中国高校计算机大赛
网络技术挑战赛

基于全同态加密的安全人脸识别系统

目录

基于全同态加密的安全人脸识别系统

一、目标问题与意义价值

人脸识别是一种基于人的相貌特征信息进行身份认证的生物特征识别技术，采用非接触的方式进行识别。人脸识别系统的应用已经相当广泛，在中国就已广泛的应用于公安、海关、金融、机场等多个重要行业及领域，以及智能门禁、考勤、移动支付等民用市场。尽管人脸识别系统如今正在飞速的发展，市场规模也逐渐变得庞大，但是如今的人脸识别系统仍然存在一些安全问题。

首先是人脸数据泄露问题，部分人脸识别系统以明文形式存储人脸数据，或者使用传统加密方案存储，一旦被攻击泄露就有很大可能造成严重后果，对用户的隐私和财产造成损伤。例如，在2020年2月27日，人工智能初创公司Clearview AI被黑，平台上超过2000家客户数据落在黑客手中，而其数据库中涵盖了约30亿张人脸数据，仅靠一张照片就可以检索出全网所有相关图片及其地址连接，对客户隐私造成了极大损伤。

此外，如今仍然存在部分的人脸识别设备无法分辨出照片等和真人的区别，导致了假冒身份情况的发生。例如，在疫情期间，四川某高校内，学生使用校长照片通过门禁进出了学校。

最后就是移动设备的摄像头质量参差不齐，不同摄像头因为像素等的差距可能会对识别造成一定的影响，比如一些摄像头像素无法清晰地拍摄到人脸，进而导致识别失败，对用户的体验有一定的影响。

人脸识别属于生物特征识别技术，由于生物特征难以改变，如果人脸数据泄露将难以弥补。对于传统加密方案，在进行人脸识别时必须先对密文解密，这就使得存储人脸数据的服务商必须有私钥，这就为数据泄露留下了隐患。本项目拟使用全同态加密算法对数据采集后的识别、存储过程全程以密文形式进行，确保用户人脸数据隐私的安全。引入全同态加密算法后，存储人脸数据的服务商可直接在密文上进行人脸相似度的计算，不再拥有私钥，安全性更高。

此外，针对使用照片进行人脸识别的问题，本项目还引入了活体检测模块。考虑到项目的目的使用场景需要简洁，有效的识别，项目采用的是静默活体检测模型，而非需要对身份进行严格识别的场景使用的交互式活体检测，后者效率过低，而前者能够保证较高的准确率的同时具有很明显的速度的提升。

最后，为了提高数据采集的规范准确，考虑到摄像头质量参差不齐，可能会对识别造成一定的影响，项目的最后目标将进行硬件的落地尝试，使用英伟达的jeston nano开发板来开发人脸识别应用。

二、设计思路与方案

人脸识别系统仍然存在的一些安全问题我们在这个项目之中尝试性的做出了一些解答，以希望改进人脸识别系统，同时兼顾安全性和计算效率的统一。如图 2.1，是我们系统的一个整体的设计方案流程。

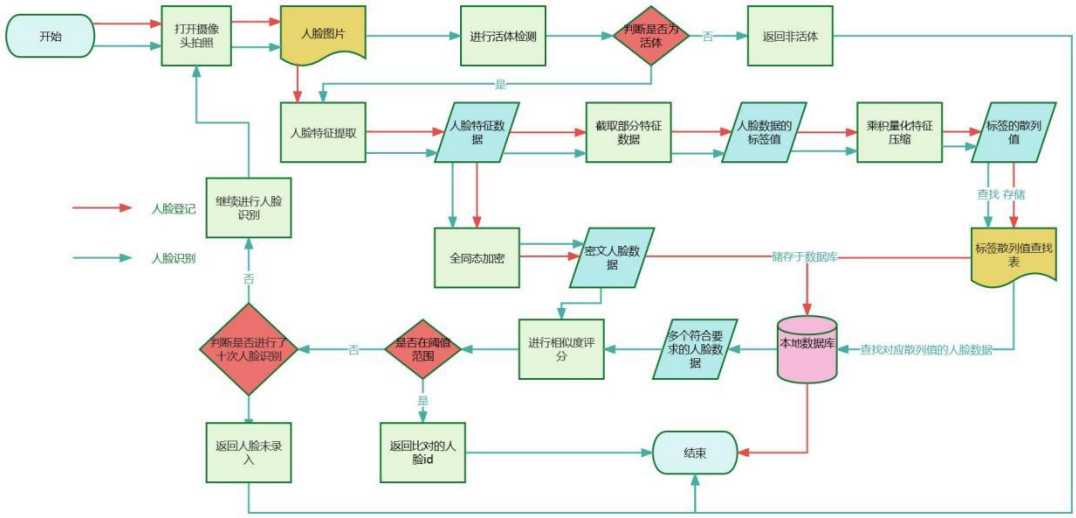


图 2.1 系统整体流程

系统中先进行人脸登记，通过摄像头进行拍照，然后经过人脸检测、对齐之后提取人脸特征，并截取部分特征值进行基于乘积量化的特征压缩，获取人脸的标签散列值，存储于数据库中，同时对提取的全部人脸特征值进行全同态加密，并存于数据库中，登记过程就结束。之后进行人脸识别，同时通过摄像头拍摄图片，然后先进行活体检测，检测是真实人脸图片后，才会提取人脸特征，否则程序运行结束，提取到人脸特征后，同样地全同态加密和特征压缩，但不存于数据库中，而是先依据标签散列值获取到一个候选人脸名单，然后又才对这个候选人脸名单做相似度计算，再判断相似度分数是否在阈值中，如果符合则返回符合人脸数据的 id，不符合的话又重复人脸识别的过程，总的需要有 10 次识别都不通过我们才判定识别的人脸未进行录入，以提高系统的识别精度。

1. 人脸检测与对齐

搭建人脸识别系统的第一步就是人脸检测，也就是在图片中找到人脸的位置。在这个过程中输入的是一张含有人脸的图像，输出的是所有人脸的矩形框。一般来说，人脸检测应该能够检测出图像中的所有人脸，不能有漏检，更不能有错检。

本项目组在查询资料后，选择如今应用最为广泛的 MTCNN 算法。MTCNN 对设备要求低，使用了级联思想，将复杂问题分解，使得模型能够在小型设备上运行，比如人脸监测模型可以在没有 GPU 的设备上运行。MTCNN 的三层网络都比较容易训练：三个级联网络都较小，训练模型时容易收敛。MTCNN 模型精度较高，在人

脸检测领域，是工业上开发人员大多都采用的选择的模型，它使用了级联思想，逐步提高精度。

MTCNN 算法是一种基于深度学习的人脸检测算法，其全称为 Multi-task Cascaded Convolutional Networks。该算法通过级联多个卷积神经网络来实现人脸检测，其中每个级联的网络都有不同的任务，包括人脸框定、人脸关键点定位和人脸属性判断等。MTCNN 算法的主要思想是通过多个级联的网络来逐步缩小搜索范围，从而提高检测的准确率和速度。具体来说，MTCNN 算法首先使用 Proposal Network（P-Net）来生成候选人脸框，然后使用 Refine Network（R-Net）来对候选框进行筛选和修正，最后使用 Output Network（O-Net）来进行人脸关键点定位和人脸属性判断。

P-Net 主要用于生成候选人脸框。具体来说，P-Net 首先将输入的图片进行卷积和池化操作，得到一系列特征图。然后，P-Net 使用滑动窗口的方式在特征图上进行扫描，对每个窗口进行分类和回归操作，得到该窗口是否包含人脸以及人脸框的位置和大小等信息。最后，P-Net 使用非极大值抑制（NMS）算法对重叠的候选框进行筛选，得到最终的候选人脸框。

R-Net 主要用于对 P-Net 生成的候选人脸框进行筛选和修正。R-Net 会首先将 P-Net 生成的候选人脸框对应的图像块进行预处理，然后将预处理后的图像块输入到 R-Net 中进行分类和回归操作。分类操作用于判断该图像块是否包含人脸，回归操作用于对候选人脸框进行位置和大小的修正。最后，R-Net 使用非极大值抑制（NMS）算法对重叠的候选框进行筛选，得到输出的人脸框。

O-Net 主要用于对 R-Net 生成的候选人脸框进行更精细的筛选和修正，是一个对 R-Net 的补充。O-Net 会进一步的对候选人脸框进行位置和大小的修正。此外，O-Net 还可以对人脸进行关键点检测，即检测人脸的眼睛、鼻子和嘴巴等关键点的位置。最后，O-Net 也会使用非极大值抑制（NMS）算法对重叠的候选框进行筛选，得到最终的人脸框和关键点位置。

人脸对齐是人脸识别中的第二个步骤，是一种常见的预处理步骤，旨在将检测到的人脸图像转换为标准的姿态和位置，以便更好地进行后续的人脸识别。在本项目中人脸对齐是通过仿射变换来实现的，通过应用仿射变换，可以将人脸图像进行平移、缩放和旋转等变换，以使得不同人脸在特征上更加一致。一般来说，人脸对齐的目标是将眼睛、鼻子和嘴巴等关键点对齐，并使得人脸的姿态和位置更加标准化。在操作时首先要在关键点定位后，根据人脸关键点的坐标，极速三出需要进行的仿射变换矩阵，使用计算得到的仿射变换矩阵，对原始图像进行仿射变换，以实现人脸对齐。常见的仿射变换包括旋转、缩放和平移。

