Selfie Filters Addition With Speech Recognition On Facial KeyPoints Using Convolutional Neural Networks

Group #19

Srishti Tomar

Akshit Goyal

Chad Bloxham

Mehul Kothari

Problem Description

Facial keypoints detection is a challenging problem in the field of computer vision since facial features vary greatly from one individual to another. In 2018, Snapchat rolled out a feature where filters animate based on facial movements and sound. We wanted to extend this functionality to the speech where a user can change objects like a hat, sunglasses.

We have detected 15 facial key points in real time and used Google's speech recognition API combined with PyAudio to maintain two threads using concurrency to incorporate both the audio and video simultaneously.

Division of Labor

Akshit Goyal

Task: Preprocessing of images, Initial Model Building, Complex Model Building & Training.

Chad Bloxham

Task: Building & Training the Complex Model, Research Over Fully Convolutional Networks

Mehul Kothari

Task: Testing and Optimization, Multithreading Over Speech And Video

Srishti Tomar

Task: Object Addition/Filter Application, Latency Optimization, Building Complex Model & Training, Speech & Video Synchronization

Requirements

- Python
- Open Cv
- Keras Libraries
- Google Speech Recognition API & PyAudio
- NLTK

Data Description And Pre-processing

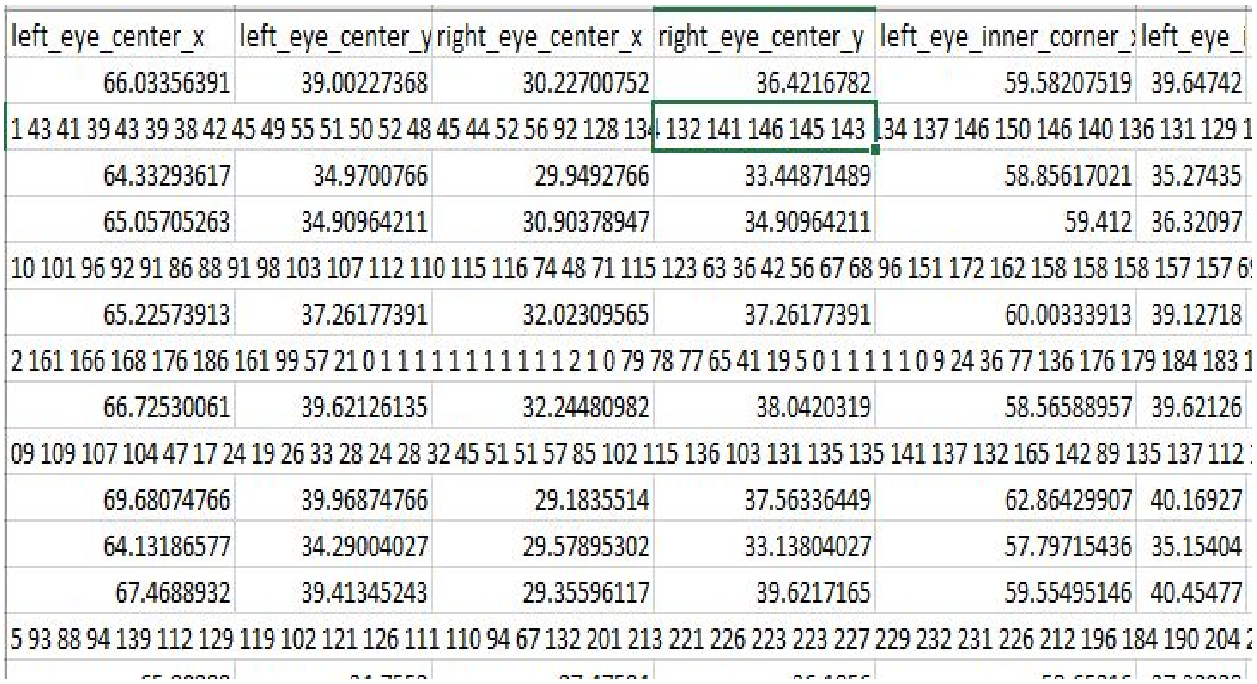
This dataset on Kaggle allows us to train a model to detect the facial key points given a facial image. It was provided by Dr. Yoshua Bengio of the University of Montreal. Each data point in the dataset contains space separated pixel values of the images in sequential order and the last 30 values of the datapoint represent 15 pairs of coordinates of the key points on the face. We have trained a CNN model to solve this classic deep learning problem.

