Computer Vision Models Assignment (Last one phew)

The goal of this assignment is to give you hands-on practice with the following concepts:

- Downloading and running vision models from hubs
- Multi-object detection in complex scenes
- Semantic Segmentation
- Pose Estimation

The objective of this assignment is for you to get some practice interacting with *pretrained* models. The computer vision community online is large and active, and many people have trained models and made them available for others to use. Sometimes, models available online have been trained on very large datasets or datasets that are not publicly available, so you would not be able to train them yourself. If you need a bespoke model for some task, a very common practice nowadays is to take a pretrained model and *fine-tune* it on your own dataset. Often this means using a pre-trained *backbone* and training a new *head* on top of it specific for your task.

Structure of the assignment

You've been provided with some skeleton code. The main entry point is `run.py`, which has two important command-line arguments: `--image` and the model type. `--image` can be set to a file path to an image, or to `live` to use your computer's webcam (must be run locally, not on the server, to use `live` mode). For example,

```
python run.py --image=solvay.jpg face
```

will run face detection using the Viola-Jones algorithm on the `solvay.jpg` image. You can also run it on your webcam:

```
python run.py --image=live face
```

There are five model types you can choose from (and the purpose of this assignment is for you to implement their behavior): `face`, `pose`, `yolo`, `segment`, and `sunglasses`. Each of these will be described in more detail below. As always, the code you need to write is marked with `YOUR CODE HERE` comments. You don't need to make any changes to `run.py` itself, but you may find it useful to look inside and see how the protocol for the specific models is defined and used. In particular, note that images being passed to the `detect_and_draw` are always in the OpenCV format (numpy

arrays of shape `(H, W, 3)` with BGR channel order and `uint8` data type), while PyTorch models generally expect `(B, C, H, W)` tensors (or a list of `[(C, H, W)]` tesnors) with RGB channel order and `float32` data type.

For each of the 4 problems below, your task is to read the documentation on how the model works and/or use your debugger to inspect the format of the model's output, and use that information to draw annotations on the image using OpenCV drawing functions.

Problem 1: Face detection with OpenCV and Viola-Jones

In this problem, you will use OpenCV's implementation of the Viola-Jones face detector to detect faces in images. You saw a demo of this in class. You will use the `cv2.CascadeClassifier` class to detect faces in images.

For problem 1, do the following:

1. Download the parameters of the Haar cascade from the OpenCV GitHub **(click here)**.
2. Fill in the missing few lines in `face.py`. Draw 1-pixel wide red rectangles around the faces. Set the `lineType` argument to `cv.LINE_AA` to make the lines look smoother.
3. Run `python run.py --image=solvay.jpg face` and verify it produces the following image:

Figure 1: Output of `python run.py --image=solvay.jpg face`

4. Run it with `--image=live` or other images of your choice and verify that it is properly detecting faces. (We will test your code on more images than just `solvay.jpg`).

You will be graded only on whether your code works as expected in steps 1 and 2 above, but you are strongly encouraged to play around with the `live` mode and develop a sense of when it works and when it fails!

Problem 2: keypoints and pose estimation

For problem 1, you downloaded and ran a pretrained model in OpenCV in the format of an XML file. For problems 2 through 4, you will download and run pretrained deep neural networks (PyTorch models). Nowadays, there are multiple "hubs" online where you can do this including

- torchvision
- torch.hub

- huggingface
- timm
- OpenMMLab
- (And probably some others that you'll be able to find easily that I haven't encountered yet)

To avoid learning a whole new tech stack, we'll use the `torchvision.models` hub for poses and segmentation (Problems 2 and 3), and a model hosted on `torch.hub` for object detection with the yolo architecture (Problem 4 below).

Some models trained on the COCO dataset combine object detection with pose estimation into a single model. Here, you'll use the Keypoint RCNN and visualize both (i) bounding boxes on objects and (ii) a skeletal pose of any detected people. This docs page will be a useful resource as you work out how to interpret the output of the model. However, while torchvision provides a set of visualization tools for you, for this assignment **you must use OpenCV drawing functions like `cv.rectangle`,`cv.putText`, and `cv.line`.** This is both so that you'll be using familiar drawing tools, and because we want to ensure that you take time to look at and interpret the outputs of the models. It is also good practice in converting back and forth between torch tensors and numpy arrays and handling datatypes.

