Détection des maladies rénales

Introduction

L’objectif principal de ce TP est de mettre en œuvre et d’améliorer un réseau de neurones convolutif (CNN) pour la classification d'images en utilisant des architectures similaires à LeNet5. Nous explorerons différentes variations de l’architecture, ainsi que l'impact de certains hyperparamètres et optimisateurs sur la performance du modèle.

L'ensemble de données utilisé dans ce projet est dédié à la **détection de diverses maladies rénales**. L'objectif principal de l'ensemble de données est de fournir un ensemble complet de caractéristiques permettant la classification des maladies rénales sur la base de données **d'imagerie médicale ou de diagnostic**. L'ensemble de données est de **3958** et est divisé en **quatre classes distinctes**, chacune représentant un type différent d'**affection rénale** :

1. **Kyste (classe 0)** : Les kystes sont des poches remplies de liquide qui peuvent se développer dans les reins. Bien que souvent bénins, ils peuvent parfois provoquer une gêne ou des complications en fonction de leur taille ou de leur nombre.
2. **Normal (classe 1)** : Cette classe correspond à des reins sains ne présentant aucun signe de maladie ou d'anomalie.
3. **Cailloux (classe 2)** : Les calculs rénaux sont des dépôts durs qui se forment dans les reins. Ils peuvent entraîner des douleurs, des infections ou d'autres problèmes graves s'ils ne sont pas pris en charge correctement.
4. **Tumeur (classe 3)** : Les tumeurs rénales sont des excroissances anormales dans les reins, qui peuvent être bénignes ou malignes. Les tumeurs nécessitent souvent une détection précoce et une intervention médicale pour un meilleur pronostic.

L'ensemble de données se compose de données étiquetées qui peuvent être utilisées pour des tâches de classification par apprentissage automatique, visant spécifiquement à identifier ces quatre affections rénales. Cette tâche de classification est cruciale pour automatiser le processus de détection dans les environnements médicaux, où un diagnostic précoce peut conduire à des traitements plus efficaces et à de meilleurs résultats pour les patients.

Ma perception

Pour le travail que j’ai pu réaliser au cours de ce projet j’en suis satisfaite Du fait que j'ai pu atteindre de bon résultats qu’il soit pour l’accuracy de test qui est de
99,94% et ou l’accuracy de validation qui est de 100%

Au cours de ce travail ,j'ai pu développé 3 modèles essentielles que nous allons décortiquer séparément dans le reste de ce rapport notamment:

1. Modèle LetNet Sans Dropout
2. Modèle LetNet Avec Dropout
3. Modèle avec une architecture différente

Préparation de l’environnement de travail

Chargement du dataset:

```
import os
from PIL import Image
from sklearn.model_selection import train_test_split
import numpy as np
from keras.src.utils import to_categorical

# Specifying the directory containing the dataset and the fol
#names for each class
data_dir = 'CT-KIDNEY-DATASET-Normal-Cyst-Tumor-Stone'
categories = ['Cyst', 'Normal', 'Stone', 'Tumor']

def preprocess_images(data_dir):
    data = []
    targets = []
    print("Starting image processing...")

    for idx, category in enumerate(categories):
        category_path = os.path.join(data_dir, category)
        if not os.path.exists(category_path):
            raise FileNotFoundError(f"Folder {category_path}"


        print(f"Processing images in category: {category}")
        files = os.listdir(category_path)

        for file in files:
            file_path = os.path.join(category_path, file)
            try:
```

```
                # Open the image and resize it to 32x32
                img = Image.open(file_path).resize((32, 32))
                #Convert image to a graysacle
                grey_img=img.convert('L')
                # Normalize pixel values
                img_array = np.array(grey_img) / 255.0

                # Reshape to include a single channel (32, 32

                img_array = img_array.reshape(32, 32, 1)

                data.append(img_array)
                targets.append(idx)  # Assign the class index
                                                              #
            except Exception as error:
                print(f"Failed to process {file_path}"

                continue

    return np.array(data), np.array(targets)

# Load and preprocess the dataset
images, labels = preprocess_images(data_dir)

# Display dataset details
print(f"Number of images: {images.shape[0]}")
print(f"Image dimensions: {images.shape[1:]}")
print(f"Number of labels: {len(labels)}")

# Convert labels to one-hot encoded format
labels = to_categorical(labels, num_classes=len(categories))

# Split the data into training, validation, and test subsets
X_train, X_intermediate, y_train, y_intermediate =
                                                    train
                        labels, test_size=0.3, random_state=4
X_val, X_test, y_val, y_test = train_test_split(X_intermediat
                                ,y_intermediate,size=0.5, ran
```

```
# Display the sizes of the splits
print("Dataset partitioning:")
print(f"Training data: {X_train.shape}, {y_train.shape}")
print(f"Validation data: {X_val.shape}, {y_val.shape}")
print(f"Testing data: {X_test.shape}, {y_test.shape}")
```

```
Processing images in category: Cyst
Processing images in category: Normal
Processing images in category: Stone
Processing images in category: Tumor
Number of images: 3958
Image dimensions: (32, 32, 1)
Number of labels: 3958
Dataset partitioning:
Training data: (2770, 32, 32, 1), (2770, 4)
Validation data: (594, 32, 32, 1), (594, 4)
Testing data: (594, 32, 32, 1), (594, 4)
```

On va préciser le dossier dans lequel se trouve les données qu'on va utilisées:

```
train_dir = Path('CT-KIDNEY-DATASET-Normal-Cyst"
```

On va trouver l'URL des dossiers contenants les images des cas normaux et des cas de pneumonie:

```
# Get the path to the normal and pneumonia
#sub-directories
Normal_Cases_dir = train_dir / 'Normal'
Cyst_Cases_dir = train_dir / 'Cyst'
Stone_Cases_dir = train_dir / 'Stone'
Tumor_Cases_dir = train_dir / 'Tumor'
```

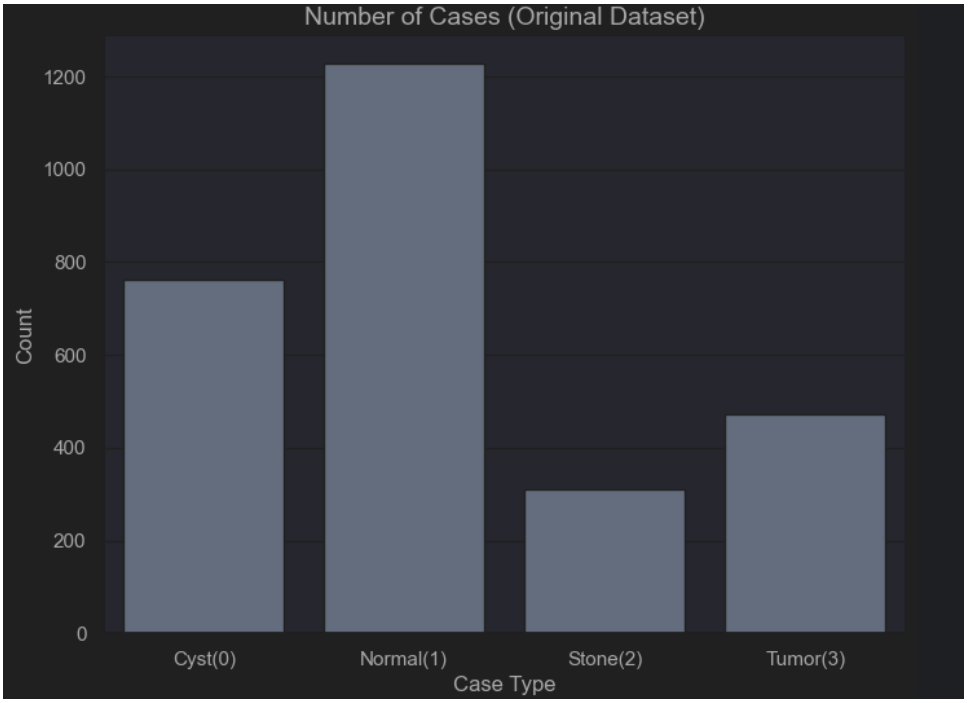
On va maintenant récupérer toutes les images:

```
# Getting the list of all the images
Normal_Cases = Normal_Cases_dir.glob('*.jpg')
Cyst_Cases = Cyst_Cases_dir.glob('*.jpg')
```

```
Stone_Cases = Stone_Cases_dir.glob('*.jpg')
Tumor_Cases = Tumor_Cases_dir.glob('*.jpg')
```

Afficher les données du Dataset d’origine:

```
# Plotting the Graph for the original dataset
plt.figure(figsize=(8, 6))  # Set the size of the graph
sns.barplot(x=list(cases_count_original.keys()),
                        y=list(cases_count_original.values())
plt.title('Number of Cases (Original Dataset)'
                                                          ,
plt.xlabel('Case Type', fontsize=12)
plt.ylabel('Count', fontsize=12)
plt.xticks(range(len(cases_count_original.keys())),
                   ['Cyst(0)', 'Normal(1)', 'Stone(2)'

plt.show()
```



Affichage le nombre d'images par catégorie:

```
import os
from collections import defaultdict
```

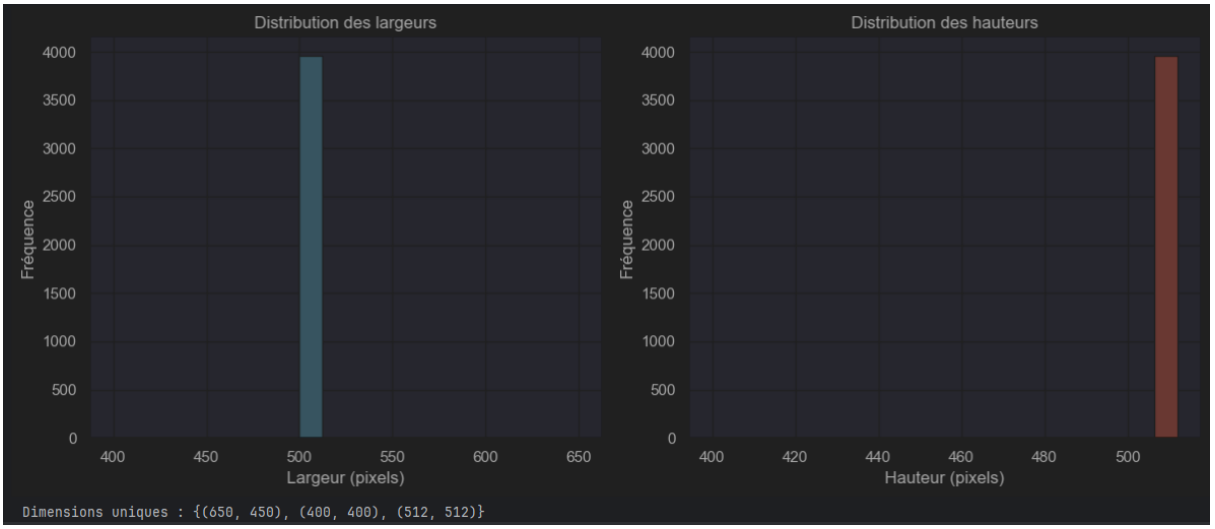
```
from PIL import Image
import matplotlib.pyplot as plt

# Répertoire racine
root_dir = './CT-KIDNEY-DATASET-Normal-Cyst-Tumor-Stone'

# Initialiser un dictionnaire pour les chemins
# d'images par catégorie
image_paths = defaultdict(list)

# Parcourir les sous-dossiers et
#charger les fichiers d'images
for category in os.listdir(root_dir):
    category_path = os.path.join(root_dir, category)
     # Vérifier si c'est un dossier
    if os.path.isdir(category_path):
        for file_name in os.listdir(category_path):
            file_path = os.path.join(category_path

                        # Vérifier si c'est un fichier
            if os.path.isfile(file_path):
                image_paths[category].append(file_path)

# Afficher le nombre d'images par catégorie
for category, paths in image_paths.items():
    print(f"{category}: {len(paths)} images")
```

```
Cyst: 1124 images
Normal: 1738 images
Stone: 425 images
Tumor: 671 images
```

Vérification les dimensions des images:

```
# Vérifier les dimensions des images
image_shapes = []
```

```
for category, paths in image_paths.items():
    for path in paths:
        try:
            with Image.open(path) as img:
                image_shapes.append(img.size)
        except Exception as e:
            print(f"Erreur avec l'image {path}: {e}")

# Distribution des largeurs et hauteurs
if image_shapes:  # Vérifier que la liste n'est pas vide
    widths, heights = zip(*image_shapes)

    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.hist(widths, bins=20, color='skyblue')
    plt.title("Distribution des largeurs")
    plt.xlabel("Largeur (pixels)")
    plt.ylabel("Fréquence")

    plt.subplot(1, 2, 2)
    plt.hist(heights, bins=20, color='salmon')
    plt.title("Distribution des hauteurs")
    plt.xlabel("Hauteur (pixels)")
    plt.ylabel("Fréquence")

    plt.tight_layout()
    plt.show()

    print(f"Dimensions uniques : {set(image_shapes)}")
else:
    print("Aucune image chargée correctement.")
```



Vérification les formats des images:

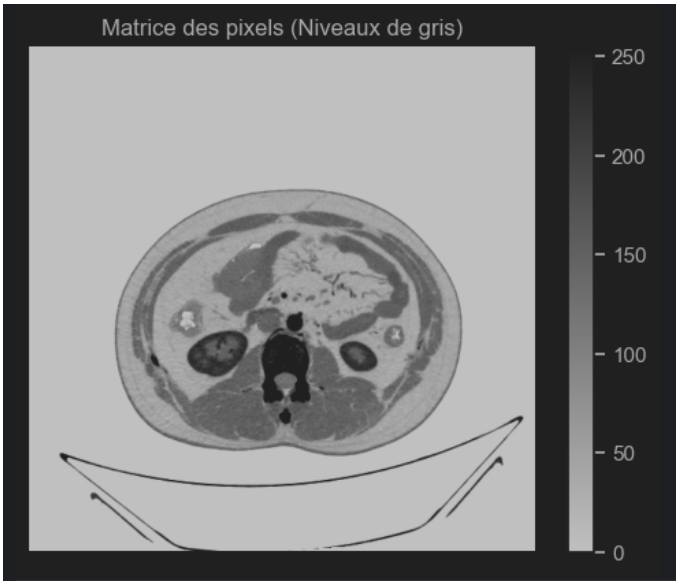
```
# Vérifier les formats des images
formats = []

for category, paths in image_paths.items():
    for path in paths:
        try:
            with Image.open(path) as img:
                formats.append(img.format)
        except Exception as e:
            print(f"Erreur avec l'image {path}: {e}")

# Compter les formats
from collections import Counter
format_counts = Counter(formats)
print("Répartition des formats d'images :"
                                          , format_coun
```

```
Répartition des formats d'images : Counter({'JPEG': 3958})
```

Visualisation des pixels d'une image (niveaux de gris):

```
import os
import numpy as np
```

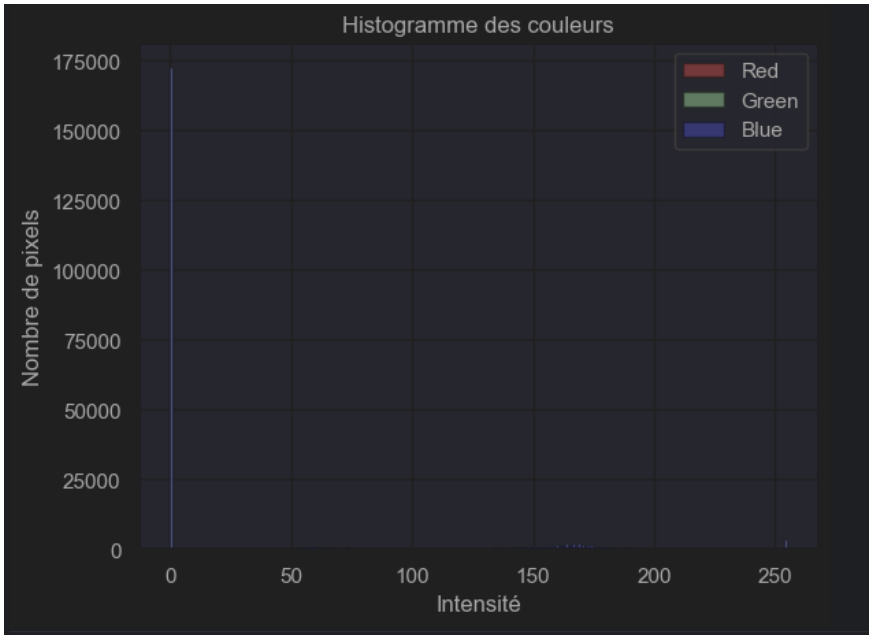
```
import matplotlib.pyplot as plt
from PIL import Image

# Exemple de chemin pour une image spécifique
example_img_path = os.path.join(root_dir,
                    'Normal', 'Normal- (980).jpg')

# Charger et afficher l'image en niveaux de gris
image = Image.open(example_img_path).convert("L")
image_array = np.array(image)

plt.imshow(image_array, cmap='gray')
plt.colorbar()
plt.title("Matrice des pixels (Niveaux de gris)")
plt.axis('off')
plt.show()
```



Histogramme des couleurs:

```
def plot_color_histogram(image_path):
    image = Image.open(image_path).convert("RGB")
    image_array = np.array(image)

    colors = ['Red', 'Green', 'Blue']
    for i, color in enumerate(colors):
```

```
        plt.hist(image_array[..., i].flatten()
                        , bins=256, alpha=0.6, label=color
                        , color=color.lower())

    plt.title("Histogramme des couleurs")
    plt.xlabel("Intensité")
    plt.ylabel("Nombre de pixels")
    plt.legend()
    plt.show()

# Exemple d'utilisation pour une image du dataset
example_img_path = os.path.join(root_dir
                    , 'Tumor', 'Tumor- (584).jpg')
plot_color_histogram(example_img_path)
```

