TensorRT高级篇

CNN分类器

学习实现一个完整的CNN分类器案例

1. 模型里面没有softmax操作，在这里采用包裹一层加上softmax节点后再导出模型，这样使得后处理得到的直接就是概率值，避免后处理上再做softmax
2. 在c++代码中，则充分采用指针偏移的方式，提升cpu上预处理的效率
3. 对于bgr->rgb也避免使用cvtColor实现，而是简单的改变赋值时的索引，提升效率

YoloV5目标检测

学习yolov5如何导出模型并利用起来

1. 修改export_onnx时的导出参数，使得动态维度指定为batch，去掉width和height的指定

2. 导出时，对yolo.py进行修改，是的后处理能够简化，并将anchor合并到onnx中

3. 预处理部分采用warpaffine，描述对图像的平移和缩放

UNet场景分割

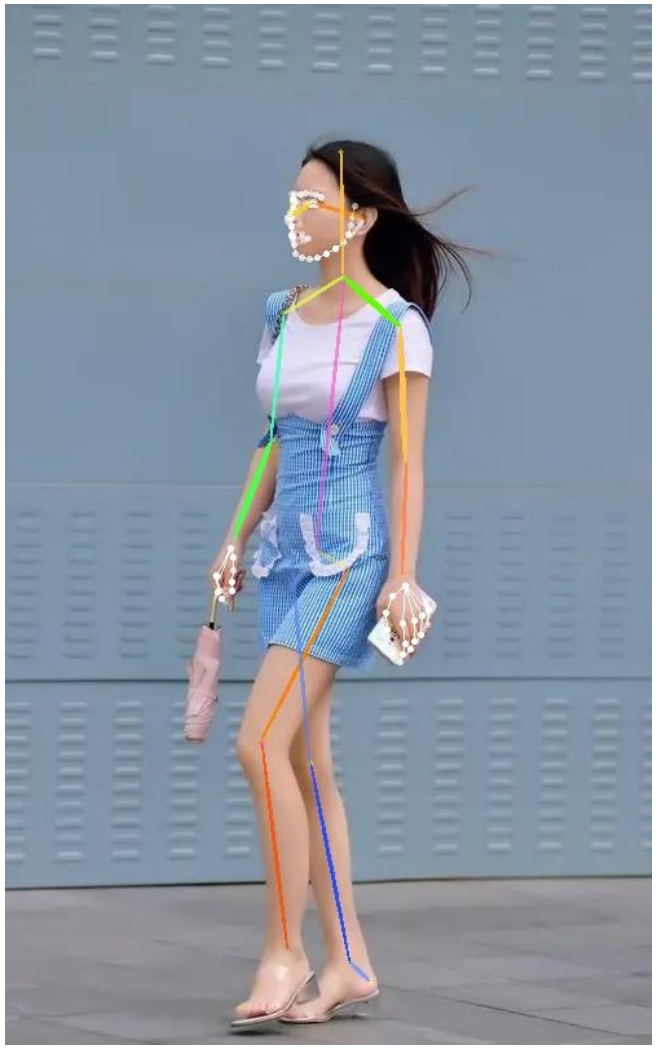
学习如何处理场景分割的案例

1. 场景分割的预处理后处理逻辑

2. 预处理采用warpaffine时，后处理可以使用逆变换得到mask

AlphaPose姿态检测

1. 由于后处理比较复杂，放到cuda去做比较麻烦，这里把计算放到了onnx中



Mmdetection案例

1. 这节课的主要目的是：通过调试分析mmdet代码，把yolox模型导出，并在tensorrt上推理得到结果

2. 其中涉及到调试和分析的方法技巧

onnxrunntime案例

1. onnx是microsoft开发的一个中间格式，而onnxruntime简称ort是microsoft为onnx开发的推理引擎
2. 允许使用onnx作为输入进行直接推理得到结果
3. onnxruntime有python/c++接口，支持CPU、CUDA、tensorRT等不同后端，实际CPU上比较常用
4. ort甚至在未来还提供了训练功能
5. 学习使用onnxruntime推理YoloV5并拿到结果

openvino案例

1. openvino是Intel开发的基于intel计算设备的推理引擎
2. 他可以利用CPU发挥最好的性能，还能使用到新款CPU提供的NN运算能力
3. 还可以利用外置的计算棒实现更好的推理性能

