

# 机器智能大作业----人脸识别系统代码报告

---

1611453 徐宇成

2019年1月18日

## 机器智能大作业----人脸识别系统代码报告

### 第一部分 环境配置

1. ubuntu 与 python 虚拟环境的配置

2. CUDA 与 CUDNN 的配置

在环境配置上有问题请联系 : hilbertxu@outlook.com

### 第二部分 人脸识别系统的原理以及使用工具

3. 实验目的

4. 系统工作原理

4.1 卷积神经网络层级结构

4.1.1 数据输入层

4.1.2 卷积层

4.1.3 池化层

4.1.4 全连接层

4.1.5 Dropout 层

4.1.6 Batch\_Normalization

4.1.7 激活层

4.2 实验计划

4.2.1 Keras & Tensorflow

4.2.2 Pytorch

5. 使用工具

5.1 Keras & Tensorflow

5.2 Pytorch

5.3 GPU, CUDA 与 CUDNN

### 第三部分 系统实现步骤以及方法

6. 数据加载以及预处理

6.1 数据加载

6.1.1 Keras & Tensorflow

6.1.2 Pytorch

6.2 数据预处理

6.3 数据增强 (可选)

7. 建立模型

7.1 Keras & Tensorflow

7.2 Pytorch

8. 训练并保存模型

8.1 计算交叉熵

8.2 选择优化器

8.3 计算精确度

- 8.4 回调函数
  - 9. 使用训练好的模型进行预测
    - 9.1 使用摄像头进行实时识别
    - 9.2 对测试图片进行批量预测
  - 第四部分 试验结果及结果评估
    - 10. 模型训练结果
      - 10.1 Keras & Tensorflow训练结果
        - 10.1.1 使用DataShuffleSplit训练
        - 10.1.2 使用KFold交叉验证训练
      - 10.2 Pytorch
    - 11. 模型测试结果
      - 11.1 Keras & Tensorflow
        - 11.1.1 图片测试效果
        - 11.1.2 实时视频流识别效果
      - 11.2 Pytorch
    - 12. 结果评估
  - 第五部分 实验反思以及未来计划
    - 13.1 实验反思
    - 13.2 未来计划
  - 参考文献
- 

## 第一部分 环境配置

---

### 1. ubuntu 与 python虚拟环境的配置

---

操作系统：Ubuntu 16.04

语言版本：Python 3.6.5 (Virtualenv)

需配置前端框架：Keras, Tensorflow-gpu, Pytorch

需配置Python依赖：Numpy, Scikit-learn, matplotlib, opencv, PIL, h5py

详细的配置方法以及运行指令见项目目录下的README.md文件

### 2. CUDA 与 CUDNN 的配置

---

CUDA和CUDNN用来对训练加速，可以提升10倍左右的训练速度

详细安装教程，可能出现的问题以及错误解决见我之前写过的[wiki栏目](#)

内容包含在ubuntu下安装显卡驱动， CUDA， CUDNN以及源码编译安装  
Opencv 3.3.1， 源码安装Caffe以及Darknet。

在环境配置上有问题请联系：[hilbertxu@outlook.com](mailto:hilbertxu@outlook.com)

## 第二部分 人脸识别系统的原理以及使用工具

### 3. 实验目的

---

学习对大规模数据进行预处理的方法，以及如何更高效的装载训练数据

学习深度学习框架（keras， tensorflow， pytorch）的使用，如何搭建神经网络模型，如何使用训练数据训练模型，如何对模型进行苹果

学习如何使用训练好的模型对样本进行预测

学习如何使用训练好的模型对于摄像头中的视频流进行实时识别

### 4. 系统工作原理

---

#### 4.1 卷积神经网络层级结构

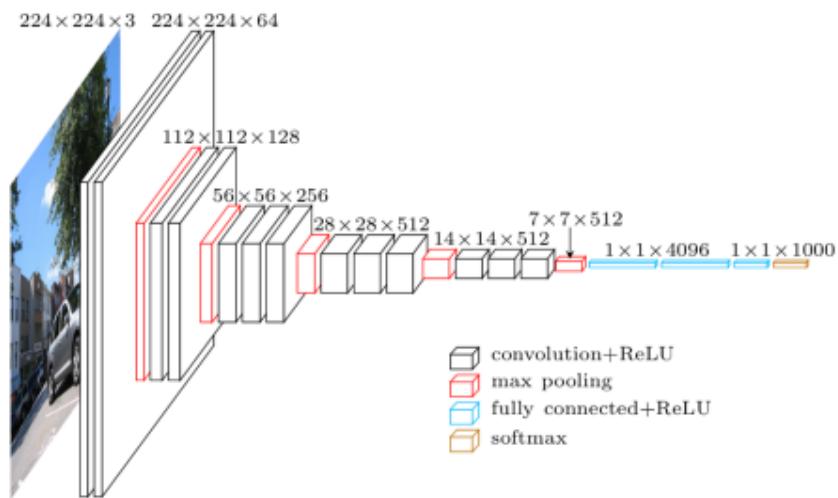
我的识别系统使用了迁移学习能力较强的VGG-16网络，其网络结构如下所示：

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input ( $224 \times 224$ RGB image)					
conv3-64	conv3-64 LRN	conv3-64 <b>conv3-64</b>	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 <b>conv3-128</b>	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 <b>conv1-256</b>	conv3-256 conv3-256 <b>conv3-256</b>	conv3-256 conv3-256 conv3-256 <b>conv3-256</b>
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 conv3-512 <b>conv3-512</b>
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 conv3-512 <b>conv3-512</b>
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: Number of parameters (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

整个网络可以分成特征提取器和分类器两部分，特征提取器主要包括卷积层，池化层，卷积层用来提取图像的高维特征，池化层用来对输入的特征图进行压缩，一方面使特征图变小，简化网络计算复杂度；一方面进行特征压缩，提取主要特征。



#### 4.1.1 数据输入层

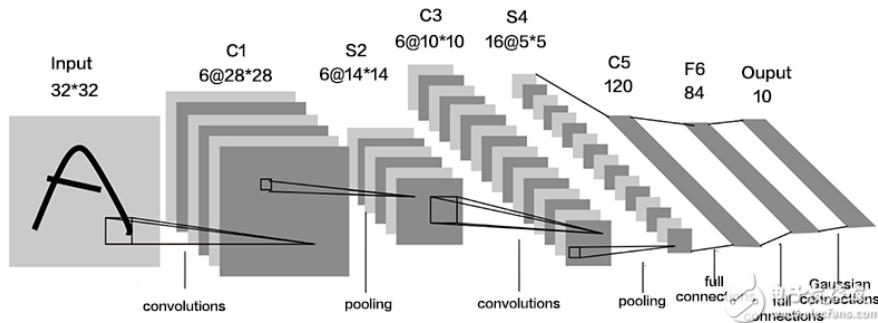
输入数据是一个Batch的图像数据

是一个四维张量,尺寸为 : [batch\_size, image\_size, image\_size, image\_channels]

#### 4.1.2 卷积层

卷积层中通过生成多个随机权重的卷积掩模对图像进行卷积操作来实现图像高维特征的提取。卷积层可以形象的理解为给图片加上滤镜，对于每一张图像，n个卷积核就相当于给原始图片上加上了n种滤镜，可以获取到原始图像的n个不同的feature maps。卷积神经网络中，通过多层卷积操作，可以提取出每一类图像的高维特征。

卷积层中每个卷积核(神经元)都与前一层中位置接近的区域中的多个神经元相连，区域的大小取决于卷积核的大小，在文献中被称为“感受野”。卷积核在工作时，会有按照给定的移动步长规律地扫过输入特征，在感受野内对输入特征做矩阵元素乘法求和并叠加偏差量。

