AUTOMATIC CLASSIFICATION OF PLANKTON IMAGES BY DEEP LEARNING

PROJECT SUPERVISORS: Frédéric Precioso, Eric Debreuve, Jean-Olivier Irisson

POLYTECH NICE-SOPHIA

Anne LEGENDRE - Baptiste POUTHIER - Mathilde SCHNEIDER

Acknowledgement

We would like to express our sincere gratitude to our supervisors Mr. Eric Debreuve and Mr. Frédéric Precioso for providing guidance and advices throughout the course of the project.

We would like to thank Mr. Cédric Bailly, Mr. Thomas Mahiout, Mr. Thomas Jalabert for their help with the code.

We are also very thankful for the resources provided by the Laboratoire d'Océanographie de Villefranche/Mer and Mr. Jean-Olivier Irisson.

A. OBJECTIVES

The aim of the project is to continue an already existing study : the statistical learning for the automatic classification of zooplankton images. It is carried out in collaboration with the Oceanography Laboratory of Villefranche-sur-Mer (LOV) which has already tested 3 approaches :

- The use of classical features proposed by environmental biologist to feed a random forest classifier.
- A Deep Learning method (simultaneous learning of characteristics and classifier).
- Deep Learning combined with a Random forest classifier (hybrid method).

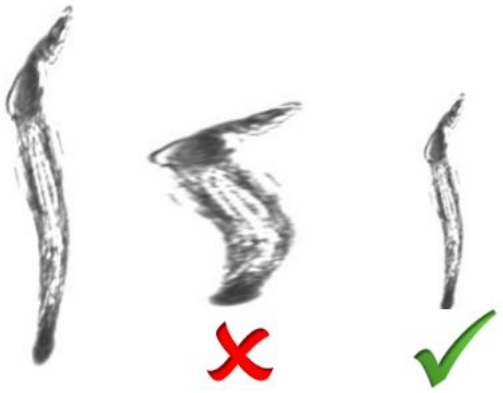
Our goal is to compare, select and combine the characteristics imagined by biologists and those learned by the Deep Learning approach and modify the Deep Learning architecture to improve the classification performance.

B. ANALYSIS OF THE PREVIOUS STUDY

1) Dataset

The dataset comes from the LOV. Some of the species are overrepresented while others are underrepresented.

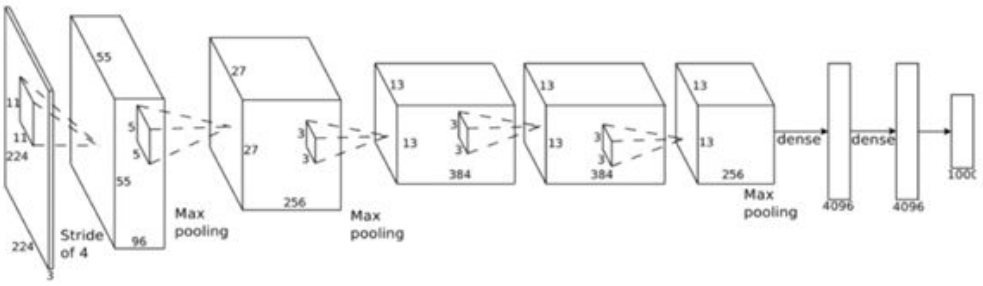
The images should have the same size while keeping their original shape. For this purpose the image is enlarged or shrunk in a uniform way so that its larger dimension corresponds to the required resolution and the image is completed with white pixels.



In addition, a Data Augmentation phase has been done in order to enlarge the size and the balance of images between the different classes. The used transformations are rotations, random flipping, translation, light shear and Gaussian noise. These transformations also make the learning more robust. Indeed, by creating many more or less fuzzy images in random directions or positions, the algorithm should be more robust when classifying unusual images.

2) Architecture

The network uses convolutions as well as different layers of processing functions: activation function, pooling, dropout. In this section we will describe the different steps used in the architecture.

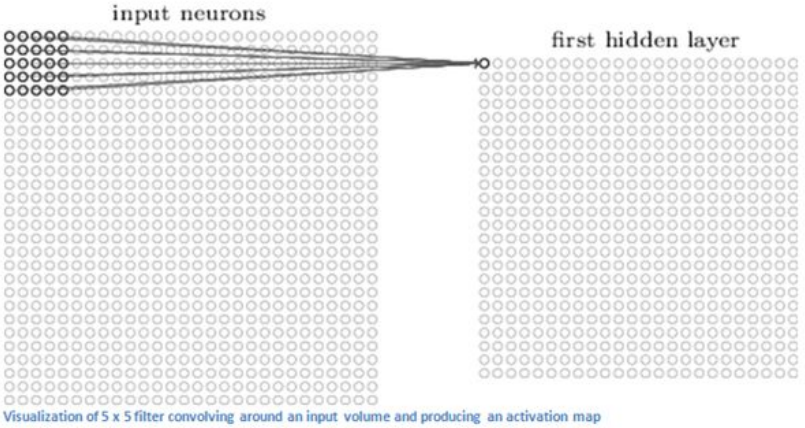


Credit:
https://stackoverflow.com/questions/42733971/convolutional-layer-to-fully-connected-layer-in-cnn

3) Convolutional layers

a. Generalities

Using filters that slide across the width and height of the input matrix (which is containing the grey levels of each pixel of the image), we compute the element-wise multiplication between the filter values and the input values at any position. Then, we sum the obtained products for each filter. This will generate the feature map (or the activation map). Each filter will produce its activation map that gives the responses of that filter at every spatial position. By stacking these feature maps, we get the output map.



Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

Credit:
https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

Convolution parameters:

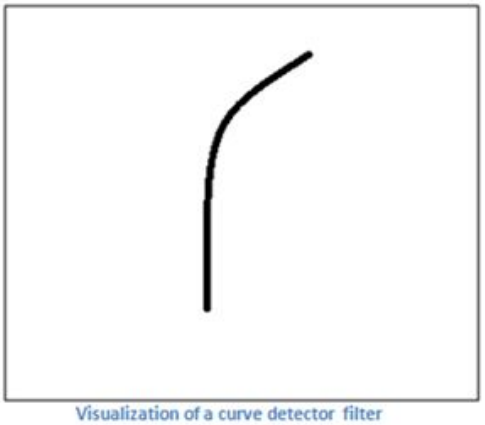
- Input matrix: 95x95
- Grey level (no RGB)
- Filters: 3x3

b. Filters

The filter is a pixel matrix in which there will be higher numerical values along the area that Is a shape of a curve.

