LSS算法核心原理详细解读

一、背景

自动驾驶感知的目标是从多个传感器提取语义表示，并将这些表示融合到一个单一的“鸟瞰”坐标框架中，以供运动规划使用。**我们提出了一种新的端到端架构，可以直接从任意数量的摄像机中提取给定图像数据的场景鸟瞰图表示。**

LSS是一篇发表在ECCV2020上有关自动驾驶感知方向的论文，具体子任务是object 分割和 map 分割。

论文：https://arxiv.org/pdf/2008.05711

code：https://github.com/nv-tlabs/lift-splat-shoot

二、核心

对于每个相机，将每个图像单独“Lift”成一个特征截锥。然后将所有的截锥“Splat”到鸟瞰网格中。

Lift: 将 2D 图像经过efficient-B0,提取环视图像特征，通过neck做特征融合，最后使用depthnet （1x1卷积）做深度估计（24，105，8，22），深度估计前41维度经过softmax得到深度概率（24，41，8，22），后64维度为图像语义信息（24，64，8，22），深度概率和语义信息做外积成3D伪点云（24，41，8，22，64）。

Splat: 将 3D 伪点云映射到视锥Bev中，拿掉高度维度，拍扁成 BEV。 Splat 操作可以使用诸如 PointPillar、VoxelNet 等方法。

该算法是BEV领域中的一大基石

Shoot: 根据生成的BEV网格做路径规划。

三、模型参数

3.1 Bev网格参数

LSS源码中，BEV视角下的 x轴和y轴方向的感知距离，以及BEV网格的单位大小尺寸如下：

- 感知范围
  x轴方向的感知范围 -50m ~ 50m；y轴方向的感知范围 -50m ~ 50m；z轴方向的感知范围 -10m ~ 10m；
- BEV单元格大小
  x轴方向的单位长度 0.5m；y轴方向的单位长度 0.5m；z轴方向的单位长度 20m；

- BEV的网格尺寸
  200 x 200 x 1；
- 深度估计范围
  由于LSS需要显式估计像素的离散深度，论文给出的范围是 4m ~ 45m，间隔为1m，也就是算法会估计41个离散深度；

3.2 坐标转换参数

模型用到的参数主要包括以下7个参数，分别是imgs，rots，trans，intrinsic，post_rots，post_trans，binimgs；

- imgs：输入的环视相机图片，imgs = (bs, N, 3, H, W)，N代表环视相机个数；
- rots：由相机坐标系->车身坐标系的旋转矩阵，rots = (bs, N, 3, 3)；
- trans：由相机坐标系->车身坐标系的平移矩阵，trans=(bs, N, 3)；
- intrinsic：相机内参，intrinsic = (bs, N, 3, 3)；
- post_rots：由图像增强引起的旋转矩阵，post_rots = (bs, N, 3, 3)；
- post_trans：由图像增强引起的平移矩阵，post_trans = (bs, N, 3)；
- binimgs：由于LSS做的是语义分割任务，所以会将真值目标投影到BEV坐标系，将预测结果与真值计算损失；具体而言，在binimgs中对应物体的bbox内的位置为1，其他位置为0；

