

基于卷积神经网络的
实时人脸识别与表情监测系统
V1.0

系统原理与程序设计说明书

目录

一、 引言	1
1.1 人脸识别发展现状	1
1.2 开发背景及意义	1
二、 基本原理	1
2.1 数据输入层	3
2.2 卷积计算层	3
2.3 ReLU 激励层	5
2.4 池化层:	6
2.5 全连接层:	8
三、 系统设计与实现	10
3.1 总体实现框架	10
3.2 人脸数据集处理	11
3.4 人脸识别分类器设计与模型训练	15
3.5 表情识别分类器设计与模型训练	20
3.6 图形化界面设计与项目完善	27
3.7 重点开发工作	32
四、 软件运行展示与分析	42
五、 总结	48

一、引言

1.1 人脸识别发展现状

人脸识别，是基于人脸特征进行身份识别的一种生物识别技术。主要采用摄像头或者屏幕视频流采集人脸信息，并自动跟踪人脸进行识别。

近年来，随着人工智能的发展以及国家经济发展、安全防卫的需要，我国人脸识别市场不断扩大，技术水平不断提升，在算法方面已取得世界领先地位。伴随着人工智能的持续发展，智能化时代的悄然到来，以人脸识别为代表的生物识别技术越来越普及。从安防、支付、金融到教育、医疗和交通，“刷脸”日渐成为常态，为人们的生产与生活带来了诸多智能、安全与便捷。

1.2 开发背景及意义

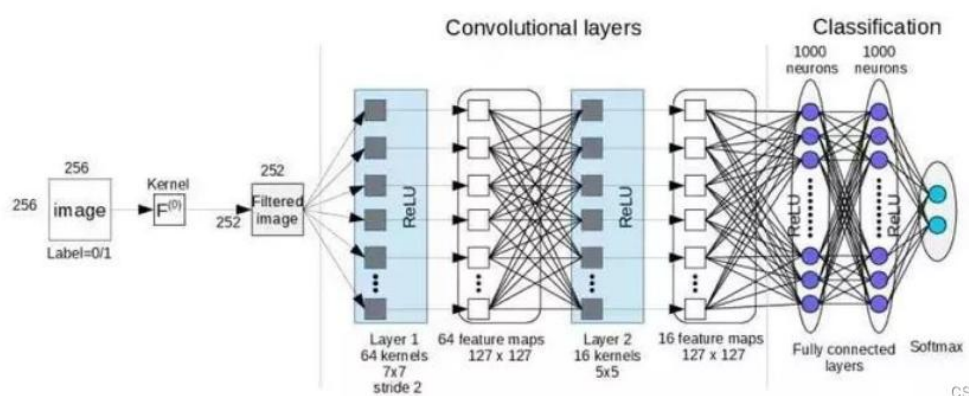
随着社会的发展，科技的进步，生活节奏也逐渐加快，这时候就会暴露出许多问题，同时也有越来越多的人患上了抑郁症。于是“基于卷积神经网络的实时人脸识别和表监测系统”也就应运而生。本系统通过实时监控并采集患者的面部特征信息，进行有效的训练后可以起到识别人像和人物表情的功能，如此方便患者的监测和治疗。监控画面一但出现了 sad 等消极情绪，立马就找出对应患者并对其进行医治。如此一来，方便了医院的实时监控，也就加快了治疗的进程，利于患者康复，也会防止患者出现过激行为伤害自己和他人。

二、基本原理

卷积神经网络(CNN)是一种前馈神经网络，它包括卷积计算并具有较深的结构，因此是深度学习的代表性算法之一。随着科技的不断进步，人们在研究人脑

组织时受 启发创立了神经网络。神经网络由很多相互联系的神经元组成，并且可以在不同的神经元之间通过调整传递彼此之间联系的权重系数 x 来增强或抑制信号。简而言之，卷积神经网络（Convolutional Neural Networks）是一种深度学习模型或类似于人工神经网络的多层感知器，常用来分析视觉图像。

卷积神经网络的架构：



如上图所示，卷积神经网络架构与常规人工神经网络架构非常相似，特别是在网络的最后一层，即全连接。此外，还注意到卷积神经网络能够接受多个特征图作为输入，而不是向量。

卷积网络的层级结构：

一个卷积神经网络主要由以下 5 层组成：

- 数据输入层/ Input layer
- 卷积计算层/ CONV layer
- ReLU 激励层 / ReLU layer
- 池化层 / Pooling layer
- 全连接层 / FC layer

2.1 数据输入层：

该层要做的处理主要是对原始图像数据进行预处理，其中包括：

去均值：

把输入数据各个维度都中心化化为 0，如下图所示，其目的就是把样本的中心拉回到坐标系原点上。

归一化：

幅度归一化到同样的范围，如下所示，即减少各维度数据取值范围的差异而带来的干扰，比如，有两个维度的特征 A 和 B，A 范围是 0 到 10，而 B 范围是 0 到 10000，如果直接使用这两个特征是有问题的，好的做法就是归一化，即 A 和 B 的数据都变为 0 到 1 的范围。

PCA/白化：

用 PCA 降维；白化是对数据各个特征轴上的幅度归一化。

2.2 卷积计算层：

这一层就是卷积神经网络最重要的一个层次，也是“卷积神经网络”的名字来源。在这个卷积层，有两个关键操作：

1. 局部关联，每个神经元看做一个滤波器(filter)
2. 窗口(receptive field)滑动，filter 对局部数据计算

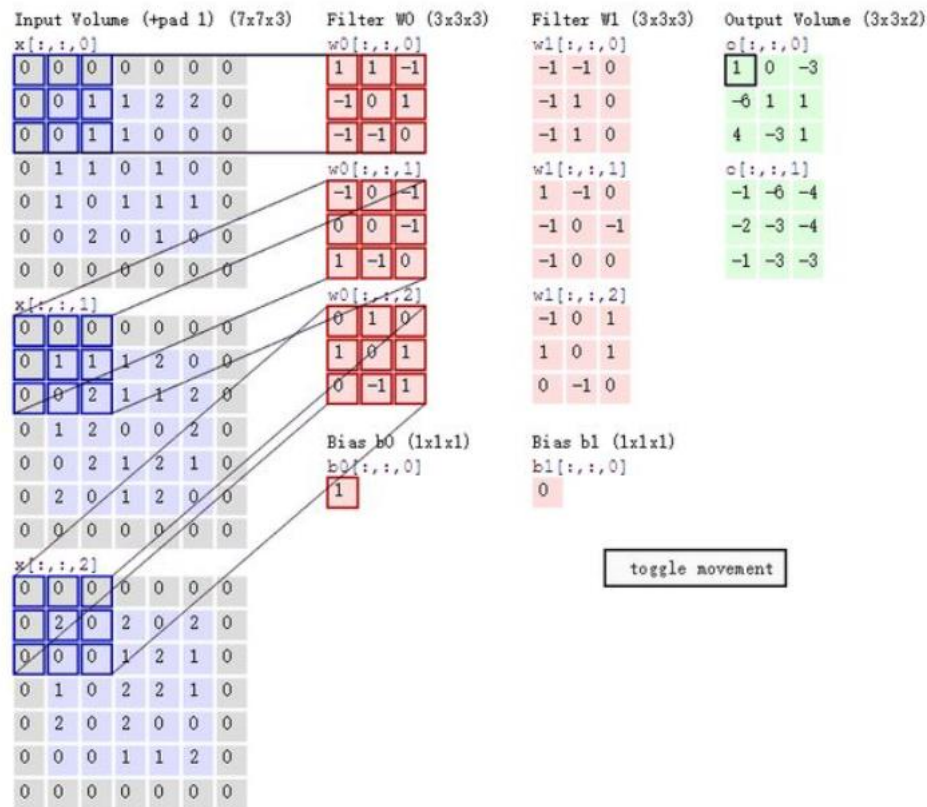
这一部分的原理也可以根据 python 中卷积层计算所用函数辅助解释，在 torch 库中的 Conv2d 函数中有五个参数，分别 in_channels,out_channels,kernel_size, stride, padding，其中要改变卷积层中每一层的行为，有两个主要参数是可以调整的。选择了过滤器的尺寸以后，还需要选择步幅（stride）和填充（padding）。

步幅控制着过滤器围绕输入内容进行卷积计算的方式。Stride=1 时，过滤器通过每次移动一个单元的方式对输入内容进行卷积。过滤器移动的距离就是步幅。步幅的设置通常要确保输出内容是一个整数而非分数。举一个例子，想象一个 7×7 的输入图像，一个 3×3 过滤器（简单起见不考虑第三个维度），步幅为 1。则输出就会变成 5×5 的图像，当 stride=2 时，输出又会变成 3×3 的图像。

所以移动了两个单元，输出内容会减小。注意，如果试图把步幅设置成 3，那就会难以调节间距并确保感受与输入图像匹配。正常情况下，程序员如果想让接受域重叠得更少并且想要更小的空间维度（spatial dimensions）时，会增加步幅。

当想要输出与输入同样大小是，padding 参数，也就是填充值，就会发挥它的作用，比如当你把 $5 \times 5 \times 3$ 的过滤器用在 $32 \times 32 \times 3$ 的输入上时，此时希望输出与输入大小相同，可以对这个层应用大小为 2 的零填充（zero padding）。零填充在输入内容的边界周围补充零。如果用两个零填充，就会得到一个 $36 \times 36 \times 3$ 的输入。

卷积的计算：



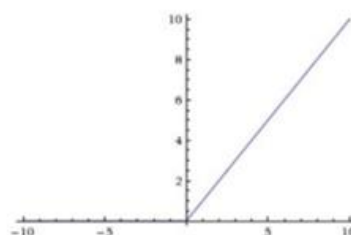
卷积层由几个卷积单元组成。每个卷积单元的参数可以通过反向传播算法进行优化，其目的是提取输入的各种特征，但是卷积层的第一层只能提取低级特征，例如边、线和角。更多层的可以提取更高级的特征，卷积层到一定数量可以对人脸面部图像进行特征提取。一般卷积层结构卷积层可以包含多个卷积面，并且每个卷积面都与一个卷积核相关联。每次执行卷积层计算时，都会生成与之相关的多个权重参数，这些权重参数的数量与卷积层的数量相关，即与卷积层所用的函数有直接的关系。

2.3 ReLU 激励层：

操作：把卷积层输出结果做非线性映射。

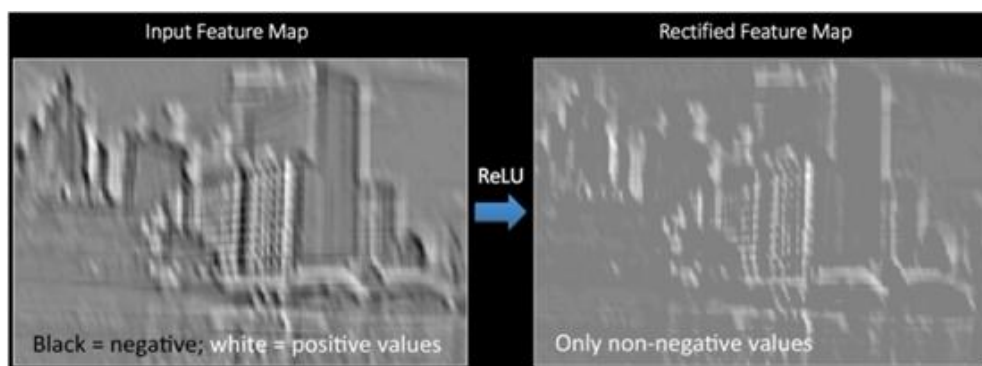
CNN 采用的激活函数一般为 ReLU(The Rectified Linear Unit/修正线性单元)，它的特点是收敛快，求梯度简单，但较脆弱，图像如下：

$$\text{Output} = \text{Max}(\text{zero}, \text{Input})$$



ReLU 是一个基于元素的操作(应用于每个像素), 它将 feature map 中的所有负像素值替换为 0。卷积是一个线性操作——对应位置上元素的矩阵乘法和加法操作,但是在学习训练时大多数的真实数据是非线性的, 所以考虑通过引入一个非线性函数如 ReLU, ReLU 的目的是在学习之前引入非线性这一概念。

从下图可以清楚地理解 ReLU 操作。它显示了将 ReLU 操作应用下图获得的一个特性映射。输出特征映射在这里也被称为“矫正”特征映射。也可以使用其他非线性函数, 如 tanh 或 sigmoid, 来代替 ReLU, 但是 ReLU 在大多数情况下性能更好。



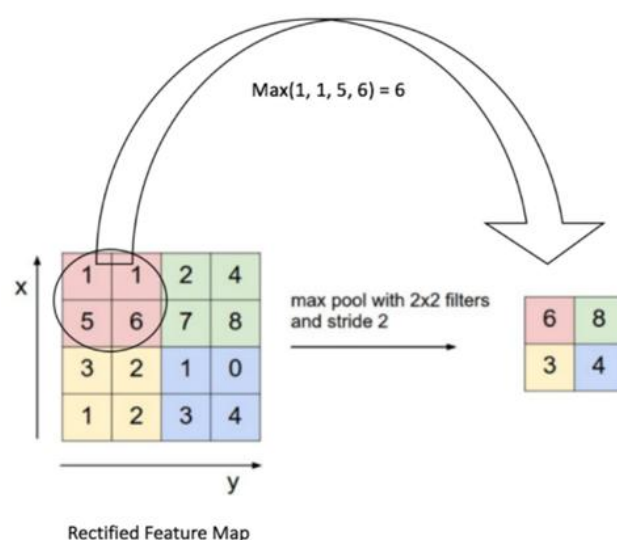
2.4 池化层:

空间池化(也称为子采样或下采样)降低了每个特征图的维数, 但保留了最重要的信息。空间池化可以有不同的类型:Max、Average、Sum 等。

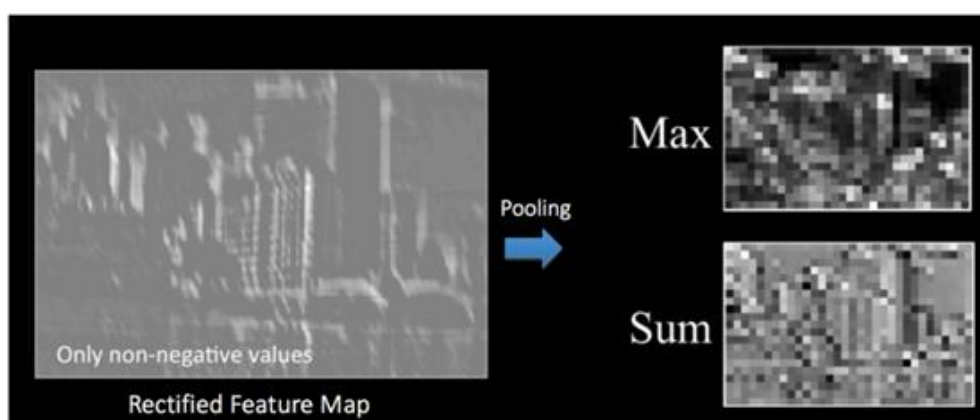
在最大池化的情况下, 定义一个空间邻域(例如, 一个 2×2 的窗口), 并从该窗口内的修正特征映射中取最大的元素。也可以取平均值(平均池)或该窗口中所

有元素的总和，而不是取最大的元素。在实践中，最大池化已经被证明可以更好地工作。

下图是一个使用 2×2 窗口对经过卷积+ ReLU 操作得到的修正后的 Feature map 进行最大池化操作的例子。



下图显示了池化对在上图中的 ReLU 操作后接收到的修正后的特征映射的影响。



在识别图像的过程中，有时会遇到较大的图像，此时希望减少训练参数的数量，这时需要引入池化层。池化层的特点包括：

1. 特征不变性，也就是在图像处理中经常提到的特征的尺度不变性，池化操作就是图像的 **resize**，平时一张狗的图像被缩小了一倍还能认出这是一张狗的照

片，这说明这张图像中仍保留着狗最重要的特征，人眼一立即就能判断图像中画的是一只狗，图像压缩时去掉的信息只是一些无关紧要的信息，而留下的信息则是具有尺度不变性的特征，是最能表达图像的特征。

2. 特征降维，一幅图像含有的信息是很大的，特征也很多，但是有些信息对于做图像任务时没有太多用途或者有重复，可以把这类冗余信息去除，把最重要的特征抽取出来，这也是池化操作的一大作用。

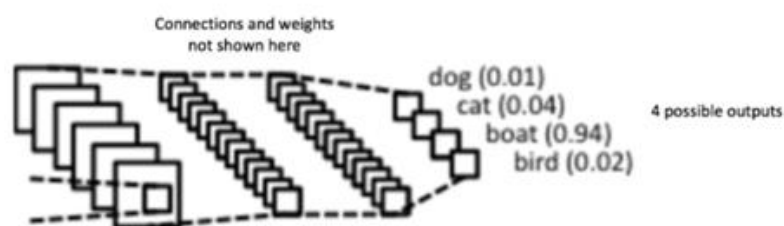
3. 在一定程度上防止过拟合，更方便优化。

简而言之，如果输入是图像的话，那么池化层的最主要作用就是压缩图像。

2.5 全连接层：

全连接层是一个传统的多层感知器，在输出层使用 softmax 激活函数（其他分类器如 SVM 也可以使用）。“全连接”意味着上一层的每个神经元都与下一层的每个神经元相连。

卷积层和池化层的输出表示输入图像的高级特征。全连接层的目的是利用这些特征根据训练数据集将输入图像分类成不同的类。例如，开始执行的图像分类任务有四种可能的输出，如卷积层和池化层的输出表示输入图像的高级特征。全连接层的目的是利用这些特征将训练数据集将输入图像分类成不同的类。例如，开始执行的图像分类任务有四种可能的输出，如下图：

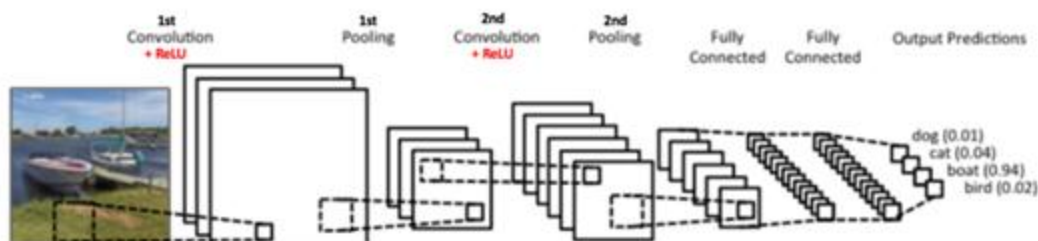


除了分类之外，添加一个完全连接的层也是学习这些特征的非线性组合的一种廉价方法。来自卷积层和池化层的大多数特性可能对分类任务很好，但这些特性的组合可能更好。

全连通层输出概率之和为 1。这是通过在全连接层的输出层使用 **Softmax** 作为激活函数来保证的。**Softmax** 函数取一个由任意实值分数组成的向量，并将其压缩为一个由 0 到 1 之间的和为 1 的值组成的向量。

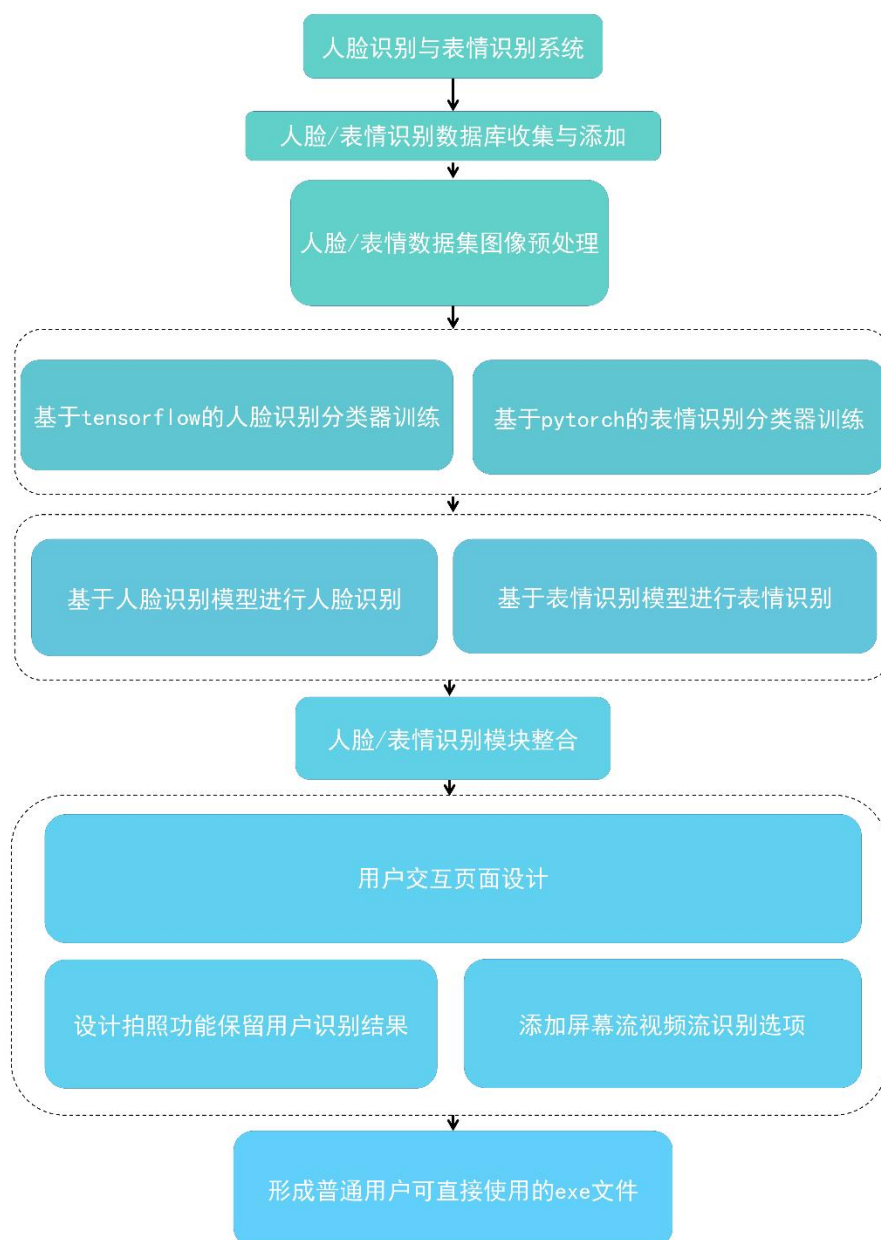
如下图所示，我们两组卷积、ReLU 和 Pooling 层-第 2 个卷积层使用 6 个过滤器对第 1 个 Pooling 层的输出进行卷积，总共产生 6 个 feature map。然后将 ReLU 分别应用于所有这六个特性映射。然后，对这六个修正后的特征映射分别执行最大池化操作。

这些层一起从图像中提取有用的特征，在网络中引入非线性，降低特征维数，同时使特征在缩放和转换时具有一定的等变性。



三、系统设计与实现

3.1 总体实现框架：



系统设计思路为：先分别设计人脸识别以及表情识别系统，保留二者的接口，再利用接口将两个模块拼接合并。

在分别设计两个模式识别模块时，采用了模式识别系统的流程化设计思想，即：

- ①先分别进行人脸以及表情数据库的收集，后续训练将会建立在此数据集上
- ②利用 `pytorch`、`tensorflow` 库分别搭建人脸识别、表情识别训练框架，并依托于采集到的数据集进行人脸识别与表情识别训练
- ③留存训练模型后，结合搭建好的卷积框架，实时对视频流以及屏幕流进行识别
- ④在接口处将两模块进行整合
- ⑤测试整体模块的性能是否合乎预期，能否同时对实时视频流进行表情和人脸识别
- ⑥将整合后的模块进行封装并设计 `ui`，实现用户不需要直接与程序交互，点击按键选择识别模式或拍照记录
- ⑦将程序封装为 `.exe` 文件，以便用户能在未安装相关支持库的情况下仍能正常运行程序。

3.2 人脸数据集处理：

人脸数据集获取：本系统的人脸数据集由测试团队的五个成员提供，通过运行 `face_add.py` 文件进行人脸采集。

调用库：`os`，`dlib`，`cv2`，`numpy`，`PIL`，`random`

`os`：创建人脸文件夹

`dlib`：检测人脸所在范围

`numpy`：将图像像素矩阵化，方便后续图像运算

PIL: ImageGrab 模块可用于将屏幕或剪贴板的内容复制到 PIL 图像存储器中。

cv2: 用于图像处理（包括灰度图像与彩色图像转化，图像三色通道的提取和组合以及图像的展示）

random: 生成随机数，用于改变采集图像的亮度和对比度

（1）摄像头采集过程：首先定义 `img_change` 函数，用于改变图像的亮度和对比度，然后使用 `dlib.get_frontal_face_detector` 函数来加载 `dlib` 自带的人脸检测器，若决定录入新的人像，则接着用 `os.path.exists` 函数和 `os.makedirs` 函数创建人脸图像文件夹用于装载图像，再然后用 `cap.read` 函数调取摄像头，如果是人像，则采集，并保存在，`num+1`，若不是人像，则 `num` 不变，不采集，采集数量达到 10000 张跳出循环，输出 ‘finished’，若不录入新的人像，则输出 ‘不录入，运行结束’。在采集过程中，使用了最开始定义的 `img_change` 函数，为了随机的改变对比度和亮度，以此增加样本的多样性。

（2）屏幕视频流采集过程：同摄像头采集一样，只是在 `cap.read` 函数调用的时候用 `ImageGrab.grab` 函数将屏幕上的图像复制到图像存储器，再进行后续处理（同摄像头采集）

（3）数据预处理：对于摄像头采集，用 `resize` 函数对图像进行裁剪，而对于屏幕流采集，则应先使用 `split` 函数和 `merge` 函数提取和重组 `rgb` 通道，再用 `resize` 函数对图像进行裁剪。

代码如下：

```
import cv2
import dlib
import os
import random
from PIL import ImageGrab
import numpy as np
```

```
faces_add_path = './data/faces_'
size = 64

"""改变图片的相关参数: 亮度与对比度"""
def img_change(img, light=1, bias=0):
    width = img.shape[1]
    height = img.shape[0]
    for i in range(0, width):
        for j in range(0, height):
            for k in range(3):
                tmp = int(img[j, i, k]*light + bias)
                if tmp > 255:
                    tmp = 255
                elif tmp < 0:
                    tmp = 0
                img[j,i,k] = tmp
    return img

"""特征提取器:dlib 自带的 frontal_face_detector"""
detector = dlib.get_frontal_face_detector()
# cap = cv2.VideoCapture(0)
BOX = (0, 0, 1100, 640)
num = 1
while True:
    print("是否录入新的人脸信息 (y or n) ? ")
    if input() == 'y':
        add_user_name = input("请输入你的姓名:")
        print("请看电脑摄像头")
        faces_add_path = faces_add_path + add_user_name
        if not os.path.exists(faces_add_path):
            os.makedirs(faces_add_path)
        while (num <= 10000):
            print('Being processed picture %s' % num)
            # success, img = cap.read()
            img = np.array(ImageGrab.grab(bbox=BOX))
            gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            """使用特征提取器进行人脸检测"""
```

```
dets = detector(gray_img, 1)
for i, d in enumerate(dets):
    x1 = d.top() if d.top() > 0 else 0
    y1 = d.bottom() if d.bottom() > 0 else 0
    x2 = d.left() if d.left() > 0 else 0
    y2 = d.right() if d.right() > 0 else 0
    face = img[x1:y1, x2:y2]
    face = cv2.resize(face, (size, size))
    b, g, r = cv2.split(face) # 分别提取 B、G、R 通道
    face = cv2.merge([r, g, b]) # 重新组合为 R、G、B
    cv2.imshow('image', face)
    cv2.imwrite(faces_add_path + '/' + str(num) + '.jpg', face)
    num += 1

key = cv2.waitKey(30)
if key == 27:
    break
else:
    print('Finished!')
    break
else :
    print("不录入,运行结束")
    break
```

