

PROJET MACHINE LEARNING



DETECTION DE LA TUMEUR CEREBRALE

Préparé par:

OUCHEN Oualae

HADDIOUI Souhayla

BOUDRAR Wafae

Encadré par:

Mme. DAOUDI Najima

REMERCIEMENT

Avant d'entamée notre rapport, un énorme merci et grand bravo à notre encadrante du projet, Professeur Najima Daoudi , pour ses efforts, pour la formation que nous avons acquise tout au long de son encadrement.

Nos remerciements également à toute les membres de ce groupe ayant contribué à l'élaboration de ce projet.



SOMMAIRE

3	Introduction
4	Transfer d'apprentissage
5	Les réseaux de neurones convolutifs (CNN)
6	Dataset
7	Pre-processing
10	VGG 16
13	Resnet
16	Exploitation des données
18	Conclusion

INTRODUCTION

Les tumeurs cérébrales sont des anomalies de croissance cellulaire dans le cerveau ou ses environs. Elles peuvent être bénignes (non cancéreuses) ou malignes (cancéreuses) et peuvent se développer à partir de différents types de cellules cérébrales. Ces tumeurs peuvent entraîner une variété de symptômes, allant des maux de tête et des changements de personnalité à des problèmes de vision et de coordination. Leur diagnostic et leur traitement peuvent être complexes et nécessitent souvent une approche multidisciplinaire impliquant des neurologues, des neurochirurgiens, des oncologues et d'autres spécialistes. Cette introduction fournira un aperçu général des tumeurs cérébrales, y compris leur classification, leurs symptômes, leurs méthodes de diagnostic et leurs options de traitement.

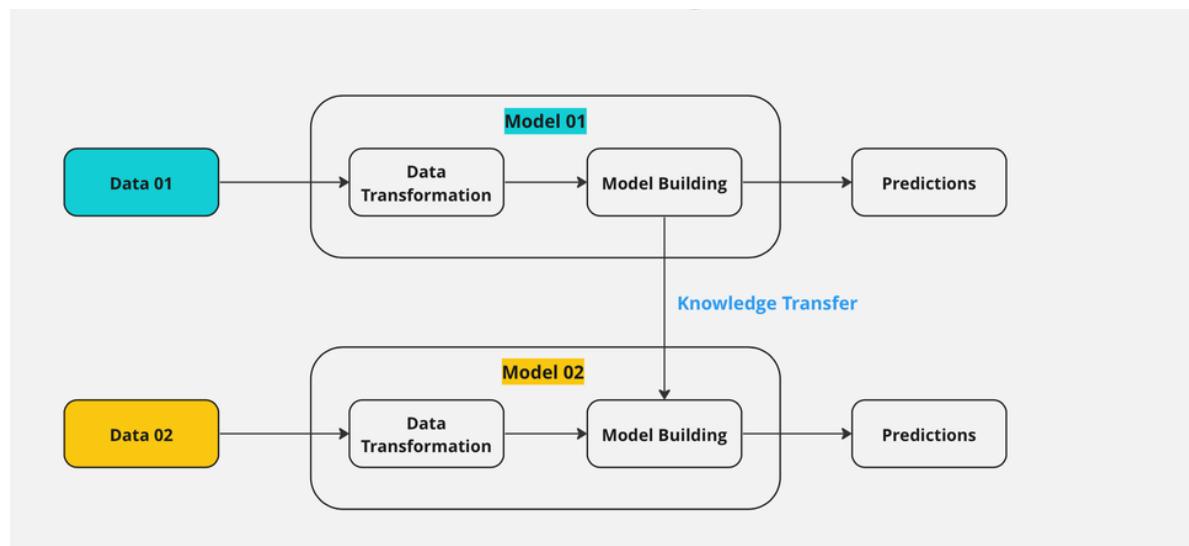
Dans notre projet, nous chercherons à déterminer, à l'aide d'une analyse d'image, la présence ou l'absence d'une tumeur cérébrale chez une personne.

TRANSFER D'APPRENTISSAGE

Le transfert d'apprentissage (ou "transfer learning" en anglais) est une technique dans le domaine de l'apprentissage automatique où un modèle pré-entraîné sur une tâche est réutilisé comme point de départ pour une nouvelle tâche similaire ou différente. Voici un résumé concis du transfert d'apprentissage :

1. Utilisation d'un modèle pré-entraîné : Un modèle déjà entraîné sur un ensemble de données massif, tel qu'ImageNet, est utilisé comme point de départ.
2. Réutilisation des poids : Les poids appris par le modèle pré-entraîné sont conservés, souvent dans les couches basses du réseau, tandis que les couches supérieures peuvent être adaptées à la nouvelle tâche.
3. Fine-tuning : Après avoir gelé les premières couches (ou toutes les couches sauf les dernières), le modèle est entraîné sur de nouvelles données spécifiques à la tâche pour ajuster les poids et améliorer les performances.
4. Avantages : Le transfert d'apprentissage permet d'économiser du temps et des ressources en tirant parti des connaissances acquises par le modèle pré-entraîné. Il est particulièrement utile lorsque les données d'entraînement pour la nouvelle tâche sont limitées.
5. Adaptabilité : Le transfert d'apprentissage peut être utilisé dans une variété de scénarios, y compris la classification d'images, la détection d'objets, la segmentation sémantique, et bien d'autres.