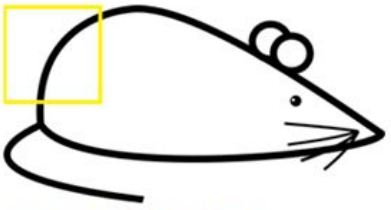
0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter



Visualization of a curve detector filter

Original image



Visualization of the filter on the image

Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

Credit:

https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

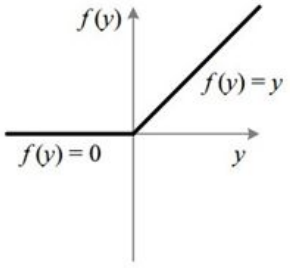
4) Activation function

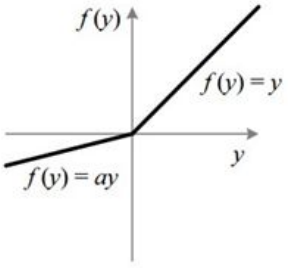
a. Generalities

After each convolution, we use an activation function.

b. Details

Chosen function: ReLU (Rectified Linear Unit) and more precisely LeakyReLU





ReLU: f(x) = max(0,x)

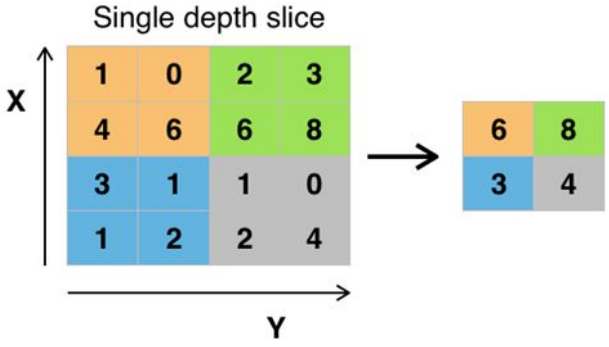
LeakyReLU: f(x) = x si x>0
f(x) = ax si x≤0

Credit: *https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6*

5) Pooling

a. Generalities

Once we know that a specific feature is remarkable in the input matrix, its exact position is not as important as its relative position to the other features. This reduces the output's dimensions, makes the network less sensitive to small deformations and manages overfitting. Pooling has an importance in the conservation of translation invariance



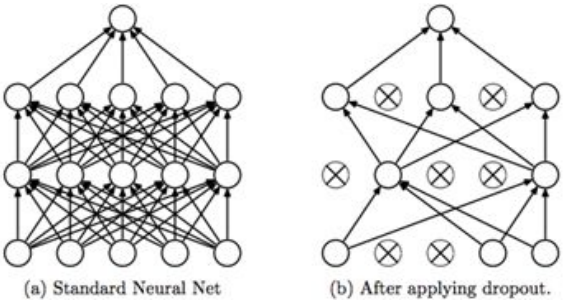
Credit:
https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

b. Details

Pooling method used: MaxPooling (it extracts, in a subregion of the feature map, the maximum value i.e. the most important information).

6) Dropout

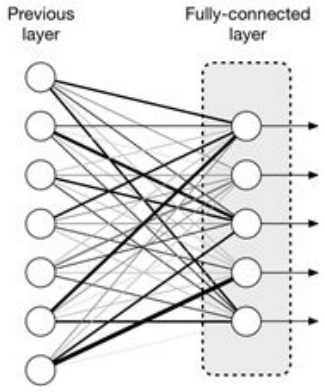
Dropping out random set of activation by killing some activated neuron force network to be redundant: the network should work even if some activations are missing. This avoid overfitting too.



Credit:
https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

7) Fully connected layer

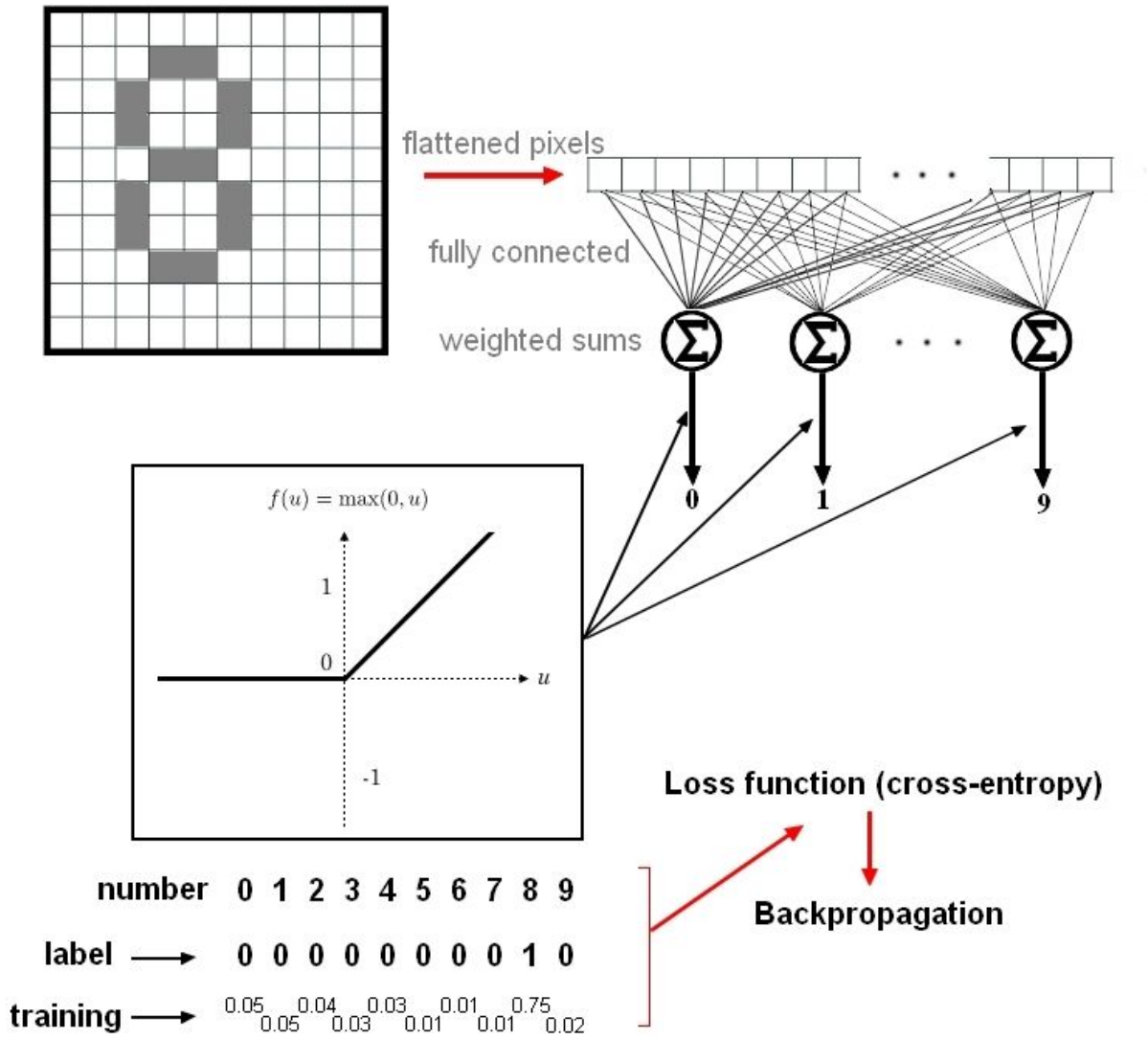
Placed at the end of the deep network, the fully connected layer makes possible to distinguish in output the probability that the input belongs to one of the classes. The output layer is as large as there are categories. This layer looks at the output of the previous layer and determines which features most correlate to a particular class. Of course, this layer also has weight and bias. There are as many output neurons as classes.



