第 1 章　概论

第 1.1 节　人体姿态估计的发展概况

略。

第 1.2 节　各种分类方法的发展概述及研究现状

1.2.1、各种分类方法的概述以及发展

涉及到姿态识别，识别出摄像头前的具体是哪一种姿态必然要实现分类，最早的有基于专家系统的动物识别分类系统，再有相比起来比较复杂的支持向量机 SVM 分类方法，基于手动提取特征的机器学习方法，更进一步地有通过设计良好的网络使得网络自己学习并提取特征的深度学习方法。

1.2.2、关于专家系统的动物识别分类的概述以及研究近况

类似于多重 if-else 嵌套判断，或是 switch-case 分支语句。它用产生式规则表示知识，满足特定的条件之后可以大致推断类别，根据更多的条件进一步可以推断输入的更细节的类别所属，要求可以随时改变规则，增添或是删除规则库，一遍又一遍地遍历规则库，然后加入拓扑序列，当输入动物的特征值到达一定的程度足以确定具体的某一个动物的时候，停止并输出结果。动物识别系统的缺陷是当规则库特别庞大的时候代码会随之增长起来，效率不太理想[12][13]。

1.2.3、关于支持向量机分类的概述以及研究近况

简单来讲支持向量机就是一条直线以最大的空隙间隔分离两类代表不同对象的数据，扩展到高纬度就是超平面分离数据，再扩展到多分类，可以用将整体二分类再分类的方法实现多分类。涉及到的数据包括线性可分，线性不可分等。SVM 地理论基础是统计学，优化问题的约束是降低训练过程中的误差[11]。

支持向量机的优点是解决小样本问题，即在数据量规模比较小的时候表现良好，同时在非线性还有高维度，超平面分类问题中存在一定的优势，还能进一步的应用于机器学习的函数拟合问题中去，但是缺点是模型的稳定性不太友好，微小的输入上面的变化会使得他的模型难以收敛，同时如果数据量是巨大规模的，那么会导致他的运算复杂度变得异常高。不适宜处理这一类的大规模数据。总体来讲 SVM 不如集成模型[14]。

1.2.4、关于机器学习与深度学习分类的概述以及研究近况

机器学习分类与深度学习分类的主要区别在于特征的提取，机器学习比较强调自己手动提取较为明显的特征，然后进行分类，分类方法可自己选择多种形式的分类方法，例如可以进行全连接 BP 神经网络[15]模型做分类，当然也可以使用 SVM 支持向量机。而深度学习则强调标注好的数据样本，设计良好的网络模型，让模型去自动提取特征。进一步做分类。

第 2 章　骨骼数据的获取

第 2.1 节 安装 OpenPose

由于良好的接口，并且由于 OpenPose 可接受的输入很广泛，几乎涵盖各种格式的图片，视频，并且支持网络 web 摄像头，它的输出格式也可进行调整，有 JSON，YML，XML 格式，也可以是视频图片格式，通过内部代码结构调整甚至可以输出自己想要的数据格式，以及保存到特定的文件方便使用。除了检测出人体的关键骨骼信息之外，OpenPose 还能检测人脸的 KeyPoint，手部的 KeyPoint，本系统选择使用 OpenPose 作为骨骼信息获取的来源。同时由于分类的网络是由 Python 设计的，因此选择 OpenPose 的 python 接口。关于 OpenPose 的安装可分为以下几个主要的步骤：

1. 安装显卡驱动以及 CUDA，CuDNN，需要注意的是要匹配电脑显卡型号，不然会造成 Caffe 运行时出现错误而导致整个程序无法启动运行；
2. 通过 Git clone 的方式将 OpenPose 下载到本地；
   git clone https://github.com/CMU-Perceptual-Computing-Lab/openpose
3. 使用 Cmake 来编译原项目，新建 build 目录生成结果。选择 visual studio 2017 x64 平台. 由于网络原因可能下载比较耗时间，可以自己手动下载所需的 model 以及所需的 Caffe，OpenCV 等主要依赖项。
4. Cmake 完成后，用 Visual Studio 2017 打开项目，将其解决方案配置为 Release&x64，按 F5 运行。
5. 因为我们需要运行 python 接口，因此要返回 Cmake 将 BUILD_PYTHON 勾选，于是我们可以接入 python 里面 import openpose。

OpenPose 在 Ubuntu 上面的安装相比 Windows 的更加方便一点。

1. 首先需要从源码下载并编译最新版本的 Cmake。
2. 安装匹配于电脑显卡的 cuda 和匹配 cuda 的 CuDNN，并且能够正确运行 sample 文件；
3. 通过执行源码里面的 install_deps.sh 安装 Caffe，通过 sudo apt install libopencv-dev 安装 OpenCV；
4. 使用 Cmake GUI 使用 Unix MakeFiles 来配置项目。
5. 勾选 BUILD_PYTHON 重新在 Cmake GUI 里面生成一下 Python API 接口；
6. 通过 make -j`nproc`来生成最终结果。

关于 OpenPose 环境的搭建在 Windows 和 Ubuntu 下都尝试过，Ubuntu 下搭建环境更为方便，并且 Ubuntu 运行速度相比 Windows 更快速。

第 2.2 节 关于 OpenPose 网络模型算法的具体细节

关于 OpenPose 的算法，主要有以下几个方面：

1. 对于在图像中出现的所有真实人体进行回归，分别去回归每一个人的每一个骨骼关键点。

2. 通过“center map”去除对其他人的影响和响应。

3. 第三部是通过重复地对预测出来的热力图片进行一个重新定义得到最终的结果。在重新定义结果的时候，需要引入一个 loss，所属中间层。这样的好处是保证在姿态预估网络进行到深层网络的时候不至于发生梯度爆炸，梯度突然消失的情况。这种思路有效地提高了回归后的准确度。

4. 在本项目的使用过程中，没有使用多人姿态预估，只调用了单人的参数来估计骨骼关节点位置，但是多人的却是 OpenPose 的一大特色，他的实现是基于 CPM 构造的。因此计算量很大，对于显存也有一定的消耗。而且划分是依据 PAF 的。

5. CMU 自己搭建了一个采集姿态的球，采集了特别多的数据集，并且数据集的质量很好，因此模型的鲁棒性非常好，能应对绝大多数的情形，相比之前的 DensePose，最基本的一点是光线对于模型的影响不再是很大的了，数据采集的时候就考虑了光线等众多因素。

6. 进一步的去增加时间序列，追踪一段时间内人体骨骼关键点的位置变化的轨迹。

7.在进行局部图匹配的时候 OpenPose 使用的特殊算法是匈牙利算法[16]，匈牙利算法的核心是寻找增广路径，进一步求二分图的最大匹配，并根据相应的结果来对应人的图域整体。

8. 关于除了人体姿态预估方面的，OpenPose 提供了双手关键点和脸部七十个关键点的检测，手部二十一个关键点，可以进一步做手势识别等。

总的来讲，OpenPose 姿态预估模型先将数据集输入图片经过十层的 VGG19[22]网络提取特征，然后再进一步用上面的 VGG19 提取的特征分为两个主要分支，一个去预测关键点的亲和度向量，另外的一个特征分支去预测关键点置信度。通过上面两个分支的处理后，然后再将关键点进行聚类，最后将骨架组装起来，展示出来的就是人体骨骼关键点的连接图。

