U-Net原理分析与代码解读

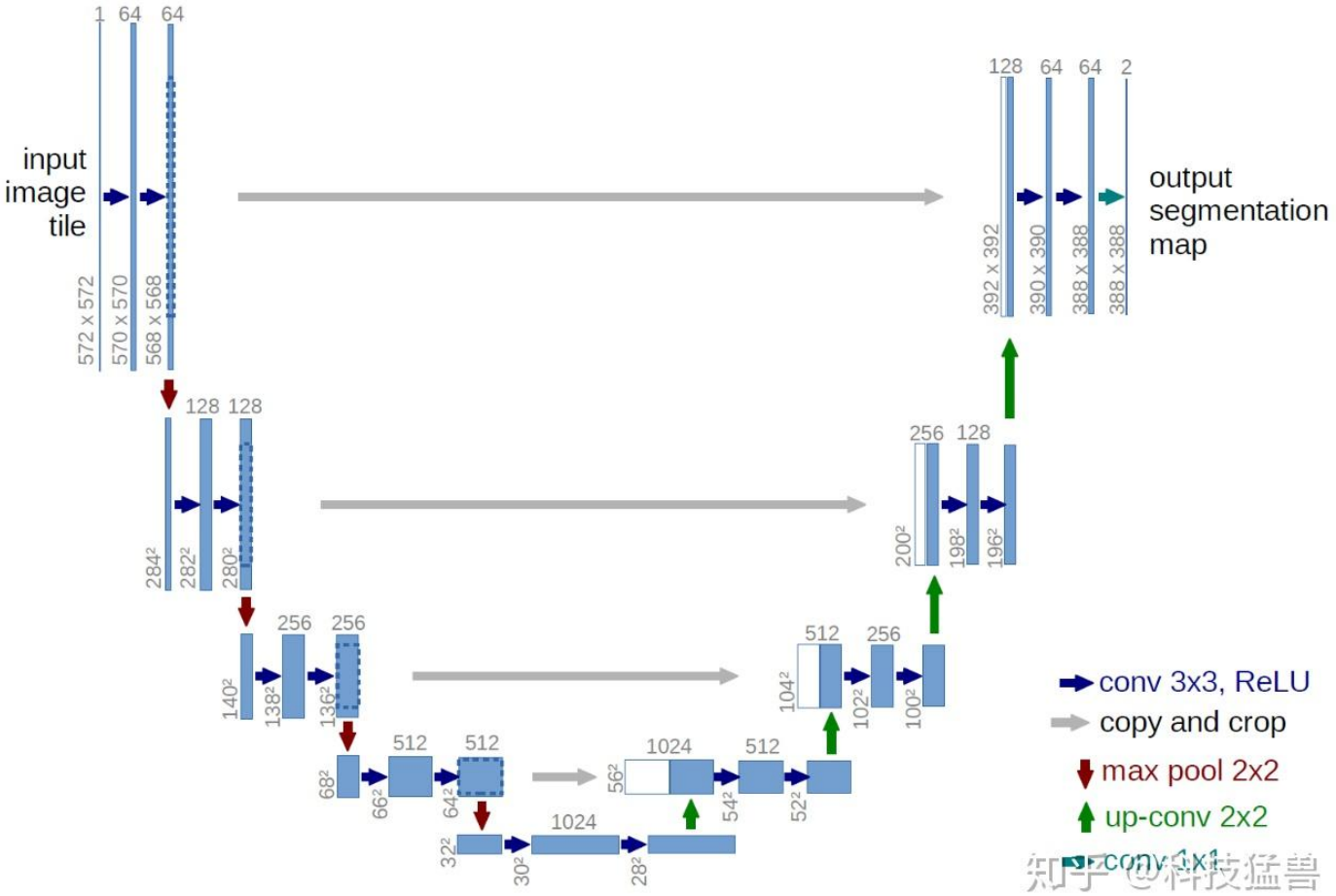
Unet 背景介绍：

Unet 发表于 2015 年，属于 FCN 的一种变体。Unet 的初衷是为了解决生物医学图像方面的问题，由于效果确实很好后来也被广泛的应用在语义分割的各个方向，比如卫星图像分割，工业瑕疵检测等。

