

人体姿态估计相关工作总结

一、PC 端人体姿态估计（AlphaPose）

1、AlphaPose 基础介绍

它是一种自上而下的人体姿态估计方法，提出了一种专门用于区域多人姿态估计的框架（RMPE），主要由三部分组成：

- （1）SSTN（对称空间变换网络）：在不精准的区域框中提取到高质量的人体区域
- （2）PNMS（参数化姿态非极大抑制）：去除冗余姿态
- （3）PGPG（姿态引导区域框生成器）：增强训练数据

整体框架图如下所示：

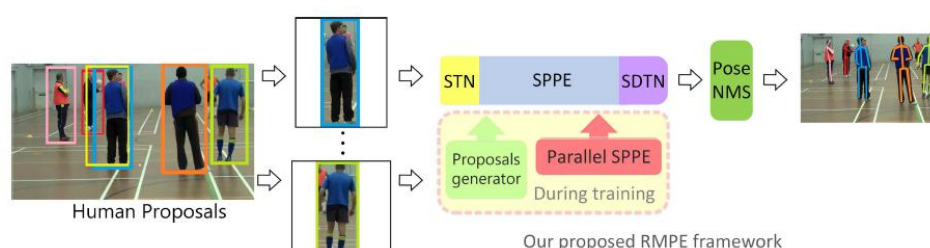


Figure 3. Pipeline of our RMPE framework. Our Symmetric STN consists of STN and SDTN which are attached before and after the SPPE. The STN receives human proposals and the SDTN generates pose proposals. The Parallel SPPE acts as an extra regularizer during the training phase. Finally, the parametric Pose NMS (p-Pose NMS) is carried out to eliminate redundant pose estimations. Unlike traditional training, we train the SSTN+SPPE module with images generated by PGPG.

SSTN=STN+SPPE+SDTN

大概流程：（1）先通过目标检测算法，将一幅图像 **S** 中的人分别框出来，此时相当于得到单个人体的原始图像 **A**；（2）将单个人体的原始图像输入 STN（空间变换网络），通过一些翻转平移等变换，可得到一个高质量的人体框 **B**（目标图），即此时的人体框更准确；（3）将此时高质量的人体框图 **B** 输入 SPPE（单人姿态估计），可得到单人姿态估计线条图 **C**；（4）将 **C** 图输入 SDTN（空间反变换网络），即相当于将姿态估计线条映射回原始图像 **A**，因为可能有冗余姿态，所以加上 PNMS，这样最后可在最原始图像 **S** 中表现出每一个人体的姿态估计线条。

重点：

Symmetric STN and Parallel SPPE（对称空间变换网络，并行单人姿态估计）

目标检测算法得到的人体区域框不是非常适合 SPPE，因为 SPPE 算法是专门针对单个人的图像进行训练的，并且对于定位错误十分敏感。通过微小变换、修剪的方法可以有效的提高 SPPE 的效果。SSTN+Parallel SPPE 可以在不完美的人体区域检测结果下有效的增强 SPPE 的效果，结构如图所示。

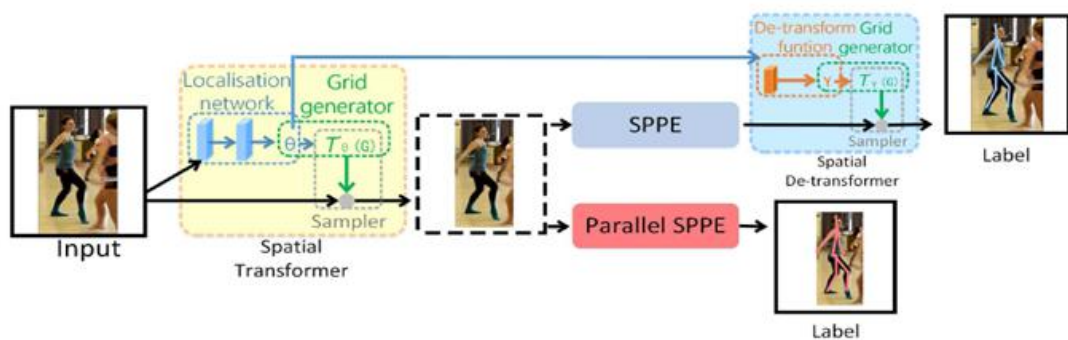


Figure 4. An illustration of our symmetric STN architecture and our training strategy with parallel SPPE. The STN used was developed by Jaderberg *et al.* [22]. Our SDTN takes a parameter θ , generated by the localization net and computes the γ for de-transformation. We follow the grid generator and sampler [22] to extract a human-dominant region. For our parallel SPPE branch, a center-located pose label is specified. We freeze the weights of all layers of the parallel SPPE to encourage the STN to extract a dominant single person proposal.

