



Accueil > Cours > Classez et segmentez des données visuelles > TP : Implémentez votre premier réseau de neurones avec Keras

## Classez et segmentez des données visuelles

15 heures

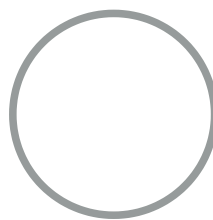


Difficile

Mis à jour le 23/05/2019



### TP : Implémentez votre premier réseau de neurones avec Keras



00:00



Passons à la pratique ! Dans ce chapitre, vous allez apprendre à utiliser Keras, une bibliothèque très intuitive de *Deep Learning* en Python.

L'objet de notre étude est VGG-16, une version du réseau de neurones convolutif très connu appelé VGG-Net. Nous allons d'abord l'implémenter de A à Z pour découvrir Keras, puis nous allons voir comment classer des images de manière efficace. Pour cela, nous allons exploiter le réseau VGG-16 pré-entraîné fourni par Keras, et mettre en oeuvre le *Transfer Learning*.

C'est parti !

## Architecture de VGG-16

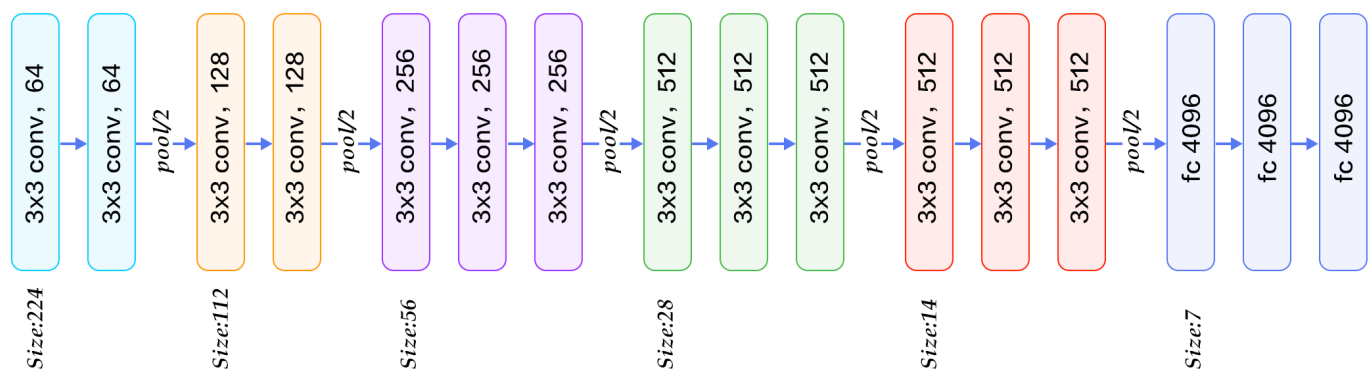


Avant de vous lancer dans l'implémentation d'un réseau de neurones, vous devez impérativement comprendre son architecture dans les moindres détails ! Nous allons donc passer un peu de temps à étudier la configuration des différentes couches de VGG-16.

