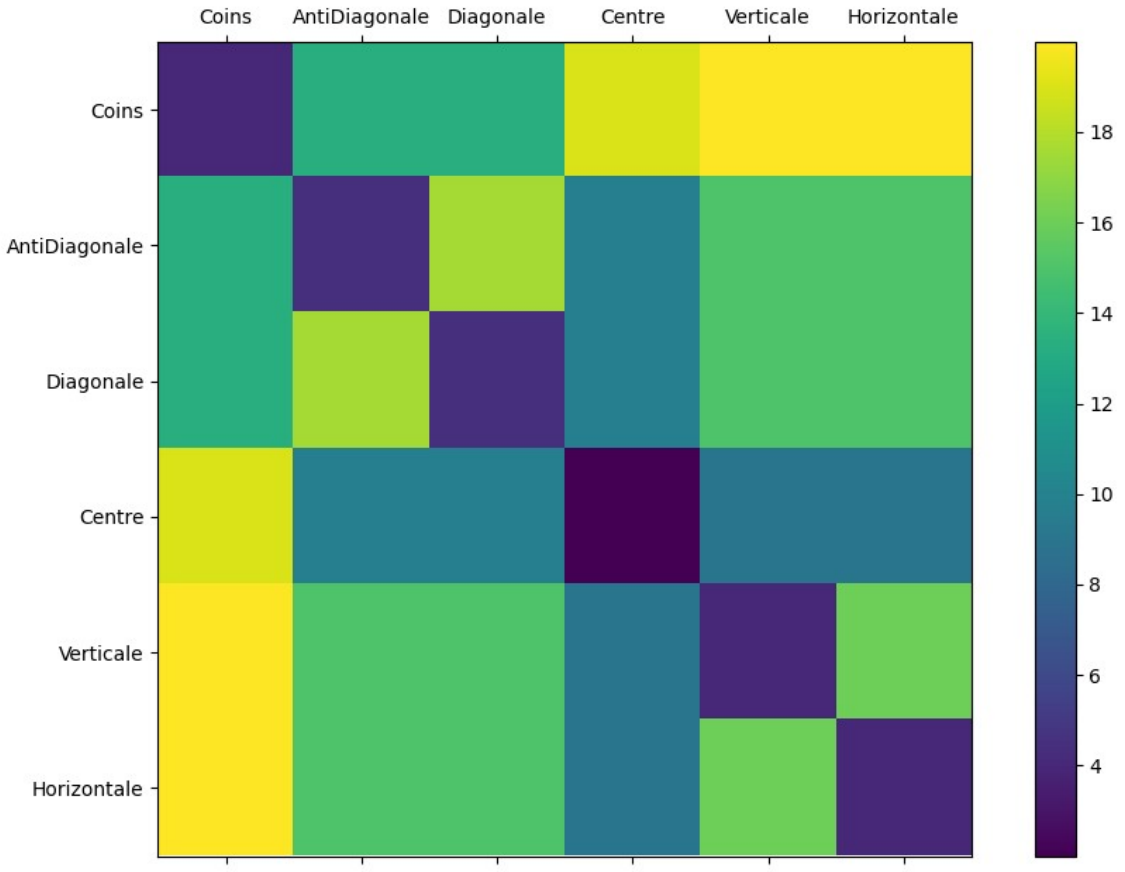
Correction sur la matrice de l’entropie croisée :

Le code précédent ignorait les 0 du calcul de l’entropie, ce qui la rend fausse. Il faut ajouter un epsilon positif proche de 0.

Ce qui nous donne la nouvelle matrice corrigée :

