RAPPORT TP

angles des dents

Réalisé par :
Hamza Kharbouch

Encadré par :
Pr. Hrimech

Table des matières

1 Introduction **2**

2 Data Augmentation **3**
- 2.1 Horizontal Image Flipping 3
- 2.2 Resizing and Cropping Image 3
- 2.3 Horizontal Tensor Flipping 3
  - 2.3.1 Resizing and Cropping Tensor 4
- 2.4 Coordinate Transformation 4
  - 2.4.1 Coordinate Transformation in JSON 4
- 2.5 Coordinate Flipping in JSON 5
- 2.6 Coordinate Flipping 5
- 2.7 Data Augmentation Loop 6

3 Outlines **8**
- 3.1 outline script 8
- 3.2 Saving Outlines 9

4 Model training **11**
- 4.1 Model Definition 11
- 4.2 Image Preprocessing Function 12
- 4.3 Training Loop 12

5 Results **15**

Chapitre 1

Introduction

Teeth, integral components of the human anatomy, play a crucial role in various aspects of life, from aiding in digestion to contributing to one's overall appearance. In this project, we delve into the realm of dental image analysis, employing techniques to enhance and extract valuable insights from images of teeth. The objective is to develop a model that can accurately predict the coordinates of specific points on teeth, aiding in various dental applications such as treatment planning and assessment.
To achieve this goal, our project follows a multi-step approach. We begin by collecting a diverse set of teeth images, which serve as the foundation for our model. The augmentation of these images provides a robust dataset for training, allowing the model to generalize well to different scenarios. Additionally, we utilize JSON files containing annotated coordinates as ground truth data, guiding the model to learn the spatial relationships between key points on teeth.
Through this project, we aim to showcase the potential of leveraging image processing and deep learning techniques in the dental domain. By elucidating the methodology employed for data collection, augmentation, and model training, we provide a comprehensive overview of our approach to teeth image analysis.

Chapitre 2

Data Augmentation

2.1 Horizontal Image Flipping

The `flip_image_horizontal` function horizontally flips an input image using the `transforms.functional.hflip` transformation.

```
def flip_image_horizontal(img):
    transform = transforms.functional.hflip
    flipped_img = transform(img)
    return flipped_img
```

2.2 Resizing and Cropping Image

```
def resize_and_crop_image(image_path, increment):
    img = Image.open(image_path)
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Resize((img.size[1], img.size[0] + increment)),
        transforms.CenterCrop((img.size[1], img.size[0])),
    ])
    resized_and_cropped_img = transform(img)
    return resized_and_cropped_img
```

2.3 Horizontal Tensor Flipping

```
def flip_image_horizontal_tensor(img_tensor):
    return torch.flip(img_tensor, dims=[3])
```

2.3.1 Resizing and Cropping Tensor

```
def resize_and_crop_image_tensor(img_tensor, increment):
    batch_size, num_channels, height, width = img_tensor.size()
    new_width = width + increment

    # Resize along the width dimension
    resized_tensor = torch.nn.functional.interpolate(img_tensor,
    size=(height, new_width), mode='bilinear')

    # Calculate cropping range for original width
    crop_start = 0
    crop_end = width

    if increment > 0:
        crop_end = min(crop_end, new_width - increment)

    cropped_tensor = resized_tensor[:, :, :, crop_start:crop_end]
    # Crop to original width

    return cropped_tensor
```

2.4 Coordinate Transformation

```
def transform_coordinates(points, original_width, increment):
    width_ratio = (original_width + increment) / original_width

    transformed_points = []
    for x, y in points:
        transformed_x = x * width_ratio - increment / 2
        transformed_points.append((transformed_x, y))

    return transformed_points
```

