Identification of Digits from Sign Language Images

Aprendizagem Automática – Professor: Pétia Georgieva

Hugo Correia, 108215 50%
DETI
Universidade de Aveiro

Joaquim Rosa, 109089 50%
DETI
Universidade de Aveiro

***Abstract*—This work presents the development of machine learning models for the recognition of hand gestures and sign language, aimed at facilitating communication between deaf or speech-impaired individuals and those without these limitations. We utilized a dataset of sign language digit images provided by Kaggle to implement and compare various machine-learning models. This study details the adaptations and optimizations made to the models, based on previous works that positively influenced our outcomes. The models were evaluated for their accuracy and efficiency, proving to be promising tools for reducing the communication barriers faced by hearing-impaired individuals. The results indicate that advanced image processing techniques and deep learning can significantly improve the automatic interpretation of sign language, promoting more inclusive communication.**

***Index Terms*—Sign language recognition, Digit recognition, Machine learning**

I. Introduction

Sign language provides an essential bridge for communication between individuals with hearing impairments and those who can speak; however, it is regrettable that a considerable communication gap still exists. This gap is primarily due to the substantial effort required to learn sign language. Consequently, trained sign language interpreters are crucial for bridging this gap. There has been a noticeable increase in demand for such interpreters across various sectors including healthcare, legal, and educational fields over recent years.

II. State of Art

The domain of sign language recognition has witnessed considerable evolution due to advances in computer vision and machine learning. Researchers have explored various approaches, ranging from traditional image processing techniques to sophisticated deep learning algorithms, to enhance the accuracy and efficiency of sign language interpretation systems.

Early attempts at sign language recognition primarily relied on hand-crafted features and classifiers such as Hidden Markov Models (HMMs) and Support Vector Machines (SVMs). For instance, Starner and Pentland's seminal work [1] utilized HMMs to recognize American Sign Language (ASL) gestures from video sequences. While effective to an extent, these methods were often limited by the manual feature extraction process and the complexities involved in modeling temporal dependencies for dynamic gestures.

With the advent of deep learning, Convolutional Neural Networks (CNNs) have become the cornerstone of many state-of-the-art sign language recognition systems. The automatic feature extraction capability of CNNs allows for capturing intricate patterns in sign language gestures. A significant contribution in this area is the work by Rastgoo et al. [2], which demonstrated the superiority of CNN-based models over traditional machine-learning approaches. Their models not only achieved higher accuracy but also demonstrated robustness against variations in background and lighting conditions.

Another notable advancement is the incorporation of temporal information in sign language recognition. 3D CNNs and recurrent neural network architectures like Long Short-Term Memory (LSTM) networks have been employed to account for the sequential nature of sign language. Adaloglou et al. [3] provided a comprehensive analysis of how such models can effectively learn spatio-temporal features, leading to improved recognition of dynamic gestures.

Moreover, the application of transfer learning, where pre-trained models on large datasets are fine-tuned for sign language recognition tasks, has also gained traction. The ability to leverage knowledge from related vision tasks has been instrumental in achieving high accuracies even with limited sign language data. For example, Koller et al. [4] showcased how transfer learning could mitigate the scarcity of labeled sign language datasets, which is a persistent challenge in the field.

Despite these technological strides, the development of sign language recognition systems that can function in real-world scenarios remains challenging. Issues such as the diversity of sign languages, signer independence, and the need for real-time interpretation are yet to be fully addressed. The current research aims to contribute to this evolving field by proposing models that are not only accurate but also computationally efficient, paving the way for their deployment in practical applications.

III. Dataset Description and Visualization

For the development of this project, we utilized the "Sign Language Digits Dataset" provided by Kaggle [5]. This dataset comprises two npy files: one containing 2062 images of digits represented in sign language, each with dimensions of 64x64 pixels and in grayscale color space; the other file contains the corresponding labels for each image in one-hot encoding format.

A. Sample Distribution Analysis

It's worth noting that this dataset was already balanced, with each label having a similar number of examples. The label with the fewest examples had 204, while the one with the most had 208. Consequently, no preprocessing regarding class balancing was required.

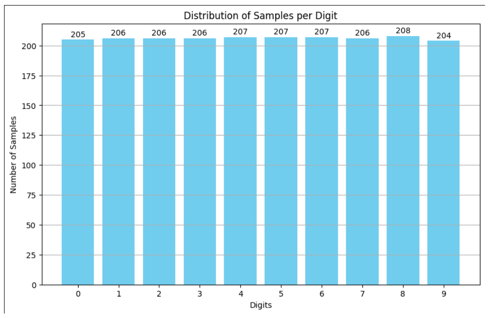


Fig. 1: Distribution of Samples per Digit

B. Sample Visualization

This dataset contains various examples of images representing digits from 0 to 9 in sign language, totaling 10 labels, as we can observe in the following figure:

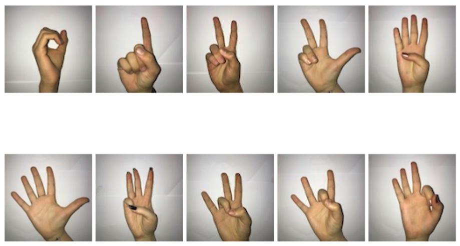


Fig. 2: Sample Examples of each digit

C. Mean Images Analysis

The Mean Images Analysis is a critical step in exploring the characteristics of our dataset at a glance. By averaging the pixel intensities of all images corresponding to each digit, we obtain a visual 'template' that highlights the most defining features of each class:

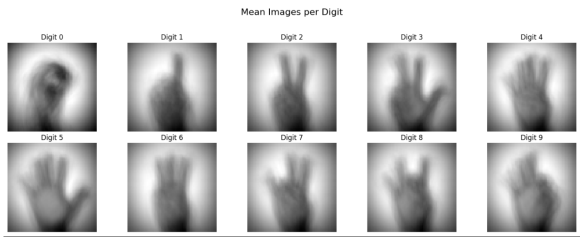


Fig. 3: Mean Images per digit

These mean images serve several important purposes in the context of machine learning:

- They allow us to visually assess the homogeneity within each class of digits. A clear and distinct mean image suggests that the members of the class share common traits.
- They can reveal if certain digits have more variation than others, which might affect the difficulty level for a model to learn and differentiate them.
- They help in identifying if the dataset contains biases or anomalies that could lead to overfitting or poor generalization of a trained model.

D. Pixel Value Distribution Analysis

In machine learning models for image recognition, understanding the distribution of pixel values can be crucial. The pixel intensity distribution for each digit can give us insights into the variability of the data and help determine if certain preprocessing steps like normalization or standardization are necessary. In the following image, box plots are represented showing the distribution of pixel values for each digit in our sign language dataset:

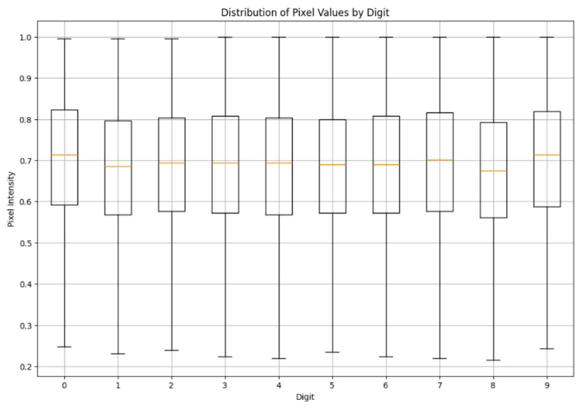


Fig. 4: Distribution of Pixel Values by Digit

With the visualization of these box plots, we can draw some conclusions:

- The median pixel intensity is consistent across digits, suggesting no significant difference in brightness or darkness among them.
- The compact IQR indicates that the majority of the pixel values are not widely spread out but rather clustered around the median.
- The absence of significant outliers suggests that extreme pixel values are not common for the images within each digit category.

Since there is no substantial variation in pixel intensity distribution between the digits, our machine-learning models will need to identify more subtle and complex patterns in the data to accurately classify each digit.

E. Correlation Heat Map Analysis

A Correlation Heatmap is a graphical representation of the correlation matrix between various features in a dataset. In the context of image processing, each pixel can be considered a feature, and the correlation heatmap helps us understand the relationship between different pixels. The analysis of a correlation map brings certain benefits, such as pattern identification, assistance in feature selection, and providing insight into data structure. The following figure shows the heatmap of the correlation matrix of our image dataset, allowing us to explore the relationships between pixel intensities across our dataset:

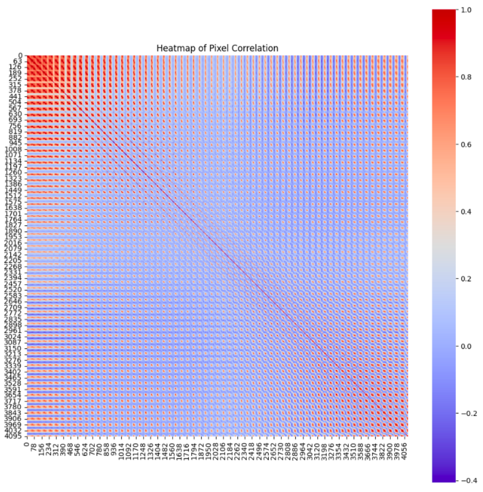


Fig. 5: Heat Map of Pixel Correlation

With the analysis of the heat map, we can make some observations:

- The corners of the heatmap show blocks of high positive correlation. This pattern suggests that pixels located in similar positions across different images—specifically the corners of the images—tend to have similar values. This similarity could be due to consistent background areas or edges that are common across the dataset.
- The heatmap exhibits a strong diagonal line of perfect positive correlation, which is expected as this represents each pixel's correlation with itself. However, the surrounding off-diagonal areas show variations in correlation, indicating diverse pixel relationships.
- There is a visible transition from blue to red as we move from the corners toward the center along the diagonal. This gradient may represent the transition from the image background to the more varied central regions where the digit symbols are located.