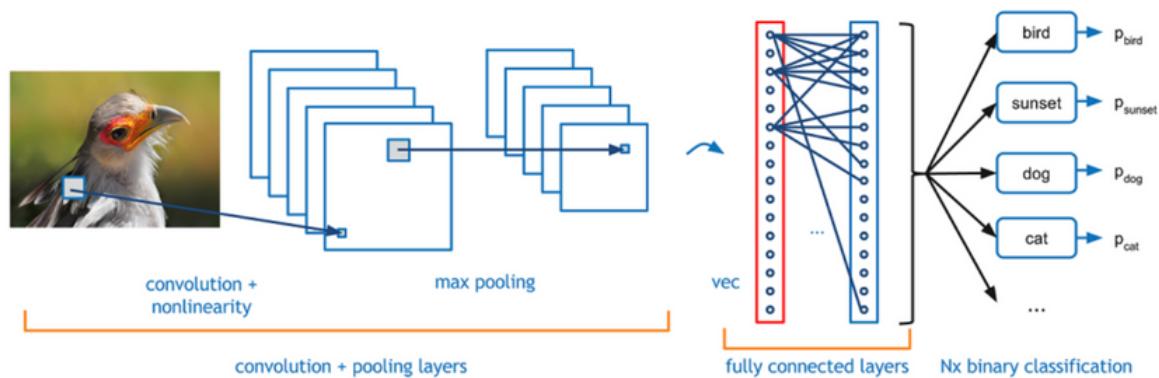
En résumé, le transfert d'apprentissage est une méthode efficace pour améliorer les performances des modèles d'apprentissage automatique sur de nouvelles tâches en réutilisant les connaissances acquises par des modèles pré-entraînés.



LES RESEAUX DE NEURONES CONVOLUTIFS

Les réseaux de neurones convolutifs (CNN), ou Convolutional Neural Networks en anglais, sont une classe de réseaux de neurones profonds conçus spécifiquement pour le traitement d'images. Voici un résumé concis des caractéristiques principales des CNN :

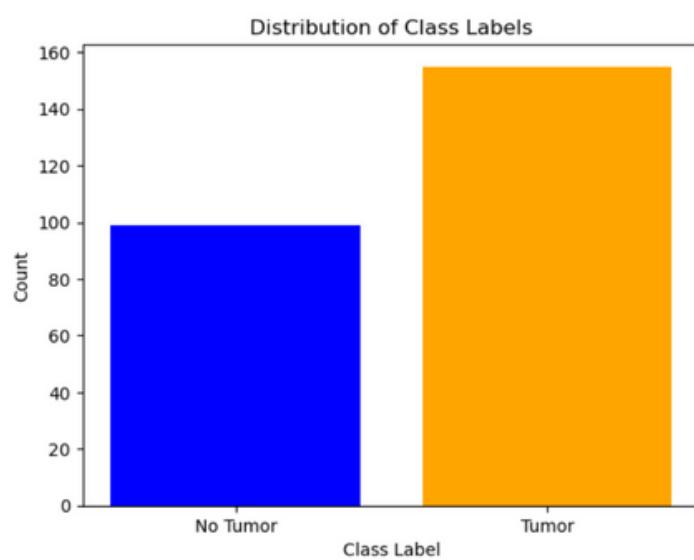
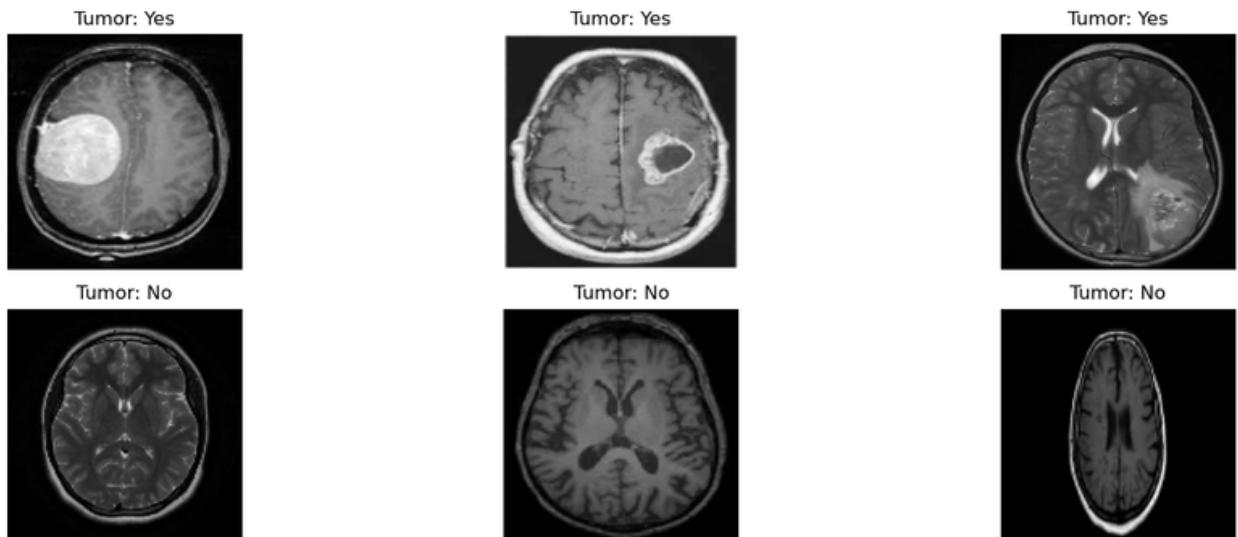
1. **Architecture** : Les CNN sont composés de différentes couches, notamment des couches de convolution, de regroupement (pooling) et des couches entièrement connectées.
2. **Convolution** : Les couches de convolution sont chargées de détecter des motifs et des caractéristiques dans l'image en appliquant des filtres sur des régions locales de l'image.
3. **Regroupement (pooling)** : Les couches de regroupement (pooling) réduisent la dimensionnalité des caractéristiques en prenant la valeur maximale ou moyenne dans des régions adjacentes.
4. **Activation** : Les fonctions d'activation telles que ReLU (Rectified Linear Unit) sont utilisées pour introduire de la non-linéarité dans le modèle.
5. **Stratégies de régularisation** : Pour éviter le surajustement, des techniques de régularisation telles que le dropout ou la régularisation L2 sont souvent utilisées.
6. **Prétraitement automatique des caractéristiques** : Les CNN sont capables d'apprendre automatiquement des caractéristiques à partir des données brutes, réduisant ainsi la dépendance à l'égard de l'extraction manuelle des caractéristiques.
7. **Utilisation** : Les CNN sont largement utilisés dans diverses applications de vision par ordinateur, notamment la classification d'images, la détection d'objets, la segmentation sémantique, etc.



