



# **SKIN DISEASE CNN CLASSIFIER**

**Machine Learning project 2019/2020**

**Giuseppe Fontana 1870948**  
**Fabio Galassi 1869688**



## Summary

<b>Introduction</b>	3
Problem Description	3
Tools Used & Datasets	3
Approach & Methodology	4
The Dataset Structure	4
Epidermal Diseases	5
<b>Notebook 1</b>	8
Feature Engineering	8
Data Augmentation	10
Dictionary Update	12
Image Re-Scaling	13
Training Set, Test Set, Validation Set	13
<b>Notebook 2</b>	14
Classifier	14
Batch Normalization	15
Flatten	15
Relu Activation Function	15
Softmax Activation Function	16
Adam Optimizer	16
Adadelta	17
Categorical Cross-Entropy	18
Dropout	18
Training Phase	18
Fit	19
ModelCheckpoint	20
EarlyStopping	20
History	20
Models	21
Tests	25
Results Analysis	26
References	28

# Introduction

## Problem Description

Our machine-learning project is oriented to the visual recognition of lesions, imperfections and diseases of the skin, that are more widespread and relevant from a dermatological point of view. The problem we want to address is the automated diagnosis of pigmented skin lesions starting from the analysis of photos that show such diseases.

The goal is reached by carrying out a convolutional neural network able to perform multi-class classification on images.

## Tools Used & Datasets

We started using **Python 3** embedded in **Anaconda** (which is a package management tool) and we adopted the **.ipynb** file format for the work: it is a format that contains both computer code (python in our case) and rich text elements (such as paragraphs, equations, figures, links, etc...).

**Notebook** documents are both human-readable documents containing the analysis description and python interpreter-runnable for the project implementation; moreover, the Anaconda execution environment allows to keep memory content among execution of many code blocks, with the purpose to make easier the development (allowing incremental changes without having to re-initialize the context and without the need to repeat computationally more expensive instructions).

**Google Colaboratory** (also known as **Colab**) is a free Jupyter notebook environment that runs in the cloud and stores its notebooks on Google Drive; it mainly allows to perform programming and execution on a quite powerful hardware. We leveraged Colab in the latter phases of the work (training and testing)

**Scikit-learn** is an open source machine learning library for Python programming language. It features various machine learning algorithms and it is designed to interoperate with the Python numerical and scientific library NumPy.

**Pandas** is a software library written in Python programming language for data manipulation and analysis, used for read/write operations on .csv files and DataFrame structure methods at runtime.

**Matplotlib** is a python library for plotting a large number of graph types and PIL another library for image management.

**HAM10000** ("Human Against Machine with 10000 training images") is a collection of dermatoscopic images from different populations, acquired and stored by different modalities. The dataset consists of 10015 dermatoscopic images which can serve for academic machine learning purposes. Cases include a representative collection of all important diagnostic categories in the realm of pigmented lesions.

# Approach & Methodology

The whole work is divided into 2 jupyter notebooks:

- In the first notebook there is the initial analysis of the dataset, the data augmentation part, the rescaling of images and the creation of CSV sheets to facilitate the next steps;
- The second notebook deals with classifier definitions, parameter tuning, training and testing of classifiers, analysis of results and performance.

This split facilitated the progress of the work because in the first part the heaviest code to execute (creation and re-scaling of images) was run only once locally, while the second notebook was developed on **Google Colab**, that offered virtualized environments with **12 GB** of **RAM** and dedicated **GPUs**, and reduced the execution time of training and testing from several hours to some tens of minutes.

## The Dataset Structure

The dataset consists of a folder with 10015 images and a .csv sheet representing the image metadata; it contains the following columns:

- **lesion\_id**: the lesion identifier, it has the same value for each image of the specific patient's lesion;
- **image\_id**: the name of the image of the disease; it is the identification key of the dataset since each tuple is precisely related to an image;
- **dx**: the type of disease found in the image, that is the label of our classifier; it can take 7 different values:
  - Actinic keratoses and intraepithelial carcinoma / Bowen's disease (**akiec**)
  - Basal cell carcinoma (**bcc**),
  - Benign keratosis-like lesions (solar lentigines / seborrheic keratoses and lichen-planus like keratosis, **bkl**),
  - Dermatofibroma (**df**),
  - Melanoma (**mel**),
  - Melanocytic nevi (**nv**) and
  - Vascular lesions (angiomas, angiokeratomas, pyogenic granulomas and hemorrhage, **vasc**).
- **dx\_type**: type of confirmation of the disease; it can be histopathology (*histo*), follow-up examination (*followup*), *expert consensus (consensus)*, or *confirmation by in-vivo confocal microscopy (confocal)*
- **age**: the age of patient (up to 85 years old)
- **sex**: the gender of the patient
- **localization**: on which body side the disease is localized

## Epidermal Diseases

### **Actinic keratosis and intraepithelial carcinoma / Bowen's disease (akiec)**

An actinic keratosis is a rough, scaly patch on your skin that develops from years of exposure to the sun. It's most commonly found on your face, lips, ears, back of your hands, forearms, scalp or neck. Also known as a solar keratosis, an actinic keratosis enlarges slowly and usually causes no signs or symptoms other than a patch or small spot on your skin. A small percentage of actinic keratosis lesions can eventually become skin cancer. Carcinoma in situ is a vitiated, superficial growth of cancerous cells on the skin's outer layer. It is not a severe condition but could develop into a full form of invasive skin cancer if not detected early or well managed. It is also known as carcinoma in situ in the literature or as Bowen disease after John T. Bowen, an American dermatologist who first described the condition in 1912.

### **Basal cell carcinoma (bcc)**

Basal cell carcinoma is a type of skin cancer; it begins in the basal cells, which are a type of cell within the skin that produces new skin cells as old ones die off.

Basal cell carcinoma often appears as a slightly transparent bump on the skin, though it can take other forms. Basal cell carcinoma occurs most often on areas of the skin that are exposed to the sun, such as your head and neck.

Most basal cell carcinomas are thought to be caused by long-term exposure to ultraviolet (UV) radiation from sunlight. Avoiding the sun and using sunscreen may help protect against basal cell carcinoma.

### **Benign keratosis-like lesions (bkl)**

These diseases are mostly solar lentigines, seborrheic keratoses and lichen-planus like keratosis.

A seborrheic keratosis is a common non cancerous skin growth. Seborrheic keratoses are usually brown, black or light tan. The growths look waxy, scaly and slightly raised. They usually appear on the head, neck, chest or back. These keratoses don't need treatment, but you may decide to have them removed if they become irritated by clothing or you don't like how they look.

Lichen planus is an inflammatory disorder that appears as purplish, flat-topped bumps. Bumps may appear in clusters or lines.

Solar lentigines are small, flat dark areas on the skin. They vary in size and usually appear on areas exposed to the sun, such as the face, hands, shoulders and arms.

Age spots can look like cancerous growths. True age spots don't need treatment, but they are a sign the skin has received a lot of sun exposure and are an attempt by your skin to protect itself from more sun damage.

**Dermatofibroma (df)**

Dermatofibromas are hard solitary slow-growing papules (rounded bumps) that may appear in a variety of colours, usually brownish to tan; they are often elevated or pedunculated. A dermatofibroma is associated with the dimple sign; by applying lateral pressure, there is a central depression of the dermatofibroma. Although typical dermatofibromas cause little or no discomfort, itching and tenderness can occur.

**Melanoma (mel)**

Melanoma, the most serious type of skin cancer, develops in the cells (melanocytes) that produce melanin, the pigment that gives your skin its color. Melanoma can also form in your eyes and, rarely, inside your body, such as in your nose or throat.