因此归纳来讲，人体姿态预估一般是自顶向下的方法，先找出图片中的人，然后单独的对每一个图像中的个体找出它的人体骨骼关键点连接起来，这种方法如果在人都没有找到的情况下是不可能找到对应人的骨骼关键点的，并且人数越多的时候，计算量越大，耗时耗资源的程度和人数呈现正相关的线性关系，OpenPose 反其道，使用自底向上的设计模式，先找关键点，然后再拼人体框架，为了克服无法利用全局上下文信息的缺陷，OpenPose 又设计了 PAF 部分区域亲和，联合学习关键点和整体之间的联系。另外一个 OpenPose 的亮点是图像中人数的多少不会对总体耗时产生影响，它利用了贪婪分析算法，对全局上下文进行足够的编码，提前消耗一部分计算资源。在这种前提下图像中人的个数不会对总体产生影响。

第 2.3 节 通过 OpenPose 获取骨骼数据

OpenPose 的采集装置里面就有 Kinect 摄像头，VGA 工业摄像头，以及高清摄像头。body25 的数据说明如下：他的骨骼姿态包括二十五个点，其中分别是 0 号位置是鼻子，22 号位置是左脚大拇指，19 号位置是右脚大拇指，2 号位置是左肩膀，5 号位置是右肩膀，1 号位置是脖子，3 号位置是左胳膊肘，6 号位置是右肘，7 号位置是右手手腕，4 号位置是左手腕，8 号位置是髋关节，9 号位置是左髋关节，12 号位置是右髋关节，13 号位置是右腿膝盖，10 号位置是左膝盖，11 号位置是左脚踝，14 号位置是右脚踝，15 号位置是左眼，16 号位置是右眼，18 号位置是右耳，17 号位置是左耳，右脚小指是 20，23 号位置对应的是左脚小指，24 号位置是左脚脚后跟，21 号位置是右脚后跟这二十五个骨骼关节点的 x 轴信息，y 轴信息，还有每一个点的置信度 c，在太极姿态所需要的骨骼点中，只需要 x 轴信息，y 轴信息，没有加入置信度。同时设置当置信度大于 0.001 的时候就标示为骨骼信息。

首先引入 OpenPose，其次加载 OpenPose 所需要的 body25 数据集，通过以下三条

语句来启动 OpenPose：

```
opWrapper1 = openpose.WrapperPython() # 封装成 python 的 openpose 接口。
opWrapper1.configure(params) # params 是一个 dict，是配置参数
opWrapper1.start() #启动 OpenPose
```

通过 OpenCV 将视频读取到 NumPy 矩阵中去，然后传给 OpenPose 入口处理

```
Datum1 = openpose.Datum()
Datum1.cvInputData = frame #frame 是 OpenCV 通过 imread()读取的每一帧
opWrapper.emplaceAndPop([Datum1])
keyPoints = Datum1.poseKeypoints.tolist() #变成普通的 list 形式
```

因此我们便可以得到所有的 25 个人体骨骼信息（因为这里我们使用的是 BODY_25 数据集）的关键点的横坐标，纵坐标，还有每一个点的置信度，置信度是指。

第 3 章　算法设计

第 3.1 节 骨骼节点特征的设计与提取

为了便于神经网络的训练以及损失函数快速的收敛，更为了提取高层信息，我们选取了差别比较明显的十五组人体姿态骨骼关键点之间的距离作为特征，同时选择了十五组夹角作为角度特征整合到一个数组中去。这样我们就手动地提取了高层信息，并不需要设计特征提取网络层来让网络提取特征。

3.1.1、距离特征的设计与提取

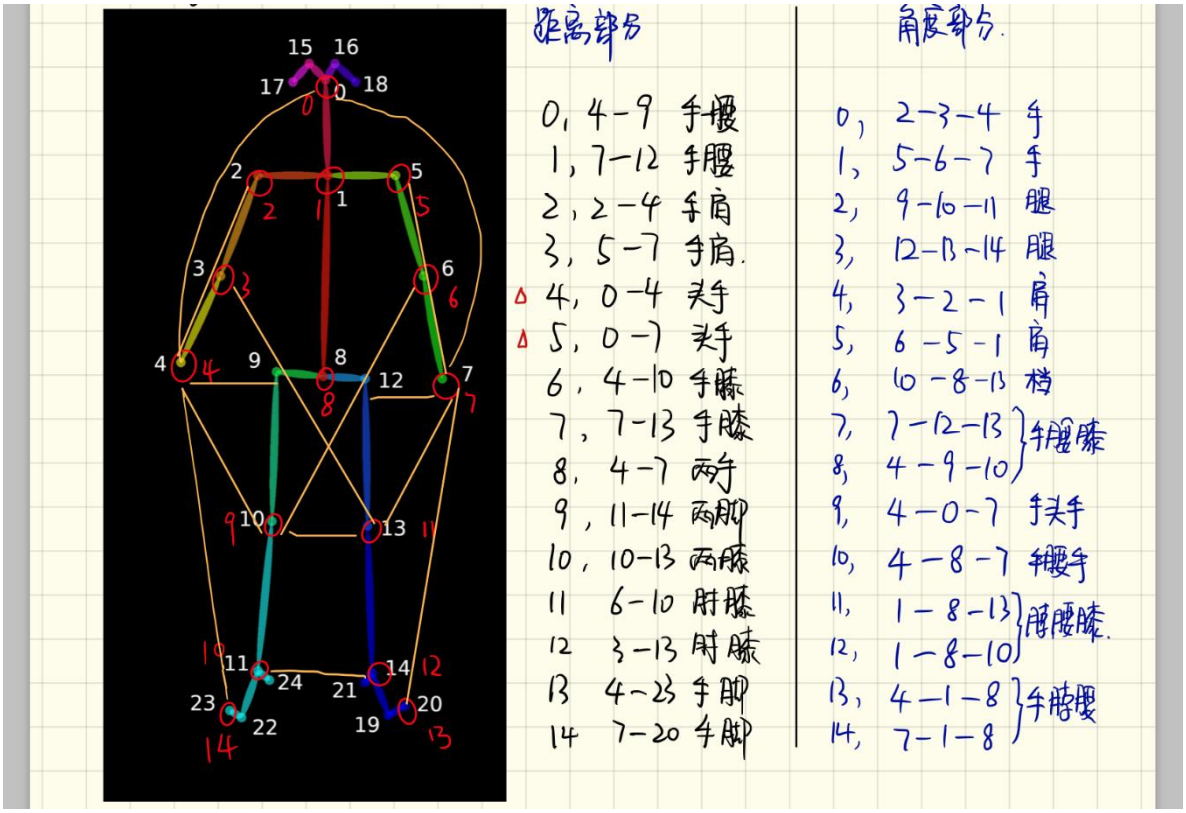

图 3.1 具体的特征设计详细

如图 3.1 中所示：距离部分是 0：4-9 手和腰关节之间的距离；1：7-12 手和腰关节之间的距离；2：2-4 手和肩膀之间的距离；3：5-7 手和肩膀之间的距离；4：0-4 头和手之间的距离；5：0-7 头和手之间的距离；6：4-10 膝盖和手之间的距离；7：7-13 膝

盖和手之间的距离；8：4-17 两手之间的距离；9：11-14 两脚之间的距离；10：10-13 两膝盖之间的距离；11：6-10 肘和膝盖之间的距离；12：3-13 肘和膝盖之间的距离；13：4-23 手和脚之间的距离；14：7-20 手和脚之间的距离。之所以要提取以上关键点之间的距离是因为他在太极姿态中的变化最为明显，例如两个手指间的距离，两个脚之间的距离，头分别和左手，右手之间的距离，在特定的姿态中他们有明显的变化，但是有些距离就没有必要，比如 2 和 3（肩膀和胳膊肘）之间的距离，这两个关键点是相邻的，具体的原因是不管做什么动作，只要人体在摄像头前的位置固定，这一组距离就不会发生变化，他们是两个相邻的节点，因此在设计的时候避开了相邻的关键点，并选取了在特定的太极二十三组姿态中差别最为明显的十五组距离特征存到指定的文件里。

