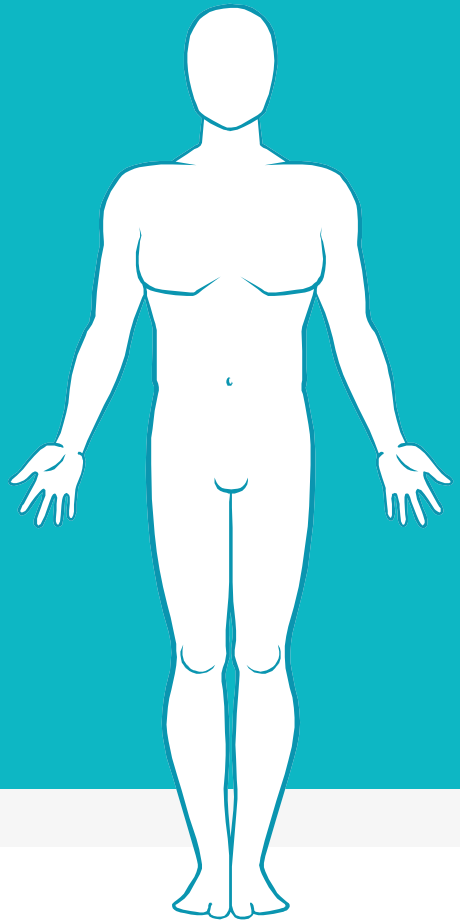
Computer Vision in Medical Applications – Brain MRI Segmentation



Nagarjun Lakshmipathy (1982435)
Kevin Do Cao (2054459)
Mirza Mujtaba Hussain (1989740)

COMPUTER VISION IN MEDICAL APPLICATIONS

Computer Vision has had great impact on the medical industry. It has led to many innovations in:

- Early detection in diagnostics
- Personalized treatment
- Improved surgical planning and guidance
- Automated data analysis

MRI Segmentation

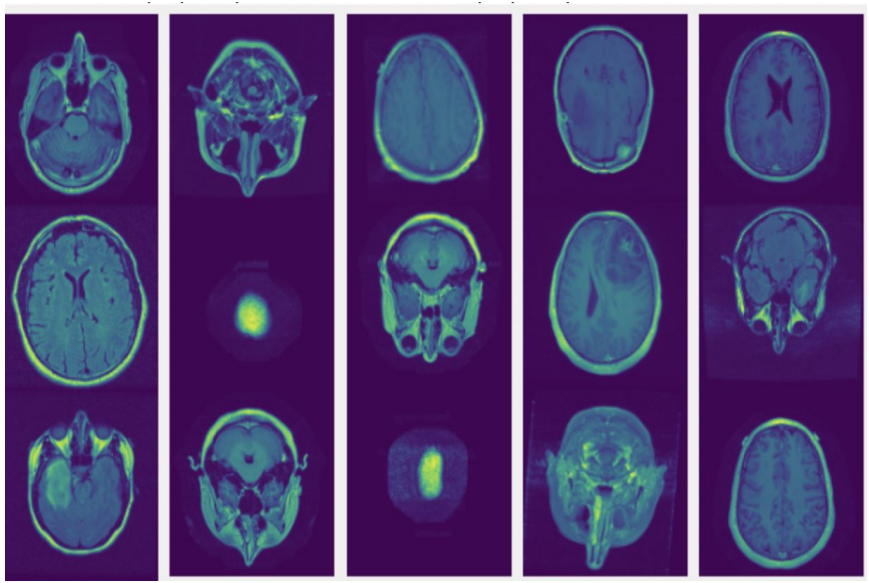
MRI segmentation is the process of separating an MRI image into different anatomical structures or regions of interest.

2D MRI Segmentation	**3D MRI Segmentation**
2D MRI segmentation involves segmenting structures in individual MRI slices	3D MRI segmentation involves segmenting structures across a volume of MRI slices
Segmentation is faster and requires less computational resources	Gives more accurate segmentation but requires more computational resources.

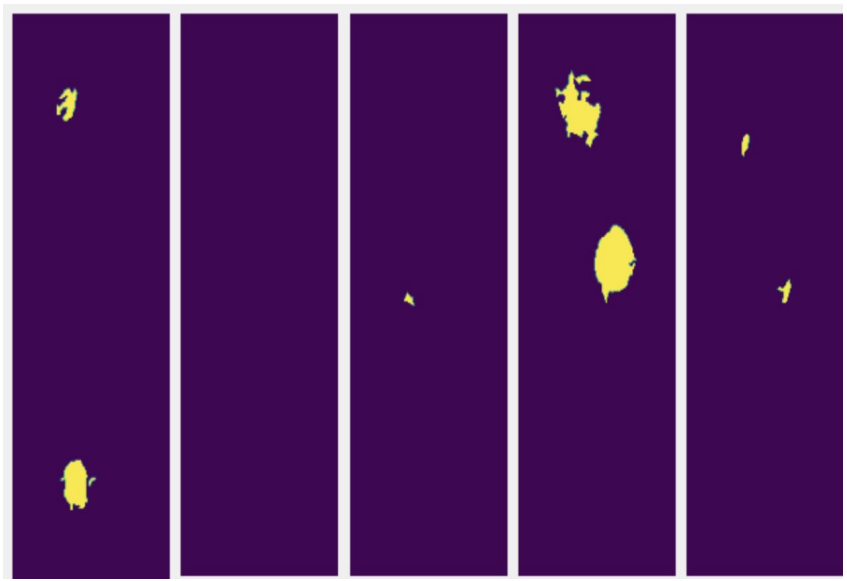
Data Used

- The dataset used for development was obtained from The Cancer Imaging Archive (TCIA)
- Involves 110 cases of lower-grade glioma patients
- Contains data with original MRI scans and highlighted duplicates displaying regions of abnormalities

Sample of Dataset – 2D Segmentation



Data



Mask

Data Preprocessing

Using Albumentations:

- Resize (128,128)
- Horizontal Flip
- Vertical Flip
- Normalize

```
img = (img - mean * max_pixel_value) / (std * max_pixel_value)
```

```
SIZE = 128

transforms = A.Compose([
    A.Resize(width=SIZE, height=SIZE, p=1.0),
    A.HorizontalFlip(p=0.5),
    A.VerticalFlip(p=0.5),
    A.Normalize(p=1.0),
    ToTensor(),
])
```

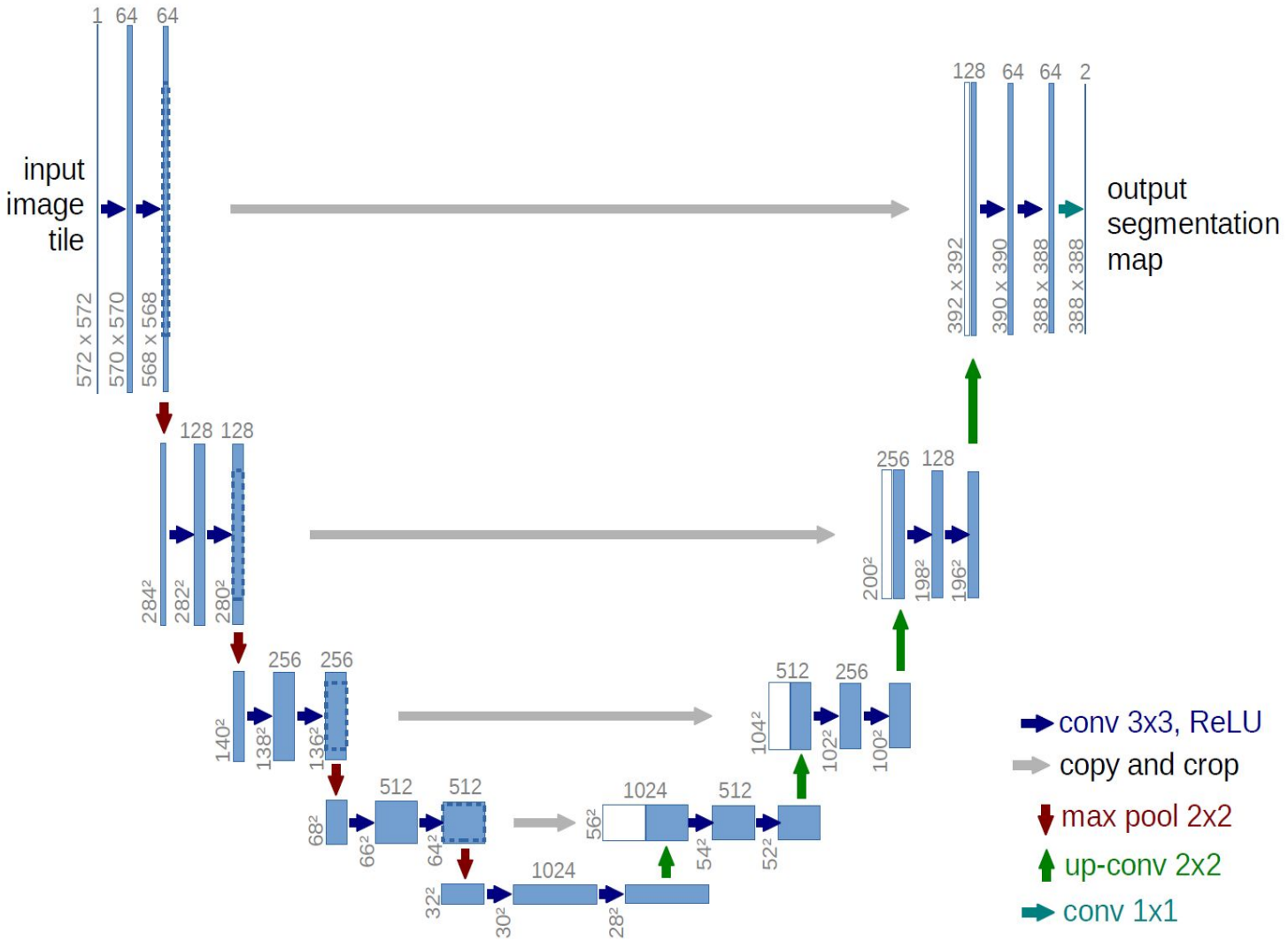
INTRODUCTION TO RESEARCH PAPERS

RESEARCH PAPER: U-NET[1]