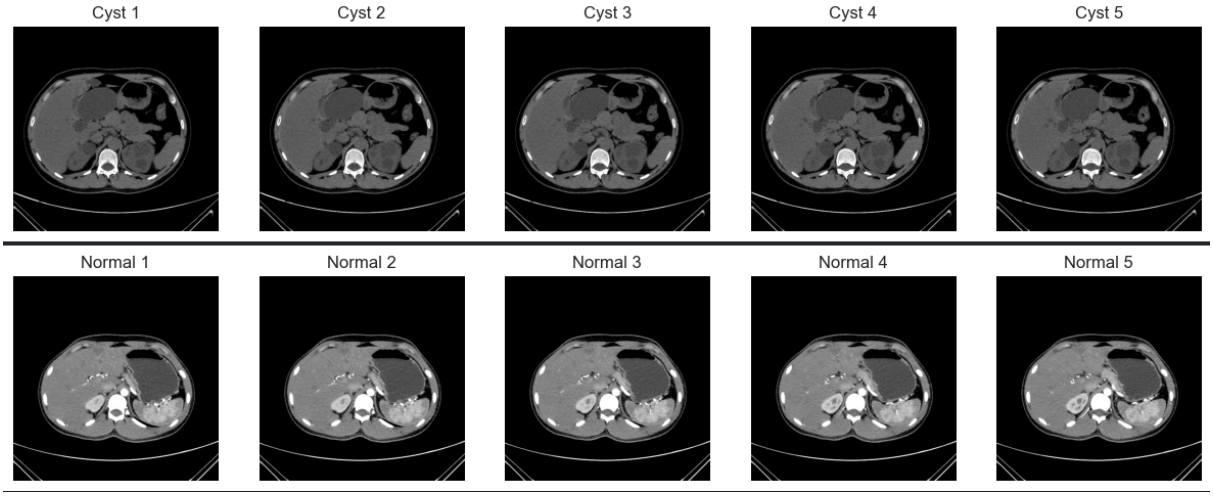


Images CT scannées et disponibles:

```
import os
import re
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Define the root directory where the images are stored
root_dir = './CT-KIDNEY-DATASET-Normal-Cyst-Tumor-Stone'
```

```
# Define the categories and their corresponding prefixes
categories = {
    'Cyst': 'Cyst-',
    'Normal': 'Normal-',
    'Stone': 'Stone-',
    'Tumor': 'Tumor-'
}

# Function to display images from a given category
def display_images_from_category(category_name, prefix):
    category_path = os.path.join(root_dir, category_name)

    # List the image files that match the category's
    #prefix pattern and have numbers in parentheses
    image_files = [f for f in os.listdir(category_path)
    if f.startswith(prefix)
                   and f.endswith(('png', 'jpg', 'jpeg'))
                   and re.search(r'\(\d+\)', f)]

    # Check if images were found
    if not image_files:
        print(f"Aucune image trouvée pour"
        "{category_name} avec le préfixe {prefix}.")
        return

    # Sort images to display in a consistent order
    #(if needed)
    image_files.sort()

    # Display a few images
    fig, axes = plt.subplots(1,5,figsize=(15, 5))
    for i, ax in enumerate(axes):
        if i < len(image_files):
            img_path = os.path.join(category_path
            , image_files[i])
            img = mpimg.imread(img_path)
            ax.imshow(img)
            ax.axis('off')  # Hide axes
```

```
            ax.set_title(f'{category_name} {i+1}')
        else:
                # Hide remaining axes if not enough images
            ax.axis('off')
    plt.show()

# Display a few images from each category
for category_name, prefix in categories.items():
    display_images_from_category(category_name
```



Vérifier le notebook joint pour voir le reste des images affichées.

Chargement des images et divisions du Dataset e données d’entrainement de validation et de test:

```
# Load the dataset images and labels
images, labels = preprocess_images(data_dir)

# Split the dataset into training, validation, and test sets
X_train, X_temp, y_train, y_temp = train_test_split(
                                                images, label

X_val, X_test, y_val, y_test = train_test_split(
                                            X_temp, y_temp, test_
```

```
# Print statistics about the splits
print(f"Total images: {images.shape[0]}")
print(f"Training set: {X_train.shape}, {y_train.shape}")
print(f"Validation set: {X_val.shape}, {y_val.shape}")
print(f"Test set: {X_test.shape}, {y_test.shape}")
```

```
Starting image processing...
Processing images in category: Cyst
Processing images in category: Normal
Processing images in category: Stone
Processing images in category: Tumor
Total images: 3958
Training set: (2770, 32, 32, 1), (2770,)
Validation set: (594, 32, 32, 1), (594,)
Test set: (594, 32, 32, 1), (594,)
```

Résolution du problème de déséquilibre du Dataset en utilisant la méthode **SMOTE**:

```
from imblearn.over_sampling import SMOTE
import numpy as np

# Check the original shape of X_train
print("Original shape of X_train:", X_train.shape)

# Ensure X_train is not empty
if X_train.size == 0:
    raise ValueError("X_train is empty! Please"
          " check your data loading pipeline.")

# Get the total number of rows in the training data
num_train_samples = len(X_train)

# Ensure the data has the correct number
#of features (not zero)
if X_train.shape[1] == 0:
    raise ValueError("X_train has no features."
    " Please check the feature extraction"
    "process.")
```

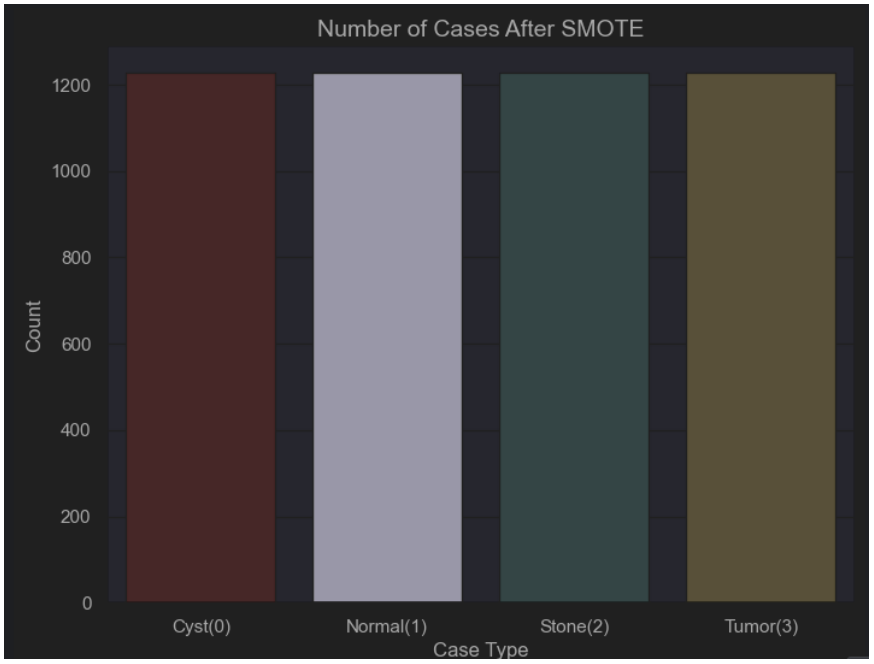
```
# Reshape the training data from 4D to
#2D (required for SMOTE)
X_train_reshaped =X_train.reshape(num_train_samples,-1)

# Check the new shape after reshaping
print("Shape after reshaping:", X_train_reshaped.shape)

# Apply SMOTE to balance the dataset
X_train_balanced, y_train_balanced =
smote.fit_resample(X_train_reshaped, y_train)

# Reshape the balanced training data back
# to its original shape (4D)
X_train_balanced = X_train_balanced.reshape(-1, 32, 32, 1)

print("Original training dataset size:"
                                    , X_train.shape[0])
print("Balanced training dataset size:"
                    , X_train_balanced.shape[0])
print("Original testing dataset size:"
                                    , y_train.shape[0])
print("Balanced testing dataset size:"
                    , y_train_balanced.shape[0])
```

```
Original shape of X_train: (2770, 32, 32, 1)
Shape after reshaping: (2770, 1024)
Original training dataset size: 2770
Balanced training dataset size: 4908
Original testing dataset size: 2770
Balanced testing dataset size: 4908
```

La Dataset après avoir été équilibrée:

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

```
# Count the values of the balanced labels
unique,counts=np.unique(y_train_balanced,return_counts=True)
cases_count_balanced = dict(zip(unique, counts))

# Define custom colors for each category
custom_colors=['#ffb5b5','#272343','#bae8e8','#f0d78c']

# Plotting the distribution of labels
plt.figure(figsize=(8, 6))
sns.barplot(x=list(cases_count_balanced.keys()),
            y=list(cases_count_balanced.values()),
            hue=list(cases_count_balanced.keys()),
            palette=custom_colors,
            legend=False)

# Set titles and labels
plt.title('Number of Cases After SMOTE',fontsize=14)
plt.xlabel('Case Type', fontsize=12)
plt.ylabel('Count', fontsize=12)

# Set x-ticks for the labels
plt.xticks(range(len(cases_count_balanced)),
['Cyst(0)', 'Normal(1)', 'Stone(2)', 'Tumor(3)'])

# Display the plot
plt.show()
```



Convertir les libellés entiers en libellés codés à chaud:

```
# Update the number of classes based on your dataset
num_classes = len(set(labels))

# One-hot encode the labels
y_train=to_categorical(y_train,num_classes=num_classes)
y_val=to_categorical(y_val,num_classes=num_classes)
y_test=to_categorical(y_test, num_classes=num_classes)

# Print shapes to verify
print(f"y_train shape: {y_train.shape}")
print(f"y_val shape: {y_val.shape}")
print(f"y_test shape: {y_test.shape}")
```

```
y_train shape: (2770, 4)
y_val shape: (594, 4)
y_test shape: (594, 4)
```

```
print("Forme des données d'images (train):"
,X_train.shape)
print("Forme des labels (train):"
, y_train.shape)
```

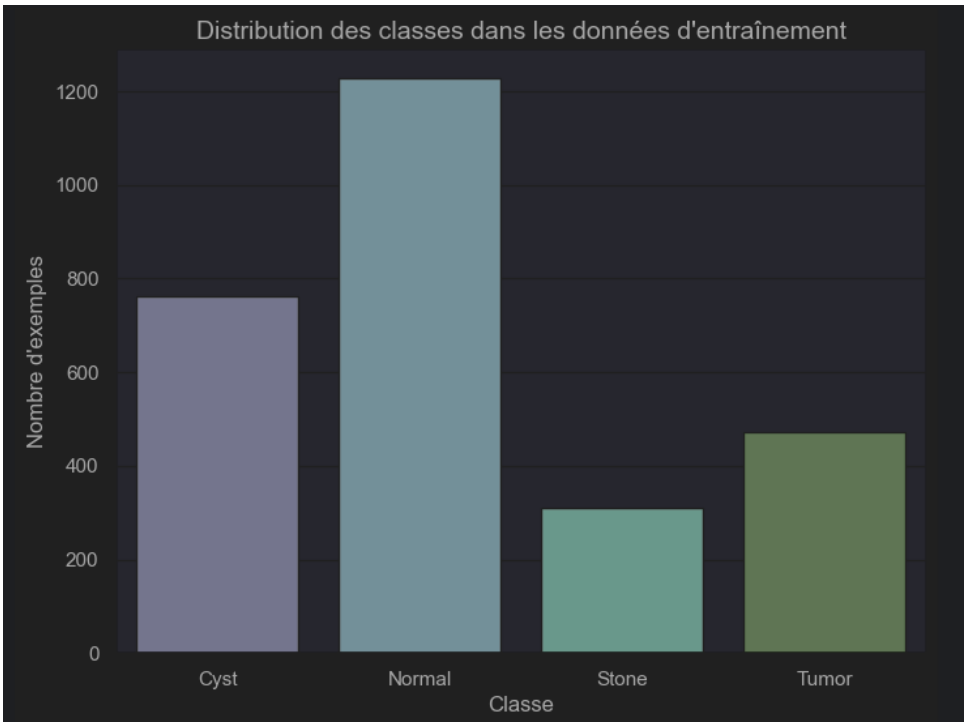
```
print("Forme des données d'images (validation):"
, X_val.shape)
print("Forme des labels (validation):"
, y_val.shape)

print("Forme des données d'images (test):"
, X_test.shape)
print("Forme des labels (test):"
, y_test.shape)
```

```
Forme des données d'images (train): (2770, 32, 32, 1)
Forme des labels (train): (2770, 4)
Forme des données d'images (validation): (594, 32, 32, 1)
Forme des labels (validation): (594, 4)
Forme des données d'images (test): (594, 32, 32, 1)
Forme des labels (test): (594, 4)
```

Visualisation de distribution des données:

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Assuming your labels and classes are in `y_train`
#, `y_val`, or `y_test`
df = pd.DataFrame({'classname':
['Cyst', 'Normal', 'Stone', 'Tumor']})
df['count'] = [sum(y_train[:, i])
for i in range(y_train.shape[1])]

sns.set_theme(style="darkgrid")
plt.figure(figsize=(8, 6))
sns.barplot(x='classname', y='count', data=df
, hue='classname', palette="viridis", dodge=False)
plt.title("Distribution des classes dans les"
" données d'entraînement", fontsize=14)
plt.xlabel("Classe", fontsize=12)
```

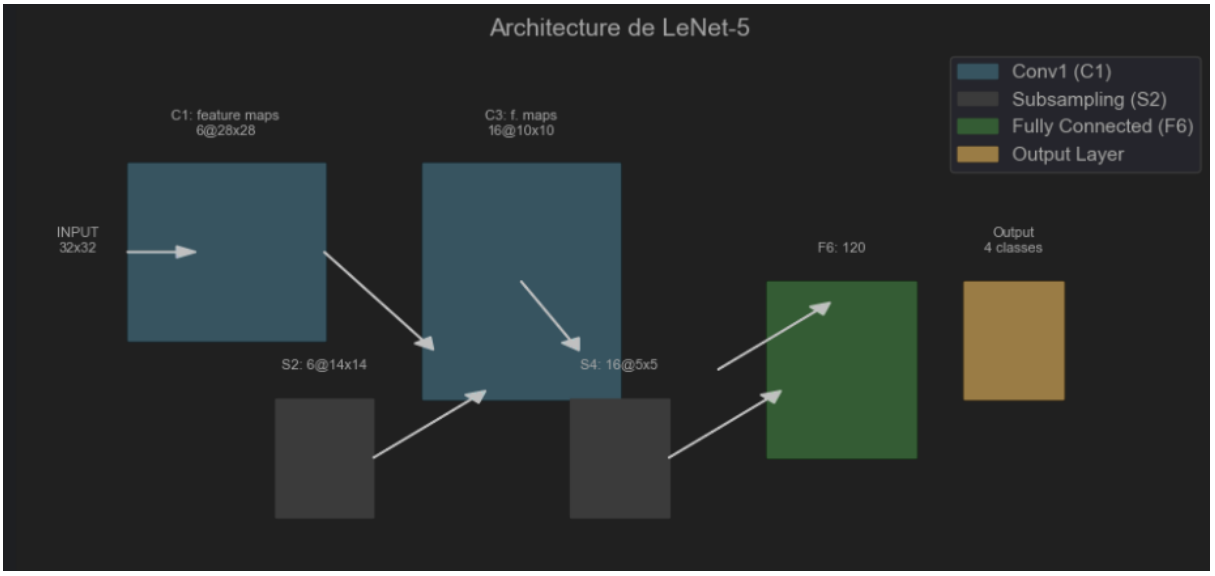
```
plt.ylabel("Nombre d'exemples", fontsize=12)
plt.legend([], [], frameon=False)
plt.show()
```



1. Implémentation de l’Architecture LeNet5

1.1. Architecture Initiale (LeNet5): Modèle1

LeNet5 est composé de plusieurs couches convolutives et de sous-échantillonnage (pooling), suivies de couches pleinement connectées. Dans ce modèle on va implémenter une architecture sans Dropout. Voici l'architecture :



1. **Entrée** : Image de taille .

   32×32×132 \times 32 \times 1

2. **Couches convolutives** :

   - **C1** : Convolution de filtres (avec fonction d’activation ReLU).

     66

     5×55 \times 5

   - **S2** : Sous-échantillonnage (pooling) (MaxPooling).

     2×22 \times 2

   - **C3** : Convolution de filtres .

     1616

     5×55 \times 5

   - **S4** : Sous-échantillonnage (MaxPooling).

     2×22 \times 2

   - **C5** : Convolution de filtres .

     120120

     5×55 \times 5

3. **Couches entièrement connectées** :

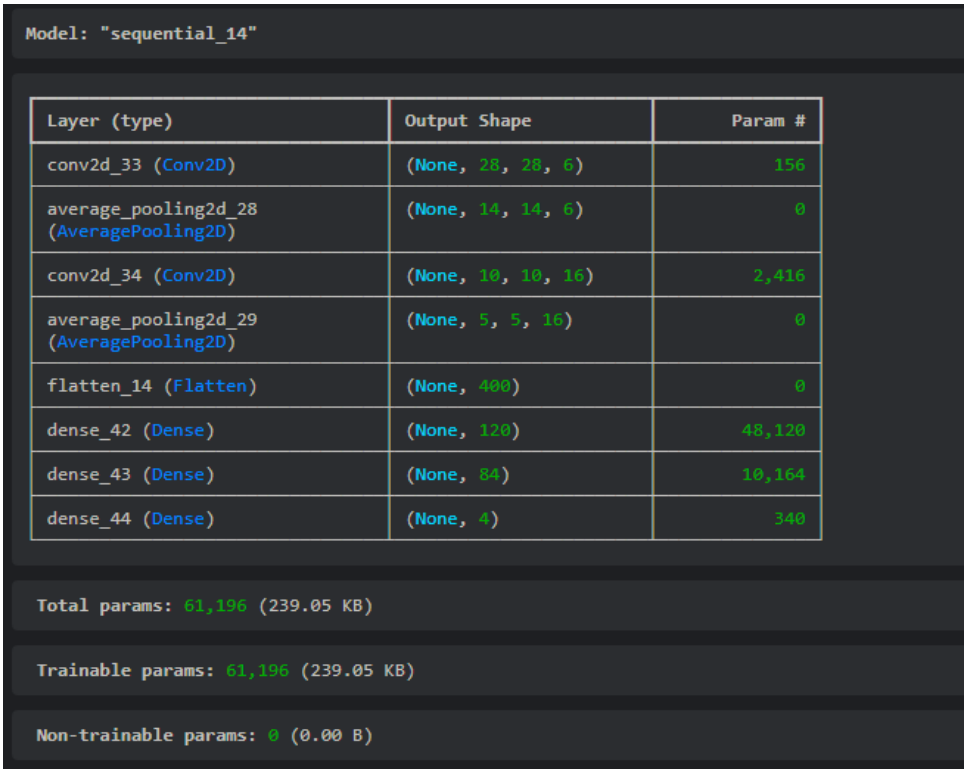
   - **F6** : 84 neurones (fonction d’activation ReLU).
   - **Sortie** : 10 neurones pour classification multi-classe (fonction softmax).

Code Python pour LeNet5 :

