Fully Convolutional Networks for Semantic Segmentation
(2015 CVPR)

Jonathan Long, Evan Shelhamer, Trevor Darrell - UC Berkeley

Summary

January 22, 2019

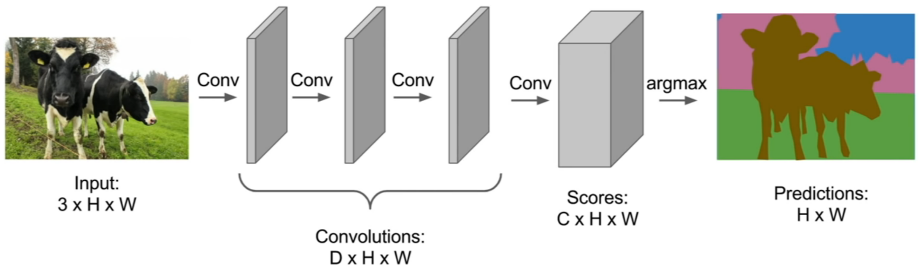
---

1 Introduction

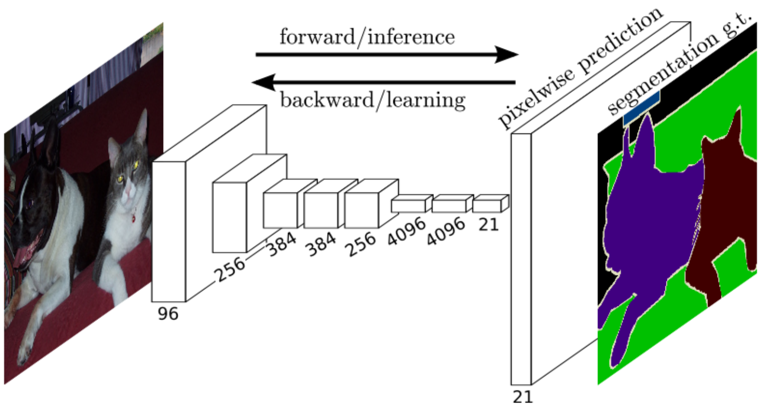
In this work, they propsoe a fully convolutionnal network (no fully connected layers) capable of outputing class predictions in the pixel level (same spatial dimensions as the inputs) using different types of upsampling (stich - transposed convolution) and using skip connections to merge the where (local information - earlier in the network) and what (global information - later in the network) and obtain both coherent pixel predictions and edge information.

2 Why Fully convolutionnal network ?

To be able to predict the class for each pixels of the image, we must obtain an output with the same spatial dimentions as the input, one possible way to do this is to use a convolution network with out any pooling layer nor any strides, and using the necessary pooling to conserve the spatial dimensions, and in the final layer we'll obtain a volume of size H x W x Classes, where H and W are the sizes of the inputs image. The main problem with this approach is the computationnal complexity and force the usage very limited network in terms of the size, one of the main usage of pooling layers is to reduce the number of operations needed in the forward / backward steps.

