Identification of Digits from Sign Language Images

Aprendizagem Automática – Professor: Pétia Georgieva

José Santos, 98279
DETI
Universidade de Aveiro

Henrique Sousa, 98324
DETI
Universidade de Aveiro

***Abstract*—The purpose of this work is to implement and compare machine learning models capable of identifying digits from sign language images. In this paper, we tried to obtain a good result with the models using a dataset provided by Kaggle. Some changes are discussed, based on the work of others that positively affected our work.**

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

I. Introduction

In recent years, there has been growing interest in using machine learning to develop computer vision systems capable of recognizing sign language gestures. Such systems could be used to improve communication between hearing and non-hearing individuals, as well as to facilitate the development of new technologies for the deaf and hard-of-hearing community.

In this paper, we present a novel approach to digit recognition from sign language images using machine learning. We explore several different models, including neural networks, support vector machines, and decision trees, and compare their performance on a dataset of sign language images. We also investigate the impact of some preprocessing techniques.

II. State of the art

Over the past few years, there have been several studies and projects focused on recognition from sign language images using machine learning techniques. Many of these approaches have utilized deep learning methods such as convolutional neural networks (CNNs), which have been shown to be effective in image recognition tasks.

TODO: Add some references to the state of the art

III. Dataset Analysis

For the development of this work, we used a dataset provided by Kaggle that contains 2062 images of sign language digits. The dataset is well balanced, with a minimum of 204 examples for a label and a maximum of 208, ensuring that every label has a similar number of examples (Figure 1). The dataset includes data for the digits 0 to 9, resulting in 10 labels in total as we can see on the figure 2.

The images are provided in a *.npy* format, but we found it easier to work with the raw images to manipulate them and apply preprocessing techniques to improve the model's performance. The dataset's size and balanced distribution make it an ideal choice for training and evaluating machine learning models for digit recognition from sign language images.

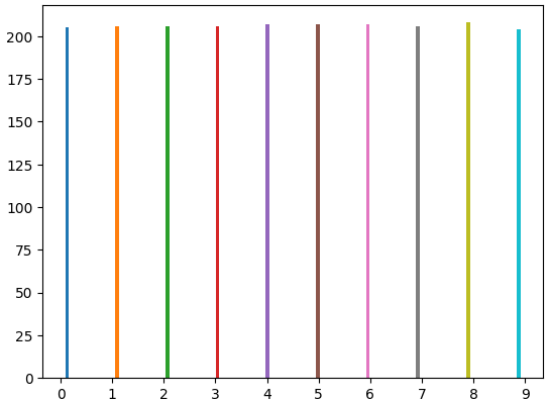

Fig. 1. Dataset balanced distribution

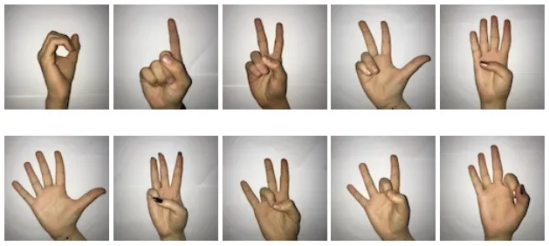

Fig. 2. Dataset label examples

IV. Dataset Preprocessing

A. Image Augmentation

The first thing we did to the images was to augment them to create more data for model training. We used *imgaug.augmenters* to perform the augmentation. The following changes were made to the images:

- ***Rotate*** - Rotate the image by a random angle between -20 and 20 degrees.
- ***Gaussian noise*** - Add gaussian noise with standard deviation of 0 to 0.05*255
- ***Gamma contrast*** - Change the contrast of the image by a random factor between 0.5 and 1.5.

For every image in the dataset, we created 5 new images with the above changes. Due to the small size of the dataset, we decided to perform the augmentation offline, before splitting the dataset into training and test data.

B. Image Preprocessing

We also resized the images to 50x50 pixels (down from the original size of 64x64), as we found that this size was sufficient for the models to achieve good results. Then, we converted the images to grayscale and flattened them, as we found that this improved the performance of the models.

Finally, we split the dataset into training and test data. We used an 80/20 split, with 80% of the data being used for training and 20% for testing. Resulting in 9897 total images for the training data and 2475 for the test data.