Credit: *http://machinethink.net/blog/mps-matrix-multiplication/*

8) Loss function and backpropagation

By comparing the result of a training with the input image label, it is possible to establish a loss function that depends on all the weights and the bias of the network. To do this, we use a function that report the distance between the experimental results and the expected results. The process allows to minimise the error of this loss function. The gradient vector is pointing towards the local minimum, according to a chosen learning rate. The gradient being calculated according to all the weight and bias, the components of this vector represent each of the small deviations from the current weights and bias that must be corrected to minimize the error function. Backpropagation is the principle of redefining weights and bias according to its small deviations.



9) Hyperparameters and precise architecture

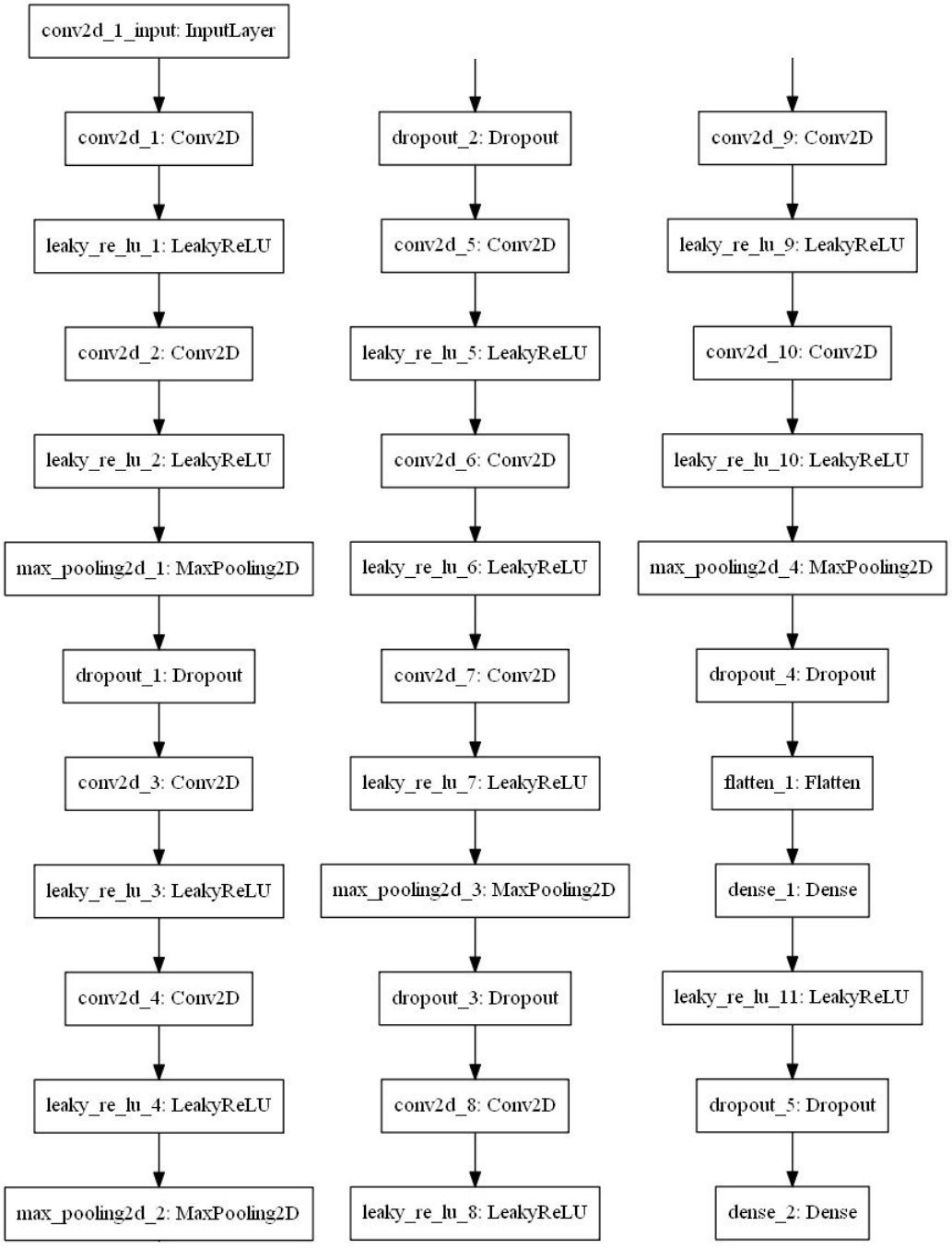
a. Hyperparameters

How many layers ? -> ~16

How many convolution layer? -> ~7

Filter size: 3x3

b. Architecture details



10) Hierarchical architectures

The dataset is composed by 60 species of plankton and by a taxonomy of these species.

