Number Detection & Hand Tracking using OpenCV and TensorFlow

CSI 4133 (Final Submission)
November 29th, 2021

Aiden Stevenson Bradwell

300064655, abrad060@uottawa.ca

University of Ottawa, Ottawa, Ontario

Part A: Number Detection using OpenCV & TensorFlow

Methodology

The solution for the number-detection component of the project is an interworking of Python 3 using OpenCv, and TensorFlow. The initial challange was to find the numbers within the image. To do so, a combination of image-preprocessing, canny edge, and contour detection was used to find all strong edges in the image. These edges were then filtered in a helper method which found the 4 strongest contours (most likely to be the numbers), and organized from left to right (placement in image).

The program then proceeds to cut out the bounding boxes into their own frames, and pass them through a TensorFlow network trained to identify canny-edged numbers.

Following an online tutorial[1] to train a classification network, a 12 layer convolutional network was implemented to identify the found numbers. Its overall success was 97.4% (SEE BELOW) in identifying all 4 numbers correctly.

Implementation (Method of Detection)

- Request image name from user (including extension)
- Gaussian blur image with a kernel of (3, 3)
- Convert image to grayscale
- Run Canny Edge Detection on image with low thresh of 177 and high thresh of 204
- Detect contours on canny image cv2.RETR_EXTERNAL and cv2.CHAIN_APPROX_NONE settings
- Filter four largest contours by area
- Sort contours from left to right (position on image)
- For each contour:
    - Separate the contour's bounding box from the original image
    - Gaussian blur with kernel of (5, 5)
    - Convert to grayscale
    - Run canny edge detection with low thresh of 177 and high thresh of 204
    - Convert image to RGB
    - Resize image to 70w x 84h (same size the network was trained with)
    - Run image prediction with trained network
    - Return strongest prediction of number value
- Output 4 combined expected values.

Network Information & Training Data

Number of Epochs: 20
Batch Size: 32

Number of Images Per Training Category

Zeros – 1368
Ones – 588
Twos -- 144
Threes -- 109
Fours -- 104
Fives -- 184
Sixes -- 119
Sevens -- 324
Eights -- 558
Nines -- 493

Network Structure

Pre-processing

0: Random Flip

1: Random Rotation

2: Random Zoom

3: Rescaling

Main Network

4: 2D Convolutional (16 filters, kernel size 3, Relu activation)

5: 2D Max Pooling

6: 2D Convolutional (32 filters, kernel size 3, Relu activation)

7: 2D Max Pooling

8: 2D Convolutional (64 filters, kernel size 3, Relu activation)

9: 2D Max Pooling

10: Dropout (Rate of 0.2)

11: Dense Layer (128 units, Relu activation)

