Learning Module

Workshop

“Deep Learning and Its Applications in Assisting Human”

2024

Contents

CHAPTER 1

Introduction to Image Segmentation

Image segmentation is a critical process in image analysis, serving as a foundational step for various applications in computer vision, medical imaging, and artificial intelligence. It involves partitioning an image into distinct segments, making it easier to analyze and interpret the content of the image. The effectiveness of image segmentation significantly influences the performance of subsequent tasks such as object detection, recognition, and classification [1].

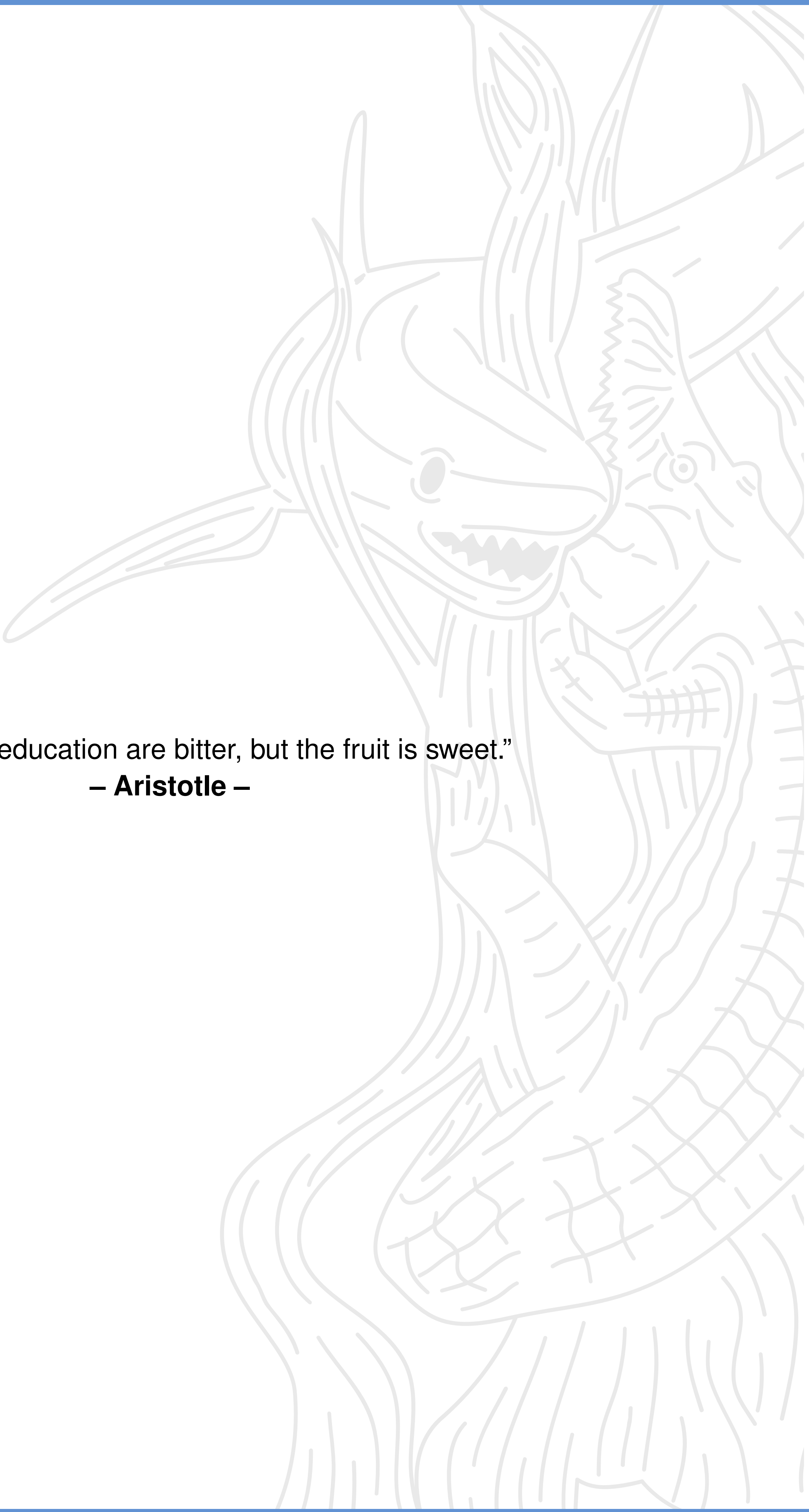
Numerous techniques have been developed for image segmentation, each with its advantages and limitations. Traditional methods include thresholding, edge detection, and region-based segmentation, which have been widely used but often struggle with challenges like noise and varying illumination conditions [2]. For instance, thresholding techniques can fail in the presence of shadows or highlights, while edge detection may not accurately capture the boundaries of objects in complex scenes [3].

Recent advancements in segmentation techniques have seen the integration of machine learning and deep learning approaches, which have shown remarkable improvements in accuracy and efficiency. For example, fuzzy C-means clustering has been effectively applied in medical image segmentation, particularly for MRI scans, where it helps in delineating structures such as tumors from surrounding tissues [4]. Moreover, hybrid approaches that combine different segmentation methods, such as fuzzy clustering with level set methods, have been proposed to enhance segmentation performance by leveraging the strengths of each technique [5].

In the medical field, accurate image segmentation is vital for diagnosis and treatment planning. Techniques such as the U-Net architecture have been specifically designed for medical image segmentation, demonstrating superior performance in tasks like tumor detection and organ delineation [6, 7]. The ability to segment medical images accurately not only aids in visualizing anatomical structures but also plays a crucial role in computer-aided diagnosis systems, enhancing the overall efficiency and accuracy of medical evaluations [8].

Despite the progress made, challenges remain in the field of image segmentation. The variability in image quality, the presence of noise, and the need for high-quality annotated datasets for training deep learning models continue to pose significant hurdles [9, 10]. Future research is likely to focus on developing more robust algorithms that can handle these challenges while improving the interpretability and usability of segmentation results across various applications [11, 12].

Image segmentation is a dynamic and evolving field that is essential for effective image analysis. The integration of traditional methods with advanced machine learning techniques is paving the way for more accurate and efficient segmentation solutions, particularly in complex domains such as medical imaging and autonomous systems.



“The roots of education are bitter, but the fruit is sweet.”
– Aristotle –

CHAPTER 2
Image Segmentation Basics

In this chapter, we will cover the basics of image segmentation, including common techniques and approaches. We'll also explain the theory behind these methods and how they are used in practical applications.

2.1 WHAT IS IMAGE SEGMENTATION?

- **Definition**: Divides an image into distinct regions for easier analysis.
- **Goal**: Separates meaningful areas, like objects or backgrounds, for clearer interpretation.
- **Use**:
  - Medical imaging (e.g., identifying tumors or organs in scans)
  - Object detection (e.g., separating cars from pedestrians in traffic)
  - Satellite imagery (e.g., land use classification)
  - Robotics and autonomous driving

2.2 COMMON TECHNIQUES IN IMAGE SEGMENTATION

Image segmentation employs various techniques, from simple methods to advanced models. Key approaches include:

a. Thresholding

- Converts images to binary (black and white) based on a threshold.
- Pixels above threshold: one class; below threshold: another class.
- Useful for basic object-background separation.

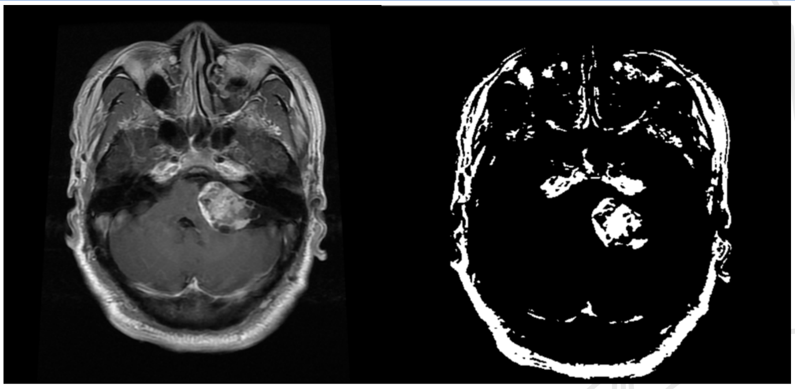


Figure 2.1: Example of Thresholding

b. Edge Detection

- Identifies object boundaries by detecting sharp pixel intensity changes.
- Common methods: Sobel operator, Canny edge detector.
- Useful for highlighting object outlines.

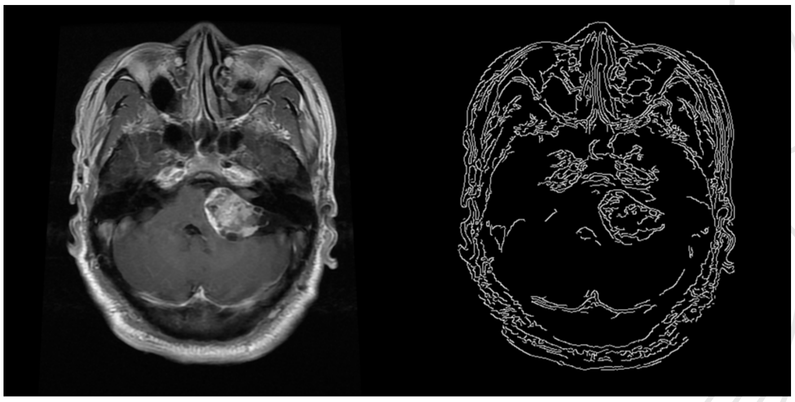


Figure 2.2: Example of Edge Detection

c. Region-Based Methods

- Groups pixels with similar properties (color, intensity) into regions.
- Example: Watershed algorithm treats an image like a topographic map.
- Effective for complex images needing more than basic separation.

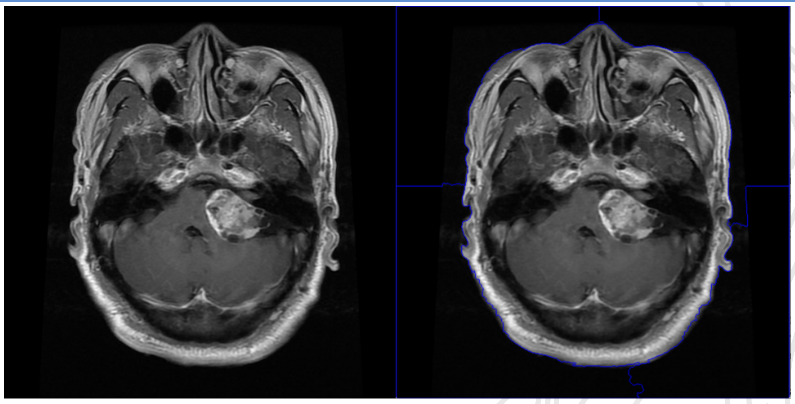


Figure 2.3: Example of Region-Based Methods

d. Machine Learning-Based Segmentation

- Uses neural networks, like U-Net, for precise segmentation.
- Processes images in layers to predict segmentation masks.
- Widely applied in fields like medical imaging.

2.3 APPLICATIONS OF IMAGE SEGMENTATION

Image segmentation is widely used in fields, especially in medical imaging. It helps:

- Segments organs or tumors in MRI or CT scans.
- Assists in diagnosis and treatment planning.

As shown in Figure 1 and Figure 2, the segmentation results demonstrate the application of segmentation techniques on various types of medical imaging data. Figure 1 illustrates the segmentation of target areas in MRI scans using color masks to differentiate relevant objects, while Figure 2 shows segmentation on panoramic dental images, highlighting individual tooth structures separately. These examples emphasize the importance of image segmentation in facilitating the analysis and interpretation of medical data for diagnosis and treatment planning.

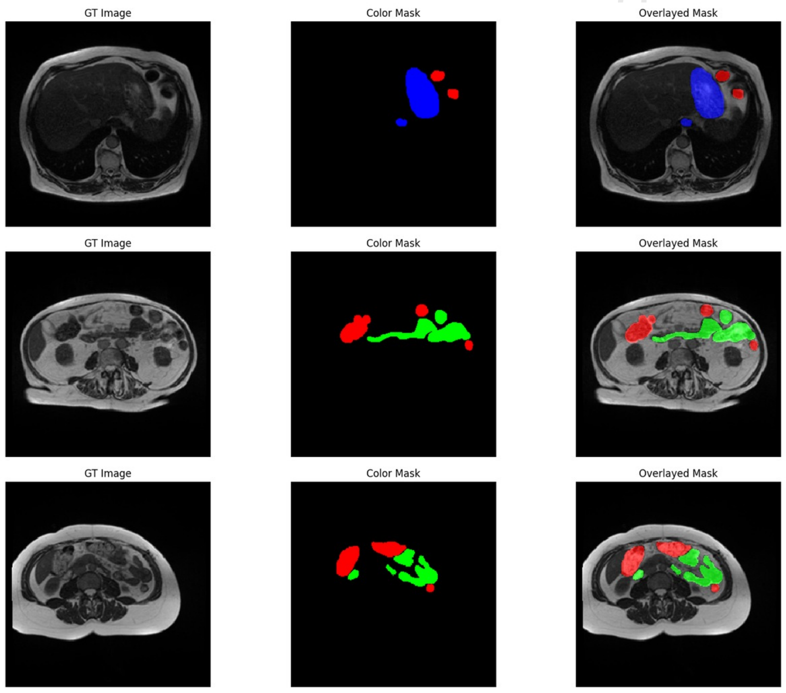


Figure 2.4: Example of Segmentation on MRI Scans with Color Mask Overlays

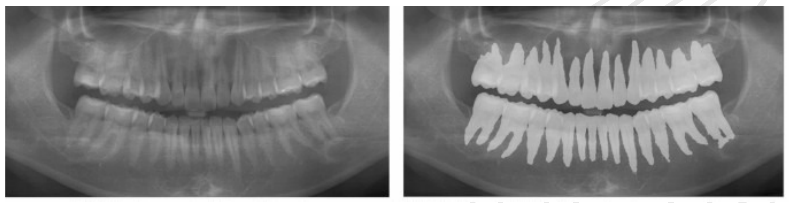


Figure 2.5: Segmentation of Panoramic Dental Images for Identifying Tooth Structures.

CHAPTER 3 Environment Setup for Image Segmentation in Google Colab

In this chapter, we will guide you through setting up your development environment for image segmentation using Google Colab.

3.1 STEP 1 : ENABLE GPU SUPPORT IN GOOGLE COLAB

To leverage GPU acceleration, ensure that the runtime is properly configured. Follow these steps:

1. Click on **Runtime** in the menu.
2. Select **Change runtime type**.
3. Choose **GPU** as the hardware accelerator.
4. Click **Save**.

With this setup, Colab will utilize GPU resources where available, significantly speeding up model training.

3.2 STEP 2: VERIFY TENSORFLOW AND CUDA SETUP

Since TensorFlow is pre-installed in Google Colab, we only need to confirm that it is accessible and that the GPU is detected.

3.2.1 Verify TensorFlow Installation

Run the following code snippet to confirm that TensorFlow is installed and functioning:

```
import tensorflow as tf
print("TensorFlow Version:", tf.__version__)
```

3.2.2 Check if GPU is Detected

Execute the command below to verify if TensorFlow can detect the GPU:

```
if tf.config.list_physical_devices('GPU'):
print("GPU is available and ready to use.")
else:
print("GPU not detected.")
```

3.3 STEP 3: INSTALL ADDITIONAL DEPENDENCIES

While TensorFlow and many essential libraries are included in Colab, you may need to install additional packages. You can do this using `pip` directly within Colab.
To install necessary packages, run:

```
!pip install -q tensorflow-addons matplotlib
```

3.4 STEP 4: VERIFY CPU AND GPU COMPUTATION

Run a simple computation to verify that the environment is set up correctly. This will test both CPU and GPU performance.