2.4.1 Coordinate Transformation in JSON

```
def transform_coordinates_in_json(jsondata, original_width, increment):
    json_data = copy.deepcopy(jsondata)
    width_ratio = (original_width + increment) / original_width
```

```
    for obj in json_data['annotation']['object']:
        xmin = int(obj['bndbox']['xmin'])
        xmax = int(obj['bndbox']['xmax'])

        transformed_xmin = xmin * width_ratio - increment / 2
        transformed_xmax = xmax * width_ratio - increment / 2

        obj['bndbox']['xmin'] = str(int(transformed_xmin))
        obj['bndbox']['xmax'] = str(int(transformed_xmax))

    return json_data
```

2.5 Coordinate Flipping in JSON

```
def flip_coordinates_in_json(jsondata, image_width):
    json_data = copy.deepcopy(jsondata)

    for obj in json_data['annotation']['object']:
        xmin = int(obj['bndbox']['xmin'])
        xmax = int(obj['bndbox']['xmax'])

        flipped_xmin = image_width - xmin
        flipped_xmax = image_width - xmax

        obj['bndbox']['xmin'] = str(flipped_xmin)
        obj['bndbox']['xmax'] = str(flipped_xmax)

    return json_data
```

2.6 Coordinate Flipping

```
def flip_coordinates(points, image_width):
    flipped_points = [(image_width - x, y) for x, y in points]
    return flipped_points
```

2.7 Data Augmentation Loop

The following code segment demonstrates how to use the above functions for data augmentation :

```
original = 448
increments = torch.tensor([0, 10, 20, 30, 40])
resized_tensors = []

for increment in increments:
    augmented_tensor = resize_and_crop_image_tensor
    (your_dataset_tensor, increment)
    flipped_augmented_tensor = flip_image_horizontal_tensor
    (augmented_tensor)
    resized_tensors.append(augmented_tensor)
    resized_tensors.append(flipped_augmented_tensor)

flipped_tensor = flip_image_horizontal_tensor(your_dataset_tensor)

resized_tensors.append(flipped_tensor)

augmented_dataset = torch.stack(resized_tensors, dim=0)

print("Augmented dataset size:", augmented_dataset.size())
```

This loop iterates over specified increments, applies resizing and flipping transformations to the dataset tensor, and prints the resulting size of the augmented dataset.
Now, let's proceed with the JSON file manipulations inside the loop :

```
for i in range(48):
    input_file_path = jolp + f"/{i}.json"

    with open(input_file_path, 'r') as file:
        data = json.load(file)

        for j, increment in enumerate(increments):
            augmented_data = transform_coordinates_in_json
            (data, original, increment)

            operation_dir_1 = os.path.join(ajolp, f"operation_{2*j}")
            if not os.path.exists(operation_dir_1):
```

```
        os.makedirs(operation_dir_1)

    output_file_path_1 = os.path.join(operation_dir_1, f"{i}.json")
    with open(output_file_path_1, 'w') as output_file_1:
        json.dump(augmented_data, output_file_1, indent=2)

    flipped_augmented_data = flip_coordinates_in_json(data, original)

    operation_dir_2 = os.path.join(ajolp, f"operation_{2*j+1}")
    if not os.path.exists(operation_dir_2):
        os.makedirs(operation_dir_2)

    output_file_path_2 = os.path.join(operation_dir_2, f"{i}.json")
    with open(output_file_path_2, 'w') as output_file_2:
        json.dump(flipped_augmented_data, output_file_2, indent=2)
```

In this part of the loop, the code reads JSON files, transforms coordinates, creates directories, and writes the augmented JSON data to new files. The process is repeated for both resized and flipped coordinates.

Chapitre 3

Outlines

In this part we will see how to get the outlines of our pictures for better focus on what is important

3.1 outline script

```
def draw_teeth_contours(image_path):
    # Read the image
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Apply unsharp masking for sharpness enhancement
    blurred = cv2.GaussianBlur(image, (0, 0), 3)
    sharpness = cv2.addWeighted(image, 1.5, blurred, -0.5, 0)

    # Apply Canny edge detector
    edges = cv2.Canny(sharpness, 50, 150)

    # Find contours
    contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL,
    cv2.CHAIN_APPROX_SIMPLE)

    # Draw contours on the original image
    result_image = cv2.drawContours(cv2.cvtColor(image, cv2.COLOR_GRAY2BGR),
    contours, -1, (0, 255, 0), 2)

    cv2_imshow(result_image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

FIGURE 3.1 – Result

3.2 Saving Outlines

then we save the outlines in a seperate directory that we will make the model train on later

