Fully convolutional models and semantic segmentation

Transposed Convolutions

Transposed Convolutions help in upsampling the previous layer to a higher resolution or dimension. Upsampling is a classic signal processing technique which is often accompanied by interpolation. The term transposed can be confusing since we typically think of transposing as changing places, such as switching rows and columns of a matrix. In this case when we use the term transpose, we mean transfer to a different place or context.

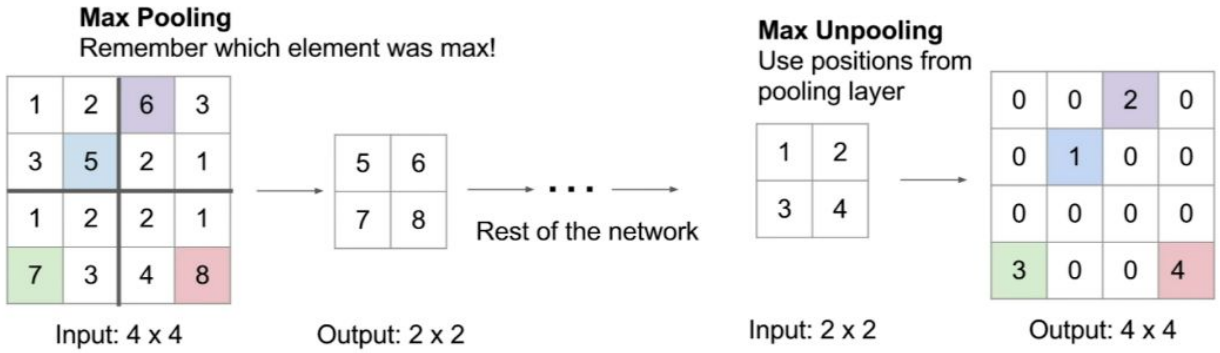
We can use a transposed convolution to transfer patches of data onto a sparse matrix, then we can fill the sparse area of the matrix based on the transferred information. Helpful animations of convolutional operations, including transposed convolutions, can be found here. As an example, suppose you have a 3x3 input and you wish to upsample that to the desired dimension of 6x6. The process involves multiplying each pixel of your input with a kernel or filter. If this filter was of size 5x5, the output of this operation will be a weighted kernel of size 5x5. This weighted kernel then defines your output layer.

Methods of upsampling

1. Max "unpooling"

2. Learnable upsampling (with transposed convolutions)

The first is pictured below.



Learnable upsampling, transposed convolutions

We've been going over learnable upsampling layers. The upsampling part of the process is defined by the strides and the padding. Let's look at a simple representation of this. If we have a 2x2 input and a 3x3 kernel and a stride of 2, we can expect an output of dimension 4x4. The following image gives an idea of the process.

Semantic Segmentation

In semantic segmentation, we want to input an image into a neural network and we want to output a category for **every pixel** in this image. For example, for the below

image of a couple of cows, we want to look at every pixel and decide: is that pixel part of a cow, the grass or sky, or some other category?

Small image of cows in a field.

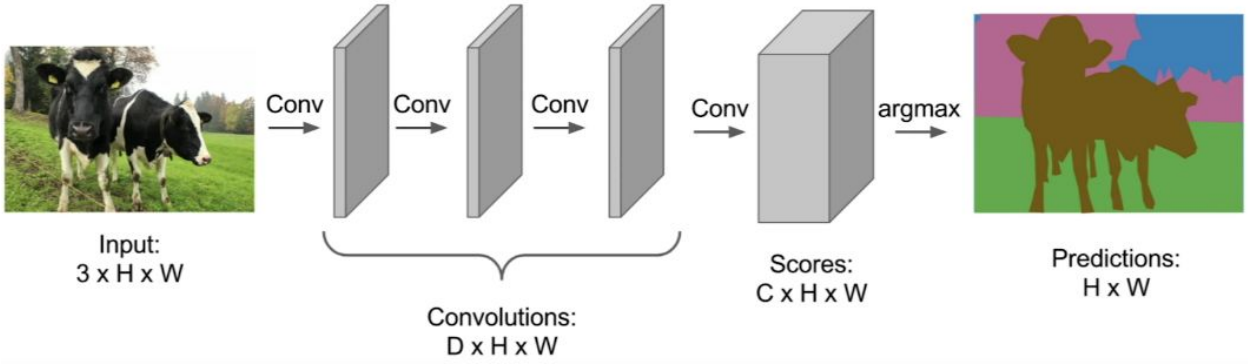
We'll have a discrete set of categories, much like in a classification task. But instead of assigning a single class to an image, we want to assign a class to every pixel in that image. So, how do we approach this task?

