Project Log on WGEBML Kin Recognition

1 Related Work

1.1 Main Paper - WGEML

- The main parts of the paper are the face detection, the four face descriptors: LBP, HOG, SIFT, VGG, the penalty graphs and intrinsic graph and then using the graphs to figure out how the faces in the images are related.

2 Implementation Notes

2.1 Testing

- A folder for unit tests is made to correspond to each of the modules of the source code. This folder is under the src file and the test file is further subdivided into each source folder.
- Unit testing is done using a combination of pytest and coverage. A make command is used to run the coverage command which references a .coveragerc file which makes sure that none of the __init__.py files, the venv or test files are included in the coverage report.

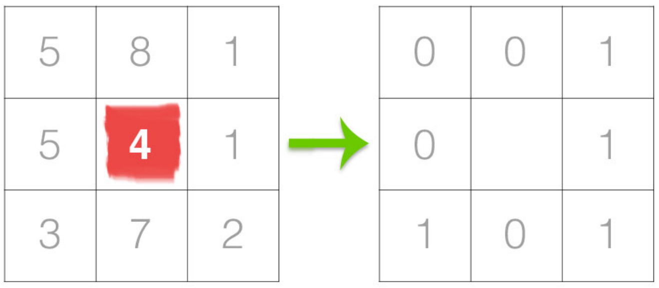
2.2 Face Detection

- Firstly, OpenCV2 was used to create a base implementation to draw a rectangle around a person's face in an image. This was done using the pre-trained classifier in "haarcascade_frontalface_default.xml". This allowed us to take a file image and output another saved file image which was the original picture with a rectangle around each face. The next step is to output an image of just the face and nothing else with the same dimensions.
- We were able to save the face on its own to an external image and change the dimensions of the outputted picture as needed. The current dimensions of the output is 64×64 as that is what the paper specified.

2.3 Feature Vectors

2.3.1 LBPs

- First, it was necessary to read a paper on LBPs applied to faces (Face Description with Local Binary Patterns: Application to Face Recognition).
- From the paper, it was found that there were 59 labels that each pixel can belong to. It can either be uniform or non-uniform and we only cared about the uniform labels. These were values where there were only at most 2 bit transitions circularly. For example, 10000001 (2 transitions) was uniform but 10101000 (6 transitions) isn't.
- It was necessary to first get the LBP value for each pixel in the image. This was done by looking at the direct neighbors of the pixel and determining if they are greater than or less than the pixel. If they were greater than the pixel, the value of that cell would be 1, otherwise it would be 0. Then, the value of the pixel in question was determined by looking at the pixel to the left and going counterclockwise and creating the bit string. In the following example:

  And so the value for the pixel becomes $01011100_2 = 92$. This value isn't uniform so this would have been marked as -1 in the process to mark it as non-uniform. This is done with every pixel in the image. For the pixels on the border, a neighborhood of size 3×3 was still taken but any "neighbors" that weren't in the image were assumed to be 0. So, for the top pixels, the 3 pixels above it were assumed to be 0, for example.
- After getting the LBP value for each pixel, the face image is split into 8×8 rectangular blocks and the vector is computed in each block. The vector is just a histogram of the values that each pixel could have been. Since there are 58 uniform values and 1 for any non-uniform values, there were 59 values that the pixels could have taken so the vector for each block would correspond to:

  $$[\text{count}(-1), \text{count}(0), ..., \text{count}(255)]$$

  Where the uniform values are ordered in ascending order.

  The vectors for each block are then concatenated to each other where the top left block is first and then the block to the right of it and so on going row by row. This outputs the 3776 dimensional vector for each face image for a 64×64 image.

2.3.2 Histogram of Gradients

- First, to compute the gradients of the image, the Sobel operator was used. To approximate the gradient in the x direction, first the kernel:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

  Was convolved on the image and then to get the gradient in the y direction, the kernel:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

  Was convolved on the image. This let us get the gradients in both the x and y direction for each pixel and then that was converted into magnitude and angle. For each pixel, the maximum magnitude and maximum angle for the 3 channels was taken to be the magnitude and angle for that pixel. For example, if a pixel had magnitudes $(1, 2, 3)$ for the (R, G, B) channels, then 3 would be the magnitude of that pixel. We also required that the angles be unsigned, so they must be between $0°$ and $180°$.