Data Preprocessing part deals with the images and their various features, some of them are mentioned below:

- Uniform aspect ratio: One of the first steps is to ensure that the images have the same size and aspect ratio.
- Image Scaling: Once we've ensured that all images are square (or have some predetermined aspect ratio), it's time to scale each image appropriately.
- Normalizing Image Inputs: Helps in faster convergence.
- Dimensionality Reduction: Changing it into grayscale

Using 96*96 Image Size & Gray Scale

The image size depends upon the primary memory available and the size of the training dataset being used. We used a laptop with 8GB RAM. We tried to resize the image in 200*200 which led to memory errors. We then changed the resolution to 96*96 and the issue was resolved. We used grayscale images instead of colored ones in order to reduce dimensionality and increase processing efficiency. Grayscale images have only one value per pixel compared to colored images that have three values (R, G and B) per pixel.

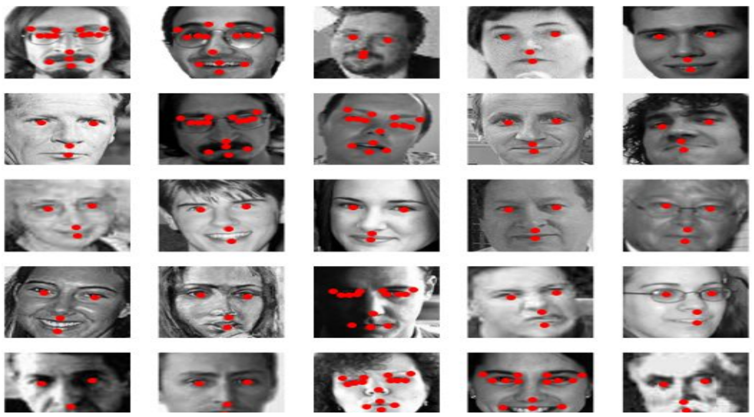


left_eye_center_x	left_eye_center_y	right_eye_center_x	right_eye_center_y	left_eye_inner_corner_x	left_eye_i
66.03356391	39.00227368	30.22700752	36.4216782	59.58207519	39.64742
1 43 41 39 43 39 38 42 45 49 55 51 50 52 48 45 44 52 56 92 128 134 132 141 146 145 143 134 137 146 150 146 140 136 131 129 1					
64.33293617	34.9700766	29.9492766	33.44871489	58.85617021	35.27435
65.05705263	34.90964211	30.90378947	34.90964211	59.412	36.32097
10 101 96 92 91 86 88 91 98 103 107 112 110 115 116 74 48 71 115 123 63 36 42 56 67 68 96 151 172 162 158 158 158 157 157 6					
65.22573913	37.26177391	32.02309565	37.26177391	60.00333913	39.12718
2 161 166 168 176 186 161 99 57 21 0 1 1 1 1 1 1 1 1 1 1 1 2 1 0 79 78 77 65 41 19 5 0 1 1 1 1 1 0 9 24 36 77 136 176 179 184 183 1					
66.72530061	39.62126135	32.24480982	38.0420319	58.56588957	39.62126
09 109 107 104 47 17 24 19 26 33 28 24 28 32 45 51 51 57 85 102 115 136 103 131 135 135 141 137 132 165 142 89 135 137 112					
69.68074766	39.96874766	29.1835514	37.56336449	62.86429907	40.16927
64.13186577	34.29004027	29.57895302	33.13804027	57.79715436	35.15404
67.4688932	39.41345243	29.35596117	39.6217165	59.55495146	40.45477
5 93 88 94 139 112 129 119 102 121 126 111 110 94 67 132 201 213 221 226 223 223 227 229 232 231 226 212 196 184 190 204					
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Initial DataSet File