四、整体流程

4.1 算法pipline

（1）生成视锥，并根据相机内外参，将视锥中的点投影到ego车体坐标系中；

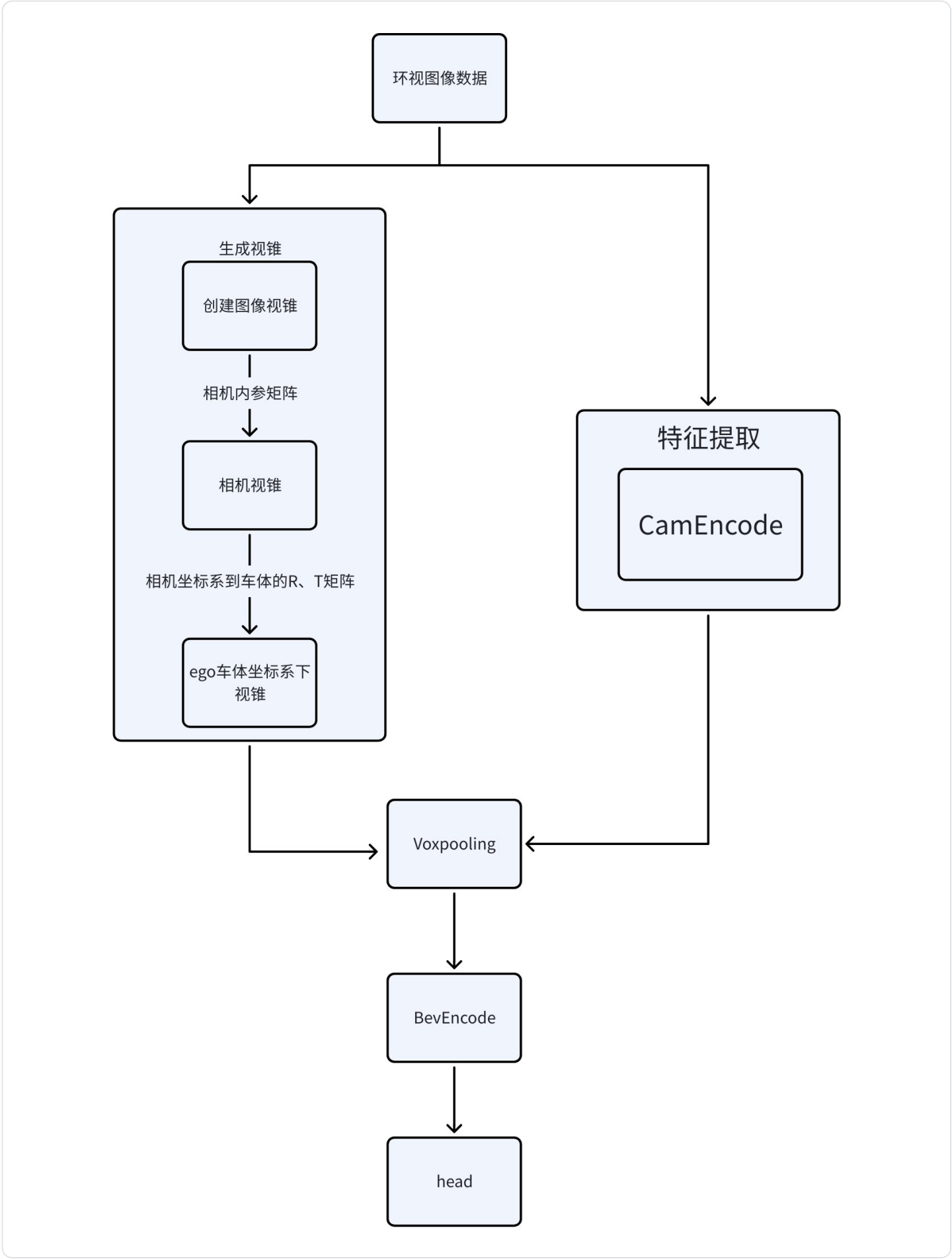
（2）对环视图像进行特征提取，并构建图像特征伪点云；

（3）利用变换后的ego 坐标系的点和图像特征点云，通过 voxel pooling 将环视图像特征转换为BEV特征；

（4）对生成的BEV特征，利用BEV encoder 做进一步特征融合。

(5) 利用特征融合后的BEV特征，添加network head获得任务结果

4.2 代码流程图


环视图像数据
生成视锥
创建图像视锥
相机内参矩阵
相机视锥
相机坐标系到车体的R、T矩阵
ego车体坐标系下视锥
特征提取
CamEncode
Voxpooling
BevEncode
head

五、代码详细分析

5.1 生成视锥：

1）原理介绍

假设用K表示相机内参矩阵，d表示三维点P在相机坐标系下的深度，该点在图像坐标系下的坐标为$(u,v,d)^T$，那么该点在相机坐标系下的坐标$(X_c,Y_c,Z_c)^T$可以表示为：

$$\begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} = K^{-1} \begin{pmatrix} ud \\ vd \\ d \end{pmatrix}$$

用R表示由相机坐标系转换到自车坐标系的旋转矩阵，$\mathbf{t}$表示由相机坐标系转换到自车坐标系的平移向量，那么自车坐标系下的点$(X_{ego},Y_{ego},Z_{ego})^T$可以表示为：

$$\begin{pmatrix} X_{ego} \\ Y_{ego} \\ Z_{ego} \end{pmatrix} = R \begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} + \mathbf{t}$$