图解：表示了 SSTD + Parallel SPPE 模块的结构，SDTN 结构接收一个由定位网络生成的参数 θ ，然后为反向转换计算参数 γ 。我们使用网格生成器和采样器去提取一个人的所在区域，在 Parallel SPPE 中，制定一个中心定位姿态标签。我们固定 Parallel SPPE 的所有层的所有权重来增强 STN 去提取一个单人姿态区域。

STN and SDTN（spatial de-transformer network，空间反变换网络）：STN 能很好地自动选取 ROI，使用 STN 去提取一个高质量的人体区域框。（ROI 是 Region of Interest 的简写，指的是在“特征图上的框”； 1）在 Fast RCNN 中， Roi 是指 Selective Search 完成后得到的“候选框”在特征图上的映； 2）在 Faster RCNN 中，候选框是经过 RPN 产生的，然后再把各个“候选框”映射到特征图上，得到 Rols

STN（论文阅读）可以被安装在任意 CNN 的任意一层中——将原来的一层结果 U ，变换到了 V ，中间并没有卷积的操作。通过 U 到 V 的变换，相当于又生成了一个新数据，而这个数据变换不是定死的而是学习来的，即使是学习来的，那它就有让 loss 变小的作用，也就是说，通过对输入数据进行简单的空间变换，使得特征变得更容易分类（往 loss 更小的方向变化）。另外一方面，有了 STN，网络就可以动态地做到旋转不变性，平移不变性等原本认为是 Pooling 层做的事情，同时可以选择图像中最终要的区域（有利于分类）并把它变换到一个最理想的姿态（比如把字放正）。在得到高质量的人体检测框后，可以使用现成的 SPPE 算法来继续高精度的人体姿态检测，在训练阶段中，SSTD 和 SPPE 一起进行 fine-tuned。

注意：不准确的检测框经过 STN+SPPE+SDTN，STN 对人体区域框中的姿态进行形态调整，输入 SPPE 做姿态估计后得到姿态线（人体骨骼框架），再用 SDTN 把姿态线映射到原始的人体区域框中，以此来调整原本的框，使框变成精准的。

Parallel SPPE：为了进一步帮助 STN 去提取更好的人体区域位置，在训练阶段添加了一个 Parallel SPPE 分支。这个 Parallel SPPE 和原来的 SPPE 用同一个 STN 模块，和 SPPE 并行处理时候，忽略 SDTN 模块。这个分支的人体姿态标签被指定为中心,更准确的说，SPPE 网络的输出直接和人体真实姿态的标签进行对比。在训练过程中会关闭 Parallel SPPE 的所有层

（????），我们固定这个分支的权重，其目的是将中心位置的位姿误差反向传播到 STN 模块。如果 STN 提取的姿态不是中心位置，那么 Parallel SPPE 会返回一个较大的误差。通过这种方式，我们可以帮助 STN 聚焦在正确的中心位置并提取出高质量的区域位置。Parallel SPPE 只有在训练阶段才会产生作用。

Discussions: Parallel SPPE 可以看作是训练阶段的正则化过程，有助于避免局部最优的情况（STN 不能把姿态转换到提取到人体区域框的居中位置）。但是 SDTN 的反向修正可以减少网络的错误进而降低陷入局部最优的可能性。这些错误对于训练 STN 是很有影响的。通过 Parallel SPPE，可以提高 STN 将人体姿态移动到检测框中间的能力。

2、训练

Alphapose 主要由两个部分的训练组成。

第一部分是目标检测，这里用的 yolov3，输入尺寸为 320*320，原文用的 yolov3-spp，输入尺寸为 608*608，它只比 yolov3 多了最后的 spp 结构（空间金字塔池化：为了解决图片大小不一的问题），我们这里已经固定了输入图片的尺寸，同时我们的训练数据集都是官方的标准数据集，大小基本一致，所以 SPP 在这里基本不起什么作用，因此没有使用该结构。基于 darknet 训练，生成 .weights 文件，利用的 coco 数据集，训练过程参照 (<https://pjreddie.com/darknet/yolo/>)，最后训练了 120000 步，保存为 yolov3-120000.weights 模型文件，比官方提供的 yolov3-spp.weights 大小少了 10M 左右（中间还尝试了 pytorch 版本的 yolov3 的训练，但是效果不好）。

第二部分是 SPPE 部分（姿态估计）训练，主干网络可以使用 ResNet50 和 ResNet101，我这里使用了 ResNet101，同样使用的是 coco 数据集，分为两个阶段，第一阶段不使用 PGPG（相当于数据增强，增强训练数据），训练 50 个 epoch，验证集准确率接近 88%，在第一阶段训练的基础上进行第二阶段训练，加上 PGPG，训练了 10 个 epoch 之后，验证集准确率并没有提高甚至还有下降的趋势，应该是第一阶段训练已经达到了饱和状态，因此中断训练，最后只使用第一阶段训练的模型 model_49.pkl。

3、效果测试

本来官方源码测试结果是只有人体骨骼关键点和骨骼，这里我加上了人体目标检测框，使得目标更醒目

对视频测试:

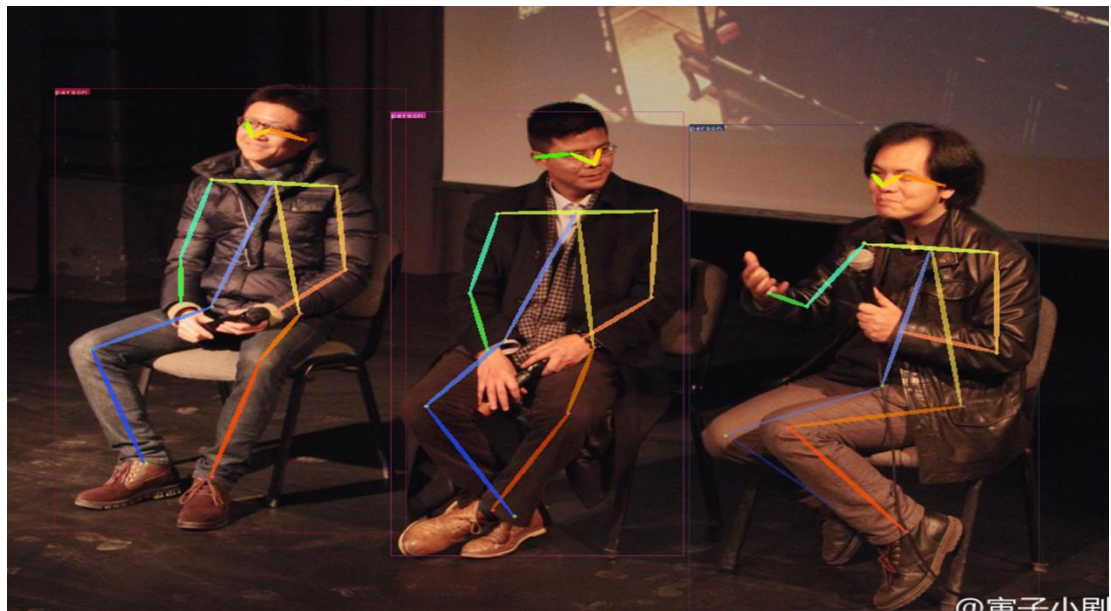
利用训练好的 `model_49.pkl` 以及目标检测模型 `yolov3_120000.weights` 对 10fps 的原始视频进行测试:

```
ai@GPU5:/data/ai/zgm/AlphaPose-pytorch$ CUDA_VISIBLE_DEVICES=1,2 python video_demo.py --video /data/ai/zgm/AlphaPose-pytorch/examples/video/test.avi --outdir examples/res7 --save_video  
fps 10.0  
Loading YOLO model..  
Loading pose model from train_sppe/exp/coco/exp1/model_49.pkl  
start jishi-----  
100%|██████████████████████████████| 795/795 [01:31<00:00, 8.64it/s]  
cost time : 91.98217034339905
```



对图片测试:

```
ai@GPU5:/data/ai/zgm/AlphaPose-pytorch$ CUDA_VISIBLE_DEVICES=1,2 python demo.py
--list examples/list3.txt --indir /data/ai/zgm/AlphaPose-pytorch/examples/demo -
-outdir examples/demo_results --save_img
Loading YOLO model..
load pose model
Loading pose model from train_sppe/exp/coco/exp1/model_49.pkl
start jishi-----
100%|████████████████████████████████████████████████████████████████████████████████| 14/14 [00:00<00:00, 15.26it/s]
cost time : 0.9207022190093994
Finish Model Running
```



4、评估

利用 cocoapi 对 alphapose 的模型进行评估测试，测试集选取的是 coco_val2017，共 5000 张图片，评估结果如下：

```
DONE (t=0.01s).
Average Precision (AP) @[ IoU=0.50:0.95 | area= all | maxDets= 20 ] = 0.607
Average Precision (AP) @[ IoU=0.50 | area= all | maxDets= 20 ] = 0.860
Average Precision (AP) @[ IoU=0.75 | area= all | maxDets= 20 ] = 0.633
Average Precision (AP) @[ IoU=0.50:0.95 | area=medium | maxDets= 20 ] = 0.544
Average Precision (AP) @[ IoU=0.50:0.95 | area= large | maxDets= 20 ] = 0.706
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 20 ] = 0.645
Average Recall (AR) @[ IoU=0.50 | area= all | maxDets= 20 ] = 0.876
Average Recall (AR) @[ IoU=0.75 | area= all | maxDets= 20 ] = 0.667
Average Recall (AR) @[ IoU=0.50:0.95 | area=medium | maxDets= 20 ] = 0.585
Average Recall (AR) @[ IoU=0.50:0.95 | area= large | maxDets= 20 ] = 0.733
ai@GPU5:/data/ai/zgm/AlphaPose-pytorch$
```