IV. Dataset Preprocessing

A. Dataset Reorganization

Observing the image samples together with their associated labels, we noticed that the dataset was not properly organized. In other words, the images of a particular digit were associated with a different label, as can be seen in the following image:

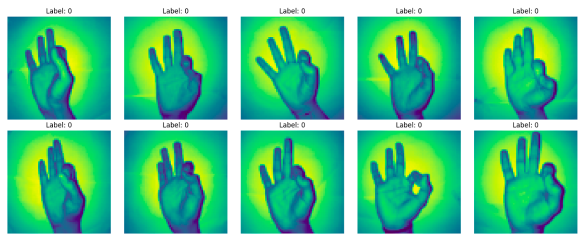


Fig. 6: Examples of images of the digit 9 associated with label 0

Although this label swap did not affect the training of the models, as we would later need correct labels for result visualization, we reorganized the dataset so that the images were sorted by the digit they represented.

B. Dataset Augmentation

To improve the training and subsequently the results of our models, we performed dataset augmentation, as our original dataset had a limited number of samples. To increase the dataset size, we applied some transformations to the existing data samples:

- **Rotation:** Rotate the image by a random angle between -30 and 30 degrees.
- **Gaussian Noise:** Add gaussian noise with a variance of 0.01^2.
- **Gamma Contrast:** Change the contrast of the image by a random factor between 0.8 and 1.2.
- **Translation:** Translate the image by a random value between -5 and 5 in both the x and y directions.

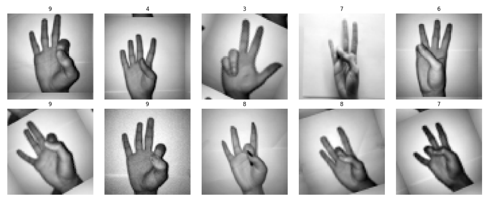


Fig. 7: Examples of the new Data Created

For every image in the dataset, we created 12 new images with the above changes, 3 images for each transformation. The total number of examples were extended to 26806.

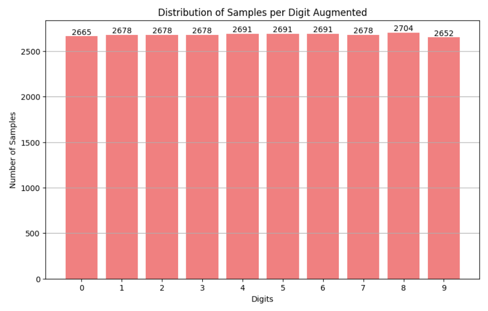


Fig. 8: Distribution of Samples per Digit (Augmented)

V. Models

Given the chosen problem, we decided to use Logistic Regression, Convolutional Neural Network, Artificial Neural Network, and Support Vector Machine models, and determine which of the models would yield the best results.

To improve the performance of each model, we decided to initially train each model using predefined parameters that we considered reasonable, with the original dataset, then we applied hyperparameter tuning to each model using the original dataset, and finally, we also applied hyperparameter tuning to each model in conjunction with the aforementioned dataset augmentation. After doing this, we analyzed the improvements that hyperparameter tuning and data augmentation would yield in comparison to the original models. Hyperparameter tuning is the process of finding the best values for the parameters that result in the best performance of the model on the given dataset. To implement this and get the best model performance, we created a function that given a model, the parameters to test and the dataset, it will try the different combinations of parameters and return the best model and its performance.

In the Logistic Regression, Convolutional Neural Network, and Artificial Neural Network models, we performed a data splitting of 80% for training the model, 20% for validation, and 20% for testing.

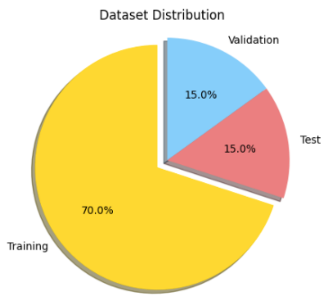


Fig. 9: Dataset Distribution for Logistic Regression, Convolutional Neural Network, and Artificial Neural Network models

In the Support Vector Machine model, we performed a data splitting of 80% for training the model and 20% for testing.

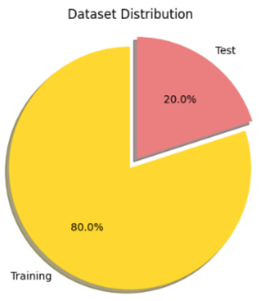


Fig. 10: Dataset Distribution for Support Vector Machine model

VI. Models Using Original Dataset

A. Logistic Regression One-vs-all Multi-class Classification

Logistic regression is a supervised machine-learning algorithm used for classification. When dealing with a binary classification problem (two classes), logistic regression is straightforward and can be used to model the probability of an instance belonging to a specific class. However, when dealing with more than two classes, we need to handle this using the One-vs-All classification method, as we did for this problem.

In One-vs-All classification, we create a logistic regression model for each class in our dataset. For each model, the class we are modeling is treated as the "positive" class, and all other classes are treated as the "negative" class.

For the implementation of this model, we used the Keras library in Python, using the following hyperparameters in our initial approach:

- **learning_rate:** 0.1
- **decay:** 1^{-6}
- **momentum:** 0.9
- **nesterov:** True

After training the model, we obtained the following plots showing the evolution of accuracy and loss during training and validation across epochs:

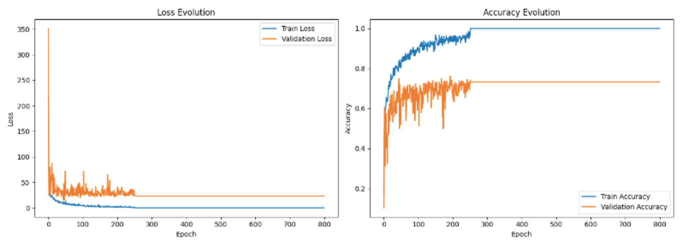
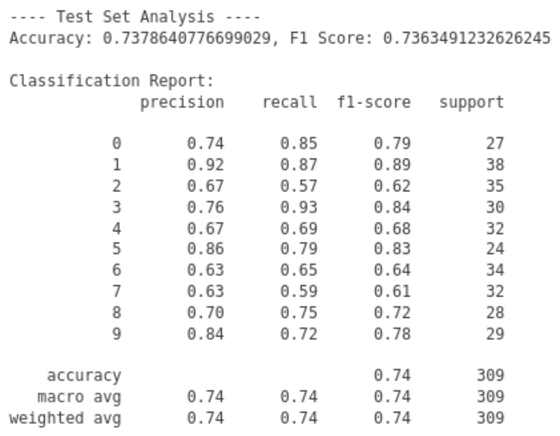


Fig. 11: Evolution of Accuracy and Loss During Training

These plots show us that the validation loss and accuracy are not following the training loss and accuracy values, indicating that we are dealing with a case of overfitting, where our model is learning the training data too well, but is not generalizing well to new data.

Due to the above problems, our model managed to obtain an accuracy and F1 score of 0.74:

We can better visualize the performance of the Logistic Regression model by looking at the following confusion matrix in which the rows symbolize the true label, and the columns the label predicted by the algorithm; this way, we can see which were predicted correctly (we can easily see by accompanying the diagonal traced), and the amount of incorrect predictions made with another specific digit.



```
---- Test Set Analysis ----
Accuracy: 0.7378640776699029, F1 Score: 0.7363491232626245

Classification Report:
              precision    recall  f1-score   support

           0       0.74      0.85      0.79        27
           1       0.92      0.87      0.89        38
           2       0.67      0.57      0.62        35
           3       0.76      0.93      0.84        30
           4       0.67      0.69      0.68        32
           5       0.86      0.79      0.83        24
           6       0.63      0.65      0.64        34
           7       0.63      0.59      0.61        32
           8       0.70      0.75      0.72        28
           9       0.84      0.72      0.78        29

    accuracy                           0.74       309
   macro avg       0.74      0.74      0.74       309
weighted avg       0.74      0.74      0.74       309
```

Fig. 12: Logistic Regression Classification Report

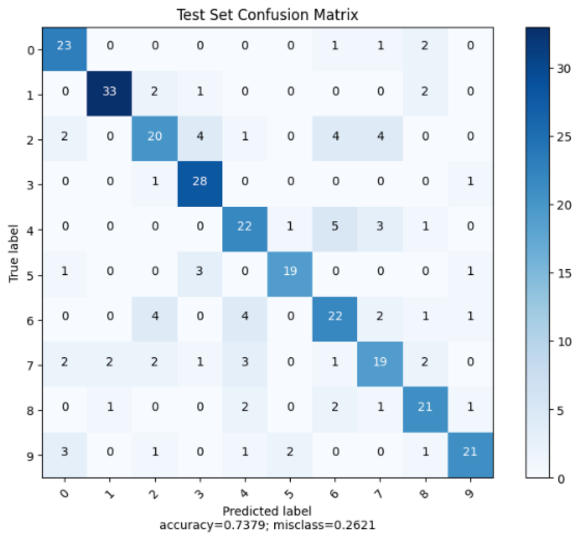


Fig. 13: Logistic Regression Confusion Matrix

For example, we can observe that the model had some difficulty in distinguishing between a 4 and a 6, resulting in lower precision for these two labels.

B. Convolutional Neural Network

A Convolutional Neural Network (CNN) is a type of neural network architecture commonly used in computer vision tasks such as image recognition and object segmentation. Inspired by the organization of the animal visual cortex, a CNN is designed to automatically learn patterns and features from images through specialized layers, which is why we chose this model.

