群智感知课程设计报告

实验名称：智能门控

班级：物联网 1802

姓名：王云鹏

学号：8213180228

日期：2021.7.18

目录

一、课设内容

作为智能家居中不可或缺的一部分，智能门控是我开发的关注点，下面给予简短的介绍。

基于 resnet 人脸识别的智能门控系统。智能门控是基于计算机视觉技术与深度学习的人脸识别系统，其硬件部分包括一块树莓派 4B 与外接的摄像头。软件部分包括一个残差网络用于实现对于人脸的识别，同时利用 PyQt 实现了一个图形用户界面，让用户只需轻击按钮即可完成完成对于人脸的识别，从而让自动控制系统打开大门。

二、功能需求分析

1. 项目背景

电影《钢铁侠》中的 AI 管家贾维斯几乎满足了人类对于智能家居的一切幻想——主动、无感、准确。“他”像一位追随多年的仆人一样了解你的 喜好，只要一个命令便可完成一套指令。而 AI 没有感情，它所以依靠的便是环境感知进而根据用户习惯进行调控，为其营造舒适的家居环境。它体现的智能技术不仅影响了我们的沟通方式，而且影响了我们在家中的生活方式。从控制我们灯光的语音助手到在工作期间监控前门的安全系统，这些技术在 21 世纪的住宅中日渐流行。

视觉和传感交互在智能家居设备的应用将成为语音之后的新兴增长点，智能家居设备将向多模态交互进一步发展。预计到 2021 年 24%的智能家居设备将搭载视觉或传感交互功能。视觉传感交互技术

的提升将进一步催化智能家居设备中的可移动性产品发展。

可视化操作是消费电子进化过程中，由功能机走向智能机的转折点。屏幕的植入，催生了操作系统的演进，进而实现了如今的屏幕时代。除了智能音箱之外，电冰箱、化妆镜、开关面板等我们日常生活中接触到的各种产品都开始屏幕化，不少产品还融入了智能生态圈。从厨卫到客厅，大屏无论是显示还是操控都更适合智能家居的应用需求。

在现在的商场或者是购物中心里，随处可见智能门的身影，而随着智能门投资的发展，智能门已经逐渐地扩大了自己的范围，不少的家庭都用上了智能门。也可以说，智能门，通过智能系统主机芯片，实现像机器人一样，守护我们的家园。这也是科技日益发展下的门业市场的发展趋势，也是智能化家居生活的重要组成部分。

2. 功能要求描述

人脸录入：界面上应当显示摄像头画面，识别出人脸的轮廓，同时用户可以通过按钮的点击完成人脸的录入

人脸识别：界面上应当显示摄像头画面，识别出人脸的轮廓，同时软件将识别出当前人脸是否是已经录入人脸的用户

三、配置环境

操作系统：Raspbian GNU/Linux 10 (buster)

Linux 内核版本：Linux raspberrypi 4.19.118-v7l+

Python 版本：Python 3.7.3

主要的包版本：

PyQt4

numpy==1.17.1

scipy==1.3.2

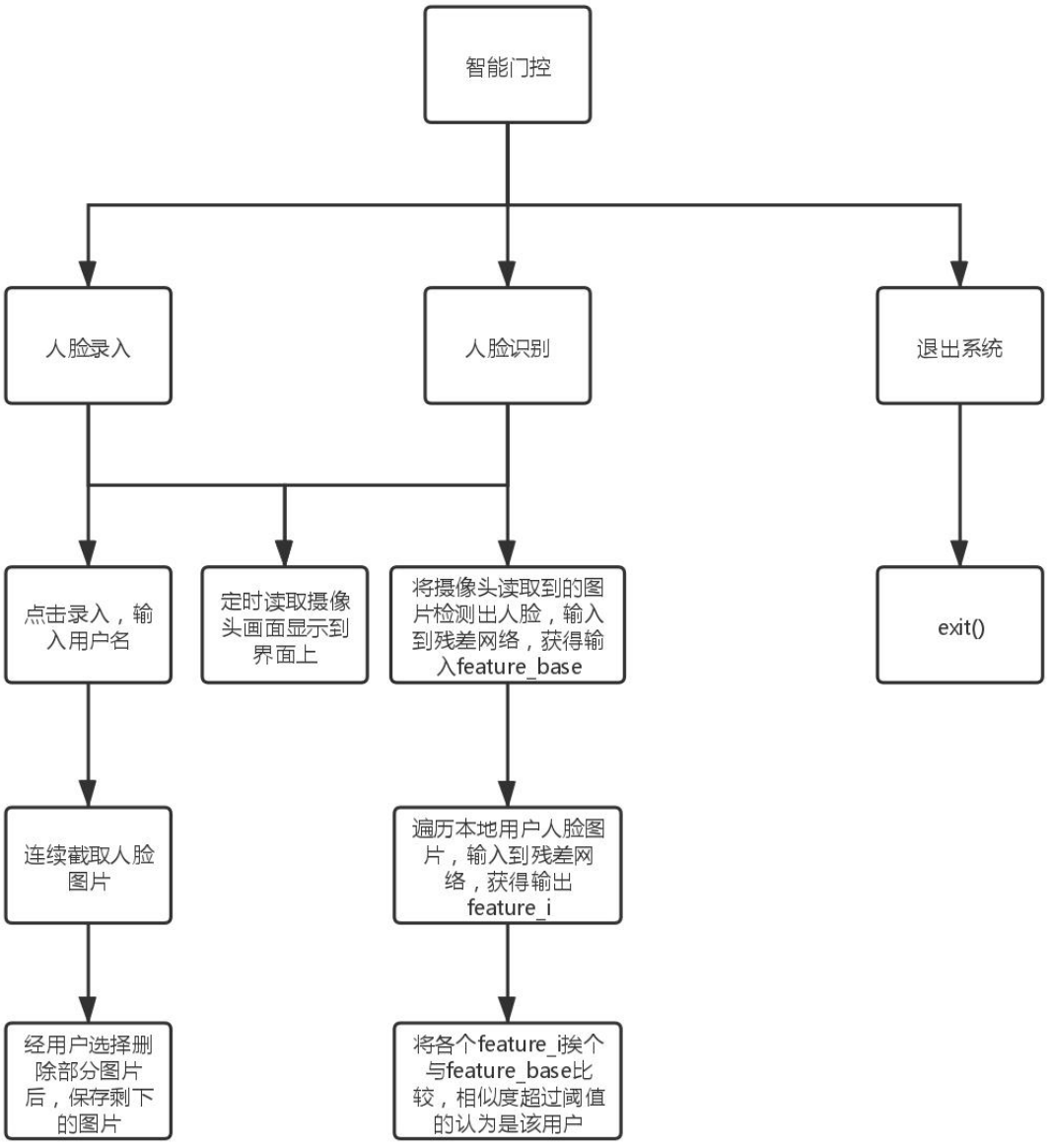
Keras==2.3.1

opencv-python==4.5.1.48

opencv-contrib-python ==4.5.2.52

四、系统设计

智能门控模块可分为三个部分：人脸录入，人脸识别，退出系统。人脸录入让用户输入用户名并且保存对应的用户人脸到本地；人脸识别将摄像头读取到的人脸图片输入深度网络获得特征，再将这个特征与本地保存的所有用户的人脸照片对应的特征进行比较，若相似度超过阈值即认定为是已录入的用户。退出系统即一键关闭软件的功能。