Knowing the warning signs of skin cancer can help ensure that cancerous changes are detected and treated before the cancer has spread. Melanoma can be treated successfully if it is detected early.

**Melanocytic nevi (nv)**

Nevi are a common type of skin growth. They often appear as small, dark brown spots and are caused by clusters of pigmented cells. Nevi generally appear during childhood and adolescence. Most people have 10 to 40 nevi, some of which may change in appearance or fade away over time. Most of them are harmless. Rarely, they become cancerous.

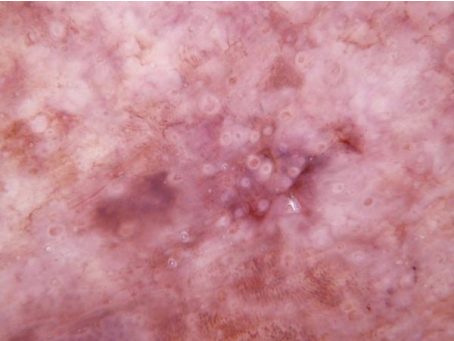

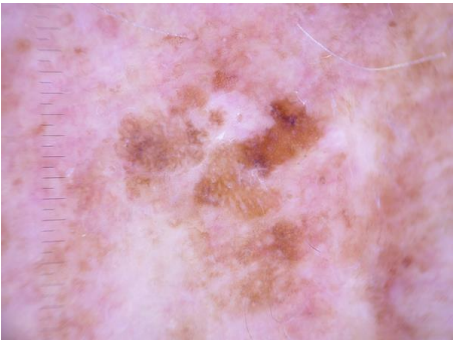

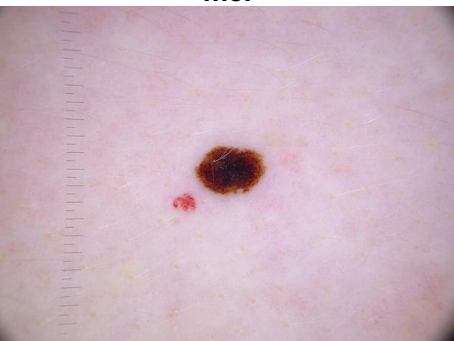


**Vascular lesions (vasc)**

Angiomas, angiokeratomas, pyogenic granulomas and hemorrhage belong to this classification.

Cutaneous vascular lesions comprise of all skin disease that originate from or affect blood or lymphatic vessels, including malignant or benign tumors, malformations and inflammatory disease. While some vascular lesions are easily diagnosed clinically and dermoscopically, other vascular lesions can be challenging as many of them share similar dermoscopic features.



Here we have some examples of images taken by the dataset:

<b>akiec</b> 	<b>bcc</b> 
<b>bkl</b> 	<b>df</b> 
<b>mel</b> 	<b>nv</b> 
<b>vasc</b> 	

# Notebook 1

## Feature Engineering

To begin with, we opened the .csv sheet as Pandas.DataFrame structure and made some feature deletion in order to have a simple dictionary containing the names of the images and the respective diseases; the features we want are only the ones from images, that will be treated as matrix of colored pixels in the RGB format.

We deleted from the DataFrame the not relevant features for image recognition and we ran the label encoding on the "dx" label.

The result has been saved in csv format:

```
# apre il dataset originale, toglie le features superflue e lo ritorna insieme all'array delle labels
def initialize_dataset():
    dataset = pd.read_csv(csv_path, encoding = "ISO-8859-1")
    labels = set(dataset["dx"])
    new_ds = pd.DataFrame(dataset)
    columns=["lesion_id", "dx_type", "age", "sex", "localization"]
    new_ds = new_ds.drop(columns=columns, axis=0)
    return new_ds, labels
```

```
# INIZIALIZZAZIONE

#creazione dataframe in memoria con colonne [image_id, dx]
ds, labels = initialize_dataset()

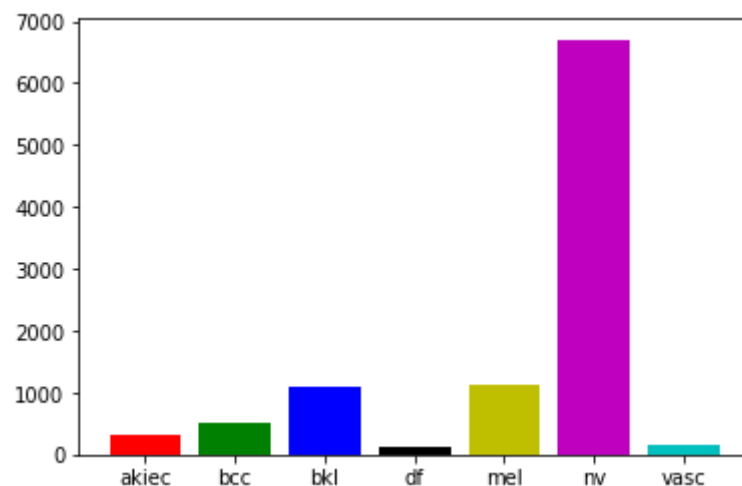
# label encoding delle labels (dx)
encoder = preproc.LabelEncoder()
encoded = lab_encode(ds, encoder)
encoded = encoded.sort_index()

encoded.to_csv(csv_enc, index=False)
encoded
```

	image_id	dx
0	ISIC_0027419	2
1	ISIC_0025030	2
2	ISIC_0026769	2
3	ISIC_0025661	2



Checking the dictionary one can see that there is a strong imbalance in the data: there are over 6000 images of Melanocytic nevi (class 5), 1099 images for Benign keratosis-like lesions (class 2) and a few hundred for the other classes (only 115 of Dermatofibroma).



```
# analisi dei valori per data augmentation
aug_size = [0,0,0,0,0,0,0]
stats(encoded, aug_size)
```

```
class 0 :      327
ugmented per image: 19
```

```
class 1 :      514
ugmented per image: 12
```

```
class 2 :     1099
ugmented per image:  5
```

```
class 3 :      115
ugmented per image: 57
```

```
class 4 :     1113
ugmented per image:  5
```

```
class 5 :     6705
ugmented per image:  0
```

```
class 6 :      142
ugmented per image: 46
```

An imbalance too marked in the available data depends on the occurrence of the different diseases in the patients; these frequencies are not at all fair but are plausible; for example, there are high rates of nevi in Oceania, North America, Europe, South Africa and Latin America, some people have hundreds of nevi and for scientific reasons there are no patients with such huge numbers in other diseases.

In order to obtain a better performing classifier, we have decided to practice data augmentation on the minority classes; this choice will allow during the training phase to see

the set of images according to a more uniform trend and avoid underfitting on the less frequent classes.

