Sean Yang

6/3/2019

Pd 4

Abstract

My project uses facial gestures to flip pages on e-books. A facial gesture is a face with the mouth and the cheeks moved in a specific way. My program takes in an image with a facial gesture, classifies the gesture, and simulates a keyboard press to flip a page on an e-book. Convolutional neural networks (CNN) are currently the standard way to perform classifications. CNNs have issues of being slow on CPU, requires lots of training data, and not being as flexible as some of the other machine learning algorithms. My program addresses the issue using the k-nearest neighbors algorithm (k-NN) and feature extraction with facial key-points to perform the classification instead. It produces a program that is more flexible and requires less training data. The programs generally perform a page flip under 2 seconds. When I tested my program, I rarely experienced any false flips.

Introduction

I was sitting on my bed and reading on my kindle during summer break. I found it inconvenient having to hold the Kindle and flip pages. So I stood my kindle next to an object. However, I still had to flip pages by touching the Kindle quite often. I can imagine this being an even bigger issue for people with disabilities afflicting their hands. I thought it would be nice to have a way to read without hands, so I came up with the idea of using facial gestures to flip e-book pages. This project can benefit lots of people with disabilities.

CNNs are a common way to solve classification problems. However, a CNN is not as flexible as k-NN. If I want software that packs a pre-trained model, I could produce software that functions like mine with a CNN. There are models that can detect emotions. Users just have to show the desired emotion to flip the page. This can be limiting for users in some cases. Users don’t really have a choice. Also, users might not want to show a sad face every they want to flip a page. My program allows the users to record the specific gestures they want to use.

Methods

My program is written in Python. It uses OpenCV for loading in the camera frame, dlib for face detection and facial key point detection, scikit learn for the k-NN algorithm, and tkinter for making the GUI. The hardware is fairly simple. All you need is a computer and a camera. In my case, I have a laptop with a camera and that is all I need. The model for facial keypoint detection is trained on the iBUG 300-W dataset. My program also requires the user to record images for the gestures the k-NN classifier is trained on.

Here is a basic description of how my software works. When the software initializes, it loads in the labeled training set and finds the feature vector for the k-NN classifier. To find the feature vectors, it detects the face in the camera. Then it takes the detected face and performs a 68-point facial landmark detection. Then, it generates a feature vector from the key points by comparing them with the center keypoint on the nose.

The program has a set of recorded and labeled faces from which to create vectors for comparison. Every frame of the camera image input will generate a feature vector to be compared. In the end, it uses k-NN to compare the gestures to classify them. After that, it performs a keyboard simulation to flip the page. In the next sections, I will go over each of the steps more specifically.

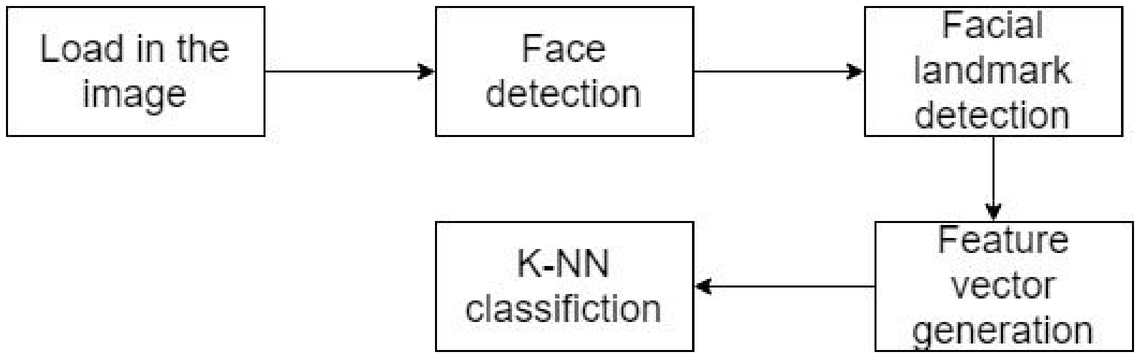

Image 1. Flow chart of my process

Loading in the image

My program uses OpenCV to load in a live camera feed. When I first tried loading in a video on the linux desktop in the computer systems research lab, it kept breaking. It turns out, loading in the frames was fine, but displaying it in a window broke the program. It was some sort of QT issue I couldn't fix it. However, running the exact same code worked fine on my laptop. I stopped using the school computer from that point on.

Code:

```
cap = cv2.VideoCapture(0)
```

This function creates a variable that is used to load the images from the camera frame later.

```
while(True):
     ret, img = cap.read()
```

This snippet continuously reads in the current frame of image from the camera. The variable "img" contains the image matrix.