DATASET

Nous avons importé notre ensemble de données depuis Kaggle : "Brain MRI for Brain Tumor Detection". Notre ensemble de données se compose de 253 images, parmi lesquelles 155 présentent une tumeur et 98 n'en présentent pas.

Voici une visualisation de quelques images, où les images présentant des tumeurs sont étiquetées "oui" et celles sans tumeur sont étiquetées "non".



PRE-PROCESSING

Le prétraitement d'images est une étape essentielle dans le traitement des données avant leur utilisation dans des applications telles que la vision par ordinateur ou l'apprentissage automatique.

1. Acquisition de données et structuration des données:

- a. Niveau de structuration: Non structurées
- b. Modèle de données: images
- c. Source: Kaggle (web)
- d. Facilité de traitement: Difficiles à traiter

2. Exploration des données

3. Transformation, normalisation & encodage:

```
output_size = (128, 128)

example_image = cv2.imread('C:\\\\Users\\\\dell\\\\Desktop\\\\archive\\\\yes\\\\Y1.jpg')
example_image = cv2.cvtColor(example_image, cv2.COLOR_BGR2RGB)

contour_image, bounding_box_image, cropped_image, resized_image = crop_brain_region(example_image, output_size)

plt.figure(figsize=(15, 10))

plt.subplot(2, 2, 1)
plt.imshow(contour_image)
plt.title("Contour")

plt.subplot(2, 2, 2)
plt.imshow(bounding_box_image)
plt.title("Bounding Box")

plt.subplot(2, 2, 3)
plt.imshow(cropped_image)
plt.title("Cropped")

plt.subplot(2, 2, 4)
plt.imshow(resized_image)
plt.title("Resized")

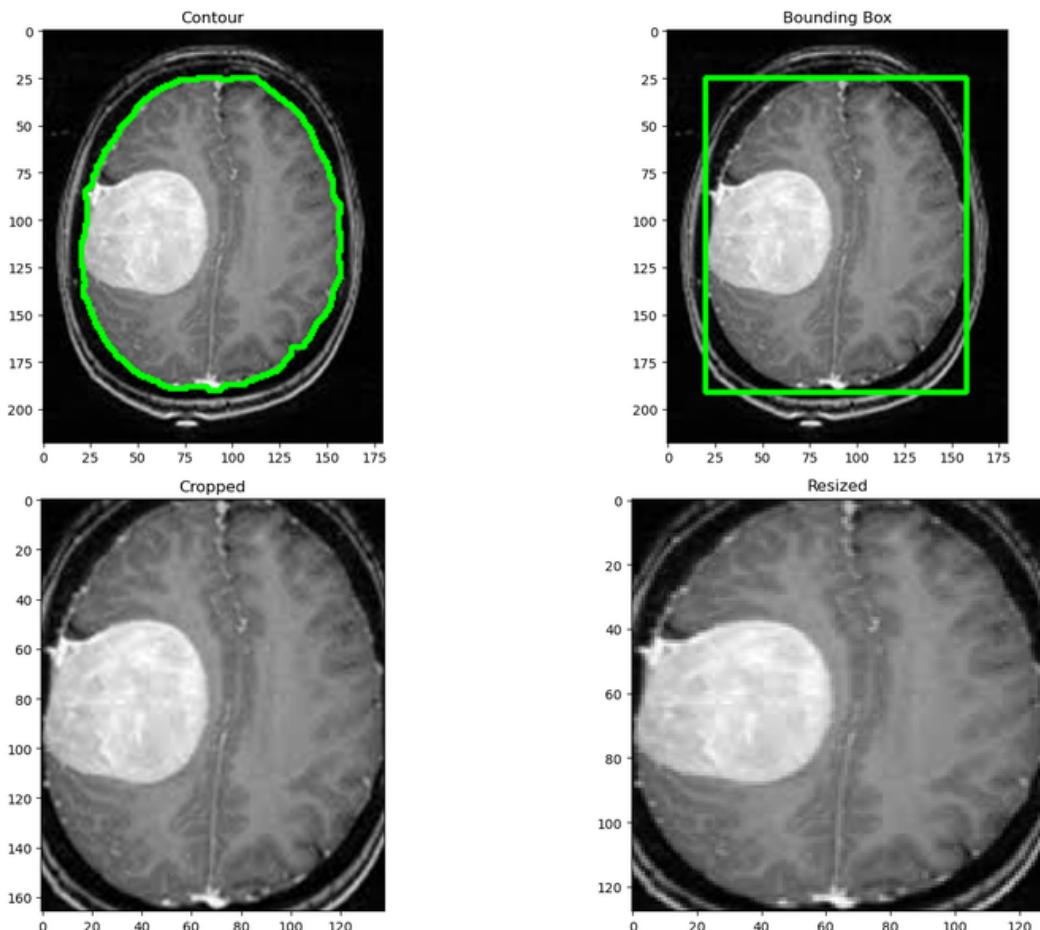
plt.tight_layout()
plt.show()

all_cropped = []

# Applying the crop function to each one of our images
for image in data:
    _, _, _, resized_image = crop_brain_region(image, output_size)
    all_cropped.append(resized_image)
```

Résultat:

PRE-PROCESSING



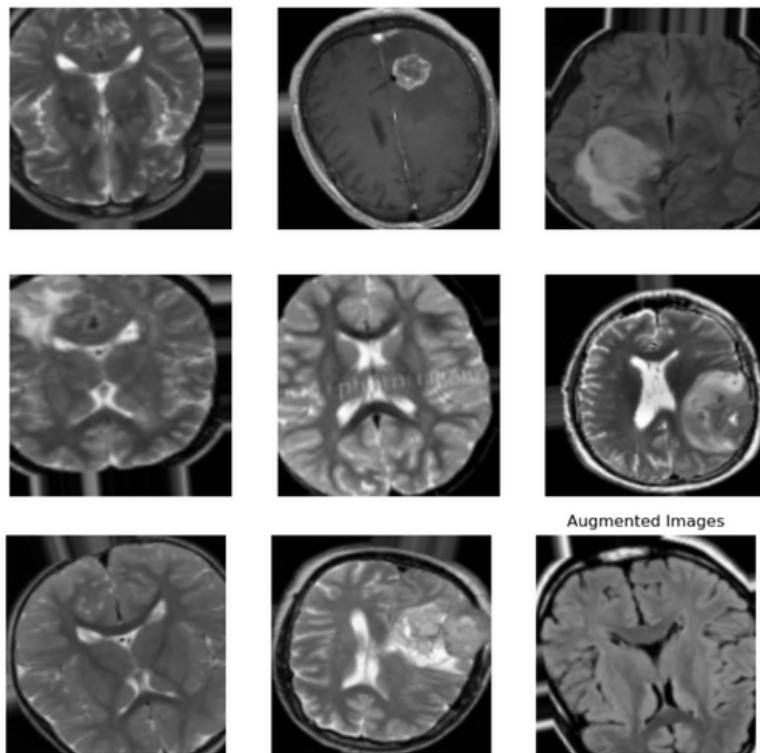
```
print("Train data shape:", train_images.shape)
print("Train labels shape:", train_labels.shape)
print("Validation data shape:", val_images.shape)
print("Validation labels shape:", val_labels.shape)
print("Test data shape:", test_images.shape)
print("Test labels shape:", test_labels.shape)
```

```
Train data shape: (203, 128, 128, 3)
Train labels shape: (203,)
Validation data shape: (26, 128, 128, 3)
Validation labels shape: (26,)
Test data shape: (25, 128, 128, 3)
Test labels shape: (25,)
```

PRE-PROCESSING

4.Agrégation et enrichissement des données: à cet égard nous avons fait une augmentation de la data

```
|: from tensorflow.keras.preprocessing.image import ImageDataGenerator  
  
train_datagen = ImageDataGenerator(  
    rotation_range=20,  
    horizontal_flip=True,  
    vertical_flip=True,  
    width_shift_range=0.1,  
    height_shift_range=0.1,  
    shear_range=0.1,  
    zoom_range=0.1,  
)  
  
val_datagen = ImageDataGenerator()  
  
train_generator = train_datagen.flow(  
    train_images, train_labels,  
    batch_size=32  
)  
  
val_generator = val_datagen.flow(  
    val_images, val_labels,  
    batch_size=32  
)
```



Augmented Images

VGG 16

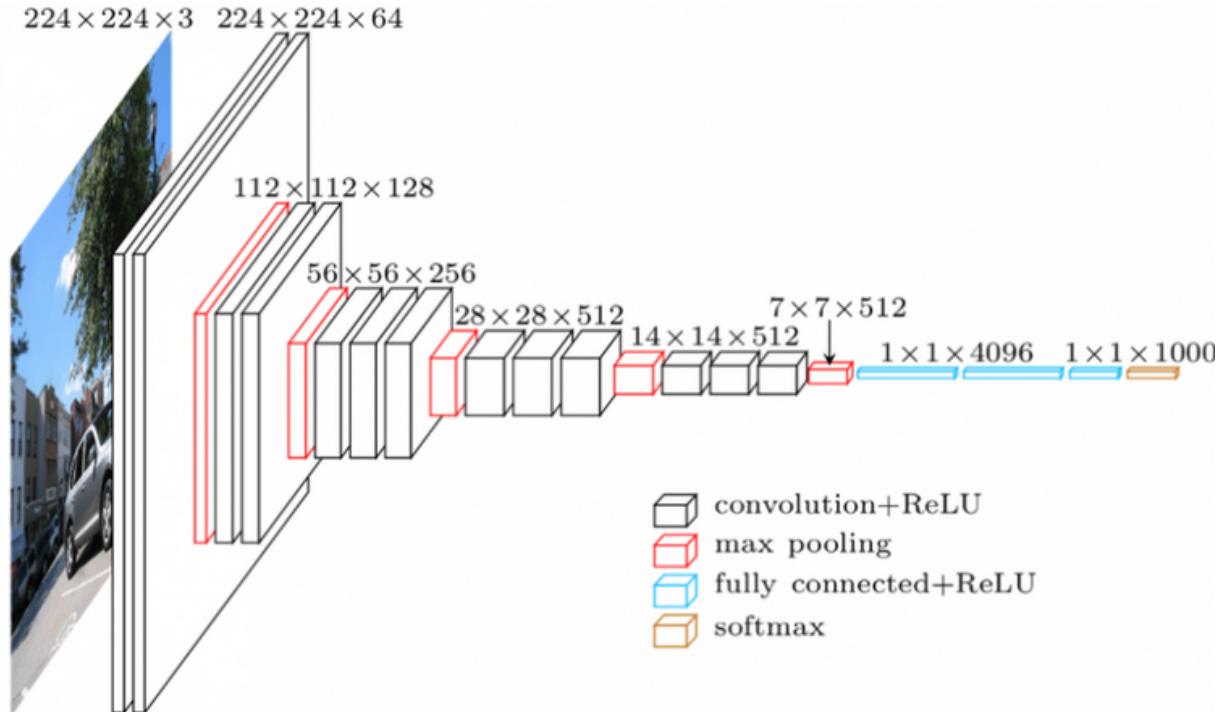
VGG-16 est un modèle de réseau de neurones convolutifs (CNN) profond largement utilisé en vision par ordinateur pour des tâches telles que la classification d'images. Voici un résumé des principales caractéristiques de VGG-16 :

