CS 280 Computer Vision
Spring 2018 Efros, Malik, Yu Lecture 21

1 Lecture

Recap

Last time we talked about perceptual organization. We see things as groups, not raw pixel values.

Image Segmentation

Segmentation is highly ambiguous. People have different interpretations of what constitutes a group. Say we represent segmentations hierarchically as **percept trees**, where upper nodes represent general groups and are themselves segmented (e.g. "woman" might be segmented into "head", "skirt," etc. on the next level). While segmenting, *a human might choose to elaborate some of the subtrees and ignore others.*

In the image segmentation task, we would like to recognize contour cues in an image and estimate boundaries from these. Given a dataset, we should extract "ground truth" boundaries (aggregations of human segmentations) and compare these to our generated boundaries during evaluation.

There are two main problems:

1. How do we quantify the difference between predicted segmentations and ground truth segmentations?
2. How do we define and use local contour cues to support boundary estimation?

Evaluating the Discrepancy Between Segmentations

We can compute correspondence via a minimum cost assignment on a bipartite graph. In this way, we only compare the stuff that is actually meant to represent the same thing.

Contour Cues

We can utilize individual gradient features as local contour cues. Under the L*a*b* color space, we can take the **brightness gradient** $BG(x, y, r, \theta)$ (a difference of L* distributions), the **color gradient** $CG(x, y, r, \theta)$ (a difference of a*b* distributions), or the **texture gradient** $TG(x, y, r, \theta)$ (a χ^2 difference of texton histograms; recall that textons are quantized V1-like filter responses). [We're looking at pixel (x, y) in a local window, and we parametrize the detected boundary by the orientation: "is there a boundary of orientation θ in this window?"]

We also have to learn how to combine these cues. Maybe the texture gradient says there's a boundary, while the brightness gradient says there's no boundary. Which do we listen to? We can learn weights for this purpose by fitting against the ground truth labels.

So far, we're only making decisions based on local information.

Region Segmentation

As it stands, we're going from images to boundaries. But labeling every pixel as a boundary doesn't tell us which pixels belong to a segment. How do we turn what we have into pixel (/region) segmentation?

Pairwise pixel affinity: the following distributed cues are useful for determining whether pairs of pixels are in the same region.

- **Proximity:** if pixels are closer together, the chance of them belonging to the same region is higher.
- **Region cue / patch similarity:** the local contexts of the pixels should match.
- **Boundary cue / intervening contour:** boundaries induce region segmentation. If pixels are separated by a boundary, the chance of them belonging to the same region is low.
  - To find an intervening contour: draw a line between the two pixels and register the maximum boundary response encountered while walking along it.

Putting Everything Together

Given an image, we can extract texture, brightness, and color features and then use these for boundary and region processing. In boundary processing, we figure out the contour-ness. In region processing, we compare patch similarities.

Currently, we're calculating pairwise interaction forces. In the end we want to assign one value to every pixel. We can take a graph approach to this, turning every pixel into a vertex and drawing edges between nearby pairs of pixels. For each edge, we'll assign a weight representing how likely the pixels are to belong to the same group. Such a graph summarizes the relationships between pixels.

We can then partition the vertices s.t. similarity within each group is high and similarity between different groups is low. *Segmentation becomes a graph partitioning problem.*

Note: if we have n points, the similarity matrix will be $n \times n$. But it will be very sparse.

To partition the graph, we'd like to cut off the weakest links (i.e. find **cuts**). The cuts should have minimal between-group connections *and* maximal within-group connections. We want to minimize

$$Ncut(A, B) = \frac{cut(A, B)}{vol(A)} + \frac{cut(A, B)}{vol(B)}$$

where $cut(A, B)$ is the number of connections between A and B and $vol(A)$ is the number of connections contained within A. We normalize the cuts by their total volume, meaning *balanced cuts*.

To be clear, the matrix we create is a pairwise affinity matrix (in the slides, it's called W) of size $n \times n$, where $n = (nrows)(ncols)$ (original image dimensions). Row $W[i, :]$ contains pixel i's affinity values across the entire image, and can be reshaped into an $nrows \times ncols$ **affinity pattern** in order to see patterns more openly. The matrix is sparse because we only compute affinity values for nearby pixels.

In total: we take an image, (1) construct the graph affinities $W = W(I, \Theta)$ based on color, texture, etc., (2) compute the eigenvector $X(W)$ for which $WX = \lambda DX$ [this assigns each pixel a value], and (3) threshold each value to get our segmentation.

*This is **spectral graph segmentation**.*

Other Computer Vision Tasks

- **Semantic segmentation:** give a semantic label to every pixel in the image.
- **Classification and localization:** what is it, and where is it (give a bounding box)?
- **Object detection:** what are they, and where are they? The previous, except with multiple objects.
- **Instance segmentation:** do object detection, and give a detailed contour/fill for each object.

To perform semantic segmentation, we can use a fully convolutional network to make predictions for all pixels at once. Note: semantic segmentation is different from instance detection in that it doesn't differentiate between instances (e.g. two cows next to each other are collectively "cow").

In-network upsampling: max unpooling. When max pooling, remember which element in each block was the maximum. When max unpooling, put the current value into the proper max location in each block (and have every other value in the block be 0).

Learnable upsampling: transposed convolution. Otherwise known as deconvolution, this involves a filter that moves `stride` pixels in the output for every pixel in the input. We can represent normal convolution as a matrix multiplication:

$$x * a = Xa$$

$$\begin{bmatrix} x & y & z & 0 & 0 & 0 \\ 0 & x & y & z & 0 & 0 \\ 0 & 0 & x & y & z & 0 \\ 0 & 0 & 0 & x & y & z \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ ax + by + cz \\ bx + cy + dz \\ cx + dy \end{bmatrix}$$

(We have a 1D image $\begin{bmatrix} a & b & c & d \end{bmatrix}$, and a 1D filter $\begin{bmatrix} x & y & z \end{bmatrix}$. The stride is 1 and the padding is 1.)

A *transposed* convolution multiplies the image by the transpose of the same matrix:

$$x *^T a = X^T a$$

$$\begin{bmatrix} x & 0 & 0 & 0 \\ y & x & 0 & 0 \\ z & y & x & 0 \\ 0 & z & y & x \\ 0 & 0 & z & y \\ 0 & 0 & 0 & z \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} ax \\ ay + bx \\ az + by + cx \\ bz + cy + dx \\ cz + dy \\ dz \end{bmatrix}$$

When the stride is 1, transposed convolution is just a regular convolution with different padding rules. When the stride is ≥ 1, transposed convolution is no longer a normal convolution. Below, the first row is normal 1D convolution and the second row is transposed 1D convolution. Both have stride 2.

$$\begin{bmatrix} x & y & z & 0 & 0 & 0 \\ 0 & 0 & x & y & z & 0 \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ bx + cy + dz \end{bmatrix}$$

$$\begin{bmatrix} x & 0 \\ y & 0 \\ z & x \\ 0 & y \\ 0 & z \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} ax \\ ay \\ az + bx \\ by \\ bz \\ 0 \end{bmatrix}$$

Note: the regular convolution takes 6 pixels to 2 pixels, while the transposed convolution takes 2 to 6. transposed convolutions allow us to go from small images to large images, and learn the weights to do so.

While performing classification and localization, we can treat localization as a regression problem. For object detection, we have a variable number of outputs; we could classify a window at all positions, but for computational efficiency we'll probably want to use region proposals instead.

***Mask R-CNN** does instance segmentation, and is the state of the art. It can also do keypoint detection, treating each keypoint as a one-point segmentation mask.*