Get started Open in app

François Ponchon

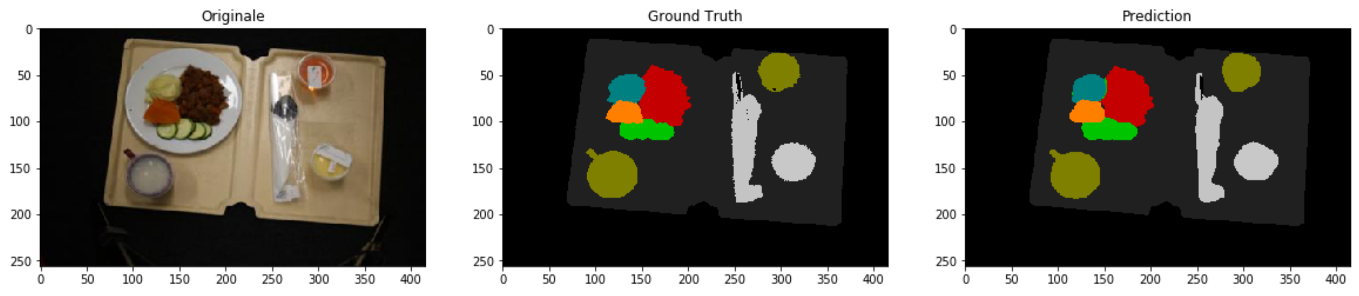
Follow 5 Followers About

Pytorch, Google Colaboratory and UNet

François Ponchon Mar 31, 2020 · 7 min read

In this story, i'll try to explain you how i trained my first UNet neural network on a TrayFood Dataset via Google Colab and PyTorch.

https://www.kaggle.com/thezaza102/tray-food-segmentation



What we'll be able to do with this Algorithm

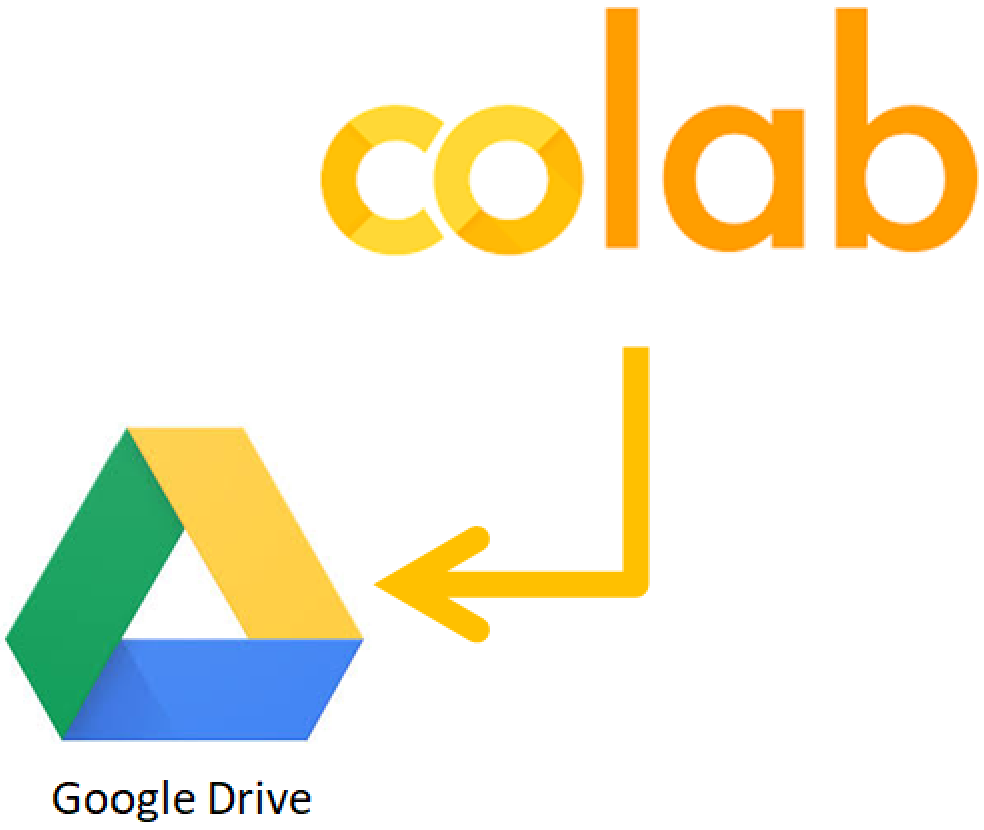
The Beggining…

When i started to work on DeepLearning, i had an ultrabook without any GPU, it quickly became hard to continue…

Asus Ultrabook —What a powerfull workstation !

Google Colab

After several months, i wanted to train from scratch a UNet on my own data. It was impossible for me to train it on my local machine !
Then i discovered Google Colaboratory as a training plateform.



Google Colab and Google Drive interaction

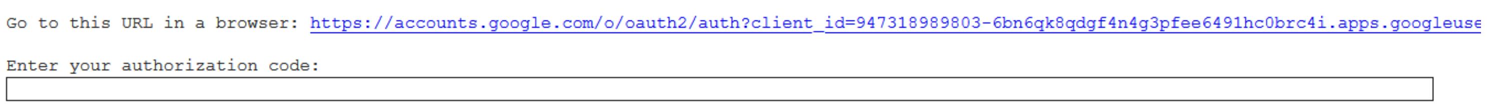
The first very interesting thing with Google Colab, is that we can virtualy connect it to the Google Drive storage.

In this case, i use data from kaggle. I can do another way, but i also have some other datasets with my labelled data, it is possible to sync it from my computer to my drive, in order to compute it in Google Colaboratory.

To connect to my drive, i just need 2 lines of code.

```python
from google.colab import drive
drive.mount('/content/drive/')
```

Connect GoogleDrive to Google Colaboratory



```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleuse

Enter your authorization code:
```

Prompt where to paste your GoogleDrive token

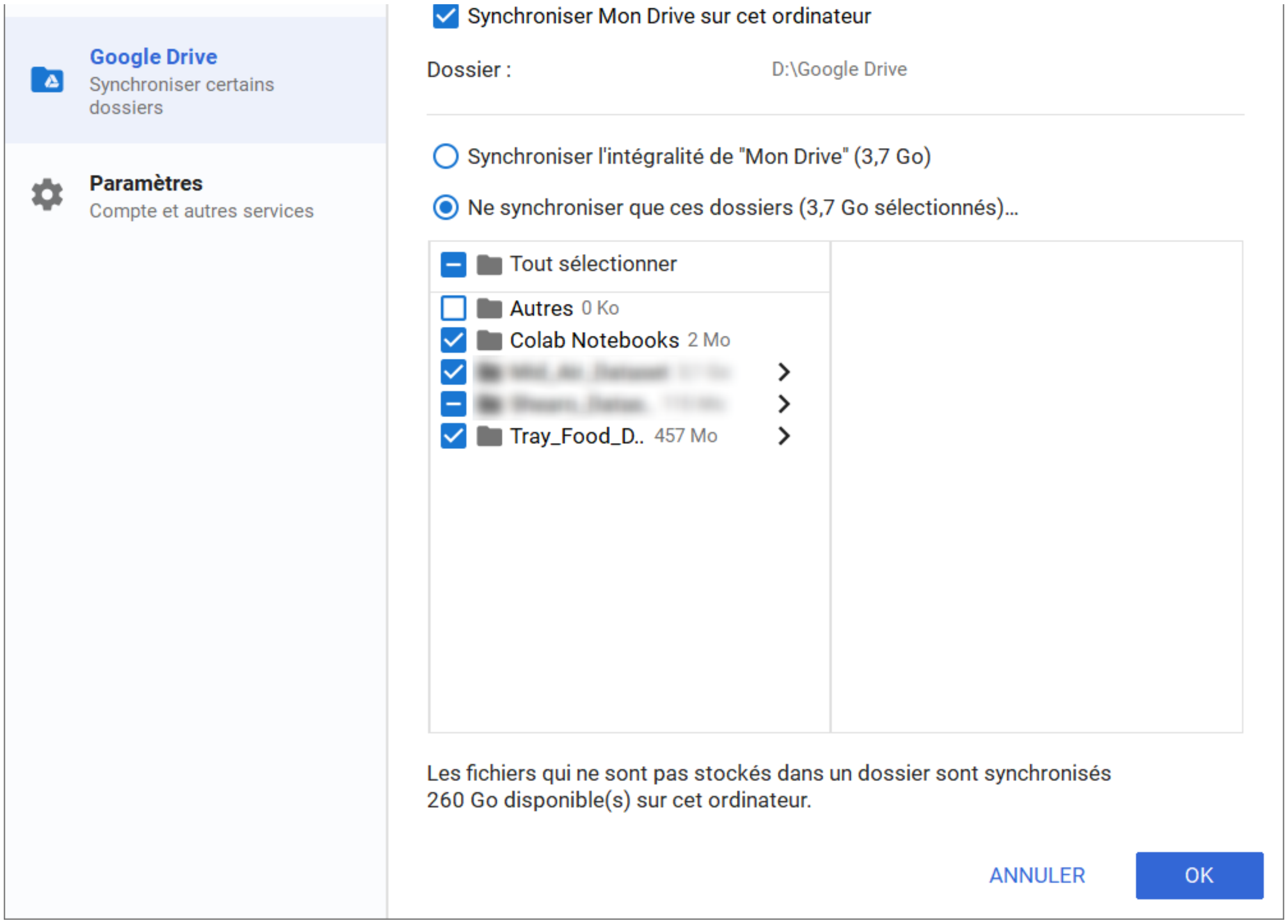
We can access to the storage with python code like on your local machine, simply with the python os library.
In my case, i'll store everything inside the folder "Tray_Food_Dataset".

