

# **Compte rendu de deep learning TP 8**

**Elodie Difonzo  
Roqyun Ko**

L'objectif de ce TP est de mettre en place un réseau de neurones simple pour en comprendre le fonctionnement et la façon de l'entraîner.  
 Nous débuterons par une partie mathématique puis nous implémenterons le réseau avec la librairie PyTorch.

## **Partie 1 : Architecture VGG16**

1. Sachant que les couches fully connected comptent la majorité des paramètres du modèle, estimer grossièrement le nombre de paramètres de VGG16 (en utilisant les tailles données sur la Figure 1).

n° couche	Couche	Taille d'entrée	Taille de sortie	Nombre de poids	Nombre de biais	Taille d'image en sortie
1	Convolution	3	64	$3 \cdot (3 \cdot 3) \cdot 64$	$1 \cdot 64$	$224 \times 224$
2	Convolution	64	64	$64 \cdot (3 \cdot 3) \cdot 64$	$1 \cdot 64$	$224 \times 224$
3	Max Pooling	64	64	-	-	$112 \times 112$
4	Convolution	64	128	$64 \cdot (3 \cdot 3) \cdot 128$	$1 \cdot 128$	$112 \times 112$
5	Convolution	128	128	$128 \cdot (3 \cdot 3) \cdot 128$	$1 \cdot 128$	$112 \times 112$
6	Max Pooling	128	128	-	-	$56 \times 56$
7	Convolution	128	256	$128 \cdot (3 \cdot 3) \cdot 256$	$1 \cdot 256$	$56 \times 56$
8	Convolution	256	256	$256 \cdot (3 \cdot 3) \cdot 256$	$1 \cdot 256$	$56 \times 56$
9	Convolution	256	256	$256 \cdot (3 \cdot 3) \cdot 256$	$1 \cdot 256$	$56 \times 56$
10	Max Pooling	256	256	-	-	$28 \times 28$
11	Convolution	256	512	$256 \cdot (3 \cdot 3) \cdot 512$	$1 \cdot 512$	$28 \times 28$
12	Convolution	512	512	$512 \cdot (3 \cdot 3) \cdot 512$	$1 \cdot 512$	$28 \times 28$

13	Convolution	512	512	$512 \cdot (3 \cdot 3) \cdot 512$	$1 \cdot 512$	$28 \times 28$
14	Max Pooling	512	512	-	-	$14 \times 14$
15	Convolution	512	512	$512 \cdot (3 \cdot 3) \cdot 512$	$1 \cdot 512$	$14 \times 14$
16	Convolution	512	512	$512 \cdot (3 \cdot 3) \cdot 512$	$1 \cdot 512$	$14 \times 14$
17	Convolution	512	512	$512 \cdot (3 \cdot 3) \cdot 512$	$1 \cdot 512$	$14 \times 14$
18	Max Pooling	512	512	-	-	$7 \times 7$
20	Fully Connected	$25088$ ( $= 512 \cdot 7 \cdot 7$ )	4096	$512 \cdot 7 \cdot 7 \cdot 4096$	$1 \cdot 4096$	$1 \times 4096$
21	Fully Connected	4096	4096	$4096 \cdot 4096$	$1 \cdot 4096$	$1 \times 4096$
22	Fully Connected	4096	1000	$4096 \cdot 1000$	$1 \cdot 1000$	$1 \times 1000$

$(3 * 3) * (3 * 64) +$

$64 * 64 +$

$64 * 128 +$

$128 * 128 +$

$128 * 256 +$

$256 * 256 +$

$256 * 256 +$

$256 * 512 +$

$6 * 512 +$

$512 * 7 * 7 * 4096 +$

$4096 * 4096 +$

$4096 * 1000 = 138\ 344\ 128\ poids$

$2 * 64 + 2 * 128 + 3 * 256 + 6 * 512 + 2 * 4096 + 1000 = 13\ 416\ biais$

**138 357 544 paramètres**

**2. Quelle est la taille de sortie de la dernière couche de VGG16 ? À quoi correspond-elle ?**

La taille de sortie est 1000. Elle correspond à la taille des classes d'images.

**3. Bonus : Appliquer le réseau sur plusieurs images de votre choix et commenter les résultats de classification.**

Les classes de VGG16 :

{0: 'tench, Tinca tinca',

1: 'goldfish, Carassius auratus',

2: 'great white shark, white shark, man-eater, man-eating shark, Carcharodon carcharias',

...

352: 'impala, Aepyceros melampus',

353: 'gazelle',

...

