目 录

第1章 基于 Csharp 和 OpenVINO 部署 PP-TinyPose 模型

1.1 项目概述

1.1.1 OpenVINO™

OpenVINO™是英特尔基于自身现有的硬件平台开发的一种可以加快高性能计算机视觉和深度学习视觉应用开发速度工具套件，用于快速开发应用程序和解决方案，以解决各种任务（包括人类视觉模拟、自动语音识别、自然语言处理和推荐系统等）。

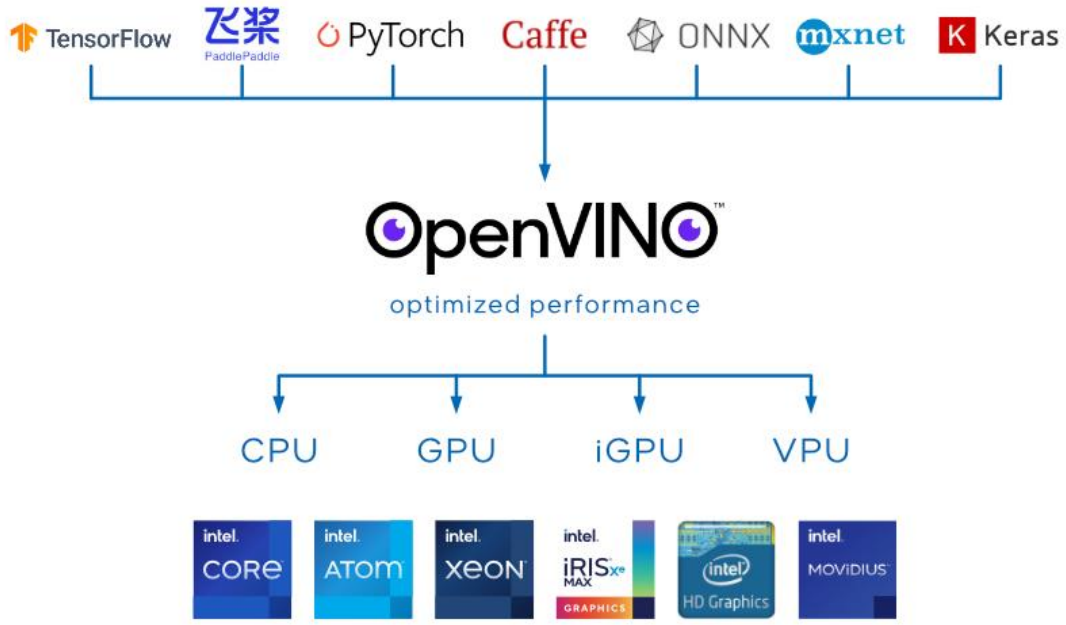

图 1 OpenVINO™ 工具结构图

该工具套件基于最新一代的人工神经网络，包括卷积神经网络 (CNN)、递归网络和基于注意力的网络，可扩展跨英特尔® 硬件的计算机视觉和非视觉工作负载，从而最大限度地提高性能。它通过从边缘到云部署的高性能、人工智能和深度学习推理来为应用程序加速，并且允许直接异构执行。极大的提高计算机视觉、自动语音识别、自然语言处理和其他常见任务中的深度学习性能；使用使用流行的框架（如 TensorFlow，PyTorch 等）训练的模型；减少资源需求，并在从边缘到云的一系列英特尔®平台上高效部署；支持在 Windows 与 Linux 系统，且官方支持编程语言为 Python 与 C++语言。

OpenVINO™ 工具套件 2022.1 版于 2022 年 3 月 22 日正式发布，与以往版本相比发生了重大革新，提供预处理 API 函数、ONNX 前端 API、AUTO 设备插件，并且支持直接读入飞桨模型，在推理中中支持动态改变模型的形状，这极大地推动了不同网络的应用落地。2022 年 9 月 23 日，OpenVINO™ 工具套件 2022.2 版推出，对 2022.1 进行了微调，以包括对英特尔最新 CPU 和离散 GPU 的支持，以实现更多的人工智能创新和机会。

1.1.2 PaddleDetection 关键点识别模型 PP-TinyPose

飞桨(PaddlePaddle)是集深度学习核心框架、工具组件和服务平台为一体的技术先进、

功能完备的开源深度学习平台，已被中国企业广泛使用，深度契合企业应用需求，拥有活跃的开发者社区生态。提供丰富的官方支持模型集合，并推出全类型的高性能部署和集成方案供开发者使用。是中国首个自主研发、功能丰富、开源开放的产业级深度学习平台。

PaddleDetection 为基于飞桨 PaddlePaddle 的端到端目标检测套件，内置 30+模型算法及 250+预训练模型，覆盖目标检测、实例分割、跟踪、关键点检测等方向，其中包括服务器端和移动端高精度、轻量级产业级 SOTA 模型、冠军方案和学术前沿算法，并提供配置化的网络模块组件、十余种数据增强策略和损失函数等高阶优化支持和多种部署方案，在打通数据处理、模型开发、训练、压缩、部署全流程的基础上，提供丰富的案例及教程，加速算法产业落地应用。

图 2 PaddleDetection 应用

PP-TinyPose 是 PaddleDetecion 针对移动端设备优化的实时关键点检测模型，可流畅地在移动端设备上执行多人姿态估计任务。借助 PaddleDetecion 自研的优秀轻量级检测模型 PicoDet，同时提供了特色的轻量级垂类行人检测模型。PP-TinyPose 可以基于人体 17 个关键点数据集训练后，识别人体关键点，获得人体姿态，如图 3 所示。

图 3 PP-TinyPose 识别效果图

1.1.3 项目介绍

该项目基于 OpenVINO™ 模型推理库，在 C#语言下，调用封装的 OpenVINO™ 动态链接库，部署推理 PP-TinyPose 人体关键点识别模型，实现了在 C#平台调用 OpenVINO™ 部署 PP-TinyPose 人体关键点识别模型。

如图 4 所示，PaddlePaddle 向我们提供了完整的人体关键点识别解决方案，主要包括

行人检测以及关键点检测两部分。人体检测主要是实现行人位置检测，在多人关键点识别任务中，可以做行人区域划分等工作，此处飞桨提供了轻量级 PicoDet 行人识别模型，用于行人区域识别。关键点识别采用的是基于 Lite-HRNet 骨干网络的 PP-TinyPose 模型，并增加了 DARK 关键点矫正算法，使模型关键点识别更加精准；且该网络至此多 bath_size 推理，可以实现同时多图片推理运算。