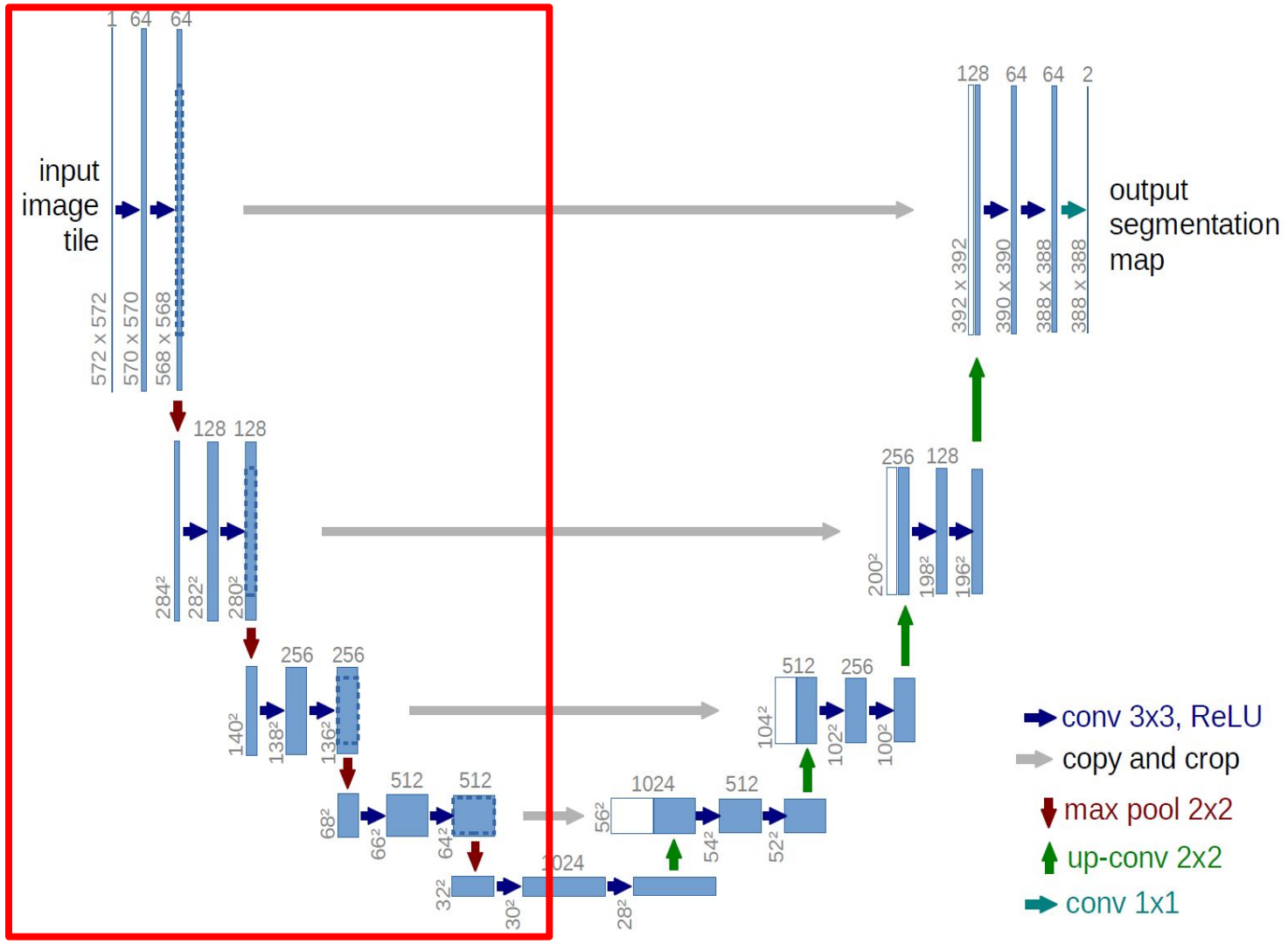
U-Net is a neural network architecture designed for semantic image segmentation, particularly in medical imaging. It has a contracting path to capture context and a expansive path to recover spatial resolution and generate an output segmentation map. The architecture includes skip connections between the two paths, allowing spatial information to be maintained and improving segmentation accuracy. U-Net has been shown to be effective in segmenting tumors, organs, and abnormal tissue in medical images, as well as in other image segmentation tasks. It is a widely used network due to its ability to capture both local and global context information.

[1] Ronneberger, Olaf, et al. “U-Net: Convolutional Networks for Biomedical Image Segmentation.” *ArXiv.org*, 18 May 2015, https://arxiv.org/abs/1505.04597.

