–Supplemental Materials–
Recognizing Actions in Videos from Unseen Viewpoints

First Author
Institution1
Institution1 address
firstauthor@i1.org

Second Author
Institution2
First line of institution2 address
secondauthor@i2.org

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.

Model	# params	Δ
(2+1)D ResNet-50	21.3M	0
(2+1)D ResNet-50 + Ours	21.5M	280k
Spherical CNNs	21.2M	-123k

B. Full Results

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

Table 2: How many cameras to use.

Method	MLB		TSH	
	Seen	Unseen	Seen	Unseen
Baseline	55.6	30.2	49.8	34.2
1 Cam	57.4	38.6	53.2	38.5
2 Cams	58.1	41.8	53.9	39.1
4 Cams	58.9	42.7	54.5	39.6
8 Cams	58.7	42.7	54.5	39.4

Table 3: Where in network to add layer.

Method	MLB		TSH	
	Seen	Unseen	Seen	Unseen
Block 1	57.8	42.1	54.3	39.2
Block 2	58.3	42.4	54.4	39.2
Block 3	58.9	42.7	54.5	39.6
Block 4	57.4	41.7	53.8	38.9
Block 5	57.1	40.9	53.3	37.7

C. PyTorch Implementation

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

```
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 (3x3) and translation (3x1)
  """
  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.rand(4)*2-1))
      self.cam_rot.append(nn.Parameter(torch.rand(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 extensic 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 dim1
    # 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

[1] Du Tran, Heng Wang, Lorenzo Torresani, Jamie Ray, Yann LeCun, and Manohar Paluri. A closer look at spatiotemporal convolutions for action recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6450–6459, 2018. 1