> **import** os
> PROJECT_PATH="/content/drive/My Drive/Tray_Food_Dataset/"

How to use libraries ?

In order to do the inference locally, it can be interesting to share code through Google Drive, between Google Colaboratory and your local machine.

In order to get it, the first things to do is to use GoogleDrive Desktop application to sync a local folder to Google Drive.



Google Drive
Synchroniser certains dossiers

Paramètres
Compte et autres services

✓ Synchroniser Mon Drive sur cet ordinateur

Dossier : D:\Google Drive

○ Synchroniser l'intégralité de "Mon Drive" (3,7 Go)
● Ne synchroniser que ces dossiers (3,7 Go sélectionnés)...

−	Tout sélectionner
☐	Autres 0 Ko
✓	Colab Notebooks 2 Mo
✓	[redacted]
−	[redacted]
✓	Tray_Food_D.. 457 Mo

Les fichiers qui ne sont pas stockés dans un dossier sont synchronisés
260 Go disponible(s) sur cet ordinateur.

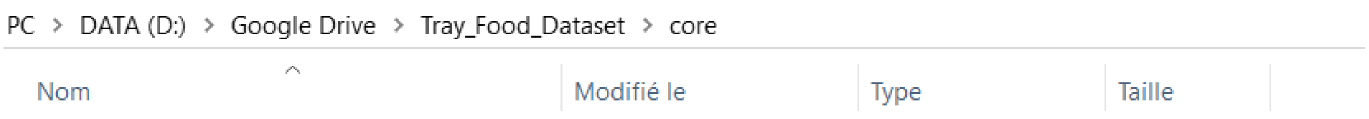
Screenshot on GoogleDrive Desktop Application — WIndows 10

In order to use libraries on Google Colaboratory, we can use **importlib.machinery**

```python
import os
PROJECT_PATH="/content/drive/My Drive/Tray_Food_Dataset/"
from importlib.machinery import SourceFileLoader
SourceFileLoader('Dataset', os.path.join(PROJECT_PATH, 'core/Dataset.py')).load_module()
SourceFileLoader('Model', os.path.join(PROJECT_PATH, 'core/Model.py')).load_module()
SourceFileLoader('Utils', os.path.join(PROJECT_PATH, 'core/Utils.py')).load_module()

from Model import UNet
from Dataset import TrayFoodDataset
from Utils import Prod
```

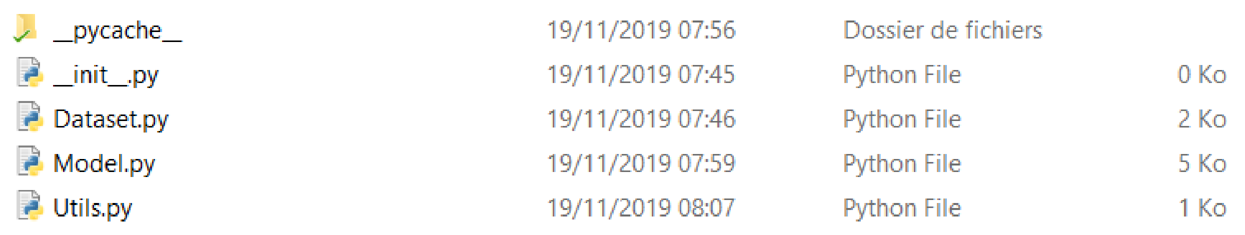
How to import libraries on Google Colaboratory

This way, we can code on local machine and re-use classes on Google Colaboratory Notebook.



PC > DATA (D:) > Google Drive > Tray_Food_Dataset > core

Nom	Modifié le	Type	Taille



__pycache__	19/11/2019 07:56	Dossier de fichiers	
__init__.py	19/11/2019 07:45	Python File	0 Ko
Dataset.py	19/11/2019 07:46	Python File	2 Ko
Model.py	19/11/2019 07:59	Python File	5 Ko
Utils.py	19/11/2019 08:07	Python File	1 Ko

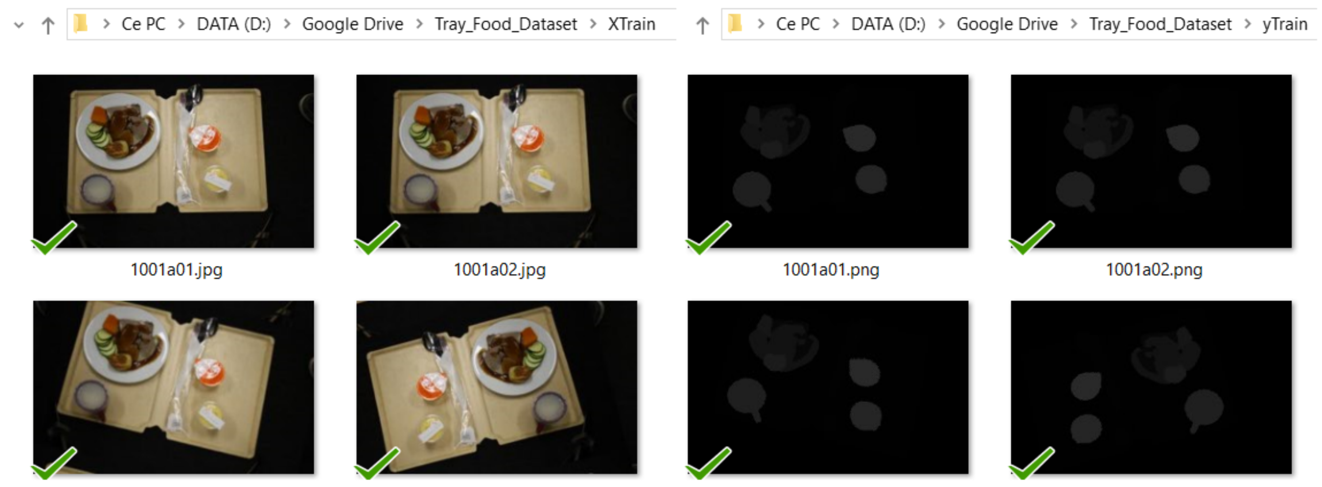
Local Directory containing Source Code

The Dataset

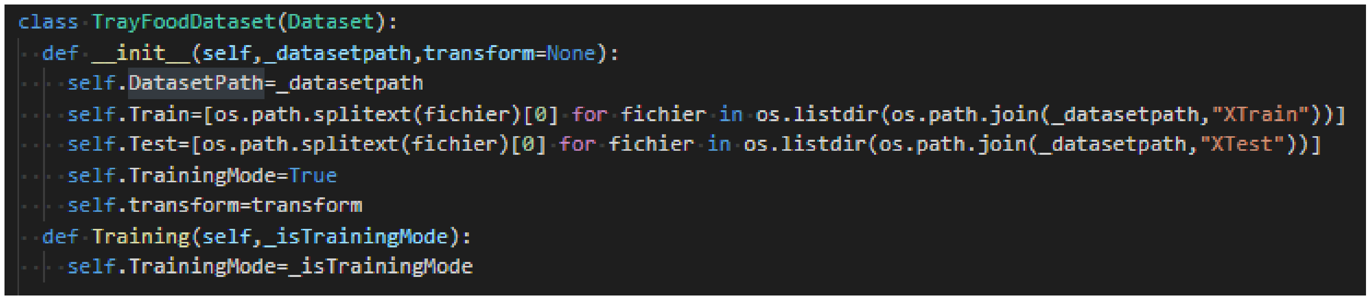
Now let's talk about the code itself.

The first thing we should code is the dataset. The way it works in Pytorch is quite simple. We just inherit from **torch.utils.data Dataset** class.