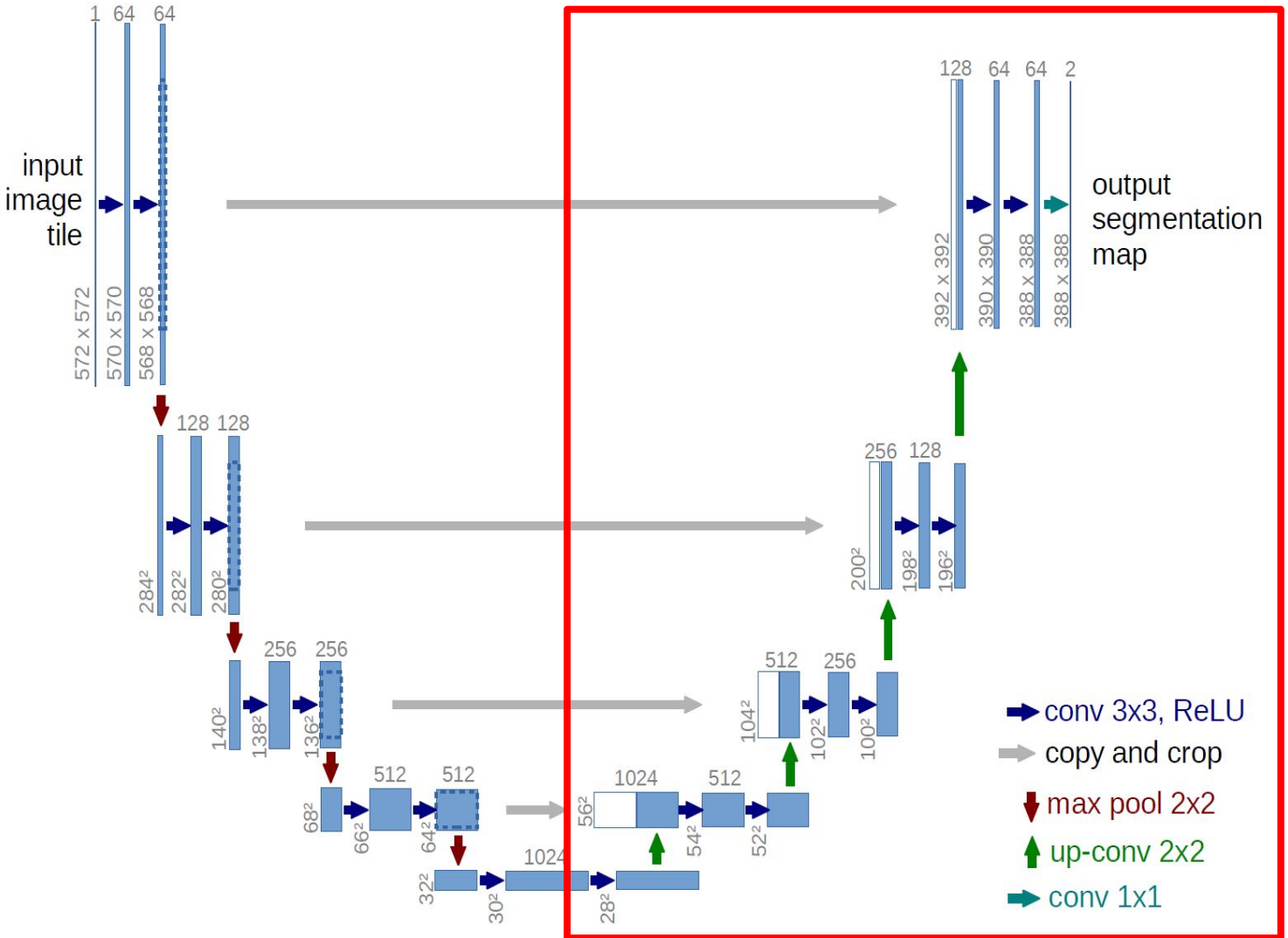
U-NET ARCHITECTURE



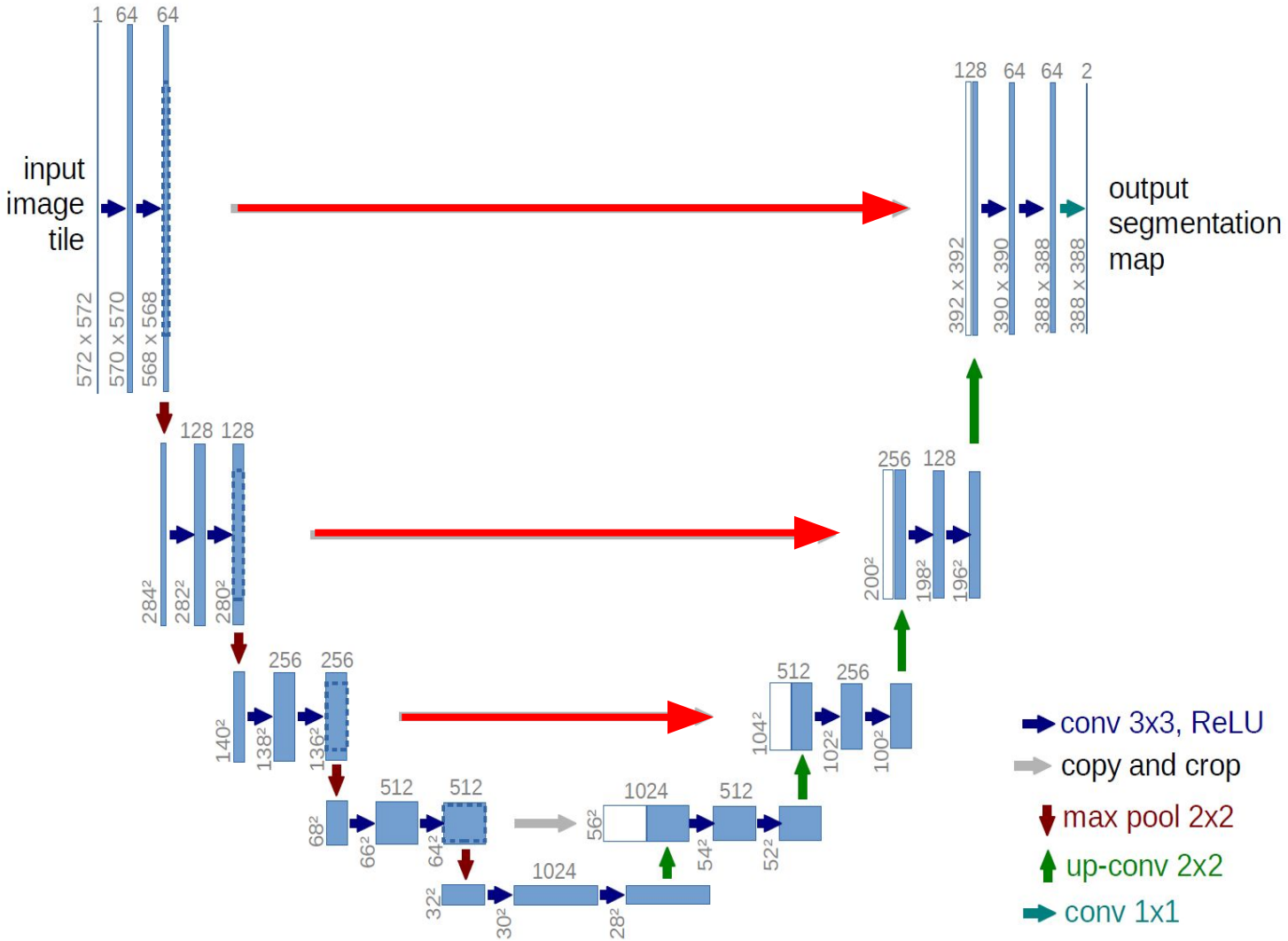
U-NET ARCHITECTURE:
CONTRACTING PATH



U-NET ARCHITECTURE:
EXPANDING PATH



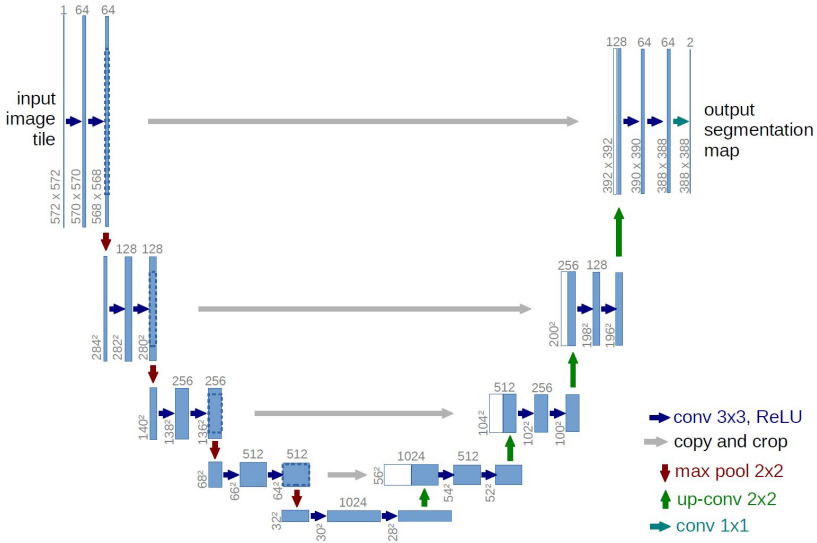
U-NET ARCHITECTURE:
SKIP CONNECTIONS



DEMONSTRATION OF CODE

U-NET Implementation

```python
class conv_block(nn.Module):
    def __init__(self,ch_in,ch_out):
        super(conv_block,self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(ch_in, ch_out, kernel_size=3,stride=1,padding=1,bias=True),
            nn.BatchNorm2d(ch_out),
            nn.ReLU(inplace=True),
            nn.Conv2d(ch_out, ch_out, kernel_size=3,stride=1,padding=1,bias=True),
            nn.BatchNorm2d(ch_out),
            nn.ReLU(inplace=True)
        )
    def forward(self,x):
        x = self.conv(x)
        return x


class up_conv(nn.Module):
    def __init__(self,ch_in,ch_out,scalefactor=2,mode_='nearest',align_corners_=None):
        super(up_conv,self).__init__()
        self.up = nn.Sequential(
            nn.Upsample(scale_factor=scalefactor,mode=mode_, align_corners=align_corners_),
            nn.Conv2d(ch_in,ch_out,kernel_size=3,stride=1,padding=1,bias=True),
            nn.BatchNorm2d(ch_out),
            nn.ReLU(inplace=True)
        )
    def forward(self,x):
        x = self.up(x)
        return x
```



Model blocks required for U-Net

```
class U_Net(nn.Module):
    def __init__(self,img_ch=3,output_ch=1):
        super(U_Net,self).__init__()

        self.Maxpool = nn.MaxPool2d(kernel_size=2,stride=2)

        self.Conv1 = conv_block(ch_in=img_ch,ch_out=64)
        self.Conv2 = conv_block(ch_in=64,ch_out=128)
        self.Conv3 = conv_block(ch_in=128,ch_out=256)
        self.Conv4 = conv_block(ch_in=256,ch_out=512)
        self.Conv5 = conv_block(ch_in=512,ch_out=1024)

        self.Up5 = up_conv(ch_in=1024,ch_out=512)
        self.Up_conv5 = conv_block(ch_in=1024, ch_out=512)

        self.Up4 = up_conv(ch_in=512,ch_out=256)
        self.Up_conv4 = conv_block(ch_in=512, ch_out=256)

        self.Up3 = up_conv(ch_in=256,ch_out=128)
        self.Up_conv3 = conv_block(ch_in=256, ch_out=128)

        self.Up2 = up_conv(ch_in=128,ch_out=64)
        self.Up_conv2 = conv_block(ch_in=128, ch_out=64)

        self.Conv_1x1 = nn.Conv2d(64,output_ch,kernel_size=1,stride=1,padding=0)
```

```
def forward(self,x):
    # encoding path
    x1 = self.Conv1(x)

    x2 = self.Maxpool(x1)
    x2 = self.Conv2(x2)

    x3 = self.Maxpool(x2)
    x3 = self.Conv3(x3)

    x4 = self.Maxpool(x3)
    x4 = self.Conv4(x4)

    x5 = self.Maxpool(x4)
    x5 = self.Conv5(x5)

    # decoding + concat path

    d5 = self.Up5(x5)
    d5 = torch.cat((x4,d5),dim=1)
    d5 = self.Up_conv5(d5)

    d4 = self.Up4(d5)
    d4 = torch.cat((x3,d4),dim=1)
    d4 = self.Up_conv4(d4)

    d3 = self.Up3(d4)
    d3 = torch.cat((x2,d3),dim=1)
    d3 = self.Up_conv3(d3)

    d2 = self.Up2(d3)
    d2 = torch.cat((x1,d2),dim=1)
    d2 = self.Up_conv2(d2)

    d1 = self.Conv_1x1(d2)

    return d1
```

INTRODUCTION TO RESEARCH PAPERS

RESEARCH PAPER: Attention U-NET[2]

The attention mechanism is a technique used in deep learning and neural network architectures to help the network focus on relevant parts of the input. This mechanism was first introduced in natural language processing tasks and has since been used in computer vision tasks as well. The attention mechanism works by assigning weights to different parts of the input, allowing the network to focus on the most important features of the input.