表情数据集获取和处理:

本系统的表情数据库采用了 FER2013 数据库, 其数据集的下载地址如下:

<https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data>

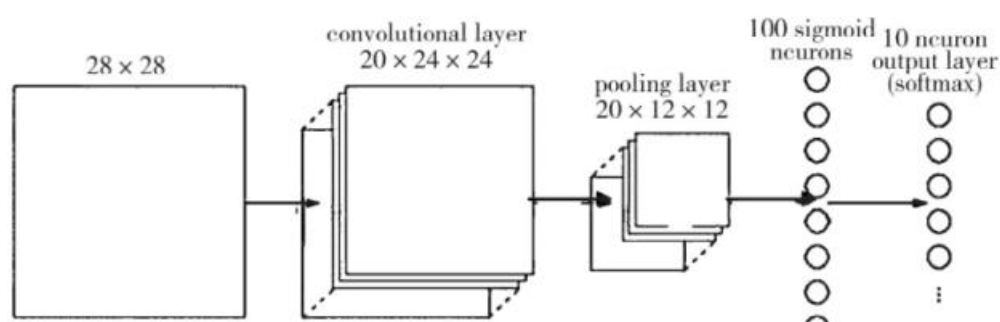
FER2013 数据集由 28709 张训练图, 3589 张公开测试图和 3589 张私有测试图组成。每一张图都是像素为 48*48 的灰度图。FER2013 数据库中一共有 7 中表情: 愤怒, 厌恶, 恐惧, 开心, 难过, 惊讶和中性。

3.3 特征提取:

在开发过程中使用了 `dlib` 函数库自带的 `frontal_face_detector()` 进行图片的特征提取, 通过 `detector=dlib.get_frontal_face_detector()` 获得脸部位置检测器, 再通过 `dets = detector(gray, 0)` 返回值是 `<class 'dlib.dlib.rectangle'>` (即一个矩形), 唯一表示这个人脸矩形框两个点坐标: 左上角 $(x1,y1)$ 、右下角 $(x2,y2)$ 来识别人脸, 需要注意的是 `detector` 中图像参数是灰度图, 要进行转换成灰度图的处理。

3.4 人脸识别分类器设计与模型训练

对于上一步的特征提取, 主要的作用是为了区分出人脸和非人脸的其他物体。将筛选出来的人脸送入分类器设计与模型训练的操作步骤之中, 进行进一步的人脸识别和表情识别的判断。对于人脸识别和表情识别均采用了卷积神经网络的方法分类器设计和模型训练。对于一个标准的卷积神经网络模型来说, 往往包含着五个主要部分, 即输入层、卷积层、池化层、全连接层和输出层, 如下图所示:



标准的卷积神经网络模型组成

对于该项目, 首先将经过 `dlib` 人脸识别分类器提取出的人脸照片进行尺寸的统一化进行数据备用。对于人脸识别的分类器设计和模型搭建部分, 利用 `tensorflow 1.13.1` 和 `keras 2.2.4` 框架进行操作。

```
from sklearn.model_selection import train_test_split

from keras.preprocessing.image import ImageDataGenerator

from keras.models import Sequential

from keras.layers import Dense, Dropout, Activation, Flatten

from keras.layers import Convolution2D, MaxPooling2D

from keras.optimizers import SGD

from keras.utils import np_utils

from keras.models import load_model

from keras import backend as K

import tensorflow
```

这是模型训练过程所需要的一些必要的 python 库。系统卷积神经网络模型在搭建和训练的主要步骤依次为：数据集读取与标签划分、CNN 模型定义及主要成分搭建、模型训练。该部分的所有代码均在 `faces_train.py` 文件中编写。

对于数据集读取和标签划分部分，在代码中定义一个 `Dataset` 类，在类中分别定义

```
def __init__(self, path_name)和 def load(self, img_rows=IMAGE_SIZE,
img_cols=IMAGE_SIZE,img_channels=3)函数。其中前者是将所需要的 images 和
labels 变量定义为空，便于后续操作进行直接赋值和引用。在 load 函数之中，传
入的参数是 dlib 人脸识别分类器所提取出的人脸图像，会将传入参数进行预处理
为 64*64 大小的图片。数据集为测试团队五位成员的人脸图像，分别以各自姓名
的命名方式存放在 data 文件夹之中，其中每个人采集了 20000 张的个人人脸图
片。开发过程中将每个图片分别提取出来，按照所属人脸进行标签编码。为了保
证后续模型训练的有效性，将数据集划分为训练集和验证集。用训练集训练模型，
用验证集检验模型训练效果并实现模型进一步优化。
```

对于 CNN 模型定义及主要成分搭建部分。借助 keras 框架。利用 Sequential

() 线性堆叠模型搭建网络结构, 在后续各层神经网络层的添加过程之中, 只需要使用 model.add() 方法即可实现。在卷积神经网络的主要组成之中, 卷积层、池化层、全连接层是重要部分。其中卷积层的作用是从输入的图片信息之中提取特征, 使用一个 3*3 的卷积模板与经过数组化处理之后图片进行卷积。池化层的作用是进行特征选择和信息过滤, 常见的主要是两种方法, 一种是 mean-pooling, 一种是 max-pooling。前者是取窗口中的均值作为处理结果, 后者则是取样本中最大值作为处理结果。在该项目中, 采用 max-pooling 方法对一小块区域取最大值, 用 2*2 大小的 pooling 窗进行池化降维, 减小复杂度。全连接层的作用是对提取的特征进行非线性组合以得到输出。但是对于层数较低的网络来说, 往往只能提取到一些低级的特征, 通过多层的网络堆叠则可以实现从低级特征中迭代提取更复杂特征的操作。具体的模型搭建代码框架如下:

```
def build_model(self, dataset, nb_classes=4):
    self.model = Sequential()
    self.model.add(Convolution2D(32, 3, 3, border_mode='same',
                                  input_shape=dataset.input_shape)) # 1 2 维卷积层
    self.model.add(Activation('relu')) # 2 激活函数层
    self.model.add(Convolution2D(32, 3, 3)) # 3 2 维卷积层
    self.model.add(Activation('relu')) # 4 激活函数层
    self.model.add(MaxPooling2D(pool_size=(2, 2))) # 5 池化层
    self.model.add(Dropout(0.25)) # 6 Dropout 层
    self.model.add(Convolution2D(64, 3, 3,
                                  border_mode='same')) # 7 2 维卷积层
    self.model.add(Activation('relu')) # 8 激活函数层
    self.model.add(Convolution2D(64, 3, 3)) # 9 2 维卷积层
    self.model.add(Activation('relu')) # 10 激活函数层
    self.model.add(MaxPooling2D(pool_size=(2, 2))) # 11 池化层
    self.model.add(Dropout(0.25)) # 12 Dropout 层
    self.model.add(Flatten()) # 13 Flatten 层
    self.model.add(Dense(512)) # 14 Dense 层, 又被称作全连接层
```

```

self.model.add(Activation('relu')) # 15 激活函数层
self.model.add(Dropout(0.5)) # 16 Dropout 层
self.model.add(Dense(nb_classes)) # 17 Dense 层
self.model.add(Activation('softmax')) # 18 分类层, 输出最终结果
self.model.summary()

```

对于模型训练部分, 采用 SGD+momentum 的优化器进行训练。相比于 SGD 优化器, 其在加入了动量之后, 参数更新可以保持之前的更新趋势, 而不会出现 SGD 优化器可能存在的梯度较小的情况。通过 SGD 优化器实现模型的学习优化, 并且采用 categorical_crossentropy 交叉熵损失函数作为来度量模型预测值与真实值之间的差异程度。由于训练数据集过于庞大, 故使用 model.fit_generator() 生成器训练模型, 利用生成器, 分批次向模型送入数据, 可以有效节省单次内存的消耗。每次送入 20 组数据进入模型训练, 总共训练 10 轮, 经过训练, 可以得到比较不错的准确率, 对于数据集的训练准确率可达 99%。

```

4487/4500 [=====>.] - ETA: 0s - loss: 0.0402 - acc: 0.9890
4490/4500 [=====>.] - ETA: 0s - loss: 0.0402 - acc: 0.9890
4493/4500 [=====>.] - ETA: 0s - loss: 0.0401 - acc: 0.9890
4496/4500 [=====>.] - ETA: 0s - loss: 0.0401 - acc: 0.9890
4499/4500 [=====>.] - ETA: 0s - loss: 0.0402 - acc: 0.9890
4500/4500 [=====] - 92s 21ms/step - loss: 0.0402 - acc: 0.9890 - val_loss: 0.0162 - val_acc: 0.9970

```

对于分类器设计和模型训练的优化方向实质上就是通过特征信息比对, 计算出给定输入分别属于 data 中各类数据集的情况的概率, 通过找到各个集合概率之中的最大值, 作为模型预测结果。

```

# 给出输入属于各个类别的概率
result_probability = self.model.predict_proba(image)

```

人脸识别预测:

通过模型训练, 完成了对于数据集的标签编码, 生成了一个储存着各类特征信息的模型结果, 即 aggregate.face.model.h5。在进行人脸识别预测的时候, 利用 opencv 通过摄像头直接读取实时摄像或者通过 PIL 库的 ImageGrab 进行读取桌面画面的方式采集图像, 获取到实时图像。在对于图像进行 dlib 人脸识别分类器

检测出人脸之后，对人脸图像进行预处理，将图像尺寸大小进行统一化为 64*64 的大小，送入模型进行计算出实时人脸图像所从属于各个模型之间的概率情况。

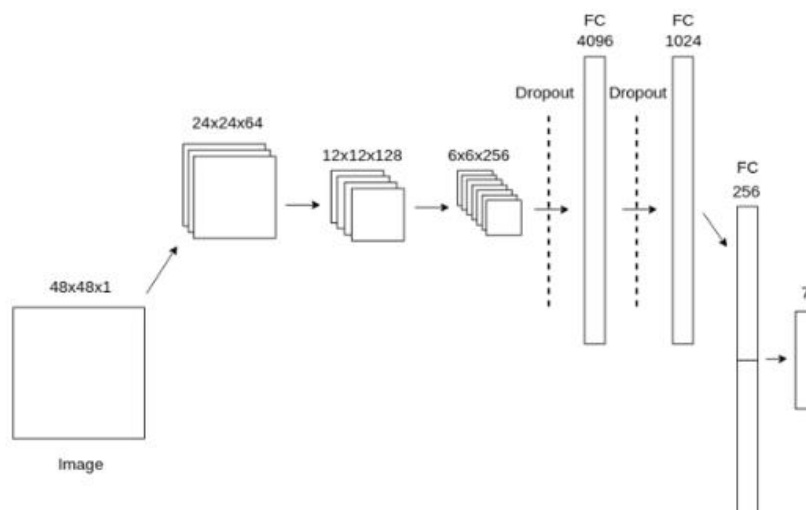
在传入参数之后，预测函数需要根据 keras 库要求的维度顺序对传入图像的数据进行重组后再带入模型计算。当模型预测结果中其可能从属于各个情况的概率之中的最大值大于 0.9 时，认为其是从属于已录入人脸的情况。而对于最大值概率仍小于 0.9 的情况，认为图像与所录入人脸的关联性均达不到可以认为是其本人的程度，将其统一归类为 faces_other 类输出预测结果。人脸预测的核心代码如下：

```
def face_predict(self, image):
    if K.image_dim_ordering() == 'th' and image.shape != (1, 3, IMAGE_SIZE,
IMAGE_SIZE):
        image = resize_image(image)
        image = image.reshape((1, 3, IMAGE_SIZE, IMAGE_SIZE))
    elif K.image_dim_ordering() == 'tf' and image.shape != (1, IMAGE_SIZE,
IMAGE_SIZE, 3):
        image = resize_image(image)
        image = image.reshape((1, IMAGE_SIZE, IMAGE_SIZE, 3))
        image = image.astype('float32')
        image /= 255
    result_probability = self.model.predict_proba(image)
    if max(result_probability[0]) >= 0.9:
        result = self.model.predict_classes(image)
        print('result:', result)
    return result[0]
    else:
        print('result:none')
        return -1
```

3.5 表情识别分类器设计与模型训练

该模型是基于 CNN 进行设计的，CNN 的原理先前已经简要介绍过，以下会通过代码围绕此模型进行更为详细的讲解：

系统采用模型的大致框架如下图所示：



从上图可以看出，在 Model 的卷积部分，输入图片 shape 为 $48 \times 48 \times 1$ ，经过一个 $3 \times 3 \times 64$ 卷积核的卷积操作，再进行一次 2×2 的池化，得到一个 $24 \times 24 \times 64$ 的 feature map 1（以上卷积和池化操作的步长均为 1，每次卷积前的 padding 为 1，下同）。将 feature map 1 经过一个 $3 \times 3 \times 128$ 卷积核的卷积操作，再进行一次 2×2 的池化，得到一个 $12 \times 12 \times 128$ 的 feature map 2。将 feature map 2 经过一个 $3 \times 3 \times 256$ 卷积核的卷积操作，再进行一次 2×2 的池化，得到一个 $6 \times 6 \times 256$ 的 feature map 3。卷积完毕，数据即将进入全连接层。进入全连接层之前，要进行数据扁平化，将 feature map 3 拉一个成长度为 $6 \times 6 \times 256 = 9216$ 的一维 tensor。随后数据经过 dropout 后被送进一层含有 4096 个神经元的隐层，再次经过 dropout 后被送进一层含有 1024 个神经元的隐层，之后经过一层含 256 个神经元的隐层，最终经过含有 7 个神经元的输出层。一般再输出层后都会加上 softmax 层，取概率最高的类别为分类结果。

