses challenges in real-life scenarios due to varying background situations. Our approach doesn't directly tackle dynamic hand motions but acknowledges their significance in comprehensive communication.

We aim to deliver technological solutions that facilitate better understanding and recognition of ASL gestures. By leveraging CNN for static images and exploring hand motion recognition through Random Forest, our project endeavors to enable smoother communication for individuals using sign language, ultimately contributing to inclusive communication platforms.

Our project's objectives align with the diverse nature of sign language communication, providing tools that facilitate effective communication through gestures, including static hand signals, facial expressions, and potentially dynamic hand motions in the future. This endeavor seeks to empower and include individuals within the speech and hearing-impaired community, fostering a more inclusive and accessible environment.

1. ASL with CNN

The problem being addressed here is sign language classification using machine learning, particularly Convolutional Neural Networks (CNNs). The dataset used contains images of hand gestures representing letters in sign language, with each image corresponding to a letter from the American Sign Language (ASL) alphabet.

The goal is to build a model that can accurately classify these hand gesture images into their respective letters. Here's the breakdown:

Problem Type:

- Classification Problem: It involves assigning a class label (a letter from the ASL alphabet) to each input image.

Dataset:

- Sign Language Dataset: Contains grayscale images of hand gestures representing individual letters from the ASL alphabet.
- Data Structure: Each image is 28x28 pixels, and the dataset includes labels corresponding to each image.

Data Cleaning:

- Reading Data: The code starts by importing necessary libraries like Pandas, NumPy, Matplotlib, and modules from Keras for deep learning.
- Loading Data: It reads two CSV files ('sign_mnist_train.csv' and 'sign_mnist_test.csv') using Pandas to create train and test DataFrames.

- Conversion to Numpy Arrays: The train and test DataFrames are converted to NumPy arrays (train_data and test_data) for manipulation and analysis.
- pd.read_csv(): Utilized to read the CSV files ('sign_mnist_train.csv' and 'sign_mnist_test.csv') into Pandas DataFrames for further processing.
- np.array(): Converts Pandas DataFrames into NumPy arrays (train_data and test_data) to facilitate numerical computations.

Data Extraction:

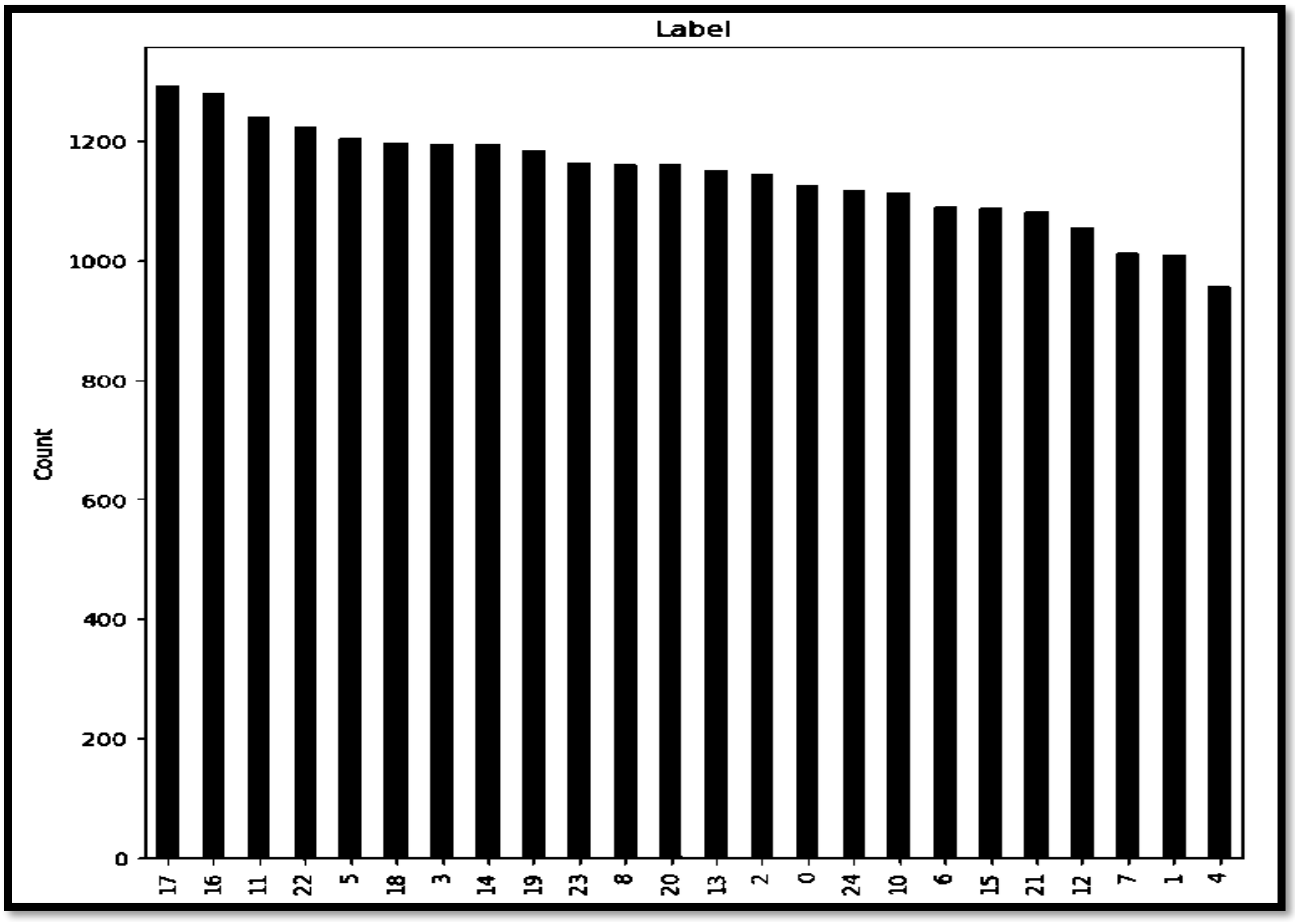
- Data as Numpy Arrays: The train and test datasets are stored as NumPy arrays for further processing.
- Class Labels: Class names are defined for the 24 alphabets represented in sign language in the dataset.
- random.randint(): Generates a random integer to select a random image from the dataset for visualization.
- plt.imshow(): Displays the selected image using Matplotlib for visual inspection.
- class_names[int(train_data[i, 0])]: Maps the numerical labels to their corresponding letter representations for display purposes.