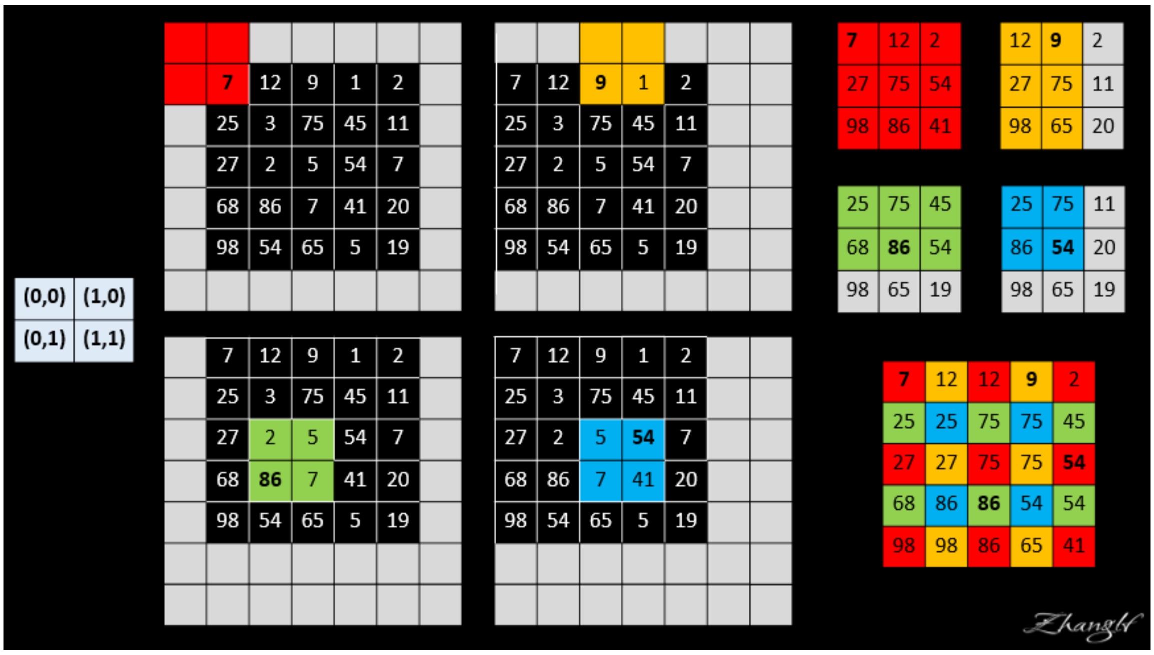


As an alternative, we can have two parts of the netword, one for convolving the inputs into smaller valume, same as the ones used in image classification, and then add a deconvolution / upsampling second part, to obtain a FCN and predict the class predictions for each pixel, and have deeper networks this way.

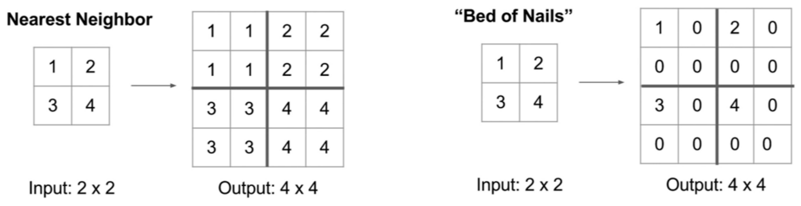


3 Upsampling

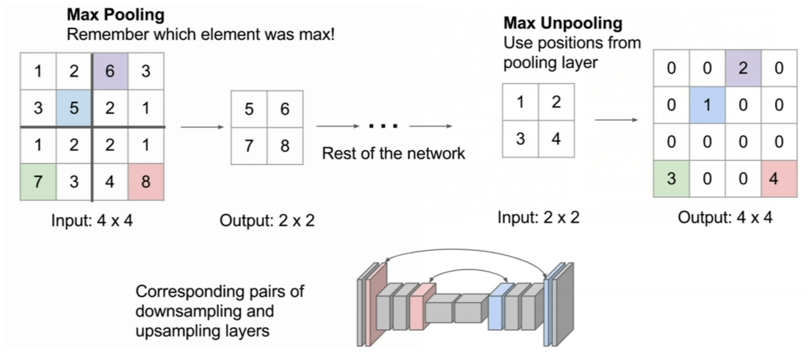
Shift-and-stitch is filter rarefaction One possible way to obtain an ouput of the same size as the input for dense predictions, is by stitching together output from shifted versions of the input. If the output is downsampled by a factor of f, shift the input x pixels to the right and y pixels down, once for every (x, y) s.t. $0 \leq x, y < f$. So with an input of size H, W, and the network downsamples the inputs with a factor of 2, we need to shit the input 4 times, and each time forward it into the network and obtain an output of size H/2, W/2, we then join / stich together the 4 outputs of size (H/2, W/2) to obtain H, W. the stitching is done in a way that the position of the shifting of the input corresponds to its position in the inputs, we could also apply one forward if we constract one inputs that correspongs to all the f^2 shifted version of the original input.