多线程知识

1. 这里的多线程主要指算法部署时所涉及的多线程内容，对于其他多线程知识需要自行补充
2. 常用组件有thread、mutex、promise、future、condition_variable
3. 启动线程，thread，以及join、joinable、detach、类函数启动为线程
4. 生产者消费者模式
5. 队列溢出的问题，生产太快，消费太慢。如何实现溢出限制
6. 生产者如何拿到消费反馈
7. RAII+接口模式的生产者消费者封装，以及多batch的体现

tensorRT封装

1. 对tensorRT的封装，更像是对推理引擎的封装
2. 封装的意义在于对技术的标准化、工具化，能够是的使用时更加便利，效率更高，定制更多的默认行为
3. 封装推理引擎的思想，还可以应用到更多其他地方。嵌入式、等等。由于大多推理引擎提供的默认方式不够友好，对其进行包装，能够很好的使得自己的代码具有复用性，一套代码多处用
4. 还可以实现，同样的封装，通过简单的配置，切换不同的推理后端。这都取决于需求
5. 我们的唯一目的就是让工作更简单，让代码复用性更强，让技术可以沉淀

tensorRT封装

1. 对builder进行封装，使得编译的接口足够简单
2. 对memory进行封装，使得内存分配复制自动管理，避免手动管理的繁琐
3. 对tensor进行封装，张量是CNN中常见的基本单元，尤其是计算偏移量的工作需要封装，其次是内存的复制、分配需要引用memory进行包装，避免使用时面对指针不好管控
4. 对infer进行封装，有了基本组件，可以拼接一个完整的推理器，而且该推理器的思想可以应用到很多框架作为底层，并不只限制于tensorRT，还可以是rknn、openvino等
5. 使用封装好的组件，配合生产者消费者，实现一个完整的yolov5推理

onnx解析器更新

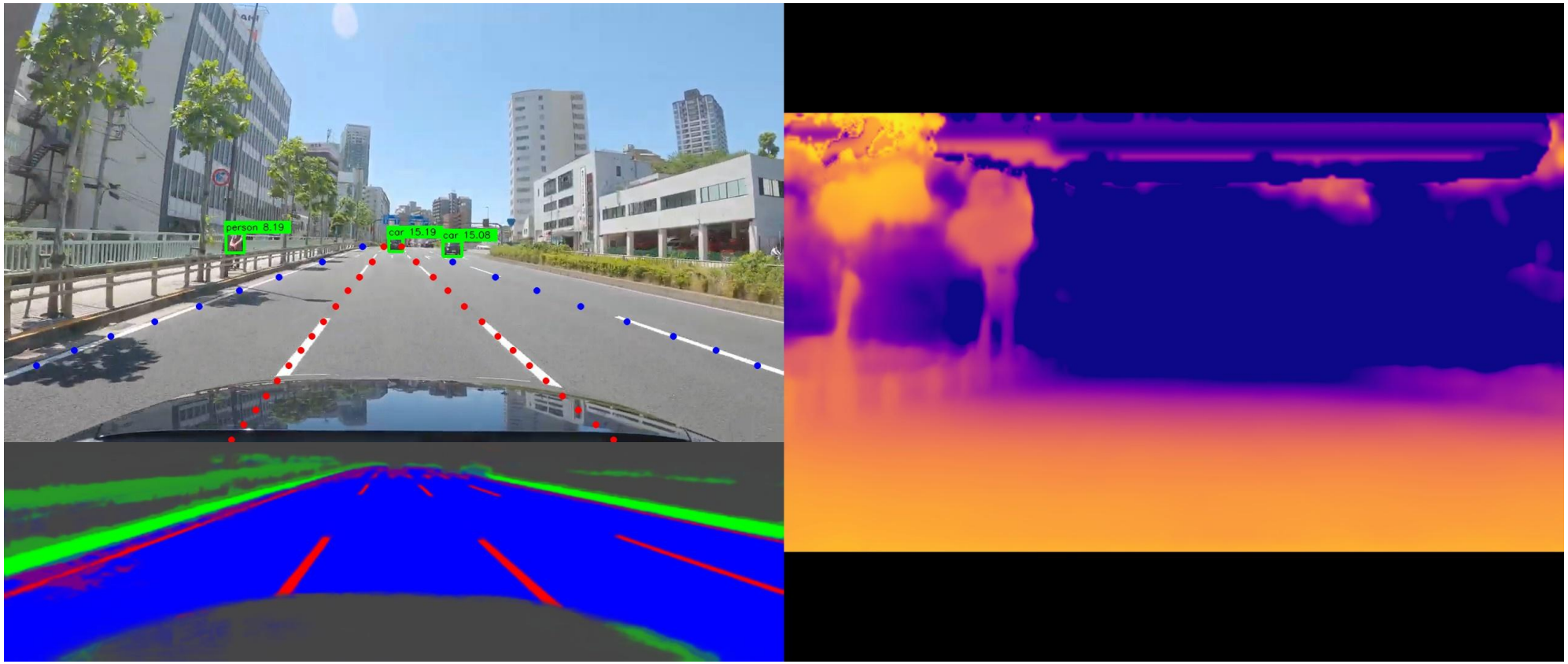
1. 这里以实际操作来更新一下onnx解析器，从网上下载并使用，使得自己可以根据需求来更新他
2. onnx解析器地址：https://github.com/onnx/onnx-tensorrt