角度部分是 0：2-3-4；1：5-6-7；2：9-10-11；3：12-13-14；4：3-2-1；5：6-5-1；6：10-8-13；7：7-12-13；8：4-9-10；9：4-0-7；10：4-8-7；11：1-8-13；12：1-8-10；13：4-1-8；14：7-1-8，如图 3.1 所示。

这样便设计出一套组合特征：距离和角度，能够提高复杂关系的拟合能力，同时由于有些角度在某些情况下的值为零值，比如 2-3-4（整个手臂：手，胳膊肘，肩膀之间的夹角的余弦值）夹角为直角的时候余弦值是零值，当 OpenPose 丢失一个点比如说 4 号位置手的信息没有检测出来的时候，返回零值，它的余弦值就和肩膀之间的夹角的余弦值）夹角为直角的时候余弦值相同了，因此仅仅依据距离信息有时候也会对数据训练产生一定的误导作用，所以就按照距离约束角度，角度约束距离的设计思路设计了独特的太极姿态识别的训练数据。这个训练数据已经是经过手动加工提纯的高层信息，不需要再添加复杂的特征提取网络层提取更深的特征信息。设计 BP 全连接就可以拟合出想要的结果来。

具体的设计如下：

距离是指两两关节点之间的欧式距离（二范数）：

$$\mathrm{A}(x_i, y_i, c_i)$$
$$\mathrm{B}(x_j, y_j, c_j)$$

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

```
# ========================== code ================================
distance0 = (keyPoint[4][0] - keyPoint[9][0]) ** 2 + (keyPoint[4][1] -
keyPoint[9][1]) ** 2
…
# ========================== code ================================
```

3.1.2、角度特征的设计与提取

其中x_i是指横坐标，y_i是指纵坐标，i是指不同的骨骼关键点。
角度是指给定骨骼点

$$A(x_i, y_i, c_i)$$
$$B(x_j, y_j, c_j)$$
$$C(x_k, y_k, c_k)$$
$$c = |AB| = d$$
$$a = |BC| = d$$
$$b = |AC| = d$$
$$\cos\theta = \frac{c^2 + a^2 - b^2}{2ac}$$

```
# ========================== code ================================
def __myAngle(A, B, C):
    c = math.sqrt((A[0] - B[0]) ** 2 + (A[1] - B[1]) ** 2) # 求二范数
    a = math.sqrt((B[0] - C[0]) ** 2 + (B[1] - C[1]) ** 2) # 求二范数
    b = math.sqrt((A[0] - C[0]) ** 2 + (A[1] - C[1]) ** 2) # 求二范数
    if 2 * a * c != 0:
        return (a ** 2 + c ** 2 - b ** 2) / (2 * a * c)
    return 0
angle0 = __myAngle(keyPoint[2], keyPoint[3], keyPoint[4])
…
# ========================== code ================================
```

所谓的“明显”是指各种动作中这一个特征会有明显的变化，比如两个脚之间的距离，两个手掌间的距离，手-胳膊肘-肩膀之间的角度，之所以要加入角度信息是因为，活体在摄像头前面距离摄像头的角度随时在改变，单纯的角度信息会有一定的干扰，而加入角度信息之后，不管活体距离摄像头的角度，只要能检测到全身的骨骼，同一个姿态它的角度信息不会因为距离摄像头的远近而发生显著的改变。因此加入了十五个角度特征信息来保证他不会受到距离的影响，也在一定程度上和距离信息形成

相应地约束作用。

第 3.2 节　分类的设计与实现

2.3.1、workspace 程序开发目录详细说明

- workspace
  - data_collection(数据采集)
    - data_collection.py （数据采集程序入口）
    - data_collection_window.py （数据采集界面设计类）
    - data_collection_window.ui （数据采集界面 UI 文件）
  - dataset (数据集)
    - taichi
      - marked_pic（CNN 训练所需图片文件夹）
        - p_2_0.jpg（最后一个下划线后面是类别，此处 0 是类别，前一个数字 2 代表大概数量）
        - ...
      - marked_pictrain.txt（CNN 所需图片的路径和类名）
      - bone_dataSet.data （全连接网络所需要的数据集）
    - marked_pic（data_collection 所采集的临时图片需经过挑选放入 taichi 文件夹）
      - p_2_0.jpg（最后一个下划线后面是类别，此处 0 是类别，前一个数字 2 代表大概数量）
      - …
    - pic_background（带有背景的 data_collection 采集的图片）
    - bone_dataSet.data（data_collection 所采集的临时骨骼点信息，需经过挑选才能放入 taichi 文件夹）
  - main_program
    - main.py（主程序入口）
    - mainWindow.py（界面类文件）
    - mainWindow.ui （界面设计 UI 文件）
  - model_pth (模型保存位置)
  - neural_network
    - runs (tensorboard 可视化,如果有必要)
    - classification23_taichi_eigenvalue.py （依据角度和距离全连接分类）
    - classification23_taichi_pic.py （依据骨骼图片进行卷积分类）
    - data_process.py （数据处理文件，角度和距离）
    - predict_eigenvalue.py （模型加载）

        - predict_pic.py （CNN 模型加载）
    - openpose_python_demos (包含一些 python 使用 openpose 的例子)
        - flags.hpp(调用 openpose 的参数设置)
        - use_camera_by_opencv.py
        - use_camera.py
    - sundry (包含一些界面设计的图片,训练后的 loss 和准确率的图片等杂项)
        - ...

2.3.2、特征值网络的设计

神经网络是用 Pytorch 实现的，因为是分类问题，设计过程受到了手写数字识别程序利用全连接网络[17]的启发，因为手写数字识别是一个很简单的十分类的问题，而太极姿势有二十四式，其中有两个是相同的姿势和名称，因此二十四分类问题少了一类变成了二十三分类。又因为涉及到干扰项（即不属于二十三类太极姿势的姿态）的影响，设计过程还得增加一个杂项，不过依旧是二十三分类问题。使用全连接神经网络,层数在最开始设计的是四层,输入数据是三十维度(原始数据包括十五维度的距离，十五维度的角度和一维度的类别)的，输出则自然是[0,23]，神经网络的第一层隐藏层的单元数一共设计了两百个，第二层隐藏层单元数是三百个，第三层隐藏层的单元数是一百个。输入数据x的形状类似于

```
x = [[],
     [],
     [],
     []]
```

len(x[0]) = 30
len(x) = 170（后续有增加）
y 的形状类似于：表示分类类别

```
y = [0,
     1,
     2,
     3,
     4
     …]
```

len(y) = 170（后续有增加）
y 是一维数组，与 x 的第一维度相对应，对应所属的姿势（分类结果）。 有了输入输出，定义的网络模型如下：

```
# ========================= code start=========================
class twentyclassifications(nn. Module):
    def __init__(self, in_dims1, n_hidden_1, n_hidden_2, n_hidden_3, out_dims1):
        super(twentyclassifications, self).__init__() # input layer()
        self.layer1 = nn.Sequential(
        nn.Linear(in_dims1, n_hidden_1), nn.ReLU(True)) # 第一层全连接的隐藏层
        self.layer2 = nn.Sequential(
        nn.Linear(n_hidden_1, n_hidden_2), nn.ReLU(True)) # 第二层全连接的隐藏层
        self.layer3 = nn.Sequential(
        nn.Linear(n_hidden_2, n_hidden_3), nn.ReLU(True)) # 第三层全连接的隐藏层
        self.layer4 = nn.Sequential(nn.Linear(n_hidden_3, out_dims1)) # output layer

    def forward(self, x):
        x = self.layer1(x) # 进行正向的传播
        x = self.layer2(x) # 进行正向的传播
        x = self.layer3(x) # 进行正向的传播
        x = self.layer4(x) # 进行正向的传播
        return x
# ========================= code end =================
```