Feature Engineering:

- Data Sanity Check: A random image and its associated label are plotted to ensure data integrity.
- Data Distribution Visualization: The distribution of labels/classes in the training dataset is visualized through a bar plot to ensure a balanced dataset.
- train['label'].value_counts().plot(kind='bar', ax=ax1): Uses Pandas' value_counts() to count occurrences of each label and Matplotlib's plot() function to create a bar plot, visualizing the distribution of classes in the training set

Exploratory Data Analysis (EDA):

Data Visualization: The code visually examines the distribution of classes in the training set to ensure a balanced dataset where we found data was fairly balanced



Model Training:

- Data Normalization: The pixel values are normalized to a range between 0 and 1.
- One-Hot Encoding: The categorical labels are converted to one-hot encoded vectors for model compatibility.
- Reshaping Data: The input data is reshaped to fit the expected input shape of the neural network.
- Model Creation: A sequential model is defined using Keras with Convolutional

Layer Descriptions:

- Conv2D Layer (32 filters, 3x3 kernel, ReLU activation):
  - Generates 32 convolutional filters.
  - Each filter operates on a 3x3 grid of pixels.
  - Uses Rectified Linear Unit (ReLU) activation to introduce non-linearity.
- MaxPooling2D Layer (2x2 pool size):
  - Performs max pooling operation with a 2x2 filter to downsample the feature maps.
  - Reduces the spatial dimensions of the data.
- Dropout Layer (20% dropout rate):
  - Introduces dropout regularization to mitigate overfitting.
  - Drops 20% of the randomly selected neurons during training.
- Conv2D Layer (64 filters, 3x3 kernel, ReLU activation) followed by MaxPooling and Dropout:
  - Similar to the first Conv2D layer but with 64 filters.
  - Followed by MaxPooling and Dropout for feature extraction and regularization.
- Conv2D Layer (128 filters, 3x3 kernel, ReLU activation) followed by MaxPooling and Dropout:
  - Another Conv2D layer with 128 filters for deeper feature extraction.
  - Followed by MaxPooling and Dropout to further downsample and regularize.
- Flatten Layer:
  - Flattens the 3D output into a 1D vector for input to the Dense layers.
- Dense Layer (128 neurons, ReLU activation):
  - Fully connected layer with 128 neurons.
  - Applies ReLU activation to introduce non-linearity.
- Dense Layer (25 neurons, Softmax activation - Output Layer):
  - Final output layer with 25 neurons (equal to the number of classes in the dataset, one for each sign).
  - Uses Softmax activation to output probabilities for each class, determining the final classification.