## Data Augmentation

To make the number of images uniform between the various classes, we have generated new "synthetic" samples so that we can obtain the same occurrences of class 5 (about 7000). Data augmentation has been implemented through the `flow()` method of the `ImageDataGenerator` object from the Keras library:

```
# Create a data generator
datagen = ImageDataGenerator(
    rotation_range=180,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.1,
    horizontal_flip=True,
    vertical_flip=True,
    #brightness_range=(0.9,1.1),
    fill_mode='nearest')

##### creazione immagini augmented
# non aumentiamo la classe 'nv', cioè la 5
class_list = ['0','1','2','3','4','6']
```

The choice of parameters for the differentiation of the samples involves small spatial variations to keep the central pixels as the most relevant and no color alterations that may mislead the classifier. The `stats()` function shown in the previous page fills a vector of integers called `aug_size` and this one contains the number of pictures to be generated starting from every single image of the original dataset; the program iterates on these values to create the new pictures.

```
for fname in img_list:
    fpath = os.path.join(src_class_dir, fname) # path completo dell'immagine src
    img = Image.open(fpath) # PIL image
    x = img_to_array(img) # this is a Numpy array with shape (3, 600, 450)
    x = x.reshape((1,) + x.shape) # this is a Numpy array with shape (1, 3, 600, 450)

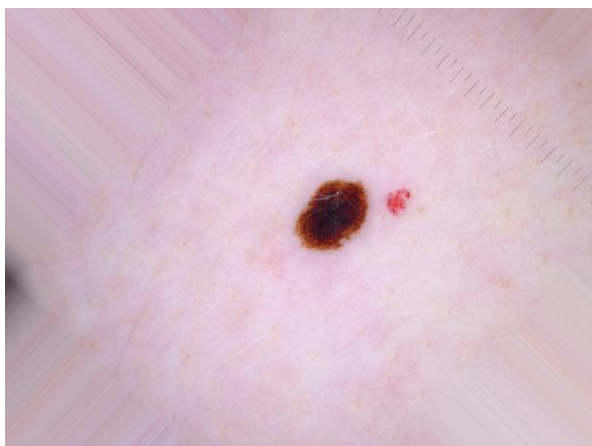
    prefix = fname.replace(".jpg", "") # prefisso delle nuove img, cioè il nome dell'originale
    # the .flow() command below generates batches of randomly transformed images
    # lo faccio fino a quando non arrivo all'aug_size
    i = 0
    for batch in datagen.flow(x, batch_size=1, save_to_dir=dst_class_dir,
                             save_prefix=prefix, save_format='jpg'):
        i += 1
        if i > aug_size[int(img_class)]:
            break
    del x
    img.close()
```

Here is an example of the original image and derived samples:

Original image



Some derived samples





## Dictionary Update

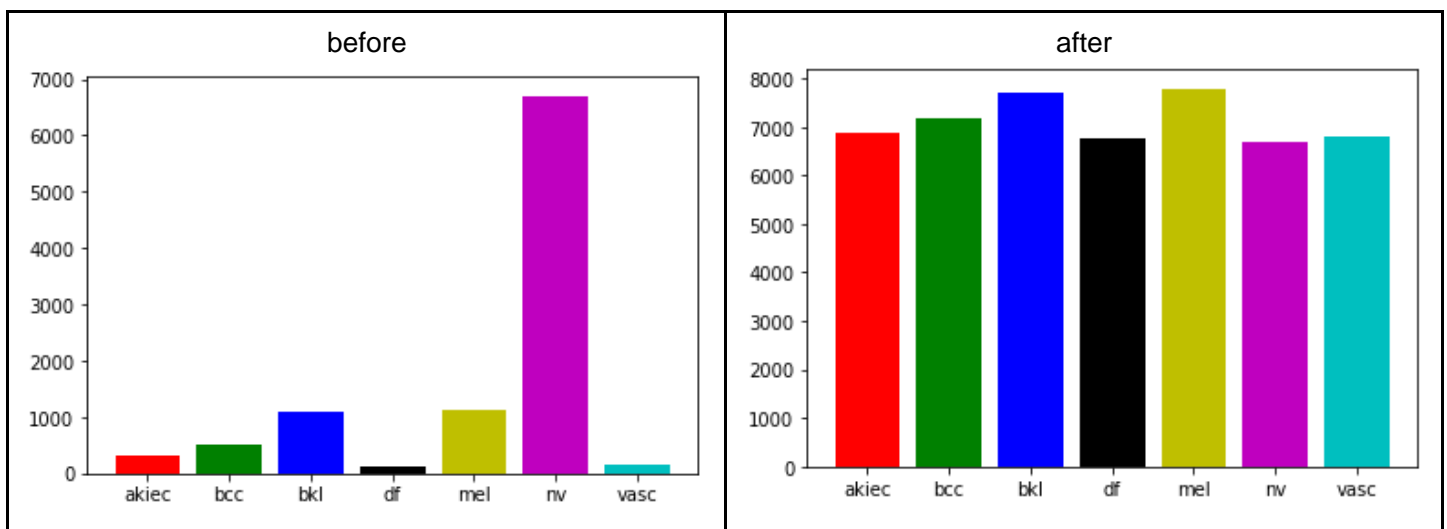
In the data augmentation phase, the new samples are stored in the folders that represent their label (directories numbered from 0 to 6); this approach was useful in updating the dictionary, which was done in few simple steps:

- a new DataFrame is initialized in memory;
- scrolling through the label folders, for each image found a new entry is added to the DataFrame with:
  - **dx**: the name of the mother directory,
  - **image\_id**: The name of the image;
- this new dictionary is saved as a csv sheet containing only augmented images and finally added to the original dictionary to build a complete dataset:

```
for elem1 in classes:
    class_dir = os.path.join(aug_dir, elem1)
    images = os.listdir(class_dir)
    for elem2 in images:
        elem2 = elem2.replace('.jpg', '')
        augmented = augmented.append({'image_id': elem2,
                                      'dx': int(elem1)}, ignore_index=True)

encoded2 = encoded.append(augmented)
encoded2.to_csv(csv_completo, index=False)
augmented.to_csv(csv_aug, index=False)
```

Here we have the trends of images before and after data augmentation:



## Image Re-Scaling

The available images have a size of 450x600 pixels and a memory usage of about 50 kb. Although these conditions may seem acceptable for learning the classifier, there was no way to complete a training without the Anaconda kernel crashing, because the devices do not have enough RAM for the operating system, for the images and for the temporary parameters of the neural network and the fit() method.

The solution adopted is the re-scaling of the images that, on one hand minimizes the necessary primary memory space, and on the other hand favors a classifier that maintains undegraded performance with a much smaller number of nodes; moreover, it speeds-up the uploading of images on Google Drive for the Colab execution.

We have tried many resizing on an image taken for each disease with different scaling factors, and we have noticed that the factor  $\frac{1}{3}$  is the best choice for information preservation (to prevent to prevent loss of important features) and memory usage that goes down with quadratic proportionality.

The resize reduces the samples from 450x600 to 150x200:

```
# rescaling immagini, src e dst sono cartelle contenenti immagini
def rescale(src_path, dst_path, size=(200,150)):
    count = 0
    for fname in os.listdir(src_path):
        if len(fname) < 2: # non è un immagine ma una cartella label
            continue
        img_rsz = Image.open(os.path.join(src_path, fname)).resize(size)
        img_rsz.save(os.path.join(dst_path, fname))
        count = count + 1
        if count % 5000 == 0:
            print("rescaled " + str(count) + " images")
    return
```

## Training Set, Test Set, Validation Set

At this point of the project we have a balanced set of more than 45000 pictures and 3 dictionaries containing respectively only starting images, only synthetic images and all the images, each one with label values.

The next steps of the project are described in Notebook 2 and are repeated each time with different portions of the data and different split percentages.