Unet 跟 FCN 都是 Encoder-Decoder 结构，结构简单但很有效。Encoder 负责特征提取，你可以将自己熟悉的各种特征提取网络放在这个位置。由于在医学方面，样本收集较为困难，作者为了解决这个问题，应用了图像增强的方法，在数据集有限的情况下获得了不错的精度。

Unet 网络结构与细节

- **Encoder**



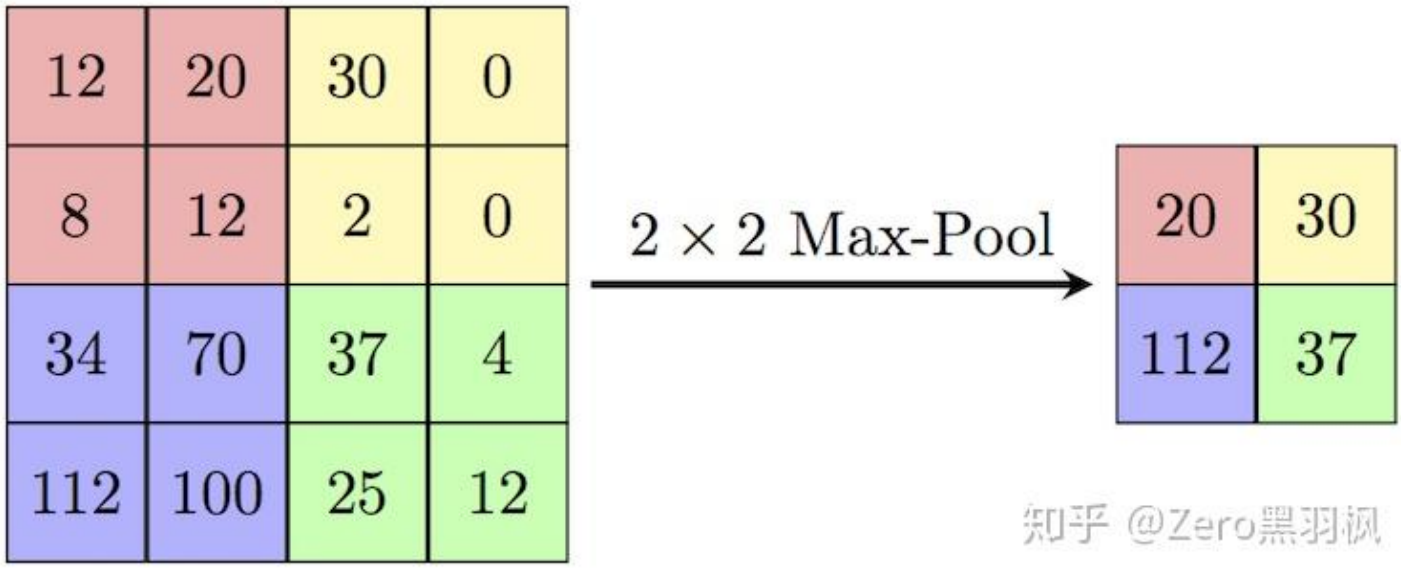
如上图，Unet 网络结构是对称的，形似英文字母 U 所以被称为 Unet。整张图都是由蓝/白色框与各种颜色的箭头组成，其中，**蓝/白色框表示 feature map；蓝色箭头表示 3x3 卷积，用于特征提取；灰色箭头表示 skip-connection，用于特征融合；红色箭头表示池化 pooling，用于降低维度；绿色箭头表示上采样 upsample，用于恢复维度；青色箭头表示 1x1 卷积，用于输出结果** 。其中灰色箭头`copy and crop`中的`copy`就是`concatenate`而`crop`是为了让两者的长宽一致

可能你会问为啥是 5 层而不是 4 层或者 6 层，emmm，这应该去问作者本人，可能对于当时作者拿到的数据集来说，这个层数的表现更好，但不代表所有的数据集这个结构都适合。我们该多关注这种 Encoder-Decoder 的设计思想，具体实现则应该因数据集而异。