rknn案例

1. rknn是瑞芯微为其NPU设计的nn推理引擎
2. NPU具有很好的int8推理性能，边缘端常选择int8进行推理
3. 这里我们仅仅是了解rknn的代码，他与咱们在tensorRT上推理的差别

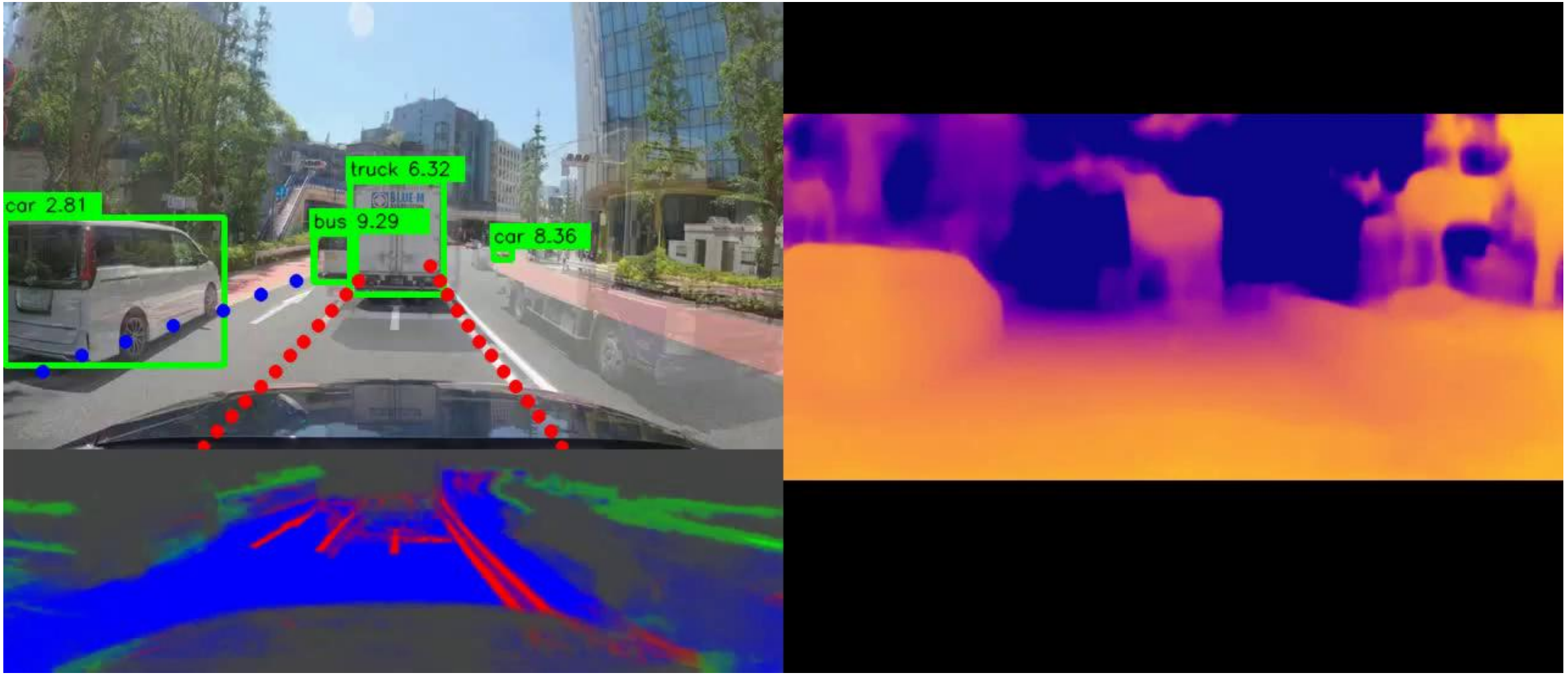
自动驾驶场景的模型案例

1. 这个案例中，存在4个模型，分别是：车辆检测YoloX、车道线检测、道路分割、深度估计
2. 学习把该案例的模型跑起来，对不同任务进行了解



https://github.com/iwatake2222/self-driving-ish_computer_vision_system

自动驾驶场景-效果展示



道路分割分析

找到道路分割的onnx，分析其onnx的大致使用逻辑，然后写出最简洁版本的predict.py

1. 打开道路分割的onnx，查看其输入与输出
