ASCII Sign Language classification

Alireza Dizaji, Std No. 20269395, username: aliirezadizaji
Michell Payano, Std No. 20265175, username: MichellP

December 12th 2023

1 Introduction

In this report we present the process and strategies of model selection followed in order to classify accurately the sign language based on images of the hand [3]. More specifically, the main goal of this work is to be able to build a model capable of classifying 24 letters of the alphabet, excluding J and Z since they require gesture motions, and then use these models to classify two given sign language images into their corresponding alphabet and then sum their respective ASCII values to provide the resultant final character corresponding to the summed ASCII value. The distribution of classes is represented in Figure 1,

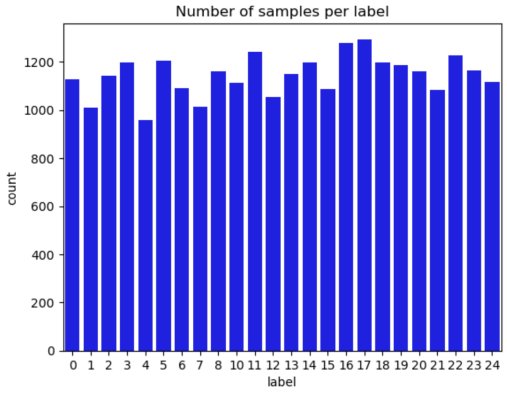

Figure 1: Number of samples per label for the training set. We can see that there is not an important difference among the samples.

Each machine-learning project requires a pipeline. The pipeline that here is used consists of data preprocessing, training, evaluation, and analysis from the evaluation and then going through this process again and again by getting results from hyper-parameter tuning.

In order to accomplish this goal, we used a Convolutional Neural Network, a Random Forest built from scratch, and the AdaBoost (Adaptive boosting) algorithm. **As a result, on public test set evaluation of Kaggle we reached 99.71%, 67%, and 54.57% on the Convolutional Neural Networks, Random forest, and adaboost methods respectively.**

2 Feature design

The data set used for this project has 784 columns, that represent an image of 28x28 pixels with grayscale values between 0-255. For the training set we have 27,455 cases and for the test set, which contains two images in each row instead of one as in the training set, 3,000 cases. Below we present in more detail how we pre-processed the data for each corresponding algorithm used:

1. **Convolutional Neural Network (CNN)**: As a first approached we normalized the data set by dividing each of this values with 255 in order to have pixel values to be between 0 and 1. This is an important step as [1] explains, we should normalize our data since it would be problematic to feed into a neural network values that all take wildly different ranges because it would make learning more difficult. Specially since neural networks apply optimization algorithms such as gradient descent, to find and update optimal weights. Another thing to consider for this data set is that we applied image augmentation, which is a popular technique used in computer vision that helps to avoid overfitting. Therefore, we applied a simple data augmentation procedure that consists on applying various random transformations to the training images on the fly. In more detail:

   - Rotation range: The image is randomly rotated in the specified degree range.
   - Width shift: The image is randomly shifted horizontally, either left or right.
   - Height shift: The image is randomly shifted vertically, either up or down.
   - Shear: Random shearing transformations are applied to the image on a clockwise direction in the specified degree range.
   - Zoom: The image is randomly zoom in and out in the specified range.

   It is important to note here that the values used for each of the transformations described above were arbitrarily selected and are relatively small, since we wanted our CNN to be trained on slightly modified versions of the original images.

2. **Random Forest / Adaboost**: We flattened the input feature into a single 784 features. Each feature space is normalized by simply dividing the input space by 255.

3 Algorithms

As we mentioned above, we used 3 models in order to compare their ability to accurately classify the sign language based on the given pixels of the several images. Below we give an overview of each learning algorithm:

1. **Convolutional Neural Network (CNN)**: This is a class of artificial deep neural network that is mainly used to process visual data. It relies on the application of convolution operation which consists of extracting patches from the input feature map and applies a non-linear transformation to each of them in order to produce an output, called the feature map or activation map. This operation is performed in the convolutional layer, whose aim is to apply filters that extract or detect patterns. Another key component of this kind of neural network, in contrast to the other ones, is the employment of pooling layers, which are basically used to reduce or downsample

the size of the feature map, and as a result it reduces the computational complexity. At the end of this model, there is the fully connected layer, operates on a flattened input where each input is connected to all neurons [2].