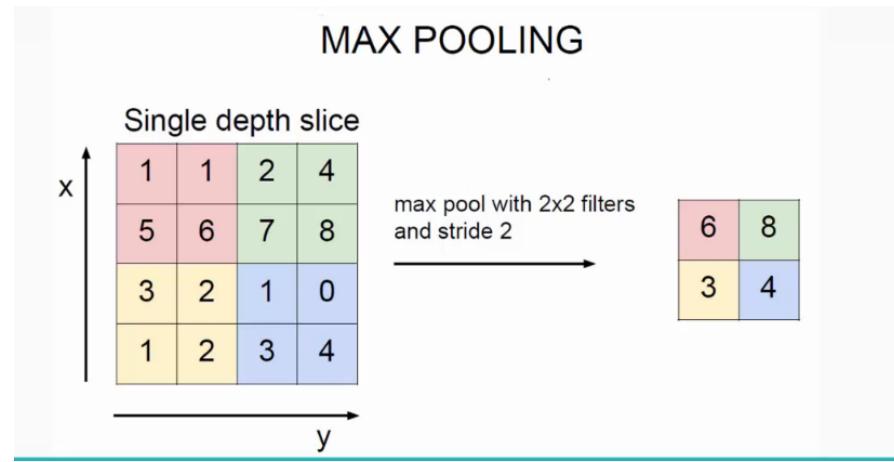


在VGG-16网络中全部使用33卷积核、22池化核，不断加深网络结构来提升性能，其优点有：

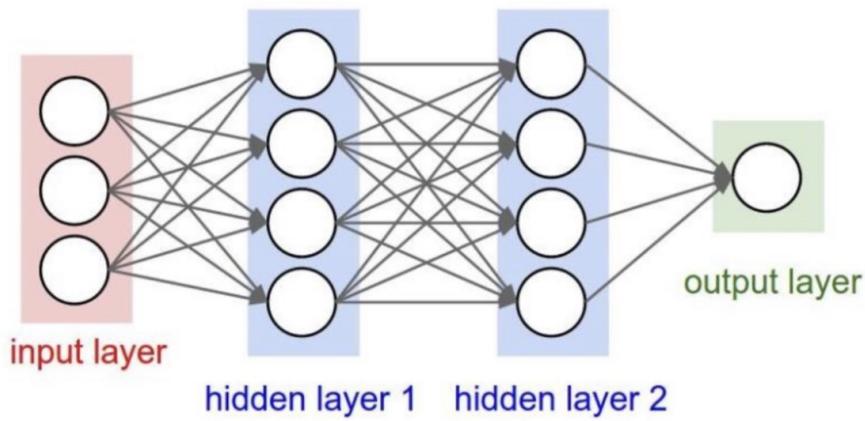
- (1) 多个一样的 $3 \times 3$ 的卷积层堆叠非常有用
- (2) 两个 $3 \times 3$ 的卷积层串联相当于1个 $5 \times 5$ 的卷积层，即一个像素会跟周围55的像素产生关联，感受野大小为55。
- (3) 三个 $3 \times 3$ 的卷积层串联相当于1个 $7 \times 7$ 的卷积层，但3个串联的 $3 \times 3$ 的卷积层有更少的参数量，有更多的非线性变换（3次ReLU激活函数），使得CNN对特征的学习能力更强。

#### 4.1.3 池化层

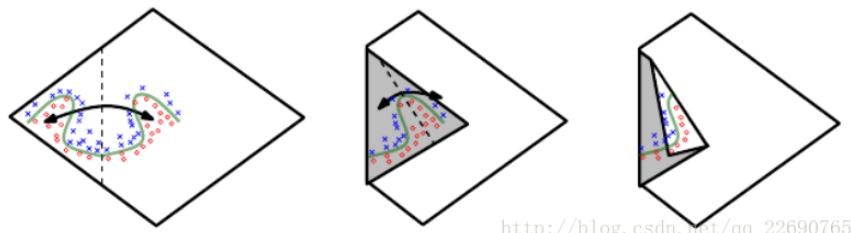
池化层一般接在卷积层之后，用于对输入的特征图进行压缩，一方面使特征图变小，简化网络计算复杂度；一方面进行特征压缩，提取主要特征。一般我们使用的是MaxPooling，即最大池化：在每一个池化区域中找最大值来代替这个区域的特征。



#### 4.1.4 全连接层



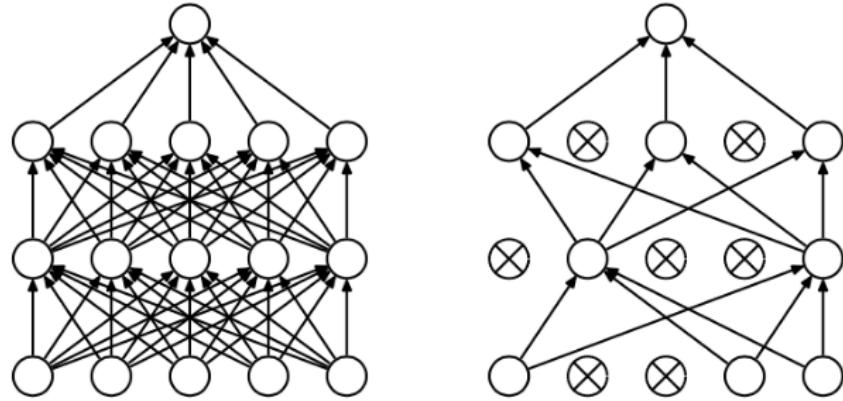
在神经网络中，全连接层的主要作用是分类。当数据是线性可分数据的时候，只需要一层隐藏层就可以实现对数据的分类；而当数据是线性不可分的数据的时候，就需要使用隐藏层来进行分类，可以认为将原始输入数据，在每一层隐含层上做了多个二分类，二分类的个数即为该隐含层的神经元个数。



#### 4.1.5 Dropout层

在大规模的神经网络中有这样两个缺点：1. 费时；2. 容易过拟合。对于一个有  $N$  个节点的神经网络，有了 dropout 后，就可以看做是  $2^N$  个模型的集合了，但此时要训练的参数数目却是不变的，这就缓解了费时的问题。

Hinton在2014年发表的论文*Dropout: A Simple Way to Prevent Neural Networks from Overfitting* 中做了这样的类比，无性繁殖可以保留大段的优秀基因，而有性繁殖则将基因随机拆了又拆，破坏了大段基因的联合适应性，但是自然选择中选择了有性繁殖，物竞天择，适者生存，可见有性繁殖的强大。dropout 也能达到同样的效果，它强迫一个神经单元，和随机挑选出来的其他神经单元共同工作，消除减弱了神经元节点间的联合适应性，增强了泛化能力。



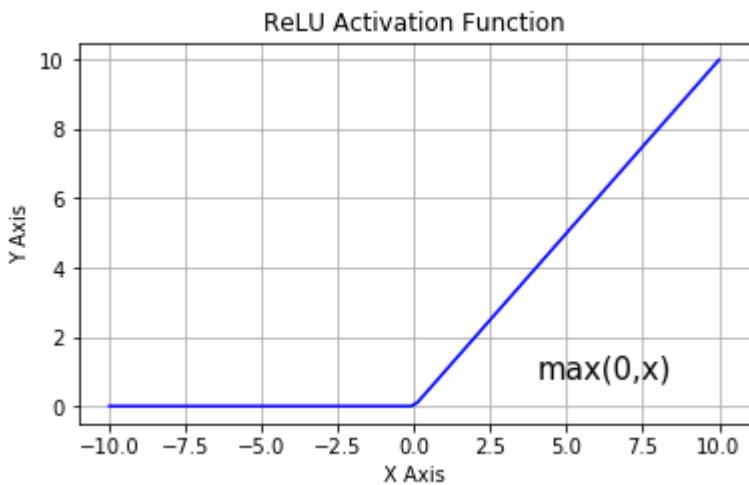
#### 4.1.6 Batch\_Normalization

我在Keras版本的训练代码中并没有加上Batch\_Normalization层，在Pytorch版本的代码中，对于每一层卷积层都加上了Batch\_Normalization层来加速巡礼那。结果十分显著，在同样地数据，同样地网络结构下，Keras大概需要20此左右的迭代，训练精度才能稳定在98%以上，而Pytorch大概需要7次迭代就可以达到98%以上的训练精度，加速效果十分显著。

Batch\_Normalization的基本思想其实相当直观：因为深层神经网络在做非线性变换前的激活输入值（就是那个 $y=Wx+B$ ， $x$ 是输入）随着网络深度加深或者在训练过程中，其分布逐渐发生偏移或者变动，之所以训练收敛慢，一般是整体分布逐渐往非线性函数的取值区间的上下限两端靠近（对于Sigmoid函数来说，意味着激活输入值 $Wx+B$ 是大的负值或正值），所以这导致反向传播时低层神经网络的梯度消失，这是训练深层神经网络收敛越来越慢的本质原因，而BN就是通过一定的规范化手段，把每层神经网络任意神经元这个输入值的分布强行拉回到均值为0方差为1的标准正态分布，其实就是把越来越偏的分布强制拉回比较标准的分布，这样使得激活输入值落在非线性函数对输入比较敏感的区域，这样输入的小变化就会导致损失函数较大的变化，意思是这样让梯度变大，避免梯度消失问题产生，而且梯度变大意味着学习收敛速度快，能大大加快训练速度。

