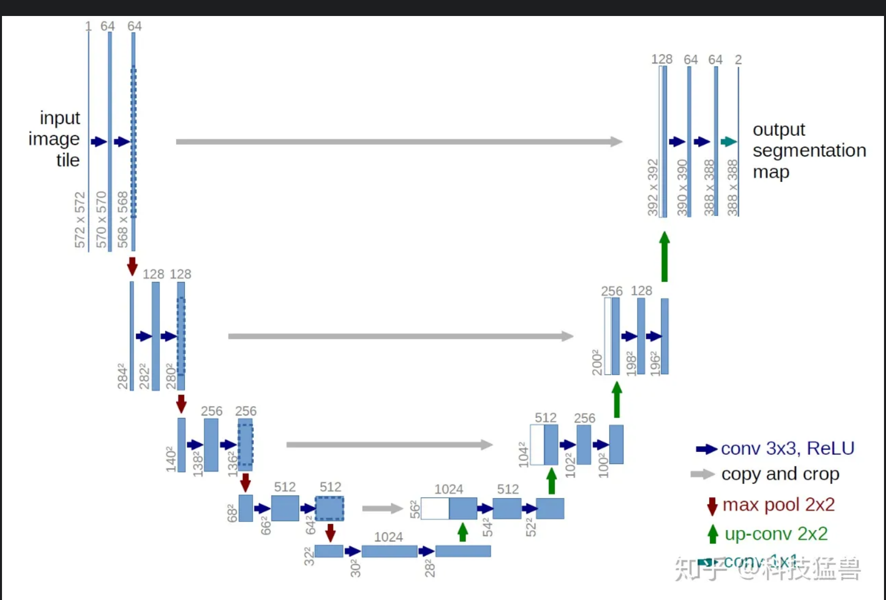
UNET



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

As shown in the figure above, the Unet network structure is symmetric and resembles the letter U, hence its name. The entire image is composed of blue/white boxes and arrows of various colors. Blue/white boxes represent feature maps; blue arrows represent 3x3 convolutions for feature extraction; gray arrows represent skip-connections for feature fusion; red arrows represent pooling for dimensionality reduction; green arrows represent upsampling for dimensionality restoration; and cyan arrows represent 1x1 convolutions for outputting results. The copy in the gray arrow "copy and crop" refers to concatenation, while the crop is used to ensure that the dimensions of the two are consistent.

```python
""" Parts of the U-Net model """

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, bias=False),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_channels, out_channels, kernel_size=3, padding=1, bias=False),
            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):
    """Upscaling then double conv"""

    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()

        # if bilinear, use the normal convolutions to reduce the number of channels
        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
            self.conv = DoubleConv(in_channels, out_channels, in_channels // 2)
        else:
            self.up = nn.ConvTranspose2d(in_channels, in_channels // 2, kernel_size=2, stride=2)
            self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        # input is CHW
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
                        diffY // 2, diffY - diffY // 2])
        # if you have padding issues, see
        # https://github.com/HaiyongJiang/U-Net-Pytorch-Unstructured-Buggy/commit/0e854509c2cea854e247a9c615f175f76fbb2e3a
        # https://github.com/xiaopeng-liao/Pytorch-UNet/commit/8ebac70e633bac59fc22bb5195e513d5832fb3bd
        x = torch.cat([x2, x1], dim=1)
        return self.conv(x)


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)
```

Define UNET as follows:

```python
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
```

Binary Cross Entropy

`torch.nn.BCELoss(weight=None, size_average=None, reduce=None, reduction='mean')`