2. 查看代码，找到onnx的预处理，分析得到预处理的逻辑
3. 针对获得的信息，编写predict.py，尝试写出来

深度估计分析

找到onnx，分析其onnx的大致使用逻辑，然后写出最简洁版本的predict.py

1. 打开深度估计的onnx，查看其输入与输出
2. 查看代码，找到onnx的预处理，分析得到预处理的逻辑
3. 针对获得的信息，编写predict.py，尝试写出来

车道线检测分析

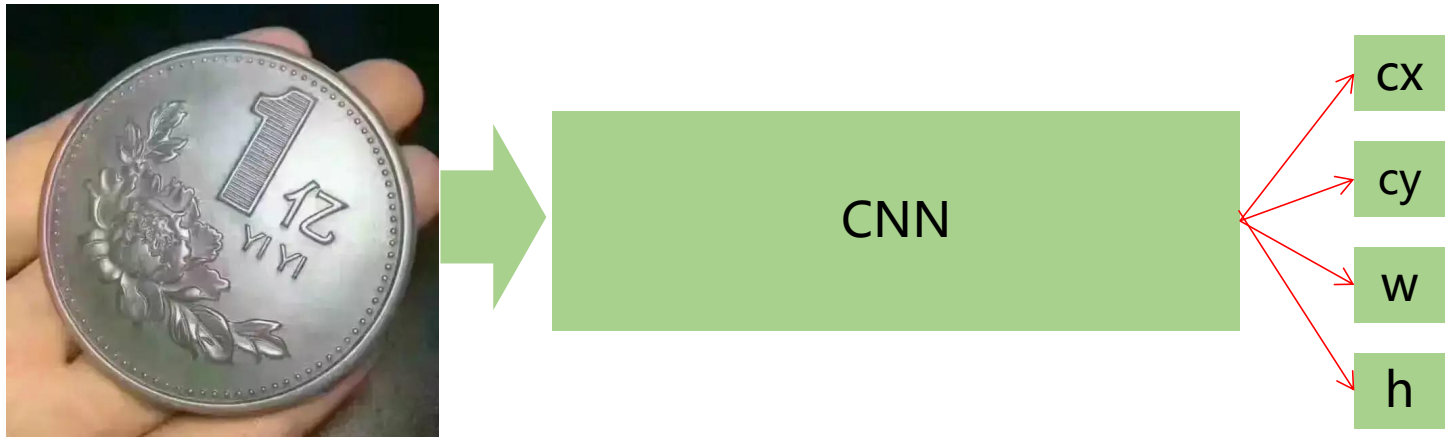
找到onnx，分析其onnx的大致使用逻辑，然后写出最简洁版本的predict.py

1. 打开深度估计的onnx，查看其输入与输出
2. 查看代码，找到onnx的预处理，分析得到预处理的逻辑
3. 针对获得的信息，编写predict.py，尝试写出来
4. 在这个案例中，由于后处理过于复杂，因此考虑合并到onnx中。使得模型尽量的简单