For problem 2, do the following:

1. Fill in the missing drawing code in `pose.py`. In particular, for every bounding box that scores above `detection_quality_threshold`:
    - Draw a red rectangle around the detected object with the label and write the label of the object 5 pixels above the rectangle using `cv.putText(img, label, (x1, y1 - 5), cv.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1, cv.LINE_AA)` where `(x1, y1)` are the top-left coordinates of the bounding box.
    - If the label is "person", draw the skeleton using `PoseEstimator.coco_keypoints` and `PoseEstimator.coco_skeleton`. You should only draw points that are detected with a confidence above `keypoint_quality_threshold`. Use `cv.circle(img, (x, y), 1, (0, 255, 0), 1, cv.LINE_AA)` to draw keypoints and `cv.line(img, (x1, y1), (x2, y2), (255, 0, 0), 1, cv.LINE_AA)` to draw the skeleton.
2. Run `python run.py --image=bowl.jpg pose` and verify it produces the following image:

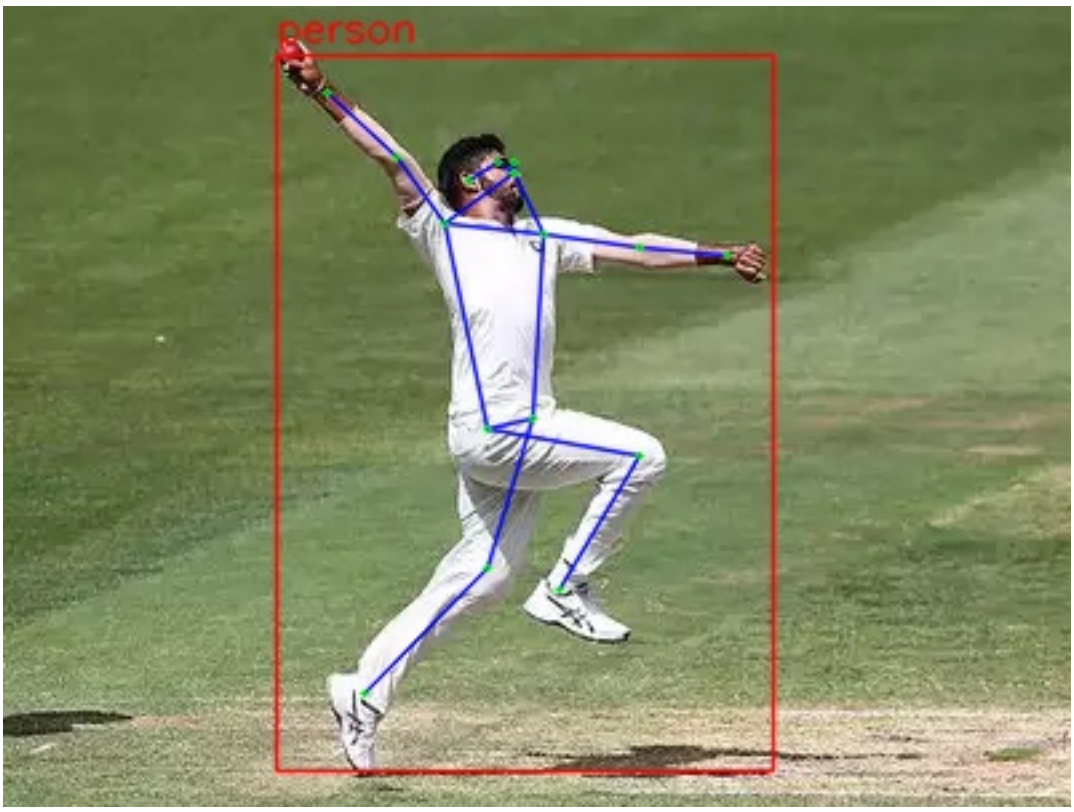

Figure 2: Output of `python run.py --image=bowl.jpg pose`

3. Run it with `--image=live` and verify that it detects poses in your webcam feed. If your personal computer supports CUDA, you may want to see if you can speed it up by moving the model and image to the GPU (not required, but a useful exercise).