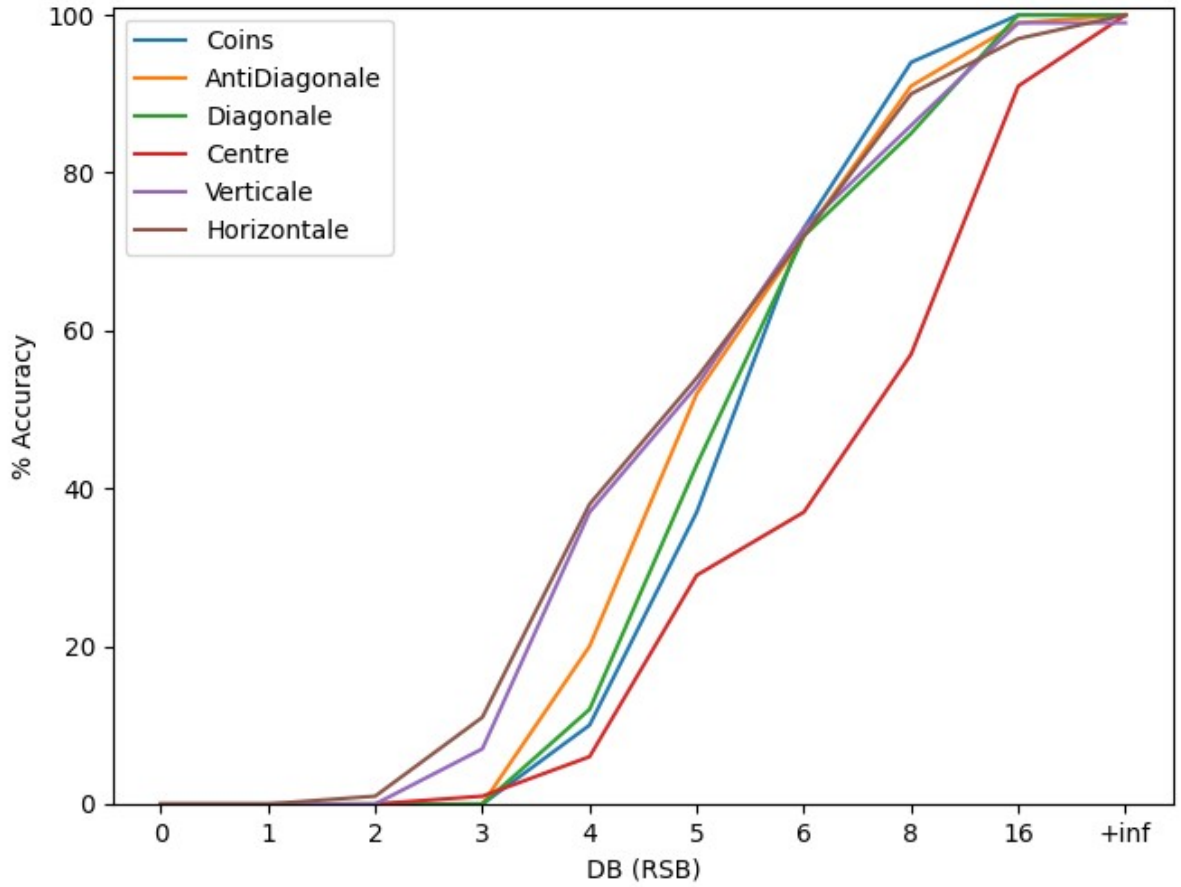


On voit bien que l’entropie croisée d’une classe est minimale lorsqu’ elle est faite avec elle-même.

On essaie ensuite de tracer les courbes de précision de nos 6 classes en fonction du rapport signal/bruit. On devrait voir logiquement que la précision augmentera plus le signal sera fort.

Pour ce faire il nous faut une fonction qui trouve l’ARGMIN des entropies croisées.
On prend une image bruitée, on fait l’entropie croisée de cette image avec les pdf des 6 autres puis on regarde où l’entropie croisée est la plus faible. Il y a de forte chances que cette image appartienne alors à cette classe.