2. 人脸特征提取

本项目对人脸识别的一些开源模型进行了调研，主要考虑到了 FaceNet 和 InsightFace 这两个开源模型及方法。

FaceNet 是由 Google 研究团队开发的人脸识别模型。它的核心思想是通过学习一个用于嵌入（embedding）人脸的高维特征空间，使得同一个人的人脸在该空间中距离更近，不同人的人脸距离更远。FaceNet 使用了三元组损失（triplet loss）来训练模型，以确保同一人的人脸特征更加接近，不同人的人脸特征更加分散。FaceNet 的输出是一个表示人脸特征的向量，可以用于人脸验证、人脸识别和人脸聚类等任务。

InsightFace 是由来自香港中文大学的研究团队开发的人脸识别模型。与 FaceNet 类似，InsightFace 也是通过学习一个高维特征空间来嵌入人脸特征。然而，InsightFace 采用了一种名为 ArcFace 的损失函数，它在三元组损失的基础上引入了角度边界（margin）来增强同一人的类内相似度并增加不同人的类间距离。这使得 InsightFace 在人脸识别任务中表现出色。

两者都是很优秀的人脸识别算法，但是考虑到已经有人做过将 FaceNet 和全同态加密结合的人脸识别系统，本项目组想要在前人的基础上为基于全同态加密的安全人脸识别提供更多的可能性，就选定了使用 InsightFace 模型。

InsightFace 使用深度卷积神经网络（DCNN）来进行人脸特征提取。提取试将准备好的人脸图像输入到网络中进行前向传播，获取网络的输出。在 InsightFace 中，输出是一个 512 维的特征向量，被称为 InsightFace 特征。这个特征向量对于同一人的不同人脸图像应该是相似的，对于不同人的人脸图像应该是不相似的。这其中使用了 ArcFace 损失函数。该损失函数通过最小化同一人脸样本与其他人脸样本之间的距离，同时最大化不同人脸样本之间的距离，以增加特征的鉴别性。

3. 全同态加密以及人脸识别

在过往的人脸识别系统中，人脸特征往往是以明文数据的形式存储于数据库的，即便我们信得过这个系统的服务提供方是可信的，我们也无法保证不存在恶意的第三方——纯恶意攻击者不会去获取数据库中所存在的赤裸裸的大量数据，在这个信息时代，数据即是资源，从资源保护的角度必须要对存储的数据进行加密。但如果仅仅是对存储的数据进行加密保护，其安全性也存在着不小的漏洞。攻击者虽然无法解密数据库中的数据，避免了数据库泄露的风险，但是攻击者对识别过程进行监听，同样会得到明文的人脸数据，这是因为在识别过程中的相关

数据运算仍然是以明文进行。基于以上问题的综合考虑，我们需要一种加密方案，它不仅能够对数据进行加密，而且能确保在使用相同计算过程的条件下，人脸数据在密文状态下的计算结果和在明文状态下的结算结果是一样的。

从而本项目就考虑到了通过全同态加密算法对采集后人脸特征数据来进行加密，人脸识别过程以及在数据库中存储时都是以密文形式进行，这就确保了用户人脸数据的隐私性。引入全同态加密算法后的系统，在采集得到明文人脸特征数据后，就马上将数据进行加密，安全性极高。但光是有足够高的安全性能这是不够的，系统的识别精确度也是一个重要的指标。本项目中提取的人脸特征都是512 位的浮点数向量，而很多全同态加密方案是只支持整数域下的计算，如果仅仅对它的整数进行识别运算的话，这势必会使得系统的识别精度较低。出于这样的考虑，本项目选择了可以对浮点数进行操作运算的 CKKS 方案，以此来作为本项目的全同态加密方案。

CKKS 是 2017 年提出的全同态加密方案。它支持浮点向量在密文空间的加乘运算并保持同态，且支持任意次加乘法的运算。本项目中全同态加密和识别结合

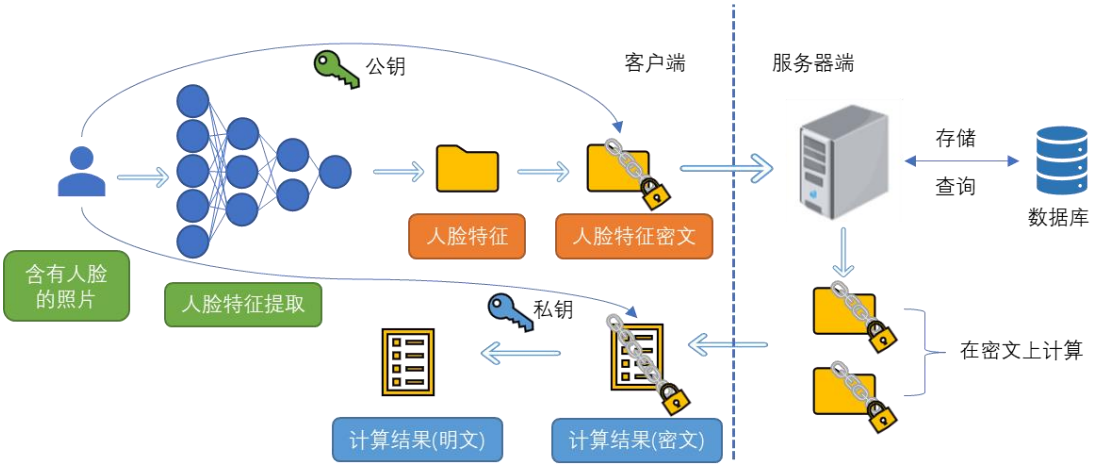


图 2.2 全同态加密识别过程

之后的工作过程如图 2.2 所示。整个过程即先使用公钥将提取的人脸特征值进行加密，然后把密文保存在数据库中，这就完成了注册；识别时同样先把要识别的人脸图像加密，然后和数据库中的密文人脸数据进行相似度计算，再将计算结果和阈值比较，符合条件则成功，不符合条件则识别失败。

在这个过程中很重要的一点是相似度计算，即计算两个人脸向量的相似分数。在我们的备选方案中有计算欧式距离和余弦距离两种方案。欧式距离其实就是在高维空间中两个点之间的绝对距离，比如在二维空间中的有两个点，分别为 $A(x_1, x_2)$ 和 $B(y_1, y_2)$，那么它们之间的欧式距离为

$$d = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} \#(2.1)$$

推广到 n 维空间中就可得到 n 维时其计算公式为

$$d=\sqrt{(x_1-y_1)^2+(x_2-y_2)^2+(x_3-y_3)^2+\ldots+(x_n-y_n)^2}=\sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}\#(2.2)$$

而余弦距离，或称为余弦相似度，则是指在高维空间中，两个向量夹角的余弦值，其二维时的计算公式为

$$d=\cos(A,B)=\frac{A\cdot B}{|A|\cdot|B|}=\frac{x_1y_1+x_2y_2}{\sqrt{x_1^2+x_2^2}\sqrt{y_1^2+y_2^2}}\#(2.3)$$

由此推出其 n 维空间时的计算公式为

$$d=\cos(A,B)=\frac{x_1y_1+x_2y_2+x_3y_3+\ldots+x_ny_n}{\sqrt{x_1^2+x_2^2+x_3^2+\ldots+x_n^2}\sqrt{y_1^2+y_2^2+y_3^2+\ldots+y_n^2}}=\frac{\sum_{i=1}^{n}x_iy_i}{\sqrt{\sum_{i=1}^{n}x_i^2}\sqrt{\sum_{i=1}^{n}y_i^2}}\#(2.4)$$

两种距离其各自具有侧重点，欧式距离的数值受到维度的影响，余弦相似度在高维的情况下也依然保持低维完全相同时相似度为 1 等性质。欧式距离体现的是距离上的绝对差异，余弦距离体现的是方向上的相对差异，并不是一个严格定义的距离。同时在计算公式上，欧式距离的计算复杂度也比余弦距离低，能够节省我们的计算开销。因此，最终我们选择了使用欧式距离的来衡量两个人脸向量之间的相似度。

4. 活体检测

针对仍然存在的某高校门禁人脸识别系统被学生使用校长照片“欺骗过机器”进出的情况，本项目还引入了活体检测模块来赋予人脸识别系统拥有分辨识别对象是否为活人的活体检测功能。

目前主流的活体解决方案分为配合式和非配合式活体，配合式活体需要用户根据提示做出相应的动作从而完成判别，而非配合式活体在用户无感的情况下直接进行活体检测，具有更好的用户体验。考虑到本项目的使用场景需要简洁、快速、有效的识别，项目采用的是非配合式活体检测，拥有更加快速的识别效率以及更好的用户体验。

非配合式活体根据成像源的不同一般分为红外图像、3D 结构光和 RGB 图像三种技术路线：红外图像滤除了特定波段的光线，天生抵御基于屏幕的假脸攻击；3D 结构光引入了深度信息，能够很容易地辨别纸质照片、屏幕等 2D 媒介的假脸攻击；RGB 图片主要通过屏幕拍摄出现的摩尔纹、纸质照片反光等一些细节信息进行判别。基于以上分析不难发现，基于 RGB 图片的活体检测与其他两种方

法相比，仅能通过图像本身的信息进行判别，但另外两种检测方法都需要额外的红外摄像头或是 3D 摄像头，这会使得我们在考虑引入硬件时成本提升，同时在一定程度上制约了我们识别系统的应用范围，因此我们最终选择了基于 RGB 的活体检测方案。