Face detection

My program relies on classifying the gestures on human faces first. Face detection is a good starting point. Since I'm only interested in facial gestures, the process preserves the most important information in the image and discards the rest of the useless information, making the program faster and more accurate.

Image 2. Result of a face detection

Initially, I used opencv's implementation of face detection because I took computer vision in my junior year and we used the library in that class. It uses a Haar cascades. The result was a program running at over 10 frames a second. However, it has false positives sometimes. For example, when I was sitting in the Sys lab, there was a wall outlet on the other side of the room. Opencv picked up the plug as a human face. This is because the plug had some features resembling that of of a human face. The top two holes kind of looks like eyes and the bottom one looks like the mouth.

Image 3. An image of a wall outlet

Later, I found a dlib implementation. It's based on the Histogram of Oriented Gradients (HOG) + Support Vector Machines (SVM), a newer approach than Haar cascades. I will go over how it works later. This has fewer false positives and is as fast as the OpenCV implementation [2].

Both face detection implementations slide boxes over an image to recognize features in the box. However, the HOG + SVM approach produces a histogram of the gradient for small patches of square cells in the image. After that, it slides the box over and uses a support vector machine to determine if the face is present in the box. In the end, I chose the dlib implementation because it has better accuracy and the dlib facial landmark detection is based on face images from the HOG + SVM method [1].

Code:

```
detector = dlib.get_frontal_face_detector()
dets = detector(img, 1)
```

This snippet creates a frontal face detector "detector." "dets" contains information on the detected face.

Facial keypoints detection

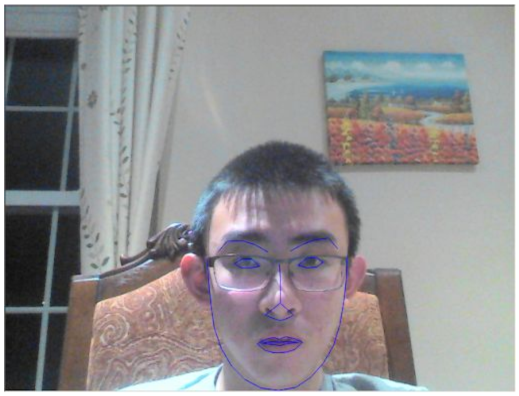
Image 4. An image with the keypoints connected and overlaid

Facial keypoint detection is an important part of the feature extraction process. It uses a cascade of regression trees [4]. The essential algorithm used here is gradient boosting. The algorithm's main idea is to repeatedly use new regression trees to learn the errors based on the result of the previous regression. The regression trees predict the shape of the face. This is a method from dlib and is trained on iBUG - 300W dataset. It's fast and fairly accurate for frontal faces. However, it has some issues with non-frontal faces; the overall orientation is fine but the borders aren't too clear. This is probably caused by not having many tilted faces in the training set. This method produces 68 keypoints with their x and y coordinates [3]. The lines on the face are produced by connecting those dots. In the next step, I will use the coordinates to produce a feature vector.

Code:

```
predictor = dlib.shape_predictor(predictor_path)
for k, d in enumerate(dets):
     shape = predictor(img, d)
```

This snippet first creates a keypoint predictor. "Predictor_path" is the path of the file that contains the predictor model. "Dets" is the variable from the face detection step. The for loop loops through all the faces from face detection. Shape is a list that contains the location of the keypoints.

Feature vectors generation

The feature vectors generation is simple. I don't want to directly compare the coordinates of the face. I want to know the relationship among the points on the face instead. Since the nose is at the center of the face, I choose it as a reference point. The program takes the 68 keypoints and compares them to the point that is the center of the nose. It generates a 68 x 2 vector.

Code:

```
for x in range(68):
     dis = distance2(rotated.part(x),comp)
     list_feature.append(dis[0]*15)
     list_feature.append(dis[1]*15)
```

This code shows the process of the feature vector generation. It first compares the x and the y offset from point on the nose. Then, it appends the offsets into a list to perform k-NN.

Rotation

The images below show examples of rotation. I added this part after I finished the gesture classification function. I came back to this because of an observation: while working on my project, I noticed that tilting the face would produce a bad result since after tilting the image, the feature vector gets changed dramatically for the same gesture. The intention of using rotation here is to help the program generalize better.

Initially, when I came up with the method, I rotated the keypoints based on the angle of the line of the nose since the nose is on the central axis of the face. The result didn't really get much better. Then I realized that tilting based on eyes are better for the adjustment process. The leftmost picture below is the original image; the middle one is rotated based on the nose; the right one is rotated based on the eyes. The rotation angle is calculated based on the key points. I pick two points that are supposed to be vertical on the nose. The program finds the angular offset using law of cosines with the line between the two points and a vertical line from the lower point [5]. The process of the rotation is like applying a transformation matrix around the center of the face to correct the tilt. Looking at the bottom right picture, the key points there look most straight. I ended up using that. The process is similar to the way I correct the angle based on the nose except it's based on two points next to the eyes and the horizontal axis.