The implementation we used was taken from a publication on Kaggle made by the user Efeergun [6] using the Keras library in Python. This CNN has the following structure:

- **Conv2D layers** apply convolutions to extract features from the image.
- **MaxPooling2D layers** reduce dimensionality, retaining the most important information.
- **Dropout layers** help prevent overfitting by randomly turning off neurons during training.
  (This pattern of layers, alternating between Conv2D, MaxPooling2D, and Dropout, repeats 3 times to extract features at different levels of abstraction.)
- A **Flatten Layer** converts the 2D feature maps into a one-dimensional vector to be used as input for the fully connected layers
- **Dense Layers** receives the feature vectors and apply learning operations on the data. The last Dense layer has a softmax activation function, which is used to obtain the probability of each class for a given input.
- The ***learning_rate*** used was 0.0002.
- The number of epochs defined was 70.

After training the model, we obtained the following plots showing the evolution of accuracy and loss during training and validation across epochs:

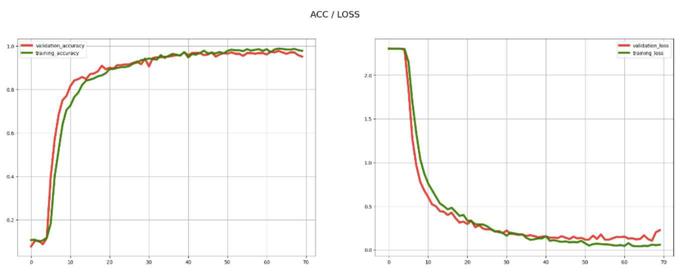
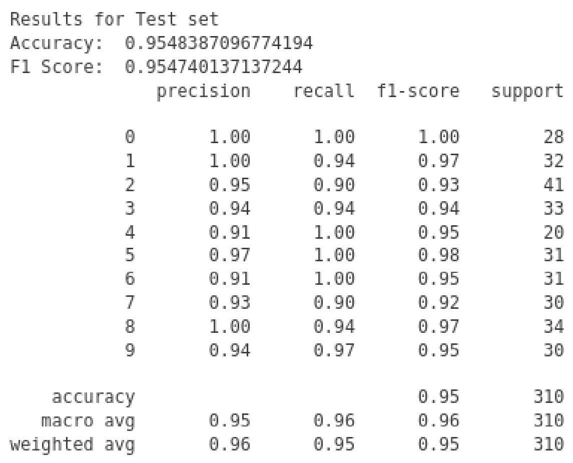


Fig. 14: Evolution of Accuracy and Loss During Training

Analyzing the graphs, we can observe that the evolution of validation accuracy and loss closely follows the evolution of training accuracy and loss, which is quite positive. This indicates that the model will likely be able to make accurate predictions on new, unseen data not encountered during training.

This analysis was confirmed, as the model achieved both an accuracy and an F1 score of 0.95:



```
Results for Test set
Accuracy:  0.9548387096774194
F1 Score:  0.954740137137244
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        28
           1       1.00      0.94      0.97        32
           2       0.95      0.90      0.93        41
           3       0.94      0.94      0.94        33
           4       0.91      1.00      0.95        20
           5       0.97      1.00      0.98        31
           6       0.91      1.00      0.95        31
           7       0.93      0.90      0.92        30
           8       1.00      0.94      0.97        34
           9       0.94      0.97      0.95        30

    accuracy                           0.95       310
   macro avg       0.95      0.96      0.96       310
weighted avg       0.96      0.95      0.95       310
```

Fig. 15: CNN Classification Report

By looking at the following confusion matrix, we can confirm that the model did a great job in classifying the images of sign language digits.

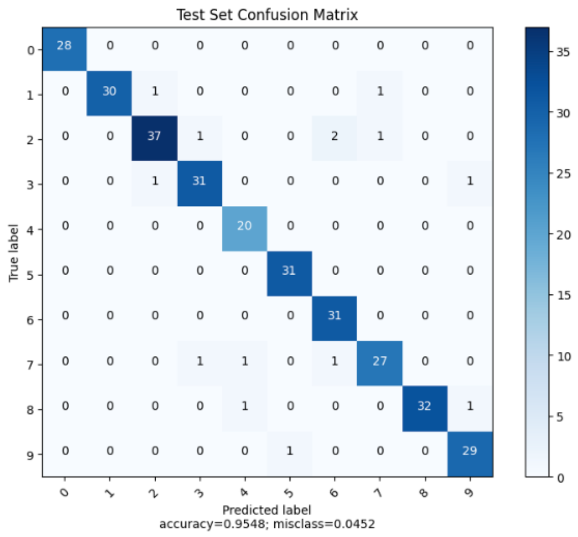


Fig. 16: CNN Confusion Matrix

C. *Artificial Neural Network*

An Artificial Neural Network (ANN) is a computational model composed of a large number of interconnected processing units, called artificial neurons or units. These networks are designed to mimic the functioning of the human brain, where neurons are interconnected and collaborate to perform complex tasks such as pattern recognition, classification, and prediction.

The implemented artificial neural network is a specific type of ANN known as a feedforward neural network, where connections between neurons do not form cycles, meaning information flows only in one direction, from the input layer to the output layer. This model was inspired by the one created by Piyush Bhedurkar, available on Kaggle [7] with some trial and error adjustments. Once again, the Keras library in Python was used to create the model, which has the following structure:

The implementation of our ANN includes L2 regularization in its dense layers. Regularization is crucial in machine learning models to prevent overfitting, especially when the dataset is not very large. By including a regularization term in the cost function, the model is penalized for having large weights, which encourages the network to maintain smaller weight values, leading to a simpler model that is less likely to overfit the training data.

- **Input Layer (Flatten) -** Flattens the input data to transform it into a one-dimensional vector.
- **Dense Layer -** It has 512 neurons, ReLU activation function, L2 regularizer with a parameter of 0.001 (helps prevent overfitting).
- **Dropout Layer -** Dropout rate of 0.3. Randomly deactivates a fraction of neurons during training, helping to prevent overfitting.
- **Dense Layer -** It has 256 neurons, also using the ReLU activation function and L2 regularizer with the same parameter of 0.001.
- **Dropout Layer -** Dropout rate of 0.2.
- **Output Layer (Dense) -** It has 10 neurons, corresponding to the number of classes in the classification problem.
- The ***learning_rate*** used was 0.00002.
- The number of epochs defined was 600.

After training the model, we obtained the following plots showing the evolution of accuracy and loss during training and validation across epochs:

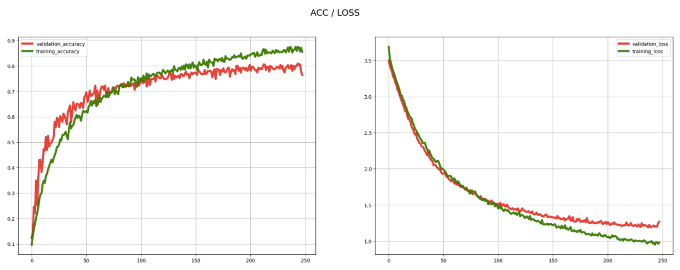
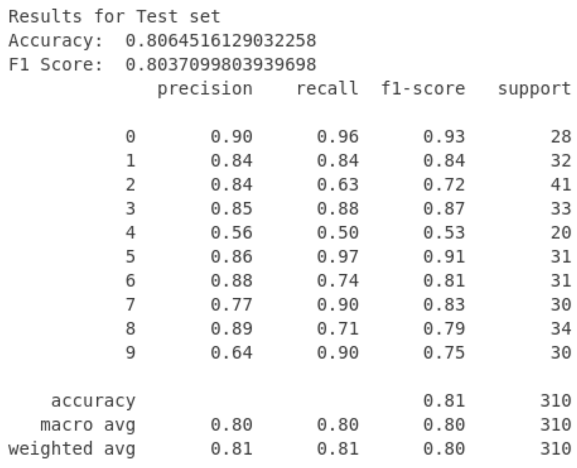


Fig. 17: Evolution of Accuracy and Loss During Training

By analysing the graphs, we can identify some overfitting, as there is a slight difference between the training and validation values.

However, the model obtained a reasonable result of 0.81 accuracy and 0.80 F1 score:



```
Results for Test set
Accuracy:  0.8064516129032258
F1 Score:  0.8037099803939698
              precision    recall  f1-score   support

           0       0.90      0.96      0.93        28
           1       0.84      0.84      0.84        32
           2       0.84      0.63      0.72        41
           3       0.85      0.88      0.87        33
           4       0.56      0.50      0.53        20
           5       0.86      0.97      0.91        31
           6       0.88      0.74      0.81        31
           7       0.77      0.90      0.83        30
           8       0.89      0.71      0.79        34
           9       0.64      0.90      0.75        30

    accuracy                           0.81       310
   macro avg       0.80      0.80      0.80       310
weighted avg       0.81      0.81      0.80       310
```

Fig. 18: ANN Classification Report

By analysing the confusion matrix below, we can see in more detail which incorrect predictions the model made:

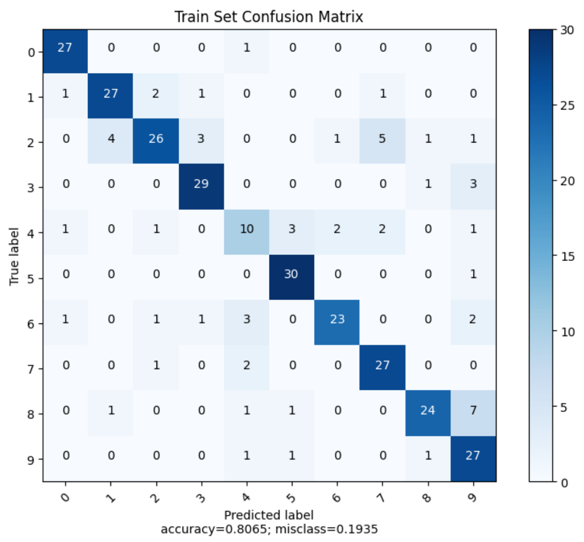


Fig. 19: ANN Confusion Matrix

For example, we can observe that this model had some difficulty in predicting the correct value for label 4.

