## Bevformer

### 1.1 DETR

1. 二分图匹配确定有效的预测框：匈牙利算法（N>M如何匹配）
2. 问题：正向推理怎么办？

DETR（DEtection TRansformer）在前向推理时，会生成固定数量的预测框。通常情况下，默认是100个。这100个预测框中，可能有一部分会被分类为“无目标”（no object），而其余的预测框会根据模型的预测对应到具体的类别和位置信息。

这是因为DETR采用了一种基于查询的机制，它会对输入图像生成固定数量的查询，然后通过Transformer解码器生成对应的预测框。即使图片中物体的实际数量少于100个，模型也会输出100个预测框，其中一些可能会被标记为无效。

如果你需要调整这个数量，可以通过修改DETR模型的num\_queries参数来改变生成的预测框数量。



1. 

Transfomer训练和测试环节输入有很大不同！

Transfomer的decoder的输入输出是错位的。

输入： S I want a beer

输出： I want a beer .



### Deformable-Detr

可变形注意力：

可变形注意力机制结合了稀疏注意力的思想和动态偏移的创新。它通过生成动态偏移来调整采样位置，从而在稀疏注意力的基础上实现更加灵活的注意力分配。具体来说，可变形注意力机制首先通过稀疏选择减少计算范围，然后在这些关键位置上应用动态偏移，以获取更加准确的注意力权重。

PS：注意！

encoder中reference\_points是直接采样（函数get\_reference\_points）获得，相当于没有进行稀疏选择,但是decoder中reference\_points是学习获得，所以这里进行了稀疏选择

重点在于只和采样偏移的点进行了多头注意力运算，所以重点在于选择采样点





问题: query feature怎么获得？是一个序列还是一群序列？图像中实际上只显示了一个

Reference point每个对应一个query

这个是在单尺度（也就是一个特征图），多尺度如下







Nlevels是特征图scale的个数（原论文中4个）。Locations以下靠CUDA代码



上面的运算流程是用CUDA代码实现的，改不了

深入理解full attention的公式：



token,multi-head(其实是直接把feature拆分)

参考：[Deformable DETR ｜ 2、backbone 、MultiHeadAttention 公式讲解\_哔哩哔哩\_bilibili](https://www.bilibili.com/video/BV1vr4y1o73B?spm_id_from=333.788.videopod.sections&vd_source=d454dd668337bd1b6955178189b20c20)



上面的公式其实是指定了某一个token作为query时的计算结果，为6x1（实际上还要重复这个操作然后计算每个token作为query的）

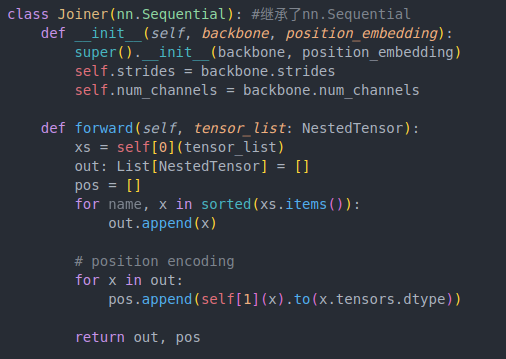


应为代码配置不好，现在BEV Former先放放

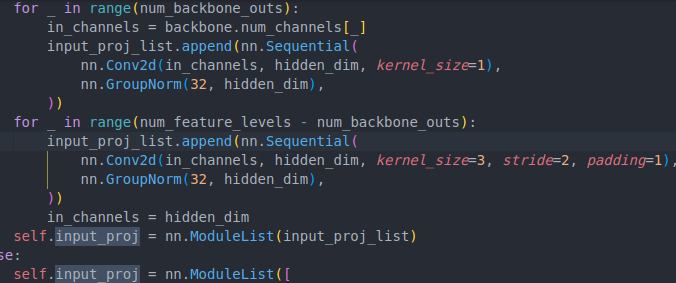
Bev Former原论文在A100上配置，4090配置挺麻烦的

deformabel-detr运行记录：

**backbone部分**

bacbone采用resnet101,position\_embbeding采用行+列编码

backbone的out采用orderedict，应为需要输出4中尺寸不同的特征图（通过out[0,1,2,3]来保存，不能放在batch中，应为特征图大小不同）