```
def draw_and_save_contours(image_path, output_path):
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    edges = cv2.Canny(image, 50, 150)
    contours, _ = cv2.findContours(edges,
                cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    contour_image = np.zeros_like(image)

    cv2.drawContours(contour_image, contours, -1, 255, 1)
    os.makedirs(os.path.dirname(output_path), exist_ok=True)
    cv2.imwrite(output_path, contour_image)
```

```
for operation_value in range(9):
    for jpg_value in range(47):
        input_path = f"{afolp}/operation_{operation_value}
                        /{jpg_value}.jpg"
        output_path = f"{acolp}/operation_{operation_value}
                        /{jpg_value}.jpg"
        draw_and_save_contours(input_path, output_path)
```

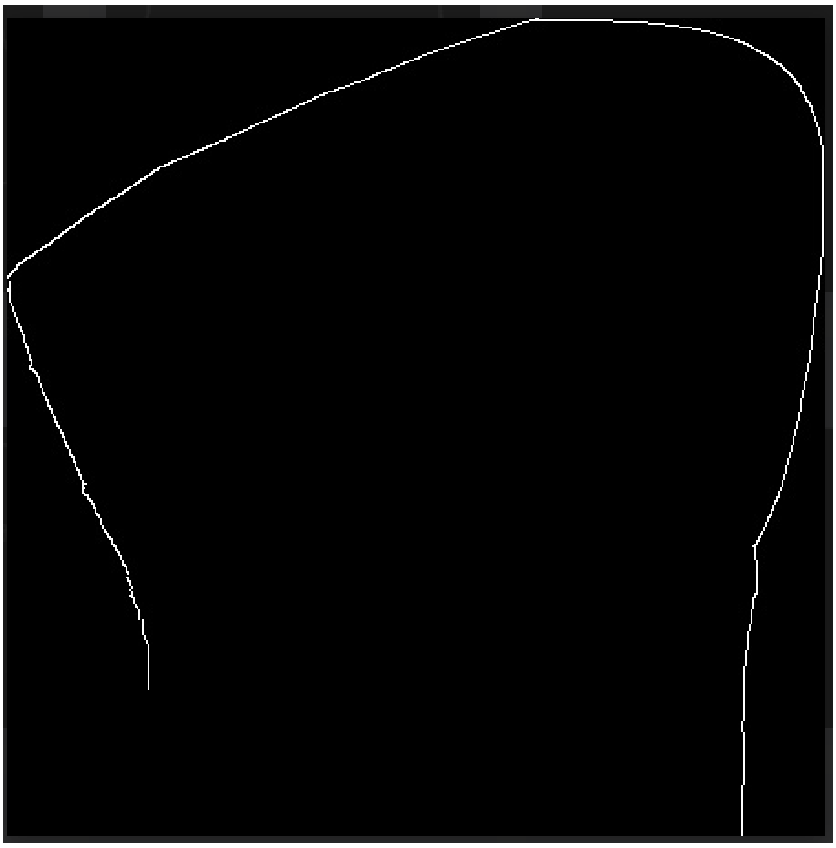

Figure 3.2 – Result

Chapitre 4

Model training

4.1 Model Definition

The following code defines a simple neural network, `CoordinatesPredictor`, for predicting coordinates from black and white images. The network consists of convolutional layers (`conv1`, `conv2`), max-pooling layers (`pool`), and fully connected layers (`fc1`, `fc2`). The output layer (`fc2`) produces two coordinates (x, y).

