Efficient Facial Landmark Tracking iOS App

Siliang Lu
School of Architecture,
Carnegie Mellon University
Pittsburgh, PA
siliang1@andrew.cmu.edu

Ningyan Zhu
Robotic Institutes,
Carnegie Mellon University
Pittsburgh, PA
nzhu@andrew.cmu.edu

Abstract

Our project aims to realize real-time facial landmark tracking on iOS mobile devices. The app utilizes the open source library called Dlib. With AVMetadatFaceObject and implementation of effiicent facial landmark tracking algorithm called cascades of regressors, the frame rate can reach around 12-13 fps in iPad. With instrumental tool for profiling, the project has utilized several methods to speed up and optimize the performances of the tracking.

1. Introduction

Facial landmark is an application of facial analysis in computer vision. It involves locating the face in an image and marking the accurate position of different facial features. It is also referred to as "face alignment " or "facial keypoint detection".

Our project aims to develop an efficient real-time facial landmark tracking APP for iOS mobile devices. We used image processing toolbox in Dlib, which is a C++ open source library. Compared to OpenCV, Dlib has a class, which can directly conducting real-time facial landmark tracking. In addition, with well commented files, Dlib is easy to be customized and compiled. However, it could be hard to link the precompiled Dlib into Xcode. In addition, it could also be challenging to speed up the app significantly.

2. Background

2.1. Face detection and tracking

Face detection problem is a kind of object detection problem. For real-time object detection, the Viola-Jones object detection frame work proposed by Paul Viola and Michael Jones in 2001 is the first frame work to provide competitive object rates[1][2]. This algorithm can provide efficient and scale invariant face detector but it also has limitations in that it can only deal with frontal faces and is sensitive to lighting conditions.

KLT algorithm is a similar object detection and tracking algorithm but can better deal with tilted and rotated faces. It tracks the face based on feature points instead of the rectangle area so it can deal with tilting and rotation of the object.

2.2. Related landmark alignment algorithms

Various landmark alignment methods has been proposed but not all of them are suitable for mobile devices. CLM (Constrained Local Model) method exhibits the state of the art performance and has many extensions. However, it is dependent on large amount of convolutional operations so it is not suitable for mobile device. An open source facial analysis SDK called Open-Face is an implementation of CLM method[3].

In addition, Dlib implements the paper named *One Millisecond Face Alignment with an Ensemble of Regression Trees* by Vahid Kazemi and Josephine Sullivan, CVPR 2014[5] for facial landmark tracking. This paper suggests a new algorithm that can speed up the process with state-of-the-art performance by identifying the essential components of prior face alignment algorithms and then incorporating them in a streamlined formulation into a cascade of high capacity regression functions learnt via gradient boosting.

2.3. 68-points facial landmark model:

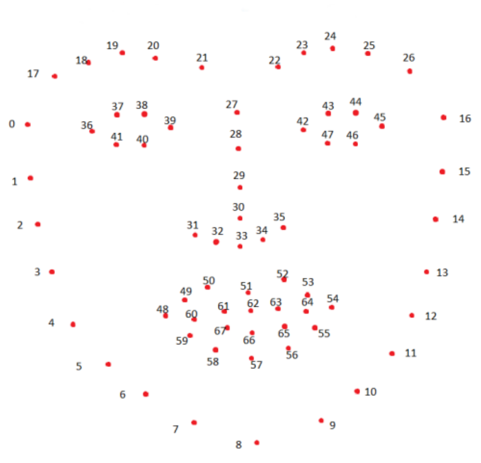


Figure 1： 68-points mark for annotation [4]

For the point distribution model, the default model provided by Dlib is a 68-point facial landmark model [4]. With the labeled landmarks on our face, we are able to add additional functions to our application like facial expression detection. As the change of relative position of the feature points reflects the shape change of the face. For example, we can tell if a person is smiling by checking the arch of the mouth's shape.

3. Approach

Our baseline version of facial landmark tracking is the work of an open source on github: zweigraf/face-landmarking-ios [6].

3.1 Flow chart:

The following figure shows the basic facial land/marking process consisting of face detection and facial landmark tracking.