One of the most important characteristics of these algorithms is that the patterns they learn are translation invariant [1], this means that the CNN can recognize certain patterns in any position, in other words, after learning a pattern in a given position, e.g in the center of the image, the model can recognize it in the corner. Another important thing to consider when using CNNs and that makes them suitable for image processing, is that they can learn spatial hierarchies of patterns, which means that each layer will learn layer patterns of the features learnt in the previous ones.

2. **Random Forest**: All the components of our random forest were implemented from scratch, and here we described a brief overview of each:

   (a) **Random forest**: The random forest consists of a multiple number of decision trees, called estimators, and each decision tree is trained on a different version of a given dataset. Different versions are generated by randomly taking samples (p_bootstrapping) with random feature selection (p_featuring). We considered a trade-off between the number of estimators, and the probability of bootstrapping during our experiments, i.e. if we use more estimators then we take less number of input samples.

   (b) **Decision tree**: Decision trees are the based estimators of random forest classifiers. To prevent overfitting, we used the "max_depth" parameter to avoid the tree expanding very deeply as this could cause overfitting issues.

   (c) **Node**: Each tree consists of multiple nodes, where at each node a question is being asked to split the input dataset into two subsets till it reaches a leaf node where we can assign the majority class of the node to the given input sample. To ask questions, we take midpoints of each feature and we check the efficiency of splitting for each midpoint by measuring the entropy of left and right subsets after splitting.

3. **AdaBoost**: We used the adaboost implemented by the scikit-learn library. We changed the base model of the current method as it basically uses a decision tree without any pre-initialized hyperparameters. We passed the decision-tree model implemented by scikit-learn where we tuned different hyperparameters for it.

4 Methodology

For the purpose of finding the best model, we performed hyperparameter tuning on each of the models described above.

To find the best model, we tried tuning hyper-parameters by splitting the training set into train/val set. The splitting was done 75% for the train set and 25% for val set. In the following, different strategies and hyperparameters used for training are explained for each model:

1. **Convolutional Neural Networks (CNN)**: For the CNN we tried basically 4 configurations. The first one consisted basically on 2 convolutional layers followed by one batch normalization layer, and then a max pooling layer before the flatten and

fully dense layer. The specific reason of why we choose the max pooling layer for this project was because it tends to work better and is most used in practice. The other 3 configurations consisted on adding "blocks" with the same structure as the first one, so for example, for the 4th configuration we have 4 blocks before the final dense layer. For each one of this we tried to tune the following hyperparameters:

(a) **Number neurons**: These are the computational units of each layer which receives the resulting signals from previous layers.

(b) **Learning rate**: This represents the step size taken during the optimization process and tells how fast the model can learn.

(c) **Strides in pooling layer**: This represents the number of pixels by which the filter moves after each operation.

(d) **Activation function**: These are the transformations (mostly non-linear) applied to the input data of each layer in order to make the neural network able to learn complex pattern in the data set.

It is important to note that there are many other hyperparameters to take into account when building a CNN or an Artifical Neural Network in general, however, due to time constraint and to reduce the computational cost, we decided to consider only the hyperameters mentioned above. Another thing we considered was to apply Bayesian optimization instead of the grid search methodology and add an early stopping criteria based on the performance on the validation set in order to make the tuning process less computational expensive.

2. **Random forest**: For random forest, we tried multiple hyperparameters, each one is described in detail in the following:

   (a) **n_estimators**: The number of decision tree instances to be created.

   (b) **p_bootstrapping**: A float number between 0 and 1, indicating how many samples are allowed to be taken randomly from the input set, which the taken samples are going to be used for a decision tree.

   (c) **max_depth**: Maximum depth which is allowed to decision tree for asking questions.

   (d) **p_featuring**: A float number between 0 and 1, indicating how many features are allowed to be taken randomly from the input set, which the taken features are going to be used for a decision tree.

   (e) **num_midpoint**: Currently, each input feature has a value space between 0 and 255. Due to the limited resources, it is not possible to explore all the midpoints and also have a fair amount of max_depth. Therefore, we also tuned the number of midpoints. First, we take the unique midpoints (because it is highly possible to have duplicate midpoints due to the high ratio of several samples per feature), and then we take the "num_midpoint" points with equal intervals from each other in their ordering.

3. **Adaboost**: For Adaboost, we tuned various hyperparameters introduced by scikit-learn. We briefly described each one in the following:

(a) **estimator**: The base classifier (or estimator) that is being used by the Adaboost algorithm. The default one is a decision tree classifier with default hyperparameter initialization of scikit-learn.