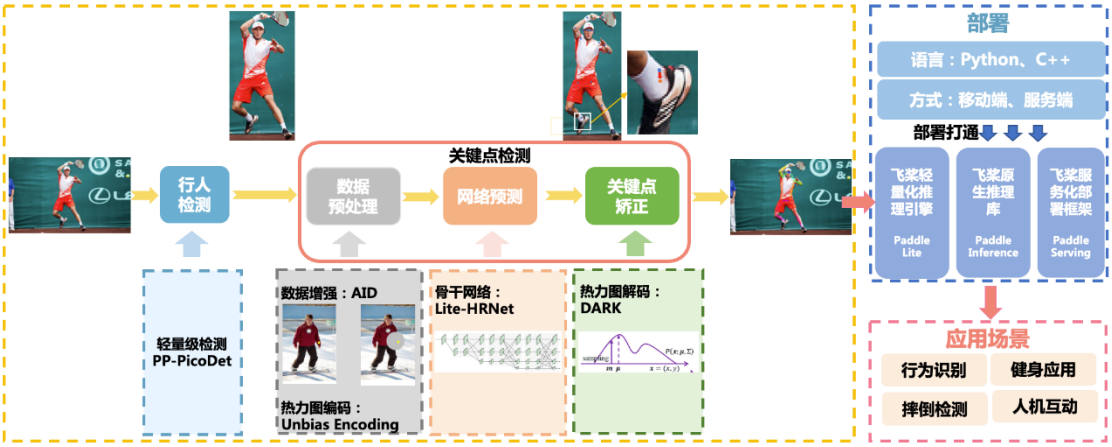

图 4 PP-TinyPose 人体关键点识别

1.1.4 项目编码环境

为了防止复现代码出现问题，列出以下代码开发环境，可以根据自己需求设置，注意 OpenVINO™ 一定是 2022 版本，其他依赖项可以根据自己的设置修改。

- 操作系统：Windows 11
- OpenVINO™：2022.2.0
- OpenCV：4.5.5
- Visual Studio：2022
- C#框架：.NET 6.0
- OpenCvSharp：OpenCvSharp4

1.1.5 源码下载方式

项目所使用的源码均已经在 Github 和 Gitee 上开源，

Github:

```
git clone https://github.com/guojin-yan/Csharp_and_OpenVINO_deploy_PP-TinyPose
```

Gitee:

```
git clone  https://gitee.com/guojin-yan/Csharp_and_OpenVINO_deploy_PP-TinyPose.git
```

1.2 C#中调用 OpenVINO™ 实现

1.2.1 OpenVINO™ 动态链接库

由于 OpenVINO™ 只有 C++和 Python 接口，无法直接在 C#中使用 OpenVINO™ 部署模型，为了实现在 C#中使用，通过动态链接库的方式实现。

该方式具体实现教程，可以参考下面这一篇 Github：

```
https://github.com/guojin-yan/OpenVinoSharp.git
```

为了方便直接使用，该项目中已经包含了上述项目创建动态链接库的代码，并将 OpenVINO™ 升级到了 2022.2 版本，在使用时可以直接配置使用。

1.2.2 C#引入动态链接库文件

在 C#中需要使用[DllImport()]方法引入动态链接库文件，其完整的使用方式如以下代码所示：

```
[DllImport(openvino_dll_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]
public extern static IntPtr set_input_image_sharp(IntPtr inference_engine, string input_node_name, ref ulong input_size);
```

针对[DllImport()]括号中的内容：openvino_dll_path 为 dll 文件路径，CharSet = CharSet.Unicode 代表支持中文编码格式字符串，CallingConvention = CallingConvention.Cdecl 指示入口点的调用约定为调用方清理堆栈。

在声明动态链接库后，就可以引入动态链接库中的方法，由于我们在 C++环境下生成的动态链接库，为了让编译器识别，需要方法名、变量类型一一对应，才可以引入成功：

表 1 C++与 C#方法对应关系

	C++	C#
返回值类型	void*	IntPtr
方法名	set_input_image_sharp	set_input_image_sharp
参数 1	void*	IntPtr
参数 2	wchar_t*	string
参数 3	size_t *	ref ulong

基于以上方法，我们将动态链接库中的所有方法引入到 C#中。

```
private const string openvino_dll_path = @"E:\Git_space\基于Csharp和OpenVINO部署PaddleOCR模型\CppOpenVinoAPI\dll\OpenVinoSharp.dll";

[DllImport(openvino_dll_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]
1 个引用
public extern static IntPtr core_init(string model_file, string device_name);

[DllImport(openvino_dll_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]
1 个引用
public extern static IntPtr set_input_image_sharp(IntPtr inference_engine, string input_node_name, ref ulong input_size);

[DllImport(openvino_dll_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]
1 个引用
public extern static IntPtr set_input_data_sharp(IntPtr inference_engine, string input_node_name, ref ulong input_size);

[DllImport(openvino_dll_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]
1 个引用
public extern static IntPtr load_image_input_data(IntPtr inference_engine, string input_node_name, ref byte image_data, ulong image_size, int type);

[DllImport(openvino_dll_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]
1 个引用
public extern static IntPtr load_input_data(IntPtr inference_engine, string input_node_name, ref float input_data);

[DllImport(openvino_dll_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]
1 个引用
public extern static IntPtr core_infer(IntPtr inference_engine);

[DllImport(openvino_dll_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]
1 个引用
public extern static void read_infer_result_F32(IntPtr inference_engine, string output_node_name, int data_size, ref float inference_result);

[DllImport(openvino_dll_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]
1 个引用
public extern static void read_infer_result_I32(IntPtr inference_engine, string output_node_name, int data_size, ref int inference_result);

[DllImport(openvino_dll_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]
1 个引用
public extern static void core_delet(IntPtr inference_engine);
```

图 5 动态链接库引入

1.2.3 C#构建 Core 类

上一步我们引入了封装的 OpenVINO™ 动态链接库，为了更方便的使用，将其封装到 Core 类中。在不同方法之间，主要通过推理核心结构体指针在各个方法之间传递，在 C# 是没有指针这个说法的，不过可以通过 IntPtr 结构体来接收这个指针，为了防止该指针被篡改，将其封装在类中作为私有成员使用。

根据模型推理的步骤，构建模型推理类：

（1）构造函数

```
public Core(string model_file, string device_name){
    // 初始化推理核心
    ptr = NativeMethods.core_init(model_file, device_name);
}
```

在该方法中，主要是调用推理核心初始化方法，初始化推理核心，读取本地模型，将模型加载到设备、创建推理请求等模型推理步骤。

（2）设置模型输入形状

```
// @brief 设置推理模型的输入节点的大小
// @param input_node_name 输入节点名
// @param input_size 输入形状大小数组
public void set_input_sharp(string input_node_name, ulong[] input_size) {
    // 获取输入数组长度
    int length = input_size.Length;
    if (length == 4) {
        // 长度为 4，判断为设置图片输入的输入参数，调用设置图片形状方法
        ptr = NativeMethods.set_input_image_sharp(ptr, input_node_name, ref input_size[0]);
    }
    else if (length == 2) {
        // 长度为 2，判断为设置普通数据输入的输入参数，调用设置普通数据形状方法
        ptr = NativeMethods.set_input_data_sharp(ptr, input_node_name, ref input_size[0]);
    }
    else {
        // 为防止输入发生异常，直接返回
        return;
    }
}
```

OpenVINO™ 2022.1 支持模型动态输入，读入模型可以不固定输入大小，在使用时固定模型的输入大小，并且可以随时修改输入形状。当前设置情况下，至此设置二维、以及四维的输入形状，在当前模型中足够使用。