Before each replay:

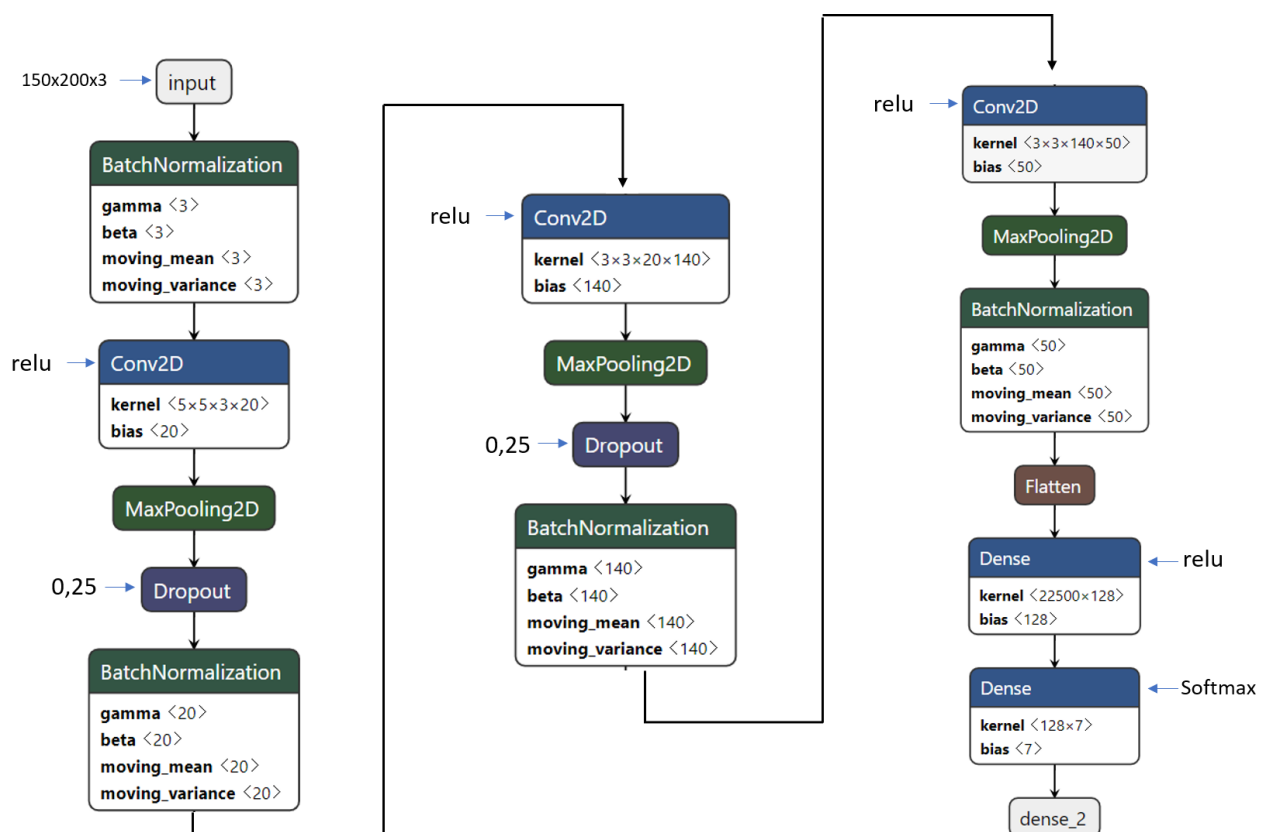
- the split percentages of the original data are chosen (e.g. for each class: training 64%, testing 20% and validation 16%);
- the respective dictionaries are created according to the proportions;
- tuples of synthetic images are added in the training set (the synthetic samples are derived from the images of this set and not from those of the other sets);
- dictionaries are saved in a dedicated folder to be uploaded later to Google Colab for running tasks in Notebook .

# Notebook 2

## Classifier

For our image classification we have decided to use a CNN classifier, which usually have good results for these tasks. The final classifier has been obtained after some tries, in which we have built the CNN, trained and tested in order to see how performant it was on our dataset, and then done some adjustment on the optimizer.

The classifier combined convolutional and pooling layers to learn global features of images. It will take as input images of size (150x200) with 3 channels. The number of classes is 7, as given in the dataset.



Convolutional layers (Conv2D) use a filter to let the image becomes abstracted to a feature map. The first layer has 20 filters with size 5x5, 140 with size 3x3 the second layer and 50 with size 3x3 the last one. Each filter applied to the image can be thought as a feature identifier.

On each convolution the “relu” activation has been applied.

After each convolution we used a pooling layer (MaxPoll2D). The pooling layer is used to reduce the computational costs since it will take (pool size=2,2) near pixels and will take only the one with max value.



## Batch Normalization

Batch Normalization is used to normalize data. Batch Normalization is a method created to solve the problem of “internal covariate shift”. Basically, while we train our network and the weights are updated, the output of every layer changes and so does the distribution of input data during iterations. Neural networks suffers from this phenomenon because they works better if the internal distribution of the data is normalized with  $\mu=0$   $\sigma=1$ . This method will

evaluate for each mini-batch the mean and variance and will normalize using  $x^{(k)}_B \leftarrow$

$x^{(k)}_B - \mu^k_B / \sqrt{\sigma^{2(k)}_B + \varepsilon}$ , so every input  $x^{(k)}_B$  is normalized by subtracting the sample mean and dividing by the square root of the variance plus an  $\varepsilon$ , an arbitrarily small constant added for numerical stability.

To restore the representation power of the network, a transformation step then follows as:

$$y^{(k)}_i = \gamma^{(k)}_i \widehat{x^{(k)}_i} + \beta^{(k)}$$

where the parameters are subsequently learned in the optimization process.

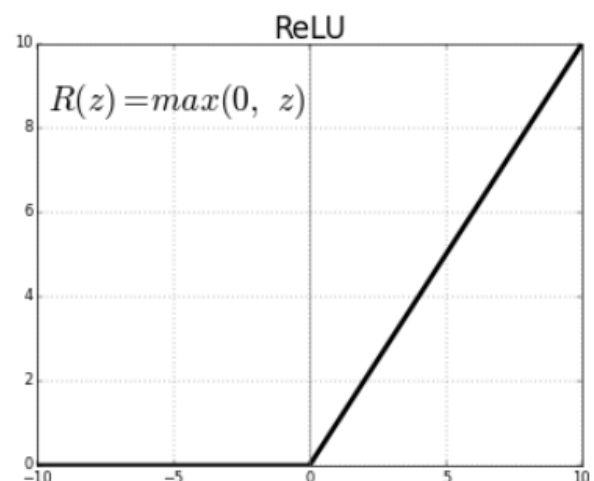
## Flatten

Flatten is used to convert the final feature maps into a one single 1D vector. This will combine all the local features found in the previous convolutional layers.

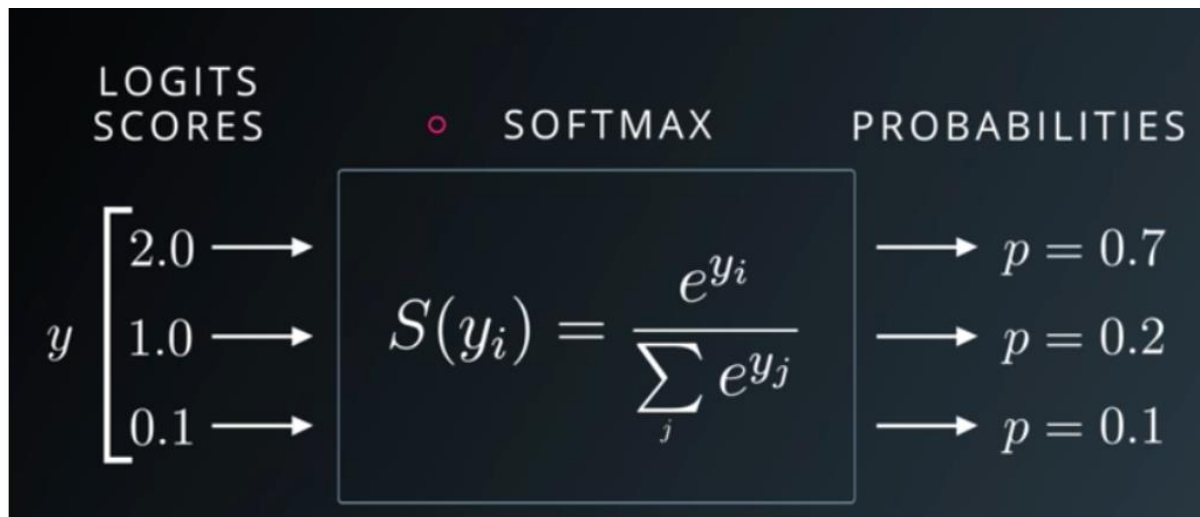
At the end we will apply the softmax activation function. The model is compiled using Adam as optimizer and categorical\_crossentropy as loss function. In the various tries the parameters of Adam has been changed to find a better result.