接着，可以通过继承 `nn.Module` 来定义自己的模型类。需要注意的是，在代码中，数据经过最后含 7 个神经元的线性层后就直接输出了，并没有经过 `softmax` 层。这是为什么呢？其实这和 `Pytorch` 在这一块的设计机制有关。因为在实际应用中，`softmax` 层常常和交叉熵这种损失函数联合使用，因此 `Pytorch` 在设计时，就将 `softmax` 运算集成到了交叉熵 `CrossEntropyLoss()` 内部，如果使用交叉熵作为损失函数，就默认在计算损失函数前自动进行 `softmax` 操作，不需要额外加 `softmax` 层。`Tensorflow` 也有类似的机制。

有了模型，就可以通过数据的前向传播和误差的反向传播来训练模型了。在此之前，还需要指定优化器（即学习率更新的方式）、损失函数以及训练轮数、学习率等超参数。以下是训练部分代码：

```
class FaceCNN(nn.Module):
    # 初始化网络结构
    def __init__(self):
        super(FaceCNN, self).__init__()

        # 第一次卷积、池化
        self.conv1 = nn.Sequential(
            # 输入通道数 in_channels, 输出通道数(即卷积核的通道数)out_channels,
            # 卷积核大小 kernel_size, 步长 stride, 对称填 0 行列数 padding
            # input:(batch_size, 1, 48, 48), output:(batch_size, 64, 48, 48),
            # (48-3+2*1)/1+1 = 48
            nn.Conv2d(in_channels=1, out_channels=64, kernel_size=3, stride=1,
padding=1), # 卷积层
            nn.BatchNorm2d(num_features=64), # 归一化
            nn.ReLU(inplace=True), # 激活函数
            # output(batch_size, 64, 24, 24)
            nn.MaxPool2d(kernel_size=2, stride=2), # 最大值池化
        )
        # 第二次卷积、池化
        self.conv2 = nn.Sequential(
            # input:(batch_size, 64, 24, 24), output:(batch_size, 128, 24, 24),
            # (24-3+2*1)/1+1 = 24
```

```
        nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3,
stride=1, padding=1),
        nn.BatchNorm2d(num_features=128),
        nn.RReLU(inplace=True),
        # output:(batch_size, 128, 12, 12)
        nn.MaxPool2d(kernel_size=2, stride=2),
# 第三次卷积、池化
self.conv3 = nn.Sequential(
    # input:(batch_size, 128, 12, 12), output:(batch_size, 256, 12, 12),
    (12-3+2*1)/1+1 = 12
    nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3,
stride=1, padding=1),
    nn.BatchNorm2d(num_features=256),
    nn.RReLU(inplace=True),
    # output:(batch_size, 256, 6, 6)
    nn.MaxPool2d(kernel_size=2, stride=2),

# 参数初始化
self.conv1.apply(gaussian_weights_init)
self.conv2.apply(gaussian_weights_init)
self.conv3.apply(gaussian_weights_init)

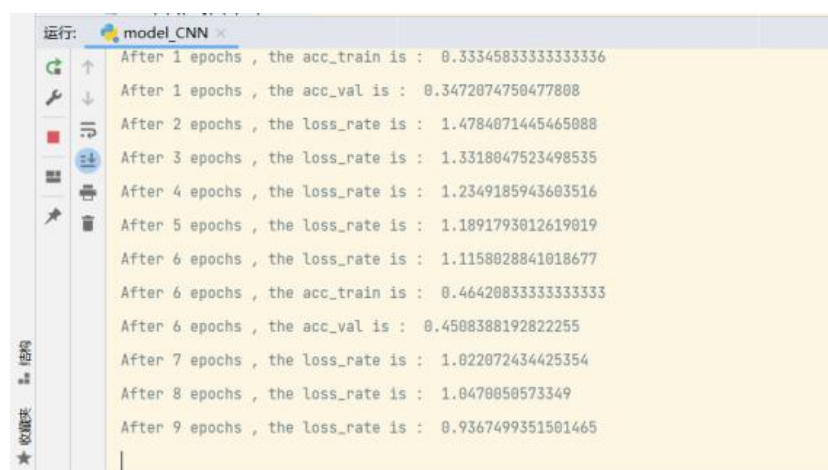
# 全连接层
self.fc = nn.Sequential(
    nn.Dropout(p=0.2),
    nn.Linear(in_features=256*6*6, out_features=4096),
    nn.RReLU(inplace=True),
    nn.Dropout(p=0.5),
    nn.Linear(in_features=4096, out_features=1024),
    nn.RReLU(inplace=True),
    nn.Linear(in_features=1024, out_features=256),
    nn.RReLU(inplace=True),
    nn.Linear(in_features=256, out_features=7),

# 前向传播
def forward(self, x):
```



```
x = self.conv1(x)
x = self.conv2(x)
x = self.conv3(x)
# 数据扁平化
x = x.view(x.shape[0], -1)
y = self.fc(x)
return y
```

在本项目中，采用的优化器是 SGD，即随机梯度下降，其中参数 `weight_decay` 为正则项系数；损失函数采用的是交叉熵；可以考虑使用学习率衰减。以下是训练过程截图：



表情识别预测：在经过 100 epochs 训练过后得到了分类器模型 `model_cnn.pkl` 文件，其中记录下了卷积神经网络的权重参数信息，在测试文件中搭建和训练模型相匹配的一致的 FaceCNN 模型，并通过已经训练好的模型传入合适的参数输出标号 0~7 对应的概率值，即：每次摄像头或屏幕流摄取的相片中调用 `dlib` 库所识别到的人脸 `resize` 成 48*48 大小并传入神经网络，从而获得捕获到的当前图像的 7 类表情标号对应的概率值，取概率最高的类别为分类结果，将结果显示在人脸对应矩形框的上方。

表情识别训练模块设计完善过程：

表情识别问题经查最常使用的数据集是“fer2013”，下载地址如下：

[fer2013 | Kaggle](https://www.kaggle.com/fer2013)

在这个数据集中，表情被标注为 7 类如下图所示：

```
98  #表情标签
99  emotion_labels = {0: 'angry', 1: 'disgust', 2: 'fear', 3: 'happy', 4: 'sad', 5: 'surprise', 6: 'neutral'}
```

因此实际上就是要解决一个 7 分类问题：

查阅了相关博客文章后发现：用传统的机器学习方法如 SVM 等等在此问题上可能效率不高、准确率不足，比较主流的算法是使用 CNN 卷积神经网络对数据集进行特征提取学习，但是由于原始的人脸图像均为 48*48 像素组成，所获取的 train.csv 文件实际上每行的第一列是表情标签，后 48*48=2304 列是人脸图像像素值，但是这样不便于后续模型训练，因此对数据集先进行了分离操作：将标签写入 emotion.csv 文件而将图像像素点写入 pixels.csv 文件，再利用 opencv 中的 imwrite 函数将像素点反写为图片，由于先前已经将 pixels 与对应的 emotion 标签进行了分离，现在重新获得图像后要重新建立各表情图像与其所属类别的关联性：显然比较优雅的一种处理方式是建立映射表（是因为读取图像时不是按照数字顺序“1、2、3、4...”而是根据首字符大小排序“0、10、100、1000...”），为了训练需要，先对训练集和验证集进行简单划分即一部分存入训练集文件夹，一部分存入验证集文件夹，然后分别遍历文件夹下的所有文件，将图片名写入列表的同时通过图片名索引出对应的表情标签将表情写入另一个列表，最后将两个列表写入同一个 csv 文件。

数据预处理过后在加载数据集时也遇到了一些问题：开始的想法是构建一个 list 读取数据，但是由于深度学习 batch_size*48*48 相对来说是一个比较大的值，如果直接用 numpy 库中 array 的数据格式来加载这些数据会占用很多内存，本身训练过程较慢将导致训练时长无法接受，因此更好的方法是直接使用 pytorch 库中的 torch.utils.Dataset，可以通过继承这个类来生成自己的数据集加载方法，解决了数据加载问题后神经网络搭建工作也正式开始：系统采用比较主流的基础框架：CNN 模型，该模型中含有三个卷积、池化层，以及第三次卷积、池

化后完成初步的特征提取，获得一个 9216×1 的特征向量，为了防止过拟合问题的出现，先加入了一层 dropout 随机失活神经元，再送入含有 4096 个神经元的隐层，随机失活神经元后再送入含有 1024 个神经元的隐层，dropout 后送入含有 256 个神经元的隐层，最后送入 7 神经元输出层，总共四层神经网络根据经验，前三层都采用 ReLU 激活函数，最后一层的激活函数选用 sigmoid 来输出各类表情的概率值，选取概率较高者为最终分类结果。

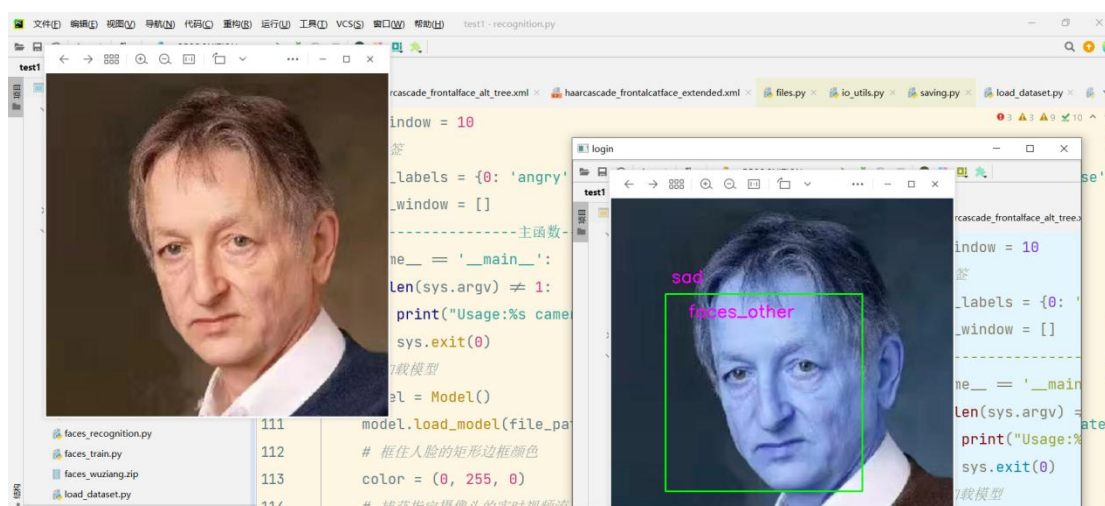
在具体实现时采用了 pytorch 来协助搭建框架，首先定义了一个参数初始化函数初始化权重，再定义了一个验证集上的验证函数用来跟踪计算验证集正确率，接着通过继承 nn.model 来定义自己的模型类，但是在编写时也发现了一定的问题就是最后输出的概率值是异常的（和不为 1），后经查阅资料发现 pytorch 设计时本身最后一层输出自带了 sigmoid 激活函数处理，因此最后将添加的 sigmoid 删去得到了正常结果。

这个四层的神经网络搭建完后开始正式进行数学运算，通过数据前向传播和误差反向传播来训练模型，其中损失函数采用的是交叉熵，进行 SGD 优化下的随机梯度下降。

但是由于图片数量以及特征提取本身运算量较大，实际的训练时间很长，例如，在指定抓取样本数量为 128 训练 10 轮、学习率 0.03 的情况下，大致需要半小时以上的训练时长，为了保证训练量足够充足，最后采用牺牲一定学习率换取更高的训练次数，每次载入 128 个样本，以 0.1 的学习率学习 100 轮。在经过数小时训练后得到了含有权重参数的训练模型。

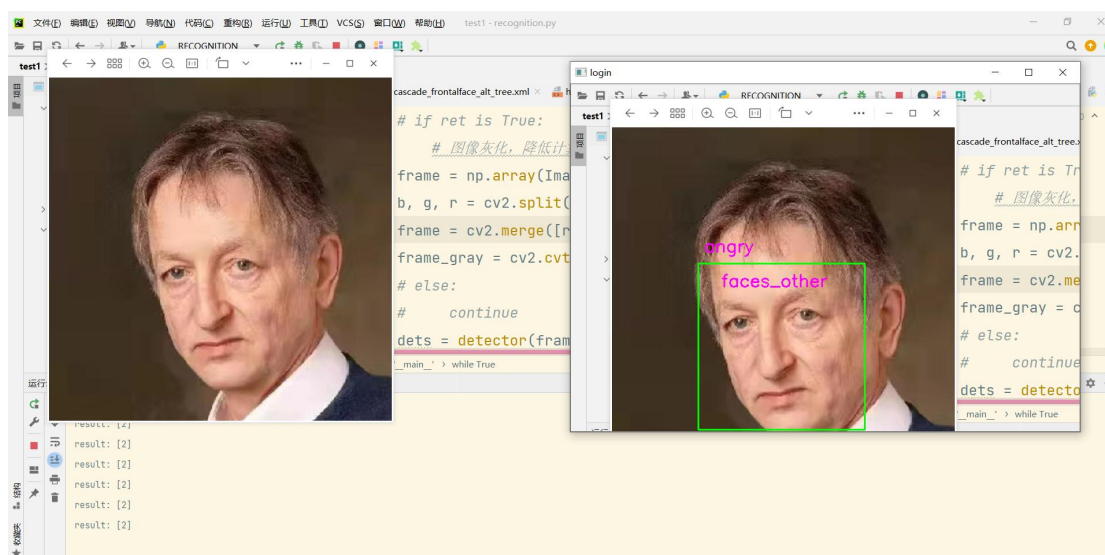
接下来就是模型实际测试的环节：神经网络结构只需要从原训练文件中照搬即可，接着用 torch.load 加载训练好的模型，调取摄像头获取人脸，使用 dlib 检测人脸区域保存到 frame 中并 resize 成 48×48 大小，传入神经网络得到输出值，并运用 np.argmax 语句提取其中的最大值，对应窗口上显示出最大概率值对应

