Classifying Hand Gestures

João Brito, M9984
Department of Computer Science
University of Beira Interior
Covilhã, Portugal
joao.pedro.brito{at}ubi.pt

***Abstract*—The present document describes the process of creating a dataset of hand gestures for image classification, with the latter being done with Convolutional Neural Networks. This work focused, in specific, on the natural digits that can be expressed using Portuguese sign language.**

I. Introduction

The task of classifying images is widely regarded has one of the leading areas of study in the field of Machine Learning. Within it, one can find the Deep Learning sub-area, where intense efforts have been made throughout the last decade. Amongst the most successful models, Convolutional Neural Networks (CNNs, in their commonly abbreviated designation) have stood out for their performance and intuitive reasoning.

Taking the above in consideration, this project involved the capturing and processing of several images of hand signs to train a CNN model, enabling it to attribute classes to never before seen instances.

As a reference point, the Portuguese sing language contemplates the following gestures for natural digits:

Fig. 1: Natural digits in sign language.

Over the next sections, the decisions regarding the creation of the dataset, the theoretical concepts behind CNNs and the actual models used will be discussed, culminating in the results seen in the experiments section (IV).

II. Hand Gestures Dataset

A. Overview

The dataset constructed for this work has the following characteristics:

- **Subjects**: 4, in total (3 for training and 1 for testing).
- **Images**: over 2000, in total (90% for training and 10% for testing, equally divided by each of the 10 classes).
- **Variation Factors**: time of day (morning, afternoon or night), subject, lighting (dark or bright), background (against a wall, the outside or the subject's body) and distance to the camera (close or far).

B. Data Capturing and Processing

Various steps were taken to arrive at a decently sized dataset, representative of real world conditions.

Firstly, the images were captured under the variation factors mentioned earlier. It was decided to record videos instead of still images. Such decision has the following benefits: ability to retrieve n frames per video (as many as needed); natural degradation factors (motion blur, hand positioning, etc...); logistics (it is more efficient to record a small amount of videos and obtain the frames afterwards, instead of taking images one by one). In the end, 270 training videos (27 per class) and 20 testing videos (2 per class) were made.

Next, the videos were cropped to fit inside a 720x720 px square. Given that Neural Networks usually take as input square images, this step was necessary to preserve the images proportions (i.e. if the original 16:9 images had been reshaped into a 1:1 ratio, they would get stretched or compressed).

Then, two ideas/versions of the work started to take shape: the first, naive, put the images through a data augmentation process (contrast changes, horizontal flips and rotation operations); the second, more interesting, coupled these techniques with a very simple and rudimentary form of semantic segmentation (based on HSV colors).

On one hand, the data augmentation process ensured that the images are as varied as possible. On the other hand, by extracting the hand from the original image, the Neural Network has a much better time classifying gestures.

More details on the exact implementation details can be found in section IV.

C. Examples

The images shown were sampled from the final dataset. They represent the raw frames, without data augmentation or hand segmentation applied to them:

Fig. 2: Dataset samples in their raw form.

Considering the aforementioned motivations for hand segmentation, the next set of images represents the same four instances as figure 2, but with data augmentation and segmentation applied to them:

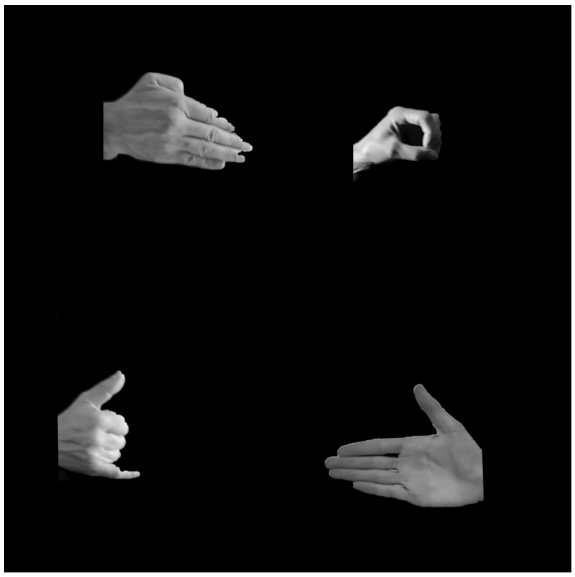


Fig. 3: Dataset samples in their totally processed form.