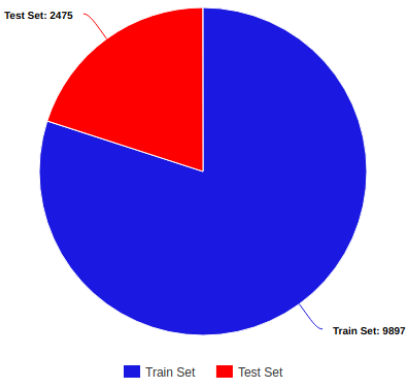

Fig. 3. Dataset Distribution

V. Models

In order to find the best training model, we tested several different models, including neural networks, support vector machines, and decision trees. We also tested different preprocessing techniques to see if they had any impact on the model's performance as discussed before. The two evaluation metrics used to measure the performance of the models are Accuracy and F1 Score. Accuracy is the proportion of correct predictions over the total number of predictions, while F1 Score considers both precision and recall of the model's predictions.

First, we tried a lot of models with their default parameters to understand which ones had value and were worth to be improved. These were the first results:

Model	Accuracy	F1 Score
Logistic Regression	0.750	0.749
Decision Tree Classifier	0.631	0.632
Random Forest Classifier	0.876	0.876
Naive Bayes	0.502	0.506
Support Vector Machines	0.888	0.888
Neural Networks (Multilayer Perceptron Classifier)	0.092	0.015

TABLE I
Models' performance with default parameters

Among all the models, Support Vector Machines has the highest accuracy and F1 Score, both of which are 0.888. Random Forest Classifier also shows a good performance with accuracy and F1 Score of 0.876. Logistic Regression comes in third place with accuracy and F1 Score of 0.750 and 0.749, respectively. The other models have lower performance, so for the next step, we will skip the Decision Tree Classifier and Naive Bayes. We will still try to find ideal parameters for the MLP Classifier because the model ended up predicting the same label for each example. This is probably because the model was not trained properly, so we will give it a chance and try to improve it.

A. Hyperparameter Tuning & Cross-Validation

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. The purpose of cross-validation is to estimate how well a machine learning model will generalize to new data. By evaluating the model on multiple subsets of the data, it is less likely that the model's performance is biased towards a particular subset of the data. Cross-validation is commonly used in machine learning to evaluate the performance of different models or to compare the performance of different hyperparameters. To implement these methods 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. The function also performs cross-validation on the training data to evaluate the model's performance. The following code shows the function's implementation:

```
def hyperparameters(model, params, X, y):
    model = GridSearchCV(model, params,
    scoring="accuracy")
    model.fit(X, y)
    print("Best Parameters:")
    print(model.bestparams)
```

These function is going to be applied in the models with different parameters to find the best model for each one.

B. Support Vector Machines

Support Vector Machines (SVM) proved to be the best-performing model for the classification task, with an initial accuracy and F1 Score of 88.8% using default parameters. In an effort to improve the performance of the SVM model, we experimented with different values for the C, kernel, degree, and gamma parameters, as shown in Table V-B.

Parameter	Values	Best Value
C	[0.001, 0.01, 0.1, 1, 10, 100, 1000]	100
kernel	[linear, poly, rbf, sigmoid]	rbf
degree	[2, 3, 4, 5]	2
gamma	[scale, auto]	scale

TABLE II
SVM Parameters

After running several iterations, we were able to identify the optimal parameter values, which resulted in a significant improvement in performance. The best-performing parameters were:

- **C** = 100

- **kernel** = rbf
- **degree** = 2
- **gamma** = scale

The SVM model with these parameters achieved an accuracy and F1 Score of 95%, which represents a significant improvement over the default parameters. We can better visualize the performance of the SVM model by looking at the confusion matrix, as shown in Figure 4.