(b) **n_estimators**: Determines the number of estimators to be generated.

(c) **learning_rate**: The weight which is applied to each estimator. A higher learning rate leads to more contribution of that classifier.

5 Results

In this section, we discuss the model selection process for each method.

1. **Convolution Neural Networks (CNN)**: As we can see in table 1 we present both the accuracy and the loss for the training and the validation sets. After trying to tune the hyperparameters of each of the 4 configurations, the one that gave the highest performance on the validation set was the one with 4 blocks, even though the other configurations, give similar results. As a result, we obtained **on the public test set of Kaggle an accuracy of 99.71%.**

	No. Blocks	Training acc	Training loss	Validation acc	Validation loss
1	1 block	0.983	0.064	0.998	0.015
2	2 blocks	0.983	0.052	0.995	0.013
3	3 blocks	0.990	0.048	0.999	0.002
4	4 blocks	0.997	0.016	1.000	0.002

Table 1: CNN performance on each configuration.

2. **Random forest**: In table 2, we examined the impact of different hyperparameters. During settings 1-3, we increased the max-depth gradually, and validation accuracy increased, then during settings 4-5, we increased the impact of increasing the number of estimators, which again caused an increase in validation accuracy. To try a higher number of estimators, we need to reduce p_bootstrapping and p_featuring to keep the model computationally efficient. The running time of experiments 6-9 takes roughly 2, 2, 4, and 8 hours respectively. The best result comes from experiments on both validation and test sets. The reason is that by increasing the number of estimators, we let the model reach more stability and capacity by generating different sub-versions of the original dataset on a great scale. **On public test evaluation, we get 67% test accuracy on the Kaggle.**

3. **Adaboost**: In table 3, we tuned different base estimator. We tried decision tree classifier but with different max_depths. We also increased gradually the number of estimators to see its impact on the generalization. We keep the learning rate the same during all our 6 experiments. The highest validation accuracy is achieved by the last one with 99.02%, and **On the public test set of Kaggle with 54.57%.**

6 Discussion

In this project, we explored the performance of three models: convolutional neural networks (CNN), random forest, and AdaBoost on the ASCII sign language classification task. The

	n_estimators	p_bootstrap	max_depth	p_feat	num_midpoint	Val Accuracy
1	2	0.8	5	1.0	10	28.83%
2	2	0.8	10	1.0	10	67.73%
3	2	0.8	15	1.0	10	78.91%
4	3	0.8	10	1.0	10	76.39%
5	5	0.8	10	1.0	10	85.03%
6	50	0.3	15	0.6	8	95.34%
7	50	0.3	15	0.6	12	98.51%
8	100	0.2	12	0.3	8	98.62%
9	**400**	**0.2**	**15**	**0.2**	**8**	**99.44%**

Table 2: Random forest accuracy performance on the validation set using 75%/25% split.

	estimator	n_estimators	learning_rate	Val Accuracy
1	DecisionTree(max_depth=10)	5	1.0	71.40%
2	DecisionTree(max_depth=12)	5	1.0	82.77%
3	DecisionTree(max_depth=15)	5	1.0	92.86%
4	DecisionTree(max_depth=15)	15	1.0	97.50%
5	DecisionTree(max_depth=15)	25	1.0	98.33%
6	**DecisionTree(max_depth=15)**	**40**	**1.0**	**99.02%**

Table 3: Adaboost accuracy performance on the validation set using 75%/25% split.

CNN outperforms the others. This happens because of the high ability of convolution operations to process the visual data. Overfitting is obvious for the random forest, and adaboost on the test set, while on the validation the training seems promising. To prevent random forest overfitting we could experiment with more strict ensemble learning. Also to get higher results, it is possible to tune hyper-parameters, including the increase in n_estimators, but it needs stronger resources. Also for the CNN we believe that it is still possible to get a higher accuracy in the public score at the kaggle competition if we tune other hyperparameters or even increasing the range of the searching space of the hyperparameters we specifies above. Also we saw that incrementing the number of blocks helped to achieve a better performance.

7 Statement of contribution

"We hereby state that all the work presented in this report is that of the author".

References

[1] Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

[2] *Convolutional Neural Networks cheatsheet*. `https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks`.

[3] *Kaggle ASCII Sign Language competition*. `https://www.kaggle.com/competitions/ascii-sign-language/overview`.