input\_proj采用索引方式：nn.ModuleList.append(nn.sequential)

encoder的输入详解:

|  |  |  |
| --- | --- | --- |
| 名称 | 含义 | 格式 |
| src\_flatten | 变为token的图片 | torch.Size([2, 12320, 256]) |
| spatial\_shapes | 特征图的大小 | torch.Size([4, 2]) |
| level\_start\_index | 每个特征图开始的索引 | torch.Size([4]) |
| valid\_ratios | 每个图片有效数据（填充） | torch.Size([2, 4, 2]) |
| lvl\_pos\_embed\_flatten | 位置编码 | torch.Size([2, 12320, 256]) |
| mask\_flatten | 掩码 | torch.Size([2, 12320]) |

说明：batch =2,d\_model=256,level=4

|  |
| --- |
| **get\_reference\_points函数详解：**  在 PyTorch 中，使用 None 可以在张量的特定位置添加一个新的维度。这种操作也被称为“维度扩展”或“升维”。在你的代码中，None 被用来扩展张量的维度，以便进行广播操作。（为啥不直接用unsqueeze） |

接下来就是使用MS\_deform\_attn函数的使用方法：

|  |  |  |
| --- | --- | --- |
| 声明参数 | 含义 | 说明 |
| d\_model | token的维度 | 和输入一致，如256 |
| n\_levels | 特征图数量 | 和输入一致，如4 |
| n\_heads | 感知头数量 | 如8 |
| n\_points | 方向点数量 | 如4 |
| 输入参数 | 含义 | 说明 |
| query | 图像组成的token+pos\_em | torch.Size([2, 12320, 256]) |
| reference\_points | 图像采样的参考点 | \*torch.Size([2, 12320, 4, 2]) |
| input\_flatten | 图像组成的token | torch.Size([2, 12320, 256]) |
| input\_spatial\_shapes | 每个特征图的尺度 | torch.Size([4, 2]) |
| input\_level\_start\_index | 每个特征图起始索引 | torch.Size([4]) |
| input\_padding\_mask | 图像的掩码（部分pad） | torch.Size([2, 12320]) |
| 输出 | 含义 | 说明 |
| output | 最终输出 | torch.Size([2, 12320, 256]) |

说明：特征图尺寸为4

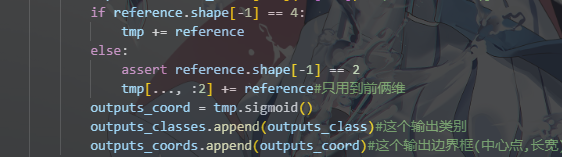
\*表示encoder和decoder是进行的refernece\_points生成策略有所不同

Encoder使用get\_reference\_points产生

下面给出decoder中cross attention调用的参数

|  |  |  |
| --- | --- | --- |
| 输入参数 | 含义 | 说明 |
| tgt | 查询向量query | torch.Size([2, 300, 256]) |
| reference\_points | 稀疏采样点 | \*torch.Size([2, 300, 4, 2]) |
| src | 图像特征的token（value） | torch.Size([2, 12320, 256]) |
| src\_spatial\_shapes | 每个特征图的尺度 | torch.Size([4, 2]) |
| level\_start\_index | 每个特征图起始索引 | torch.Size([4]) |
| src\_padding\_mask | 图像的掩码（部分pad） | torch.Size([2, 12320]) |
| 输出 | 含义 | 说明 |
| tgt2 | 最终输出 | torch.Size([2, 300, 256]) |

可以看到，最终的输出大小tgt2.shape==tgt.shape，表示查询的结果



最终边界框在outputs\_coord中，不包含置信度（不需要NMS）

最后详细学习一下如何计算loss

|  |
| --- |
| **Loss的计算方法**  这里还涉及到匈牙利损失和二分匹配算法  第一步：二分图匹配  表示ground truth集合(用控物体填充)，表示预测的集合 |
| 没有置信度分数，所以也没办法进行NMS，只能对每一个框进行遍历，取最小的损失值预测框作为真实匹配框  <https://blog.csdn.net/weixin_47936614/article/details/141398813>  计算类别损失  目标检测中会采用Focal Loss,专门用于处理类别不平衡问题的损失函数    目标检测框的损失是mxn计算的，所以最后结果都是返回mxn的矩阵  分别计算L1损失和giou损失    最后匈牙利算法进行二分图匹配（直接调用linear\_sum\_assignment函数）   * i 是预测边界框的索引。 * j 是目标边界框的索引。   第二步：计算匈牙利损失  定义的损失类似于常见目标检测器的损失    代码中实现实际上由4种loss组成  具体4种loss计算方法目前先不展开学习 |