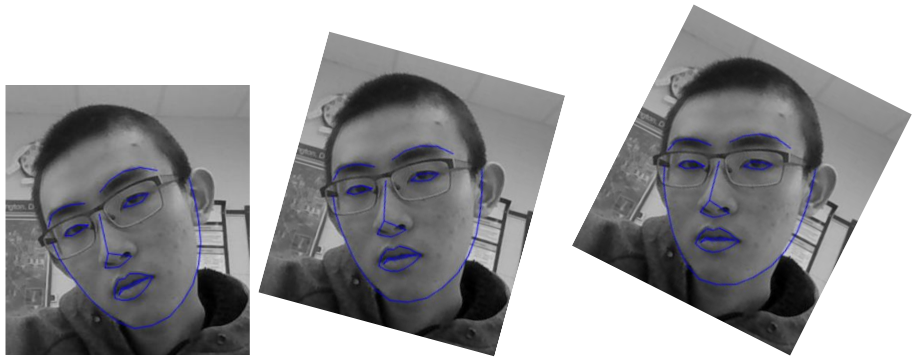

Image 5. Images with varying degrees of tilt adjustment

Code:

```
def get_angle(left_pt,right_pt):
     refpt=dlib.point(right_pt.x -1 ,right_pt.y)
     s1=1
     s2=distance(left_pt,right_pt)
     opposite=distance(left_pt,refpt)
     angle=math.acos((s1**2+s2**2-opposite**2)/(2*s1*s2))
     if right_pt.y-left_pt.y>0:
          return -angle
     else:
          return angle
```

The function creates a reference point on the left of the right point. The reference point is guaranteed to be level with the right point. Then it finds the angular offset using the law of cosines.

Classification

The k-nearest neighbors algorithm (k-NN) is responsible for the classification process [6]. It compares feature vectors using an Euclidean distance metric from the labeled training set and the image from the camera input to find the closest matches in the training set. Intuitively, the “closest” faces are likely to be the right answer. I use scikit learn for this step. It uses a data structure called kd-tree which makes the process faster than the most basic way of using for loop to loop through everything. My program compares the 4 closest points based on experimentation.

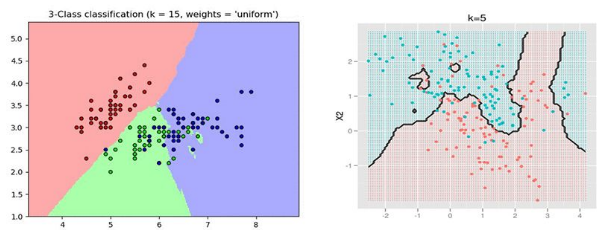

Image 6. Visualization for k-NN

Code:

```
neigh = KNeighborsClassifier(n_neighbors=4)
neigh.fit(xlist, ylist)
```

This snippet first establishes a k-NN classifier. “Xlist” contains the feature vectors and y list contains the name of the gesture.

```
Cur_gestire =
neigh.predict(np.asarray(list_feature).reshape(1,-1))[0]
```

This line of code predicts the current gesture based on the feature vectors generated from the image.

Key presses and tradeoffs

Some might think that it’s better to simulate a keyboard press every time a non-neutral gesture is detected. However, this is not ideal for a model with some errors. It would be really annoying if the software start to randomly flip pages. So I rather play it safe here and wait for consecutive frames of the same non-neutral gesture to perform a keypress. I trade some speed for accuracy during keyboard simulation here. The program currently requires 3 consecutive frames to simulate a keyboard press. I do this because the speed is still good after the modification. It still takes less than 2 seconds to run most of the time. Trading some speed for accuracy is well worth it.

Code:

```
keyboard.press('up')

keyboard.press('down')
```

Those methods are used to simulate keyboard presses.

Graphical user interface (GUI)

The software GUI is built in tkinter and is fairly simple [7]. For the classification script, I don't even have a special GUI for it. There is only an image window that displays the face with the keypoints overlaid. This is useful for debugging purposes. Users can easily hide the window. I chose to not make a complex GUI for this section of the script because it's unnecessary; when people are reading, they don't want a random GUI blocking their sight. It would be distracting. They just want something that works quietly in the background.