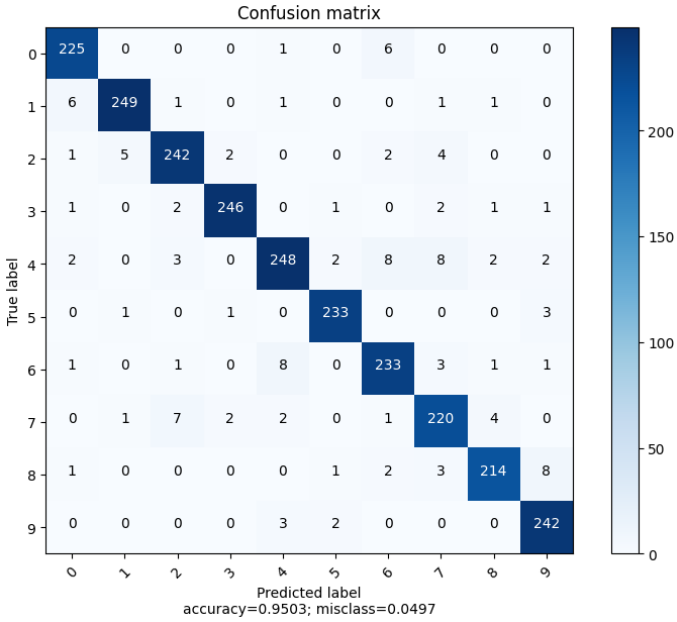

Fig. 4. SVM Confusion Matrix

As we can see in the confusion matrix, the SVM model can correctly classify most of the images. Some classes are misclassified. This is probably because the images of these classes are very similar to each other, so it is not surprising that the model has a hard time classifying them correctly. Looking

Label	Precision	Recall	F1 Score
0	0.95	0.97	0.96
1	0.97	0.96	0.97
2	0.95	0.95	0.95
3	0.98	0.97	0.97
4	0.94	0.90	0.92
5	0.97	0.98	0.98
6	0.92	0.94	0.93
7	0.91	0.93	0.92
8	0.96	0.93	0.95
9	0.94	0.98	0.96

TABLE III
SVM CLASSIFICATION REPORT

at the precision scores, which measure the proportion of true positives among all samples predicted as positive, we can see that the model performed well for most classes, with scores ranging from 0.91 to 0.98. This indicates that when the model predicted a certain class, it was usually correct.

The recall scores, which measure the proportion of true positives among all actual positive samples, are also high for most classes, ranging from 0.90 to 0.98. This suggests that the model was able to identify most instances of each class correctly.

In addition to high precision and recall scores, the F1-score, which balances these two metrics, also indicates strong performance for most classes, with scores ranging from 0.92 to 0.98. This suggests that the model achieved a good trade-off between precision and recall, and is able to correctly classify samples while minimizing false positives and false negatives. Overall, these results suggest that the classification model was effective at distinguishing between the different classes in the dataset, with high levels of precision, recall, and F1-score for most classes.

C. Random Forest Classifier

The Random Forest Classification model was the second best model with the default parameters and was very close to the SVM model, initially. The accuracy and F1 Score were 88.8%. In an effort to improve the accuracy of this model. We experimented with different values for the model parameters, as shown in Table V-C.

Parameter	Values	Best Value
n_estimators	[50, 100, 200, 500]	500
criterion	[gini, entropy]	entropy
max_depth	[None, 5, 10, 20]	None
min_samples_split	[2, 5, 10]	2
min_samples_leaf	[1, 2, 4]	1
max_features	[auto, sqrt, log2]	auto

TABLE IV
RANDOM FOREST CLASSIFIER PARAMETERS

As we see in the table above, the best parameters for the Random Forest Classifier were:

- **n_estimators** = 500
- **criterion** = entropy
- **max_depth** = None
- **min_samples_split** = 2
- **min_samples_leaf** = 1
- **max_features** = auto

The Random Forest Classifier model with these parameters achieved an accuracy and F1 Score of 89.1%. The selection of ideal parameters only barely improved the model over the default parameters. By looking at the confusion matrix in figure 5, we can confirm that the model performed decently in classifying the images of sign language digits.

In table V-C we can check the precision, recall and f1 score for each label in the dataset.