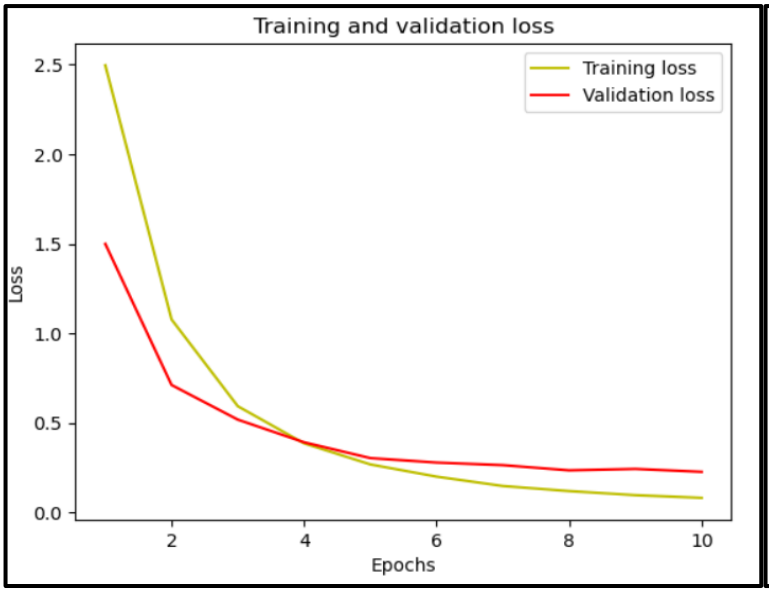
Activation Functions:

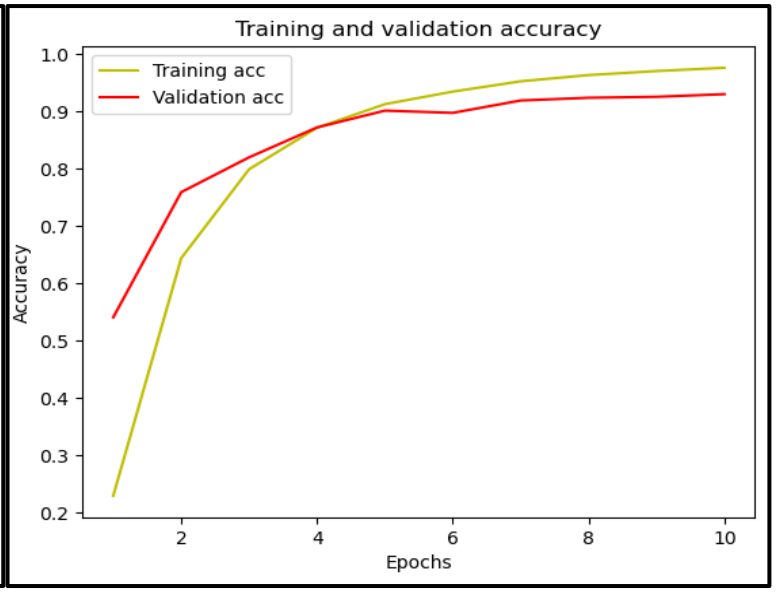
- ReLU (Rectified Linear Unit): Used in Conv2D and Dense layers except for the output layer. It introduces non-linearity by converting negative values to zero, enabling the model to learn complex patterns in the data.
- Softmax: Employed in the final Dense layer (output layer) to produce probability distributions across all classes, allowing the model to make class predictions based on these probabilities.
- to_categorical(): Converts categorical labels to one-hot encoded vectors, aiding in neural network training with Keras.
- model.add(): Sequentially adds layers to the Keras Sequential model, including Conv2D (convolutional), MaxPooling2D (pooling), Dropout (regularization), and Dense (fully connected) layers.
- model.compile(): Configures the model for training by specifying the loss function, optimizer, and evaluation metric.
- model.fit(): Trains the model on the training data for a specified number of epochs with defined batch size and validation data

Metrics Evaluation:

- Training and Validation Metrics: Plots are generated to visualize the training and validation loss as well as accuracy across epochs.
- Accuracy Calculation: The accuracy score is calculated using scikit-learn's accuracy_score.
- Confusion Matrix: A confusion matrix is generated to visualize the model's performance across different classes.
- Fractional Incorrect Predictions: The fraction of incorrect predictions per class is plotted to understand where the model struggles.
- Test Prediction: A random test image is chosen, and the model predicts its label, showcasing the predicted and true labels along with the image.

- history.history['loss'] and history.history['val_loss']: Accesses training and validation loss values from the model training history for plotting.
- plt.plot(): Generates line plots to visualize training and validation loss/accuracy across epochs.
- confusion_matrix(): Computes the confusion matrix, showcasing model performance in classifying different categories.