In our case, we have 4 folders :
- XTrain => Containing ".jpg" Training 1241 original images
- yTrain => Containing ".png" Training class masks
- XTest => Containing ".jpg" Test 6 original images
- yTest => Containing ".png" Test class masks


Ce PC > DATA (D:) > Google Drive > Tray_Food_Dataset > XTrain
Ce PC > DATA (D:) > Google Drive > Tray_Food_Dataset > yTrain
1001a01.jpg
1001a02.jpg
1001a01.png
1001a02.png

Example of images — Training Data and Labels



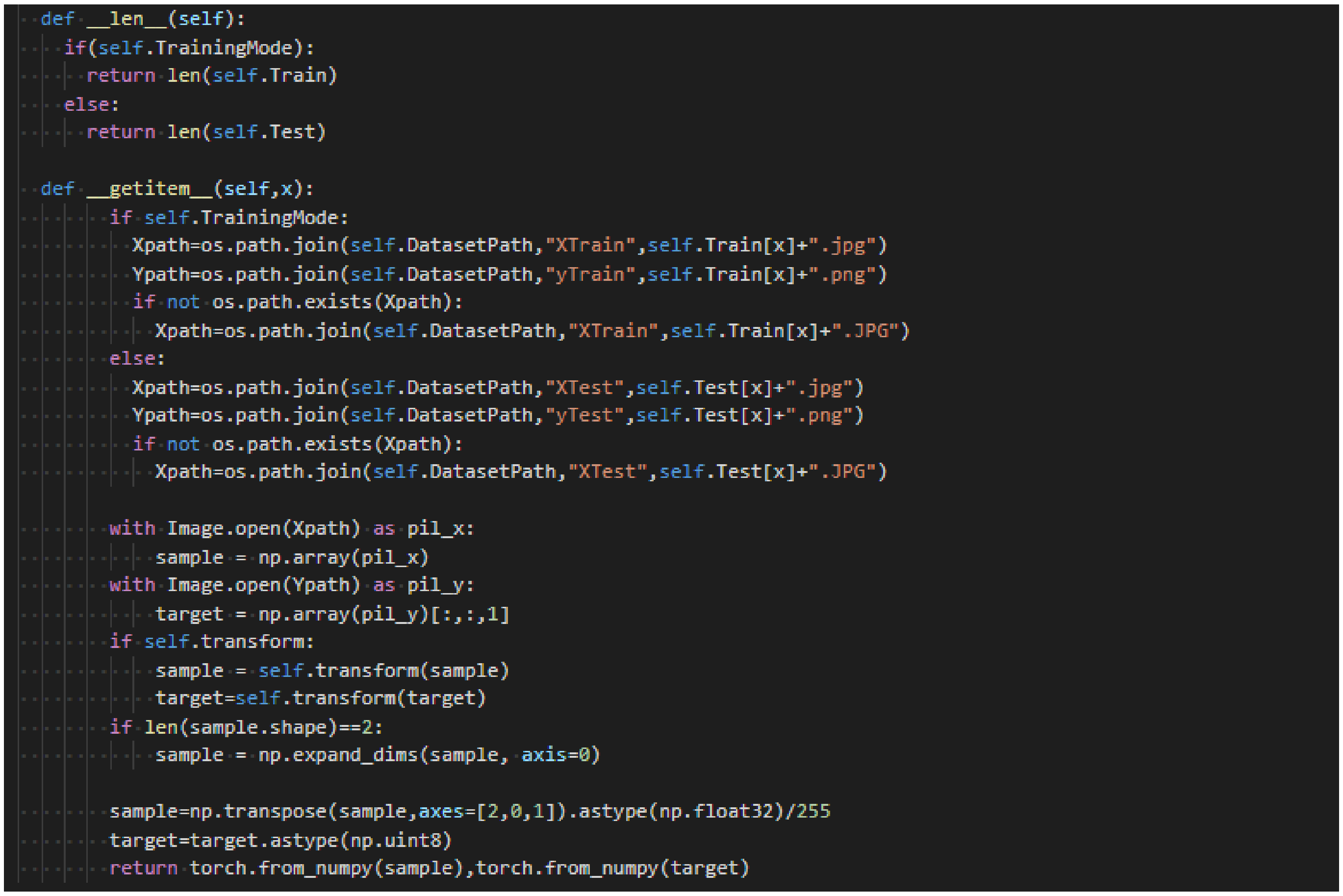
```
class TrayFoodDataset(Dataset):
    def __init__(self,_datasetpath,transform=None):
        self.DatasetPath=_datasetpath
        self.Train=[os.path.splitext(fichier)[0] for fichier in os.listdir(os.path.join(_datasetpath,"XTrain"))]
        self.Test=[os.path.splitext(fichier)[0] for fichier in os.listdir(os.path.join(_datasetpath,"XTest"))]
        self.TrainingMode=True
        self.transform=transform
    def Training(self,_isTrainingMode):
        self.TrainingMode=_isTrainingMode
```

Dataset Constructor — Index Train and Test Data in two lists

In Pytorch Dataset, it is necessary to override two methods, ***__len__(self)*** and ***__getitem__(self,idx)***

The first method is the ability to return the number of elements in the dataset.
The second one is to get the i *th* element from the Dataset.

Note that we'll separate Training and Test dataset. We'll just use Training method to switch the dataset.



```
    def __len__(self):
        if(self.TrainingMode):
            return len(self.Train)
        else:
            return len(self.Test)

    def __getitem__(self,x):
            if self.TrainingMode:
              Xpath=os.path.join(self.DatasetPath,"XTrain",self.Train[x]+".jpg")
              Ypath=os.path.join(self.DatasetPath,"yTrain",self.Train[x]+".png")
              if not os.path.exists(Xpath):
                Xpath=os.path.join(self.DatasetPath,"XTrain",self.Train[x]+".JPG")
            else:
              Xpath=os.path.join(self.DatasetPath,"XTest",self.Test[x]+".jpg")
              Ypath=os.path.join(self.DatasetPath,"yTest",self.Test[x]+".png")
              if not os.path.exists(Xpath):
                Xpath=os.path.join(self.DatasetPath,"XTest",self.Test[x]+".JPG")

            with Image.open(Xpath) as pil_x:
                sample = np.array(pil_x)
            with Image.open(Ypath) as pil_y:
                target = np.array(pil_y)[:,:,1]
            if self.transform:
                sample = self.transform(sample)
                target=self.transform(target)
            if len(sample.shape)==2:
                sample = np.expand_dims(sample, axis=0)

            sample=np.transpose(sample,axes=[2,0,1]).astype(np.float32)/255
            target=target.astype(np.uint8)
            return torch.from_numpy(sample),torch.from_numpy(target)
```

Methods overrided in order to complete the Dataset

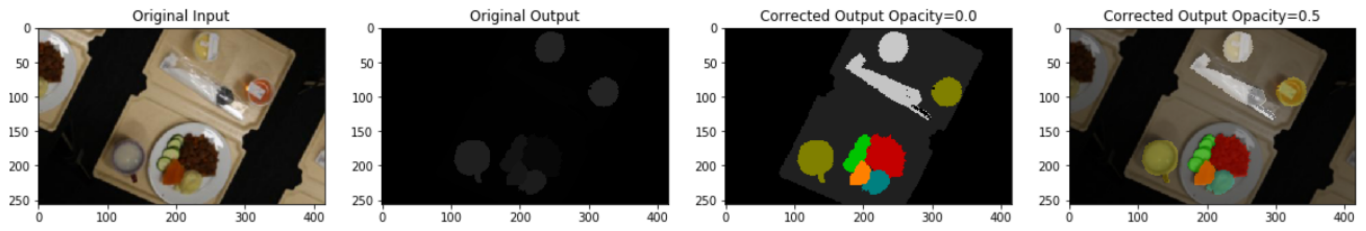
In order to check that everything work, we can create a Dataset instance, get an element and display it.
The problem is that we have already transformed image to PyTorch instance in the **__getitem__** function, then we'll have to build a displaying function in order to display it with matplotlib. We'll also use this function in order to colorize classes, for that, i stored conversion table in a csv file.

```python
import pandas as pd
import numpy as np

data = pd.read_csv("/content/drive/My Drive/Tray_Food_Dataset/classes.csv")
global colordict
colordict={}
for i,row in data.iterrows():
  colordict[row["_id"]]=np.array(row.loc["_rcolor":"_bcolor"].tolist())

def ColorTarget(Y):
  h,w=Y.shape
  print(Y.shape)
  result=np.zeros([h,w,3])
  I,J=np.nonzero(Y)
  for index in range(len(I)):
    result[I[index],J[index],:]=colordict[Y[I[index],J[index]]]
  return result

def DisplayX(X):
  Xn=X.data.numpy()
  Xt=np.transpose(Xn,axes=[1,2,0])*255
  return Xt.astype(np.uint8)

def DisplayY(X,Y,alpha=0.1):
  # Display from X and Y a colorized map with classe
  Xt=DisplayX(X)
  Yt=Y.data.numpy()
  Mask=ColorTarget(Yt)
  result=alpha*Xt+(1-alpha)*Mask
  return result.astype(np.uint8)
```