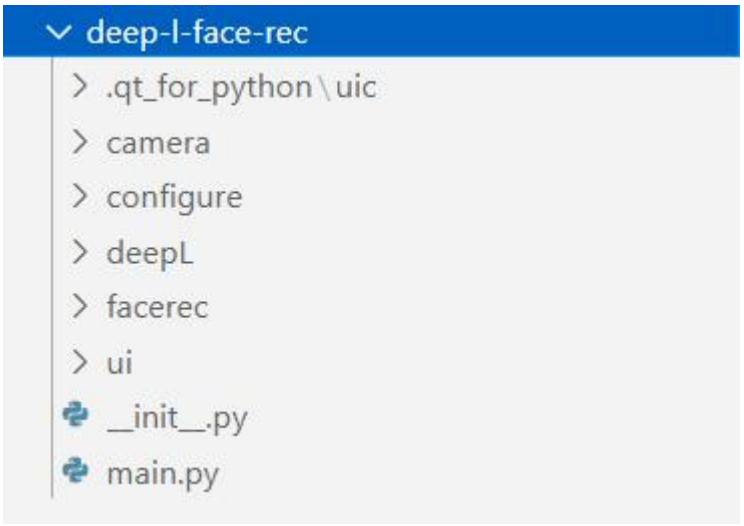
本模块的核心部分在于人脸检测与识别的 resnet 网络，人脸识别部分的实现利用了特征的比较。我们训练的残差网络做的是分类任务，将数据集中的不同人脸进行分类。那么用在人脸识别上很自然的想法是将录入的用户照片也放入网络进行训练，但是我们的代码运行在树莓派上，计算能力较弱。若用户进行一次录入就要进行一次训练，那么时间的花费是不可接受的。

所以我们采用的方法是：将用户录入的图像保存到本地，并不用于训练网络。而是在用户进行人脸识别的时候，将捕捉到的图像输入到网络获得输出向量作为特征值。然后遍历本地所有用户的头像输入到网络获取特征值，将两种特征值利用 cosine 值进行比较，相似度大于阈值的则视为是同一个人。

五、实现细节

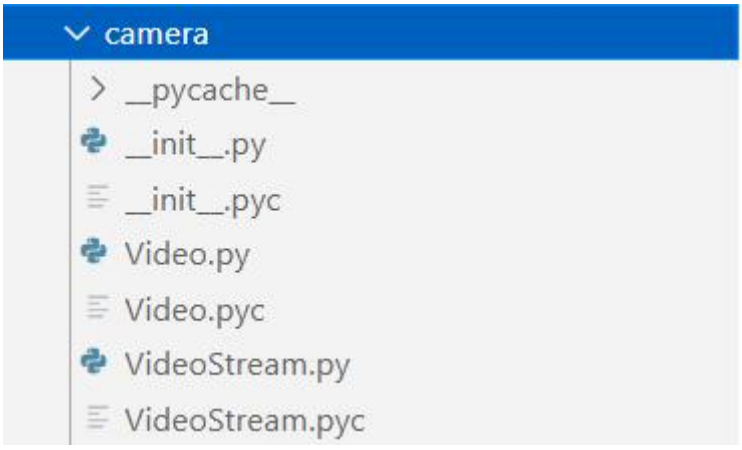
1. 文件架构

顶层文件架构



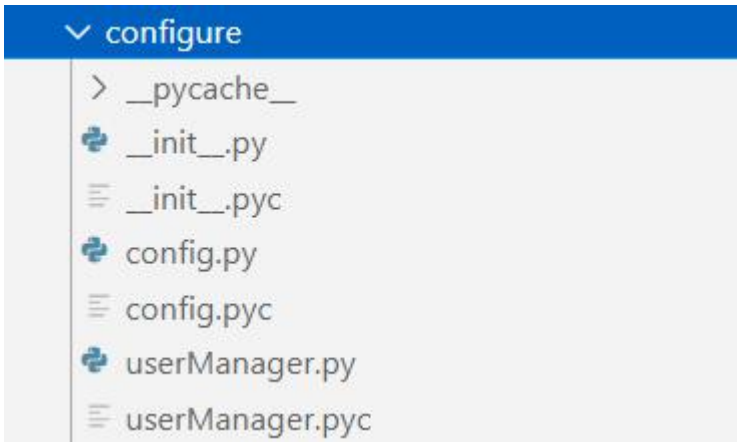
main.py 为项目主函数所在，其他文件夹则存放模块与数据等。

camera 文件架构



Video.py 中包含了 Video 类，其作用是封装 cv2.VideoCapture 函数，用于读取摄像头每帧图像的数据，可将每帧图像数据转化为 QImage 及 CVImage。