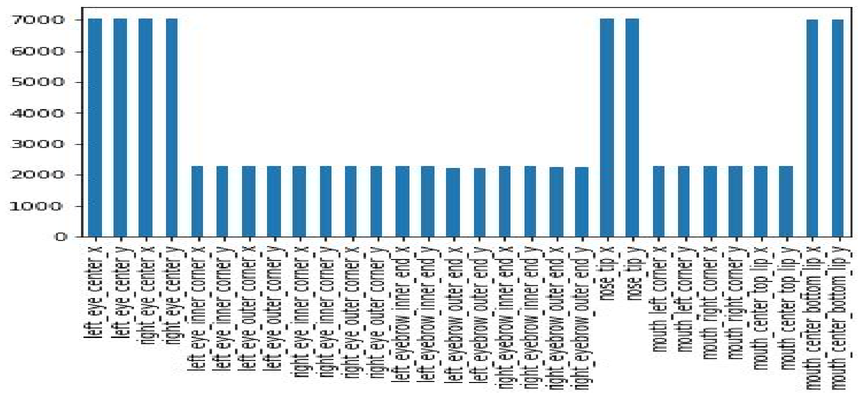
Analyzing Data

The Image column contains the face data in which the 30 first columns represent the key point data (15 x-coordinates 15 y-coordinates). The red dots shown in images are the (x,y) pixel value.



Statistical Analysis

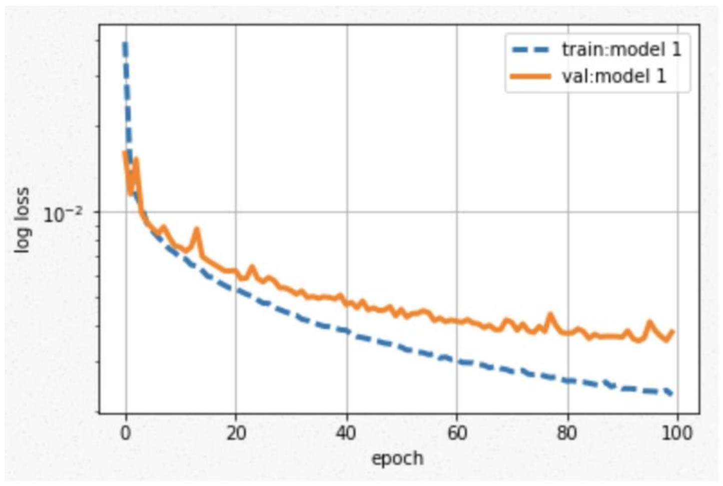
This plot tells us that only 2000 images are ‘high quality’ in this dataset with all key points while the other 5000 are low quality with only 4 key points.



Network models for Facial Key Point Recognition

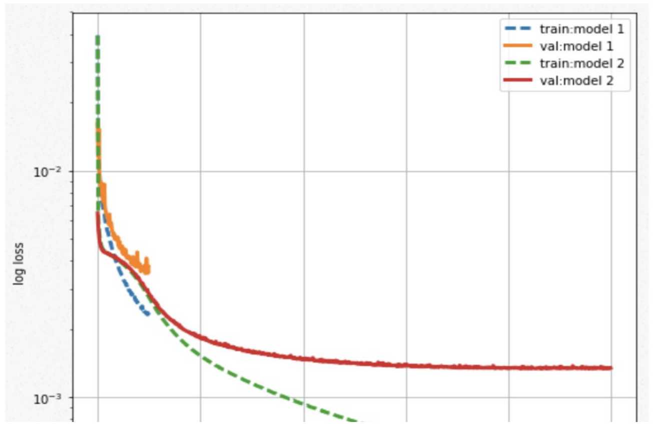
- **Single Layer Feed Forward Network for setting the baseline performance**

  The performance provided was quite decent but we tried improving the performance by increasing the complexity of the model.