#### 4.1.7 激活层

把卷积层输出结果作非线性映射。CNN采用的激活函数常为ReLU (The Rectified Linear Unit) / 修正线性单元，特点是收敛快，求梯度简单，但是较为脆弱。其函数图像如下：



## 4.2 实验计划

### 4.2.1 Keras & Tensorflow

1. 先对原始脸集进行数据预处理，对每张图片减去全部训练集图片的RGB均值，然后进行图像归一化操作。

2. 利用Keras搭建VGG-16卷积神经网络模型

3. 采取多种划分训练集和验证集的方法：

**手动划分**：使用sklearn中的StratifiedShuffleSplit API来划分，可以保证划分出来的训练集和验证集均保持原分布

**KFold交叉验证**：有两种方式，使用sklearn中的API来手动将数据集进行K折分割，以及使用keras中的sklearn wrapper进行K折验证

**网络超参数验证（GridSearch）**：这个方法的目的在于找到最合适的超参数（训练迭代次数epoch，批大小batch\_size，优化器...）

但是该方法现在没有很好的实现，会出现内存溢出的问题，是一个未完善的功能

d. 建立学生学号到训练标签的映射

e. 生成训练数据进入网络，开始训练模型。

f. 使用训练好的模型进行预测

### 4.2.2 Pytorch

1. 使用Pytorch自带的数据读取API进行数据读取，假设文件储存路径是/DataBase/student\_id/images，

**torchvision.datasets.ImageFolder()** 可以直接读取DataBse中的所有文件夹，并且生成每个文件夹的名称到训练标签的映射，同时还可以对图像进行水平翻转，剪切，去中心化等预处理操作，可以说是一个非常方便的文件读取工具了

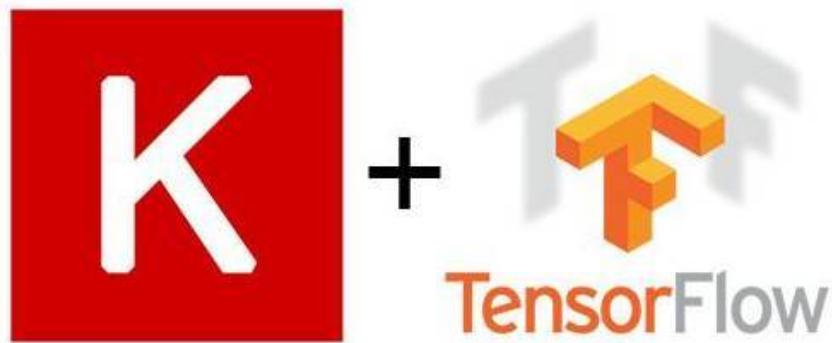
2. 将上一步生成的数据集输入网络进行训练

3. 使用训练好的模型进行预测

## 5. 使用工具

---

### 5.1 Keras & Tensorflow



Keras是一个高层神经网络API，由Python编写并且使用Tensorflow和Theano作为后端工作，使用Keras可以很方便的编写神经网络，训练网络和使用模型进行预测。但是Keras是Tensorflow的高层封装，并不能很好的理解Tensorflow的特性和工作方式，以及网络的原理；所以我在完成了Keras部分的系统之后，使用纯Tensorflow重新搭建了我的系统，使用更底层的函数进行了神经网络的搭建和训练工作（模型的搭建和训练代码在./tensorflow文件夹中，可以进行训练，由于模型的使用部分并没有区别，所以就只编写了tensorflow版本的模型搭建既训练部分），在完成了Tensorflow版本的代码编写之后，我对于神经网络中每一个网络层的结构组成以及工作原理有了更深的理解，也学习到了Tensorflow中计算图的概念。

### 5.2 Pytorch



在写Tensorflow部分的代码的时候，我认识到Tensorflow使用的是静态计算图，即我们需要先定义计算图，然后在每次运行的时候使用的都是同一个计算图。这样当输入需要变化的时候，就需要重构整个训练图，而且，在调试的时候也不能逐步调试，虽然一定程度上静态图节省了计算量，运行速度更快，但是仍然存在诸多不便。在了解到逐渐兴起的Pytorch使用的是动态计算图，即每次都会重构一个新的计算图，使用者能够用任何他们喜欢的方式进行debug，非常直观；同时Pytorch与Numpy及Python完全契合，语法和接口上具有很高的一致性，使用起来十分方便。之后当然复习任务繁重，但是我仍然对Pytorch产生了巨大兴趣，决定重新写一分Pytorch框架下的人脸识别系统。

### 5.3 GPU, CUDA 与 CUDNN

同时使用GPU和CPU，加快图形处理速度，GPU由上千个流处理器（core）作为运算器，采用单指令多线程（SIMT）模式。

CUDA（Compute Unified Device Architecture）是Nvidia公司推出的一种基于新的并行变成模型和指令集架构的通用计算架构，它能利用Nvidia公司推出的GPU的并行引擎进行计算，在处理图形的任务中，比CPU更加高效

CUDNN（The NVIDIA CUDA® Deep Neural Network library），是针对深层神经网络的显卡计算加速库。CUDA可以看作是一个工作台，上面配有很多工具，如锤子、螺丝刀等。cuDNN是基于CUDA的深度学习GPU加速库，有了它才能在GPU上完成深度学习的计算。它就相当于工作的工具。但是CUDA这个工作台买来的时候，并没有送扳手。想要在CUDA上运行深度神经网络，就要安装cuDNN，就像你想要拧个螺帽就要把扳手买回来。这样才能使GPU进行深度神经网络的工作，工作速度相较CPU快很多。

## 第三部分 系统实现步骤以及方法

### 6. 数据加载以及预处理

---

#### 6.1 数据加载

##### 6.1.1 Keras & Tensorflow

Keras版本的代码中我是直接采用了将数据全部读取到内存，然后分割成训练数据和验证数据，在分别进行预处理的方式，比较一般，而Tensorflow版本的代码中我在此基础上使用了Tensorflow自带的数据储存格式以及数据读取API，所以选择Tensorflow版本的代码来进行解释。

首先是读取数据以及生成相对应的标签。由于在训练时标签需要使用one-hot code，所以标签的编号必须是从0到class\_num-1，而在所有的类别中学号又不是连续的，因此需要生成一个学号到标签的映射表，我首先是找出所有学号中最小的一个，并将其作为标准值，然后将每个人的学号减去这个标准

值，得到相对偏移量，再对这个偏移量进行从小到大的排序，就生成了每个人学号相对与标准值的偏移量的顺序表，在这个表中每个人按照学号从小到大排序。之后每遍历一个以学号命名的文件夹时，只需要将文件夹名对应数字减去之前获得的标准值，得到偏移量之后在顺序表中查找偏移量对应的下标便是这个学号对应的训练标签，生成输出如下：

```
=====
Generating Order Sheet ||

=====

||          the original class labels is:           ||
=====

[1611455. 1611468. 1611440. 1611463. 1611436. 1611482. 1611436. 1711459.
1613378. 1611493. 1611433. 1611418. 1611446. 1611415. 1611470. 1611420.
1610763. 1611444. 1611465. 1611424. 1611412. 1611491. 1611443. 1611471.
1611409. 1611478. 1611490. 1611461. 1611447. 1611407. 1611419. 1611431.
1611453. 1611488. 1611487. 1611492. 1611450. 1611437. 1611417. 1611449.
1613550. 1611462. 1611438. 1611413. 1611483. 1611427. 1611260. 1611434.
1611458. 1611466. 1613376. 1611468. 1611421. 1611480. 1611464. 1611476.
1611426. 1611460. 1611467. 1611486. 1611473. 1611472.]



=====

the minimum class label is: 1610763.0 ||

=====

the total number of predefined classes is: 62 ||

=====

||          the relative class label sheet is:           ||
=====

[ 692.    645.    677.    700.    667.    719.    673.  100696.    2615.
 730.    670.    655.    683.    652.    707.    657.      0.    681.
 702.    661.    649.    728.    680.    708.    646.    715.    727.
 698.    684.    644.    656.    668.    690.    725.    724.    729.
 687.    674.    654.    686.   2787.    699.    675.    650.    720.
 664.    497.    671.    695.    703.   2613.    705.    658.    717.
 701.    713.    663.    697.    704.    723.    710.    709.]



=====

||          generated order sheet:           ||
=====

0.0    497.0    644.0    645.0    646.0    649.0    650.0    652.0
654.0    655.0    656.0    657.0    658.0    661.0    663.0    664.0
667.0    668.0    670.0    671.0    673.0    674.0    675.0    677.0
680.0    681.0    683.0    684.0    686.0    687.0    690.0    692.0
695.0    697.0    698.0    699.0    700.0    701.0    702.0    703.0
704.0    705.0    707.0    708.0    709.0    710.0    713.0    715.0
717.0    719.0    720.0    723.0    724.0    725.0    727.0    728.0
729.0    730.0   2613.0   2615.0   2787.0  100696.0



=====

Find All images & Generate Labels ||
```