车道线检测分析

对于常规的框回归任务，例如要求回归图像中硬币所在位置，cx cy w h

其通常直接输出4个标量值进行回归

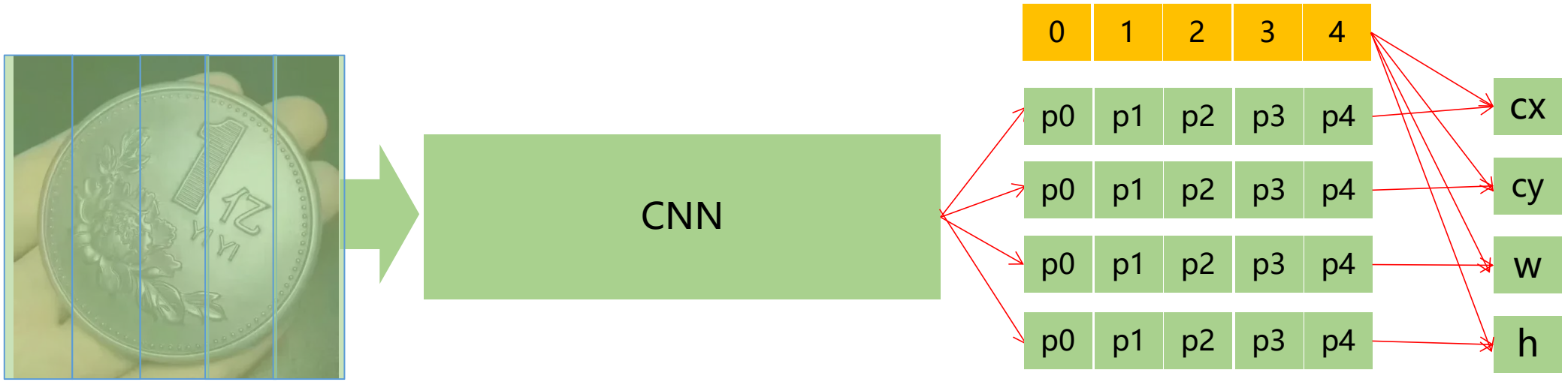


车道线检测分析

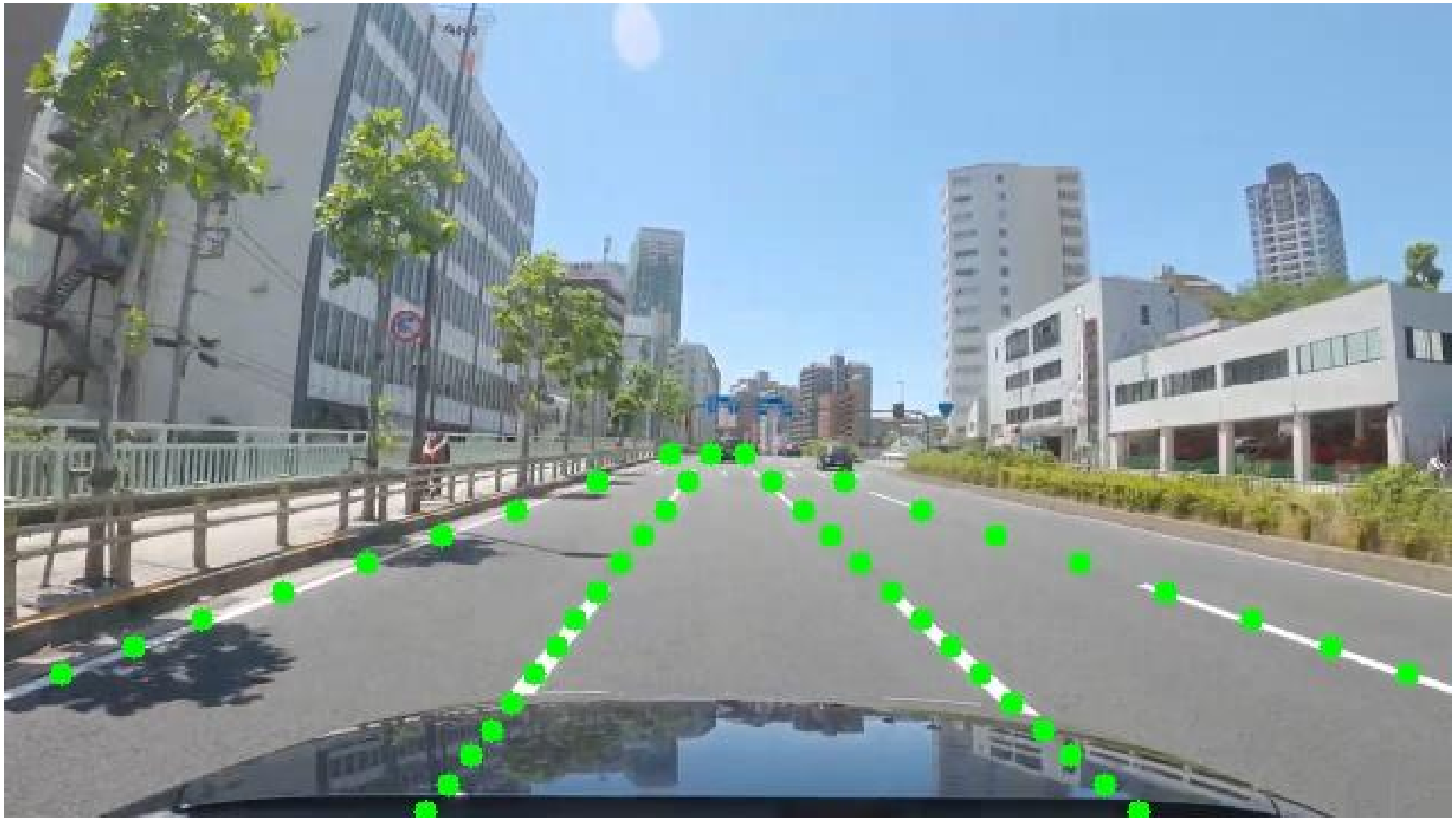
目前最新的，大家更倾向于使用位置概率点乘其位置作为输出值，属于加权和

即，将回归的坐标以n个位置概率进行表示，例如对于cx的回归，表示为5个概率，可以认为对图像划分为5块

然后cx更有可能落到哪一块上进行表述。例如落在图像中心上时，其中心概率最高。有一种attention的味道



车道线检测分析



模型的调试技巧，debug方法

调试法则：

1. 善用python工作流，联合python/cpp一起进行问题调试
2. 去掉前后处理情况下，确保onnx与pytorch结果一致，排除所有因素。这一点通常是能够保证的。例如都输入全为5的张量，必须使得输出之间差距小于1e-4，确保中间没有例外情况发生
3. 预处理一般很难保证完全一样，考虑把python的预处理结果储存文件，c++加载后推理，得到的结果应该差异小于1e-4
4. 考虑把python模型推理后的结果储存为文件，先用numpy写一遍后处理。然后用c++复现
5. 如果出现bug，应该把tensor从c++中储存文件后，放到python上调试查看。避免在c++中debug