The impacts on the model's performance will be discussed throughout the rest of this document.

III. Convolutional Neural Networks (CNNs)

A. Overview

The basis of CNNs resembles the inner workings of the human brain, namely through the use of feature extraction techniques and (artificial) neurons. The former is employed to capture characteristics from a given image (the so called features), in the form of edges, colours, shapes or textures. The latter, takes those features and gives them meaning, with an obvious output at the end. The following subsections will go over each main component of a CNN, in more detail, and the two models used for the purposes of this work.

B. Convolutional Layers

The first building block of a classical CNN (and the source of its name) is the convolutional layer, as seen in the following figure:

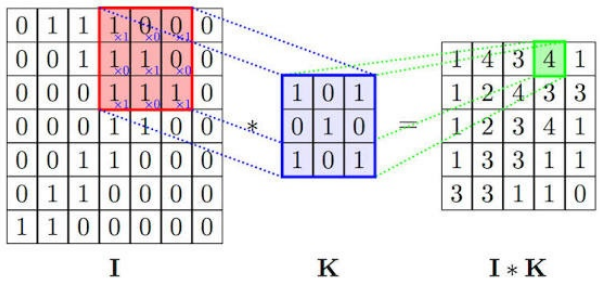


Fig. 4: Visualisation of a convolutional layer and the operations done.

Above, we can see how the input (I) and the kernel (K) are used together to form the resulting $I * K$ output. The input can be the original image or the result of previous convolutions, and the kernel is a matrix of size (k,k) and the same depth as the input. The key here is that the kernel acts has a sliding window over the input, multiplying its values (learned through training) by the selected ones on the input matrix. If, for example, we wanted to spot straight, vertical lines on an image, we would expect the network to devote one (or more) of its kernels to be a matrix with a vertical row of "1"s, flanked by "0"s. So, there are several kernels, each with a unique purpose set by the network whilst training.

C. Pooling Layers

Going hand in hand with convolutional layers, pooling layers are capable of reducing the space complexity of the input given. As noted in figure 5, pooling involves either a "max" or "average" operation:

- **Max Pooling**: takes a 4x4 grid of values, divides it in 4 sub-grids and, for each of them, chooses the biggest value;
- **Average Pooling**: takes a 4x4 grid of values, divides it in 4 sub-grids and, for each of them, computes the mean of all the values within.

Another advantage of using pooling is that it gives the features some spatial invariance, meaning they can appear in

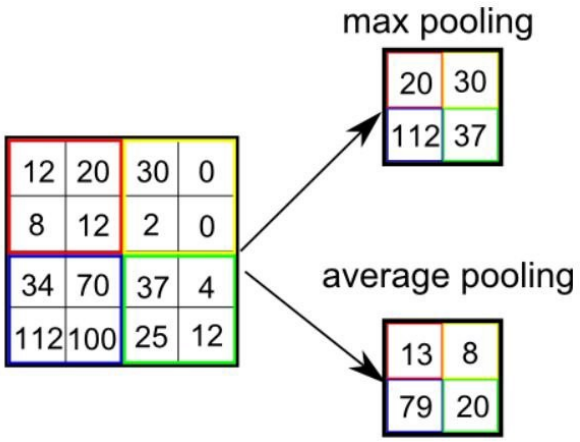


Fig. 5: Visualisation of a pooling layer in its two variants: max pooling and average pooling.

various locations on the input, with some guarantee that the pooling operation will keep them (e.g. no matter where the value 37 appears in the green sub-grid, it will always be the biggest number, in the case of max pooling).

The behaviour of pooling layers as explained in this section is common, though not entirely ubiquitous. Depending on the situation, bigger grids may be preferred.

