Supplementary Material for Voxel Proposal Network via Multi-Frame Knowledge Distillation for Semantic Scene Completion

This supplemental material presents: (1) more visualization results of competitive methods (i.e., LMSCNet [1], SSA-SC [2]) on SemanticKITTI [3] dataset; (2) code segments of CVP.

A Visualization Comparsion

We present more visualization results in Figure 1.

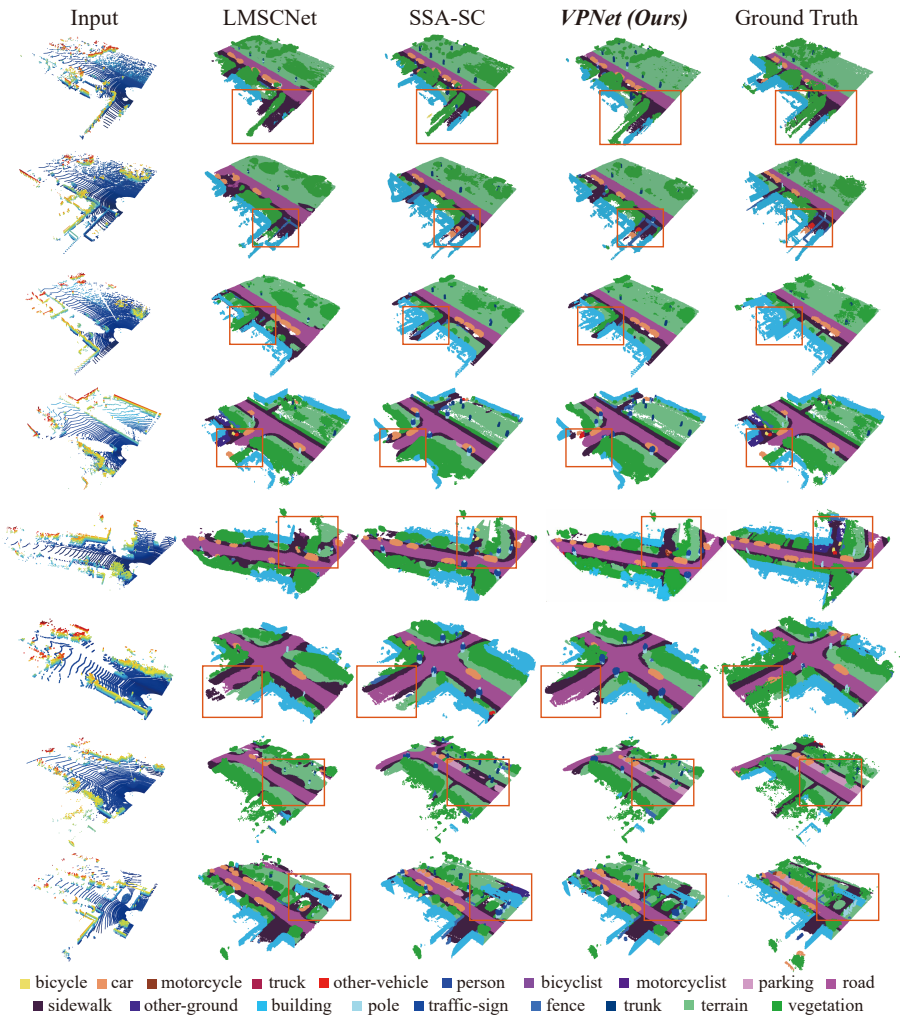

Figure 1: Completion results of different methods on SemanticKITTI validation set.