在这个太极姿态识别的神经网络中，以线性整流 RELU 作为本太极姿态预估网络模型中神经元的激活函数，激活函数表达式是$f(x) = \max(0, x)$,太极识别的全连接网络的损失函数使用的是交叉熵函数 nn.CrossEntropyLoss，CrossEntropyLoss[21] 的一个特定的优势是不需要在它的网络的最后一层里面即输出层的前添加一层 softmax 层还有另外的 log 层，太极姿态直接输出全连接层即可。太极姿态的全连接网络采用随机梯度下降进行优化 optim.SGD(model.parameters(), lr=0.000001)，其中设计的 learning rate = 0.000001，即 1e-6. 然后进行 600 次左右的迭代便是：

调整网络结构的对比试验：在三层神经网络（30,60,40,12），学习率 0.01，3000

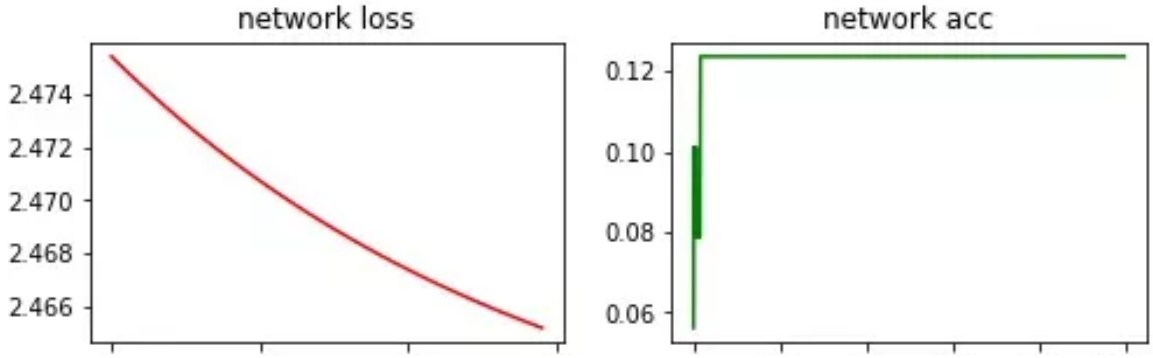

图 3.2 左边是 loss 曲线，右边为准确率

次 GPU 训练效果如图 3.2 所示。

很明显 loss 下降的很慢，几乎没有特别明显的下降，准确率特别低，只有 12%。进一步调整参数，得到第二次训练效果图片如图 3.3 所示：

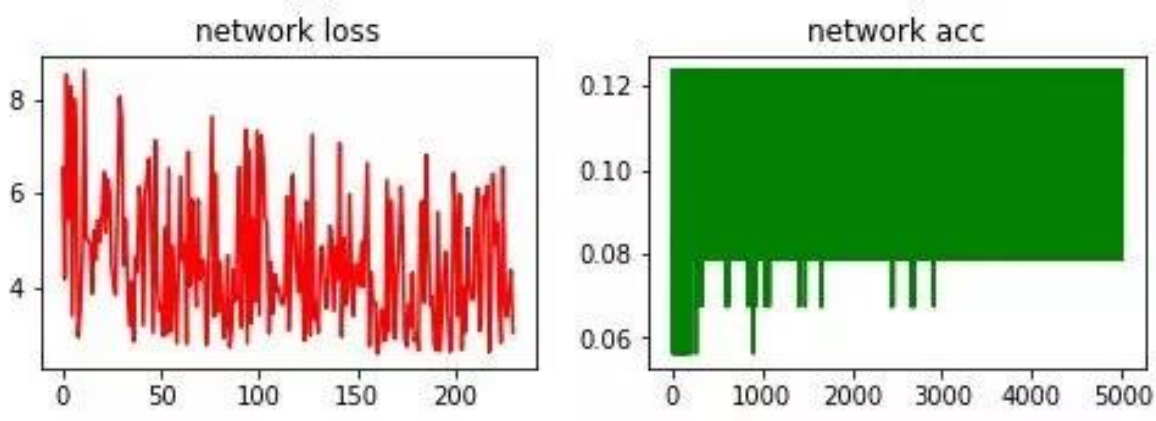

图 3.3 左边是 loss 曲线，右边为准确率

本次训练为学习率 0.01，训练次数为 5000 次，如图 3.3 所示，可以看到它的 loss 摆动很严重，对于输入数据，分析数据，其中数据主要分为三部分，前十五维和后十五维相差太大了，前十五维度是距离信息，它的范围是未开方的大整数距离范围∈（0，+∞），相比起来很大，而另外一边是十五维度的角度信息，范围∈（-1，1），相比之前是很小的数值，学习率得更小才能有良好的训练效果，学习率的作用就是在正向传播反向传播的过程中起到一个纠正权值 w 的重要作用，这里对于输入数据的数据归一化做的不太好。关于数据的归一化，在这里是指将所有的 OpenPose 采集到的数据都统一到一个大致相同的数值区间，比如[0,1]或是[-1,1]。

最终，以 GPU 训练，600 次的小规模训练，同时设置学习率为 0.000001，得到的最终效果图如图 3.4 的 network loss 和 network acc 所示，与此同时，GPU 训练所需要