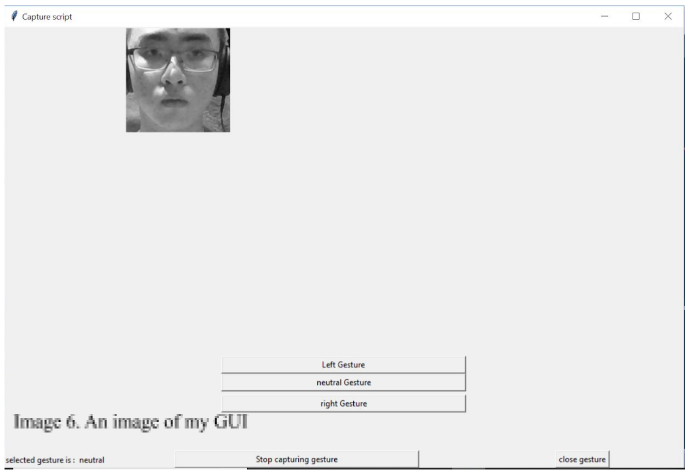

Image 6. An image of my GUI

Image 7. An image of my GUI

The capturing script GUI is also designed with simplicity in mind. Instead of giving a text field input for the name of the gestures, the program uses the button to ask for the gesture being recorded. Realistically, my program only needs 3 gestures: flip left, flip right, and do nothing if it's neutral. A button is the more intuitive approach here. There is also a video showing the detected face to show that the program is running during gesture capturing.

Principal component analysis (PCA)

I attempted to use principal component analysis after finding the keypoints. The intention here is to perform dimensional reduction to only extract the key information and make the program faster. However, it didn't work too well. Since the feature vectors of the face don't look linear, PCA might not do the job very well. Also all the gestures have too similar of a principal component because they are all pretty much on the same plane and the points don't vary too much when you make a facial gesture overall. I had terrible results with it. It's probably close to or worse than just guessing randomly. It turns out k-NN was relatively fast and PCA isn't necessary.

Why not the convolutional neural net (CNN) approach

Convolutional neural networks seem to be the standard approach for classification tasks nowadays. However, it doesn't suit my project well. I want the flexibility of user inputted gestures; it requires users to train offline if I want to use a CNN. First off, it requires lots of data to train well. Collecting data can be tiring. Secondly, people can accidentally create imbalanced classes and reduce the accuracy of the model by accidentally taking too long for a certain gesture. Thirdly, training without a GPU is a huge pain and can take a long time for bigger data sets. Training on bigger data sets with a CPU can take days. Finally, a hyperparameter such as learning rate can be situational. For example, the number of epochs taken and the learning rate can depend on the size of the dataset. The end users might not be as familiar with the parameters and could produce bad results.

I fine tuned a resnet-34 myself on 3 gestures [8]. The training set had over 200 images total. It's good at telling whether a gesture is neutral or not, but it has trouble telling the difference between left and right. The result is not satisfactory.

Potential improvements

Typically my program takes a couple of seconds to read hundreds of images. I could shorten the time to load in the images. Currently, my programs read in raw images for k-NN training data. Then, it performs a key points detection to extract the features. This provides some flexibility for tuning parameters such as rotating the key points during the reading process. However, if I directly save the feature vectors, it would be faster during loading.

My program gets slow if you have a high-quality camera. Photos taken from phones take a long time to process. Higher resolution images have more data than necessary and take longer to process for each frame. I can potentially resize the image to achieve a good result. When I was working on my program, I used the same camera, so I never had to think about the issue. However, my laptop has a decent camera so I would assume laptops should be good enough.

Conclusion

I successfully finished what I planned at the beginning of the year. I have fully working software that flips pages using the facial gestures. My program is fairly flexible. The gestures recorded on my own face have a decent chance of working with other people based on my testing. Overall, I’m happy about my result. I hope this will be able to provide a free and open source solution for people with disabilities.

Acknowledgement

I would like to thank Dr. Gabor for his continuous feedback on my project and the presentation. I would like to thank Dr. Zacharias for discussing the computer vision related topics in my project. I also would like to thank my classmates for their constant support. My project would not have gone go as smoothly without their continued support.

References

1. https://medium.com/@mithi/vehicles-tracking-with-hog-and-linear-svm-c9f27eaf521a This is an article explaining HOG + SVM
2. http://blog.dlib.net/2014/02/dlib-186-released-make-your-own-object.html Dlib vs opencv implementation of face detection comparison
3. http://dlib.net/face_landmark_detection.py.html Example code from dlib for landmark detection
4. http://www.csc.kth.se/~vahidk/face_ert.html The original paper for face alignment
5. https://stackoverflow.com/questions/14410484/solve-triangle-using-cosine-in-python Use the law of cosines in Python
6. https://scikit-learn.org/stable/modules/neighbors.html k-NN’s official docs
7. https://docs.python.org/3/library/tk.html Tkinter’s official docs
8. https://www.fast.ai/ An excellent resource for CNN and other deep learning applications