CS 280 Computer Vision
Spring 2018 Efros, Malik, Yu Lecture 22

1 Lecture

Recap: Segmentation

Prior to deep learning, the approach to contour segmentation was to take local gradients in texture/brightness/color channels ($\rightarrow$ contours), derive pairwise affinities between pixels, and finally use spectral graph partitioning to do contextual (global) integration.

In more detail: pairwise similarities are represented as a matrix W. The eigenvector of W will be a nearly optimal solution to the normalized cuts criterion. (It is a consensus, based on all of the pairwise similarities, about which pixels should go with which.) Then, after discretizing based on eigenvector component values, pixels with similar values ought to belong to the same group.

Recap: Deep Learning

The arrival of deep learning was characterized by applications to image classification. An image would pass through a CNN, end up as a feature vector ("summarizing" the entire image), pass through a fully connected layer, and end up as a bunch of class scores. Under this formulation, the input and output would always have the same dimensions.

Contour Detection

Full-image classification is all good and well, but how do we apply deep learning to contour detection? In this case, the approximated function is (*image*) $\rightarrow$ (*image*), not (*image*) $\rightarrow$ (*fixed number of outputs*). Since convolutional layers can be applied to images of any sizes, convolution will end up being very useful. With convolutional layers, the output is proportional to the input size (as opposed to being fixed).

One of the best-performing deep learning methods for contour detection is *holistically nested edge detection*. The goal of the model is to take an RGB image as input and produce a contour image (for which supervision annotations do exist) as output. To do so, it makes predictions at different scales (using different convolutional blocks), resizes these predictions back to the original resolution (using interpolation), and finally combines the information to make a holistic prediction. Notably, at every stage of the processing, there is a loss for predicting the final output – as opposed to only having a loss at the end. In this way, the goal is "pushed" as early and as often as possible. Such a scheme, in which we impose supervision at all stages, is called "deep supervision."

Note: multi-scale can be handled in many different ways, with many different network structures. Also, this works well because a lot of decisions can already be made at a local scale. We don't need huge fields of view to identify boundaries.

Semantic Segmentation

Unlike contour detection, in semantic segmentation (and object detection, etc.), we need to label each pixel as belonging to one of many categories.

Everything we've looked at so far involves a sliding window classification. We make a decision for a pixel based on a window centered around that pixel. This means that we have to extract a patch around each pixel, feed it into the CNN, and get a category decision for that pixel. *But we have to do this for every pixel in, say, a* 200×200 *image!* ***And*** *at different scales – maybe one window size isn't enough!*

For a 200×200 image, semantic segmentation requires running a CNN for classification $200 \times 200 \times$ (# scales) times! Can't we make the process more efficient? (For one thing, we're not reusing shared features between overlapping patches.)

But of course we can. First of all, we can make our network fully convolutional, so that its output is an image and it predicts the values of all pixels at once. A potential problem is the immense dimensionality of images, but this is mitigated by the fact that we can downsample our images through pooling and strided convolutions, and later upsample the images through unpooling and strided transpose convolutions.

Downsampling is fine because neighboring pixels are highly correlated; we don't need all of them.

Classification and Localization

In this problem, we want our output to be a bounding box indicating the position of a single object in the image. We need to figure out (a) *what* the image contains (cat, dog, car, etc.) and (b) *where* the object is [(x, y) coordinates and (w, h) dimensions]. Since the localization problem involves turning something like a $200 \times 200 \times 3$ image into a 4×1 bounding box output, we should treat it as a regression problem.

Since there is no unique correspondence between feature map pixels and image pixels, is it really possible to regress bounding box coordinates from final-layer features? As it happens, it is.

For a feedforward convNN, each neuron is connected to a small patch of neurons in the previous layer. Accordingly, a receptive field for the first layer is the filter size, but a receptive field (w.r.t. the input image) of a deeper layer depends on all previous layers' filter sizes and strides. We might detect a head or a torso in an earlier filter, which will respond, and then a later filter might respond, and so on. The later neurons will look at entire collections of filter responses in order to make their decisions. *But since we're just composing convolutions, finer position information should be encoded in filter dimensions.* And so there is hope to recover bounding boxes from our feature responses.