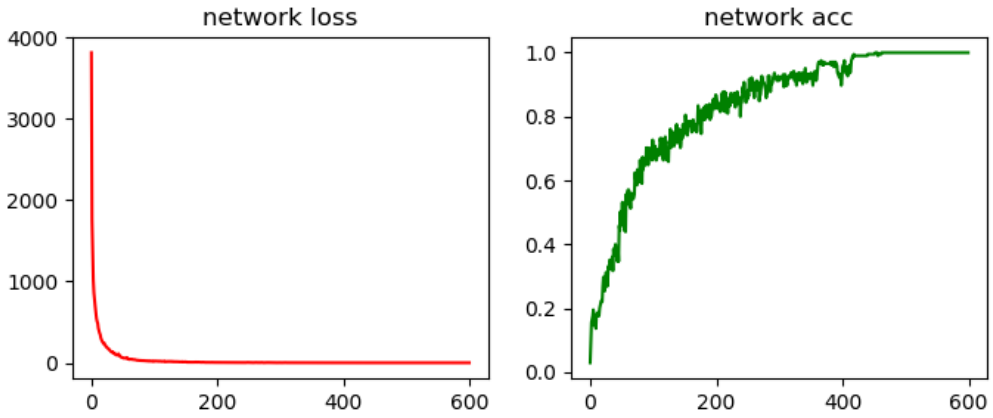

图 3.4 左边是 loss 曲线，右边为准确率

的时间平均在 2-3 秒，在 Windows 下的时间稍微比 Ubuntu 下的时间久一点。

TensorBoard 效果如图 3.5 和图 3.6：

图 3.5 TensorBoard 中多次训练后 Loss 展示

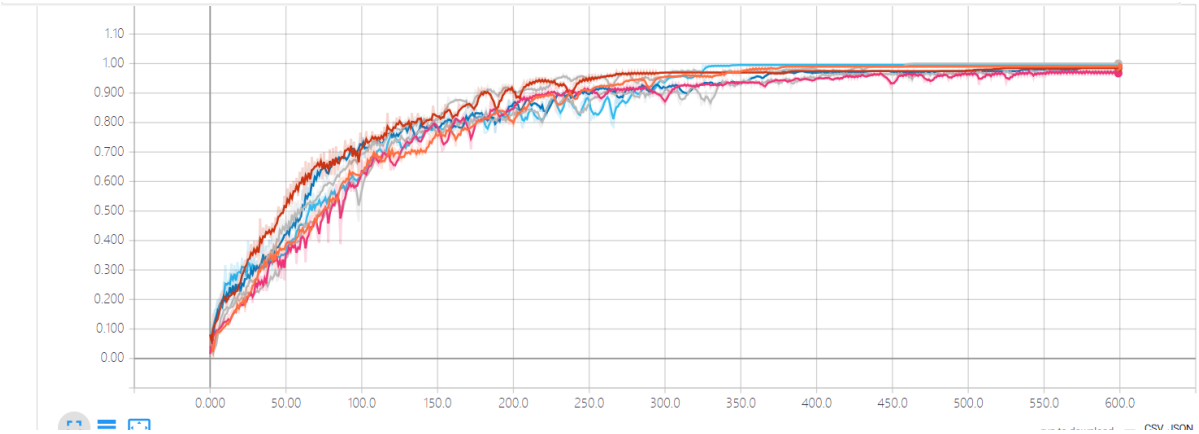

图 3.6 TensorBoard 中多次训练的准确率

2.3.3、卷积网络的设计与实现

前面关于特征值网络的训练和预测是高效的，没有卷积之类的关于图片的复杂操作，在提取了最为明显特征的情况下以极快的速度和最少占用 CPU 以及 GPU 资源的情况下达到了较为理想的结果（后面在 CPU 上面实验表明不使用 GPU 也能在一分钟之内训练完毕结果）。

因为图片涉及到卷积网络，并且图片里面的特征特别复杂，各种各样的杂乱的特征应有尽有，OpenPose 可以剥离背景图片而只显示摄像头前面的活体的骨骼，因此在

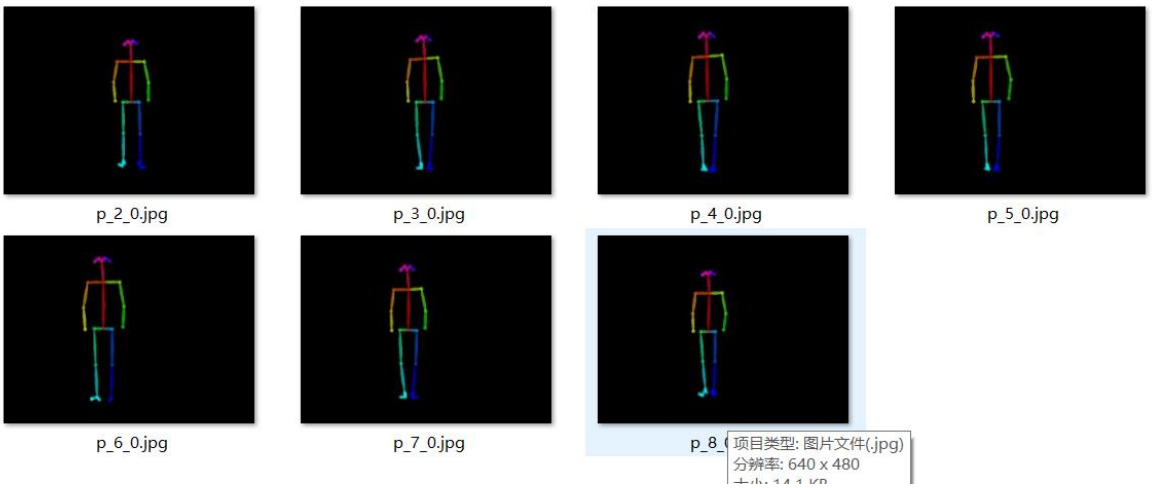

图 3.7 去除了背景的纯骨骼图片

采集特征值地时候顺便的将所有的骨骼图片保留了下来，做卷积分类，如图 3.7 所示，只有几根线条，因此没有其他复杂的特征，易于模仿简单网络进行分类。

如图采集的预备式（太极姿态的第一个要分类的姿态）的骨骼图片，每一幅图片的大小是 640×480 的大小，同样地参照和借鉴了 fashion_mnist 的设计思路，fashion-mnist[18]的图片大小，训练样本数等和经典的 MNIST 完全一致。

主要一点是关于图片数据如何加载到网络中去的问题，此处采用的方法是先将所有图片的绝对路径同时以一个空格和它的类别标签记录到一个 txt 文件中去，继承 pytorch 里面的 DataSet 只去处理那一个 txt 文件读取图片以及太极姿态的标签即可。

txt 文件内容格式如下：前面是代表姿态图片的路径（图片名字的下划线分割开来最后的数字就是类别），后面的代表太极姿态类别。

F:\datasets\taichi\taichi\marked_pic\p_124_20.jpg 20

F:\datasets\taichi\taichi\marked_pic\p_125_21.jpg 21

网络结构如图 3.8.

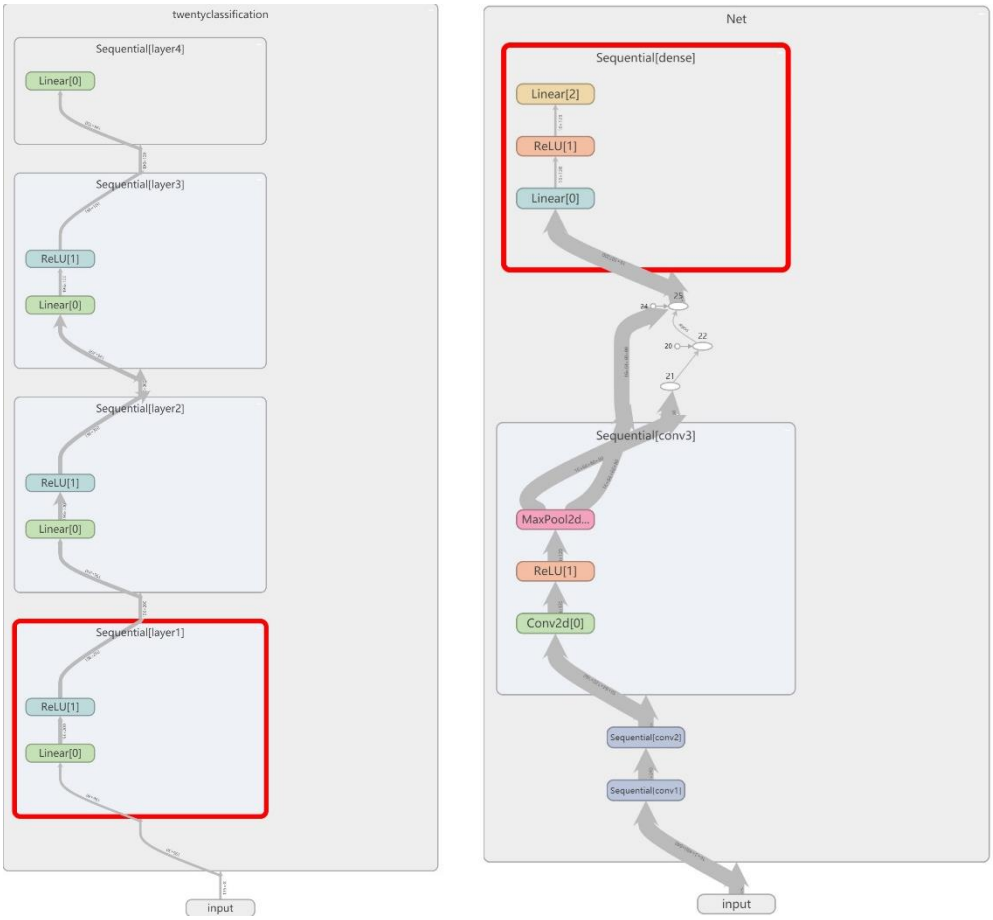

图 3.8 左边为全连接网络模型图 右边为卷积网络模型图

如图 3.8 所示，左边是四层全连接神经网络的设计细节，右边是 CNN 卷积神经网络的设计细节，这里是一层二维卷积与 ReLU 激活函数，再和一层二维池化 MaxPool2d 放在一起，三次同样的操作后与一层全连接构成整个设计的网络。