1. Architecture : VGG-16 est composé de 16 couches de convolution et de regroupement (pooling), suivies de trois couches entièrement connectées (fully connected) pour la classification. Les couches de convolution utilisent des filtres de petite taille (3×3) avec un pas (stride) de 1 et un remplissage (padding) pour maintenir la taille de l'image en entrée.
2. Profondeur : Le nombre total de couches, y compris les couches de convolution, de regroupement et entièrement connectées, est de 16, d'où le nom "VGG-16".
3. Taille des filtres : Tous les filtres de convolution utilisés dans VGG-16 ont une taille fixe de 3×3 pixels.
4. Nombre de canaux : Les filtres de convolution ont un nombre de canaux qui correspond au nombre de canaux de l'image d'entrée (généralement 3 pour les images RGB).
5. Regroupement : Après chaque ensemble de couches de convolution, une couche de regroupement (pooling) est utilisée pour réduire la taille spatiale de la carte des caractéristiques.
6. Activation : Des fonctions d'activation ReLU (Rectified Linear Unit) sont utilisées après chaque couche de convolution pour introduire de la non-linéarité dans le modèle.
7. Classification : Les couches entièrement connectées suivent les couches de convolution et de regroupement, culminant avec une couche softmax pour la classification en sortie.
8. Pré-entraînement : VGG-16 peut être pré-entraîné sur de grandes bases de données d'images, telles qu'ImageNet, pour capturer des caractéristiques générales des images. Ensuite, il peut être finement ajusté sur des ensembles de données spécifiques pour des tâches particulières.

En raison de sa simplicité et de son architecture profonde, VGG-16 est souvent utilisé comme modèle de base dans le domaine de la vision par ordinateur, bien que sa taille et son nombre de paramètres le rendent relativement coûteux en termes de calcul et de mémoire.

.

VGG 16



CONSTRUCTION DU MODEL:

```
# Charger le modèle VGG16 pré-entraîné sans la couche Dense supérieure, en spécifiant la forme d'entrée
base_model = VGG16(weights='/kaggle/input/keras-pretrained-models/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5',
                     include_top=False, input_shape=(128, 128, 3))

# Geler les poids des couches du modèle de base pour empêcher leur mise à jour pendant l'entraînement
base_model.layers[0].trainable = False

# Ajouter une couche de Global Average Pooling pour réduire les dimensions de sortie
x = GlobalAveragePooling2D()(base_model.output)

# Ajouter une couche Dense avec 128 neurones et une fonction d'activation 'relu', avec une régularisation L2
x = Dense(128, activation='relu', kernel_regularizer=regularizers.l2(0.001))(x)

# Ajouter une couche de Dropout pour régulariser le modèle et prévenir le surapprentissage
x = Dropout(0.5)(x)

# Ajouter la couche de sortie avec une seule unité et une fonction d'activation 'sigmoid' pour la classification binaire
predictions = Dense(1, activation='sigmoid')(x)

# Créer le modèle final en spécifiant les entrées et les sorties
model = Model(inputs=base_model.input, outputs=predictions)

# Compiler le modèle en spécifiant la fonction de perte, l'optimiseur et les métriques à surveiller
model.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=1e-5), metrics=['accuracy'])
```

VGG 16

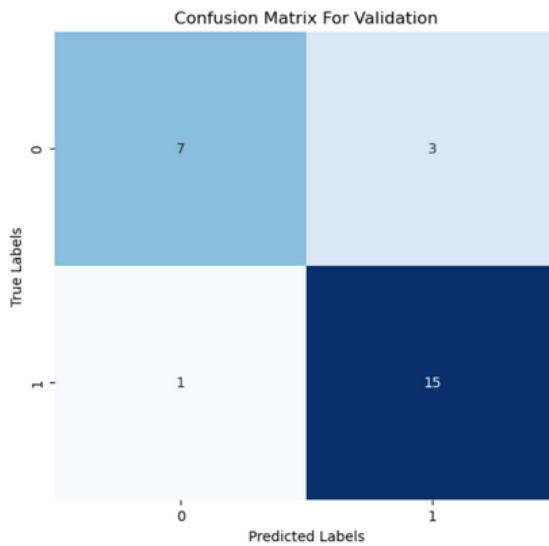
```
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Définition du callback pour réduire le taux d'apprentissage lorsque la perte de validation cesse
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=5, min_lr=1e-7)

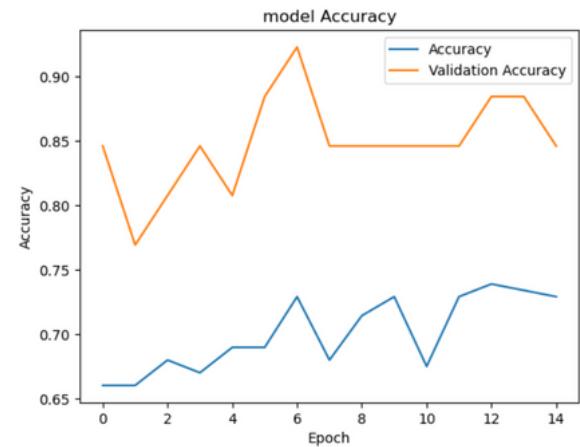
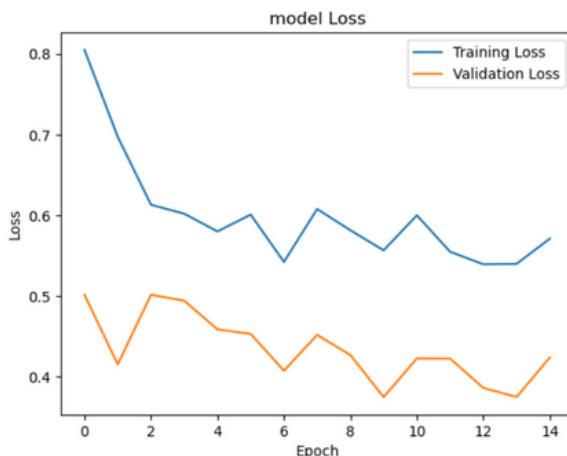
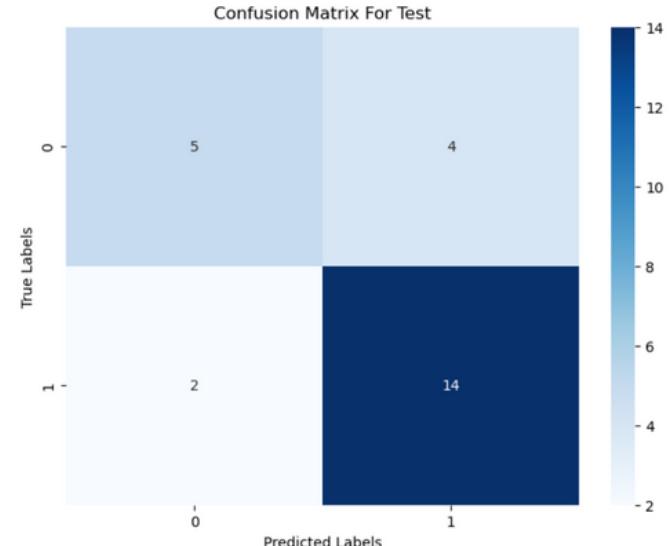
# Entraînement du modèle sur les données d'entraînement et de validation pour un nombre d'époques
history = model.fit(
    train_generator,
    epochs=200,
    validation_data=val_generator,
    callbacks=[early_stopping, reduce_lr]
)
```