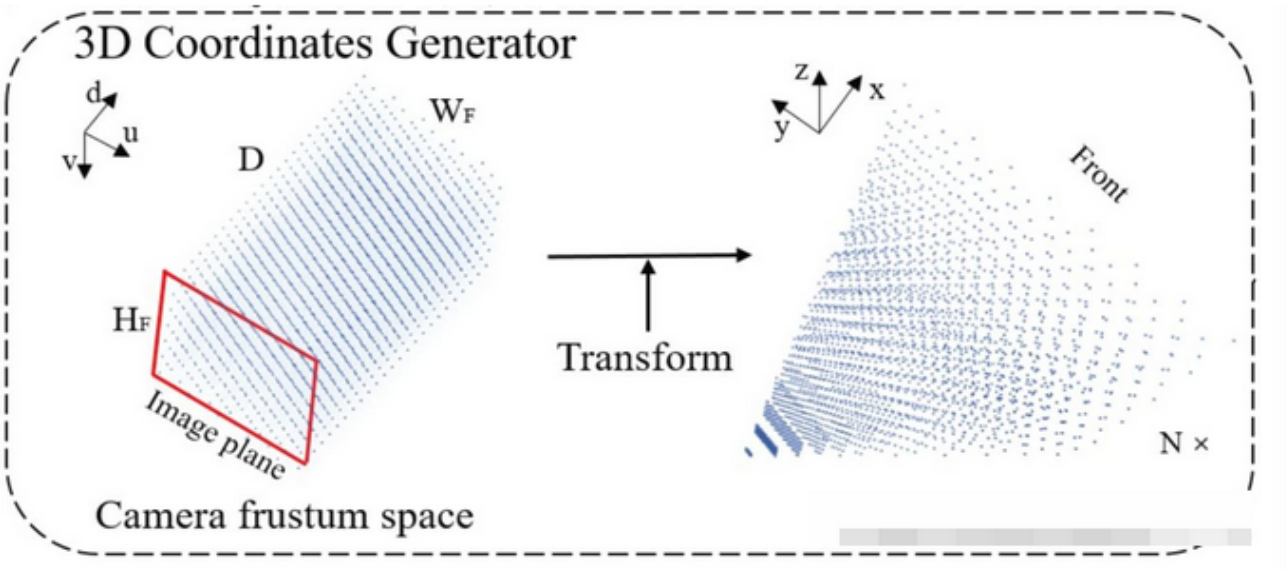


图1：图像坐标系视锥转相机坐标系视锥

2）代码段分析

a、图像视锥生成

代码块

```
#为每张图片生成视锥 ，即为上图左边立柱体视锥生成
    def create_frustum(self):
        # make grid in image plane
        # 数据增强后图片大小:ogfH:128, ogfW:352
```

```
        ogfH, ogfW = self.data_aug_conf['final_dim']
        # 下采样16倍后图像的高宽:fH:128/16=8,fw:352/16=22
        fH, fW = ogfH // self.downsample, ogfW // self.downsample

        """
        在深度方向上划分网格 ds: DxfHxfW (41x8x22)
        ['dbound'] = [4, 45, 1]->arange->[4,5,6,...,44]
        ->view->(41,1,1)->expand(扩展维度数据的尺寸)->ds:41x8x22
        """
        ds = torch.arange(*self.grid_conf['dbound'], dtype=torch.float).view(-1, 1, 1).expand(-1, fH, fW)
        D, _, _ = ds.shape  # D: 41 表示深度方向上网格的数量

        """xs:在宽度方向上划分网格, 在0到351上划分22个格子 xs: DxfHxfW(41x8x22)
           ogfW:352 -> linspace:均匀划分 -> [0,16,32..336]  大小=fW(22)
            -> view-> 1x1xfW(1x1x22)-> expand-> xs: DxfHxfW(41x8x22)
        """
        xs = torch.linspace(0, ogfW - 1, fW, dtype=torch.float).view(1, 1, fW).expand(D, fH, fW)

        """ys:在高度方向上划分网格, 在0到128上划分22个格子 ys: DxfHxfW(41x8x22)
           ogfH:128 -> linspace:均匀划分 -> [0,16,32...,112]  大小=fH(8)
            -> view-> 1xfHx1(1x8x1)-> expand-> =ys: DxfHxfW(41x8x22)
        """
        ys = torch.linspace(0, ogfH - 1, fH, dtype=torch.float).view(1, fH, 1).expand(D, fH, fW)

        """frustum: 把xs,ys,ds,堆叠到一起
           stack后-> frustum: DxfHxfWx3
           堆积起来形成网格坐标,
           frustum[i,j,k,0]就是(i,j)位置,深度为k的像素的宽度方向上的栅格坐标[41,8,22,3]
        """
        frustum = torch.stack((xs, ys, ds), -1)
        return nn.Parameter(frustum, requires_grad=False)
```

- 深度方向ds：shape:(41, 8, 22) data: [4,5,6,7,...44](建立41个深度值)
- 高度方向ys：shape(41, 8, 22) data: [0, 16, 32,..., 112]（8个高度值）
- 宽度方向xs：shape(41, 8, 22) data:[0,16, 32, ..., 336] （22个宽度值）
- Frustum 视锥： shape(41, 8, 22, 3): x =[:, :, :, 0]， y=[:, :, :, 1]，d=[:, :, :, 2]