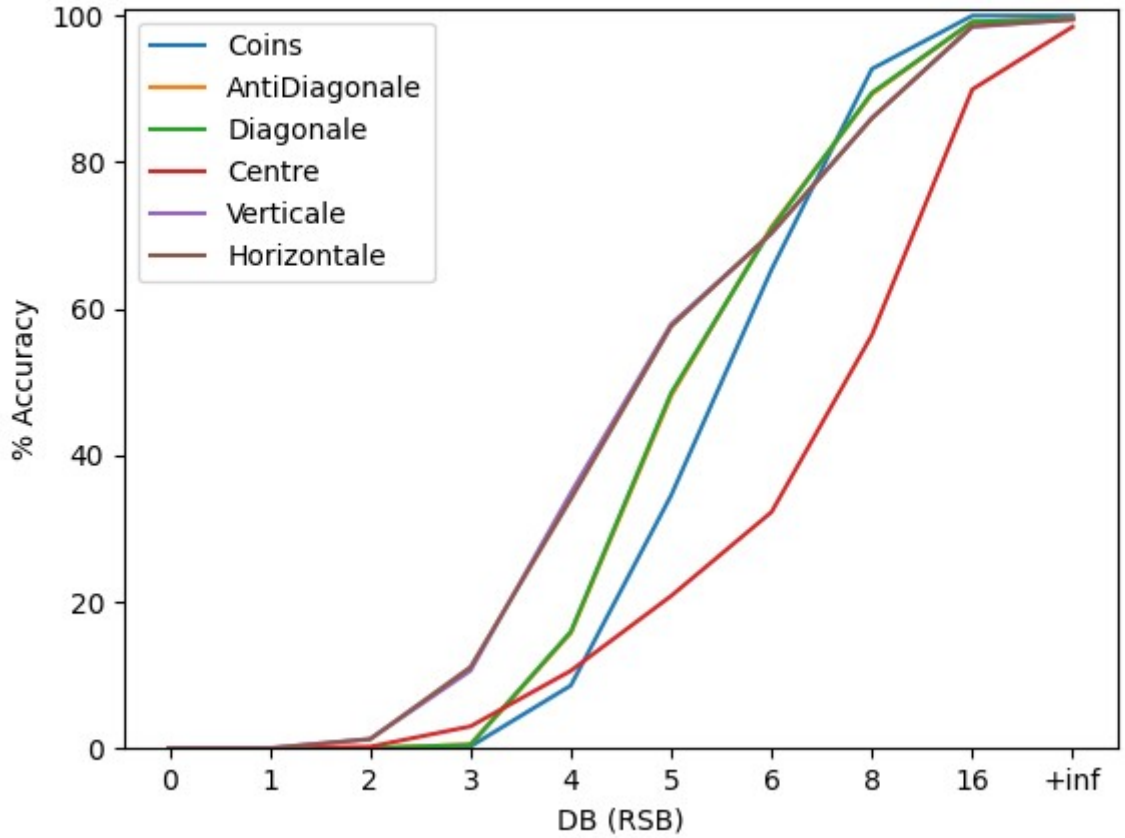
Avec 100 tirages par classe on obtient cette image :



On voit bien que lorsque que le bruit est maximal la précision est nulle, puis elle augmente plus le signal devient fort jusqu'à atteindre les 100% lorsqu'on a que du signal et pas de bruit.

Les courbes ne sont pas parfaites car nous sommes limités en temps et en énergie donc on ne peut pas générer une infinité d'images

On peut améliorer les courbes en tirant plus d'images. Ici, avec 10 000 images par classes :



On obtient des sigmoïdes avec un point d'infléxion vers les 50% d'accuracy.

Cependant on vu en cours que les courbes ne devraient pas se croiser et que la précision des diagonales devrait s'accroître plus rapidement que les autres classes en raison du plus grand nombre de pixels. Mais on voit bien la classe centre qui s'effondre et a besoin de beaucoup de signal pour être reconnue en raison de sa forte information mutuelle avec les autres classes. En effet, toutes les classes, mis à part la classe coins, partagent des pixels avec la classe centre. C'est donc la moins robuste.

À faire pour le 12 avril :

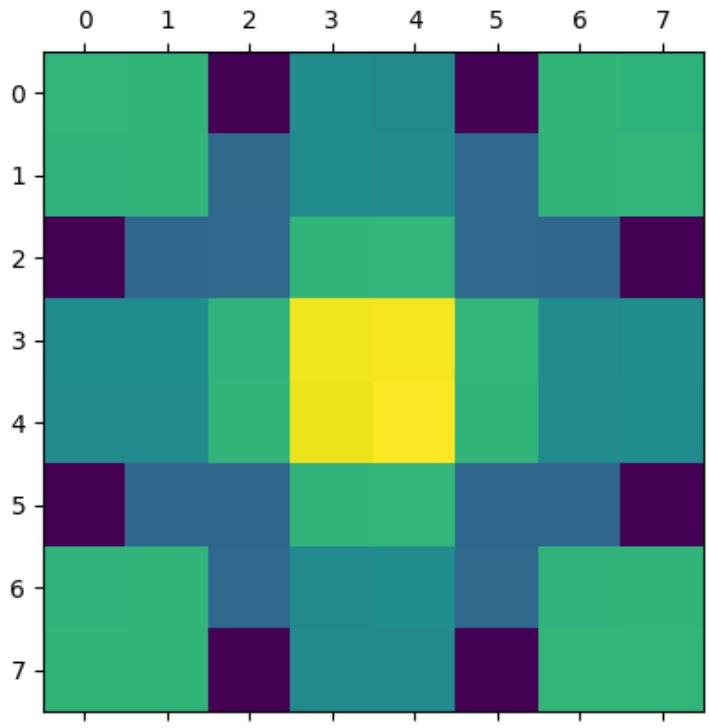
- Faire la matrice des IM pour chaque classe avec les autres.
- Faire l'estimation des J* pixels robustes, minimisant la moyenne.
- 1 Matrice d'IM par pixel, donc 64 matrices, 64 moyennes.
- Prendre les 32 premiers.
- Refaire les courbes de scores avec des bruits autres que blanc (rose, marron).