值得一提的是，原图像像素是 480*640 像素的，因此输入是 307200，也就是 64*60*80，做的是太极姿态 23 分类，输出便是 23 种太极姿态的 index，输出的结果

是一个数组，数组的最大值所在的索引便是太极姿态所属的类别。和之前的全连接的随机梯度下降优化算法不一样，这里的 CNN 使用的优化算法选择了自适应矩估计，由于输入数据是单纯的太极姿态的骨骼图片，数据集比较稀疏，网络设计的规模也比较小，因此它的效果用在判断太极姿态骨骼图片上相比随机梯度下降算法更好一点。

这个太极姿态 CNN 的模型训练的 batch_size 是 16，由于数据量有限，没有测试数据，只是反复进行十次的训练（GPU，1066 显卡），让 loss 下降，acc 上升，最终的结果是 acc 达到了 99.402%。程序中单次训练的结果如图 3.9 所示。

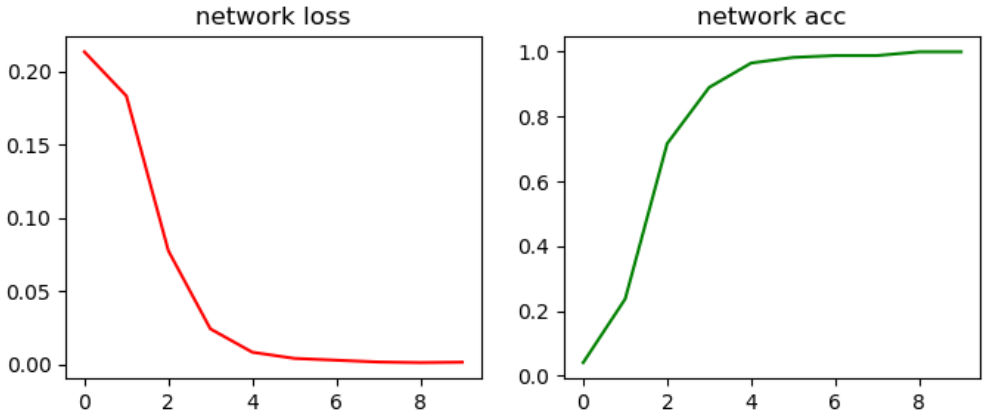

图 3.9 CNN 训练结果，左边时 loss，右边是准确率

TensorBoard 展示效果如图 3.10 所示。

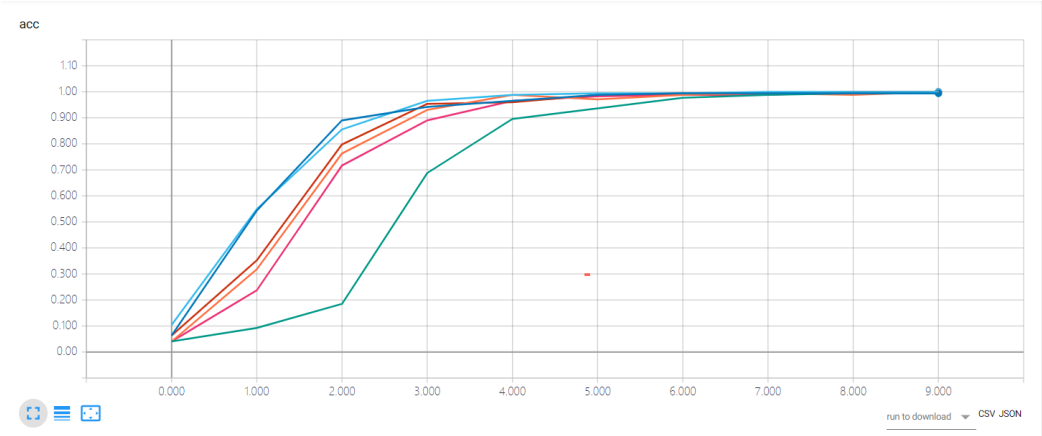

图 3.10 TensorBoard 中展示的 CNN 网络多次训练后的准确率

第 4 章　系统的设计与实现

第 4.1 节 太极姿态网络的模型保存与加载使用

关于太极姿态的全连接特征值网络的保存如下：

```
# ========================= code start================================
model.eval().cuda()
torch.save(model.state_dict(),"../model_pth/23classification_eigenvalue.pth") # 这里的 model.state_dict()是将对应参数保存至状态字典
# ========================= code end  ================================
```

通过以上代码便将太极姿态模型保存到指定的文件夹下的 23classification_eigenvalue.pth，此时保存的是 GPU 版本的，在另外一个文件里面调用首先需要将姿态识别的类导入到本文件，然后通过如下代码加载 CPU 版本的太极姿态模型：

```
# ========================= code start=======================
model = twentyclassifications(30, 200, 300, 100, 24)
model.load_state_dict(torch.load("../model_pth/23classification_eigenvalue.pth", map_location="cpu")) #（加载 CPU 版本的模型）
# ========================= code end  =======================
```

然后将实时采集到的太极姿态的骨骼数据送入 model 中进行一次正向传播即可。输出便是 23 种太极姿态的 index，输出的结果是一个数组，数组的最大值所在的索引便是太极姿态所属的类别。

而关于太极姿态的卷积神经网络[20]模型的保存与加载与太极姿态的全连接特征值网络的保存与加载有所不同，因为它有 batch_size，多了一个维度。

模型保存类似太极姿态的全连接特征值网络，加载需要添加一个零维度，并不代表实际意义，只是为了和模型格式相匹配：

```
# ========================= code start====================
pres = transforms.Compose([transforms.ToTensor()])(img) # img 可通过 OpenCV 获取，也可通过 PIL 读取图像，只要变成 numpy 矩阵即可。
```

```
pres = Variable(pres.unsqueeze(0)) # 添加维度
res = model(pres)[0].detach().numpy().tolist() # 转成普通 list
resClass = res.index(max(res)) # 寻找索引
# ========================= code end=======================
```

由于 OpenPose 的 Python API 接口，PyTorch，因此界面设计采用 PyQt5[19]来设计，搭配 PyUIC+PyCharm+QtDesigner，设计出一套较为完整的系统。主要分为两个主系统，第一是数据采集系统，数据采集系统是为了扩大数据源，使得能够完美的采集更多的信息，能够走向大众，从一个人到多个人，标准化太极姿态的所有人群，当然最好的面向的是是太极做的标准的一类人群，例如武术指导，健身教练等，从而可以去判断别人做的好不好。第二是实时预测系统，可以实时预测摄像头前一个人的姿态来判度他所属的太极姿态类别。