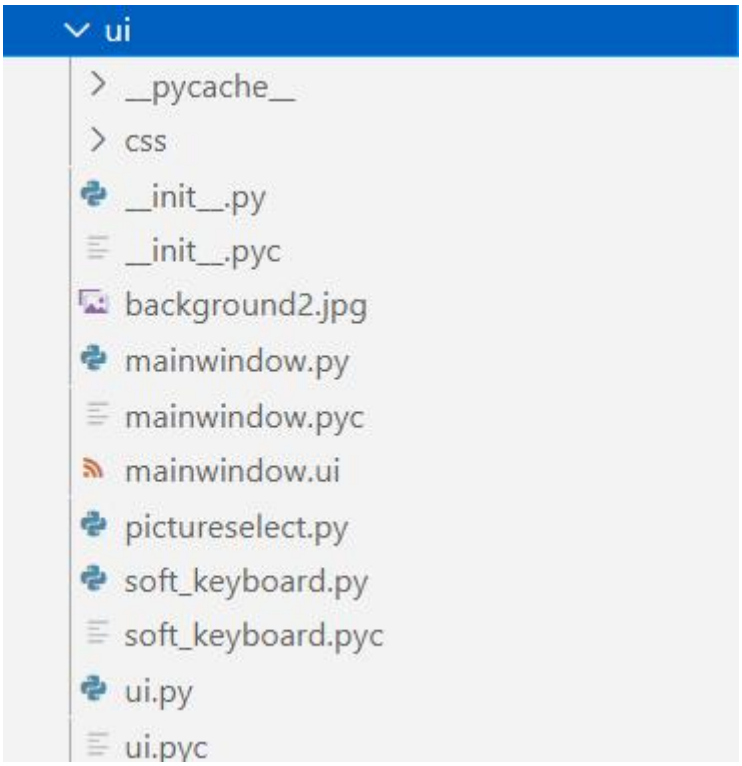
configure 文件架构



config.py 中包含了各种文件的地址，方便统一修改。

userManager.py 中包含了 UserManager 类，提供对于用户的管理功能，包括增删改查，通过对 CSV 文件的操作实现。

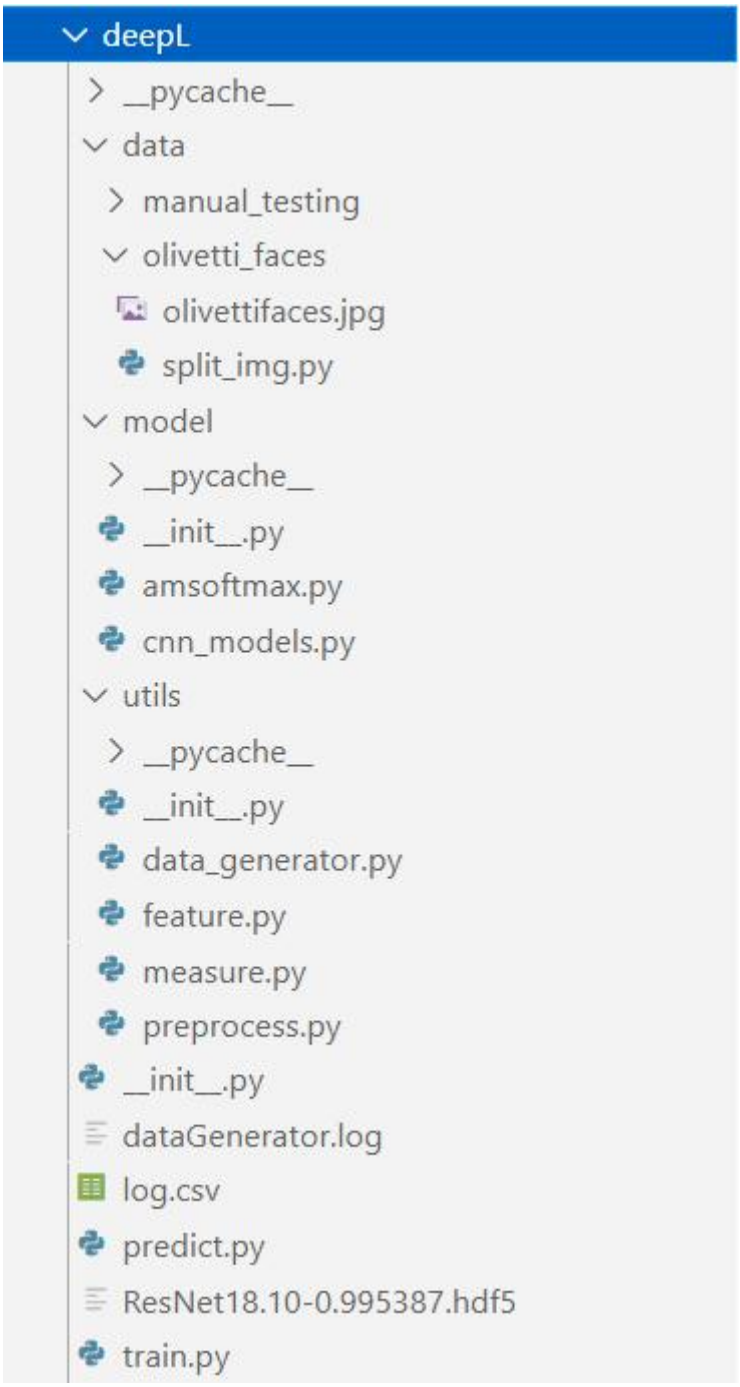
ui 文件架构



ui 模块主要是和界面有关的实现。mainwindow.py 是主界面的设计，

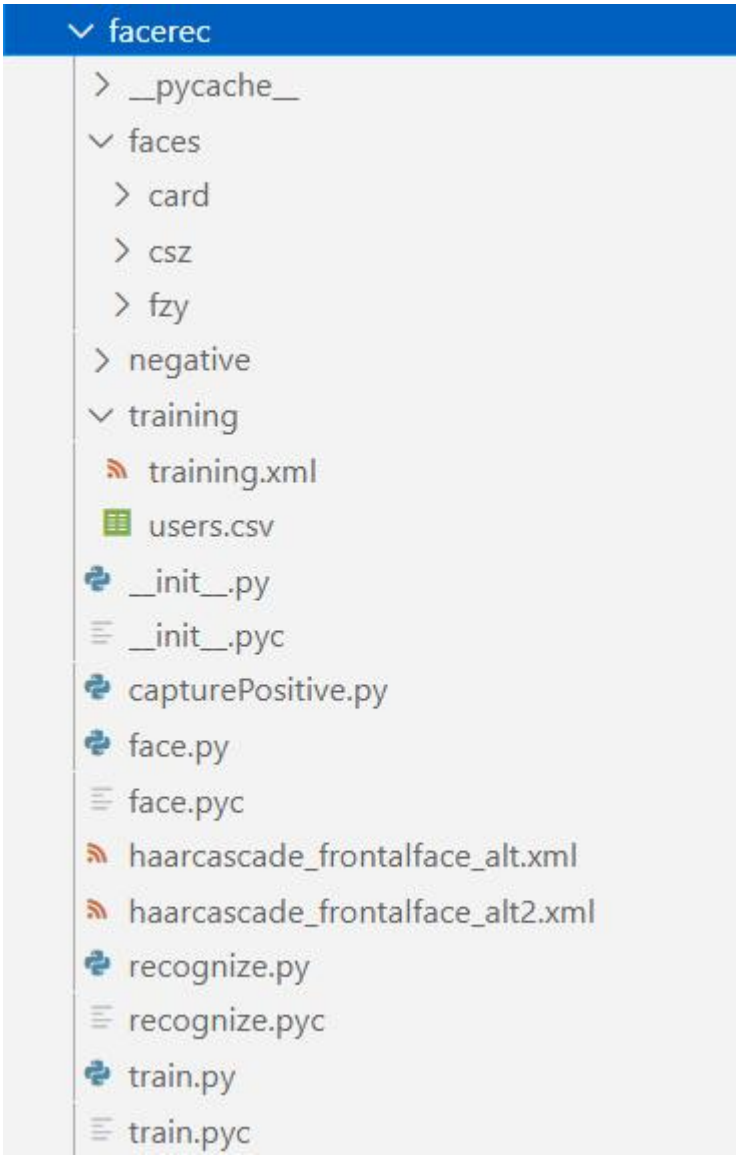
上面提供了三个按钮；pictureselect.py 是图片选择界面的设计；soft_keyboard.py 是软键盘的设计；ui.py 包括了人脸识别界面和人脸录入界面的设计。