D. Support Vector Machine

Support vector machine is another simple algorithm highly preferred by many as it produces significant accuracy with less computation power. Support Vector Machine, abbreviated as SVM, can be used for both regression and classification tasks. The idea behind it is simple: The algorithm creates a line or a hyperplane that separates the data into classes.

In our implementation, we created an SVM model using the Python sklearn library. The ***kernel*** parameter was set to ***linear***, specifying that this is a linear SVM. This means it will attempt to separate the classes using a linear hyperplane in the feature space.

The Support Vector Machine (SVM) is a powerful and versatile machine learning algorithm, particularly effective in classification problems with high-dimensional spaces. Given the relatively small size of our dataset and the complex nature of sign language recognition, we opted to employ cross-validation for the SVM to ensure that our performance estimates are both reliable and indicative of the model's ability to generalize to unseen data.

Cross-validation is executed using the k-fold method, where the dataset is randomly partitioned into k equal sized subsamples. Of the k subsamples, a single subsample is retained as the validation data for testing the model, and the remaining $k-1$ subsamples are used as training data. This process is repeated k times (the folds), with each of the k subsamples used exactly once as the validation data. The k results can then be averaged (or otherwise combined) to produce a single estimation. The advantage of this method over repeated random sub-sampling is that all observations are used for both training and validation, and each observation is used for validation exactly once.

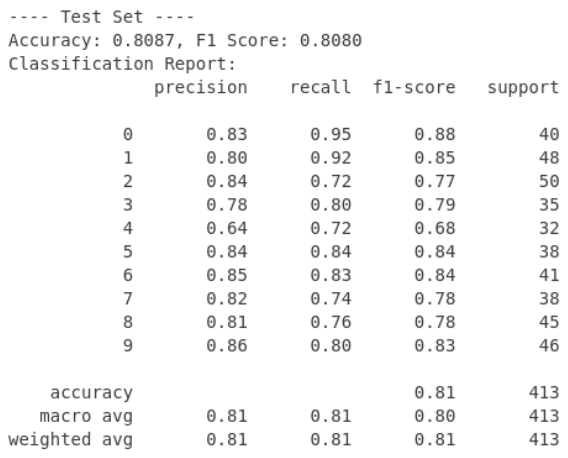
In our implementation, we have used 5-fold cross-validation, which is a standard choice, balancing the trade-off between computation time and the reliability of the estimation:

$$CV_k = \frac{1}{k}\sum_{i=1}^{k} \mathit{ModelPerformance}_i \qquad (1)$$

where CV_k represents the cross-validation performance score, k is the number of folds, and $\mathit{ModelPerformance}_i$ is the performance measure obtained on the i^{th} fold.

This approach to model evaluation is particularly beneficial for our SVM model, allowing us to fine-tune hyperparameters such as the cost parameter C, the kernel type, and the gamma parameter of the radial basis function (rbf) kernel, to optimize performance.

Our model achieved an accuracy and F1 score of 0.80, which is reasonable.


```
---- Test Set ----
Accuracy: 0.8087, F1 Score: 0.8080
Classification Report:
              precision    recall  f1-score   support

           0       0.83      0.95      0.88        40
           1       0.80      0.92      0.85        48
           2       0.84      0.72      0.77        50
           3       0.78      0.80      0.79        35
           4       0.64      0.72      0.68        32
           5       0.84      0.84      0.84        38
           6       0.85      0.83      0.84        41
           7       0.82      0.74      0.78        38
           8       0.81      0.76      0.78        45
           9       0.86      0.80      0.83        46

    accuracy                           0.81       413
   macro avg       0.81      0.81      0.80       413
weighted avg       0.81      0.81      0.81       413
```

Fig. 20: SVM Classification Report

And we can better visualize the performance of the SVM model by looking at the following confusion matrix:

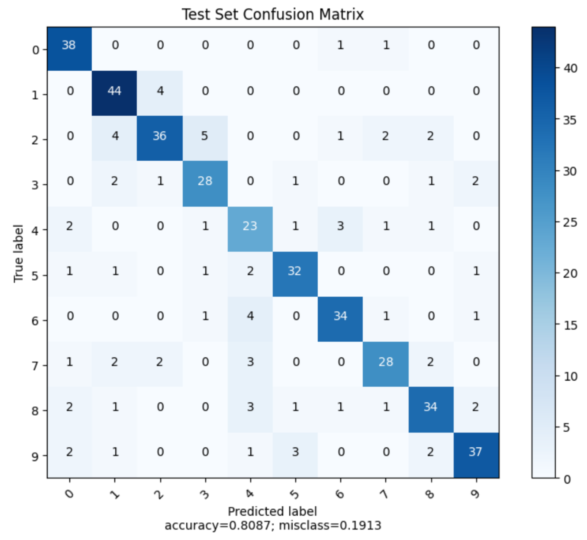


Fig. 21: SVM Confusion Matrix

VII. Tuned Models Using Original Dataset

A. *Logistic Regression One-vs-all multi-class classification*

To improve the performance of the previous model, we developed a function that trains the model 20 times, varying randomly the following hyperparameters across the following ranges:

Hyperparameter	**Values**
learning_rate	$[1^{-4} - 1^{-1}]$
momentum	[0.8 - 0.99]
nesterov	{0, 1}

TABLE I: Logistic Regression Parameters

After trying 20 different combinations, the combination of parameters that gave the best results was the following:

- **learning_rate -** 0.00026
- **momentum -** 0.84
- **nesterov -** 1

By analyzing the hyper tuning trials, we were able to draw some conclusions about how certain parameters significantly affected the model's performance:

- Models with very high learning rates (e.g., 0.0251 and 0.0367) showed very poor performance, with very low accuracies (close to 0.1) and high losses. This suggests that a high learning rate can cause the model to diverge or not converge properly.
- Models with moderate or low learning rates tend to show better accuracies and lower losses. For example, a learning rate of 0.000539611 and 0.000408221 resulted in some of the highest observed accuracies (approximately 0.92 and 0.91, respectively).
- Momentum assists in accelerating Stochastic Gradient Descent in relevant directions and appears to be a crucial factor in achieving good convergence. Momentum values of 0.89 and 0.94 were present in configurations with good performance, indicating that an appropriately high momentum value can improve performance.

The training of the model with the hyperparameters above showed some improvement in reducing overfitting compared to the previous version, as we can see in the fig. 22, but still has some, which will cause the model to make some inaccurate predictions on new data.

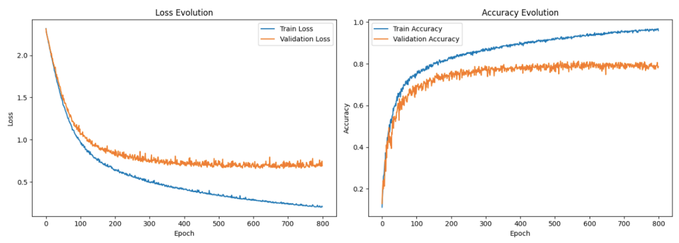
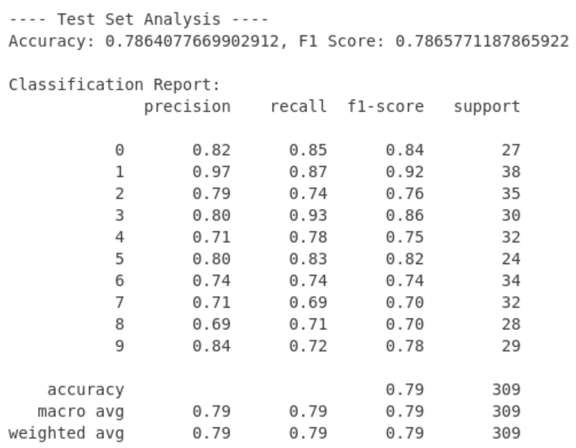


Fig. 22: Evolution of Accuracy and Loss During Training

There was an increase of 0.05 in both accuracy and F1 score compared to the model without hyperparameter tuning, resulting in both reaching 0.79:



```
---- Test Set Analysis ----
Accuracy: 0.7864077669902912, F1 Score: 0.7865771187865922

Classification Report:
              precision    recall  f1-score   support

           0       0.82      0.85      0.84        27
           1       0.97      0.87      0.92        38
           2       0.79      0.74      0.76        35
           3       0.80      0.93      0.86        30
           4       0.71      0.78      0.75        32
           5       0.80      0.83      0.82        24
           6       0.74      0.74      0.74        34
           7       0.71      0.69      0.70        32
           8       0.69      0.71      0.70        28
           9       0.84      0.72      0.78        29

    accuracy                           0.79       309
   macro avg       0.79      0.79      0.79       309
weighted avg       0.79      0.79      0.79       309
```

Fig. 23: Tuned Logistic Regression Classification Report

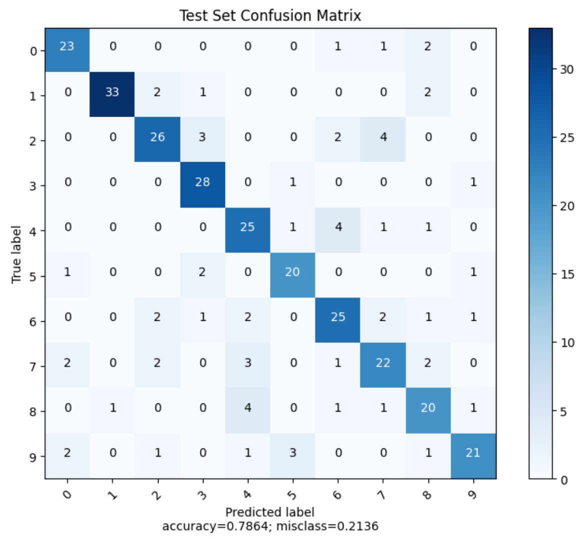


Fig. 24: Tuned Logistic Regression Confusion Matrix

B. *Convolutional Neural Network*

To improve the performance of the previous model, we developed again another function that trains the model 20 times, varying randomly the number of filters of each convolutional layer, the dropout rate of each dropout layer, the number of units of each dense layer, and the learning_rate, as we can see in the table II.