Résultats:

Accuracy on Validation Set: 0.846154



Accuracy on Test Set: 0.760000



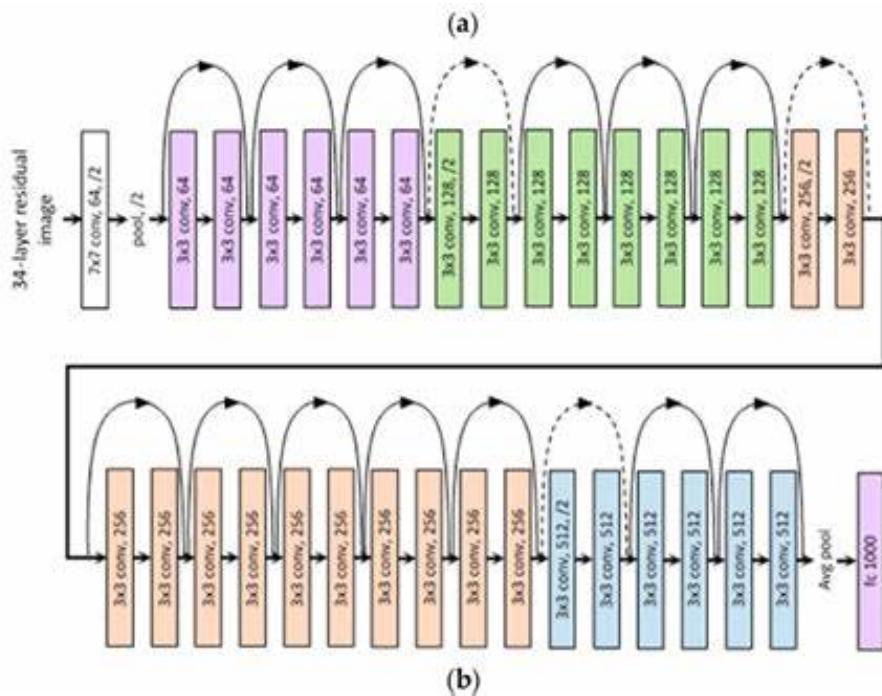
RESNET

ResNet, ou Réseaux Résiduels, est une architecture de réseau de neurones profonds qui a introduit le concept de "skip connections" ou connexions résiduelles. Voici un résumé concis des principales caractéristiques de ResNet :

1. Architecture : ResNet est composé de nombreuses couches empilées, avec des blocs résiduels qui contiennent des connexions "skip" pour sauter des couches.
2. Connexions résiduelles : Les connexions résiduelles permettent de sauter certaines couches, ce qui facilite l'apprentissage en évitant le problème de la disparition du gradient et en facilitant la propagation de l'information.
3. Blocs résiduels : Les blocs résiduels sont constitués de deux couches de convolution, entourées de connexions résiduelles. Ces blocs sont répétés plusieurs fois dans l'architecture du réseau.
4. Activation : ResNet utilise des activations ReLU après chaque couche de convolution pour introduire de la non-linéarité dans le modèle.
5. Pooling : Des couches de regroupement (pooling) sont utilisées pour réduire la taille spatiale des caractéristiques, ce qui aide à réduire la complexité du modèle.
6. Taille des filtres : ResNet utilise principalement des filtres de taille 3x3 dans ses couches de convolution.
7. Pré-entraînement : ResNet peut être pré-entraîné sur des ensembles de données massifs tels qu'ImageNet, puis finement ajusté pour des tâches spécifiques.

Grâce à ses connexions résiduelles, ResNet a permis de former des réseaux beaucoup plus profonds tout en évitant les problèmes de rétropropagation du gradient. Cela a conduit à des performances supérieures dans diverses tâches de vision par ordinateur, telles que la classification d'images et la détection d'objets.