deepL 文件架构



deepL 模块主要是关于深度学习的文件。data 文件夹下放置了训练数据 Olivetti；model 文件夹下是模型的实现与使用的损失函数的实现；utils 文件夹下有数据的生成、特征计算、特征比较、数据处理的实现；此外目录下还有网络训练的日志、与界面交互的接口、训练好的网络权重、训练网络的实现。

facerec 文件架构

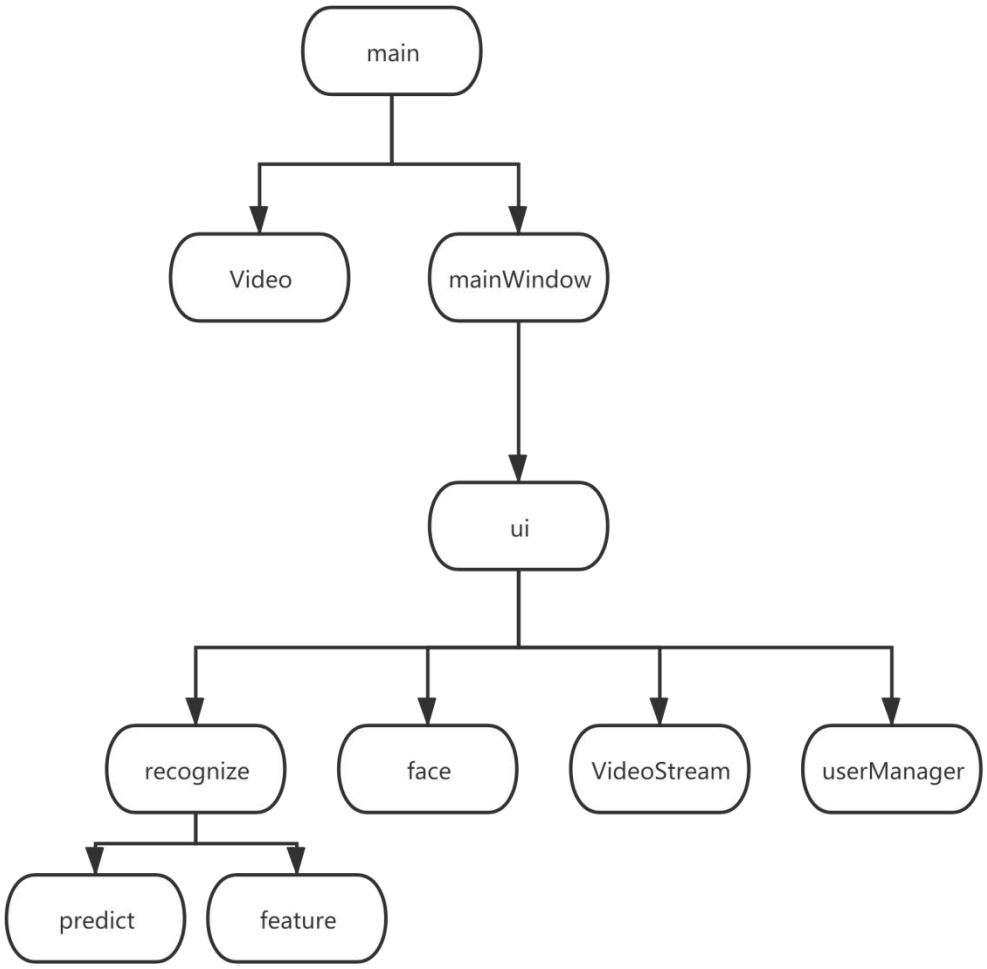


facerec 模块主要和识别的功能有关。faces 文件夹下存放了录入人脸

时录下的用户照片，分别以用户名命名；traning 文件夹下有存放用户信息的 CSV 文件；此外目录下还有对摄像头采集到的图像处理的实现 face.py，提供人脸识别接口的 recognize.py。