Displaying functions

```python
from matplotlib import pyplot as plt
plt.rcParams['figure.figsize'] = [20, 20]

datasetpath="/content/drive/My Drive/Tray_Food_Dataset"
Ds=TrayFoodDataset(datasetpath)
X,Y=Ds.__getitem__(25)
plt.subplot(1,4,1)
plt.imshow(DisplayX(X))
plt.title("Original Input")
plt.subplot(1,4,2)
plt.imshow(Y,cmap="gray",vmax=255)
plt.title("Original Output")
plt.subplot(1,4,3)
plt.imshow(DisplayY(X,Y,0.0))
plt.title("Corrected Output Opacity=0.0")
plt.subplot(1,4,4)
plt.imshow(DisplayY(X,Y,0.5))
plt.title("Corrected Output Opacity=0.5")
```

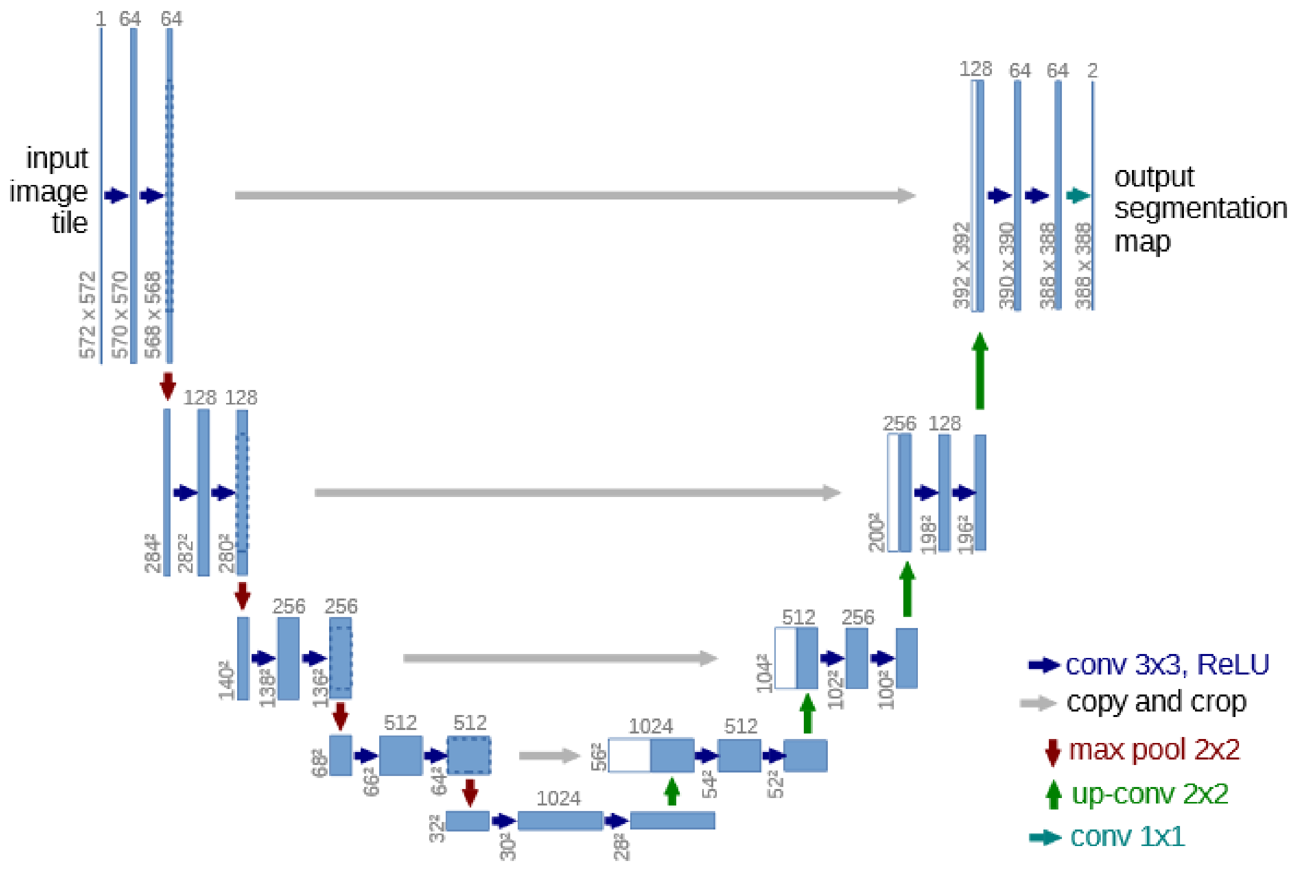
Create a Dataset Instance and Display Images



25th Element from the Dataset

The Network : UNet

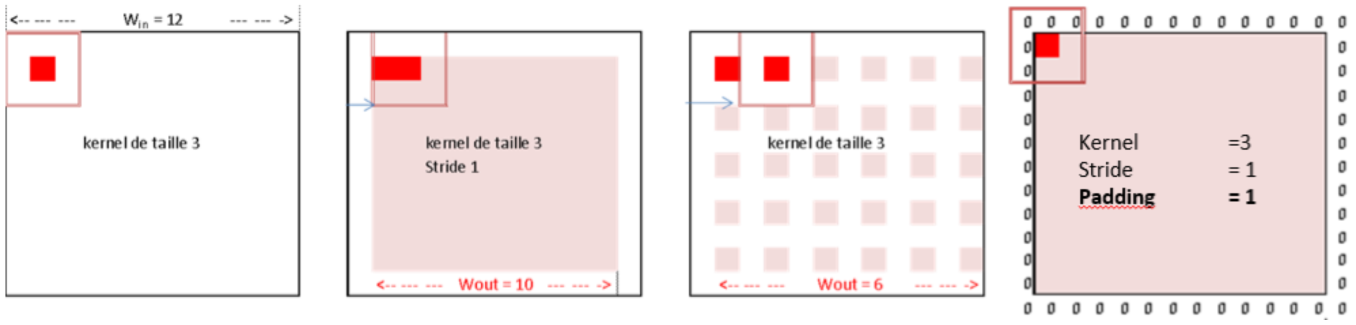
UNet paper (2015) can be found there : https://arxiv.org/pdf/1505.04597.pdf



Unet Original structure

In this paper, they propose a network which uses a 512x512px image as input. The name of the network comes from the U shape structure of convolutions / deconvolutions.

An interesting thing is that the output has a 388x388px shape. For our implementation, it is a problem, because we want to do the segmentation on the whole image.In order to avoid it, we will play with padding for getting an output size identical to the input shape.



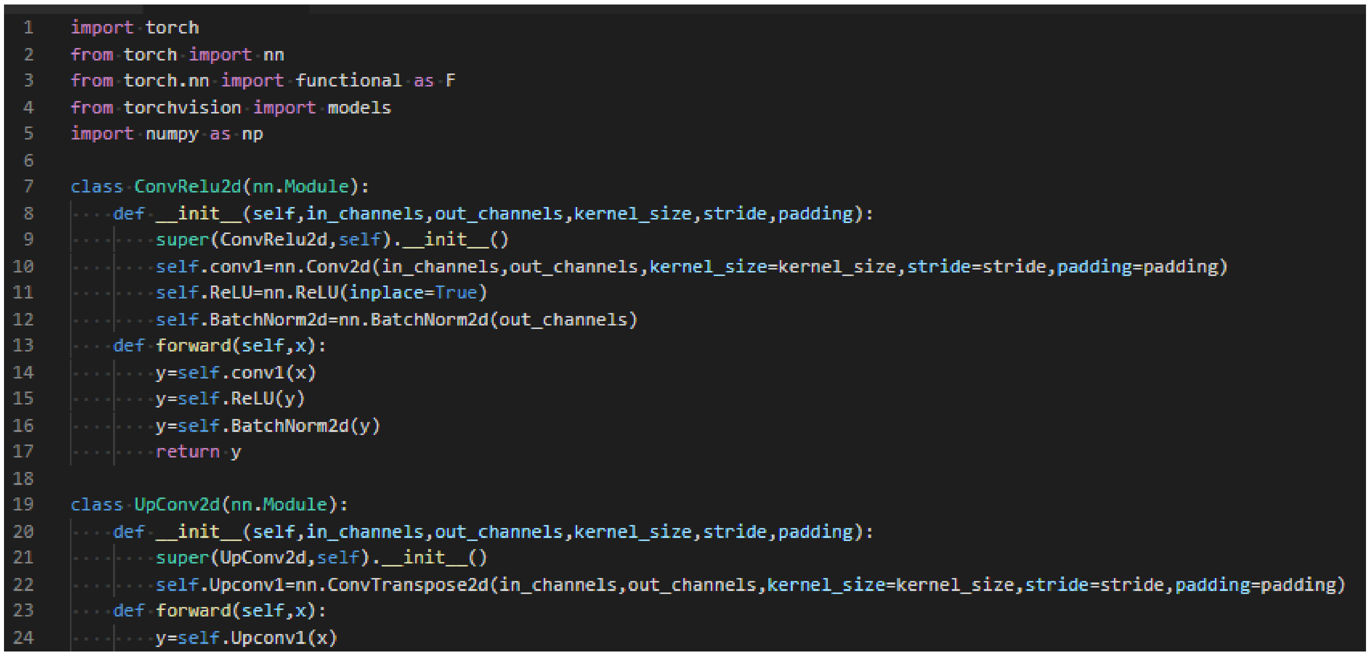
<-- --- --- W_in = 12 --- --- ->
kernel de taille 3
kernel de taille 3
Stride 1
<-- --- --- Wout = 10 --- --- ->
kernel de taille 3
<-- --- --- Wout = 6 --- --- ->
Kernel =3
Stride = 1
Padding = 1

The way to use padding in order to manage the output size

We see on the original network structure, that the shape is reduced from the first to the second layer. Without padding, the shape is reduced little by little because the convolution loses one pixel on each side. In order to avoid it, we'll use a padding value of 1.
It will allow us to get an output image with the same size of the input.