998: 'ear, spike, capitulum',

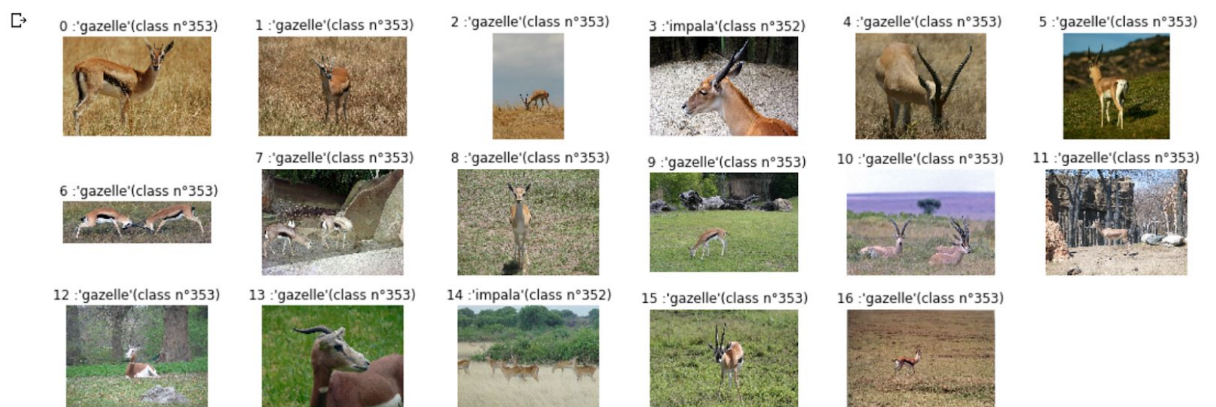
999: 'toilet tissue, toilet paper, bathroom tissue'}

Les images de gazelles:

<http://imagenet.stanford.edu/synset?wnid=n02423022>

<http://imagenet.stanford.edu/api/text/imagenet.synset.geturls?wnid=n02423022>

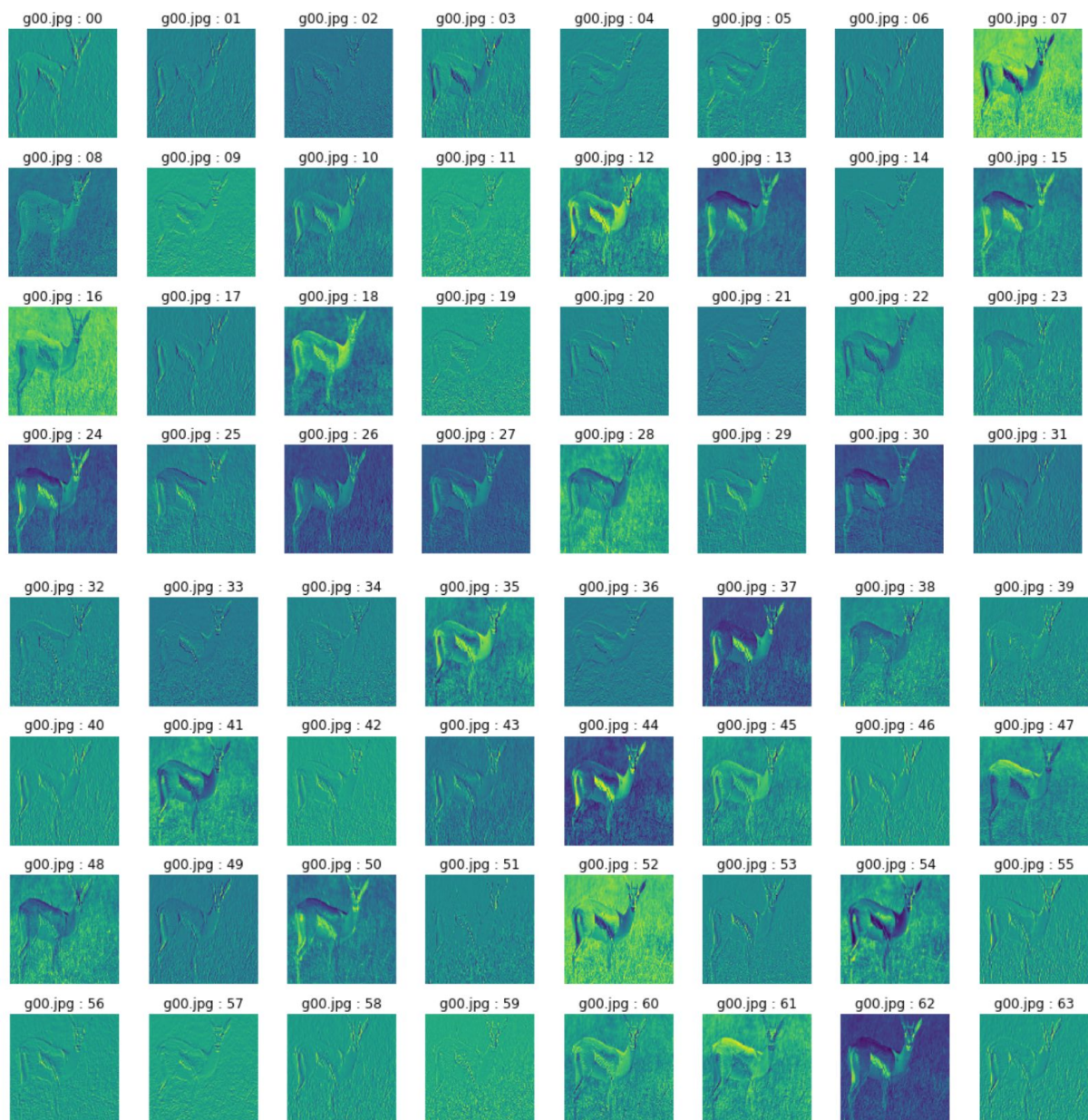
```
[21] plt.subplot(4, 6, i + 1)
      gazelle = Image.open(f'gazelles/test/a/{i:02}.jpg')
      plt.imshow(gazelle)
      plt.title(f"{i:2} : "+ labels[np.argmax(X[i])] + f' (class n°{np.argmax(X[i])})')
      plt.axis('off')
      plt.gcf().tight_layout()
```