The attention mechanism in the Attention U-Net is used to help the network focus on relevant parts of the input during the segmentation process. The attention mechanism is applied to the feature maps generated by the encoder, allowing the network to selectively attend to the most important features in the image. This can help improve the accuracy of the segmentation mask and reduce the number of false positives and false negatives.

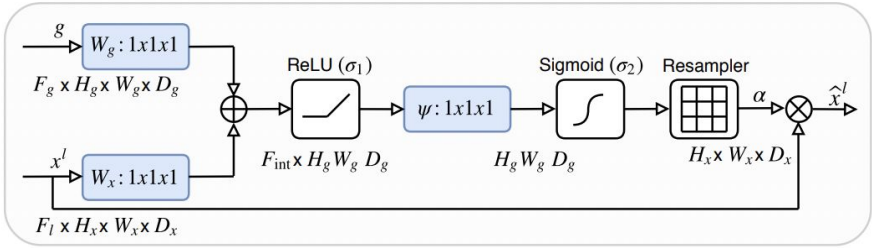
[2] Oktay, Ozan, et al. “Attention U-Net: Learning Where to Look for the Pancreas.” *ArXiv.org*, 20 May 2018, https://arxiv.org/abs/1804.03999.

DEMONSTRATION OF CODE

Attention U-NET Implementation

```
class Attention_block(nn.Module):
    def __init__(self,F_g,F_l,F_int):
        super(Attention_block,self).__init__()
        self.W_g = nn.Sequential(
            nn.Conv2d(F_g, F_int, kernel_size=1,stride=1,padding=0,bias=True),
            nn.BatchNorm2d(F_int)
            )

        self.W_x = nn.Sequential(
            nn.Conv2d(F_l, F_int, kernel_size=1,stride=1,padding=0,bias=True),
            nn.BatchNorm2d(F_int)
        )

        self.psi = nn.Sequential(
            nn.Conv2d(F_int, 1, kernel_size=1,stride=1,padding=0,bias=True),
            nn.BatchNorm2d(1),
            nn.Sigmoid()
        )

        self.relu = nn.ReLU(inplace=True)

    def forward(self,g,x):
        g1 = self.W_g(g)
        x1 = self.W_x(x)
        psi = self.relu(g1+x1)
        psi = self.psi(psi)

        return x*psi
```

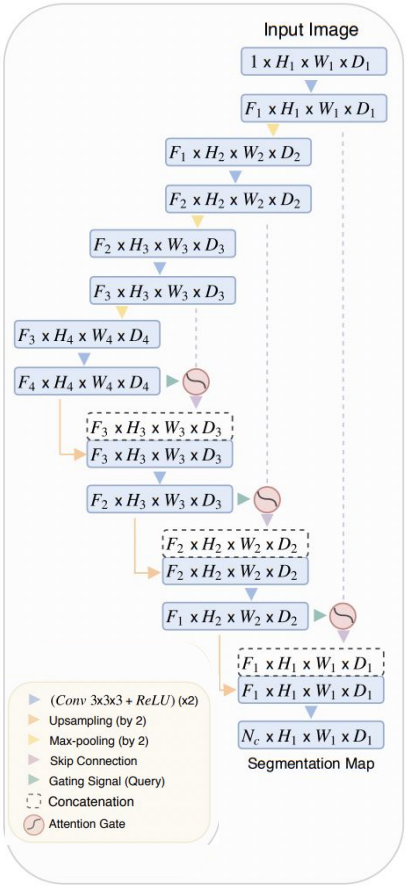
Model blocks required for Attention U-Net



Architecture of Attention Mechanism

```python
class AttentionUNet(nn.Module):
    def __init__(self,img_ch=3,output_ch=1):
        super(AttentionUNet,self).__init__()

        self.Maxpool = nn.MaxPool2d(kernel_size=2,stride=2)

        self.Conv1 = conv_block(ch_in=img_ch,ch_out=64)
        self.Conv2 = conv_block(ch_in=64,ch_out=128)
        self.Conv3 = conv_block(ch_in=128,ch_out=256)
        self.Conv4 = conv_block(ch_in=256,ch_out=512)
        self.Conv5 = conv_block(ch_in=512,ch_out=1024)

        self.Up5 = up_conv(ch_in=1024,ch_out=512)
        self.Att5 = Attention_block(F_g=512,F_l=512,F_int=256)
        self.Up_conv5 = conv_block(ch_in=1024, ch_out=512)

        self.Up4 = up_conv(ch_in=512,ch_out=256)
        self.Att4 = Attention_block(F_g=256,F_l=256,F_int=128)
        self.Up_conv4 = conv_block(ch_in=512, ch_out=256)

        self.Up3 = up_conv(ch_in=256,ch_out=128)
        self.Att3 = Attention_block(F_g=128,F_l=128,F_int=64)
        self.Up_conv3 = conv_block(ch_in=256, ch_out=128)

        self.Up2 = up_conv(ch_in=128,ch_out=64)
        self.Att2 = Attention_block(F_g=64,F_l=64,F_int=32)
        self.Up_conv2 = conv_block(ch_in=128, ch_out=64)

        self.Conv_1x1 = nn.Conv2d(64,output_ch,kernel_size=1,stride=1,padding=0)
```

```python
def forward(self,x):
    # encoding path
    x1 = self.Conv1(x)

    x2 = self.Maxpool(x1)
    x2 = self.Conv2(x2)

    x3 = self.Maxpool(x2)
    x3 = self.Conv3(x3)

    x4 = self.Maxpool(x3)
    x4 = self.Conv4(x4)

    x5 = self.Maxpool(x4)
    x5 = self.Conv5(x5)

    # decoding + concat path
    d5 = self.Up5(x5)
    x4 = self.Att5(g=d5,x=x4)
    d5 = torch.cat((x4,d5),dim=1)
    d5 = self.Up_conv5(d5)

    d4 = self.Up4(d5)
    x3 = self.Att4(g=d4,x=x3)
    d4 = torch.cat((x3,d4),dim=1)
    d4 = self.Up_conv4(d4)

    d3 = self.Up3(d4)
    x2 = self.Att3(g=d3,x=x2)
    d3 = torch.cat((x2,d3),dim=1)
    d3 = self.Up_conv3(d3)

    d2 = self.Up2(d3)
    x1 = self.Att2(g=d2,x=x1)
    d2 = torch.cat((x1,d2),dim=1)
    d2 = self.Up_conv2(d2)

    d1 = self.Conv_1x1(d2)

    return d1
```

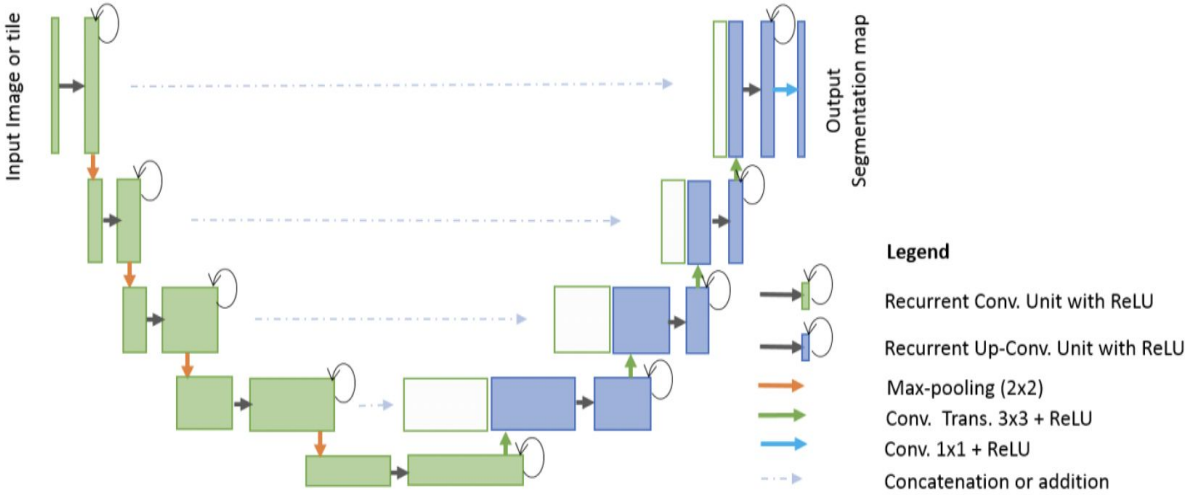


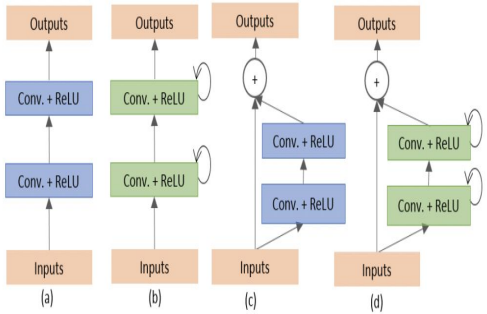
INTRODUCTION TO RESEARCH PAPERS

RESEARCH PAPER: R2U-NET[-]

R2U-Net, also known as Recurrent Residual U-Net, is an advanced neural network architecture for semantic segmentation of images. It is an extension of the popular U-Net architecture and incorporates recurrent connections, residual connections to improve the performance of the network.