3.4.1 Test TensorFlow Computation on CPU

```
import tensorflow as tf

print("Testing CPU computation...")
cpu_result = tf.reduce_sum(tf.random.normal([1000, 1000]))
print("CPU computation result:", cpu_result.numpy())
```

3.4.2 Test GPU Computation (if available)

```
if tf.config.list_physical_devices('GPU'):
with tf.device('/GPU:0'):
print("Testing GPU computation...")
gpu_result = tf.reduce_sum(tf.random.normal([1000, 1000]))
print("GPU computation result:", gpu_result.numpy())
else:
print("GPU not detected, skipping GPU test.")
```

3.5 STEP 5: SAVE AND SHARE NOTEBOOKS

Google Colab allows you to save notebooks directly to Google Drive. To access saved notebooks:

1. Click on **File** in the menu.
2. Select **Save a copy in Drive**.
3. Access it from your Google Drive at any time.

3.6 STEP 6: INSTALL AND USE JUPYTER WIDGETS (OPTIONAL)

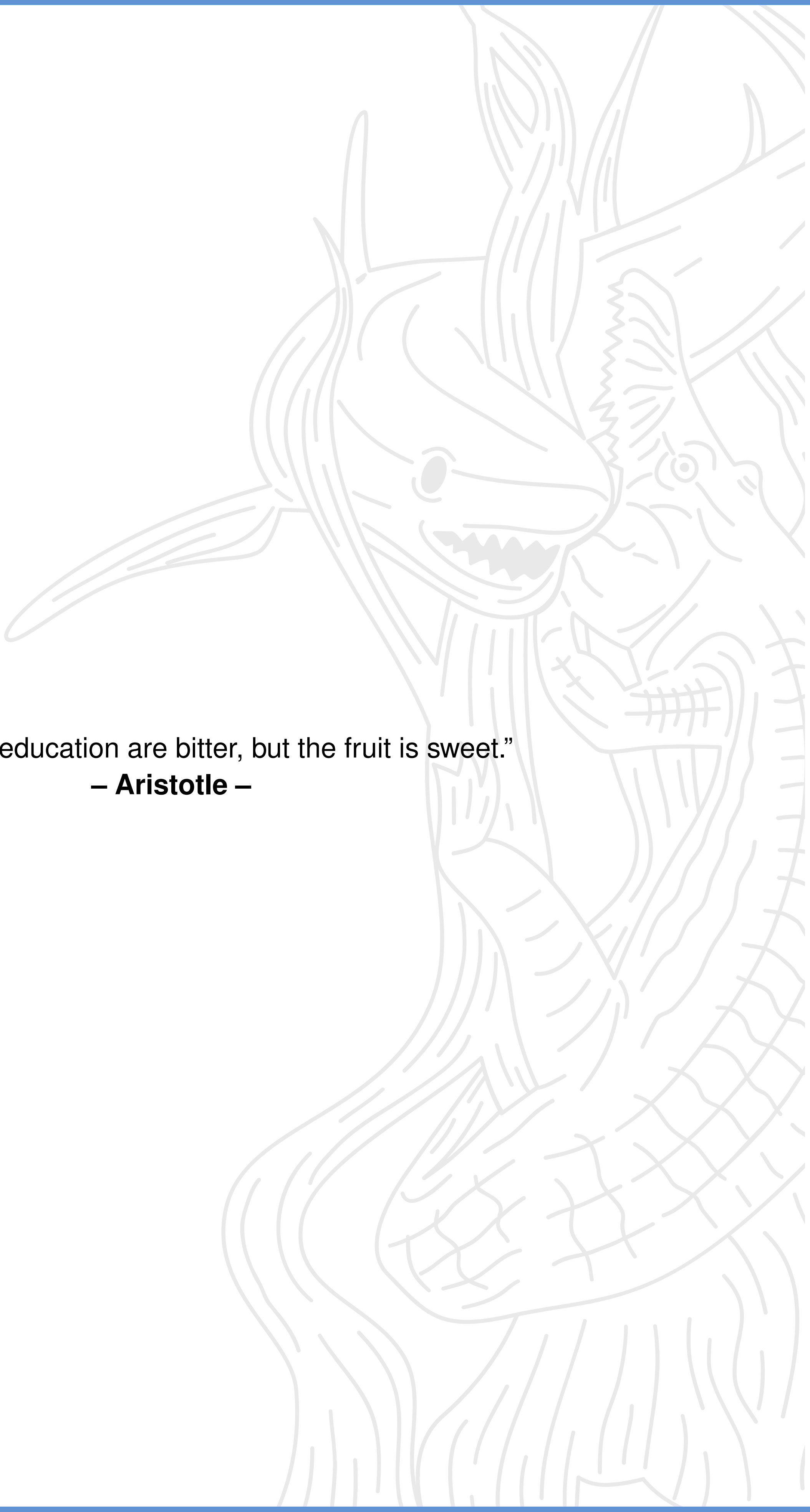
Some projects may require Jupyter widgets or additional features not enabled by default. You can install these using:

```
!pip install ipywidgets
```

To enable widgets, run:

```
from google.colab import output
output.enable_custom_widget_manager()
```

This chapter provides a streamlined way to set up and validate your environment on Google Colab. With its pre-configured settings, you can avoid the complexities of manual installations such as CUDA and cuDNN setups, allowing you to focus on developing your image segmentation models.



“The roots of education are bitter, but the fruit is sweet.”
– Aristotle –

CHAPTER 4

Evaluation Metrics for Image Segmentation

Proper evaluation of image segmentation is critical for assessing the performance of segmentation algorithms and ensuring their reliability in practical applications. Accurate evaluation metrics provide insights into the effectiveness of segmentation methods, guiding researchers and practitioners in selecting the most suitable approaches for their specific tasks. Among the most commonly used metrics for evaluating segmentation performance are Intersection over Union (IoU), Dice Coefficient, and pixel accuracy.

Intersection over Union (IoU) measures the overlap between the predicted segmentation and the ground truth. It is defined as the area of overlap divided by the area of union between the predicted and ground truth regions. Mathematically, it can be expressed as:

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} \tag{4.1}$$

where A is the predicted segmentation and B is the ground truth. IoU is particularly useful for evaluating segmentation tasks where precise boundary delineation is crucial, such as in medical imaging and autonomous driving [13]. A higher IoU score indicates better performance, with a score of 1.0 representing perfect overlap.

Dice Coefficient, similar to IoU, quantifies the similarity between two sets. It is defined as twice the area of overlap divided by the sum of the areas of the predicted and ground truth regions, given by:

$$\text{Dice} = \frac{2|A \cap B|}{|A| + |B|} \tag{4.2}$$

This metric is particularly sensitive to small changes in segmentation and is often preferred in medical applications where the detection of small structures is critical [14]. The Dice Coefficient ranges from 0 to 1, with 1 indicating perfect agreement between the predicted and ground truth segmentations.

Pixel Accuracy calculates the ratio of correctly classified pixels to the total number of pixels in the image. It is defined as:

$$\text{Pixel Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.3}$$

where TP is true positives, TN is true negatives, FP is false positives, and FN is false negatives. While pixel accuracy provides a straightforward measure of overall segmentation

performance, it can be misleading in cases of class imbalance, where one class significantly outnumbers another [15].
In addition to these common metrics, advanced metrics such as **Precision**, **Recall**, and **F1 Score** are also essential in segmentation tasks. Precision measures the proportion of true positive predictions among all positive predictions:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.4}$$

Recall assesses the proportion of true positives among all actual positives:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4.5}$$

The F1 score combines both precision and recall into a single metric, providing a balanced measure of segmentation performance:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.6}$$

This metric is particularly useful in scenarios where both false positives and false negatives carry significant consequences, such as in medical diagnostics [16].
Furthermore, the **Confusion Matrix** is a valuable tool for pixel classification, allowing for a detailed breakdown of true positive, false positive, true negative, and false negative counts. The confusion matrix provides insights into the types of errors made by the segmentation algorithm, facilitating targeted improvements [17]. By analyzing the confusion matrix, practitioners can identify specific classes that are frequently misclassified and adjust their models or preprocessing techniques accordingly.
The evaluation of image segmentation is a multifaceted process that requires a combination of metrics to accurately assess performance. The choice of metrics can significantly impact the interpretation of results and the subsequent development of segmentation algorithms. Therefore, understanding and applying the appropriate evaluation metrics is essential for advancing the field of image segmentation.

CHAPTER 5
Building a Simple U-Net for Image Segmentation

5.1 INTRODUCTION TO U-NET

- **Purpose**: U-Net is a convolutional neural network (CNN) used mainly for biomedical image segmentation.
- **Difference from Traditional CNNs**: Unlike typical CNNs for classification, U-Net enables both localization and classification of objects within an image.
- **Goal for Implementation**: This chapter will implement a simple U-Net model in Python using Keras for binary segmentation, identifying each pixel as background or as part of the object of interest.

5.2 ARCHITECTURE OF U-NET

The U-Net consists of two primary parts:

- **Structure**: U-Net has a unique encoder-decoder design, where the encoder captures context, and the decoder precisely locates objects.
- **Encoder (Contraction Path):** Extracts features using multiple convolutional and pooling layers, reducing the spatial dimension.
- **Decoder (Expansion Path):** Uses transposed convolutions to restore the spatial dimensions, combining high-level features with finer details through skip connections.

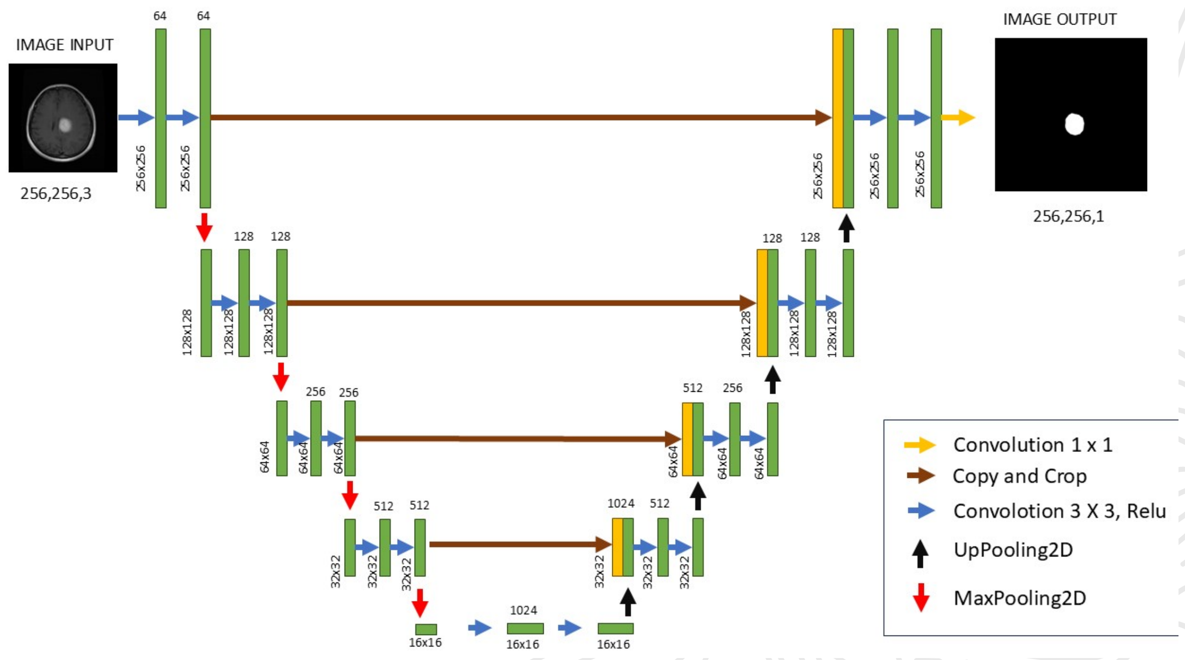


Figure 5.1: U-NET Architecture

5.3 CODE WALKTHROUGH

5.3.1 Importing Libraries

We start by importing the necessary libraries :

1. **Importing numpy**:
   Numpy is imported to handle numerical computations, particularly with arrays and matrices. It is typically imported as `np`.

2. **Importing os**:
   The `os` module is used for interacting with the operating system, such as handling file paths and directory operations.

3. **Importing cv2**:
   OpenCV is imported as `cv2` to handle advanced image and video processing tasks, including reading, writing, and transforming images.

4. **Importing pandas**:
   Pandas is used for data manipulation and analysis, often employed in handling tabular data related to image annotations or model performance.

5. **Importing random**:
   The `random` module is used for generating random numbers, often employed in data augmentation or shuffling datasets.

6. **Importing PIL.Image**:
   From the Python Imaging Library (PIL), the `Image` module is imported to handle image processing tasks like opening, manipulating, and saving images.

7. **Importing resize from skimage.transform**:
   The `resize` function from `skimage.transform` is used to resize images to specific dimensions while maintaining aspect ratio. It is often used for preprocessing images for machine learning models.

8. **Importing matplotlib.pyplot**:
   `matplotlib.pyplot` is a plotting library used to visualize data. It is often used for displaying images and their transformations during data analysis or debugging.

9. **Importing tensorflow.keras model-related modules**:
   The following modules from `tensorflow.keras` are used to define neural networks:

   - `Model`: A base class for defining neural networks.
   - `Input`: Used to define the input layer of a model.
   - `Conv2D`: A layer for applying 2D convolution to an input.
   - `MaxPooling2D`: A layer for down-sampling feature maps using maximum pooling.
   - `UpSampling2D`: Used for up-sampling input data, often in decoder architectures.
   - `Concatenate`: Combines layers or tensors along a specific axis.
   - `Conv2DTranspose`: Performs transposed convolution (deconvolution) for up-sampling.
   - `Dropout`: A regularization technique to prevent overfitting by randomly deactivating neurons during training.

10. **Importing tensorflow.keras.optimizers.Adam**:
    `Adam` is a popular optimizer that combines the benefits of AdaGrad and RMSProp. It is used to adjust the weights of a model during training.

11. **Importing tensorflow.keras.preprocessing.image.ImageDataGenerator**:
    `ImageDataGenerator` provides functionality for image augmentation such as rotation, zoom, or flipping. This helps increase the diversity of the training dataset.

12. **Importing sklearn's train_test_split**:
    The `train_test_split` function from `sklearn.model_selection` is used to split data into training and testing sets. This is useful for evaluating the performance of a machine learning model.

```python
import numpy as np
import os
import cv2
import pandas as pd
import random
from PIL import Image
from skimage.transform import resize
import matplotlib.pyplot as plt
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D,
    UpSampling2D, concatenate, Conv2DTranspose, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import MeanIoU
from sklearn.model_selection import train_test_split
```

5.3.2 Model Architecture

The following function defines the U-Net model:

```
    def simple_unet_model(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS):
    inputs = Input((IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS))
    s = inputs
```

This initializes the input layer with the specified image dimensions.