## Relu Activation Function

Relu function as shown in the graph is half rectified.  $F(z)$  is zero when  $z$  is negative and goes to  $\infty$  when  $z$  increases. Relu is the most used activation function with CNN since has fewer [vanishing gradient](#) problems compared to sigmoidal activation functions that saturate in both directions. The only problem with this function is that negative input values are directly mapped to zero, which in turns affects the resulting graph by not mapping the negative values appropriately.



## Softmax Activation Function



In a multiclass classification neural network, softmax function is applied at the last layer of the classifier. Given a net input parameter in the form of a one-hot encoded matrix, Softmax is needed in order to translate numeric output to turn them into probability. To do so, softmax applies the standard exponential function to each element  $z_i$  of the input vector  $z$  and normalize these values by dividing by the sum of all these exponentials; this normalization ensures that the sum of the components of the output vector  $\sigma(z)$  is 1. Softmax is usually used with cross-entropy loss function.

## Adam Optimizer

Adam (adaptive moment estimation) is an adaptive learning rate optimization algorithm that's been designed specifically for training deep neural networks. Adam keeps a learning rate for each network weight, which are scaled using estimations of first and second moments of gradient. It also takes advantage of momentum by using moving average of the gradient.

The  $n$ -th moment is defined as:  $m^n = E[X^n]$

To estimate the moments, Adam utilizes exponentially moving averages, computed on the gradient evaluated on a current mini-batch:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned}$$

where  $m$  and  $v$  are moving averages,  $g$  is gradient on current mini-batch,  $\beta_1$  is the exponential decay rate for the first moment estimates (usually set to 0.9) and  $\beta_2$  is the exponential decay rate for the second-moment estimates (usually 0.999).

Since  $m$  and  $v$  are estimates of first and second moments, we want to have that

$$E[m_t] = E[g_t] \text{ and } E[v_t] = E[g_t^2]$$

We can see that expanding  $m$  values, results in obtaining a pattern like this:

$$m_t = (1 - \beta_1) \sum_{i=0}^{t-1} \beta_1^{(t-1-i)} g_i$$

Since we're multiplying for decreasing amount of  $\beta$ , expanding  $m$  to greater orders means having the first values of gradients contributing less and less to the total value.

Putting all together we have:

$E[m_t] = E[(1 - \beta_1) \sum_{i=0}^t \beta_1^{(t-1)} g_i]$  Approximating  $g_i$  to  $g_t$  we can write  $E[g_t](1 - \beta_1) \sum_{i=0}^t \beta_1^{(t-1)} + \zeta$

applying the sum for a finite geometric series we finally have  $E[m_t] = E[g_t](1 - \beta^{t-1}) + \zeta$   
 So, to remove the bias given from the initial guess and look the expected value to look as we want we apply  $\widehat{m}_t = \frac{m_t}{1 - \beta^{t-1}}$  and  $\widehat{v}_t = \frac{v_t}{1 - \beta^{t-1}}$ .

Now, using those results we need to bring out how to perform the model weight update using the moving averages.

Using  $w$  to indicate the model weights,  $\eta$  the step size, we have

$$w_t = w_{t-1} - \eta \frac{\widehat{m}_t}{\epsilon + \sqrt{\widehat{v}_t}}$$

Depending on  $\eta$  values we can have different results in learning, for instance large values (e.g. 0.3) results in faster initial learning before the rate is updated.  $\epsilon$  is a very small number to prevent any division by zero in the implementation.

That's how Adam updates the weights of the model.

In our models we have tried different settings for those values in order to find better results.

The settings applied were taken from the default parameters setting used by different machine learning libraries since they are the main suggestion coming from papers, and are:

1. learning\_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08. as done by Tensorflow
2. lr=0.001, beta\_1=0.9, beta\_2=0.999, epsilon=1e-08, decay=0.0. as in Keras
3. learning\_rate=0.002, beta1=0.9, beta2=0.999, epsilon=1e-08, decay\_factor=1 as in Blocks

## Adadelta

Adadelta is an update of Adagrad, an algorithm for gradient-based optimization that adapts the learning rate to the parameters, performing smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequent features. Adadelta seeks to reduce its monotonically decreasing learning rate. the sum of gradients is recursively defined as a decaying average of all past squared gradients. At time  $t$  then, the running average depends only on the previous average and the current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2 \quad E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2.$$

The parameter update vector, based on the Adagrad one, takes the form:

$$\Delta \theta_t = \frac{-\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

$G_t \in \mathbb{R}^{d \times d}$  here is a diagonal matrix where each diagonal element  $i,i$  is the sum of the squares of the gradients w.r.t.  $\theta_i$  up to time  $t$ , while  $\epsilon$  is a smoothing term that avoids division by zero

In Adadelta we can so replace the diagonal matrix with with the decaying average over past squared gradients  $E[g^2]_t$ , so it becomes:

$$\Delta \theta_t = \frac{-\eta}{\sqrt{E[g^2]_t + \epsilon}} \odot g_t$$

Doing some evaluations and simplifications, noticing that in denominator we have the root mean squared (RMS), at the end we obtain the update rule:



$$\Delta\theta_t = \frac{-RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

## Categorical Cross-Entropy

In a multi-class classification problem as ours, categorical cross-entropy is perfectly suited. Using this loss, we will train a CNN to output a probability over the C classes available for classification. Categorical cross-entropy will compare the distribution of the predictions (the activations in the output layer, one for each class) with the true distribution, where the probability of the true class is set to 1 and 0 for the other classes. Every class is represented as a one-hot-encoded vector, so the less is the distance between the prediction of the classifier and this vector, the lower is the loss. Mathematically we have:

$$L(\theta) = -\frac{1}{n} \sum_{i=1}^n [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)] = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^m y_{ij} \log(p_{ij})$$

where i is an index for sample observations, j is an index for classes, y is the sample label which is one hot encoded, and p is the prediction for a sample, such that

$p_{ij} \in (0,1)$  and  $\sum_j p_{ij} = 1 \forall i, j$ .

Note that the binary cross-entropy is a special case for cross-entropy with m=2.

## Dropout

Dropout is a regularization technique that provides to randomly drop some nodes in one or more layers. With dropout we can set as parameter the probability of dropping a node, where 1 means that every node is kept and 0 that there is no output.

Usually dropout is used in hidden layers with a parameter set between 0.20 and 0.50, even 0.80 in some cases.

This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different “view” of the configured layer.

Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs.

