CSCI 1430 Final Project Report: SNAPMAP

Aliya Chambless, Ambika Miglani, Adam Shelby, Victoria Xu.
Brown University
9th May 2017

1. Introduction

Have you ever wanted to play around with face filters on your webcam and not just on your phone (not just on snapchat)? Our project does just that. We wanted to combine our creative visions with computer vision to make our own face filters, focusing more on turning users' faces into whacky pieces of art and not so much on features enhancement (as many snapchat filters do). Users can run our code through terminal and have fun becoming 6-eyed, 3-mouthed humans (using the -f multi flag in terminal), or turn themselves into plant nymphs (using the -f plant flag in terminal)! Shown below are the two face filters we made that users can choose from.

Figure 1. *Left* multi face filter *Right* plant face filter

2. Related Work

Citations are written into a .bib file in BibTeX format, and can be called like this: Alpher et al. [**?**]. Here's a brief intro: webpage. *Hint:* $> pdflatex %docu, bibtex %docu, pdflatex %docu, pdflatex %docu

To complete our project, we took inspiration from several tutorials that worked with feature detection and image mapping. Our primary source was a tutorial by Shantanu Tiwari, and we used the Kaggle dataset which provided us with training and testing images including openCV. We essentially used these resources in addition to the Viola jones face detector to complete our project, and used tutorials provided by OpenCV2 on using the haar cascade classifiers.

3. Method

We decided to use a CNN to train a data set instead of implementing traditional computer vision methods, because of the difference in accuracy between the two methods that we observed in our previous projects. In our webcam class, we decided to implement a cheap viola jones face detector to crop faces out of the input video frame (as the kaggle dataset was trained on close-up faces), and implementing a feature detector that worked with that restricted portion of the image.

The method of putting together our CNN was to research and look into how other people had approached the problem and adapted their models for our own use. The main CNN that we based our model off of was one done by Peter Skvarenina but used Flatten() instead of GlobalAveragePooling2D(), as we got a higher accuracy as shown in the table below. We ran it for 50 epochs each time and recorded the results before choosing our method. We also experimented with the keras ImageDataGenerator to augement the data. We wanted to add rotation to the dataset, but this decreased our accuracy to 0.5180. We believe that the keras ImageDataGenerator was not rotating the facial keypoints, so we did not use this feature.

	Global Average Pooling()	Flatten()
us/step	914	930
loss	0.0072	0.0062
accuracy	0.6092	0.6343
val_loss	0.0255	0.0247
val_acc	0.2967	0.2734

At first, the CNN was optimizing across all the images instead of for each image in the dataset, which meant that the CNN gave us the same points for every face. As shown in Figure 2 Left, the points are the same no matter what. But, once we made sure to optimize for each individual image instead of across all of them together, we got much better

results as shown in Figure 2 Right. That result was after just 5 epochs.

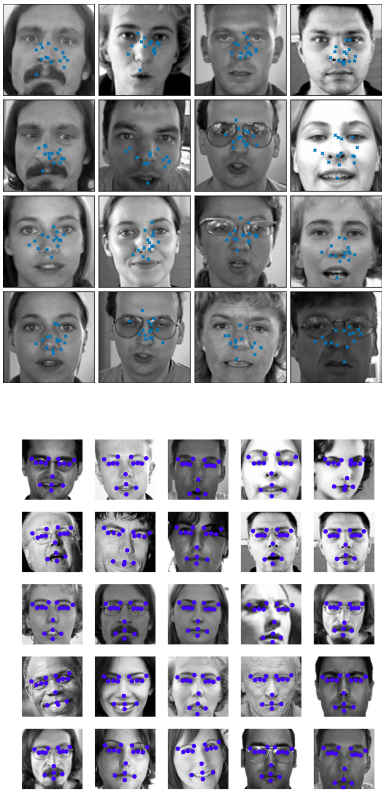

Figure 2. *Left* CNN, bad results *Right* CNN, good results after just 5 epochs

For image filter mapping, we first worked on getting a simple filter with the points for the two eyes, nose, and mouth to align with feature points detected on the face. Based on these four points, we would skew, rotate, or scale the filter image accordingly. Once that was working, we then worked on making our filters more interesting and playful.

To plot the facial features, we created a dictionary and mapped the feature names to an index in an array corresponding to an index in the fig array, which represents the positions for the features.