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

B Core code

We show some of the core code about CVP below.

```
import torch
import torch.nn as nn
import spconv.pytorch as spconv
from spconv.pytorch import functional as Fsp
import torch_scatter
from torch.nn.utils.rnn import pad_sequence
import spconv.pytorch as spconv

def conv3x3(in_planes, out_planes, stride=1, indice_key=None):
    return spconv.SubMConv3d(in_planes, out_planes, kernel_size
        =3, stride=stride, padding=1, bias=False, indice_key=
        indice_key)

def conv1x3(in_planes, out_planes, stride=1, indice_key=None):
    return spconv.SubMConv3d(in_planes, out_planes, kernel_size
        =(1, 3, 3), stride=stride, padding=(0, 1, 1), bias=
        False, indice_key=indice_key)

def conv1x1x3(in_planes, out_planes, stride=1, indice_key=None)
    :
    return spconv.SubMConv3d(in_planes, out_planes, kernel_size
        =(1, 1, 3), stride=stride, padding=(0, 0, 1), bias=
        False, indice_key=indice_key)

def conv1x3x1(in_planes, out_planes, stride=1, indice_key=None)
    :
    return spconv.SubMConv3d(in_planes, out_planes, kernel_size
        =(1, 3, 1), stride=stride, padding=(0, 1, 0), bias=
        False, indice_key=indice_key)

def conv3x1x1(in_planes, out_planes, stride=1, indice_key=None)
    :
    return spconv.SubMConv3d(in_planes, out_planes, kernel_size
        =(3, 1, 1), stride=stride, padding=(1, 0, 0), bias=
        False, indice_key=indice_key)

def conv3x1(in_planes, out_planes, stride=1, indice_key=None):
    return spconv.SubMConv3d(in_planes, out_planes, kernel_size
        =(3, 1, 3), stride=stride, padding=(1, 0, 1), bias=
        False, indice_key=indice_key)

def conv1x1(in_planes, out_planes, stride=1, indice_key=None):
    return spconv.SubMConv3d(in_planes, out_planes, kernel_size
        =1, stride=stride, padding=1, bias=False, indice_key=
        indice_key)

def extract_nonzero_features(x):
    nonzero_index = torch.sum(torch.abs(x), dim=1).nonzero()
    coords = nonzero_index.type(torch.int32)
    channels = int(x.shape[1])
    features = x.permute(0, 2, 3, 4, 1).reshape(-1, channels)
    features = features[torch.sum(torch.abs(features), dim=1).
        nonzero(), :]
    features = features.squeeze(1)
    coords= torch.unique(coords, return_inverse=False,
        return_counts=False, dim=0)
```

```
    return coords, features

class DDCM(nn.Module):
    def __init__(self, in_filters, out_filters, kernel_size=(3,
        3, 3), stride=1, indice_key=None):
        super(DDCM, self).__init__()
        self.indice_key = indice_key
        self.conv1 = conv3x1x1(in_filters, out_filters,
            indice_key=indice_key + "bef")
        self.bn0 = nn.BatchNorm1d(out_filters)
        self.act1 = nn.Sigmoid()

        self.conv1_2 = conv1x3x1(in_filters, out_filters,
            indice_key=indice_key + "bef")
        self.bn0_2 = nn.BatchNorm1d(out_filters)
        self.act1_2 = nn.Sigmoid()

        self.conv1_3 = conv1x1x3(in_filters, out_filters,
            indice_key=indice_key + "bef")
        self.bn0_3 = nn.BatchNorm1d(out_filters)
        self.act1_3 = nn.Sigmoid()

    def forward(self, x):
        shortcut = self.conv1(x)
        shortcut = shortcut.replace_feature(self.bn0(shortcut.
            features))
        shortcut = shortcut.replace_feature(self.act1(shortcut.
            features))

        x.indice_dict[self.indice_key + "bef"] = shortcut.
            indice_dict[self.indice_key + "bef"]
        x.indice_dict[self.indice_key + "bef"].ksize = [1,3,1]
        shortcut2 = self.conv1_2(x)
        shortcut2 = shortcut2.replace_feature(self.bn0_2(
            shortcut2.features))
        shortcut2 = shortcut2.replace_feature(self.act1_2(
            shortcut2.features))

        x.indice_dict[self.indice_key + "bef"] = shortcut.
            indice_dict[self.indice_key + "bef"]
        x.indice_dict[self.indice_key + "bef"].ksize = [1,1,3]
        shortcut3 = self.conv1_3(x)
        shortcut3 = shortcut3.replace_feature(self.bn0_3(
            shortcut3.features))
        shortcut3 = shortcut3.replace_feature(self.act1_3(
            shortcut3.features))
        shortcut = shortcut3.replace_feature(shortcut.features
