HOURGLASS MODEL

We used hourglass model to train our own eye landmark model on unity eye dataset. The model worked fine on the unity eye dataset or the HD images of eyes, but failed to perform on normal or low quality images of eyes.

The hourglass model we used to train our model required data in form of tfrecords.

STEPS:

1) First, we have to extract eyes from the images and create a single json file which contains the annotations of landmarks of the eyes. JSON file contains: name of the image, path and coordinates of the landmarks. Also, we have to create separate folders for train and val sets.

Script for this is in colab by the name **“eyeExtraction”**.

“EyeExtraction” : First half of the script is setting up the environment. Second half contains the code for extraction eyes from the raw dataset, create separate annotation files for train and val dataset and uploading it to the drive.

Import libraries ➔ Load darkflow model for eyes and face ➔ Read the names of the images in both train and val raw dataset and create new directories to store the eyes images. ➔ Read each train images ➔ extract face and eyes ➔ read the corresponding image annotation file ➔ subtract the top left coordinates of eye boundary box from the landmarks coordinates ➔ save the landmarks coordinates in a list ➔ save eye image ➔ at the end, save annotations list as JSON file ➔ repeat same for validation images ➔ upload all data to the drive for converting them to tfrecords.

2) After extracting eyes and adjusting the landmark’s coordinates, we have to change them to tfrecords. Script for this in colab by the name **“tfrecords”**.

“tfrecords” : Tfrecords are like JSON files that contains data of the images in a NoSQL kind of format. It significantly reduces the size of overall dataset.

read annotation file ➔ create chunks indexes ➔ read the image from the path mentioned in it. ➔ read landmarks coordinates ➔ convert image into byte stream ➔ create feature dictionary (a format requires to create tfrecords) using image and landmark coordinates ➔ after each chunk, convert them into tfrecords. ➔ repeat same for validation ➔ upload tfrecords to drive

3) After creating tfrecords, we have to train an Hourglass Network to create a model. The setup for this is in training-compute.

COMPUTE – training :

There are various folders inside /home/ubuntu/training.

1) HourglassTest:

This directory contains the resources for preprocessing, train and test hourglass model.

Hourglass model: read tfrecords ➔ convert image from byte streams to normal image ➔ create 16 different heatmaps, each for each landmark co-ordinates ➔ train model

a) train.py: to train the hourglass model.

b) demo_hourglass_pose.ipynb: to test the hourglass model.

c) preprocess.py: contains the code for preprocessing data for hourglass model.

d) test_preproccess.ipynb: contains the code to check the heatmaps generation.

e) test_tfrecords.ipynb: contains the code to check tfrecords.

f) models/: it's a directory that contains model checkpoints.

g) hourglass104.py: contains the architecture of the hourglass model. The architecture has been altered to take image of size (128,128,3) and generate heatmap of size (32,32).

2) HourglassTest2:

The structure is similar to above.

The difference between above 2 is the architecture of the HourglassTest2 takes image of size (256,256,3) and generate heatmap of size (64,64).

3) GazeML:

The main purpose of this was to train new GazeML model but it doesn't worked properly, so it is useless. The whole architecture of GazeML is explained later in this document.

4) analysis.ipyb:

This contains the code to check whether the headpose and gaze vector generated by deep-headpose and gazeML is consistent with the available datasets or not.

GAZEML

1) Original:

This one only works on video.

Parameters required:

--from_video: the path of the video file

--record_video: the name of the output file

--headless: If you don’t want to see output real time, pass this (no value)

--fps: Fps of the output video you want

Main Files:

elg_demo.py

datasources/video.py

datasources/frame.py

Flow:

a) video path taken from the user is pass to the video.py by creating an object of Video class.

b) Then an object of ELG class is created and the tensorflow session and the Video class object along with few other parameters are passed to the class.

c) ELG class initialized an queue and call functions of Video class to generate frames.

d) Video class reads the video, extract all the frames and save it to the queue and returns to the ELG class.

e) ELG class takes all the frames and create an object of Frame class and pass the queue of the frames.

f) Frame class read frames one by one.

g) For each frame, Frame class first convert it into BGR format. Then, Frame class find face from that frame and save the bounding boxes. Then for each frame, its BGR format and bounding box details is saved as dictionary format and is passed into the queue one by one.

h) ELG models then finds the eye landmarks for each frame, calculate gaze angles and then save it with other details as dictionary format and passed them into the queue.

i) Control is passed to the elg_demo.py.

j) elg_demo.py creates a cv2 VideoWriter object to save the output

k) elg_demo.py takes frames from the queue one by one, read the details, draw all the bounding boxes and other details like eye landmarks and all, on the frames and save it in the VideoWriter object.

l) End

2) Edited Version (GazeMLImageSupport):

I altered some of the code to work it on the bunch of images.