The key difference between the R2U-Net and the U-Net is the inclusion of recurrent connections and residual connections in the encoder and decoder. Recurrent connections allow the network to learn temporal dependencies and capture long-range dependencies in the input. Residual connections help mitigate the vanishing gradient problem and allow for faster training and improved performance.

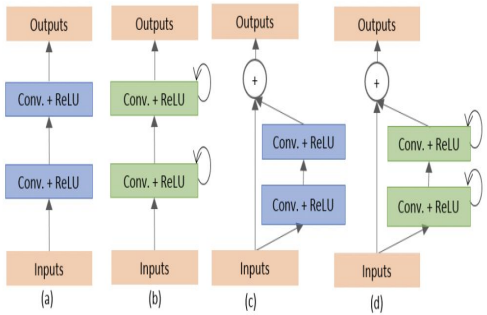

Input Image or tile
Output Segmentation map
Legend
Recurrent Conv. Unit with ReLU
Recurrent Up-Conv. Unit with ReLU
Max-pooling (2x2)
Conv. Trans. 3x3 + ReLU
Conv. 1x1 + ReLU
Concatenation or addition


Outputs
Conv. + ReLU
Inputs
(a)
(b)
(c)
(d)

DEMONSTRATION OF CODE

R2U-NET Implementation

```python
#for r2unet
class Rec_block(nn.Module):
    def __init__(self,ch_out,t=2):
        super(Rec_block,self).__init__()
        self.t = t
        self.ch_out = ch_out
        self.conv = nn.Sequential(
            nn.Conv2d(ch_out,ch_out,kernel_size=3,stride=1,padding=1,bias=True),
            nn.BatchNorm2d(ch_out),
            nn.ReLU(inplace=True)
        )
    def forward(self,x):
        for i in range(self.t):

            if i==0:
                x1 = self.conv(x)

            x1 = self.conv(x+x1)
        return x1

class RRCNN_block(nn.Module):
    def __init__(self,ch_in,ch_out,t=2):
        super(RRCNN_block,self).__init__()
        self.RCNN = nn.Sequential(
            Rec_block(ch_out,t=t),
            Rec_block(ch_out,t=t)
        )
        self.Conv_1x1 = nn.Conv2d(ch_in,ch_out,kernel_size=1,stride=1,padding=0)
    def forward(self,x):
        x = self.Conv_1x1(x)
        x1 = self.RCNN(x)
        return x+x1
```



```python
class R2U_Net(nn.Module):
    def __init__(self,img_ch=3,output_ch=1,t=2):
        super(R2U_Net,self).__init__()

        self.Maxpool = nn.MaxPool2d(kernel_size=2,stride=2)
        self.Upsample = nn.Upsample(scale_factor=2)

        self.RRCNN1 = RRCNN_block(ch_in=img_ch,ch_out=64,t=t)

        self.RRCNN2 = RRCNN_block(ch_in=64,ch_out=128,t=t)

        self.RRCNN3 = RRCNN_block(ch_in=128,ch_out=256,t=t)

        self.RRCNN4 = RRCNN_block(ch_in=256,ch_out=512,t=t)

        self.RRCNN5 = RRCNN_block(ch_in=512,ch_out=1024,t=t)


        self.Up5 = up_conv(ch_in=1024,ch_out=512)
        self.Up_RRCNN5 = RRCNN_block(ch_in=1024, ch_out=512,t=t)

        self.Up4 = up_conv(ch_in=512,ch_out=256)
        self.Up_RRCNN4 = RRCNN_block(ch_in=512, ch_out=256,t=t)

        self.Up3 = up_conv(ch_in=256,ch_out=128)
        self.Up_RRCNN3 = RRCNN_block(ch_in=256, ch_out=128,t=t)

        self.Up2 = up_conv(ch_in=128,ch_out=64)
        self.Up_RRCNN2 = RRCNN_block(ch_in=128, ch_out=64,t=t)

        self.Conv_1x1 = nn.Conv2d(64,output_ch,kernel_size=1,stride=1,padding=0)
```

```python
def forward(self,x):
    # encoding path
    x1 = self.RRCNN1(x)

    x2 = self.Maxpool(x1)
    x2 = self.RRCNN2(x2)

    x3 = self.Maxpool(x2)
    x3 = self.RRCNN3(x3)

    x4 = self.Maxpool(x3)
    x4 = self.RRCNN4(x4)

    x5 = self.Maxpool(x4)
    x5 = self.RRCNN5(x5)

    # decoding + concat path
    d5 = self.Up5(x5)
    d5 = torch.cat((x4,d5),dim=1)
    d5 = self.Up_RRCNN5(d5)

    d4 = self.Up4(d5)
    d4 = torch.cat((x3,d4),dim=1)
    d4 = self.Up_RRCNN4(d4)

    d3 = self.Up3(d4)
    d3 = torch.cat((x2,d3),dim=1)
    d3 = self.Up_RRCNN3(d3)

    d2 = self.Up2(d3)
    d2 = torch.cat((x1,d2),dim=1)
    d2 = self.Up_RRCNN2(d2)

    d1 = self.Conv_1x1(d2)

    return d1
```

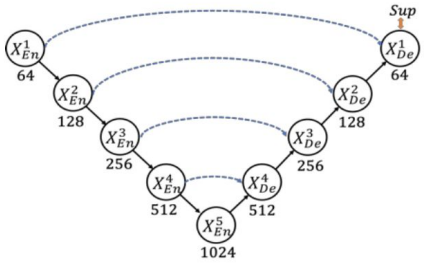
```
class AttentionR2U_Net(nn.Module):
    def __init__(self,img_ch=3,output_ch=1,t=2):
        super(AttentionR2U_Net,self).__init__()

        self.Maxpool = nn.MaxPool2d(kernel_size=2,stride=2)
        self.Upsample = nn.Upsample(scale_factor=2)

        self.RRCNN1 = RRCNN_block(ch_in=img_ch,ch_out=64,t=t)

        self.RRCNN2 = RRCNN_block(ch_in=64,ch_out=128,t=t)

        self.RRCNN3 = RRCNN_block(ch_in=128,ch_out=256,t=t)

        self.RRCNN4 = RRCNN_block(ch_in=256,ch_out=512,t=t)

        self.RRCNN5 = RRCNN_block(ch_in=512,ch_out=1024,t=t)


        self.Up5 = up_conv(ch_in=1024,ch_out=512)
        self.Att5 = Attention_block(F_g=512,F_l=512,F_int=256)
        self.Up_RRCNN5 = RRCNN_block(ch_in=1024, ch_out=512,t=t)

        self.Up4 = up_conv(ch_in=512,ch_out=256)
        self.Att4 = Attention_block(F_g=256,F_l=256,F_int=128)
        self.Up_RRCNN4 = RRCNN_block(ch_in=512, ch_out=256,t=t)

        self.Up3 = up_conv(ch_in=256,ch_out=128)
        self.Att3 = Attention_block(F_g=128,F_l=128,F_int=64)
        self.Up_RRCNN3 = RRCNN_block(ch_in=256, ch_out=128,t=t)

        self.Up2 = up_conv(ch_in=128,ch_out=64)
        self.Att2 = Attention_block(F_g=64,F_l=64,F_int=32)
        self.Up_RRCNN2 = RRCNN_block(ch_in=128, ch_out=64,t=t)

        self.Conv_1x1 = nn.Conv2d(64,output_ch,kernel_size=1,stride=1,padding=0)
```

```
def forward(self,x):
    # encoding path
    x1 = self.RRCNN1(x)

    x2 = self.Maxpool(x1)
    x2 = self.RRCNN2(x2)

    x3 = self.Maxpool(x2)
    x3 = self.RRCNN3(x3)

    x4 = self.Maxpool(x3)
    x4 = self.RRCNN4(x4)

    x5 = self.Maxpool(x4)
    x5 = self.RRCNN5(x5)

    # decoding + concat path
    d5 = self.Up5(x5)
    x4 = self.Att5(g=d5,x=x4)
    d5 = torch.cat((x4,d5),dim=1)
    d5 = self.Up_RRCNN5(d5)

    d4 = self.Up4(d5)
    x3 = self.Att4(g=d4,x=x3)
    d4 = torch.cat((x3,d4),dim=1)
    d4 = self.Up_RRCNN4(d4)

    d3 = self.Up3(d4)
    x2 = self.Att3(g=d3,x=x2)
    d3 = torch.cat((x2,d3),dim=1)
    d3 = self.Up_RRCNN3(d3)

    d2 = self.Up2(d3)
    x1 = self.Att2(g=d2,x=x1)
    d2 = torch.cat((x1,d2),dim=1)
    d2 = self.Up_RRCNN2(d2)

    d1 = self.Conv_1x1(d2)

    return d1
```