代码实现如下：

```

def Generate_OrderSheet (self):
    ...
    初始化得到的标签列表元素为string类型
    首先将其转化为np.array
    然后根据得到的np.array中元素的大小，从小到大排序得到一张次序表
    然后遍历np.array，每一个元素在次序表中搜索对应，并用次序表中这个元素的下标
    来取代np.array中该元素的值
    这样就可以把一个string类型的列表转化为一个one-hot编码方式所需求得顺序np.array数组
    ...
    #using os operation to get the number of classes
    print ("====")
    print ('||          the original class labels is:           ||')
    print ("====")
    #print labels
    folders = os.listdir (self.DATASET_ROOT_DIR)
    folders = np.asarray (folders, dtype = np.float32)
    print (folders)
    print ("=====\n")

    self.minimun_id = min(folders)
    print ("====")
    print ('the minimum class label is: ' + str(self.minimun_id) + '||')
    print ("=====\\n")

    class_num = len(folders)
    self.predefined_class = class_num
    print ("====")
    print ('the total number of predefined classes is: ' + str(class_num) + '||')
    print ("=====\\n")

    for folder in folders:
        temp = int(folder) - self.minimun_id
        self.relative_sheet = np.append (self.relative_sheet, temp)
    print ("====")
    print ('||          the relative class label sheet is:           ||')
    print ("====")
    print (self.relative_sheet)
    print ("=====\\n")

    self.order_sheet = Bubble_Sort (self.relative_sheet)
    print ("====")
    print ('||          generated order sheet:           ||')
    print ("====")
    print_matrix(self.order_sheet)
    print ("=====\\n")

```

在完成数据读取之后，为了之后调用方便，我将数据存储在Tensorflow提供的标准数据储存格式**TFRecords**文件中。TFRecords文件中储存的是经Tensorflow序列化之后的图像数据和标签数据，能够很大程度上压缩数据储存空间，并且调用的时候相当于调用裸数据流，只需要用对应的解码方式解码即可。（详细可见Tensorflow文件夹下的load\_data.py 以及 train.py文件，此处就不再罗列）

在训练时我们就可以直接使用Tensorflow中的接口

`tf.data.TFRecordDataset` 直接读取预生成好的TFReocrds文件，并读取成一个tf.dataset类的对象，再通过类函数对这个数据集中的数据进行解码和预处理。

```

def _parse_tfrecord(example_proto):
    # 定义解析用的字典
    features = {
        'image_raw': tf.FixedLenFeature(shape=[], dtype=tf.string),
        'label': tf.FixedLenFeature(shape=[1], dtype=tf.int64),
        'height': tf.FixedLenFeature(shape=[1], dtype=tf.int64),
        'width': tf.FixedLenFeature(shape=[1], dtype=tf.int64),
        'channels': tf.FixedLenFeature(shape=[1], dtype=tf.int64)
    }
    # 调用接口解析一行样本
    parsed_data = tf.parse_single_example(serialized=example_proto, features=features)
    # 获取图像基本属性
    label, height, width, channels = parsed_data['label'], parsed_data['height'], parsed_data['width'], parsed_data['channels']
    # 获取原始图像，并进行转化
    image_decoded = tf.decode_raw(parsed_data['image_raw'], tf.uint8)
    ...
    读取 TFRecord 文件过程中，解析 Example Protobuf 文件时，  

    decode_raw 得到的数据(如 image raw data) 要通过 reshape 操作恢复 shape，  

    而 shape 参数也是从 TFRecord 文件中获取时，要加 tf.stack 操作：image = tf.reshape(image, tf.stack([height, width, channels]))
    ...
    image_decoded = tf.reshape(image_decoded, tf.stack([height, width, channels]))
    # 数据增强
    image_decoded = tf.image.random_flip_left_right(image_decoded)
    #image_decoded = tf.image.per_image_standardization(image_decoded)
    # 图像归一化
    image_decoded = tf.cast(image_decoded, tf.float32)/255.0
    ...
    # 处理图像对应的标签，将其转化为 one-hot code
    label = tf.cast(label, tf.int32)
    label = tf.one_hot(label, depth=class_num, on_value=1)
    return image_decoded, label

def read_train_valid_tfrecord():
    train_file_path = os.path.abspath(os.path.join(TFRECORD_DIR, "train.tfrecords"))
    valid_file_path = os.path.abspath(os.path.join(TFRECORD_DIR, "valid.tfrecords"))

    print ("*[INFO] Reading dataset from .tfrecord files")
    # 定义 train dataset 和对应的迭代器
    dataset_train = tf.data.TFRecordDataset(train_file_path)
    # shuffle buffer 大小设置为所有训练数据的数量，使整个训练集全部打乱
    dataset_train = dataset_train.shuffle(train_data_size)
    dataset_train = dataset_train.map(_parse_tfrecord)
    dataset_train = dataset_train.batch(BATCH_SIZE, drop_remainder=True)
    dataset_train = dataset_train.prefetch(5)
    print ("Batch Size is: %d" % (BATCH_SIZE))

    # 创建 validation dataset 和对应的迭代器
    dataset_valid = tf.data.TFRecordDataset(valid_file_path)
    dataset_valid = dataset_valid.map(_parse_tfrecord)
    dataset_valid = dataset_valid.batch(BATCH_SIZE, drop_remainder=True)
    You, 29 days ago · The last day of 2018
    return dataset_train, dataset_valid

```

### 6.1.2 Pytorch

pytorch中的数据读取十分方便，默认的数据读取方式可以读取每一个类别的文件夹中的所有图片，并直接生成学号到训练标签（0 - (N-1)）的映射（实际效果和我自己手动实现的一致）。同时可以直接生成训练批次（Batch）

```

def load_train_data(path=TRAIN_DATASET):
    trainset = torchvision.datasets.ImageFolder(path,
                                                transform=transforms.Compose([
                                                    transforms.Resize((64, 64)),
                                                    transforms.CenterCrop(64),
                                                    transforms.RandomHorizontalFlip(),
                                                    transforms.ToTensor(),
                                                    # 此处训练图像的均值和标准差由 utils.py 中函数算得
                                                    transforms.Normalize((0.5229, 0.4287, 0.3647), (0.2120, 0.1877, 0.1734)),]))
    train_loader = torch.utils.data.DataLoader(
        trainset, batch_size=64,
        shuffle=True, num_workers=4
    )
    return train_loader
    You, 7 days ago · add pytorch part

```

## 6.2 数据预处理

VGG-16 网络对数据预处理的要求较低，我使用的预处理方法只有去中心化，归一化：即减去图像的RGB均值，并将像素值归一到 (-0.5 , 0.5) 的范围内。归一化的操作尤为重要，许多情况下缺少归一化会导致网络训练是loss不收敛。