Encoder (Contraction Path)

The encoder path extracts features through convolutional layers and reduces spatial dimensions with pooling:

```
    c1 = Conv2D(16, (3, 3), activation='relu', padding='same')(s)
    p1 = MaxPooling2D((2, 2))(c1)

    c2 = Conv2D(32, (3, 3), activation='relu', padding='same')(p1
       )
    p2 = MaxPooling2D((2, 2))(c2)

    c3 = Conv2D(64, (3, 3), activation='relu', padding='same')(p2
       )
    p3 = MaxPooling2D((2, 2))(c3)

    c4 = Conv2D(128, (3, 3), activation='relu', padding='same')(
       p3)
    p4 = MaxPooling2D((2, 2))(c4)

    c5 = Conv2D(256, (3, 3), activation='relu', padding='same')(
       p4)
```

Decoder (Expansion Path)

The decoder restores the original spatial dimensions using transposed convolutions and skip connections:

```
    u6 = Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='
       same')(c5)
    u6 = concatenate([u6, c4])

    u7 = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='
       same')(u6)
    u7 = concatenate([u7, c3])

    u8 = Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='
       same')(u7)
    u8 = concatenate([u8, c2])

    u9 = Conv2DTranspose(16, (2, 2), strides=(2, 2), padding='
       same')(u8)
    u9 = concatenate([u9, c1])

    outputs = Conv2D(1, (1, 1), activation='sigmoid')(u9)
```

5.3.3 Compiling the Model

We compile the model using the Adam optimizer and binary cross-entropy loss.

```
    model = Model(inputs=[inputs], outputs=[outputs])
    model.compile(optimizer=Adam(lr=1e-3), loss='binary_crossentropy', metrics=['accuracy'])
```

5.4 MODEL SUMMARY

The model summary provides an overview of the architecture:

```
    model.summary()
```

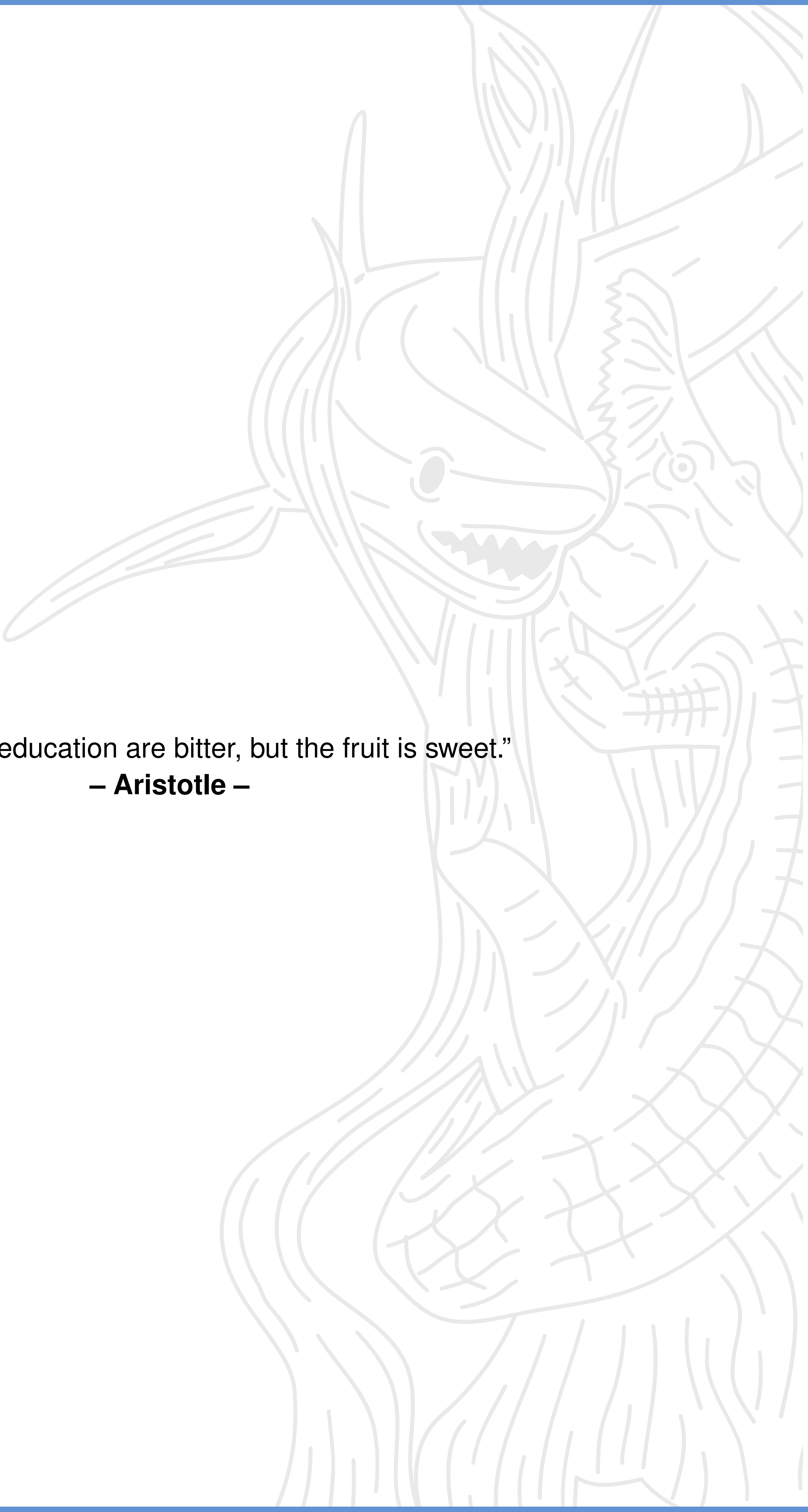
5.5 POTENTIAL IMPROVEMENTS

- **Kernel Initialization:** Use alternative initializers like `he_normal` to improve convergence.
- **Loss Function:** Use `categorical_crossentropy` for multiclass segmentation tasks.
- **Metrics:** Measure performance with the MeanIoU metric for better insights.

5.6 COMPLETE CODE

Here is the complete code for the U-Net model:

```
"""
    Simple Unet model
"""
# u-net model

from keras.models import Model
from keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D, concatenate, Conv2DTranspose, BatchNormalization, Dropout, Lambda
from keras.optimizers import Adam
from keras.metrics import MeanIoU

kernel_initializer =  'he_uniform'  # also try 'he_normal' but model not converging...


################################################################
def simple_unet_model(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS):
    #Build the model
    inputs = Input((IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS))
    #s = Lambda(lambda x: x / 255)(inputs)   #No need for this if
        we normalize our inputs beforehand
    s = inputs

    #Contraction path
    c1 = Conv2D(16, (3, 3), activation='relu', kernel_initializer
       =kernel_initializer, padding='same')(s)
    c1 = Dropout(0.1)(c1)
    c1 = Conv2D(16, (3, 3), activation='relu', kernel_initializer
       =kernel_initializer, padding='same')(c1)
    p1 = MaxPooling2D((2, 2))(c1)

    c2 = Conv2D(32, (3, 3), activation='relu', kernel_initializer
       =kernel_initializer, padding='same')(p1)
    c2 = Dropout(0.1)(c2)
    c2 = Conv2D(32, (3, 3), activation='relu', kernel_initializer
       =kernel_initializer, padding='same')(c2)
    p2 = MaxPooling2D((2, 2))(c2)

    c3 = Conv2D(64, (3, 3), activation='relu', kernel_initializer
       =kernel_initializer, padding='same')(p2)
    c3 = Dropout(0.2)(c3)
    c3 = Conv2D(64, (3, 3), activation='relu', kernel_initializer
       =kernel_initializer, padding='same')(c3)
    p3 = MaxPooling2D((2, 2))(c3)

    c4 = Conv2D(128, (3, 3), activation='relu',
       kernel_initializer=kernel_initializer, padding='same')(p3)
    c4 = Dropout(0.2)(c4)
    c4 = Conv2D(128, (3, 3), activation='relu',
       kernel_initializer=kernel_initializer, padding='same')(c4)
    p4 = MaxPooling2D(pool_size=(2, 2))(c4)

    c5 = Conv2D(256, (3, 3), activation='relu',
       kernel_initializer=kernel_initializer, padding='same')(p4)
    c5 = Dropout(0.3)(c5)
    c5 = Conv2D(256, (3, 3), activation='relu',
       kernel_initializer=kernel_initializer, padding='same')(c5)

    #Expansive path
    u6 = Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='
       same')(c5)
    u6 = concatenate([u6, c4])
    c6 = Conv2D(128, (3, 3), activation='relu',
       kernel_initializer=kernel_initializer, padding='same')(u6)
    c6 = Dropout(0.2)(c6)
    c6 = Conv2D(128, (3, 3), activation='relu',
       kernel_initializer=kernel_initializer, padding='same')(c6)

    u7 = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='
       same')(c6)
    u7 = concatenate([u7, c3])
    c7 = Conv2D(64, (3, 3), activation='relu', kernel_initializer
       =kernel_initializer, padding='same')(u7)
    c7 = Dropout(0.2)(c7)
    c7 = Conv2D(64, (3, 3), activation='relu', kernel_initializer
       =kernel_initializer, padding='same')(c7)

    u8 = Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='
       same')(c7)
    u8 = concatenate([u8, c2])
    c8 = Conv2D(32, (3, 3), activation='relu', kernel_initializer
       =kernel_initializer, padding='same')(u8)
    c8 = Dropout(0.1)(c8)
    c8 = Conv2D(32, (3, 3), activation='relu', kernel_initializer
       =kernel_initializer, padding='same')(c8)

    u9 = Conv2DTranspose(16, (2, 2), strides=(2, 2), padding='
       same')(c8)
    u9 = concatenate([u9, c1], axis=3)
    c9 = Conv2D(16, (3, 3), activation='relu', kernel_initializer
       =kernel_initializer, padding='same')(u9)
    c9 = Dropout(0.1)(c9)
    c9 = Conv2D(16, (3, 3), activation='relu', kernel_initializer
       =kernel_initializer, padding='same')(c9)

    outputs = Conv2D(1, (1, 1), activation='sigmoid')(c9)

    model = Model(inputs=[inputs], outputs=[outputs])
    model.compile(optimizer=Adam(lr = 1e-3), loss='
       binary_crossentropy', metrics=['accuracy'])
    #model.compile(optimizer=Adam(lr = 1e-3), loss='
       binary_crossentropy', metrics=[MeanIoU(num_classes=2)])
    model.summary()

    return model
```



“The roots of education are bitter, but the fruit is sweet.”
– Aristotle –

CHAPTER 6
Preparing Data for U-Net Model

In this chapter, we will walk through the process of preparing the dataset for the U-Net model. This involves loading images and masks, normalizing the data, and splitting it into training and testing sets. We will also visualize some samples to ensure the data is loaded correctly.

6.1 IMPORTING REQUIRED LIBRARIES

First, we need to import the necessary libraries for data processing and visualization.

```
from simple_unet_model import simple_unet_model  # Use normal
    U-Net model
from keras.utils import normalize
import os
import cv2
from PIL import Image
import numpy as np
from matplotlib import pyplot as plt
```

6.2 SETTING UP DIRECTORIES

Next, we define the directories where our images and masks are stored.

```
image_directory = 'data/generated_patches/images/'
mask_directory = 'data/generated_patches/masks/'
```

6.3 IMAGE SIZE CONFIGURATION

We will resize all images and masks to a uniform size of 256 $\times$ 256 pixels.

```
SIZE = 256
image_dataset = []   # Placeholder for images
mask_dataset = []    # Placeholder for masks
```

6.4 LOADING IMAGES

Now, we will load the images from the specified directory, convert them to grayscale, and resize them.

```
    images = os.listdir(image_directory)
    for i, image_name in enumerate(images):
    if (image_name.split('.')[1] == 'png'):
    image = cv2.imread(image_directory + image_name, 0)  # Load
        in grayscale
    image = Image.fromarray(image)
    image = image.resize((SIZE, SIZE))  # Resize to 256x256
    image_dataset.append(np.array(image))
```

6.5 LOADING MASKS

Similarly, we will load the masks from the mask directory, resize them, and store them in the 'mask_dataset'.

```
    masks = os.listdir(mask_directory)
    for i, image_name in enumerate(masks):
    if (image_name.split('.')[1] == 'png'):
    image = cv2.imread(mask_directory + image_name, 0)  # Load in
        grayscale
    image = Image.fromarray(image)
    image = image.resize((SIZE, SIZE))  # Resize to 256x256
    mask_dataset.append(np.array(image))
```

6.6 NORMALIZING THE IMAGES

To ensure our model converges effectively, we need to normalize the images. However, we will simply scale the mask images to the range [0, 1].

```
    # Normalize images
    image_dataset = np.expand_dims(normalize(np.array(
       image_dataset), axis=1), 3)
    mask_dataset = np.expand_dims((np.array(mask_dataset)), 3) /
       255.
```

6.7 SPLITTING THE DATASET

Next, we will split the dataset into training and testing sets using the 'train_test_split' function from Scikit-Learn.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    image_dataset, mask_dataset, test_size=0.10, random_state
    =0)

X_train_quick_test, X_test_quick_test, y_train_quick_test,
    y_test_quick_test = train_test_split(X_train, y_train,
    test_size=0.9, random_state=0)
```

6.8 VISUALIZING SAMPLE IMAGES

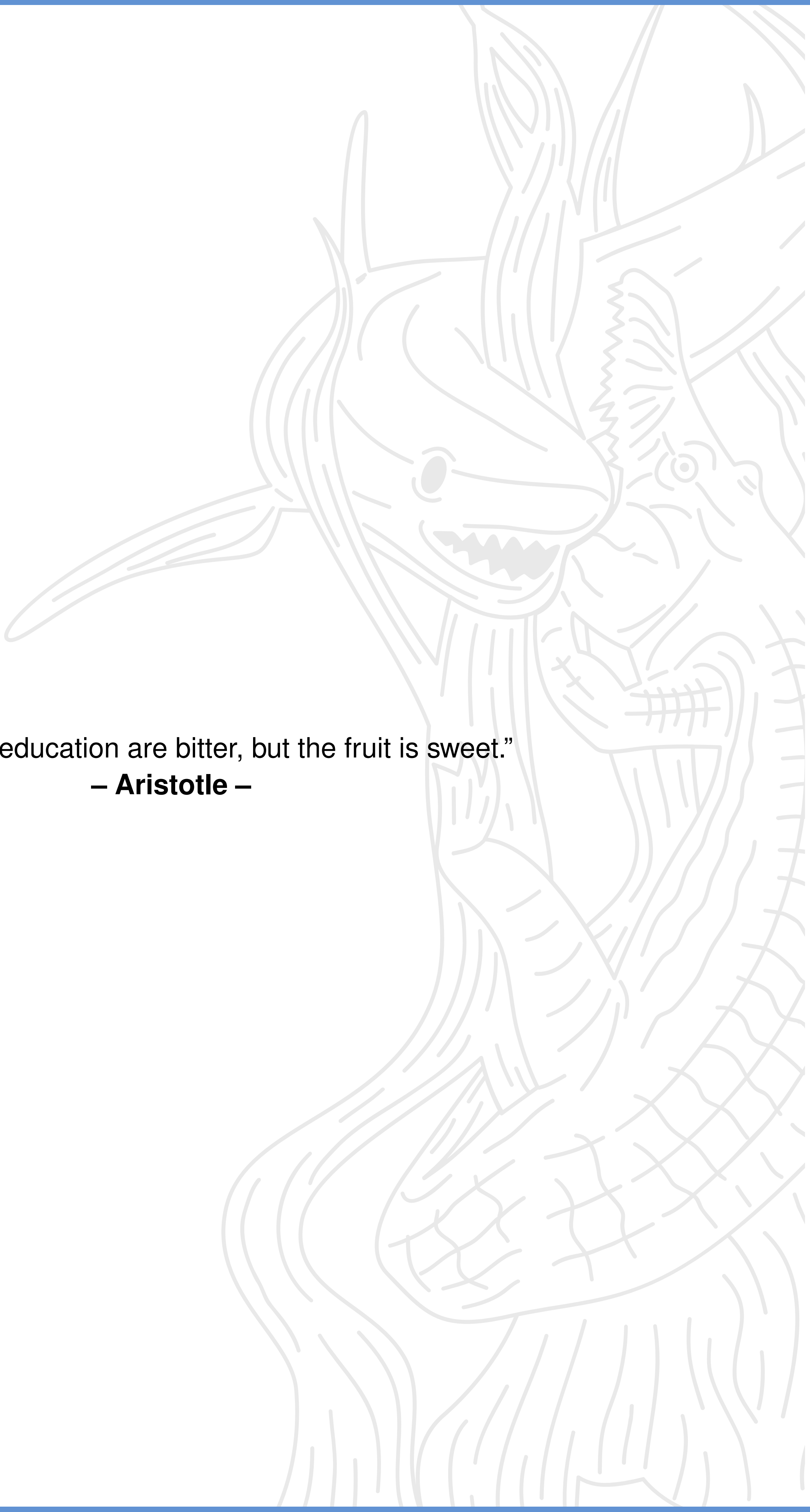
To ensure that our data is loaded correctly, we will visualize a few samples from the training dataset.