- **Convolutional Neural Network**

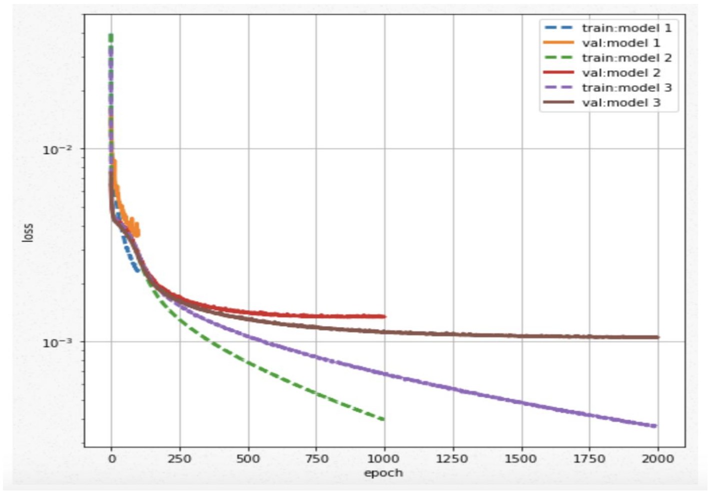
  This plot compares the baseline model and the CNN model. The loss is relatively lower using CNN. The results are better.



Different Techniques used to increase model accuracy

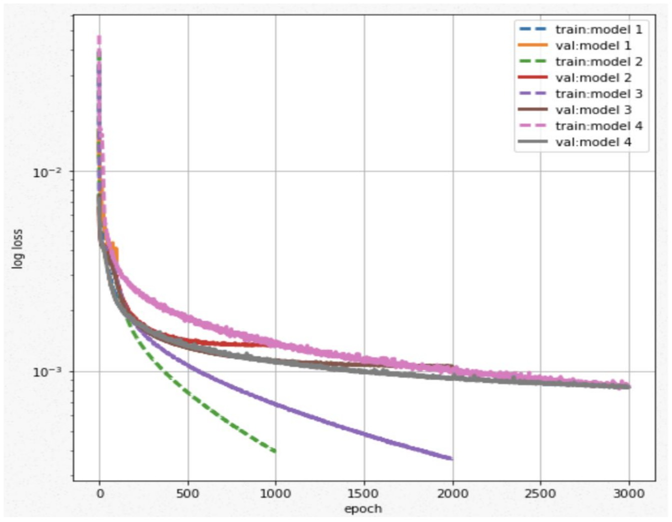
- **Data Augmentation with Flipped Pictures**

  This plot shows model performance after augmenting the data with flipped images.



- **Data Augmentation with shifting pictures**

  We tried adding dropout layer for regularization. The results did not show any considerable improvement since shifting the images serves the same purpose.



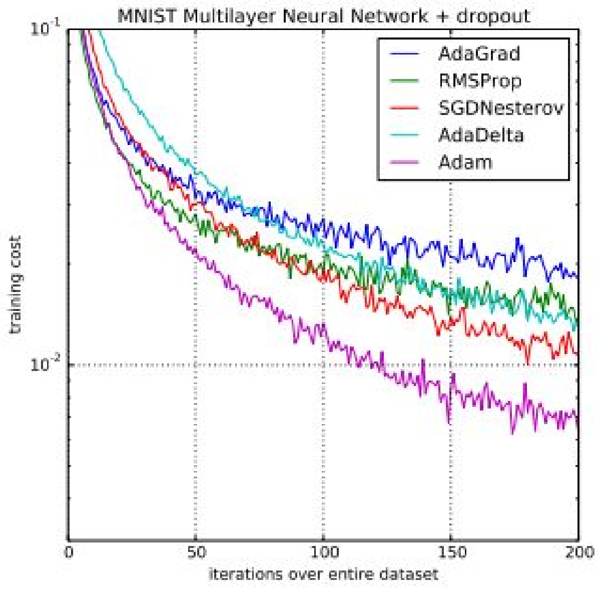
Alternate Method for Facial Key-point Recognition

Before discussing the method used to train our network, we will briefly discuss another network we considered using. In the course of configuring our CNN, we came across a paper describing another one, a “fully convolutional network” (Long, Shelhamer, and Darrell). This is a network used for general object detection and image segmentation which can be applied to facial keypoint detection. The network uses a “skip” architecture in which each set of max pooling and prediction layers are followed by a set of convolutional or deconvolutional layers. Feature extraction takes place in the early layers to detect course, semantic information. This is integrated with the finer details extracted in later layers, culminating in every pixel having its own probability of belonging to a particular class (left corner of the mouth, right eye outer corner, etc.) within the training dataset. These probabilities are represented visually as an outputted “heat map”.