Problem 3: Semantic segmentation

In this problem, you will use a pretrained model to perform semantic segmentation. Semantic segmentation is the task of assigning a class label to each pixel in an image. We will again use a pretrained model from `torchvision.models` to perform this task.

For problem 3, do the following:

1. Fill in the missing drawing code in `segment.py`. The output of the model contains a set of logits per pixel. Use `torch.argmax` to get the most likely class for each pixel, and then use the `color_per_class_bgr` array to convert each integer class label to a color (this can be done in a vectorized fashion using `image = color_per_class_bgr[label_per_pixel]`). Finally, for visualization purposes, we will blend the original image with the segmentation

mask. You should return a new image that is `1-self.alpha` times the original image and `self.alpha` times the segmentation mask. You may use `cv.addWeighted` to do this, which will ensure `uint8` outputs.

2. Run python run.py --image=jax.jpg segment

Figure 3: Output of `python run.py --image=jax.jpg segment`

3. Run it with `--image=live` and verify that it segments images in your webcam feed. Again, you may want to see if you can speed it up by moving the model and image to the GPU if you have access to one.

Problem 4: Object detection with YOLO

This one is slightly different from the above because `torchvision.models` does not contain a YOLO model, but we can get one from a third party using the `torch.` hub interface. This is why, unlike previous assignments, we need the additional `ultralytics` package in `requirements.txt`. Again, running the mdoel been done for you in `yolo.py`, but you will need to fill in the missing code to draw the output.

For problem 4, do the following:

1. Fill in the missing drawing code in `yolo.py`. Like in Problem 2, you should draw a rectangle around each above-threshold detected object. Unlike in Problem 2, you will color the bounding box according to the detected class using the provided `Colors()` object. Also unlike Problem 2, the yolo model has a `model.conf` attribute which can be set *before* running the model to change the confidence threshold and only return detections above that threshold.

   In addition to the bounding box, you should also write the label of the object 5 pixels above the rectangle using `cv.putText(img, label, (x1, y1 - 5), cv.FONT_HERSHEY_SIMPLEX, 0.5, color, 1, cv.LINE_AA)` where `(x1, y1)` are the top-left coordinates of the bounding box and `color` is the color corresponding to the object label, as defined in the `Colors()` object.