To build our network, we'll create 2 standard blocks : One for convolution and one for deconvolution.

In Pytorch, each block inherits from ***nn.Module***. it must implement ***__init__*** (constructor) and ***forward*** methods.



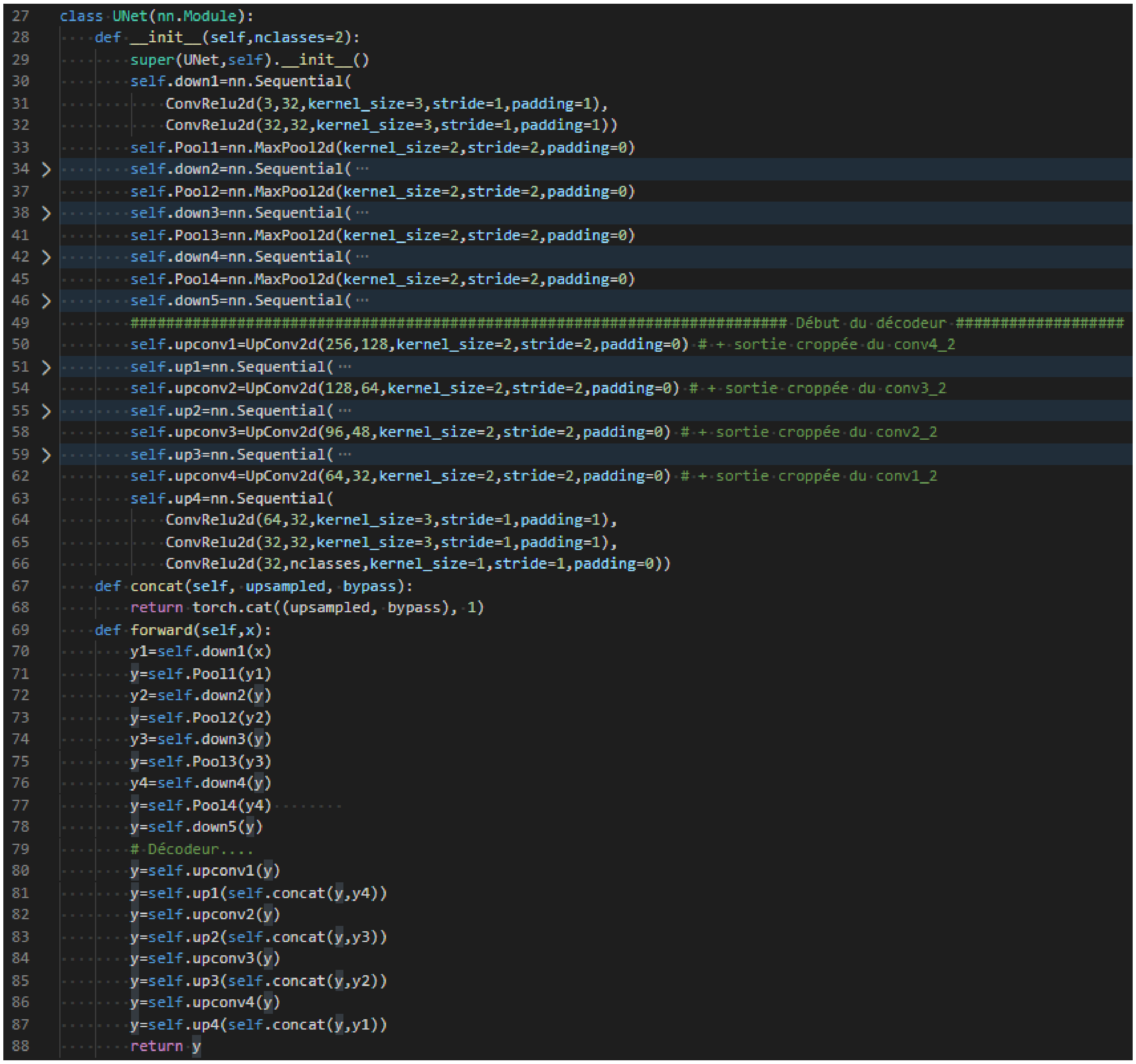
```
import torch
from torch import nn
from torch.nn import functional as F
from torchvision import models
import numpy as np

class ConvRelu2d(nn.Module):
    def __init__(self,in_channels,out_channels,kernel_size,stride,padding):
        super(ConvRelu2d,self).__init__()
        self.conv1=nn.Conv2d(in_channels,out_channels,kernel_size=kernel_size,stride=stride,padding=padding)
        self.ReLU=nn.ReLU(inplace=True)
        self.BatchNorm2d=nn.BatchNorm2d(out_channels)
    def forward(self,x):
        y=self.conv1(x)
        y=self.ReLU(y)
        y=self.BatchNorm2d(y)
        return y

class UpConv2d(nn.Module):
    def __init__(self,in_channels,out_channels,kernel_size,stride,padding):
        super(UpConv2d,self).__init__()
        self.Upconv1=nn.ConvTranspose2d(in_channels,out_channels,kernel_size=kernel_size,stride=stride,padding=padding)
    def forward(self,x):
        y=self.Upconv1(x)
```

```
        return y
```

Standards blocks we'll reuse inside the network declaration

Theses blocks can be assembled in the future network declaration. We'll first declare them in the network constructor.

The same way, we'll declare our network, and reuse our blocks :



```
class UNet(nn.Module):
    def __init__(self,nclasses=2):
        super(UNet,self).__init__()
        self.down1=nn.Sequential(
            ConvRelu2d(3,32,kernel_size=3,stride=1,padding=1),
            ConvRelu2d(32,32,kernel_size=3,stride=1,padding=1))
        self.Pool1=nn.MaxPool2d(kernel_size=2,stride=2,padding=0)
        self.down2=nn.Sequential(…
        self.Pool2=nn.MaxPool2d(kernel_size=2,stride=2,padding=0)
        self.down3=nn.Sequential(…
        self.Pool3=nn.MaxPool2d(kernel_size=2,stride=2,padding=0)
        self.down4=nn.Sequential(…
        self.Pool4=nn.MaxPool2d(kernel_size=2,stride=2,padding=0)
        self.down5=nn.Sequential(…
        ######################################################################### Début du décodeur ###################
        self.upconv1=UpConv2d(256,128,kernel_size=2,stride=2,padding=0) # + sortie croppée du conv4_2
        self.up1=nn.Sequential(…
        self.upconv2=UpConv2d(128,64,kernel_size=2,stride=2,padding=0) # + sortie croppée du conv3_2
        self.up2=nn.Sequential(…
        self.upconv3=UpConv2d(96,48,kernel_size=2,stride=2,padding=0) # + sortie croppée du conv2_2
        self.up3=nn.Sequential(…
        self.upconv4=UpConv2d(64,32,kernel_size=2,stride=2,padding=0) # + sortie croppée du conv1_2
        self.up4=nn.Sequential(
            ConvRelu2d(64,32,kernel_size=3,stride=1,padding=1),
            ConvRelu2d(32,32,kernel_size=3,stride=1,padding=1),
            ConvRelu2d(32,nclasses,kernel_size=1,stride=1,padding=0))
    def concat(self, upsampled, bypass):
        return torch.cat((upsampled, bypass), 1)
    def forward(self,x):
        y1=self.down1(x)
        y=self.Pool1(y1)
        y2=self.down2(y)
        y=self.Pool2(y2)
        y3=self.down3(y)
        y=self.Pool3(y3)
        y4=self.down4(y)
        y=self.Pool4(y4)
        y=self.down5(y)
        # Décodeur....
        y=self.upconv1(y)
        y=self.up1(self.concat(y,y4))
        y=self.upconv2(y)
        y=self.up2(self.concat(y,y3))
        y=self.upconv3(y)
        y=self.up3(self.concat(y,y2))
        y=self.upconv4(y)
        y=self.up4(self.concat(y,y1))
        return y
```