RESNET



CONSTRUCTION DU MODEL:

```

from tensorflow.keras.applications import ResNet50, ResNet101, ResNet152
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import Adam

base_model = ResNet152(weights='imagenet', include_top=False, input_shape=(224,224,3))

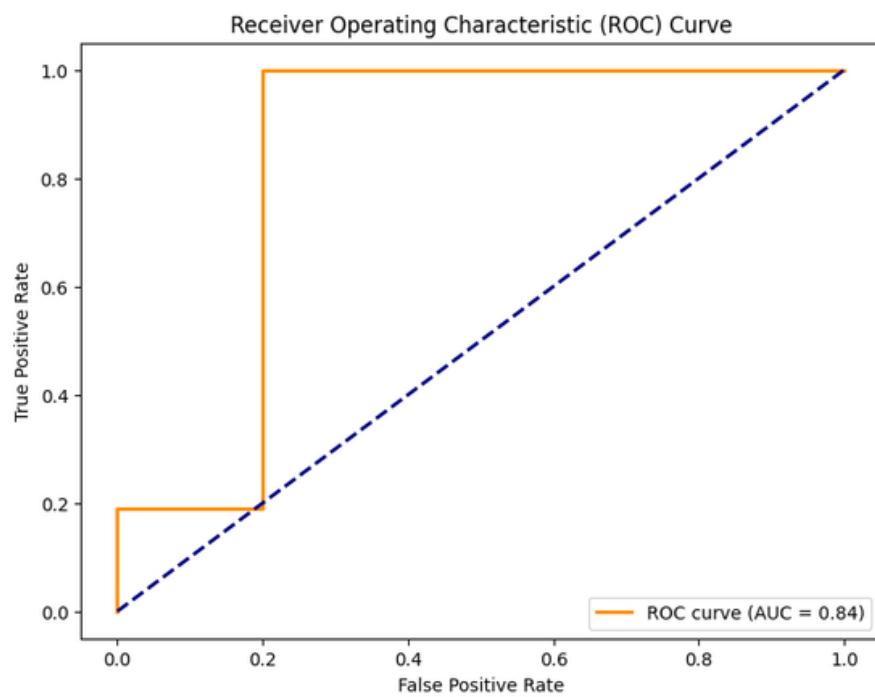
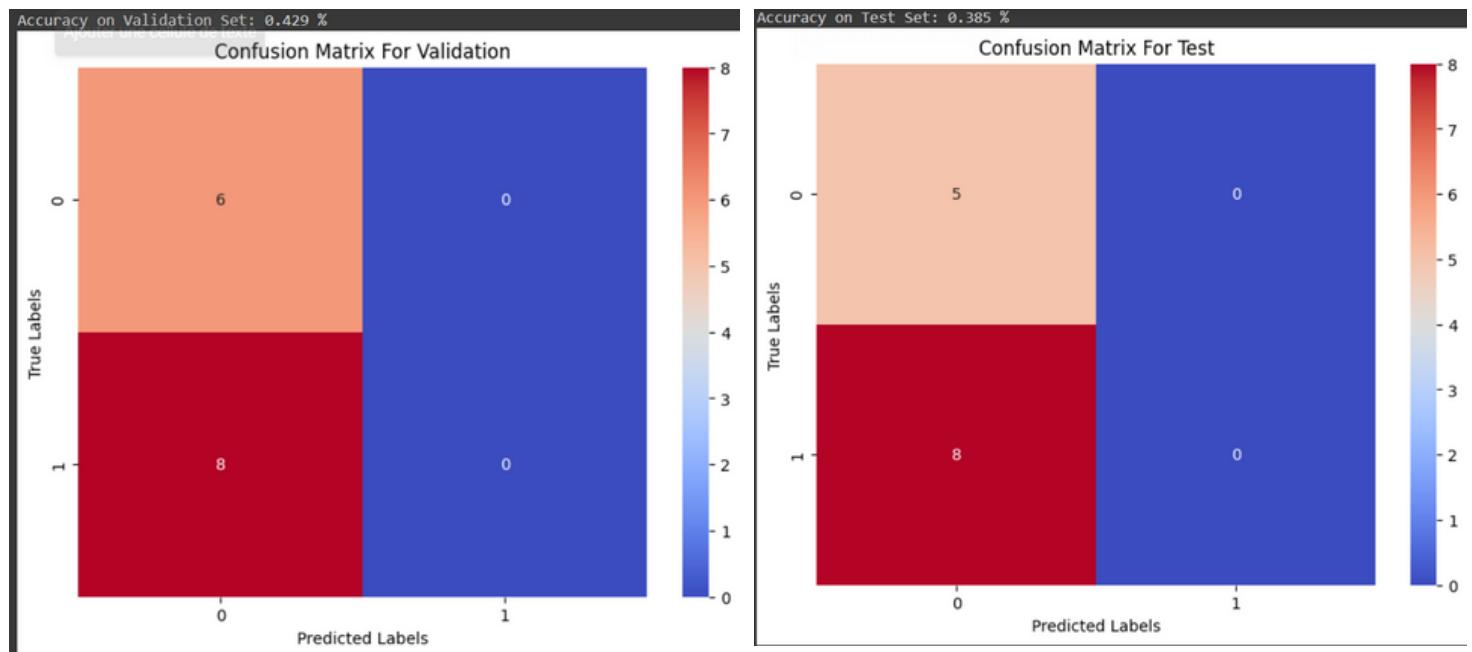
# Customize the top classification layers
x = base_model.output
x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dense(256, activation='relu')(x)
predictions = Dense(1, activation='sigmoid')(x)

model = Model(inputs=base_model.input, outputs=predictions)
model.compile(optimizer="adam",
              loss="binary_crossentropy",
              metrics=["accuracy"])