第 4.2 节 采集系统的实现细节

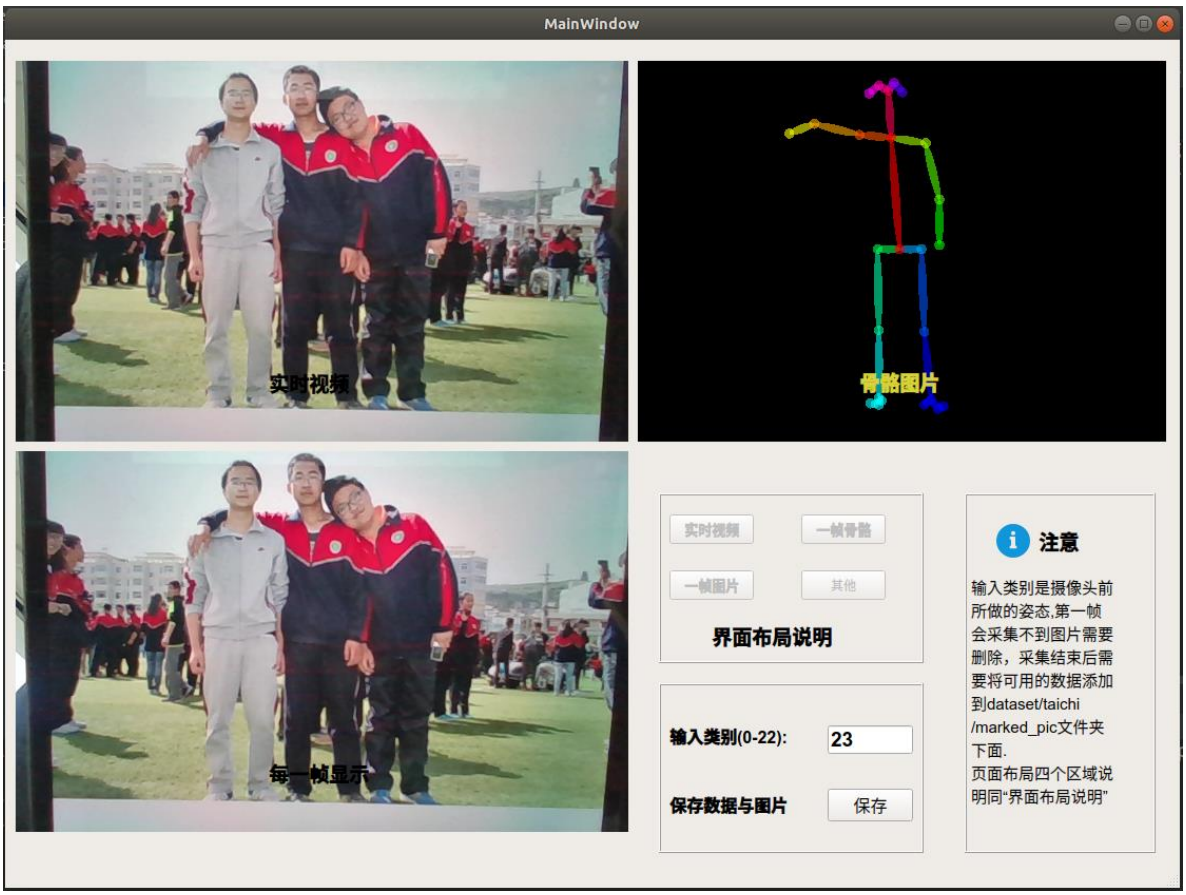

图 4.1 采集系统的界面

采集系统分为四个模块，如图 4.1 所示，第一个模块是摄像头前的实时视频信息，是随时间一直变化的。第二个模块（左下角）是需要采集的那一帧图片的抓拍，第三个模块（右上角）是同第二个模块同时的骨骼图片，这幅图片会被保存到指定的位置，可以加入训练集或者测试集中去扩大数据集，增强网络的容错以及鲁棒性。第四个模块是介绍说明，其中输入类别是指太极姿势的所属序号，0-22，如果是在测试阶段则可以使用 23 作为杂项采集到数据集里面去。输入类别是摄像头前所作的太极姿态，采集结束后可以选择性的将有用的特征值以及图片分别添加到 taichi/bone_dataSet.data（特征值文件）文件和 taichi/marked_pic（骨骼图片文件夹）文件夹里面去，为了保证数据的正确性，采集的图片并不会直接保存到网络训练所需的文件夹中，而是保存至网络训练所需的文件夹上层目录的同等文件里面。加入训练文

件需要手动选择最优质的训练源文件和特征值然后追加到指定的文件夹里。

关于采集系统的实现，由于要不断地去读取 USB 摄像头的每一帧的图片而且还需要送到 OpenPose 的接口中去处理得到结果，再送到太极姿态的神经网络模型之中去进行一次正向传播，将结果送到界面上展示，因此设计死循环是不现实的，PyQt5 中有一个 Qtimer，其中有一个以毫秒计时的设计，每一个 timeout 后面就可以刷新一次 OpenCV 通过 videoCapture 得到的图片。Python 中的多线程并不是实际意义上的多线程，Qtimer 却可以很好的弥补这一点。在本程序设置中是 30ms 刷新。

关于 Python 的多线程是在 threading 里面，用法如下：

```
# ========================== code start==============
thread1 = threading.Thread(target=function, name="fun1")# function 指的是你要在多线程里运行的函数名
thread1.start() # 开启多线程
thread1.join() # 合并到主线程 main 里面去
# ========================== code end  ==============
```

但是 Python 的多线程追溯到设计时候的 GIL 机制，并不是真正的多线程，而是模拟出来的多线程。

第 4.3 节 太极识别系统的实现细节

最终的太极识别系统的效果展示在基于人体骨骼姿态的姿势识别系统里面，这个太极姿态识别系统又有三个 Tab 页面，其中前面两个为主要的系统，网络训练和姿态预估，网络训练也是实时的去展示网络训练的过程，展示的是骨骼关节点特征值太极姿态网络。在下面两个查看详情里面可以通过 TensorBoard 查看网络的 loss 损失和 acc 准确率以及网络的结构。程序运行一次便将一次的 loss 损失函数和准确率，网络模型做一次记录，并保存至程序运行目录的 runs 文件夹下。通过激活 TensorFlow 环境，使用 tensorboard –logdir runs 在 localhost：6060 中查看结果（需要安装 TensorFlow 才可使用）。最为关键的一点是，随着程序的运行，可以设置 TensorBoard 的刷新时间，动态实时的查看网络的收敛情况。