```
import torch.nn as nn
import cv2
import numpy as np
device = torch.device("cuda")

class CoordinatesPredictor(nn.Module):
    def __init__(self):
        super(CoordinatesPredictor, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 112 * 112, 256)
        self.fc2 = nn.Linear(256, 8)  # Output 2 coordinates (x, y)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 112 * 112)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
```

```
        return x
```

4.2 Image Preprocessing Function

The `get_image_from_path` function processes an input image from a given path. It reads the image in grayscale, applies a binary threshold, and converts it to a PyTorch tensor.

```
# Function to get a black and white image as a 2D tensor from a given path
def get_image_from_path(image_path):
    """
    Get a black and white image as a 2D tensor from the given path.

    Parameters:
        image_path (str): Path to the input image.

    Returns:
        torch.Tensor: Processed image tensor.
    """
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    _, binary_image = cv2.threshold(image, 128, 255, cv2.THRESH_BINARY)
    return torch.tensor(binary_image, dtype=torch.float)
    .unsqueeze(0).unsqueeze(0)
```

4.3 Training Loop

The following code represents the training loop for the `CoordinatesPredictor` model. The loop iterates over a specified number of epochs, loading JSON data, extracting coordinates, and training the model using the Adam optimizer and Mean Squared Error (MSE) loss.

```
\begin{verbatim}
import torch.optim as optim
model = CoordinatesPredictor().to(device)

# Initialize your dataset and dataloader
transform = transforms.Compose([transforms.ToTensor()])
# Initialize the model, loss function, and optimizer
criterion = nn.MSELoss()
```

```
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
num_epochs = 50

for epoch in range(num_epochs):
    for operation_value in range(9):
        for jpg_value in range(47):
            # Load JSON data
            json_path = f"{ajolp}/operation_{operation_value}
            /{jpg_value}.json"
            with open(json_path, 'r') as file:
                data = json.load(file)

            # Extract coordinates from JSON
            coordinates = [[float(obj["bndbox"]["xmin"]),
                int(obj["bndbox"]["ymin"])]
                for obj in data["annotation"]["object"]]

            # Get the input image as a 2D tensor
            image_path = f"{acolp}/operation_{operation_value}
                            /{jpg_value}.jpg"
            input_image = get_image_from_path(image_path).to(device)

            # Assuming input_image is a torch tensor
            optimizer.zero_grad()
            outputs = model(input_image)
            target_coordinates = torch.tensor(coordinates)
                .view(-1).to(device)
            loss = criterion(outputs, target_coordinates)
            loss.backward()
            optimizer.step()

    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')

# Save the trained model
torch.save(model.state_dict(), 'coordinates_predictor_model.pth')
```

The loop processes each image in the dataset, computes the loss, and updates the model parameters. The final trained model is saved as 'coordinates_predictor_model.pth'.

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/loss.
  return F.mse_loss(input, target, reduction=self.reduction)
Epoch [1/10], Loss: 37912.7891
Epoch [2/10], Loss: 16765.1172
Epoch [3/10], Loss: 28989.5840
Epoch [4/10], Loss: 7781.4482
Epoch [5/10], Loss: 6753.6079
Epoch [6/10], Loss: 3831.1450
Epoch [7/10], Loss: 2350.6182
Epoch [8/10], Loss: 2210.8333
Epoch [9/10], Loss: 841.1655
Epoch [10/10], Loss: 6073.4912
```

FIGURE 4.1 – Training Loss Over Epochs

and after many tweaking

```
Epoch [1/10], Loss: 201.2026
Epoch [2/10], Loss: 195.1530
Epoch [3/10], Loss: 202.1035
Epoch [4/10], Loss: 153.1508
Epoch [5/10], Loss: 148.5576
Epoch [6/10], Loss: 130.4602
Epoch [7/10], Loss: 77.5002
Epoch [8/10], Loss: 66.0755
Epoch [9/10], Loss: 54.5706
Epoch [10/10], Loss: 72.6252
```