Dropout can help in models where, due to small amount of data, the model tends to overfit and learn to generalize. Those models will have a bad testing result since they do not perform well on data never seen before. We’ve though to use this technique because the initial models that we’ve tested were having some overfitting issues and poor testing results.

## Training Phase

In this section we will present the graphical representation of the trends in the training phase of the 4 models which had the best results among all others.

## Fit

In our first attempt of training, We've noticed that the classifier tended to overfit on the training set, becoming unable to generalize and perform well on new, never seen, images. To help us obtaining better results, We've adopted a couple of callbacks which are: ModelCheckpoint and EarlyStopping.

```
cl_fit = k.models.load_model(cl_name)
cl_fit.fit(X_train, y_train_one, batch_size=250, epochs=20 ,shuffle=True,
          validation_data=(X_val, y_val_one), verbose=1 , callbacks=[history,mc,es])
```

The model is trained for 20 epochs with batches of 250 elements. Before each epoch the training set used to learn is shuffled. Every image has been converted in an array of integers containing the value of pixels in a range between [0,255]. The label of validation and test set have been encoded using one hot encoding, resulting in the following format:

```
array([[1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 1.]])
```

## ModelCheckpoint

```
mc = ModelCheckpoint('/classifiers/cl_fit.h5', monitor='val_loss', mode='min', save_best_only=True)
```

ModelCheckpoint is a callback that, given a variable to monitor, will save only the trained model that better performed on that variable.

In our case it has been set to monitor the decrease of validation loss and save only the best model seen.

Since it seemed that after some epochs the model was starting to overfit, this callback helped us to have only the best option, stopping at the point when the validation loss was still decreasing.

## EarlyStopping

```
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=5)
```

Early stopping is another callback that, while monitoring a specified parameter, will stop the training phase when after some epoch the previous threshold hasn't been exceeded.

Here we have set it to monitor the validation loss parameter and stop if after 5 epochs in a row where there were no decrease.

## History

history is a callback that will save in memory the statistics performed by the model in the training phase for each epoch. At the end of the training phase one can use it to plot useful graphs to understand the trend and make some adjustments.



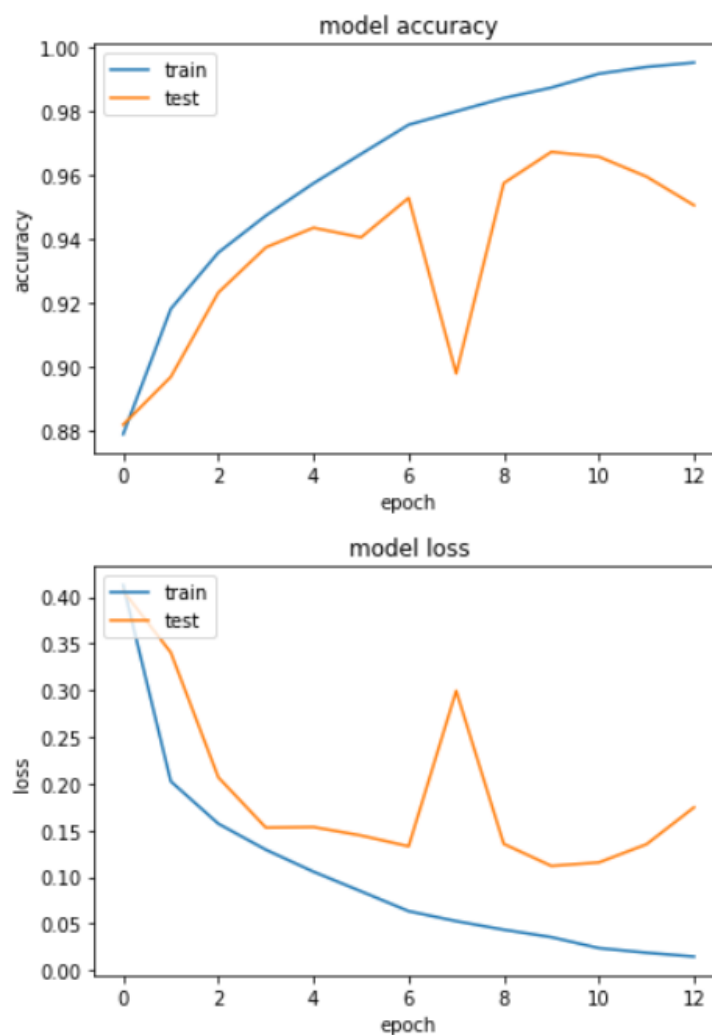
## Models

The 4 models have all the same structure, but they differ in the setting of optimization function.

The first model has been set as follow:

```
opt = k.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
return model
```

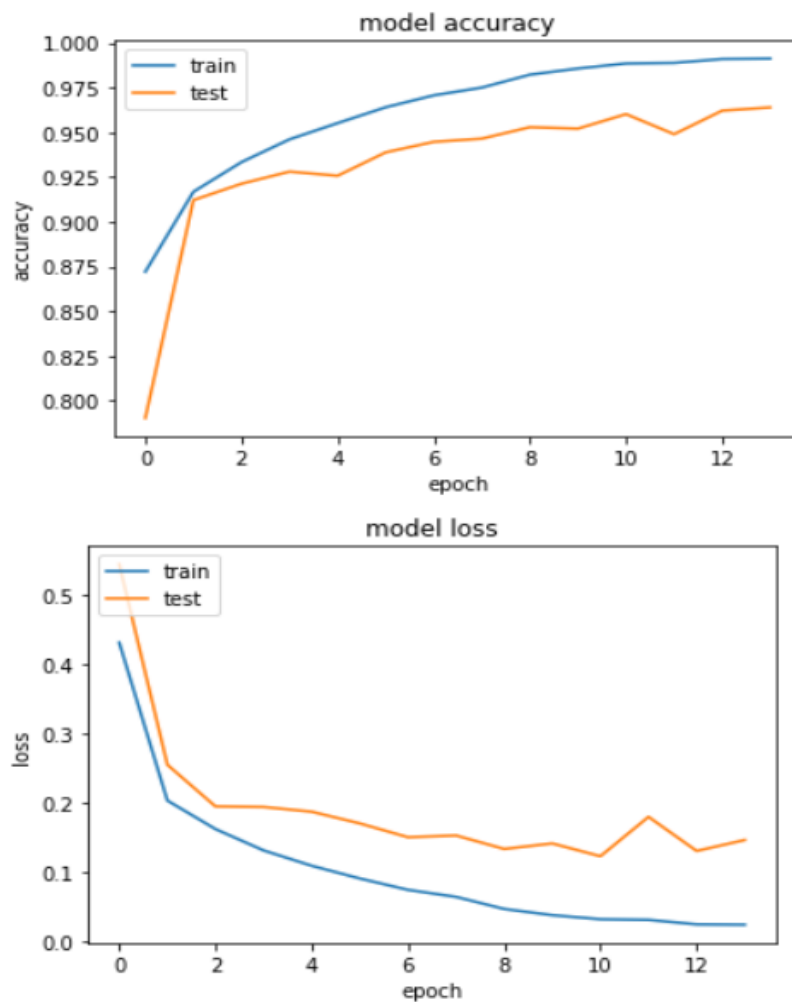
The graphs shows the trends of accuracy and loss evaluated on both training and validation set. As we can see this model had a peculiar irregularity between epoch 6 and 8. However, shortly after that, the val\_loss started to increase, indicating that the model was starting to stop to learn how to generalize on new, never seen, images. Thanks to callbacks only the best version of the model has been saved.



