WINTER IN DATA SCIENCE – MID TERM REPORT AUGMENTED REALITY FACE TRACKING AND OVERLAY SYSTEM

Nitin Bamnawath

25B3936

Table of Contents

This report summarizes my work during the first four weeks of the "Winter in Data Science" (WiDS) program. The main goal of this project was to build a computer vision application that works like a social media face filter.

The project started from scratch. In the beginning, I focused on learning the basics of Python and understanding how computers handle images. I built a simple tool to edit photos, which taught me how to resize images and sharpen them to make them look clearer.

Next, I worked on "Background Removal." I learned how to write code that can automatically cut a person out of a photo, separating them from the scenery behind them. This is a crucial step for creating stickers or overlays.

Finally, I moved on to working with video. I used a tool called MediaPipe to track human faces. I wrote a script that can access my laptop's webcam, find my face in real-time, and stick an image (like a mask or a logo) onto my nose or forehead. The system allows me to switch the position of the sticker using the keyboard.

This report explains the steps I took, the logic behind the code, and the results I achieved each week.

Introduction

1 Overview

We see Computer Vision everywhere today—from unlocking our phones with our faces to cars that can see the road. This project was my opportunity to look "under the hood" and understand the technology.

The specific focus of project was **Augmented Reality (AR)**. AR is when you take a live video of the real world and layer digital objects on top of it. The goal was to build a system that can track a face and overlay images on it, just like a instagram filter.

2 Project Roadmap

The project was broken down into four distinct stages:

- **Week 0 (The Basics):** Learning to load images and change their colors or size using code.
- **Week 1 (Clean Cutouts):** Learning how to remove the background from an image so we have a clean "sticker" to use later.
- **Week 2 (Finding the Face):** Learning how to detect faces in a pre-recorded video file.
- **Week 3 (Going Live):** Connecting everything to the live webcam and allowing the user to interact with the system in real-time.

3 Tools Used

- **Python:** The programming language used for all the logic.
- **OpenCV:** A powerful library used for processing images and accessing the camera.
- **MediaPipe:** A Google library used to detect the geometry of the face.
- **NumPy:** A scientific computing library used for performing high-speed matrix multiplication.
- **Pillow:** A library used to handle image file loading.

Week 0: Python Basics + Introduction to Computer Vision

The first thing I had to learn was how a computer "sees" an image. To us, a picture is a collection of shapes and colors. To a computer, it is just a big grid of numbers.

I used a library called **NumPy** to handle these grids.

- Every image is a matrix (a grid).
- Each cell in the grid is a pixel.
- Each pixel has three numbers representing Blue, Green, and Red.

One interesting thing I learned is that OpenCV loads images in **BGR** format (Blue-Green-Red), while most other tools use **RGB**. I had to be careful to convert between them, or else my images looked blue!

In week 0 we were asked to create a basic image editor, it was with help of python libraries such as pillow, OpenCV and used matplotlib for final output.

1. Initially file was opened with help of pillow

   ```
   from PIL import Image : Importing its library
   Image.open("image.jpg") : Importing image
   ```

2. Convert RGB to BGR
   With the help of OpenCV we will convert image format from RGB to BGR
   Note: (OpenCV handles images in BGR format by default, whereas Pillow opens them in RGB, but since we are using matplotlib for image output which supports RGB by default it doesn't pose a problem)
   ```
   cv2.cvtColor(imagecv2, cv2.COLOR_RGB2BGR)
   ```

3. Resize
   ```
   cv2.resize(imagecv2,(x,y))
   ```

4. Sharpen To sharpen the image, we defined a custom kernel and applied it using filter2D:
   ```
   kernel_sharpening = np.array([[-1,-1,-1],[-1,9,-1],[-1,-1,-1]])
   finalimage = cv2.filter2D(imagecv2, -1, kernel_sharpening)
   ```

5. Image output by matplotlib.pyplot
   ```
   plt.imshow(finalimage)
   ```

6. Saving image as
   ```
   plt.imsave("final_output.jpg", finalimage)
   ```

Week 1: Background Removal + Overlaying on Static Images

The goal for Week 1 was to implement background removal on static images without relying on external removal tools. I utilized the GrabCut Algorithm, a simpler alternative to Deep Learning Semantic Segmentation, to extracting a foreground object and overlaying it onto a new background.

Used the grabcut algorithm by OpenCV

```
cv2.grabCut(img,mask,rect,bgdModel,fgdModel,5,cv2.GC_INIT_WITH_RECT)
```

#parameter(img, mask, rect, bgmodel, fgmodel, iter, method)

By providing a bounding box (rect), we define the area we want to work with. This separates the image from its probable background. We initialized the background and foreground models (bgdModel, fgdModel) as zero arrays, which the algorithm uses internally to learn the color distribution during the 5 iterations.