### 1.3 BevFusion



Lidar的处理不特别，Camera的view transform部分是重点

## 3D Object Detection

选取几个开源的代码跑通，分别有：camera,lidar两个方案

目前先运行Mono3D

区别于语义分割的点

### 2.1 Detr3D

环境配置问题：mmcv,mmdet,mmseg，mmdet3d均可以安装在4090 CUDA11.8上

但是不能参照bevformer的md，会报错，自己去看对应的官网

问题：2.0.0以上的版本会有很多报错，移除了很多文件

Detr3d的代码是在3090上运行的，安装的库最好是老版本

4090安装的版本如下：



为了运行较老代码，mmcv版本为：

pip install mmcv-full==1.4.0 （安装不了full，但是可以安装mmcv 1.4.0）

pip install mmsegmentation==0.14.1

pip install mmdet == 2.14.0

但是mmdet3d必须从源代码安装，版本为0.17.1



[出现错误“subprocess.CalledProcessError: Command ‘[‘ninja‘, ‘-v‘]‘ returned non-zero exit status 1”解决方法\_subprocess.calledprocesserror: command '['ninja', -CSDN博客](https://blog.csdn.net/fq9200/article/details/125362088)

感觉还是pytorch版本太高，没有其他办法（CUDA 11.8只能安装2.0.0的），已经老实

### BevFusion

BevFusion可以尝试配置

注意：openMPI需要管理员权限sudo



已经老实，BevFusion需要MMCV 1.4.0，最好是3090或者A100，4090根本不行

看样子最好得自己配置一个docker镜像

尽力了，安了CUDA 11.7都没用

### CLOCS

结果级别的融合方案，非常有意思

先看NMS

这些都先放着，这里的电脑跑不了

One-stage和two-stage



2.。

### 语义分割

语义分割目前只有一篇cam+lidar的融合方法，其他基本都是基于Lidar

### SMOKE

以这篇作为mono3D的学习入门方法

中间提到了RPN （region proposal network）——这个方法首次来自于Faster-RCNN(详情请见下面)

首先梳理相机内参和外参的作用





内参反映了像素坐标系和相机坐标系的坐标变换关系

外参反映了相机坐标系和世界坐标系的坐标变换关系

另外注意kitti数据集的理解：

[KITTI 3D目标检测数据集解析（完整版）\_kitti数据集-CSDN博客](https://blog.csdn.net/qq_16137569/article/details/118873033)

3D框标注

|  |  |  |  |
| --- | --- | --- | --- |
| 字段 | 字段长度 | 单位 | 含义 |
| Type | 1 | - | 目标类型 |
| Truncated | 1 | - | 目标截断程度：0~1之间的浮点数，表示目标距离图像边界的程度 |
| Occluded | 1 | - | 目标遮挡程度：0~3之间的整数，0：完全可见 1：部分遮挡 2：大部分遮挡 3：未知 |
| Alpha | 1 | 弧度 | 目标观测角： |
| Bbox | 4 | 像素 | 目标2D检测框位置：左上顶点和右下顶点的像素坐标 |
| Dimensions | 3 | 米 | 3D目标尺寸：高、宽、长 |
| Location | 3 | 米 | 目标3D框底面中心坐标：( x , y , z ) (x, y,z)(x,y,z)，相机坐标系 |
| Rotation\_y | 1 | 弧度 | 目标朝向角： |

