Appendix

1. Training Details of Static Baseline

The static baseline that we consider in the main text is arch2 from [4], which we pre-train on CamVid [1] and CityScapes [3].

We utilise the ‘poly’ learning schedule [2] with the initial learning rates of $5 \times 10^{-2}$ and $1 \times 10^{-2}$ for the encoder and the decoder, respectively. As in [4], we set the weight for auxiliary losses to 0.3.

On CityScapes, we train for 1000 epochs with mini-batches of 28 examples each randomly scaled with the scale factor in range of $[0.5, 2.0]$ and randomly cropped to $800 \times 800$ with each side zero-padded accordingly. On CamVid, we train for 2000 epochs with mini-batches of 32 examples each randomly scaled with the scale factor in range of $[0.5, 2.0]$ and randomly cropped to $600 \times 600$ with each side zero-padded accordingly.

2. CamVid Experiments with Raw Videos

In addition to the experiments on the main set of CamVid with neighbouring annotated frames we also conduct experiments with frames extracted from the raw videos.

For both training and testing we use sequences with 3 frames, each 1/30 seconds apart, with the last frame being annotated. As the extracted frames slightly differ from those in the main set of CamVid, the qualitative numbers are not directly comparable, hence we re-train the static baseline and also re-do the search following the setup of the main experiments.

We provide the qualitative results in Table 1. Opposed to the experiments with frames spaced far from each other, here we do see dynamism for both considered cells, with the newly found cell significantly outperforming the baseline and the other cell. The structure of cell4 is visualised in Fig. 1.

3. Search Space Aggregation Operations

In addition to the definitions of all operations in the main text, we provide the code for each aggregation operation written in PyTorch [5] in Listings 1, 2 and 3.

---

Table 1. Quantitative results on the test set of CamVid on frames extracted from raw videos. For trimap IoU the width is 3. No dynamism implies that there are no connections between adjacent frames. cell0 is the cell found by the search on the main set, while cell4 is the cell found by the search using the frames extracted from the video.

<table>
<thead>
<tr>
<th>Method</th>
<th>mIoU, %</th>
<th>mAcc, %</th>
<th>gAcc, %</th>
<th>tIoU, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>per-frame baseline</td>
<td>62.2</td>
<td>73.3</td>
<td>89.9</td>
<td>38.8</td>
</tr>
<tr>
<td>w/ cell0</td>
<td>61.8</td>
<td>69.9</td>
<td>90.4</td>
<td>36.9</td>
</tr>
<tr>
<td>no dynamism</td>
<td>41.0</td>
<td>46.7</td>
<td>83.8</td>
<td>22.6</td>
</tr>
<tr>
<td>w/ cell4</td>
<td>64.1</td>
<td>74.5</td>
<td>90.6</td>
<td>40.3</td>
</tr>
<tr>
<td>no dynamism</td>
<td>39.1</td>
<td>51.3</td>
<td>78.4</td>
<td>23.9</td>
</tr>
</tbody>
</table>

---

Figure 1. Structure of cell4. Orange blocks represent operations and green blocks represent aggregation operations. Numbers inside blocks are operation identifiers as defined in the main text.
import torch
import torch.nn as nn
import torch.nn.functional as F

def resize(x1, x2):
    """Spatially resize two tensors to the largest size among them"""
    if x1.size()[2:] > x2.size()[2:]:
        x2 = nn.Upsample(size=x1.size()[2:], mode='bilinear')(x2)
    elif x1.size()[2:] < x2.size()[2:]:
        x1 = nn.Upsample(size=x2.size()[2:], mode='bilinear')(x1)
    return x1, x2

def conv(C_in, C_out, k, groups=1, stride=1, bias=False):
    return nn.Conv2d(C_in, C_out, k, stride, padding=k // 2, bias=bias, groups=groups)

class ParamSum(nn.Module):
    """ID 0: Summation with per-channel learnable weights per each input.
    Args:
    C (int) : number of input channels.
    """
    def __init__(self, C):
        super(ParamSum, self).__init__()
        self.a = nn.Parameter(torch.ones(C))
        self.b = nn.Parameter(torch.ones(C))

