FLAIR Brain Tumor segmentation with DNN: from U-Net architecture to Vision Transformers

Matteo Liotta
SM3800072
Deep Learning, 2025, UniTS - *Final Project*

Tumor individuation and segmentation

Cancer was the **second leading cause** of death in EU in 2021, with a percentage of 21.6 total number of deaths [1].

In 2015 United States estimated the emergence of 23000 new brain related cases in its territory.

Now the most common treatments for the brain tumor is the **surgery**, but its not always an easy to execute solution [2].

Even if some tumor as meningiomas can be easily segmented, there are other types which could be **hardly** localized: poorly contrasted, usually diffused, with tentacle-like structures.

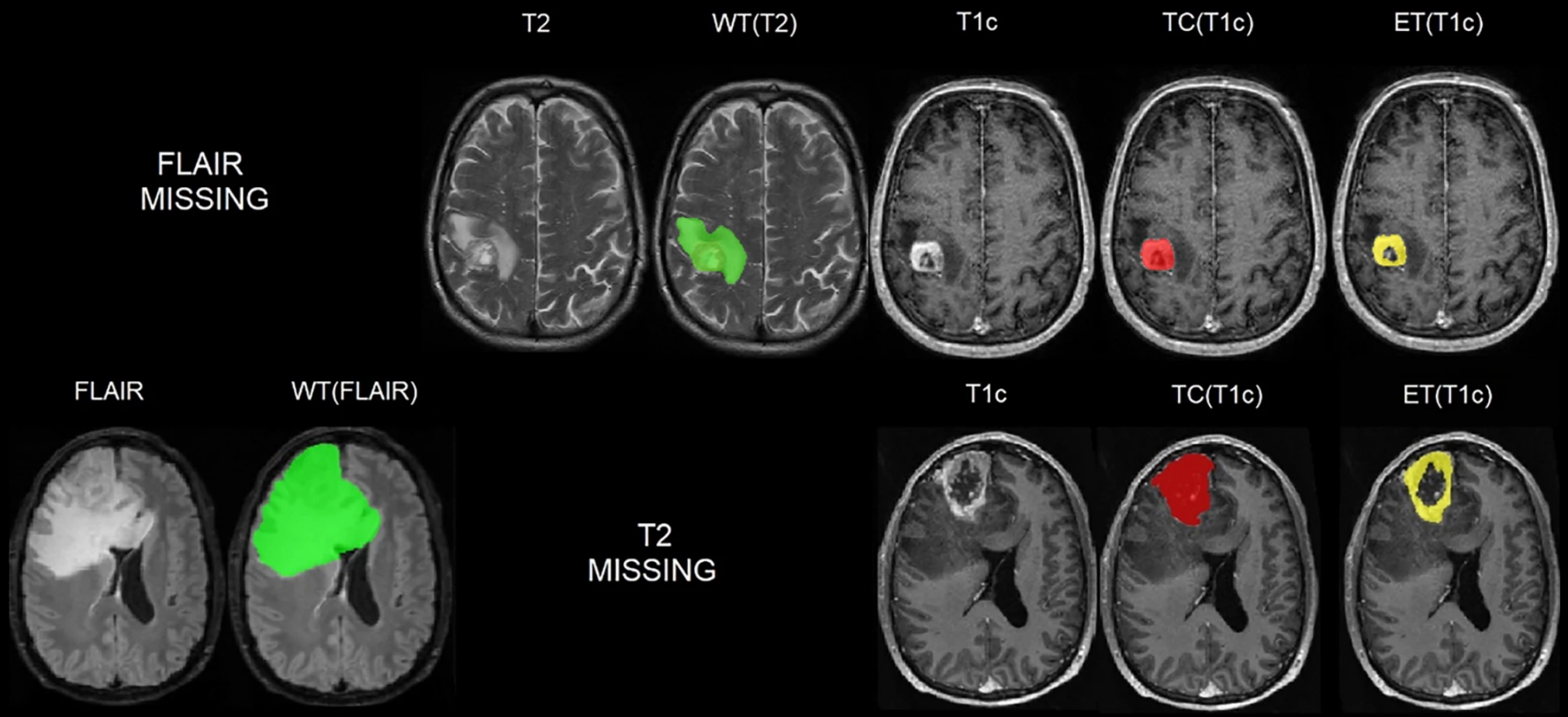


Fig 1. Sample images. Ground truth manual segmentation for a GBM patient with missing FLAIR scan (top row) and one with missing T2w (bottom row). Whole Tumor (WT) in green.