```
from keras.src.models import Sequential
from keras.src.layers import Conv2D, AveragePooling2D
, Flatten, Dense, InputLayer

# Définir le modèle
model1 = Sequential([
    # Première couche :
    #Définir la forme d'entrée avec InputLayer
    # Définir la forme d'entrée 32x32x1
    InputLayer(shape=(32, 32, 1)),

    # C1: Couche de convolution
    #avec 6 filtres 5x5, activation ReLU
    Conv2D(6, (5, 5), activation='relu'),

    # S2: SubSampling (AveragePooling) 2x2
    AveragePooling2D(pool_size=(2, 2)),

    # C3: Deuxième couche de convolution
    #avec 16 filtres 5x5, activation ReLU
    Conv2D(16, (5, 5), activation='relu'),

    # S4: SubSampling (AveragePooling) 2x2
    AveragePooling2D(pool_size=(2, 2)),

    # Aplatir pour passer aux couches denses
    Flatten(),

    # C5: Couche entièrement connectée
    #avec 120 neurones
    Dense(120, activation='relu'),

    # F6: Couche entièrement connectée
    #avec 84 neurones
    Dense(84, activation='relu'),
```

```
    # Couche de sortie avec 4 neurones
    #(pour 4 classes de classification)
    Dense(4, activation='softmax')
])

# Afficher le résumé du modèle
model1.summary()

# Compiler le modèle
model1.compile(optimizer='adam',
 loss='categorical_crossentropy',metrics=['accuracy'])
```

L’exécution est représentée comme suit:



Model: "sequential_14"

Layer (type)	Output Shape	Param #
conv2d_33 (Conv2D)	(None, 28, 28, 6)	156
average_pooling2d_28 (AveragePooling2D)	(None, 14, 14, 6)	0
conv2d_34 (Conv2D)	(None, 10, 10, 16)	2,416
average_pooling2d_29 (AveragePooling2D)	(None, 5, 5, 16)	0
flatten_14 (Flatten)	(None, 400)	0
dense_42 (Dense)	(None, 120)	48,120
dense_43 (Dense)	(None, 84)	10,164
dense_44 (Dense)	(None, 4)	340

Total params: 61,196 (239.05 KB)

Trainable params: 61,196 (239.05 KB)

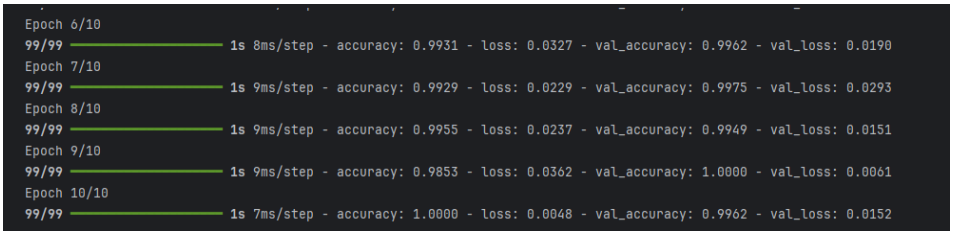
Non-trainable params: 0 (0.00 B)

Pour l’entrainement du model ce code à été écrit:

```
# Convert labels to categorical (one-hot encoding)
y_train = to_categorical(y_train, num_classes=len(categories)
y_val = to_categorical(y_val, num_classes=len(categories))

# Train the model
```

```
history1 = model1.fit(
    X_train,
    y_train,
    validation_data=(X_val, y_val),
    epochs=10,
    batch_size=32
)
```

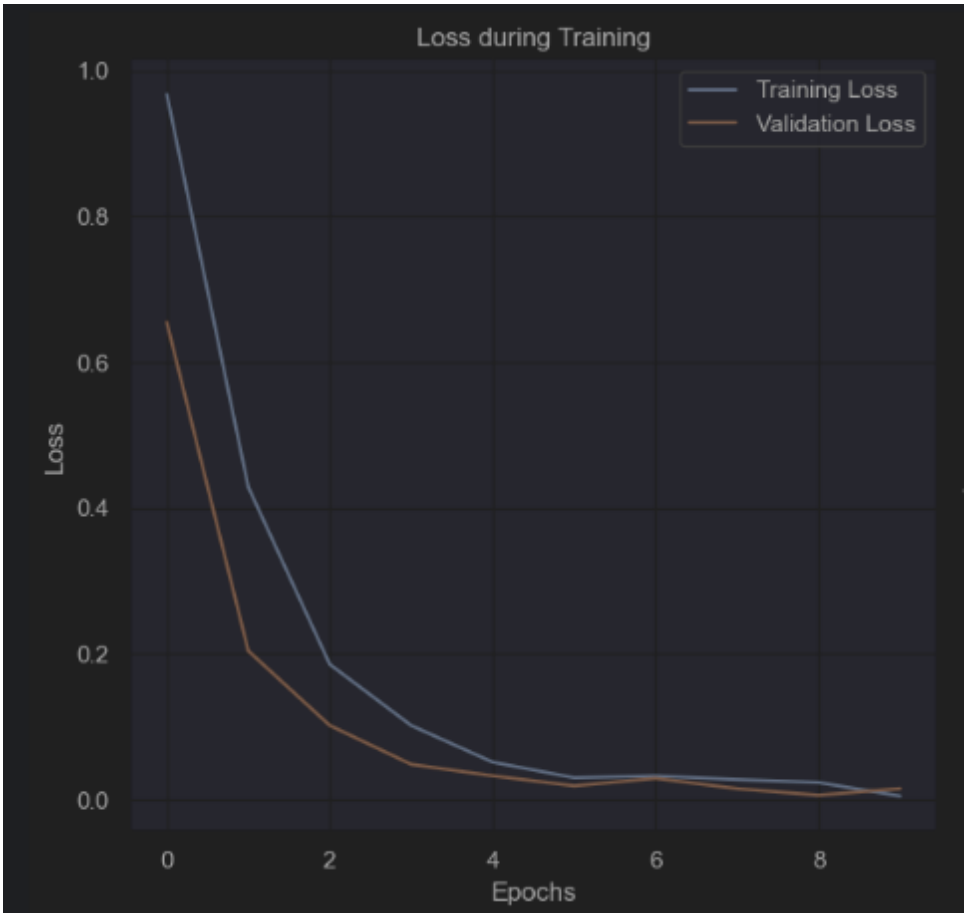
L’entrainement à donné 100% d’auccuracy et voilà le résultat:

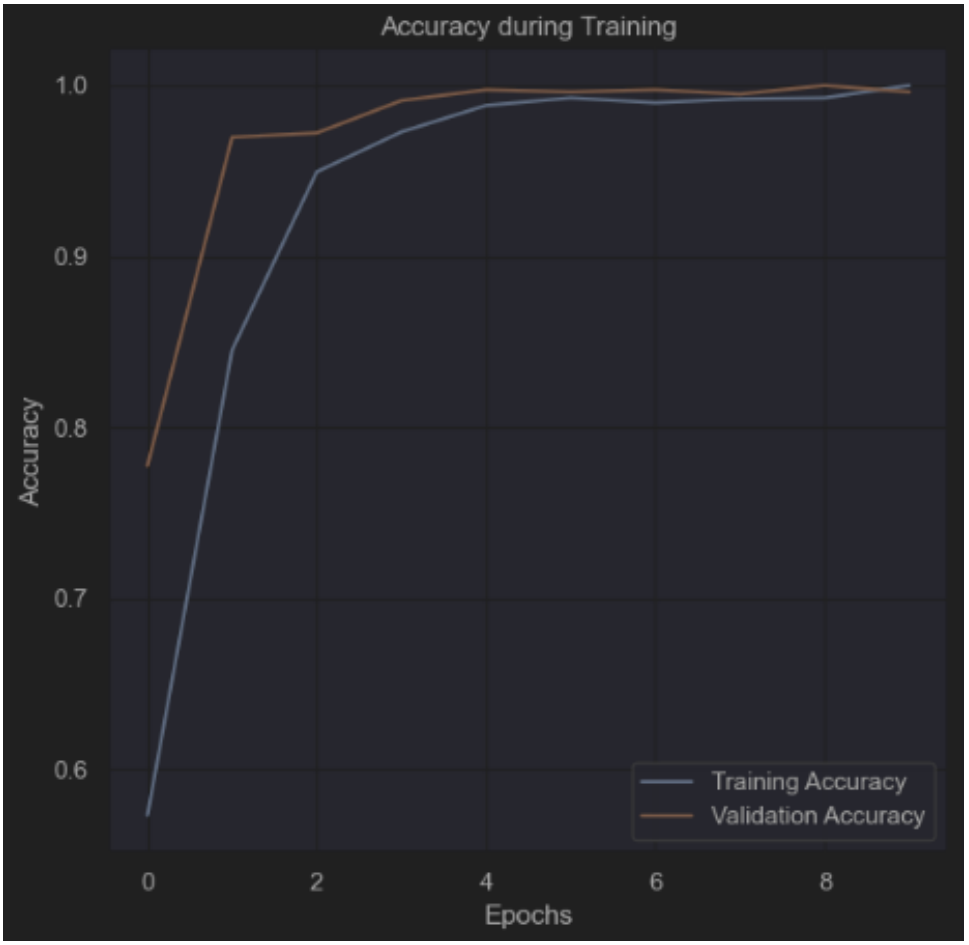


Evaluation des performances du modèle:

```
# Évaluer les performances du modèle

import matplotlib.pyplot as plt

# Extract loss and accuracy from training history
train_loss = history1.history['loss']
train_acc = history1.history['accuracy']
val_loss = history1.history['val_loss']
val_acc = history1.history['val_accuracy']

# Plot training and validation loss
plt.figure(figsize=(12, 6))

# Plot Loss
plt.subplot(1, 2, 1)
plt.plot(train_loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss')
plt.title('Loss during Training')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

```
# Plot Accuracy
plt.subplot(1, 2, 2)
plt.plot(train_acc, label='Training Accuracy')
plt.plot(val_acc, label='Validation Accuracy')
plt.title('Accuracy during Training')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()

# Evaluate the model on the test set
test_loss_model1, test_accuracy_model1
= model1.evaluate(X_test, y_test)
print(f"Test Accuracy:
{test_accuracy_model1 * 100:.2f}%")
```






```
1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - accuracy: 1.0000 - loss: 0.002818/19 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
  - accuracy: 0.9965 - loss: 0.014519/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9967 - loss:
 0.014419/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9967 - loss: 0.0144
Test Accuracy: 99.83%
```

Validation du modèle:

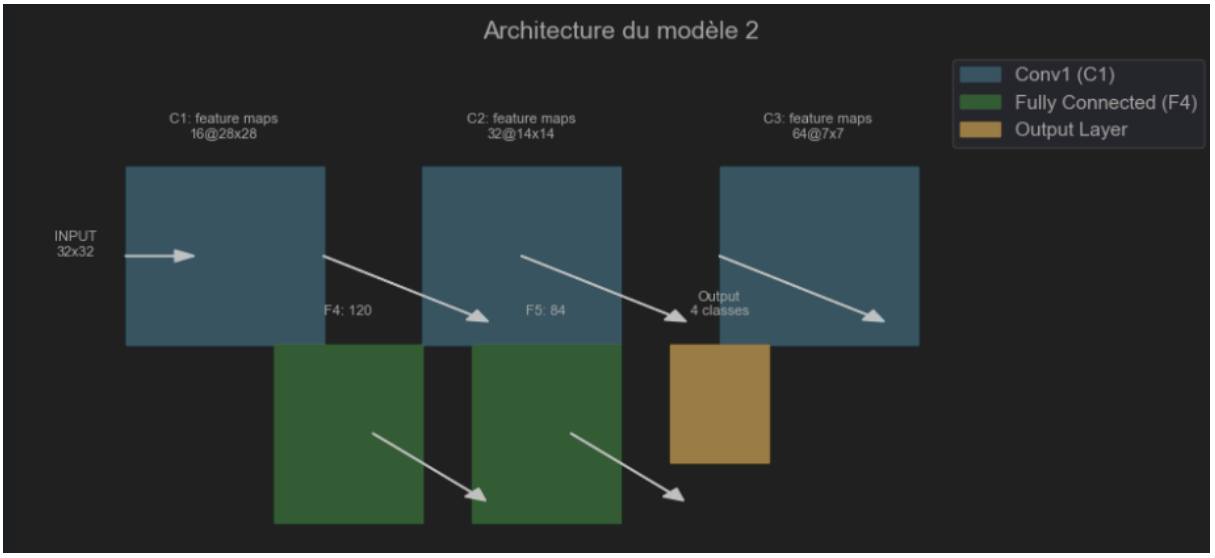
```
val_loss_model1, val_accuracy_model1
= model1.evaluate(X_test, y_test)

print(f"Validation Loss:
{val_loss_model1 * 100:.2f}%")
print(f"Validation Accuracy:
{ val_accuracy_model1 * 100:.2f}%")
```

```
1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step - accuracy: 1.0000 - loss: 0.002819/19 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
  - accuracy: 0.9967 - loss: 0.014419/19 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9967 - loss: 0.0144
Validation Loss: 1.32%
Validation Accuracy:99.83%
```

2. Amélioration de l'Architecture: Modèle avec Dropout

2.1. Nouvelle Architecture CNN Améliorée

Pour améliorer les performances de la classification, nous avons ajouté plusieurs couches convolutives et une couche de dropout pour éviter le surapprentissage (overfitting).



1. **Entrée** : Image de taille .

   32×32×132 \times 32 \times 1

2. **Couches convolutives et pooling** :

   - **C1** : Convolution de filtres suivis d’une activation ReLU.

     3232

     3×33 \times 3

   - **S2** : MaxPooling .

     2×22 \times 2

   - **C3** : Convolution de filtres avec activation ReLU.

     6464

     3×33 \times 3

   - **S4** : MaxPooling .

     2×22 \times 2

   - **C5** : Convolution de filtres avec activation ReLU.

     128128

3×33 \times 3

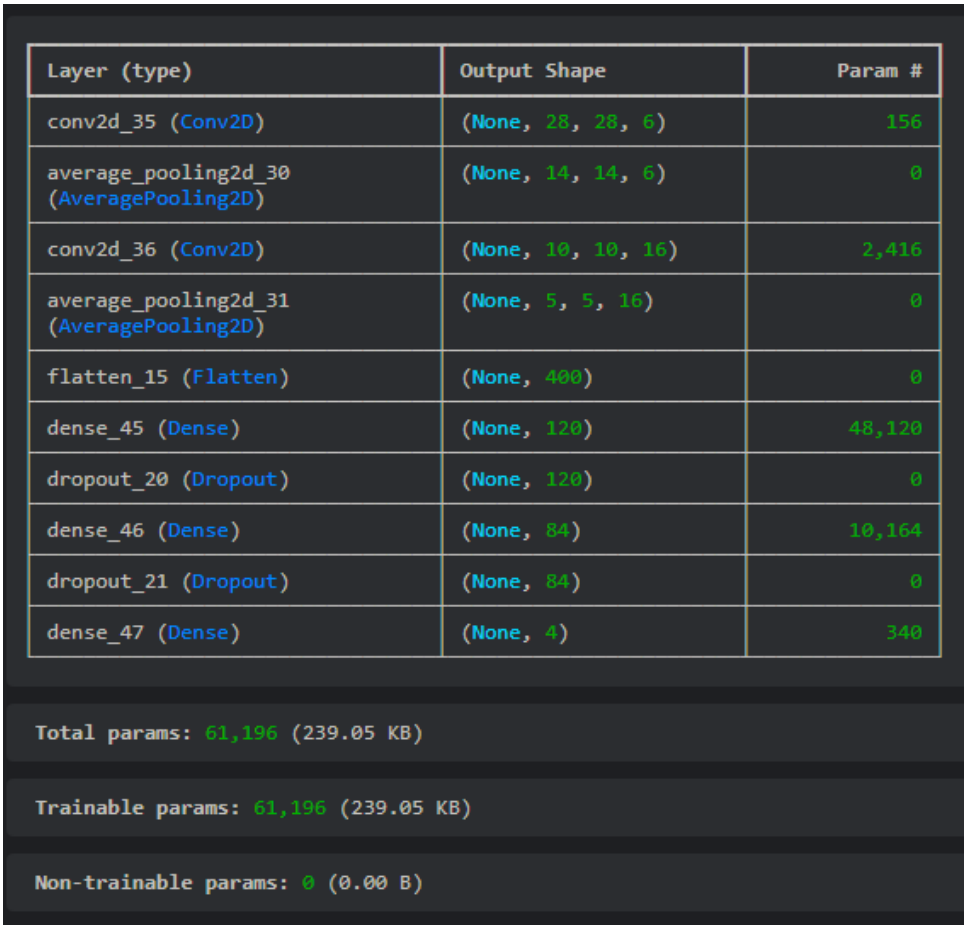
3. **Dropout** pour la régularisation (0.5).
4. **Couches entièrement connectées** :
   - **F6** : 256 neurones avec activation ReLU.
   - **Sortie** : 10 neurones pour classification multi-classe avec softmax.

