The Data

Download the Cornell data on the website https://github.com/LaurentBimont/process_Cornell
Clone the files and install them on your system

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import os
import cv2

! rm -r process_Cornell
! git clone https://github.com/LaurentBimont/process_Cornell

X = []
folder_name = 'process_Cornell/x_split/'
for file_num in range(len(os.listdir(folder_name))):
  X.extend(np.load(folder_name + 'x_{}.npy'.format(file_num)))

X = np.array(X)
Y = np.load('process_Cornell/all_Y_test_format.npy', allow_pickle=True)
Y = np.array([np.array(y) for y in Y])
```

```
Cloning into 'process_Cornell'...
remote: Enumerating objects: 146, done.
remote: Total 146 (delta 0), reused 0 (delta 0), pack-reused 146
Receiving objects: 100% (146/146), 112.18 MiB | 22.46 MiB/s, done.
Resolving deltas: 100% (31/31), done.
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:17: VisibleDeprecationW
```

Representation of the data

For the grasping representation we use 5 grasping parameters: x, y, θ, w et h. In the following, we will try to predict them with a neural network.

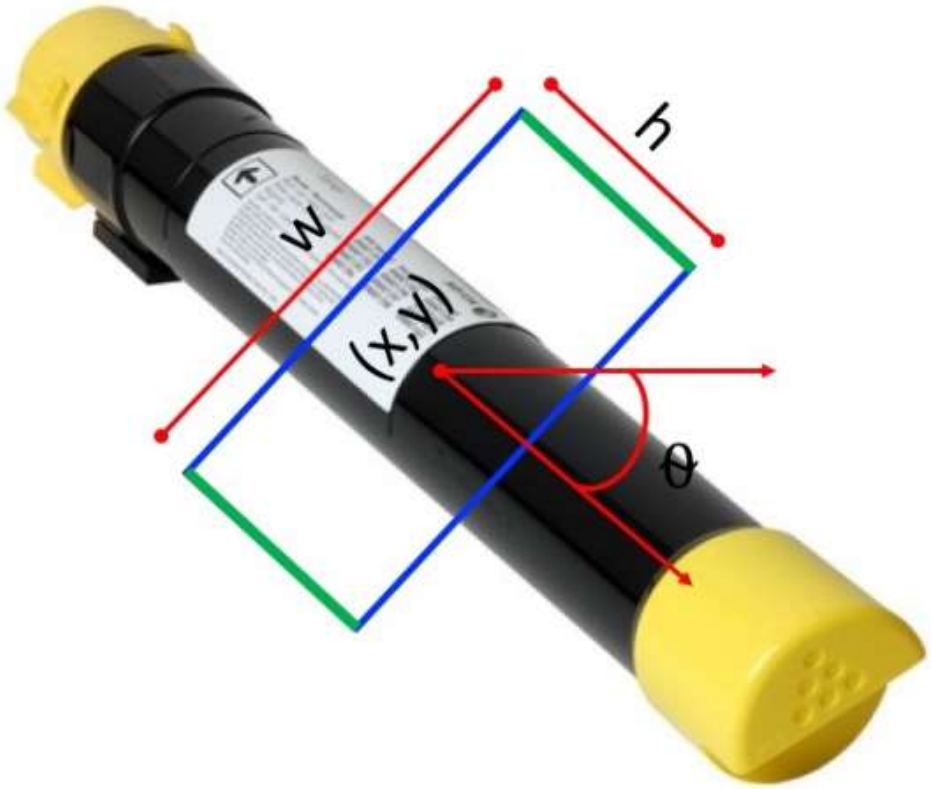


Fig. 1. An example grasp rectangle for a potential good grasp of a toner cartridge. This is a five-dimensional grasp representation, where green lines represent parallel plates of gripper, blue lines correspond to the distance between parallel plates of the grippers before grasp is performed, (x,y) are the coordinates corresponding to the center of grasp rectangle and θ is the orientation of the grasp rectangle with respect to the horizontal axis.

There are several things to note:

For an image there are several possible grasping parameters:

- The variable y_train contains all possible grasping rectangles. So it's a list list for example:
$$[(x_1, y_1, \theta_1, w_1, h_1), (x_2, y_2, \theta_2, w_2, h_2), (x_3, y_3, \theta_3, w_3, h_3)]$$
- Consideration should be given to how to use y_train to effectively train a network. To view the rectangles on an image using python, you have to convert these grasping parameters into a rectangle representation (the coordinates of the 4 sides). To move from one to the other one has the functions 'bboxes_to_grasps' and 'grasp_to_bbox'.
- The 'vizualise' function allows grasping rectangles to be displayed on an image. The green lines represent the location of the robotic grasp.
- To determine if a prediction is correct, the Jacquard metric, also known as IoU (Intersection over Union), is used. Consider a rectangle of grasping predicts $\mathcal{R}_{pred}$ and the

true $\mathcal{R}_{true}$, the intersection on the union of these rectangles must be greater than 0.25:

$$\frac{\mathcal{R}_{true} \cap \mathcal{R}_{pred}}{\mathcal{R}_{true} \cup \mathcal{R}_{pred}}$$

In addition, the angle difference should be less than 30 degrees. The 'performance' function performs this test.