If we look at the feature map for a specific channel of a convolutional filter, then the choice of channel will tell us "what" the filter is looking for, and the region of maximum response will tell us "where" the filter sees it.

Object Detection

Object detection is a coarse version of instance segmentation; it only requires a bounding box instead of an exact boundary around each object. It requires us to provide a label and a bounding box for all known objects in the image.

In object detection, the added challenge is that we need to apply a CNN to a massive number of locations and scales (asking "is it a cat, a dog, a background...?"), and it can get pretty expensive. Furthermore, different images might have different numbers of outputs. One answer to these problems is to find blobby image regions that are likely to contain objects, and to propose these regions as candidates for search. In a relatively short amount of time, we can come up with, say, 1000 region proposals.

Then, instead of running the CNN $200 \times 200 \times$ (# scales) times, we can warp each proposed region into a fixed size and only run the CNN on those. For each region, we'll essentially be doing the classification and localization task.

And we can do even better. A lot of the proposed regions will overlap, but we process them as if they're independent. So let's try to reuse the features. We'll compute the features over the entire image only *once*, and then just pull out specific regions of the feature map to get region-based features. In order to turn this into a fixed-length feature (as required by FC layers), we can use spatial pyramid matching (SPM) representations.

Spatial pyramid matching: like a histogram over different spatial regions. (A histogram of features over the entire image would throw away spatial information.)

In the **fast R-CNN (SPP-net)** pipeline, we run the CNN once to compute features for the entire image, then later pick out features corresponding to each area and do classification on those.

Note: currently, the region proposal is still external (some other algorithm gives us the regions). But we've already computed all these features, which again encode potential bounding box information. Why not just do the proposals based on the features themselves? As part of an in-house **region proposal network**, we can classify each pixel as "object" or "not object" and also do box regression relative to each pixel:

- Regress anchor boxes (aka predefined bounding box shapes) at each pixel
- Measure IoU of predicted bounding box overlapped with ground truth bounding box
- Convert into binary output (if IoU > 0.7, the anchor box is "good," otherwise it's "bad")

The RPN is fully convolutional, trained end-to-end, and learns n scores and n boxes (using n anchors) at every location. Adding the RPN to our object detection pipeline gives it the name "**faster R-CNN**."

On the whole, the keys to efficient CNN-based object detection are **feature sharing** [share convolutional feature maps among proposal regions (fast R-CNN) *and* between proposal and detection (faster R-CNN)] and **efficient multi-scale solutions** (e.g. multi-scale anchor boxes).

Instance Segmentation

As part of the **mask R-CNN** scheme, we can pick out a region and do segmentation within that region. Mask R-CNN can also do keypoint detection (e.g. where a head or a shoulder are). In this way it can be used for pose estimation.

Modeling Figure/Ground

A figure is something "thing-like"; it has a shape and is closer to us. By contrast, the ground is "not thing-like" and is farther away.

One of the key concepts in figure/ground organization is **contour ownership**. The figure owns the contour on the boundary between figure and ground. This is important for the perception of shape.

Some cues for figure/ground include *Gestalt principles* (implementation-wise, we can cluster on local shapes and train logistic F/G classifiers on such "shapeme" cues) and *label consistency at junctions* (implementation-wise, we can infer the probability of certain F/G configurations based on data).

Can we use deep learning to segment into figure/ground? First, we can pass an image through a CNN to produce each pixel's affinities with each of its neighbors. Then we can fill the pairwise affinity matrix, do global integration as before, and perform an angular embedding to combine all of our information into a figure/ground segmentation image.

- For the embedding step, we start with pairwise local relationships (the output of normalized cuts).
- As output: a value for each pixel representing a globally consistent embedding of the pairwise affinities.

In angular embedding, things lying on different radial lines belong to different depths.