- Once the magnitudes and the unsigned angles are obtained for the image, the image is split up into blocks of size $n \times n$, in which in our case, it is first 16×16 then 8×8. For each block, a 9-dimensional vector is obtained which is the histogram of angles for that block. The labels of the histogram are the angles:

$$[0, 20, 40, 60, 80, 100, 120, 140, 160]$$

  So if a gradient has angle $0°$, then it would count towards the first bin. Given a pixel with gradient with magnitude m and angle θ, if θ is one of the labels, then you would add m to the bar with label θ. For example, if $\theta = 0$, then vec$[0]$ += m. If θ is between labels ϕ_1 and ϕ_2, then vec$[\phi_1]$ += $\frac{\phi_2 - \theta}{20} \cdot m$ and vec$[\phi_2]$ += $\frac{\theta - \phi_1}{20} \cdot m$. In other words, the amount that goes to each label is weighted with respect to the magnitude of the gradient and how close to the labels the angle of the gradient is.

- The vectors for each of the 256 blocks are created for when there are 16×16 blocks and then for the 64 blocks for when there are 8×8 blocks. The paper then doesn't seem to normalize the vectors so **that is a potential improvement on the algorithm** as normalization tends to improve performance.

- For a 64×64 face image, the vector that will come out of it will be a 2880-dimensional vector. The vectors for the 16×16 blocks are concatenated first and then the ones for the 8×8 blocks.

- Much of the information from this comes from the paper “Histograms of Oriented Gradients for Human Detection”

2.3.3 SIFT

- Read the paper ”Distinctive Image Features from Scale-Invariant Keypoints” which was what introduced the SIFT algorithm.

- The first thing to do is to create octaves for the given image. An octave is a set of the given image being blurred multiple times. For example, in the first octave, you’ll have the original image as the first image, and then you’ll blur it a bit for the next image and then that image will be blurred for the next image in the octave, etc. In the second octave, the image is halved in size and the same blurring happens. So, if the original image was of size 64×64, then the images in the second octave will be 32×32 and in the third it would be 16×16 and so on. A specified number of octaves and blurred images are used for the SIFT algorithm. The way that the image is blurred is as follows:

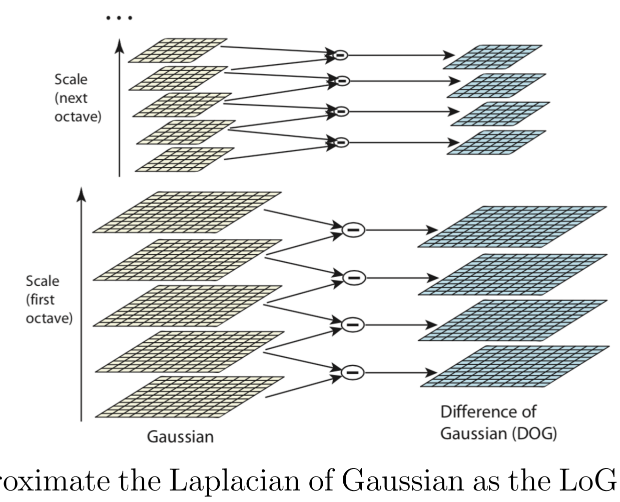
  $$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

  Where x, y is the coordinate in the image, I is the function mapping coordinates to the value of the image at that coordinate, σ is the amount of blurring, $*$ represents convolving G on the image and:

  $$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

  Which is the Gaussian blur.

- From here, for each octave, a difference of Gaussians is created to help find the keypoints of the image. The difference of Gaussians is just the octave[i] - octave[i - 1]:

  This is used to approximate the Laplacian of Gaussian as the LoG helps find the edges of the image by blurring the image a bit and then finding the second order derivatives.

It is first blurred as taking the Laplacian straight away would be sensitive to noise. However, this is computationally expensive. The Difference of Gaussians is a good approximation of the scale invariant Laplacian, $\sigma^2\nabla^2 G$.