### 6.3 数据增强（可选）

数据增强主要是用于在数据量较小时，人工通过改变一系列图像属性来生成大量数据。这样可以在原始数据较少时仍然能够使网络有较强的泛化能力以及避免过拟合现象。以下是常用数据增强方法（以Keras为例，详见[keras/src/train\\_model.py](#)）：

```

#使用实时数据提升
else:
    #定义数据生成器用于数据提升，其返回一个生成器对象datagen, datagen每被调用一次
    #次其生成一组数据（顺序生成），节省内存，其实就是python的数据生成器
    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, #是否对数据标准化（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

    #计算整个训练样本集的数量以用于特征值归一化、ZCA白化等处理
    datagen.fit(self.train_image)

#define callback funcs
#在使用shuffleSplit划分训练集的时候使用EarlyStopping来检测
#val_loss是否不再下降，也可以防止过拟合现象

#利用生成器开始训练模型
hist = self.VGG_16.model.fit_generator(datagen.flow(self.train_image, self.train_label,
                                                    batch_size = self.batch_size),
                                       samples_per_epoch = self.train_image.shape[0],
                                       nb_epoch = self.nb_epoch,
                                       validation_data = (self.valid_image, self.valid_label),
                                       callbacks=self.callbacks)

```

## 7. 建立模型

---

### 7.1 Keras & Tensorflow

在Keras和Tensorflow中建立模型的方式比较一致，毕竟Keras还是运行在Tensorflow后端上的。两者都是建立一个顺序模型，然后向模型中添加网络层，除输入层外每一层的输入都是上一层的输出，输入层的输入是每一个批次的图像数据，模型最后会输出分类的结果。Keras代码实现如下（Tensorflow版本的见/tensorflow/src/model.py）：

```

def model_for_scikit(self):
    #when you use different Dataset
    #remember to change input_shape, nb_classes
    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(64, 3, 3, border_mode='same',
                                input_shape = (64,64,3)))
    self.model.add(Activation('relu'))
    self.model.add(Convolution2D(64, 3, 3, border_mode='same'))
    self.model.add(Activation('relu'))
    self.model.add(MaxPooling2D(pool_size=(2, 2)))
    self.model.add(Dropout(0.25))

    self.model.add(Convolution2D(128, 3, 3, border_mode='same'))
    self.model.add(Activation('relu'))
    self.model.add(Convolution2D(128, 3, 3, border_mode='same'))
    self.model.add(Activation('relu'))
    self.model.add(MaxPooling2D(pool_size=(2, 2)))
    self.model.add(Dropout(0.25))

    self.model.add(Convolution2D(256, 3, 3, border_mode='same'))
    self.model.add(Activation('relu'))
    self.model.add(Convolution2D(256, 3, 3, border_mode='same'))
    self.model.add(Activation('relu'))
    self.model.add(MaxPooling2D(pool_size=(2, 2)))
    self.model.add(Dropout(0.25))

    self.model.add(Convolution2D(512, 3, 3, border_mode='same'))
    self.model.add(Activation('relu'))
    self.model.add(Convolution2D(512, 3, 3, border_mode='same'))
    self.model.add(Activation('relu'))
    self.model.add(MaxPooling2D(pool_size=(2, 2)))
    self.model.add(Dropout(0.25))

    self.model.add(Convolution2D(512, 3, 3, border_mode='same'))
    self.model.add(Activation('relu'))
    self.model.add(Convolution2D(512, 3, 3, border_mode='same'))
    self.model.add(Activation('relu'))
    self.model.add(MaxPooling2D(pool_size=(2, 2)))
    self.model.add(Dropout(0.25))

    self.model.add(Flatten())
    self.model.add(Dense(4096))
    self.model.add(Activation('relu'))
    self.model.add(Dropout(0.5))

    self.model.add(Dense(4096))
    self.model.add(Activation('relu'))
    self.model.add(Dropout(0.5))

    #nb_classes
    self.model.add(Dense(62))
    self.model.add(Activation('softmax'))
    #plot_model(self.model, to_file='../model/VGG-16.png', show_shapes=True)

```

## 7.2 Pytorch

Pytorch中并不是一个完整的网络，他将网络分为了特征提取层（features）和分类层（classifier），个人认为这样很好地体现了网络中各个部分的功能，很有助于理解。特征提取层的输入是原始训练图像数据，输出是提取到的图像特征，而分类层的输入特征提取层提取到的特征，输出是分类结果。以下是代码实现如下：

```

#Class vgg inheritance from torch.nn.Module
You, 5 days ago | 1 author (You)
class VGG(torch.nn.Module):
    def __init__(self, vgg_name):
        #调用父类的构造函数
        super(VGG, self).__init__()
        #定义网络中的特征提取层, 即卷积层
        self.features = self._make_layers(cfg[vgg_name])
        #定义网络中的分类层, 即全链接层
        #使用dropout层来缓解过拟合现象
        self.classifier = torch.nn.Sequential(
            #fc6
            torch.nn.Linear(512*2*2, 4096),
            torch.nn.ReLU(),
            torch.nn.Dropout(0.5),

            #fc7
            torch.nn.Linear(4096, 4096),
            torch.nn.ReLU(),
            torch.nn.Dropout(0.5),

            #fc8
            torch.nn.Linear(4096, CLASS_NUM)
        )
        self._initialize_weight()

    #生成网络层
    def _make_layers(self, cfg):
        layers = []
        input_channels = 3
        for layer in cfg:
            if layer == 'M':
                layers += [torch.nn.MaxPool2d(kernel_size=2, stride=2)]
            else:
                layers += [torch.nn.Conv2d(input_channels, layer, kernel_size=3, padding=1),
                           torch.nn.BatchNorm2d(layer),
                           torch.nn.ReLU(inplace=True)]
                input_channels = layer
        #在网络层最后加上一个均值池化层
        layers += [torch.nn.AvgPool2d(kernel_size=1, stride=1)]
        #将储存网络层的列表转化为torch中的Sequential模型
        return torch.nn.Sequential(*layers)

    #初始化网络层中的权重
    def _initialize_weight(self):
        for m in self.modules():
            if isinstance(m, torch.nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
                if m.bias is not None:
                    m.bias.data.zero_()
            elif isinstance(m, torch.nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()
            elif isinstance(m, torch.nn.Linear):
                m.weight.data.normal_(0, 0.01)
                m.bias.data.zero_()

    #定义训练中的前向传播过程
    def forward(self, input_batch):
        features_op = self.features(input_batch)
        features_op = features_op.view(features_op.size(0), -1)
        classify_op = self.classifier(features_op)
        return classify_op

```

## 8. 训练并保存模型

---

各个框架下模型的训练部分思路都一致，在此就不一一赘述，只是介绍一下训练的思路。

### 8.1 计算交叉熵

一张图片进入模型后，模型的输出在经过Softmax层进行归一化之后，得到的是一个N维的数组，N是训练数据的类别数。这个数组中第*i*个元素的值，就是这张图片对应第*i*类的可能性（0-1），而这张图片的标签同样是一个N维的数组，标签数组中只有一个元素为1，其余都是0，这个元素的位置就是这张图片对应的类别。

交叉熵计算公式如下：

$$H(p, q) = - \sum_{i=1}^n p(x_i) \log(q(x_i))$$

代码实现如下（Tensorflow的代码实现最为直观，故以其为例）：

```
def loss_op(logits, label_batches, regular):
    """
    tf.nn.softmax_cross_entropy_with_logits(记为f1)
    tf.nn.softmax_cross_entropy_with_logits_v2(记为f2)
    tf.nn.sparse_softmax_cross_entropy_with_logits(记为f3)
    之间的区别。
    f1和f3对于参数logits的要求都是一样的，即“未经处理的，直接由神经网络输出的数值(最后一层全连接层的输出)”，
    比如 [3.5, 2.1, 7.89, 4.4]。两个函数不一样的地方在于labels格式的要求，f1的要求labels的格式和logits类似，比如[0, 0, 1, 0]。
    而f3的要求labels是一个数值，这个数值记录着ground truth所在的索引。以[0, 0, 1, 0]为例，这里真值1的索引为2。
    所以f3要求labels的输入为数字2(tensor)。一般可以用tf.argmax()来从[0, 0, 1, 0]中取得真值的索引。
    f1和f2之间很像，实际上官方文档已经标记出f1已经是deprecated 状态，推荐使用f2。
    两者唯一的区别在于f1在进行反向传播的时候，只对logits进行反向传播，labels保持不变。而f2在进行反向传播的时候，
    同时对logits和labels都进行反向传播，如果将labels传入的tensor设置为stop_gradients，就和f1一样了。
    ...
    You, a few seconds ago • Uncommitted changes

    with tf.variable_scope('loss_op', reuse=tf.AUTO_REUSE):
        cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits=logits, labels=label_batches)
        cost = tf.reduce_mean(cross_entropy)
    return cost
```