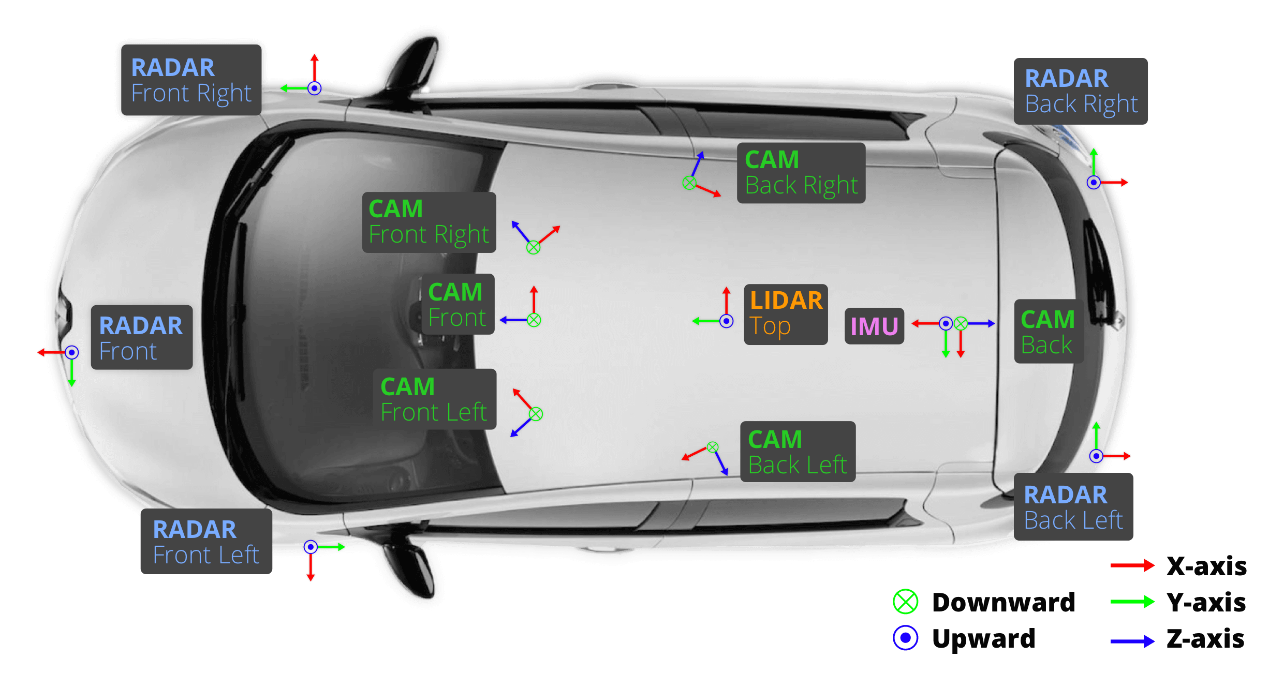
注意：Alpha和Rotation\_y可以相互转化，关系如下：（这里是相对于数据车的相机作为参考）



偏航角的计算方法：

### 2.6 Metrics

首先学习nuScenes tracking task和指标的含义



* [目标检测](https://so.csdn.net/so/search?q=%E7%9B%AE%E6%A0%87%E6%A3%80%E6%B5%8B&spm=1001.2101.3001.7020)(object Detection)：是指在图像或视频帧中识别并定位一个或多个感兴趣的目标对象的过程 。
* 目标跟踪(object Tracking)：是指在**视频帧序列中**连续地监测和定位一个或多个**目标对象的位置**的过程。

具体区别：

1.输入数据不同：目标检测处理单一图像，目标跟踪处理视频序列（多帧图像）

2.目标不同：目标检测关注于定位物体，目标跟踪关注于物体的变化（还有ID）

数据格式：

|  |
| --- |
| └── NUSCENES\_TRAINVAL\_DATASET\_ROOT  ├── samples <-- key frames  ├── sweeps <-- frames without annotation  ├── maps <-- unused  ├── lidarseg <-- lidar semantic labels  └── v1.0-trainval <-- metadata and annotations  └── NUSCENES\_TEST\_DATASET\_ROOT  ├── samples <-- key frames  ├── sweeps <-- frames without annotation  ├── maps <-- unused  ├── lidarseg <-- lidar semantic labels  └── v1.0-test <-- metadata  说明：  1.samples所有传感器的数据关键帧（包括cam+lidar+radar）  2.sweeps所有传感器的数据插帧（没有标注，但是会用来作为补充数据帧）  3.maps场景的地图（一般用不上）  4.v1.0-trainval所有数据标注文件      说白了，sample代表一个关键帧，sample\_data代表这个关键帧所有传感器的数据 |