            + shortcut2.features + shortcut3.features)

        shortcut = shortcut.replace_feature(shortcut.features *
             x.features + x.features)
        return shortcut

class ConfidentVoxelProposal(nn.Module):
    def __init__(self, in_planes, spatial_shape, noise_channels
       =4):
        super(ConfidentVoxelProposal, self).__init__()
        self.in_planes = in_planes
        self.spatial_shape = spatial_shape
```

```
        self.noise_channels = noise_channels

        ####   add channel 4 for random noise
        self.mlp_0= nn.Sequential(nn.Conv1d(in_planes + 3 +
            noise_channels, in_planes * 2 , kernel_size=1,
            stride=1, bias=False), nn.BatchNorm1d(in_planes *
            2), nn.ReLU())

        self.mlp_1 = nn.Sequential(nn.Conv1d(in_planes * 2,
            in_planes, kernel_size=1, stride=1, bias=False), nn
            .BatchNorm1d(in_planes), nn.ReLU())

        self.mlp_2 = nn.Sequential(nn.Conv1d(in_planes, 3,
            kernel_size=1, stride=1, bias=True), nn.Tanh())

        self.ReconNet = DDCM(in_planes, in_planes, indice_key="
            recon")
        self.fusion = WeightedFusion(in_planes)

    def forward(self, dense_voxels, batch_size):
        ## Step 0 : calculate offset
        #### position embedding
        coords, features = extract_nonzero_features(
            dense_voxels)
        origin_sparse_voxel = spconv.SparseConvTensor(features,
             coords, self.spatial_shape, batch_size)
        #### pad
        counts = torch.unique(coords[:,0], return_counts=True)
            [-1]
        batch = torch.arange(batch_size).unsqueeze_(0).repeat(
            torch.max(counts),1).reshape(-1,1).to(dense_voxels.
            device)
        features = torch.cat((coords[:,1:], features), dim=1)
        features = torch.split(features, counts.tolist())
        features = pad_sequence(features, batch_first=True,
            padding_value=0)
        coords = torch.cat((batch, features[:,:,:3].reshape
            (-1,3)), dim=1)
        #### random noise and position_embedding
        features[:,:,0] /= self.spatial_shape[0]
        features[:,:,1] /= self.spatial_shape[1]
        features[:,:,2] /= self.spatial_shape[2]
        #### b0
        noise_b0 = torch.normal(mean=0, std=torch.ones((
            batch_size, self.noise_channels, torch.max(counts))
            , device=dense_voxels.device))
        cat_features_noise_0 = torch.cat((features.permute
            (0,2,1), noise_b0), dim=1)

        offset_b0 = self.mlp_2(self.mlp_1(self.mlp_0(
            cat_features_noise_0))).permute(0,2,1).reshape
            (-1,3)
        offset_b0 = offset_b0 * torch.tensor([self.
            spatial_shape[0] / 2, self.spatial_shape[1] / 2,
            self.spatial_shape[2] / 2 ], device=dense_voxels.
            device).float()
        new_bxyz_b0 = torch.cat((coords[:,0].unsqueeze(1),
            coords[:,1:] + offset_b0), dim=1)
```

```
coords_b0 = torch.cat((coords[:,0].unsqueeze(1), (
    coords[:,1:] + offset_b0).floor()), dim=1).int()
## Step 1 : propagate feature
#### feature = feature / (offset)^2^0.5 + 0.0001
features_b0 = features[:,:,3:].reshape(-1, self.
    in_planes) * (1.0 / torch.sqrt(torch.sum(offset_b0
    **2,dim=1)) + 0.001).unsqueeze(1)
#### concat the new coords and the new features for
    mean at the same time with right correspondence
features_b0 = torch.cat((coords_b0, features_b0), dim
    =1)
features_b0 = torch_scatter.scatter_mean(features_b0,
    torch.unique(coords_b0, return_inverse=True, dim=0)
    [-1], dim=0)
coords_b0 = features_b0[:,:4].int()
features_b0 = features_b0[:,4:]
coords_b0[:,1] = torch.clamp(coords_b0[:,1], 0, self.
    spatial_shape[0] - 1)
coords_b0[:,2] = torch.clamp(coords_b0[:,2], 0, self.
    spatial_shape[1] - 1)
coords_b0[:,3] = torch.clamp(coords_b0[:,3], 0, self.
    spatial_shape[2] - 1)
post_sparse_voxels_b0 = spconv.SparseConvTensor(
    features_b0, coords_b0.int(), self.spatial_shape,
    batch_size)

#### b1
noise_b1 = torch.normal(mean=0, std=torch.ones((
    batch_size, self.noise_channels, torch.max(counts))
    , device=dense_voxels.device))
cat_features_noise_1 = torch.cat((features.permute
    (0,2,1), noise_b1), dim=1)