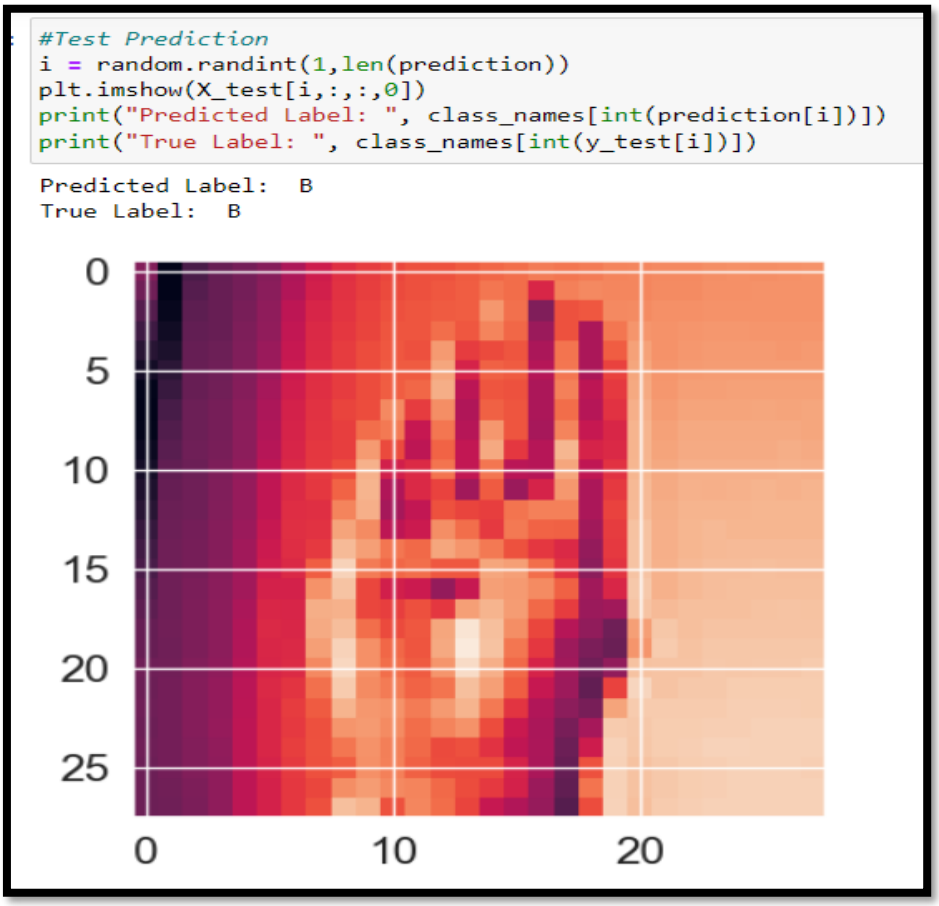


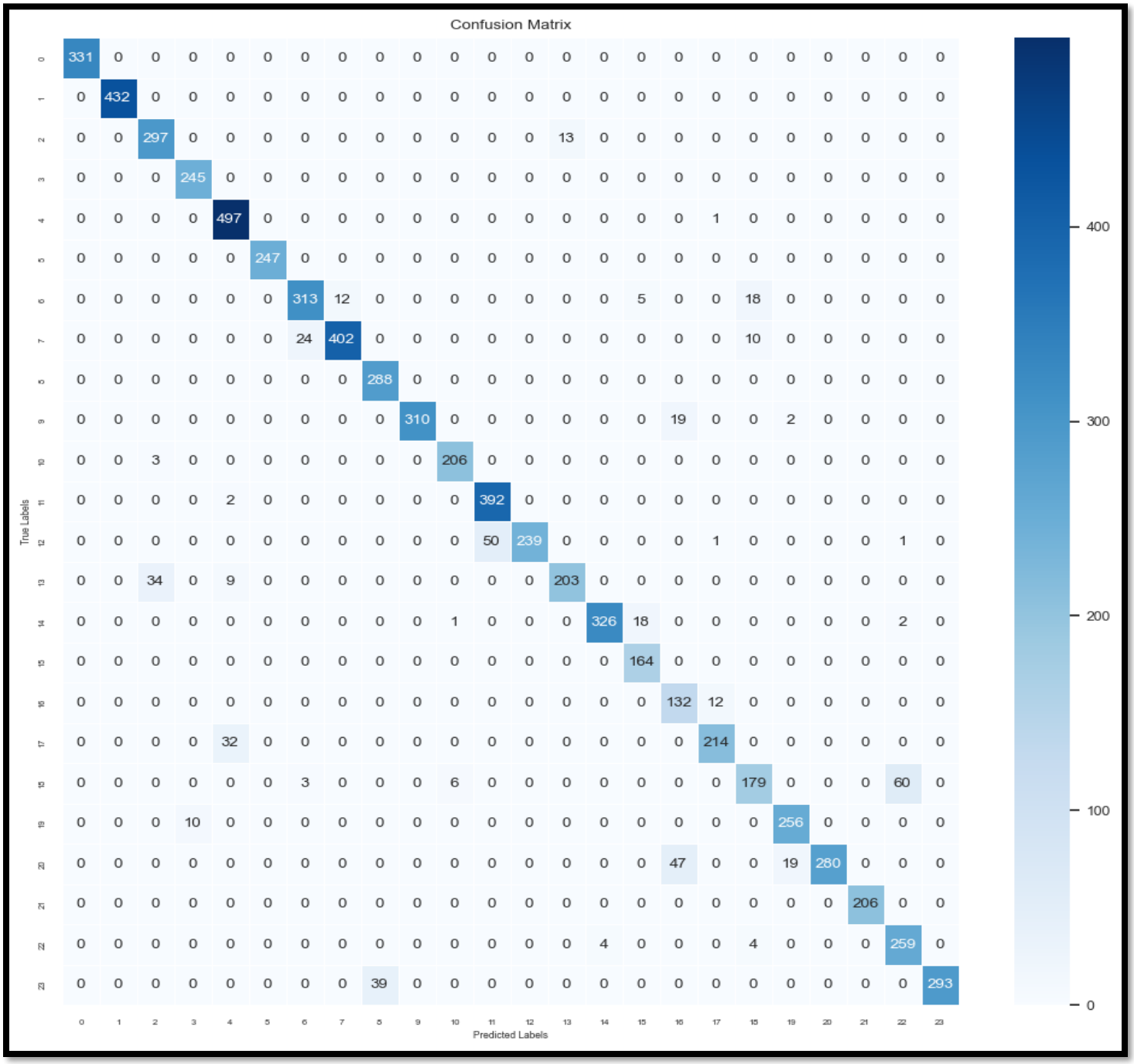
Test Prediction:

- plt.imshow(): Displays a random test image from the test set using Matplotlib.
- np.argmax(): Determines the index of the maximum value along an axis, used here to predict the class label for test data.
- class_names[int(prediction[i])] and class_names[int(y_test[i])]: Maps predicted and true numerical labels to their corresponding letters for display.

```
#Test Prediction
i = random.randint(1,len(prediction))
plt.imshow(X_test[i,:,:,0])
print("Predicted Label: ", class_names[int(prediction[i])])
print("True Label: ", class_names[int(y_test[i])])
```

```
Predicted Label:  B
True Label:  B
```



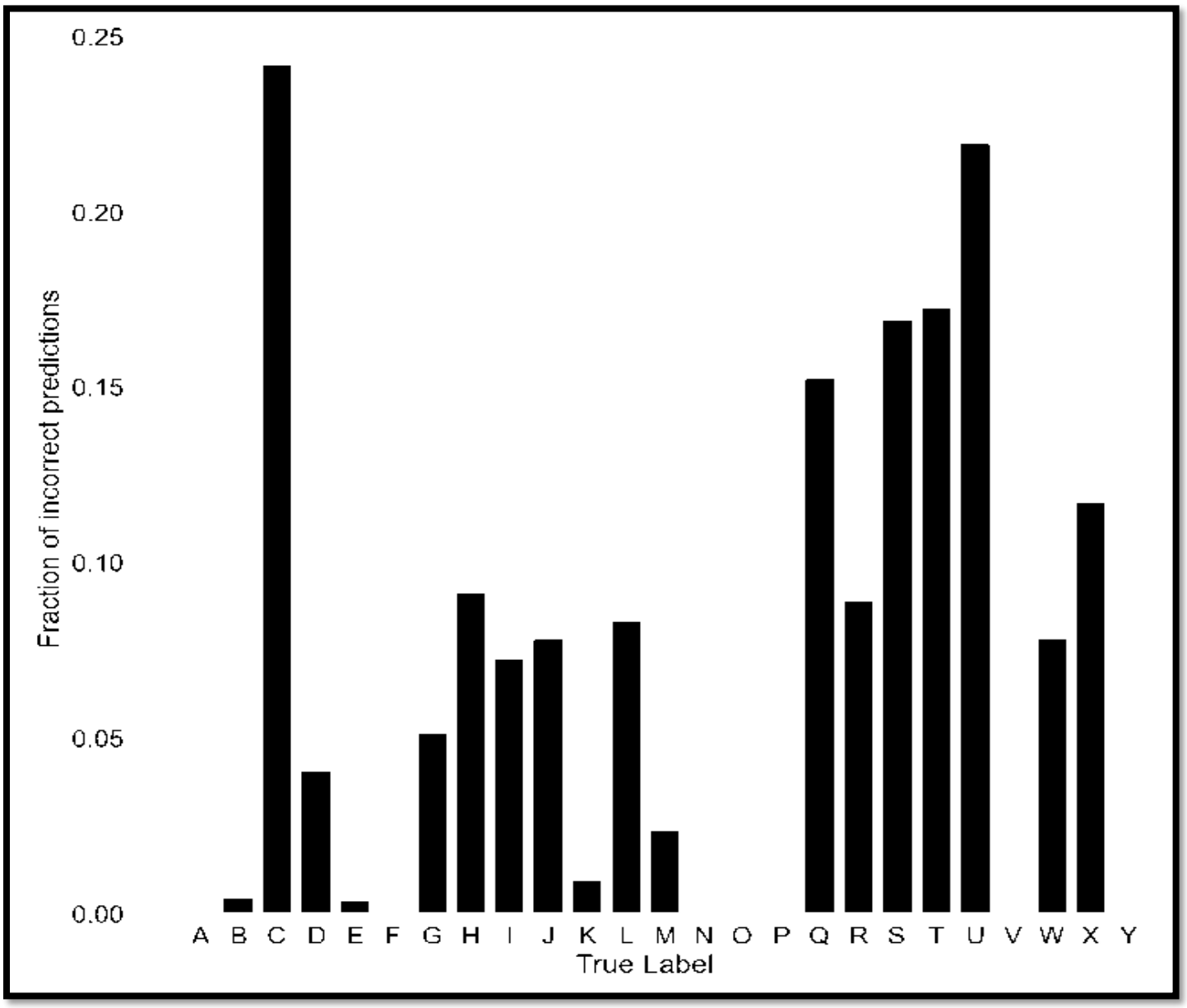


- Training Loss:
  - Epoch 1: Training loss starts at 2.5015 and progressively decreases over epochs.
  - Epoch 10: It reaches 0.0719. This indicates the decrease in the error made by the model on the training data as it learns more about the patterns in the data.

- Validation Loss:
  - Epoch 1: Validation loss begins at 1.3530 and decreases consistently across epochs.
  - Epoch 10: It stabilizes at 0.1910. The validation loss also decreases, indicating that the model generalizes well to unseen data.

- Training Accuracy:
  - Epoch 1: Training accuracy starts at 0.2334 and increases steadily.
  - Epoch 10: It reaches 0.9764, signifying the proportion of correctly predicted labels in the training dataset.

- Validation Accuracy:
  - Epoch 1: Validation accuracy starts at 0.5753 and increases notably.
  - Epoch 10: It reaches 0.9357, representing the proportion of correctly predicted labels in the validation dataset.



Out of all the letters that were misclassified: C,S,T,U were the highest.

2. ASL with Random Forest Classifier

Problem Type:

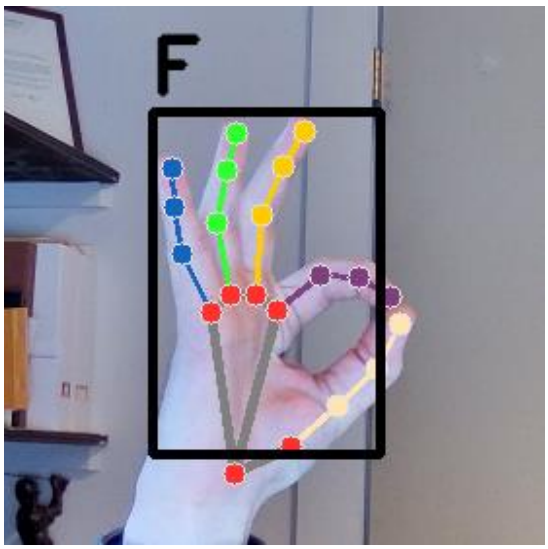
- Similar to the Convolutional Neural Network approach, this is a classification problem. The model is to take each input image and classify it into one of the 24 letters from the ASL alphabet.

Dataset/Collection:

- Captures images for 24 ASL classes through a webcam, allowing users to perform each gesture and collecting multiple images per class.
- The dataset contains color images of hand gestures representing the individual letters of the ASL alphabet.
- Each image is 640x480 pixels and has a corresponding label.
- The images that constitute this dataset are a combination of images that were gathered by each group member. Each member collected 200 images per letter, equating to 19,200 total images in the dataset (200 images * 24 letters * 4 group members).
- For the image collection process, we utilized the Open Computer Vision library (opencv) to capture and save images of each group member performing the various hand gestures.
- cv2.VideoCapture(0): Initializes the webcam to capture video.
- cv2.imshow(), cv2.waitKey(): Displays the video feed and waits for a key press ('q') to start collecting images for each gesture.
- cv2.imwrite(): Saves images in directories corresponding to the ASL classes

Data Preparation/Cleaning:

- Combines collected images into a centralized directory structure, merging data from multiple folders into a single dataset for unified processing.
- To prepare the data, we use the MediaPipe library. This step in the process takes the images, in a numpy array format, and outputs three components:
  - Handedness, which represents whether the detected hands are left or right.
  - Landmarks, which represent 21 distinct features of the hands, such as fingertips, the wrist, and other joints, and are composed of x, y, and z coordinates. The x and y coordinates are normalized to [0.0, 1.0] by the image width and height, respectively. The z coordinate represents the depth, with the depth at the wrist being the origin.
  - World Landmarks, which present the 21 Landmarks in world coordinates. Each x, y, and z represents real-world 3D coordinates in meters with the origin at the hand's geometric center.
- For our purposes, we are only interested in the second output, the landmarks. An example of what this looks like is below