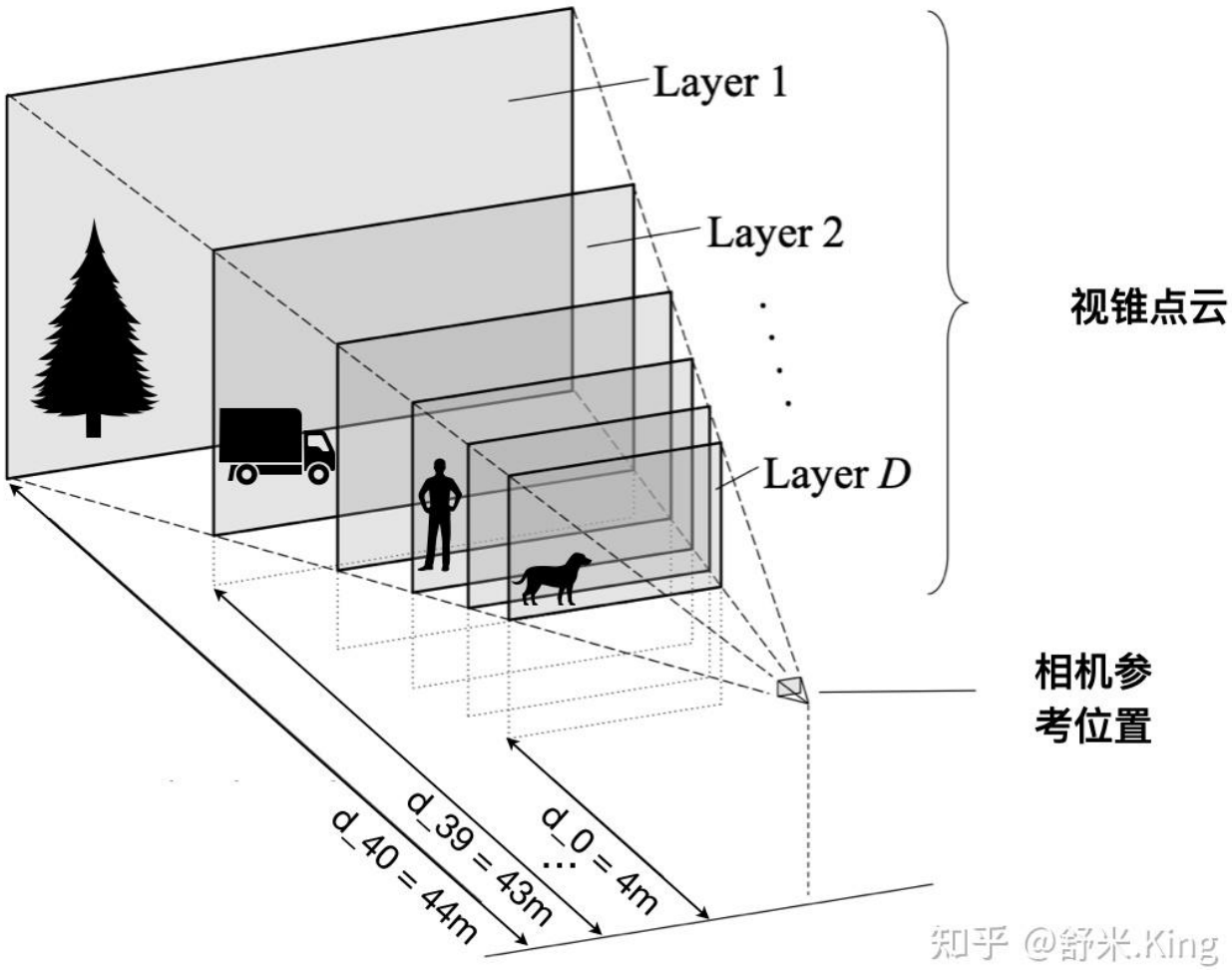


图2：相机视锥模型，由深度范围在4m～44m、间隔1m的41个平面组成

b、相机视锥 和 ego视锥生成

代码块

```
    def get_geometry(self, rots, trans, intrins, post_rots, post_trans):
            """Determine the (x,y,z) locations (in the ego frame)
            of the points in the point cloud.
            Returns B x N x D x H/downsample x W/downsample x 3
            """
            B, N, _ = trans.shape  #trans.shape: [4,6,3]
            print("trans.shape:",trans.shape)
            # 抵消图像增强以及预处理对像素的变化
            # self.frustum[B x N x D x H x W x 3]: 为每张图像生成一个棱台状的点云

            points = self.frustum - post_trans.view(B, N, 1, 1, 1, 3)
            # test_points = self.frustum
            # self.show_frustum(test_points)
            points = torch.inverse(post_rots).view(B, N, 1, 1, 1, 3,
    3).matmul(points.unsqueeze(-1))

```

```
        # cam_to_ego
        # 图像坐标系 -> 归一化相机坐标系 -> 相机坐标系 -> 车身坐标系
        """图像坐标系（柱体）->归一化相机坐标系（棱柱）
            xs,ys,lamda-> xs*lamda, ys*lamda, lamda
           对点云中预测的宽度和高度上的栅格坐标，将其乘以深度上的栅格坐标。
        """
        points = torch.cat((points[:, :, :, :, :, :2] * points[:, :, :, :, :, 2:3],
                            points[:, :, :, :, :, 2:3]
                            ), 5) #[4,6,41,8,22,3,1]
        """
           相机->ego坐标系:
           相机内参矩阵取逆:intrins: 3*3, ->inverse(取逆)
           ego 坐标系下点云坐标=
               点云视锥 *相机内参逆矩阵 * 相机坐标系到ego坐标系旋转矩阵 + 平移矩阵
        """
        combine = rots.matmul(torch.inverse(intrins))
        points = combine.view(B, N, 1, 1, 1, 3, 3).matmul(points).squeeze(-1)
        points += trans.view(B, N, 1, 1, 1, 3) #加上相机坐标系到车身坐标系的平移矩阵
        return points
```

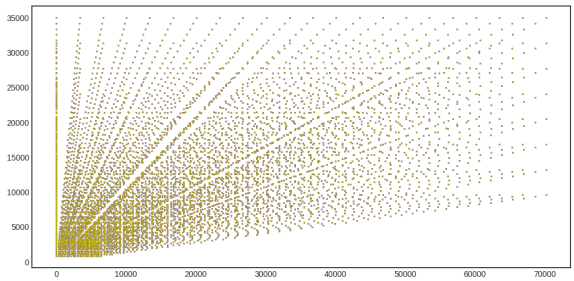


图3：归一化相机坐标系视锥图

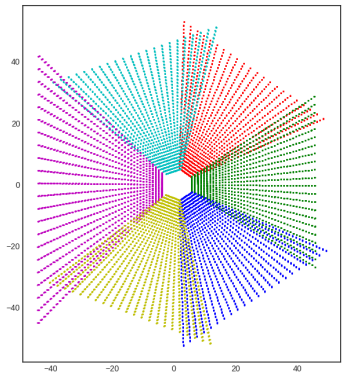
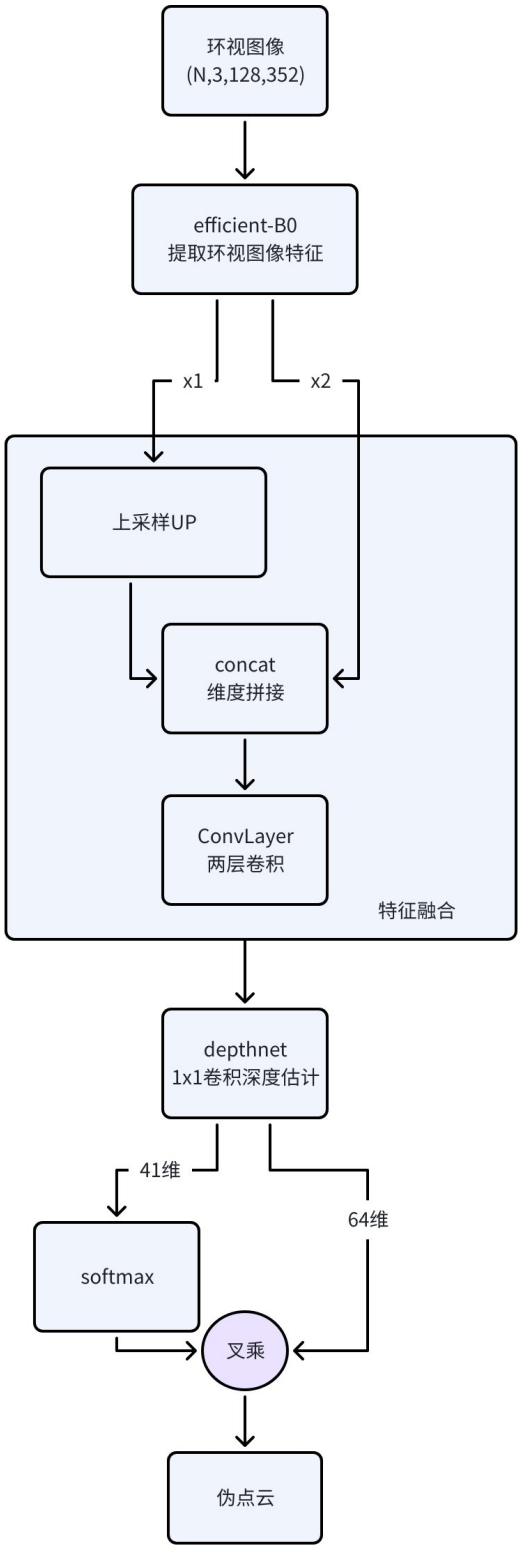