2. 代码调用关系

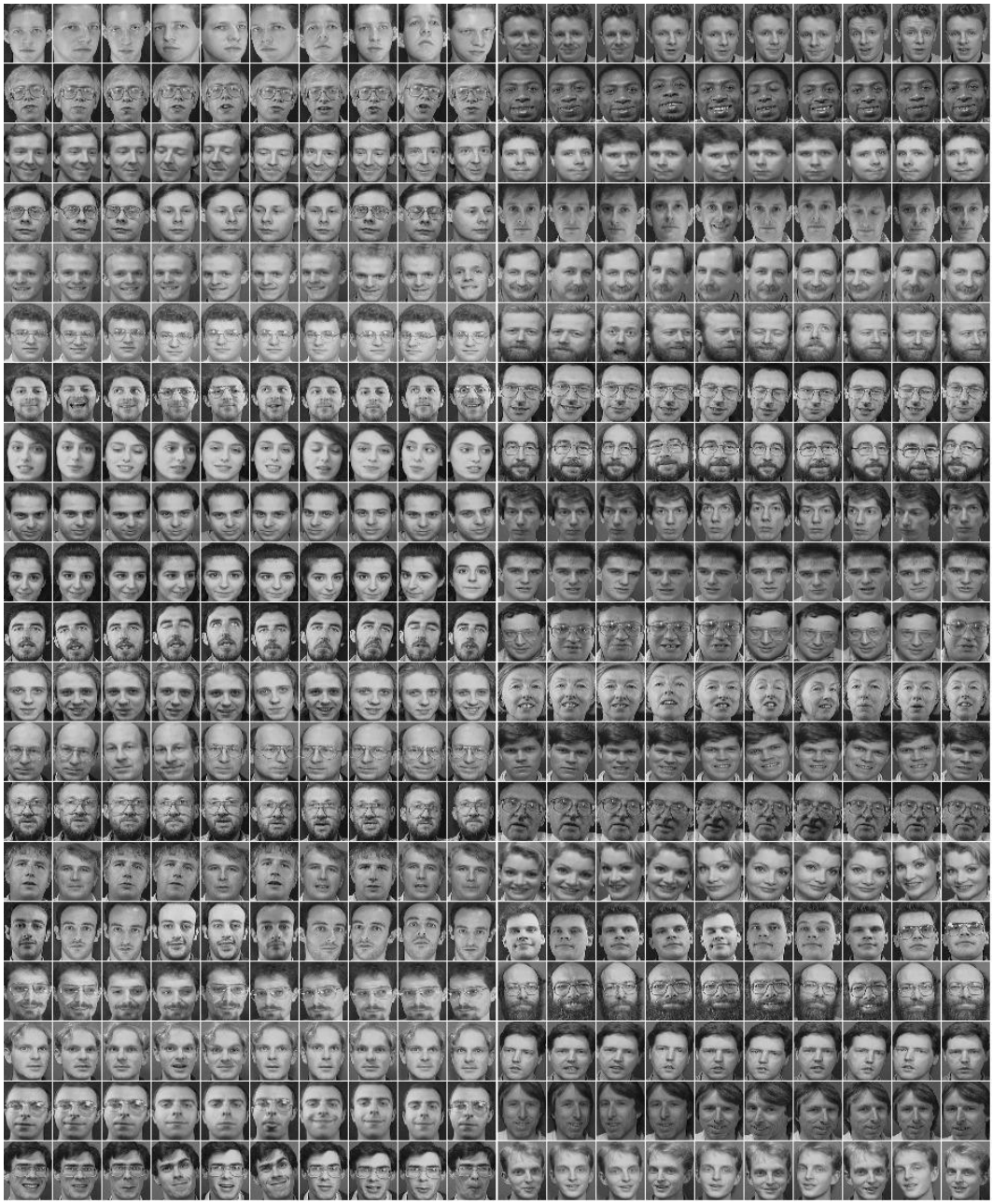
这里列出主要的代码文件调用关系，其他的文件，例如 config.py 为大多数文件所使用，不再列出。



可以看到，main 函数启动了 mainWindow，也就是主界面。主界面提供的按钮可以进入 ui 文件中的两个界面，在这里完成了软件的主要功能。

3. 训练数据结构

本次的训练数据来源于开源数据集 Olivetti faces，获取链接：https://cs.nyu.edu/~roweis/data.html。图片展示如下：



这个开源数据集共有来自 40 个人的 400 张图片，每个人 10 张图片。当然这是个较小的数据集，所以使用数据增强技术将提高网络的鲁棒性。

4. 数据的处理

数据处理由 data_generator.py 下的 DataGenerator 类提供，实现细节如下。

读入图片

首先读入图片

```
olive = cv2.imread(path)
```

分割图片

然后是分割图片，由于我们的数据集共有 20 行，每行有两个人的 20 张图片，每个人 10 张。这个可以在上一小节的图片示例中看出。那么我们需要做的就是将对应像素位置的图片分割出来，每行每列遍历分割即可。分割后的图片将与对应人的 label 一起存储下来，为之后网络的训练做准备。

```
for row in range(20):#20 行
for column in range(20):#20 列
    img = olive[row * 57:(row + 1) * 57, column * 47:(column + 1) * 47]
    data.append((img, label))#label 是 0-39
    if count % 10 == 0 and count != 0:#每过 10 列，就是下一张人脸
        label += 1
    count += 1
```

数据增强

最后是数据增强，我们可以将所有图片进行随机的裁剪与随机角度的旋转，重复几遍即可得到大量数据。然后将这些数据都保存起来。

```
for i in range(0, expand_size):#扩充 expan_size 倍
    index = i % data_size
    # 确保不会超过 data_size 的大小,然后在 data List 中循环遍历
    crop_img = self._do_random_crop(data[index][0])
    expand_list.append((crop_img, data[index][1]))
    rotate_img = self._do_random_rotation(data[index][0])
    expand_list.append((rotate_img, data[index][1]))
     data.extend(expand_list)
```

数据打乱

最后，让我们把数据打乱。

```
random.shuffle(data)
```

5. 网络结构的实现

网络结构的实现在 cnn_models.py 中，实现细节如下。

resnet 基本结构 basicblock

首先是 x1 输入进入一个卷积层得到 x2，然后 x2 经过激活函数 relu 得到 x3，然后 x3 再次进入一个卷积层得到 x4，然后将 x4 与 x1 相加后得到 x5，然后将 x5 经过激活函数 relu 得到输出 y。结构如下所示。

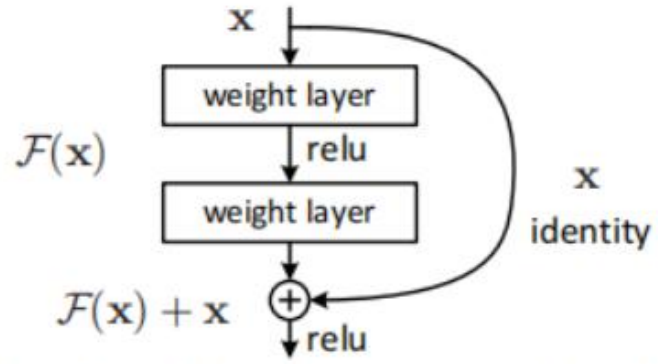


Figure 2. Residual learning: a building block.

第一个 weight layer

y = ZeroPadding2D(padding=1)(x)

y= Conv2D(filters, kernel_size, strides=stride, kernel_initializer='he_normal')(y)

y = BatchNormalization()(y)

y = Activation("relu")(y)

第二个 weight layer

y = ZeroPadding2D(padding=1)(y)

y = Conv2D(filters, kernel_size, kernel_initializer='he_normal')(y)

y = BatchNormalization()(y)

其中将 x4 与 x1 相加的步骤被称为残差连接，这也是为什么这种网络被叫做残差网络。残差连接的好处在于，将前面的输出张量与后面的输出张量相加，从而将前面的表示重新注入下游数据流中，这有助于防止信息处理流程中的信息损失。