    def forward(self, x, y):
        x, y = resize(x, y)
        return (self.a.expand(x.size(0), -1)[:, :, None, None] * x +
                self.b.expand(y.size(0), -1)[:, :, None, None] * y)

class ConcatReduce(nn.Module):
    """ID 1: Channel-wise concatenation followed by grouped 1x1 convolution.
    Args:
    C (int) : number of input channels (also the number of groups).
    """
    def __init__(self, C):
        super(ConcatReduce, self).__init__()
        self.conv1x1 = nn.Sequential(
            nn.BatchNorm2d(2 * C),
            nn.ReLU(),
            conv(2 * C, C, 1, groups=C))

    def forward(self, x, y):
        x, y = resize(x, y)
        z = torch.cat([x, y], 1)
        return self.conv1x1(z)

Listing 1: Aggregation Operations 0-1.
class PredOP(nn.Module):
    """ID 2: (weight) predictive operation, where
    the first input becomes a set of spatial convolutional
    filters (weights) applied on the second one.
    Args:
      C (int) : number of input channels.
      ksize (int, default=3) : kernel size of the resultant convolution.
    """
    def __init__(self, C, ksize=3):
        super(PredOP, self).__init__()
        self.ksize = ksize
        self.conv = nn.Sequential(
            nn.ReLU(), conv(C, C, 3, groups=C),
            nn.ReLU(), conv(C, C, 3, groups=C),
            nn.ReLU(), conv(C, ksize * ksize, 1), nn.Softmax(dim=1))
    def forward(self, x, y):
        x, y = resize(x, y)
        b, c, h, w = y.size()
        x = (self.conv(x)
            .permute(0, 2, 3, 1)
            .contiguous().view(b, h*w, self.ksize**2, 1))
        p = self.ksize // 2
        cols = F.unfold(
            y, kernel_size=self.ksize, dilation=p, padding=p, stride=1)  # im2col
        out = torch.matmul(
            cols.permute(0, 2, 1).contiguous().view(b, -1, c, self.ksize**2), x)
        out = out.permute(0, 2, 1, 3).contiguous().view(b, c, h, w)
        return out

class BilSampling(nn.Module):
    """ID 3: Bilinear sampling of the first input with the affine grid
    predicted based on the values of the second input.
    Args:
      C (int) : number of input channels.
    """
    def __init__(self, C):
        super(BilSampling, self).__init__()
        self.conv_loc = nn.Sequential(conv(C, 3 * 2, 1), nn.ReLU())
        self.fc_loc = nn.Linear(3 * 2, 3 * 2)
    def forward(self, x, y):
        x, y = resize(x, y)
        yconv = self.conv_loc(y).mean(2).mean(2)
        theta = self.fc_loc(yconv).view(-1, 2, 3)
        grid = F.affine_grid(theta, x.size())
        x = F.grid_sample(x, grid)
        return x + y

Listing 2: Aggregation Operations 2-3.
class Conv3d(nn.Module):
    """ID 4: 3D-convolution, where two inputs are stacked together forming a new dimension with 2x3x3 grouped convolution applied on top.

    Args:
        C (int) : number of input channels (also the number of groups).
        ksize (int, default=3) : kernel size in (2, ksize, ksize) convolution.
    """
    def __init__(self, C, ksize=3):
        super(Conv3d, self).__init__()
        p = int(ksize // 2)
        self.conv = torch.nn.Conv3d(C, C, kernel_size=(2, ksize, ksize), padding=(0, p, p), groups=C, bias=False)

    def forward(self, x, y):
        x, y = resize(x, y)
        return self.conv(torch.stack([x, y], 2)).squeeze(2)

class DenseAttention(nn.Module):
    """ID 5: Element-wise multiplication between the first input and the sigmoid-activated second one.
    """
    def __init__(self):
        super(DenseAttention, self).__init__()

    def forward(self, x, y):
        x, y = resize(x, y)
        return x * F.sigmoid(y)

Listing 3: Aggregation Operations 4-5.
References