Fully-Convolutional Network (FCN) Approach

If your goal is to preserve the spatial information in our original input image, you might think about the simplest solution: simply never discard any of that information; never downsample/maxpool and don't add a fully-connected layer at the end of the network.

We could use a network made entirely of convolutional layers to do this, something called a fully convolutional neural network. **A fully convolutional neural network preserves spatial information.**

This network would take in an image that has true labels attached to each pixel, so every pixel is labeled as grass or cat or sky, and so on. Then we pass that input through a stack of convolutional layers that preserve the spatial size of the input (something like 3x3 kernels with zero padding). Then, the final convolutional layer outputs a tensor that has dimensions CxHxW, where C is the number of categories we have.



FCN architecture.

Predictions

This output Tensor of predictions contains values that classify every pixel in the image, so if we look at a single pixel in this output, we would see a vector that looks like a classification vector -- with values in it that show the probability of this single pixel being a cat or grass or sky, and so on. We could do this pixel level classification all at once, and then train this network by assigning a loss function to each pixel in the image and doing backpropagation as usual. So, if the network makes an error and classifies a single pixel incorrectly, it will go back and adjust the weights in the convolutional layers until that error is reduced.

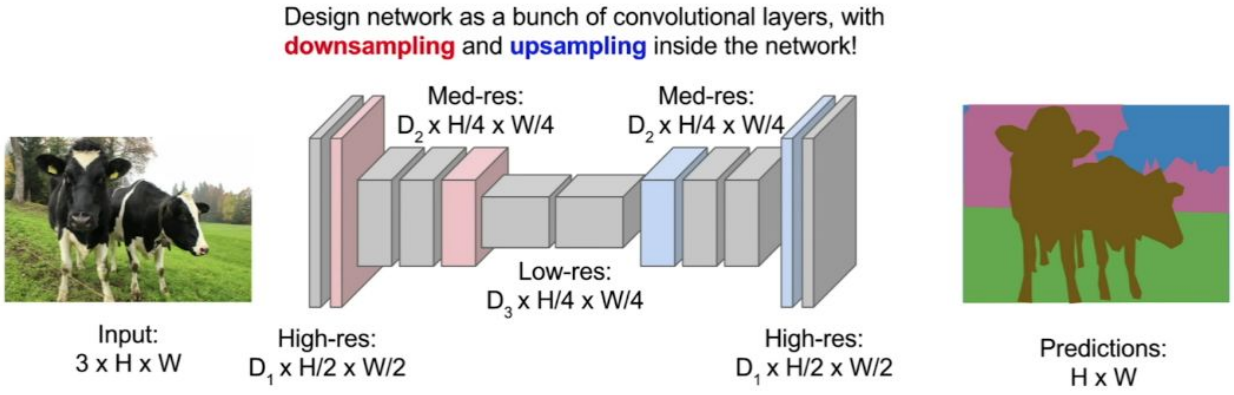
Limitations of This Approach

- It's very expensive to label this data (you have to label every pixel), and
- It's computationally expensive to maintain spatial information in each convolutional layer

So...

Downsampling/Upsampling

Instead, what you usually see is an architecture that uses downsampling of a feature map and an upsampling layer to reduce the dimensionality and, therefore, the computational cost, of moving forward through these layers in the middle of the network.



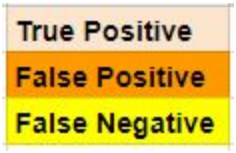
Downsampling/upsampling an an FCN.

