Understanding the U-Net Architecture with Code

The U-Net is made up of contracting path (downsampling), bottleneck (middle part) and expansive path (upsampling).

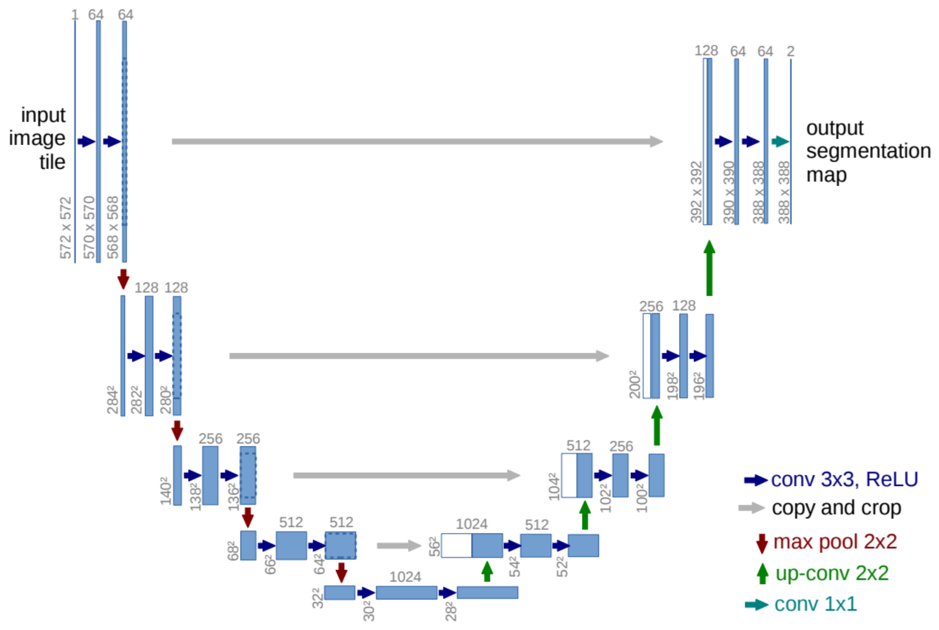

Figure 1: U-Net Architecture

Contents

1 Contracting Path (Downsampling)

The contracting path captures the context of the input image by applying repeated convolution and pooling operations. Each step in the contracting path contains:

- **Blue arrow**: Two 3×3 convolutional layers (with ReLU activation after each convolution layer).
- **Red arrow**: A 2×2 max pooling layer with stride 2 for downsampling.

1.1 Blue arrow: Convolutional block

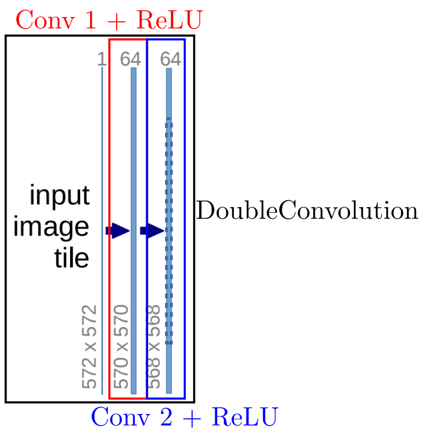

Figure 2: Double Convolutional Block

The double convolution block can be defined as:

```python
class DoubleConvolution(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv_block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.conv_block(x)
```

Explanation of the code:

- The padding of 1 ensures that the spatial dimensions remain the same after convolution. Though in the original paper Ronneberger, Fischer, and Brox 2015, there was no padding; we can see this in figure 1 where we go from 572×572 to 570×570 by the first convolution with kernel size 3×3, which makes sense.
- The `inplace=True` argument in ReLU allows for memory optimization by modifying the input directly (instead of allocating memory).

1.2 Red arrow: Maxpooling

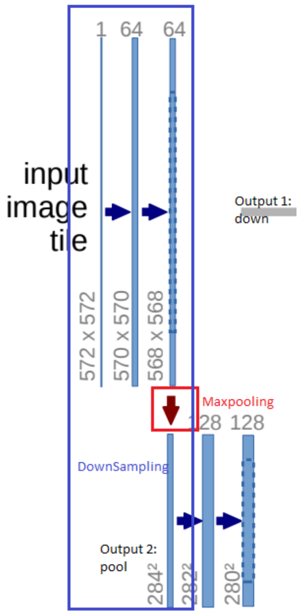

Figure 3: Downsampling Block

The downsampling operation can be implemented as:

```python
class DownSampling(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = DoubleConvolution(in_channels, out_channels)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        down = self.conv(x)
        pool = self.pool(down)

        return down, pool
```

Explanation of the code:

- We apply a single `DoubleConvolution` block to the input, and then we apply the maxpooling operation on the result, which is shown in figure 3.
- We return both the output of the `DoubleConvolution` block which will be used for the skip connection and the output of the maxpooling operation which will be used for the next downsampling block.