RGB 活体检测算法是一个分类任务，将人脸图像分为三种类别——真脸，2D 假脸（纸质或电子照片等），3D 假脸（3D 人脸磨具或人皮面具等），算法实现是通过多种方式来综合判断的，比如在检测 2D 假脸中，会把检测的边框扩大，去检测人脸的周围环境中是否有照片或者屏幕的边缘，同时也会检测拍摄的的图片上面是否拥有摩尔纹，以此作为一个辨别出电子照片的重要依据。而对于判断 3D 人脸磨具或是人皮面具这样的 3D 假脸，项目中通过光线反光来实现，这是因为这些 3D 假脸的材质和真人皮肤的差异使得光照之后反光不一样，基于此来作为判断依据。但在项目中最为重要的一个判别依据是傅里叶频谱图。

因为真人脸图片和假脸图片的频谱差异很大，如图 2.3 所示，可以发现真人脸的高频信息从图像的中心向外呈发散状，而假脸的高频信息分布比较单一，仅沿着水平和垂直方向延伸。因此在项目中引入了傅里叶频谱图进行辅助检测，这一方法也极大提升了模型算法的性能以及精确度。

图 2.3 真假人脸之间的傅里叶频谱图

5. 基于乘积量化的特征压缩

全同态加密方案的一大缺点在于计算开销过于巨大，如果在识别时对数据库中的每一条数据都和待识别人脸数据进行欧式距离计算的话，这会大大延长整个识别过程的时间，这就使得用户的体验感会大大折扣，没有实际应用的意义。因此本项目中使用预分类来帮助我们在计算欧式距离之间先筛选出一部分人脸数据，这一部分人脸数据是和待识别的人脸具有一定相似度的，然后再对这一部分人脸数据来计算欧式距离，这会大大减少识别过程所需要的时间。

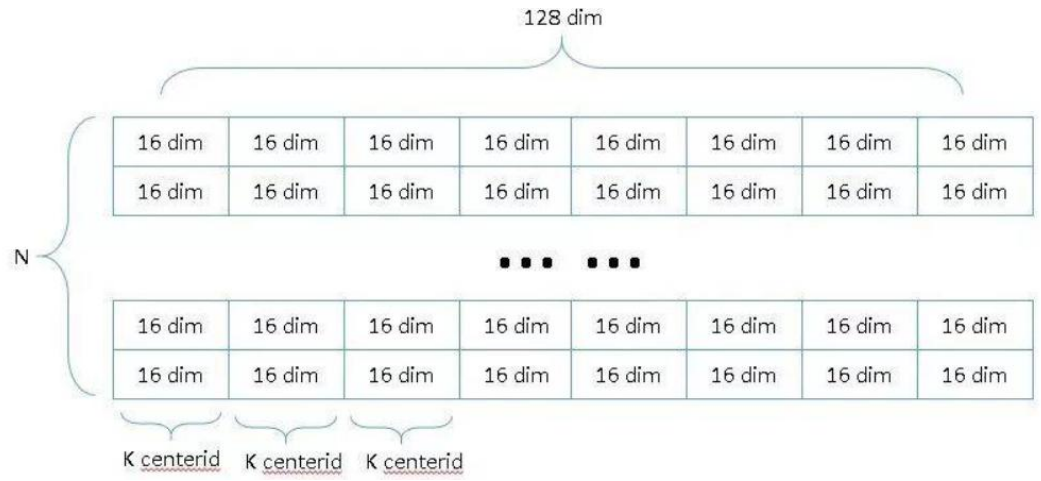


图 2.4 乘积量化分段

预分类的实现我们使用到了乘积量化的思想。乘积量化源于图像检索，本质上是对向量的压缩。假设总共有 N 个数据点，数据维数为 D 维，采用暴力检索方式，计算复杂度则维 N*D。但如果对向量进行了压缩，那么就可以降低计算复杂度。如上图 2.4 所示，设原始向量的维数 D 为 128 维，将向量分为 M=8 段，每段长度则为 16 维。

然后采用聚类方法，对于每段训练一个 K=256 中心的聚类模型，然后用聚类中心的 id(8bits)表示每一段，且每一段都是独立的聚类模型和聚类中心点。如下图 2.5 所示。

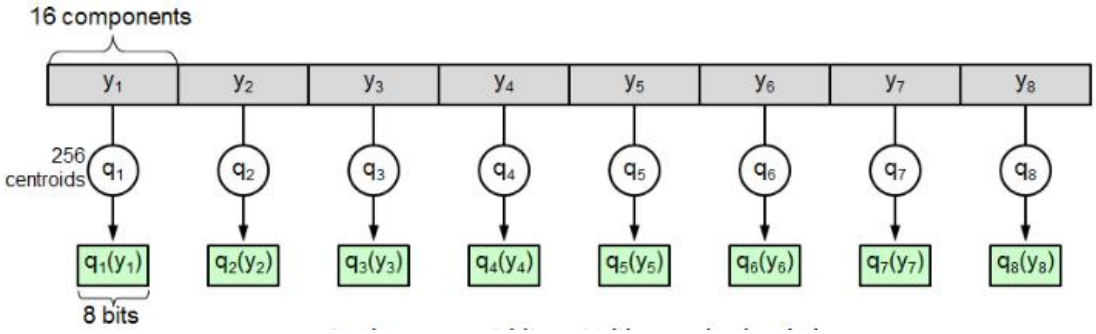


图 2.5 训练聚类模型

因此，一个 128 维的向量就被压缩为了一个 8 维的向量。

而对于压缩后向量的计算问题，有对称距离和非对称距离两种计算方法。其中对称距离是直接对压缩后的向量 x，y 进行欧式距离计算；而非对称距离则是需要计算压缩向量 x 与 q(y)(k 个聚类中心 id)，制成查找表，然后进一步操作。其中对称距离误差相对较大。由于本项目中使用 PQ 量化仅仅是作为索引来进行使用，并且是为了找出多个候选目标，选择对称距离或非对称距离并无太大的影响，因此选择了对称距离。

具体流程实现如下图 2.6 所示：

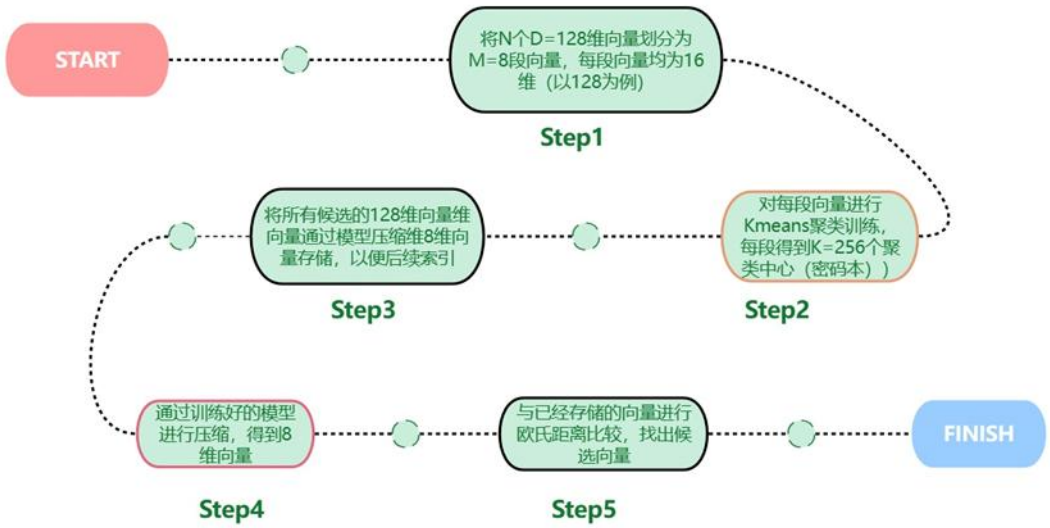


图 2.6 乘积量化特征压缩流程

三、具体方案实现

1. Web 前端人脸检测

前端人脸检测使用 faceapi.js，该开源项目使用 js 代码实现了人脸检测，且具有较好效果。faceapi.js 有多个人脸检测模型，本项目使用其中的 Tiny Face Detector 模型，因为该模型占用资源更少、速度更快，在移动设备上也适用。前端可以实时检测人脸，当人脸识别度大于 0.9 时，则认为捕捉到了清晰的人脸，然后自动拍照并上传至后端，具体流程如下图 3.1。