的表情。在取得一定效果后，继续尝试了运用 pillow 库的 ImageGrab 功能识别电脑上正在播放视频的人脸表情，也就是，但是遇到了一个问题，最终窗口显示颜色怪异不仅影响观感同时造成识别有误，像下图中所示：



经过查阅资料后发现，原因是 ImageGrab 在颜色提取时的颜色通道提取顺序是 bgr 而非正常的 rgb，在调整通道顺序，加入以下代码后，解决了这个问题：

1. `frame = np.array(ImageGrab.grab(bbox=B0X))`
2. `b, g, r = cv2.split(frame)` # 分别提取 B、G、R 通道
3. `frame = cv2.merge([r, g, b])` # 重新组合为 R、G、B



完善该表情识别功能后，将该模块整合进入了人脸识别模块，也圆满达成了该系统的设计意图：即希望实时地对指定人物的表情进行监测。

3.6 图形化界面设计与项目完善

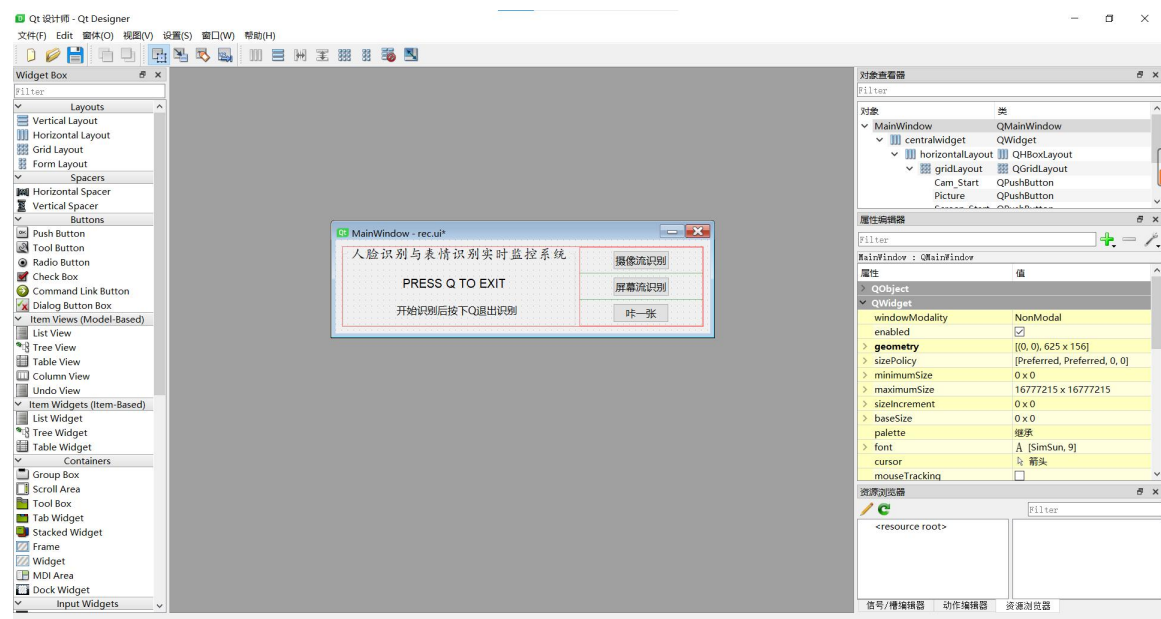
3.6.1 开发工具选择

由于该系统的设计依赖于 Python 语言，因此考虑基于 python 进行 UI 界面开发。全面考虑了各种 UI 界面设计开发工具，最终选择了功能完善且强大的 PyQt5 工具进行图形化界面开发。PyQt5 是一个创建 GUI 应用程序的工具包，它是 Python 编程语言和 Qt 库的成功融合，而 Qt 库则是最强大的库之一。PyQt5 实现了一个 Python 模块集。它有超过 300 类，将近 6000 个函数和方法。它是一个多平台的工具包，可以运行在所有主要操作系统上，包括 UNIX，Windows 和 Mac。PyQt5 采用双许可证，开发人员可以选择 GPL 和商业许可。在此之前，GPL 的版本只能用在 Unix 上，从 PyQt5 的版本 4 开始，GPL 许可证可用于所有支持的平台。基于以上多种优良特性，PyQt5 成为了该系统界面开发的选择。

3.6.2 界面设计过程

在设计开始前，首先需要全面了解 PyQt5 的设计语言。查阅资料后了解了此次界面设计所需要的必要原理和指令，以及调用的框架和方式方法。在学习过程中重点了解了“信号”和“槽”的概念，这是编写 UI 的最基本思路。接着开始了解图形化设计工具 Qt Designer。这是一个便捷的设计界面，通过拖动相应模块并进行相应的布局(layout)设计（水平排列的元素使用 Horizontal layout，垂直排列元素使用 Vertical layout）和 StyleSheet 参数调整即可设计出美观的界面。经过设计

和功能相适应的原则，设计出如下界面：



内部元素有：QMainWindow,QWidget,QPushButton,QGroupBox,QLabel.而这些都是将在后续界面和程序连接起来的重要调用对象。

3.6.3 界面与主体程序的链接

这一步需要用到 PyUIC 对.ui 文件进行 python 语言的转换，上述界面的.ui 文件转换如下：

```
from PyQt5 import QtCore, QtGui, QtWidgets

class Ui_MainWindow(object):

    def setupUi(self, MainWindow):

        MainWindow.setObjectName("MainWindow")

        MainWindow.resize(625, 356)

        MainWindow.setStyleSheet("*:hover { color: red; }")

        MainWindow.setToolButtonStyle(QtCore.Qt.ToolButtonIconOnly)

        ...

        self.Screen_Start = QtWidgets.QPushButton(self.centralwidget)

        font = QtGui.QFont()

        font.setFamily("微软雅黑")

        font.setPointSize(10)
```

```

        self.Screen_Start.setFont(font)
        self.Screen_Start.setObjectName("Screen_Start")
        self.gridLayout.addWidget(self.Screen_Start, 1, 0, 1, 1)
        self.horizontalLayout.addLayout(self.gridLayout)
        self.horizontalLayout_3.addLayout(self.horizontalLayout)
        MainWindow.setCentralWidget(self.centralwidget)
        self.retranslateUi(MainWindow)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)

    def retranslateUi(self, MainWindow):
        _translate = QtCore.QCoreApplication.translate
        MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow"))
        self.groupBox.setTitle(_translate("MainWindow", "人脸识别与表情识
别实时监控系统"))
        self.label.setText(_translate("MainWindow", "PRESS Q TO EXIT"))
        self.label_2.setText(_translate("MainWindow", "开始识别后按下 Q 退出
识别"))
        self.Cam_Start.setText(_translate("MainWindow", "摄像流识别"))
        self.Picture.setText(_translate("MainWindow", "咔一张"))
        self.Screen_Start.setText(_translate("MainWindow", "屏幕流识别"))

```

这个是 ui 文件的 python 代码段。由于项目开发灵活性的需求，进而寻求到一种更灵活的动态开发方式，直接调用.ui 文件然后使用类的调用方法，给出按钮按下的信号和接受信号的“槽”函数。代码如下：

```

from PyQt5 import uic
from PyQt5.QtWidgets import QApplication
class Face:
    def __init__(self):
        # 从文件中加载 UI 定义
        self.ui = uic.loadUi("ui\\rec.ui")
        self.ui.setWindowTitle("人脸识别系统")
        self.ui.setFixedSize(515, 150)
        self.ui.Cam_Start.clicked.connect(self.cam_rec) # cam_rec 为“槽”函数
        self.ui.Picture.clicked.connect(self.take_picture) # take_picture 为
“槽”函数

```



```
self.ui.Screen_Start.clicked.connect(self.screen_rec) # screen_rec 为“槽”函数
```

标黄部分即为自定义的按钮名称：

Cam_Start	QPushButton
Picture	QPushButton
Screen_Start	QPushButton

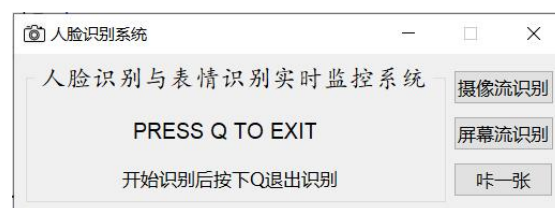
编写好 `__init__(self)` 后，后续继续定义“槽”函数：

```
def cam_rec(self)
def take_picture(self)
def screen_rec(self)
```

在“槽”函数内即可编写相应的识别源代码。原理是在按下相应名称的 QPushButton 后发射一个“信号”到“槽”函数，即调用函数操作。最后需要一个入口主函数：

```
# UI 主函数
if __name__ == '__main__':
    App = QApplication(sys.argv)
    App.setWindowIcon(QIcon('logo.png')) # 加上 logo
    win = Face()
    win.ui.show()
    sys.exit(App.exec_())
```

进行一个类的实例化即可调用 UI 界面并和相应的界面联系起来。运行界面如下：



3.6.4 整体项目完善与代码优化

在之前的 UI 调用框架下，对于界面功能的完善重心转移到了槽函数的编写上。依据最简单的代码结构，在开始识别的信号下，直接调用相应槽函数内部的识别的源代码即可，结果代码运行变得迟缓，原因是重复的函数定义和模型调用

与计算分类使得代码运行内存占用量大大增加，导致程序出现开启运行迟缓，出现影响使用系统使用体验的情况。


基于此，在充分分析两种识别模式（摄像流和屏幕流识别）的共性和个性后，编写新的函数(`def rec(self)`)，将代码共性作为识别的头部代码段，中部则使用全局变量 `flag` 来进行条件判断执行两种模式下代码段的个性部分。对应接收信号的槽函数则改变为更改全局变量 `flag` 然后执行识别对应的函数即可，修改如下：

```
def __init__(self):
    ...
    self.ui.Cam_Start.clicked.connect(self.cam_flag)
    self.ui.Picture.clicked.connect(self.take_picture)
    self.ui.Screen_Start.clicked.connect(self.screen_flag)
# 执行两种识别
def rec(self):...
# 调取摄像头 标志为 1
def cam_flag(self):
    global flag
    flag = 1
    Face.rec(self)
# 调取屏幕 标志为 2
def screen_flag(self):
    global flag
    flag = 2
    Face.rec(self)
```

优化后源码代码量减轻优化了 39.7%，代码运行速度显著提升。

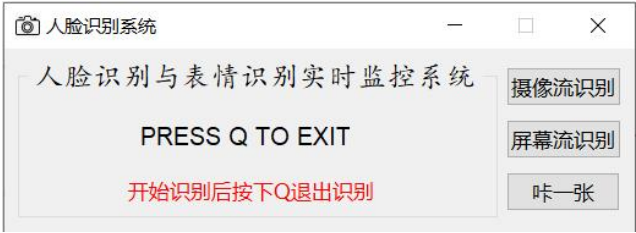
源码部分优化完毕后，基于使用方面，做出了如下完善：

- ① 缩小界面，调整字体大小，做成类似遥控器界面：增加美观度
- ② 增加系统显示图标增加美观度；

 人脸识别系统



- ③ 增加提示（按 Q 退出识别）：便于用户正确使用系统界面；



④ 优化按钮状态，做到鼠标悬浮时，按钮变红，提升操作体验感；



3.7 重点开发工作

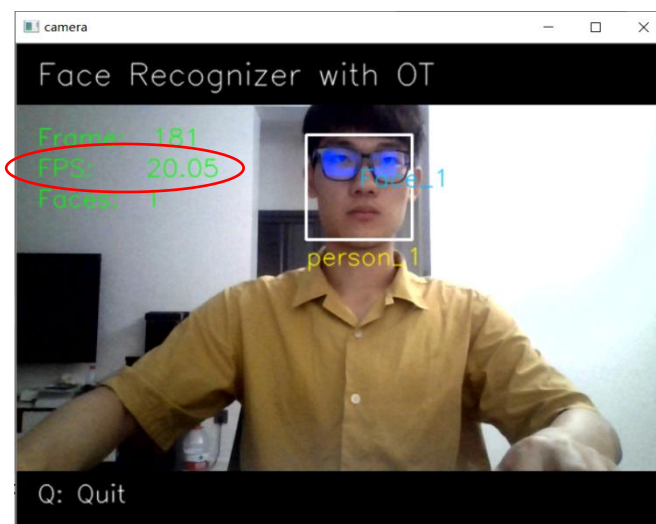
整个系统设计工作拆分为大致七个板块，分别是模式识别系统分类策略测试比较与选择、数据集寻找下载于数据适应性预处理、人脸识别分类器设计实现与模型训练、表情识别分类器设计实现与模型训练、人脸与表情识别整体连接与代码整理、图形化界面设计与代码整体框架梳理与优化、整体程序封装与测试。

分类策略测试比较与选择：

经典人脸识别算法是一种内核算法，是通用统计算法：PCA 和 SVM。主成分分析（PCA）是一种具有许多实际应用的通用统计方法。当在人脸识别过程中使用时，PCA 旨在减少源数据的大小，同时保留最相关的信息。它生成一组加权特征向量，这些特征向量依次构建特征脸——大量不同的人脸图像。特征脸的线性组合代表训练集中的每个图像。PCA 用于从训练图像集的协方差矩阵中接收这些特征向量。对于每张图像，计算其主要成分（从 5 到 200）。其他组件编码面部和噪声之间的细微差异。识别过程包括将未知图像的主要成分与所有其他图像的成分进行比较。