- os.listdir(), os.makedirs(), os.path.join(): Manages directories for organizing collected data.
- shutil.copy(): Copies files from source to target directories, avoiding filename conflicts
- mp_hands.Hands(): Initializes MediaPipe Hands for hand landmark detection.
- cv2.imread(), cv2.cvtColor(): Reads and converts images to RGB for processing.
- results = hands.process(): Processes the frame using MediaPipe Hands to detect hand landmarks.
- Subsequently, the code stores the processed data and labels into a 'data.pickle' file for future utilization or model training. In handling inconsistencies where the extracted features don't match the anticipated length, the code alerts about the mismatch and excludes the specific image from being added to the dataset.

Model Training:

- Data Normalization: The landmark's x and y coordinates have been normalized between [0.0, 1.0].
- The landmark's z coordinates are not normalized.
- Reshaping Data: The data and labels are loaded into their respective numpy arrays to input into the model.
- Model Creation: Using Scikit-Learn's ensemble methods, a Random Forest Classifier model is defined.
- Model Training: The model is trained on the training data using the model.fit() function.
- pickle.dump(), pickle.load(): Saves and loads preprocessed data and models using pickle files.
- train_test_split(): Splits data into training and testing sets.
- GridSearchCV(): Searches for the best hyperparameters for the Random Forest Classifier model.
- model.predict(): Uses the trained model for predictions.
- model.predict(): Predicts ASL gestures based on the hand landmarks detected.
- cv2.rectangle(), cv2.putText(): Draws bounding boxes and displays predicted gestures on the video feed.

GridSearchCV systematically explores the defined parameter grid, evaluates models with different hyperparameters using cross-validation, and identifies the combination of hyperparameters that yields the best performance for the Random Forest Classifier in ASL gesture recognition. This process helps to optimize the model's performance by selecting the most suitable hyperparameters

Best parameters found: {'max_depth': 30, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 300}

Test Prediction

- mp.solutions.hands:
  - **mp_hands** is an instance of the **Hands** class provided by MediaPipe.
  - It represents the module responsible for hand tracking and detecting landmarks in the captured video frames or images.
- mp.solutions.drawing_utils:
  - **mp_drawing** is an instance of the **drawing_utils** module from MediaPipe.
  - It contains utility functions to help visualize and draw annotations (like hand landmarks) on the frames.
- mp.solutions.drawing_styles:
  - **mp_drawing_styles** is an instance that holds various styles for drawing annotations.
  - It provides predefined styles that can be applied to the drawn annotations, such as hand landmarks or connections.
- Prediction Loop: The code enters an infinite loop to continuously process video frames from the webcam (cap). Within this loop:
  - Frame Processing: Each frame read from the webcam is converted to RGB color space to match MediaPipe's requirements.
  - Hand Landmark Detection: MediaPipe processes the frame to detect hand landmarks (results = hands.process(frame_rgb)).
  - Visualization: Detected hand landmarks are drawn on the frame using mp_drawing.draw_landmarks.