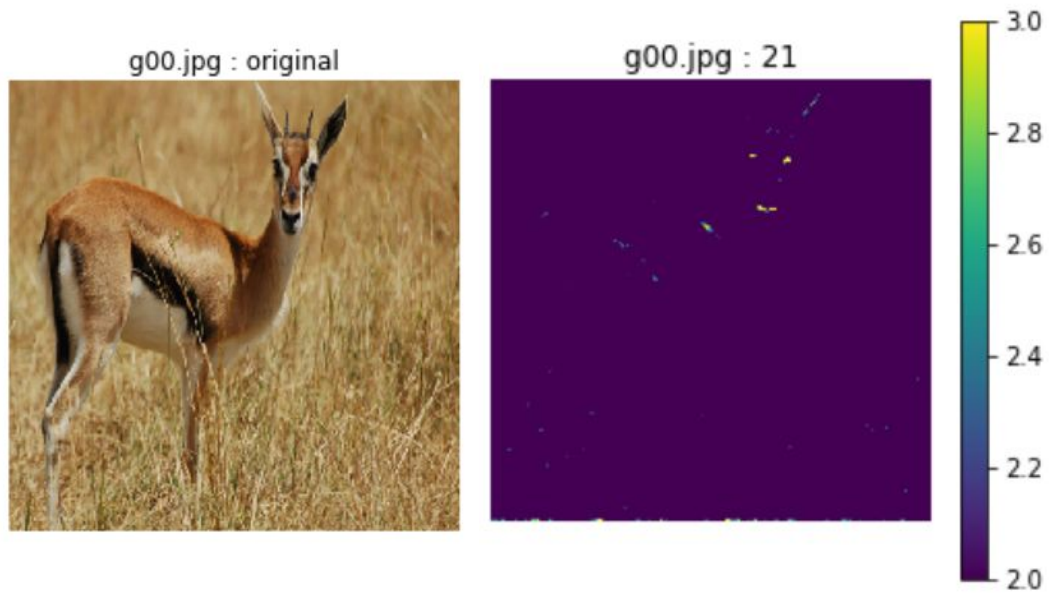


**4. Bonus : Afficher des images correspondant à différentes cartes obtenues après la première convolution. Comment interpréter ces cartes ?**

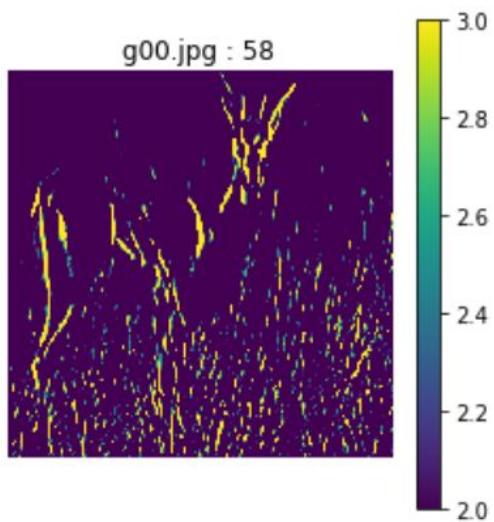
n° couche	Couche	Taille d'entrée	Taille de sortie	Nombre de poids	Nombre de biais	Taille d'image en sortie
1	Convolution	3	64	$3 \cdot (3 \cdot 3) \cdot 64$	$1 \cdot 64$	$224 \times 224$

**Les 64 cartes sorties obtenues à la couche n°1 :**





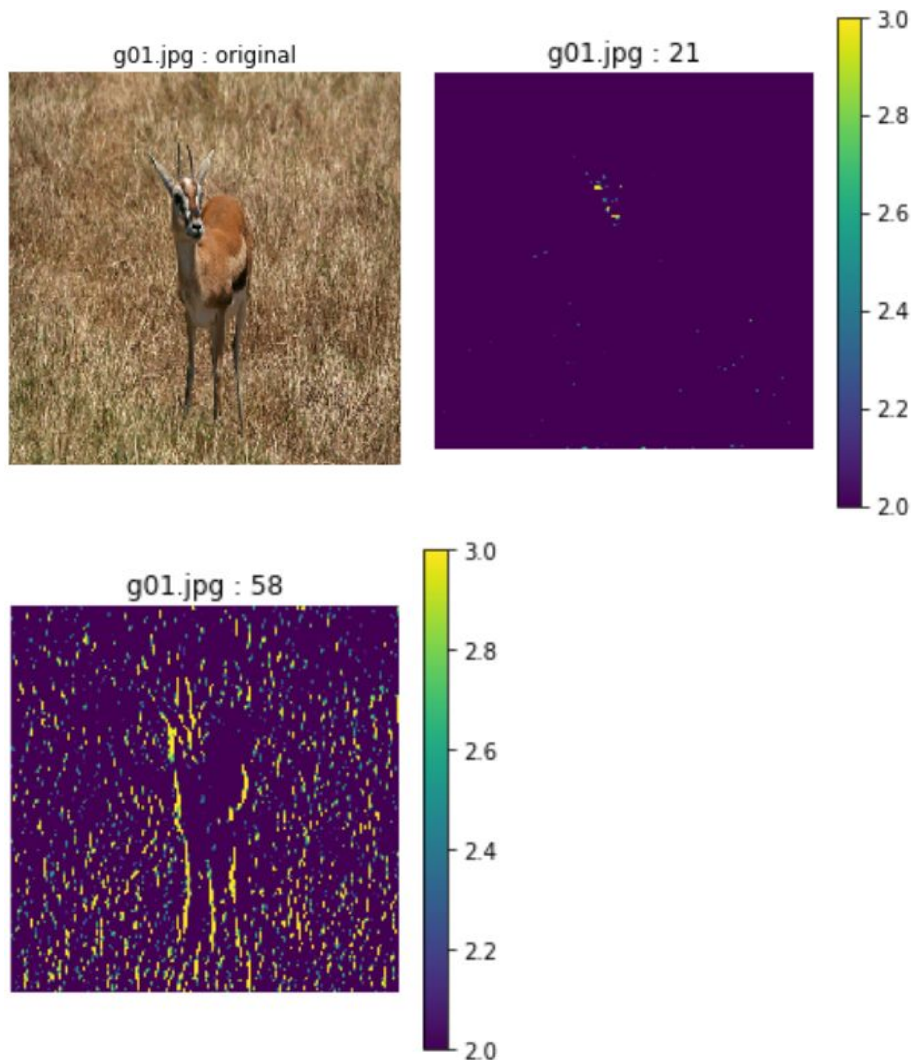
On étudie les régions les plus activées de ces cartes avec un seuil à 2. On constate que la carte n°21 extrait les yeux et le nez de la gazelle.