D. Fully Connected Layers

This final component is, essentially, a MLP (Multilayer Perceptron): a collection of (artificial) neurons stacked together in layers (much like the neuron arrangement in the human brain). In practice, the input of the first layer of neurons is a flat version of the feature map captured and, per layer, each neuron receives the same input as the others. What sets them apart, though, is how they weigh those inputs, essentially performing a linear combination with the inputs given and the weights learned via training (i.e. how important each input is):

$$z = W_0 * x_0 + W_1 * x_1 + ... + W_n * x_n \quad (1)$$

Next, the inner product is given to an activation function (such as ReLU or Sigmoid):

$$a = activation\_function(z) \quad (2)$$

A schematic of fully connected layers can be seen in figure 6. It is worth noting that there are, basically, three types of layers:

- **Input Layer**: the first layer, takes the input given (in a CNN, this layer receives a flat version of the feature maps computed during the feature extraction stage).
- **Hidden Layers**: the intermediate layers, called "hidden" because they don't interact directly with the input or the output. Most of the processing happens through these layers.
- **Output Layer**: the final layer, produces the output of the network (which will later be compared to the ground truth, using a loss/cost function).

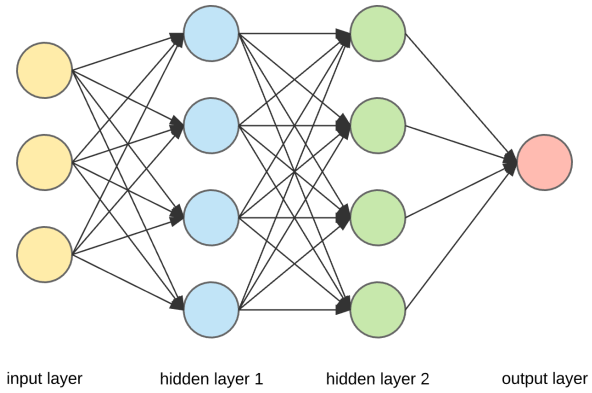


Fig. 6: Visualisation of fully connected layers.

E. Models Used

For experimental purposes, two CNN models were tested, namely:

1) VGG19: One of the several versions of the classical VGG architecture [1], the VGG19 network has 19 main layers (16 convolutional layers and 3 fully connected layers). There are also pooling layers (5, in fact), though not represented in the network's designation.

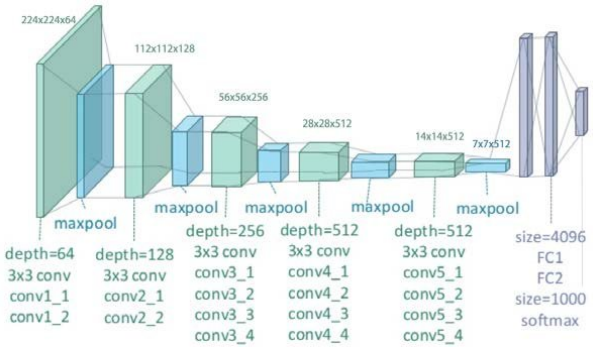


Fig. 7: Architecture of the VGG19 network.

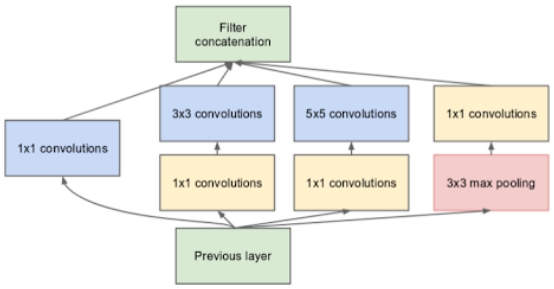
It is widely used in the field for quick experimentation, given the capacity provided and relatively small training times.

2) InceptionV3: Based on the original Google Inception architecture [2], this third version attempted to rethink some architectural aspects. To understand those changes, let us take a look at the basic building block of the this type of architecture:

The main problem tackled by the original Inception module was the fact that objects, in an image, can appear in many different locations (to the right, to the left, far away, up close, etc...). Thus, choosing the right kernel size *a priori* can be challenging. Therefore, the goal of this module is to allow multiple filters/kernels to be used for feature extraction (in figure 8, we can see 1x1, 3x3 and 5x5 convolutions).