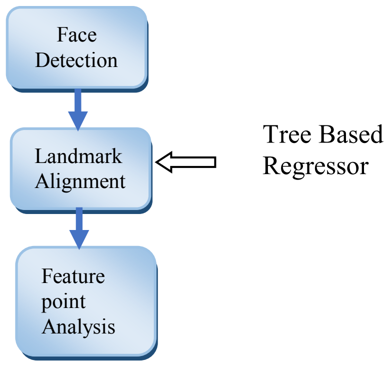


Figure 2：Outline of the process

In addition, Figure 3 shows how the input data is processed in the Xcode. Asshown in the flow chart, after the capture outputs added to the session, the rectangle bounds based on the detected faces (CVMetaDataFaceObject) from CVMetaDataOutput as well as the CMSampleBuffer are used as arguments to copy the pixel data from buffer and conduct facial landmark tracking in the Dlib library.

After completing image processing, the Dlib 2d array is copied back into the buffer, which is added into the layer. Finally, the user shall see the landmarks successfully tracking the facial expressions after the layer being added into the UIView.

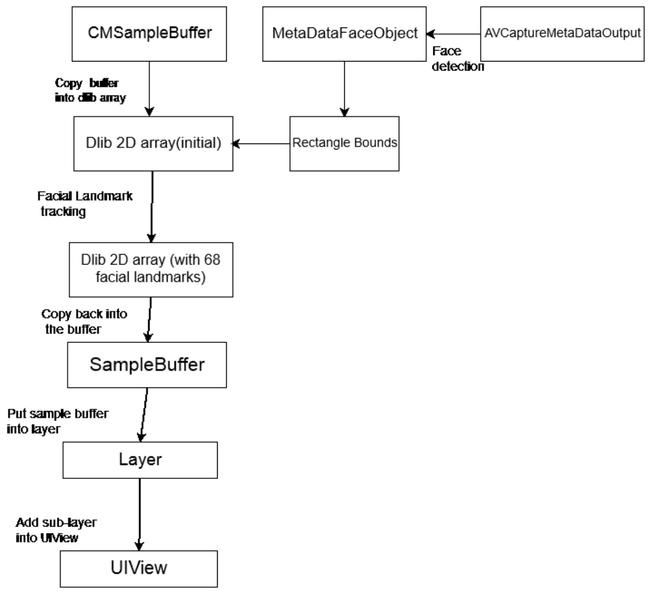


Figure 3：Flow chart of showing the whole process in Xcode

3.2 Algorithm:

As mentioned before, the algorithm is based on the paper called *One Millisecond Face Alignment with an Ensemble of Regression Trees.* Similar to supervised descent gradient (SVD), the implemented algorithm utilizes a cascade of regressors, as shown in Eq.1

$$\hat{\mathbf{S}}^{(t+1)} = \hat{\mathbf{S}}^{(t)} + r_t(I, \hat{\mathbf{S}}^{(t)}) \quad \text{(Eq.1)}$$

where the vector $S = (X_1^T, X_2^T, \ldots, X_p^T)^T \in R^{2p}$ denotes the positions of all the p facial landmarks in image I. $\hat{S}^{(t+1)}$ is an updated estimate based on the current shape estimate.

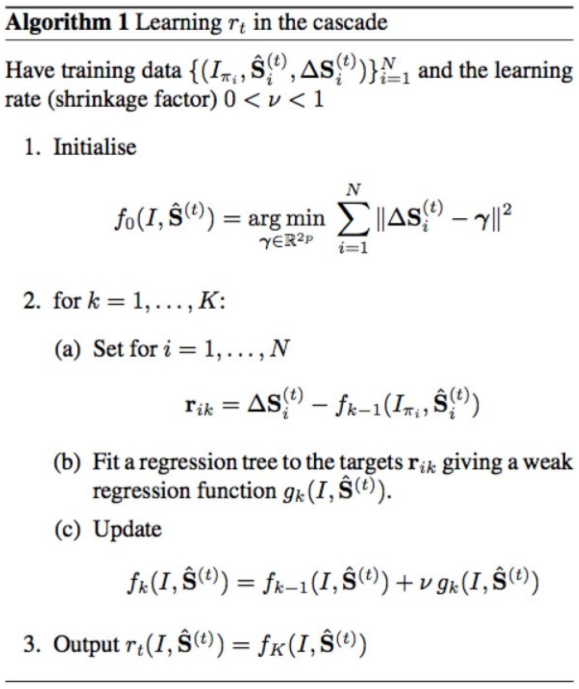
Each cascasde of regressor can estimate a new vector of facial landmarks to be used as an input for the next cascade. In this algorithm, the regressor is the gradient tree boosting algorithm with a sum of square error loss. As mentioned in the paper, this instroduces some form of geometric invariance into the process and a more precise semantic location on the face can be indexed as the cascade proceeds[5].