可以看到当 $\text{IoU}=0.5$ 时， $\text{AP}=0.86$ ，这个评测结果相当不错。

二、将 Alphapose 部署到 docker

流程如下：

- 1、往 docker 上部署模型的前期工作主要参考 svn 上的学习资料文件夹下的模型发布平台，具体实例的部署参考对应的 pdf，包括如何部署实例，及实例的调用等。
- 2、svn 个人文件夹下的 Alphapose_detection-0.2.3.zip 中主要包含 model.json，model.conf，need 文件夹，model_dir 文件夹和主文件 Alphapose_demo.py，Alphapose_demo.py 主要继承 AbstractModelAction，重写 load 方法加载自己的模型，重写 action 方法，实现自己功能，model_dir 放的是模型文件，need 是需要调用的文件（由于 docker 上没有 gpu，只有 cpu，因此需要将所有代码文件全部修改成 cpu 版本），model.conf 对用到模型目录、函数名和类名的配置说明；model.json 主要是定义模型的名字和模型的版本（注意的是：模型的名字与版本，要与后面在 docker 上模型的定义管理名字和版本要一致）



打包成 zip（只能 zip 包）包之后，首先进行 pc 端权限的配置：

模型实例部署

1. 配置host文件

C:/Windows/System32/drivers/etc/host 文件中加入：

192.168.51.142 fms-oss-test.vemic.com

192.168.51.14 dev-umc-ptl.vemic.com

192.168.51.14 dev-umc.vemic.com

2. 登录192.168.51.14

3. 进入模型系统选项

进入模型管理：

（1）进行模型定义管理，首先定义一个模型，注意模型定义的名字和上传的 zip 包中的 json 中模型的名字应该一致，包括版本号也一致；

（2）自定义模型管理：添加刚才定义的模型，包括定义名，版本号，存储源 FFS 默认，上传对应的 zip 包，上传完之后，还有维度信息的设置（主要是维度类型：设置图片；其他也可以设置为文本）

（3）模型发布管理：对定义的模型进行发布，版本信息是 python，实例信息：fms-python3-tf18-56.123_9260（部门部署 docker 服务的机器）；模型信息：选择刚定义的模型名；

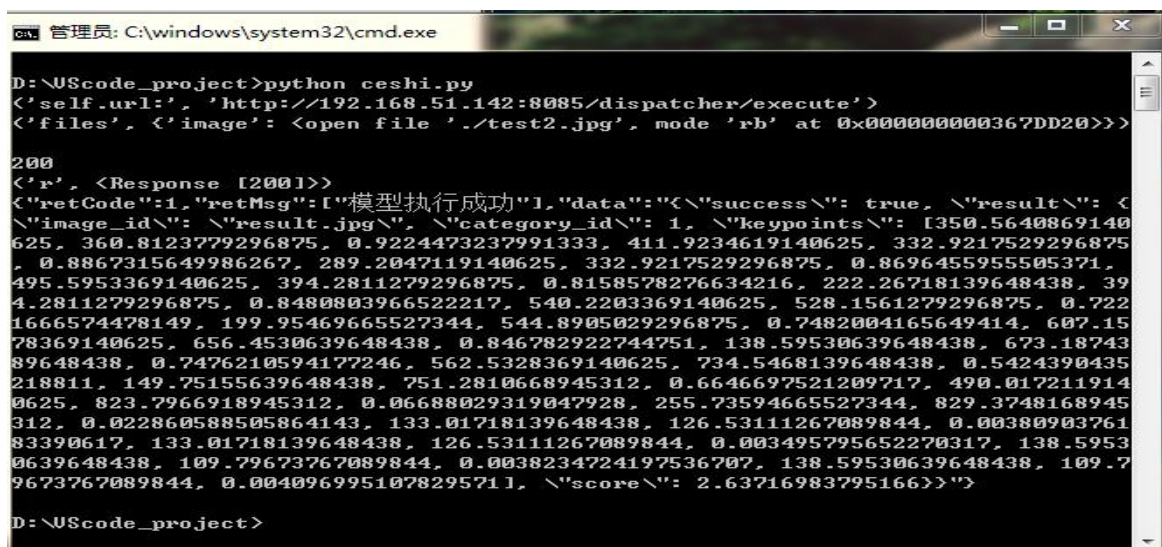
3、模型发布启动成功后，通过 python 的 http 进行文件传输并查看效果，代码如下：

```

1  # -*- coding: utf-8 -*-
2  # @Time    : 2018/12/12  9:37
3  # @Author  : zhugaoming
4  # @FileName: ceshi.py
5
6
7  #import httplib2
8  import urllib
9  import requests
10 import json
11 import sys
12 import io
13
14
15 path = './test2.jpg'
16 class FMSClient(object):
17     def __init__(self, ip, port, timeout):
18         self.url = "http://" + ip + ":" + port + "/dispatcher/execute"
19         print ("self.url:",self.url)
20         self.timeout = timeout
21
22 if __name__ == '__main__':
23
24     client = FMSClient("192.168.51.142", "8085", 5000)
25     #httpClient = httplib2.HTTPConnectionWithTimeout(client.url, 8085, timeout=client.timeout)
26     #print ("httpClient:",httpClient)
27
28
29     files = {'image': open(path, 'rb')}
30     print ("files",files)
31
32     params = {'modelName':"Alphapose_detection", 'modelVersion': "0.2.3"}
33     r = requests.post(client.url, params, files = files)
34     print(r.status_code)
35
36     print("r",r)
37     print(r.text)