2. Run `python run.py --image=amsterdam.jpg yolo` and verify it produces the following image:

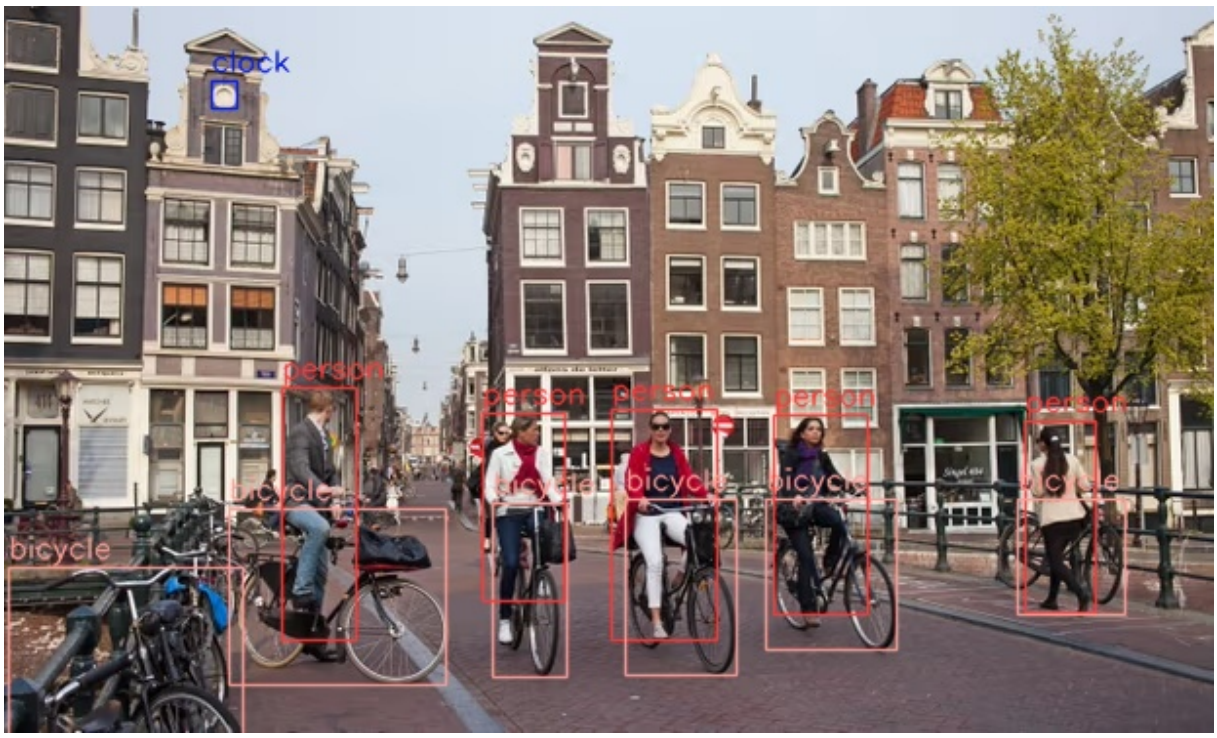

Figure 4: Output of `python run.py --image=amsterdam.jpg yolo`

3. Run it with `--image=live` and verify that it detects poses in your webcam feed. Same instructions as above.

Problem 5: Sunglasses filter

Now that you've seen face detection and keypoint pose estimation, this problem involves:

1. detecting faces in an image
2. for each detected face, detecting keypoints on the face
3. drawing a pair of sunglasses on the face

Modify the missing code in sunglasses.py. The face detector and pose estimator objects are already configured for you (downloaded from dlib). **Click here** for a good tutorial on how to use them to detect keypoints. Importantly, pay attention to whether the keypoint IDs are zero-indexed or one-indexed.

Here's how I suggest writing the `detect_and_draw` method… For each detected face,