```
print('Input dimensions: ', X.shape)
print('Output dimensions: ', Y.shape)
print('First element of Y', Y[0])
```

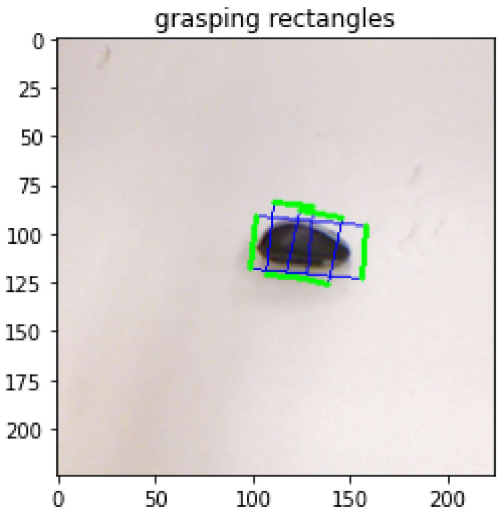
```
Input dimensions:  (750, 224, 224, 3)
Output dimensions:  (750,)
First element of Y [[139.80952381 125.69980952  84.80557109  25.18899086  54.9706415
 [121.9047619  121.73942857  81.31588232  26.18337566  60.55401418]
 [104.         119.26247619  84.80557109  22.86983774  70.18123129]]
```

Fonctions utiles

```
## Draw the grasping rectangles on an  image
def draw_rectangle(mes_rectangles, image):
  '''
  mes_rectangles is a list of 4 points of a rectangle
  image is a numpy array where we draw the grasping rectangles
  '''
  for rectangle in mes_rectangles:
      point1, point2 = tuple([int(float(point)) for point in rectangle[0]]), tuple(
          [int(float(point)) for point in rectangle[1]])
      point3, point4 = tuple([int(float(point)) for point in rectangle[2]]), tuple(
          [int(float(point)) for point in rectangle[3]])
      cv2.line(image, point1, point2, color=(0, 0, 255), thickness=1)
      cv2.line(image, point3, point4, color=(0, 0, 255), thickness=1)
      cv2.line(image, point2, point3, color=(0, 255, 0), thickness=2)
      cv2.line(image, point4, point1, color=(0, 255, 0), thickness=2)
  return image


def vizualise(x, y):
  '''
  x : is a raw image
  y : is a list of lists with the grasping parameters

  even if you visualize only one grasping parameter,
  there must be a list of list
  '''
  tot_rect = []
  for box in y:
    if box[0]<500:
      rect = grasp_to_bbox(box)
```