The following figure shows the algorithm used to learn each regressor in the cascade from the traning data[5].

According to the algorithm, for each cascade, the prcess consists of initializing, learning, updating and finally output.
To learn the first regressor, the set of training triplets $(I_{\pi_i}, \hat{S}_i^{(0)}, \Delta S_i^{(0)})$ is utilized as training data. Then the initilization utilized gradient tree boosting with a sum of square error loss to a suitable tree. In addition, each cascade regressor is built based on K weak regression functions g_k . and learn from N training images. Therefore, the next regressor r1 can be achieved by setting

$$\hat{S}_i^{(t+1)} = \hat{S}_i^{(t)} + r_t(I_{\pi_i}, \hat{S}_i^{(t)}) \quad \text{(Eq.2)}$$

$$\Delta S_i^{(t+1)} = S_{\pi_i} - \hat{S}_i^{(t+1)} \quad \text{(Eq.3)}$$



```
Algorithm 1 Learning $r_t$ in the cascade

Have training data $\{(I_{\pi_i}, \hat{\mathbf{S}}_i^{(t)}, \Delta\mathbf{S}_i^{(t)})\}_{i=1}^{N}$ and the learning
rate (shrinkage factor) $0 < \nu < 1$
  1. Initialise
       $f_0(I, \hat{\mathbf{S}}^{(t)}) = \arg\min_{\gamma \in \mathbb{R}^{2p}} \sum_{i=1}^{N} \|\Delta\mathbf{S}_i^{(t)} - \gamma\|^2$
  2. for $k = 1, \ldots, K$:
     (a) Set for $i = 1, \ldots, N$
           $\mathbf{r}_{ik} = \Delta\mathbf{S}_i^{(t)} - f_{k-1}(I_{\pi_i}, \hat{\mathbf{S}}_i^{(t)})$
     (b) Fit a regression tree to the targets $\mathbf{r}_{ik}$ giving a weak
         regression function $g_k(I, \hat{\mathbf{S}}^{(t)})$.
     (c) Update
           $f_k(I, \hat{\mathbf{S}}^{(t)}) = f_{k-1}(I, \hat{\mathbf{S}}^{(t)}) + \nu\, g_k(I, \hat{\mathbf{S}}^{(t)})$
  3. Output $r_t(I, \hat{\mathbf{S}}^{(t)}) = f_K(I, \hat{\mathbf{S}}^{(t)})$
```

Figure 4： Algorithm of learning r in the cascade [5]

Since each regressor is a regression tree, the runtime complexity on a single image is constant O(TKF) where T is the number of cascases, K is the number of weak regressors and F is the depth of each regression tree.

3.3 Software:

3.3.1 Dlib vs OpenCV:

Instead of using OpenCV, the app uses image-processing toolbox of an open source library called Dlib. Dlib is a modern C++ toolkit containing machine learning algorithms and tools to solve real world problems[7]. Similar to OpenCV, Dlib can be integrated into C++ project by including the header files. Building precompiled Dlib library into Xcode requires creating a static library called libdlib and setting appropriate flags in building setting regarding linking, architecture, etc.

Unlike OpenCV, Dlib has a class called shape predictor, which can directly doing facial landmark and head pose estimation.

3.3.2 Frameworks:

Besides the fast algorithm in Dlib implemented efficiently in this app, the available frameworks in Xcode, particularly AV-Foundation and Core Media are also a key factor for successful implementation in iOS devices.

3.3.2.1 AVCaptureSession and AVCaptureDevice

AVCaptureSession is used to perform a real-time capture[8]. After configuring the object of AVCaptureSession atomically, the session is ready for running. AVCaputreDevice represents a physical capture device and the properties associated with that device. A capture device provides inputs data to an AVCaptureSession object[8].

The default frame rate of an AV-CaptureSession object can be set by using session-Preset (normally, it is 30). Moreover, the default frame rate can also be overridden by changing active-Format property as well as active-VideoMin/MaxFrameDuration of an AV-CaptureDevice object. However, directly configuring active-Format causes the capture session no longer automatically configures the capture format[8], it is necessary to be careful with the order of configuring active-Format of an AVCaptureDevice object and the function of commit-Configuration() of an AV -CaptureSession object.

