MACHINE LEARNING CHALLENGE

PART 1 - Feature Engineering

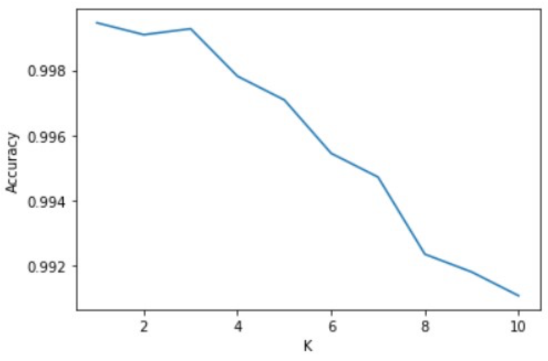
The decision was made to split the entire training dataset into training and validation subsets with a proportion of 0.8 and 0.2, respectively. In the training data, all classes occurred between 957 and 1294 times, leading to the assumption of balanced classes, and consequently, macro accuracy was used. The modelling strategy, initially focused on the KNN model, prompted consideration of engineering some features of the datasets. However, it was concluded that this step should be avoided for the following reasons:

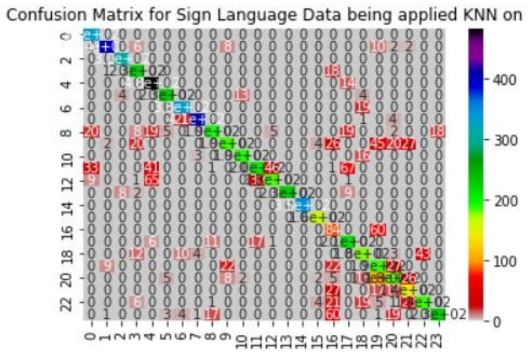
1) When modeling with KNN, the primary goal was to focus on the distance between pixels, and thus, transforming or extracting pixel values did not align with the analytical framework. Additionally, concerns arose regarding the potential impact of removing noisy images from the training subset, which could drastically alter data variance. Therefore, feature engineering for KNN was not included in the project's scope.
2) Conversely, for modeling with CNN, reliance was placed on the CNN feature extractor capabilities, allowing for automatic extraction of features during the training process, rather than implementing it manually. For the CNN, in the preprocessing stage, the LabelBinarizer method was used to transform the labels, which were Pandas objects, into Numpy arrays. After reshaping the images into the size 28x28x1 from 1D to 3D, the data was prepared for input into the CNN model.

PART 2 – Learning Models and Algorithms

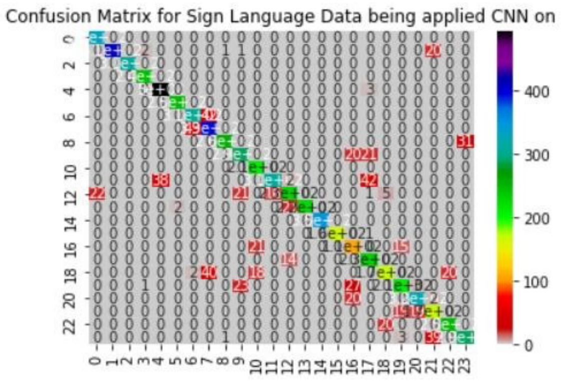
After coding and evaluating the performance metrics of the selected models on the training set, it was observed that both the KNN and CNN models demonstrated robust accuracy scores. Subsequently, both final models were trained on the combined training and validation sets. The decision to select these two models for Task 1 was driven by the following factors:

1) KNN was chosen due to its solid performance on the test set. The accuracy plot for KNN hyperparameter tuning indicated that out-of-sample validation with K=1 was the optimal choice, supported by its related confusion matrix. The accuracy on the validation set was 0.9995, while the final accuracy for KNN was 0.8104. Additionally, KNN operates by locating all data points in a multidimensional space (in this case, 784-dimensional) and calculating the distance from a new input to every data point, selecting the best distance based on the majority vote of the k nearest neighbors.





2) Traditionally, neural networks (NNs) are considered an advanced machine learning technique due to their high capability in identifying important features. CNN, a type of deep learning model specifically designed for processing data with a gridded pattern (such as images), is inspired by the structure of the brain's visual cortex and is designed to learn spatial hierarchies of features automatically and adaptively, from low- to high-level patterns. Consequently, a CNN was chosen because it is particularly suited for images and can leverage spatial dependencies for predictions. The final accuracy for CNN on the test set was 0.9078.



By comparing the two final confusion matrices, it became apparent that both models encountered difficulties with classes 16 and 12.

PART 3 – Parameters Tuning

Hyperparameters for the two models were tuned as follows:

1) For KNN, as previously mentioned, the hyperparameter tuning process was performed by selecting the best K-value, resulting in K=1 with an accuracy score of 0.8104.

2) For CNN, Keras, one of the Python libraries used for fast computations, was employed along with Grid-Search, varying batch sizes and the number of epochs across 1 and 2 layers, as detailed in the summary table below. The highest test-accuracy score was recorded for a single CNN layer, with a batch size of 512 and 5 epochs, resulting in a final score of 0.9078. All combinations reached a validation accuracy score of 1.0, leading to the decision based on the shortest training time.