```
        rect = [float(item) for vertex in rect for item in vertex]
        grasp = bboxes_to_grasps(rect)
        new_rect = grasp_to_bbox(grasp)
        tot_rect.append(new_rect)
    image = draw_rectangle(tot_rect, x)
    plt.imshow(image)
    plt.title('grasping rectangles')


def bboxes_to_grasps(box):
    '''
    convert a rectangle into the grasping parameters
    '''
    x = (box[0] + (box[4] - box[0])/2)
    y = (box[1] + (box[5] - box[1])/2)
    if box[0] == box[2]:
        tan = 30
    else:
        tan = -(box[3] - box[1]) / (box[2] - box[0])
    tan = max(-11, min(tan, 11))
    w = np.sqrt(np.power((box[2] - box[0]), 2) + np.power((box[3] - box[1]), 2))
    h = np.sqrt(np.power((box[6] - box[0]), 2) + np.power((box[7] - box[1]), 2))
    angle = np.arctan(tan) * 180/np.pi
    return x, y, angle, h, w


def grasp_to_bbox(grasp):
    '''
    convert the grasping parameters into a rectangle
    '''
    x, y, theta, h, w = tuple(grasp)
    theta = theta * np.pi/180
    edge1 = [x - w/2*np.cos(theta) + h/2*np.sin(theta), y + w/2*np.sin(theta) + h/2*np.cos(t
    edge2 = [x + w/2*np.cos(theta) + h/2*np.sin(theta), y - w/2*np.sin(theta) + h/2*np.cos(t
    edge3 = [x + w/2*np.cos(theta) - h/2*np.sin(theta), y - w/2*np.sin(theta) - h/2*np.cos(t
    edge4 = [x - w/2*np.cos(theta) - h/2*np.sin(theta), y + w/2*np.sin(theta) - h/2*np.cos(t
    return [edge1, edge2, edge3, edge4]


from shapely.geometry import Polygon

def performance(Y_pred, Y_true):
    '''
    Y_pred and Y_true are the two grasping parameters
    '''
    grasp_pred = grasp_to_bbox(Y_pred)
    grasp_true = grasp_to_bbox(Y_true)

    p_pred = Polygon(grasp_pred)
    p_true = Polygon(grasp_true)

    iou = p_pred.intersection(p_true).area / (p_pred.area + p_true.area - p_pred.intersectic
    theta_pred, theta_true = Y_pred[2], Y_true[2]
    if iou > 0.25 and (np.abs(theta_pred-theta_true) < 30 or np.abs(theta_pred % 180-theta_t
        return True
```

```
    else:
        return False



n = np.random.randint(500)
vizualise(X[n], Y[n])
```



Build your first network for the grasping prediction

For this you must:

- Define a network architecture
- Define a regression loss function
- Define a Y format well adaptated (the current format is not adaptated)
- Set a function to evaluate the number of successfull grasping on the test data test place une fonction pour déterminer le nombre de grasping (you have to reach a success rate of 40-50%)

Architecture + Loss function

Complete the following function with layers, compilation method and suitable loss function

- Test of different architectures
- Choose an optimizer (https://keras.io/optimizers/)
- Choose a loss function suitable for regression (https://keras.io/losses/)

NB: you can use "transfer learning"

```
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Input, Dense, Dropout, BatchNormalization

def model_1():
  model = Sequential()
```

```
###############
## TO DO     ##
###############

model.compile(''' TO DO''')
return model
```

Training

Prepare the data so that you can train your network.

- Separate your data in train / test thanks to the function: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
- The Y has the following format: (number of photos, number of possible graspings, 5). However, to function correctly, Y must be in the format (nb photo, 5). Y must therefore be reworked by selecting only one grasping parameter among all those possible. Do this by selecting only the first parameter of grasping for example.
- Train your model by choosing a batch_size and the number of epochs

Remarks:

- you are free to set up callbacks or other functions to improve your network
- You are also free to view the training history to analyze what happened and suggest improvements to your network
- You are free to use transfer learning or not.
- Do not use data augmentation, we will do it after

```
## Data preparation
# To avoid having to reload data all the time, make a copy of the original data
X1, Y1 = np.copy(X), np.copy(Y)


#### Data separation into x_train, y_train
x_train, x_test, y_train, y_test = '''TO DO: Use the function correctly train_test_split''

y_test_copie = np.copy(y_test)    # We keep a y_test in the initial format

### Give Y an acceptable format ()
y_train = ''' TO DO '''
y_test = ''' TO DO '''
# test should return (something, 5)
print('test', y_train.shape)


## Training
model = model_1()
model.fit(''' TO DO ''')
```

Once you have obtained a satisfactory model, you can save it so that you do not have to retrain it each time.