```
import random
image_number = random.randint(0, len(X_train_quick_test))
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.imshow(np.reshape(X_train_quick_test[image_number], (256,
     256)), cmap='gray')
plt.title('Input Image')
plt.subplot(122)
plt.imshow(np.reshape(y_train_quick_test[image_number], (256,
     256)), cmap='gray')
plt.title('Ground Truth Mask')
plt.show()
```

6.9 IMAGE DIMENSIONS

Finally, we will extract the dimensions of the images for use in the model.

```
IMG_HEIGHT = image_dataset.shape[1]
IMG_WIDTH = image_dataset.shape[2]
IMG_CHANNELS = image_dataset.shape[3]
```



“The roots of education are bitter, but the fruit is sweet.”
– Aristotle –

CHAPTER 7 Training the U-Net Model

In this chapter, we will train the U-Net model using the dataset we prepared earlier. We will also evaluate the model's performance and visualize the training and validation loss and accuracy over epochs.

7.1 GETTING THE STANDARD MODEL

We start by defining a function to get the standard U-Net model and then create an instance of the model.

```
def get_standard_model():
return simple_unet_model(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS)

model_standard = get_standard_model()
```

7.2 TRAINING THE MODEL

Next, we fit the model to the training data. You can start with pre-trained weights if available by uncommenting the following line.

```
# If starting with pre-trained weights.
# model.load_weights('mitochondria_with_jacard_50_epochs.hdf5
   ')

history_standard = model_standard.fit(X_train_quick_test,
   y_train_quick_test,
batch_size=16,
verbose=1,
epochs=10,
validation_data=(X_test, y_test),
shuffle=False)
```

We save the model after training.

```
model_standard.save('mitochondria_standard.hdf5')
```

7.3 EVALUATING THE MODEL

We can evaluate the model on the test data and print the accuracy.

```python
_, acc = model_standard.evaluate(X_test, y_test)
print("Accuracy of Standard Model is = ", (acc * 100.0), "%")
```

7.4 VISUALIZING TRAINING AND VALIDATION LOSS

We will plot the training and validation loss for each epoch to visualize the model's learning.

```python
loss = history_standard.history['loss']
val_loss = history_standard.history['val_loss']
epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'y', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

7.5 VISUALIZING STANDARD COEFFICIENT

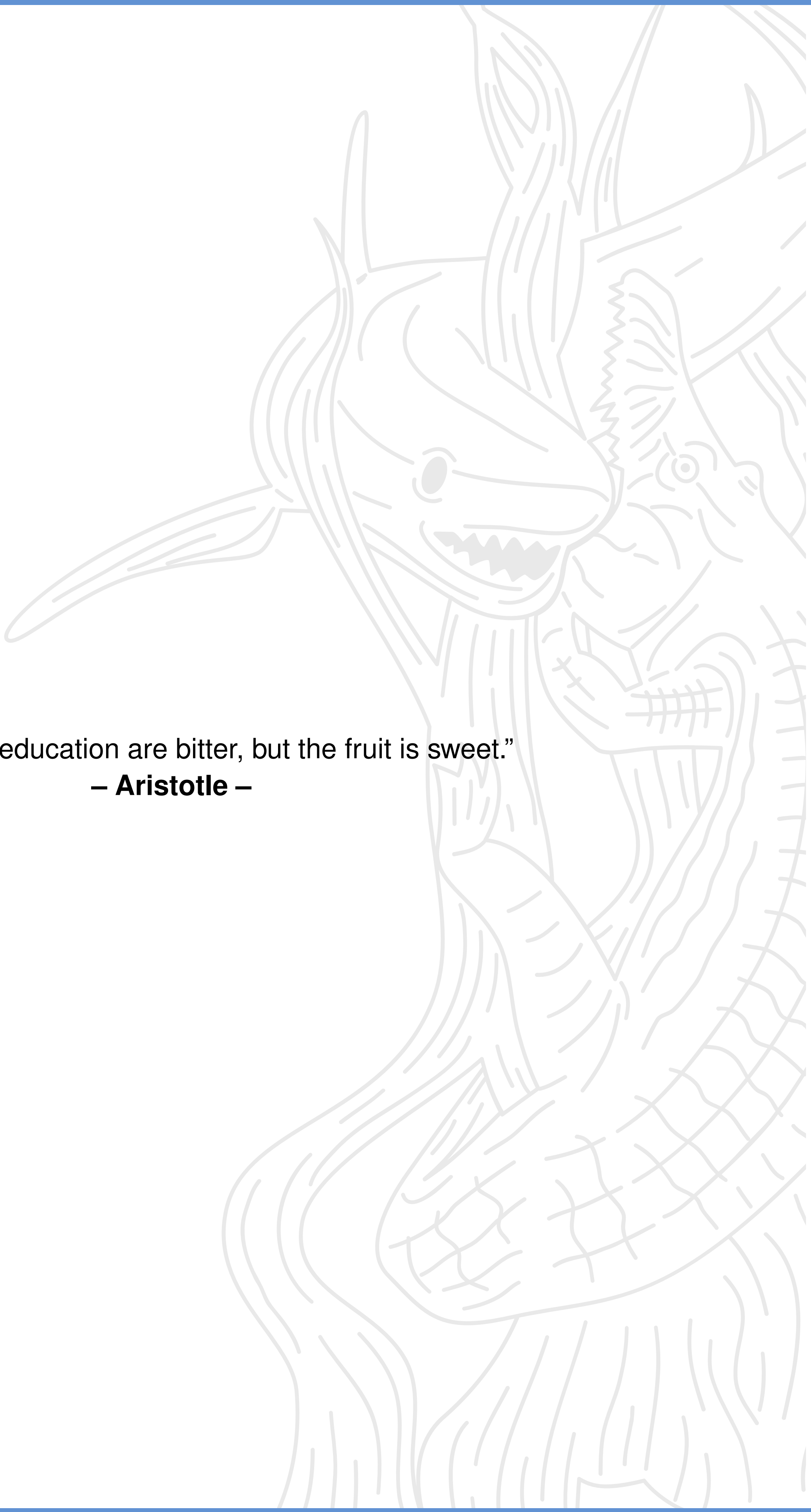
Next, we visualize the training and validation coefficient, which is useful for assessing model performance beyond simple accuracy metrics.

```python
standard_coef = history_standard.history['standard_coef']
val_standard_coef = history_standard.history['
   val_standard_coef']

plt.plot(epochs, standard_coef, 'y', label='Training Standard
    Coeff.')
plt.plot(epochs, val_standard_coef, 'r', label='Validation
   Standard Coeff.')
plt.title('Training and Validation U-Net Coefficient')
plt.xlabel('Epochs')
plt.ylabel('U-Net Coefficient')
plt.legend()
plt.show()
```

7.6 VISUALIZING TRAINING AND VALIDATION ACCURACY

Finally, we plot the training and validation accuracy to see how well the model is performing.

```
    acc = history_standard.history['accuracy']
    val_acc = history_standard.history['val_accuracy']

    plt.plot(epochs, acc, 'y', label='Training Accuracy')
    plt.plot(epochs, val_acc, 'r', label='Validation Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
```



“The roots of education are bitter, but the fruit is sweet.”
– Aristotle –

CHAPTER 8
Evaluating the U-Net Model

In this chapter, we will evaluate the trained U-Net model using Intersection over Union (IoU) as a metric. We will also visualize the model's predictions on both test images from the dataset and external images.

8.1 ASSIGNING THE MODEL

First, we assign our trained model to a variable for easier reference during testing.

```
model = model_standard  # Assign model to one of the models
    instead of changing the entire code for testing.
```

8.2 CALCULATING INTERSECTION OVER UNION (IOU)

To evaluate the model's performance, we calculate the IoU score, which measures the overlap between the predicted segmentation and the ground truth mask.

```
# IOU
y_pred = model.predict(X_test)
y_pred_thresholded = y_pred > 0.5

intersection = np.logical_and(y_test, y_pred_thresholded)
union = np.logical_or(y_test, y_pred_thresholded)
iou_score = np.sum(intersection) / np.sum(union)
print("IoU score is: ", iou_score)
```

8.3 PREDICTING ON A FEW IMAGES

Next, we will test the model on a random image from the test set to visualize its predictions.

```
test_img_number = random.randint(0, len(X_test))
test_img = X_test[test_img_number]
```

```
    ground_truth = y_test[test_img_number]
    test_img_norm = test_img[:,:,0][:,:,None]
    test_img_input = np.expand_dims(test_img_norm, 0)
    prediction = (model.predict(test_img_input)[0,:,:,0] > 0.5).
       astype(np.uint8)
```

We can also predict on an external image to evaluate the model’s generalization.

```
    test_img_other = cv2.imread('data/test_images/01-1_256.jpg',
       0)
    test_img_other_norm = np.expand_dims(normalize(np.array(
       test_img_other), axis=1), 2)
    test_img_other_norm = test_img_other_norm[:,:,0][:,:,None]
    test_img_other_input = np.expand_dims(test_img_other_norm, 0)

    # Predict and threshold for values above 0.5 probability
    prediction_other = (model.predict(test_img_other_input)
       [0,:,:,0] > 0.5).astype(np.uint8)
```

8.4 VISUALIZING PREDICTIONS

Now we will visualize the test image, ground truth, and model predictions, including the external image and its prediction.

```
    plt.figure(figsize=(16, 8))
    plt.subplot(231)
    plt.title('Testing Image')
    plt.imshow(test_img[:,:,0], cmap='gray')
    plt.subplot(232)
    plt.title('Testing Label')
    plt.imshow(ground_truth[:,:,0], cmap='gray')
    plt.subplot(233)
    plt.title('Prediction on Test Image')
    plt.imshow(prediction, cmap='gray')
    plt.subplot(234)
    plt.title('External Image')
    plt.imshow(test_img_other, cmap='gray')
    plt.subplot(235)
    plt.title('Prediction of External Image')
    plt.imshow(prediction_other, cmap='gray')
    plt.show()
```

Finally, we can save the prediction of the external image.

```
    plt.imsave('data/results/output2.jpg', prediction_other, cmap
       ='gray')
```

CHAPTER 9
Results of Image Segmentation

This section highlights the outcomes of the segmentation model applied to test MRI scans. The figures below show the testing image, ground truth mask, predicted mask, and the overlay of the prediction on the original image.

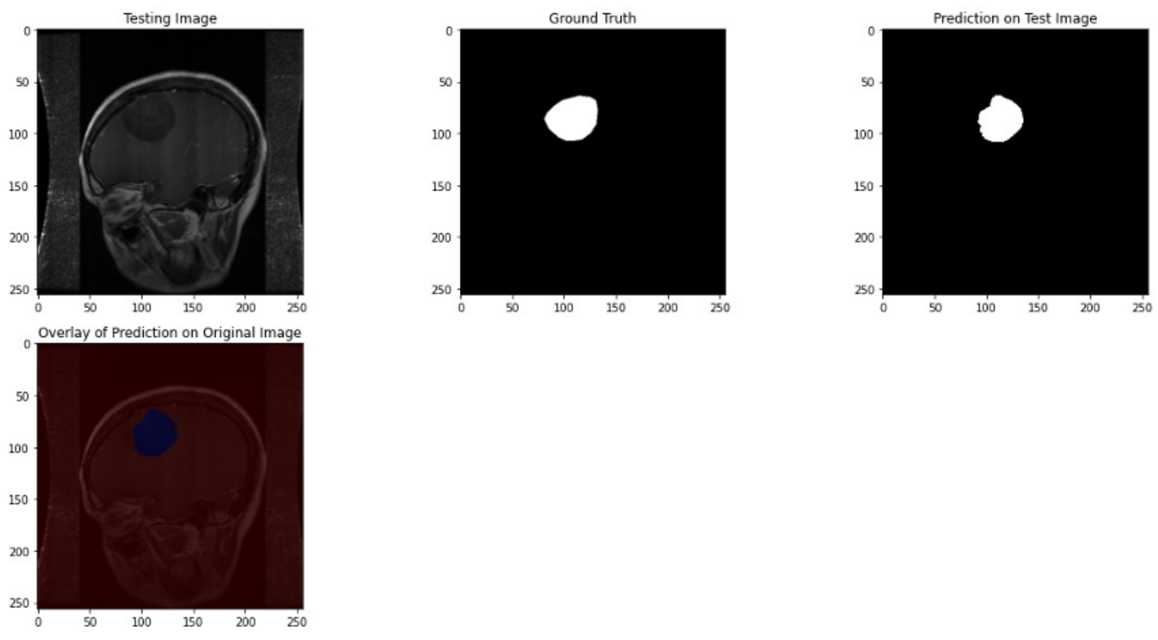


Figure 9.1: MRI Scan 1- Segmentation Result

In Figure **??**, the segmentation results are displayed. The details are as follows:

- **Top-Left:** Original testing image from the MRI scan.
- **Top-Right:** Ground truth mask representing the true target region.
- **Bottom-Right:** Predicted mask generated by the segmentation model.
- **Bottom-Left:** Overlay of the predicted mask on the original image, allowing for a visual comparison with the ground truth.