图4：环视图像车体坐标系视锥图

5.2 环视图像特征提取

1）代码流程图



2）代码pipline讲解：

a、backbone:

efficientNet-B0 提取环视图像特征，输入shape=(bs，N, 3, H, W)，输出 （24，512，8，22）

- 24：B x N -> B=4，N=6 ： 6张图片（相机的数量为6）
- fH：8；fW：22

b、Neck:

后两层特征组一个融合， 输入（x1, x2）, 对x1上采样， x1 和 x2通道拼接，Concat之后，使用ConvLayer(两层卷积)做特征融合

c、Lift 操作：

a. 使用depthnet做深度估计（1x1卷积），512->105（64+41），输出（24，105，8，22）；
   - 105个特征通道中前64个通道为语义通道，后41个通道为深度通道
   - 该部分网络进行了深度估计，通过训练所得

b. 前41维拿出来，通过softmax预测深度方向的概率分布，后64维是语义特征；

c. 利用广播机制，将语义通道24 x 64 x 1 x 8 x 22 与深度通道24 x 1 x 41 x 8 x 22进行**外积，最终获得24 x 64 x 41 x 8 x 22特征图**
   - 取出上述某一帧中一张图片的特征图：64 x 41 x 8 x 22，其中的每个特征点的维度为 64 x 41
     - 如果该特征点的表征为3m距离的狗，则64个语义通道内，表示狗的语义通道的值最大；3m对应的深度通道距离最大。两个通道利用广播的机制相乘，最大的通道即可表示为3m距离的狗
     - 广播机制如下所示，c x a 获得Matirx_(c x a)，可以通过该方式获得最大的通道

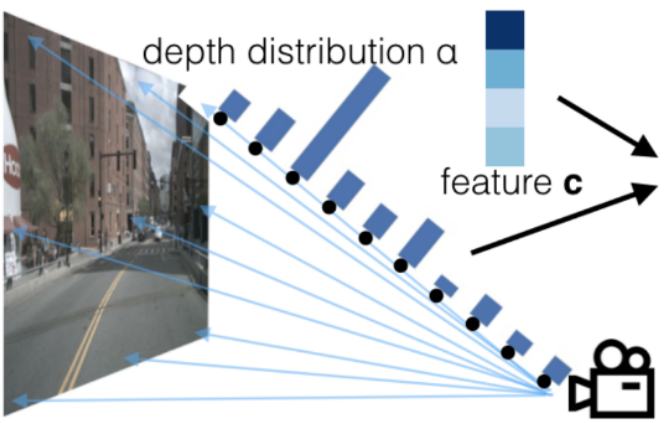


图5：深度估计概率示意图

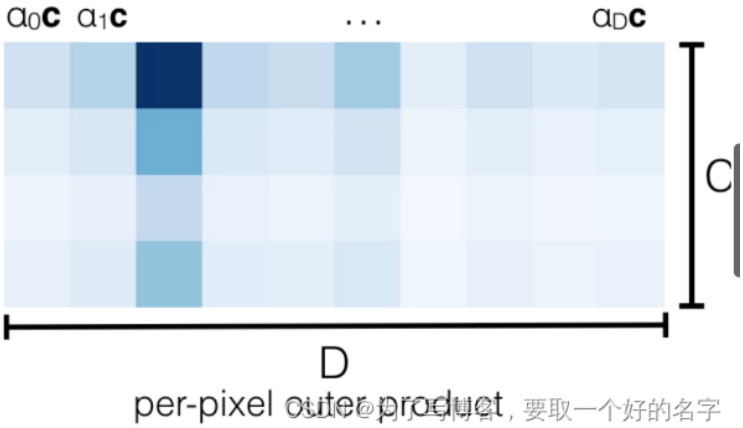


图6：深度特征：外积操作示意图

3）代码段分析

代码块

```
# 利用深度的概率和图像特征的积，完成对每个栅格点的距离预测。
def get_depth_feat(self, x):
    #通过EfficientNet提取特征,Up上采样到512->24x512x8x22
    x = self.get_eff_depth(x)
    # 1x1卷积，将数据维度变化为24x105x8x22
    x = self.depthnet(x)
    #前面d=41 depth求softmax概率分布 -> depth:24x41x8x22
    depth = self.get_depth_dist(x[:, :self.D])
```

```
        #depth*后面的64特征维度 外积->new_x:24x64x41x8x22
        new_x = depth.unsqueeze(1) * x[:, self.D:(self.D + self.C)].unsqueeze(2)
        return depth, new_x
            #通过efficientnet提取特征

    补:
    self.depthnet = nn.Conv2d(512, self.D + self.C, kernel_size=1, padding=0)

    def get_depth_dist(self, x, eps=1e-20):
        return x.softmax(dim=1)
```