（3）加载推理数据

```
// @brief 加载推理数据
// @param input_node_name 输入节点名
// @param input_data 输入数据数组
public void load_input_data(string input_node_name, float[] input_data) {
```

```
            ptr = NativeMethods.load_input_data(ptr, input_node_name, ref input_data[0]);
        }
        // @brief 加载图片推理数据
        // @param input_node_name 输入节点名
        // @param image_data 图片矩阵
        // @param image_size 图片矩阵长度
        public void load_input_data(string input_node_name, byte[] image_data, ulong image_size, int type) {
            ptr = NativeMethods.load_image_input_data(ptr, input_node_name, ref image_data[0], image_size,
type);
        }
```

加载推理数据主要包含图片数据和普通的矩阵数据，其中对于图片的预处理，也已经在 C++中进行封装，保证了图片数据在传输中的稳定性。

（5）模型推理

```
        // @brief 模型推理
        public void infer() {
            ptr = NativeMethods.core_infer(ptr);
        }
```

（6）读取推理结果数据

```
        // @brief 读取推理结果数据
        // @param output_node_name 输出节点名
        // @param data_size 输出数据长度
        // @return 推理结果数组
        public T[] read_infer_result<T>(string output_node_name, int data_size) {
            // 获取设定类型
            string t = typeof(T).ToString();
            // 新建返回值数组
            T[] result = new T[data_size];
            if (t == "System.Int32") { // 读取数据类型为整形数据
                int[] inference_result = new int[data_size];
                NativeMethods.read_infer_result_I32(ptr, output_node_name, data_size, ref
inference_result[0]);
                result = (T[])Convert.ChangeType(inference_result, typeof(T[]));
                return result;
            }
            else { // 读取数据类型为浮点型数据
                float[] inference_result = new float[data_size];
                NativeMethods.read_infer_result_F32(ptr, output_node_name, data_size, ref
inference_result[0]);
                result = (T[])Convert.ChangeType(inference_result, typeof(T[]));
                return result;
            }
        }
```

在读取模型推理结果时，支持读取整形数据和浮点型数据，且需要知晓模型输出数据

的大小，这就要求我们对自己所使用的模型有很好的把握。

（7）清除地址

```
// @brief 删除创建的地址
public void delet() {
    NativeMethods.core_delet(ptr);
}
```

此处的清除地址需要调用 fengzhuangd 额地址删除方法实现，不可以直接删除 C#中创建的 IntPtr，这样会导致内存泄漏，影响程序性能。

通过上面的封装，比可以在 C#平台下，调用 Core 类，间接调用 OpenVINO™ 推理套件部署自己的模型了。

1.3 PP-PicoDet 行人识别

1.3.1 模型介绍-PP-PicoDet

PP-PicoDet 是由百度提出新型移动端实时检测模型，该模型拥有更高的 mAP，预测速度更快，在 ARM CPU 下可达 150FPS，并且支持 PaddleLite /MNN /NCNN /OpenVINO 等预测库部署。针对关键点识别这一块，飞桨提供了专门的行人检测训练模型，可以在直接官网下载使用。

导出的包含后处理的模型当前是无法在 OpenVINO™ 上使用的，因此在导出模型时需要关闭模型的后处理，将后处理放到代码中实现，

表 2 Picodet_s_320_lcnet_pedestrian Paddle 格式模型信息

	Input	Output	
名称	x	concat_8.tmp_0	transpose_8.tmp_0
形状	[bath_size, 3, 320, 320]	[bath_size, 2125, 4]	[bath_size, 1, 2125]
数据类型	Float32	Float32	Float32

表 2 为 Paddle 格式下模型的输入与输出相关信息，当前模型是导出的官方不带后处理的模型，是当前模型是没有固定 bath_size 的，并不能直接在推理 OpenVINO™ 使用，需要将其转为 ONNX，在转换时指定 bath_size 大小。

1.3.2 模型下载与转换

（1）PaddlePaddle 模型下载方式：

命令行直接输入以下模型导出代码，使用 PaddleDetecion 自带的方法，下载预训练模型并将模型转为导出格式。

导出 picodet_s_320_lcnet_pedestrian 模型：

```
python tools/export_model.py -c
configs/picodet/application/pedestrian_detection/picodet_s_320_lcnet_pedestrian.yml -o export.benchmark=False
export.nms=False
weights=https://bj.bcebos.com/v1/paddledet/models/keypoint/tinypose_enhance/picodet_s_320_lcnet_pedestrian.p
dparams --output_dir=output_inference
```

导出 picodet_s_192_lcnet_pedestrian 模型：

```
python tools/export_model.py -c
configs/picodet/application/pedestrian_detection/picodet_s_192_lcnet_pedestrian.yml -o export.benchmark=False
export.nms=False
weights=https://bj.bcebos.com/v1/paddledet/models/keypoint/tinypose_enhance/picodet_s_192_lcnet_pedestrian.p
dparams --output_dir=output_inference
```

此处导出模型的命令与我们常用的命令导出增加了 export.benchmark=False 和 export.nms=False 两个指令，主要是关闭模型后处理以及打开模型极大值抑制。如果不关闭模型后处理，模型会增加一个输入，且在模型部署时会出错。

（2）转换为 ONNX 格式：

该方式需要安装 paddle2onnx 和 onnxruntime 模块。导出方式比较简单，比较注意的是需要指定模型的输入形状，用于固定模型批次的大小。在命令行中输入以下指令进行转换：

```
paddle2onnx --model_dir output_inference/picodet_s_320_lcnet_pedestrian --model_filename model.pdmodel --
params_filename model.pdiparams --input_shape_dict "{'image':[1,3,320,320]}" --opset_version 11 --save_file
picodet_s_320_lcnet_pedestrian.onnx
```

1.3.3 构建行人识别 C# PicoDet 类

此处为了让代码更加规范，构建专门用于行人识别的模型推理类 PicoDet。

（1）设置成员变量

```
// 成员变量
private Core predictor; // 模型推理器
private string input_node_name = "image"; // 模型输入节点名称
private string output_node_name_1 = "concat_8.tmp_0"; // 模型预测框输出节点名
private string output_node_name_2 = "transpose_8.tmp_0"; // 模型预测置信值输出节点
private Size input_size = new Size(0,0); // 模型输入节点形状
private int output_length = 0; // 模型输出数据长度
```

在该项目中我们使用的关键点预测模型主要不同点是输入与输出节点的形状不同，其他相关信息基本一致，所以将相同的内容封装为成员变量。其中模型的输入与输出节点名一定要正确，前后不得有空格。

（2）构造方法

```
public PicoDet(string mode_path, string device_name) {
    predictor = new Core(mode_path, device_name);
}
```

在该推理类中，其核心是模型推理器，因此我们需要在构造函数中对模型推理其进行初始化，需要指定模型路径以及设备名称。

（3）设置模型输入输出节点形状方法

```
public void set_shape(Size input_image_size, int output_length)
{
    this.input_size = input_image_size;
    this.output_length = output_length;
}
```

对于行人识别模型我们在模型转换时已经固定了输入形状，此处主要是用于设置图片的缩放比例。另一点模型的输出大小由于在转换的时候未自动生成，因此需要通过模型推理工具进行查询，此处已经获取到输入为 320×320 的模型输出大小为 2125，输入大小为 192×192 的模型输出大小为 765.