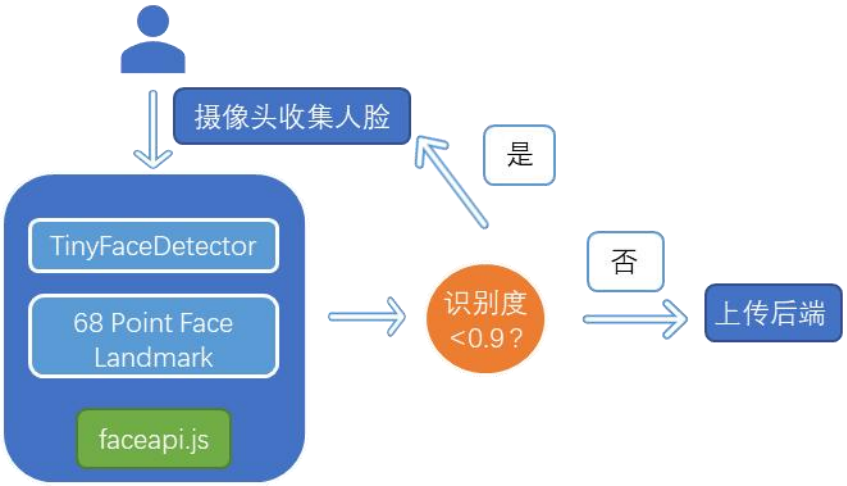


图 3.1 前端人脸检测流程

前端 js 代码中定义了一个 beginDetection()函数，该函数主要功能是获取摄像头拍摄的照片，然后调用 faceapi.js 接口检测人脸，并调用 faceapi.draw.drawDetections()绘制人脸边框展示给用户。使用 js 中的

setInterval()函数每0.2秒调用一次beginDetection()即可实现人脸实时检测。

```
setInterval(() => {this.beginDetection()}, 200);   //实时人脸检测
async beginDetection() {
    /*人脸检测*/
    const detections = await faceapi.detectAllFaces(input, new faceap
    i.TinyFaceDetectorOptions()).withFaceLandmarks(true);
    const displaySize = { width: input.width, height: input.height };
    /*绘制边框*/
    const canvas = this.$refs.overlay;
    canvas.getContext('2d').clearRect(0, 0, this.videoWidth, this.vid
    eoHeight);
    await faceapi.matchDimensions(canvas, displaySize);
    const resizedDetections = await faceapi.resizeResults(detections,
     displaySize);
    await faceapi.draw.drawDetections(canvas, resizedDetections);
}
```

为了保证后端顺利检测并提取人脸特征，前端在人脸识别度小于0.9时继续采集图像，当图像中人脸识别度不低于0.9时就将图像上传至后端。

```
for(var i = 0; i < detections.length; ++i)
{   /*判断照片中是否含有人脸*/
    if(detections[i].detection.classScore < 0.9)  //如果人脸识别度小于
0.9就继续检测
        continue;
    else
    //将图像传输至后端
}
```

2. CKKS方案的Python实现

CKKS是2017年提出的同态加密方案。它支持浮点向量在密文空间的加减乘运算并保持同态，但是只支持有限次乘法的运算。在加密的数据上训练机器学习模型或者计算加密数据之间的“距离”等领域，CKKS方案是迄今为止最好的同

态加密方案。本项目使用全同态加密算法 CKKS 将人脸原始数据加密，在密文上进行相应比较运算，从而避免人脸数据的泄露。

2.1 编译 SEAL Python 拓展

微软的 SEAL 库目前仅有 C++代码，由于人脸识别项目大多采用 Python 实现，因此需要将 C++代码迁移到 Python 上。本项目使用 pybind11 库实现使用 Python 调用 C++代码。首先按照 SEAL 官方文档编译 SEAL 代码，生成头文件和动态链接库。Pybind11 封装 C++的代码使用 https://github.com/Huelse/SEAL-Python 提供的源码，该开源项目封装了主要的 SEAL 接口，编译后生成 pyd 拓展即可在 Python 上调用 SEAL 代码。该开源代码序列化密文的方式是将密文写入文件，获取密文时则从文件中读取，这种方式增加了 I/O 操作，频繁的 I/O 操作会降低速度，因此本项目对该开源代码做了微小的修改，添加了从内存字节流中读取密文的功能，减少 I/O 操作时间。

```
/*在 SEAL-Python 的 wrapper.cpp 中的 Ciphertext 类添加以下代码*/
// ciphertext.h
py::class_<Ciphertext>(m, "Ciphertext")
/*其余部分*/
.def("load_bytes", [](Ciphertext& cipher, const SEALContext& context,
 const py::bytes &data) {
    size_t buffer_size = PyBytes_GET_SIZE(data.ptr());
    const char* pointer = PyBytes_AsString(data.ptr());
    cipher.load(context, (seal_byte *)pointer, buffer_size);
});
```

添加以上代码后即可在 Python 中通过 Ciphertext. load_bytes()从 Python 字节流中读入密文，省去了先写入文件再读取文件步骤，减少了 I/O 时间。

2.2 密钥生成

对于 CKKS 方案参数，本项目使用 4096 阶多项式，多项式系数的模数由 3 个素数组成，分别是 36,32,36 比特，生成代码如下。

```
from seal import *
#创建加密参数
parms = EncryptionParameters(scheme_type.ckks)
poly_modulus_degree = 4096
parms.set_poly_modulus_degree(poly_modulus_degree)
parms.set_coeff_modulus(CoeffModulus.Create(poly_modulus_degree, [36,
 32, 36]))

scale = 2.0 ** 32  #ckks 的缩放因数
context = SEALContext(parms)
#接着创建密钥
keygen = KeyGenerator(context)
public_key = keygen.create_public_key()
secret_key = keygen.secret_key()
rl_key = keygen.create_relin_keys()
gal_key = keygen.create_galois_keys()
```

密钥生成由客户端侧完成，其中 public_key、secret_key 保留在客户端侧，parms、relin_key、galois_key 需要发送至服务器侧，用于后续计算。

```
#将密钥相关的参数保存至文件
parms.save('parms.bin')
public_key.save('pub_key.bin')
secret_key.save('sec_key.bin')
rl_key.save("rl_key.bin")
gal_key.save("gal_key.bin")
```

2.3 加密人脸数据

客户端首先导入生成的加密参数文件、公钥文件，接着使用公钥加密人脸数据，然后将密文写入文件，最后将密文文件发送至服务器。

导入加密参数的代码如下。

```
scale = 2.0 ** 32
# 导入加密参数
parms = EncryptionParameters(scheme_type.ckks)
parms.load('./keys/parms.bin')
context = SEALContext(parms)
# 创建 CKKS Encoder，用于编码明文
ckks_encoder = CKKSEncoder(context)
# 导入公钥
public_key = PublicKey()
public_key.load(context, './keys/pub_key.bin')
# 创建 encryptor 用于加密
encryptor = Encryptor(context, public_key)
```

本项目人脸数据明文为 512 维向量，CKKS Encoder 用于编码明文，然后将编码的明文加密并写入文件，最后将密文文件发送至服务器。加密代码如下。

```
#读取需要加密的人脸数据
res = temp2.reshape((1, 512)).astype(np.float)# 将人脸特征数据格式化
data = res[0].tolist()
#将数据编码为明文
plain = ckks_encoder.encode(data, scale)
#加密
cipher = encryptor.encrypt(plain)
cipher.save('./cipher.bin')
```

2.4 计算人脸相似度

服务端首先导入加密参数文件、relin_key 和 galois_key，接着生成 Evaluator 用于在密文上执行加减乘操作。导入代码如下。

```
parms = EncryptionParameters(scheme_type.ckks)
parms.load('./keys/parms.bin')
context = SEALContext(parms)
evaluator = Evaluator(context)
rl_key = RelinKeys()
rl_key.load(context, './keys/rl_key.bin')
gal_key = GaloisKeys()
gal_key.load(context, './keys/gal_key.bin')
```

本项目在计算人脸相似度时采用欧式距离，公式如(3.1)式所示。

$$d=\sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}\#(3.1)$$

SEAL 的 CKKS 方案仅支持加法、减法、乘法运算，不支持开根号。本项目的处理方式是，在远程服务器端计算根号下的值，即d^2。客户端解密后再开方，从而得到人脸识别阈值d。本项目人脸数据是 512 维向量，SEAL 在密文上的计算也是以向量形式，可以方便计算欧氏距离。SEAL 还支持密文向量的循环移位操作，结合密文上的加法和循环移位操作，可以用于计算密文向量所有元素的和。