Hyperparameter	Values
conv_1_filters	{16, 32, 48, 64}
conv_2_filters	{16, 32, 48, 64}
conv_3_filters	{16, 32, 48, 64}
conv_4_filters	{16, 32, 48, 64}
conv_5_filters	{16, 32, 48, 64}
conv_6_filters	{32, 64, 96, 128}
conv_7_filters	{16, 32, 48, 64}
dropout_1	[0.1 - 0.5]
dropout_2	[0.1 - 0.5]
dropout_3	[0.1 - 0.5]
dense_1_units	{64, 128, 192 e 256}
dense_2_units	{32, 64, 96 e 128}
dense_3_units	{16, 32, 48 e 64}
learning_rate	[1^{-5} - 1^{-2}]

TABLE II: CNN Parameters

After trying 20 different combinations, the combination of parameters that gave the best results was the following:

- **conv_1_filters -** 64
- **conv_2_filters -** 48
- **conv_3_filters -** 48
- **conv_4_filters -** 64
- **conv_5_filters -** 32
- **conv_6_filters -** 128
- **conv_7_filters -** 32
- **dropout_1 -** 0.5
- **dropout_2 -** 0.5
- **dropout_3 -** 0.4
- **dense_1_units -** 256
- **dense_2_units -** 64
- **dense_3_units -** 16
- **learning_rate -** 0.000287

By analyzing the hyper tuning trials, we were able to draw some conclusions about how certain parameters significantly affected the model's performance:

- We observed significant performance variation with different learning rates. Very low learning rates (such as 1.11e-05) and very high ones (such as 0.009) resulted in poor performance. Moderate values closer to 0.0001 to 0.003 seem more consistent in producing better results.
- Models with larger units in dense layers (for example, 256) often showed better performance.
- Variations in dropout usage have a considerable impact on controlling overfitting. Very high dropout values can lead to the loss of important information during training, while very low values may not be sufficient to mitigate overfitting.

The training model with the hyperparameters mentioned above was able to deliver even better results than the previous model, which already performed well. In the graphs below, we can see that the accuracy and loss values of the validation set closely follow those of the training set, and these values are quite high, indicating that we are seeing a good fit and that this model should be able to generalize well to new data.

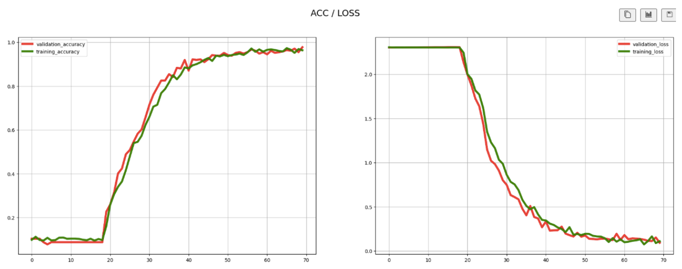
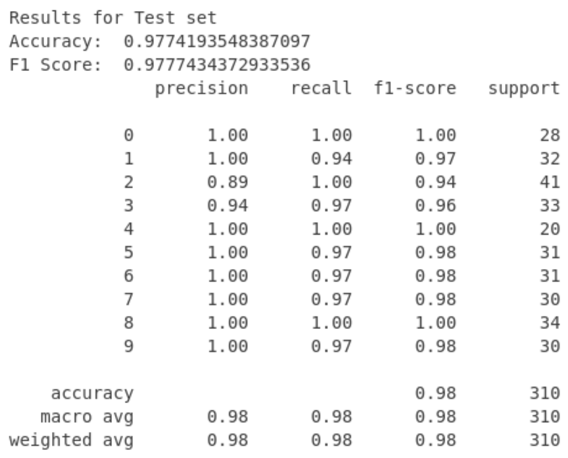


Fig. 25: Evolution of Accuracy and Loss During Training

There was an increase of 0.03 in both accuracy and F1 score compared to the model without hyperparameter tuning, resulting in both reaching 0.98, each is a very positive result:



```
Results for Test set
Accuracy:  0.9774193548387097
F1 Score:  0.9777434372933536
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        28
           1       1.00      0.94      0.97        32
           2       0.89      1.00      0.94        41
           3       0.94      0.97      0.96        33
           4       1.00      1.00      1.00        20
           5       1.00      0.97      0.98        31
           6       1.00      0.97      0.98        31
           7       1.00      0.97      0.98        30
           8       1.00      1.00      1.00        34
           9       1.00      0.97      0.98        30

    accuracy                           0.98       310
   macro avg       0.98      0.98      0.98       310
weighted avg       0.98      0.98      0.98       310
```

Fig. 26: Tuned CNN Classification Report

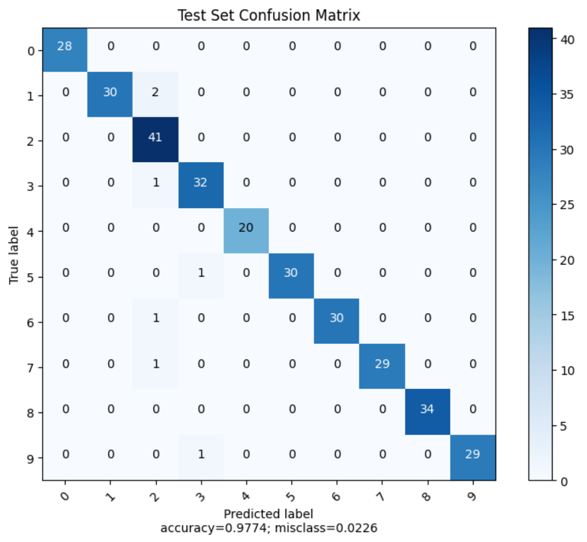


Fig. 27: Tuned CNN Confusion Matrix

C. *Artificial Neural Network*

To improve the performance of the previous model, we developed another function that trains the model 20 times, randomly varying the number of neurons, the L2 regularization rate and the dropout rate for each hidden layer (Dense), and the learning_rate, as we can see in the table III.

Hyperparameter	Values
units_layer_1	{256, 512, 768 e 1024}
l2_layer_1	$[1^{-4}$ - $1^{-2}]$
dropout_layer_1	[0.1 - 0.5]
units_layer_2	{128, 256, 384 e 512}
l2_layer_2	$[1^{-4}$ - $1^{-2}]$
dropout_layer_2	[0.1 - 0.5]
learning_rate	$[1^{-5}$ - $1^{-2}]$

TABLE III: ANN Parameters

After trying 20 different combinations, the combination of parameters that gave the best results was the following:

- **units_layer_1 -** 1024
- **l2_layer_1 -** 0.0016786
- **dropout_layer_1 -** 0.3
- **units_layer_2 -** 384
- **l2_layer_2 -** 0.0006236
- **dropout_layer_2 -** 0.2
- **learning_rate -** 7.8111^{-05}

By analyzing the hyper tuning trials, we were able to draw some conclusions about how certain parameters significantly affected the model's performance:

- Learning rate remains one of the most impactful hyper-parameters. Variations in the learning rate showed significant differences in performance metrics, where very high or very low rates led to worse results due, respectively, to training instability or very slow convergence.
- The number of units per layer had a considerable effect on the results. Models with a higher number of units tend to show better performance, indicating that increased model capacity is beneficial up to a certain point.
- Dropout rates also showed to have a substantial impact, with higher rates being useful in some cases to mitigate overfitting, especially in models with many units.

Compared to the graphs for the previous model, here we can see a slight improvement in the convergence values for accuracy and loss, both for validation and training. However, the small difference between the validation and training values persists:

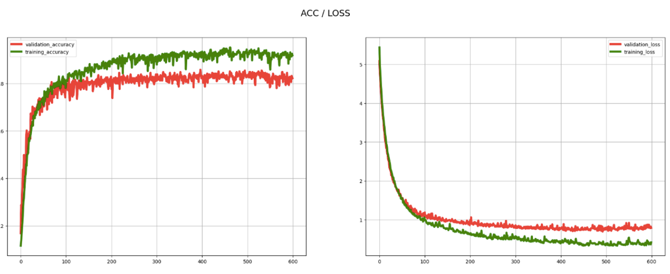
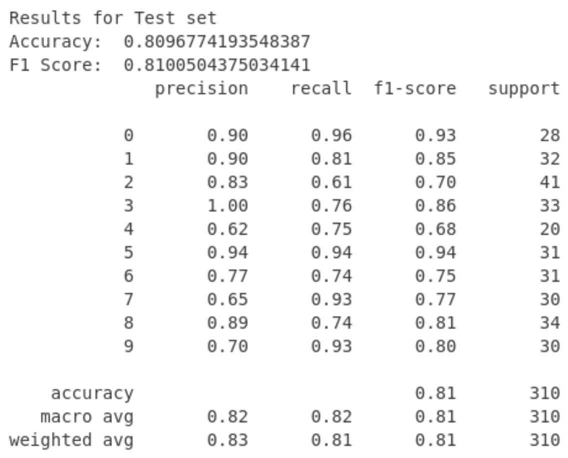


Fig. 28: Evolution of Accuracy and Loss During Training

Due to what was mentioned earlier, the accuracy and F1 score values did not show significant changes compared to the previous model, again obtaining 0.80 for both accuracy and F1 score:


```
Results for Test set
Accuracy:  0.8096774193548387
F1 Score:  0.8100504375034141
                 precision    recall  f1-score   support

           0       0.90      0.96      0.93        28
           1       0.90      0.81      0.85        32
           2       0.83      0.61      0.70        41
           3       1.00      0.76      0.86        33
           4       0.62      0.75      0.68        20
           5       0.94      0.94      0.94        31
           6       0.77      0.74      0.75        31
           7       0.65      0.93      0.77        30
           8       0.89      0.74      0.81        34
           9       0.70      0.93      0.80        30

    accuracy                           0.81       310
   macro avg       0.82      0.82      0.81       310
weighted avg       0.83      0.81      0.81       310
```