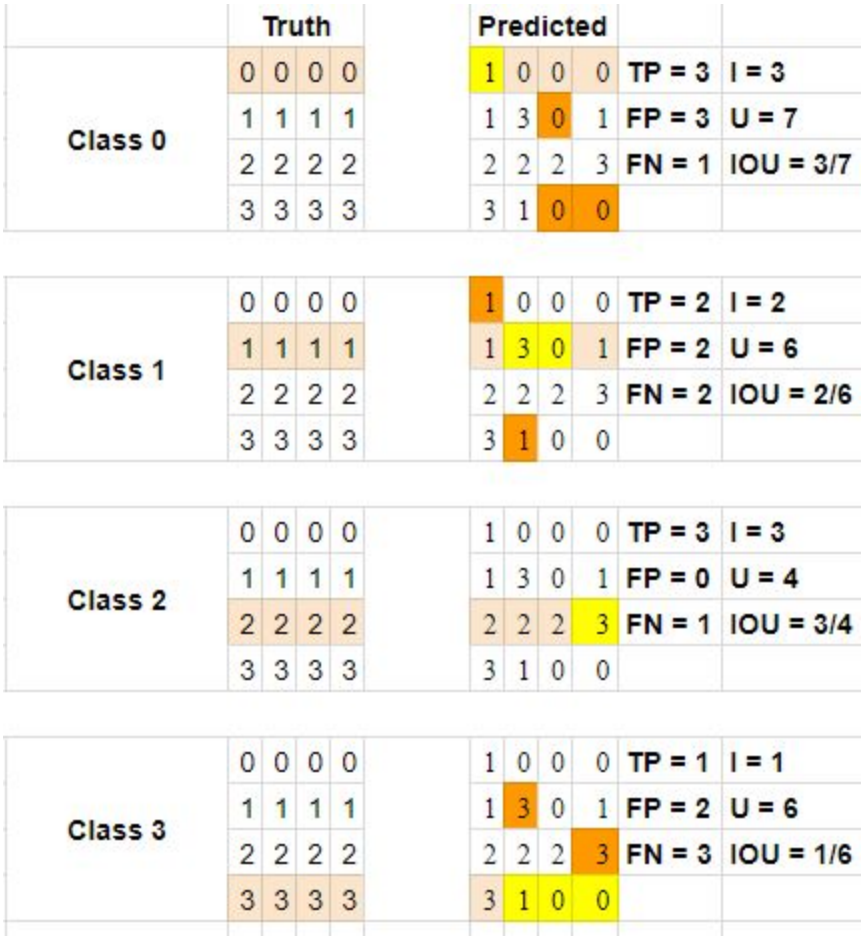
So, what you'll see in these networks is a couple of convolutional layers followed by downsampling done by something like maxpooling, very similar to a simple image classification network. Only, this time, in the second half of the network, we want to increase the spatial resolution, so that our output is the same size as the input image, with a label for every pixel in the original image.

Let's walk through an example IOU calculation.

Steps

- count true positives (TP)
- count false positives (FP)
- count false negatives (FN)
- Intersection = TP
- Union = TP + FP + FN
- IOU = Intersection/Union





In the above, the left side is our ground truth, while the right side contains our predictions. The highlighted cells on the left side note which class we are looking at for statistics on the right side. The highlights on the right side note true positives in a cream color, false positives in orange, and false negatives in yellow (note that all others are true negatives - they are predicted as this individual class, and should not be based on the ground truth).

We'll look at the first class, Class 0, and you can do the same calculations from there for each.

For Class 0, only the top row of the 4x4 matrix should be predicted as zeros. This is a rather simplified version of a real ground truth - in reality, the zeros could be anywhere in the matrix. On the right side, we see 1,0,0,0, meaning the first is a false negative, but the other three are true positives (aka 3 for Intersection as well). From there, we need to find anywhere else where zero was falsely predicted, and we note that happens once on the second row, and twice on the fourth row, for a total of three false positives.

To get the Union, we add up TP (3), FP (3) and FN (1) to get seven. The IOU for this class, therefore, is 3/7.

