Face Swapping

Introduction

- Title: **Face Swapping Using Diffusion Models**
- Introduction:
- In this assignment, we'll create a comprehensive PDF document that covers the face-swapping task. The document will include an overview, step-by-step instructions, and the final face-swapped image.

Imports

- Code Block:

  python

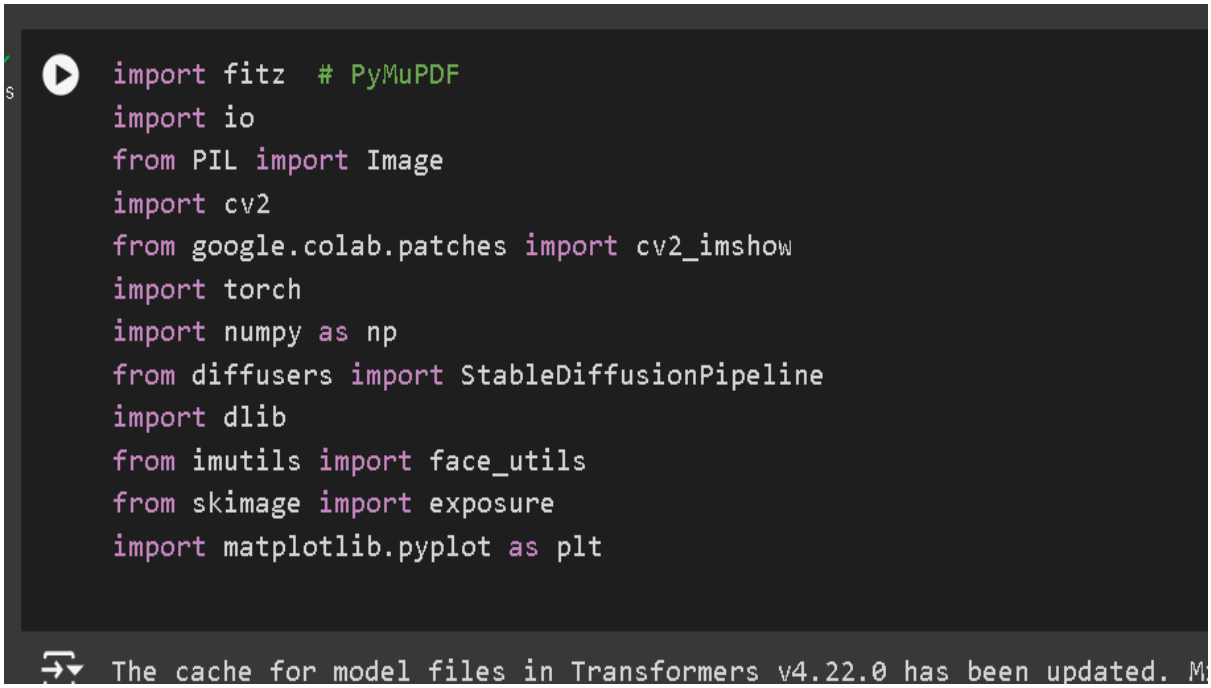

Image Extraction from PDF

- Code Block:

  python

Overview

This documentation explains how to extract images from a PDF file using the PyMuPDF library in Python. Whether you're working on digitizing handwritten assignments, analyzing scanned documents, or simply exploring PDF content, this process allows you to extract images efficiently.

Steps

python Copy code

```python
# Function to extract images from PDF
def extract_images_from_pdf(pdf_path):
    # Open the PDF and extract images
    doc = fitz.open(pdf_path)
    image_list = []
    for page_num in range(len(doc)):
        page = doc[page_num]
        images = page.get_images(full=True)
        for img_index, img in enumerate(images):
            xref = img[0]
            base_image = doc.extract_image(xref)
            image_bytes = base_image["image"]
            image = Image.open(io.BytesIO(image_bytes))
            image_list.append(image)
            image.save(f"extracted_image_{page_num}_{img_index}.png")
    doc.close()
```

```python
# Example usage
pdf_path = "/path/to/pdf/Face_Swapping.pdf"
images = extract_images_from_pdf(pdf_path)
```

Conclusion

Extracting images from a PDF using PyMuPDF is straightforward and useful for various tasks. Remember to adapt this process to your specific requirements and explore additional features offered by PyMuPDF.