La carte n°58 extrait le bord des herbes et le corps de la gazelle.



On prend la 2ème image et on étudie les mêmes cartes.



La carte n°21 n'affiche que le nez et les yeux (ou un oeil) de la gazelle et la carte n°58 extrait le bord des herbes et le corps. Les cartes extraient les mêmes features qu'avant même avec l'image différente.

On conclut que les cartes extraient leurs propres features de manière fixe.

##### **5. Pourquoi ne pas directement apprendre VGG16 sur 15 Scène ?**

On ne peut pas apprendre VGG16 sur "15 Scènes" car les images fournies sont en niveaux de gris (1 couche) mais VGG16 prend des images en RGB (3 couches).

##### **6. En quoi le préapprentissage sur ImageNet peut aider à la classification de 15 Scene ?**

Apprendre un réseau de grande taille prend une longue durée même avec l'accélération de calcul avec le GPU. Les modèles extraient les features similaires des images pour faire les classifications similaires. Donc, on peut réutiliser les poids obtenus pour entraîner le modèle

VGG16 sur nos images afin de réduire le temps d'optimisation du modèle car il est déjà plus ou moins optimal pour faire la classification.

**7. Quelles sont les limites de cette approche par feature extraction ?**

Il faut que l'extraction de particularités soit correctement fait. Si le modèle VGG16 extrait, par exemple, des caractéristiques moins importantes que les autres d'une image, alors on risque de mal classer les images.

**8. Quelle est l'influence de la couche à laquelle les features sont extraites ?**

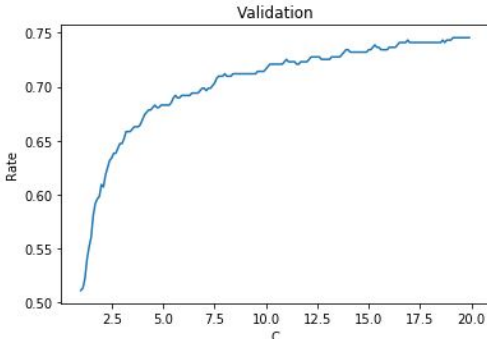
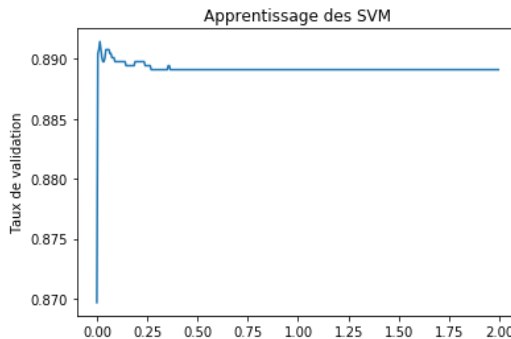
La couche à laquelle on extrait les features a 4096 neurones, activés ou non. C'est-à-dire qu'elle extrait 4096 features à partir d'une image pré-traitée.

**9. Les images de 15 Scene sont en noir et blanc, alors que VGG16 attend des images RGB. Comment contourner ce problème ?**

On duplique la couche de niveaux de gris 2 fois plus afin d'émuler une image RGB (3 couches).