（4）图片模型预测方法

```
public List<Rect> predict(Mat image)
{
    // 设置图片输入
    // 图片数据解码
    byte[] input_image_data = image.ImEncode(".bmp");
    // 数据长度
    ulong input_image_length = Convert.ToUInt64(input_image_data.Length);
    // 设置图片输入
    predictor.load_input_data(input_node_name, input_image_data, input_image_length, 0);
    // 求取缩放大小
    double scale_x = (double)image.Width / (double)this.input_size.Width;
    double scale_y = (double)image.Height / (double)this.input_size.Height;
    Point2d scale_factor = new Point2d(scale_x, scale_y);
    // 模型推理
    predictor.infer();
    // 读取模型输出
    // 读取置信值结果
    float[] results_con = predictor.read_infer_result<float>(output_node_name_2, output_length);
    // 读取预测框
    float[] result_box = predictor.read_infer_result<float>(output_node_name_1, 4 * output_length);
    // 处理模型推理数据
    List<Rect> boxes_result = process_result(results_con, result_box, scale_factor);
    return boxes_result;
}
```

模型预测过程主要实现将图片数据按照输入要求预处理输入数据，并加载到模型上，进行模型推理，以及最后将模型结果按照要求进行处理等一套图片推理流程。如果想使用 PicoDet 进行图片推理，直接调用该方法就可以实现一张图片的完整推理。方法返回值为行人位置矩形框。

（5）推理结果处理方法

```
private List<Rect> process_result(float[] results_con, float[] result_box, Point2d scale_factor)
{
    // 处理预测结果
    List<float> confidences = new List<float>();
    List<Rect> boxes = new List<Rect>();
    for (int c = 0; c < output_length; c++)
    {    // 重新构建
        Rect rect = new Rect((int)(result_box[4 * c] * scale_factor.X), (int)(result_box[4 * c + 1] *
scale_factor.Y),
```

```
                    (int)((result_box[4 * c + 2] - result_box[4 * c]) * scale_factor.X),
                    (int)((result_box[4 * c + 3] - result_box[4 * c + 1]) * scale_factor.Y));
                boxes.Add(rect);
                confidences.Add(results_con[c]);
            }
            // 非极大值抑制获取结果候选框
            int[] indexes = new int[boxes.Count];
            CvDnn.NMSBoxes(boxes, confidences, 0.5f, 0.5f, out indexes);
            // 提取合格的预测框
            List<Rect> boxes_result = new List<Rect>();
            for (int i = 0; i < indexes.Length; i++)
            {
                boxes_result.Add(boxes[indexes[i]]);
            }
            return boxes_result;
        }
```

模型直接读取的结果为 n 个预测结果的置信值以及位置框(x)，该预测结果是以图片大小为输入大小时的位置，所以需要按照原图片大小将图片转换为实际大小，另一点由于预测结果未进行非极大值抑制，因此还需要对最后结果进行非极大值抑制，最终获取实际的行人位置。

1.4 PP-TinyPose 人体姿态识别

1.4.1 模型介绍- PP-TinyPose

PP-TinyPose 是 PaddlePaddle 提供了关键点识别模型，PP-TinyPose 在单人和多人场景均达到性能 SOTA，同时对检测人数无上限，并且在微小目标场景有卓越效果，助力开发者高性能实现异常行为识别、智能健身、体感互动游戏、人机交互等任务。同时扩大数据集，减小输入尺寸，预处理与后处理加入 AID、UDP 和 DARK 等策略，保证模型的高性能。实现速度在 FP16 下 122FPS 的情况下，精度也可达到 51.8%AP，不仅比其他类似实现速度更快，精度更是提升了 130%。

表 3 PP-TinyPose 256×192 Paddle 模型信息

	Input	Output	
名称	image	conv2d_441.tmp_1	argmax_0.tmp_0
形状	[bath_size, 3, 256, 192]	[bath_size, 17, 64, 48]	[bath_size,17]
数据类型	Float32	Float32	Int64

表 3 为 Paddle 格式下 PP-TinyPose 256×192 模型的输入与输出相关信息，除此以外，飞桨还提供了输入大小为 128×96 的模型，这两类模型在部署时所有操作基本一致，主要差别就是输入与输出的形状不同。分析模型的输入和输出可以获取以下几个点：

第一模型的输入与 conv2d_441.tmp_1 节点输出形状，呈现倍数关系，具体是输入的长宽是输出的四倍， 因此我们可以通过输入形状来推算输出的大小。

第二模型 conv2d_441.tmp_1 节点输出为预测出的 17 个点的灰度图，因此后续在进行数据处理是，只需要寻找到最大值所在位置，就可以找到近似的位置。

第三通过模型，我们可以获取到 argmax_0.tmp_0 节点的输出为 conv2d_441.tmp_1 节点

输出的灰度图中最大值所在的点。

1.4.2 模型下载与转换

（1）PaddlePaddle 模型下载方式：

命令行直接输入以下代码，或者浏览器输入后面的网址即可。

```
wget https://bj.bcebos.com/v1/paddledet/models/keypoint/tinypose_enhance/tinypose_256x192.zip
```

下载好后将其解压到文件夹中，便可以获得 Paddle 格式的推理模型。

（2）转换为 ONNX 格式：

该方式需要安装 paddle2onnx 和 onnxruntime 模块。在命令行中输入以下指令进行转换：

```
paddle2onnx --model_dir inference_model/tinypose_128x96 --model_filename model.pdmodel --params_filename
model.pdiparams --opset_version 11 --save_file tinypose_128_96.onnx
```

由于 PP-TinyPose 支持多 bath_size 推理，因此在模型转换时无需指定形状尺寸。

（3）转换为 IR 格式

利用 OpenVINO™ 模型优化器，可以实现将 ONNX 模型转为 IR 格式。

```
在 OpenVINO™ 环境下，切换到模型优化器文件夹，直接使用下面指令便可以进行转换。
cd .\openvino\tools
mo –input_model tinypose_128_96.onnx
```

经过上述指令模型转换后，可以在当前文件夹下找到转换后的三个文件。

1.4.3 构建人体姿势识别 PPTinyPose 类

由于人体姿态识别设计比较复杂的数据处理，因此在此处我们将人体姿态识别模型推理方案设计为一个推理类，用于支持实现人体关键点检测，以达到人体姿态识别。最终效果为输入行人图片，推理后获得绘制人体姿态图像。

（1）成员变量

```
private Core predictor; //  模型推理器
private string input_node_name = "image"; //  模型输入节点名称
private string output_node_name_1 = "conv2d_441.tmp_1"; //  模型输出节点名称
private string output_node_name_2 = "argmax_0.tmp_0"; //  模型输出节点名称
private Size input_size = new Size(0, 0); //  模型输入节点形状
private Size output_size = new Size(0, 0); //  模型输出节点形状
private Size image_size = new Size(0, 0); //  待推理图片形状
```

目前设计的推理类支持输入形状为 256×192 和 128×96 大小的两类模型，且两类模型的输入输出的名称一致，因此将一些会用到的变量设为成员变量，方便后面调用。