In model 2 the optimizer has been set using parameters suggested by tensorflow:

```
#tensorflow suggested parameters
opt = k.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=10e-8)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
return model
```

In this model the trend of the graph is really more regular than the previous one. Again, thanks to callbacks we had an early stopping when the model stopped to generalize and started to overfit.

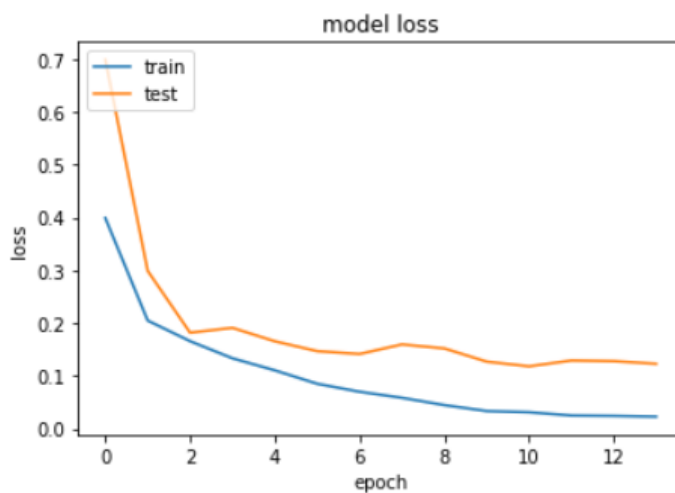
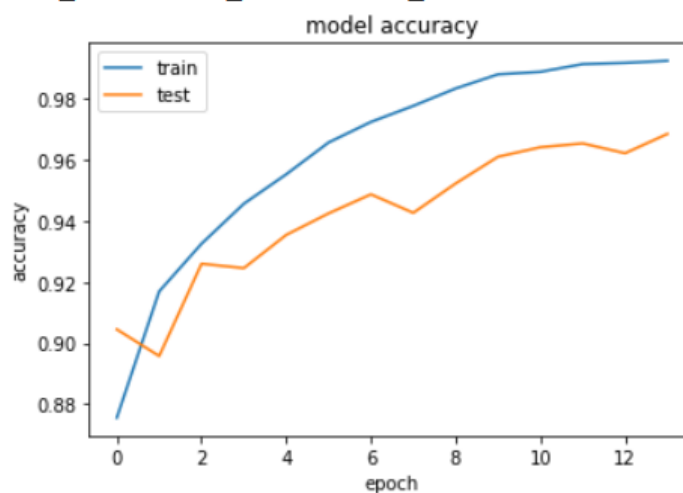


In model 3 the optimizer has been set with parameters suggested by keras:

```
#keras suggested parameters
opt = k.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=10e-8)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
return model
```

Even this third model has a smooth trend. The training phase goes on for a couple of epochs more with respect to the previous model. Apparently, the val loss after a big decrease in the initial phase, tends to become constant fairly quickly, which let us think that even going on for more epochs the final results wont change.

```
dict_keys(['val_loss', 'val_accuracy', 'loss', 'accuracy'])
```

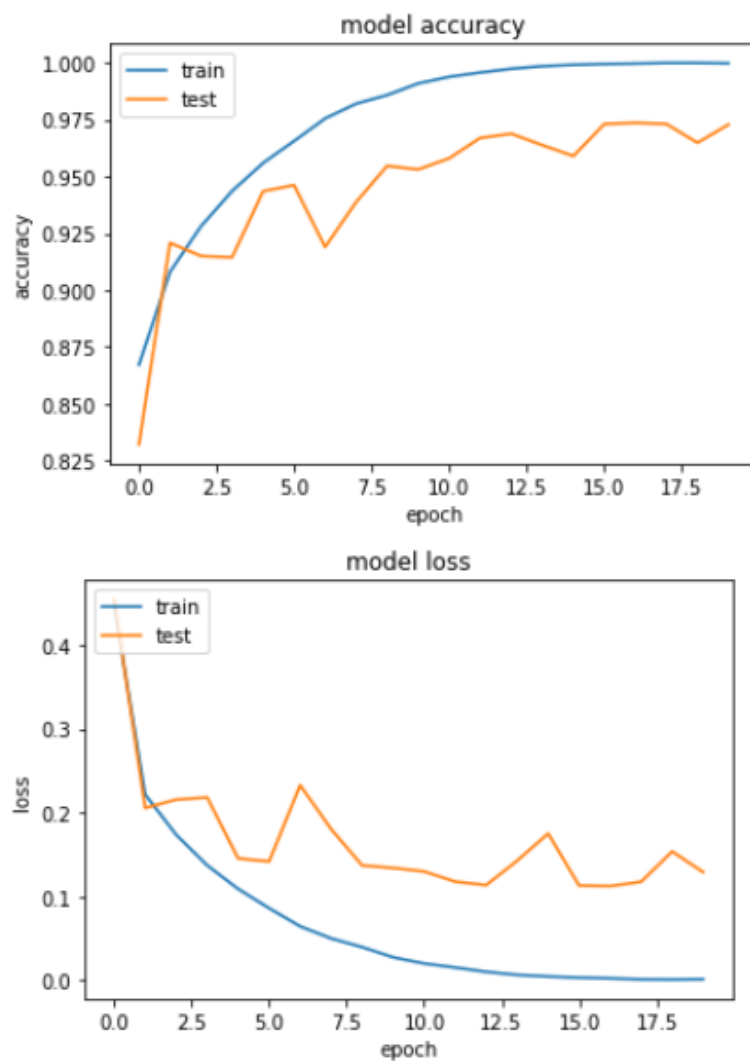




Finally, in our fourth model we choose to change the optimizer, using Adadelta instead of Adam. This decision has been taken because the test result of the previous three models, even if were satisfying, they hardly differed from each other.

```
model.compile(optimizer='adadelta', loss='categorical_crossentropy', metrics=['accuracy'])  
return model
```

This time the graph was really irregular, with the val\_loss very far from the loss. However, despite the trend, looking at values on y axis we can see that val\_accuracy has the highest esteem ever reached. From this point of view this model seems to be the best candidate among all its predecessors.



# Tests

We have then tested classifiers using the test set, which is a portion equal to 20% of the total original dataset. All the images presented in the test set are new for the classifier, and so a good candidate to make an evaluation about the ability to generalize of the model.

## Model 1

	precision	recall	f1-score	support
0.0	0.81	0.79	0.80	66
1.0	0.92	0.82	0.87	103
2.0	0.78	0.77	0.78	220
3.0	0.57	0.74	0.64	23
4.0	0.65	0.69	0.67	223
5.0	0.93	0.93	0.93	1341
6.0	0.97	1.00	0.98	29
accuracy			0.87	2005
macro avg	0.80	0.82	0.81	2005
weighted avg	0.87	0.87	0.87	2005

F1-score: 0.87

## Model 2

	precision	recall	f1-score	support
0.0	0.82	0.76	0.79	66
1.0	0.85	0.84	0.85	103
2.0	0.81	0.67	0.73	220
3.0	0.55	0.78	0.64	23
4.0	0.67	0.74	0.70	223
5.0	0.93	0.93	0.93	1341
6.0	0.63	0.93	0.75	29
accuracy			0.87	2005
macro avg	0.75	0.81	0.77	2005
weighted avg	0.87	0.87	0.87	2005