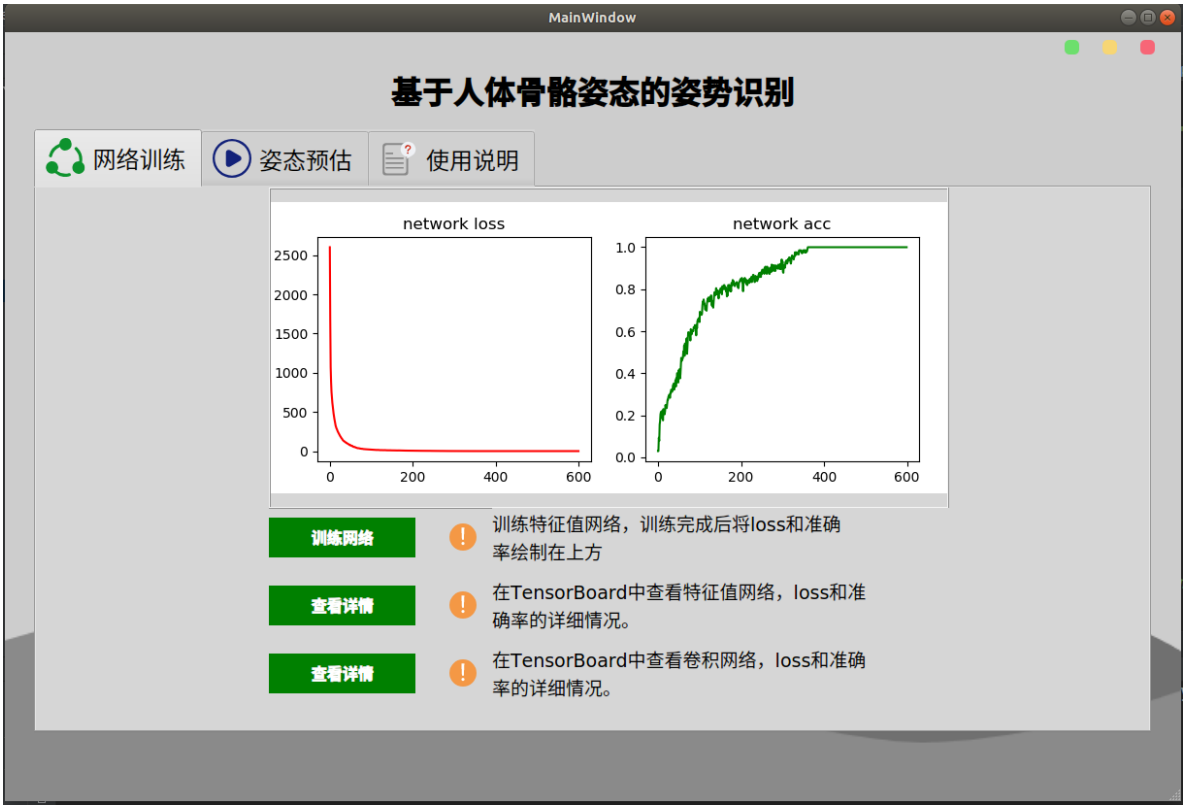

图 4.2 识别系统的网络训练 Tab

如图 4.2 所示，第一个 tab 展现太极姿态识别网络训练的结果，点击训练网络先会去执行网络的训练，网络训练先加载采集的特征数据，然后进行四层全连接网络的六百次训练，然后通过 matplot 绘制一幅图片保存到一个指定文件夹下（这里保存至 workspace/sundry 下），然后加载这幅图片展现在界面上，其余两个是在 TensorBoard 里面展示效果，第二个 tab 则是整个太极姿态识别系统核心的展现，如图 14 所示，其中一个是实时的摄像头捕获到的视频所展示的 Qlabel，通过 pixmap 的形式转化成图片展示出来，同时由于是实时的展示太极姿态，因此这里做到的是用 QTimer 每隔 30ms 的刷新间隔，由于太极姿态预估最后输出的结果是一个数字，分别对应的是索引 0 代表太极姿态第一个姿态:“预备势”，索引 1 代表太极姿态第二个姿态:“起势”，索引 2 代表太极姿态第三个姿态:“左右野马分鬃”……索引 22 代表太极姿态第二十三个姿态:“十字手”，索引 23 代表太极姿态第二十四个姿态:“太极拳”，这二十三种姿态外加一个杂项或者干扰项，为了程序设计的合理性，起名为“太极拳”，当网络输出一个索引后，会对应太极姿态的名称。实时地展现在 label 上面。

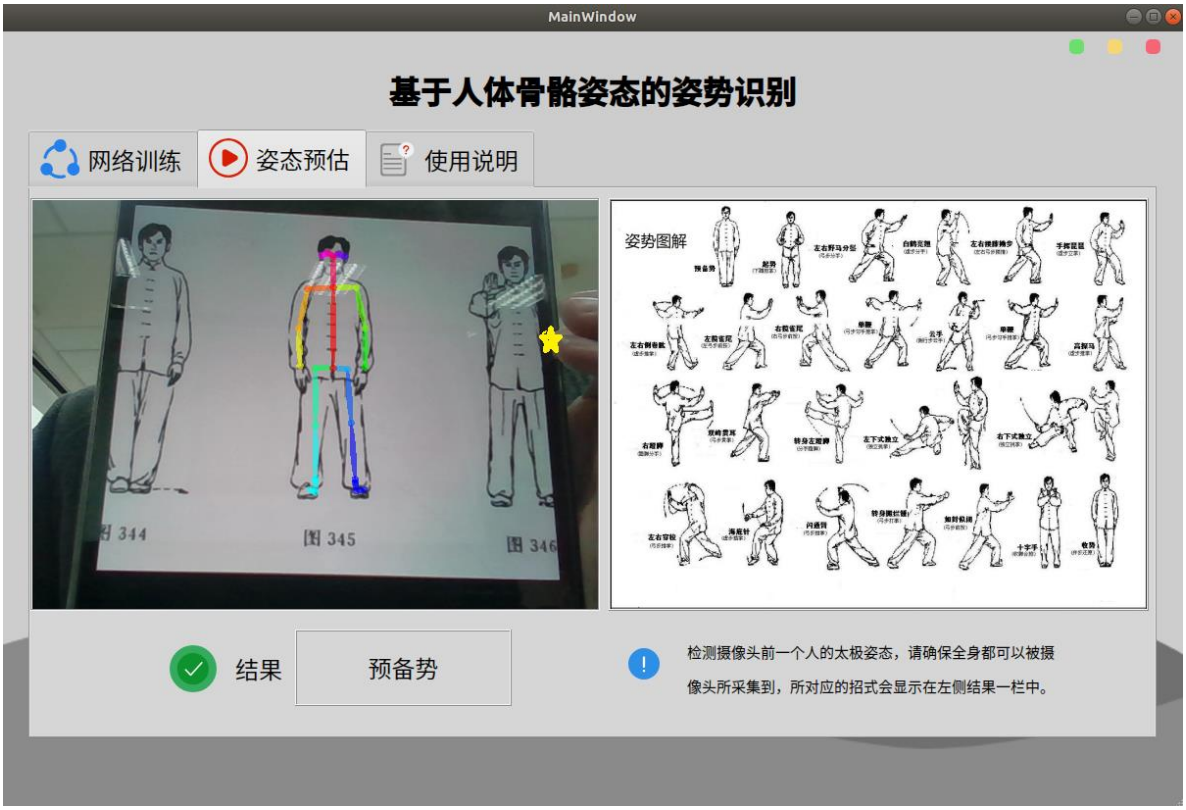

图 4.3 识别系统的识别 Tab

关于 PyQt 中各个部件的美化是依据 setStyleSheet 来设置各种美化的效果的。这种语法类似于 HTML 以及 CSS 语法，易于美化界面。由于有 Qtimer 的设计，视频是实时 30ms 的间隔刷新，同时，下方实时地展示经过太极姿态预估模型一次次正向传播后的结果。需要说明的是，这个过程十分消耗显存，因为实时地在调用搭建在 GPU 上面的 OpenPose，并且太极姿态识别的网络的设计也是基于 GPU 的，在不插电源的情况下对于结果和性能有一定的影响。

第三个 tab 是使用说明，主要介绍了基于人体骨骼姿态的太极识别系统的各个界面，各个 label，各个按钮的主要功能以及使用方法，更多的细节以及源码公开在 GitHub 上面。

第 5 章 系统总结

略。