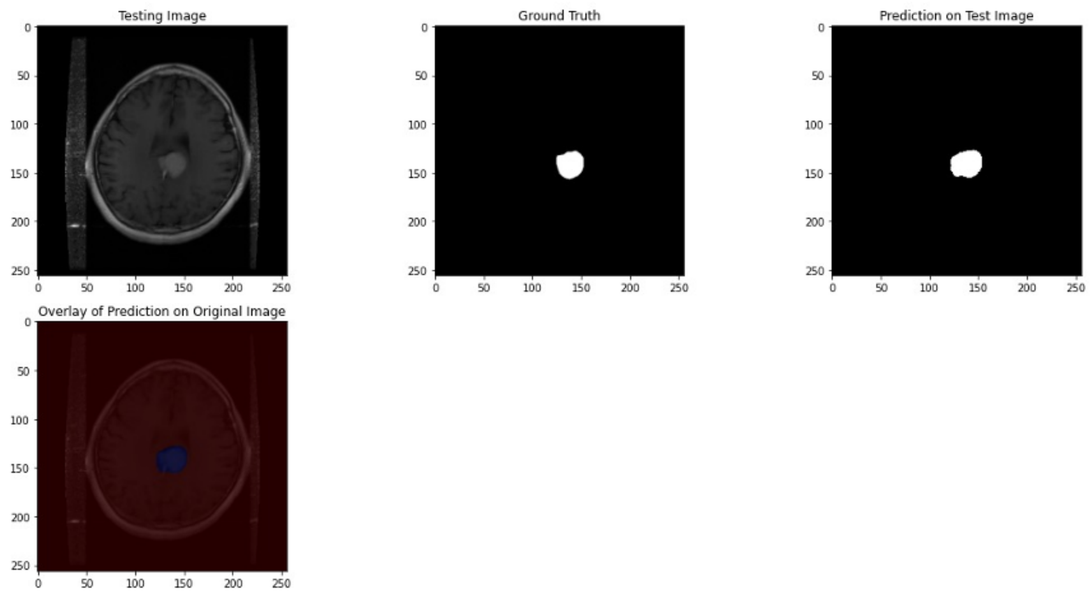


Figure 9.2: MRI Scan 2- Segmentation Result

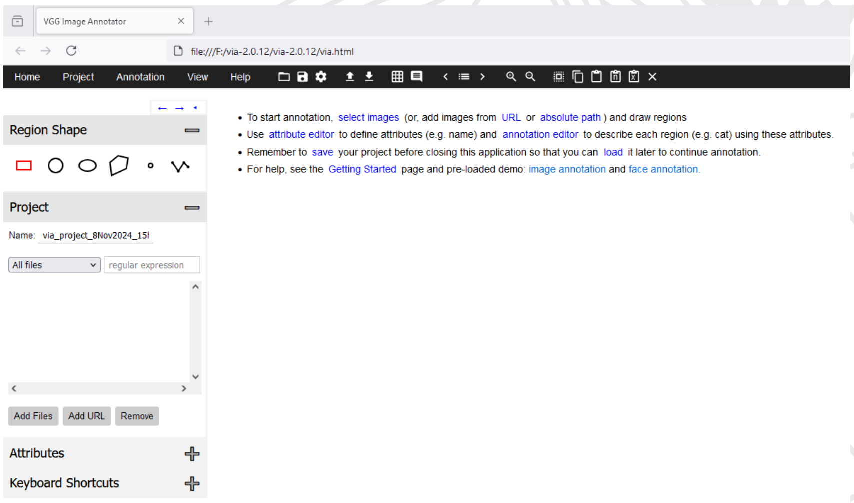
Similarly, Figure **??** shows another example of segmentation. The components are identical to those in Figure **??**:

- **Top-Left:** Original testing image.
- **Top-Right:** Ground truth mask showing the accurate target region.
- **Bottom-Right:** Predicted mask created by the model.
- **Bottom-Left:** Overlay of the predicted mask on the original scan to evaluate alignment.

The segmentation results presented in Figures **??** and **??** demonstrate the effectiveness of the model. The overlays confirm that the predicted masks align well with the ground truth. Accurate segmentation is critical for medical diagnostics, and these results indicate the robustness of the applied algorithm.

CHAPTER 10
Labeling Process Guide

1. **Display the initial interface of the application.**
   This screen is the starting point where you set up your project and prepare to add images for annotation. It shows options for managing files and accessing the attribute editor.

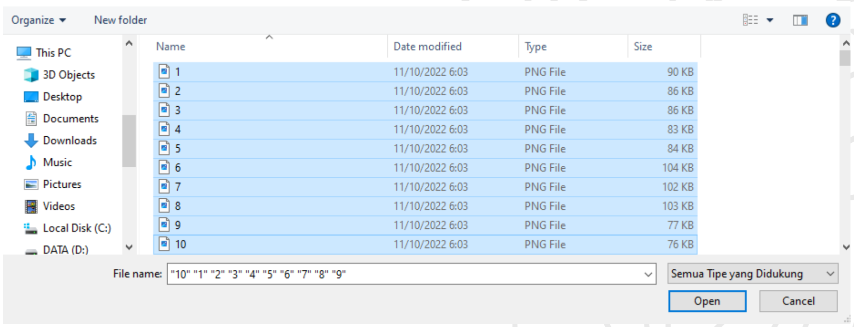


Name: via_project_8Nov2024_15l

- To start annotation, select images (or, add images from URL or absolute path) and draw regions
- Use attribute editor to define attributes (e.g. name) and annotation editor to describe each region (e.g. cat) using these attributes.
- Remember to save your project before closing this application so that you can load it later to continue annotation.
- For help, see the Getting Started page and pre-loaded demo: image annotation and face annotation.

Figure 10.1: Initial Interface

2. **Open the file you want to annotate.**
   In this step, you select the image file that you'll be working on. You can add images from various sources, such as URLs or a specified file path.



Name	Date modified	Type	Size
1	11/10/2022 6:03	PNG File	90 KB
2	11/10/2022 6:03	PNG File	86 KB
3	11/10/2022 6:03	PNG File	86 KB
4	11/10/2022 6:03	PNG File	83 KB
5	11/10/2022 6:03	PNG File	84 KB
6	11/10/2022 6:03	PNG File	104 KB
7	11/10/2022 6:03	PNG File	102 KB
8	11/10/2022 6:03	PNG File	103 KB
9	11/10/2022 6:03	PNG File	77 KB
10	11/10/2022 6:03	PNG File	76 KB

File name: "10" "1" "2" "3" "4" "5" "6" "7" "8" "9"
Semua Tipe yang Didukung

Figure 10.2: Open File

3. **Confirm that the file has been successfully opened.**
   This screen provides feedback that your selected file is ready for annotation. It ensures that the image has been loaded correctly, allowing you to proceed with defining regions.

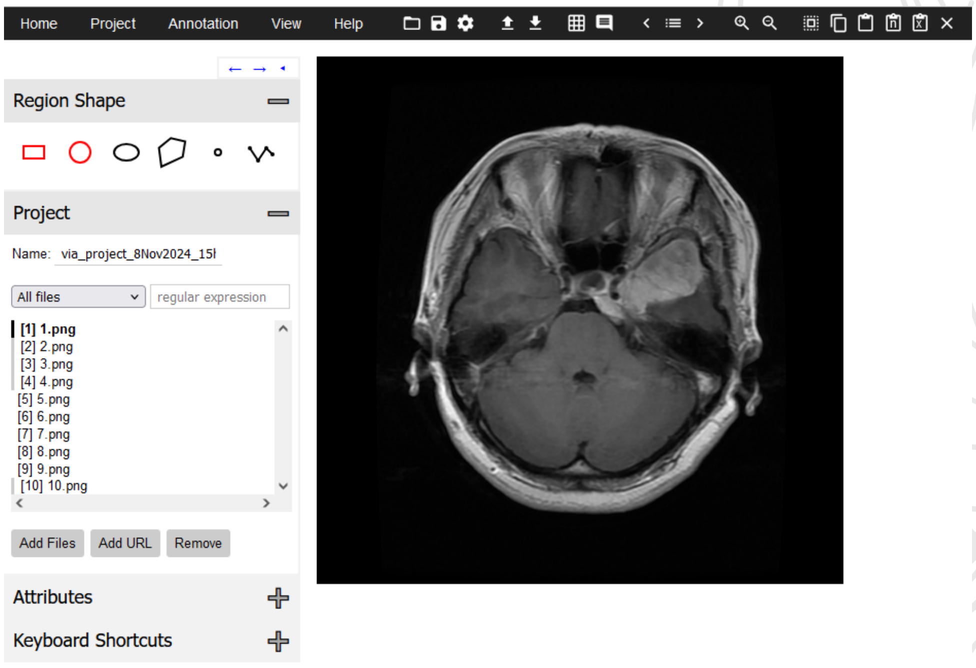


Figure 10.3: File Opened

4. **The labeling process is in progress.**
   Here, you start creating annotations by drawing shapes or regions over parts of the image. You can define different regions and assign attributes to each area based on your labeling criteria.

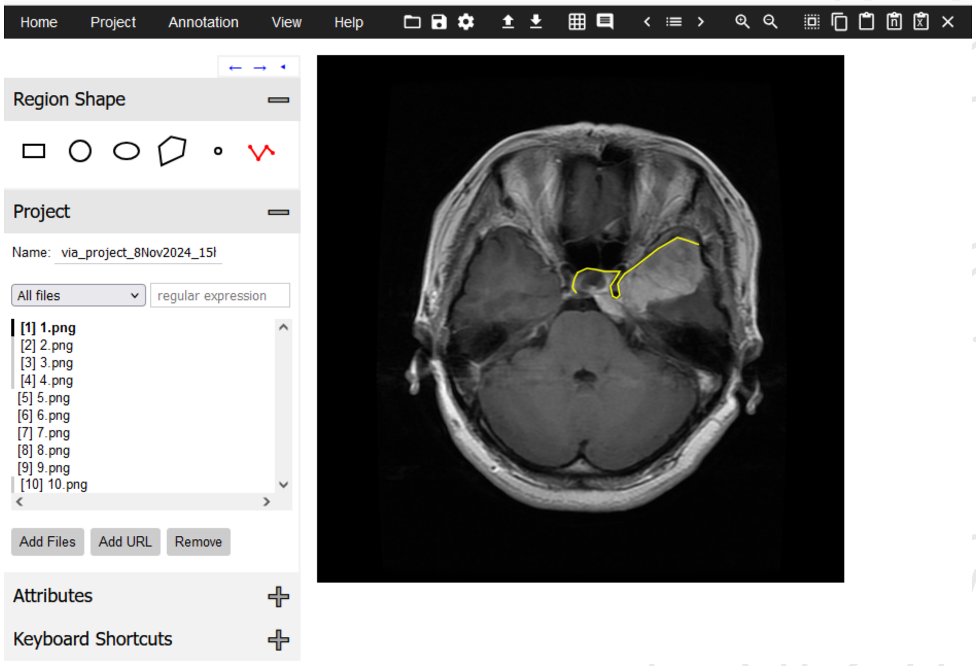


Figure 10.4: Labeling in Progress

5. **The labeling process is completed.** This screen shows that all regions have been labeled, and any necessary attributes have been applied. It's a checkpoint to review the annotations before finalizing.

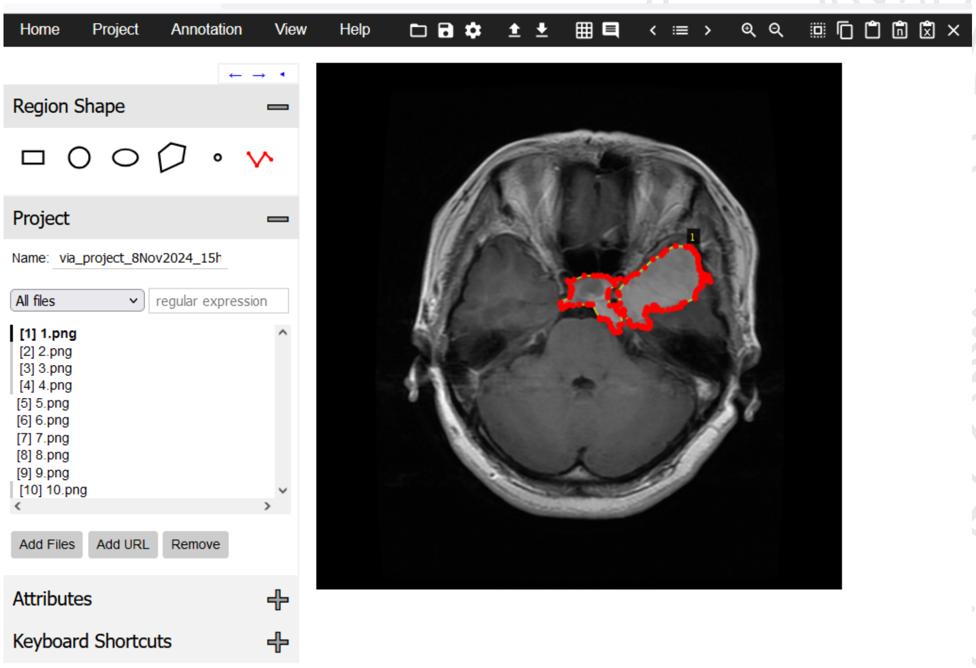


Figure 10.5: Labeling Complete

6. **Final result of the labeling process.** This displays a summary of the annotations, showing each labeled region and its attributes. You can review this to ensure that everything is accurately labeled according to your project goals.

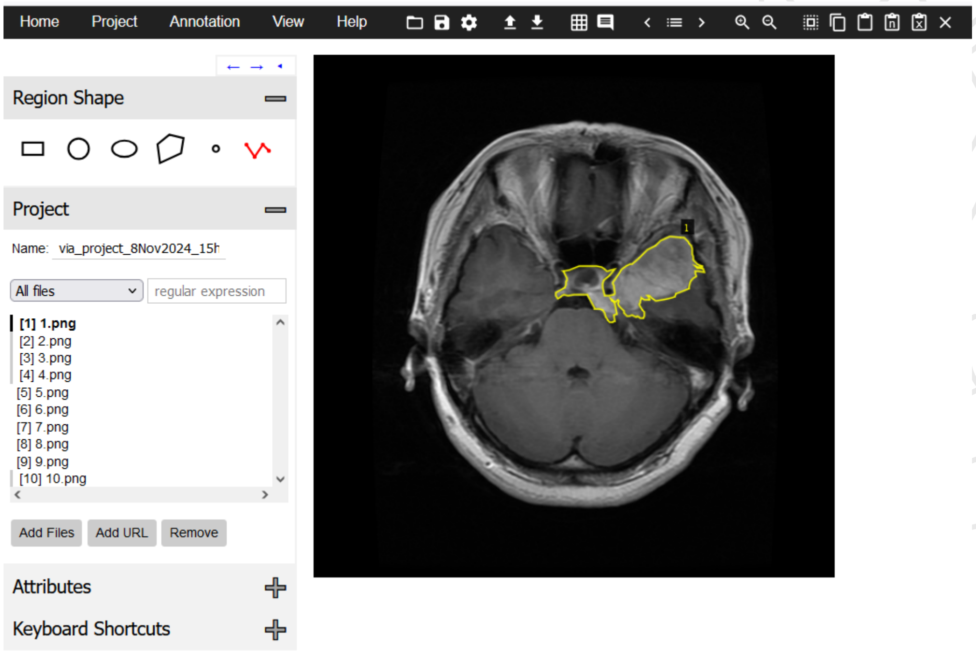


Figure 10.6: Final Result

7. **Export the labeled data.** Once labeling is complete, this step allows you to export the annotations. The export can be done in various formats, making the data ready for further analysis or use in machine learning models.