F1-score: 0.87

## Model 3

	precision	recall	f1-score	support
0.0	0.75	0.73	0.74	66
1.0	0.77	0.86	0.82	103
2.0	0.78	0.72	0.75	220
3.0	0.59	0.74	0.65	23
4.0	0.68	0.70	0.69	223
5.0	0.94	0.93	0.93	1341
6.0	0.72	1.00	0.84	29
accuracy			0.87	2005
macro avg	0.75	0.81	0.77	2005
weighted avg	0.87	0.87	0.87	2005

F1-score: 0.87

## Model 4

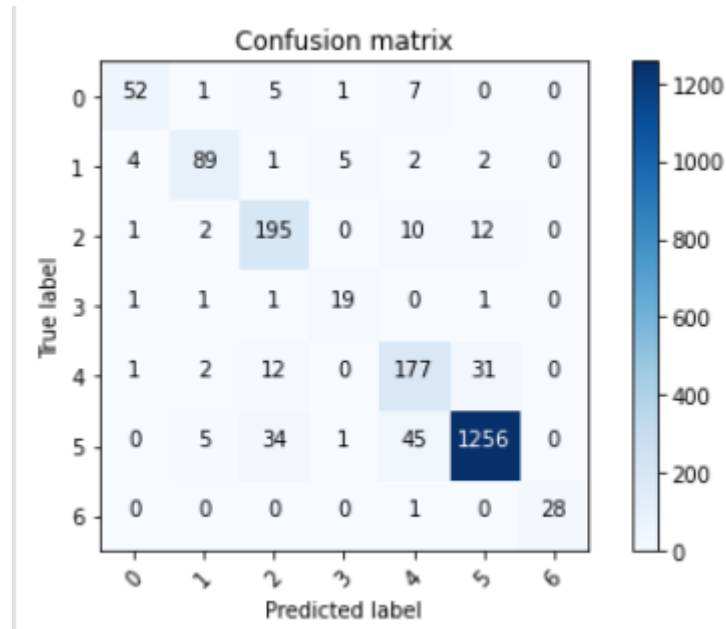
	precision	recall	f1-score	support
0.0	0.88	0.79	0.83	66
1.0	0.89	0.86	0.88	103
2.0	0.79	0.89	0.83	220
3.0	0.73	0.83	0.78	23
4.0	0.73	0.79	0.76	223
5.0	0.96	0.94	0.95	1341
6.0	1.00	0.97	0.98	29
accuracy			0.91	2005
macro avg	0.85	0.87	0.86	2005
weighted avg	0.91	0.91	0.91	2005

F1-score: 0.91

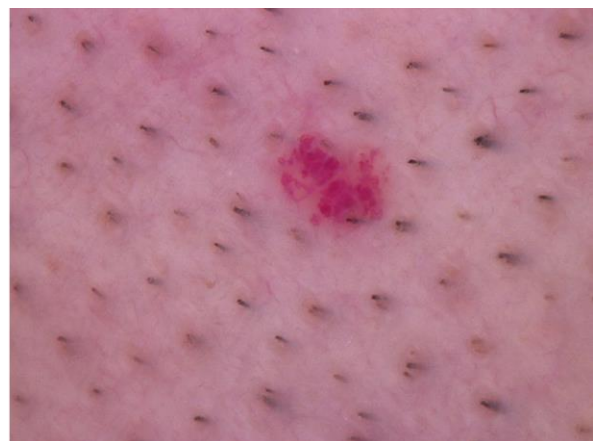
The results of the test phase represents what we thought looking to graphs. Model 4 , thanks to the characteristics of Adagrad, have the best results. The first three models have quite the same results on average, even if the first one has slightly better precision with respect to the other 2. The final F1-score of model 4, which is 91%, can be regarded as a good score, especially considering the hard nature of this type of classification. For this reason, model 4 is the one chosen as final model for this task.

## Results Analysis

The model previously shown are the best results that, in all the tests done, we have obtained. Has can be seen, the model has good performances on quite every class. The precision value ( $\frac{TP}{TP+FP}$ ) is high on every class, reaching the 80% on 4/6 of them. Even the recall ( $\frac{TP}{TP+FN}$ ) values are good on quite every class. In particular, the F1-score result is 0.91 and for us seems a really good value. The results indicates that, overall, a really good amount of elements is correctly classified in the correct predicted label.

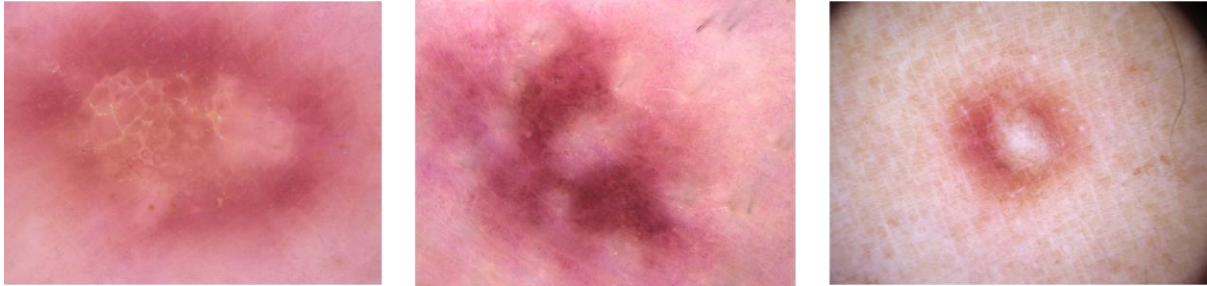


From both confusion matrix and final result of the prediction, we can see that some classes have better values than others, but this can be related to the characteristics of the images itself. For instance, class 6, which are “vascular lesions”, has reached high scores in every model that we’ve tested, and not only on the ones showed. This is reconnected, as said, to the structure of the images of this class which are hard to confuse with other kind of skin lesions, even to human eye , because are small red stains and are all really similar to each other:





Some other classes are harder, particularly 3 (dermatofibroma) and 4 (melanoma). For instance, looking at the confusion matrix for dermatofibroma, we can see that on 23 prediction it failed 4 of them, misclassifying the images as 4 different labels. Looking at dermatofibroma description we can see that it can take several different shapes and even different colors, making it harder to correctly classify since can be confused with other classes. For instance, looking at those three images we can see that there is not a particular pattern followed by this disease:



Another thing that comes out from confusion matrix is that images belonging classes 4 (Melanoma) 5 (Melanocytic nevi) and 2 (Benign keratosis-like lesions) are often confused with each other, showing that these families of skin lesions are similar.

Although a bigger image dataset would help to improve the classification task, we can feel satisfied with the results achieved especially on other classes in which we have a F1-score that is higher than 80%.

## References

Dataset:	<a href="#">Skin Cancer MNIST: HAM10000</a>
Diseases:	<a href="#">Mayo Clinic - Mayo Clinic</a>
Batch normalization:	<a href="#">How to use Batch Normalization with Keras? – MachineCurve</a>
Relu:	<a href="#">Activation Functions in Neural Networks</a>
Softmax:	<a href="#">Softmax Activation Function Explained</a>
Adam optimizer:	<a href="#">Gentle Introduction to the Adam Optimization Algorithm for Deep Learning</a> <a href="#">Adam — latest trends in deep learning optimization.</a>
Categorical cross-entropy:	<a href="#">A Gentle Introduction to Cross-Entropy for Machine Learning</a> <a href="#">AI glossary   Deep learning definitions</a> <a href="#">Should I use a categorical cross-entropy or binary cross-entropy loss?</a>
Dropout:	<a href="#">A Gentle Introduction to Dropout for Regularizing Deep Neural Networks</a> <a href="#">How to explain dropout regularization in simple terms? - Cross Validated</a>