Compte Rendu Final.

Finalement, nous choisirons d’utiliser la divergence de Kullback-Leibler pour le choix des 32 pixels les plus discriminants et on restera sur du bruit blanc.

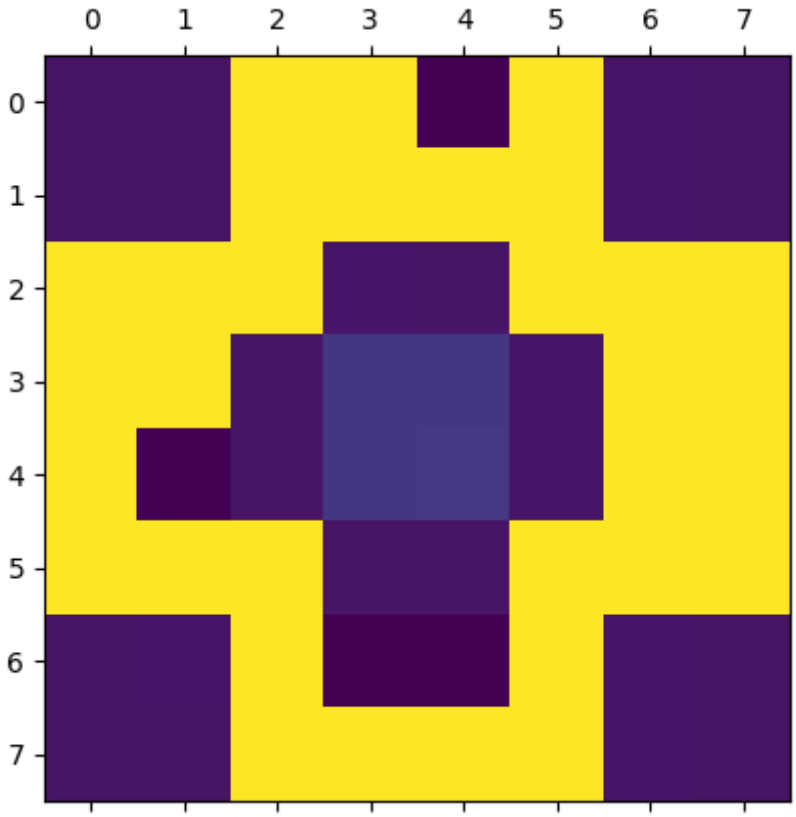
La méthode utilisée est la suivante :

Pour chaque pixel j de 1 à 64, nous calculons la divergence de Kullback-Leibler du pixel j pour tous les couples de classes A et B.
Ce qui nous donne 36 valeurs de divergences par pixel. On en fait ensuite la moyenne.



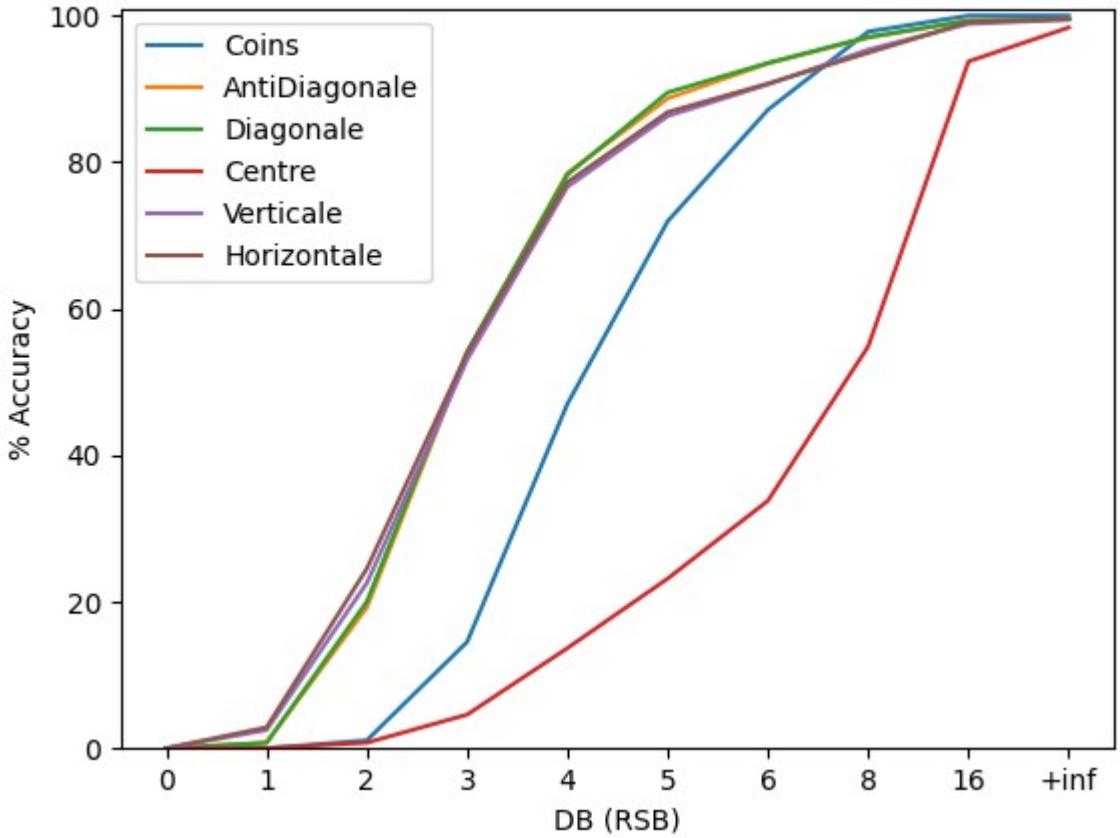
On observe que les pixels discriminants se trouvent aux abords des coins et du centre, tandis que les moins discriminants se situent dans les coins et surtout au centre, ce qui est attendu car le centre partage le plus de pixel avec les autres classes.

