Fully Convolutional Networks for Semantic Segmentation

1 Citation

Long, Jonathan, Evan Shelhamer, and Trevor Darrell. "Fully convolutional networks for semantic segmentation." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

`https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Long_Fully_Convolutional_Networks_2015_CVPR_paper.pdf`

2 Abstract

We get state of the art on semantic segmentation by training a fully convolutional network that goes from input image to segmentation. To enable this, we use deconvolutional layers to upsample the small feature maps. Since the small feature maps lose information, we have skip connections that combine the small feature maps with previous larger feature maps. This, with the deconvolutional upsampling, produces great segmentations.

3 Introduction

A CNN can be used to make a prediction for every pixel in an image.

We don't use patchwise training, superpixels, or conditional random fields.

The issue with a CNN is that its pooling layers help figure out "what" is in the image, but not "where" it is. We bring back the "where" information using skip connections and deconvolution layers for upsampling.

4 Related Work

AlexNet, VGG, GoogLeNet do great on image classification and transfer learning can apply them to other tasks.

Others have applied fully convolutional networks to other tasks. Some have even done fully convolutional training.

People have densely applied CNNs for segmentation, but they also include other steps like superpixels, patchwise training, image pyramids, and CRFs.

5 Fully Convolutional Networks

The input for semantic segmentation is a $w \times h$ image with d color channels. A fully convolutional network is one that only uses convolution (or deconvolution) layers (we also allow for pooling layers and activation functions). The output is another image (with a different number of channels - one per class). We can then compute a loss:

$$\ell(\mathbf{x};\theta) = \sum_{i,j} \ell^{'}(\mathbf{x}_{i,j};\theta)$$

You could compute this loss (and backpropagate) for patches of the image, but it's more efficient to do it for the whole image.

We take models pretrained on ImageNet (AlexNet, VGG, GoogLeNet) and turn their fully connected layers into convolutional layers. If you apply them densely over an input image, the resulting output feature map is spatially smaller (i.e. smaller width and height) than the input image. For segmentation, we need to produce an output that has the same width/height as the image.

One approach is this. Suppose the output map is a downsampling of the input map by a factor of f. You can shift the input image by x, y for $0 \leq x, y < f$ and interlace the outputs to get the desired $w \times h$ size for the output. However, this increases compute usage by a factor of f^2. So, a trick is to use the a trous (hole) algorithm from the wavelet community. We don't take either approach because they are not that helpful.

The approach we take to fix the output dimension is to use deconvolution layers. Basically, it applies a $1/f$ stride and upsamples the image. Bilinear interpolation is one technique for upsampling, but rather than fixing this kernel, we let the network learn the best upsampling kernels.

Instead of taking image patches and training on those, you can alternatively just train on the entire image and randomly drop pieces of the loss (i.e. drop $\ell^{'}(\mathbf{x}_{i,j};\theta)$ terms for random i, j) This is more efficient, but it doesn't let you correct for imbalanced classes like sampling patches does (i.e. you could oversample positive patches if the positive class is rare), but we don't find that to be a problem if the imbalance is not extreme. In fact, we don't find you don't even need to drop loss terms - just keep them all.

6 Segmentation Architecture

We take our ImageNet classifiers, make them fully convolutional, add upsampling layers, and fine-tune. Then we skip connections and fine-tune again.

Our dataset is PASCAL VOC. Our loss is per-pixel multinomial logistic loss. Our metric is mean pixel intersection over union (mean computed over all classes, including background).

For each ImageNet model, we chop off its classification layer and turn the fully connected layers into convolutional layers. We then add a 1×1 conv layer with 21 classes (one for each PASCAL VOC class and the background) and a deconvolution layer to bilinearly upsample. Then we fine-tune. Interestingly, VGG (not GoogLeNet) does the best here. We call this architecture the FCN-32s. The 32 comes from total downsampling.

Now, let's add a skip connection. Let's first cut the output stride to 16 instead of 32. We then put a 21 1×1 conv layer after $pool_4$ to get per-class predictions. We upsample 2x with a deconvolution layer and then sum with the original feature map before it goes into our fully connected layer (which, if you remember, has been converted into a convolutional layer). After going through the FC layer, we upsample back to original image size. This is FCN-16s. It is initialized with FCN-32s's weights and new weights are set to zero. We then fine-tune with a smaller learning rate. This gives a gain of 3 mean intersection-over-union (IU).

Next we take the same approach with the $pool_3$ layer and get FCN-8s. This gives a little gain too.

Let's talk about some implementation details. We train with SGD with momentum, where learning rates were picked with line search. We used Dropout if the original classification net used it. Fine-tuning is done on the entire network, not just the classification layers. It takes 3 days on a single GPU for FCN-32s and 1 day for skip connection upgrade.

We simulated patch sampling by randomly dropping loss terms, but it didn't help.

About 3/4 of our labels are for background, but we didn't do any correction for imbalanced classes.

The final upsampling is always bilinear, but the others are initialize at bilinear and then tuned.

Data augmentation with mirrors and jittering did not help.

7 Results

Our datasets are PASCAL VOC, NYUDv2, and SIFT Flow.

Let n_{ij} be the number of class i pixels predicted to be class j. Let $t_i = \sum_j n_{ij}$ be the number of class i pixels. Let n_{cl} be the number of classes. We consider these metrics:

$$\text{Pixel Accuracy} = \frac{\sum_i n_{ii}}{\sum_i t_i}$$

$$\text{Mean Accuracy} = \frac{1}{n_{cl}} \sum_i \frac{n_{ii}}{t_i}$$

$$\text{Mean IU} = \frac{1}{n_{cl}} \sum_i \frac{n_{ii}}{t_i + \sum_j n_{ji} - n_{ii}}$$

$$\text{Frequence Weighted IU} = \frac{1}{\sum_k t_k} \sum_i \frac{t_i n_{ii}}{t_i + \sum_j n_{ji} - n_{ii}}$$

On Mean IU, we get a 20% relative improvement over the old state of the art (we get 62.2 on PASCAL VOC 2012) and we can handle 5 frames a second (300x speedup).

NYUDv2 is taken from a Kinect depth sensor, so it also has a depth channel in the image. We find that the depth info does not help much because it's hard to propagate info all the way back through our deep models during fine tuning.

Sift Flow requires two classifications per pixel (a semantic category and a geometric category), so we duplicate our classification components and fine tune the resulting "two headed" network. This was just as good as training two independent models.

8 Conclusion

Fully convolutional networks do great on semantic segmentation if you combine upsampling (via deconvolution) with skip connections.