Additionally, Inception V3 evolved on such principles in the following ways (amongst others):

- **New modules**: Three new types of modules were proposed (called "A", "B" and "C" here for convenience):



(b) Inception module with dimension reductions

Fig. 8: Inception module in its original form.

  - **Module A**: Replaced 5x5 kernels with two consecutive 3x3 kernels for parameter efficiency (5*5 = 25 $>$ 2*(3*3) = 18, with "bigger" meaning "worse").
  - **Module B**: Replaced 3x3 kernels with a 3x1 kernel and a 1x3 filter, for the same reasons as above.
  - **Module C**: Added more parallel kernels, meaning that the "Filter Concatenation" block would receive more information, allowing higher dimensional representations.
- **Number of Auxiliary Classifiers**: Reduced the number of auxiliary classifiers to just one (previously, there were two). These classifiers help to back-propagate the gradient information to earlier layers in the network.
- **Efficient Grid Size Reduction**: Added a new way of reducing the grid size (height x width) of feature maps. Whereas traditional pooling layers are used for this specific purpose, the authors decided to use pooling and convolution operations to achieve a similar result (each of those operations accounts to half of the processed input).

The improvements mentioned helped this deep neural network (48 layers deep, instead of the original's 27) achieve great results compared to the state of the art methodologies.

IV. Experiments

The experiments shown in this section were made possible by using TensorFlow [3] and its module TF-Slim [1].

For starters, some hyper-parameters were defined, through some quick experimentation, and left unchanged afterwards (i.e. the experiments did not focus specifically on them):

- **Epochs**: 60/75.
- **Dropout (Keep)**: 0.5 for VGG19 and 0.8 for InceptionV3.
- **Label Smoothing**: 0.1 (α value).
- **Image Size**: 224x224 px for VGG19 and 299x299 px for InceptionV3.

Note that Label Smoothing was applied to penalise the model for being too confident with its predictions, something that could lead to greater loss values.

[1] https://github.com/google-research/tf-slim

For the experimentation phase of the first idea, the VGG19 model was chosen to start with. After some tests, the results were not satisfactory, as one can see with the following figures:

Fig. 9: Training plot of the first version of this work.

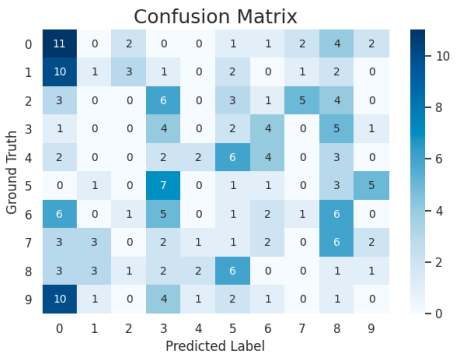


Fig. 10: Confusion matrix of the first version.

The first version of this project (using images with just data augmentation) made it very difficult for the chosen CNN model to learn any significant information. Despite a decent training accuracy, the testing phase showed how poorly it generalised (a clear sign of overfitting).

With such bad results, any improvement was most welcome. Therefore, a new, more sophisticated data pipeline was devised:

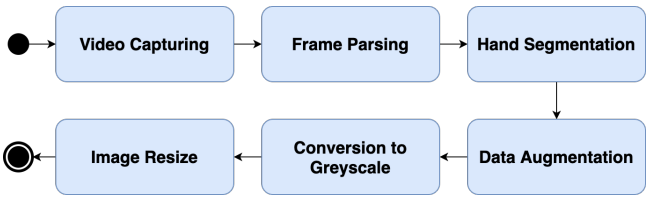


Fig. 11: Data processing pipeline.

Almost every main step in the pipeline was done using OpenCV [4]. To start, the hand segmentation process was quite tedious, given the amount of source videos. To ease this task, a Python script was used whose job was to display a video, show some controls and ask the user to chose the right HSV

threshold to determine what was a hand and what was not (figure 12).

Fig. 12: Snippet of the hand segmentation task.