```

返回结果如下，返回的是关键点坐标，由于是 cpu 执行，一张图片从输入到最后返回结果需要 10 多秒的时间：



```

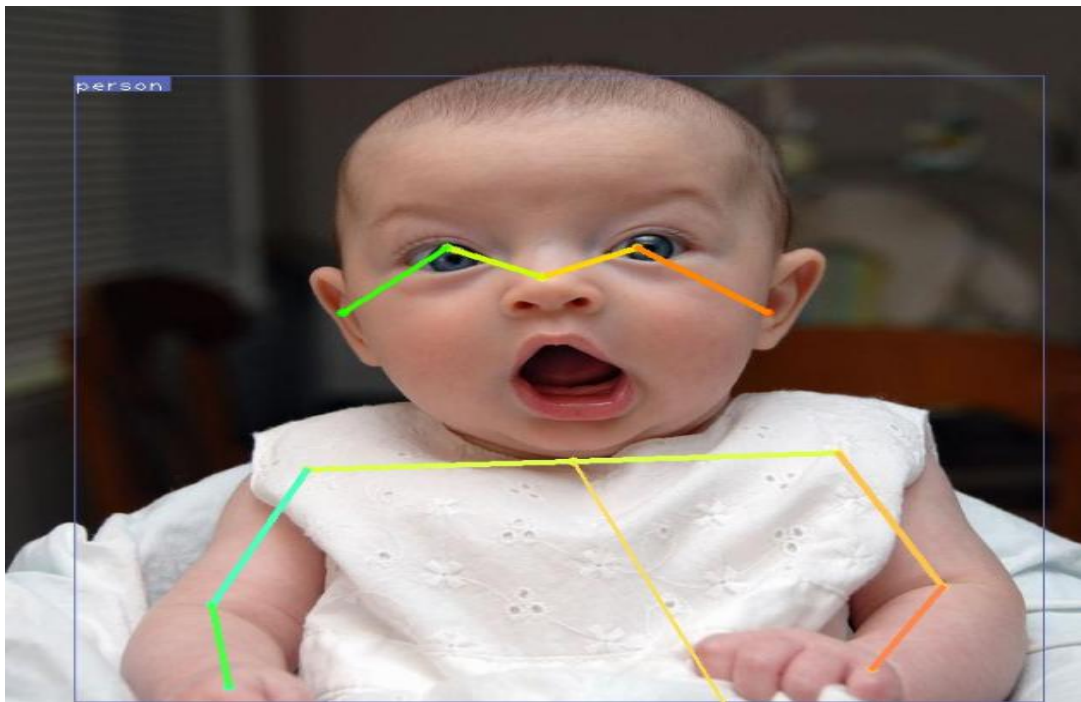
管理员: C:\windows\system32\cmd.exe

D:\VSCode_project>python ceshi.py
<'self.url':, 'http://192.168.51.142:8085/dispatcher/execute'>
<'files', <'image': <open file './test2.jpg', mode 'rb' at 0x00000000367DD20>>>

200
<'r', <Response [200]>>
<'retCode':1,"retMsg":["模型执行成功"],"data":{"\success\: true, \result\: <
\image_id\: \result.jpg\", \category_id\: 1, \keypoints\: [350.5640869140
625, 360.8123779296875, 0.9224473237991333, 411.9234619140625, 332.9217529296875
, 0.8867315649986267, 289.2047119140625, 332.9217529296875, 0.8696455955505371,
495.5953369140625, 394.2811279296875, 0.8158578276634216, 222.26718139648438, 39
4.2811279296875, 0.8480803966522217, 540.2203369140625, 528.1561279296875, 0.722
1666574478149, 199.95469665527344, 544.8905029296875, 0.7482004165649414, 607.15
78369140625, 656.4530639648438, 0.846782922744751, 138.59530639648438, 673.18743
89648438, 0.7476210594177246, 562.5328369140625, 734.5468139648438, 0.5424390435
218811, 149.75155639648438, 751.2810668945312, 0.6646697521209717, 490.017211914
0625, 823.7966918945312, 0.06688029319047928, 255.73594665527344, 829.3748168945
312, 0.022860588505864143, 133.01718139648438, 126.53111267089844, 0.00380903761
83390617, 133.01718139648438, 126.53111267089844, 0.003495795652270317, 138.5953
0639648438, 109.79673767089844, 0.0038234724197536707, 138.59530639648438, 109.7
9673767089844, 0.0040969951078295711, \score\: 2.63716983795166}>>"}
D:\VSCode_project>