5.3 Voxpooling 得到BEV特征

- **目的**：利用变换后的ego 坐标系的点和图像特征点云，通过 voxel pooling 将环视图像特征转换为BEV特征；

1) 代码pipline 功能块讲解

a、*视锥分配到预设bev栅格中*

- *向右向上平移【50,50,10],除以栅格【0.5，0.5，20】取整，*获得**栅格坐标**。
- 每个栅格大小 ds：[0.5,0.5, 20]， nx:栅格数：[200,200,1], bx:第一个栅格中心：[-49.75,-49.75,0]

代码块

```
    geom_feats = ((geom_feats - (self.bx - self.dx/2.)) / self.dx).long()
    #(self.bx - self.dx/2)  ->  初始点：[-50,-50,-10]
```

视锥图变化前后对比如下：

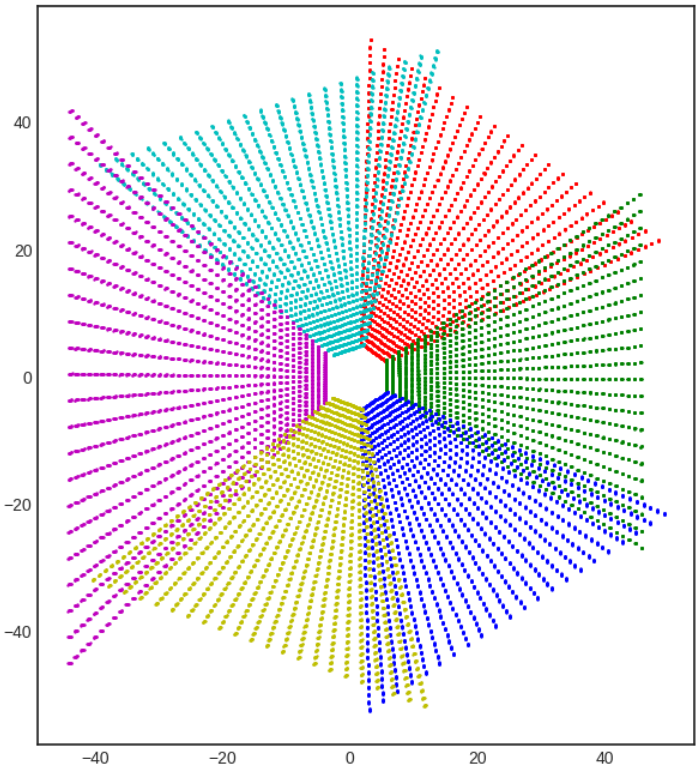


图7：环视图像视锥原始图

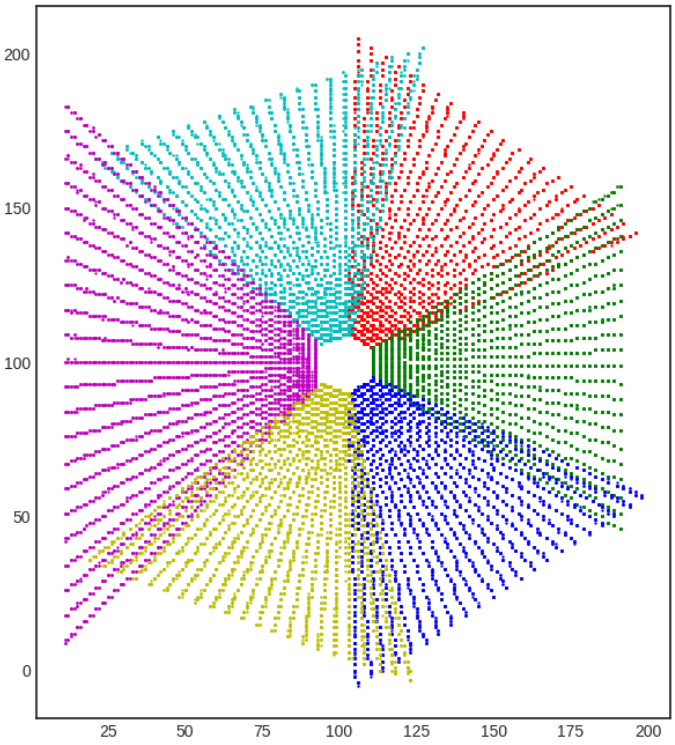


图8：环视图像ego视锥平移取整到Bev栅格示意图。

b、过滤Bev 栅格图以外的视锥点和对应图像特征

将视锥点云平铺到一维向量中

- 已有图像特征 x：B x N x D x H x W x C；视锥点 geom_feats：B x N x D x H x W x 3
- 将图像特征**铺开**，变为（B x N x D x H x W） x C；将视锥点**铺开**，变为（B x N x D x H x W） x 3 **,**视锥点提取batch,变为（B x N x D x H x W） x 4，视锥点与图像特征**一一对应**
- *通过kept过滤，只保留[200, 200,1]内的点，dx是[0.5, 0.5, 20]，所以就是保留[100, 100, 20]米的特征。 检测范围就是自车前50米后50米、左50米右50米。*