The algorithm [5] responsible for extracting hands was really simple: look at every pixel in each frame and choose if it should be kept or discarded (given a black shade). There are some rough spots, especially around the edges or even the hand itself, but given how straight forward it was to implement, the results are decent enough.

The pipeline progresses with the usual data augmentation step, leading to a conversion to greyscale images. In a problem like this, colour is not that important. What sets gestures apart from each other is the shape made by the hand, with no regard for colour. Thus, every image was converted to a greyscale equivalent, with a resizing operation at the end.

The dataset was now ready to be feed the InceptionV3 model, this time around. This model was chosen to get a sense of what a more powerful architecture can produce, given the refinements made to the image processing pipeline.

The following table displays the experiments made, highlighting the best configuration (in bold and in figures 13/14):

CNN	Learning Rate	Batch Size	Test Accuracy
InceptionV3	$1 * 10^{-4}$	64	80.7%
	$1 * 10^{-5}$	64	79.38%
InceptionV3	$1 * 10^{-4}$	32	82.4%
	$1 * 10^{-4}$	16	**83.2%**

TABLE I: Impact of some parameters on the training process.

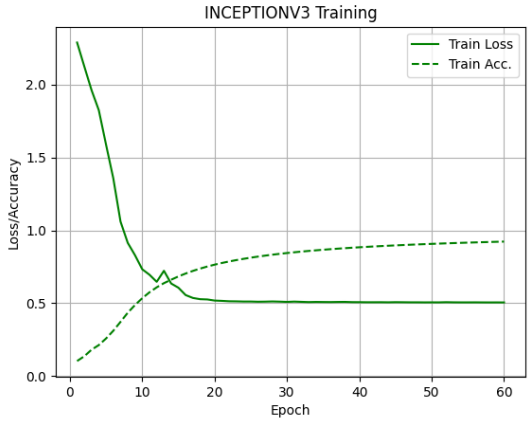


Fig. 13: Training plot of the second version of this work.

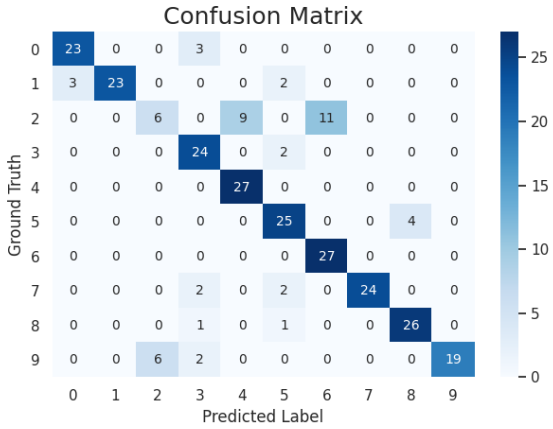


Fig. 14: Confusion matrix of the second version.

This time, the results were far better: the training process started right away (instead of halting for a few epochs like in figure 9) and the confusion matrix was much clearer. The only obvious flaw seemed to happen with digit ”2”. Such problem may have to do with the fact that, both ”4” and ”6” have a finger pointed upwards, like ”2”. Given the imperfections in the hand segmentation step, the ”2”’s pinky finger may have been cropped or dwarfed, leading the model to some confusion. Better segmentation techniques could improve these results even further.

V. Conclusion

The present document described a project whose aim was to classify hand gestures. Several experiments were made with regards to the hyper-parameters and image processing pipeline. With a project of this nature, the main takeaways are that creating a dataset is laborious, with a need for craft and attention to detail. The choice of the model shouldn’t be taken lightly, as well, as it can hinder the final results. Overall, it was a very interesting introduction (or revision) of Deep Learning concepts, both in theory and in practice.

REFERENCES

[1] Understanding the VGG19 Architecture . [Online] https://iq.opengenus.org/vgg19-architecture/

[2] A Simple Guide to the versions of the Inception Network. [Online] https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202

[3] TensorFlow. [Online] https://www.tensorflow.org/

[4] OpenCV. [Online] https://opencv.org/

[5] Skin Detection: A Step-by-Step Example using Python and OpenCV. [Online] https://www.pyimagesearch.com/2014/08/18/skin-detection-step-step-example-using-python-opencv/