Although this method exhibits high performance, the output of our network was simpler to work with when overlaying our filters (15 positions as opposed to an array of probabilities), and we had already moved forward with this implementation.

Training

To train our network, we use Adam (adaptive moment estimator) optimizer. This is the most efficient and highest performing algorithm provided by the keras.optimizers module. The Adam algorithm is an improvement on the Root Mean Square Propagation algorithm. RMS Propagation is a variant on classical Stochastic Gradient Descent in which each parameter has its own learning rate calculated using an average of the recent gradients. Adam calculates learning rates using the first moment (mean) and second moment (uncentered variance) of the gradients. The following figure (from the paper introducing Adam by Kingma and Ba) shows the superior performance of Adam in comparison to other popular training algorithms:



Per-parameter learning rates are especially important in convolutional neural networks because CNNs have can have vastly different gradients due to weight sharing and the fact that different types of layers are used (convolutional, max pooling). This makes Adam the optimal training algorithm for our network, as well as all other networks designed for computer vision applications. For our error function, we use the mean squared error, which is the standard choice in many optimization problems involving stochastic gradient descent or any of its variants.

We trained our network using 1000 epochs with a batch size of 200. This took approximately 2 hours in total.

Main Program Overview

After importing all required modules, we define a function for the speech recognition portion of the application. It will be called in every iteration of the while true loop (explained further later). In this function, we invoke the built-in microphone to listen for our input. Once it registers something, a Google API is used to convert the detected audio into text. If we say “green”, “black”, “red”, “gray”, “pink”, or “blue”, and the correct text is returned by the Google API, the filter images (hat, glasses, mustache) will change.

After we have defined this function, we load our CNN model (which was built and trained in separate files). We then initialize arrays for all of our filter images as well as the color -> index dictionaries. Finally, we can enter our while true loop in which we capture frames from our webcam and perform the key-point detection/filter overlay.

Adding Facial Filters in real time

In order to incorporate voice commands with filter addition, we implemented multithreading. The first thread captures video and adds filters to them, the second thread deals with voice recognition. A shared variable, speechToAction, is used to switch between these two threads to ensure parallelization. When the user gives a voice command, the **speechrecognition** library converts the input into an audio file(.py format), searches for the corresponding words over the internet and returns the tokens as an output.

Hat Filters

We used a cascade classifier to form a bounding box over faces in the captured frame which is then used by the model to predict the facial key-points. We then computed the proper offsets in terms of height and width and re-sized the images accordingly so that the filter will adjust as and when the faces move. The cascade classifier returns the position of the top left corner(x,y) of the bounding box and the width and height of the bounding box.

Moustache and Sunglass Filters

The width, height, and positional offsets of the moustache and sunglasses are computed using the positions of relevant key-points: Sunglasses key-points are 7, 8, 9, and 10 (eye corners) and Moustache key-points are 10, 11, 12, 13 (lower nose and mouth region). Since the key-point positions are computed every time the frame changes, the height, width, and positional offset change accordingly to fit the face.

Application

We can extend this application as a part of shopping websites to enhance customers' shopping experience online through augmented reality. This will result in greater consumer engagement. Though an exact fit would be ideal, even an approximation of fit with a virtual item will help shoppers pick and choose from the wide variety of products available online.

Future Scope

The following improvements could be made:

- Reducing the lag while changing filters
- Improving model accuracy by changing hyperparameters
- Including more filter options and better image resolution
- Accommodating different face orientations which would require a training dataset with faces at various angles and detection of additional key-points to determine the orientation. It would also require additional filter images for different possible head poses/orientations

Conclusion

Facial key-point detection is a challenging problem due to the fact that individual features and key-points differ greatly from one another. We have combined facial key-point detection and selfie filter addition along with voice-based commands. However, there are a number of improvements, as mentioned above, that can be made to our application such as increasing the speed of voice recognition and reducing lag in the changing of filters.

References

- OpenCV Face Detection Documentation: https://docs.opencv.org/3.4.3/d7/d8b/tutorial_py_face_detection.html
- Facial Keypoints Detection: https://www.kaggle.com/c/facial-keypoints-detection
- Speech Recognition: https://pypi.org/project/SpeechRecognition/