Parameters required:

--from_video: the path of the video file/ directory where images are stored

--record_video: the name of the output file

--is_image: If the path pass in the from_video is of the directory of images, then pass this (no value)

--fps: Fps of the output video you want

Flow: (For video, it is same as above)

a) image path and is_image=True taken from the user is pass to the video.py by creating an object of Video class.

b) Then an object of ELG class is created and the tensorflow session and the Video class object along with few other parameters are passed to the class.

c) ELG class initialized an queue and call functions of Video class to generate frames.

d) Video class reads the images from the directory and saves it along with name of each image to the queue and returns to the ELG class.

e) ELG class takes all the frames and create an object of Frame class and pass the queue of the frames.

f) Frame class read frames one by one.

g) For each frame, Frame class first convert it into BGR format. Then, Frame class find face from that frame and save the bounding boxes. Then for each frame, its BGR format, it's name and bounding box details is saved as dictionary format and is passed into the queue one by one.

h) ELG models then finds the eye landmarks for each frame, calculate gaze angles and then save it with other details as dictionary format and passed them into the queue.

i) Control is passed to the elg_demo.py.

j) elg_demo.py creates a cv2 VideoWriter object to save the output and create an empty list to save the details.

k) elg_demo.py takes frames from the queue one by one, read the details, draw all the bounding boxes and other details like eye landmarks and all, on the frames and save it in the VideoWriter object.

l) elg_demo.py save the image name, eye_label(which eye it is), left eye gaze vector as one row and image name, eye_label(which eye it is), right eye gaze vector as another row in the list.

m) elg_demo.py then convert the data list into numpy and then pandas dataframe and save the csv with same name as –record_video.

Visualization

main ones:

gazePlotterNew.py: This contains the code to create graph on poll image. This can create 4 types of Graph:

a) Fixation   b) Heatmap   c) Fixation-Scanpath   d) Heatmap-Scanpath

This file only deals with creation of graph, not data handling and manipulation.

Flow of any graph contains first: draw_display ➔ change data to dictionary format ➔ draw graphs.

Code is easily understandable if you know Mathplotlib.

imageVisualizationGaze.py:

This is the main file that was deployed on frontend. This was used to create all the graphs for poll Image. Code for filter was remaining in the multi-users case.

Parameters required:

a) -client: cient_id   b) -content: content_id   c) -age: age (“All” if age filter not applied by user)   d)-gender: gender (“All” if age filter not applied by user)   e) -user: user_id
f) -image: path of poll image.

Explanations given in the code.

Test ones:

Folders: 200CSV and 1200CSV contains 20 csv files containing 200 and 1200 rows respectively.

Product: contains the data used in creating graphs on the Pepsi Advertisement.

pygazeanalyser: contains modules that is used to create graph on images and videos.

Explanation in the code.

Deployed Version V1

V1 uses deep-head-pose and GazeMLImageSupport.

FILES:

a) takeFrames.py: This file read frames from video and save it as images in the directory.

b) calibration.py: This file contains the code for calibration.

Flow:

read calibration gaze data ➔ read calibration headpose data ➔ reading calibration images ➔for each calibration image, separating dependent variables (gaze and headpose) and independent variable (normalized gaze location on screen) ➔ training models ➔ reading reaction video gaze & headpose data ➔ reading reaction video's frames images ➔ finding gaze location for each frame using models ➔storing data in the list frame wise ➔ create clusters for fixation and change data accordingly ➔ save data

Code is commented.

c) finalScipt.py: This file contains the code for automating the backend process.

Flow:

1) Infinite loop

2) Connect to sql database.

3) Extract the data for top 5 unanalyzed reaction data.

4) If no unanalyzed reaction data, then close the database connection, and sleep for a minute and return to step 2).

5) Loop through each unanalyzed data:

6) Copy the data from s3 bucket to the compute under GazeData/ClientId/PollId/UserId.

7) Check for reaction video.

8) If no reaction video: if this was the last unanalyzed data, then sleep for 1 minute and go to the last step else go to the step 5).

9) Extract frames from video and save it as images under directory VideoName.

10) Find fps.

11) If fps less than 15, then video is corrupted, so update sql table and go to step 5).

12) Run the headpose script for the reaction video and save the data in the folder VideoName_GazeOutput/Headpose.

13) Run the gazeML script for the reaction video and save the data in the folder VideoName_GazeOutput/Gaze.

14) Create the calibration directory under VideoName_GazeOutput/Calibration.

15) Copy the calibration images to the last step directory.

16) Run the headpose script for the calibration images and save the data in the folder VideoName_GazeOutput/CalibrationGazeOutput/Headpose.

17) Run the gazeML script for the calibration images and save the data in the folder VideoName_GazeOutput/CalibrationGazeOutput/gaze.

18) Run the calibration script.

19) Transfer the output from compute to s3 bucket.

20) Update the SQL table.

21) Commit the changes.

22) Close the connection. Move to step 2).