1. Pick four keypoints that define a quadrilateral around the eyes, something like this:

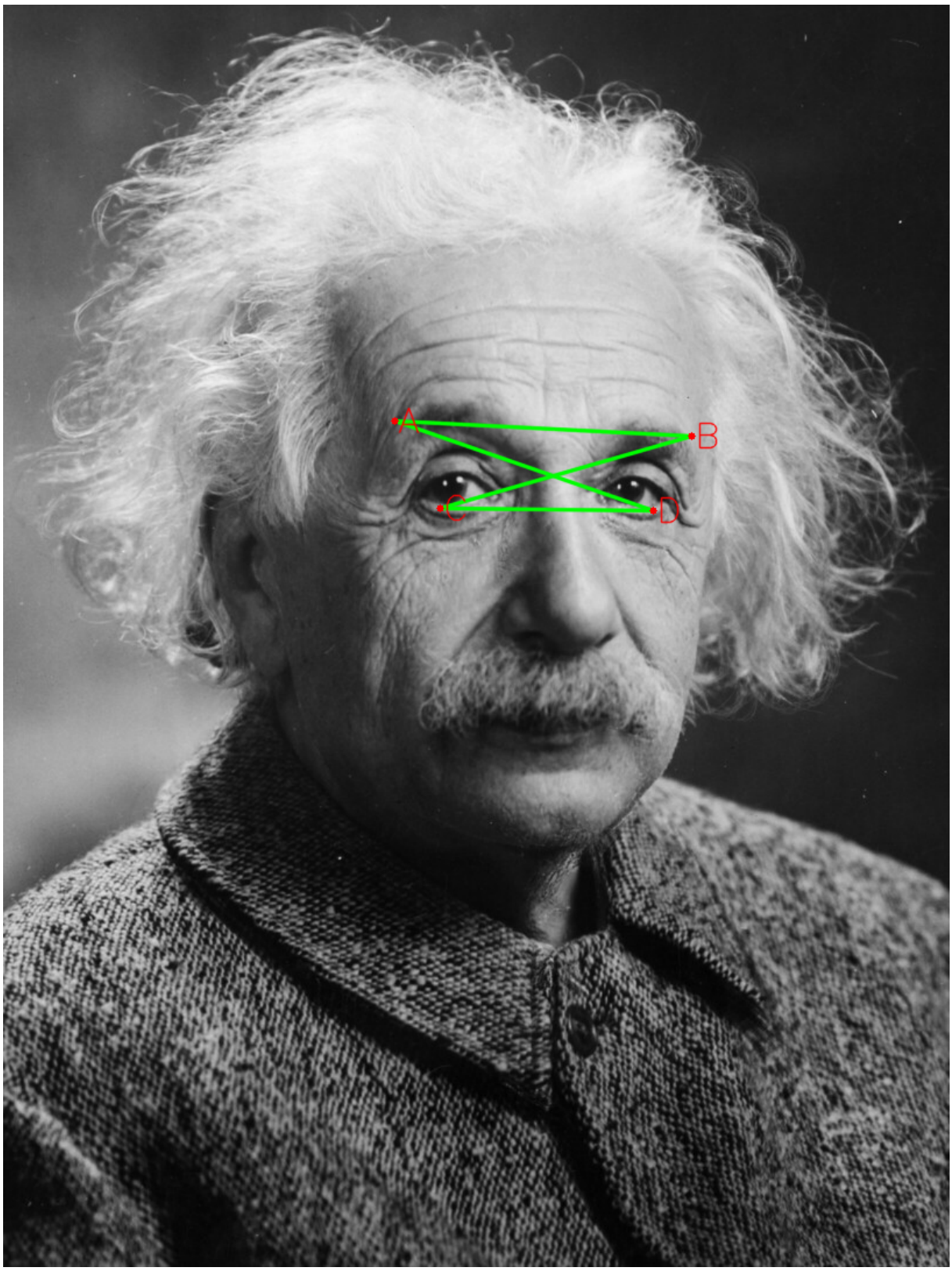

Figure 5: Four facial keypoints ABCD

2. Pick four corresponding keypoints on the provided sunglasses image, something like this:

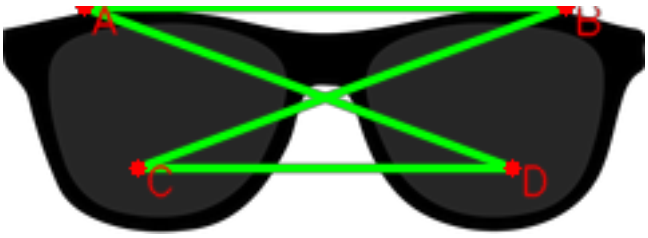

Figure 6: Four sunglasses keypoints ABCD

3. Use `cv.getPerspectiveTransform` to solve for the homography that maps the sunglasses into the face image coordinate frame.
4. Use `cv.warpPerspective` to do the warping.
5. Use a weighted average of the two images, using the 4th (alpha) channel of the sunglasses image as the weights. Something like `output = (1 - alpha) * face + alpha * sunglasses` should work.