服务端在密文上的操作如下图 3.2 所示。

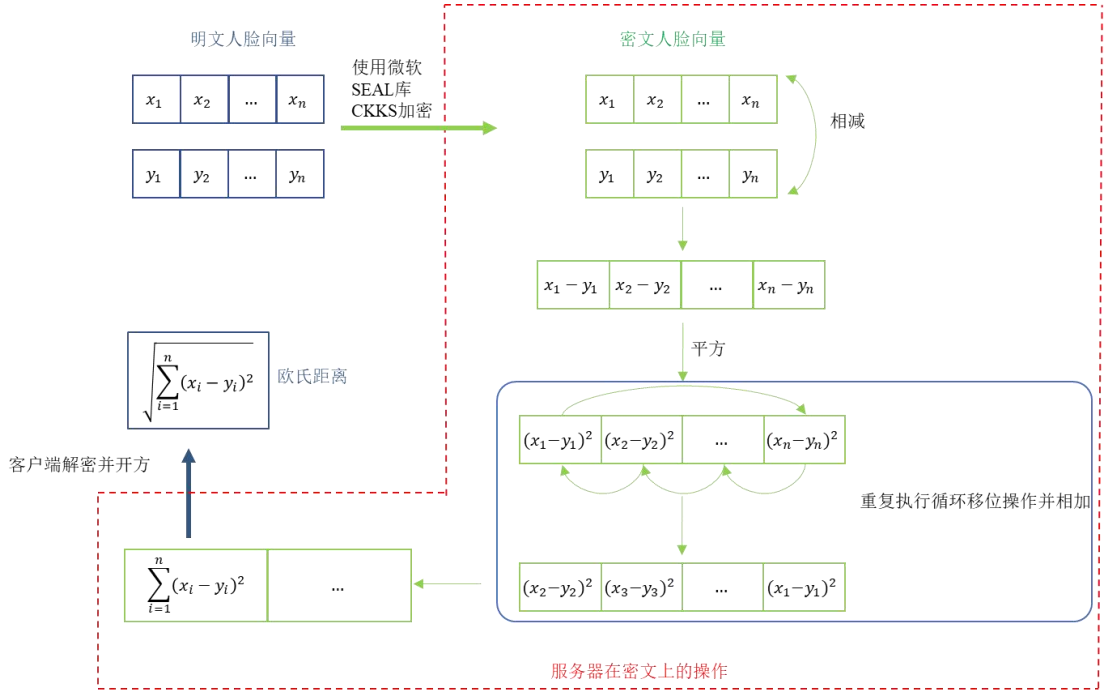


图 3.2 服务端在密文上的操作

在计算密文向量所有元素的和时，先保存密文向量的副本，然后在密文向量副本上每次循环左移 1 位，并与原密文向量相加，执行一次后，相加后的密文第 1 个元素便是原密文向量前两个元素的和，由于人脸特征为 512 维向量，需要重复执行 511 次，最终密文向量的第一个元素便是(3.1)公式中的d^2。这样效率很慢，时间复杂度为$O(n)$，本项目使用另一种算法进行优化，时间复杂度为$O(logn)$，提高了服务器在计算人脸相似度时的速度。在计算 512 维向量所有元素之和时，每次将一半的元素相加。第一次将 512 维向量循环左移 256 位，分为前后 256 维向量，并相加；然后循环左移 128 位，并相加，直到循环左移一位。算法示例如下图 3.3 所示。

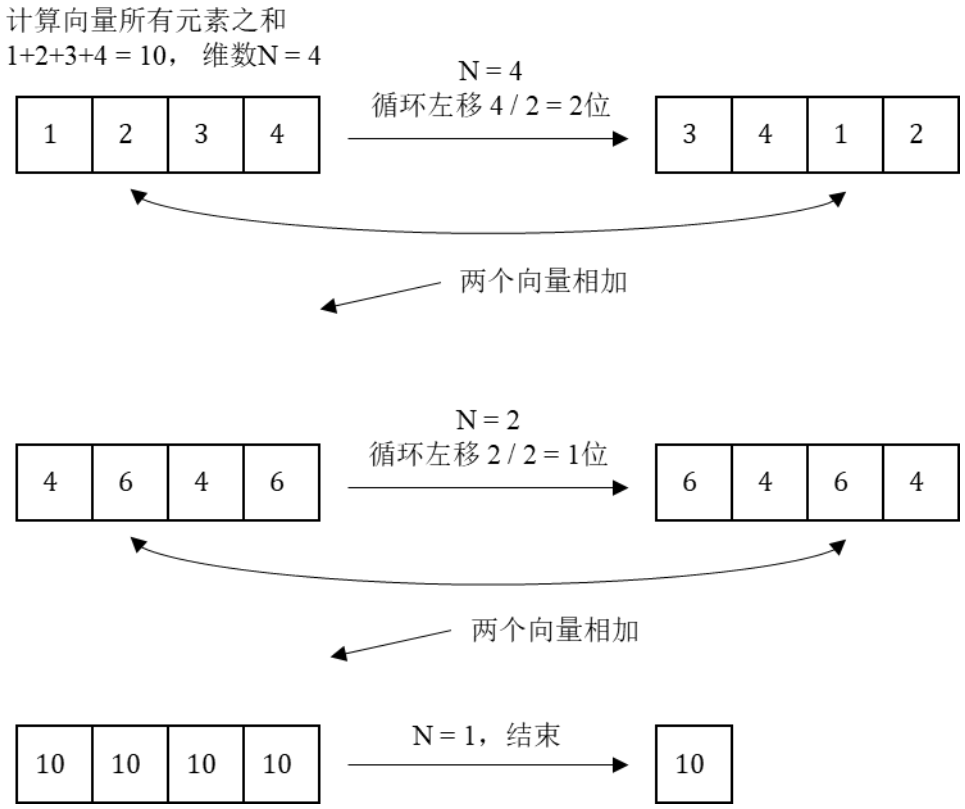


图 3.3 改进后欧式距离算法

```
cipher1 = Ciphertext()
cipher2 = Ciphertext()
cipher1.load_bytes(context, file_byte1)
cipher2.load_bytes(context, file_byte2)
diff = evaluator.sub(cipher1, cipher2)
evaluator.square_inplace(diff)  # 计算平方
evaluator.relinearize_inplace(diff, rl_key)  # 使用 rl_key 可以减少 nois
e budget 损耗
evaluator.rescale_to_next_inplace(diff)

step = 512
while True:
    step /= 2
    cipher_temp = evaluator.rotate_vector(diff, int(step), gal_key)
    diff = evaluator.add(diff, cipher_temp)
    if step == 1:
        break
return diff  # 返回密文 diff
```

具体代码实现如上所示。需要计算相似度的两个密文已保存在 file_byte1 和 file_byte2 中，使用 evaluator.sub 计算密文向量的差，evaluator.square_inplace 计算向量的平方。因为乘法操作使得密文变大，因此使用 relin_key 和 evaluator.relinearize_inplace 减少乘法运算后的密文大小。evaluator.rotate_vector 和 galois_key 可以对密文向量进行循环移位操作。

服务器最终计算结果保存到了密文 diff 中，将该密文返回客户端，由客户端解密并执行开方操作得到最终人脸向量的欧式距离。

2.5 客户端解密

客户端首先加载加密参数、secret_key，使用 Decryptor 对密文进行解密，并将明文解码，最后使用 math.sqrt 计算最终人脸相似度。具体代码实现如下。

```
parms = EncryptionParameters(scheme_type.ckks)
parms.load('./keys/parms.bin')
context = SEALContext(parms)


secret_key = SecretKey()
secret_key.load(context, './keys/sec_key.bin')
ckks_encoder = CKKSEncoder(context)
decryptor = Decryptor(context, secret_key)
cipher = Ciphertext()
cipher.load_bytes(context, file_data)
plain = decryptor.decrypt(cipher)
ret = ckks_encoder.decode(plain)
flag_diff = math.sqrt(ret[0])
distance.append(flag_diff)
```

3. 人脸检测和特征提取

3.1 人脸检测

人脸检测使用了 MTCNN 算法，基于 MxNet 来构建神经网络，同时使用 cuda 来加速程序的执行，具体实现逻辑为：首先加载 MTCNN 模型，并设置最小的人脸尺寸阈值以及可选的边界框扩充参数 margin，默认为 8。然后设置人脸检测算法的三个阈值参数，用于控制人脸检测的准确度和召回率，在检测时调用 MTCNN 模型的 detec_face 方法来进行人脸检测，返回人脸边界框数组 box_2d_array

和人脸关键点数组 point_2d_array。之后将边界框数组进行形状重塑，使其每行包含 5 个元素（4 个坐标值和一个置信度）以及提取边界框数组中的前 4 个元素，即边界框的坐标值，再紧接着调用 get_new_box 函数对每个边界框进行扩充，增加边界框的大小。扩充后的边界框存储在 new_box 变量中。把扩充后的边界框添加到 box_list 列表中，将 box_list 转换为 NumPy 数组，并将数据类型转换为整型。最终得到包含扩充后边界框的 NumPy 数组。最后对人脸关键点数组进行转置，使其每行包含一个人脸的关键点坐标，结束后返回扩充后的人脸边界框数组 box_2d_array_3 和人脸关键点数组 box_2d_array_1 作为检测结果。重要代码如下所示。

3.2 人脸对齐

人脸对齐是依据仿射变换来进行的，在具体实现时，遵循如下的步骤：首先定义了左眼、右眼和右嘴角三个关键点在图像宽高的百分比，这些百分比值将用于计算在剪裁图像中的坐标。然后获取剪裁图像的边界框坐标，从而对原始图像进行裁剪，获得人脸区域的图像数据，再计算裁剪图像的宽度和高度。获得裁剪图像的宽度和高度以后，以此来创建裁剪图像的尺寸数组 clipped_image_size。再之后将左眼、右眼和右嘴角在剪裁图像中的坐标计算为相对于剪裁图像左上角

```
# 左眼、右眼、右嘴角这 3 个关键点在图像宽高的百分比
affine_percent_1d_array = np.array([0.3333, 0.3969, 0.7867, 0.4227, 0.7,
0.7835])
# 获取剪裁图像数据及宽高信息
x1, y1, x2, y2 = box_1d_array
# 剪裁人脸区域
clipped_image_3d_array = original_image_3d_array[y1:y2, x1:x2]
clipped_image_width = x2 - x1
clipped_image_height = y2 - y1
clipped_image_size=np.array([clipped_image_width,clipped_image_height])
# 左眼、右眼、右嘴角这 3 个关键点在剪裁图中的坐标
old_point_2d_array = np.float32([
    [point_1d_array[0]-x1, point_1d_array[5]-y1],
    [point_1d_array[1]-x1, point_1d_array[6]-y1],
    [point_1d_array[4]-x1, point_1d_array[9]-y1]
    ])
# 左眼、右眼、右嘴角这 3 个关键点在仿射变换图中的坐标
new_point_2d_array = (affine_percent_1d_array.reshape(-1, 2)
    * clipped_image_size).astype('float32')
affine_matrix=cv2.getAffineTransform(old_point_2d_array,new_point_2d_ar
ray)
# 做仿射变换，并缩小像素至 112 * 112
new_size = (112, 112)
clipped_image_size = (clipped_image_width, clipped_image_height)
affine_image_3d_array = cv2.warpAffine(clipped_image_3d_array,
    affine_matrix, clipped_image_size)
affine_image_3d_array_1 = cv2.resize(affine_image_3d_array, new_size)
```