Les 32 plus grandes moyennes de divergences sont donc nos 32 pixels les plus discriminants et pertinents.

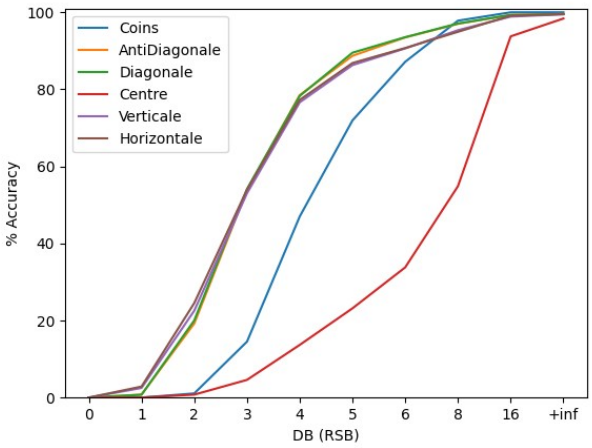


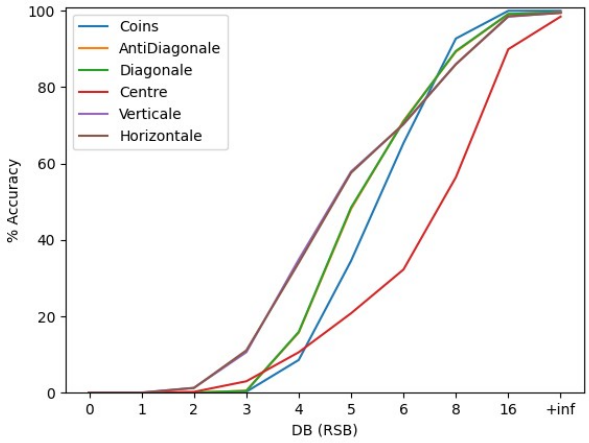
Ici avec seulement les 32 pixels les plus pertinents, même si le centre devrait être en jaune. L’image varie avec différentes exécutions du programme.

Nous relançons nos calculs de courbes en ne prenant en compte que ces pixels.



Ici, avec 10 000 tirages, on remaque bien que les courbes sont similaires mais croissent plus vites, cela s'explique par le fait que les 32 pixels les plus pertinents suffisent pour notre programme. Aussi ce procédé augmente la robustesse des courbes et économise des ressources de calculs, d'énergie et de temps.





Comparaison des deux méthodes de calcul de courbe : à gauche la nouvelle méthode utilisant uniquement les 32 pixels pertinents, à droite la méthode classsique.