The tree-based hierarchy makes use of this knowledge because its structure is similar to that of taxonomy. This hierarchy is made up of tree-based classifiers. Each classifier is trained to classify one part of the tree. The CNN-block of these classifiers is used for the extraction of features while the decision is made by the MLP-block. Each classifier develops features specific to its own part of the tree. A tree-based classifier is longer to train than a flat one because the whole dataset must be used at every level of the tree. The results are also equivalent or less precise than with a flat structure, but the improvements are easier because the inefficient sub-trees are identifiable and can be fine-tuned.

The error-based structure takes the results of the classifiers into account. The idea is to group classes which are difficult to distinguish and to create a new classifier whose only task is their distinction. This classifier is supposed to be better at differentiating these classes. A confusion matrix C is used in order to do so. C is such that $C(i,j)$ is the frequency of classifying an image in class j knowing that its real class is i. Clustering methods like hierarchical clustering, spectral clustering or k-means can be used but most of them require $C(i,j) = C(j,i)$, which is not always fulfilled. That is why C and C^T are multiplied by α (α<1) before the symmetric matrix is constructed. The multiplication enables to have reciprocal relations rather than unilateral ones. The confusion matrix becomes $\alpha C + \alpha C^T$.

The chosen method is inspired by the hierarchical clustering. The user chooses a threshold β, which is the acceptance value of the error. A new training is needed if the coefficient is greater than β. All values $C(i,j) \leq \beta$ are replaced by 0 because they don't need to be trained again, then the greatest coefficient $C(n,m)$ ($n \neq m$) is taken. Species n and m are grouped and the vectorial space is modified such that n is a linear combination of n and m. $C(n,m)$ becomes 0. This operation is repeated until no coefficient is positive except diagonal terms. In the end the matrix contains the clusters.

11) YOLO

YOLO is an object-detection algorithm. It only looks one time à the image (You Only Look Once) and try to detect objects. For this, it predicts bounding boxes that encloses each object and gives it a confidence score. Then it classifies it.

This model was implemented as a convolutional neural network. The initial convolutional layers of the network extract features from the image while the fully connected layers predict the output probabilities and coordinates. The network has 24 convolutional layers followed by 2 fully connected layers. However, instead of the inception modules used by GoogLeNet it simply uses 1 × 1 reduction layers followed by 3 × 3 convolutional layers. The full network is shown in Figure 3.

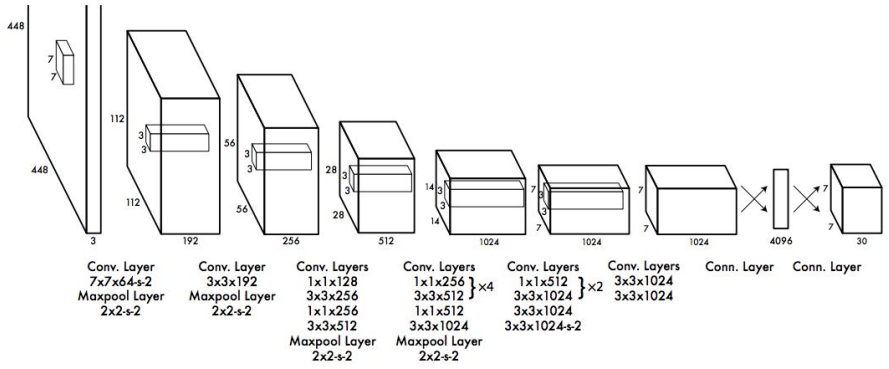


Figure 3: The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

Credit: *https://pjreddie.com/media/files/papers/yolo.pdf*

You can play with different parameters with YOLO : the algorithm divides the image into a SxS grid and predicts B bounding boxes for each grid cell and C classes possibilities. These predictions are encoded as an S × S × (B ∗ 5 + C) tensor. The previous students chose to use the Darkflow library.

Sadly, YOLO didn't come as efficient as hopped. It had worst results than a classical CNN. This can be explained by the limitations of YOLO. It encounters problems with the classification and detection of group of small objects like birds, this falls directly onto our plancton superposition problem. Also, YOLO treats errors the same in small bounding boxes versus large boxes, but, a small error in a big box doesn't have the same consequences as in a small box.

The choice of using YOLO can be wise because of its speed. A good question to ask ourselves is how to better YOLO when 2 planktons are superimposed ? Is it possible ?

We chose not to explore this approach after talking to the authors of the previous report, it was clear that YOLO didn't perform well.

C. APPLICATION

After studying the previous work, we understood that the best results were obtained by using the usual CNN without tree classifiers (flat classification). We chose to go further into it by testing it. First we tested 1 epoch on our computer, then on the NEF server to be able to train our network.

1) Dataset

Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called overfitting. To avoid it, it is common practice when performing a machine learning experiment to hold out part of the available data as a test set.

2) Cross validation

One round of cross-validation involves partitioning a sample of data into complementary subsets, performing the analysis on one subset (called the training set), and validating the analysis on the other subset (called the validation set or testing set). To reduce variability, in most methods multiple rounds of cross-validation are performed using different partitions, and the validation results are combined (e.g. averaged) over the rounds to give an estimate of the model's predictive performance.

We chose to use **train_test_split** from the **sklearn.cross_validation** module. This splits arrays or matrices into random train and test subsets using cross validation.

3) Results

We used the keras library to build and train our network. The function model.fit_generator takes for arguments the tain inputs and outputs and the batch size. As a results, it stocks :

- The **accuracy** for each epoch of training process
- The **validation accuracy**, meaning the accuracy when our model is tested with new data (test set)
- The **loss** for each epoch of the training process
- The **validation loss**, meaning the loss when our model is tested with new data (test set)

What makes for a good model ?

We want to estimate the ability of our model to generalize to now data. To do so, we need to look at the validation accuracy (**val_acc** when using keras) because the validation set contains only data that the model never sees during training and therefore can not memorize.

It is also important to understand overfitting situations : if the accuracy (**acc**) keeps improving while the validation accuracy (**val_acc**) gets worse, meaning the model starts to memorize the data.