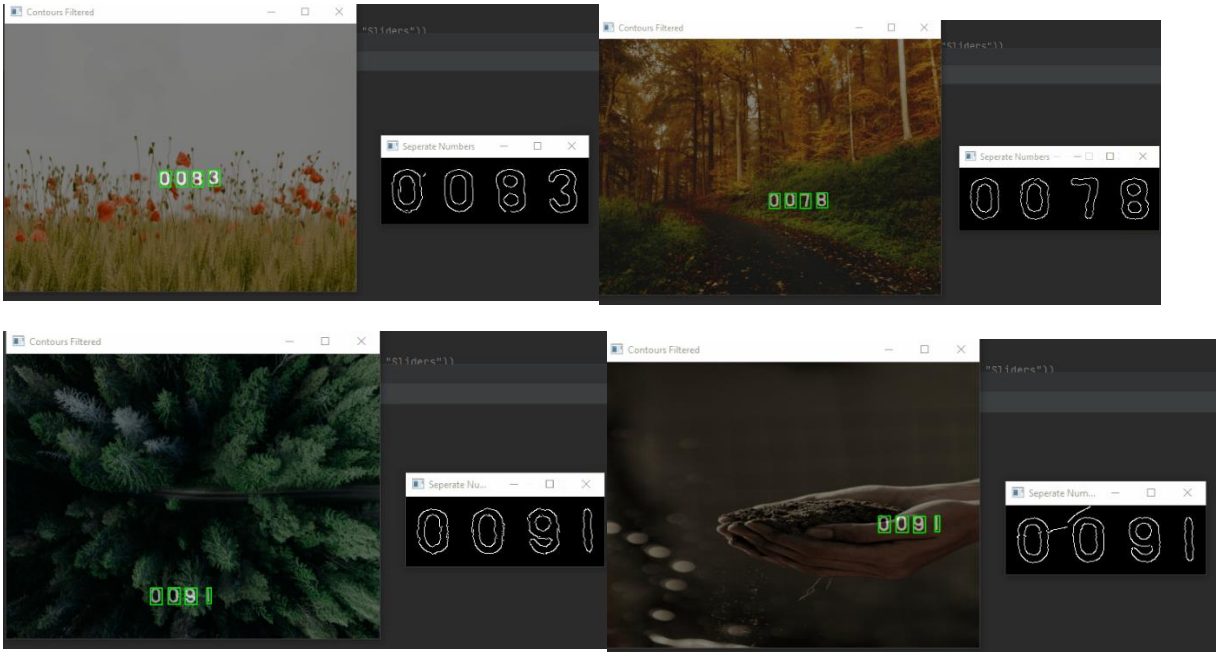
12: Dense Layer (10 units, Relu activation)

Limitations

In its current state, this program does have some limitations. Human analysis of the network's success has determined that it correctly identifies all 4 numbers in non-corrupted images 98.187% of the time (1294/1324 images). It was not trained to recognize corrupt data, and incorrectly identifies any corrupt numbers (where corrupt images are defined as any image which does not include 4 clear numbers).

Future work can be done to introduce a "Corrupt" data class which the network can be trained to signal and ignore. For now, there exists a list which warns the user if the attempt to call the network on one of the 24 incorrectly identified images.

Screenshots



Conclusions

In summary, while unable to successfully identify all 4 numbers rom all 1324 images provided in the assignment, my solution to this problem can successfully do so for 98.2% of non-corrupted samples. Using a self-trained neural network to identify the **value** numbers, preprocessing and contours are used to detect the **location** of the numbers within the frame.

Part B: Hand Detection & Motion Tracking

Methodology

For each frame of a given video, the hands need to be detected. To do so, a pretrained algorithm provided in an online tutorial [2] is used to find the skeleton of the hand. The tutorial was originally designed to predict hand-gestures, however for the purposes of this project only the hand detection was needed.

From the skeleton, the palm was calculated, and added to a stored list of points (with a capped size of 75). Depending on how many hands are detected in the frame, the point will either be added to the HAND1 or HAND2 sub list of this storage list. During the visualization of these lists (the blue line present in the display), a prevention method is in place to stop confusion between two separate hands being tracked. If for some reason, the program begins identifying alternate hands, instead of a large jump in the line, the program simply cuts the first line and begins a new line across the screen. This allows for the two hands to be tracked separately, without a jumping between them (IE differentiate between the two hands).

This is flawed however as if the user crosses their hands behind one-another too closely, there may be a joining of the two lines, as they have crossed the 75-pixel threshold required for them to be considered members of the same list.

In summary, the program aims to track motion through the visualization of previous locations and current locations.