UNet Implementation — The depth of output is the class count

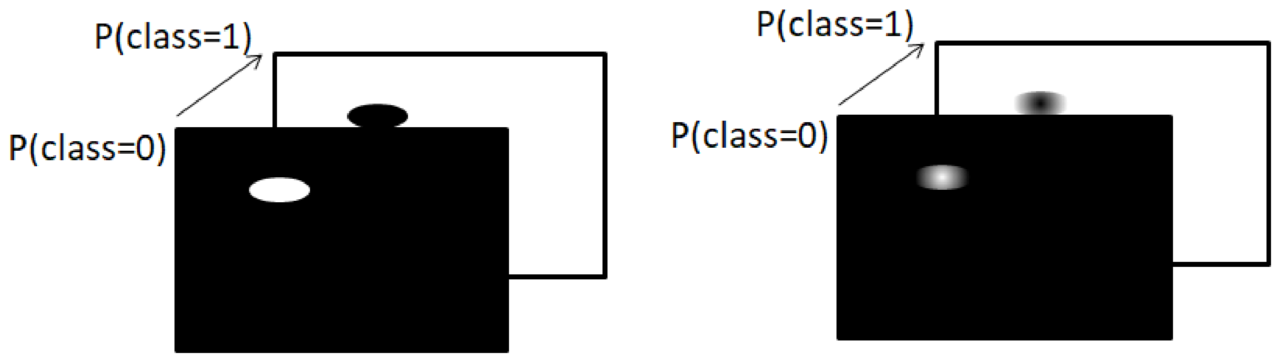
Now our network is declared, we'll build our training loop and loss.

Training Loop

Loss

Firstly we have to define the Loss function we'll use. In our case, we'll have 43 different classes.

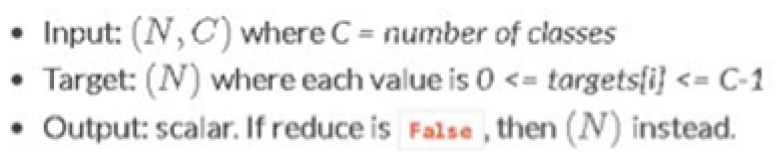
We'll use CrossEntropyLoss because our problem is to estimate for each pixel what is the distribution between classes.
This loss allows us to compare the distribution we've obtained through the network to the expected distribution.
Let's imagine we have two classes :



The Loss will compare the reference (left) to prediction (right)

For each pixel, we'll have the two values, P(class=0) and P(class=1). And the expected class

The CrossEntropyLoss is already implemented in Pytorch, it works like that :



- Input: (N, C) where C = *number of classes*
- Target: (N) where each value is *0 <= targets[i] <= C-1*
- Output: scalar. If reduce is `False`, then (N) instead.

PyTorch documentation

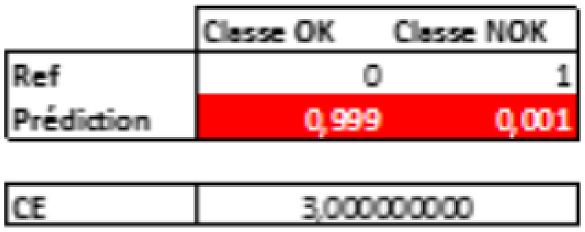
As an input, it expects one matrix and one vector…
The matrix will be the output image from our network (flattened) as N Pixels, and C is the number of class (each pixels have C values).
The Target will be the label (flattened) as N Pixels, each pixels contains a number representing the class index.

$$\boxed{CE_{Image} = -\frac{1}{N}\sum_{k=1:N}\log(Pred(k, c = Ref(k)))}$$

Target: (N) where each value is $0 <= targets[i] <= C-1$

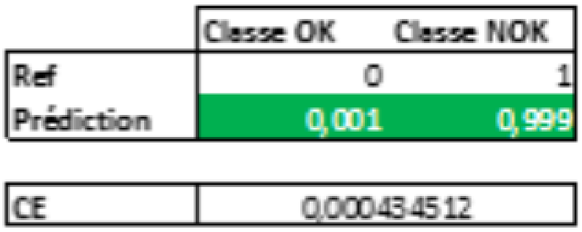
What will compute the CrossEntropyLoss

We can simulate it easily with an Excel Worksheet :



	Classe OK	Classe NOK
Ref	0	1
Prédiction	0,999	0,001

CE	3,000000000



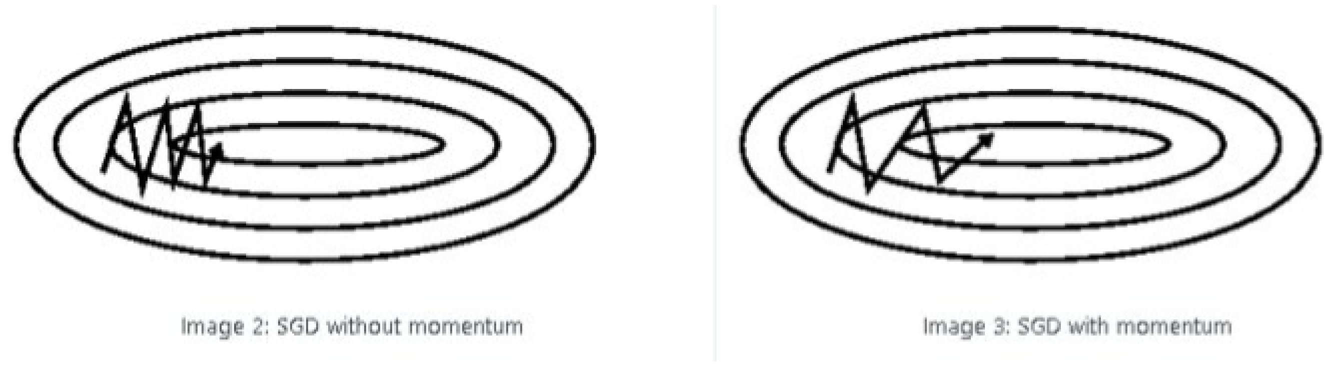
	Classe OK	Classe NOK
Ref	0	1
Prédiction	0,001	0,999

CE	0,000434512

Value increase when we have a distribution far from the expected distribution

Optimizer

The optimizer is the way we'll compute the gradient descent



SGD Gradient Descent

In order to explain it, i advise this very interesting article : https://ruder.io/optimizing-gradient-descent/index.html

Loop

In order to build our training loop, we'll first create instances of our objects :

```
from torch import optim
import torch.nn
import multiprocessing
```

```
num_worker=int(multiprocessing.cpu_count()/2)
if torch.cuda.is_available():
    device = torch.device('cuda')
    torch.cuda.empty_cache()
else:
    device = torch.device('cpu')

model=UNet(nclasses)
if os.path.exists(Model_Path):
  model.load_state_dict(torch.load(Model_Path))

model=model.to(device)
optimizer = torch.optim.SGD(model.parameters(), lr = Lr, momentum=momentum)
criterion=torch.nn.CrossEntropyLoss()
```

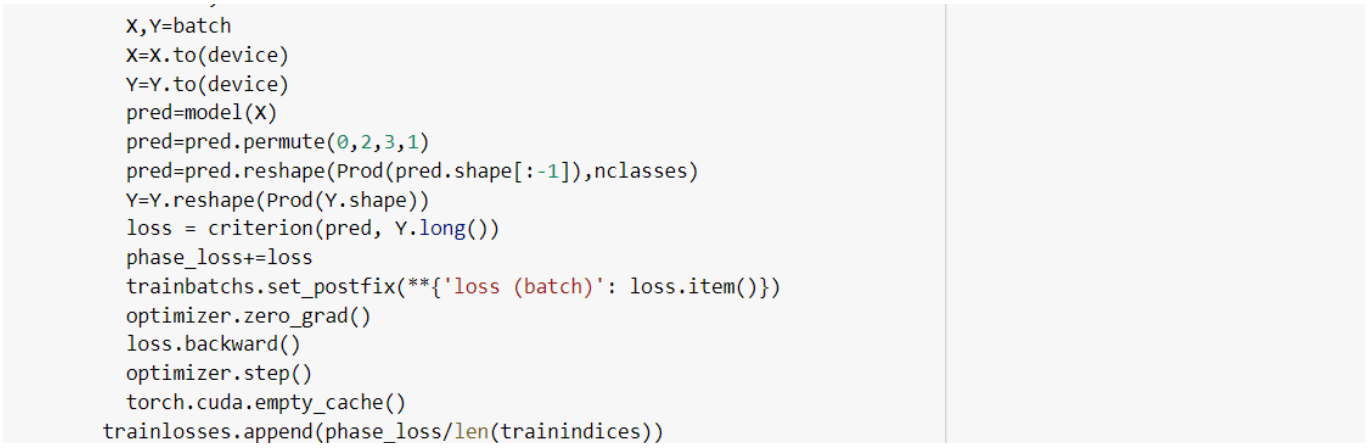
Instances of model, optimizer and criterion (loss)

```
import tqdm as tqdm
import pandas as pd
from torch.utils.data import SubsetRandomSampler,DataLoader

np.random.seed(29)
NbSamples=float(len(Ds))
Ds.Training(True)
indices=np.arange(NbSamples,dtype=np.uint8)
Limit=int((1-ValidationPercentage)*NbSamples)
epochs=[]
trainlosses=[]
vallosses=[]
```