Code Python pour la nouvelle architecture CNN :

```
model2 = Sequential([
    # C1: Couche de convolution
    #avec 6 filtres 5x5, activation ReLU, entrée 32x32x1
    # Définir la forme d'entrée 32x32x1
    InputLayer(shape=(32, 32, 1)),
    Conv2D(6, (5, 5), activation='relu'),

    # S2: Sous-échantillonnage (AveragePooling) 2x2
    AveragePooling2D(pool_size=(2, 2)),

    # C3: Deuxième couche de convolution
    #avec 16 filtres 5x5, activation ReLU
    Conv2D(16, (5, 5), activation='relu'),

    # S4: Sous-échantillonnage (AveragePooling) 2x2
    AveragePooling2D(pool_size=(2, 2)),

    # Aplatir pour passer aux couches denses
    Flatten(),

    # C5: Couche entièrement connectée avec 120 neurones
    Dense(120, activation='relu'),

    Dropout(0.5),

    # F6: Couche entièrement connectée avec 84 neurones
    Dense(84, activation='relu'),
```

```
    Dropout(0.5),

    # Couche de sortie avec 4 neurones
    #(pour 4 classes de classification)
    Dense(4, activation='softmax')
])

# Afficher le résumé du modèle
model2.summary()
```

Le sommaire après l'exécution:



Layer (type)	Output Shape	Param #
conv2d_35 (Conv2D)	(None, 28, 28, 6)	156
average_pooling2d_30 (AveragePooling2D)	(None, 14, 14, 6)	0
conv2d_36 (Conv2D)	(None, 10, 10, 16)	2,416
average_pooling2d_31 (AveragePooling2D)	(None, 5, 5, 16)	0
flatten_15 (Flatten)	(None, 400)	0
dense_45 (Dense)	(None, 120)	48,120
dropout_20 (Dropout)	(None, 120)	0
dense_46 (Dense)	(None, 84)	10,164
dropout_21 (Dropout)	(None, 84)	0
dense_47 (Dense)	(None, 4)	340

Total params: 61,196 (239.05 KB)

Trainable params: 61,196 (239.05 KB)

Non-trainable params: 0 (0.00 B)

Compilation du modèle:

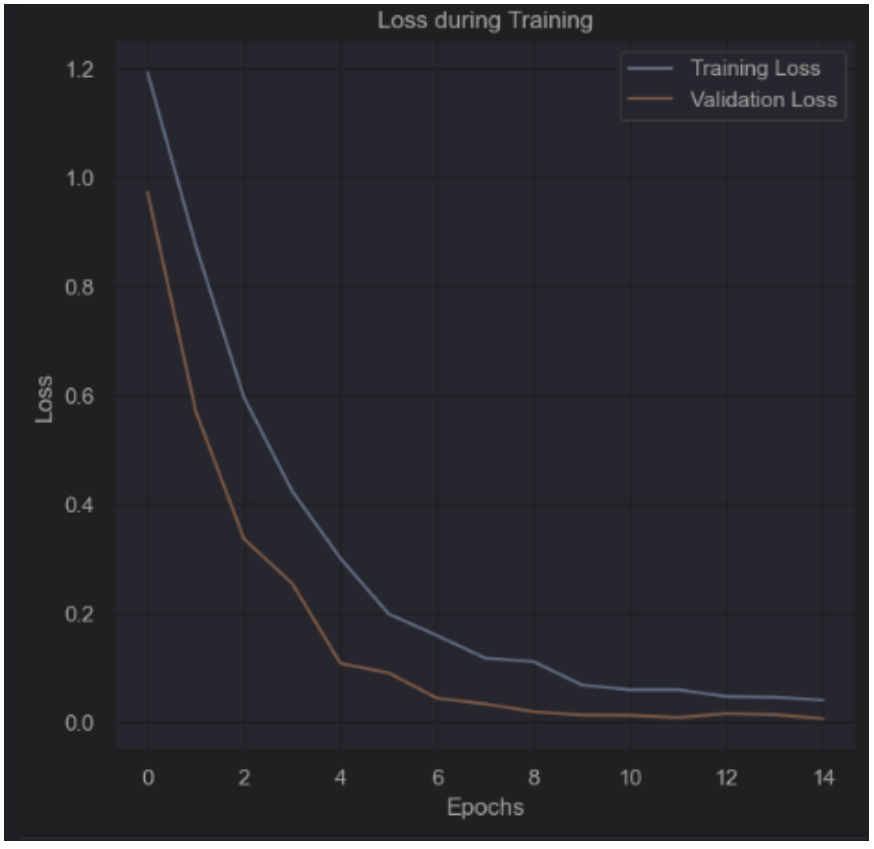
```
model2.compile(optimizer='adam'
, loss='categorical_crossentropy'
, metrics=['accuracy'])
```

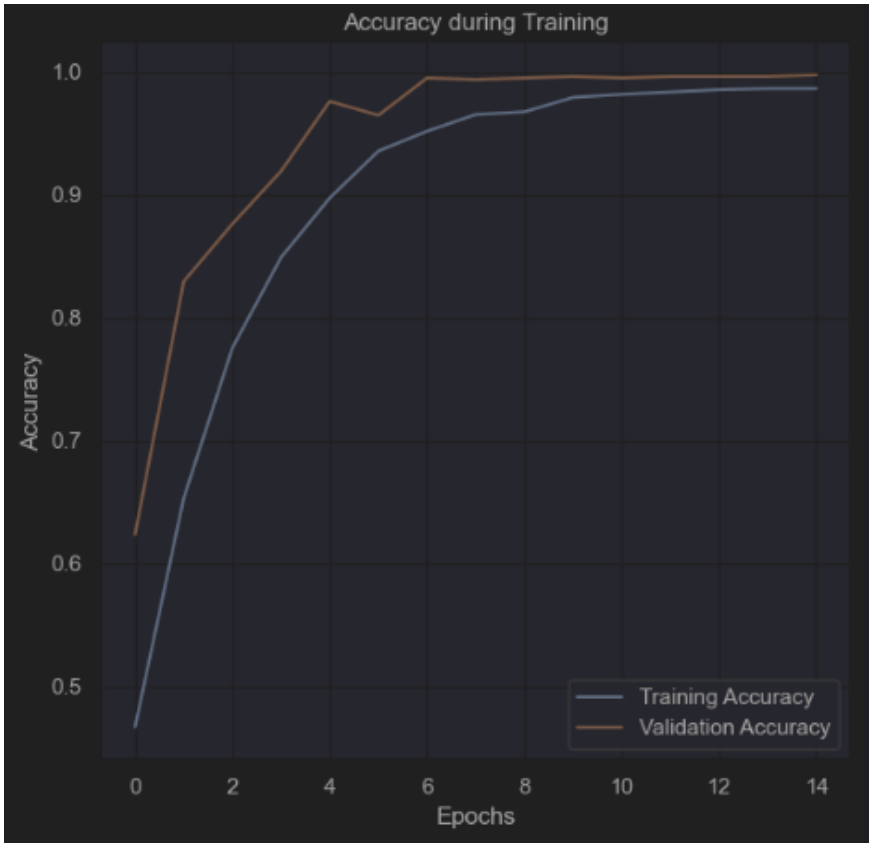
Entrainement du deuxième modèle:

```
history2 = model2.fit(
    X_train,
    y_train,
    validation_data=(X_val, y_val),
    epochs=15,
    batch_size=32
```

```
Epoch 11/15
99/99 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9841 - loss: 0.0565 - val_accuracy: 0.9949 - val_loss: 0.0131
Epoch 12/15
99/99 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9803 - loss: 0.0718 - val_accuracy: 0.9962 - val_loss: 0.0090
Epoch 13/15
99/99 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9879 - loss: 0.0433 - val_accuracy: 0.9962 - val_loss: 0.0164
Epoch 14/15
99/99 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9835 - loss: 0.0572 - val_accuracy: 0.9962 - val_loss: 0.0143
Epoch 15/15
99/99 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9856 - loss: 0.0402 - val_accuracy: 0.9975 - val_loss: 0.0070
```

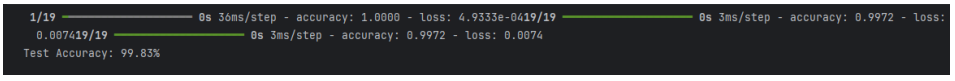
Les courbes d’entrainement et de validation:





Evaluation des performances du modèle:

```
# Évaluer les performances du modèle
test_loss_model2, test_accuracy_model2
= model2.evaluate(X_test, y_test)
print(f"Test Accuracy:
{test_accuracy_model2 * 100:.2f}%")
```



Evaluation des performances sur la validation:

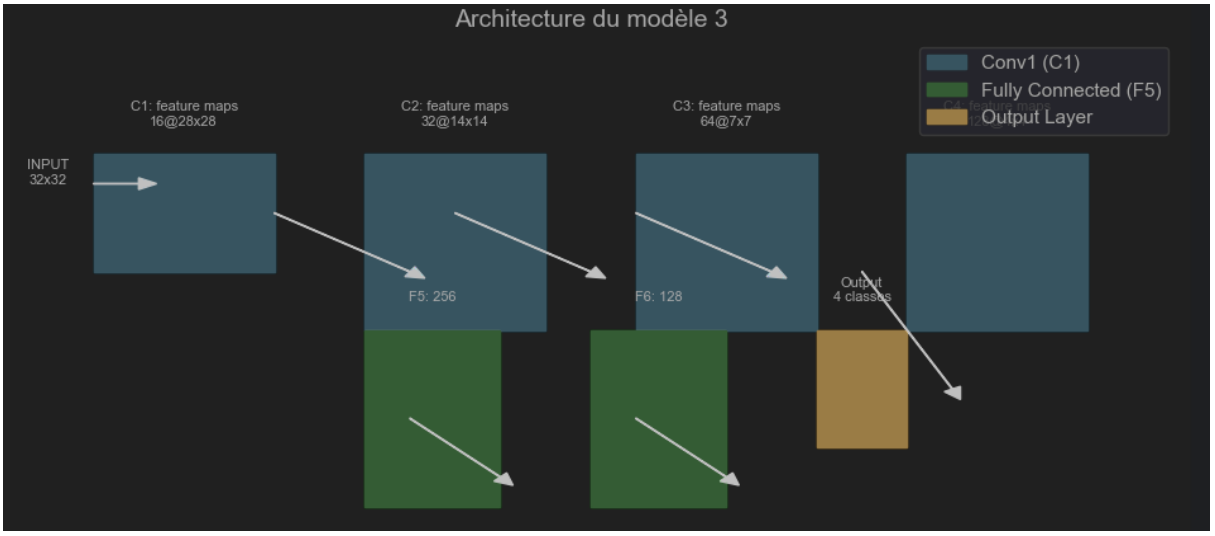
```
# Évaluer les performances du modèle sur la validation
val_loss_model2, val_accuracy_model2
 = model2.evaluate(X_val, y_val)

print(f"Validation Loss:
 {val_loss_model2:.2f}")
```

```
print(f"Validation Accuracy:
{val_accuracy_model2 * 100:.2f}%")
```

```
1/25 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - accuracy: 0.9688 - loss: 0.0534 2/25 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9766 - loss:
 0.040121/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9946 - loss: 0.010825/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9952 -
 loss: 0.009925/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9952 - loss: 0.0099
Validation Loss: 0.01
Validation Accuracy: 99.75%
```

3. Autre Architecture CNN avec plus de couches convolutifs et de couches de Dropout

Architecture du model 3:



voilà le troisième modèle réalisé:

```
model3 = Sequential([
    # Première couche
    #Définir la forme d'entrée avec InputLayer
    # Définir la forme d'entrée 32x32x1
    InputLayer(shape=(32, 32, 1)),

    # C1: Couche de convolution
    #avec 32 filtres 3x3, activation ReLU
    Conv2D(32, (3, 3), activation='relu'),

    # Dropout après la première couche convolutive
    Dropout(0.2),
```

```
    # C2: Couche de convolution
    #avec 64 filtres 3x3, activation ReLU
    Conv2D(64, (3, 3), activation='relu'),
    # Sous-échantillonnage (AveragePooling) 2x2
    AveragePooling2D(pool_size=(2, 2)),
    Dropout(0.3),

    # C3: Couche de convolution
    #avec 128 filtres 3x3, activation ReLU
    Conv2D(128, (3, 3), activation='relu'),
    # Sous-échantillonnage (AveragePooling) 2x2
    AveragePooling2D(pool_size=(2, 2)),
    Dropout(0.4),

    # C4: Couche de convolution
    #avec 256 filtres 3x3, activation ReLU
    Conv2D(256, (3, 3), activation='relu'),
    Dropout(0.4),

    # Aplatir pour passer aux couches
    #entièrement connectées
    Flatten(),

    # C5: Couche entièrement connectée
    #avec 512 neurones
    Dense(512, activation='relu'),
    Dropout(0.5),

    # F6: Couche entièrement connectée
    #avec 256 neurones
    Dense(256, activation='relu'),
    Dropout(0.5),

    # Couche de sortie avec 4 neurones
    #(pour 4 classes de classification)
    Dense(4, activation='softmax')
])
```

```
model3.compile(optimizer='adam'
, loss='categorical_crossentropy'
, metrics=['accuracy'])
```

Le sommaire est ainsi représenté comme suit:

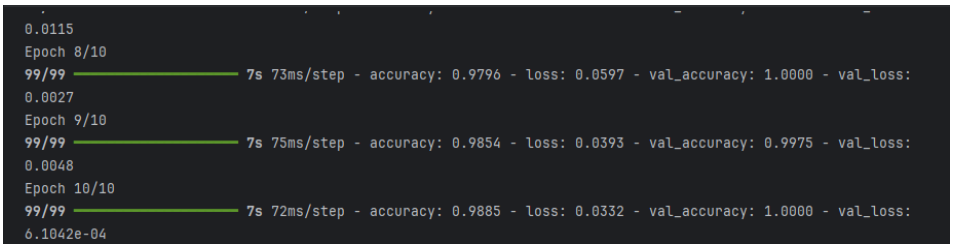
Layer (type)	Output Shape	Param #
conv2d_37 (Conv2D)	(None, 30, 30, 32)	320
dropout_22 (Dropout)	(None, 30, 30, 32)	0
conv2d_38 (Conv2D)	(None, 28, 28, 64)	18,496
average_pooling2d_32 (AveragePooling2D)	(None, 14, 14, 64)	0
dropout_23 (Dropout)	(None, 14, 14, 64)	0
conv2d_39 (Conv2D)	(None, 12, 12, 128)	73,856
average_pooling2d_33 (AveragePooling2D)	(None, 6, 6, 128)	0
dropout_24 (Dropout)	(None, 6, 6, 128)	0
conv2d_40 (Conv2D)	(None, 4, 4, 256)	295,168
dropout_25 (Dropout)	(None, 4, 4, 256)	0
flatten_16 (Flatten)	(None, 4096)	0
dense_48 (Dense)	(None, 512)	2,097,664
dropout_26 (Dropout)	(None, 512)	0
dense_49 (Dense)	(None, 256)	131,328
dropout_27 (Dropout)	(None, 256)	0
dense_50 (Dense)	(None, 4)	1,028

```
Total params: 2,617,860 (9.99 MB)

Trainable params: 2,617,860 (9.99 MB)

Non-trainable params: 0 (0.00 B)
```

Entrainement du modèle:

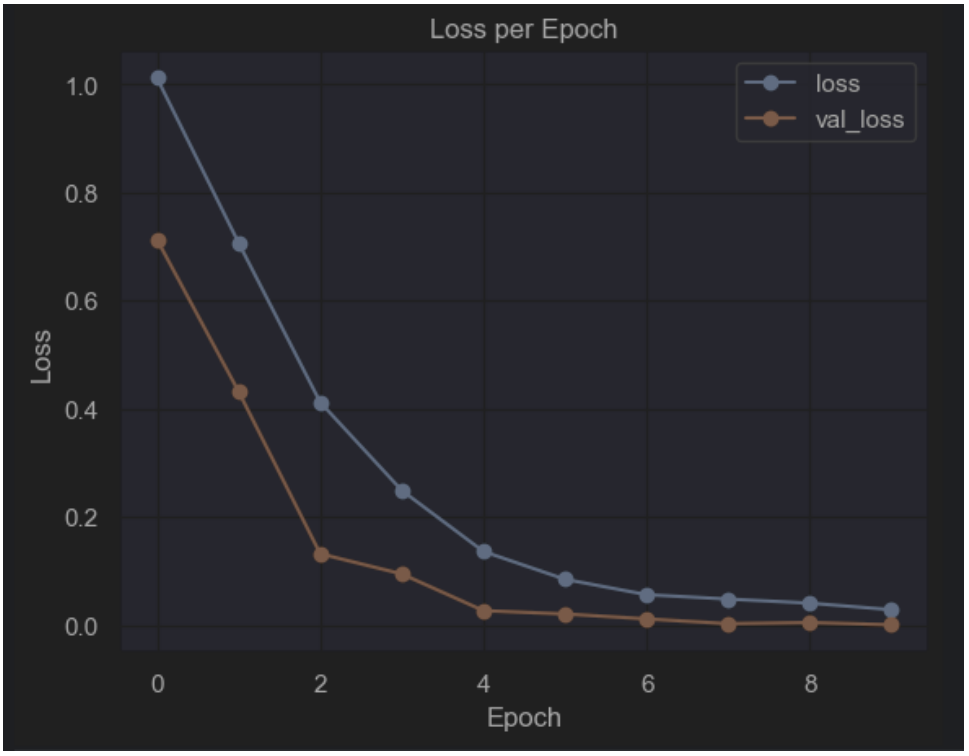
```
history3 = model3.fit(X_train
, y_train
, validation_data=(X_val, y_val)
, epochs=10, batch_size=32)
```

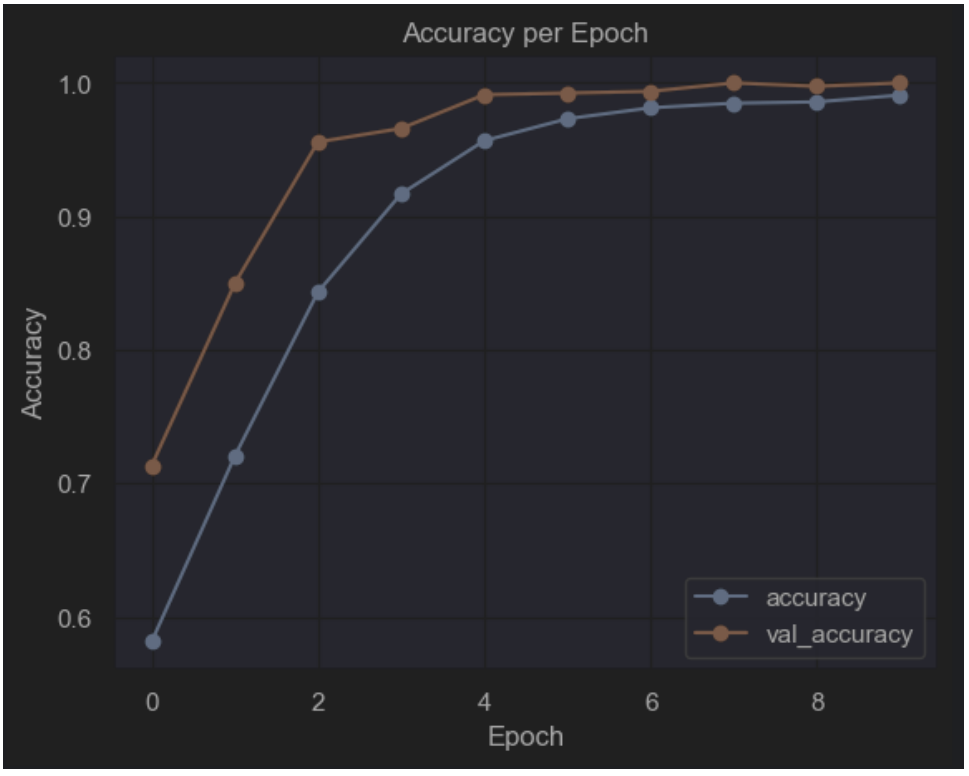
Le résultat à donné 98% d’auccuracy:



Les courbes d’entrainement et de validation:

```
import pandas as pd
import matplotlib.pyplot as plt

# Convertir l'historique en DataFrame
history3= pd.DataFrame(history3.history)

# Tracer les courbes d'entraînement et de validation
history3[['loss', 'val_loss']]
.plot(title='Loss per Epoch', xlabel='Epoch'
                            , ylabel='Loss', marker='o')
history3[['accuracy', 'val_accuracy']]
.plot(title='Accuracy per Epoch'
    , xlabel='Epoch', ylabel='Accuracy', marker='o')

plt.show()
```





Evaluation des performances du modèle:

```
# Évaluer les performances du modèle
test_loss_model3, test_accuracy_model3
= model3.evaluate(X_train, y_train)
```

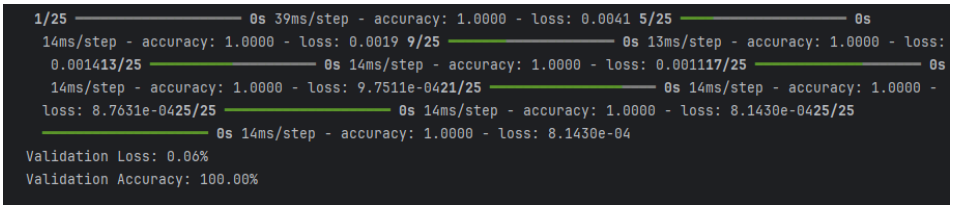
```
print(f"Test Accuracy:
{test_accuracy_model3 * 100:.2f}%")
```

Le résultat de l’exécution:

Evaluation des performances de la validation:

```
# Évaluer les performances du modèle sur la validation
val_loss_model3, val_accuracy_model3
= model3.evaluate(X_val, y_val)
print(f"Validation Loss:
{val_loss_model3 * 100:.2f}%")
print(f"Validation Accuracy:
{val_accuracy_model3 * 100:.2f}%")
```

Le résultat de la validation:



Comparaison entre les trois modèles:

```
import pandas as pd
# Dictionnaire contenant les résultats
results = {
 'Model': ['Model 1', 'Model 2', 'Model 3'],
 'Test Accuracy':[test_accuracy_model1
 , test_accuracy_model2
 , test_accuracy_model3],
 'Validation Accuracy':[val_accuracy_model1
```

```
 ,val_accuracy_model2
 ,val_accuracy_model3],
 'Test Loss': [test_loss_model1
 , test_loss_model2
 , test_loss_model3],
 'Validation Loss': [val_loss_model1
 , val_loss_model2
 , val_loss_model3],
}

# Convertir les colonnes de précision en pourcentage
results['Validation Accuracy']=
[acc * 100 for acc in results['Validation Accuracy']]
results['Test Accuracy'] =
[acc * 100 for acc in results['Test Accuracy']]

# Créer un DataFrame pour afficher les résultats
results_df = pd.DataFrame(results)

# Afficher les valeurs en pourcentage avec 2 décimales
pd.options.display.float_format = '{:,.2f}%'.format
print(results_df)
```

	Model	Test Accuracy	Validation Accuracy	Test Loss	Validation Loss
0	Model 1	99.83%	99.83%	0.01%	0.01%
1	Model 2	99.83%	99.75%	0.01%	0.01%
2	Model 3	99.94%	100.00%	0.00%	0.00%

Bilan

D’après les résultats obtenus On voit clairement que le modèle ayant une architecture CNN avec plusieurs couches convolutives et de couches Dropout est le meilleure par rapport aux autres qu’il soit au niveau de test ou de validation .

---

4. Expérimentations et Ajustements des Hyperparamètres

4.1. Taille du lot ("batch size")

La taille du lot affecte à la fois le temps de calcul par époque et la convergence du modèle. Par exemple, un **batch size de 32** est souvent utilisé, mais des tailles plus petites (comme 16) ou plus grandes (comme 64) peuvent être testées pour optimiser la performance.

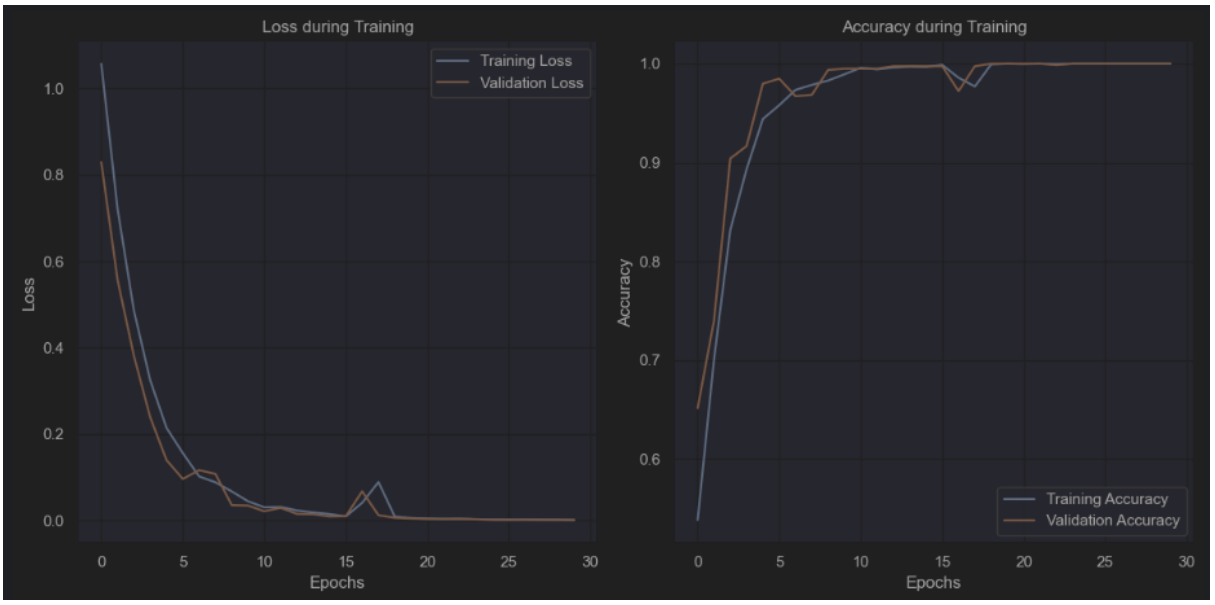
Pour le test effectué nous avons utilisé 20 époques et 64 comme taille de lot pour tester.

On va garder dans ce test l'optimiseur **ADAM**

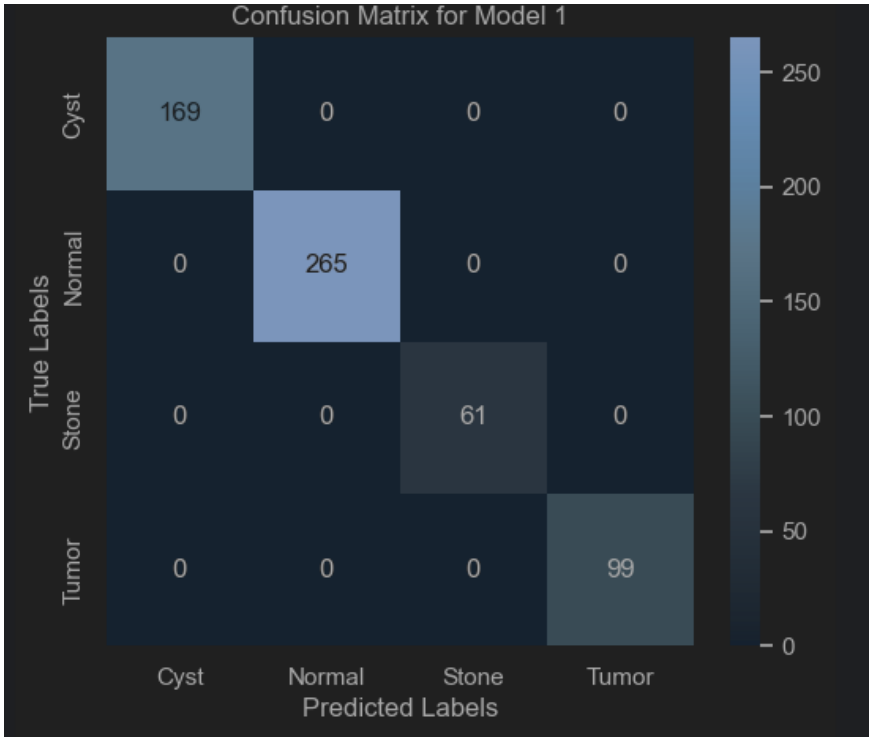
Les modifications mises en place ont permis d'améliorer les performances de chaque modèle en fonction des hyperparamètres choisis.

- **Pour le modèle LetNet5 sans Dropout**

les courbes de pertes et de précision obtenues:

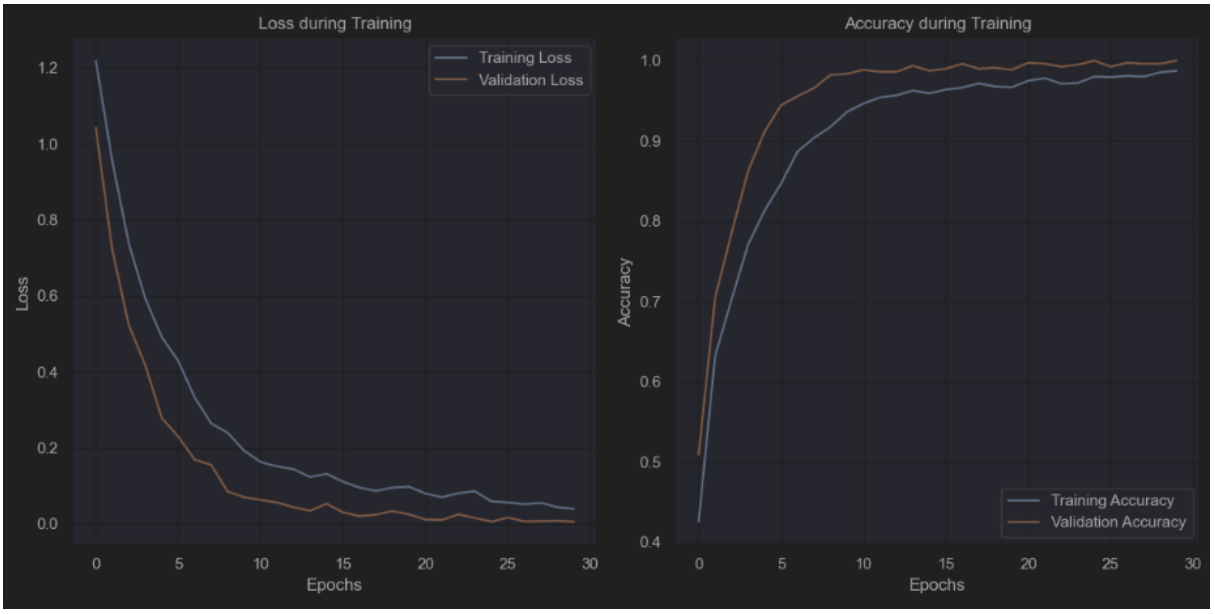


La matrice de Confusion obtenue:

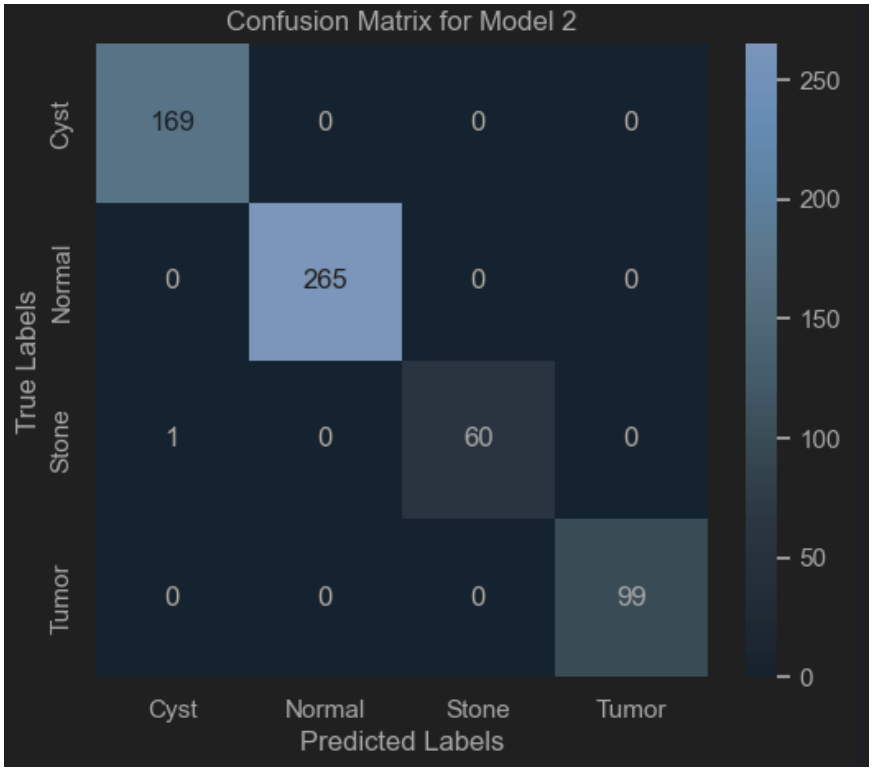


- **Pour le modèle LetNet5 avec Dropout**

Voici la courbe de perte et de précision :