offset_b1 = self.mlp_2(self.mlp_1(self.mlp_0(
    cat_features_noise_1))).permute(0,2,1).reshape
    (-1,3)
offset_b1 = offset_b1 * torch.tensor([self.
    spatial_shape[0] / 2, self.spatial_shape[1] / 2,
    self.spatial_shape[2] / 2 ], device=dense_voxels.
    device).float()
new_bxyz_b1 = torch.cat((coords[:,0].unsqueeze(1),
    coords[:,1:] + offset_b1), dim=1)
coords_b1 = torch.cat((coords[:,0].unsqueeze(1), (
    coords[:,1:] + offset_b1).floor()), dim=1).int()
features_b1 = features[:,:,3:].reshape(-1, self.
    in_planes) * (1.0 / torch.sqrt(torch.sum(offset_b1
    **2,dim=1)) + 0.001).unsqueeze(1)
features_b1 = torch.cat((coords_b1, features_b1), dim
    =1)
features_b1 = torch_scatter.scatter_mean(features_b1,
    torch.unique(coords_b1, return_inverse=True, dim=0)
    [-1], dim=0)
coords_b1 = features_b1[:,:4].int()
features_b1 = features_b1[:,4:]
coords_b1[:,1] = torch.clamp(coords_b1[:,1], 0, self.
    spatial_shape[0] - 1)
coords_b1[:,2] = torch.clamp(coords_b1[:,2], 0, self.
    spatial_shape[1] - 1)
coords_b1[:,3] = torch.clamp(coords_b1[:,3], 0, self.
    spatial_shape[2] - 1)
```

```
post_sparse_voxels_b1 = spconv.SparseConvTensor(
    features_b1, coords_b1.int(), self.spatial_shape,
    batch_size)

#### b2
noise_b2 = torch.normal(mean=0, std=torch.ones((
    batch_size, self.noise_channels, torch.max(counts))
    , device=dense_voxels.device))
cat_features_noise_2 = torch.cat((features.permute
    (0,2,1), noise_b2), dim=1)
offset_b2 = self.mlp_2(self.mlp_1(self.mlp_0(
    cat_features_noise_2))).permute(0,2,1).reshape
    (-1,3)
offset_b2 = offset_b2 * torch.tensor([self.
    spatial_shape[0] / 2, self.spatial_shape[1] / 2,
    self.spatial_shape[2] / 2 ], device=dense_voxels.
    device).float()
new_bxyz_b2 = torch.cat((coords[:,0].unsqueeze(1),
    coords[:,1:] + offset_b2), dim=1)
coords_b2 = torch.cat((coords[:,0].unsqueeze(1), (
    coords[:,1:] + offset_b2).floor()), dim=1).int()
features_b2 = features[:,:,3:].reshape(-1, self.
    in_planes) * (1.0 / torch.sqrt(torch.sum(offset_b2
    **2,dim=1)) + 0.001).unsqueeze(1)
features_b2 = torch.cat((coords_b2, features_b2), dim
    =1)
features_b2 = torch_scatter.scatter_mean(features_b2,
    torch.unique(coords_b2, return_inverse=True, dim=0)
    [-1], dim=0)
coords_b2 = features_b2[:,:4].int()
features_b2 = features_b2[:,4:]
coords_b2[:,1] = torch.clamp(coords_b2[:,1], 0, self.
    spatial_shape[0] - 1)
coords_b2[:,2] = torch.clamp(coords_b2[:,2], 0, self.
    spatial_shape[1] - 1)
coords_b2[:,3] = torch.clamp(coords_b2[:,3], 0, self.
    spatial_shape[2] - 1)
post_sparse_voxels_b2 = spconv.SparseConvTensor(
    features_b2, coords_b2.int(), self.spatial_shape,
    batch_size)

sparse_voxel_proposal_b0 = Fsp.sparse_add(
    post_sparse_voxels_b0, origin_sparse_voxel)
sparse_voxel_proposal_b1 = Fsp.sparse_add(
    post_sparse_voxels_b1, origin_sparse_voxel)
sparse_voxel_proposal_b2 = Fsp.sparse_add(
    post_sparse_voxels_b2, origin_sparse_voxel)

dense_voxel_proposal = self.fusion(
    sparse_voxel_proposal_b0, sparse_voxel_proposal_b1,
     sparse_voxel_proposal_b2)
coords, features = extract_nonzero_features(
    dense_voxel_proposal)
sparse_voxel_proposal = spconv.SparseConvTensor(
    features, coords, self.spatial_shape, batch_size)
sparse_voxel_proposal = self.ReconNet(
    sparse_voxel_proposal)
dense_voxel_proposal = sparse_voxel_proposal.dense()
del batch, noise_b0, noise_b1, noise_b2
```

```
    return dense_voxel_proposal, new_bxyz_b0, new_bxyz_b1,
        new_bxyz_b2
```

References

[1] Luis Roldao, Raoul de Charette, and Anne Verroust-Blondet. Lmscnet: Lightweight multiscale 3d semantic completion. In *2020 International Conference on 3D Vision (3DV)*, pages 111–119. IEEE, 2020.

[2] Xuemeng Yang, Hao Zou, Xin Kong, Tianxin Huang, Yong Liu, Wanlong Li, Feng Wen, and Hongbo Zhang. Semantic segmentation-assisted scene completion for lidar point clouds. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3555–3562. IEEE, 2021.

[3] Jens Behley, Martin Garbade, Andres Milioto, Jan Quenzel, Sven Behnke, Cyrill Stachniss, and Jurgen Gall. Semantickitti: A dataset for semantic scene understanding of lidar sequences. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9297–9307, 2019.