```
## Save the weights of your model
model.save_weights('model_1.h5')
## To load a previously trained model (that of the final model or that of a checkpoint)
model = model_1()
model.load_weights('Magat
```

Performance

Evaluating the exact capacities of your network is essential to be able to improve it as you go. The value of the loss function is not very meaningful, so we need to develop our own metric.

Here we are going to write two functions allowing to know the successful grasping rate and to visualize a prediction of our model.

Write a function to test the performance of my network

This function should make it possible to return the success rate of the predictions of a model. As input, it will receive y_pred in format (number of photos, 5) and y_test_copie in format (number of photos, number of possible grasp, 5). Be careful not to count several admissible graspings of the same photo!

On output, it will return the number of good predictions divided by the total number of predictions.

You are going to need **performance()**

```
def test_performance(y_pred, y_test):
  ###############
  #  TO DO      #
  ###############

### TestMagat
y_pred = model.predict(x_test)
print('Performance of my trained model : {:.1%}'.format(test_performance(y_pred, y_test_cc
```

Write a function to view a prediction of my network

model is a deep learning model and x is an image on which we have to calculate the prediction

Make this function display an image with the predicted rectangle materialized on it.

Write yourself a script to test this function

```
def show_prediction(model, x):
  ###############
  #  TO DO      #
  ###############

#### Test
###############
#  TO DO      #
###############
```

So here is a first network which should be able to easily reach 50% success.

To improve the accuracy of the network, we will act on two levers:

- Write a custom loss function to take into account the initial y
- Increase the amount of data available using data augmentation

Define your own Loss Function

The y format is $g_{true} = [(x_1, y_1, \theta_1, w_1, h_1), (x_2, y_2, \theta_2, w_2, h_2), (x_3, y_3, \theta_3, w_3, h_3)]$. Previously, we decided to compare the output of the network $g_{pred} = (x, y, \theta, w, h)$ to only the first possible grasping rectangle $g = [(x_1, y_1, \theta_1, w_1, h_1)]$ to train the network with a loss function $\mathcal{L}$ of your choice:

$$\mathcal{L}(g, g_{pred})$$

To take into account all the possible grasping rectangles, and hopefully improve training, we will use another loss function:

$$\mathcal{L}_{new}(g_{true}, g_{pred}) = \min_{g \in g_{true}} \mathcal{L}(g, g_{pred})$$

This function is not canonical and we will have to write it by ourselves. Keras allows this, here is how canonical functions are defined (https://github.com/keras-team/keras/blob/master/keras/losses.py from line 602). We will therefore imitate them.

Several remarks:

- This function will be called internally in keras, it is necessary to use keras functions to do the mathematical operations on the variables y_pred, y_true (they are of type tensor). Instead of np.sum or np.min, we will use K.sum or K.min.
- Keras doesn't really like having Ys with changing shapes (as a reminder: (nb of photos, nb of possible grasp, 5)) we will therefore reformat all the ys so that they all have the same shape: (nb of photos, 40, 5). It will therefore be necessary to artificially add graspings to most y while ensuring that they do not affect the loss function. In fact, we will add $[1000, 1000, 1000, 1000, 1000]$ grasping parameters that have no chance of being a minimum. We will therefore have a new y of the form: $g_{new,true} = [(x_1, y_1, \theta_1, w_1, h_1), (x_2, y_2, \theta_2, w_2, h_2), (x_3, y_3, \theta_3, w_3, h_3), \dots, (1000,$

Reformat the Y.

Y is of size (750,) and contains different numbers of elements on its axis 1 (this is why no dimension is specified). Add graspings [1000, 1000, 1000, 1000, 1000] so that all images have 40 possible graspings. The function will return a numpy array