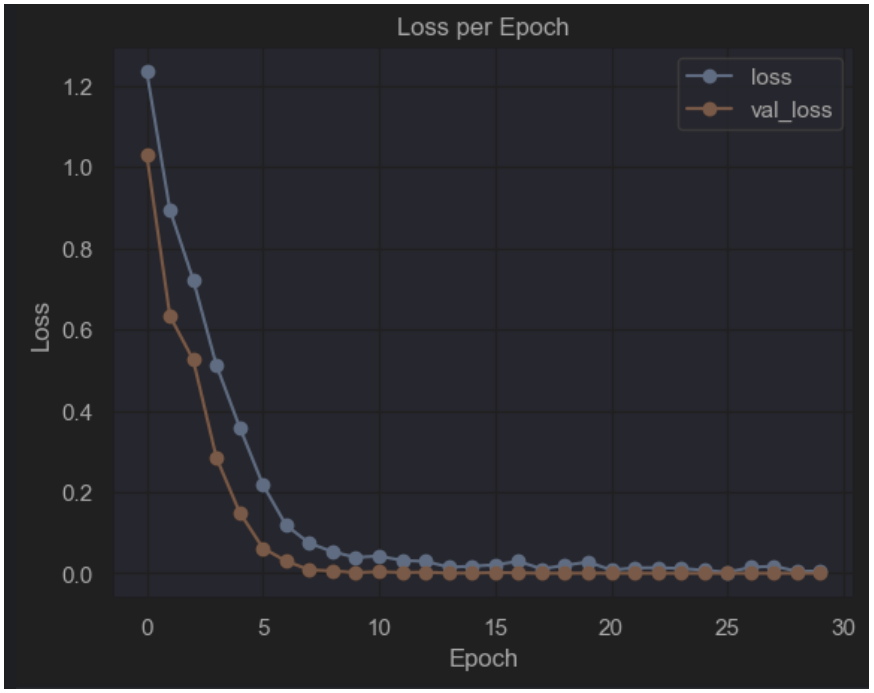


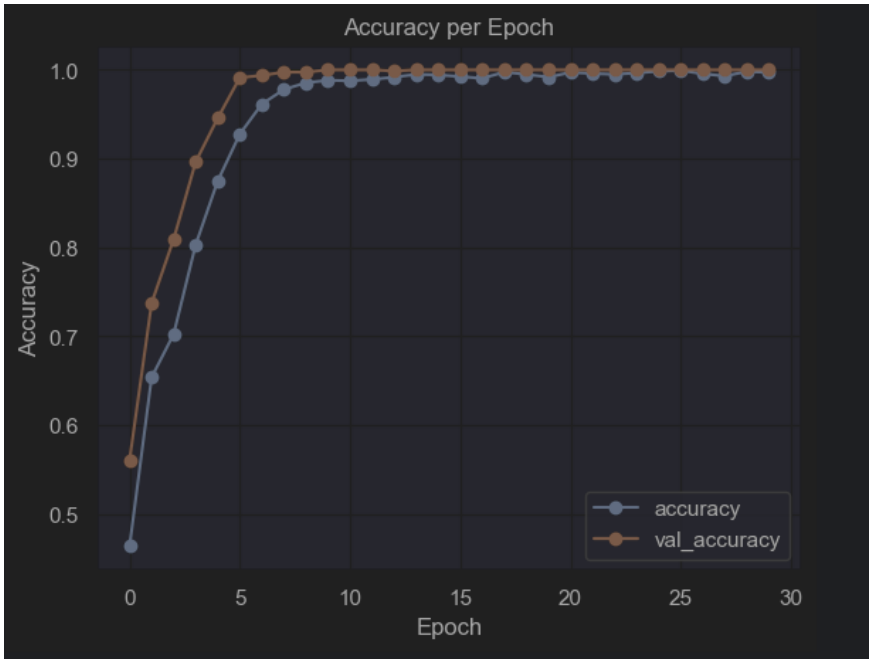
La matrice de confusion:



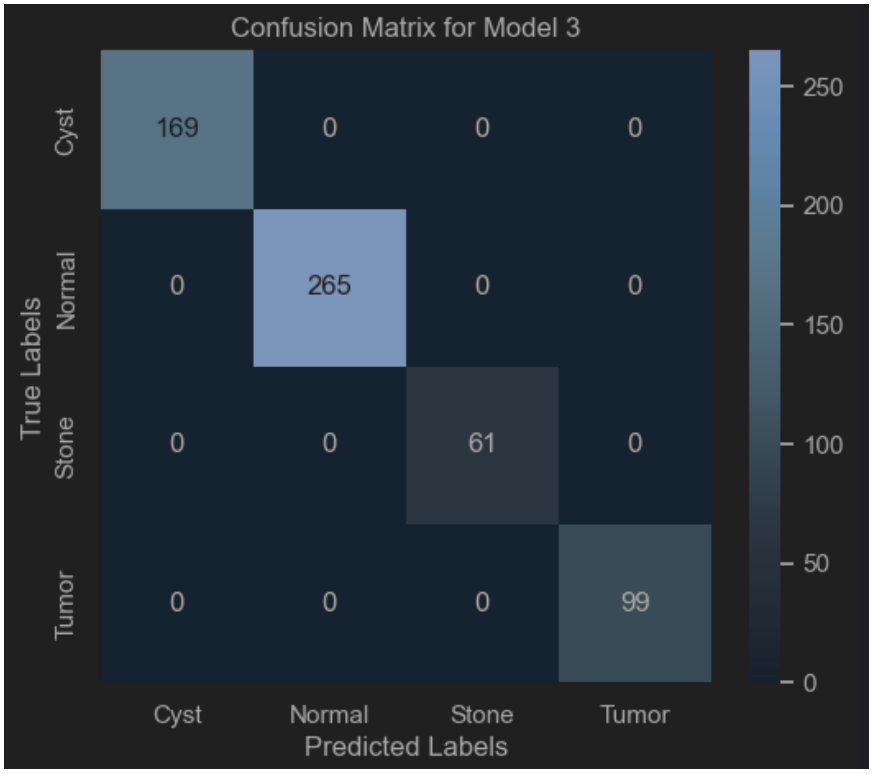
- **Pour le dernier Model**

Voici la courbe de perte et de précision :





La matrice de confusion:



4.2. Fonctions d’optimisation

Nous avons comparé plusieurs optimisateurs pour évaluer leurs performances :

- **SGD** : Stochastic Gradient Descent standard.
- **SGD avec Momentum** : Améliore l’apprentissage en accumulant un momentum.

- **Adam** : Optimiseur adaptatif qui ajuste automatiquement le taux d'apprentissage.

Tests réalisés :

- **Utilisation de l'optimiseur Adam**

  Résultats:

  Les étapes précédentes montrent clairement les résultats de l'utilisation de l'optimiseur **Adam**.

```
Model 1 - Precision: 1.0, Recall: 1.0, F1 Score: 1.0
Model 2 - Precision: 0.9985294117647059, Recall: 0.9959016393442623, F1 Score: 0.997196421170677
Model 3 - Precision: 1.0, Recall: 1.0, F1 Score: 1.0
```

  Une autre exécution à été faite en utilisation l'optimiseur Adam et 20 comme nombre d'époques:

	Model	Test Accuracy	Validation Accuracy	Test Loss	Validation Loss
0	Model 1	99.83%	99.83%	0.00%	0.00%
1	Model 2	99.83%	99.75%	0.01%	0.01%
2	Model 3	100.00%	100.00%	0.00%	0.00%

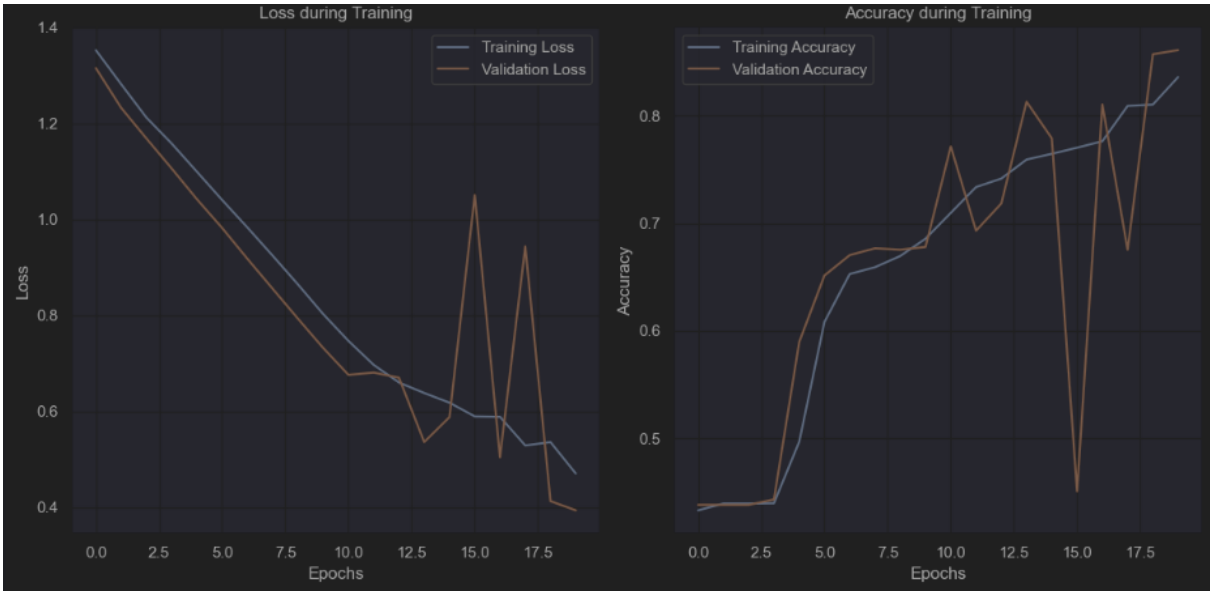
```
Model 1 - Precision: 0.9990601503759399, Recall: 0.9959016393442623, F1 Score: 0.997463074504677
Model 2 - Precision: 0.9990601503759399, Recall: 0.9959016393442623, F1 Score: 0.997463074504677
Model 3 - Precision: 1.0, Recall: 1.0, F1 Score: 1.0
```

- **Utilisation de l'optimiseur SGD:**

  On va maintenant changer l'optimiseur et utiliser le **SGD** à la place de **Adam**.

  Résultats d'utilisation:

▼ Pour le premier modèle LetNet5 sans Dropout:



Analyses des résultats obtenus:

1. Courbe de perte :

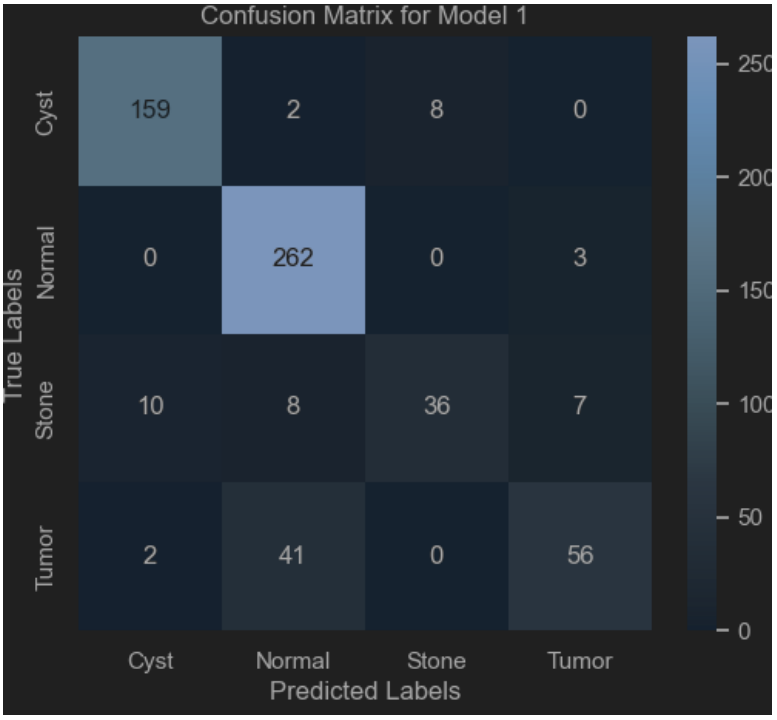
- On voit clairement d’après les résultats obtenus que la perte a connue une diminution progressive au fil des époques qu’il soit pour l’ensemble d’entraînement ou de validation.
- On déduit, de surcroit, que la courbe de validation montre plus d'oscillations et est plus irrégulière que celle d'entraînement.
- L'écart entre les deux courbes augmente surtout à la fin de la courbe, ainsi que l’adaptation parfaite du modèle montre l’existence d’un surapprentissage(Overfitting).

2. Courbe de précision :

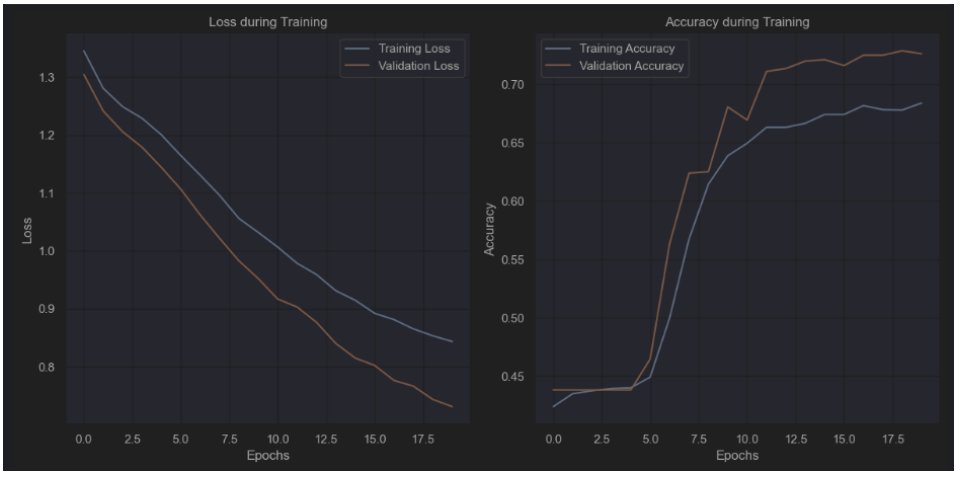
- La précision augmente pour les deux ensembles.
- La précision d'entraînement atteint près de 84.09%.
- La précision de validation atteint 86.36%.
- Les oscillations importantes dans la validation indiquent une certaine instabilité dans l'apprentissage

Le modèle parvient à apprendre efficacement, mais des indications de surapprentissage sont observées.

L'analyse de la matrice de confusion révèle également la présence de plusieurs prédictions erronées.



▼ Pour le deuxième modèle LetNet5 avec Dropout:



Analyses des résultats obtenus:

1. Courbe de perte :

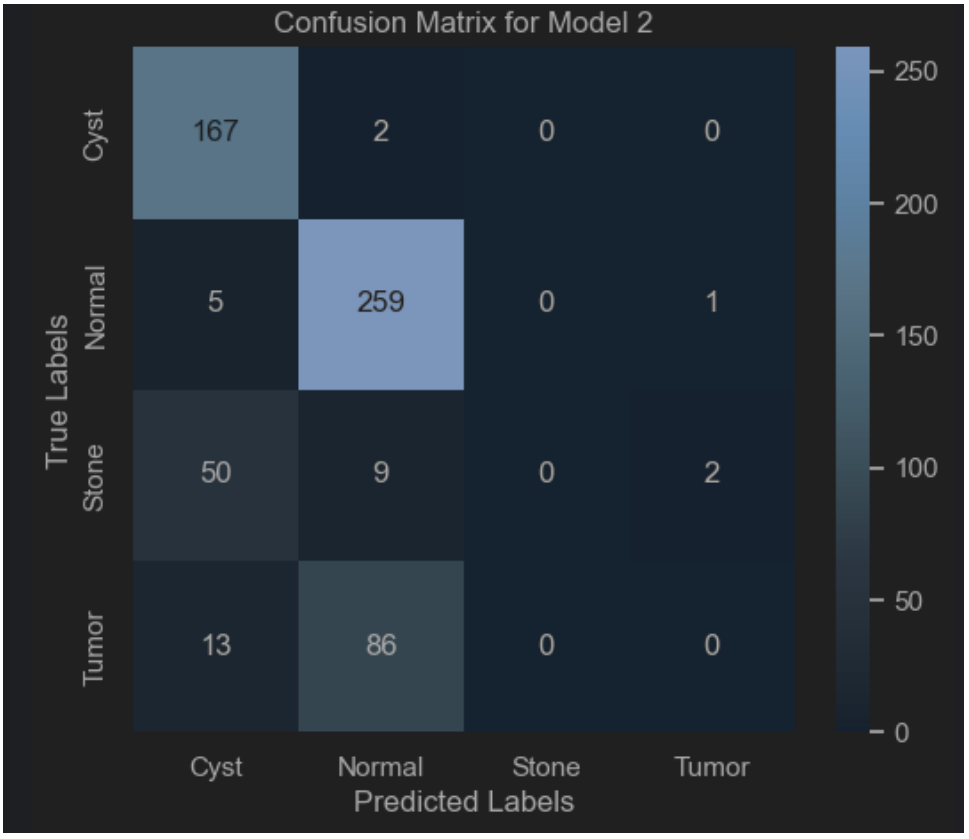
- Les courbes d’entraînement et de validation affichent une diminution progressive.
- La perte de validation est inférieure à celle d’entraînement.
- Les courbes sont relativement lisses, sans fluctuations importantes.
- La diminution reste constante jusqu’à la fin des 20 époques.