（2）构造方法

```
public PPTinyPose(string mode_path, string device_name)
{
    predictor = new Core(mode_path, device_name);
}
```

模型推理器为模型推理类的核心，需要将本地模型读取到内存中和加载到指定设备

中，因此将该推理类初始化时就将该工作完成。

（3）形状设置方法

```
public void set_shape(Size input_image_size)
{
    this.input_size = input_image_size;
    this.output_size = new Size(input_image_size.Width / 4, input_image_size.Height / 4);
}
```

此处需要设置模型的输入与输出节点的形状，由于模型导出时未固定形状，因此我们需要设置模型的输出形状；另一点模型推理获得的结果需要进行缩放边换，才可以获得与原图匹配的结果。由于模型的输入与输出存在倍数关系，因此只需要指定输入形状即可。

（4）行人姿态预测方法

行人姿态预测方法输入为一张行人图片数据，输出为绘制完行人姿态的图像。该方法主要实现推理数据加载（包括数据预处理）、模型推理以及预测结果处理等步骤。

```
public Mat predict(Mat image)
{
    this.image_size.Width = image.Cols;
    this.image_size.Height = image.Rows;
    // 设置输入形状
    ulong[] input_size = new ulong[] { 1, 3, (ulong)(this.input_size.Width), (ulong)(this.input_size.Height) };
    predictor.set_input_sharp(input_node_name, input_size);
    // 设置图片输入
    // 图片数据解码
    byte[] input_image_data = image.ImEncode(".bmp");
    // 数据长度
    ulong input_image_length = Convert.ToUInt64(input_image_data.Length);
    // 设置图片输入
    predictor.load_input_data(input_node_name, input_image_data, input_image_length, 2);
    // 模型推理
    predictor.infer();
    // 读取模型输出
    // 读取模型位置输出
    long[] result_pos = predictor.read_infer_result<long>(output_node_name_2, 17);
    // 单个预测点数据长度
    int point_size = output_size.Width * output_size.Height;
    // 读取预测结果
    float[] result = predictor.read_infer_result<float>(output_node_name_1, 17 * point_size);
    // 处理模型输出结果
    float[,] points = process_result(result);
    // 绘制人体姿态
    draw_poses(points, ref image);
    return image;
}
```

PPTinyPose 模型数据处理方式不是直接缩放变换，是通过仿射变换将图像转换为输入

大小，该变换方式实现方式会在后面讲解。

（5）其他方法

其他四个私有方法主要是实现模型输出数据的后处理，主要包括处理关键点预测结果、获取模型输出最大点位置、获取仿射变换矩阵以及绘制预测结果。其实现原理在数据处理方式中讲解。

```
private float[,] process_result(float[] het_map)
{
    float[,] point_meses = new float[17, 3];
    ……
    return point_meses;
}
private int[] get_max_point(float[,] map, ref float maxval)
{
    int[] index = new int[2];
    ……
    return index;
}
Mat get_affine_transform(Point center, Size input_size, int rot, Size output_size, bool inv = false)
{
    ……
    return Cv2.GetAffineTransform(src, dst);
}
void draw_poses(float[,] points, ref Mat image)
{
    ……
}
```

1.4.4 处理关键点预测结果

由于人体关键点检测模型使用了热力图编解码数据增强，因此模型最终的输出结果处理相比其他模型来说比较复杂，总共分为 6 个过程，如图 6 所示。具体实现的过程数学原理大家可以参考参考文献[[1]。

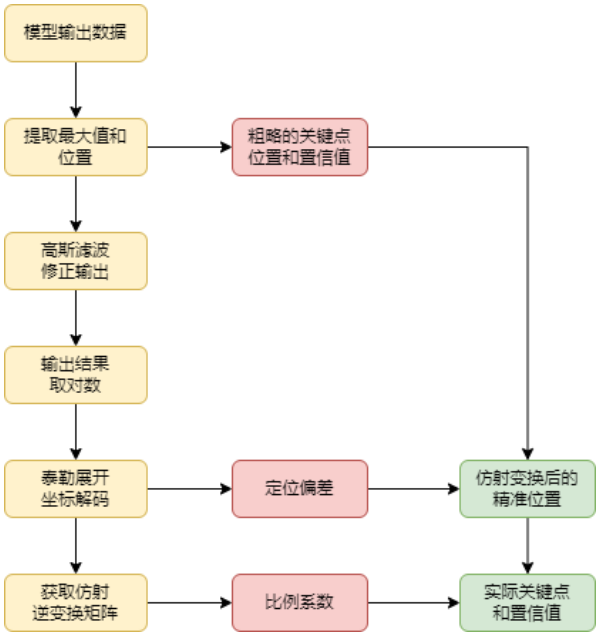

图 6 关键点预测结果处理流程

```
private int[] get_max_point(float[,] map, ref float maxval)
{
    int height = map.GetLength(0);
    int width = map.GetLength(1);
    int[] index = new int[2];
    int[] index_h = new int[height];
    float[] maxval_h = new float[height];
    for (int h = 0; h < height; h++)
    {
        float val = map[h, 0];
        for (int w = 0; w < width; w++)
        {
            if (val < map[h, w])
            {
                val = map[h, w];
                maxval_h[h] = val;
                index_h[h] = w;
            }
        }
    }
    float maxval_temp = maxval_h[0];
    for (int h = 0; h < height; h++)
    {
        if (maxval_temp < maxval_h[h])
        {
            maxval_temp = maxval_h[h];
            index[1] = h;
            index[0] = index_h[h];
            maxval = maxval_temp;
        }
    }
    return index;
}
```

图 7 get_max_point 方法实现

（1）提取最大值和位置

直接读取模型输出为一个二维矩阵，其大小为输出的 1/4，其矩阵最大值位置代表了当前位置为预测的关键点，当前位置的数值就作为该点预测的置信值。此处我们封装了一个方法 get_max_point()，用于实现提取最大点的位置。

模型输出为 17 个关键点预测图像，在提取和细化时需要一个一个点进行，所以此处依次读取每一个点的结果图像进行处理。

```
// 提取一个点结果图像
float[,] map = new float[this.output_size.Width, this.output_size.Height];
for (int h = 0; h < this.output_size.Width; h++)
{
    for (int w = 0; w < this.output_size.Height; w++)
    {
        map[h, w] =
        het_map[p * this.output_size.Width * this.output_size.Height + h * this.output_size.Height + w];
    }
}
```

然后将提取的结果点图像调用 get_max_point()方法，

```
// 通过获取最大值获得点的粗略位置
float maxval = 0;
int[] index_int = get_max_point(map, ref maxval);
// 保存关键点的信息
point_meses[p, 0] = index_int[0];
point_meses[p, 1] = index_int[1];
point_meses[p, 2] = maxval;
```

（2）高斯滤波修正输出

首先构建高斯滤波输入图像，主要是将点结果矩阵转为指定的输入格式。

```
// 高斯图像背景
Mat gaussianblur = Mat.Zeros(this.output_size.Width + 2, this.output_size.Height + 2,MatType.CV_32FC1);
Mat roi = new Mat(new List<int>() { this.output_size.Width, this.output_size.Height }, MatType.CV_32FC1,
map); // 将点结果转为 Mat 数据
Rect rect = new Rect(1,1, this.output_size.Height, this.output_size.Width);
roi.CopyTo(new Mat(gaussianblur, rect)); // 将点结果放在背景上
```