Metrics Evaluation: Random Forest Classifier Result Analysis

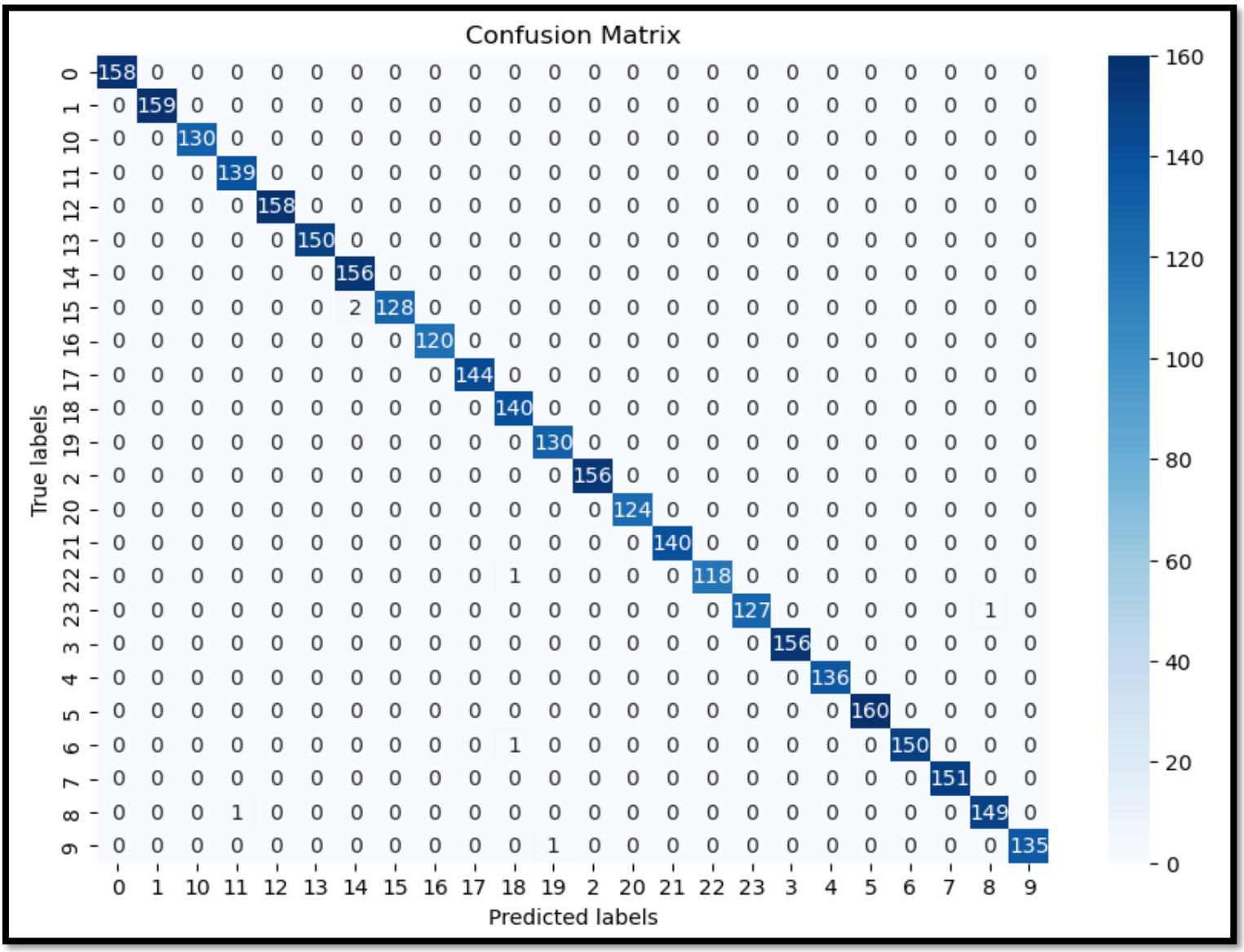
- The Random Forest classification model achieved an accuracy of 99.795%.
- This represents a relatively modest improvement over the Convoluted Neural Network model's accuracy of 93.168%.
- However, as discussed previously and we will discover through a thorough analysis of the Random Forest results, it should be recognized that both models have performed incredibly well.
- The metrics we have utilized to analyze this model's performance are precision, recall, and F1 scores.
- Each metric is calculated twice, using different averages.
- The micro average calculates the metric globally by counting the total true positives, false negatives, and false positives.
- The macro average calculates the metrics for each label, and finds their unweighted mean.
- Using the macro average does not take label imbalance into account, though this has no impact as the data is not imbalanced. For the sake of brevity, these metrics are calculated for the entire dataset instead of for each class.

Precision:
- **Micro:** 99.80%
- **Macro:** 99.80%

Recall:
- **Micro:** 99.80%
- **Macro:** 99.78%

F1 Score:
- **Micro:** 99.80%
- **Macro:** 99.79%

- Overall, these metrics are very homogenized. This suggests a balanced performance.
- The model exhibits high precision, recall, and F1 scores.
- The macro recall and F1 scores are slightly lower, 0.02% and 0.01% respectively, than the micro recall and F1 scores. This suggests that there are some categories that are mislabeled relatively more often than other categories.



The confusion matrix offers a detailed breakdown of correct and incorrect responses

Key observations:

- The model misclassified the letter Q (15) the most. It was predicted to be P (14) two times.
- The model predicted the letter X (22) to be T (18) once.
- The model predicted Y (23) to be I (8) once.
- The model predicted G (6) to be T (18) once.
- The model predicted I (8) to be M (11) once.
- The model predicted K (9) to be U (19) once.

There is no clear trend with these misclassifications, suggesting that there is very low-to-no bias in the model. Some categories were mislabeled relatively more than others as suspected. However, relative carries a lot of weight in this case as 18 out of the 24 categories had no misclassifications.

Conclusion:

Both the Convoluted Neural Network and Random Forest Classifier models proved to have impressive performance. However, the Random Forest model is the clear winner with an accuracy of 99.795% versus CNN's 93.168%. When it comes to real world applications, we believe that both models could prove useful. Although the Random Forest model proved to be incredibly accurate, the training data we used was gathered in a controlled manner, and may lose accuracy when trained on a more diverse dataset. The images used to train the CNN model had a variety of backgrounds, different objects in the picture, and the like. Though this could have led to the model having a lower accuracy, this could also better reflect real-world conditions and may be worth the trade off. These models could lay the foundation for improvements in AI-driven ASL communication.