FIGURE 4.2 – Training Loss Over Epochs improved

Chapitre 5

Results

here are the results of the predecting model plotted on top of the image

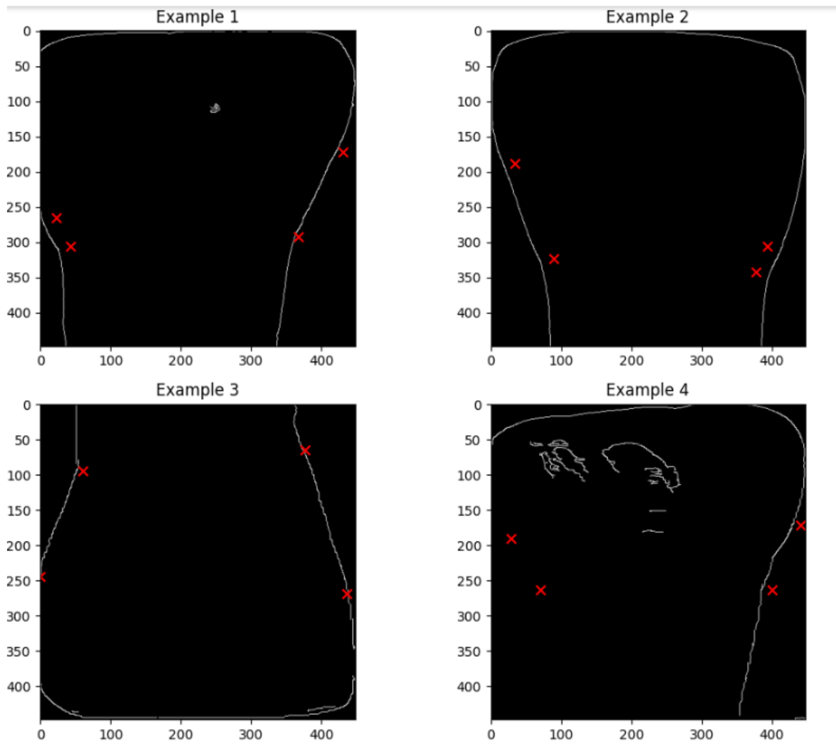

FIGURE 5.1 – Training results

to put them more clearly on top of the original images

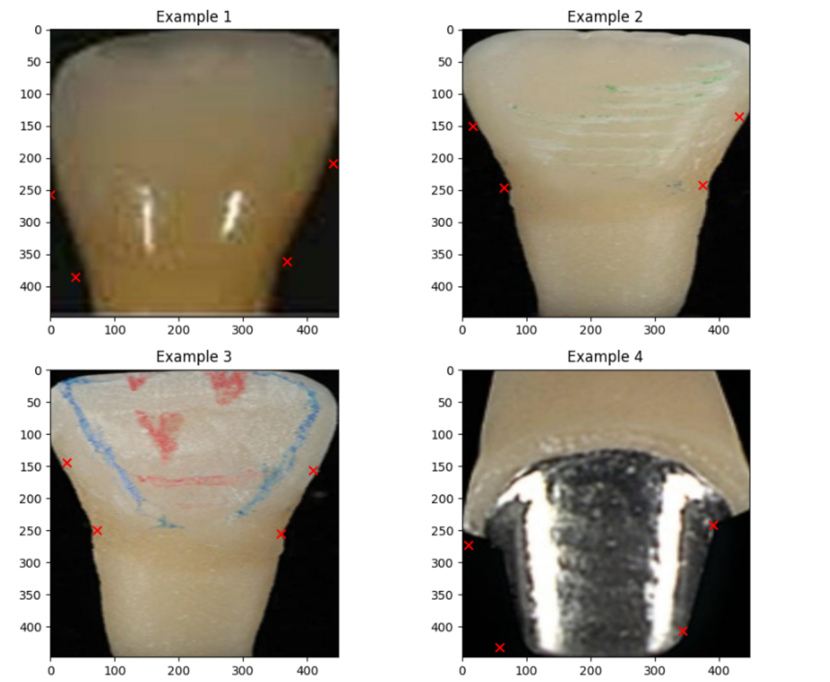

Figure 5.2 – Training results 2