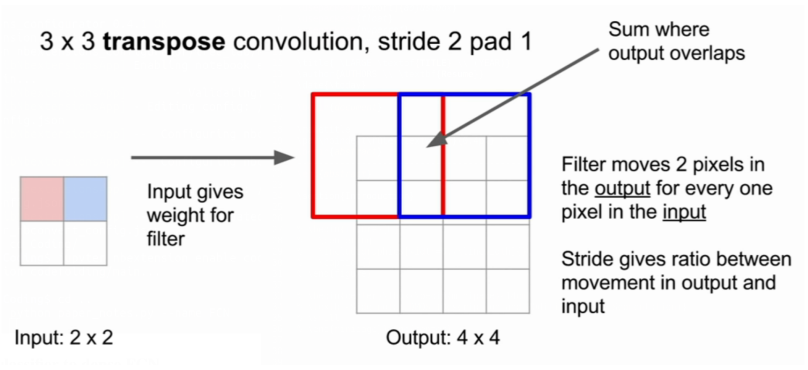
Interpolation / Max unpooling Another possible way to apply unpsampling is two use a form of non parametrized (non learnable parameters) to upsample the outputs, either bu using interpolation (nearest neighboor or bed of nails).



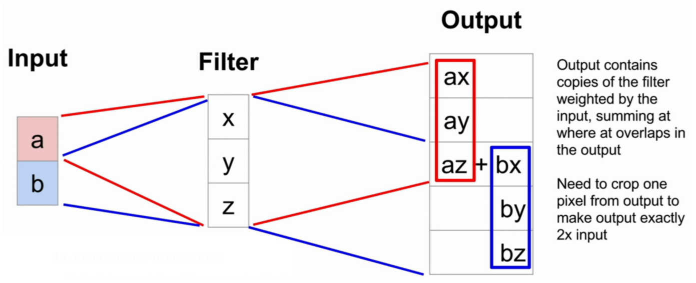
Or max unpooling, in wich we save the indexes the pooling took palce in one place of the network, and copy the values to the same location later on.


Max Pooling
Remember which element was max!
Input: 4 x 4
Output: 2 x 2
Rest of the network
Max Unpooling
Use positions from pooling layer
Input: 2 x 2
Output: 4 x 4
Corresponding pairs of downsampling and upsampling layers

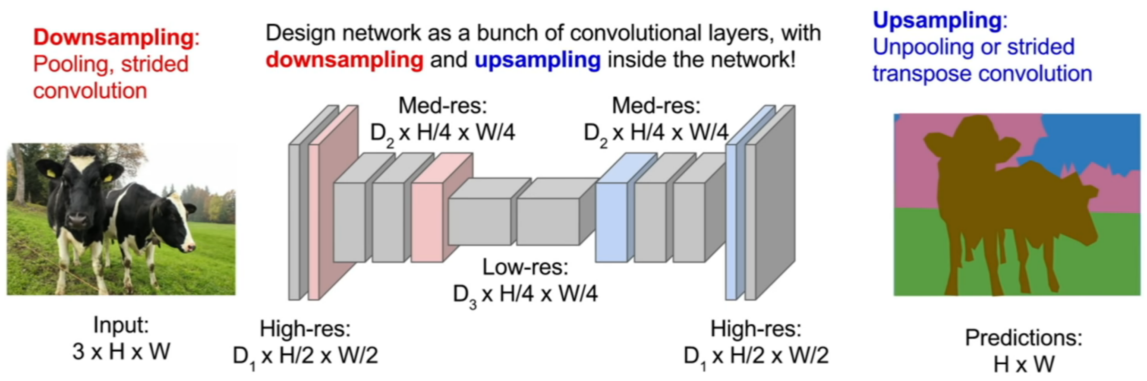
Deconvolution - Backwards strided convolution - Transposed convolution As oposed to applying non learnable upsampling operations, we can also apply learnable filters, but this time to upsample the inputs, in normal convolution, we pass a filter over an image, apply element wise operation and sum the results, so per one convolution of the filter we obtain one values, in transposed convolution, we start from an input, and for each element of the input, we multiply it by a given filter and then create an output, if the results overlaps with an earlier pass, we add them, this might give us different scales of the output and one of the solutions is to use bigger striden to avoid intersections. This is the same operation applied to get the gradient from one layer to the one before it, using the incoming layer (inputs) and the weights (filters)