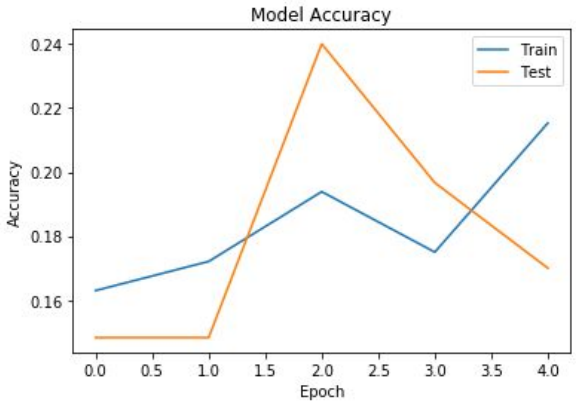
a. Variation of epochs

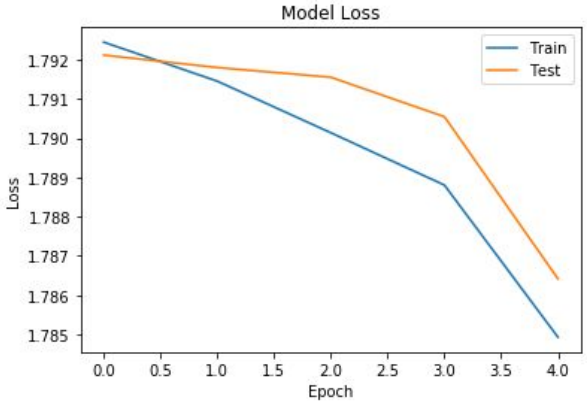
An epoch represents a training run over all of the training set. During that run the parameters of the model are adjusted according to the loss function. The result is a set of parameters which have a certain ability to generalize to new data. That ability is reflected by the validation accuracy.

Parameters:

- *Number of classes : 6*
- *Number of images : 6000 (1000 in each class)*
- *Batch size : 70*

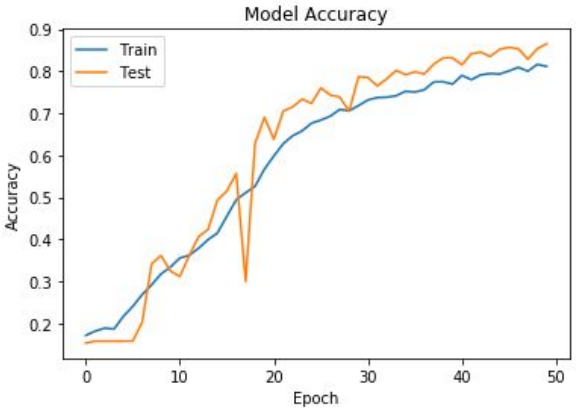
- 5 epochs

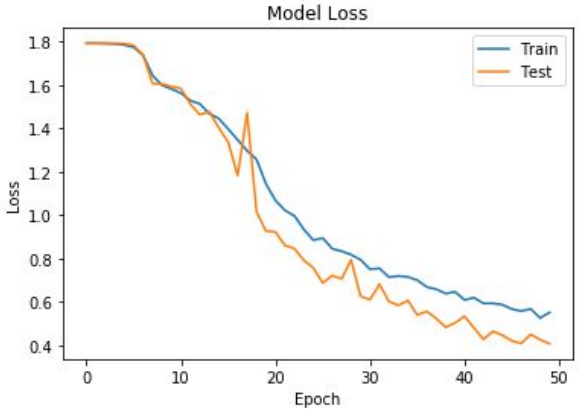





```
68/68 [==============================] - 262s 4s/step - loss: 1.7849 - acc: 0.2151 -
val_loss: 1.7864 - val_acc: 0.1700
```

- 50 epochs






```
68/68 [==============================] - 380s 6s/step - loss: 0.5517 - acc: 0.8115 -
val_loss: 0.4068 - val_acc: 0.8650
```

Interpretation:

With 5 epochs, the model is 17% accurate but with 50 epochs, we obtain an accuracy of 87%. It is clear that the number of epoch needs to be high enough to give the model a chance to be efficient.

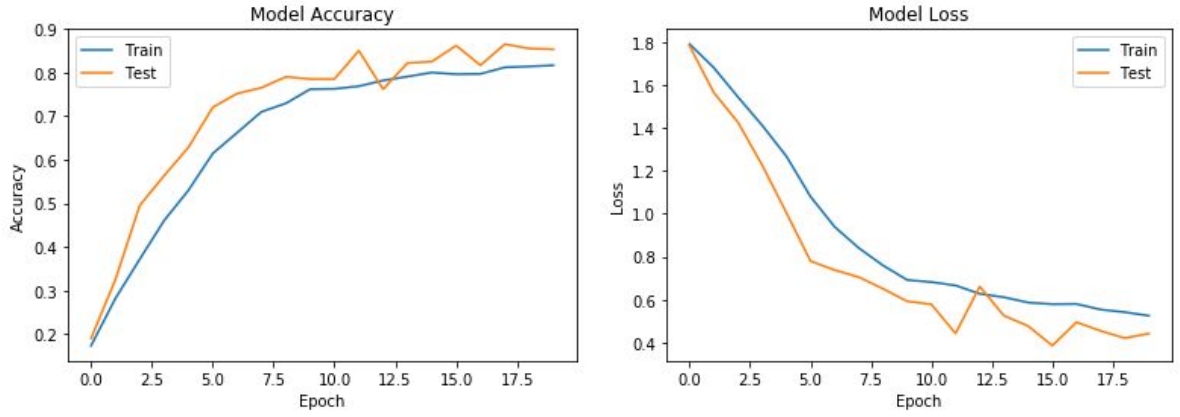
b. Variation of batch size

Batch size defines the number of samples that is going to be propagated through the network. For example, if **batch_size = 100**, the model takes the first 100 samples of the training dataset and trains the network. The process repeats until all samples are propagated through the network. The main advantage to dividing the dataset is that it requires less memory to train the model. Generally, networks train faster with little batches because it updates weights after each propagation. Usually, the smaller the batch the less accurate estimate of the gradient.