RESEARCH PAPER: U-NET 3+[-]

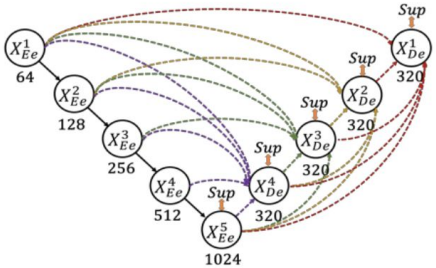
U-Net 3+ is an upgraded variant of the U-Net architecture used for image segmentation tasks. It builds on the standard U-Net architecture by incorporating Full-scale Skip Connections, Full-scale Deep Supervision and Classification-guided Module (CGM).We have only implemented the Full-scale Skip Connections in this project . It helps to incorporate low-level details with high-level semantics from feature maps in different scales.

Difference between UNet and UNet 3+



(a) UNet

(Plain skip connections)



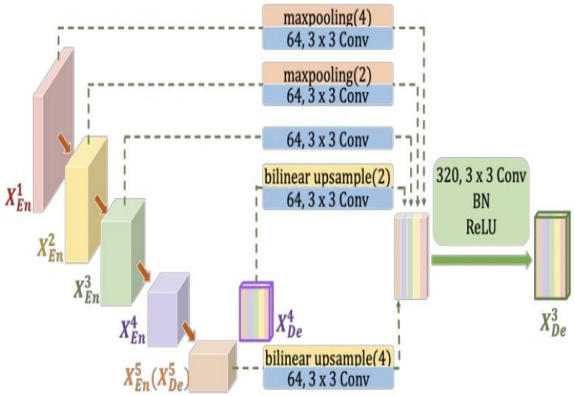
(c) UNet 3+

(Full-scale skip connections)

DEMONSTRATION OF CODE

U-NET 3+ Implementation (Only Full-scale Deep Supervision)

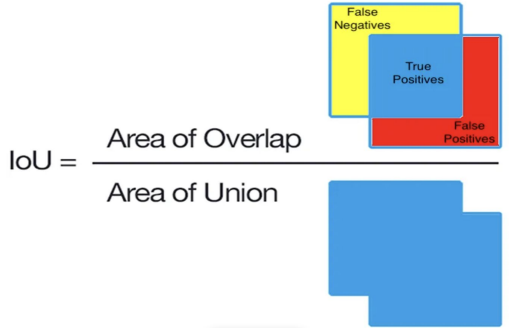
```python
class ThreePlusDecoder(torch.nn.Module):
    def __init__(self, center_out_channels, level, unet_depth=5):
        super(ThreePlusDecoder, self).__init__()

        self.unet_depth = unet_depth
        self.max_pool = torch.nn.MaxPool2d
        self.operation_list = torch.nn.ModuleList()

        #from center(bottlneck) to the decoder in consideration
        center_up_factor = int(2 ** (self.unet_depth - level))
        self.operation_list.append(
            up_conv(center_out_channels,64,scalefactor=center_up_factor,mode_='bilinear',align_corners_=True))

        #from previos decoders to the decoder in consideration
        for i in range(1, self.unet_depth - level):
            up_scaling_factor = int(2 ** (self.unet_depth - level - i))
            in_channels = 320
            self.operation_list.append(
                up_conv(in_channels,64,scalefactor=up_scaling_factor,mode_='bilinear',align_corners_=True))

        #from same enocoder level to the decoder in consideration
        current_scale = int(2 ** (self.unet_depth - level))
        current_in_channels = center_out_channels // current_scale
        self.current_level_operation = conv_block(current_in_channels, 64)
        self.operation_list.append(self.current_level_operation)

        #from encoder above to the decoder in consideration
        for i in range(1, level):
            kernel_size = int(2 ** i)
            in_channels = current_in_channels // int(2 ** i)
            self.operation_list.append(
                torch.nn.Sequential(
                    self.max_pool(kernel_size),
                    conv_block(in_channels, 64),
                )
            )


        self.final =  torch.nn.Conv2d(64 * self.unet_depth, 64 * self.unet_depth,kernel_size=(3, 3), padding=(1, 1))
        self.relu = torch.nn.ReLU(inplace=True)
        self.batchnorm=torch.nn.BatchNorm2d(64 * self.unet_depth)

    def forward(self, *args):
        out_list = []
        for idx, element in enumerate(args):
            out_list.append(self.operation_list[idx](element))

        x = torch.cat(out_list, dim=1)
        x= self.final(x)
        return self.relu(self.batchnorm(x))
```

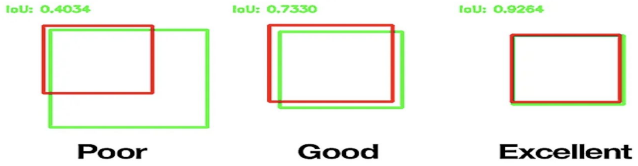


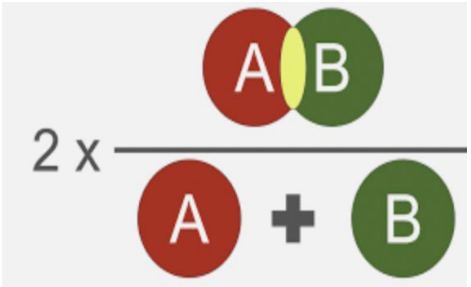
```
class UNet3(torch.nn.Module):
    def __init__(self,img_ch=3,output_ch=1):
        super(UNet3,self).__init__()


        self.center_channels = 1024
        self.Maxpool = nn.MaxPool2d(kernel_size=2,stride=2)

        self.enc_1 = conv_block(img_ch,64)
        self.enc_2 = conv_block(64,128)
        self.enc_3 = conv_block(128,256)
        self.enc_4 = conv_block(256,512)

        self.center = conv_block(512, self.center_channels)

        self.dec_4 = ThreePlusDecoder(self.center_channels,  level=4)
        self.dec_3 = ThreePlusDecoder(self.center_channels, level=3)
        self.dec_2 = ThreePlusDecoder(self.center_channels, level=2)
        self.dec_1 = ThreePlusDecoder(self.center_channels, level=1)

        self.final = torch.nn.Conv2d(320, output_ch, kernel_size=(3, 3), padding=(1, 1))

    def forward(self, x):
        enc_1 = self.enc_1(x)
        pool_1=self.Maxpool(enc_1)
        enc_2= self.enc_2(pool_1)
        pool_2=self.Maxpool(enc_2)
        enc_3= self.enc_3(pool_2)
        pool_3=self.Maxpool(enc_3)
        enc_4= self.enc_4(pool_3)
        pool_4=self.Maxpool(enc_4)
        center = self.center(pool_4)

        dec_4 = self.dec_4(center, enc_4, enc_3, enc_2, enc_1)
        dec_3 = self.dec_3(center, dec_4, enc_3, enc_2, enc_1)
        dec_2 = self.dec_2(center, dec_4, dec_3, enc_2, enc_1)
        dec_1 = self.dec_1(center, dec_4, dec_3, dec_2, enc_1)

        x = self.final(dec_1)
        return x
```

Loss Function

```
class BCEwithDiceLoss(nn.Module):
    def __init__(self):
        super(BCEwithDiceLoss, self).__init__()

    def forward(self, inputs, targets, smooth=1):

        #apply sigmoid to inputs
        inputs = nn.Sigmoid()(inputs)

        #flatten label and prediction tensors
        inputs = inputs.view(-1)
        targets = targets.view(-1)
        #calculate dice loss
        intersection = (inputs * targets).sum()
        dice = 1- (2.*intersection + smooth)/(inputs.sum() + targets.sum() + smooth)

       # calculate BCE
        bce = F.binary_cross_entropy_with_logits(inputs, targets)

        return dice + bce
```