此处我们使用 OpenCvSharp 中自带的高斯滤波方法实现，滤波器直径为 3。经过高斯滤波后，获取滤波前后的最大值，按照滤波前后最大值比值关系整体缩放点数据矩阵。

```
Cv2.GaussianBlur(gaussianblur, gaussianblur, new Size(3, 3), 0); // 高斯滤波
gaussianblur = new Mat(gaussianblur, rect); // 提取高斯滤波结果
double max_temp = 0;
double min_temp = 0;
Cv2.MinMaxIdx(gaussianblur, out min_temp, out max_temp); // 获取高斯滤波后的最大值
Mat mat = new Mat(this.output_size.Width, this.output_size.Height, MatType.CV_32FC1, maxval/ max_temp);
gaussianblur = gaussianblur.Mul(mat); // 滤波结果乘滤波前后最大值的比值
```

（3）输出结果取对数

```
float[,] process_map = new float[this.output_size.Width, this.output_size.Height];
for (int h = 0; h < this.output_size.Width; h++)
{
    for (int w = 0; w < this.output_size.Height; w++)
    {
        float temp = gaussianblur.At<float>(h, w);
        if (temp < 1e-10)
        {
            temp = (float)1e-10;
        }
      temp = (float) Math.Log(temp);
      process_map[h, w] = temp;
    }
}
```

（4）泰勒展开坐标解码

按照泰勒展开公式，求解定位偏差，用于修正前面定位得到的初略误差。

```
// 基于泰勒展开的坐标解码
int py = index_int[1];
int px = index_int[0];
if ((2 < py) && (py < this.output_size.Width-2) && (2 < px) && (px < this.output_size.Height-2))
{
    // 求导数和偏导数
    float dx = 0.5f * (process_map[py, px + 1] - process_map[py, px - 1]);
    float dy = 0.5f * (process_map[py + 1, px] - process_map[py - 1, px]);
    float dxx = 0.25f * (process_map[py, px + 2] - 2 * process_map[py, px] + process_map[py, px - 2]);
    float dxy = 0.25f * (process_map[py + 1, px + 1] - process_map[py - 1, px + 1]
        - process_map[py + 1, px - 1] + process_map[py - 1, px - 1]);
    float dyy = 0.25f * (process_map[py + 2 * 1, px] - 2 * process_map[py, px] + process_map[py - 2 * 1, px]);
    // 构建相应的倒数矩阵
    Mat derivative = new Mat(2, 2, MatType.CV_32FC1, new float[] { dx, 0, dy, 0 });
    Mat hessian = new Mat(2, 2, MatType.CV_32FC1, new float[] { dxx, dxy, dxy, dyy });
    if (dxx * dyy - dxy * dxy != 0)
    {
        Mat hessianinv = new Mat();
        Cv2.Invert(hessian, hessianinv); // 矩阵求逆
        mat = new Mat(2, 2, MatType.CV_32FC1, -1);
        hessianinv = hessianinv.Mul(mat); // 矩阵取 -
        Mat offset = new Mat();
        Cv2.Multiply(hessianinv, derivative, offset); // 矩阵相乘
        offset = offset.T(); // 矩阵转置
        // 获取定位偏差
        double error_x = offset.At<Vec2d>(0)[0];
        double error_y = offset.At<Vec2d>(0)[1];
        // 修正横纵坐标
        point_meses[p, 0] = px + (float)error_x;
        point_meses[p, 1] = py + (float)error_y;

    }
}
```

图 8 泰勒展开坐标解码过程

（5）获取仿射逆变换矩阵

```
// 获取反向变换矩阵
Point center = new Point(this.image_size.Width / 2, this.image_size.Height / 2); // 变换中心点
Size input_size = new Size(this.image_size.Width, this.image_size.Height); // 输入尺寸
```

```
int rot = 0; // 旋转角度
Size output_size = new Size(this.output_size.Height, this.output_size.Width); // 输出尺寸
Mat trans = get_affine_transform(center, input_size, rot, output_size, true); // 变换矩阵
```

PPTinyPose 模型数据缩放方式为仿射变换方式，因此在进行结果处理时，需要获取仿射变换逆变换矩阵，get_affine_transform 为封装的矩阵变换求解方式，其主要输入参数为变换中心点、输入形状、旋转角度、输出尺寸。

（6）获取变换关系

上一步已经获取逆变换矩阵，由于我们已经求取完输出点图的坐标位置，所以此处我们直接根据变换关系对点的位置进行变换。

```
// 获取变换结果
double scale_x_1 = trans.At<Vec3d>(0)[0];
double scale_x_2 = trans.At<Vec3d>(0)[1];
double scale_x_3 = trans.At<Vec3d>(0)[2];
double scale_y_1 = trans.At<Vec3d>(1)[0];
double scale_y_2 = trans.At<Vec3d>(1)[1];
double scale_y_3 = trans.At<Vec3d>(1)[2];
// 变换预测点的位置
for (int p = 0; p < 17; p++)
{
    point_meses[p, 0] = point_meses[p, 0] * (float)scale_x_1 + point_meses[p, 1] * (float)scale_x_2
    + 1.0f * (float)scale_x_3;
    point_meses[p, 1] = point_meses[p, 0] * (float)scale_y_1 + point_meses[p, 1] * (float)scale_y_2
    + 1.0f * (float)scale_y_3;
}
```

1.4.5 绘制人体姿态

```
void draw_poses(float[,] points, ref Mat image)
{
    // 连接点关系
    int[,] edgs = new int[17, 2] { { 0, 1 }, { 0, 2}, {1, 3}, {2, 4}, {3, 5}, {4, 6}, {5, 7}, {6, 8},
        {7, 9}, {8, 10}, {5, 11}, {6, 12}, {11, 13}, {12, 14},{13, 15 }, {14, 16 }, {11, 12 } };
    // 颜色库
    Scalar[] colors = new Scalar[18] { new Scalar(255, 0, 0), new Scalar(255, 85, 0), new Scalar(255, 170, 0),
        new Scalar(255, 255, 0), new Scalar(170, 255, 0), new Scalar(85, 255, 0), new Scalar(0, 255, 0),
        new Scalar(0, 255, 85), new Scalar(0, 255, 170), new Scalar(0, 255, 255), new Scalar(0, 170, 255),
        new Scalar(0, 85, 255), new Scalar(0, 0, 255), new Scalar(85, 0, 255), new Scalar(170, 0, 255),
        new Scalar(255, 0, 255), new Scalar(255, 0, 170), new Scalar(255, 0, 85) };
    // 绘制阈值
    double visual_thresh = 0.4;
    // 绘制关键点
    for (int p = 0; p < 17; p++)
    {
        if (points[p, 2] < visual_thresh)
        {
            continue;
        }
        Point point = new Point((int)points[p, 0], (int)points[p, 1]);
        Cv2.Circle(image, point, 2, colors[p], -1);
    }
    // 绘制
    for (int p = 0; p < 17; p++)
    {
        if (points[edgs[p, 0], 2] < visual_thresh || points[edgs[p, 1], 2] < visual_thresh)
        {
            continue;
        }

        float[] point_x = new float[] { points[edgs[p, 0], 0], points[edgs[p, 1], 0] };
        float[] point_y = new float[] { points[edgs[p, 0], 1], points[edgs[p, 1], 1] };

        Point center_point = new Point((int)((point_x[0] + point_x[1]) / 2), (int)((point_y[0] + point_y[1]) / 2));
        double length = Math.Sqrt(Math.Pow((double)(point_x[0] - point_x[1]), 2.0) + Math.Pow((double)(point_y[0] - point_y[1]), 2.0));
        int stick_width = 2;
        Size axis = new Size(length / 2, stick_width);
        double angle = (Math.Atan2((double)(point_y[0] - point_y[1]), (double)(point_x[0] - point_x[1]))) * 180 / Math.PI;
        Point[] polygon = Cv2.Ellipse2Poly(center_point, axis, (int)angle, 0, 360, 1);
        Cv2.FillConvexPoly(image, polygon, colors[p]);

    }
}
```