的偏移量 old_point_2d_array，紧接着根据百分比坐标和剪裁图像的尺寸，计算仿射变换后的关键点在仿射图像中的坐标 new_point_2d_array，之后调用 OpenCV 库中的 getAffineTransform 方法根据原始关键点和目标关键点，计算仿射变换矩阵 affine_matrix，然后就可以利用 warpAffine 方法和仿射变换矩阵对剪裁的人脸图像进行仿射变换，最后也可以将变换后的图像调整到指定的尺寸。重要代码如下。

3.3 人脸特征提取

人脸特征提取使用的是 InsightFace 模型来实现，最后输出为 512 位的浮点数特征向量。具体方案实现为：首先加载人脸特征提取模型，即加载模型的符号（symbol）、参数（arg_params）和辅助参数（auxiliary_params）。然后，获取模型的所有层，并选择输出层作为特征提取层。接下来，创建一个模型对象 model，并指定运行环境为 CPU 或是 GPU。通过 model.bind 绑定输入数据的形状，设置为(batch_size, 3, 112, 112)，表示每批次处理 batch_size 张 112x112 像素的 RGB 图像，batch_size 可以自行设置。

在正式提取前，要进行数据的预处理。首先检查输入图像数组的维度，如果是三维数组，则使用 np.expand_dims 函数在第一维度上进行扩展，使其成为四维数组。然后，它检查输入图像的数量是否与当前的 batch_size 相同，以及模型是否已加载，如果不满足条件，则更新 batch_size 并加载模型。接下来，调用自定义的 get_feedData 方法获取输入数据的数据批次。然后，调用模型的 forward 方法进行前向传播，并获取输出结果。其中,outputs[0] 是输出结果的第一个张量。将输出结果转换为 numpy 数组，并使用 preprocessing.normalize 函数对特征向量进行归一化处理。最后，返回特征向量数组。

其中，自定义的 get_feedData 方法的作用在于获得数据批次，首先会将输入的 image_4d_array（包含多个人脸图像的四维数组）进行处理。对于每张图像，它检查图像的高度和宽度是否为 112 像素，如果不是，则使用 OpenCV 的 resize 函数将图像调整为 112x112 的大小。然后，将处理后的图像添加到 image_list 中。接着，将 image_list 转换为 numpy 数组，并使用 np.transpose 函数将通道维度移动到合适的位置。最后，将数据包装成 mxnet 的 DataBatch 对象，并返回该对象作为数据批次。

```
def get_feature_2d_array(self, image_4d_array):
    if len(image_4d_array.shape) ==  3:
        image_4d_array = np.expand_dims(image_4d_array, 0)
    assert len(image_4d_array.shape) == 4, 'image_ndarray shape length
is not 4'
    image_quantity = len(image_4d_array)
    if image_quantity != self.batch_size or not self.model:
        self.batch_size = image_quantity
        self.model = load_model(batch_size=self.batch_size)
    feed_data = self.get_feedData(image_4d_array)
    self.model.forward(feed_data, is_train=False)
    outputs = self.model.get_outputs()
    output_2D_Array = outputs[0]
    output_2d_array = output_2D_Array.asnumpy()
    feature_2d_array = preprocessing.normalize(output_2d_array)
    return feature_2d_array
```

```
def get_feedData(self, image_4d_array):
    image_list = []
    for image_3d_array in image_4d_array:
        height, width, _ = image_3d_array.shape
        if height!= 112 or width != 112:
            image_3d_array = cv2.resize(image_3d_array, (112, 112))
        image_list.append(image_3d_array)
    image_4d_array_1 = np.array(image_list)
    image_4d_array_2 = np.transpose(image_4d_array_1, [0, 3, 1, 2])
    image_4D_Array = nd.array(image_4d_array_2)
    image_quantity = len(image_list)
    label_1D_Array = nd.ones((image_quantity, ))
    feed_data        =        mx.io.DataBatch(data=(image_4D_Array,),
label=(label_1D_Array,))
    return feed_data
```

重要实现代码如下。

4. 预分类模块

使用了开源库 nanopq。本项目中是截取了人脸特征向量(512 维)的 128 维作为压缩前的向量，我们将其划分为 8 块，即每块有 16 维。如下代码所示：

```
PQ=nanopq.PQ(M=8)
```

调用了 nanopq 库，并生成了 pq 对象以便后续使用。进一步通过代码：

```
PQ.fit(Xt)
```

其中 Xt 为训练数据，为一个矩阵，每行为一条数据，通过人脸数据库提取特征向量得到，进而通过聚类算法得到了 256 个聚类中心(密码本)。然后可以进一步的使用：

```
PQ_query=PQ.encode(query)
```

其中 query 向量为需要进行压缩的向量，为人脸特征向量的部分维数，压缩后便得到了 PQ_query，为 8 维向量。并且在进行查询过程中，就通过计算压缩后的 8 维向量的欧式距离来近似查找。

最后关于模型的保存，使用到了 pickle 库，通过其中的 load 方法来对模型进行了保存，以便后续的使用方便。

```
pickle.dump(pq0, open('./static/model/pq_1.pkl', 'wb'))
```

并且以后就可以通过加载该模型来进行调用，将部分人脸特征向量进行压缩，降低至 8 维，最后作为其索引标签。然后通过该 8 维向量进行欧式距离比较来判断是否是相似人脸。具体代码如下:

```
d=np.sqrt(np.sum(pow(np.array(x)-np.array(y), 2)))
```

5.活体检测

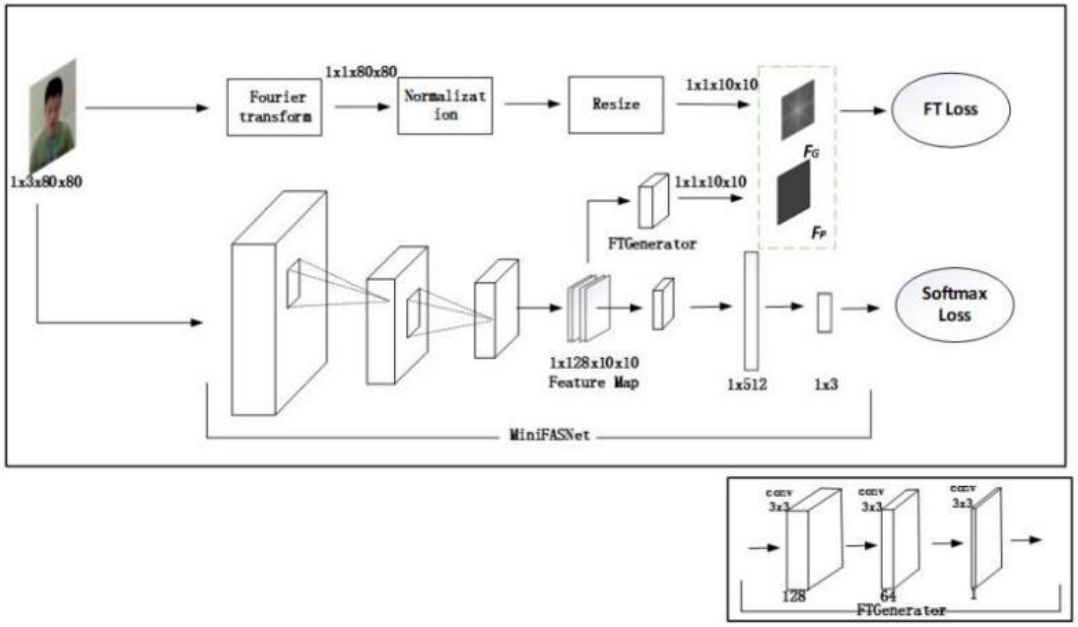
我们引入的小视科技团队的静态活体检测算法是单模态 RGB 活体检测算法，这个开源项目静态活体检测模型能够在一定程度上防止恶意的攻击者利用照片放置在摄像头之前来欺骗过识别比对，从而实现身份伪认证。

该算法在收到采集信息的第一步就进行了一次检测的小方法，对收集到的画面进多尺度 patch 截取，如图 3.4，在检测到存在明显边框的时候（即使存在照片）就判断为假脸，这可以节省掉后面的判断，减少开销。

图 3.4 多角度 patch 截取

从数据方面上来说这个模块将数据划分为三类，分别为真脸、2D 假脸（纸质攻击、视频攻击） 及 3D 假脸（3D 人脸、3D 模具等）在活体检测算法中，将数据划分为三个类别也能得到比较好的效果，毕竟这三类中每一类别多少都是

有一定共性的。由于不同攻击媒介、不同拍摄设备、不同分辨率、不同成像质量，被证实对活体检测算法都存在直接的影响，因此，如何规范输入数据就左右了模型算法的表现能力。其具体做法是对输入数据进行一系列限制，即对活体的输入限制在特定的分辨率区间，对输入图片均限制为从实时流中截取的视频帧，我们在开源的代码里，发现到了对输入图像有高宽 4:3 的一个比例要求。