Parameters:

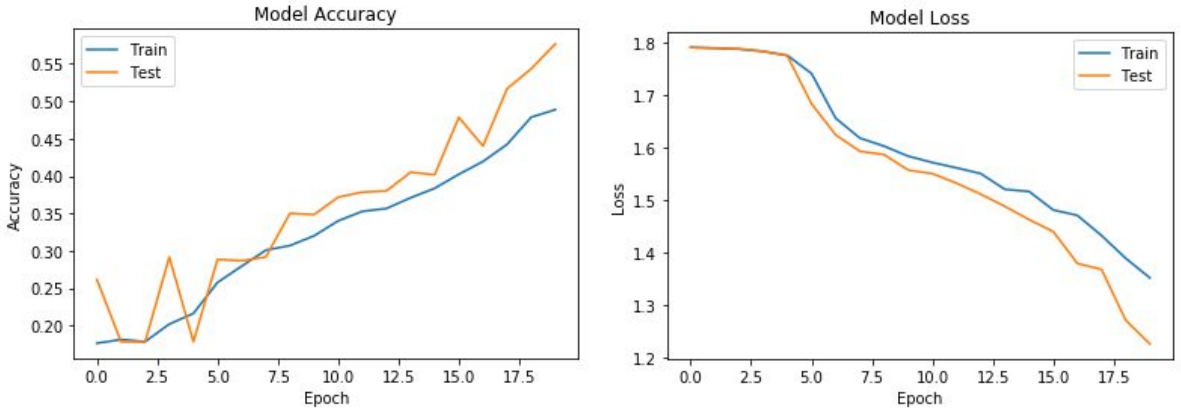
- *Number of classes : 6*
- *Number of images : 6000 (1000 in each class)*
- *Number of epochs : 20*

- Batch size : 10




```
480/480 [==============================] - 335s 698ms/step - loss: 0.5245 - acc: 0.8165 -
val_loss: 0.4413 - val_acc: 0.8533
```

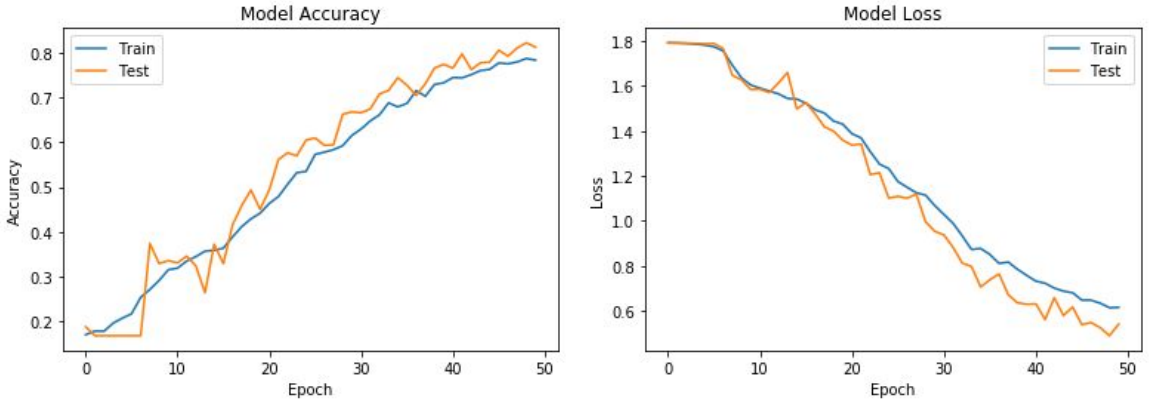
- Batch size : 100




```
48/48 [==============================] - 260s 5s/step - loss: 1.3519 - acc: 0.4887 - val_loss:
1.2261 - val_acc: 0.5767
```

At first, we were surprised to get a poor accuracy with a much bigger batch size. We assumed that having too big of a batch size has a negative effect on the accuracy, as much as a small one. We wanted to first increasing the number of epochs to verify our result.

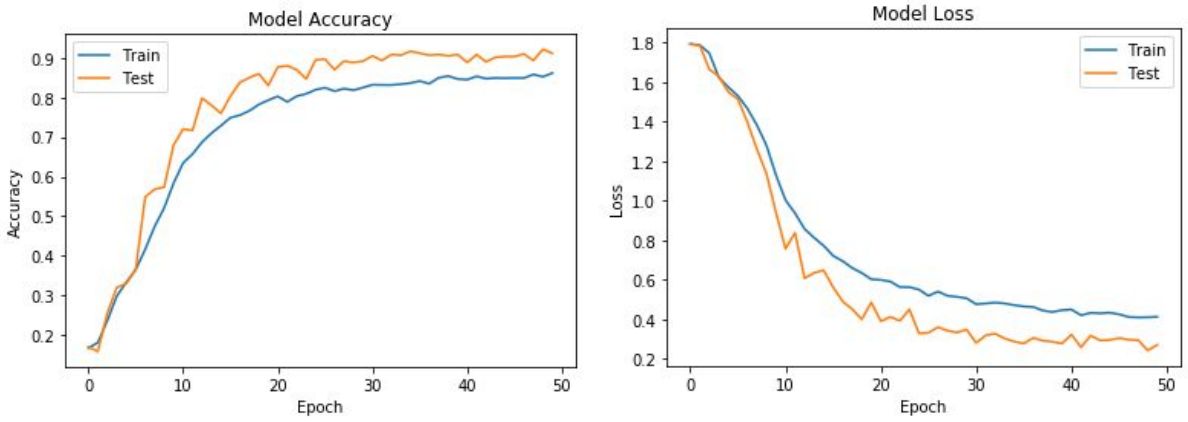
- Batch size = 100, 50 epochs




```
48/48 [==============================] - 270s 6s/step - loss: 0.6147 - acc: 0.7844 - val_loss:
0.5400 - val_acc: 0.8133
```

Even with 50 epochs, we got 81% accuracy, wich was still lower than with a batch size of 10 (85%). We then found an in between batch size, hoping to get a better result.