## 10. Comparez votre résultat aux résultats obtenus avec les BoW.

Résultat : “Feature extraction” par BoW (TP3)	Résultat : “Feature extraction” par VGGrelu7
 <pre> In [15]: C = 5.80          clf = svm.SVC(C=C, gamma='scale')          clf.fit(train_data, train_label)          print(clf.score(test_data, test_label)) 0.688195991091314  In [16]: C = 7          clf = svm.SVC(C=C, gamma='scale')          clf.fit(train_data, train_label)          print(clf.score(test_data, test_label)) 0.6937639198218263  In [17]: C = 9          clf = svm.SVC(C=C, gamma='scale')          clf.fit(train_data, train_label)          print(clf.score(test_data, test_label)) 0.6937639198218263  In [18]: C = 19.9          clf = svm.SVC(C=C, gamma='scale')          clf.fit(train_data, train_label)          print(clf.score(test_data, test_label)) 0.7360801781737194 </pre>	 <pre> Apprentissage des SVM C = 0.0001, VAL : 86.96817420435511 % C = 0.0051, VAL : 89.04522613065326 % C = 0.0101, VAL : 89.07872696817421 % C = 0.0151, VAL : 89.14572864321609 % C = 0.0201, VAL : 89.07872696817421 % C = 0.0251, VAL : 89.01172529313233 % C = 0.0301, VAL : 88.97822445561138 % C = 0.0351, VAL : 88.97822445561138 % C = 0.0401, VAL : 89.01172529313233 % C = 0.0451, VAL : 89.07872696817421 % C = 0.0501, VAL : 89.07872696817421 % </pre>

La feature extraction par VGG16relu7 est beaucoup plus précise que le BoW.

## 11. Plutôt que d'apprendre un classifieur indépendant, est-il possible de n'utiliser que le réseau de neurones ? Si oui, expliquer comment.

Oui, c'est possible. On remplace la dernière couche FC de la taille 1000 avec une couche FC de taille 15.

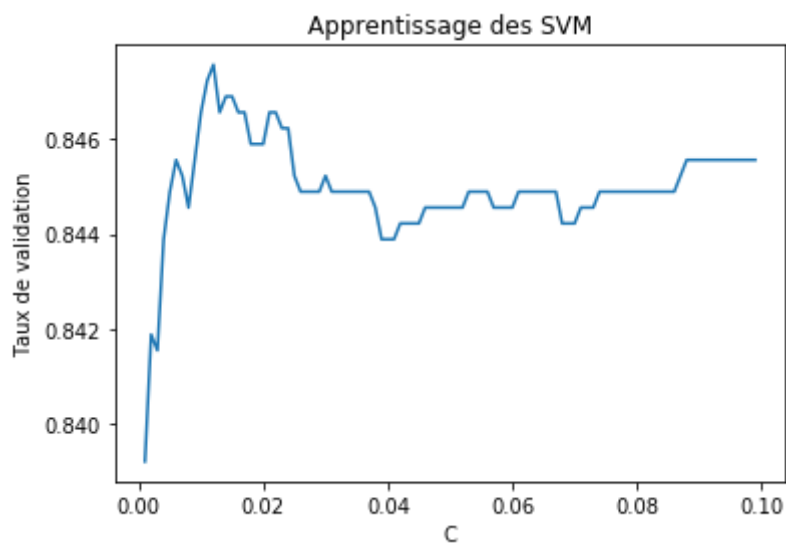
12. Pour chaque amélioration testée, expliquer ses justifications et commenter les résultats obtenus.

## Amélioration du modèle

### 1. Modification de couches / Réglage du paramètre C :

#### a. Modification apportée (n°1):

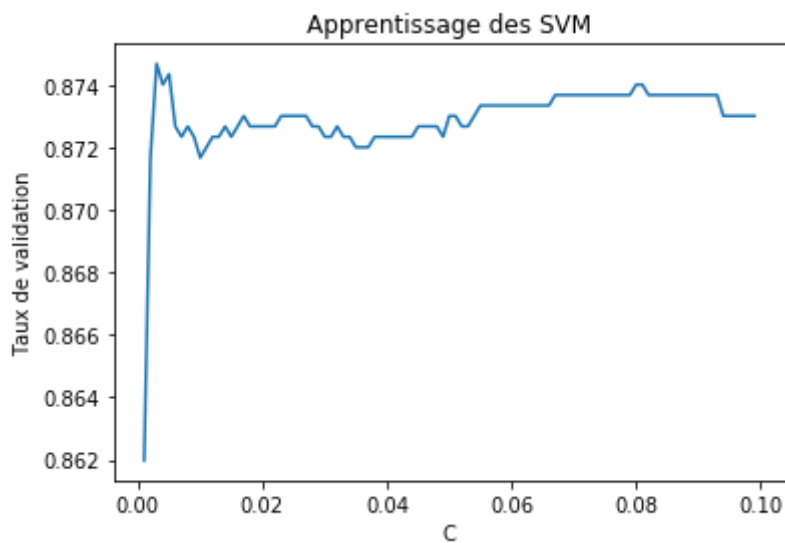
n° couche	Couche	Taille d'entrée	Taille de sortie	Nombre de poids	Nombre de biais	Taille d'image en sortie
20	Fully Connected	25088 (= $512 \cdot 7 \cdot 7$ )	8196	$512 \cdot 7 \cdot 7 \cdot 8196$	$1 \cdot 8196$	$1 \times 8196$
21	Fully Connected	8196	8196	$8196 \cdot 8196$	$1 \cdot 8196$	$1 \times 8196$



Le score maximal de l'ensemble de test : 84.7571189279732 % ( C = 0.0120)