不要急着写C++，多用python调试好

模型的调试技巧，debug方法

实现一个模型的流程：

1. 先把代码跑通predict，单张图作为输入。屏蔽一切与该目标不符的东西，可以修改删除任意多余的东西
2. 自行写一个新的python程序，简化predict的流程，掌握predict所需要的最小依赖和最少代码
3. 如果第二步比较困哪，则可以考虑，直接在pred = model(x)这个步骤上研究，例如直接在此处写 torch.onnx.export(model, (pred,) ...)。或者直接把pred的结果储存下来研究，等等
4. 把前处理、后处理分析出来并实现一个最简化版本
5. 利用简化版本进行debug、理解分析。然后考虑预处理后处理的合理安排，例如是否可以把部分后处理放到onnx中
6. 导出onnx，在c++上先复现预处理部分，是的其结果接近（因为大多时候不能得到一样的结果）
7. 把python上的pred结果储存后，使用c++读取并复现所需要的后处理部分。确保结果正确
8. 把前后处理与onnx对接起来，形成完整的推理

Python扩展模块

这里我们学习如何为python写c++的扩展模块，使用pybind11

1. 这里实现了对yolov5的推理封装

2. 封装了一个c++类对应到python中

3. python的底层大都使用c++进行封装，可以利用c++的计算性能和python的便利性

谢谢!