Fig. 29: Tuned ANN Classification Report

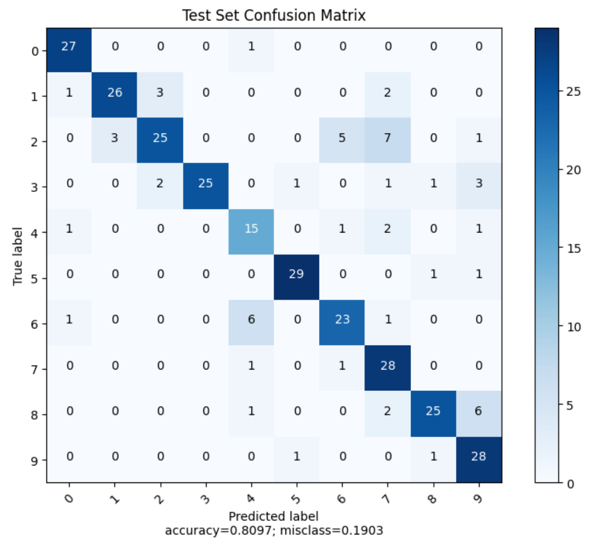


Fig. 30: Tuned ANN Confusion Matrix

D. Support Vector Machine

To improve the performance of the previous model, we developed another function that searches for the best hyperparameters. However, this time, we tested all possible combinations using the following parameters, presented in the table IV, resulting in a total of 192 combinations. We repeated this process 5 times for each combination, resulting in a total of 960 training sets.

Hyperparameter	Values
C	{0.1, 1, 10, 100}
kernel	{rbf, poly, sigmoid, linear}
gamma	{scale, auto, 0.01, 0.001}
degree	{2, 3, 4}

TABLE IV: SVM Parameters

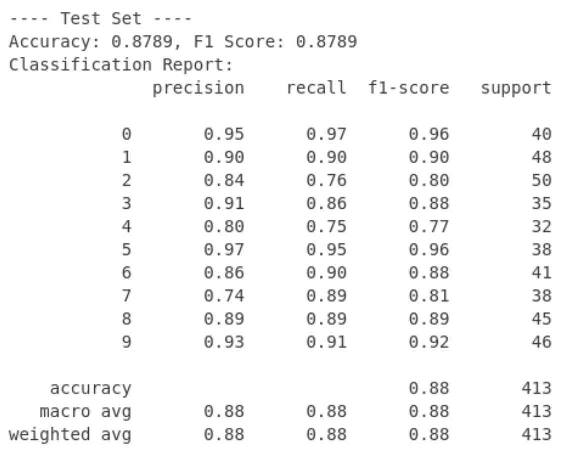
After trying the 192 combinations, the combination of parameters that gave the best results was the following:

- **C -** 10

- **degree -** 2
- **gamma -** 0.01
- **kernel -** rbf

Since the library used for hyperparameter tuning of this model does not allow us to save logs of the tuning trials, and due to the high number of trials that were performed, we did not conduct any analysis on the influence that certain hyperparameters could have on the model's performance.

In comparison with the previous model, this model with hyperparameter tuning achieved an increase of 0.08 in both accuracy and F1 score values, scoring 0.88 in both, which is an improvement in the model's performance.



```
---- Test Set ----
Accuracy: 0.8789, F1 Score: 0.8789
Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.97      0.96        40
           1       0.90      0.90      0.90        48
           2       0.84      0.76      0.80        50
           3       0.91      0.86      0.88        35
           4       0.80      0.75      0.77        32
           5       0.97      0.95      0.96        38
           6       0.86      0.90      0.88        41
           7       0.74      0.89      0.81        38
           8       0.89      0.89      0.89        45
           9       0.93      0.91      0.92        46

    accuracy                           0.88       413
   macro avg       0.88      0.88      0.88       413
weighted avg       0.88      0.88      0.88       413
```

Fig. 31: Tuned SVM Classification Report

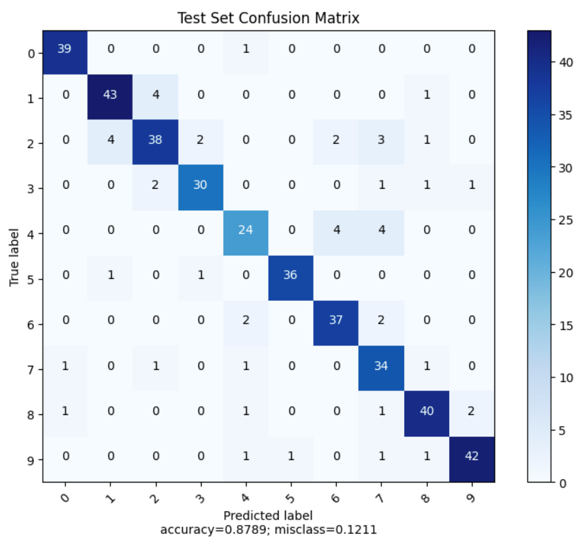


Fig. 32: Tuned SVM Confusion Matrix

VIII. Tuned Models Using Augmented Dataset

With small datasets, models learn the training data very well, sometimes even memorizing it. This leads to an overfitting problem, where the model fits too closely to the training data and fails to generalize well to new data.

Recognizing that the dataset size might be limiting the performance of our models, we conducted dataset augmentation and confirmed our suspicions, as the overall performance of all models increased.

A. Logistic Regression One-vs-all multi-class classification

In contrast to what was observed in the previous versions of this model, we can now see that the accuracy and loss values of both validation and training are much closer. This indicates a reduction in the overfitting that was present.

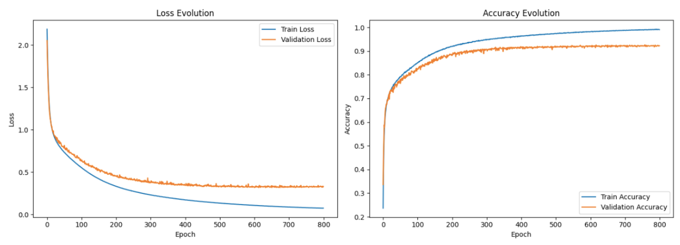
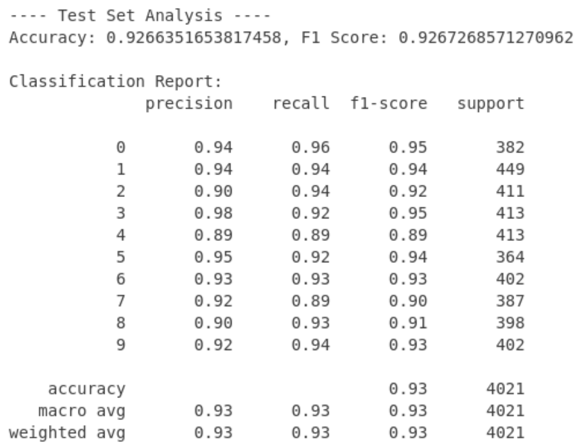


Fig. 33: Evolution of Accuracy and Loss During Training

Compared to the accuracy and F1 score values of the tuned model with the normal dataset, this model saw an increase of 0.14, resulting in an accuracy and F1 score of 0.93. It's worth noting that this model showed the highest increase in these values compared to its previous version:



```
---- Test Set Analysis ----
Accuracy: 0.9266351653817458, F1 Score: 0.9267268571270962

Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.96      0.95       382
           1       0.94      0.94      0.94       449
           2       0.90      0.94      0.92       411
           3       0.98      0.92      0.95       413
           4       0.89      0.89      0.89       413
           5       0.95      0.92      0.94       364
           6       0.93      0.93      0.93       402
           7       0.92      0.89      0.90       387
           8       0.90      0.93      0.91       398
           9       0.92      0.94      0.93       402

    accuracy                           0.93      4021
   macro avg       0.93      0.93      0.93      4021
weighted avg       0.93      0.93      0.93      4021
```

Fig. 34: Tuned Logistic Regression with Augmented Dataset Classification Report

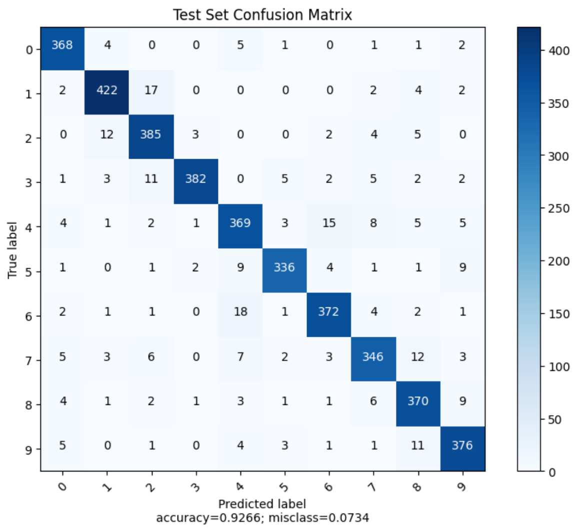


Fig. 35: Tuned Logistic Regression with Augmented Dataset Confusion Matrix

B. Convolutional Neural Network

Similar to the previous model, the accuracy and loss values of the validation and training sets remain very close, indicating that we still have a good fit.