### b. Modification apportée (n°2):

n° couche	Couche	Taille d'entrée	Taille de sortie	Nombre de poids	Nombre de biais	Taille d'image en sortie
20	Fully Connected	25088 (= $512 \cdot 7 \cdot 7$ )	16384	$512 \cdot 7 \cdot 7 \cdot 16384$	$1 \cdot 16384$	$1 \times 8196$
21	Fully Connected	16384	16384	$16384 \cdot 16384$	$1 \cdot 16384$	$1 \times 16384$



**Le score maximal de l'ensemble de test : 87.47 % ( C = 0,003 )**

### c. Conclusion

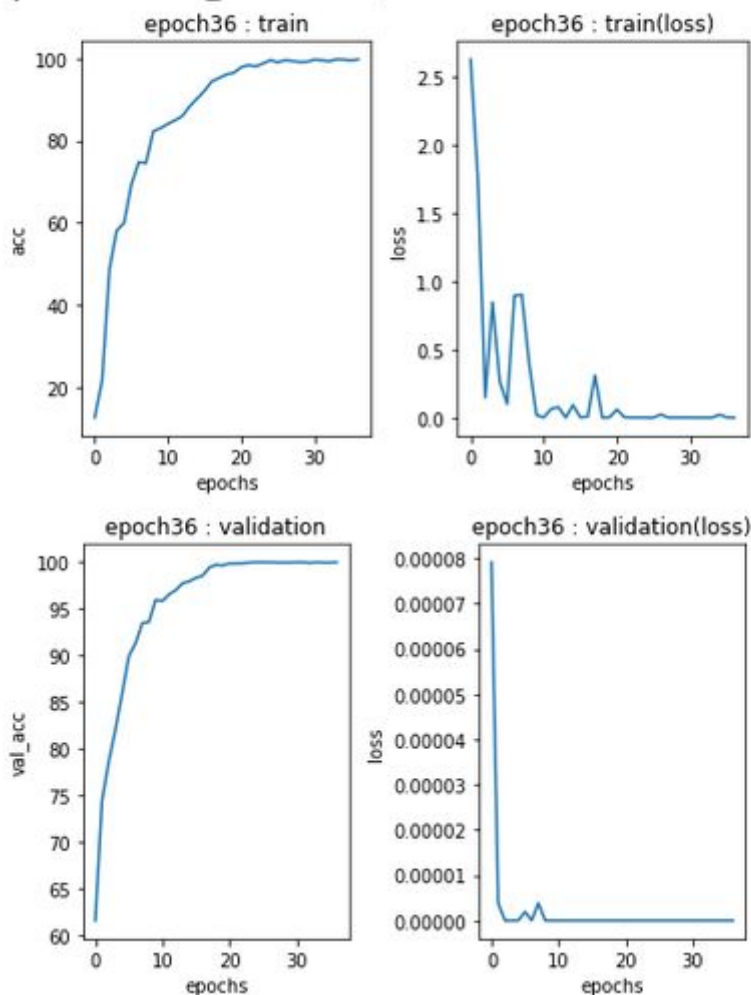
On observe que l'augmentation du nombre de neurones dans une couche linéaire influence légèrement le résultat (89,15 % -> 84,75 % -> 87,47 %). L'utilisation d'une couche linéaire permet de quantifier les features. On devine que le réseau peut extraire plus de features avec le nombre de neurones augmenté et la précision augmente mais il faut choisir le nombre de neurones correcte pour la meilleure classification.

## 2. Nouvelle couche de classification (FC Layer - 15 Scènes)

### a. Résultat

```
Sequential(  
  (0): Linear(in_features=25088, out_features=1024, bias=True)  
  (1): ReLU(inplace=True)  
  (2): Dropout(p=0.5, inplace=False)  
  (3): Linear(in_features=1024, out_features=1024, bias=True)  
  (4): ReLU(inplace=True)  
  (5): Dropout(p=0.5, inplace=False)  
  (6): Linear(in_features=1024, out_features=15, bias=True)  
)
```

```
=====EPOCH36=====  
epoch36 : acc = 1496/1500 = 99.73  
epoch36 : val_acc = 2985/2985 = 100.00
```



### b. Conclusion

Le réseau s'entraîne sans sur-apprentissage et atteint la précision de 100 %.

Ce résultat est beaucoup mieux que la classification par SVM linéaire.

On suppose que la classification par une couche linéaire marche mieux car il y a beaucoup plus d'hyper-paramètres qu'on peut définir alors que le SVM n'a qu'un seul hyper paramètre, C, qui sert à définir la tolérance d'erreur (mauvaise classification).