```
def preprocess_data_min_mse(Y):
  ###############
  #  TO DO      #
  ###############

#### Test :  must see (750, 224, 224, 3) (750, 40, 5)(750,)
y11 = np.copy(Y)
yy = preprocess_data_min_mse(y11)
print(yy.shape, Y.shape)
```

Now let's write the loss function. Be inspired by canonical functions. I am giving you the first line of the cost function which changes the format of y_pred (to y_pred_temp) to be able to apply our cost function. Its shape is:
$g_{pred,temp} = [(x, y, \theta, w, h), (x, y, \theta, w, h), (x, y, \theta, w, h), \ldots, (x, y, \theta, w, h)]$

- y_pred : (batch_size, 5)
- y_true : (batch_size, nb_of_possible_grasping_area, 5)
- y_pred_temp : (batch_size, nb_of_grasping_areas, 5) with the same prediction iterated x times

You can use K.square, K.abs, K.mean, K.min according your loss function. As numpy, you can set which axis you can evaluate.

```
# Implement it: only 5 lines of code,
# think carefully about the dimensions of the variables you use

import tensorflow.keras.backend as K

def my_function(y_true, y_pred):
    y_pred_temp = K.repeat(y_pred, K.shape(y_true)[1])
    ###############
    #  TO DO      #
    ###############


# Test : by running the test below, you should get:
# [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
# 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
# 1. 1.]
y_true = tf.zeros((50, 20, 5))
y_pred = tf.ones((50, 5))
A = min_mse(y_true, y_pred)
with tf.Session() as sess:  print(A.eval())
```

Recreate your model using your cost function

```
def model_2():
  ##############################
  # paste your model           #
  ##############################
  model.compile(optimizer='''TO DO''', loss=my_function)
  return model
```

Train it and evaluate the impact of the loss function

```
## DATA preparation
X2, Y2 = np.copy(X), np.copy(Y)
X2, Y2 = preprocess_data_min_mse(X2, Y2)
x_train, x_test, y_train, y_test =

###############
#  TO DO      #
###############


test_performance(model_2, y_pred)
```

Data augmentation

The number of data (750) on which we train is rather low for a neural network. We are therefore going to implement data augmentation to improve all that. To avoid having to store all the new data in hard, we will use a data generator which will randomly generate the data at the time of training before forgetting them. Unfortunately, there is no such function ready for regression problems, we will have to code it ourselves!

We are going to create a DataGenerator class which, by relying on the basic data, will perform geometric transformations on the X and Y to fill batches of images and thus train a model.

This supposes performing transformations on the images. We will use the python reference library for image processing: ** opencv **. First we will create and test the functions, then we will incorporate them into a DataGenerator

Brightness

To begin with, you can change the brightness of the images. This will have an impact when we want to deploy our model on a real system because the lighting is not constant (however on our test base, this may not have an impact). The change in brightness doesn't alter the grasping coordinates, so we just need to change the values of X without touching Y.

Create a function to randomly change the brightness of an image.

You are autonomous to look for methods on the internet. Here are some functions that you might find useful

- cv2.cvtColor(img, cv2.COLOR_BGR2HSV) : convert a RGB image into a HSV image
- np.random.randint(min, max) : generates a random value useful for changing the brightness randomly

(Hint 1: You can convert your RGB image to its HSV representation. In this representation, the V channel corresponds to the brightness. By changing this value and then converting back to RGB you will change the brightness of your image.)