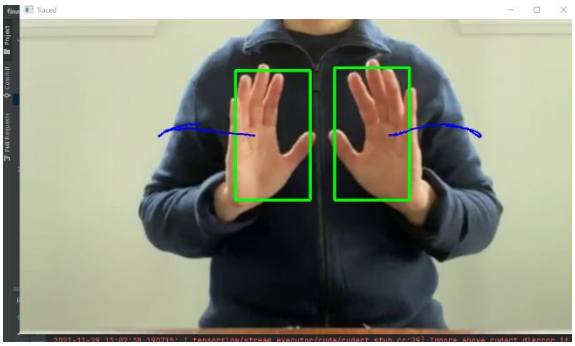
Implementation (Method of Detection)

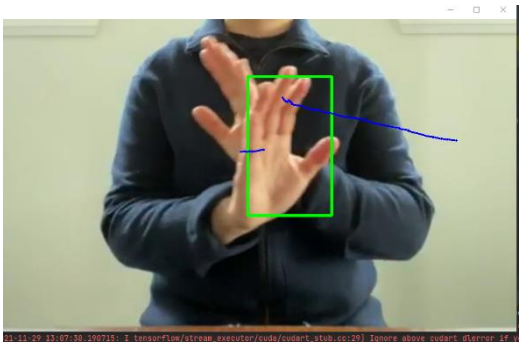
- Create video input & output handlers
- Initialize a storage list, containing a sub list for both HAND1 and HAND2
- For each frame in the video:
    - Convert frame to RGB
    - Use pretrained hand-detection **mp_hand_gesture.pb** to detect hands in the frame
    - Detect palm of hand from the returned objects
    - Add the coordinates of the palm to the appropriate list (where the first object detected in HAND1 and the second is HAND2)
    - Draw curve of movement for both hand sub-lists
        - Two stored points are considered part of the same motion-curve if they are at most 75px away from each other
        - Max of 75 points stored for either hand (creates fading line)
    - Display and save processed frame

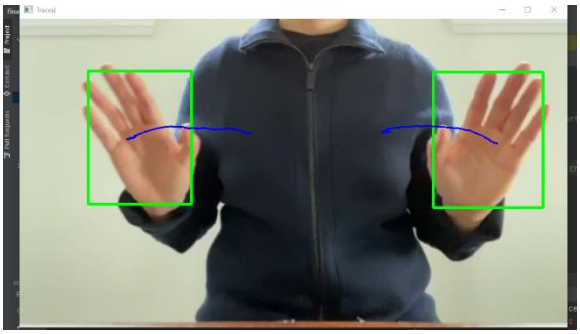
External Code Usage

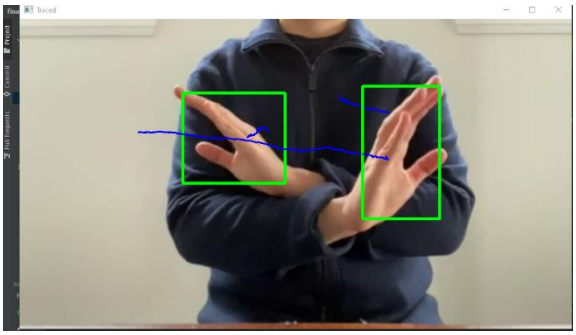
The externally pretrained neural network is taken from a tutorial on hand gesture recognition. The author of this tutorial did not include their name, or git information. The tutorial can be found here (https://techvidvan.com/tutorials/hand-gesture-recognition-tensorflow-opencv/), and parts of it are used in my solution. Specifically, the network and the use of the network to find the skeleton through google-media pipe are used.

Screenshots









Conclusions

The solution to part B is quite simple, and uses the training work of another programmer. The main work done by myself is the tracing and detection of motion within the video. My original program misunderstood the assignment and was a webcam based-live tracker, which was then simplified down to the simpler video-reader solution.

References

1. https://www.tensorflow.org/tutorials/images/classification

TensorFlow. (2021, November 11). *Image classification : Tensorflow Core*. TensorFlow. Retrieved November 29, 2021, from https://www.tensorflow.org/tutorials/images/classification.

2. https://techvidvan.com/tutorials/hand-gesture-recognition-tensorflow-opencv/

Real-time hand gesture recognition using tensorflow & opencv. TechVidvan. (2021, July 21). Retrieved November 29, 2021, from https://techvidvan.com/tutorials/hand-gesture-recognition-tensorflow-opencv/.