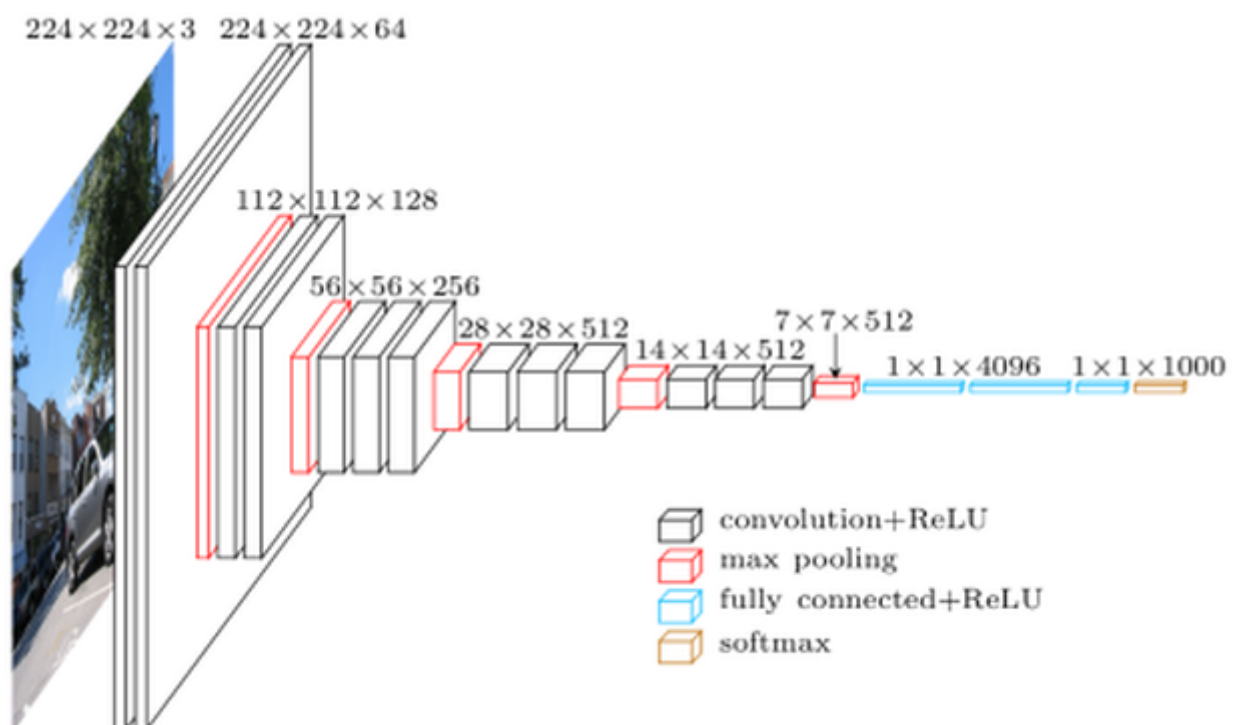
VGG-16 est constitué de plusieurs couches, dont 13 couches de convolution et 3 *fully-connected*. Il doit donc apprendre les poids de 16 couches.

Il prend en entrée une image en couleurs de taille  $224 \times 224$  px et la classe dans une des 1000 classes. Il renvoie donc un vecteur de taille 1000, qui contient les probabilités d'appartenance à chacune des classes.

L'architecture de VGG-16 est illustrée par les schémas ci-dessous :



Architecture de VGG-16



## Représentation 3D de l'architecture de VGG-16

Chaque couche de convolution utilise des filtres en couleurs de taille  $3 \times 3$  px, déplacés avec un pas de 1 pixel. Le *zero-padding* vaut 1 pixel afin que les volumes en entrée aient les mêmes dimensions en sortie. Le nombre de filtres varie selon le "bloc" dans lequel la couche se trouve. De plus, un paramètre de biais est introduit dans le produit de convolution pour chaque filtre.

Chaque couche de convolution a pour fonction d'activation une ReLU. Autrement dit, il y a toujours une couche de correction ReLU après une couche de convolution.

L'opération de *pooling* est réalisée avec des cellules de taille  $2 \times 2$  px et un pas de 2 px – les cellules ne se chevauchent donc pas.

Les deux premières couches *fully-connected* calculent chacune un vecteur de taille 4096, et sont chacune suivies d'une couche ReLU. La dernière renvoie le vecteur de probabilités de taille 1000 (le nombre de classes) en appliquant la fonction *softmax*. De plus, ces trois couches utilisent un paramètre de biais pour chaque élément du vecteur en sortie.

Maintenant, vérifions si vous avez bien compris les chapitres précédents...

**Petit exercice :** retrouvez les dimensions du volume renvoyé par chaque couche.

**Solution :** il suffit d'appliquer les formules données dans le chapitre précédent !

Par exemple, détaillons le calcul pour le volume en sortie du premier bloc. La première couche de convolution a pour paramètres  $K = 64$ ,  $F_C = 3$ ,  $S_C = 1$  et  $P = 1$ . Elle renvoie donc un volume de dimensions  $W_C \times H_C \times D_C$  avec  $W_C = H_C = \frac{224-3+2}{1} + 1 = 224$  et  $D_C = 64$ . La couche de ReLU ne modifie pas les dimensions du volume en entrée. On applique le même raisonnement pour les couches de convolution et ReLU suivantes.

La couche de *pooling* du premier bloc reçoit donc en entrée un volume de dimensions  $224 \times 224 \times 64$ . Comme elle a pour paramètres  $F_P = 2$  et  $S_P = 2$ , elle renvoie un volume de dimensions  $W_P \times H_P \times D_P$  avec  $W_P = H_P = \frac{224-2}{2} + 1 = 112$  et  $D_P = 64$ . Finalement, le volume en sortie du premier bloc est de dimensions  $112 \times 112 \times 64$ .