3.5 整体框架实现

```
 def crop(self, org_img, bbox, scale, out_w, out_h, crop=True):
        if not crop:
            dst_img = cv2.resize(org_img, (out_w, out_h))
        else:
            src_h, src_w, _ = np.shape(org_img)
            left_top_x, left_top_y, \
                right_bottom_x, right_bottom_y = self._get_new_box(s
rc_w, src_h, bbox, scale)
            img = org_img[left_top_y: right_bottom_y+1,
                          left_top_x: right_bottom_x+1]
            dst_img = cv2.resize(img, (out_w, out_h))
        return dst_img
```

该算法也有类似于 visionlabs 单模态活检算法，采用了人工特征模式输入深度学习网络的做法。傅里叶频谱图一定程度上能够反应真假脸在频域的差异，因此该算法采用了一种基于傅里叶频谱图辅助监督的静默活体检测方法，模型架构由分类主分支和傅里叶频谱图辅助监督分支构成，团队发现真人和假人图像经过傅里叶变换后，其高频信息表现形式差距很明显，假脸的高频信息分布比较

```
class CropImage:
    @staticmethod
    def _get_new_box(src_w, src_h, bbox, scale):
        x = bbox[0]
        y = bbox[1]
        box_w = bbox[2]
        box_h = bbox[3]
        scale = min((src_h-1)/box_h, min((src_w-1)/box_w, scale))
        new_width = box_w * scale
        new_height = box_h * scale
        center_x, center_y = box_w/2+x, box_h/2+y
        left_top_x = center_x-new_width/2
        left_top_y = center_y-new_height/2
        right_bottom_x = center_x+new_width/2
        right_bottom_y = center_y+new_height/2
        if left_top_x < 0:
            right_bottom_x -= left_top_x
            left_top_x = 0
        if left_top_y < 0:
            right_bottom_y -= left_top_y
            left_top_y = 0
        if right_bottom_x > src_w-1:
            left_top_x -= right_bottom_x-src_w+1
            right_bottom_x = src_w-1
        if right_bottom_y > src_h-1:
            left_top_y -= right_bottom_y-src_h+1
            right_bottom_y = src_h-1
        return int(left_top_x), int(left_top_y),\
               int(right_bottom_x), int(right_bottom_y)
```

单一，仅沿着水平和垂直方向延伸，而真脸的高频信息从图像的中心向外呈发散状。

6. 硬件实现

本项目一开始选定实现 openMV 来作为项目的硬件落地设备，但是在实际操作时发现，其摄像头的分辨率过于低，无法满足本项目的需求，后来又选择了 Nvidia 的 Jetson nano 作为硬件落地设备，但由于该开发板的底层架构和平时所使用电脑的架构不一致，带来了诸多的困难，再加之时间紧迫，所以在选拔赛截止日前，本项目组也没有将项目实现完全硬件落地，这一部分会在未来进一步的完善，以使得项目达到最佳的效果。

我们现如今仅仅将硬件作为一个摄像头拍摄图片，并且将图像传输到服务器

端进行进一步的操作。具体实现如下:

Jetson nano 等嵌入式设备通常使用 CSI 摄像头，本项目通过浏览器调用摄像 头，但是 Chromium、Firefox 等主流浏览器均无法识别 CSI 摄像头，因此本项目 使用 v4l2loopback 和 gst-launch-1.0 工具创建虚拟摄像头，并将物理摄像头挂载 到虚拟摄像头上，从而使浏览器顺利识别摄像头。

Jetson nano 自带 gst-launch-1.0 工具，该工具用于将物理摄像头的视频流推送 到虚拟摄像头上。根据 https://github.com/umlaeute/v4l2loopback 的指南下载并在 Jetson nano 上安装 v4l2loopback，该工具可以创建虚拟摄像头，本项目的 Jetson nano 中 Jetpack 版本为 4.5.1，支持的 v4l2loopback 的版本为 0.9.1。 安装完毕后，打开 Jetson nano 的终端，运行以下命令打开 v4l2loopback 工具。

接着运行如下命令，创建虚拟摄像头，该摄像头为/dev/video2。

```
sudo modprobe v4l2loopback #打开 v4l2loopback 工具
```

```
#创建编号为 2 的虚拟摄像头设备
sudo rmmod v4l2loopback
sudo modprobe v4l2loopback exclusive_caps=1 video_nr=2
```

```
gst-launch-1.0 nvarguscamerasrc ! nvvidconv ! video/x-raw, width=1920, height=1080, fr amerate=30/1, format=YUY2 ! identity drop-allocation=1 ! v4l2sink device=/dev/video2
```

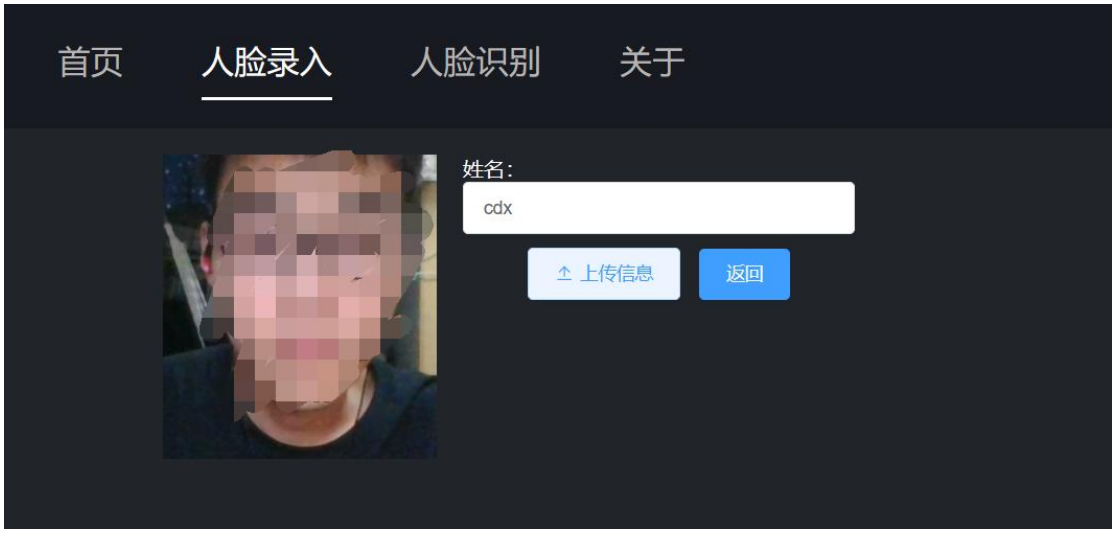
创建虚拟摄像头成功后，使用 gst-launch-1.0 工具将物理摄像头的视频流推送到 虚拟摄像头上，命令如下。

四、运行结果与应用效果

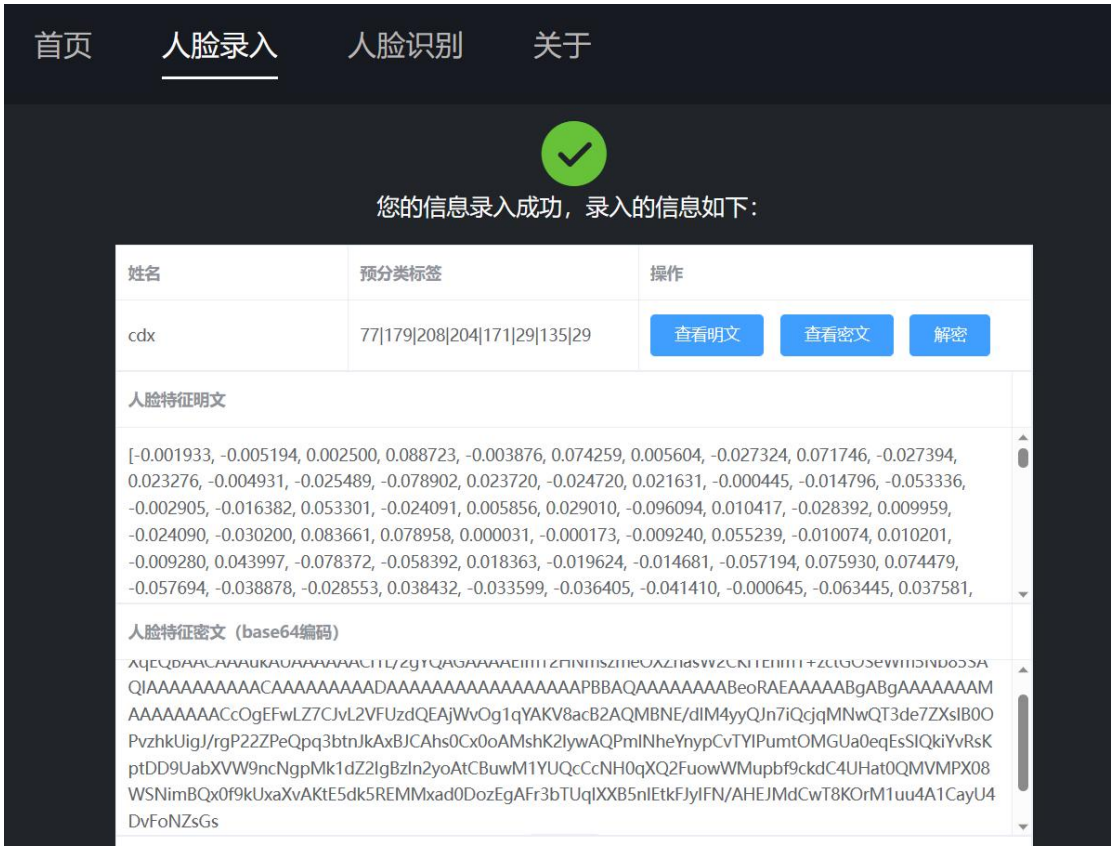
为了更好的演示整个流程，使用 web 系统来展示我们的项目细节:

1. 首先登录 web 界面对一个人脸进行录入注册：

人脸录入



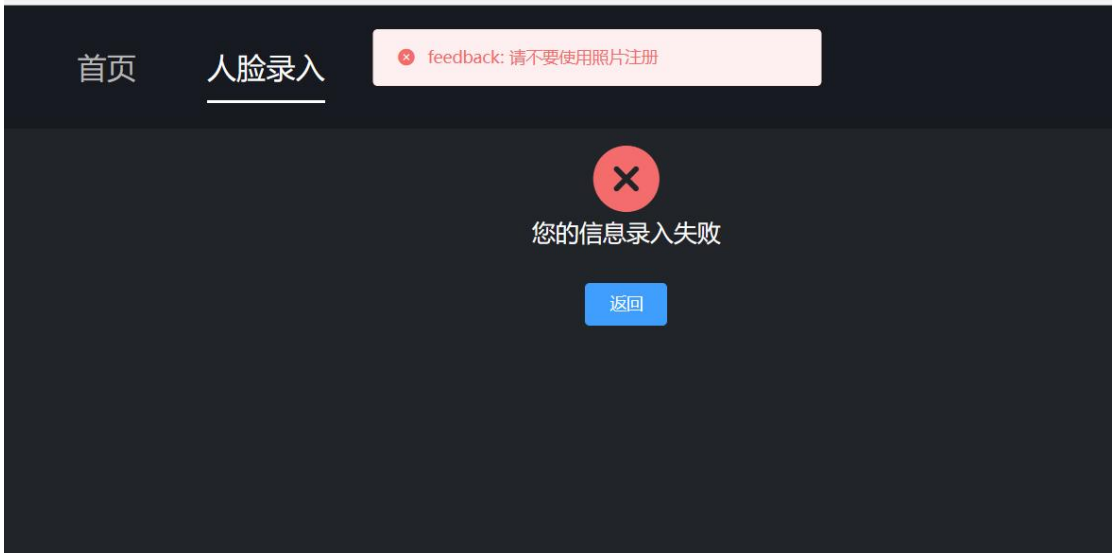
人脸信息填好后上传录入成功.



2. 随后使用一张由人脸图片放置在摄像头之前，测试简单的活体检测，结果会返回报错：

人脸图片（非真人）录入

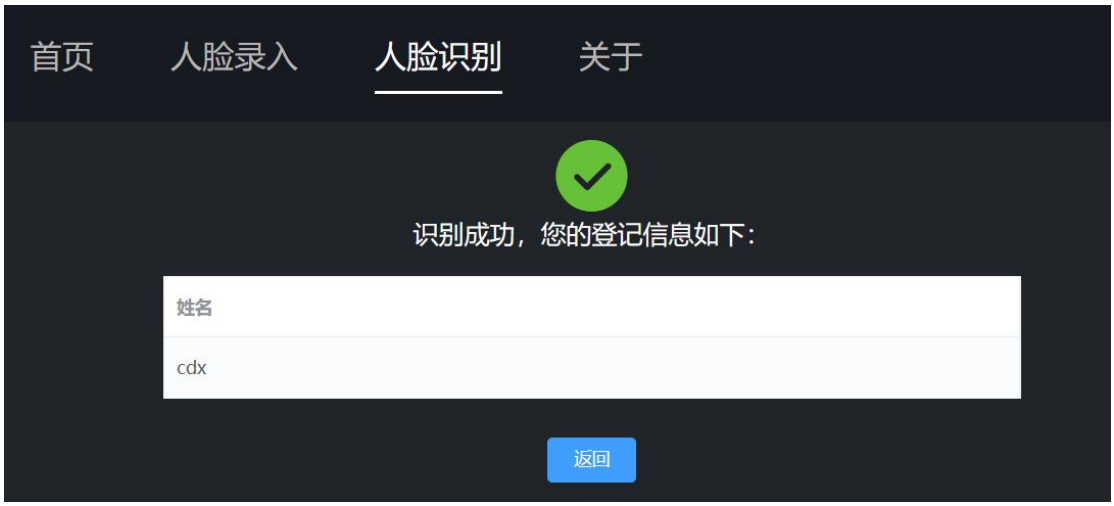
录入失败



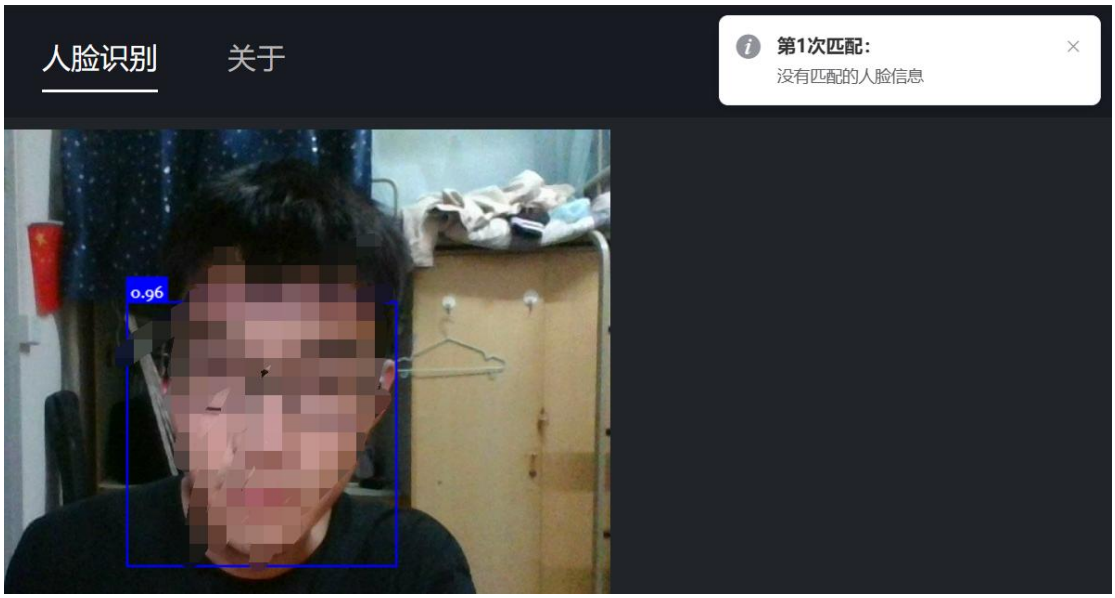
3. 测试用真人进行人脸识别，显示识别是否存在这个人脸：

真人人脸检测

检测成功



检测失败，人脸不存在



4. 测试时为了更好的调试代码，未对数据的输出进行完全的隐藏以观察效率，下面是后台隐藏输出前的一些输出截图：

服务端密文计算检测

```
服务端密文计算用时0.028919900000005327
服务端密文载入用时0.0018056000000115091
服务端密文计算用时0.027032599999984086
服务端密文载入用时0.0010374999999953616
服务端密文计算用时0.024978599999997186
服务端密文载入用时0.0013108999999644766
服务端密文计算用时0.025127499999996417
服务端密文载入用时0.0019111000000293643
服务端密文计算用时0.023913100000015675
服务端全同态部分耗时： 1.7725714999999695 s
```

预分类数据检测

```
预分类flag: 0.7636363636363637
预分类flag: 0.8076923076923077
预分类flag: 0.75
预分类flag: 0.8148148148148148
预分类flag: 0.7924528301886793
预分类flag: 0.7636363636363637
预分类flag: 0.76
预分类flag: 0.7407407407407407
预分类flag: 0.7407407407407407
```

最终识别流程相关信息

```
decrypted distance (square): 1.900470958983714
 decrypted distance (root): 1.3785756994027256
decrypted distance (square): 2.022981228325187
 decrypted distance (root): 1.422315446138861
decrypted distance (square): 0.1727204174405878
 decrypted distance (root): 0.41559645985088445
客户端全同态部分用时: 0.14650280000000748 s
识别结果: {'code': 0, 'id': 504}
识别用时: 3.6256040999999755 s
```

五、创新与特色

与现有的基于全同态加密的人脸识别系统不同，本项目进行了进一步的优化与创新，引入了预分类方案，活体检测，以及利用了硬件来进行实现。具体的内容如下所示：

① 预分类模块

对人脸特征向量截取了部分，再进行特征压缩，得到散列值，然后将具体的散列值值来作为对应人脸的索引，在进行人脸查询的过程中，先对索引进行查找，根据所有找到一个候选列表，其中包含多个人脸，然后再对其中的密文进行全同态加密，能够大大地提高人脸识别效率。

② 活体检测

活体检测通过静默实体检测来进一步确保人脸识别系统的安全性，避免了系统会被人脸照片，视频，人偶等非活体手段攻破，而进一步被恶意攻击者利用，对用户的隐私财产带来严重危害。

③ 硬件实现

利用了硬件来进行拍摄识别，能够提高系统的便利性以及可用性，相对于过去普遍软件的实现来说是最具有进步性的地方。