3.3.2.2 AVMetaDataFaceObject

AVMetaDataFaceObject defines the features of a single detected face[8]. Based on functions of face detection in OpenCV, AVMetadataFaceObject class can access the face detection data like yaw angle and roll angle of a detected face.

3.3.2.3 AVMetaDataFaceObject

CMSampleBuffer contains zero or more compressed sample of a particular media type[8]. As part of sample buffer, CVImageBuffer provides a convenient interface for managing image data[8]. For instance, pixel buffers, derived from CVImageBuffer can be a source of CIImage or be used for image processing work conducted by third party library like Dlib.

3.4 Architecture:

3.4.1 Multithreading:

The concurrent queues enable iOS apps to run faster than that with serialized code. In Xcode, two kinds of queues can realize concurrent programming. One is called Grand Central Dispatch (GCD) queues and the other is called NS-Operation Queue. Dispatch queues and NS-Operation can effectively speed up with independent blocks of codes executed by different queues.

4 Results

4.1 68 facial landmark tracking

The following figure shows the results of app. The facial landmarks are able to fit the right positions quite well. However, when rotating with a larger degree, the tracker may fail.

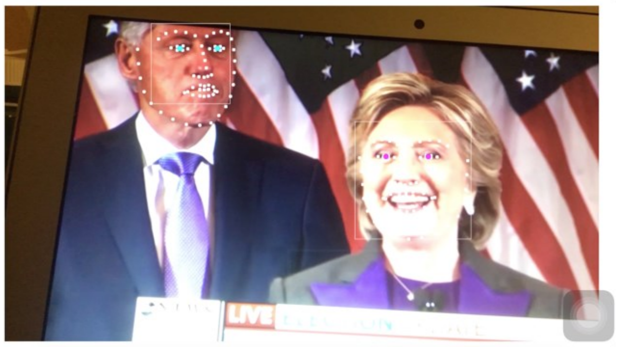


Figure 5： Screenshot showing the result of smile detection

4.2 Smile detection

When smile is detected in the face, the point in the eye will turn red from blue. This function is not 100% accurate as when the alignment of landmarks fails, it will fail to track the mouth, thus causing the inaccuracy of smile detection.

4.3 Speed Benchmark:

4.3.1 Frame Rate Measurement:

We firstly used Core Animation module to measure the frame rate of different versions. The original version has frame rate around 30 fps. However, it dramatically drops down to 11 fps when doing facial landmark with Dlib. As shown in Figure 6, by setting the frame rate to be the maximum available frame rate of iOS, the speeded app can reach around 60 fps. However, it is still not satisfactory when tracking facial landmarks.

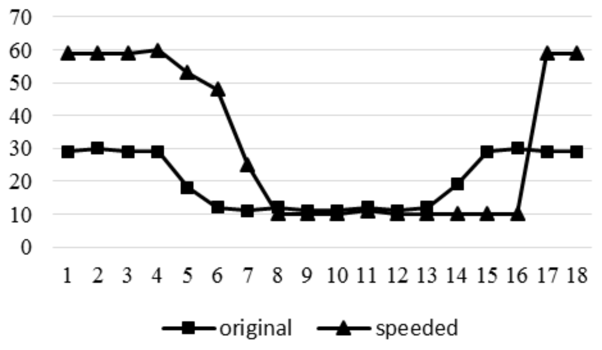
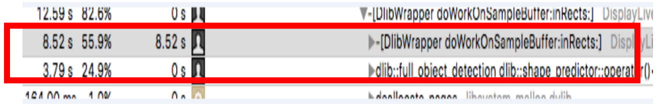


Figure 6: frame rate comparison between the original version and the version with maximum available frame rate (unit: fps)

4.3.2 Time profile:

In order to test the most time-consuming part of the program, we utilized instrumental tool available in Xcode. Figure. shows the most two expensive parts of the program. As a result, copying pixel data from Dlib 2D array back into the image buffer consumes most, followed by image processing of facial landmark tracking in Dlib. In addition, among three channels (RGB), copying the blue channel consumes most.



```
12.59 s  82.6%      0 s   ▼-[DlibWrapper doWorkOnSampleBuffer:inRects:] DisplayLive
 8.52 s  55.9%   8.52 s    ▶-[DlibWrapper doWorkOnSampleBuffer:inRects:] DisplayLi
 3.79 s  24.9%      0 s    ▶dlib::full_object_detection dlib::shape_predictor::operator()
164.00 ms  1.0%     0 s    ▶deallocate_pages libsystem_malloc.dylib
```

Figure 7: the illustration of the most time-consuming parts

Therefore, we focused on speeding up the part of copying data from image buffer into Dlib and copying back.