**Petit exercice, le retour :** combien de paramètres VGG-16 doit-il apprendre pendant la phase d'entraînement ?

**Solution :** VGG-16 apprend 138 357 544 paramètres ! Pour trouver ce petit nombre, il suffit de compter les poids de toutes les couches de convolution et *fully-connected*, sans oublier les paramètres de biais.

Par exemple, pour la première couche de convolution, le réseau doit apprendre 64 filtres en couleurs (donc de profondeur 3) de taille  $3 \times 3$ , ainsi qu'un paramètre de biais pour chaque filtre.

Cela fait un total de  $(3*3*3)*64 + 64 = 1792$  paramètres.

Le nombre de paramètres d'une couche *fully-connected* s'obtient en multipliant le nombre d'éléments en sortie avec celui en entrée, et en ajoutant le nombre de biais. Ainsi, le réseau doit apprendre  $7*7*512*4096 + 4096 = 102\,764\,544$  paramètres pour la première couche *fully-connected*,  $4096*4096 + 4096 = 16\,781\,312$  pour la deuxième, et  $4096*1000 + 1000 = 4\,097\,000$  pour la dernière.

## Implémentation de VGG-16



Maintenant que vous maîtrisez l'architecture de VGG-16, nous pouvons passer à la partie la plus rigolote : l'implémentation !

Implémenter un réseau de neurones avec Keras revient à créer un modèle `Sequential` et à l'enrichir avec les couches correspondantes dans le bon ordre. L'étape la plus difficile est de définir correctement les paramètres de chacune des couches – d'où l'importance de bien comprendre l'architecture du réseau !

La création du modèle se fait comme ci-dessous :

python

```
1 from keras.models import Sequential
2
3 my_VGG16 = Sequential() # Création d'un réseau de neurones vide
```

Les couches s'ajoutent soit en tant que paramètres d'entrée du constructeur `Sequential()`, soit une par une avec la méthode `mon_reseau_VGG.add()`.

Les couches de convolution, *pooling* et *fully-connected* correspondent à des instances des classes respectives [Conv2D](#), [MaxPooling2D](#) et [Dense](#) du module `keras.layers`. Une couche ReLU peut être créée soit en instanciant la classe [Activation](#), soit en ajoutant un argument au constructeur de la couche qui la précède.

Les fonctions d'initialisation (ou constructeurs) des classes présentent une multitude de paramètres, plus ou moins sophistiqués. Comme l'implémentation basique de VGG-16 ne nécessite que certains d'entre eux, je ne vais pas vous expliquer tous les paramètres disponibles.

Néanmoins, je vous encourage à lire la documentation de chaque classe pour bien comprendre le format et l'ordre des paramètres en entrée du constructeur.

Pour construire une couche de convolution, nous devons préciser le nombre de filtres utilisés, leur taille, le pas et le *zero-padding*. Ils correspondent respectivement aux arguments `filters`, `kernel_size`, `strides` et `padding` du constructeur de la classe `Conv2D`.

Seulement deux valeurs sont possibles pour `padding` : `'same'` ou `'valid'`. La couche de convolution réduit la taille du volume en entrée avec l'option `'valid'`, mais la conserve avec `'same'`. Pour VGG-16, on utilisera donc toujours l'option `'same'`.

S'il s'agit de la toute première couche de convolution, il faut préciser dans l'argument `input_shape` les dimensions des images en entrée du réseau. Pour VGG-16, `input_shape = (224, 224, 3)`.

Enfin, pour indiquer la présence d'une couche ReLU juste après la couche de convolution, on ajoute l'argument `activation = 'relu'`.

Une couche de *pooling* est définie par la taille des cellules de *pooling* et le pas avec lequel on les déplace. Ces paramètres sont précisés dans les arguments respectifs `pool_size` et `strides` du constructeur de la classe `MaxPooling2D`.

A ce stade, vous pouvez déjà implémenter quasiment tout le réseau VGG-16 ! Par exemple, la construction du premier bloc de couches est détaillée ci-dessous :

python

```
1 from keras.models import Sequential
2 from keras.layers import Conv2D, MaxPooling2D
3
4 my_VGG16 = Sequential() # Création d'un réseau de neurones vide
5
6 # Ajout de la première couche de convolution, suivie d'une couche ReLU
7 my_VGG16.add(Conv2D(64, (3, 3), input_shape=(224, 224, 3), padding='same', activation='relu'))
8
9 # Ajout de la deuxième couche de convolution, suivie d'une couche ReLU
10 my_VGG16.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
11
12 # Ajout de la première couche de pooling
13 my_VGG16.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
```

Il ne reste plus qu'à ajouter les couches *fully-connected*, en créant des objets de la classe `Dense`. Ce type de couche reçoit en entrée un vecteur 1D. Il faut donc convertir les matrices 3D renvoyées par la dernière couche de *pooling*. Pour cela, on instancie la classe `Flatten` juste avant la première couche *fully-connected*.