- Batch size = 30




```
160/160 [==============================] - 204s 1s/step - loss: 0.4125 - acc: 0.8617 -
val_loss: 0.2705 - val_acc: 0.9117
```

Interpretation:

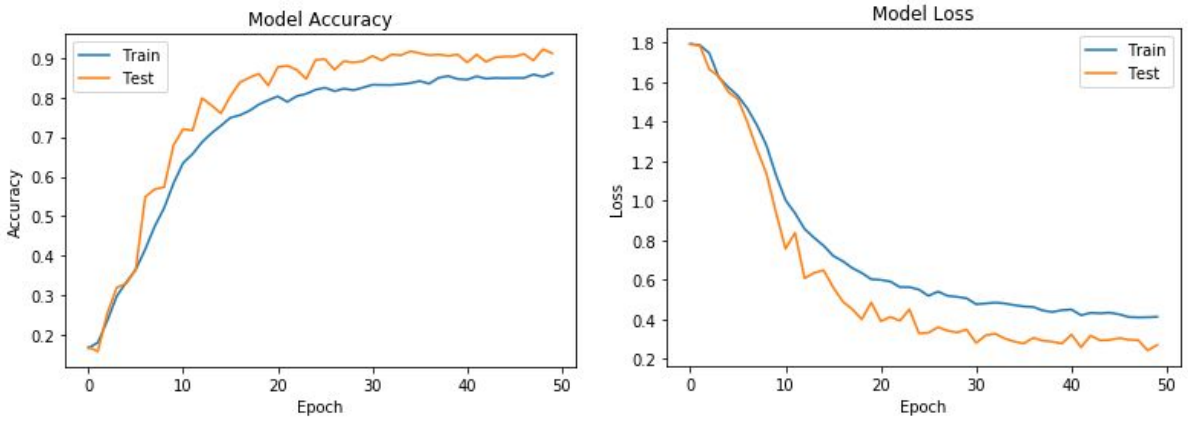
With a batch size of 30, we got a very satisfying accuracy of 91%. We can infer that depending on the problem, and most importantly the dataset, we need to adapt the batch size to get the best accuracy.

c. Variation of the number of images

Parameters:

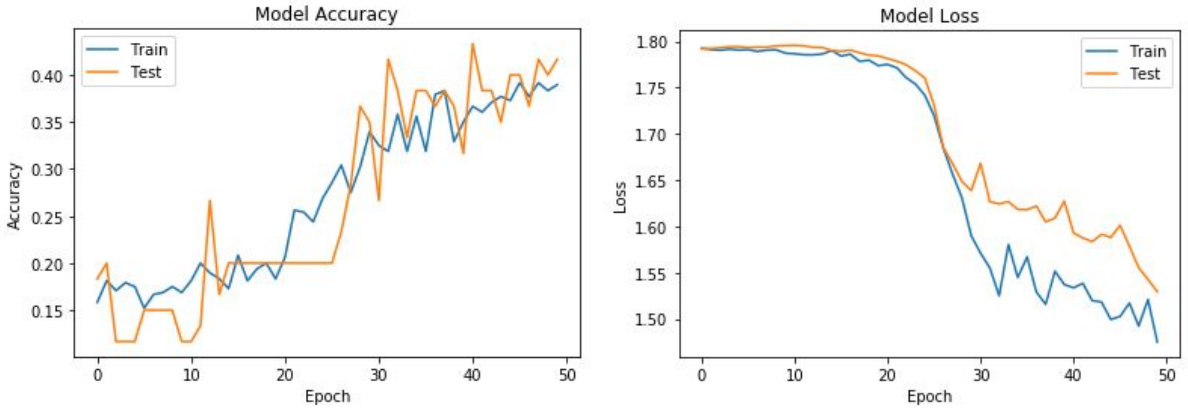
- *Number of classes : 6*
- *Batch size : 30*
- *Number of epochs : 50*

- 6000 images (1000 in each class)




```
160/160 [==============================] - 204s 1s/step - loss: 0.4125 - acc: 0.8617 -
val_loss: 0.2705 - val_acc: 0.9117
```

- 600 images (100 in each class)




```
16/16 [==============================] - 21s 1s/step - loss: 1.4752 - acc: 0.3896 -
val_loss: 1.5295 - val_acc: 0.4167
```

Interpretation:

We obtained 91% accuracy with 6000 images in the dataset. With 600 images, the accuracy decreased drastically to 41%. It is necessary to have a big enough dataset to ensure accuracy.

4) NEF

NEF is the Inria computing cluster. It is very powerful and useful for executing long and heavy tasks while our personal computers are not adapted.

The platform is open for all people with an Inria account but the GPUs can be bought by the different teams so that they have priority when executing jobs. It means that our training is killed when someone with priority submit a job to the GPU we are using. Control points have been set up in our code in order to get around this problem. These control points memorize the last computed weights and the time of the killing.

NEF is using the OAR system, that is why its syntax is specific. Here is a submission example :
oarsub -S ./start-naif.sh -t besteffort -p "gpu='YES' and host='nefgpu09.inria.fr" -p 'mem >150000' -l /nodes=1, walltime=20:00:00.

- **oarsub -S ./start-naif.sh** submits the script
- **-t besteffort** submits the job to the best effort queue
- **-p "gpu='YES' and host='nefgpu09.inria.fr"** reserves the GPU "nefgpu09"
- **-p 'mem >150000'** changes the memory allocated to the job
- **-l /nodes=1** uses one single node
- **walltime=20:00:00** reserves the node for at most 20 hours

The result we get for the following parameters :

- Number of classes : 121
- Number of images : 30 336
- Batch size : 64
- Number of epochs : 100

Result :

```
Epoch 00100: val_acc did not improve from 0.72907
Test accuracy = 0.6977587343441002
```

This is our best result because this training matches the theoretical utilisation of the algorithm after the training phase.

D. CONCLUSION

During this project we had to continue an existing project which aims to classify plankton images using deep learning. The main objectives were to understand and compute the code and to manage the NEF cluster. We had some other interesting axis of research we could have explore: confusion matrix and layer classifier.

The first problem we faced was the reuse of an already existing code. It was difficult to understand the architecture of the classes and the code itself because of the lack of comments. We also had difficulties to use the NEF cluster because we were new to this platform and the documentation was not adapted to beginners. Once those problems solved, we computed different programs with different parameters to explain the theoretical approach we had before computing the programs. We faced a lot of problems regarding the time of computation: it was impossible to run our program with 121 classes using our personal CPU. So we focused on smaller parameters. At the end of the project, we found out how to use NEF and then we had the chance to compute bigger programs.

It is important to remember to adapt every parameters depending on the problem. The number of epochs, batch size etc have to be tuned in order to fit the problem. There is very little chance that the parameters chosen can be relevant for another problem.

Nevertheless, we can say that :

- The number of epochs have to be high enough
- The batch size need to be adapted to the size of the dataset (not too high, not too low)
- The dataset needs to be big and rich enough to train our model

We can also keep from this study that using the usual CNN without tree classifiers (flat classification) and cross validation gives really good results.

This project being precursor to our future internships, we feel ready to work about deep learning and to use computation machines. We acquired strong knowledge in machine learning and we had the chance to be guided about how to improve deep learning architectures.