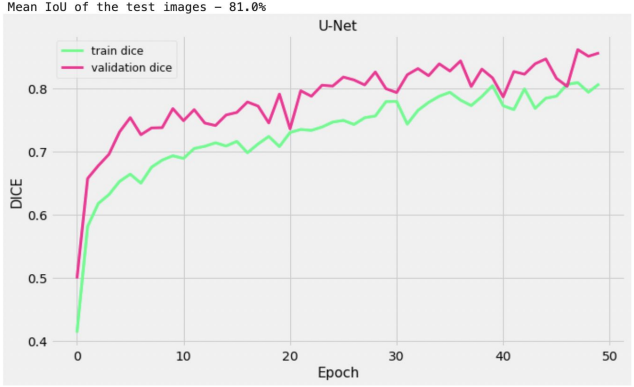


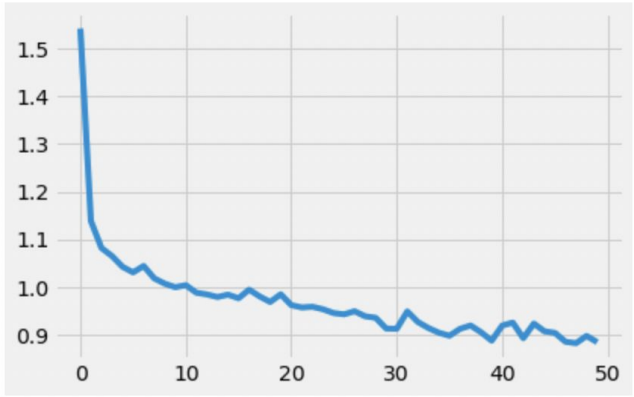


COMPARISON BETWEEN NETWORKS

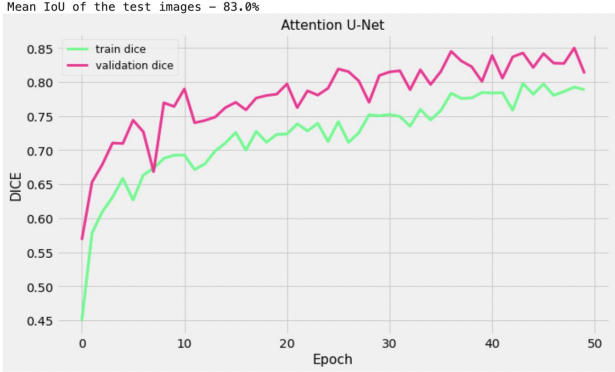
Comparison of Results between variants of U-NETs

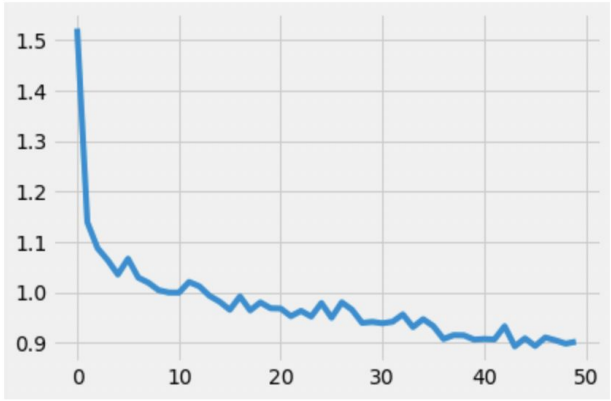
U-Net
Mean IoU of the test images – 81.0%



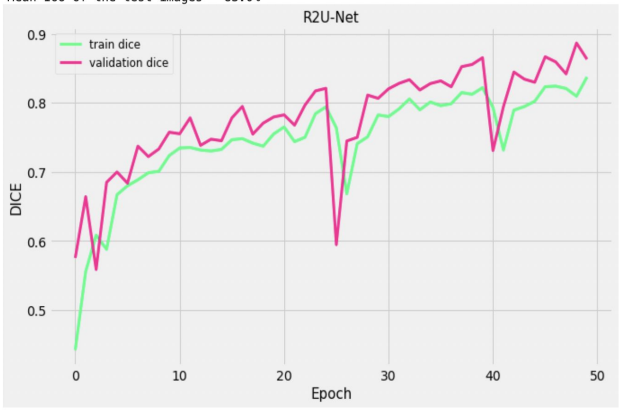


Attention U-Net
Mean IoU of the test images – 83.0%

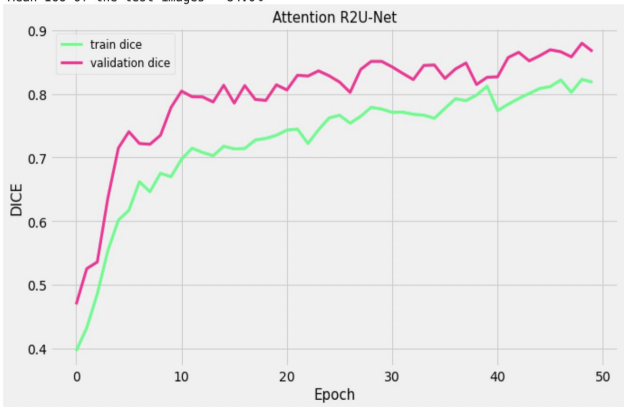


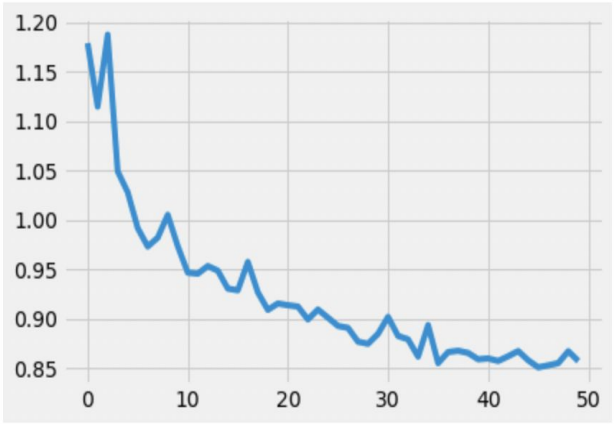


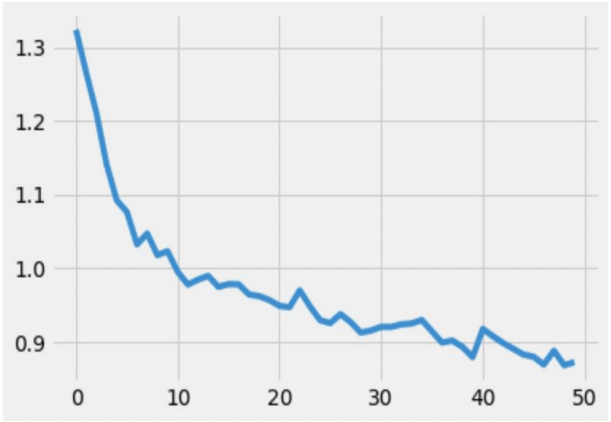
R2U-Net
Mean IoU of the test images – 83.0%



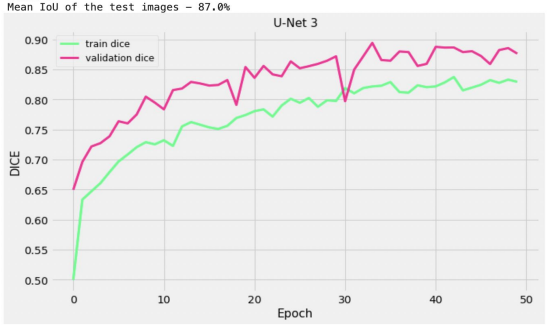
Attention R2U-Net
Mean IoU of the test images – 84.0%