L'argument `units` du constructeur de `Dense` permet de préciser la taille du vecteur en sortie. De plus, si une correction ReLU ou softmax est effectuée juste après la couche *fully-connected*, on l'indique dans le paramètre `activation`.

Ainsi, les trois dernières couches *fully-connected* et leur fonction d'activation (ReLU pour les deux premières, softmax pour la dernière) sont ajoutées de la manière suivante :

python

```
1 from keras.layers import Flatten, Dense
2
3 my_VGG16.add(Flatten()) # Conversion des matrices 3D en vecteur 1D
4
```

```
5 # Ajout de la première couche fully-connected, suivie d'une couche ReLU
6 my_VGG16.add(Dense(4096, activation='relu'))
7
8 # Ajout de la deuxième couche fully-connected, suivie d'une couche ReLU
9 my_VGG16.add(Dense(4096, activation='relu'))
10
11 # Ajout de la dernière couche fully-connected qui permet de classifier
12 my_VGG16.add(Dense(1000, activation='softmax'))
```

Vous êtes désormais capables d'implémenter VGG-16 en entier : lancez-vous ! Pour vérifier que votre implémentation est bonne, vous pouvez la comparer avec celle fournie par Keras [ici](#).

Une fois implémenté, vous devez d'abord "compiler" votre modèle avec la méthode [Sequential.compile\(\)](#), puis l'entraîner avec [Sequential.fit\(\)](#). Mais la phase d'entraînement peut prendre au moins un mois, même avec un GPU ! Nous allons donc comment gagner du temps dans les deux prochaines sections.

## Utilisation du VGG-16 pré-entraîné



Vous savez maintenant comment implémenter un réseau de neurones convolutif de A à Z avec Keras. Dans cette partie, nous allons apprendre à classifier des images avec le modèle VGG-16 fourni par Keras et pré-entraîné sur ImageNet.

La première étape consiste à charger ce modèle avec la classe [VGG16](#) de

`keras.applications.vgg16` :

python

```
1 from keras.applications.vgg16 import VGG16
2
3 model = VGG16() # Création du modèle VGG-16 implémenté par Keras
```

Par défaut, le constructeur `VGG16()` crée le réseau VGG-16 pré-entraîné sur ImageNet. Si à l'avenir, pour d'autres projets, vous souhaitez initialiser aléatoirement les poids, il faudra préciser `weight=None` en argument.

Le constructeur possède d'autres paramètres pour faire du *Transfer Learning*, que nous allons utiliser dans la partie suivante.

Nous allons utiliser ce réseau pré-entraîné pour classer une image dans une des 1000 catégories d'ImageNet.

Nous devons d'abord charger l'image et la pré-traiter afin qu'elle respecte bien les spécifications des images en entrée de VGG-16. Pour cela, nous allons utiliser les fonctions du module

`keras.preprocessing.image` et `keras.preprocessing.vgg16` :

1. VGG-16 reçoit des images de taille (224, 224, 3) : la fonction `load_img` permet de charger l'image et de la redimensionner correctement

2. Keras traite les images comme des tableaux numpy : `img_to_array` permet de convertir l'image chargée en tableau numpy
3. Le réseau doit recevoir en entrée une collection d'images, stockée dans un tableau de 4 dimensions, où les dimensions correspondent (dans l'ordre) à (nombre d'images, largeur, hauteur, profondeur). Pour l'instant, nous donnons qu'une image en entrée : `numpy.reshape` permet d'ajouter la première dimension (nombre d'images = 1) à notre image.
4. Enfin, `preprocess_input` permet d'appliquer les mêmes pré-traitements que ceux utilisés sur l'ensemble d'apprentissage lors du pré-entraînement.