If we do this for all the classes and average the IOUs, we get:

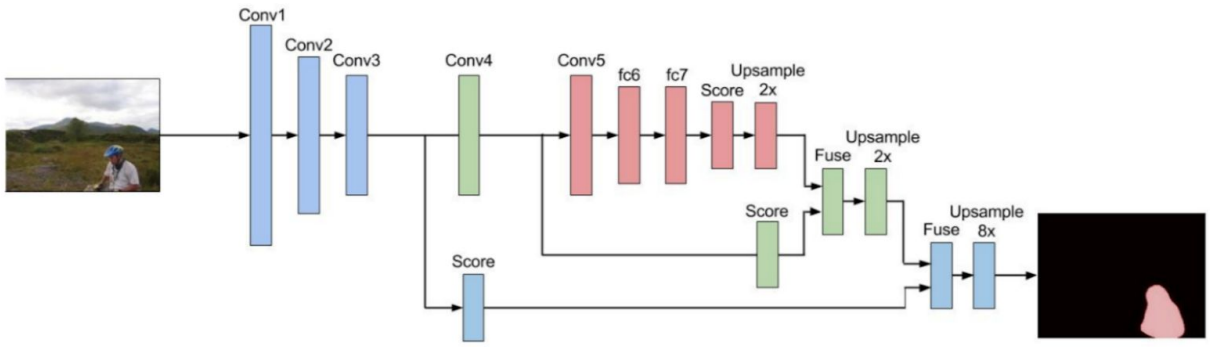
Mean IOU = [(3/7) + (2/6) + (3/4) + (1/6)] / 4 = 0.420

FCN-8, Architecture

Let's focus on a concrete implementation of a fully convolutional network. We'll discuss the FCN-8 architecture developed at Berkeley. In fact, many FCN models are derived from this FCN-8 implementation. The encoder for FCN-8 is the VGG16 model pretrained on ImageNet for classification. The fully-connected "score" layers are replaced by 1-by-1 convolutions. The code for convolutional score layer like looks like:

```
self.score = nn.Conv2d(input_size, num_classes, 1)
```

The complete architecture is pictured below.



There are skip connections and various upsampling layers to keep track of a variety of spatial information. In practice this network is often broken down into the **encoder** portion with parameters from VGG net, and a decoder portion, which includes the upsampling layers.

FCN-8, Decoder Portion

To build the decoder portion of FCN-8, we'll upsample the input, that comes out of the convolutional layers taken from VGG net, to the original image size. The shape of the tensor after the final convolutional transpose layer will be 4-dimensional: (batch_size, original_height, original_width, num_classes).

To define a transposed convolutional layer for upsampling, we can write the following code, where 3 is the size of the convolving kernel:

```
self.transpose = nn.ConvTranspose2d(input_depth, num_classes, 3, stride=2)
```

The transpose convolutional layers increase the height and width dimensions of the 4D input Tensor. And you can look at the PyTorch documentation, here.

Skip Connections

The final step is adding skip connections to the model. In order to do this we'll combine the output of two layers. The first output is the output of the current layer. The second output is the output of a layer further back in the network, typically a pooling layer.

In the following example we combine the result of the previous layer with the result of the 4th pooling layer through elementwise addition (`tf.add`).

```
# the shapes of these two layers must be the same to add them
input = input + pool_4
```

We can then follow this with a call to our transpose layer.

```
input = self.transpose(input)
```

We can repeat this process, upsampling and creating the necssary skip connections, until the network is complete!

FCN-8, Classification & Loss

The final step is to define a loss. That way, we can approach training a FCN just like we would approach training a normal classification CNN.

In the case of a FCN, the goal is to assign each pixel to the appropriate class. We already happen to know a great loss function for this setup, cross entropy loss!

Remember the output tensor is 4D so before we perform the loss, we have to reshape it to 2D, where each row represents a pixel and each column a class. From here we can just use standard cross entropy loss.

That's it, we now have an idea of how to create an end-to-end model for semantic segmentation. Check out the original paper to read more specifics about FCN-8.

Supporting Materials

FCN8 Paper