A. Architecture Details

Our base model used a standard (2+1)D ResNet-50 [1]. The camera transform is inserted into the network usually after the 3rd block (in the main paper we compared all locations). Usually this network used 256 channels for the representation and we used 3 cameras (i.e., 3 different 2D projections). The total number of parameters of the 3 main models is summarized in Table 1. Our layer adds only 280k parameters (only about 1% of the parameters), but significantly improves performance on unseen views. It further has significantly better runtime performance than spherical CNNs.

Table 1: Comparison of the number of parameters in the 3 main models. Adding the geometric projection layer only adds 280k parameters, but greatly improves performance.

<table>
<thead>
<tr>
<th>Model</th>
<th># params</th>
<th>∆</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2+1)D ResNet-50</td>
<td>21.3M</td>
<td>0</td>
</tr>
<tr>
<td>(2+1)D ResNet-50 + Ours</td>
<td>21.5M</td>
<td>280k</td>
</tr>
<tr>
<td>Spherical CNNs</td>
<td>21.2M</td>
<td>-123k</td>
</tr>
</tbody>
</table>

B. Full Results

The full numerical results from plots in the paper are provided here.

Table 2: How many cameras to use.

<table>
<thead>
<tr>
<th>Method</th>
<th>MLB Seen</th>
<th>MLB Unseen</th>
<th>TSH Seen</th>
<th>TSH Unseen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>55.6</td>
<td>30.2</td>
<td>49.8</td>
<td>34.2</td>
</tr>
<tr>
<td>1 Cam</td>
<td>57.4</td>
<td>38.6</td>
<td>53.2</td>
<td>38.5</td>
</tr>
<tr>
<td>2 Cams</td>
<td>58.1</td>
<td>41.8</td>
<td>53.9</td>
<td>39.1</td>
</tr>
<tr>
<td>4 Cams</td>
<td>58.9</td>
<td>42.7</td>
<td>54.5</td>
<td>39.6</td>
</tr>
<tr>
<td>8 Cams</td>
<td>58.7</td>
<td>42.7</td>
<td>54.5</td>
<td>39.4</td>
</tr>
</tbody>
</table>

Table 3: Where in network to add layer.

<table>
<thead>
<tr>
<th>Method</th>
<th>MLB Seen</th>
<th>MLB Unseen</th>
<th>TSH Seen</th>
<th>TSH Unseen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1</td>
<td>57.8</td>
<td>42.1</td>
<td>54.3</td>
<td>39.2</td>
</tr>
<tr>
<td>Block 2</td>
<td>58.3</td>
<td>42.4</td>
<td>54.4</td>
<td>39.2</td>
</tr>
<tr>
<td>Block 3</td>
<td>58.9</td>
<td>42.7</td>
<td>54.5</td>
<td>39.6</td>
</tr>
<tr>
<td>Block 4</td>
<td>57.4</td>
<td>41.7</td>
<td>53.8</td>
<td>38.9</td>
</tr>
<tr>
<td>Block 5</td>
<td>57.1</td>
<td>40.9</td>
<td>53.3</td>
<td>37.7</td>
</tr>
</tbody>
</table>
C. PyTorch Implementation