Ainsi, on prépare l'image comme ci-dessous :

python

```
1 from keras.preprocessing.image import load_img, img_to_array
2 from keras.applications.vgg16 import preprocess_input
3
4 img = load_img('cat.jpg', target_size=(224, 224)) # Charger l'image
5 img = img_to_array(img) # Convertir en tableau numpy
6 img = img.reshape((1, img.shape[0], img.shape[1], img.shape[2])) # Créer la collection d'images (un seul échantillon)
7 img = preprocess_input(img) # Prétraiter l'image comme le veut VGG-16
```

Nous pouvons maintenant donner l'image en entrée du réseau et prédire sa classe :

python

```
1 y = model.predict(img) # Prédire la classe de l'image (parmi les 1000 classes d'ImageNet)
```

On obtient la sortie finale du réseau, c'est-à-dire une liste de 1000 probabilités.

Les classes correspondant à ces probabilités ne sont pas explicitement données. La fonction

`decode_predictions` de `keras.applications.vgg16` permet alors de récupérer cette information. Ainsi, on peut faire un top 3 des classes les plus probables de l'image :

python

```
1 from keras.applications.vgg16 import decode_predictions
2
3 # Afficher les 3 classes les plus probables
4 print('Top 3 :', decode_predictions(y, top=3)[0])
```

## Transfer Learning



Dans la section précédente, nous avons utilisé le réseau VGG-16 fourni par Keras pour résoudre le même problème de classification que celui sur lequel il a été pré-entraîné (classification à 1000 classes avec ImageNet). En pratique, vous serez très probablement confrontés à un nouveau problème de classification. Dans ce cas, savoir mettre en oeuvre le *Transfer Learning* vous sera très utile !

Je vous encourage tout d'abord à bien vous remettre en tête les stratégies possibles, introduites dans le chapitre précédent : *fine-tuning* total, extraction des *features*, et *fine-tuning* partiel.

Dans les trois cas, il faut remplacer les dernières couches *fully-connected* qui permettent de classifier l'image dans une des 1000 classes (ImageNet) par un classifieur plus adapté à notre problème. Par exemple, supposons qu'on veuille différencier un chat d'un chien (classification binaire). La suppression des dernières couches se fait en ajoutant l'argument

`include_top = False` lors de l'import du modèle pré-entraîné. Dans ce cas, il faut aussi préciser les dimensions des images en entrée ( `input_shape` ) :

python

```
1 from keras.applications.vgg16 import VGG16
2 from keras.layers import Dense
3
4 # Charger VGG-16 pré-entraîné sur ImageNet et sans les couches fully-connected
5 model = VGG16(weights="imagenet", include_top=False, input_shape=(224, 224, 3))
6
7 # Récupérer la sortie de ce réseau
8 x = model.output
9
10 # Ajouter la nouvelle couche fully-connected pour la classification à 10 classes
11 predictions = Dense(10, activation='softmax')(x)
12
13 # Définir le nouveau modèle
14 new_model = Model(inputs=model.input, outputs=predictions)
```

## Stratégie #1 : *fine-tuning* total

Ici, on entraîne tout le réseau, donc il faut rendre toutes les couches "entraînables" :

python

```
1 for layer in model.layers:
2     layer.trainable = True
```

## Stratégie #2 : extraction de features

On entraîne seulement le nouveau classifieur et on ne ré-entraîne pas les autres couches :

python

```
1 for layer in model.layers:
2     layer.trainable = False
```

## Stratégie #3 : *fine-tuning* partiel

On entraîne le nouveau classifieur et les couches hautes :

python

```
1 # Ne pas entraîner les 5 premières couches (les plus basses)
2 for layer in model.layers[:5]:
3     layer.trainable = False
```

## Entraînement du réseau

Il ne reste plus qu'à compiler le nouveau modèle, puis à l'entraîner :

python

```
1 # Compiler le modèle
```



```
2 new_model.compile(loss="categorical_crossentropy", optimizer=optimizers.SGD(lr=0.0001,
momentum=0.9), metrics=["accuracy"])
3
4 # Entraîner sur les données d'entraînement (X_train, y_train)
5 model_info = new_model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, verbose=2)
6
```

☐ J'AI TERMINÉ CE CHAPITRE ET JE PASSE AU SUIVANT

**APPRENEZ À CONSTRUIRE UN CNN ET  
← GAGNEZ DU TEMPS AVEC LE TRANSFER  
LEARNING**

**QUIZ : PARTIE 3**



## Les professeurs

### Pascal Monasse

Docteur en mathématiques appliquées, chercheur en vision par ordinateur à l'École des Ponts ParisTech.

### Kimia Nadjahi

Ingénieure en Machine Learning et vision par ordinateur. Enseignante à OpenClassrooms.

**OpenClassrooms**




**Entreprises**



**En plus**



 Français ▼