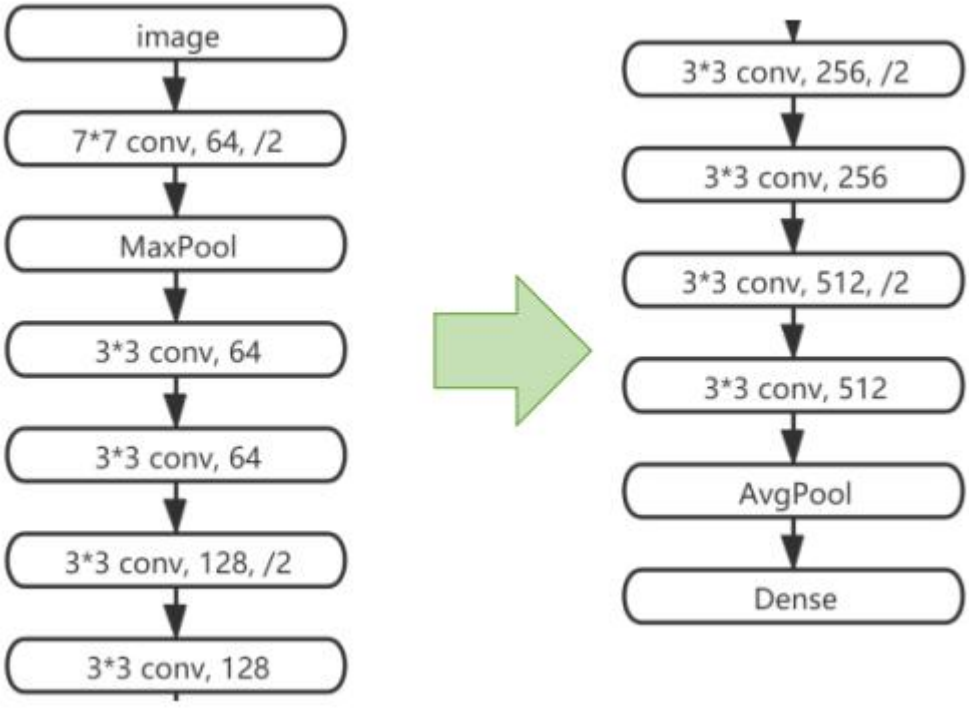
残差连接

if is_first_block:#不同的 filters 之间调整大小

```
shortcut=Conv2D(filters,kernel_size=1,strides=stride,kernel_initializer='he_normal')(x)
shortcut = BatchNormalization()(shortcut)
else:
shortcut = x
y = Add()([y, shortcut])
```

同时，这次的实现中采用了批标准化的技术。即使在训练过程中均值和方差随时间发生变化，批标准化也可以适应性地将数据标准化。批标准化的工作原理是，训练过程中在内部保存已读取每批数据均值和方差的指数移动平均值。批标准化的主要效果是，它有助于梯度传播（这一点和残差连接很像），因此允许更深的网络。

总体网络结构



网络结构如上图，由于过长，分成了两段。相比传统的 resnet18，本设计的网络结构为 5 层。残差结构并未显示在图中。

6. 训练代码实现

数据生成器

首先根据网络的参数生成数据生成器

generator=DataGenerator(dataset="olivettifaces",path="./data/olivetti_faces/olivettifaces.jpg",batch_size=batch_size,input_size=input_shape,is_shuffle=True,data_augmentation=10,validation_split=.2)

编译设置模型

然后设置模型参数

model=wrap_cnn(model=eval(cnn),feature_layer="feature",input_shape=input_shape,num_classes=num_classes)model.compile(optimizer='adam',loss=amsoftmax_loss,metrics=['accuracy'])#模型配置

设置早停

设置回调，动态调整训练过程中的参数，当被监测的数量不再提升，则停止训练。

early_stop = EarlyStopping('loss', 0.1, patience=patience)

monitor: 被监测的数据。

min_delta: 在被监测的数据中被认为是提升的最小变化， 例如，小于 min_delta 的绝对变化会被认为没有提升。

patience: 没有进步的训练轮数，在这之后训练就会被停止。

调整学习速率

当标准评估停止提升时，降低学习速率。当学习停止时，模型总是会受益于降低 2-10 倍的学习速率。 这个回调函数监测一个数据并且当这个数据在一定「有耐心」的训练轮之后还没有进步， 那么学习速率就会被降低。

```
reduce_lr=ReduceLROnPlateau('loss',factor=0.1,patience=int(patience/2),
verbose=1)
```

monitor: 被监测的数据。

factor: 学习速率被降低的因数。新的学习速率 = 学习速率 * 因数

patience: 没有进步的训练轮数，在这之后训练速率会被降低。

verbose: 整数。0：安静，1：更新信息。

断点保存

在每个训练期之后保存模型。

```
model_checkpoint=ModelCheckpoint(model_names,monitor='loss',
verbose=1,save_best_only=True,save_weights_only=False)
```

filepath 可以包括命名格式选项，可以由 epoch 的值和 logs 的键（由 on_epoch_end 参数传递）来填充。例如：如果 filepath 是 weights.{epoch:02d}-{val_loss:.2f}.hdf5， 那么模型被保存的的文件名就会有训练轮数和验证损失。

filepath: 字符串，保存模型的路径。

monitor: 被监测的数据。

verbose: 详细信息模式，0 或者 1 。

save_best_only: 如果 save_best_only=True， 被监测数据的最佳模型就不会被覆盖。

save_weights_only: 如果 True，那么只有模型的权重会被保存(model.save_weights(filepath))， 否则的话，整个模型会被保存(model.save(filepath))。

训练模型

使用 Python 生成器（或 Sequence 实例）逐批生成的数据，按批次训练模型。

model.fit_generator(generator=generator.flow('train'),steps_per_epoch=int(training_set_size/batch_size),epochs=num_epochs,verbose=1,callbacks=callbacks,validation_data=generator.flow('validate'),validation_steps=int(validation_set_size) / batch_size)#模型训练

generator: 一个生成器

steps_per_epoch: 在声明一个 epoch 完成并开始下一个 epoch 之前从 generator 产生的总步数（批次样本）。 它通常应该等于你的数据集的样本数量除以批量大小。

epochs: 训练模型的迭代总轮数。一个 epoch 是对所提供的整个数据的一轮迭代，如 steps_per_epoch 所定义。

verbose: 0, 1 或 2。日志显示模式。 0 = 安静模式, 1 = 进度条, 2 = 每轮一行。

callbacks: keras.callbacks.Callback 实例的列表。在训练时调用的一系列回调函数。

validation_data: 验证数据

validation_steps: 仅当 validation_data 是一个生成器时才可用。在停止前 generator 生成的总步数（样本批数）。

7. 主界面设计

主界面的实现由 mainwindow.py 中的 Ui_MainWindow 类完成。其采用垂直排布，其主要由四个部分组成：最上方的日期 label，第二行的时间计时器，第三行的信息 label，第四行的三个按钮。这四个部分是挨个加入垂直排布布局的。

```
self.verticalLayout = QtGui.QVBoxLayout(self.centralwidget)
self.verticalLayout.addWidget(self.label_date)
self.verticalLayout.addWidget(self.lcd_time)
self.verticalLayout.addWidget(self.label_welcome)
self.verticalLayout.addLayout(self.gridLayout_buttons)
```

girdLayout_buttons 是三个按钮的布局，同样是一个垂直排布，也是挨个加入的三个按钮。然后这整个的排布呈现一个 1：2：4：1 的比例。同时加上了一个背景图片，来源于中南大学 D 座后的风景。

```
palette = QPalette()
pix=QPixmap("ui/background2.jpg")
```

palette.setBrush(QPalette.Background, QBrush(pix))