```


进入后台可以看到测试效果图：



至此，将 Alphapose 部署到 docker 工作完成。

三、人体姿态估计移动端的实现

1、背景说明

由于 Alphapose 最后训练得到的网络模型一共有 400 多兆，如此大的模型移植到移动端会造成内存溢出闪退的问题、同时我们知道目前的移动手机基本都是基于 cpu 的，而从上面部署到 docker 可以看出，只使用 cpu 的话，一张图片就需要 10 多秒的时间，而移动端需要具备实时性，因此将 Alphapose 移植到移动端可行性不大，因此需要考虑其他网络模型。经过大量的调研和检索，最后只找到一个成功移植到移动端的人体姿态估计 demo，(<https://github.com/edvardHua/PoseEstimationForMobile>)，这个 demo 是基于 CPM 网络的。

CPM 网络 (<https://blog.csdn.net/mpsk07/article/details/79522809>) 使用 CNN 进行人体姿态估计，它的主要贡献在于使用顺序化的卷积架构来表达空间信息和纹理信息。顺序化的卷积架构表现在网络分为多个阶段，每一个阶段都有监督训练的部分。前面的阶段使用原始图片作为输入，后面阶段使用之前阶段的特征图作为输入，主要是为了融合空间信息，纹理信息和中心约束。另外，对同一个卷积架构同时使用多个尺度处理输入的特征和响应，既能保证精度，又考虑了各部件之间的远近距离关系。

2、实现过程

说明：因为 CPM 是直接对整张原始输入图片进行学习和姿态估计，它利用卷积神经网络的方法从单目 RGB 图像中内隐地利用图像的 feature 与图像相关的空间模型学习了人体姿态。这可能导致后面特征提取受到背景影响，因此在此基础上，我们在 CPM 之前加入了人体目标检测，即首先进行人体目标的检测，检测到人体后，将 heatmap 作为原始输入图片（通俗讲就是将人体检测框图作为输入），利用 CPM 进行姿态的估计，这个思路和 Alphapose 十分相似。

训练：这里我使用 yolov3-tiny 作为目标检测的模型，它是 yolov3 的压缩版，适用于移动端。这里依然使用 darknet 进行模型的训练，这里数据集我使用的是 pascal_voc2007，训练过程依然参照（<https://pjreddie.com/darknet/yolo/>）。

训练注意点：

(1) 这里我只训练了 person 一类，因此需要将下载下来的 VOC2007 数据集进行处理，挑选出只有 person 类的图片和标注数据。处理过程如下：

a: 根据上面的链接教程将下载下来的数据集整合成一个数据集 VOCdevkit，将文件夹下的三个子文件夹全部加上后缀“_all”，将/VOCdevkit/VOC2007/ImageSets_all/Main 下的 person_trainval.txt 和 person_test.txt 合并，然后根据训练和测试文件将包含人的图片名筛选出来，保存为 person_name.npy，同时将所有的 person 图片保存到 JPEGImages 中；

b: 根据 person_name.npy 和 Annotations_all 将图片名，标签和位置写入 txt 文件，保存到 person_label 文件夹

c: 根据 person_label 和 JPEGImages_all 生成 person 的所以标注文件.xml 保存到 Annotations 中。

d: 根据 ImageSets_all/Main 下的 person_train.txt，person_test.txt 和 person_val.txt 将含有 person 的图片序号找出来，分别保存为 train.txt，test.txt 和 val.txt 放入 ImageSets/Main

至此，只有 person 类的的数据全部处理好，然后继续根据上述链接教程执行 voc_label.py 及后续操作。其中 voc.data 中 classes 改为 1，voc.names 中只有 person。

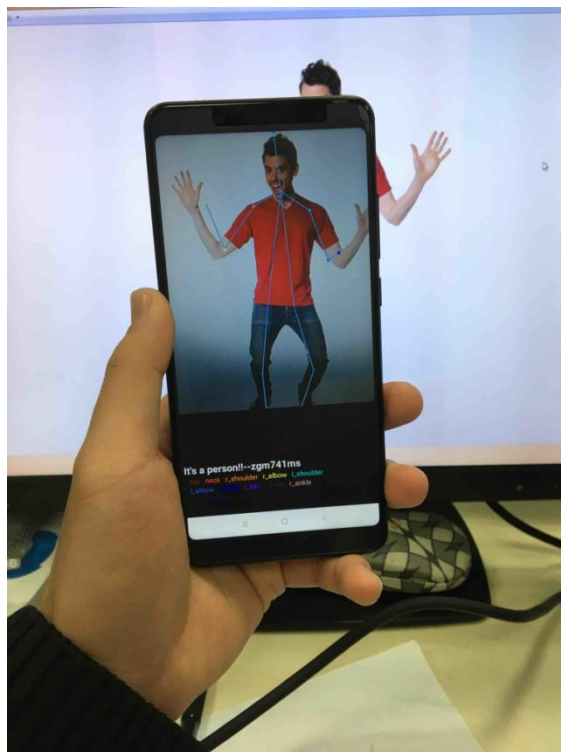
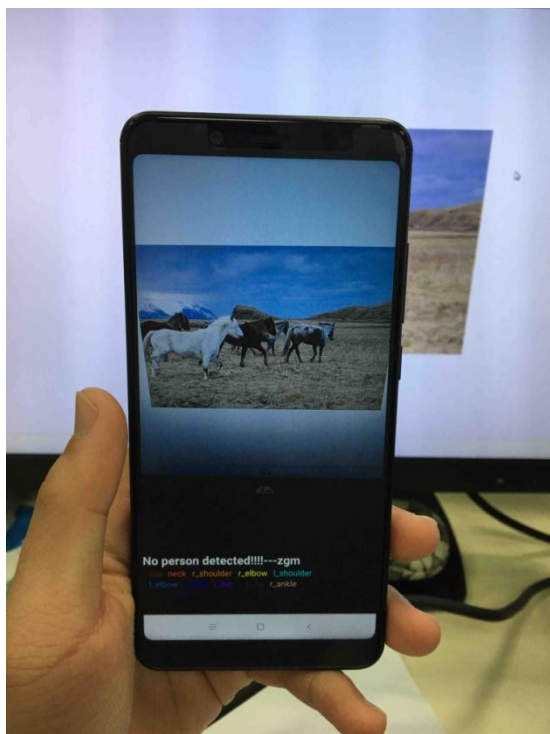
(2) 数据处理好后，开始修改 yolov3.cfg 文件，修改其中的 classes 和 filters 等参数。