If all goes well, you'll generate images like this:

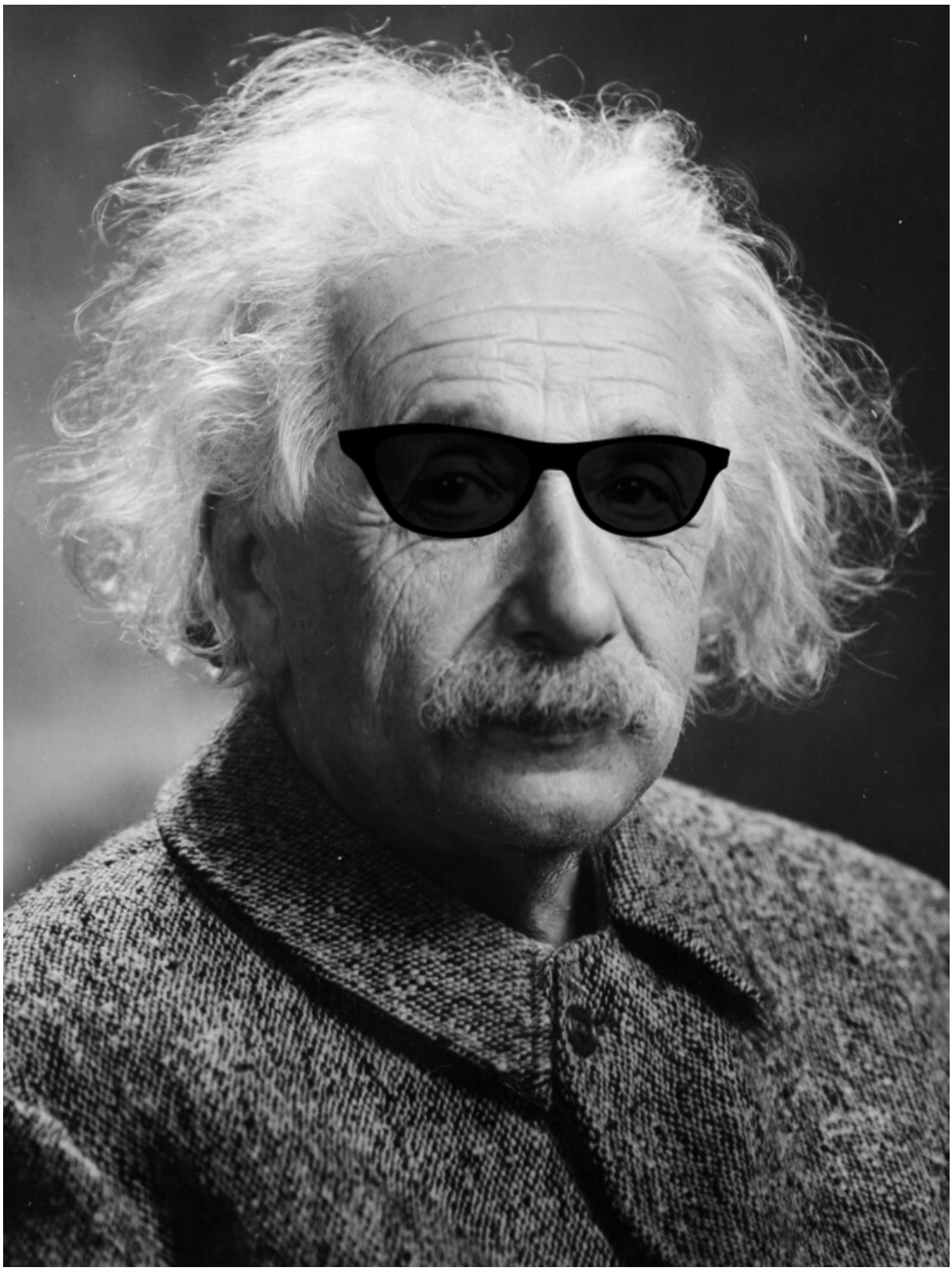

Figure 7: Output of `python run.py --image=einstein.jpg sunglasses`

And you should be able to run it “live” like you would a snapchat filter. Note that you’ll get more robust results if you define the face quadrilateral in terms of facial features that move around less (e.g. if you map the sunglasses to where the eyebrows are, then they will move/stretch as you raise your eyebrows).

Collaboration and Generative AI disclosure

Did you collaborate with anyone? Did you use any Generative AI tools? Briefly explain what you did in the `collaboration-disclosure.txt` file.