MainWindow.setPalette(palette)

这个主界面的设计效果可以参考本报告第六章“实验现象”部分。

其次，三个按钮通过信号与槽函数连接到执行函数：

self.btn_face.clicked.connect(self.btn_face_clicked)

self.btn_register.clicked.connect(self.btn_register_clicked)

self.btn_exit.clicked.connect(self.btn_exit_clicked)

这三个执行函数在按钮被点击时执行。其中 btn_face 被点击时将启用人脸识别界面，btn_register 被点击时将启用人脸录入界面，btn_exit 被点击时将退出系统。

8. 人脸识别界面设计

人脸识别界面由 ui.py 中的 FaceRec 类完成。这个界面主要由上方的信息 lable，中间的 VideoFrame，下面的信息 label 和最下方的一个按钮组成。其位置由函数设置。

self.video_frame.setGeometry(QtCore.QRect(250, 100, 480, 360))

self.pushButton_back.setGeometry(QtCore.QRect(450, 600, 80, 50))

self.label_title.setGeometry(QtCore.QRect(450, 10, 100, 50))

self.label_info.setGeometry(QtCore.QRect(350, 500, 300, 50))

下方的按钮通过信号与槽函数连接到执行函数：

self.pushButton_back.clicked.connect(self.pushButton_back_clicked)

按钮将在被点击时执行，这个按钮执行的是返回功能，即被点击时返回到主界面。

9. 人脸录入界面设计

人脸录入界面由 ui.py 中的 FaceRegister 类实现。这个界面主要由上方的信息 label，中间的 VideoFrame，下方的两个按钮，以及一个在点击后才会出现的进度条组成。其位置由函数手动设定。

```
self.label_title.setGeometry(QtCore.QRect(450, 50, 100, 60))
self.progressBar.setGeometry(QtCore.QRect(300, 700, 440, 60))
self.video_frame.setGeometry(QtCore.QRect(250, 100, 480, 360))
self.pushButton_capture.setGeometry(QtCore.QRect(450, 500, 100, 60))
self.pushButton_back.setGeometry(QtCore.QRect(450, 600, 100, 60))
```

下方的按钮通过信号与槽函数连接到执行函数：

```
self.pushButton_capture.clicked.connect(self.pushButton_capture_clicked)
self.pushButton_back.clicked.connect(self.pushButton_back_clicked)
```

其中 pushButton_capture 被点击时，将会唤起录入功能。pushButton_back 被点击时将会返回主界面。

10. 界面与网络调用接口

这个接口由 predict.py 中的 model_predict 函数提供，这个函数输入一个函数句柄，一个待匹配的人脸图像。返回匹配到的用户 id 与匹配置信度（如果匹配成功的话）。下面给出该函数完成功能的核心步骤：

首先令待匹配的人脸图片输入网络，取得网络的输出作为特征值

```
base_feature=get_feature(template_image)
```

然后从存放用户照片的目录下搜寻匹配的用户

```
for root, dirs, files in os.walk(data_path):
    for name in dirs:
        path_faces = os.path.join(root, name)
        for file in os.listdir(path_faces):
            path_face=os.path.join(path_faces, file)
```

上面的代码搜寻了存放图片的用户目录，获得了某个用户的人脸照片的路径存入 path_face。接下来将读取用户的人脸照片

```
face = cv2.imread(path_face)
```

然后将用户的人脸图片输入网络，获得网络的输出作为特征值

```
face_feature = get_feature(face)
```

然后计算待匹配人脸的特征值与用户人脸的特征值的相似度

```
similarity = cosine_similarity(base_feature, face_feature)
```

若置信度大于阈值，则认为该待匹配的人脸图片与用户的人脸图片来自于同一人，赶回这个用户的 id 号与匹配置信度。

```
if similarity > threshold:
    label = manager.getUserByName(name)['id']
    return label, similarity
```