支持向量机 (SVM) 是一种机器学习算法，它使用两组分类原则来区分人脸和“非人脸”。对于每个类别，SVM 模型都会接收一个标记的训练数据集来对新的测试数据进行分类。研究人员将线性和非线性 SVM 训练模型应用于人脸识别。最近的结果表明，非线性训练机具有更大的余量和更好的识别和分类结果。但是这些经典传统算法在面对大量数据和小量数据的多分类时（比如测试团队 5 位成员的人脸识别分类）就达不到理想的识别分类结果。

在前期过程中自然而然的注意到基于 Opencv+dlib+OT 的实时视频人脸检测，对于特征提取过程直接调用 dlib 的 landmark 人脸提取器即可，实现起来较为便捷。实现的识别效果准确率较高，视频 FPS 也能维持在 20 左右，比较流畅。从资料中不断截取代码尝试组合也实现了此方法下的分类，识别效果如下：



但是到了多人识别部分，该分类模型就显得十分卡顿，虽然分类效果比较理想，但是画面呈现效果不尽如人意。

于是选题方向来到了更为前沿先进的人脸识别方法：卷积神经网络（CNN）。卷积神经网络（CNN）是人工神经网络（ANN）和人工智能发展的突破之一。它是深度学习中最流行的算法之一，深度学习是一种机器学习，模型学习直接对图像、视频、文本或声音执行分类任务。该模型在多个领域显示出令人印象深刻的结果：计算机视觉、自然语言处理（NLP）和最大的图像分类数据集（Image Net）。CNN 是一个普通的神经网络，带有新的层——卷积层和池化层。CNN 可以有几十个和几百个这样的层，每个层都通过反向传播误差和不断优化损失函数可以学会检测不同的成像特征。在查阅大量有关卷积神经网络的人脸识别论文和资料后，发现其优势明显，在识别准确率方面可以达到很高的水平（接近 99%），且同时，卷积神经网络发展迅速，训练框架比较成熟，转化应用起来十分方便，因此很适合本多人人脸识别检测和表情分类检测。于是确定系统设计应用的方法为卷积神经网络。

人脸识别分类器设计实现与模型训练：

对于人脸识别分类器的设计来说，神经网络模型的搭建过程是比较复杂的。随后明确了卷积层的作用是提取特征，池化层的作用是进行特征选择和信息过滤，全连接层的作用是对提取的特征进行非线性组合以得到输出。并且单层的神经网络结构往往只能提取到一些简单的特征，对于人脸识别这种比较复杂的特征来说，多层的神经网络堆叠往往可以在提取特征上达到更好的效果。

此外在模型搭建和训练部分，还需要考虑一个重要问题，那就是编码，即为不同的人脸数据集赋予不同的标签编码方式。现如今常用的标签编码方式主要有两种，一种是整数编码，给每个类别值都分配一个整数值，而另一种则是 One-hot 独热编码，为每个整数值都创建一个二值变量。比如，用 1 表示红色（red），2 表示绿色（green），3 表示蓝色（blue）。这种方式被称为标签编码或者整数编码，可以很轻松地将它还原回类别值。对于某些变量来说，这种编码就足够了。整数之间存在自然的排序关系，机器学习算法可以理解并利用这种关系。但对于不存在次序关系的类别变量，仅使用上述的整数编码是不够的。实际上，使用整数编码会让模型假设类别间存在自然的次序关系，从而导致结果不佳或得到意外的结果（预测值落在两个类别的中间）。这种情况下，就要对整数表示使用 One-Hot 编码了。One-Hot 编码会去除整数编码，并为每个整数值都创建一个二值变量。就比如一个二分类问题，编码方式有 $[0, 1]$, $[1, 0]$ 两种，对于三分类问题，则有 $[0, 0, 1]$, $[0, 1, 0]$, $[1, 0, 0]$ 三种编码方式，以此类推。现在多分类 CNN 网络的输出通常是 softmax 层，而它的输出是一个概率分布，从而要求输入的标签也以概率分布的形式出现，进而算交叉熵。One-hot 其实就是给出了样本的真实概率分布，其中一个样本数据概率为 1，其他全为 0。计算损失交叉熵时，直接用 $1 * \log(1/\text{概率})$ ，就直接算出了交叉熵，作为损失。One-hot 也作为现如今机器学习领域数据处理方面所使用的较为广泛的编码方式，也具备一定优越性，于是便选择使用 One-hot 独热编码的方式来实现对不同类别的数据集进行标签。

对于人脸识别卷积神经网络的模型搭建,选择使用现如今的主流深度学习框架 tensorflow 进行完成。系统利用lfw 开源数据集中的人脸图片作为训练数据集来源之一,这里面包含了来自数千人的共计约 14000 张的人脸图片,将这个文件夹中的所有图片进行统一提取并重命名为数字命名 1、2、3…….jpg 的形式存放于 faces_other 文件夹中。另外采集了 10000 张自己的人脸图片,保存在 faces_my 文件夹之中。

在本系统 CNN 模型第一个版本之中,对于数据集的划分时的 One-hot 编码只是固定地写成了二分类问题,并且网络结构的输出层选用的是 sigmoid,这就导致了模型只能识别两种人脸,也就是 my face 和 other face。而实质上完全可以采用利用算特征点的欧式距离这种非常简单的方式就可以完成识别单个人的脸,然后再将其他不属于这个人的脸统一归类为 others 即可,并且这样的识别方法可以得到更高的识别精度,完全没有必要这么大费周章。考虑到二分类问题人脸识别分类器的应用场景局限性,需要将二分类的人脸识别向多分类的人脸识别问题进行转变。

对于多分类问题来说,会涉及到很多不同的人的脸,这对模型精度要求比较高,怎样才能保证对于添加多人的人脸照片也能保持正确,这就需要考虑到神经网络的层之间的关系。对于单层的卷积神经网络往往只能提取出较为简单的特征信息,而想要提取到更多的复杂信息,需要更多层的神经网络层进行堆叠,此外需要更多的训练样本和训练轮数来保证机器学习效果。

在使用 tensorflow 来作为核心开发框架的核心下,引入了 keras 深度学习框架作为模型搭建的辅助工具,使用 keras.models 中的 Sequential 线性堆叠模型。

以下是改进后的多分类人脸识别分类器模型的卷积神经网络构造核心代码:

```
def build_model(self, dataset, nb_classes=4):

    self.model = Sequential()

    # 以下代码将顺序添加 CNN 网络需要的各层，一个 add 就是一个网络层

    self.model.add(Convolution2D(32, 3, 3, border_mode='same',
                                   input_shape=dataset.input_shape)) # 1 2 维卷积层

    self.model.add(Activation('relu')) # 2 激活函数层

    self.model.add(Convolution2D(32, 3, 3)) # 3 2 维卷积层

    self.model.add(Activation('relu')) # 4 激活函数层

    self.model.add(MaxPooling2D(pool_size=(2, 2))) # 5 池化层

    self.model.add(Dropout(0.25)) # 6 Dropout 层

    #池化层 特征选择和信息过滤

    self.model.add(Convolution2D(64, 3, 3, border_mode='same')) # 7 2 维卷积层

    self.model.add(Activation('relu')) # 8 激活函数层

    self.model.add(Convolution2D(64, 3, 3)) # 9 2 维卷积层

    self.model.add(Activation('relu')) # 10 激活函数层

    self.model.add(MaxPooling2D(pool_size=(2, 2))) # 11 池化层

    self.model.add(Dropout(0.25)) # 12 Dropout 层

    #卷积神经网络中的卷积层和池化层能够对输入数据进行特征提取，全连接层的作用则是对
    提取的特征进行非线性组合以得到输出

    self.model.add(Flatten()) # 13 Flatten 层

    self.model.add(Dense(512)) # 14 Dense 层,又被称作全连接层

    self.model.add(Activation('relu')) # 15 激活函数层

    self.model.add(Dropout(0.5)) # 16 Dropout 层

    self.model.add(Dense(nb_classes)) # 17 Dense 层
```

```
self.model.add(Activation('softmax')) # 18 分类层, 输出最终结果

# 输出模型概况

self.model.summary()
```

总共添加了共计 18 个层的网络结构, 这样多层次的神经网络可以实现对于多个人的人脸识别, 并且在最后对于测试团队所有人的人脸进行测试时也达到了不错的识别效果, 并且为了检验模型学习效果, 将数据集一分为二, 分别为训练集、验证集, 对训练模型的训练效果进行检验, 并且实现再优化。

对于训练数据类别划分与标签编码, 利用 `keras.utils` 中包含的 `one-hot` 编码功能来对 `data` 文件夹所存储的各类人的人脸信息进行统一的编码, 大大简化了代码编写难度。

对于模型训练部分, 需要选择一个合适的优化器, 来作为优化方法, 寻找模型最优解。常见的优化器有标准梯度下降法 (GD)、批量梯度下降法 (BGD)、随机梯度下降法 (SGD)、动量优化法、Adam 算法等等。开发阶段选择了两款优化器进行测试, 一种是 SGD+momentum 的优化器进行训练, 另外一种则是 adam 优化器, 采用 `categorical_crossentropy` (交叉熵) 来作为损失函数, 度量模型预测值与真实值的差异程度来检验模型学习效果的好坏。最终系统采用图像分类领域使用更为广泛的 SGD+momentum 的优化器进行训练。相比于 SGD 优化器, 其在加入了动量之后, 参数更新可以保持之前的更新趋势, 而不会出现 SGD 优化器存在的求解时会卡在梯度较小点的情况, 对于图像识别方向有着不错的优势。

确定了优化器, 利用数据生成器 `model.fit_generator()` 来分批次向模型输送数据, 可以有效减少单次内存的消耗, 此外在训练数据的时候, 使用了数据增强, 即对提供的训练数据中利用旋转、翻转、加噪声等方法创造新的训练数据,

有意识的提升训练数据规模，增加模型训练量，提高模型的泛化能力。

```
datagen = ImageDataGenerator(
    featurewise_center=False, # 是否使输入数据去中心化(均值为0),
    samplewise_center=False, # 是否使输入数据的每个样本均值为0
    featurewise_std_normalization=False, # 是否数据标准化(输入数据除以数据集的标准差)
    samplewise_std_normalization=False, # 是否将每个样本数据除以自身的标准差
    zca_whitening=False, # 是否对输入数据施以ZCA白化
    rotation_range=20, # 数据提升时图片随机转动的角度(范围为0~180)
    width_shift_range=0.2, # 数据提升时图片水平偏移的幅度(单位为图片宽度的占比, 0~1之间的浮点数)
    height_shift_range=0.2, # 同上, 只不过这里是垂直
    horizontal_flip=True, # 是否进行随机水平翻转
    vertical_flip=False) # 是否进行随机垂直翻转
```

在 data 文件夹中包含了共计五个人的人脸图片，每个人的人脸图片信息均为 10000 张，在训练模型的时候使用 CPU 进行模型训练，每次输送 20 个数据，总共训练 10 轮，CPU 训练的效率比较慢，一次完整的训练需要耗时一个多小时，为了提高训练效果，最终借助 GPU 进行模型训练，通过配置 CUDA 10.0 和 CUDNN 7.6，借助 tensorflow-gpu 1.13.1，在模型训练速度上得到显著提升，相较于 CPU 训练速度提升了 5 倍以上，大大提高了训练模型时的速度，在学习率为 0.01 的情况下，最终在测试集上可以达到 99% 的识别准确率。精确度可以达到比较高的水平。

开始模型训练：

```
2022-07-13 19:03:03.375467: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this
2022-07-13 19:03:03.582382: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1433] Found device 0 with properties:
name: NVIDIA GeForce GTX 1650 major: 7 minor: 5 memoryClockRate(GHz): 1.56
pciBusID: 0000:01:00:0
totalMemory: 4.00GiB freeMemory: 3.24GiB
2022-07-13 19:03:03.583060: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1512] Adding visible gpu devices: 0
2022-07-13 19:03:05.398751: I tensorflow/core/common_runtime/gpu/gpu_device.cc:984] Device interconnect StreamExecutor wi
2022-07-13 19:03:05.398931: I tensorflow/core/common_runtime/gpu/gpu_device.cc:990] 0
2022-07-13 19:03:05.399038: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1003] 0: N
2022-07-13 19:03:05.401391: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created TensorFlow device (/job:local
2022-07-13 19:03:06.083595: I tensorflow/stream_executor/dso_loader.cc:152] successfully opened CUDA library cublas64_100.

1/4500 [.....] - ETA: 6:19:23 - loss: 1.6060 - acc: 0.2500
4/4500 [.....] - ETA: 1:35:44 - loss: 1.5941 - acc: 0.2625
7/4500 [.....] - ETA: 55:12 - loss: 1.6042 - acc: 0.2357
11/4500 [.....] - ETA: 35:33 - loss: 1.6037 - acc: 0.2409
15/4500 [.....] - ETA: 26:22 - loss: 1.6036 - acc: 0.2300
```

模型训练结束：

```
4487/4500 [=====>.] - ETA: 0s - loss: 0.0402 - acc: 0.9890
4490/4500 [=====>.] - ETA: 0s - loss: 0.0402 - acc: 0.9890
4493/4500 [=====>.] - ETA: 0s - loss: 0.0401 - acc: 0.9890
4496/4500 [=====>.] - ETA: 0s - loss: 0.0401 - acc: 0.9890
4499/4500 [=====>.] - ETA: 0s - loss: 0.0402 - acc: 0.9890
4500/4500 [=====] - 92s 21ms/step - loss: 0.0402 - acc: 0.9890 - val_loss: 0.0162 - val_acc: 0.9970
```

在识别运行过程中，系统利用交叉熵损失函数计算出输入图像从属于各个数据集概率，当最大概率大于 0.9 时，将其视为是识别出事先所录入的人脸，当所计算出的各个概率的最大概率值仍小于 0.9 时，将其处理为不属于所录入的人脸，将其统一归类为 faces_other，并在屏幕上实时显示。