### 3. Etude des méthode de la réduction de la dimension et leurs impactes sur la performance et le temps d'exécutions

#### a. MaxPool2d

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace=True)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace=True)
  (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace=True)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace=True)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace=True)
  (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (18): ReLU(inplace=True)
  (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace=True)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace=True)
  (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (25): ReLU(inplace=True)
  (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (27): ReLU(inplace=True)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace=True)
  (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
```

#### Feature extraction

Batch 000/188

Batch 050/188

Batch 100/188

Batch 150/188

2.5965974628925323 seconds per batch

Batch 000/374

Batch 050/374

Batch 100/374

Batch 150/374

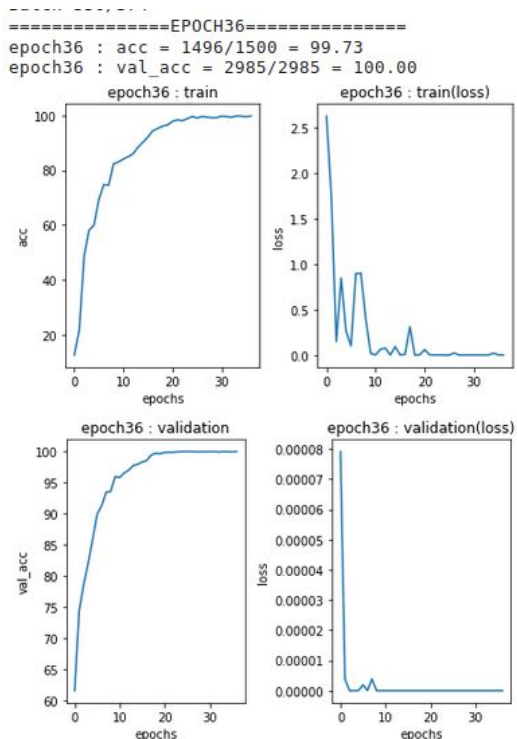
Batch 200/374

Batch 250/374

Batch 300/374

Batch 350/374

5.150318294763565 seconds per batch





## b. AvgPool2d

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace=True)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace=True)
  (9): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace=True)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace=True)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace=True)
  (16): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (18): ReLU(inplace=True)
  (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace=True)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace=True)
  (23): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (25): ReLU(inplace=True)
  (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (27): ReLU(inplace=True)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace=True)
  (30): AvgPool2d(kernel_size=2, stride=2, padding=0)
)
```