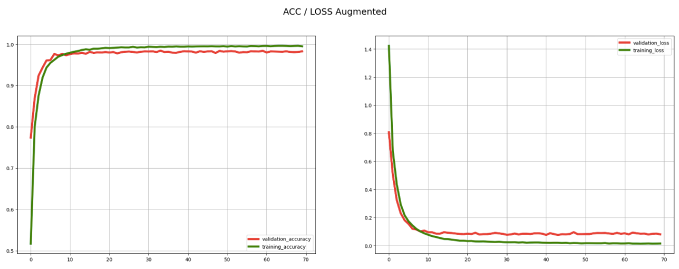
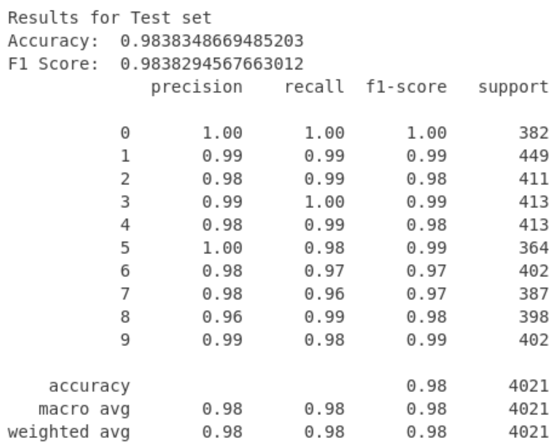


Fig. 36: Evolution of Accuracy and Loss During Training

There was still a very slight improvement in the accuracy and F1 score values of the model compared to the previous version, reaching a value of 0.98. The improvement was not very significant since the model already had good performance and was already fitting well.



```
Results for Test set
Accuracy:  0.9838348669485203
F1 Score:  0.9838294567663012
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       382
           1       0.99      0.99      0.99       449
           2       0.98      0.99      0.98       411
           3       0.99      1.00      0.99       413
           4       0.98      0.99      0.98       413
           5       1.00      0.98      0.99       364
           6       0.98      0.97      0.97       402
           7       0.98      0.96      0.97       387
           8       0.96      0.99      0.98       398
           9       0.99      0.98      0.99       402

    accuracy                           0.98      4021
   macro avg       0.98      0.98      0.98      4021
weighted avg       0.98      0.98      0.98      4021
```

Fig. 37: Tuned CNN with Augmented Dataset Classification Report

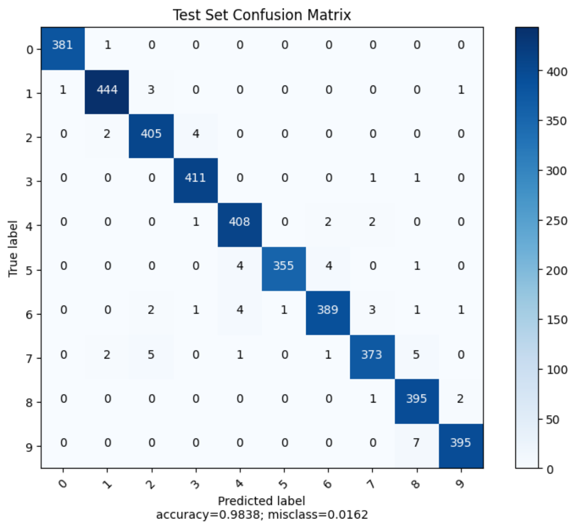


Fig. 38: Tuned CNN with Augmented Dataset Confusion Matrix

C. Artificial Neural Network

Compared to the tuned model with the normal dataset, the accuracy and loss values of the validation set are now tracking the training values, and their convergence has also improved:

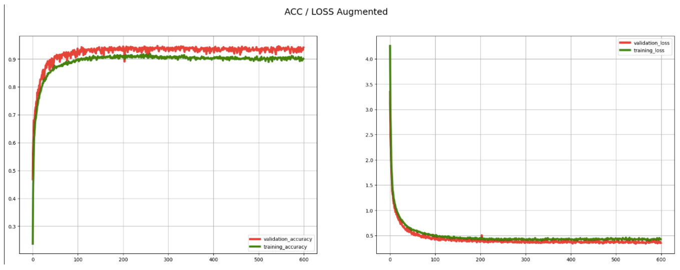
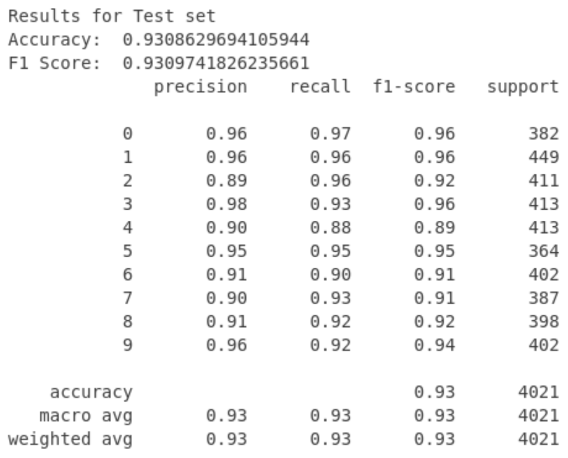


Fig. 39: Evolution of Accuracy and Loss During Training

Compared to its previous version, this model also experienced a considerable increase in both accuracy and F1 score values, reaching 0.93 in both metrics, which is 0.12 higher than the tuned model with the normal dataset.

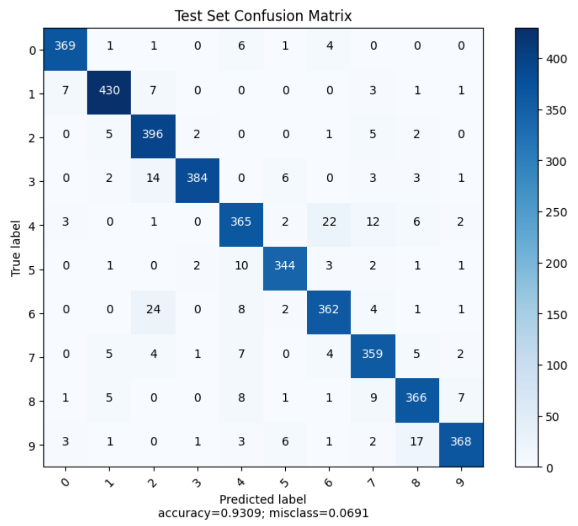


```
Results for Test set
Accuracy:  0.9308629694105944
F1 Score:  0.9309741826235661
              precision    recall  f1-score   support

           0       0.96      0.97      0.96       382
           1       0.96      0.96      0.96       449
           2       0.89      0.96      0.92       411
           3       0.98      0.93      0.96       413
           4       0.90      0.88      0.89       413
           5       0.95      0.95      0.95       364
           6       0.91      0.90      0.91       402
           7       0.90      0.93      0.91       387
           8       0.91      0.92      0.92       398
           9       0.96      0.92      0.94       402

    accuracy                           0.93      4021
   macro avg       0.93      0.93      0.93      4021
weighted avg       0.93      0.93      0.93      4021
```

Fig. 40: Tuned ANN with Augmented Dataset Classification Report

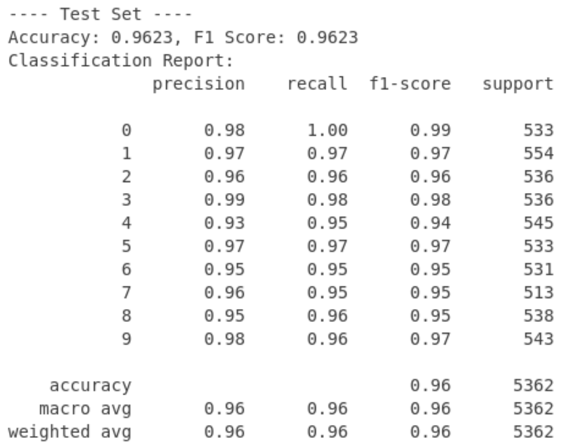


	0	1	2	3	4	5	6	7	8	9
0	369	1	1	0	6	1	4	0	0	0
1	7	430	7	0	0	0	0	3	1	1
2	0	5	396	2	0	0	1	5	2	0
3	0	2	14	384	0	6	0	3	3	1
4	3	0	1	0	365	2	22	12	6	2
5	0	1	0	2	10	344	3	2	1	1
6	0	0	24	0	8	2	362	4	1	1
7	0	5	4	1	7	0	4	359	5	2
8	1	5	0	0	8	1	1	9	366	7
9	3	1	0	1	3	6	1	2	17	368

Fig. 41: Tuned ANN with Augmented Dataset Confusion Matrix

D. Support Vector Machine

In comparison with the model tuned with the original dataset, this model achieved an increase of 0.08 in both accuracy and F1 score values, reaching an accuracy and F1 score of 0.96. This indicates that the model has made a good adjustment and is generalizing well to new data.