学习模型：

BEVFusion,TransFusion,SImpleTrack,Tracker,Sparse4D,BEVTrack

表一 多模态

|  |  |  |
| --- | --- | --- |
| 名称 | 数据 | 结构 |
| BEVFusion | cam+lidar |  |
| TransFusion | Cam+lidar |  |
| MCTack | Cam+lidar |  |
| MV2DFusion | Cam+lidar |  |

表二 视觉

|  |  |  |
| --- | --- | --- |
| 名称 | 数据 | 结构 |
| Sparse4D | cam |  |
| \*SparseAD | cam |  |

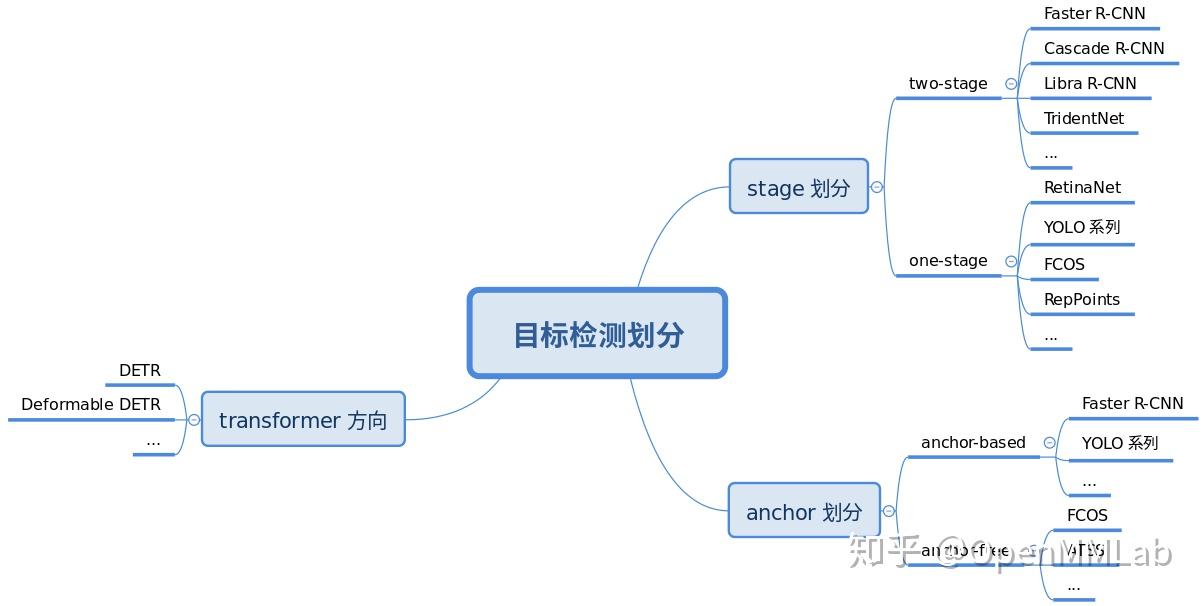
表三 激光雷达

|  |  |  |
| --- | --- | --- |
| 名称 | 数据 | 结构 |
| SimpleTrack | lidar |  |
| BEVTrack | lidar |  |
| LEGO | lidar |  |

|  |
| --- |
| **总结计划**  1.首先研究基于检测的方法，包括但不限于：sparse4d,transfusion  2.然后研究跟踪的方法，总结检测和跟踪的区别  3.最后探索多传感器融合的跟踪方法，并扩展到openPCD框架 |