Feature extraction

Batch 000/188

Batch 050/188

Batch 100/188

Batch 150/188

2.608605742454529 seconds per batch

Batch 000/374

Batch 050/374

Batch 100/374

Batch 150/374

Batch 200/374

Batch 250/374

Batch 300/374

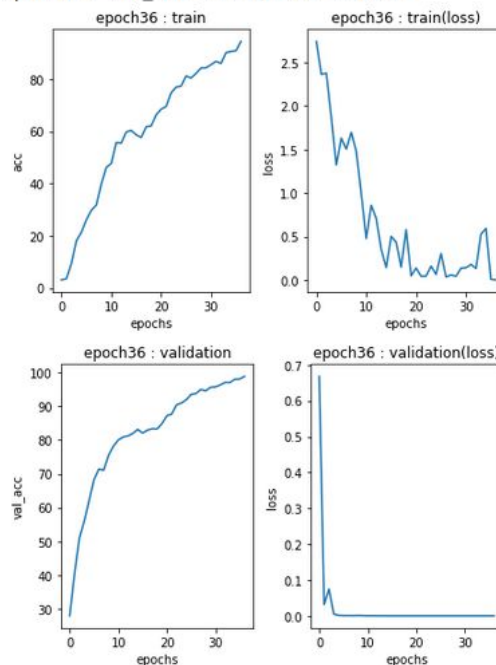
Batch 350/374

5.161018371582031 seconds per batch

=====EPOCH36=====

epoch36 : acc = 1417/1500 = 94.47

epoch36 : val\_acc = 2952/2985 = 98.89





### c. SumPool2d (LPPool2d, p = 1)

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): LPPool2d(norm_type=1, kernel_size=2, stride=2, ceil_mode=False)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace=True)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace=True)
  (9): LPPool2d(norm_type=1, kernel_size=2, stride=2, ceil_mode=False)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace=True)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace=True)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace=True)
  (16): LPPool2d(norm_type=1, kernel_size=2, stride=2, ceil_mode=False)
  (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (18): ReLU(inplace=True)
  (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace=True)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace=True)
  (23): LPPool2d(norm_type=1, kernel_size=2, stride=2, ceil_mode=False)
  (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (25): ReLU(inplace=True)
  (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (27): ReLU(inplace=True)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace=True)
  (30): LPPool2d(norm_type=1, kernel_size=2, stride=2, ceil_mode=False)
)
```

#### Feature extraction

Batch 000/188

Batch 050/188

Batch 100/188

Batch 150/188

2.771664947271347 seconds per batch

Batch 000/374

Batch 050/374

Batch 100/374

Batch 150/374

Batch 200/374

Batch 250/374

Batch 300/374

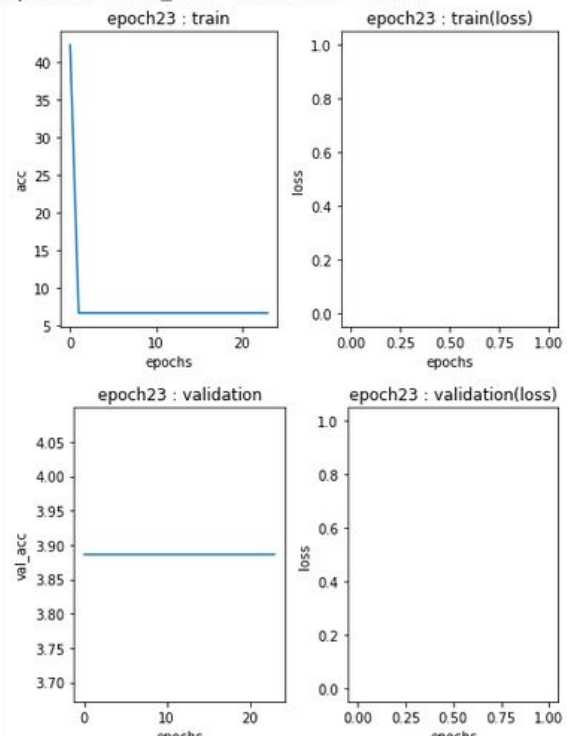
Batch 350/374

5.466719776391983 seconds per batch

=====EPOCH23=====

epoch23 : acc = 100/1500 = 6.67

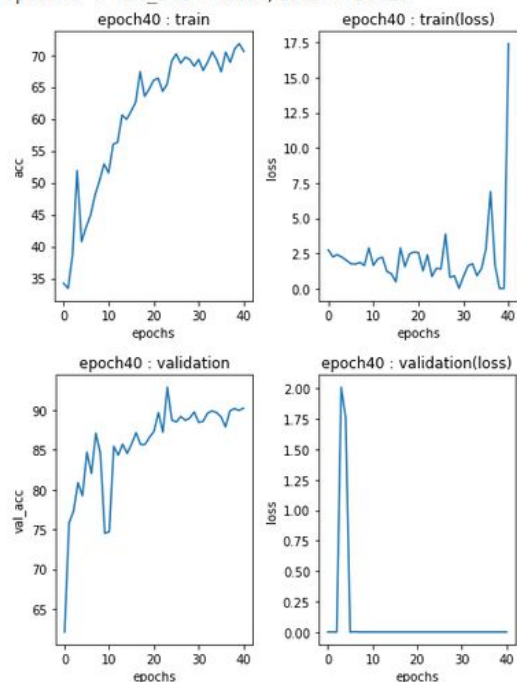
epoch23 : val\_acc = 116/2985 = 3.89



## d. Norm2Pool2d (LPPool2d, p = 2)

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): LPPool2d(norm_type=2, kernel_size=2, stride=2, ceil_mode=False)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace=True)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace=True)
  (9): LPPool2d(norm_type=2, kernel_size=2, stride=2, ceil_mode=False)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace=True)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace=True)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace=True)
  (16): LPPool2d(norm_type=2, kernel_size=2, stride=2, ceil_mode=False)
  (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (18): ReLU(inplace=True)
  (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace=True)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace=True)
  (23): LPPool2d(norm_type=2, kernel_size=2, stride=2, ceil_mode=False)
  (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (25): ReLU(inplace=True)
  (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (27): ReLU(inplace=True)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace=True)
  (30): LPPool2d(norm_type=2, kernel_size=2, stride=2, ceil_mode=False)
)
```

=====EPOCH40=====  
 epoch40 : acc = 1059/1500 = 70.60  
 epoch40 : val\_acc = 2694/2985 = 90.25



Feature extraction

Batch 000/188

Batch 050/188

Batch 100/188

Batch 150/188

2.7971440851688385 seconds per batch

Batch 000/374

Batch 050/374

Batch 100/374

Batch 150/374

Batch 200/374

Batch 250/374

Batch 300/374

Batch 350/374

5.495275557041168 seconds per batch

### e. Benchmark

n°	Epoch d'échantillon	pooling	train_acc	val_acc	stable (test)	stable (train)	temps d'exécution par mini-batch (test)	temps d'exécution par mini-batch (train)
1	37	Max	99.73 %	100 %	Oui	Oui	2.60 s	5.15 s
2	37	Avg	94.47 %	98.89 %	Oui	Oui	2.61 s	5.16 s
3	24	Sum	6.67 %	3.89 %	Non	Non	2.77s	5.47 s
4	41	Norm2	70.6 %	90.25 %	Oui	Non	2.80 s	5.50 s

Avec le pooling moyen, l'évolution de la précision et de la loss est moins stable qu'avec le pooling max, mais le modèle est aussi performant que le réseau avec le pooling max. Il atteint autant de précision et prend autant de temps pour exécuter le modèle.

En revanche, l'utilisation de la couche LPPool2d (pooling somme et pooling norme 2) rend le temps d'exécution 6 ~ 8 %. La performance du réseau baisse aussi. Le réseau avec le pooling somme ne marche pas du tout. Le réseau avec le pooling norme 2 a une haute précision mais il est moins performant que le pooling max et le pooling moyen. Surtout, il a une grande valeur de loss et l'évolution de loss est instable.

Le pooling max n'extrait que les features principaux alors que le pooling moyen retire aussi les informations contextuelles lors du sous-échantillonnage. Néanmoins, les informations contextuelles ne sont pas nécessaires pour classer les images. Par conséquent, les couches de réseaux contiennent les informations inutiles au fur et à mesure lors de l'utilisation du pooling moyen (pooling norme 1) et ses variations (pooling somme / pooling norme 2). De plus, le pooling sum retourne les valeurs qui ne sont pas normalisées. Donc, les features extraits ne sont pas à l'échelle de même grandeur. Cela rend le réseau difficile à classer correctement.

### f. Conclusion

Pooling max > Pooling moyen > Pooling norme 2 > Pooling somme

Surtout, il est déconseillé d'utiliser pooling norme 2 et pooling somme avec le réseau VGG16.