3 x 3 transpose convolution, stride 2 pad 1
Sum where output overlaps
Input gives weight for filter
Filter moves 2 pixels in the output for every one pixel in the input
Stride gives ratio between movement in output and input
Input: 2 x 2
Output: 4 x 4

Here is an other example in 1D (images from Lecture 11 cs231n)

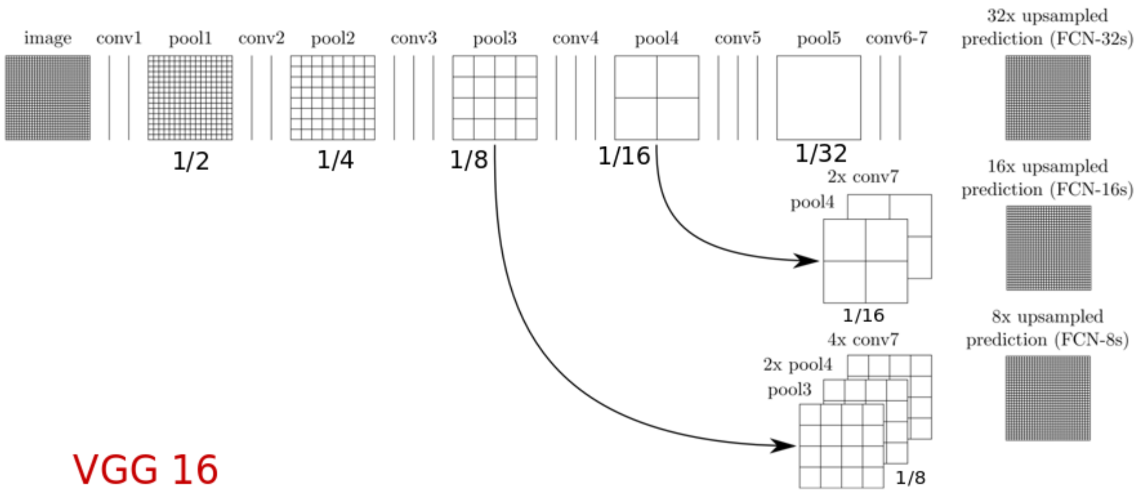

Input
a
b
Filter
x
y
z
Output
ax
ay
az + bx
by
bz
Output contains copies of the filter weighted by the input, summing at where at overlaps in the output
Need to crop one pixel from output to make output exactly 2x input

4 FCN architecture


Downsampling: Pooling, strided convolution
Design network as a bunch of convolutional layers, with downsampling and upsampling inside the network!
Upsampling: Unpooling or strided transpose convolution
Med-res: D_2 x H/4 x W/4
Med-res: D_2 x H/4 x W/4
Low-res: D_3 x H/4 x W/4
Input: 3 x H x W
High-res: D_1 x H/2 x W/2
High-res: D_1 x H/2 x W/2
Predictions: H x W

In the fully convolutionnal network, there is three version:

- FCN 32s: Starting directly from conv7 (convolutionalized fc7), we directly apply an an upsampling x32 with learnable parameters.
- FCN 16s: We first start by upsamling conv7 with a factor of two to obtain volume with the same spatial dimensions as pool4, and then we apply a conv1x1 to reduce the depth of volume at pool4 to match conv7, and we sum the two, and then we apply an upsampling with factor x16 to get to the original input dimensions.
- FCN 8s: We continue in this fashion by fusing predictions from pool3 with a 2 upsampling of predictions fused from pool4 and conv7, building the net FCN-8s.