Prepare the training Loop

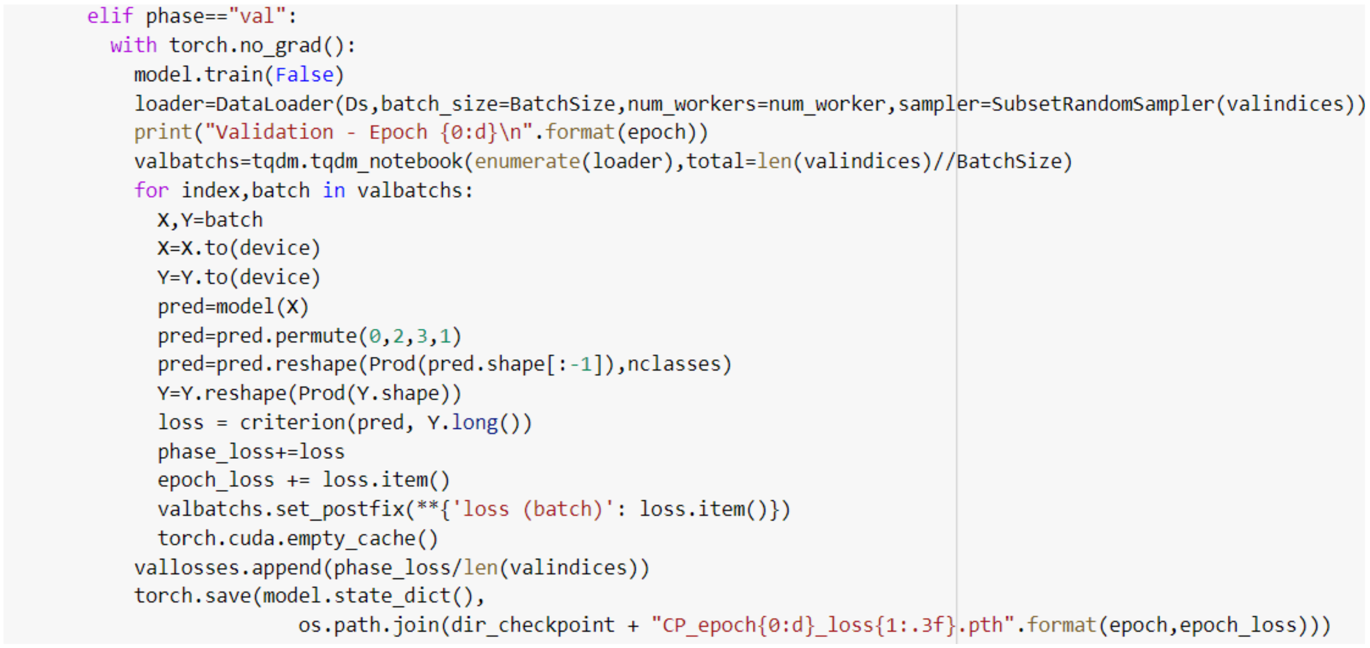
We'll then first implement the training loop :

```
for epoch in range(Epoch):
  epoch_loss=0
  epochs.append(epoch)
  np.random.shuffle(indices)
  trainindices=indices[:Limit]
  valindices=indices[Limit+1:]
  print("Nb Train : {0:d} \nNb Val : {1:d}".format(len(trainindices),len(valindices)))
  for phase in ["train","val"]:
      phase_loss=0.0
      if phase=="train":
        loader=DataLoader(Ds,batch_size=BatchSize,num_workers=num_worker,sampler=SubsetRandomSampler(trainindices))
        model.train(True)
        print("Training - Epoch {0:d}\n".format(epoch))
        trainbatchs=tqdm.tqdm_notebook(enumerate(loader),total=len(trainindices)//BatchSize)
        for index,batch in trainbatchs:
```



```
      X,Y=batch
      X=X.to(device)
      Y=Y.to(device)
      pred=model(X)
      pred=pred.permute(0,2,3,1)
      pred=pred.reshape(Prod(pred.shape[:-1]),nclasses)
      Y=Y.reshape(Prod(Y.shape))
      loss = criterion(pred, Y.long())
      phase_loss+=loss
      trainbatchs.set_postfix(**{'loss (batch)': loss.item()})
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
      torch.cuda.empty_cache()
    trainlosses.append(phase_loss/len(trainindices))
```

Training Loop — Phase “Train”



```
    elif phase=="val":
      with torch.no_grad():
        model.train(False)
        loader=DataLoader(Ds,batch_size=BatchSize,num_workers=num_worker,sampler=SubsetRandomSampler(valindices))
        print("Validation - Epoch {0:d}\n".format(epoch))
        valbatchs=tqdm.tqdm_notebook(enumerate(loader),total=len(valindices)//BatchSize)
        for index,batch in valbatchs:
          X,Y=batch
          X=X.to(device)
          Y=Y.to(device)
          pred=model(X)
          pred=pred.permute(0,2,3,1)
          pred=pred.reshape(Prod(pred.shape[:-1]),nclasses)
          Y=Y.reshape(Prod(Y.shape))
          loss = criterion(pred, Y.long())
          phase_loss+=loss
          epoch_loss += loss.item()
          valbatchs.set_postfix(**{'loss (batch)': loss.item()})
          torch.cuda.empty_cache()
        vallosses.append(phase_loss/len(valindices))
        torch.save(model.state_dict(),
                      os.path.join(dir_checkpoint + "CP_epoch{0:d}_loss{1:.3f}.pth".format(epoch,epoch_loss)))
```

Training Loop — Phase “Validation”

Note that :
- We randomise indices in each epoch. In our case, we don’t have much data then it is important to use everything. (I won’t talk about augmentation in this story).
- We reshape the prediction before we use the Loss function as explained before.
- We can compute indicators we want during the process
- Network weights are saved every epochs

Test

Once it’s done, we’ll reuse the best network we trained and use it on our small test dataset :

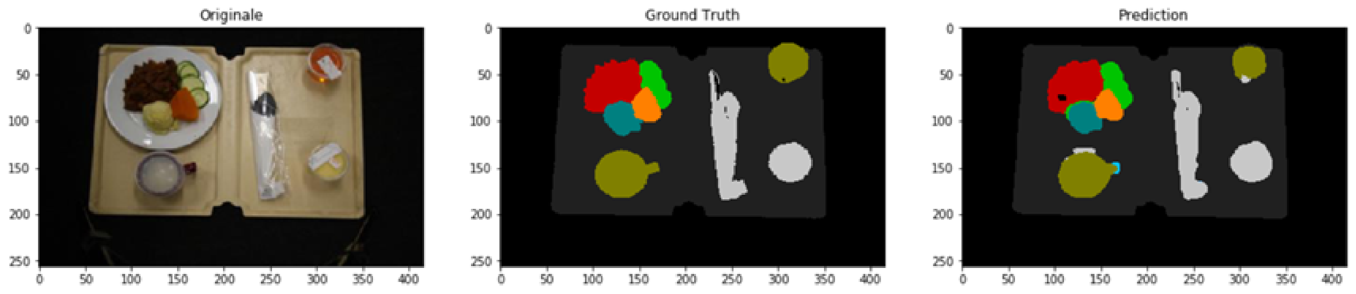
```
from torch import optim
import torch.nn
import multiprocessing

nclasses=43

num_worker=int(multiprocessing.cpu_count()/2)
if torch.cuda.is_available():
    device = torch.device('cuda')
    torch.cuda.empty_cache()
else:
    device = torch.device('cpu')

net=UNet(nclasses)
datasetpath="/content/drive/My Drive/Tray_Food_Dataset"
Model_Path="/content/drive/My Drive/Tray_Food_Dataset/Training4/CP_epoch23_loss0.646.pth"

DsT=TrayFoodDataset(datasetpath)
#On passe le Dataset en mode test
DsT.Training(False)

if os.path.exists(Model_Path):
  net.load_state_dict(torch.load(Model_Path))
```