```

RESNET

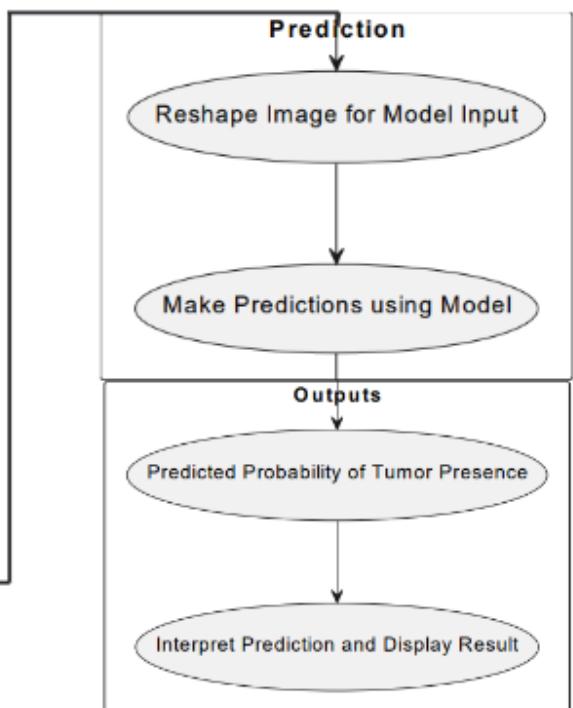
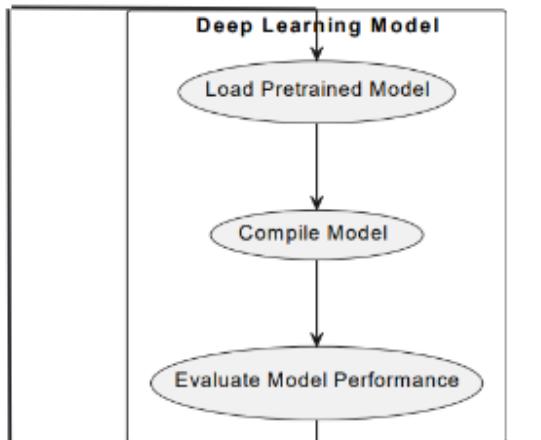
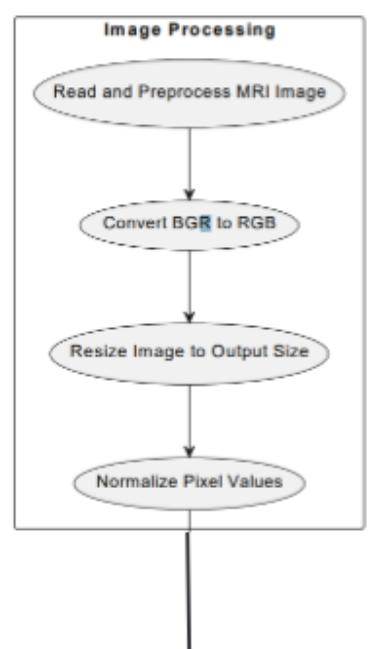
Résultat:



EXPLOITATION DES DONNEES

ANALYSE CONCEPTUELLE

MRI Image Analysis Process



EXPLOITATION DES DONNEES

Après avoir comparé les deux modèles utilisés, nous avons décidé d'exploiter les données en utilisant le modèle VGG 16, car il a démontré des performances supérieures en termes d'exactitude (accuracy)

On commence par l'importation des bibliothèques nécessaires:

```
: import os
from tensorflow.keras.models import save_model

model = model| 

# Define the path where you want to save the model
save_path = os.path.join(r'C:\Users\dell\Desktop\ML', 'model.h5')

# Save the model to the specified path
save_model(model, save_path)
```

```
import cv2
import numpy as np
from tensorflow.keras.models import load_model
# Fonction pour prétraiter l'image IRM
def preprocess_image(image_path, output_size):
    # Lire l'image depuis le chemin spécifié
    img = cv2.imread(image_path)
    # Convertir l'image de l'espace de couleur BGR à RGB
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    # Redimensionner l'image à la taille de sortie spécifiée
    img = cv2.resize(img, output_size)
    # Normaliser les valeurs de pixels pour les ramener entre 0 et 1
    img = img / 255.0
    return img

# Chemin de l'image IRM que vous souhaitez tester
image_path = 'Y4.jpg'

# Prétraiter l'image IRM
preprocessed_image = preprocess_image(image_path, output_size=(128, 128))

# Remodeler l'image pour correspondre à la forme d'entrée attendue par le modèle
input_image = np.expand_dims(preprocessed_image, axis=0)

# Charger le modèle entraîné
model_path = 'C:\\\\Users\\\\dell\\\\Desktop\\\\ML\\\\model.h5'
model = load_model(model_path)

# Compiler le modèle avec les métriques appropriées
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

EXPLOITATION DES DONNEES

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Faire des prédictions
prediction = model.predict(input_image)

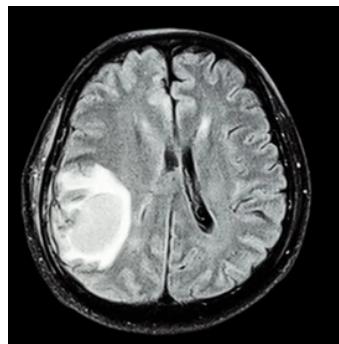
# Obtenir la probabilité prédite
predicted_probability = prediction[0]

# Afficher la probabilité
print("Probabilité de tumeur:", predicted_probability)

# Interpréter la prédition
if prediction[0] >= 0.5:
    print("L'image IRM n'indique pas la présence d'une tumeur.")
else:
    print("L'image IRM indique la présence d'une tumeur.")
```

Premier test:

Image utilisé:

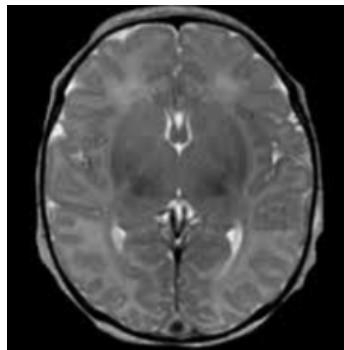


Résultats:

```
1/1 ————— 0s 164ms/step
Probabilité de tumeur: [0.11930704]
L'image IRM indique la présence d'une tumeur.
```

Deuxième test:

Image utilisé:



Résultats:

```
1/1 ————— 0s 124ms/step
Probabilité de tumeur: [0.8961098]
L'image IRM n'indique pas la présence d'une tumeur.
```

CONCLUSION

A la fin de ce rapport, on avoue que la réalisation de ce travail était une expérience assez enrichissante, Ce projet nous a permis d'approfondir nos connaissances théoriques.

