Understanding the U-Net Architecture with Code

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

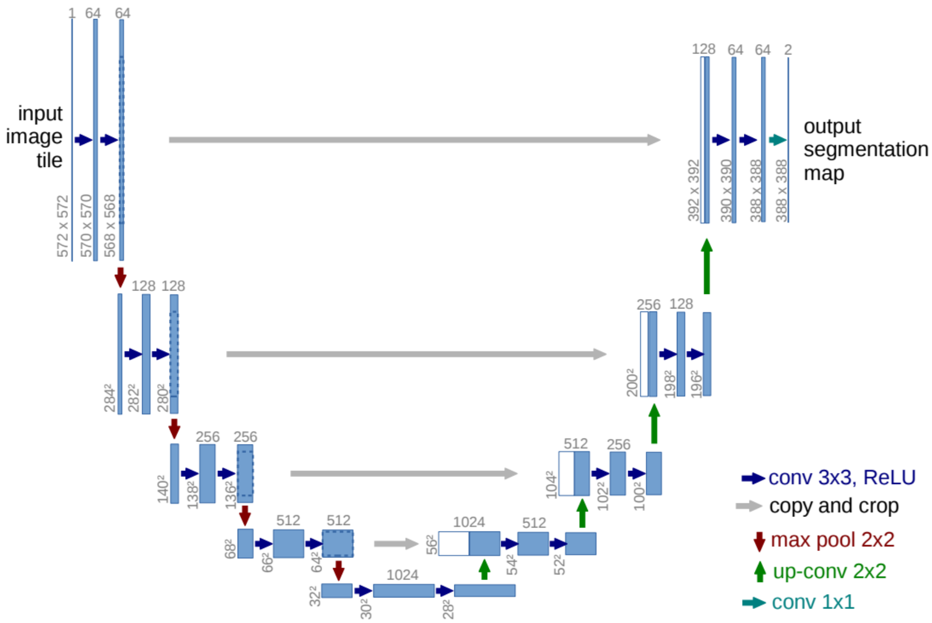

Figure 1: U-Net Architecture

Contents

1 Contracting Path (Downsampling) **2**

1.1 Blue arrow: Convolutional Block 2

1.2 Red arrow: Maxpooling 2

2 Bottleneck **3**

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

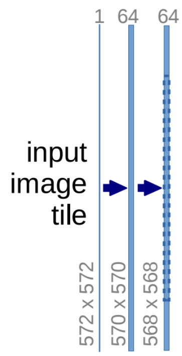

Figure 2: Double Convolutional Block

The double convolution block can be defined as:

```
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

The downsampling operation can be implemented as:

```
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:

- The `down` feature map is retained for later use in skip connections.
- The `pool` is forwarded deeper into the network.

2 Bottleneck

At the bottom of the U, the network contains two 3×3 convolutions without pooling. This stage learns the most abstract features before upsampling begins.

3. Expansive Path (Upsampling)

The expansive path reconstructs the segmentation map through upsampling and concatenation with features from the contracting path. Each step includes:

- A 2×2 transposed convolution for upsampling.
- Concatenation with the corresponding feature map from the contracting path.
- Two 3×3 convolutions (with ReLU activation).

The implementation of an upsampling block:

```
class UpSampling(nn.Module):
    def __init__(self, in_channels, out_channels):
        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)
```

4. Skip Connections

Skip connections play a critical role in U-Net. They allow high-resolution features from the encoder to be reused in the decoder, which helps retain fine-grained spatial information lost during pooling. They are implemented by concatenating encoder features with the decoder's upsampled output:

$$x = \mathrm{Concat}(\mathrm{Upsample}(x_{\mathrm{decoder}}), x_{\mathrm{encoder}}) \tag{1}$$

This helps the network make better predictions, especially near object boundaries.

5. Final Output

The final layer is typically a 1×1 convolution to reduce the number of feature maps to the number of classes in the segmentation task.

```
self.out = nn.Conv2d(64, num_classes, kernel_size=1)
```

Conclusion

U-Net's strength lies in its symmetric structure and use of skip connections. This design enables the network to combine both high-level abstract features and low-level spatial information, making it highly effective for segmentation tasks with limited data.