### MM-openlab框架学习

|  |
| --- |
| MMCV  **配置**  学习 from mmcv import Config的用法，包括：  配置config.py，继承关系  **注册器**  学习from mmcv import Registry的用法，包括：  如何创建注册器的子注册器，例如：  所有下游代码库中所有MODELS注册器都是MMCV MODELS注册器的子注册器  替代原本pytorch注册神经网络的方法  不同Registry的实例的区别是：其实都是MMCV\_MODELS的实例  严格来说使用方法没有区别  **执行器**  学习runner类的用法  **卷积神经网络**  BaseModule 继承自 torch.nn.Module, 它们之间唯一的不同是 BaseModule 实现了 init\_weight  Sequential 继承自 BaseModule 和 torch.nn.Sequential  ModuleList 继承自 BaseModule 和 torch.nn.ModuleList  Init\_cfg是用来初始化模型的  **Hook**  钩子编程（hooking），也称作“挂钩”，是计算机程序设计术语，指通过拦截软件模块间的函数调用、消息传递、事件传递来修改或扩展操作系统、应用程序或其他软件组件的行为的各种技术。处理被拦截的函数调用、事件、消息的代码，被称为钩子（hook） |



|  |
| --- |
| **MMDetection**    训练过程：   1. 任何一个 batch 的图片先输入到 backbone 中进行特征提取，典型的骨干网络是 ResNet 2. 输出的单尺度或者多尺度特征图输入到 neck 模块中进行特征融合或者增强，典型的 neck 是 FPN 3. 上述多尺度特征最终输入到 head 部分，一般都会包括分类和回归分支输出 4. 在整个网络构建阶段都可以引入一些即插即用增强算子来增加提取提取能力，典型的例如 SPP、DCN 等等 5. 目标检测 head 输出一般是特征图，对于分类任务存在严重的正负样本不平衡，可以通过正负样本属性分配和采样控制 6. 为了方便收敛和平衡多分支，一般都会对 gt bbox 进行编码 7. 最后一步是计算分类和回归 loss，进行训练 8. 在训练过程中也包括非常多的 trick，例如优化器选择等，参数调节也非常关键   MMDetection中实现的部分  1.backbone:mmdet/models/backbones  *# 骨架的预训练权重路径*  pretrained**=**'torchvision://resnet50',  backbone**=**dict(  type**=**'ResNet', *# 骨架类名，后面的参数都是该类的初始化参数*  depth**=**50,  num\_stages**=**4,  out\_indices**=**(0, 1, 2, 3),  frozen\_stages**=**1,  norm\_cfg**=**dict(type**=**'BN', requires\_grad**=**True),  norm\_eval**=**True,  style**=**'pytorch'),  2.neck: mmdet/models/necks  neck**=**dict(  type**=**'FPN',  in\_channels**=**[256, 512, 1024, 2048], *# 骨架多尺度特征图输出通道*  out\_channels**=**256, *# 增强后通道输出*  num\_outs**=**5), *# 输出num\_outs个多尺度特征图*   1. 训练测试算法流程：     在 MMDetection 中 DataLoader 输出的内容不是 pytorch 能处理的标准格式    前面说个，训练和验证的时候实际上调用了 model 内部的 train\_step 和 val\_step 函数 |

|  |
| --- |
| **MMDetection3d** |

|  |
| --- |
| **3D Multiple Object Tracking ( 3D MOT)**  Nuscenes数据集中包含tracking相关的数据 |

### 2.8 Sparse4D v1

|  |
| --- |
| **环境配置问题：**  1.需要利用mmdet3D库处理数据集  数据集处理的命令  python tools/create\_data.py nuscenes --root-path /data/wangzhaohui/github/Sparse4D/data/nuscenes --out-dir /data/wangzhaohui/github/Sparse4D/data/nuscenes\_cam --extra-tag nuscenes  主要是为了产生几个pkl文件  2.其他一堆问题 |

|  |
| --- |
| **核心流程**      核心代码就是forward\_train，代表了模型的前向传播过程    Nuscenes每帧数据  Img\_metas:4\*4矩阵，T\_global，代表目前车辆的世界坐标系位姿  Gt\_bboxes\_3d: LiDARInstance3DBoxes,代表每个3D框在雷达坐标系的位姿  总之很麻烦  具体流程：  **1.特征提取部分**    最终提取的feature\_maps是4个不同尺度对应6张图像的特征图  **2.检测头head部分**  这部分的重点在于理解提出的Deformabel 4D Aggregation模块基本结构 |