Encoder 由卷积操作和下采样操作组成，文中所用的卷积结构统一为 **3x3 的卷积核，padding 为 0 ，striding 为 1** 。没有 padding 所以每次卷积之后 feature map 的 H 和 W 变小了，在 skip-connection 时要注意 feature map 的维度(其实也可以将 padding 设置为 1 避免维度不对应问题)，pytorch 代码：

```
nn.Sequential(nn.Conv2d(in_channels, out_channels, 3),
                nn.BatchNorm2d(out_channels),
                nn.ReLU(inplace=True))
```

上述的两次卷积之后是一个 **stride 为 2 的 max pooling** ，输出大小变为 1/2 *(H, W)：



pytorch 代码：

```
nn.MaxPool2d(kernel_size=2, stride=2)
```

上面的步骤重复 5 次，最后一次没有 max-pooling，直接将得到的 feature map 送入 Decoder。

- **Decoder**

feature map 经过 Decoder 恢复原始分辨率，该过程除了卷积比较关键的步骤就是 upsampling 与 skip-connection。

Upsampling 上采样常用的方式有两种：1.**FCN 中介绍的反卷积**；2. **插值** 。这里介绍文中使用的插值方式。在插值实现方式中，bilinear 双线性插值的综合表现较好也较为常见 。

双线性插值的计算过程没有需要学习的参数，实际就是套公式，这里举个例子方便大家理解(例子介绍的是参数 align_corners 为 Fasle 的情况)。

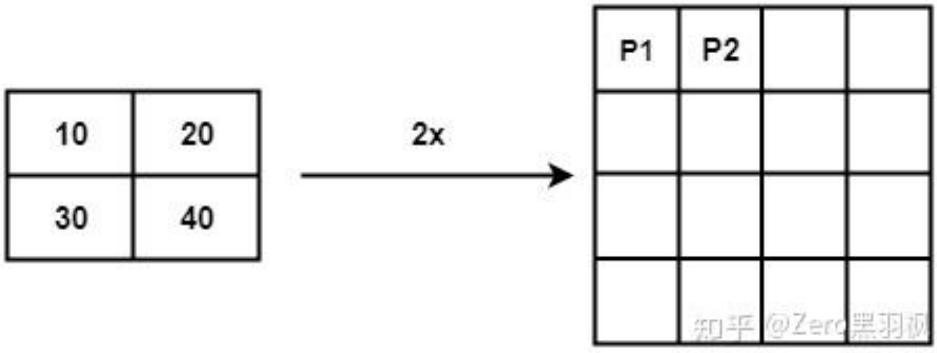
```
In [1]: import torch
        import numpy as np

        src = torch.Tensor(np.asarray([[[[10, 20], [30, 40]]]]))
        print("src: ")
        print(src)
        upsample = torch.nn.Upsample(scale_factor=2, mode="bilinear", align_corners=False)
        print("dst:")
        print(upsample(src))
```

2x2 ↓ 4x4

```
src:
tensor([[[[10., 20.],
          [30., 40.]]]])
dst:
tensor([[[[10.0000, 12.5000, 17.5000, 20.0000],
          [15.0000, 17.5000, 22.5000, 25.0000],
          [25.0000, 27.5000, 32.5000, 35.0000],
          [30.0000, 32.5000, 37.5000, 40.0000]]]])
```

知乎 @Zero黑羽枫

例子中是将一个 2x2 的矩阵通过插值的方式得到 4x4 的矩阵，那么将 2x2 的矩阵称为源矩阵，4x4 的矩阵称为目标矩阵。双线性插值中，目标点的值是由离他最近的 4 个点的值计算得到的，我们首先介绍如何找到目标点周围的 4 个点，以 P2 为例。