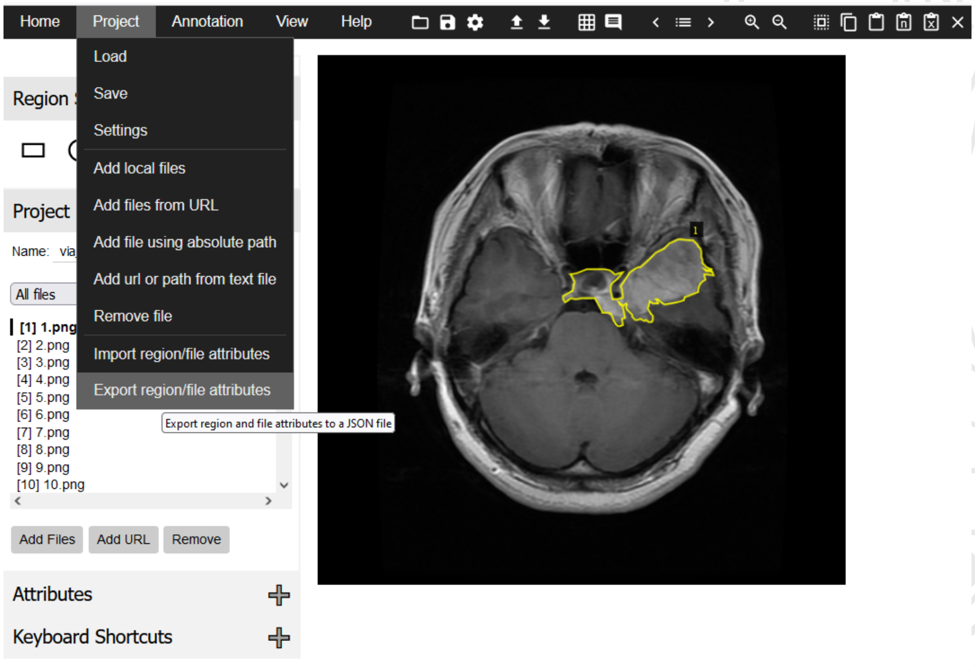


Figure 10.7: Export Data

CHAPTER 11
Costum Dataset

11.1 IMPORTING LIBRARIES

1. **Importing numpy**: Numpy is imported as `np` to handle numerical computations, particularly with arrays and matrices.
2. **Importing os**: Used for interacting with the operating system, such as handling file paths and directory operations.
3. **Importing json**: Used for parsing and generating JSON data, which is often employed for managing image annotations or configurations in machine learning projects.
4. **Importing cv2**: OpenCV is imported as `cv2` to handle advanced image and video processing tasks, including reading, writing, and transforming images.
5. **Importing pandas**: Useful for data manipulation and analysis, often employed in handling tabular data related to image annotations or model performance.
6. **Importing random**: Used for generating random numbers, often employed in data augmentation or shuffling datasets.
7. **Importing PIL.Image**: From the Python Imaging Library (PIL), `Image` is imported to handle image processing tasks like opening, manipulating, and saving images.
8. **Importing polygon from skimage.draw**: Used for generating coordinates of pixels inside a polygon, useful in image processing tasks such as annotation or masking.
9. **Importing resize from skimage.transform**: Resizes images to specific dimensions while maintaining aspect ratio, often used for preprocessing images for machine learning models.
10. **Importing matplotlib.pyplot**: A plotting library used to visualize data, often for displaying images and their transformations during data analysis or debugging.
11. **Importing tensorflow.keras model-related modules**:
    - **Model**: A base class for defining neural networks.
    - **Input**: Used to define the input layer of a model.

- **Conv2D**: A layer for applying 2D convolution to an input.
- **MaxPooling2D**: A layer for down-sampling feature maps using maximum pooling.
- **UpSampling2D**: Used for up-sampling input data, often in decoder architectures.
- **Concatenate**: Combines layers or tensors along a specific axis.
- **Conv2DTranspose**: Performs transposed convolution (deconvolution) for up-sampling.
- **Dropout**: Regularization technique to prevent overfitting by randomly deactivating neurons during training.

12. **Importing tensorflow.keras.optimizers.Adam**: A popular optimizer that combines the benefits of AdaGrad and RMSProp, used to adjust the weights of a model during training.

13. **Importing tensorflow.keras.preprocessing.image.ImageDataGenerator**: Provides functionality for image augmentation such as rotation, zoom, or flipping, which helps increase the diversity of the training dataset.

14. **Importing sklearn's train_test_split**: Splits data into training and testing sets, useful for evaluating the performance of a machine learning model.

```python
import numpy as np
import os
import json
import cv2
import pandas as pd
import random
from PIL import Image
from skimage.draw import polygon
from skimage.transform import resize
import matplotlib.pyplot as plt
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D,
   UpSampling2D, concatenate, Conv2DTranspose, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import MeanIoU
from sklearn.model_selection import train_test_split
```

11.2 DEFINING PATHS FOR DATA PROCESSING

1. **Setting image directory path**: The `image_directory` variable specifies the path to the folder containing images for segmentation. This path will be used to load images for preprocessing and training the model.

2. **Setting mask JSON file path**: The `mask_json_path` variable specifies the path to the VIA JSON file containing annotations for the images. This file is critical for creating masks from annotated regions for training a segmentation model.

3. **Setting mask save directory**: The `mask_save_directory` variable specifies the folder path where generated masks will be saved. Masks generated from annotations will be stored in this directory for later use.

4. **Printing contents of the image directory**: The `os.listdir()` function is used to list all files and folders in the `image_directory`. This helps verify the files present and troubleshoot potential path issues. The output provides an overview of the available image files for processing.

```python
image_directory = '/content/drive/MyDrive/Resources/4
    _Image_Segmentation/Dataset_Custom/Image/'  # Path to images
mask_json_path = '/content/drive/MyDrive/Resources/4
    _Image_Segmentation/Dataset_Custom/
    Brain_Segmentation_Workshop_json_Fix.json'  # Path to VIA JSON
mask_save_directory = '/content/Masks/'  # Path to save masks
print(f"Image directory contents: {os.listdir(image_directory)}")
```

11.3 ENSURING DIRECTORY STRUCTURE AND DEFINING UTILITY FUNCTI

1. **Ensuring mask save directory exists**: The `os.makedirs()` function creates the specified directory if it does not already exist. The `exist_ok=True` parameter ensures no error is raised if the directory already exists, preventing interruptions in the script.

2. **Function to create binary mask from polyline data**: The `create_mask_from_polyline` function generates a binary mask from polygon coordinates (x and y points).

   - **Parameters**:
     - `image_shape`: Shape of the image to determine mask dimensions.
     - `points_x` and `points_y`: Arrays of x and y coordinates defining the polygon.
   - **Process**:
     - A zero-initialized array of the image dimensions is created.
     - The `polygon()` function calculates the rows (`rr`) and columns (`cc`) inside the polygon.
     - The `np.clip()` ensures polygon coordinates are within image boundaries.
     - The pixels inside the polygon are set to 1, representing the mask.

3. **Loading JSON annotations**: The `json.load()` function loads annotations from the VIA JSON file containing segmentation data. This step prepares the annotations for further processing, such as creating masks or mapping them to images.

```python
# Ensure mask save directory exists
os.makedirs(mask_save_directory, exist_ok=True)

# Function to create binary mask from polyline data
def create_mask_from_polyline(image_shape, points_x, points_y):
    mask = np.zeros(image_shape[:2], dtype=np.uint8)
    rr, cc = polygon(points_y, points_x)
    rr = np.clip(rr, 0, image_shape[0] - 1)
    cc = np.clip(cc, 0, image_shape[1] - 1)
    mask[rr, cc] = 1
    return mask

# Load JSON annotations
with open(mask_json_path, 'r') as f:
    annotations = json.load(f)
```

11.4 PROCESSING IMAGES AND GENERATING MASKS

1. **Start processing images**: A message is printed to indicate the beginning of image and mask processing.
2. **Iterating through JSON annotations**: The script loops through the keys and values of the `annotations` dictionary to process each image.
3. **Debugging with filename**: The filename of the current image from the JSON is printed for troubleshooting.
4. **Constructing the image path**: The full path to the image is created using `os.path.join()` and normalized for cross-platform compatibility with `os.path.normpath()`.
5. **Debugging with image path**: The generated image path is printed to ensure the correct path is used.
6. **Checking if the image exists**: A check is performed using `os.path.exists()` to ensure the image file exists before processing.
7. **Loading the image**: The image is opened using `PIL.Image` and converted to an RGB array. Its original shape is also stored for further processing.
8. **Initializing an empty mask**: An empty mask of the same dimensions as the image is created using `np.zeros()`.
9. **Checking for regions in the annotation**: If no regions are found in the annotation, a message is printed and the loop continues to the next image.
10. **Generating the mask from regions**: For each region in the annotation, the shape attributes are extracted, and a mask is created using `create_mask_from_polyline`.
    - The `shape_attributes` contain the coordinates of the region's polygon points (`all_points_x` and `all_points_y`).
    - A region mask is generated and added to the main mask.
11. **Saving the generated mask**: The mask is saved as a PNG image in the `mask_save_directory`. If an error occurs during saving, it is caught and reported.
12. **Handling missing images**: If the image file is not found, a warning is printed.
13. **Completion message**: A final message is printed to indicate the processing is complete.

```python
print("Processing images and generating masks...")
for key, annotation in annotations.items():
    filename = annotation['filename']

    # Debug: print the filename from the JSON
    print(f"Filename from JSON: {filename}")

    # Construct the full path to the image
    image_path = os.path.normpath(os.path.join(image_directory,
       filename))

    # Debug: print the generated image path
    print(f"Generated image path: {image_path}")

    if os.path.exists(image_path):
        print(f"Processing: {filename}")

        # Load the image
        image = Image.open(image_path).convert("RGB")
        image = np.array(image)
        original_shape = image.shape

        # Initialize an empty mask
        mask = np.zeros(original_shape[:2], dtype=np.uint8)

        # Check if regions exist
        if 'regions' not in annotation or not annotation['regions
           ']:
            print(f"No regions found for {filename}")
            continue

        # Process regions to generate the mask
        for region in annotation['regions']:
            shape_attributes = region['shape_attributes']
            points_x = shape_attributes['all_points_x']
            points_y = shape_attributes['all_points_y']
            print(f"Region points: X={points_x}, Y={points_y}")

            # Create mask for the region
            region_mask = create_mask_from_polyline(
               original_shape, points_x, points_y)
            if region_mask is not None:
                 mask += region_mask

        # Save the mask
        try:
            mask_filename = os.path.splitext(os.path.basename(
               filename))[0] + ".png"
            mask_path = os.path.join(mask_save_directory,
               mask_filename)
            Image.fromarray((mask * 255).astype(np.uint8)).save(
               mask_path)
            print(f"Saved mask to {mask_path}")
        except Exception as e:
            print(f"Error saving mask for {filename}: {e}")
    else:
        print(f"Image not found: {image_path}")

print("Processing complete.")
```

11.5 DEFINING PATHS FOR IMAGE AND MASK DIRECTORIES

1. **Setting the image directory path**: The `image_directory` variable specifies the location of the folder containing the input images for segmentation.
    - This directory will be used to load images for processing and generating segmentation masks.
2. **Setting the mask directory path**: The `mask_directory` variable specifies the location where the generated segmentation masks will be saved.
    - This ensures all masks are stored in a dedicated directory, facilitating organized access for future use.

```
image_directory = '/content/drive/MyDrive/Resources/4
   _Image_Segmentation/Dataset_Custom/Image/'  # Path to images
mask_directory = '/content/Masks/'
```

11.6 INITIALIZING VARIABLES FOR DATASET PREPARATION

1. **Defining the image size**: The `SIZE` variable is set to 256, representing the dimension to which all images and masks will be resized.
    - Standardizing the size of the dataset ensures compatibility with neural networks, which often require fixed input dimensions.
2. **Initializing the image dataset**: The `image_dataset` list is initialized as an empty list to store processed images.
    - This will hold images resized to the defined `SIZE` during preprocessing.
3. **Initializing the mask dataset**: The `mask_dataset` list is initialized as an empty list to store corresponding segmentation masks.
    - This list will contain masks aligned with the images in `image_dataset`, resized to the same dimensions for proper training.

```
SIZE = 256
image_dataset = []  # List to store images
mask_dataset = []   # List to store masks
```

11.7 LOADING, RESIZING, AND NORMALIZING IMAGES AND MASKS

1. **Loading and resizing images from** `image_directory`: The script iterates through all files in the `image_directory`, sorts them to maintain order, and processes only `.png` files.
   - Files are sorted alphabetically to maintain consistent alignment between images and masks.
   - Each `.png` file is read using OpenCV (`cv2.imread`) and converted to PIL format for resizing.
   - Images are resized to the dimensions specified by `SIZE` (256x256) using `Image.NEAREST` to ensure pixel-level accuracy.
   - Resized images are converted back to NumPy arrays and appended to the `image_dataset` list.
2. **Loading and resizing masks from** `mask_directory`: Similarly, all files in the `mask_directory` are processed, with only `.png` files being considered.
   - Masks are loaded in grayscale mode (`cv2.imread` with `0` flag) since they represent binary segmentation.
   - Resizing and appending follow the same logic as for the images, ensuring masks are aligned with their corresponding images.
3. **Normalizing image data**: The `image_dataset` is converted to a NumPy array and normalized by dividing pixel values by 255.0, scaling them to the range [0, 1].
   - This step is crucial for neural networks to work efficiently with consistent input ranges.
4. **Normalizing and reshaping mask data**: The `mask_dataset` is converted to a NumPy array and rescaled to [0, 1]. An additional dimension is added using `np.expand_dims()` to match the expected input shape for neural networks.
5. **Debugging dataset size**: The script prints the number of processed images and masks for verification.
   - This step ensures that the datasets are correctly processed and ready for model training.

```python
# Load and resize images from image_directory
images = sorted(os.listdir(image_directory))  # Sort to ensure
    correct order
for i, image_name in enumerate(images):
    if image_name.split('.')[1] == 'png':  # Ensure correct file
       format
        image = cv2.imread(os.path.join(image_directory,
            image_name), 1)  # Read image
        image = Image.fromarray(image)  # Convert to PIL format
        image = image.resize((SIZE, SIZE), Image.NEAREST)  #
            Resize image
        image_dataset.append(np.array(image))  # Append to list

# Load and resize masks from mask_directory
masks = sorted(os.listdir(mask_directory))  # Sort to ensure
    correct order
for i, image_name in enumerate(masks):
    if image_name.split('.')[1] == 'png':  # Ensure correct file
        format
        mask = cv2.imread(os.path.join(mask_directory, image_name
            ), 0)  # Read mask (grayscale)
        mask = Image.fromarray(mask)  # Convert to PIL format
        mask = mask.resize((SIZE, SIZE), Image.NEAREST)  # Resize
            mask
        mask_dataset.append(np.array(mask))  # Append to list

# Normalize image data
image_dataset = np.array(image_dataset) / 255.0
# Do not normalize masks, just rescale to 0 to 1
mask_dataset = np.expand_dims((np.array(mask_dataset)), axis=3) /
    255.0  # Expand dims for mask

# Debug: Check the number of images and masks
print(f'Number of images: {len(image_dataset)}')
print(f'Number of masks: {len(mask_dataset)}')
```

11.8 SPLITTING DATASET INTO TRAINING AND TESTING SETS

1. **Splitting the dataset:** The `train_test_split` function from `sklearn.model_selection` is used to split the dataset into training and testing sets.
   - `image_dataset`: The set of input images to be split.
   - `mask_dataset`: The corresponding set of segmentation masks to be split.
   - `test_size=0.10`: Specifies that 10% of the dataset will be reserved for testing.
   - `random_state=0`: Ensures reproducibility by using a fixed seed for the random split.
   - `shuffle=False`: Prevents shuffling of the dataset, keeping the original order intact.
2. **Resulting datasets:** After splitting, the following datasets are created:
   - `X_train`: Training images.
   - `X_test`: Testing images.
   - `y_train`: Training masks.
   - `y_test`: Testing masks.
3. **Purpose:** Splitting the dataset ensures that the model is trained on one subset (`X_train`, `y_train`) and evaluated on a separate subset (`X_test`, `y_test`) to measure its performance on unseen data.

```python
# Split dataset into train and test
X_train, X_test, y_train, y_test = train_test_split(
    image_dataset, mask_dataset, test_size=0.10, random_state=0,
        shuffle=False
)
```