Code source utilisé :

Génération des données :

```python
import numpy as np
import math
import pickle
import matplotlib.pyplot as plt
from PIL import Image
SEUIL = 0.35
EPSILON = 0.00001
TIRAGES = 10000
CLASSES = {'Coins' : 0, 'AntiDiagonale' : 1, 'Diagonale' : 2,
           'Centre' : 3, 'Verticale' : 4, 'Horizontale': 5}


def get_rdm_hline(seuil=SEUIL):
    res = np.zeros((8, 8), dtype=int)
    res = res + EPSILON
    for row in range(3,5):
        for col in range(8):
            if np.random.rand() > seuil:
                res[row, col] = 1

    return res

def get_rdm_vline(seuil=SEUIL):

    return get_rdm_hline(seuil).transpose()


def get_rdm_o1line(seuil=SEUIL):
    res = np.zeros((8, 8), dtype=int)
    res = res + EPSILON

    res[0, 0] = np.random.rand() > seuil
    for diag in range(7):
        res[diag + 1, diag] = np.random.rand() > seuil
        res[diag, diag + 1] = np.random.rand() > seuil
        res[diag + 1, diag + 1] = np.random.rand() > seuil
    return res
```

```python
def get_rdm_o2line(seuil=SEUIL):
    res = np.zeros((8, 8), dtype=int)
    res = res + EPSILON

    res[7, 0] = np.random.rand() > seuil
    for diag in range(7):
        res[7 - diag - 1, diag] = np.random.rand() > seuil
        res[7 - diag, diag + 1] = np.random.rand() > seuil
        res[7 - diag - 1, diag + 1] = np.random.rand() > seuil
    return res


```

```python
def get_rdm_centre(seuil=SEUIL):
    res = np.zeros((8, 8), dtype=int)
    res = res + EPSILON
    liste = [[3, 3], [4, 3], [3, 4], [4, 4]]
    for centre in liste:
        res[centre[0], centre[1]] = np.random.rand() > seuil

    return res

```

```python
def get_rdm_corner(seuil=SEUIL):
    res = np.zeros((8, 8), dtype=int)
    res = res + EPSILON
    liste = [[0, 0], [1, 0], [0, 1], [1, 1], [7, 0], [6, 0], [7, 1], [6, 1], [0, 6], [1, 6], [0, 7], [1, 7], [7, 6], [6, 6], [7, 7], [6, 7]]
    for corner in liste:

        res[corner[0], corner[1]] = np.random.rand() >= seuil

    return res
```

```python
def gen_hline():
    filename = "hlines.txt"
    file = open(filename, 'wb')
    data = []
    for i in range(TIRAGES):
        data.append(get_rdm_hline())
    pickle.dump(data, file)

    file.close()

    return data

def gen_vline():
    filename = "vlines.txt"
    file = open(filename, 'wb')
    data = []
    for i in range(TIRAGES):
        data.append(get_rdm_vline())
    pickle.dump(data, file)

    file.close()
    return data

def gen_o1line():
    filename = "o1lines.txt"
    file = open(filename, 'wb')
    data = []
    for i in range(TIRAGES):
        data.append(get_rdm_o1line())
    pickle.dump(data, file)

    file.close()
    return data

```

```
150  def gen_o2line():
151      filename = "o2lines.txt"
152      file = open(filename, 'wb')
153      data = []
154      for i in range(TIRAGES):
155          data.append(get_rdm_o2line())
156      pickle.dump(data, file)
157      file.close()
158      return data
159
160  def gen_centre():
161      filename = "centre.txt"
162      file = open(filename, 'wb')
163      data = []
164      for i in range(TIRAGES):
165          data.append(get_rdm_centre())
166      pickle.dump(data, file)
167      file.close()
168      return data
169
170  def gen_corner():
171      filename = "corner.txt"
172      file = open(filename, 'wb')
173      data = []
174      for i in range(TIRAGES):
175          data.append(get_rdm_corner())
176      pickle.dump(data, file)
177      file.close()
178      return data
179
180
181  def lire(filename):
182      file = open(filename, 'rb')
183      test = pickle.load(file)
184      for x in test :
185          print(x)
186
```

Formules d’entropies :

```python
def entropie(PX):
    res = 0
    PX = PX.flatten()
    for x in PX:

        res -= x * math.log2(x)
    return res

def infomut(PXY):
    ePXY = entropie(PXY.np.flatten())
    PX = np.sum(PXY, 0)
    PY = np.sum(PXY, 1)
    ePX = entropie(PX)
    ePY = entropie(PY)
    return ePXY - ePX - ePY

```

```python
def H_croise(B, C):
    res = 0
    C = C.flatten()
    B = B.flatten()
    for i in range(len(B)):

        res -= B[i] * math.log2(C[i])
    return res

def dkl(p, q):

    return H_croise(p, q) - entropie(p)
```