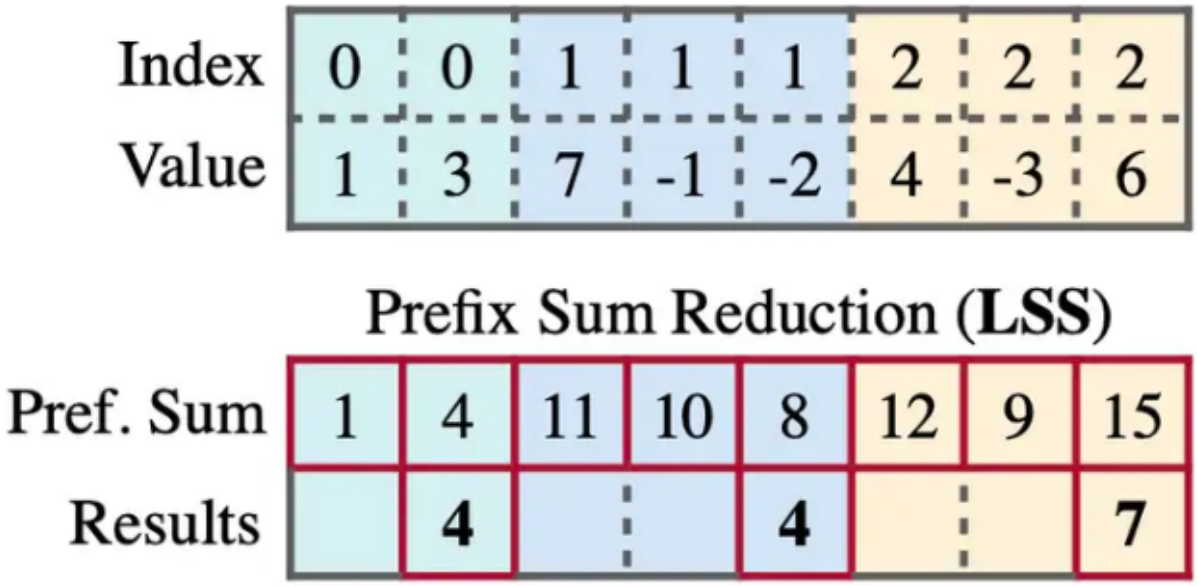
代码块

```
#过滤操作 将所有小于0，大于200的XY， 小于0大于1的Z都过滤
#x.shape=[173184, 64]  geom_feats.shape=[173184, 4]
kept = (geom_feats[:, 0] >= 0) & (geom_feats[:, 0] < self.nx[0])\
    & (geom_feats[:, 1] >= 0) & (geom_feats[:, 1] < self.nx[1])\
    & (geom_feats[:, 2] >= 0) & (geom_feats[:, 2] < self.nx[2]) # kept[true,False]
x = x[kept]
geom_feats = geom_feats[kept]
```

c、**ranks值 索引**

- **ranks 排序意义：ranks**相同的点在同一batch，也在同一栅格中。
- 计算**ranks**值，表示每个点在展开geom_feats 一维数组中的位置。
  - 公式：ranks= X x 200 x 1 x B + Y x 1 x B + Z x B + batch

d、**Cumulative Sum(CumSum) Trick**



算法目的：**计算分布在同一Bev网格中所有点的特征值和。（注意，只是相邻栅格同一维度特征求和，64维特征不混同）**

算法优点：空间换时间。累积求和可以将原本需要多次重复计算的部分提前计算好，从而减少时间复杂度。

- 排序：根据 BEV网格ID ranks排序索引，并对 图像特征x, geom_feats, ranks 相应重新排序。
- 累积和计算：对所有按BEV网格Index 排序后的视锥体特征x，对所有特征执行累积求和。 得到如上Pref.Sum行的累积求和值。
- 差分提取: 当ranks后一位与前一位不同时，取True。然后提取出True位置的累加值。

- 聚合特征：计算同一Bev网格中所有点的特征值和，即相邻网格的累计和差值

代码块

```
def cumsum_trick(x, geom_feats, ranks):
    x = x.cumsum(0)#特征在68527维度 累积和[68527,64]
    kept = torch.ones(x.shape[0], device=x.device, dtype=torch.bool)
    #kept.shape:[68527],data=[True,...,True]
    kept[:-1] = (ranks[1:] != ranks[:-1])
    #ranks 后一位和前一位不同，kept 为 True

    x, geom_feats = x[kept], geom_feats[kept]
    #过滤，每个网格只留一个点。
    x = torch.cat((x[:1], x[1:] - x[:-1]))
    # x 返回落在Bev栅格图中单个网格中所有点的特征值和
    return x, geom_feats
```