2. Courbe de précision :

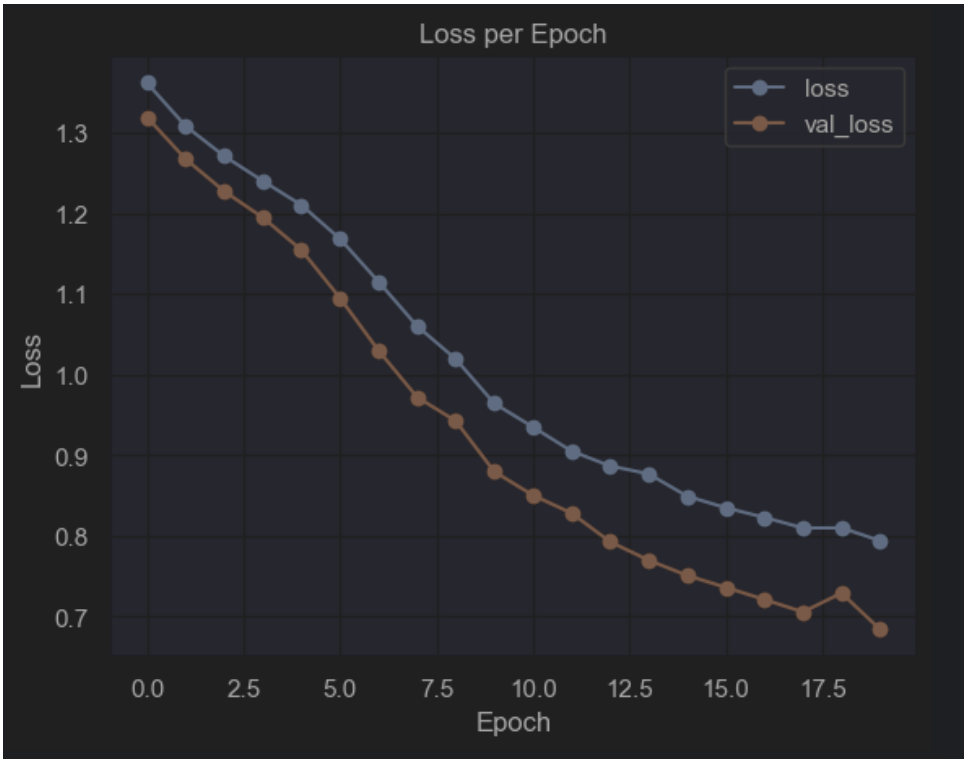
- La précision progresse de manière graduelle pour les ensembles d'entraînement et de validation.
- La précision de validation dépasse celle d'entraînement.
- L'écart entre les deux courbes est significatif et stable.

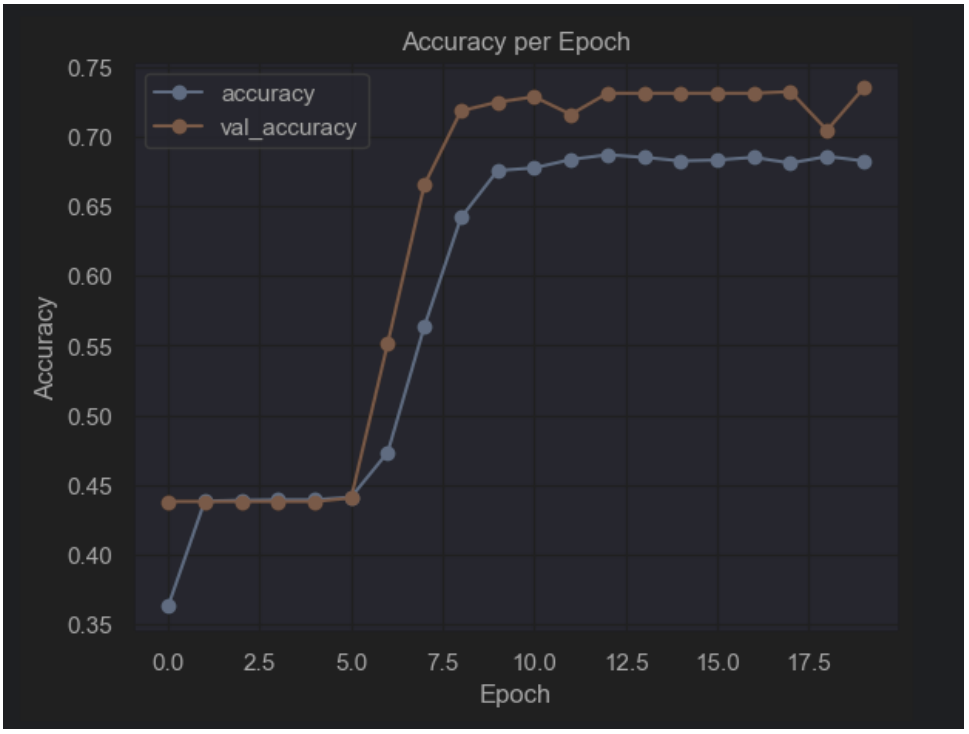
3. Conclusion :

- Les performances sur l'ensemble de validation surpassent celles sur l'ensemble d'entraînement, ce qui indique un problème de sous-apprentissage (underfitting).
- Le modèle continue d'apprendre tout au long des 20 époques.
- Les courbes, plus régulières que dans l'exemple précédent, reflètent un apprentissage plus stable.
- La matrice révèle plusieurs prédictions erronées.



▼ Pour le dernier modèle:





Analyse des résultats obtenus:

1. Courbe de perte :

- On voit clairement une diminution rapide de la perte au fil des époques.
- La perte de validation décroît plus rapidement que celle d’entraînement.

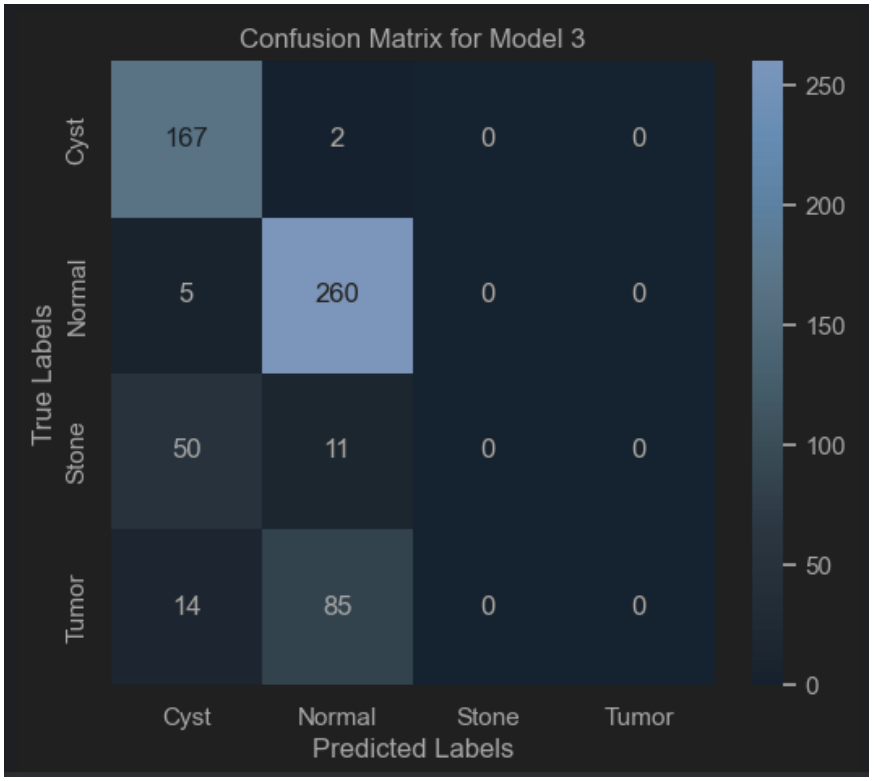
- La convergence n’est pas encore atteinte à l’époque 20.

2. Courbe de précision :

- La précision stagne autour de 45 % jusqu’à l’époque 5.
- Une amélioration significative des performances est notée après l’époque 5.
- La précision de validation atteint environ 75 %, tandis que celle d’entraînement atteint environ 70 %.
- L’écart entre les courbes de validation et d’entraînement augmente vers la fin.

3. Conclusion :

- Le modèle présente des signes de sous-apprentissage (underfitting) et nécessiterait un nombre d’époques d’entraînement plus élevé pour améliorer ses performances.
- La matrice de confusion montre également des valeurs faussement prédites:



Les résultats d’utilisation de l’optimiseur SGD:

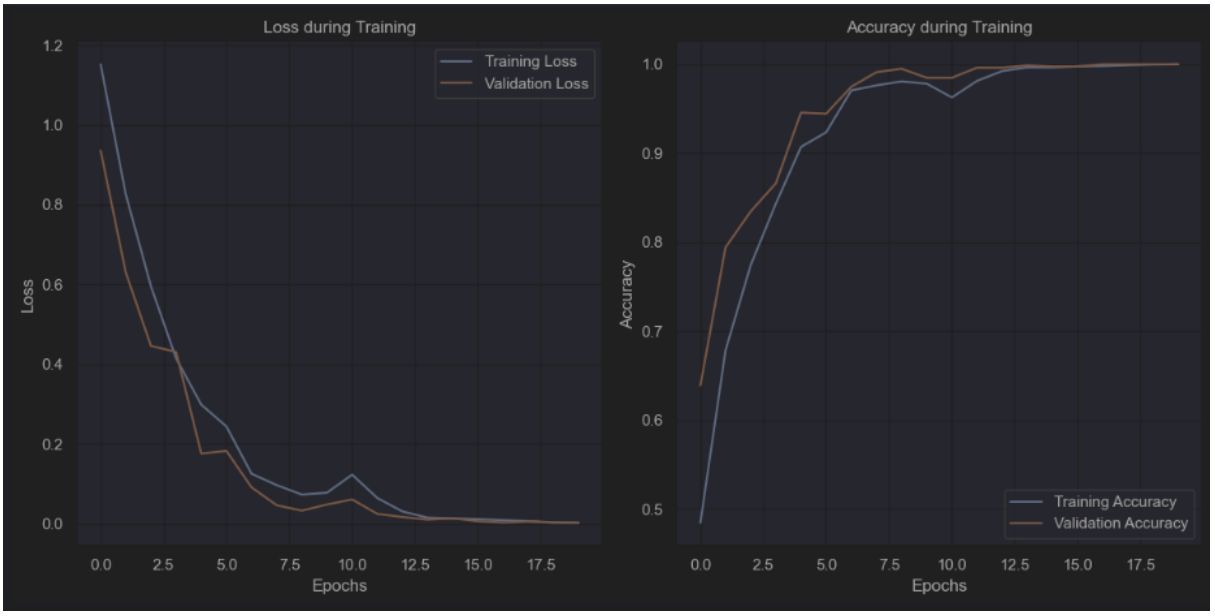
	Model	Test Accuracy	Validation Accuracy	Test Loss	Validation Loss
0	Model 1	86.36%	86.36%	0.38%	0.38%
1	Model 2	71.72%	72.60%	0.72%	0.73%
2	Model 3	70.59%	73.48%	0.73%	0.68%

```
Model 1 - Precision: 0.8583879827363937, Recall: 0.7713320369331695, F1 Score: 0.8015926691532227
Model 2 - Precision: 0.3595415969399952, Recall: 0.4913810427598526, F1 Score: 0.4152177898949315
Model 3 - Precision: 0.3584710254710728, Recall: 0.49232443898626777, F1 Score: 0.4148405762637972
```

- **Utilisation de l’optimiseur SGD avec Momentum:**

Résultats d’utilisation:

▼ Pour le premier modèle LetNet5 sans Dropout:



Analyse des résultats obtenus:

1. Courbe de perte:

1.1. **Observation générale** :

- La perte (loss) diminue régulièrement pour les ensembles d'entraînement et de validation, indiquant une amélioration progressive du modèle.
- Les courbes montrent une convergence cohérente avec des valeurs de perte proches de zéro en fin de formation.

1.2. **Détail des phases** :

- **Début de l'entraînement (0 à ~5 époques)** :
  Une diminution rapide de la perte est observée, suggérant que le modèle apprend rapidement les caractéristiques principales des données.
- **Milieu de l'entraînement (~5 à 10 époques)** :
  La diminution ralentit, indiquant une stabilisation progressive de l'apprentissage.

- **Fin de l'entraînement (>10 époques)** :
  La perte atteint un plateau avec des valeurs très faibles, ce qui suggère que le modèle s’est bien ajusté.

1.3. Validation vs Entraînement :

- Les pertes de validation et d'entraînement suivent des trajectoires similaires, avec une perte de validation légèrement inférieure.
- Cela montre que le modèle généralise bien et qu'il n'y a pas de surapprentissage (overfitting).

---

2. Courbe de précision

2.1. Observation générale :

- La précision augmente progressivement pour les ensembles d'entraînement et de validation.
- Les courbes atteignent une précision élevée (environ 100 % pour l'entraînement et la validation).

2.2. Détail des phases :

- **Début de l'entraînement (0 à ~5 époques)** :
  Une augmentation rapide de la précision est visible, confirmant un apprentissage rapide des patterns de base.
- **Milieu de l'entraînement (~5 à 10 époques)** :
  La précision continue d'augmenter à un rythme plus lent, indiquant un affinement des paramètres.
- **Fin de l'entraînement (>10 époques)** :
  Les courbes se stabilisent avec une précision proche de 100 %, montrant une convergence.

2.3. Validation vs Entraînement :

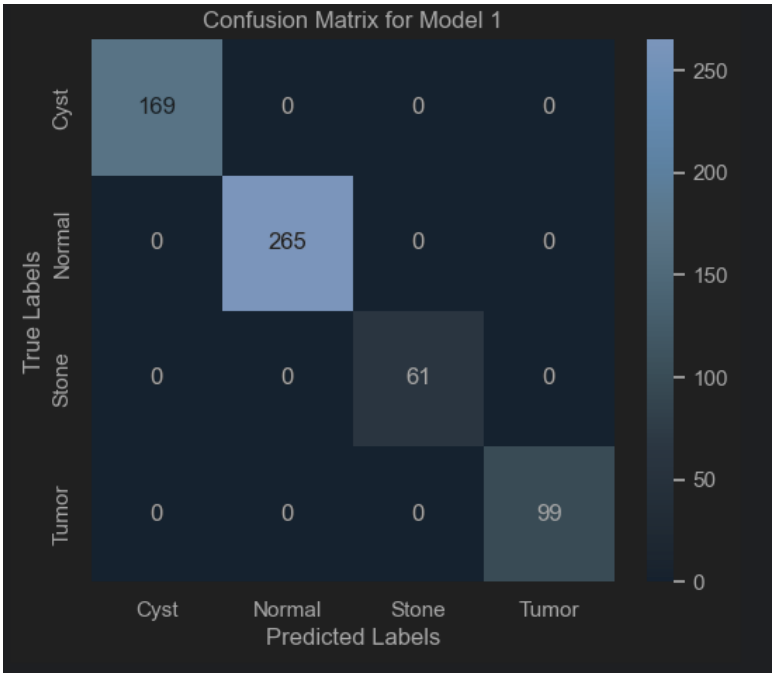
- Les courbes de précision de validation et d'entraînement sont très proches, avec un léger avantage pour la validation.
- Cela indique une bonne généralisation, sans signes d'underfitting ou d'overfitting.

---

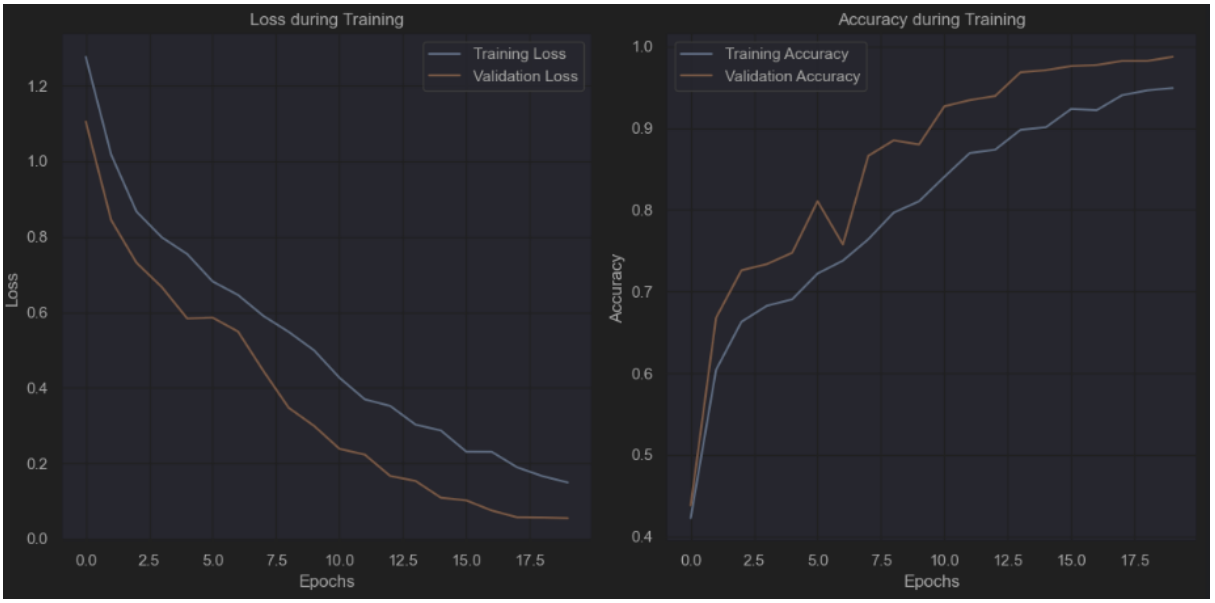
3. **Conclusion** :

- Le modèle utilisant **SGD avec Momentum** montre des performances solides, avec une bonne convergence des courbes de perte et de précision.