4.3.2.1 Removing blue channel

The first updated version is simply to remove the blue channel. The comparison of frame rates between the original one and the updated one is shown in Figure 8. The frame rate is still around 10 fps when tracking facial landmarks.

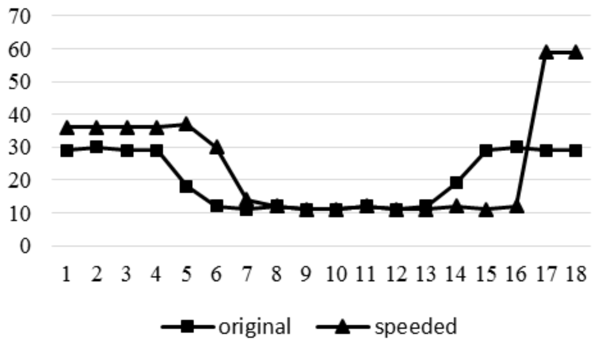


Figure 8: frame rate comparison between the original version and the version removing blue channel (unit: fps)

4.3.2.2 Dispatch queue/NSOperation

4.3.2.2.1 Optimizing of copying data from image buffer into Dlib

Figure 9 shows the comparisons between the updated version (dispatch queue) and the original version. Unfortunately, the frame rate of doing facial landmark tracking steadily is still around 10 fps.

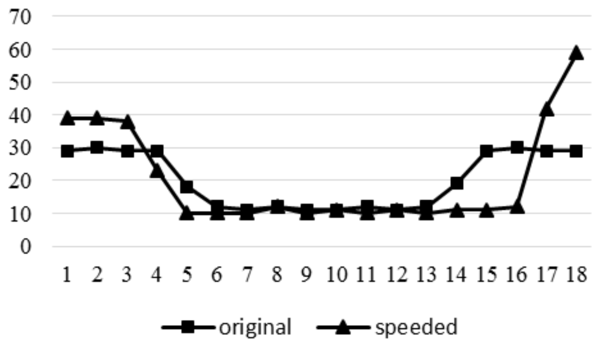
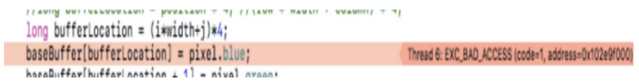


Figure 9: frame rate comparison between the original version and the version using dispatch queue (unit:fps)

Since both Dispatch queue and NSOperation aim to realize concurrent programming in Xcode, the difference of speed performance between them are not significantly. Therefore, none of the methods used above can speed up the original version significantly.

4.3.2.2.2 Optimizing of copying data from Dlib into image buffers

We stuck with the optimization by parallelizing the part where Dlib 2D array is copied(written) into the image buffer. The threads were not implemented well as shown in Figure 10. We are still working on that. The reason might be exclusive state of writes conflicts with the shared memory using concurrent queues or other misuses of concurrent queues in Xcode.



```
long bufferLocation = (i*width+j)*4;
baseBuffer[bufferLocation] = pixel.blue;                Thread 6: EXC_BAD_ACCESS (code=1, address=0x102e9f000)
```

Figure 10: The failure of using multithreading to copy pixel data from Dlib back into the image buffer.

5 Future work

Since we have not successfully implemented the parallelization of copying pixel data from Dlib back into the image buffer, we have to finish that.

In addition, the failure of face alignment when face moving quickly or rotating with a large degree motivates us to optimize the code like adding more cascades.

Last but not least, we could use other facial landmark training models besides 68-point landmark model to implement facial landmark tracking.

6 List of work

Equally Distributed.

7 GitHub pages:

https://github.com/lusiliang93/16623-facial-landmark.git

8 References

[1] Viola, Jones. Rapid object detection using a boosted cascade of simple features, 2001

[2] Viola, Jones. Robust Real-time Object Detection, IJCV, 2001.

[3] https://github.com/TadasBaltrusaitis/OpenFace.git

[4] http://dlib.net/files/shape_predictor_68_face_landmarks.dat.bz2

[5] Vahid Kazemi, Josephine Sullivan. One Millisecond Face Alignment with an Ensemble of Regression Trees, CVPR 2014.

[6] https://github.com/zweigraf/face-landmarking-ios

[7] http://dlib.net/

[8] https://developer.apple.com/reference