	1 Convolutional Layer			*2 Convolutional Layers*		
# Batch	**5 epochs**	**10 epochs**	**15 epochs**	**5 epochs**	**10 epochs**	**15 epochs**
Batch size 128	Acc: 1.0 Time: 105.72s	Acc: 1.0 Time: 210.38s	Acc: 1.0 Time: 312.54s	Acc: 1.0 Time: 133.25s	Acc: 1.0 Time: 262.02s	Acc: 1.0 Time: 390.54s
Batch size 256	Acc: 1.0 Time: 103.19s	Acc: 1.0 Time: 200.07s	Acc: 1.0 Time: 301.77s	Acc: 1.0 Time: 129.06s	Acc: 1.0 Time: 256.72s	Acc: 1.0 Time: 383.12s
Batch size 512	Acc: 1.0 Time: 101.38s	Acc: 1.0 Time: 203.01s	Acc: 1.0 Time: 306.26	Acc: 1.0 Time: 128.10s	Acc: 1.0 Time: 261.28s	Acc: 1.0 Time: 380.10s

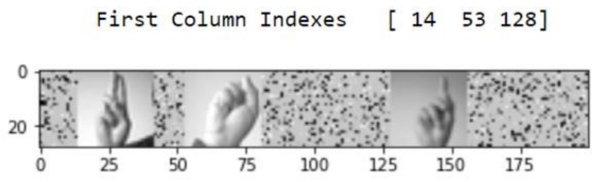
PART 4 – Performance of the Solution

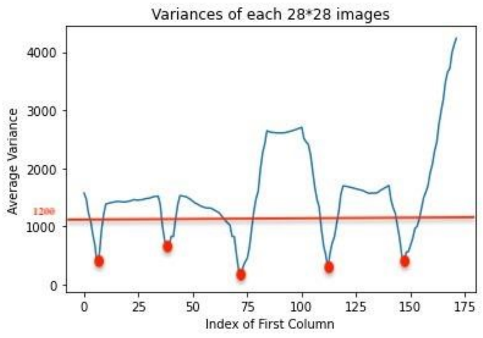
For Task 2, two different image detection models were implemented. One allowed for the extraction and prediction of the indexes of sign images, while the other method facilitated the retrieval of information from a single image without considering overlapping areas. These two methods were combined to classify different image specifications. These are the functions we implemented[1]:

1) **find_indexes**: function that locates the left edges of the images and uses a variance through the images to find the indexes for detecting images correctly 95% of the times and detects all the images with a threshold value of 1200 (see

[1] The prediction functions have a fallback-mechanism for which they add an arbitrary prediction (‘***05000811’; “FAIL” encoded in the targets***) when they do not detect any images. By doing so, we avoided the occurrence of an error when making all predictions that would crash our final file.

plots below):





2) **im_detect_cnn** and **im_detect_knn**: these two functions get as input a single 28*200 image and run the **find_indexes** function. They then create an array that contains the 28*28 images for all found indices (which flattens all of them to be able to use only the KNN). At the end, they make predictions using the KNN for single-digit classes, therefore the leading 0 is added (for example, 8 becomes “08”).

3) **im_detect_knn**: function that gets as input a single 28*200 image and runs the **find_indexes** function. It then creates an array that contains the 28*28 images for all found indices and it flattens all of them to be able to use KNN model. Finally, it makes predictions using the KNN for single-digit classes, therefore the leading 0 is added (for example, 8 becomes “08”).

4) **proba_pred**: function that classifies each 28x28 area of the sequence which has 172 images per sequence. If the probability of the predictions has a probability above the threshold using CNN, then the function keeps the prediction and skips the next 27 pixels because it assumes that they are in the same image. The function also uses image detection to separate the images contained in an input-image and then predicts each image using the CNN from task 1, but due to the encoding of the target, the labels obtained with argmax do not pay respect to classes 9 and 25.

Overall, predictions were made using two different methods: one with CNN and KNN, and three predictions with the second method by adjusting the threshold for accepting predictions. The input was a single large 28x200 image, and predictions were made for each 28x28 image within the larger image. The predicted class and its probability were stored. A recursive function was then initialized to skip the image if the probability matched a predefined threshold (0.99). The predicted probabilities were high, and a significant difference was noted between thresholds of 0.99, 0.9999, and 1.0. Therefore, high thresholds were chosen to avoid recording too many incorrect predictions. It is important to note that the two methods used for Task 2 complement each other’s weaknesses, which is why both were employed.

References:

- https://www.kaggle.com/code/madz2000/cnn-using-keras-100-accuracy
- https://www.tensorflow.org/tutorials/images/cnn
- https://towardsdatascience.com/simple-guide-to-hyperparameter-tuning-in-neural-networks-3fe03dad8594
- https://analyticsindiamag.com/hands-on-guide-to-sign-language-classification-using-cnn/