Code pour la matrice des entropies croisées :

```python
def moyenne(LM): # Retourne la moyenne d'apparition des pixels

    res = np.full((8, 8), EPSILON,dtype=float)

    for x in LM:
        res += x

    res = res / len(LM)
    return res
```

```python
def moyenne_all(): # Retourne un tableau avec les moyennes pour chaque
                   # classe toujours dans
                   # l'odre habituel [coins, antidiag, diag, centre, verticale, horizontale]
    liste_moyennes = [moyenne(gen_corner()), moyenne(gen_o2line()),
                      moyenne(gen_o1line()), moyenne(gen_centre()),
                      moyenne(gen_vline()), moyenne(gen_hline())]

    return liste_moyennes
```

```python
def normalise(M):

    return M/M.sum()

def normalise_all():    # Retourne la pdf des 6 classes
    lm = moyenne_all()
    liste_norma = []
    for m in lm:
        liste_norma.append(normalise(m))
    return liste_norma
```

```python
def H_croise_all(): # Retourne la matrice des entropies croisées par couples de classes.

    ln = normalise_all()

    res = np.zeros((6, 6),dtype=float)
    for i in range(len(ln)):
        for j in range(len(ln)):
            res[i, j] = ((H_croise(ln[i], ln[j])+ H_croise(ln[j], ln[i]))/2)

    fig, ax = plt.subplots()

    pos = ax.matshow(res)

    ax.set_xticks(range(len(res)))
    ax.set_yticks(range(len(res)))
    ax.set_xticklabels(["Coins", "AntiDiagonale", "Diagonale", "Centre", "Verticale", "Horizontale"])
    ax.set_yticklabels(["Coins", "AntiDiagonale", "Diagonale", "Centre", "Verticale", "Horizontale"])
    fig.colorbar(pos, ax=ax)


    plt.savefig('matrice_des_H_croise.png')
    plt.show() #pour afficher les images


    return res
```

Pour tracer les courbes avec la méthode classique :

```
246  # Dictionnaire de fonctions (toujours dans l'odre habituel) pour faciliter l'implementation dans le code.
247  FCLASSES = {0 : gen_corner, 1 : gen_o2line, 2 : gen_o1line,
248              3 : gen_centre, 4 : gen_vline, 5: gen_hline}
```

```
251  # Retourne si le notre système trouve bien l'appartenance d'une image à sa classe.
252  def arg(T, forme_img, pdf):
253      T = T + EPSILON
254      T = normalise(T)
255
256      res = []
257      for i in pdf:
258          res.append((H_croise(T, i) + H_croise(i, T)))
259      if min(res) == res[forme_img]: #ARGMIN
260          return True
261      else:
262          return False
263
264  def bruitage(M, db): # Prend une image et la retourne bruitée.
265      n = 64/(10**(db/10))
266      L = np.arange(0, 64, 1, dtype=int)
267      np.random.shuffle(L)
268
269      for i in range(int(n)):
270
271          if M[L[i]//8,L[i]%8] > EPSILON:
272              M[L[i]//8,L[i]%8] = EPSILON
273
274          elif M[L[i]//8,L[i]%8] == EPSILON:
275              M[L[i]//8,L[i]%8] = 1
276      return M
277
```

```
297  def courbe(classe : int, pdf): # Donne l'image (l'accuracy) pour chaque x
298
299
300
301      x = [0, 1, 2, 3, 4, 5, 6, 8, 16, '+inf']
302      y = []
303
304      for db in x:
305          data = FCLASSES[classe]()
306          score = 0
307          for d in data :
308              test = d
309              if db != '+inf':
310                  test = bruitage(d, db)
311              if arg(test, classe, pdf):
312                  score += 1
313          y += [(score/TIRAGES)*100]
314      return y
315
```

```python
def dessine_courbe(): # Dessine les courbes avec matplotlib.
    x = [0, 1, 2, 3, 4, 5, 6, 8, 16, '+inf']
    xticks = ['0', '1', '2', '3', '4', '5', '6', '8', '16', '+inf']

    y = []
    pdf = normalise_all()
    fig, ax = plt.subplots()
    for c in CLASSES:
        y = courbe(CLASSES[c], pdf)
        print(c)
        print(y)

        ax.plot(x, y, label=c)



    #plt.xlim([0, 17])
    plt.ylim([0, 100+1])
    plt.xticks(xticks)
    plt.yticks(np.arange(0, 100+1, 20))

    plt.ylabel('% Accuracy')
    plt.xlabel('DB (RSB)')

    plt.legend()
    plt.savefig('Courbe_Scores.png')

    plt.show()