人脸分类决策代码：

```
faceID = model.face_predict(face)
cv2.rectangle(frame, (x2- 10, x1 - 10), (y2+ 10, y1 + 10), color, thickness=2)
# face_id判断 (
for i in range(len(os.listdir('./data/'))):
    if i == faceID:
        # 文字提示是谁
        cv2.putText(frame, os.listdir('./data/')[i],
                    (x2 + 30, x1 + 30), # 坐标
                    cv2.FONT_HERSHEY_SIMPLEX, # 字体
                    1, # 字号
                    (255, 0, 255), # 颜色
                    2) # 字的线宽
    if faceID == -1:
        cv2.putText(frame, 'faces_other',
                    (x2 + 30, x1 + 30), # 坐标
                    cv2.FONT_HERSHEY_SIMPLEX, # 字体
                    1, # 字号
                    (255, 0, 255), # 颜色
                    2) # 字的线宽
```

图形化界面设计与代码整体框架梳理与优化：

对于图形化界面的设计，查阅资料了解广泛了解后，得知，若目标是设计出一个美观且控件丰富功能强大的 UI 界面，PyQt5 和 PySide2 是很好的选择，但是由于 PySide2 版权的原因，就选择了 PyQt5 作为此次界面开发的工具。

利用图形化界面设计中的“信号”和“槽”的概念，在实现时，编写了调用界面的类 class，然后将类实例化调用，显示界面，循环等待用户的操作，直到用户点击关闭界面。到这里一切顺利，紧接着就遇到了问题。在运行代码时候，出现了加载 FaceCNN 模型错误的报错，但是代码中写明了 FaceCNN 这一个类。于是在网上大量查找相关报错的解决方法，说是这类错误需要将模型类显性调用，但是代码中已经显性调用了，于是还找不到解决本问题的方法。然而在尝试性的

将 FaceCNN 类剪切到定义的图形化界面类的外部后，模型能够正常加载然后代码也能正常运行了。接着就是进行摄像流和屏幕流的识别切换，和添加截图功能。其中遇到了摄像流和屏幕流切换时的代码结构优化问题，优化解决方法如下：

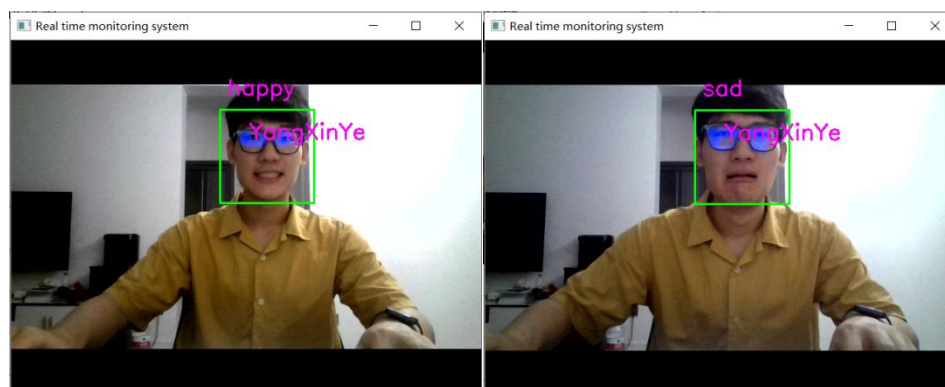
在充分分析两种识别模式（摄像流和屏幕流识别）的共性和个性后，编写新的函数(`def rec(self)`)，将代码共性作为识别的头部代码段，中部则使用全局变量 `flag` 来进行条件判断执行两种模式下代码段的个性部分。对应接收信号的槽函数则改变为更改全局变量 `flag` 然后执行识别对应的函数即可优化程序并且最后肉眼可见地提升了程序运行速度和响应速度。

最后收尾工作主要为：

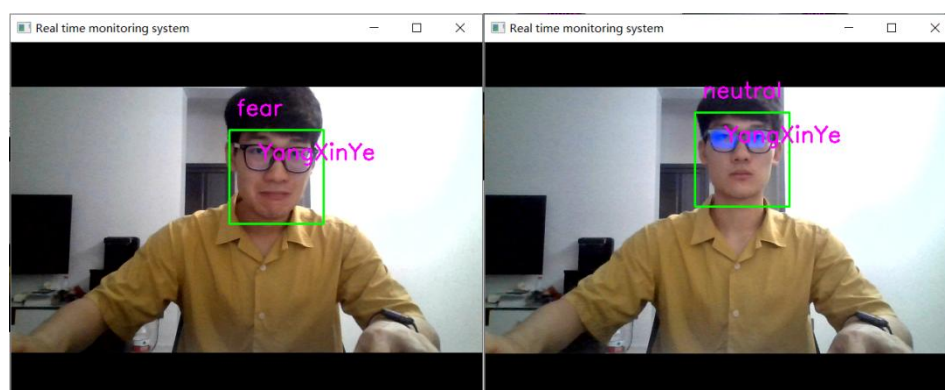
- ① 优化代码注释行，精简注释方便阅读；
- ② 删除不需要的.py 文件，优化文件传输内存；
- ③ 统一文件命名，如数据集文件；
- ④ 美化界面设计，增加操作体验感和交互感。
- ⑤ 尝试打包成.exe 可执行文件，提高系统作品的迁移性。