Overlay Logic After generating the mask, I inverted it to create a "hollowed out" space in the background image and filled it with the foreground object. This creates a seamless cutout effect.

Then by using this we got a mask

```
img_backcut = img_back[coordinates_y : coordinates_y + y , coordinates_x :
coordinates_x + x]
img_backcut = img_backcut * ( 1 - mask_resize[:,:,np.newaxis] ) +
img_resize
```

"Since our image has 3 channels (Blue, Green, Red) but our mask only has 1 channel (Grayscale), we use np.newaxis to add a third dimension to the mask. This allows us to multiply the 3D image array by the mask without shape errors."

`img_backcut * (1 - mask):` *This turns the pixels black (0) where the Pepsi logo will go. It "punches a hole" in the background.*

`+ img_resize`*: This fills that black hole with the Pepsi logo pixels.*

Week 2: Facial Landmarks Detection (no webcam)

For Week 2, the project advanced from static images to real-time video processing. I developed an Augmented Reality (AR) application that tracks a user's face and overlays the image like in Week 1 onto the nose in real-time.

Facial Landmark Detection I utilized MediaPipe Face Mesh, which maps 468 distinct 3D landmarks on the user's face.

```
facial_landmarks.landmark[4] #nose land mark
```

Nose Tip (Index 4): Used as the anchor point for the center of the overlay.

Cheeks (Indices 234 & 454): Used to calculate the width of the face.

Nose Tip (Index 4): Used as the anchor point for the center of the overlay. Cheeks (Indices 234 & 454): Used to calculate the width of the face.

Dynamic Scaling Logic Instead of a static size, the overlay resizes dynamically as the user moves closer or further from the camera.

```
# Calculate real-time face width
face_width = right_cheek_x - left_cheek_x

# Maintain Aspect Ratio for the overlay height
face_height = int(face_width * (original_overlay_height /
original_overlay_width))
```

6. Week 3: Real-Time Webcam Implementation

6.1 System Architecture

In Week 3, I integrated all previous components into a real-time application using the laptop's webcam. The system runs in an infinite `while` loop that performs the following steps roughly 30 times per second:

1. **Capture:** `cv2.VideoCapture` grabs a frame.
2. **Process:** MediaPipe detects the 468 landmarks.
3. **Calculate:** The code computes the position and size of the face.
4. **Render:** The overlay is resized and mathematically added to the frame.
5. **Display:** The final composite image is shown to the user.

6.2 Coordinate Mapping

The AI model returns coordinates as percentages (0.0 to 1.0). To draw on the screen, these must be converted to pixels.

- **Formula:** `Pixel Coordinate = Normalized Value * Screen Resolution`

Centering Logic: Simply drawing an image at the nose coordinates places the top-left corner of the image on the nose. To center it, I implemented an offset calculation:

- `Start X = Nose X - (Image Width / 2)`
- `Start Y = Nose Y - (Image Height / 2)`

6.3 User Interaction

To make the system interactive, I added keyboard controls:

- **'1' Key:** Moves the overlay to the **Forehead**.
- **'2' Key:** Moves the overlay to the **Nose**.
- **'3' Key:** Moves the overlay to the **Mouth**.
- **'q' Key:** Quits the application safely.

This required creating a "State Variable" in the code that remembers the current active mode and updates the coordinate logic accordingly.

.

Conclusion

1 Challenges

- **Coordinate Negativity:** When a user moves to the edge of the screen, the calculated start coordinates for the overlay often become negative. Attempting to draw at a negative index causes the program to crash.
    - *Solution:* I implemented a manual clipping check. If `x < 0`, the code slices the overlay image to remove the off-screen portion before drawing.
- **Color Space Confusion:** OpenCV reads BGR, but MediaPipe expects RGB. Forgetting to convert the frame before processing resulted in the face detection failing or being inaccurate.
    - *Solution:* Added a dedicated `cv2.cvtColor` step at the start of the main loop.
- **Latency:** Adding too many heavy operations (like high-resolution resizing) slowed down the video feed.
    - *Solution:* I prioritized efficient math (NumPy vectorization) over Python loops to keep the frame rate high.

2 Conclusion

This project has successfully demonstrated the end-to-end pipeline of a computer vision application.

1. **Week 0** established the mathematical foundation of pixel manipulation.
2. **Week 1** provided the tools to create clean digital assets via background removal.
3. **Week 2** enabled the system to "understand" the geometry of the user.
4. **Week 3** synthesized these skills into a live, interactive product.

The final result is a robust AR system that replicates the core functionality of commercial filters, built entirely from scratch using Python code. This experience has provided deep insights into matrix algebra, algorithmic efficiency, and the practical challenges of real-time software engineering.