第一个公式，目标矩阵到源矩阵的坐标映射：

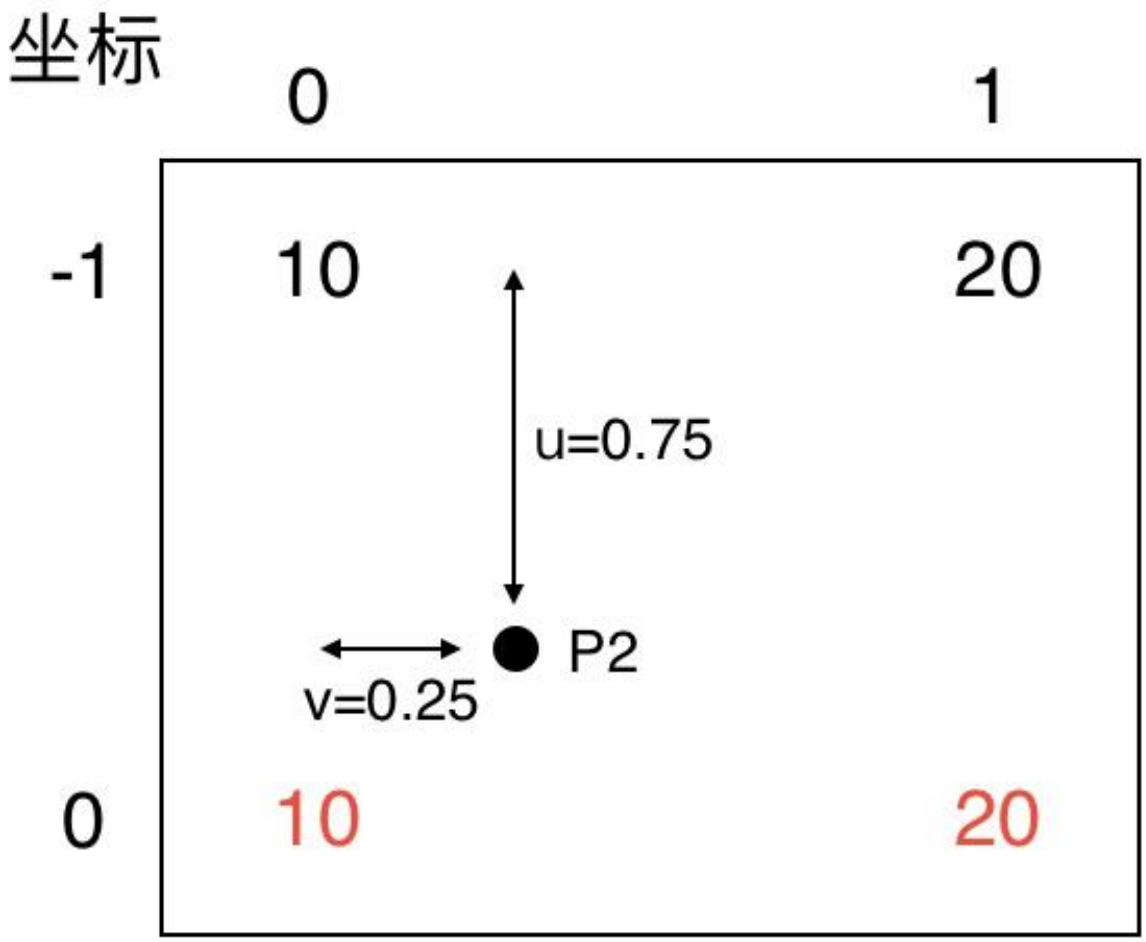
$$X_{src} = (X_{dst} + 0.5) * (\frac{Width_{src}}{Width_{dst}}) - 0.5 \qquad (1)$$

$$Y_{src} = (Y_{dst} + 0.5) * (\frac{Height_{src}}{Height_{dst}}) - 0.5 \qquad (2)$$

为了找到那 4 个点，首先要找到目标点在源矩阵中的**相对位置**，上面的公式就是用来算这个的。P2 在目标矩阵中的坐标是 (0, 1)，对应到源矩阵中的坐标就是 (-0.25, 0.25)。坐标里面居然有小数跟负数，不急我们一个一个来处理。我们知道双线性插值是从坐标周围的 4 个点来计算该坐标的值，(-0.25, 0.25) 这个点周围的 4 个点是(-1, 0), (-1, 1), (0, 0), (0, 1)。为了找到负数坐标点，我们将源矩阵扩展为下面的形式，中间红色的部分为源矩阵。