Deep Learning approaches for the segmentation task

The idea is to introduce an exploration in DL models for efficient tumoral region segmentation:

- We need input **images** (dependent on the dataset used)
- To produce **binary segmentation mask** output

Some dataset problem choice emerged:

- The **image size** increase with the **quality** and image definition
- The **dataset size** increase with the **quality** of its instances and instances **number**

The proposed dataset is just some gigabytes heavy, while professional ones are about ~ 100 GB.

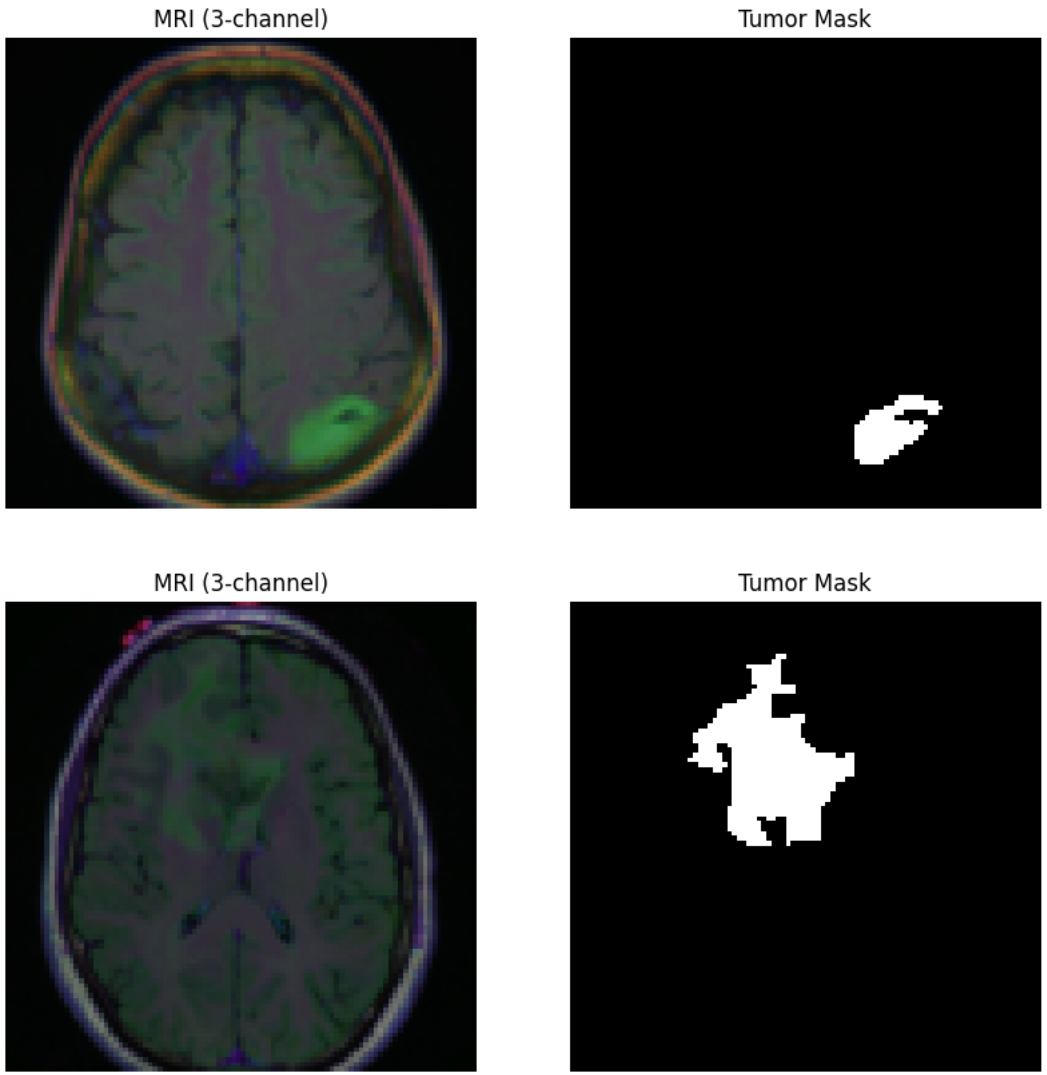
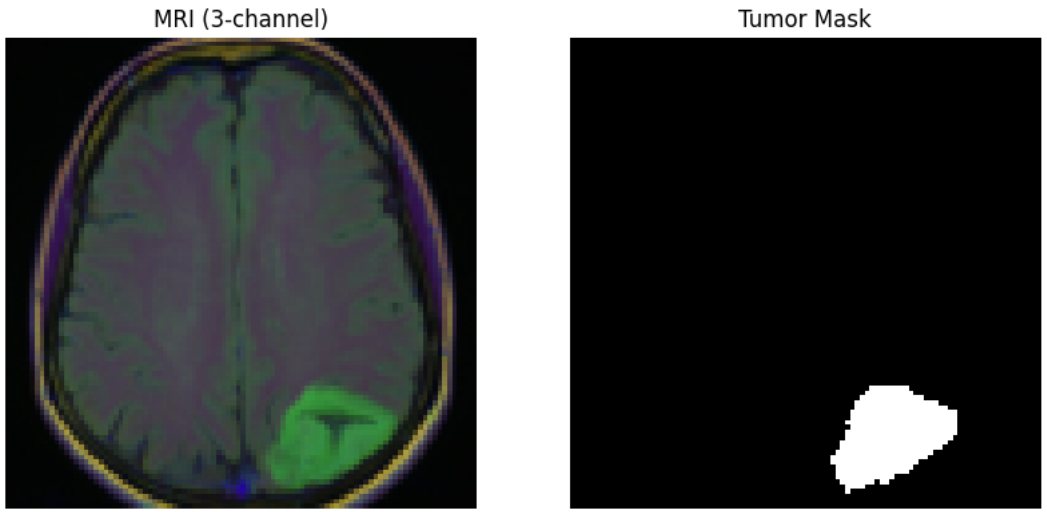