E. CODE

```python
import numpy as np
from skimage import transform
from skimage import io
import os
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Flatten
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import Conv2D
from keras.preprocessing.image import ImageDataGenerator
from sklearn.cross_validation import train_test_split
from keras.layers.pooling import MaxPooling2D
from keras.optimizers import SGD
from keras.callbacks import ModelCheckpoint
from pathlib import Path
import glob
import matplotlib.pyplot as plt

NUM_CLASSES = 6  # 38
IMG_SIZE = 95
batch_size = 10

d = dict()


def preprocess_img(img):
    # rescale to standard size
    img = transform.resize(img, (IMG_SIZE, IMG_SIZE))

    # roll color axis to axis 0
    img = np.rollaxis(img, -1)
    img = img.reshape(img.shape + (1,))

    return img


def get_class(img_path):
    return img_path.split("/")[-2]


```

```python

def cnn_model():
    model = Sequential()

    model.add(Conv2D(32, (3, 3), padding='same',
                     input_shape=(IMG_SIZE, IMG_SIZE, 1)))
    model.add(LeakyReLU(alpha=(1 / 3)))
    model.add(Conv2D(16, (3, 3)))
    model.add(LeakyReLU(alpha=(1 / 3)))
    model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
    model.add(Dropout(0.2))

    model.add(Conv2D(64, (3, 3), padding='same'))
    model.add(LeakyReLU(alpha=(1 / 3)))
    model.add(Conv2D(32, (3, 3)))
    model.add(LeakyReLU(alpha=(1 / 3)))
    model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
    model.add(Dropout(0.2))

    model.add(Conv2D(128, (3, 3), padding='same'))
    model.add(LeakyReLU(alpha=(1 / 3)))
    model.add(Conv2D(128, (3, 3)))
    model.add(LeakyReLU(alpha=(1 / 3)))
    model.add(Conv2D(64, (3, 3)))
    model.add(LeakyReLU(alpha=(1 / 3)))
    model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
    model.add(Dropout(0.2))

    model.add(Conv2D(256, (3, 3), padding='same'))
    model.add(LeakyReLU(alpha=(1 / 3)))
    model.add(Conv2D(256, (3, 3)))
    model.add(LeakyReLU(alpha=(1 / 3)))
    model.add(Conv2D(128, (3, 3)))
    model.add(LeakyReLU(alpha=(1 / 3)))
    model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
    model.add(Dropout(0.2))

    model.add(Flatten())
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=(1 / 3)))
    model.add(Dropout(0.5))
    model.add(Dense(NUM_CLASSES, activation='softmax'))
    return model


```

```
def prepareData():
    root_dir = '/Users/annelegendre/Polytech/MAM4/MAM4S2/Projet/pfe_plankton_classif-master/test6c/classes';

    imgs = []
    labels = []
    k = 0
    all_img_paths = glob.glob(os.path.join(root_dir, '*/*.jpg'));
    #all_img_paths = glob.glob(root_dir);
    np.random.shuffle(all_img_paths)
    for img_path in all_img_paths:
        img = preprocess_img(io.imread(img_path))
        label = get_class(img_path)
        if (label not in d):
            d[label] = k;
            k = k + 1;
        imgs.append(img)
        labels.append(label)
        if (len(labels) % 100 == 0):
            print(str(len(labels)) + " imgs treated...");
    print(str(len(labels)) + " Done")

    label_chiffre = [];
    for label in labels:
        label_chiffre.append(d.get(label));

    X = np.array(imgs, dtype='float32')
    # Make one hot targets
    Y = np.eye(NUM_CLASSES, dtype='uint8')[label_chiffre]

    return X, Y


def lr_schedule(epoch):
    return lr * (0.1 ** int(epoch / 10))

imagesKaggles = "/Users/annelegendre/Polytech/MAM4/MAM4S2/Projet/pfe_plankton_classif-master/test6c/Kaggles.npy"
imagesKagglesLabels = "/Users/annelegendre/Polytech/MAM4/MAM4S2/Projet/pfe_plankton_classif-master/test6c/Kaggleslab.npy"

pathImages = Path(imagesKaggles);
pathImagesLabels = Path(imagesKagglesLabels);

if (pathImages.exists()):
    print("Load Data...")
    X = np.load(imagesKaggles);
    Y = np.load(imagesKagglesLabels);
else:
    X, Y = prepareData();
    np.save(imagesKaggles, X);
    np.save(imagesKagglesLabels, Y);

X_train, X_val, Y_train, Y_val = train_test_split(X, Y,
                                                  test_size=0.2)  # , random_state=42)
```

```

X_val, X_test, Y_val, Y_test = train_test_split(X_val, Y_val, test_size=0.5)

train_datagen = ImageDataGenerator(
    featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    rotation_range=45,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.1,
    zoom_range=0.2,
    fill_mode='nearest',
    cval=0.,
    horizontal_flip=True,
    vertical_flip=True,
    preprocessing_function=None
)


train_datagen.fit(X_train) #training


## WITHOUT DATA AUGMENTATION
model = cnn_model()


# let's train the model using SGD + momentum
lr = 0.003
sgd = SGD(lr=lr, momentum=0.9, nesterov=True)

checkpoint = ModelCheckpoint(
    '/Users/annelegendre/Polytech/MAM4/MAM4S2/Projet/pfe_plankton_classif-master/test6c/save',
    monitor='val_acc', verbose=1, save_best_only=True, mode='max')
callbacks_list = [checkpoint]
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])
h1=model.fit_generator(train_datagen.flow(X_train, Y_train, batch_size=batch_size),
                    steps_per_epoch=X_train.shape[0] // batch_size,
                    epochs=20,
                    validation_data=(X_val, Y_val),
                    callbacks=callbacks_list)  # validation_split pour split auto


```

```
Y_pred = model.predict(X_test)

predClasses = Y_pred.argmax(axis=-1)
trueClasses = Y_test.argmax(axis=-1)

acc = np.sum(predClasses == trueClasses) / np.size(predClasses)
print("Test accuracy = {}".format(acc))

# summarize history for accuracy
plt.plot(h1.history['acc'])
plt.plot(h1.history['val_acc'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'])
plt.show()
# summarize history for loss
plt.plot(h1.history['loss'])
plt.plot(h1.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'])
plt.show()
```