We provide the code here to implement the camera projection layer.

```python
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

device = torch.device('cuda')

def rotation_tensor(theta, phi, psi, b=1):
    """
    Takes theta, phi, and psi and generates the 3x3 rotation matrix. Works for batched ops
    As well, returning a Bx3x3 matrix.
    """
    one = torch.ones(b, 1, 1).to(device)
    zero = torch.zeros(b, 1, 1).to(device)
    rot_x = torch.cat((
        torch.cat((one, zero, zero), 1),
        torch.cat((zero, theta.cos(), theta.sin()), 1),
        torch.cat((zero, -theta.sin(), theta.cos()), 1),
    ), 2)
    rot_y = torch.cat((
        torch.cat((phi.cos(), zero, -phi.sin()), 1),
        torch.cat((zero, one, zero), 1),
        torch.cat((phi.sin(), zero, phi.cos()), 1),
    ), 2)
    rot_z = torch.cat((
        torch.cat((psi.cos(), -psi.sin(), zero), 1),
        torch.cat((psi.sin(), psi.cos(), zero), 1),
        torch.cat((zero, zero, one), 1)
    ), 2)
    return torch.bmm(rot_z, torch.bmm(rot_y, rot_x))

class CameraProps(nn.Module):
    """
```
Generates the extrinsic rotation and translation matrix for the current camera. Takes some feature as input, then returns the rotation matrix \((3 \times 3)\) and translation \((3 \times 1)\).

```python
def __init__(self, channels):
    super(CameraProps, self).__init__()
    self.cam = nn.Conv2d(channels, 128, 3)
    self.cam2 = nn.Linear(128, 32)
    self.rot = nn.Linear(32, 3)
    self.trans = nn.Linear(32, 3)

def forward(self, x):
    x = F.relu(self.cam(x))
    # averages x over space, time
    # then provides 3x3 rot and 3-dim trans
    x = torch.mean(torch.mean(x, dim=2), dim=2)
    x = F.relu(self.cam2(x))
    b = x.size(0)
    r = self.rot(x)
    return rotation_tensor(r[:, 0], r[:, 1], r[:, 2], b), self.trans(x).view(b, 3, 1, 1)

class CameraProjection(nn.Module):
    
    Does the camera transforms and multi-view projection described in the paper.

    def __init__(self, num_cameras):
        super(CameraProjection, self).__init__()
        self.cameras = nn.ParameterList()
        self.cam_rot = nn.ParameterList()
        for c in range(num_cameras):
            self.cameras.append(nn.Parameter(torch.randn(4)*2-1))
            self.cam_rot.append(nn.Parameter(torch.randn(3)*np.pi))

def forward(self, x, rot, trans):
    # X is a list of [F, x, y, z] feature maps
    # or X is a [C, W, H] feature map
    # rot, trans are the extinsic camera parameters
    if isinstance(x, list):
        # if it is a list, process each feature map
        # resulting in a [C, W, H] as input
        output = [self.forward(f, rot, trans) for f in x]
        return torch.cat(output, dim=1) # channels is dim 1
    # x is now a [F, x, y, z] input where F is the feature
    fts = x[:, :, :-3] # get feature value, a B x F x H x W tensor
    pt = x[:, :, -3:] # get 3D point locations, a B x 3 x H x W tensor
    # rot is a 3x3 matrix
    # pw is 3x3 matrix applied along dim
    pw = torch.einsum('bphw,bpq->bqhw', pt, rot)
    pw += trans # add 3D translation
    # pw is now world coordinates at each feature map location
    # we do 2D projection next
```

views = []
for r, c in zip(self.cam_rot, self.cameras):
    rot = rotation_tensor(r[0].view(1, 1, 1), r[1].view(1, 1, 1), r[2].view(1, 1, 1))
    cam_pt = torch.einsum('bphw, pq->bqhw', pw, rot.squeeze(0))

    proj = torch.stack([(cam_pt[:, 0]*c[0] + c[2]),
                        (cam_pt[:, 1]*c[1] + c[3])], dim=-1)
    proj = torch.tanh(proj) # apply tanh to get values in [-1,1]
views.append(F.grid_sample(fts, proj))
return torch.cat(views, dim=1)

This layer can easily be inserted anywhere into a CNN. For example, assume the following code generates a ResNet. Then the camera transform is used as:

class Net(nn.Module):
    def __init__(self, ...):
        self.layers = # ResNet Layers
        self.cam_props = CameraProps(channels)
        self.camera_proj = CameraProjection(num_cams)

    def forward(self, video):
        x = video
        for i, layer in enumerate(self.layers):
            x = layer(x)
            if i = apply_camera_layer_loc:
                rot, trans = self.cam_props(x)
                x = self.camera_proj(x, rot, trans)
        return x

References