### 2.9 Sparse4D v3

V3和v1差别还是比较大的，需要结合v2一起看

|  |
| --- |
| **核心流程**    Input:img,data  V3输入的是连续6个关键帧的数据！（不一定，这里应该是batch=6）  Gt\_boxes\_3d:每个box9个值，没用使用mmdet3d格式的BaseInstance3DBoxes  LiDARInstance3DBoxes(tensor**=**[[l**/**2, w**/**2, 0, l, w, h, 0]])  构建一个boxinstance只需要7个值      这个深度图模块输入时encder的不同尺度featuremap,输入焦距，输出不同维度的depths map（逻辑很奇怪）    输入的gt\_depth居然已经整理为这种格式了  六个相机的feat.shape torch.Size([1, 6, 256, 64, 176])合起来输入一个大的图  gt\_depth必须处理，应为lidar的数据无法覆盖整个图像  所以参与监督的深度图其实也是一个阉割版  6张周围图融合各个焦距获得一个角度的深度图（有点离谱）  **Temporal Instance Denoising**    产生dn\_anchor，需要产生正样本和负样本，总之直接把三个种类的anchor进行cat  **Decoupled Attention**    **Tracking** |
| id和bboxes有一一对应的关系  跟踪的环节    跟踪的环节其实是严重依赖于检测结果：cls,anchor的结果都会直接影响id的产生 |

### 3.0 PF-Track

说明;主体目标为3DMOT，关键在于研究它的Tracking策略

### 3.1 Our Work

|  |
| --- |
| **结构化数据输入**  1.首先实现记录前N时刻的bbox/feature queue  2.然后实现解码使用的Decoder  3.最后获得tracking结果，需要loss |

## 2D目标检测方法

### 3.1 Yolov1

Yolov1:one-stage的代表性方法

Anchor-free:



通过代码深入学习yolo的输入数据格式和loss计算方法



Yolo需要将数据集的label处理为的格式，表示每一个grid的所有bounding\_box负责的obeject(每一个grid只能负责一个obeject，也就是说两个框的中心如果落在一个grid上就无法识别了)

### 3.2 R-CNN

R-CNN:two-stage的代表性方法

|  |
| --- |
| **R-CNN**算法流程：  1.通过Selective Search算法对一张图像产生1k~2k个候选区域  2.对每个候选区域resize为统一大小的尺寸  3.对这个图片进行分类 |



### 3.3 Faster-RCNN



Faster-RCNN结构如上：重点关注RPN结构如何打造的

|  |
| --- |
| 1. **Conv layers**  无论输入图片多大，网络都会整形为MxN，（如1000x600）,然后经过conv后变为M/16xN/16(变为60x40), 60\*40\*512(特征图通道固定为512)  2. **RPN(Region Proposal Networks)**  负责计算预测框的大小，然后与先验框叠加就是最终的建议框  3. **RoiPoolingConv**  这里需要使用from torchvision.ops import RoIPool  pooled\_features = roi\_pool(features, rois)  features 是输入特征图，形状为 (N, C, H, W)，其中 N 是批次大小，C 是通道数，H 和 W 是特征图的高度和宽度。  rois 是感兴趣区域，形状为 (num\_rois, 5)，每行格式为 (batch\_index, x1, y1, x2, y2) |



总之可以直接计算ROI对应区域的特征图



在训练过程中，正样本和负样本还需要进行创建（同一个真实框会对应上多个建议框，阈值一般不会太高）

Loss部分使用smooth L1 loss函数

torch.where 用于计算平滑L1损失，是一个条件选择函数

## 多传感器融合定位

### 4.1 卡尔曼滤波与多传感器定位



上图是五个最为重要的方程

参考网站：<https://zhuanlan.zhihu.com/p/433560568>

推导非常详细，核心问题其实就是：通过最小化的MSE均值来求取，



基本方程：

|  |  |  |
| --- | --- | --- |
| 名称 | 公式 | 说明 |
| 状态方程 |  |  |
| 观测方程 |  |  |
| 预测方程 |  |  |
| 校正方程 |  |  |

基本步骤：

|  |  |  |
| --- | --- | --- |
| 步骤 | 公式 | 说明 |
| 预测 |  | 使用上一刻估计值计算先验值 |
| 卡尔曼增益 |  | 计算这一刻的先验误差的协方差 |
| 卡尔曼增益 |  | 计算卡尔曼增益 |
| 矫正 |  | 根据观测值和增益对先验值矫正 |
| 递推 |  | 计算这一刻的后验误差的协方差 |