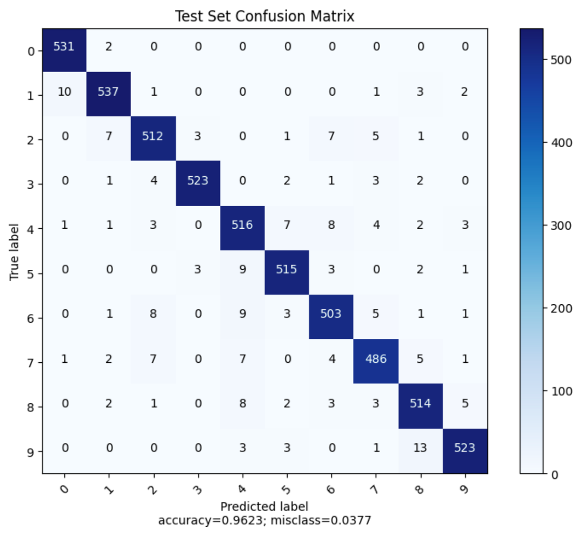


```
---- Test Set ----
Accuracy: 0.9623, F1 Score: 0.9623
Classification Report:
              precision    recall  f1-score   support

           0       0.98      1.00      0.99       533
           1       0.97      0.97      0.97       554
           2       0.96      0.96      0.96       536
           3       0.99      0.98      0.98       536
           4       0.93      0.95      0.94       545
           5       0.97      0.97      0.97       533
           6       0.95      0.95      0.95       531
           7       0.96      0.95      0.95       513
           8       0.95      0.96      0.95       538
           9       0.98      0.96      0.97       543

    accuracy                           0.96      5362
   macro avg       0.96      0.96      0.96      5362
weighted avg       0.96      0.96      0.96      5362
```

Fig. 42: Tuned SVM with Augmented Dataset Classification Report



	0	1	2	3	4	5	6	7	8	9
0	531	2	0	0	0	0	0	0	0	0
1	10	537	1	0	0	0	0	1	3	2
2	0	7	512	3	0	1	7	5	1	0
3	0	1	4	523	0	2	1	3	2	0
4	1	1	3	0	516	7	8	4	2	3
5	0	0	0	3	9	515	3	0	2	1
6	0	1	8	0	9	3	503	5	1	1
7	1	2	7	0	7	0	4	486	5	1
8	0	2	1	0	8	2	3	3	514	5
9	0	0	0	0	3	3	0	1	13	523

Fig. 43: Tuned SVM with Augmented Dataset Confusion Matrix

IX. Model Comparison

Looking at fig. 44, we can see that in general there was an improvement in accuracy between the models with the default hyperparameters with the original dataset and the models with the tuned hyperparameters with the original dataset, and then between the latter and the models with tuned hyperparameters and augmented dataset, showing that these techniques were effective in improving the performance of our models.

We also observed that among all the models we built, the CNN models are the ones that most accurately identified the digits in the images, which is not surprising to us since CNNs are widely used in image recognition, analysis, and classification.

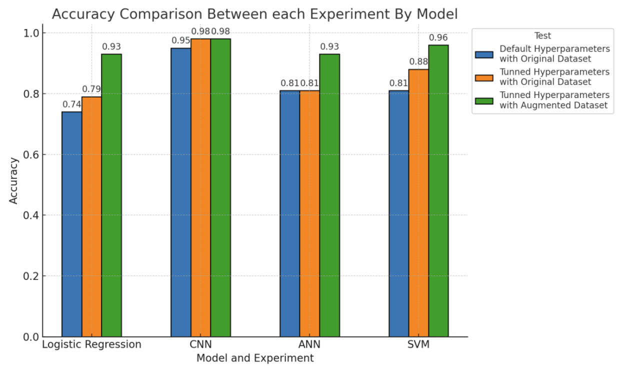


Fig. 44: Accuracy Comparison Between each Experiment by Model

In the fig. 45 below, we have represented all the accuracies achieved by the models with hyperparameter tuning using the augmented dataset, showing that all models achieved high performance (above 0.9 accuracy) when applying these techniques.

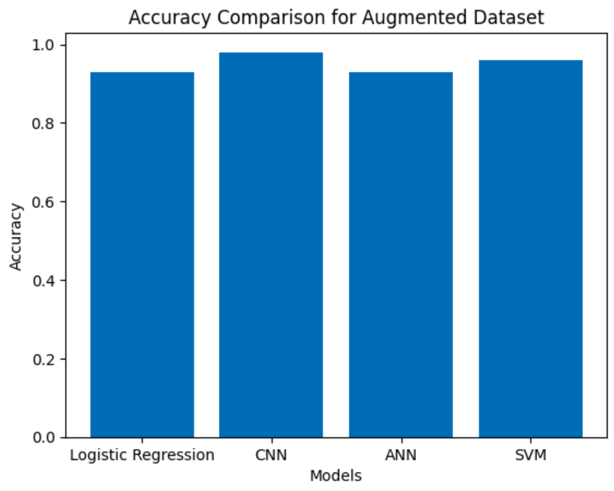


Fig. 45: Final Results using hyperparameter tuning and dataset augmentation

X. Novelty and Contributions

While there are numerous studies focusing on sign language recognition, our approach is distinguished by its attempt to identify all ten digits from sign language images using a comprehensive set of machine learning techniques. Prior work on the Kaggle platform [8] has often limited the scope to classifying between two digits. Although these studies provide a valuable foundation, they do not address the complexity introduced by a ten-class system. Our work expands on these foundational models by not only increasing the number of classes but also by enhancing the model's ability to generalize through sophisticated preprocessing and augmentation techniques.

In our initial experiments, Logistic Regression was the least accurate. However, with systematic hyperparameter tuning and extensive data augmentation, we significantly improved its performance—from a baseline accuracy of 0.74 to 0.93. This enhancement underscores the impact of methodical tuning and dataset expansion on the model's ability to generalize, a key challenge noted in related work [9]

Our contributions are threefold:

- **Extended Classification Capability:** Unlike many models that focus on binary classification of sign language digits, our models are trained to distinguish among all ten digits, tackling a more complex and realistic task.
- **Advanced Preprocessing Techniques:** We demonstrated that preprocessing steps such as mean image analysis and pixel value distribution analysis are critical for understanding and preparing our data effectively, leading to better model performance.
- **Effective Use of Data Augmentation:** By implementing data augmentation, we addressed the overfitting issue prevalent in models trained on small datasets. Our results validate that augmentation can significantly enhance model robustness and accuracy.

These enhancements not only improve upon the existing approaches by offering a more robust and accurate system but also contribute practical insights into the challenges of working with sign language digit datasets. Such advancements are crucial for developing assistive technologies that are both reliable and scalable.

To achieve the results presented in this work, we leveraged several advanced software libraries and tools, which were integral to our machine learning pipeline from model creation to hyperparameter tuning.

A. Machine Learning Libraries

Scikit-learn provides a wide array of efficient tools for machine learning and statistical modeling including classification, regression, clustering, and dimensionality reduction. We particularly utilized its `SVC` module to implement our Support Vector Machine models and `GridSearchCV` for exhaustive parameter tuning to find the optimal settings for our models. This library's robust, well-optimized functions enable rigorous statistical evaluations to be carried out with minimal code.

B. Deep Learning Frameworks

TensorFlow and **Keras** were used for building, training, and validating deep learning models such as Convolutional Neural Networks (CNNs) and Artificial Neural Networks (ANNs). TensorFlow offers a comprehensive, flexible ecosystem of tools, libraries, and community resources that lets researchers push the state-of-the-art in ML, and makes it easy to deploy AI applications. Keras, a high-level neural networks API, written in Python and capable of running on top of TensorFlow, simplifies many operations and is widely used for rapid prototyping.

C. Model Building and Optimization

Our models were defined and compiled using Keras functionalities such as `Sequential`, `Conv2D`, MaxPooling2D, Dense, Flatten, and Dropout. These classes and functions provide an intuitive way to stack layers for deep learning models in a modular fashion. We used Adam optimizer for its adaptive learning rate capabilities, which make it more efficient than classical stochastic gradient descent.

D. Hyperparameter Tuning

To fine-tune the hyperparameters of our deep learning models, we employed RandomSearch from Keras Tuner. This tuner explores a given parameter space randomly and identifies the combination of parameters that results in the optimal model performance. The use of this tuner helped in systematically exploring wide-ranging configurations and significantly enhanced our models' accuracy.

E. Enhancing Model Training

EarlyStopping is another critical feature used during the training of our models. It is a form of regularization used to avoid overfitting by halting the training process if the model's performance ceases to improve on a held-out validation dataset for a specified number of epochs. This approach ensures that we maintain an efficient training process and prevent our models from learning noise in the training data.

These tools collectively supported the robust development of our machine learning models, enabling complex operations to be performed with high levels of abstraction and efficiency. The use of such advanced libraries and frameworks not only facilitated the rapid development and iteration of models but also ensured that our approaches are grounded in the latest and most effective practices in machine learning.

XI. Conclusions

In conclusion, the CNN algorithm, with the correct configuration, can achieve the best accuracy possible for this kind of problem (with image recognition). Yet, we learned that certain strategies could be applied to improve the other algorithms, like hyperparameter tuning and data augmentation.

References

[1] T. Starner, "Visual recognition of american sign language using hidden markov models," Ph.D. dissertation, Massachusetts Institute of Technology, 1995.

[2] R. Rastgoo, K. Kiani, and S. Escalera, "Sign language recognition: A deep survey," *Expert Systems with Applications*, vol. 164, p. 113794, 2021.

[3] N. Adaloglou, T. Chatzis, I. Papastratis, A. Stergioulas, G. T. Papadopoulos, V. Zacharopoulou, G. J. Xydopoulos, K. Atzakas, D. Papazachariou, and P. Daras, "A comprehensive study on deep learning-based methods for sign language recognition," *IEEE Transactions on Multimedia*, vol. 24, pp. 1750–1762, 2021.

[4] O. Koller, H. Ney, and R. Bowden, "Deep hand: How to train a cnn on 1 million hand images when your data is continuous and weakly labelled," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 3793–3802.

[5] "Sign language digits dataset," Kaggle, Dec. 2017. [Online]. Available: https://www.kaggle.com/datasets/ardamavi/sign-language-digits-dataset/data

[6] Efeergun, "Deep learning on sign language digits," Kaggle, November 2018. [Online]. Available: https://www.kaggle.com/code/efeergun96/deep-learning-on-sign-language-digits/notebook

[7] Pamhohhgkgm, "Deep neural network for sign language dataset," Kaggle, Oct 2019. [Online]. Available: https://www.kaggle.com/code/pamhohhgkgm/deep-neural-network-for-sign-language-dataset

[8] M. Serce, "Sign language with logistic regression," Kaggle, August 2020. [Online]. Available: https://www.kaggle.com/code/mete6944/sign-language-with-logistic-regression

[9] D. Mathew, "Sign language detection using data augmentation," Kaggle, September 2018. [Online]. Available: https://www.kaggle.com/code/honeysingh/sign-language-detection-using-data-augmentation