- From here we look for the keypoints in the image using the Difference of Gaussians. This is done by finding the local maxima and minima of the DoG. Once you have the extrema of the difference of Gaussians, we need to refine the approximation of the keypoint because the actual keypoint is probably between pixels. Thus, we can use a Taylor expansion around the proposed keypoint of the scale-space function $D(\sigma, x, y)$ where σ is the blur level in the DoG. This Taylor expansion looks like:

$$D(\mathbf{x}) = D + \frac{\partial D}{\partial \mathbf{x}}^T \mathbf{x} + \frac{1}{2}\mathbf{x}^{\mathbf{T}} \frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x}$$

Where D is the value of D at the proposed keypoint, $\mathbf{x} = (\sigma, x, y)^T$, $\frac{\partial D}{\partial \mathbf{x}}^T = (\frac{\partial D}{\partial \sigma}, \frac{\partial D}{\partial x}, \frac{\partial D}{\partial y})$. Letting this equal 0, we get that the offset from our keypoint is:

$$\hat{\mathbf{x}} = -\frac{\partial^2 D}{\partial \mathbf{x}^2}^{-1} \frac{\partial D}{\partial \mathbf{x}}$$

Each of these were calculated using the numpy gradient function which is an approximation of the actual gradient of D. If our offset is greater than 0.5 in any dimension, then we want to try again since that means it's closer to another sample point. We keep trying again until we get an offset which is close to the sample point. We then find the value at the subpixel extrema:

$$D(\hat{\mathbf{x}}) = D + \frac{1}{2}\frac{\partial D}{\partial \mathbf{x}}^T \hat{\mathbf{x}}$$

And if the value of the extrema is less than 0.03×255, then we throw it out since it is an unstable extrema and has low contrast.

- Furthermore, we also eliminate any keypoints that are potentially on an edge. We can do this by looking at the hessian of the keypoint. We have that the Hessian is:

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{yx} & D_{yy} \end{bmatrix}$$

To determine whether something is a corner, we care about the eigenvalues of $\mathbf{H}$, or more specifically, the ratio between the eigenvalues. Using the trace and determinant of $\mathbf{H}$, we can find that:

$$\frac{\mathrm{Tr}(\mathbf{H})^2}{\mathrm{Det}(\mathbf{H})} = \frac{(r+1)^2}{r}$$

Where r is the ratio of the eigenvalues. Thus, letting the maximum ratio that the eigenvalues are allowed to be at be $r_0 = 10$, we just need to find if:

$$\frac{\mathrm{Tr}(\mathbf{H})^2}{\mathrm{Det}(\mathbf{H})} < \frac{(r_0+1)^2}{r_0}$$

If it is, then this is a proper keypoint. If it isn't then this is an edge so we can discard it.

- The actual paper that I was implementing didn't require the keypoints at all unfortunately. This was only realized after the keypoint orientation function was implemented. What the paper did was take 7×7 overlapping patches and, with each patch, obtained a SIFT-like feature vector. This just involved getting the gradient at each point in the patch and getting the magnitude and angle of the gradients. Then, the patch was split into a 4×4 grid and for each square in that grid, an 8 dimensional vector was obtained by doing something similar to HoG where a histogram of the angles was obtained but with only 8 bins and not having the angles being unsigned. This length 8 vector is then weighted by the Gaussian function where the center is from the center of the patch and each of these weighted 8-dimensional vectors are appended together and normalized. This is then appended to the overall feature vector. This is done for each patch in the face image and this gets a 6072-dimensional feature vector if the face image is of size 64×64.

Licensing

1. The licensing is given here `https://github.com/wiseman/sift/blob/master/LICENSE.ubc`
2. The US version of the patent is here `https://patents.google.com/patent/US6711293B1/en`

2.3.4 VGG

- Firstly, the original VGG16 architecture was looked at before looking at the one used for faces specifically. The main differences were in the number of neurons in the softmax layer and that there was a Dropout layer after the first 2 Dense layers.
- At first, I was going to train the network using the data but then was informed that this could take days and is very time-consuming.
- The alternatives that came from this were to either fine tune the network using Keras's built-in VGG function which was already trained on ImageNet and fine tune it to the needed dataset or to take the weights from `https://www.robots.ox.ac.uk/~vgg/software/vgg_face/`.
- The weights in the url were only in the forms of a matconvnet, torch, or caffe formats so if this was to be used, it would need to be transformed into something usable by Keras. This was done with the matconvnet format and using loadmat from scipy. The main difference from this and my version of the architecture, however, was that they had Convolutional layers instead of Dense layers but this was easily solved by reshaping the weights from the Convolutional layers to what was required for the Dense layers.