- Les pertes faibles et la précision élevée, tant pour l'entraînement que pour la validation, confirment l'efficacité de l'optimiseur dans ce contexte.
- **Améliorations possibles** :
  - Tester sur un jeu de données plus complexe ou introduire des augmentations de données pour évaluer davantage la robustesse du modèle.
  - Évaluer les performances sur des données totalement inédites pour confirmer la généralisation.
- La matrice de confusion montre également des valeurs faussement prédites:



▼ Pour le deuxième modèle avec Dropout:



Analyse des résultats obtenus:

1. **Analyse des courbes de perte** :

1.1. Observation générale :

- Les pertes d'entraînement et de validation diminuent régulièrement tout au long des époques.
- La perte de validation est inférieure à celle d'entraînement, indiquant une bonne généralisation.

1.2. Détail des phases :

- **Début de l'entraînement (0 à ~5 époques)** :
  Une réduction rapide de la perte est observée, marquant une phase d'apprentissage initial efficace.
- **Milieu et fin de l'entraînement (~5 à 18 époques)** :
  La diminution reste progressive mais ralentit, montrant que le modèle s'approche de la convergence.

1.3. Validation vs Entraînement :

- La perte de validation reste constamment inférieure à la perte d'entraînement, ce qui est un indicateur positif.
- Les courbes sont proches, confirmant l'absence de surapprentissage.

---

2. **Analyse des courbes de précision** :

2.1. Observation générale :

- La précision augmente progressivement pour les ensembles d'entraînement et de validation.
- Une précision élevée est atteinte (près de **100 %** pour la validation et un peu moins pour l'entraînement).

2.2. Détail des phases :

- **Début de l'entraînement (0 à ~5 époques)** :
  La précision augmente rapidement, montrant que le modèle capture les patterns de base.
- **Milieu de l'entraînement (~5 à 10 époques)** :
  Une amélioration continue est observée, bien que plus lente, suggérant un affinement des poids.
- **Fin de l'entraînement (>10 époques)** :
  Les courbes se stabilisent avec des valeurs proches de la précision maximale, confirmant une convergence.

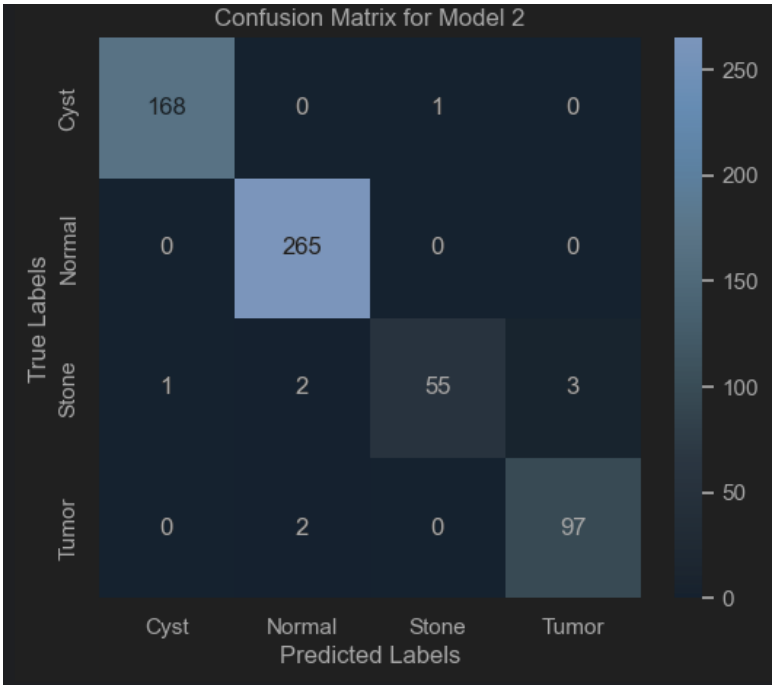
2.3. Validation vs Entraînement :

- La précision de validation dépasse légèrement celle d'entraînement dans les dernières époques.
- Cela indique que le modèle est bien régularisé et généralise efficacement.

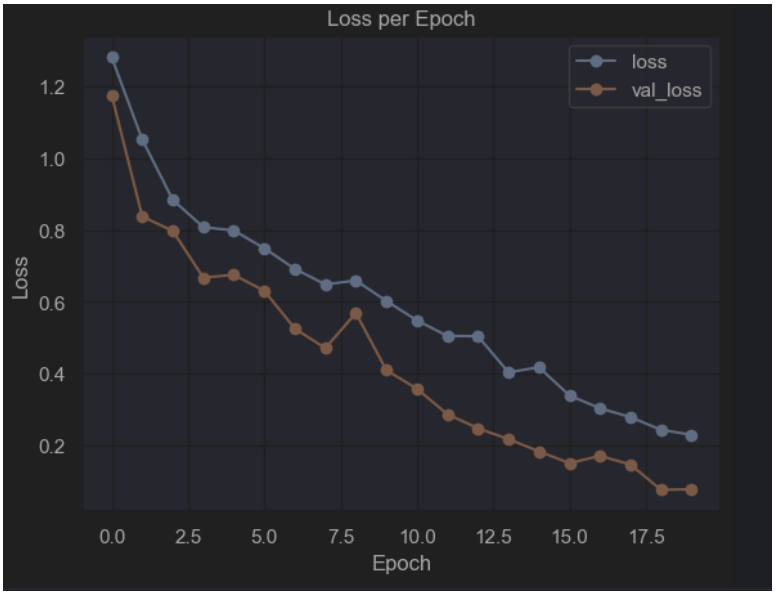
---

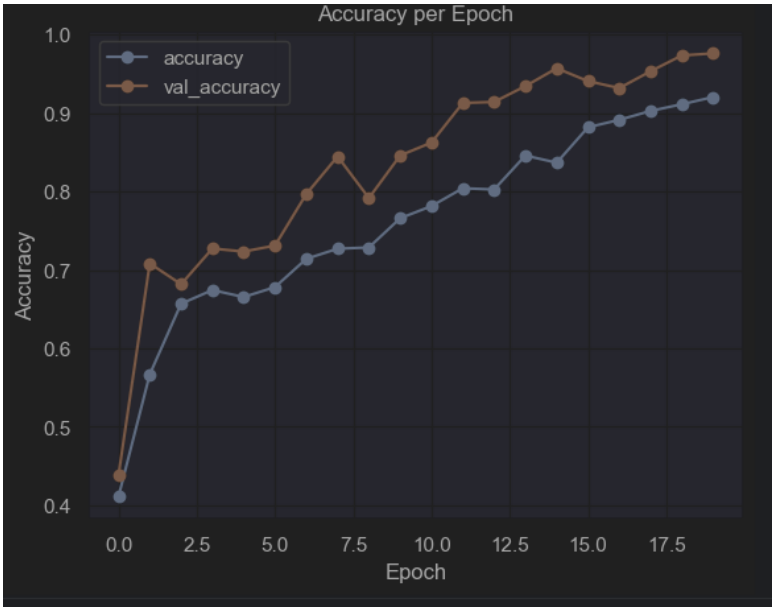
3. **Conclusion** :

- **Résultats globaux** : Ce modèle affiche une bonne généralisation, sans overfitting ou underfitting. La précision et la perte sont optimisées de manière stable.
- la matrice de confusion montre cependant que le modèle donne des résultats faussement prédits:



▼ Pour le dernier modèle:





L’analyse des courbes de **perte** et **précision** obtenues avec l'optimiseur SGD (Stochastic Gradient Descent) avec momentum est donné comme suit:

---

1. **Graphique de la perte (Loss)**

- **Observation** :
  - La courbe `loss` (entraînement) et `val_loss` (validation) diminuent de façon constante, suggérant une bonne convergence du modèle.
  - La `val_loss` reste légèrement en dessous de la `loss`, ce qui est un bon signe car cela indique que le modèle généralise bien sur les données de validation.
  - Pas de signe évident de sur-apprentissage (overfitting), car les deux courbes suivent des tendances similaires.
- **Analyse** :
  - L'utilisation de l'optimiseur SGD avec momentum semble aider à une descente efficace du gradient, ce qui est visible par une convergence régulière.
  - Le modèle est bien entraîné avec des données équilibrées et un bon choix d'hyperparamètres comme le taux d'apprentissage.

---

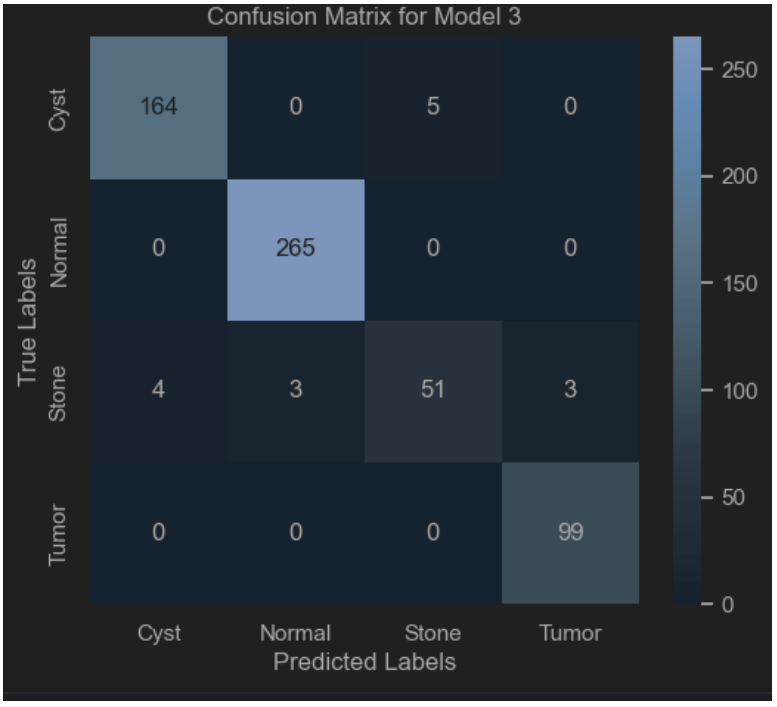
2. **Graphique de la précision (Accuracy)**

- **Observation** :
  - La `accuracy` (entraînement) et `val_accuracy` (validation) augmentent régulièrement.

- La précision sur les données de validation est légèrement au-dessus de celle de l'entraînement, ce qui indique un bon niveau de généralisation.
- À la fin de l'entraînement, la précision dépasse les **90%** pour les données de validation.

- **Analyse** :
  - Le modèle parvient à améliorer ses performances de manière cohérente sur les données d'entraînement et de validation.
  - Il n'y a pas de signe de sous-apprentissage (underfitting), car la `accuracy` est élevée dans les deux cas.
  - Le momentum dans SGD a probablement aidé à surmonter des

La matrice de confusion a ainsi donné:



> Les résultats d’utilisation de l’optimiseur SGD avec Momentum:

	Model	Test Accuracy	Validation Accuracy	Test Loss	Validation Loss
0	Model 1	100.00%	100.00%	0.00%	0.00%
1	Model 2	98.48%	98.74%	0.05%	0.05%
2	Model 3	97.57%	97.60%	0.08%	0.08%

```
Model 1 - Precision: 1.0, Recall: 1.0, F1 Score: 1.0
Model 2 - Precision: 0.9828389522259267, Recall: 0.9688800410742403, F1 Score: 0.9754093788907008
Model 3 - Precision: 0.9615747418370333, Recall: 0.9516199437384809, F1 Score: 0.9561336873457245
```

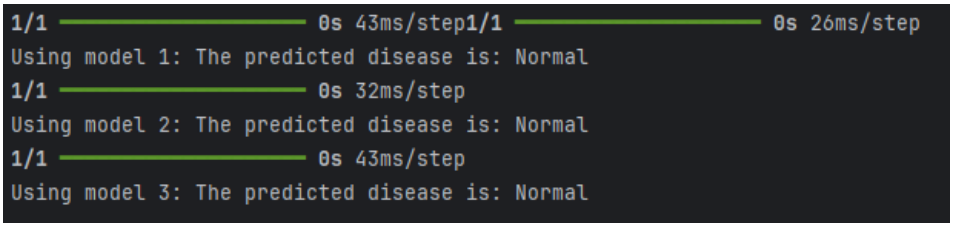
Bilans :

- En utilisant **SGD**, nous avons observé une convergence lente et le test montre des résultats incorrectes.

```
import numpy as np
from keras.src.utils import load_img, img_to_array

# Mapping class indices to disease labels
class_labels = {0: 'Cyst', 1: 'Normal'
                           , 2: 'Stone', 3: 'Tumor'}

def predict_disease(model, image_path):
    """
    Predicts the disease type for a given input image.

    Parameters:
    - model: Trained model for prediction.
    - image_path: Path to the input image.

    Returns:
    - Predicted disease label.
    """
    # Load and preprocess the image
    img = load_img(image_path, target_size=(32, 32)
                                           , color_mode='gr
    img_array = img_to_array(img)
    img_array = img_array / 255.0
    img_array = np.expand_dims(img_array, axis=0)

    # Make prediction
    predictions = model.predict(img_array)
    predicted_class = np.argmax(predictions,axis=1)[0]

    # Return the corresponding disease label
```

```
    return class_labels[predicted_class]

# Example usage
image_path
='CT-KIDNEY-DATASET-Normal-Cyst-Tumor-Stone'
                                        '/Tumor/Tumor- (562).
img = cv2.imread(str(image_path))

## Using the Model 1
predicted_disease_m1
= predict_disease(model1, image_path)
print(f"Using model 1: The predicted disease is:
{predicted_disease_m1}")

## Using the Model 2
predicted_disease_m2
= predict_disease(model2, image_path)
print(f"Using model 2: The predicted disease is:
{predicted_disease_m2}")

## Using the Model 3
predicted_disease_m3
= predict_disease(model3, image_path)
print(f"Using model 3: The predicted disease is:
{predicted_disease_m3}")
```

Résultat:

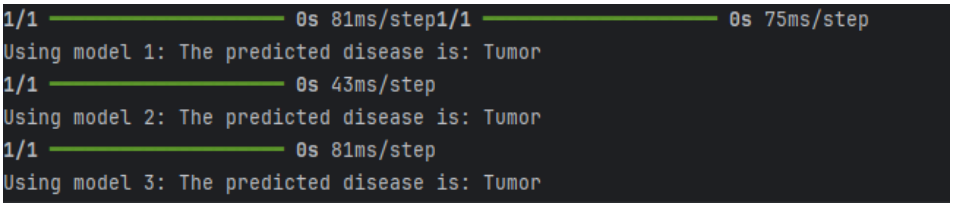


- **SGD** avec Momentum à donné également pour le deuxième modèle des résultats erronés:

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step
Using model 1: The predicted disease is: Tumor
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
Using model 2: The predicted disease is: Normal
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
Using model 3: The predicted disease is: Tumor
```

- **Adam** a fourni des résultats plus rapides, correctes et stables.

avec l'exécution du même code nous avons obtenu des résultats correctes:



---

5. Conclusion et Critique du Modèle

5.1 Avantages

- **Modèle performant** avec une bonne précision sur le jeu de validation.
- **Réduction du surapprentissage** grâce aux couches de dropout.
- **Amélioration de la capacité à extraire des caractéristiques** grâce à l'ajout de couches de convolution.

5.2 Limitations

- **Temps d'entraînement élevé** pour les modèles plus profonds.
- **Nécessité de plus de données** pour améliorer la généralisation.
- **Possibilité de surajuster le modèle** avec des couches trop profondes.

---

6. Accélération par GPU

L'entraînement du modèle sur GPU a considérablement réduit le temps nécessaire pour chaque époque par presque la moitié du temps. Le calcul de l'exactitude et de la perte a été effectué à l'aide de Google Colab, où l'usage du GPU a permis de réduire de manière significative le temps de calcul.

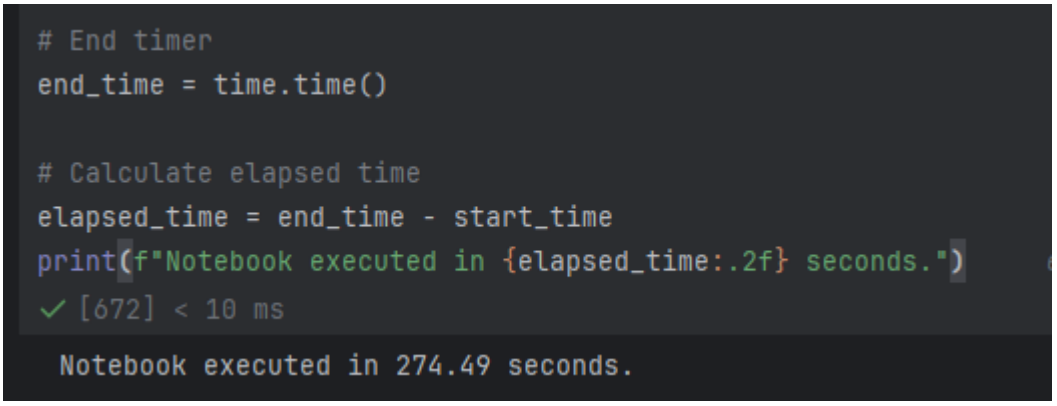
Voici le résultat de calcul:

Sur google Collab l’exécution a pris **124.24** secondes

```
# End timer
end_time = time.time()

# Calculate elapsed time
elapsed_time = end_time - start_time
print(f"Notebook executed in {elapsed_time:.2f} seconds.")
```

```
Notebook executed in 124.24 seconds.
```

Sur la machine locale l’exécution a pris 274.49.



Conclusion

Ce TP nous a permis d'explorer plusieurs architectures CNN et d'améliorer les performances du modèle en ajustant des hyperparamètres comme la taille du lot et l'optimiseur. L'ajout de couches de convolution supplémentaires a permis d'améliorer l'extraction des caractéristiques des images, mais cela a également augmenté le temps de calcul.