## 8.2 选择优化器

每个Batch的图片进入到网络之后我们可以计算得到这个Batch的分类输出的平均Loss，而我们希望的就是网络预测地尽可能准确，反映到Loss上就是Loss尽可能地小。这个时候就需要用到优化器来对网络中的参数（权重，偏置）进行调整。优化器的最终目标是寻找到全局最优解，即最优的网络参数，在提出了Adam算法之后，很多人会直接选用自适应学习率的Adam优化器，但是事实上根据我的多次实验，在图像多分类问题中，效果较好的优化器应该是：

带动量的随机梯度下降优化器，即SGD（Stochastic Gradient Descent）+ Momentum。基本策略可以理解为随机梯度下降像是一个盲人下山，不用每走一步计算一次梯度，但是他总能下到山底，只不过过程会显得扭扭曲曲。使用动量（Momentum）的随机梯度下降算法（SGD）主要思想是引入了一个积攒历史梯度信息动量来加速SGD计算过程。

自适应学习率优化算法中的Adam算法，Adam中动量直接并入了梯度一阶矩（指数加权）的估计。其次，相比与缺少修正因子导致二阶矩估计可能在训练初期具有很高偏置的RMSProp，Adam包括偏置修正，修正从原点初始化的一阶矩（动量项）和（非中心的）二阶矩估计。

在Keras中的实现分别如下：

```
#optimizers
sgd = SGD(lr = 0.01, decay = 1e-6,
           momentum = 0.9, nesterov = True) #采用SGD+momentum的优化器进行训练，首先生成一个优化器对象
adam = optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)
```

### 8.3 计算精确度

虽然精确度是我们衡量模型好坏很重要的标准，但是在训练过程中并未使用到精确度。精确度的计算也比较简单可直观的表视为：

**Acc** = 正确识别样本数/样本总数

### 8.4 回调函数

在Keras版本的代码中我引入了一系列的回调函数，来可视化训练过程，记录训练过程，调整学习率，以及实现EarlyStoping功能。

```
#early stopping
#10次迭代val_loss不下降则停止训练
early_stopping = EarlyStopping(monitor='val_loss', patience=10, verbose=1, mode='auto')

#使用Checkpoint来记录训练过程
checkpoint = ModelCheckpoint(
    CHECK_POINT, monitor='val_acc', verbose=1,
    save_best_only=True, save_weights_only=False,
    mode='auto', period=10
)
#使用ReduceLROnPlateau在val_loss不再下降的时候降低学习率
reducclr = ReduceLROnPlateau(
    monitor='val_loss', factor=0.1,
    patience=10, verbose=0, mode='auto',
    epsilon=0.0001, cooldown=0, min_lr=0
)
global LOG_DIR
#使用Tensorboard可视化训练过程
tensorboard = TensorBoard(
    log_dir='LOG_DIR', histogram_freq=0, batch_size=64, write_graph=True,
    write_grads=False, write_images=False, embeddings_freq=0,
    embeddings_layer_names=None, embeddings_metadata=None
)

self.callbacks = [tensorboard, early_stopping, checkpoint, reducclr]
```

## 9. 使用训练好的模型进行预测

---

### 9.1 使用摄像头进行实时识别

基本思路是先使用OpenCV中的人脸分类器将视频流每一帧中的人脸截取出来，然后作为输入数据输进网络中进行预测，然后得到预测结果

### 9.2 对测试图片进行批量预测

对图片进行批量预测可以类比训练的过程，只是在测试的时候模型参数固定使用训练好的参数不再变化。

注意！

在使用模型进行预测时，需要保证输入图片的测试数据经过与训练数据一样的数据预处理。否则可能出现精度较低的情况

## 第四部分 试验结果及结果评估

### 10. 模型训练结果

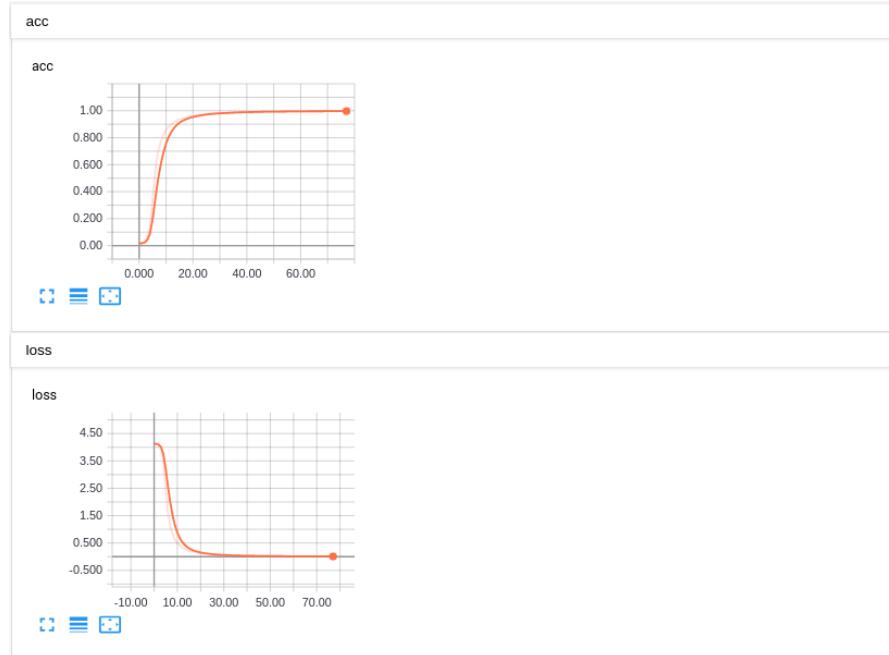
#### 10.1 Keras & Tensorflow训练结果

##### 10.1.1 使用DataShuffleSplit训练

在加入EarlyStopping机制以及学习率衰减机制之后模型在60次迭代之后停止了训练，以下是训练结果输出：

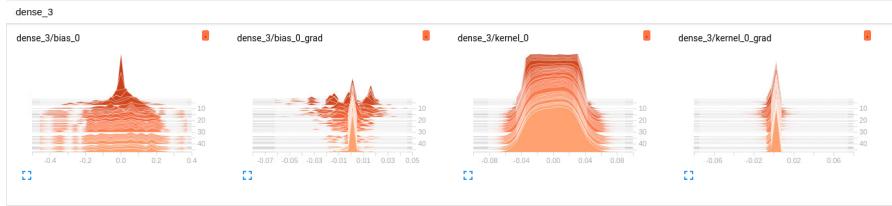
```
Epoch 699/70: val_acc improved from 0.98393 to 0.98409, saving model to /home/kamerider/machine_learning/face_recognition/keras/checkpoint/Shufflesplit/model_70-0.98.npz
Epoch 71/280
[=====] - 62s 1ms/step - loss: 0.6111 - acc: 0.9986 - val_loss: 0.6886 - val_acc: 0.9853
[=====] - 62s 1ms/step - loss: 0.6099 - acc: 0.9971 - val_loss: 0.6724 - val_acc: 0.9858
[=====] - 64s 1ms/step - loss: 0.6099 - acc: 0.9971 - val_loss: 0.6724 - val_acc: 0.9858
Epoch 73/280
[=====] - 62s 1ms/step - loss: 0.6099 - acc: 0.9971 - val_loss: 0.6749 - val_acc: 0.9849
[=====] - 62s 1ms/step - loss: 0.6099 - acc: 0.9971 - val_loss: 0.6749 - val_acc: 0.9849
[=====] - 65s 1ms/step - loss: 0.6092 - acc: 0.9972 - val_loss: 0.6730 - val_acc: 0.9850
Epoch 75/280
[=====] - 62s 1ms/step - loss: 0.6087 - acc: 0.9972 - val_loss: 0.6730 - val_acc: 0.9850
[=====] - 65s 1ms/step - loss: 0.6087 - acc: 0.9972 - val_loss: 0.6710 - val_acc: 0.9862
Epoch 76/280
[=====] - 64s 1ms/step - loss: 0.6111 - acc: 0.9964 - val_loss: 0.6692 - val_acc: 0.9865
[=====] - 64s 1ms/step - loss: 0.6111 - acc: 0.9964 - val_loss: 0.6692 - val_acc: 0.9865
[=====] - 65s 1ms/step - loss: 0.6077 - acc: 0.9975 - val_loss: 0.6813 - val_acc: 0.9837
Epoch 78/280
[=====] - 66s 2ms/step - loss: 0.6085 - acc: 0.9975 - val_loss: 0.6721 - val_acc: 0.9866
Epoch 699/70: early stopping
0/262 [=====] - 0s 6ms/step
acc: 100.0%
```