We ran into some issues to do with maintaining a high resolution and frame rate once the filters were applied. In Figure 3 are screenshots of the result of our code with lower frame rate and resolution. A screenshot of the result when run with a higher frame rate and resolution can be found in Figure 4 under the results section.

4. Results

Once changing the filter mapping code, matchup, from a for loop to numpy operations, the framerate increased significantly. Likewise, we mapped the locational data back to the scale of the high resolution filter and video image to preserve resolution as opposed to resizing the filter to 96x96 and failing to rescale. These improvements can be viewed below in Figure 3, and also in the accompanying video titled final_results.

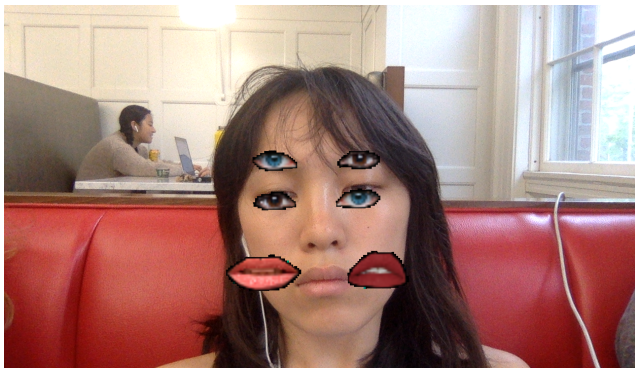

Figure 3. *Top* Low-res, low frame rate

Figure 4. High-res, faster frame rate

This is the code before, using a for loop to map:

```
filter_over_spots = np.array(np.where(
    rotated_filter != 0))
for index in range(len(
    filter_over_spots[0])):
     point = [filter_over_spots[0][index
         ], filter_over_spots[1][index]]
     displacement_from_center = [
         rotated_center[0] - point[0],
         rotated_center[1] - point[1]]
     try:
          face_image[face_nose[0] -
              displacement_from_center
              [0]][face_nose[1] -
              displacement_from_center[1]]
               = rotated_filter[point[0],
              point[1]]
     except IndexError:
          0
```

This is the code after, using numpy operations:

```
# creates mask for filter, where zeroes are replaced with -5000's
    mask = np.ma.array(rotated_filter, mask=(rotated_filter == 0), fill_value = -5000.)
    fixed_filter = mask.filled()
    filter_half_ht, filter_half_wd = np.array(fixed_filter.shape) / 2

    # takes a sample of the face where the filter would be imposed and negates it
    sample = face_image[max(face_nose[0] - math.floor(filter_half_ht), 0):min(face_nose[0] + math.ceil(filter_half_ht), face_image.shape[0]),max(face_nose[1] - math.floor(filter_half_wd), 0):min(face_nose[1] + math.ceil(filter_half_wd), face_image.shape[1])]
    sample = np.negative(sample)
    sample_half_ht, sample_half_wd = np.array(sample.shape) / 2

    # takes a portion of the filter to fit the sample of the face
    filter_overlay = fixed_filter[max(rotated_center[0] - math.floor(sample_half_ht), 0):min(rotated_center[0] + math.ceil(sample_half_ht),fixed_filter.shape[0]), max(rotated_center[1] - math.floor(sample_half_wd), 0):min(rotated_center[1] + math.ceil(sample_half_wd), fixed_filter.shape[1])]

    # assures that the filter portion to fit the face and the face portion being fit are same-shaped
    small_shapes = np.minimum(np.array(sample.shape), np.array(filter_overlay.shape))
    assert (small_shapes[0] == sample.shape[0] == filter_overlay.shape[0] and
            small_shapes[1] == sample.shape[1] == filter_overlay.shape[1])

    # applies the filter to the sample of the face in question, and returns
    cast = np.absolute(np.maximum(sample, filter_overlay))
    face_image[max(face_nose[0] - math.floor(filter_half_ht), 0):min(face_nose[0] + math.ceil(filter_half_ht), face_image.shape[0]),max(face_nose[1] - math.floor(filter_half_wd), 0):min(face_nose[1] + math.ceil(filter_half_wd), face_image.shape[1])] = cast
```

This is a video of us running our program:

https://drive.google.com/a/brown.edu/file/d/1fPn9_pMG97O-7Fgcn8ldlk9SWiFLqInf/view?usp=drivesdk

4.1. Discussion

What about your method raises interesting questions? Are there any trade-offs? What is the right way to think about the changes that you made?