ADD COST LOSS OVER ITERATIONS FOR EVERY MODEL

D. Multilayer Perceptron

This Neural Network approach was the model that delivered the worst results with the default parameters with a very bad accuracy and f1 score of 0.092 and 0.015, respectively. The model got these values because it was not able to predict the labels correctly and kept predicting every image with the same label. This could be indicating that our model was not able

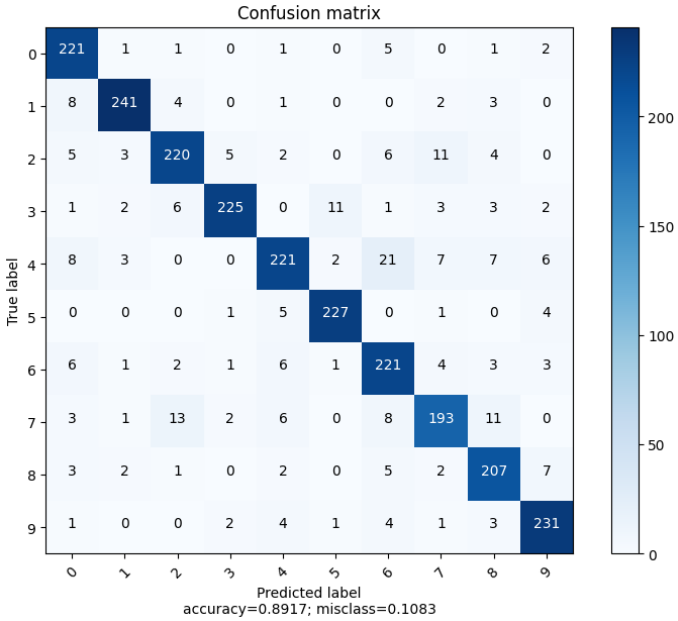

Fig. 5. Random Forest Confusion Matrix

Label	Precision	Recall	F1 Score
0	0.86	0.95	0.91
1	0.95	0.93	0.94
2	0.89	0.86	0.87
3	0.95	0.89	0.92
4	0.89	0.80	0.85
5	0.94	0.95	0.95
6	0.82	0.89	0.85
7	0.86	0.81	0.84
8	0.86	0.90	0.88
9	0.91	0.94	0.92

TABLE V
RANDOM FOREST CLASSIFICATION REPORT

to converge in order to understand how to diferentiate every label, so we didn't discard it in the initial phase and tried to get better results with it. This model is resource intensive so we were not able to test every possible combination of the parameters seen on table V-D below.

Parameter	Values	Best Value
solver	[adam, lbfgs]	lbfgs
max_iter	[250, 500, 1000]	1000
hidden_layer_sizes	From 1 to 3 hidden layers	(256, 512, 128)
activation	[relu, tanh]	relu
alpha	[0.0001, 0.001, 0.01]	0.0001
learning_rate	[constant, invscaling, adaptive]	adaptive
learning_rate_init	[0.001, 0.01, 0.1]	0.001

TABLE VI
MULTILAYER PERCEPTRON PARAMETERS

So we tried manually without using the hyperparameters function we developed some combinations with the parameters seen below. The best results we got were with the parameters seen below:

- **solver** = lbfgs
- **max_iter** = 1000
- **hidden_layer_sizes** = (256, 512, 128)
- **activation** = relu
- **alpha** = 0.0001
- **learning_rate** = adaptive
- **learning_rate_init** = 0.001

These parameters produced better results than the ones seen above, generating the confusion matrix in figure 6.

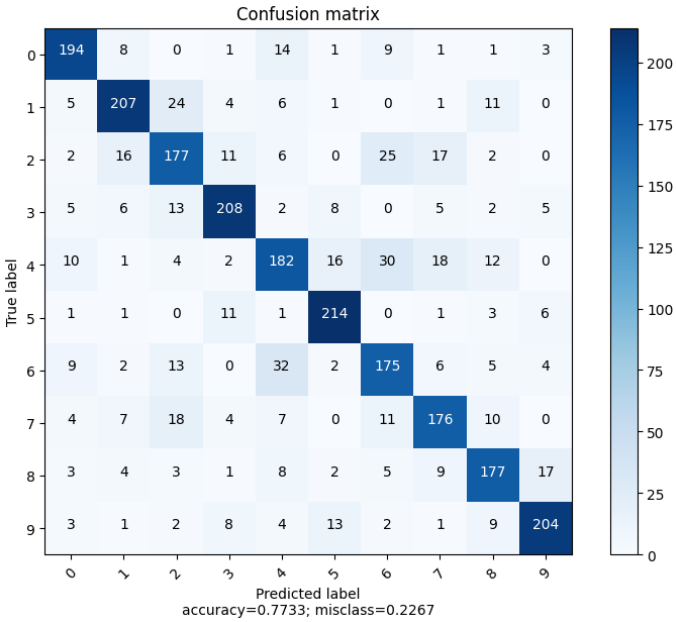

Fig. 6. MLP Confusion Matrix

As we can see, now the model can differentiate the different labels, but still misclassified a lot of examples which resulted in accuracy and f1 score of 77,3%. Although, this was the best improvement compared with the model with the default parameters it is still far from being the best-performing model for this problem. The precision, recall and f1 score for each label can be observed below in table V-D.