图 9 draw_poses 方法

绘制人体姿态方法主要是将模型预测的人体关键点，按照指定关系连接在一起，构成人体姿态图。edgs 变量表示人体关节点连接关系，由于预测结果关键点是按照指定顺序排列的，所以连接关系是固定的。

此处主要分为两步，分别是绘制人体关键点和绘制人体姿态，由于人体不是每时每刻都能够保持在照片上出现，因此对于未在照片上出现的人体关键点置信度会较低且会不准，所以此处设一个阈值，用于评判预测点的有效性。对于人体姿态绘制采用椭圆型曲线，其最终绘制效果如**错误!未找到引用源。** 所示。

1.4.6 仿射变换矩阵

```
Mat get_affine_transform(Point center, Size input_size, int rot, Size output_size, bool inv = false)
{
    Point2f shift = new Point2f(0.0f, 0.0f);
    // 输入尺寸宽度
    int src_w = input_size.Width;

    // 输出尺寸
    int dst_w = output_size.Width;
    int dst_h = output_size.Height;

    // 旋转角度
    double rot_rad = 3.1715926 * rot / 180.0;
    int pt = (int)(src_w * -0.5);
    double sn = Math.Sin(rot_rad);
    double cs = Math.Cos(rot_rad);

    Point2f src_dir = new Point2f((float)(-1.0 * pt * sn), (float)(pt* cs));
    Point2f dst_dir = new Point2f(0.0f, (float)(dst_w * -0.5));
    Point2f[] src = new Point2f[3];
    src[0] = new Point2f((float)(center.X + input_size.Width * shift.X), (float)(center.Y + input_size.Height * shift.Y));
    src[1] = new Point2f(center.X + src_dir.X + input_size.Width * shift.X, center.Y + src_dir.Y + input_size.Height * shift.Y);
    Point2f direction = src[0] - src[1];
    src[2] = new Point2f(src[1].X - direction.Y, src[1].Y - direction.X);

    Point2f[] dst = new Point2f[3];
    dst[0] = new Point2f((float)(dst_w * 0.5), (float)(dst_h* 0.5));
    dst[1] = new Point2f((float)(dst_w * 0.5 + dst_dir.X), (float)(dst_h * 0.5 + dst_dir.Y));
    direction = dst[0] - dst[1];
    dst[2] = new Point2f(dst[1].X - direction.Y, dst[1].Y - direction.X);

    // 是否为反向
    if (inv) { return Cv2.GetAffineTransform(dst, src);}
    else { return Cv2.GetAffineTransform(src, dst);}
}
```

图 10 get_affine_transform 方法

OpenCV 中封装了 GetAffineTransform(src, dst)方法，用于获取仿射变换矩阵，不过该方法的输入为变换前后三个点的对应坐标，因此封装的 get_affine_transform()方法主要用于计算对应的三个点坐标，其求解原理可以参考仿射变换关系说明。

由于模型的输入也要求对图片数据进行仿射变换，目前直接克隆的 OpenVinoSharp 库没有该方法实现，因此此处需要在 OpenVinoSharp 项目 load_image_input_data()方法增加这种数据处理方式，此处不做详述，具体可以参考源码文件，后续也会对 OpenVinoSharp 进行更新，增加模型部署是用到的一些特殊处理方式。

1.5 PP-TinyPose 人体姿态识别模型在 OpenVINO™ 部署效果

1.5.1 行人识别

（1）初始化 PicoDet 行人识别类

```
// 行人检测模型
string mode_path_det =
@"E:\Text_Model\TinyPose\picodet_v2_s_320_pedestrian\picodet_s_320_lcnet_pedestrian.onnx";
// 设备名称
string device_name = "CPU";
PicoDet pico_det = new PicoDet(mode_path_det, device_name);
```

首先初始化行人识别类，将本地模型读取到内存中，并将模型加载到指定设备中，由于此处所用电脑未配备英特尔 GPU，所以使用 CPU 进行模型处理。

（2）设置输入输出形状

```
Size size_det = new Size(320, 320);
pico_det.set_shape(size_det, 2125);
```

根据我们使用的模型，设置模型的输入输出形状。

（3）行人预测

```
// 测试图片
string image_path = @"E:\Git_space\基于 Csharp 和 OpenVINO 部署 PP-TinyPose\image\demo_3.jpg";
Mat image = Cv2.ImRead(image_path);
List<Rect> result_rect = pico_det.predict(image);
```

在进行模型推理时，使用 OpenCvSharp 读取图像，然后带入预测，最终获取行人预测框。最后将行人预测框绘制到图片上，如图 11 所示。

图 11 行人位置预测结果

1.5.2 人体姿态识别

（1）初始化 P 人体姿势识别 PPTinyPose 类

```
// 关键点检测模型
// onnx 格式
string mode_path_pose = @"E:\Text_Model\TinyPose\tinypose_128_96\tinypose_128_96.onnx";
// 设备名称
string device_name = "CPU";
PPTinyPose tiny_pose = new PPTinyPose(mode_path_pose, device_name);
```

首先初始化人体姿势识别 PPTinyPose 类，将本地模型读取到内存中，并加载到设备上。

（2）设置输入输出形状

```
Size size_pose = new Size(128, 96);
tiny_pose.set_shape(size_pose);
```

PP-TinyPose 模型输入与输出有对应关系，因此只需要设置输入尺寸

（3）行人预测

```
// 测试图片
```

```
string image_path = @"E:\Git_space\基于 Csharp 和 OpenVINO 部署 PP-TinyPose\image\demo_3.jpg";
Mat image = Cv2.ImRead(image_path);
Mat result_image = tiny_pose.predict(image);
```

在进行模型推理时，使用 OpenCvSharp 读取图像，然后带入预测，最终获取人体姿态结果，如图 12 所示。

图 12 人体姿态绘制效果图

1.5.3 总体预测效果

人体姿态识别主要包括行人区域识别和关键点识别两步，将上面两步同时实现，就可以实现完整的人体姿态检测，检测效果如图 12 所示。

对于第一幅图识别效果可以看出，第一幅图已经将 17 个关键点都识别了出来，其中对人体四肢和关节识别较为准确，对于人脸上的关键点识别有些偏差，主要是人脸上面关键点较为集中，且篇幅较小，所以识别较困难；对于第二幅图，右手臂与身体重叠，因此检测出来的的置信度较低，此处我们设置了阈值，因此在绘图时被忽略掉了。