以下是通过tensorboard可视化的训练过程中的loss曲线和accuracy曲线：

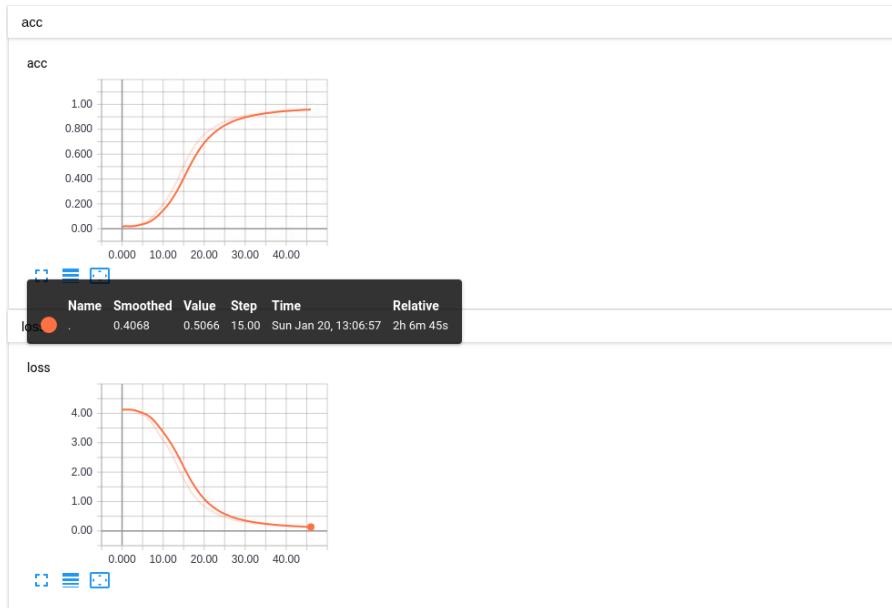


##### 10.1.2 使用KFold交叉验证训练

同样加入了EarlyStopping机制以及学习率衰减机制，以下是训练过程中最后一层全连接层（分类层）的参数更新情况：



以下是训练过程中loss和accuracy的变化曲线：



## 10.2 Pytorch

Pytorch中由于我使用了Batch\_Normalization层来加速训练，只需要10次左右的迭代就能获得很好的训练效果。以下是Pytorch的训练效果以及所使用的网络结构：

```

VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (9): ReLU(inplace)
    (10): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (12): ReLU(inplace)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (14): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (16): ReLU(inplace)
    (17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (19): ReLU(inplace)
    (20): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (26): ReLU(inplace)
    (27): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (29): ReLU(inplace)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (32): ReLU(inplace)
    (33): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (36): ReLU(inplace)
    (37): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (38): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (39): ReLU(inplace)
    (40): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (42): ReLU(inplace)
    (43): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (44): AvgPool2d(kernel_size=1, stride=1, padding=0)
  )
  (classifier): Sequential(
    (0): Linear(in_features=2048, out_features=4096, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=62, bias=True)
  )
)

[EPPOCH 1/10] val_loss: 0.083 val_acc: 0.941====> Step: 971ms 671/679 1ms | Loss: 0.281 | Acc: 96.875% ((2/64)64)))
[EPPOCH 2/10] val_loss: 0.014 val_acc: 0.724====> Step: 914ms 671/679 1ms | Loss: 0.789 | Acc: 81.250% ((52/64)64))
[EPPOCH 3/10] val_loss: 0.006 val_acc: 0.884====> Step: 953ms 671/679 1ms | Loss: 0.478 | Acc: 98.625% ((58/64)64))
[EPPOCH 4/10] val_loss: 0.005 val_acc: 0.915====> Step: 916ms 671/679 1ms | Loss: 0.186 | Acc: 92.188% ((59/64)64))
[EPPOCH 5/10] val_loss: 0.003 val_acc: 0.940====> Step: 916ms 671/679 1ms | Loss: 0.138 | Acc: 96.875% ((62/64)64))
[EPPOCH 6/10] val_loss: 0.003 val_acc: 0.951====> Step: 945ms 671/679 1ms | Loss: 0.115 | Acc: 96.875% ((62/64)64))
[EPPOCH 7/10] val_loss: 0.002 val_acc: 0.955====> Step: 902ms 671/679 1ms | Loss: 0.098 | Acc: 97.188% ((63/64)64))
[EPPOCH 8/10] val_loss: 0.002 val_acc: 0.955====> Step: 902ms 671/679 1ms | Loss: 0.098 | Acc: 98.438% ((63/64)64))
[EPPOCH 9/10] val_loss: 0.002 val_acc: 0.973====> Step: 930ms 671/679 1ms | Loss: 0.046 | Acc: 96.875% ((62/64)64))
[EPPOCH 10/10] val_loss: 0.001 val_acc: 0.978====> Step: 967ms 671/679 1ms | Loss: 0.027 | Acc: 100.000% ((64/64)64))

Finished Training