11.9 VISUALIZING IMAGES AND CORRESPONDING MASKS

1. **Performing a sanity check:** A random image and its corresponding mask are visualized to verify the correctness of the dataset.
    - **Purpose:** Ensures that the images and their masks are correctly aligned after preprocessing and splitting.
2. **Setting up the visualization layout:** A figure with two subplots is created to display the image and its mask side by side.
    - The figure's size is defined using `figsize=(12, 6)` to control the layout.
3. **Displaying the image:** The randomly selected image from the training dataset is reshaped to the correct dimensions (`SIZE x SIZE x 3`) and displayed.
    - **Details:** The reshaping ensures that the image has the correct shape, and the `imshow()` function renders the image with a title for clarity.
4. **Displaying the mask:** The corresponding mask is reshaped to `SIZE x SIZE` and displayed in grayscale using a colormap.
    - **Details:** The mask is displayed to verify its alignment with the image, and grayscale is used as masks are binary (0 or 1).
5. **Displaying the figure:** The figure, containing both the image and its corresponding mask, is rendered side by side for visual inspection.
    - **Outcome:** Visual confirmation that the image and its mask match correctly, ensuring the integrity of the dataset before model training.

```python
# Sanity check: Visualize some images and their corresponding
   masks
image_number = random.randint(0, len(X_train))
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.imshow(np.reshape(X_train[image_number], (SIZE, SIZE, 3)))  #
    Display image
plt.title('Image')
plt.subplot(122)
plt.imshow(np.reshape(y_train[image_number], (SIZE, SIZE)), cmap=
   'gray')  # Display mask
plt.title('Mask')
plt.show()
```

11.10 DEFINING IMAGE DIMENSIONS

1. **Extracting image height:** The `IMG_HEIGHT` variable is assigned the height of the images in the dataset.
   - **Purpose:** Ensures the model knows the height of the input images, extracted from the dataset's shape.
2. **Extracting image width:** The `IMG_WIDTH` variable is assigned the width of the images in the dataset.
   - **Purpose:** Provides the model with the input width, derived from the dataset dimensions.
3. **Extracting the number of image channels:** The `IMG_CHANNELS` variable is assigned the number of channels in the images (e.g., 3 for RGB).
   - **Purpose:** Informs the model about the input's channel depth, typically 3 for RGB or 1 for grayscale images.
4. **Explanation:** The `image_dataset.shape` provides the dimensions of the dataset in the format `(num_images, height, width, channels)`. This ensures the model dynamically adapts to the dataset's actual dimensions without hardcoding values.
5. **Example:** If `image_dataset.shape = (100, 256, 256, 3)`:
   - `IMG_HEIGHT = 256`
   - `IMG_WIDTH = 256`
   - `IMG_CHANNELS = 3`

```
IMG_HEIGHT  = image_dataset.shape[1]
IMG_WIDTH   = image_dataset.shape[2]
IMG_CHANNELS = image_dataset.shape[3]
```

11.11 U-NET MODEL DEFINITION

1. **Defining the U-Net model:** The `simple_unet_model` function defines a U-Net architecture for image segmentation tasks.
   - **Inputs:** `IMG_HEIGHT`, `IMG_WIDTH`, and `IMG_CHANNELS` specify the dimensions of the input image.
2. **Contraction Path (Encoder):** The encoder progressively reduces the spatial dimensions while increasing the number of feature channels.
   - Two convolutional layers with ReLU activation are applied.
   - A `Dropout` layer helps prevent overfitting.

   - `MaxPooling2D` reduces spatial dimensions by half.

   The layers involved:

   - `Conv2D`: Extracts spatial features with a kernel size of `(3, 3)`.
   - `MaxPooling2D`: Reduces image dimensions while retaining important features.

   This pattern is repeated with increasing filters (16, 32, 64, 128, 256).

3. **Bottleneck:** The bottleneck serves as the transition between encoder and decoder.
   - The number of filters is highest here (256), capturing complex spatial patterns.
   - Dropout is increased to 30% to handle high feature density.
4. **Expansion Path (Decoder):** The decoder progressively upsamples the feature maps and concatenates them with the corresponding encoder outputs.
   - `Conv2DTranspose`: Performs transposed convolution for upsampling.
   - `concatenate`: Merges decoder features with corresponding encoder features to retain spatial information.

   This pattern is repeated with decreasing filters (128, 64, 32, 16) until the original resolution is restored.
5. **Output Layer:** The final layer outputs a single-channel binary mask using the sigmoid activation function.
   - `Conv2D(1, (1, 1))`: Reduces the channels to 1 for binary segmentation.
   - `sigmoid`: Maps the output to a range of [0, 1], representing probabilities.
6. **Compiling the Model:** The model is compiled with the Adam optimizer, binary crossentropy loss, and accuracy as a metric.
   - `Adam`: An adaptive optimizer for efficient weight updates.
   - `binary_crossentropy`: Suitable for binary segmentation tasks.
7. **Model Summary:** The model structure is printed for verification.
8. **Return the Model:** The compiled model is returned.
9. **Purpose:** The U-Net model efficiently captures spatial and contextual features, making it ideal for segmentation tasks like medical imaging or object detection.

```python
# U-Net Model Definition
def simple_unet_model(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS):
    inputs = Input((IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS))

    # Contraction path
    c1 = Conv2D(16, (3, 3), activation='relu', kernel_initializer
        ='he_uniform', padding='same')(inputs)
    c1 = Dropout(0.1)(c1)
    c1 = Conv2D(16, (3, 3), activation='relu', kernel_initializer
        ='he_uniform', padding='same')(c1)
    p1 = MaxPooling2D((2, 2))(c1)

    c2 = Conv2D(32, (3, 3), activation='relu', kernel_initializer
       ='he_uniform', padding='same')(p1)
    c2 = Dropout(0.1)(c2)
    c2 = Conv2D(32, (3, 3), activation='relu', kernel_initializer
       ='he_uniform', padding='same')(c2)
    p2 = MaxPooling2D((2, 2))(c2)

    c3 = Conv2D(64, (3, 3), activation='relu', kernel_initializer
       ='he_uniform', padding='same')(p2)
    c3 = Dropout(0.2)(c3)
    c3 = Conv2D(64, (3, 3), activation='relu', kernel_initializer
       ='he_uniform', padding='same')(c3)
    p3 = MaxPooling2D((2, 2))(c3)

    c4 = Conv2D(128, (3, 3), activation='relu',
       kernel_initializer='he_uniform', padding='same')(p3)
    c4 = Dropout(0.2)(c4)
    c4 = Conv2D(128, (3, 3), activation='relu',
       kernel_initializer='he_uniform', padding='same')(c4)
    p4 = MaxPooling2D(pool_size=(2, 2))(c4)

    c5 = Conv2D(256, (3, 3), activation='relu',
       kernel_initializer='he_uniform', padding='same')(p4)
    c5 = Dropout(0.3)(c5)
    c5 = Conv2D(256, (3, 3), activation='relu',
       kernel_initializer='he_uniform', padding='same')(c5)

    # Expansive path
    u6 = Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='
       same')(c5)
    u6 = concatenate([u6, c4])
    c6 = Conv2D(128, (3, 3), activation='relu',
       kernel_initializer='he_uniform', padding='same')(u6)
    c6 = Dropout(0.2)(c6)
    c6 = Conv2D(128, (3, 3), activation='relu',
       kernel_initializer='he_uniform', padding='same')(c6)

    u7 = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='
       same')(c6)
    u7 = concatenate([u7, c3])
    c7 = Conv2D(64, (3, 3), activation='relu', kernel_initializer
       ='he_uniform', padding='same')(u7)
    c7 = Dropout(0.2)(c7)
    c7 = Conv2D(64, (3, 3), activation='relu', kernel_initializer
       ='he_uniform', padding='same')(c7)

    u8 = Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='
       same')(c7)
    u8 = concatenate([u8, c2])
    c8 = Conv2D(32, (3, 3), activation='relu', kernel_initializer
       ='he_uniform', padding='same')(u8)
    c8 = Dropout(0.1)(c8)
    c8 = Conv2D(32, (3, 3), activation='relu', kernel_initializer
        ='he_uniform', padding='same')(c8)

    u9 = Conv2DTranspose(16, (2, 2), strides=(2, 2), padding='
        same')(c8)
    u9 = concatenate([u9, c1])
    c9 = Conv2D(16, (3, 3), activation='relu', kernel_initializer
        ='he_uniform', padding='same')(u9)
    c9 = Dropout(0.1)(c9)
    c9 = Conv2D(16, (3, 3), activation='relu', kernel_initializer
        ='he_uniform', padding='same')(c9)

    outputs = Conv2D(1, (1, 1), activation='sigmoid')(c9)

    model = Model(inputs=[inputs], outputs=[outputs])
    model.compile(optimizer=Adam(learning_rate=1e-3), loss='
        binary_crossentropy', metrics=['accuracy'])
    model.summary()

    return model
```

11.12 INITIALIZING AND TRAINING THE U-NET MODEL

1. **Initializing the U-Net model:** The `simple_unet_model` function is called with the image dimensions to create an instance of the U-Net model.
   - **Parameters:**
     - `IMG_HEIGHT`: Height of the input images.
     - `IMG_WIDTH`: Width of the input images.
     - `IMG_CHANNELS`: Number of channels in the input images (e.g., 3 for RGB).
   - **Outcome:** The model architecture is built, compiled, and ready for training.
2. **Training the U-Net model:** The `fit` function trains the model using the input images and masks.
   - **Parameters:**
     - `image_dataset`: Input images used for training.
     - `mask_dataset`: Corresponding ground truth masks for the input images.
     - `batch_size=32`: Specifies that 32 samples are processed at a time during training.
     - `epochs=100`: The model will iterate 100 times over the entire dataset.
     - `verbose=1`: Provides detailed progress output during training.
     - `validation_split=0.2`: Reserves 20% of the training data for validation to monitor the model's performance on unseen data.
3. **Resulting Output:**

- **Training History:** The `history` object contains details about training, such as loss and accuracy for both training and validation sets.

4. **Purpose:** Training optimizes the model's weights to minimize the loss function and improve accuracy, preparing it for segmentation tasks.

```
# Initialize model
model = simple_unet_model(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS)
######################################################
# Train the model
history = model.fit(image_dataset, mask_dataset,
                    batch_size=32,
                    epochs=100,
                    verbose=1,
                    validation_split=0.2)  # Use 20% of data for
                       validation
```

11.13 SAVING TRAINING HISTORY TO A CSV FILE

1. **Creating a DataFrame for the training history:** The `history.history` attribute is converted into a pandas DataFrame.
   - **Details:**
     - `history.history` contains the training and validation metrics (e.g., loss, accuracy) for each epoch.
     - Converting it into a DataFrame organizes the data for easier analysis and export.
2. **Saving the training history to a CSV file:** The DataFrame is saved as a CSV file using the `to_csv()` method.
   - **Details:**
     - The file is opened in write mode (`mode='w'`), ensuring any existing file with the same name is overwritten.
     - The training history is exported to `unet_history_df.csv`, enabling further analysis or documentation.
3. **Purpose:** Storing the training history provides a persistent record of the model's performance across epochs.
   - It allows for:
     - Plotting metrics such as loss and accuracy trends.
     - Comparing the performance of different model configurations.
     - Debugging or revisiting results after training is complete.

```
unet_history_df = pd.DataFrame(history.history)
with open('unet_history_df.csv', mode='w') as f:
    unet_history_df.to_csv(f)
```

11.14 PLOTTING TRAINING AND VALIDATION LOSS AND ACCURACY

1. **Plotting Training and Validation Loss:** This section visualizes the training and validation loss for each epoch.

   - **Steps:**
     - Extract the `loss` (training) and `val_loss` (validation) from `history.history`.
     - Define the `epochs` range using the number of loss values.
     - Plot:
       * `loss` in yellow (`'y'`) for training loss.
       * `val_loss` in red (`'r'`) for validation loss.
     - Add labels and a legend to identify the curves.
     - Use `plt.show()` to display the plot.
   - **Purpose:** To monitor overfitting or underfitting by comparing training and validation loss trends.

2. **Plotting Training and Validation Accuracy:** This section visualizes the training and validation accuracy for each epoch.

   - **Steps:**
     - Extract the `accuracy` (training) and `val_accuracy` (validation) from `history.history`.
     - Use the same `epochs` range as defined earlier.
     - Plot:
       * `accuracy` in yellow (`'y'`) for training accuracy.
       * `val_accuracy` in red (`'r'`) for validation accuracy.
     - Add labels, a legend, and display the plot.
   - **Purpose:** To track how well the model is learning by observing the accuracy improvement during training and validation.

3. **Outcome:** These plots provide insights into the model's performance trends, helping to:

   - Detect overfitting (e.g., increasing validation loss while training loss decreases).
   - Identify epochs where accuracy saturates, signaling the need for early stopping or further tuning.

```
#plot the training and validation accuracy and loss at each epoch
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'y', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
```

```python
plt.legend()
plt.show()
############################################################

############################################################
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
plt.plot(epochs, acc, 'y', label='Training Accuracy')
plt.plot(epochs, val_acc, 'r', label='Validation Accuracy')
plt.title('Training and validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

11.15 VISUALIZING TRAINING HISTORY: COMBINED ACCURACY AND LOSS

1. **Setting up the visualization:** A single plot is used to visualize both training and validation accuracy and loss trends.
   - **Details:**
     - `figsize=(12, 6)` specifies the size of the plot, making it wide enough to clearly display the data.
2. **Plotting training and validation accuracy:** The `accuracy` and `val_accuracy` metrics from the training history are plotted.
   - **Details:**
     - The `accuracy` curve shows how well the model performs on the training data.
     - The `val_accuracy` curve indicates performance on the validation data.
3. **Plotting training and validation loss:** Similarly, `loss` and `val_loss` metrics are added to the plot.
   - **Details:**
     - The `loss` curve tracks the model's error on the training data.
     - The `val_loss` curve shows error on the validation data.
4. **Adding titles and labels:** Titles and axis labels provide context for the plot.
   - **Details:**
     - `Epochs` is used as the x-axis to represent training iterations.
     - `Accuracy / Loss` is used as the y-axis to unify the two types of metrics.
5. **Adding a legend:** The `legend` makes it easy to distinguish between the curves.
   - **Details:**

- The legend helps in identifying which curve corresponds to training or validation performance.

6. **Displaying the plot:** The `plt.show()` function renders the combined plot.

   - **Details:**
     - `plt.show()` displays the plot on the screen.

7. **Purpose:** This visualization helps in:

   - Comparing training and validation performance across epochs.
   - Detecting overfitting or underfitting (e.g., divergence between training and validation metrics).
   - Monitoring overall training progress in a compact and unified view.