### 因子图优化与多传感器定位

（比KF抽象得多）

## 机器学习算法归纳

百度面试需要。。。

**线性回归算法**

线性回归的目标是最小化损失函数，损失函数用于度量模型预测的准确度。最常用的损失函数是均方误差（MSE），直接使用梯度下降的方法即可求得

线性回归的几种变体：

1.简单线性回归：只涉及一个自变量和一个因变量。

2.多元线性回归：涉及多个自变量和一个因变量。

3.岭回归：通过引入L2正则化来解决多元线性回归中的过拟合问题。

4.套索回归：通过引入L1正则化来解决多元线性回归中的过拟合问题，并且能够进行特征选择。

**支持向量机算法**

支持向量机（Support Vector Machine，SVM）是一种监督学习算法，可以用于分类或回归任务。SVM的基本模型是在特征空间上找到一个最优的超平面，这个超平面能够最好地将不同类别的数据分开，同时保持最大的间隔（margin）。对于分类问题，SVM寻找一个超平面，使得距离最近的两个不同类别的数据点之间的距离最大，这样的超平面称为最大间隔超平面。

支持向量机其实有点像一个线性层

**最近邻居/k-近邻算法**

k-近邻（k-Nearest Neighbors，k-NN）算法是一种基本的机器学习算法，用于分类和回归任务。它的工作原理非常简单：对于一个未知类别的数据点，k-NN算法会在训练集中找到与它最接近的k个数据点，然后根据这k个“邻居”的类别来预测未知数据点的类别。如果k-NN用于回归任务，那么预测值通常是k个邻居的平均值。

**逻辑回归算法**

**决策树算法**

决策树（Decision Tree）是一种常见的机器学习算法，用于分类和回归任务。它是一种监督学习算法，以树形结构进行决策。每个内部节点代表一个特征，每个分支代表一个特征的测试结果，每个叶节点代表一个分类或回归结果。

**k-平均算法**

k-平均（k-Means）算法是一种无监督的聚类算法，它的目的是将相似的数据点分组到一起，形成多个簇（cluster）。算法通过迭代过程来优化簇内点的均方误差，直到达到收敛条件。

重复步骤：

a.分配步骤：将每个数据点分配到最近的质心所代表的簇。

b.更新步骤：重新计算每个簇的质心

**随机森林算法**

随机森林（Random Forest）是一种基于决策树的集成学习算法。它通过构建多个决策树并进行投票来提高预测的准确性和稳定性。随机森林在处理分类和回归任务时都非常有效，尤其是对于具有大量特征的数据集。（不会）

**朴素贝叶斯算法**

朴素贝叶斯（Naive Bayes）算法是一种基于贝叶斯定理的简单概率分类器，它假设特征之间相互独立。尽管这种独立性假设在现实世界中通常不成立，但朴素贝叶斯分类器在许多实际应用中仍然非常有效。

**降维算法**

降维（Dimensionality Reduction）是机器学习中的一个重要概念，它涉及到减少数据集的维数，同时尽可能地保留数据中的有用信息。降维的目的是为了简化模型，提高计算效率，减少过拟合的风险，并可能提高模型的泛化能力。

一些常用的降维方法。

1.主成分分析（PCA）：PCA是一种流行的线性降维方法，它通过保留数据中方差最大的几个主成分来降低维数。（使用频率较高）

2.因子分析（Factor Analysis）：因子分析类似于PCA，但它试图找到数据中的隐藏因子，这些因子可以解释数据中的变异。

3.线性判别分析（LDA）：LDA是一种线性降维方法，它通过最大化不同类别之间的距离来降低维数，同时最小化类内距离。

4.t-SNE：t-SNE是一种非线性降维方法，它通过模拟数据点在低维空间中的局部结构来降低维数。

5.自编码器（Autoencoder）：自编码器是一种神经网络，它试图通过重建原始输入来学习数据的低维表示。

6.局部线性嵌入（LLE）：LLE是一种非线性降维方法，它通过保持数据点在低维空间中的局部邻域结构来降低维数。

7.等距映射（Isomap）：等距映射是一种非线性降维方法，它通过保持数据点之间的测地距离来降低维数。