An interesting discussion our method raises is the trade-off between efficiency and aesthetic intrigue. Because the viola jones face detection was a cheap way to isolate the face and crop it from the video still, we used it to make the runtime of the CNN better. However, this interestingly limits our ability to manipulate the face through filters, as we are limited to filter elements that fit within the face box, excluding filter options from extending outside the face box. For example, in our implementation, crowns or tentacles extending past the face are more complicated to implement because we work with just the face cutout from the viola-jones face detector. This was a trade off we found worth it to ensure frame-rate speed and efficiency.

The last main problem that we could not solve entirely was the issue of rotating the filter when the face in the frame rotated. The model wasn't able to detect a face when it was rotated too much, therefore not applying a filter to the face. This is definitely partially caused by the fact that our data set did not include rotated faces. We thought we could solve this issue by adding a rotation data augmenter, but adding this caused our accuracy to drop to 0.3. We weren't quite sure how train a data set using another method other than rotation augmenters. If we were to build upon this project we would focus on being able to map a filter onto a face in any position, as this would also be relevant and necessary for a project that involved the creation of 3D, physics simulated filters.

5. Conclusion

In conclusion, we were able to create custom face filters for computer webcams that were more oriented to morphing faces rather than enhancing features, which was our goal. In the end, we didn't achieve the 3D physics motion for our face filters that we had aimed for, but successfully implemented a 2D, more static version. We also created a project that can be easily added to with more face filters, any square image with black in the background can be used as a face filter! This is a level of customization and creation we could never have achieved using commercial software like Snapchat.

There are definitely places of improvement involving using a better dataset (that includes rotated and turned faces) and more reactive, 3D filters. However, there are also clear paths to take to get there from this initial basis.

References

[Peter Skvarenina]: https://towardsdatascience.com/detecting-facial-features-using-deep-learning-2e23c8660a7a

https://stackoverflow.com/questions/18689823/pandas-dataframe-replace-nan-values-with-average-of-columns http://flothesof.github.io/convnet-face-keypoint-detection.html
https://docs.opencv.org/3.4.1/d7/d8b/tutorial/$_{p}y/_{f}ace/_{d}etection.html$

Appendix

Team contributions

Adam Shelby I built the function to overlay the filter image on the image of the face. This was initially very quick, as it didn't focus on efficiency, resolution, or even working in color. Over time though, as all of those functions became focuses, the project became more involved. I also was the primary worker on incorporating that function into the main loop so that the logic of the outer loop agreed with the logic of the image combination.

Aliya Chambless I mainly worked on loading the data and the CNN. After briefly trying to write my own CNN structure (and failing) I tested out different preexisting facial detection CNN structures. One structure I tried optimized the distance error over all the images, so it outputted the same 15 points for any image. I eventually settled on Peter Skvarenina's structure from Medium.com. I did use Flatten() instead of GlobalAveragePooling2D() based on Florian's suggestion. I also attempted to use the panda fillna() with the mean values of each column to get more data and therefore a better accuracy, but it took too long to run. I also had to save the model using keras.save and pipline using pickle so I could inverse transform the model's output data. The pipeline transforms the keypoints to a 96 by 96 array, so I had to apply additional transformations to resize it to a variable face size.

Ambika Miglani I mainly worked on the webcam file and worked on issues that came up within that class. The first issue we ran into was how to map features onto a an image that include more than just a single face(as the data set we trained on were all frontal views of a face filling up the entire frame). A bit of research was done and we decided on moving forward with the viola-jones facial detector.I was consdering using a CV2 tracking function (i.e. comparing frames to the frame before rather than running a new detection every frame), as I thought that that would be faster and more accurate. However the haarcascades and feature point mapping seemed to be working and we decided to move forward with that.

Victoria Xu I worked on three main things: webcam, filter images, and command line flags. For the webcam, I used cv2 to access and input each frame of the webcam into the viola jones face detector. I then cropped a copy of the video frame to feed into the CNN model that Aliya worked on. Once obtaining the points, I formatted the information into dictionaries and images to feed into the matchup code used for filter mapping that Adam wrote. For creating the filter images, I photoshopped square filters based around coordinate points hard-coded from our mockup face filter, this way they would all map the same way. These square filters are larger than 96x96 to obtain higher resolution. Finally, for the command line flags, I implemented two options of filter image for users to choose from when running our code.