```python
# Visualize training history
plt.figure(figsize=(12, 6))
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation
    Accuracy')
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Accuracy and Loss')
plt.xlabel('Epochs')
plt.ylabel('Accuracy / Loss')
plt.legend()
plt.show()
```

11.16 VISUALIZING PREDICTIONS ON TEST DATA

1. **Selecting a test image and its ground truth:** A specific test image and its corresponding ground truth mask are selected from the test dataset.

   - `test_img = X_test[7]` is the 7th image from the test set.
   - `ground_truth = y_test[7]` is the corresponding ground truth mask for the selected image.

2. **Preparing the test image for prediction:** The test image is expanded to match the input shape expected by the model.

   - `test_img_input = np.expand_dims(test_img, 0)` adds an additional batch dimension.
   - The model expects inputs in the shape `(batch_size, height, width, channels)`.

3. **Generating the prediction:** The model predicts the segmentation mask for the test image.

   - `prediction = (model.predict(test_img_input)[0, :, :, 0] > 0.5).astype(np.uint8)`.

- The model's output is a probability map, which is thresholded at 0.5 to convert to binary values (0 or 1).

4. **Setting up the visualization:** A figure with three subplots is created to display the test image, ground truth, and prediction.
   - `plt.figure(figsize=(16, 8))` creates a wide figure to fit all subplots.
5. **Displaying the test image:** The test image is displayed in the first subplot.
   - `plt.subplot(231)` selects the first subplot.
   - `plt.imshow(test_img, cmap='gray')` renders the test image in grayscale.
6. **Displaying the ground truth mask:** The ground truth segmentation mask is displayed in the second subplot.
   - `plt.subplot(232)` selects the second subplot.
   - `plt.imshow(ground_truth[:, :, 0], cmap='gray')` renders the ground truth mask.
7. **Displaying the prediction:** The model's prediction is displayed in the third subplot.
   - `plt.subplot(233)` selects the third subplot.
   - `plt.imshow(prediction, cmap='gray')` renders the predicted mask.
8. **Rendering the visualization:** The `plt.show()` function displays the figure.
   - `plt.show()` renders the final output with all subplots visible.
9. **Purpose:** This visualization allows comparison between:
   - The original test image.
   - The ground truth mask.
   - The predicted segmentation mask.

   It provides qualitative insights into the model's performance on unseen data.

```
test_img = X_test[7]
ground_truth=y_test[7]
test_img_input=np.expand_dims(test_img, 0)
prediction = (model.predict(test_img_input)[0,:,:,0] > 0.5).
    astype(np.uint8)

plt.figure(figsize=(16, 8))
plt.subplot(231)
plt.title('Testing Image')
plt.imshow(test_img, cmap='gray')
plt.subplot(232)
plt.title('Testing Label')
plt.imshow(ground_truth[:,:,0], cmap='gray')
plt.subplot(233)
plt.title('Prediction on test image')
plt.imshow(prediction, cmap='gray')

plt.show()
```

11.17 CALCULATING THE INTERSECTION OVER UNION (IOU) FOR A SING

1. **Defining the number of classes:** The number of classes in the segmentation task is specified.
   - Sets the number of classes for binary segmentation (background and foreground).
2. **Initializing the MeanIoU metric:** An instance of the MeanIoU metric is created from TensorFlow Keras.
   - The MeanIoU metric computes the Intersection over Union (IoU) for each class and averages the values across all classes.
3. **Updating the metric state with ground truth and prediction:** The metric state is updated to calculate the IoU for the given ground truth and predicted mask.
   - The ground truth mask is used for comparison with the predicted mask.
4. **Printing the Mean IoU:** The IoU value is computed and displayed.
   - The result returns the calculated IoU value.
5. **Purpose:** The Intersection over Union (IoU) is a critical metric for segmentation tasks, measuring the overlap between the predicted and ground truth masks. It provides a numerical evaluation of the model's performance for a specific test image.

```
#IoU for a single image
#######################################################
n_classes = 2
IOU_keras = MeanIoU(num_classes=n_classes)
IOU_keras.update_state(ground_truth[:,:,0], prediction)
print("Mean IoU =", IOU_keras.result().numpy())
```

11.18 CALCULATING IOU FOR ALL TEST IMAGES AND COMPUTING THE A

1. **Initializing a list to store IoU values**: An empty list is created to store the IoU for each test image.
2. **Iterating through the test set**: A loop is used to process each image in the test set.
   - The total number of test images is determined, and for each image, the corresponding test image and ground truth mask are selected.
3. **Preparing the image for prediction**: Each test image is expanded to match the input shape expected by the model.
4. **Generating predictions**: The model predicts the segmentation mask for the current test image.

- The predictions are thresholded and converted to binary masks (0 or 1).

5. **Calculating IoU for the current image**: The MeanIoU metric is used to compute IoU for the current image.
   - The IoU metric is updated with the ground truth and predicted masks, and the IoU value is retrieved as a scalar.
6. **Storing the IoU value**: The computed IoU value is appended to the list of IoU values.
7. **Creating a DataFrame of IoU values**: The IoU values are converted into a pandas DataFrame for easier analysis.
8. **Filtering out invalid IoU values**: Any IoU value of 1.0 (which could be an outlier) is removed from the list.
9. **Calculating the mean IoU**: The mean IoU is computed from the filtered values, providing an overall evaluation of the model's segmentation performance.
   - A higher mean IoU indicates better segmentation quality.
10. **Purpose**: This process evaluates the segmentation performance across the entire test set by computing and averaging the IoU values for all test images. The mean IoU offers a single metric for assessing model performance.

```python
#Calculate IoU and average
##################################################
IoU_values = []
for img in range(0, X_test.shape[0]):
    temp_img = X_test[img]
    ground_truth=y_test[img]
    temp_img_input=np.expand_dims(temp_img, 0)
    prediction = (model.predict(temp_img_input)[0,:,:,0] > 0.5).
       astype(np.uint8)

    IoU = MeanIoU(num_classes=n_classes)
    IoU.update_state(ground_truth[:,:,0], prediction)
    IoU = IoU.result().numpy()
    IoU_values.append(IoU)

    #print(IoU)

df = pd.DataFrame(IoU_values, columns=["IoU"])
df = df[df.IoU != 1.0]
mean_IoU = df.mean().values
print("Mean IoU is: ", mean_IoU)
```

11.19 EVALUATING SEGMENTATION METRICS FOR TEST DATASET

1. **Importing required metrics:** Metrics for segmentation evaluation are imported from `sklearn.metrics`.

2. **Initializing lists to store metrics:** Lists are created to store IoU, precision, recall, F1-score, and accuracy for each test image.
3. **Iterating through the test dataset:** A loop is used to calculate metrics for each test image and its corresponding ground truth.
   - The current test image and its corresponding ground truth mask are selected for evaluation.
4. **Preparing the image for prediction:** The test image is expanded to match the input shape expected by the model.
5. **Generating predictions:** The model predicts the segmentation mask for the test image.
6. **Calculating IoU:** The MeanIoU metric computes the IoU for the predicted mask.
7. **Flattening ground truth and prediction:** Ground truth and prediction masks are flattened to a 1D array for calculating other metrics.
8. **Calculating precision, recall, F1-score, and accuracy:** Various metrics are calculated by comparing the flattened ground truth and predicted values:
   - `precision` measures the proportion of true positive predictions.
   - `recall` measures the ability of the model to identify all relevant positive instances.
   - `f1` is the harmonic mean of precision and recall, balancing both metrics.
   - `accuracy` measures the overall proportion of correct predictions.

```python
from sklearn.metrics import precision_score, recall_score,
    f1_score, accuracy_score

# Initialize lists to store metrics for each image
IoU_values = []
precision_values = []
recall_values = []
f1_values = []
accuracy_values = []

# Loop through the test dataset
for img in range(0, X_test.shape[0]):
    temp_img = X_test[img]
    ground_truth = y_test[img]
    temp_img_input = np.expand_dims(temp_img, 0)

    # Generate prediction
    prediction = (model.predict(temp_img_input)[0, :, :, 0] >
        0.5).astype(np.uint8)

    # Calculate IoU
    IoU = MeanIoU(num_classes=n_classes)
    IoU.update_state(ground_truth[:, :, 0], prediction)
    IoU_value = IoU.result().numpy()
    IoU_values.append(IoU_value)

    # Flatten ground truth and prediction for metric calculations
    gt_flat = ground_truth[:, :, 0].flatten()
    pred_flat = prediction.flatten()

    # Calculate metrics
    precision = precision_score(gt_flat, pred_flat, zero_division
       =0)
    recall = recall_score(gt_flat, pred_flat, zero_division=0)
    f1 = f1_score(gt_flat, pred_flat, zero_division=0)
    accuracy = accuracy_score(gt_flat, pred_flat)

    # Append metrics to their respective lists
    precision_values.append(precision)
    recall_values.append(recall)
    f1_values.append(f1)
    accuracy_values.append(accuracy)

# Create a DataFrame to store all metrics
metrics_df = pd.DataFrame({
    "IoU": IoU_values,
    "Precision": precision_values,
    "Recall": recall_values,
    "F1-Score": f1_values,
    "Accuracy": accuracy_values
})

# Calculate mean metrics
mean_metrics = metrics_df.mean()
print("Mean Metrics:")
print(mean_metrics)

# Save metrics to a CSV file
metrics_df.to_csv('segmentation_metrics_summary.csv', index=False
   )
```

Bibliography

[1] A. Abdulateef and A. Salman, “A Comprehensive Review of Image Segmentation Techniques,” *Iraqi Journal for Electrical and Electronic Engineering*, vol. 17, no. 2, pp. 18–30, 2021. doi:10.37917/ijeee.17.2.18.

[2] M. Saleem and S. Vinitha, “A Study on Image Segmentation Techniques,” *Asian Journal of Computer Science and Technology*, vol. 8, no. 2, pp. 2020, 2019. doi:10.51983/ajcst-2019.8.s2.2020.

[3] S. Patra et al., “Segmentation techniques used in image recognition and SAR Image Processing,” *IOSR Journal of Computer Engineering*, vol. 16, no. 2, pp. 37–43, 2014. doi:10.9790/0661-16213743.

[4] A. Ali et al., “Multi-resolution MRI Brain Image Segmentation Based on Morphological Pyramid and Fuzzy C-mean Clustering,” *Arabian Journal for Science and Engineering*, vol. 40, no. 9, pp. 2551–2561, 2015. doi:10.1007/s13369-015-1791-x.

[5] A. Kumar et al., “A Hybrid Approach for Image Segmentation Using Fuzzy Clustering and Level Set Method,” *International Journal of Image Graphics and Signal Processing*, vol. 4, no. 2, pp. 1–8, 2012. doi:10.5815/ijigsp.2012.06.01.

[6] M. Benazzouz et al., “Modified U-Net for cytological medical image segmentation,” *International Journal of Imaging Systems and Technology*, vol. 32, no. 1, pp. 1–12, 2022. doi:10.1002/ima.22732.

[7] Y. Wang et al., “Medical image segmentation using deep learning: A survey,” *IET Image Processing*, vol. 16, no. 5, pp. 1031–1045, 2022. doi:10.1049/ipr2.12419.

[8] Anonymous, “The Application of Support Vector Machines in Medical Image Segmentation,” *Machine Learning Theory and Practice*, vol. 2022, pp. 1–12, 2022. doi:10.38007/ml.2022.030204.

[9] X. Liu, “Image Segmentation Techniques for Intelligent Monitoring of Putonghua Examinations,” *Advances in Mathematical Physics*, vol. 2022, pp. 1–12, 2022. doi:10.1155/2022/4302666.

[10] A. Haldorai and S. Anandakumar, “Image Segmentation and the Projections of Graphic Centred Approaches in Medical Image Processing,” *International Journal of Advanced Information and Communication Technology*, vol. 2021, pp. 1–12, 2021. doi:10.46532/ijaict-202108028.

[11] A. Khan, “Image Segmentation Techniques: A Survey,” *Journal of Image and Graphics*, vol. 1, no. 4, pp. 166–170, 2014. doi:10.12720/joig.1.4.166-170.

[12] R. Gehlot and A. Kumar, “The Image Segmentation Techniques,” *International Journal of Image Graphics and Signal Processing*, vol. 2017, pp. 1–12, 2017. doi:10.5815/ijigsp.2017.02.02.

[13] M. Everingham et al., “The Pascal Visual Object Classes Challenge,” *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, 2010. doi:10.1007/s11263-009-0275-4.

[14] L. R. Dice, “Measures of the Amount of Ecologic Association Between Species,” *Ecology*, vol. 26, no. 3, pp. 297–302, 1945. doi:10.2307/1932409.

[15] L. Zhang et al., “A Review of Image Segmentation Techniques for Medical Applications,” *Journal of Healthcare Engineering*, vol. 2018, pp. 1–12, 2018. doi:10.1155/2018/4568452.

[16] Y. Yang et al., “F1 Score: A New Metric for Evaluating Image Segmentation Algorithms,” *arXiv preprint arXiv:1803.08656*, 2018. doi:10.48550/arxiv.1803.08656.

[17] D. M. Powers, “Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation,” *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37–63, 2011. doi:10.1007/s11263-009-0275-4.

[18] Chen, L. C., Papandreou, G., Schroff, F., & Adam, H. (2018). DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4), 834-848.

[19] Bousabarah, K., et al. (2020). Deep convolutional neural networks for automated segmentation of brain metastases trained on clinical data. *Radiation Oncology*, 15(1), 1-10.

[20] Ronneberger, O., Fischer, P., & Becker, A. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)* (pp. 234-241).

[21] Hesamian, M. H., et al. (2019). Deep Learning Techniques for Medical Image Segmentation: Achievements and Challenges. *Journal of Digital Imaging*, 32(4), 582-596.

[22] Minaee, S., et al. (2021). Image Segmentation Using Deep Learning: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(8), 2465-2481.

[23] Zhou, Z., & Yang, Y. (2019). Normalization in Training U-Net for 2-D Biomedical Semantic Segmentation. *IEEE Robotics and Automation Letters*, 4(4), 3484-3491.

[24] Milletarì, F., Navab, N., & Ahmadi, S. A. (2016). V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation. In *2016 Fourth International Conference on 3D Vision (3DV)* (pp. 565-571).

[25] Zhang, Y., et al. (2020). DENSE-INception U-net for medical image segmentation. *Computer Methods and Programs in Biomedicine*, 196, 105395.

[26] Zhao, Y., et al. (2023). RGSB-UNet: Hybrid Deep Learning Framework for Tumour Segmentation in Digital Pathology Images. *Bioengineering*, 10(8), 957.

[27] Gao, Y., et al. (2022). Medical Image Segmentation Algorithm for Three-Dimensional Multimodal Using Deep Reinforcement Learning and Big Data Analytics. *Frontiers in Public Health*, 10, 879639.

[28] Napte, B., & Mahajan, A. (2022). Deep Learning based Liver Segmentation: A Review. *Revue D Intelligence Artificielle*, 36(6), 619-628.

[29] Ferdinandus, S. et al. (2022). Covid-19 Lung Segmentation using U-Net CNN based on Computed Tomography Image. In *2022 2nd International Conference on Computer Vision and Image Processing (CVIP)* (pp. 1-5).

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-Based Learning Applied to Document Recognition*, Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, 1998.

[31] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016.

[32] X. Glorot and Y. Bengio, *Understanding the Difficulty of Training Deep Feedforward Neural Networks*, Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, vol. 9, pp. 249-256, 2010.

[33] K. He, X. Zhang, S. Ren, and J. Sun, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, Proceedings of the IEEE International Conference on Computer Vision, pp. 1026-1034, 2015.

[34] V. Nair and G. E. Hinton, *Rectified Linear Units Improve Restricted Boltzmann Machines*, Proceedings of the 27th International Conference on Machine Learning, pp. 807-814, 2010.

[35] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.

[36] A. L. Maas, A. Y. Hannun, and A. Y. Ng, *Rectifier Nonlinearities Improve Neural Network Acoustic Models*, Proceedings of the 30th International Conference on Machine Learning, pp. 1-9, 2013.

[37] B. Xu, N. Wang, T. Chen, and M. Li, *Empirical Evaluation of Rectified Activations in Convolutional Network*, Proceedings of the 30th International Conference on Machine Learning, vol. 37, pp. 1509-1517, 2015.