5 Details and results

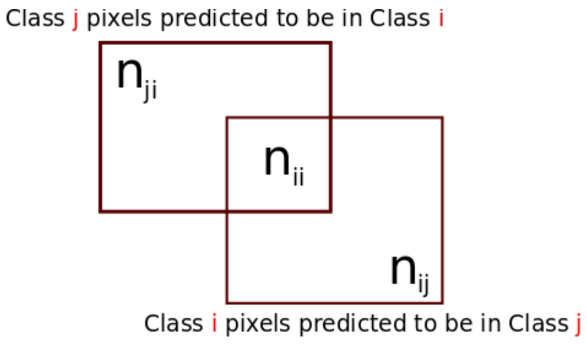
5.1 Details

- Optimization: SGD with momentum. minibatch size of 20 images and fixed learning rates of 10^3, 10^5, and 5^5 for FCN-AlexNet, FCN-VGG16, and FCN-GoogLeNet. We use momentum 0.9, weight decay of 5^4 or 2^4, and doubled learning rate for biases.
- Zero-initialize the class scoring layer
- Dropout was included where used in the original classifier nets.

5.2 Resutlts

Metrics: Let n_{ij} be the number of pixels of class i predicted to belong to class j, where there are n_{cl} different classes, and let $t_i = \sum_j n_{ij}$ be the total number of pixels of class i. The metrics computed are:

- pixel accuracy: $\sum_i n_{ii} / \sum_i t_i$
- mean accuraccy: $(1/n_{\text{cl}}) \sum_i n_{ii}/t_i$
- mean IU: $(1/n_{\text{cl}}) \sum_i n_{ii} / \left(t_i + \sum_j n_{ji} - n_{ii}\right)$
- frequency weighted IU: $(\sum_k t_k)^{-1} \sum_i t_i n_{ii} / (t_i + \sum_j n_{ji} - n_{ii})$


Class j pixels predicted to be in Class i
n_{ji}
n_{ii}
n_{ij}
Class i pixels predicted to be in Class j

	mean IU VOC2011 test	mean IU VOC2012 test	inference time
R-CNN [12]	47.9	-	-
SDS [17]	52.6	51.6	~ 50 s
FCN-8s	**62.7**	**62.2**	~ **175 ms**

6 Implementation details

One important detail in the implementation of the FCN by the original authors is the usage of a padding of 100 in the input, and the reason for this, they state: *Why pad the input?: The 100 pixel input padding guarantees that the network output can be aligned to the input for any input size in the given datasets, for instance PASCAL VOC. The alignment is handled automatically by net specification and the crop layer. It is possible, though less convenient, to calculate the exact offsets necessary and do away with this amount of padding.*

For an input of size 100x100, we'll get the following shapes:

```
INPUT size: torch.Size([1, 3, 100, 100])
pool3 size: torch.Size([1, 256, 38, 38])
pool4 size: torch.Size([1, 512, 19, 19])
pool5 size: torch.Size([1, 512, 10, 10])
score_fr size: torch.Size([1, 7, 4, 4])
upscore2 size: torch.Size([1, 7, 10, 10])
score_pool4 size: torch.Size([1, 7, 19, 19])
upscore_pool4 size: torch.Size([1, 7, 22, 22])
score_pool3 size: torch.Size([1, 7, 38, 38])
upscore8 size: torch.Size([1, 7, 184, 184])
```

After 5 POOLs: (300 = 100+200 — 150 — 75 (one padding) — 38 —- 19 —- 10), we then use a 7x7 conv layer to flatten the volume with a number of channels = 4096, and then two conv 1x1 to get to a number of channels = classes, now without any paddings we would have end up with 1x1xCx1, but with 100 padding the shape is 1x1x4x4,

- first we take 1xCx4x4 and we upsample it with a deconvolution with a filter of size 4 and stride 2, so 4x4 becomes 10x10,
- we convolve pool 4, to have the correct number of channels to add them with 1xCx4x4,
- we add pool4: 1xCx19x19 with the ouputs: 1xCx10x10, so we only choose pool4[5:15],
- now we need to use pool3, so we convolve it to get the correct number of channles,
- and then we add pool3 : 1xCx38x38, to the upsampled version of the addition of pool4 and the ouputs : 1xCx38x38 + 1xCx22x22, we only take the elements of pool3 [9:31].
- , and we finnaly upsample the whole x8, using a kernel with size 16 and stride 8, 22*8 + 8 = 184.