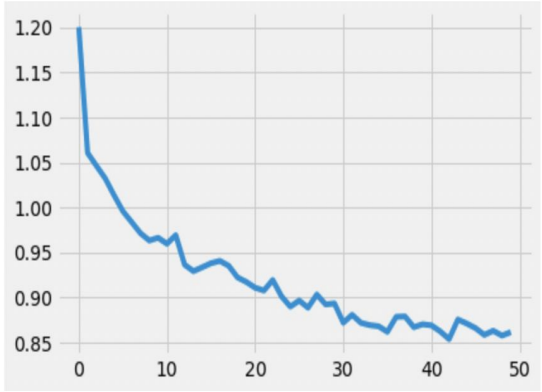


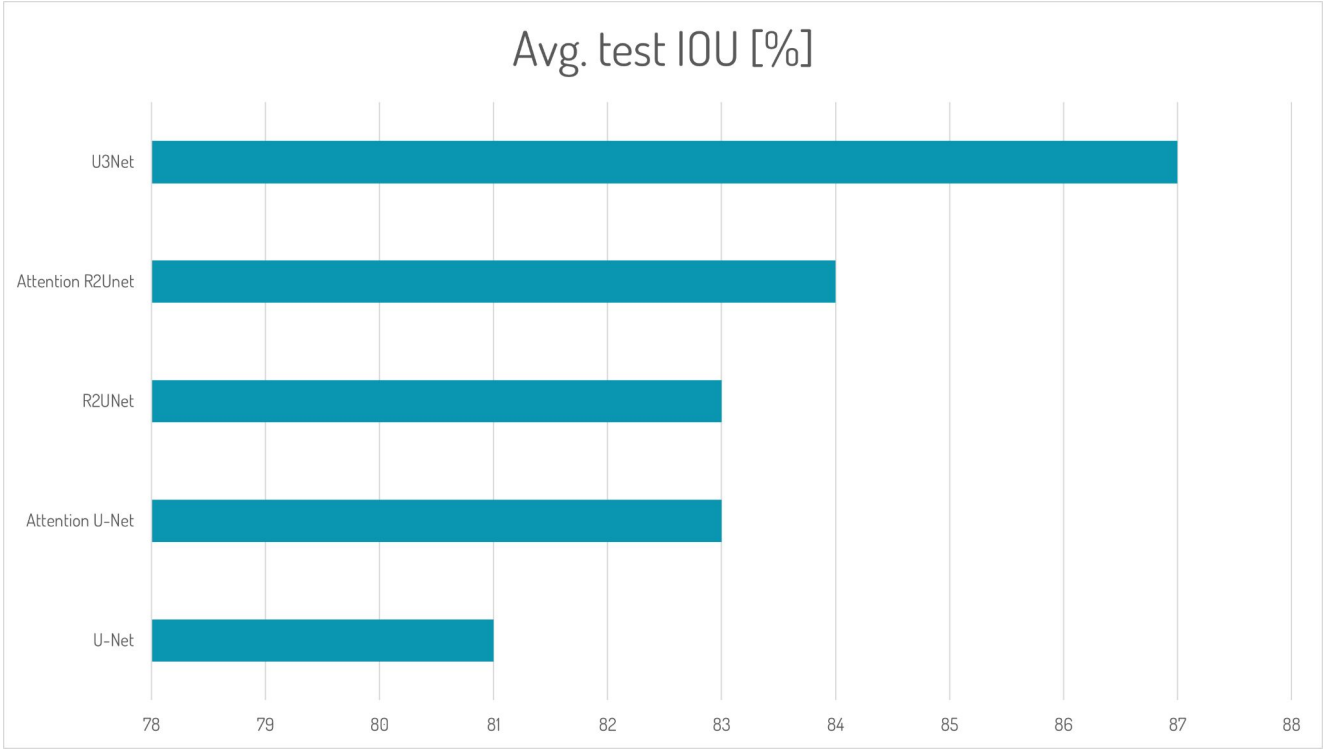




U-Net 3
Mean IoU of the test images – 87.0%

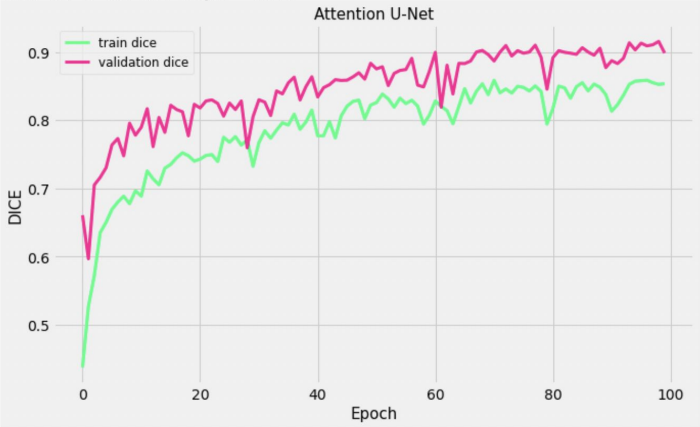


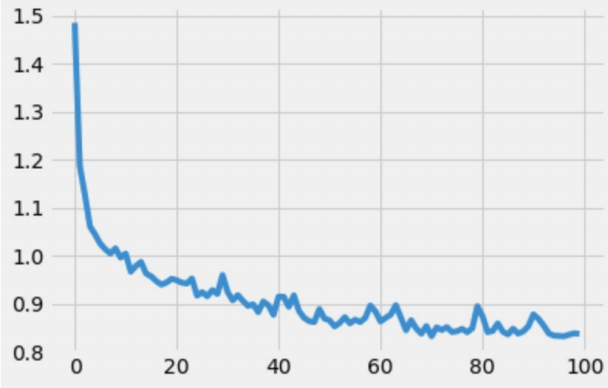



Avg. test IOU [%]
U3Net
Attention R2Unet
R2UNet
Attention U-Net
U-Net
78
79
80
81
82
83
84
85
86
87
88

Some variations with number of Epochs and Loss function we tried

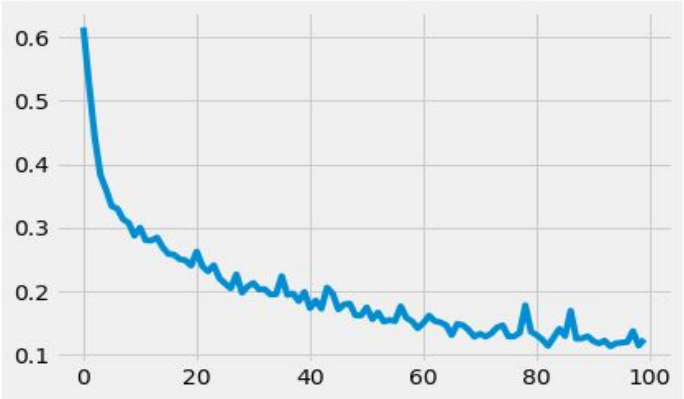
Attention U-Net
Mean IoU of the test images - 88.0%





Dice + BCE

R2U-Net
Mean IoU of the test images - 94.0%



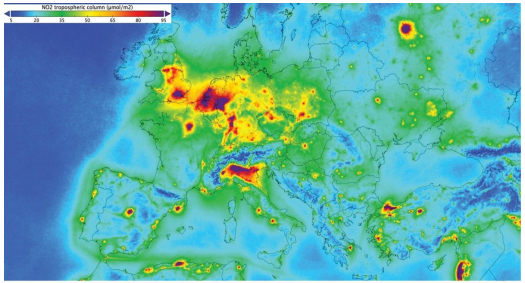
Dice

APPLICATION IN OTHER DOMAINS

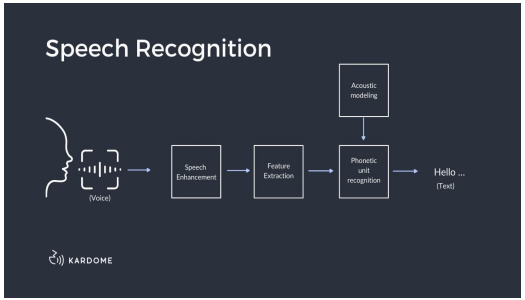
APPLICATION IN OTHER DOMAINS

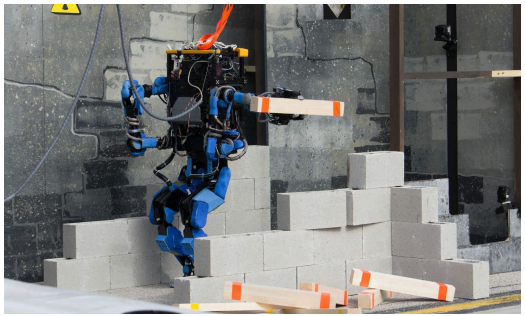
- Environmental Monitoring
- Speech Recognition
- Geology & Cosmology
- Robotics

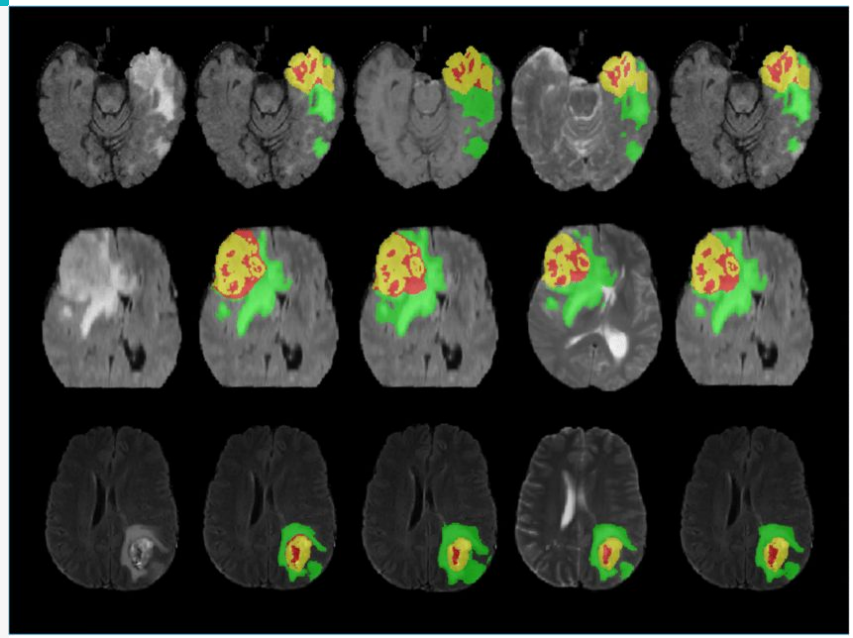
Due to their versatility and flexibility, U-Nets can be used for many different tasks.



According to the Air Quality in Europe report published in 2018 by the European Environment Agency (EEA), 19 EU Member States recorded nitrogen dioxide concentration above the annual permissible limit. Imagery from Sentinel-5P







THANKS!

Any questions?