e、将图像特征点按照对应视锥点的位置（即BEV栅格的位置）填入BEV特征图中，splat掉高度维度，获得最终的BEV特征图

代码块

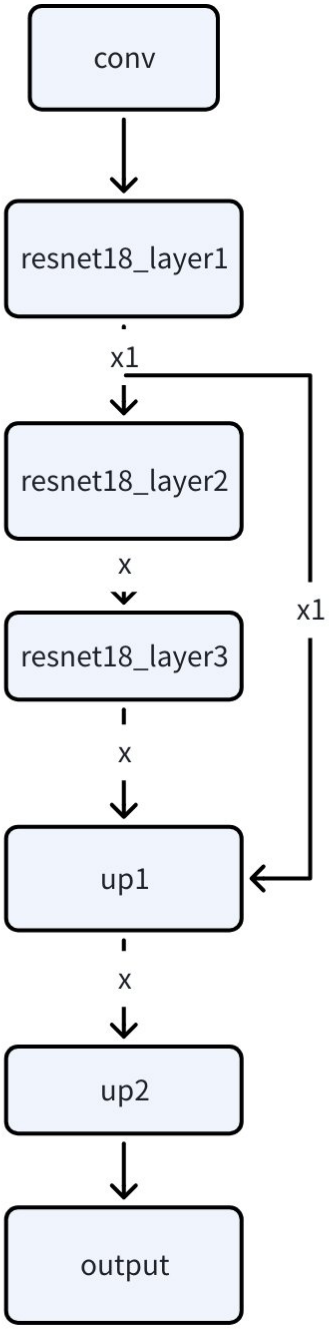
```
#图像特征映射到BEVgrid下 -> 特征final[4,64,1,200,200]
# geom_feats:[29072, 4])
# x:[29072,64]
final = torch.zeros((B, C, self.nx[2], self.nx[0], self.nx[1]),
device=x.device)
final[geom_feats[:, 3], :, geom_feats[:, 2], geom_feats[:, 0],
geom_feats[:, 1]] = x
# splat 高度维度
final = torch.cat(final.unbind(dim=2), 1)
```

5.4 BevEncode

代码块

```
class BevEncode(nn.Module):
    def __init__(self, inC, outC):
        super(BevEncode, self).__init__()

        trunk = resnet18(pretrained=False, zero_init_residual=True)
        self.conv1 = nn.Conv2d(inC, 64, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = trunk.bn1
```

```
        self.relu = trunk.relu

        self.layer1 = trunk.layer1
        self.layer2 = trunk.layer2
        self.layer3 = trunk.layer3

        self.up1 = Up(64+256, 256, scale_factor=4)
        self.up2 = nn.Sequential(
            nn.Upsample(scale_factor=2, mode='bilinear',
                                align_corners=True),
            nn.Conv2d(256, 128, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, outC, kernel_size=1, padding=0),
        )

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        x1 = self.layer1(x)
        x = self.layer2(x1)
        x = self.layer3(x)

        x = self.up1(x, x1)
        x = self.up2(x)

        return x
```



5.5 Head

- 依据不同的任务添加不同的Head即可

六、总结

-

优点：BEV基石

1. **解决遮挡问题**
   LSS通过“Lift”步骤显式估计每个像素的深度分布，将2D图像特征提升到3D空间，有效减少遮挡对感知结果的影响。这种基于深度估计的方法能够恢复被遮挡物体的潜在位置信息，提升感知鲁棒性16。
2. **多模态数据统一融合**
   LSS支持多相机数据的端到端融合，通过“Splat”将不同视角的特征投影到统一的BEV坐标系中，避免了传统后融合方法的信息丢失问题，同时降低了多传感器标定误差的影响。
3. **纯视觉低成本方案**
   作为纯视觉BEV感知的基础，LSS无需依赖昂贵的激光雷达，仅通过单目或多目相机即可实现3D感知，显著降低了自动驾驶系统的硬件成本。

缺点：

1. **计算资源消耗大**
   LSS在“Lift”和“Splat”阶段需要处理高维特征（如H×W×D×C的视锥点云），导致显存占用和计算量剧增，尤其在远距离感知时需增大BEV网格尺寸，进一步加剧资源消耗。
2. **几何压缩导致细节丢失**
   BEV特征通过高度轴压缩和栅格化池化（Sum Pooling）生成，导致物体高度信息和局部细节丢失，影响3D检测（如边界框精度）和小目标感知能力。
3. **依赖深度估计精度**
   若深度估计存在偏差（如单目相机的深度歧义性），投影到BEV的特征位置可能不准确，进而影响感知结果。
4. **时序信息利用不足**

原始LSS主要针对单帧数据处理，未有效融合时序信息，难以应对动态场景中的运动目标跟踪和自车运动补偿问题。

改进：

1. **结合稀疏表示优化计算效率**
   - **稀疏BEV与实例级增强**：如LSSInst算法引入稀疏实例级表示，通过自适应采样关键点（如物体中心或边缘）补充BEV缺失的细节，提升3D检测精度。
   - **动态点云筛选**：仅在物体可能存在的区域进行深度采样，减少无效计算（如地平线Sparse4D系列通过可学习关键点优化特征采样）。
2. **时序融合与运动补偿**
   - 引入时序BEV编码器，结合历史帧特征和自车运动信息，补偿动态目标的位移（如LSSInst中的运动补偿模块）。
   - 使用4D关键点建模时空一致性，提升长时序场景下的感知稳定性。

3. **硬件部署优化**
    - **算子级优化**：如地平线征程5芯片通过替换大尺寸乘法操作为分步计算、优化BEV池化索引操作（如用Grid Sample替代传统映射），显著降低延迟。
    - **量化与轻量化**：采用低比特量化（如int16）和轻量级网络结构（如EfficientNet）平衡精度与算力需求。
4. **多模态融合增强**
    - **占用网络（Occupancy Network）**：结合激光雷达或4D毫米波雷达的点云信息，通过体素化占用检测弥补纯视觉深度估计的不足（如特斯拉方案)。
    - **Transformer增强特征融合**：利用注意力机制动态加权多视角特征，提升异构传感器（如相机与雷达）的融合效果。
5. **自监督与数据驱动优化**
    - 通过自动标注和影子模式收集大规模训练数据，提升深度估计的泛化能力。
    - 引入数据增强策略（如BEVRotate）提升模型对视角变化的鲁棒性。