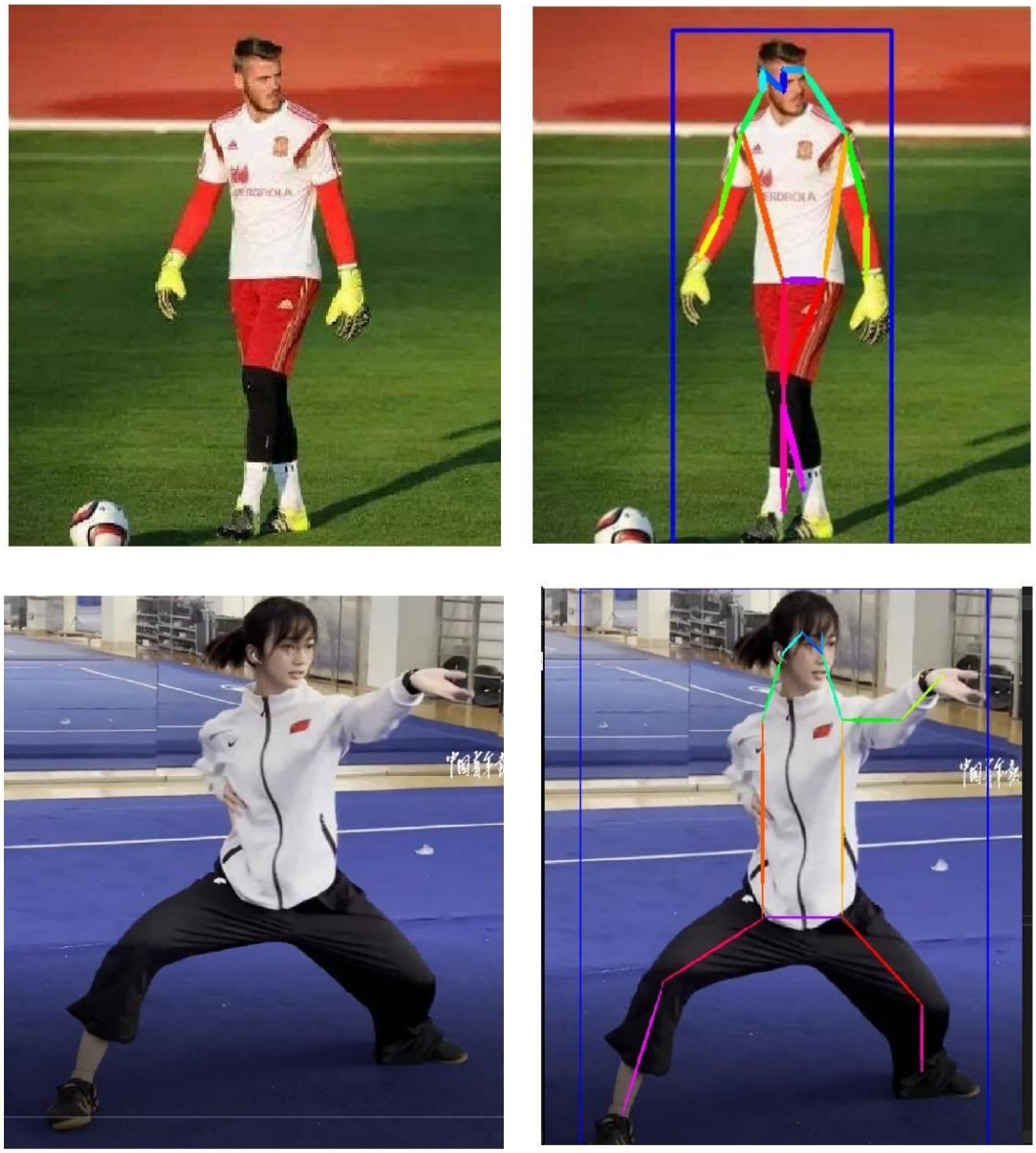

图 13 人体姿态识别输出效果

1.5.4 时间测试

由于 PP-PicoDet 模型无法转为 IR 模型格式，因此此处只测试 ONNX 模型运行时间。通过之前测试，C#平台与 C++平台运行时间没有太大的差距，所以此处只测试 C#平台，并与 Paddle Inference 进行对比，获得以下结果，如表 4 所示。

表 4 PP-PicoDet 与 PP-TinyPose 模型运行时间(ms)

平台	模型名称	PP-PicoDet 320×320				PP-TinyPose 256×192			
	模型格式	模型读取	加载数据	模型推理	结果处理	模型读取	加载数据	模型推理	结果处理
OpenVINO C#	ONNX	295	17	7	4	763	8	105775	15

Paddle Inference	Paddle	10	127	1	3.5	65	4

注： 模型读取：读取本地模型，加载到设备，创建推理通道；
加载数据：将待推理数据进行处理并加载到模型输入节点；
模型推理：模型执行推理运算；
结果处理：在模型输出节点读取输出数据，并转化为我们所需要的结果数据。

通过表 4 可以看出，在 C#中，利用 OpenVINO 部署 PP-PicoDet 模型推理速度比使用 Paddle Inference 快了很多，目前 PP-TinyPose 模型在 OpenVINO 部署还存在问题，模型推理时间较慢。

1.6 总结

在该项目中，基于 C#和 OpenVINO 部署 PP-PicoDet 和 PP-TinyPose 模型，实现了行人识别以及人体姿态识别，打通了 C#、OpenVINO、PP-PicoDet 和 PP-TinyPose 模型之间的障碍。在该项目中，除了利用动态链接库在 C#中实现调用 OpenVINO 推理套件这一个难点之外，还有以下几个难点：

第一、PP-PicoDet 模型处理：目前我们直接在飞桨网站下载的部署模型，是不能够在 OpenVINO 中部署使用，主要原因 PP-PicoDet 的后处理，加入了模型输入与实际图片的比例关系，增加了后处理这一过程，所以使用时会出现错误。此处采用去掉模型后处理这一过程，将后处理放在代码中实现。

第二、PP-TinyPose 输入与输出结果处理问题：利用 OpenVINO 部署 PP-TinyPose 模型最大的障碍就是模型的输入处理以及输出处理，因为该模型的训练采用了热力图修正，因此我们也需要按照该方式处理模型的输入与输出，并且模型的输出热度图解码坐标点处理起来较为复杂，在进行处理时要细心实现。

参考文献

[1] Zhang F, Zhu X, Dai H, et al. Distribution-aware coordinate representation for human pose estimation[C]//Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2020: 7093-7102.

[2] @misc{ppdet2019, title={PaddleDetection, Object detection and instance segmentation toolkit based on PaddlePaddle.}, author={PaddlePaddle Authors}, howpublished = {\url{https://github.com/PaddlePaddle/PaddleDetection}}, year={2019}}