```

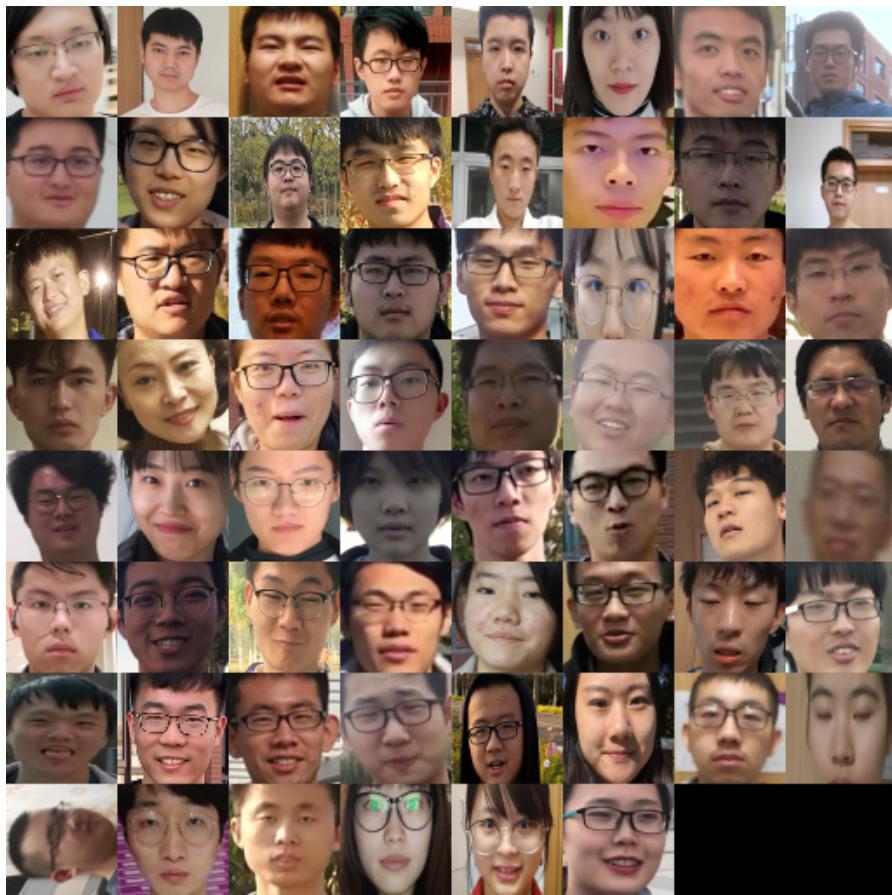
## 11. 模型测试结果

### 11.1 Keras & Tensorflow

选用效果最好的Kfold交叉验证训练得到的模型进行测试

#### 11.1.1 图片测试效果

以下是测试图片图片和识别结果：



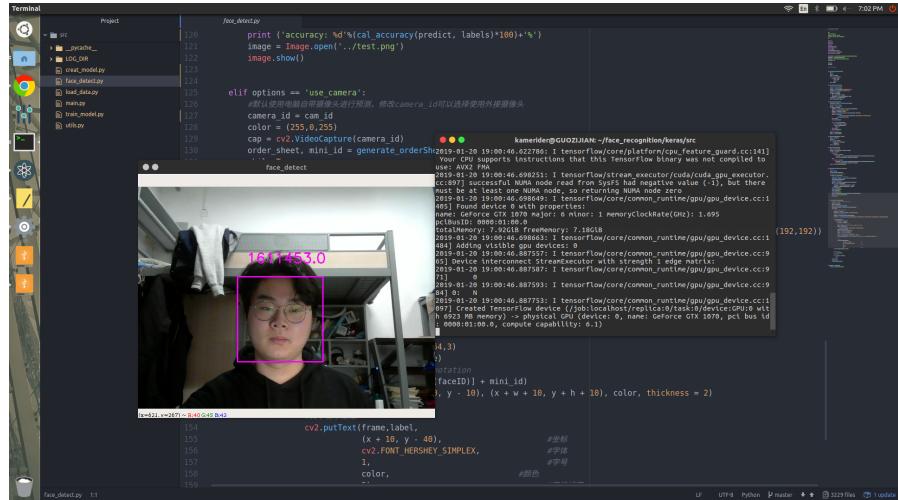
```
[predict]
1611455 1611408 1611440 1611463 1611430 1611482 1611436 1711459
1613378 1611493 1611433 1611418 1611446 1611415 1611470 1611420
1610763 1611444 1611465 1611424 1611412 1611491 1611443 1611471
1611409 1611478 1611490 1611461 1611447 1611407 1611419 1611431
1611453 1611488 1611487 1611492 1611450 1611437 1611417 1611449
1613550 1611462 1611438 1611413 1611483 1611427 1611260 1611434
1611458 1611466 1613376 1611468 1611421 1611480 1611464 1611476
1611426 1611460 1611467 1611486 1611473 1611472

[truth]
1611455 1611408 1611440 1611463 1611430 1611482 1611436 1711459
1613378 1611493 1611433 1611418 1611446 1611415 1611470 1611420
1610763 1611444 1611465 1611424 1611412 1611491 1611443 1611471
1611409 1611478 1611490 1611461 1611447 1611407 1611419 1611431
1611453 1611488 1611487 1611492 1611450 1611437 1611417 1611449
1613550 1611462 1611438 1611413 1611483 1611427 1611260 1611434
1611458 1611466 1613376 1611468 1611421 1611480 1611464 1611476
1611426 1611460 1611467 1611486 1611473 1611472

accuracy: 100%
```

### 11.1.2 实时视频流识别效果

可以识别出来我，并且正确显示了学号



```
accuracy: 100%
image = Image.open("./test.png")
image.show()

elif options == "use_camera":
    camera_id = 0
    cols = (255, 255)
    cap = cv2.VideoCapture(camera_id)
    order_sheet, mini_id = generate_order_sheet()
    face_detect

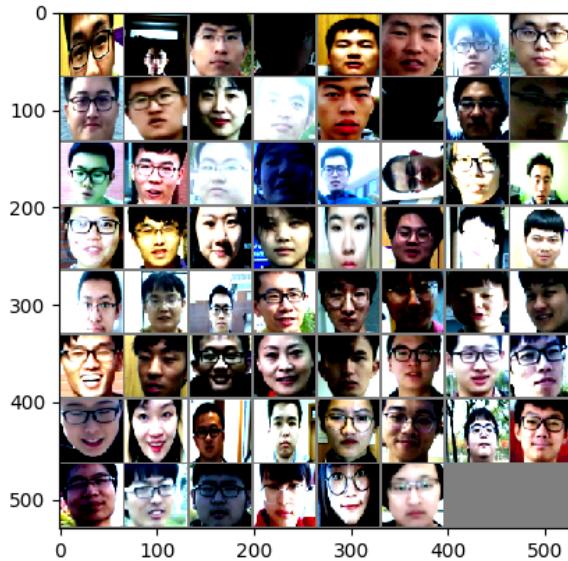
cv2.putText(frame,label,
           (x + 10, y + 40),
           cv2.FONT_HERSHEY_SIMPLEX,
           1,
           color,
           2)

cv2.imshow('frame',frame)
cv2.waitKey(1)
```

## 11.2 Pytorch

Pytorch中我只写了一个简单的测试图像识别demo，以下是使用Pytorch训练出来的模型进行识别的效果，Pytorch中提供了很方便的显示多张图片的API：

测试用图片：



识别结果：

```
[Truth]:  
tensor([22, 60, 43, 42, 23, 24, 36, 40, 59, 25, 53, 20, 7, 37, 17, 6, 34, 39,  
2, 50, 61, 14, 57, 26, 54, 9, 48, 56, 46, 30, 0, 3, 28, 10, 11, 58,  
33, 55, 32, 8, 5, 1, 15, 47, 4, 21, 41, 29, 44, 49, 12, 16, 52, 35,  
18, 38, 27, 19, 13, 45, 51, 31])  
[Predict]:  
tensor([22, 60, 43, 42, 23, 24, 36, 40, 59, 25, 53, 20, 7, 37, 17, 6, 34, 39,  
2, 50, 61, 14, 57, 26, 54, 9, 48, 56, 46, 30, 0, 3, 28, 10, 11, 58,  
33, 55, 32, 8, 5, 1, 15, 47, 4, 21, 41, 29, 44, 49, 12, 16, 52, 35,  
18, 38, 27, 19, 13, 45, 51, 31], device='cuda:0')  
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

## 12. 结果评估

---

从上面的测试图像可以看到，Keras和Pytorch训练出来的模型都能够在测试图像上有很好的表现，但是在实时视频流的识别上就有所不足。在使用摄像头进行实时识别的时候，会出现误识别的情况，分析可能是受到了光照条件的影响，以及在使用OpenCV从逐帧图像中截取人脸时截取框大小选择不太合适，应该保证截取出来的人脸应该和训练集一样保留少许背景，而不是全部都是人脸。在将截取范围略微改大之后，识别效果好了很多。

以及在制作训练集的时候，重复照片过多，光照条件差别不大，每个人的照片差别不大，所以导致了模型的泛化能力不强，考虑在进一步的实验中，对于128x128的原始图像，不采用直接缩放到64x64的预处理方法。而使用中心随机剪裁的方法，即每次在图像中心区域随机剪裁出64x64的区域作为训练数据。同时在图像数据预处理的时候，加上对图像颜色亮度的调整来模拟不同的光照条件。

## 第五部分 实验反思以及未来计划

---

### 13.1 实验反思

这是我第一次从头到尾完全自己编写代码完成基于Tensorflow和Pytorch的深度学习任务，先前仅有使用Keras搭建简易神经网络的经验。在这一次实验中我尝试了多种不同的框架，不同的训练方法，以及不同的参数进行了训练。在这个过程中遇到了许许多多的问题，尤其是在数据的读取与预处理的部分以及训练部分，让我对于这种大规模图像数据的处理上积累了许多的实战经验。最开始我会因为图像标签没有转化成one-hot码而出问题，之后花了一整个晚上研究出了如何建立训练标签到实际学号的映射表，当我发现Pytorch自带的API里面使用的方法和我一样时，也是十分有成就感的。

这次大作业让我在深度学习任务上的实战能力大大提升，对于经典机器学习算法的理解更加深刻，俗话说“纸上得来终觉浅，绝知此事要躬行”，对于概念十分复杂的机器学习和深度学习，实战是再好不过的学习途径了。

## 13.2 未来计划

我将会继续完善并维护代码，争取在大三期间完善Pytorch下的功能解决Tensorflow版本代码中仍然存在的问题，以及在现有功能的基础上完善前端，制作出来易于用户使用的UI界面。

项目Github地址：[https://github.com/HilbertXu/face\\_recognition.git](https://github.com/HilbertXu/face_recognition.git)

若有问题请联系：[hilbertxu@outlook.com](mailto:hilbertxu@outlook.com)

## 参考文献

[1] [https://tensorflow.google.cn/api\\_docs/python/tf](https://tensorflow.google.cn/api_docs/python/tf)

[2] <https://keras.io/zh/>

[3] <https://github.com/machrisaa/tensorflow-vgg>

[4] <https://github.com/yunjey/pytorch-tutorial>

[5] <http://www.tensorfly.cn/tfdoc/tutorials/overview.html>