四、软件运行展示与分析

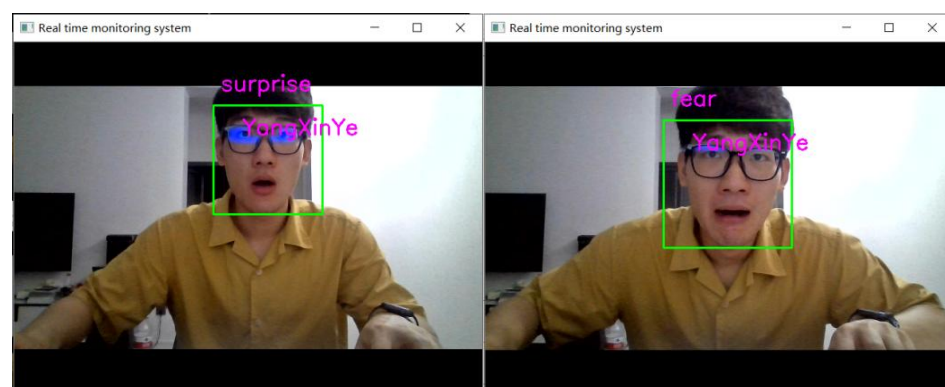
运行结果——摄像流：



成功识别人脸且识别出 happy 和 sad 表情

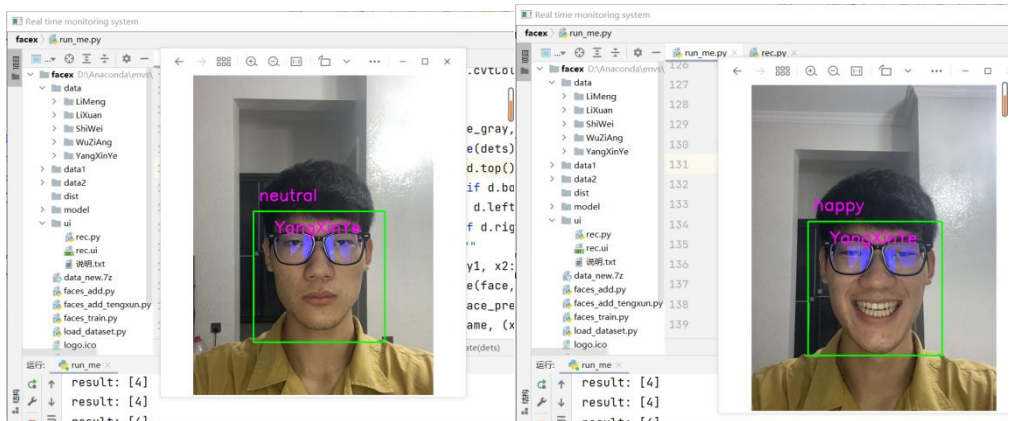


成功识别人脸且识别出 fear 和 neutral 表情



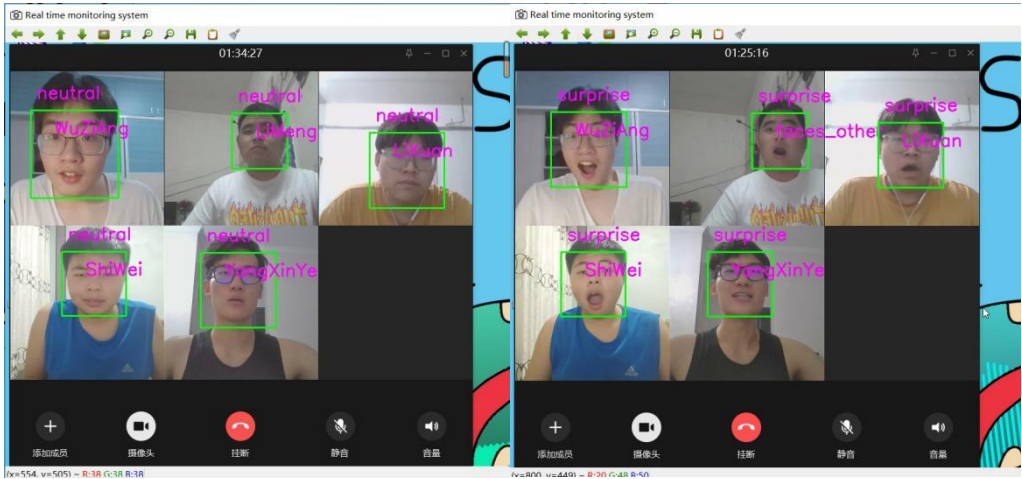
成功识别人脸且识别出 surprise 和另外一种 fear 表情

运行结果——屏幕流：



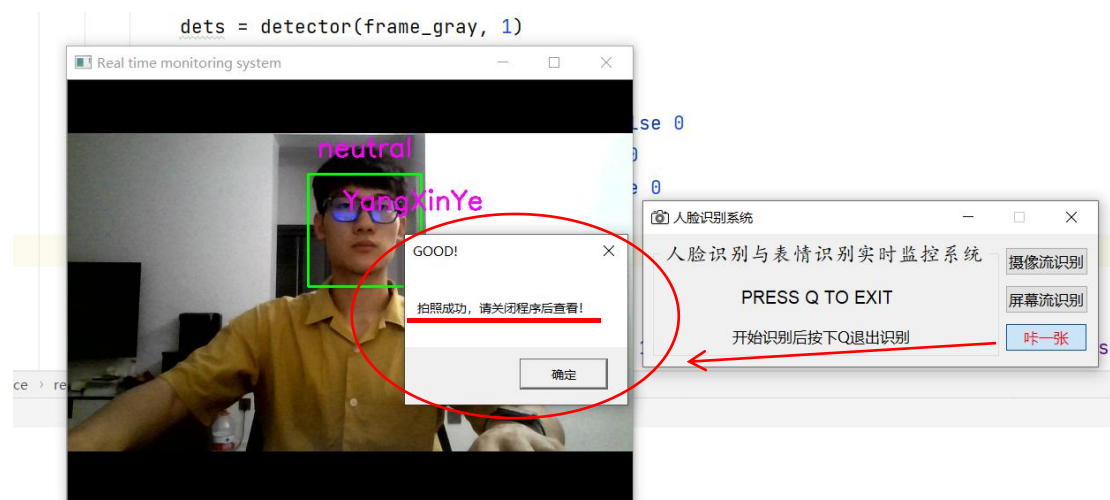
在屏幕上成功读取到人脸进行识别并识别出 neutral 和 happy 表情

运行结果——多人识别：

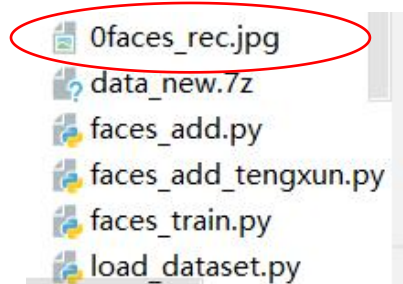


多人实时视频会议时读取屏幕人脸进行识别，全员分别为 neutral 和 surprise。

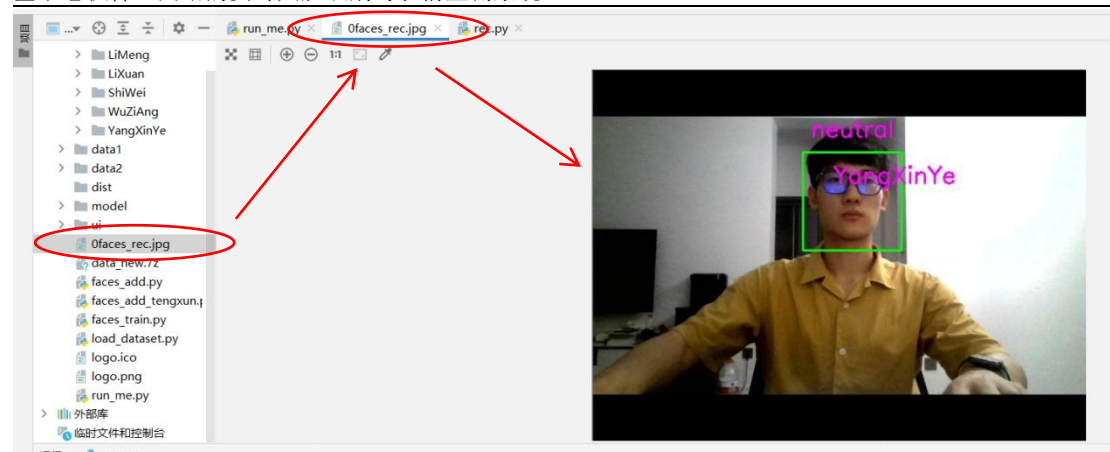
运行结果——截图功能：



截图成功后弹窗提示

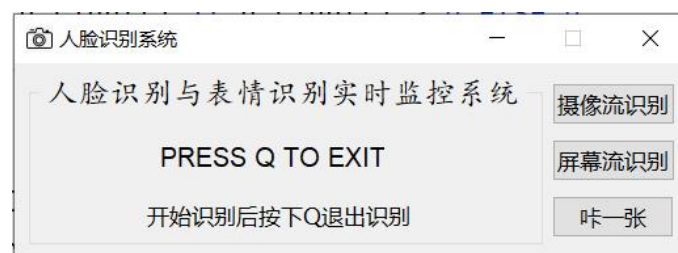


关闭程序后出现截图 0face_rec.jpg

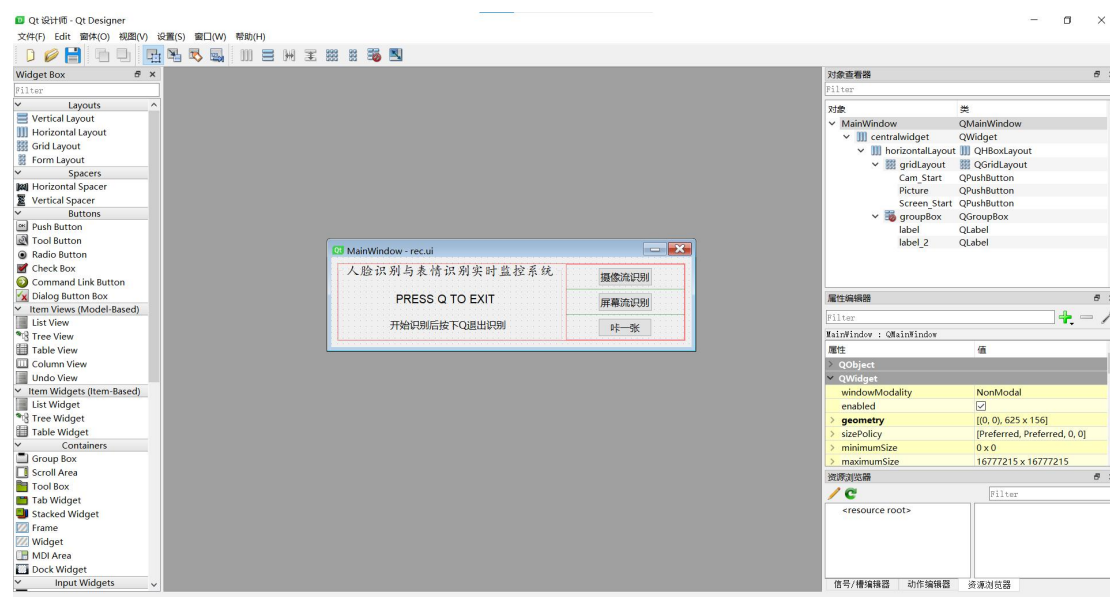


点击查看截图文件为刚才识别窗口截图

运行结果——界面设计：



最终界面

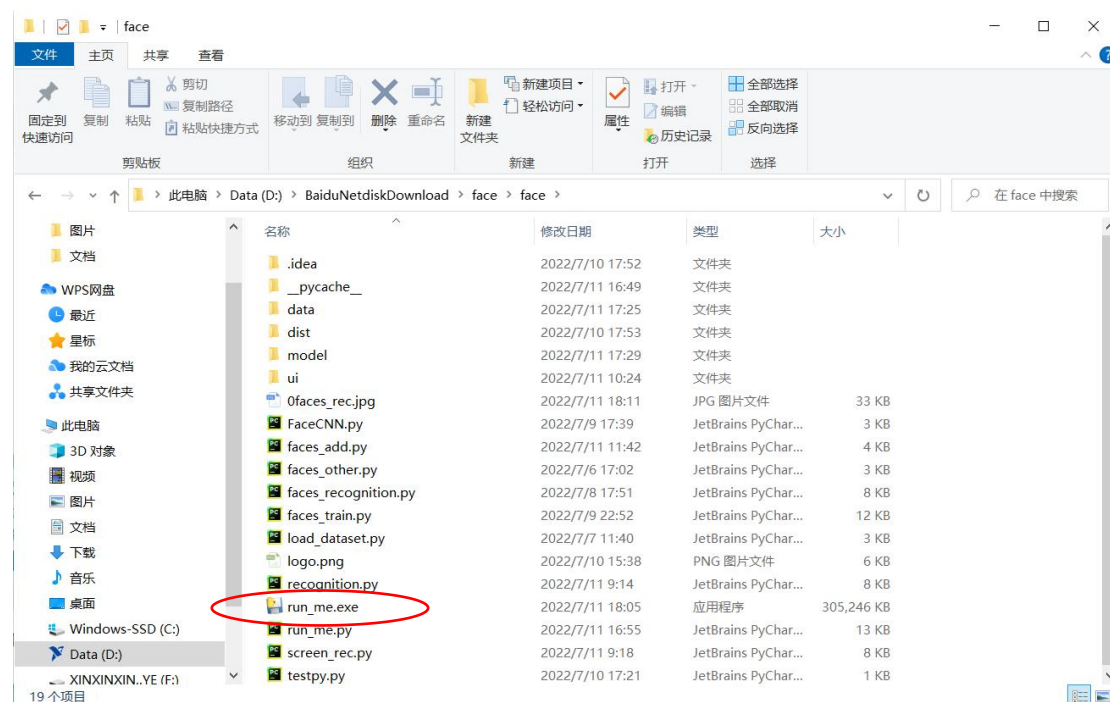


Qt Designer 设计界面

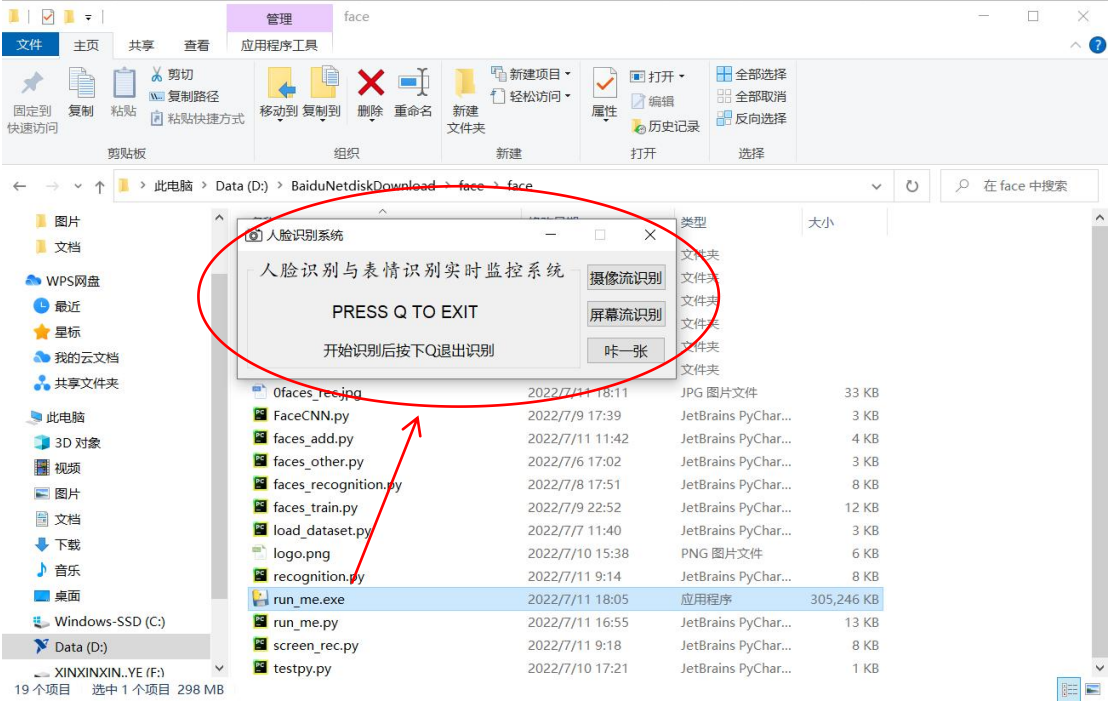
```
102         font.setPointSize(10)
103         self.Screen_Start.setFont(font)
104         self.Screen_Start.setObjectName("Screen_Start")
105         self.gridLayout.addWidget(self.Screen_Start, 1, 0, 1, 1)
106         self.horizontalLayout.addLayout(self.gridLayout)
107         self.horizontalLayout_3.addLayout(self.horizontalLayout)
108         MainWindow.setCentralWidget(self.centralwidget)
109
110         self.retranslateUi(MainWindow)
111         QtCore.QMetaObject.connectSlotsByName(MainWindow)
112
113     def retranslateUi(self, MainWindow):
114         _translate = QtCore.QCoreApplication.translate
115         MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow"))
116         self.groupBox.setTitle(_translate("MainWindow", "人脸识别与表情识别实时监控系统"))
117         self.label.setText(_translate("MainWindow", "PRESS Q TO EXIT"))
118         self.label_2.setText(_translate("MainWindow", "开始识别后按下Q退出识别"))
119         self.Cam_Start.setText(_translate("MainWindow", "摄像流识别"))
120         self.Picture.setText(_translate("MainWindow", "咋一张"))
121         self.Screen_Start.setText(_translate("MainWindow", "屏幕流识别"))
122
```

121 行 UI 界面对应的 Python 代码

运行结果——可执行文件封装运行：



打包目录下的.exe 可执行文件



双击可执行文件后系统正常运行

分析：

总体情况下看，本基于卷积神经网络的实时人脸识别与表情监测系统能够基本实现预期效果。调取摄像头实现人脸识别的实时监控，读取屏幕可以实时监测屏幕视频图片等文件，还能用于多人会议的面部表情和身份识别与监控，最后给系统通过图形化界面设计与打包封装，增加了其作为一个系统的完备性与可迁移性，实用性与功能性大大增强。

预定的目标为调取摄像头，读取屏幕，拍照，多人识别等，基于功能和设计相适应的原则，设计出相应按钮，调整代码结构等。起初为了实现读取摄像头和屏幕流之间的切换，直接将代码重复的写进槽函数中，殊不知这大大增加了代码冗余量，拖慢了程序运行速度，也大大增加了系统的计算量和内存占有量。调整后，代码运行较为流畅，代码冗余量降低 39.7%。

五、总结

系统设计概述：

系统完备性较好，工作量较大，功能丰富，能够实现测试团队内成员的单人识别和多人识别，能够较灵敏的识别出当前人物的实时表情。拥有自行设计的图形化控制界面，功能强大，界面简洁清晰，迁移性优异。

功能模块概述：

系统按粗粒度划分，可分为：人脸识别模块，表情识别模块，图形化界面模块。其中人脸识别与表情识别共用 dlib 库主动识别并实时截取出的人脸图像，该图像将会归一化到相同的尺寸送入人脸识别以及表情识别网络进行预测。预测结果将会在人脸检测框附近标出。对于实时显示的结果，可于图形化界面窗口点击“咔一张”按钮进行屏幕截取并保存至默认路径。此外通过图形化界面可切换视频流来源，当点击图像视频流时，系统打开电脑前置摄像头并针对视频中的人脸进行识别标注；当点击屏幕流识别时，系统实时截取电脑界面中的人脸进行识别标注。

系统的部署方案概述：

当前该软件的支持范围是电脑端，且需为 windows10 操作系统。因此在部署时需要使用 windows10 及以上进行部署。另外，该模型的人脸识别模块还支持用户数据的自录入，即用户通过自己的屏幕前置摄像头对用户人脸进行采集足够数量的照片，进而送入训练模块进行训练，从而得到自己所需的人脸识别系统。如果使用 GPU 进行训练，建议显卡配置在 NVIDA2060 以上。

推荐的一种部署环境如下：

- Dell G5 5500
- 处理器 Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz

- 显示适配器 1 Intel(R) UHD Graphics
- 显示适配器 2 NVIDIA GeForce RTX 2060
- 机带 RAM 16.0 GB (15.8 GB 可用)
- anaconda 4.13.0
- PyQt5
- PyCharm Community 2021.2

针对实际应用场景的部署，以部署在医疗系统为例：

实际流程如下：

① 确定需求和目标：首先，需要确定部署人脸加表情识别系统的具体目标和需求。这可能包括确定系统所需的功能、性能要求、数据类型和格式等。

② 获取数据：接下来，需要获取医疗数据库中的数据，并将其准备好用于训练人脸加表情识别系统的模型。这可能包括清理和预处理数据，并将其分割为训练集、验证集和测试集。

③ 训练模型：使用获取的数据训练人脸加表情识别系统的模型。这可能包括使用机器学习算法调整模型的参数，以使模型能够尽可能准确地识别人脸和表情。

④ 部署系统：将训练好的模型部署到计算机或移动设备上。这可能包括确定模型的输入和输出格式以及与确定的硬件平台的接口，并安装必要的软件。

产品应用前景：

所设计的人脸+表情识别系统是一种能够识别人脸并判断其表情的软件系统。这种系统使用计算机视觉技术来检测和分析人脸的各个特征，包括眼睛、鼻子、

嘴巴和脸颊。通过对这些特征的分析，系统可以判断人脸的表情，如笑容、愤怒或悲伤。

人脸+表情识别系统的运用前景非常广阔。这种系统可以用于许多不同的领域，如安全监控、市场营销、娱乐和医疗保健等。

在安全监控领域，人脸+表情识别系统可以用于识别犯罪嫌疑人或警告警察有危险的人。在市场营销领域，这种系统可以用于分析客户的表情，以了解他们对产品或服务的反应。在娱乐领域，这种系统可以用于制作虚拟角色，使其能够真实地表现出人类的各种表情。在医疗保健领域，这种系统可以用于监测病人的情绪状态，帮助医生更好地诊断和治疗疾病。故总体上看，该软件系统的运用前景很广阔。