```
import cv2

def change_luminosity(img):
    ###############
    #  a remplir  #
    ###############

## Test
img = X[0]
lum_img = change_luminosity(img)
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.subplot(1, 2, 2)
plt.imshow(lum_img)
```

Flip

Flip consists of simply flipping the image up and down or left to right. In this case you must also change the grasping coordinates to Y. The easiest way is to go back to the pixel coordinates with the **grasp2bbox** function. Perform the geometric transformation and then go back to the grasping representation using **bbox2grasp**.

To make flips easily you can use these functions: `np.fliplr` ot `np.flipud`

Create a flip function which takes as input an image (img) and the grasping parameters (y_list). This function will randomly return the same image, the image flipped left to right or the image flipped top to bottom. Please also change the associated ones.

Write a script to test your function

```
def flip(img, y_list):
  ###############
  #  TO DO      #
  ###############


## Test
###############
```

```
#  TO DO      #
###############
```

The DataGenerator class

We need to create a class which will generate batches of data based on the geometric transformations we have just created. This class will be called dataGenerator and we will give a variable of this class to fit_generator ()

The class must therefore follow a certain format so that Keras understands how to use it. The basic skeleton can be found on this site: https://stanford.edu/~shervine/blog/keras-how-to-generate-data-on-the-fly

The skeleton below has been retouched a bit. To create a variable of this class, you must give X and Y data, the desired batch size and indicate whether you want to mix the data or not.

You need to integrate the two functions as an instance of the class and then use them in the instance *__data_generation* to create batches of data as you go. *__data_generation* should return x and y data in the respective form (batch_size, height, width, 3) and (batch_size, 40, 5).

Here's what you need to do:

- Add your flip and brightness functions to the classroom
- In __data_generation (): fill in to make random transformations to selected X and Y.

```
class DataGenerator(tf.keras.utils.Sequence):
    'Generates data for Keras'
    def __init__(self, x, y, batch_size=32, shuffle=True):
        'Initialization'
        self.batch_size = batch_size
        self.X = x
        self.Y = y
        self.shuffle = shuffle
        self.on_epoch_end()

    def __len__(self):
        'Denotes the number of batches per epoch'
        return int(np.floor(self.X.shape[0] / self.batch_size))

    def __getitem__(self, index):
        'Generate one batch of data'
        # Generate indexes of the batch
        indexes = self.indexes[index*self.batch_size:(index+1)*self.batch_size]
        # Find list of IDs
        list_IDs_temp = [k for k in indexes]
        # Generate data
        X, y = self.__data_generation(list_IDs_temp)
        return X, y

    def on_epoch_end(self):
```

```
        'Updates indexes after each epoch'
        self.indexes = np.arange(self.X.shape[0])
        if self.shuffle == True:
            np.random.shuffle(self.indexes)

    def change_luminosity(self, img):
      ###############
      #  TO DO      #
      ###############

    def flip(self, img, y_list):
      ###############
      #  TO DO      #
      ###############

    def __data_generation(self, list_IDs_temp):
        'Generates data containing batch_size samples' # X : (n_samples, *dim, n_channels)
        # Initialization
        X = np.ones((self.batch_size, 224, 224, 3))
        y = 1000 * np.ones((self.batch_size, 40, 5), dtype=np.float)
        # Generate data
        for i, ID in enumerate(list_IDs_temp):
            x_temp, y_temp = self.X[ID], self.Y[ID]
            ###############
            #  TO DO      #
            ###############
        return X, y


#### Test: should output : (20, 224, 224, 3) (20, 100, 5)
testing_generator = DataGenerator(X, Y, batch_size=20)
x, y = testing_generator[0]
print(x.shape, y.shape)
vizualise(x[0], y[0])
```

Train a new network with the data generator. Then test its performance

```
## Data preparation
X3, Y3 = np.copy(X), np.copy(Y)
# X2, Y2 = preprocess_data_min_mse(X2, Y2)
x_train, x_test, y_train, y_test =

## Creation of the train DataGenerator
training_generator = DataGenerator(x_train, y_train, batch_size=10, shuffle=True)

## Training
model_3 = model_2()
model.fit_generator(training_generator, epochs=100, validation_data=(x_test, y_test))


y_pred = model.predict(x_test)
print(y_pred.shape)
print('Performance of my trained model : {:.1%}'.format(test_performance(y_pred, y_test)))
```