2 Bottleneck

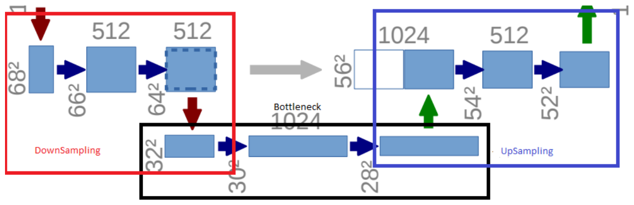

Figure 4: Bottleneck Block

The bottleneck block is also `DoubleConvolution`, without maxpooling. This stage learns the most abstract features before upsampling begins.

3 Expansive Path (Upsampling)

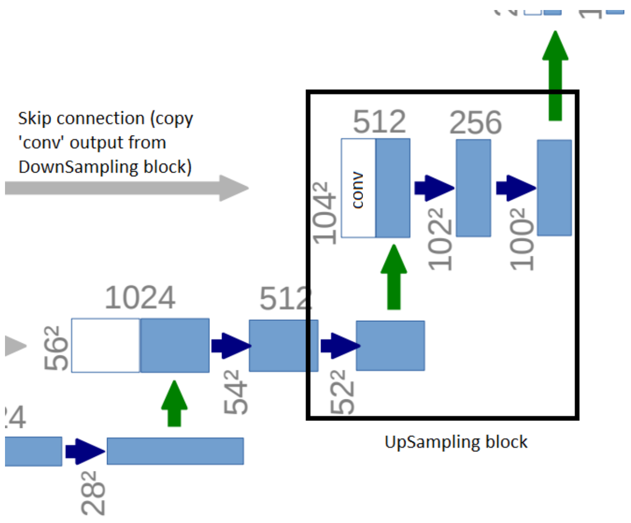
Skip connection (copy 'conv' output from DownSampling block)
512
256
conv
104²
102²
100²
1024
512
56²
54²
52²
28²
UpSampling block

Figure 5: UpSampling Block

The `UpSampling` block can be implemented as:

```python
class UpSampling(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.up = nn.ConvTranspose2d(in_channels, in_channels//2, kernel_size=2, stride=2)
        self.conv = DoubleConvolution(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        x = torch.cat((x1, x2), dim=1)
        return self.conv(x)
```

Explanation of the code:

- The `ConvTranspose2d` layer is used for upsampling. Its a 2×2 deconvolution (reverse of regular convolution - it increases spatial resolution).

- The output of the `ConvTranspose2d` layer is concatenated with the corresponding feature map from the contracting path (the skip connection). i.e. this is the output `conv` of the `DownSampling` block.
- Finally, `DoubleConvolution`

4 U-Net: Skip connections & Final layer

The skip connections are implemented by concatenating the feature maps from the contracting path with the corresponding feature maps in the expansive path. This allows the network to retain spatial information lost during downsampling. It is not `ResidualBlock` connections, where we add the feature maps: instead we concat them.

Here is the final U-Net coming together:

```
1  class UNet(nn.Module):
2      def __init__(self, in_channels, num_classes):
3          super().__init__()
4          self.dconv1 = DownSampling(in_channels, 64)
5          self.dconv2 = DownSampling(64, 128)
6          self.dconv3 = DownSampling(128, 256)
7          self.dconv4 = DownSampling(256, 512)
8
9          self.bottle_neck = DoubleConvolution(512, 1024)
10
11         self.uconv1 = UpSampling(1024, 512)
12         self.uconv2 = UpSampling(512, 256)
13         self.uconv3 = UpSampling(256, 128)
14         self.uconv4 = UpSampling(128, 64)
15
16         self.out = nn.Conv2d(64, num_classes, kernel_size=1)
17
18     def forward(self, x):
19         d1,p1 = self.dconv1(x)
20         d2,p2 = self.dconv2(p1)
21         d3,p3 = self.dconv3(p2)
22         d4,p4 = self.dconv4(p3)
23
24         b = self.bottle_neck(p4)
25
26         u1 = self.uconv1(b, d4)
27         u2 = self.uconv2(u1, d3)
28         u3 = self.uconv3(u2, d2)
29         u4 = self.uconv4(u3, d1)
30
31         out = self.out(u4)
32         return out
```

Explanation of the code:

- Line 16: the final output layer is a 1×1 convolution that reduces the number of feature maps to the number of classes in the segmentation task.
- Line 19-22: the downsampling blocks. We get dX and pX where dX is the output of the `DoubleConvolution` block and pX is the output of the maxpooling operation. The pX is used for the next downsampling block and dX is used for the skip connection (lines 26-29).
- Line 24: the bottleneck block.

- Line 26-29: the upsampling blocks. We get uX which is the output of the `UpSampling` block. The input is the dX which we talked about before (the skip connection), and the output is the uX which is used for the next upsampling block.
- Line 31: the final output layer. The output is the number of classes in the segmentation task.