(3) 开始正常训练。训练了 180000 步，loss 从一开始的 103 下降到现在的 0.3 左右，手动终止了训练过程。保存为 yolov3-tiny_180000.weights 文件，大小为 33M 左右。

模型转换：由于 darknet 生成的是.weights 模型文件，而移植到手机端需要 pb 文件，本来与 darknet 配套的工具 darkflow 可以直接将.weights 文件转换为 pb 文件，但是仅限于 yolov3 以下版本，而我这里的 yolov3-tiny 是属于 yolov3 版本，因此不可以直接转换，只能先将.weights 文件转换为 keras 对应的 h5 模型（<https://github.com/qgwweee/keras-yolo3/>），然后再将 h5 转换为 pb 格式（https://github.com/amir-abdi/keras_to_tensorflow）。最后成功转换得到 yolov3-tiny.pb。（由于部署到移动端需要给出相应的 input 和 output，因此重新加载 pb 模型文件，利用 tensorboard 进行可视化处理，可以清晰看到网络各层的结构）

导入手机：在 Andriod studio 中打开之前的 PoseEstimationForMobile 的 demo，将 yolov3-tiny.pb 放入 assets 文件夹，并将 tensorflow 源码的官方 andriod_demo 中的 TensorFlowYoloDetector.java 和 Classifier.java 文件导入相应文件夹，一系列环境编译好之后，根据上面的 tensorboard 可视化结果，给出 input 和 output 并开始编写相应的代码将 yolove-tiny 目标检测和人体姿态估计串联起来。

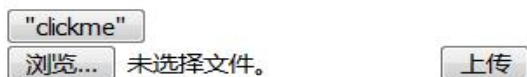
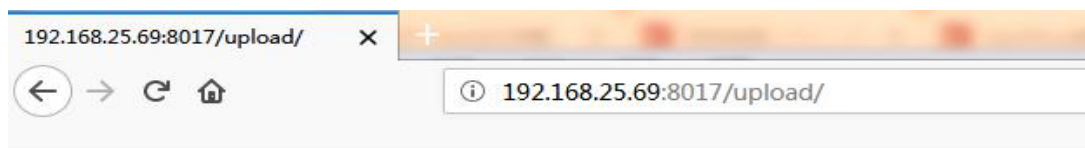
效果如下所示：当图片中没有 person 时候，会给出相应的提示，有 person 时候也会给出提示并绘制出关键点和骨骼。但是也会发现延迟较高，在红米 note5 上平均延迟 700 多 ms，同时关键点和骨骼也没有完全契合人体，但总体姿态基本相似，这可能与目标检测时候 heatmap 区域过大有关（即人体检测框范围较大），也可能是关键点和骨骼映射回原图时距离和角度参数设置的不好有关。