10	10	20	20
10	10	20	20
30	30	40	40
30	30	40	40

我们规定 f(i, j) 表示 (i, j)坐标点处的像素值，对于计算出来的对应的坐标，我们统一写成 (i+u, j+v) 的形式。那么这时 i=-1, u=0.75, j=0, v=0.25。把这 4 个点单独画出来，可以看到目标点 P2 对应到源矩阵中的**相对位置** 。



知乎 @Zero黑羽枫

第二个公式，也是最后一个。

f(i + u, j + v) = (1 - u) (1 - v) f(i, j) + (1 - u) v f(i, j + 1) + u (1 - v) f(i + 1, j) + u v f(i + 1, j + 1)

目标点的像素值就是周围 4 个点像素值的加权和，明显可以看出离得近的权值比较大例如 (0, 0) 点的权值就是 0.750.75，*离得远的如 (-1, 1) 权值就比较小，为 0.25*0.25，这也比较符合常理吧。把值带入计算就可以得到 P2 点的值了，结果是 12.5 与代码吻合

上了，nice。

pytorch 里使用 bilinear 插值：

```
nn.Upsample(scale_factor=2, mode='bilinear')
```

CNN 网络要想获得好效果，skip-connection 基本必不可少。Unet 中这一关键步骤融合了底层信息的位置信息与深层特征的语义信息，pytorch 代码：

```
torch.cat([low_layer_features,
deep_layer_features], dim=1)
```

这里需要注意的是，**FCN 中深层信息与浅层信息融合是通过对应像素相加的方式，而 Unet 是通过拼接的方式。**

那么这两者有什么区别呢，其实 在 ResNet 与 DenseNet 中也有一样的区别，Resnet 使用了对应值相加，DenseNet 使用了拼接。**个人理解在相加的方式下，feature map 的维度没有变化，但每个维度都包含了更多特征，对于普通的分类任务这种不需要从 feature map 复原到原始分辨率的任务来说，这是一个高效的选择；而拼接则保留了更多的维度/位置 信息，这使得后面的 layer 可以在浅层特征与深层特征自由选择，这对语义分割任务来说更有优势。**

代码解读：

网络模块定义：

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class DoubleConv(nn.Module):
    """(convolution => [BN] => ReLU) * 2"""

    def __init__(self, in_channels, out_channels, mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
```

```
        )

    def forward(self, x):
        return self.double_conv(x)


class Down(nn.Module):
    """Downscaling with maxpool then double
conv"""

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
        )

    def forward(self, x):
        return self.maxpool_conv(x)


class up(nn.Module):
    ''' up path
        conv_transpose => double_conv
    '''
```

```
    def __init__(self, in_ch, out_ch,
Transpose=False):
        super(up, self).__init__()

        #  would be a nice idea if the upsampling
could be learned too,
        #  but my machine do not have enough
memory to handle all those weights
        if Transpose:
            self.up = nn.ConvTranspose2d(in_ch,
in_ch//2, 2, stride=2)
        else:
            # self.up =
nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)
            self.up =
nn.Sequential(nn.Upsample(scale_factor=2,
mode='bilinear', align_corners=True),

 nn.Conv2d(in_ch, in_ch//2, kernel_size=1,
padding=0),

 nn.ReLU(inplace=True))
        self.conv = double_conv(in_ch, out_ch)
        self.up.apply(self.init_weights)
```

```
    def forward(self, x1, x2):
        '''
            conv output shape = (input_shape -
Filter_shape + 2 * padding)/stride + 1
        '''

        x1 = self.up(x1)

        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = nn.functional.pad(x1, (diffX // 2,
diffX - diffX//2,
                                    diffY // 2,
diffY - diffY//2))

        x = torch.cat([x2,x1], dim=1)
        x = self.conv(x)
        return x

    @staticmethod
    def init_weights(m):
        if type(m) == nn.Conv2d:
            init.xavier_normal(m.weight)
            init.constant(m.bias,0)
```