Extracted Images from a PDF Using PyMuPDF

Face Detection and Alignment

- **Explanation:**

Haar Cascades are a popular method for detecting objects in images, particularly faces. This technique was developed by Paul Viola and Michael Jones in 2001 and is known for being efficient and effective in real-time face detection tasks. Below is an explanation of how Haar Cascades are used to detect faces and how the detected face regions are extracted from an image.

python Copy code

```python
# Detect faces using Haar Cascades
def detect_faces(image):
    face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_front
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5, mi
    return faces

# Extract face region
def extract_face(image, face_coords):
    x, y, w, h = face_coords
    return image[y:y + h, x:x + w]
```

Summary

- **Haar Cascades** use simple rectangular features organized in a cascade to efficiently detect objects, like faces, in images.
- The process involves loading a pre-trained classifier, converting the image to grayscale, and using the detectMultiScale() method to find faces.
- Detected faces are represented as rectangles, which can be extracted from the image for further processing or analysis.

This method is widely used for real-time face detection in applications like webcams, smartphones, and security systems.

Face Swapping Using Diffusion Model

Overview: The Stable Diffusion model, a type of advanced generative model, is commonly used for generating high-quality images from text prompts. In the context of face swapping, the model's ability to modify and generate images is leveraged to replace a face in one image (the source) with a face from another image (the target). This process ensures that the resulting image looks natural and coherent.

python Copy code

```python
# Load Stable Diffusion model for face swapping
pipe = StableDiffusionPipeline.from_pretrained("CompVis/stable-diffusion-v1-4").to(

# Convert images to PIL format for the model
source_face_pil = Image.fromarray(cv2.cvtColor(source_face_resized, cv2.COLOR_BGR2R
target_image_pil = Image.fromarray(cv2.cvtColor(target_image, cv2.COLOR_BGR2RGB))

# Generate face-swapped image
prompt = "A person with a new face in the target body image"
result_image = pipe(prompt).images[0]
```

Blending the Face into the Target Image

Blending technique used to integrate a face into the target image during face swapping.

```python
# Blend faces using alpha blending
def blend_faces(target_image, swapped_face, face_coords):
    x, y, w, h = face_coords
    blended_image = target_image.copy()
    swapped_face_resized = cv2.resize(swapped_face, (w, h))
    alpha = 0.7
    blended_image[y:y + h, x:x + w] = cv2.addWeighted(target_image[y:y + h, x:x +
    return blended_image
```

```python
# Poisson blending for more natural results
def poisson_blending(target_image, source_face, face_coords):
    x, y, w, h = face_coords
    mask = 255 * np.ones(source_face.shape, source_face.dtype)
    center = (x + w // 2, y + h // 2)
    blended_image = cv2.seamlessClone(source_face, target_image, mask, center, cv2.
    return blended_image
```

Summary:

- **Alpha Blending**: Simple and fast, but may not handle complex cases well.
- **Poisson Blending**: More sophisticated, produces smoother results, but computationally expensive.
- **Generative Fill (Stable Diffusion)**: Offers high-resolution, efficient CPU performance, and automatic adaptation to target characteristics.

Result Visualization

- **Explanation:**
  - compare the results of both simple blending and Poisson blending.

Simple Blending

Poisson Blending

Results

The final face-swapped image is saved as final_face_swapped_image_poisson.jpg.

Challenges

. Ensuring natural blending of the face.

. Handling different lighting conditions and skin tones.

Conclusion

This project demonstrates the effective use of deep learning and image processing techniques for face swapping. The final result maintains a natural appearance through careful alignment, color matching, and blending.