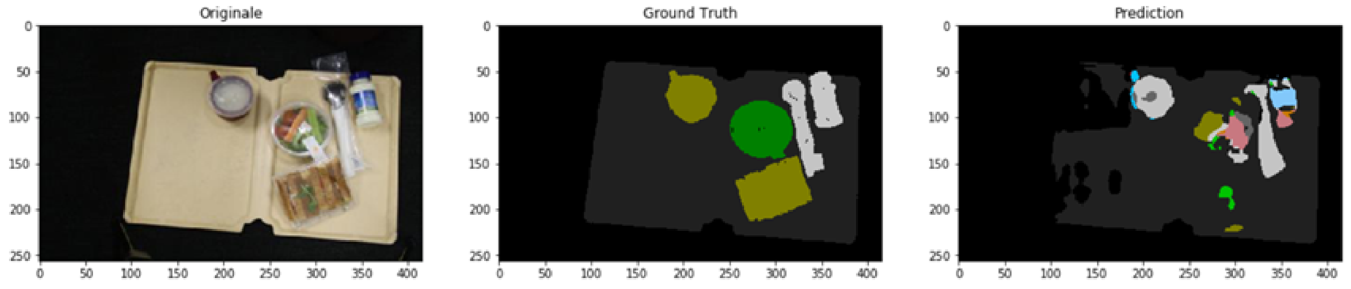
Test Environnement Loading

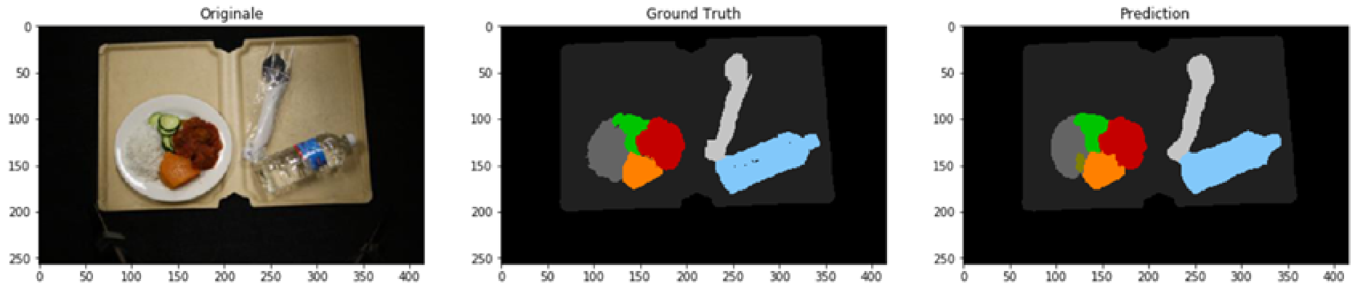
We used the method **DsT.Training(False)** in order to use test images.

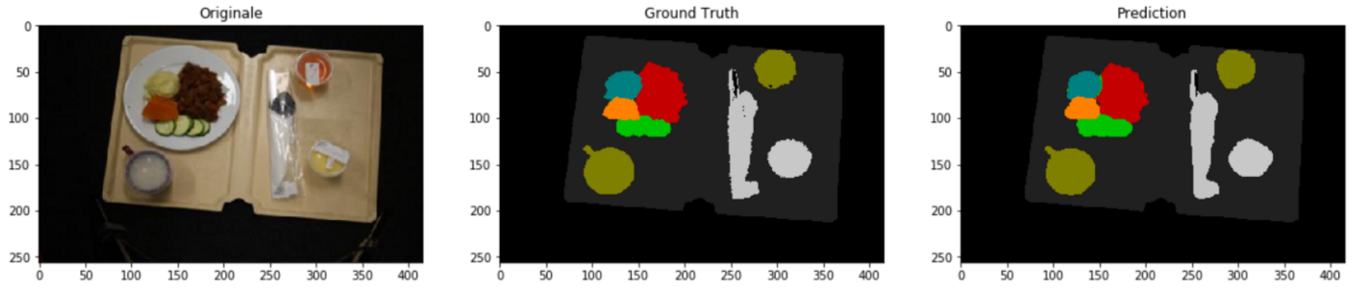
```
X,Y=DsT.__getitem__(4)
Xnet=torch.from_numpy(np.expand_dims(X,axis=0))

Xnet=Xnet.to(device)
net.to(device)
with torch.no_grad():
  net.train(False)
  pred=net(Xnet).squeeze()
  pred=torch.max(pred,0).indices

from matplotlib import pyplot as plt
plt.rcParams['figure.figsize'] = [20, 20]

plt.subplot(1,3,1)
plt.imshow(DisplayX(X))
plt.title("Originale")
plt.subplot(1,3,2)
plt.imshow(DisplayY(X,Y,0))
plt.title("Ground Truth")

plt.subplot(1,3,3)
plt.imshow(DisplayY(X,pred.cpu(),0))
plt.title("Prediction")
```

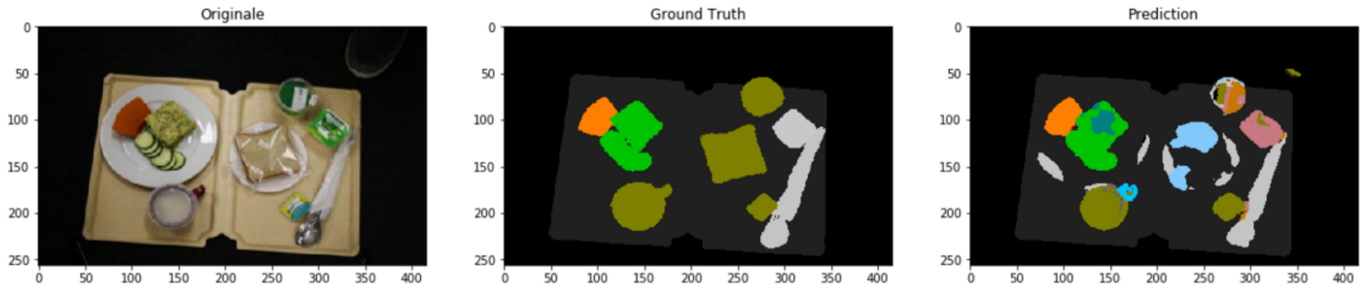
Test on the 4th image of the Test Dataset

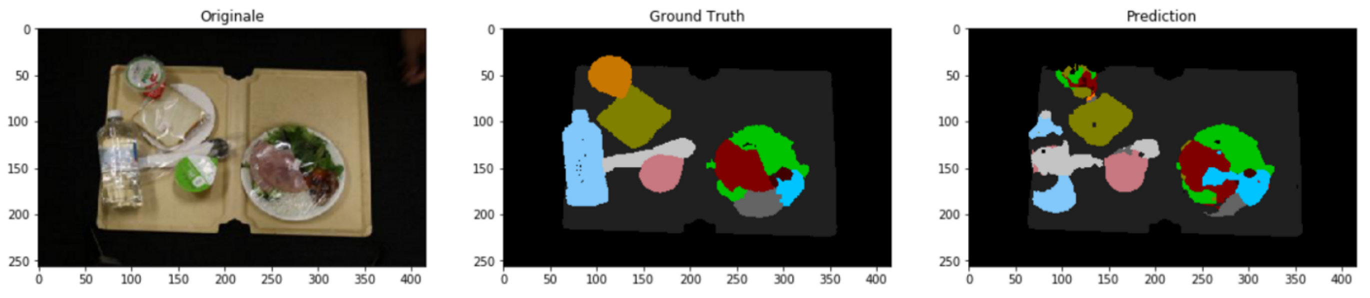


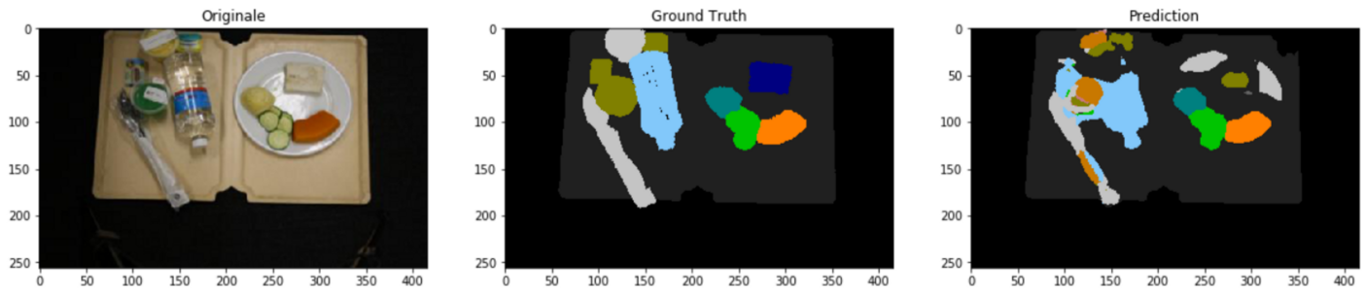












The result on the test dataset is not perfect, but not so bad with few data like that. It would be interesting to try data augmentation with horizontal and vertical flip for example and of course to get a bigger dataset than this one.

I hope this post helped to understand how to work with google colab, Pytorch and UNet network.

Thanks for reading !

Pytorch Unet Colab Deep Learning Step By Step