若置信度未超过阈值，则接着搜索下一个用户，看是否匹配。若搜索完都未找到匹配的用户，则返回空值。

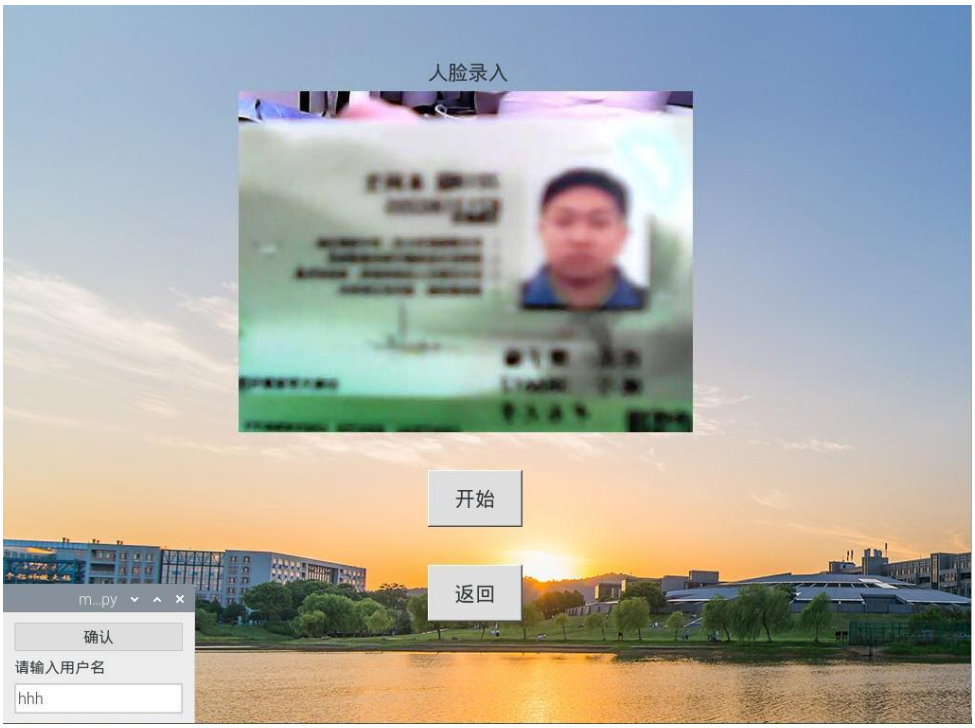
```
return None
```

六、实验现象

人脸识别系统的主界面，包括三个按钮，分别是：人脸识别，人脸录入，退出系统。

点击人脸录入按钮后出现录入的界面，中央的是摄像头画面，在这里用中南大学某学生校园卡上的头像照片做演示。

点击开始后，提示输入用户名，我们输入“hhh”。



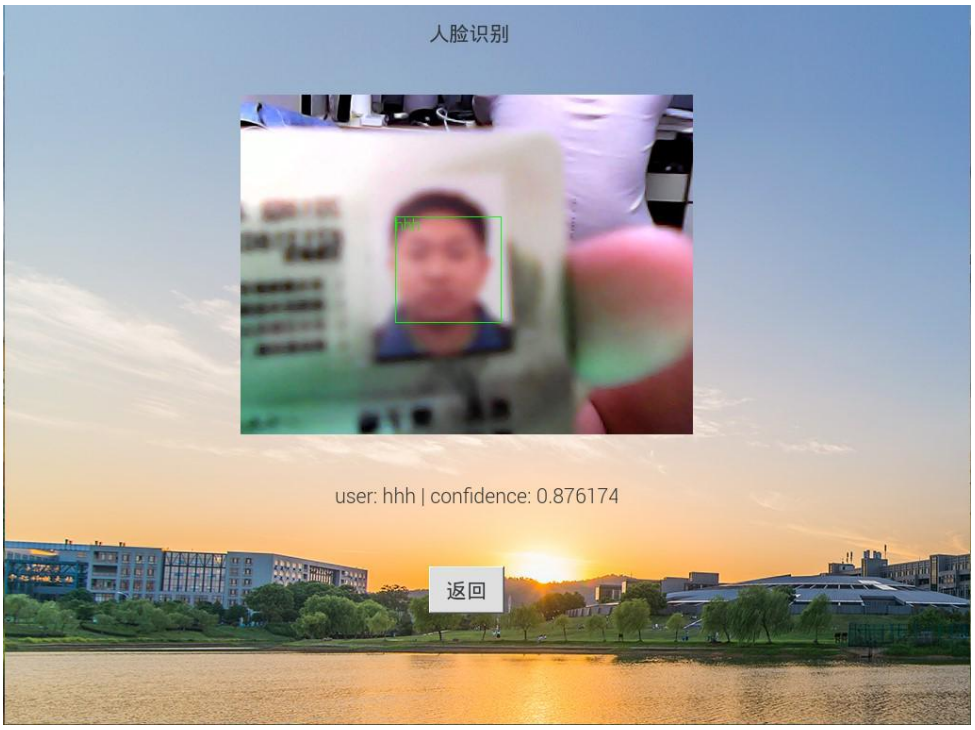
点击确认后，系统开始截取人脸图片传到后台并保存，下方显示进度条。

截取结束后，出现删除界面，可以将质量不高的图片进行删除。

再点击录入后，系统提示“人脸录入应完成”，点击 ok 即可完成录入。

再点击下方的返回按钮回到主界面后，点击人脸识别按钮，进入人脸识别界面。此时中央显示摄像头画面，上面用绿色方框圈出人脸，并显示用户名“hhh”。下方同时

显示用户名，以及识别的置信度。



七、课设收获与感想

这次课设源于我的群智感知课程，我一直对深度学习的图像处理能力深深着迷，觉得卷积网络的设计巧夺天工。同时能够达到上千层的残差网络更是真正的深度学习，不同于往常的十几层的网络（例如VGG之类的），残差网络特殊的残差结构完美的解决了梯度消失的问题。于是趁这个课设的机会，我实现了残差网络用于人脸识别。

这个课设的界面是用PyQt画的，我以往就有用PyQt画界面的经验，所以界面部分并不费力，而且也不是本次课设的重点，所以在本报告中也没有详细的展示，有兴趣的话我可以当面讲述这个界面的部分。这里给出Qt的官方帮助文档地址：https://www.kancloud.cn/apachecn/pyqt4-doc-zh/1947160

网络的实现利用了Keras框架，这个框架的使用非常简便，只需调用接口即可轻松实现卷积层等复杂的设计。我之前读过《Python深度学习》，这本书是Keras之父写的，内容主要是深度学习的介绍与Keras的使用。这本书避开了神经网络底层的繁琐数学公式，只介绍不同网络的特点。这里给出Keras的中文帮助文档地址：https://keras.io/zh/

这个项目的全部代码实在是太多，从界面到网络的实现，还有数据处理等等，因此就不一一贴在报告中。我将关于深度学习部分的代码贴在了本报告的第五节，同时在旁边写了丰富的注释，可以毫不费力的看懂。为了方便获取，我将其上传到了gitee上，链接：https://gitee.com/longtimeno-see/deep-l-face-rec.git

关于人脸识别部分的实现利用了特征的比较，这个和鲁老师上课说过的**“模式就蕴含在向量之中”、“模式就是网络学习到的知识”、“网络学习的就是向量之间的相似度，向量的夹角代表了不同向量间的相似度”**不谋而合。我训练的残差网络做的是分类任务，将数据集中的不同人脸进行分类。那么用在人脸识别上很自然的想法是将录入的用户照片也放入网络进行训练，但是我们的代码运行在树莓派上，计算能力较弱。若用户进行一次录入就要进行一次训练，那么时间的花费是不可接受的。想一想就很能明白，假如作为用户的你点击按钮录入的自己的头像，然后再点击人脸识别的按钮，希望软件能识别出你。可软件告诉你说：不好意思，我还在训练，你得等几个小时。这毫无疑问是无法接受的时间延迟。所以我采用的方法是：将用户录入的图像保存到本地，并不用于训练网络。而是**在用户进行人脸识别的时候，将捕捉到的图像输入到网络获得输出向量作为特征值。然后遍历本地所有用户的头像输入到网络获取特征值，将两种特征值利用 cosine 值进行比较，相似度大于阈值的则视为是同一个人。**这个想法毫无疑问与鲁老师上课说到的模式与向量相似度的内容是吻合的。

报告到这里也该结束了，这也意味着一学期群智感知课程的结束。然而课程会结束，我学到的知识与能力却会跟随我终身。非常感谢鲁老师的悉心教诲与督促。老师采用的上课模式自由而充满创新，与我们学生的互动贯穿课程，起到了极好的引导作用。再次感谢鲁老师一学期的陪伴，您的教导学生定会牢记。