```

Calcul des courbes avec les 32 pixels pertinents :

```python
def bruitage_32px(M, db, top_px): # La version avec seulement les 32 pixels les plus pertinents.

    n = 64/(10**(db/10))
    L = np.arange(0, 64, 1, dtype=int)
    np.random.shuffle(L)

    for i in range(int(n)):
        if  (L[i]//8,L[i]%8) in top_px:
            if M[L[i]//8,L[i]%8] > EPSILON:
                M[L[i]//8,L[i]%8] = EPSILON

            elif M[L[i]//8,L[i]%8] == EPSILON:
                M[L[i]//8,L[i]%8] = 1
    return M


```

```python
def courbe_32px(classe: int, pdf, tp): # La version avec seulement les 32 pixels les plus pertinents.


    x = [0, 1, 2, 3, 4, 5, 6, 8, 16, '+inf']
    y = []

    for db in x:
        data = FCLASSES[classe]()
        score = 0
        for d in data :
            test = d
            if db != '+inf':
                test = bruitage_32px(d, db, tp)
            if arg(test, classe, pdf):
                score += 1
        y += [(score/TIRAGES)*100]
    return y

```

```python
def dessine_courbe_32px(): # La version avec seulement les 32 pixels les plus pertinents.
    x = [0, 1, 2, 3, 4, 5, 6, 8, 16, '+inf']
    xticks = ['0', '1', '2', '3', '4', '5', '6', '8', '16', '+inf']

    y = []
    pdf = normalise_all()
    tp = get_top_pixels_indices(Matrice_DKL())
    fig, ax = plt.subplots()
    for c in CLASSES:
        y = courbe_32px(CLASSES[c], pdf, tp)
        print(c)
        print(y)

        ax.plot(x, y, label=c)



    #plt.xlim([0, 17])
    plt.ylim([0, 100+1])
    plt.xticks(xticks)
    plt.yticks(np.arange(0, 100+1, 20))

    plt.ylabel('% Accuracy')
    plt.xlabel('DB (RSB)')

    plt.legend()
    plt.savefig('Courbe_Scores.png')

    plt.show()

```

```python
def Matrice_DKL(): # Retourne la matrice des moyennes des divergences de Kullback-Leibler
    ln = normalise_all()

    res = np.zeros((8, 8),dtype=float)
    for px_i in range(8):
        for px_j in range(8):
            som = 0
            for classe_A in range(6):

                for classe_B in range(6):
                    pjA = ln[classe_A][px_i][px_j] # Probabilité dans la classe A du pixel [i,j]
                    pjB = ln[classe_B][px_i][px_j] # Probabilité dans la classe B du pixel [i,j]
                    som += pjA * math.log2(pjA / pjB) + (1 - pjA) * math.log2((1 - pjA) / (1 - pjB))
                    #som += dkl(Aj, Bj)
            moy = som / 36
            #res[i, j] = dkl(ln[i], ln[j])
            res[px_i, px_j] = moy
    fig, ax = plt.subplots()

    return res


```

Calcul de la matrice de la moyenne des divergences :

Code emprunté, courtoisie de Swailem Abdullah L3 info :

```python
def get_top_pixels_indices(DIV): # Avoir les indices des 32 pixels pertinents.
    # Convertir la matrice DIV en une liste de tuples (valeur, (i, j))
    indices = [(i, j) for i in range(8) for j in range(8)]

    # Trier les indices par valeur en ordre décroissant
    sorted_indices = sorted(indices, key=lambda x: DIV[x[0], x[1]], reverse=True)

    # Récupérer les indices des 32 pixels avec les plus grandes valeurs
    top_pixels_indices = sorted_indices[:32]

    return top_pixels_indices


```

```python
def adjust_matrix(matrix, top_pixels_indices):
    # Créer une copie de la matrice originale pour éviter les modifications non désirées
    adjusted_matrix = matrix.copy()

    # Parcourir les indices des 32 pixels avec les plus grandes valeurs
    for i in range(matrix.shape[0]):
        for j in range(matrix.shape[1]):
            # Vérifier si l'indice (i, j) est dans top_pixels_indices
            if (i, j) not in top_pixels_indices:
                # Si l'indice n'est pas dans top_pixels_indices, supprimer la valeur du pixel
                adjusted_matrix[i, j] = 1


    return adjusted_matrix

```

Tests :

```python
DIV = Matrice_DKL()
TP = get_top_pixels_indices(DIV)
top_pixels_indices = get_top_pixels_indices(DIV)
plt.matshow(DIV)#adjust_matrix(DIV, TP))
plt.colorbar()
plt.show()

dessine_courbe_32px()
dessine_courbe()

```