Label	Precision	Recall	F1 Score
0	0.82	0.84	0.83
1	0.82	0.80	0.81
2	0.70	0.69	0.69
3	0.83	0.82	0.83
4	0.69	0.66	0.68
5	0.83	0.90	0.86
6	0.68	0.71	0.69
7	0.75	0.74	0.75
8	0.76	0.77	0.77
9	0.85	0.83	0.84

TABLE VII
MULTILAYER PERCEPTRON REPORT

E. Logistic Regression

Logistic Regression with the default parameters got an accuracy and f1 score of 0.75 and 0.74, respectively. To improve the model we tried to change its parameters.

Parameter	Values	Best Value
solver	[lbfgs]	lbfgs
max_iter	[100, 400, 800]	1000
C	[0.1, 1, 10, 100, 1000]	10
class_weight	[balanced]	balanced
penalty	[l2]	l2

TABLE VIII
LOGISTIC REGRESSION PARAMETERS

In this case the parameters were a little bit more limited, because has we have a multiclass problem and a small dataset *lbfgs* was the only viable option for the solver parameter, which limited our penalty parameter to *l2* and the class_weight parameter to *balanced*. So we tried to vary the max_iter and C parameters, and the set of parameter with the best results were:

- **solver** = lbfgs
- **max_iter** = 1000
- **C** = 10
- **class_weight** = balanced
- **penalty** = l2

These parameters only got a small increase in performance, achieving an accuracy and f1 score of 0.78 and generating the confusion matrix in figure 7.

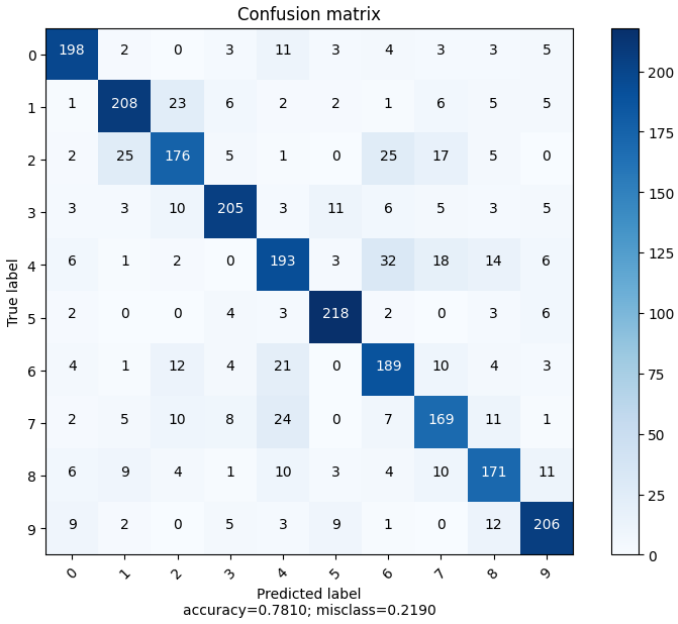

Fig. 7. Logistic Regression Confusion Matrix

The precision, recall and f1 score for each label can be observed below in table V-E.

Label	Precision	Recall	F1 Score
0	0.85	0.85	0.85
1	0.81	0.80	0.81
2	0.74	0.69	0.71
3	0.85	0.81	0.83
4	0.71	0.70	0.71
5	0.88	0.92	0.90
6	0.70	0.76	0.73
7	0.71	0.71	0.71
8	0.74	0.75	0.74
9	0.83	0.83	0.83

TABLE IX
LOGISTIC REGRESSION REPORT

64fe8ad0fc2c

VI. MODEL COMPARISON

A. Image Augmentation Result

VII. CONCLUSION

REFERENCES

[1] Akanksha Telagamsetty, Sign Language Digits Classification https://medium.com/analytics-vidhya/sign-language-classification-