```
class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels,
out_channels, kernel_size=1)

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

网络结构整体定义：

```
class Unet(nn.Module):
    def __init__(self, in_ch, out_ch, gpu_ids=[]):
        super(Unet, self).__init__()
        self.loss_stack = 0
        self.matrix_iou_stack = 0
        self.stack_count = 0
        self.display_names = ['loss_stack',
'matrix_iou_stack']
        self.gpu_ids = gpu_ids
        self.bce_loss = nn.BCELoss()
        self.device = torch.device('cuda:
{}'.format(self.gpu_ids[0])) if
torch.cuda.is_available() else torch.device('cpu')
```

```
        self.inc = inconv(in_ch, 64)
        self.down1 = down(64, 128)
        # print(list(self.down1.parameters()))
        self.down2 = down(128, 256)
        self.down3 = down(256, 512)
        self.drop3 = nn.Dropout2d(0.5)
        self.down4 = down(512, 1024)
        self.drop4 = nn.Dropout2d(0.5)
        self.up1 = up(1024, 512, False)
        self.up2 = up(512, 256, False)
        self.up3 = up(256, 128, False)
        self.up4 = up(128, 64, False)
        self.outc = outconv(64, 1)
        self.optimizer =
torch.optim.Adam(self.parameters(), lr=1e-4)
        # self.optimizer =
torch.optim.SGD(self.parameters(), lr=0.1,
momentum=0.9, weight_decay=0.0005)

    def forward(self):
        x1 = self.inc(self.x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x4 = self.drop3(x4)
        x5 = self.down4(x4)
```

```
        x5 = self.drop4(x5)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        x = self.outc(x)
        self.pred_y = nn.functional.sigmoid(x)

    def set_input(self, x, y):
        self.x = x.to(self.device)
        self.y = y.to(self.device)

    def optimize_params(self):
        self.forward()
        self._bce_iou_loss()
        _ = self.accu_iou()
        self.stack_count += 1
        self.zero_grad()
        self.loss.backward()
        self.optimizer.step()

    def accu_iou(self):
        # B is the mask pred, A is the malanoma
        y_pred = (self.pred_y > 0.5) * 1.0
        y_true = (self.y > 0.5) * 1.0
        pred_flat = y_pred.view(y_pred.numel())
```

```
        true_flat = y_true.view(y_true.numel())

        intersection = float(torch.sum(pred_flat *
true_flat)) + 1e-7
        denominator = float(torch.sum(pred_flat +
true_flat)) - intersection + 2e-7

        self.matrix_iou = intersection/denominator
        self.matrix_iou_stack += self.matrix_iou
        return self.matrix_iou

    def _bce_iou_loss(self):
        y_pred = self.pred_y
        y_true = self.y
        pred_flat = y_pred.view(y_pred.numel())
        true_flat = y_true.view(y_true.numel())

        intersection = torch.sum(pred_flat *
true_flat) + 1e-7
        denominator = torch.sum(pred_flat +
true_flat) - intersection + 1e-7
        iou = torch.div(intersection, denominator)
        bce_loss = self.bce_loss(pred_flat,
true_flat)
        self.loss = bce_loss - iou + 1
        self.loss_stack += self.loss
```

```
    def get_current_losses(self):
        errors_ret = {}
        for name in self.display_names:
            if isinstance(name, str):
                errors_ret[name] =
float(getattr(self, name)) / self.stack_count
        self.loss_stack = 0
        self.matrix_iou_stack = 0
        self.stack_count = 0
        return errors_ret

    def eval_iou(self):
        with torch.no_grad():
            self.forward()
            self._bce_iou_loss()
            _ = self.accu_iou()
            self.stack_count += 1
```

小结：

Unet 基于 Encoder-Decoder 结构，通过拼接的方式实现特征融合，结构简明且稳定。

参考：

Zero黑羽枫：语义分割网络 U-Net 详解