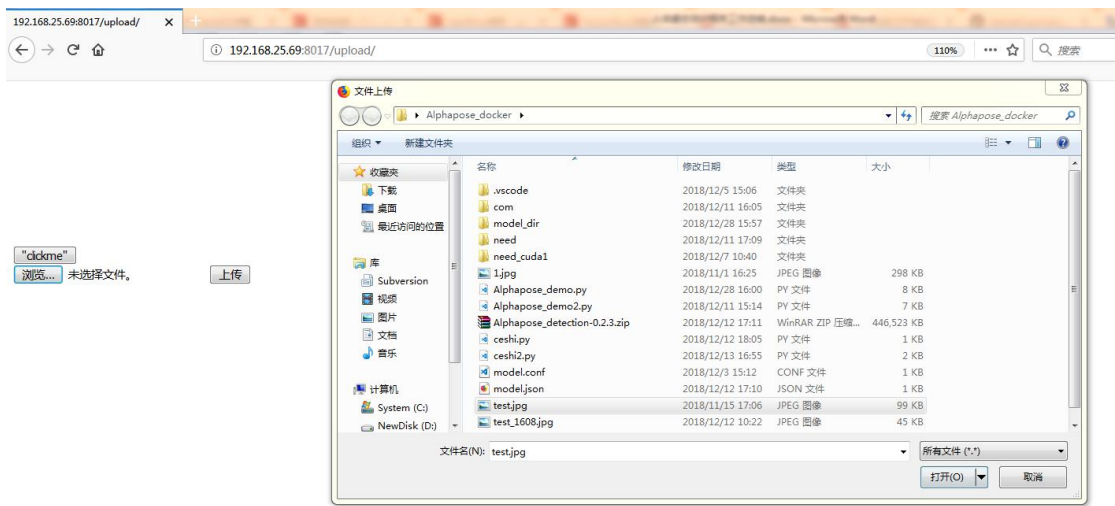
四、利用 Django 在 web 端调用 Alphapose 模型

配置好 django 环境后，新建了一个 django 项目，然后这里我直接将之前在 docker 上部署模型时用到的各个文件直接拷贝到新建的 django 项目文件夹下，编写相关的代码，调用相应的文件，最后的页面展示如下，由于使用的是 cpu 运行，所以一张图片从上传到最后展示结果需要 10 多秒的时间。

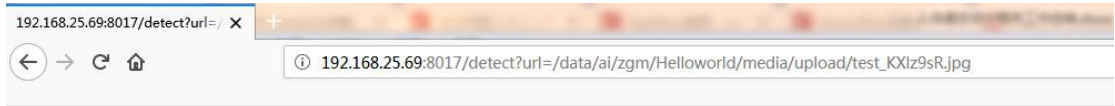
浏览器中输入：<http://192.168.25.69:8017/upload/>，页面如下：



点击浏览，选择图片点击上传：



最后检测效果：



人体检测框图：



骨骼及框标注图：

检测及标注时间：12.811578273773193 s

后台运行:

```
[02/Jan/2019 08:00:41] "GET /media/detect/test_KXlz9sR.jpg HTTP/1.1" 200 50571
^Caig@GPU5:/data/ai/zgm/Helloworld$ python manage.py runserver 192.168.25.69:8017
Performing system checks...

System check identified no issues (0 silenced).
January 02, 2019 - 08:00:33
Django version 2.0.2, using settings 'Helloworld.settings'
Starting development server at http://192.168.25.69:8017/
Quit the server with CONTROL-C.
[02/Jan/2019 08:00:45] "GET /upload/ HTTP/1.1" 200 527
IMG object (None)
save-----
name upload/test_KXlz9sR.jpg
url: /data/ai/zgm/Helloworld/media/upload/test_KXlz9sR.jpg
[02/Jan/2019 08:00:51] "POST /upload/ HTTP/1.1" 302 0
Loading YOLO model..
Loading pose model from model_dir/model_49.pkl
load done
orgurl media/upload/test_KXlz9sR.jpg
detecturl: media/detect/test_KXlz9sR.jpg
draw-----
x tensor([ 0.0000, 42.5215, 59.4622, 477.6977, 617.0540, 0.9966,
          1.0000, 0.0000])
label: person
draw end!!!!
result--> {'image_id': 'result.jpg', 'category_id': 1, 'score': 2.6857316493988037, 'keypoints':
[210.88671875, 137.01165771484375, 0.8600544929504395, 230.84765625, 117.05072021484375, 0.8777764
439582825, 198.91015625, 125.03509521484375, 0.9743959307670593, 298.71484375, 137.01165771484375,
0.8379865288734436, 182.94140625, 144.99603271484375, 0.019127780571579933, 378.55859375, 268.753
84521484375, 0.6409925222396851, 139.02734375, 272.74603271484375, 0.6892663836479187, 450.4179687
5, 428.44134521484375, 0.7321545481681824, 63.17578125, 432.43353271484375, 0.6824381351470947, 35
0.61328125, 316.66009521484375, 0.7779421210289001, 178.94921875, 384.52728271484375, 0.5383805036
5448, 326.66015625, 624.0585327148438, 0.2059108316898346, 166.97265625, 620.0663452148438, 0.1486
3349497318268, 354.60546875, 340.61322021484375, 0.002318961312994361, 111.08203125, 424.449157714
84375, 0.008198081515729427, 234.83984375, 356.58197021484375, 0.003756407182663679, 338.63671875,
308.67572021484375, 0.009946860373020172]}
cost_time: 12.322558879852295
detect end-----
[02/Jan/2019 08:01:06] "GET /detect?url=/data/ai/zgm/Helloworld/media/upload/test_KXlz9sR.jpg HTTP
/1.1" 200 285
[02/Jan/2019 08:01:06] "GET /media/face/test_KXlz9sR.jpg HTTP/1.1" 200 81581
[02/Jan/2019 08:01:06] "GET /media/upload/test_KXlz9sR.jpg HTTP/1.1" 200 100547
[02/Jan/2019 08:01:06] "GET /media/detect/test_KXlz9sR.jpg HTTP/1.1" 200 95854
```