Fig 2. Dataset [4] Sample images.



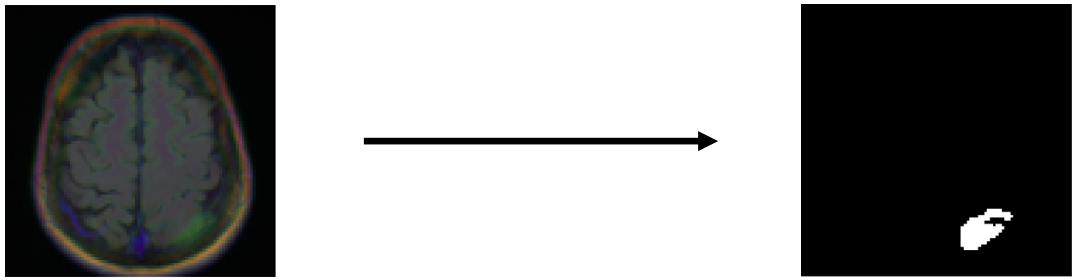
This dataset contains ~4000 brain MR images together with manual FLAIR abnormality segmentation masks.

They correspond to 110 patients included in The Cancer Genome Atlas (TCGA) lower-grade glioma collection with at least fluid-attenuated inversion recovery (FLAIR) sequence […].

Problem Formalization

We can formulate the actual network independently of the architecture as a function

$$model : M_{NxNx3}(\mathbb{R}) \longrightarrow M_{NxNx1}(\mathbb{Z}_2)$$



Mapping an order 3 tensor into a binary matrix.

Problem Formalization

Even if… in all considered models we'll be returning with a **sigmoid** transformation:

$$model: M_{NxNx3}(\mathbb{R}) \longrightarrow M_{NxNx1}([0,1])$$

If so each $m_{i,j}$ we can derive the proper model definition and pixel output interpretation

$$with\ M = M_{NxNx1}([0,1]), m_{i,j} \in M \qquad m_{i,j} \sim p(M_{i,j} = tumoral)$$

$$proper\ model \coloneqq \mathbb{I}(model(M) \geq threshold)$$

So,

$$proper\ model: M_{NxNx3}(\mathbb{R}) \longrightarrow M_{NxNx1}(\mathbb{Z}_2)$$

Loss Choice

Optimality depends not only on the right architecture choice but also on an appropriate choice of the loss function.

Different loss functions have been proposed in literature w.r.t. different behaviors of interest.

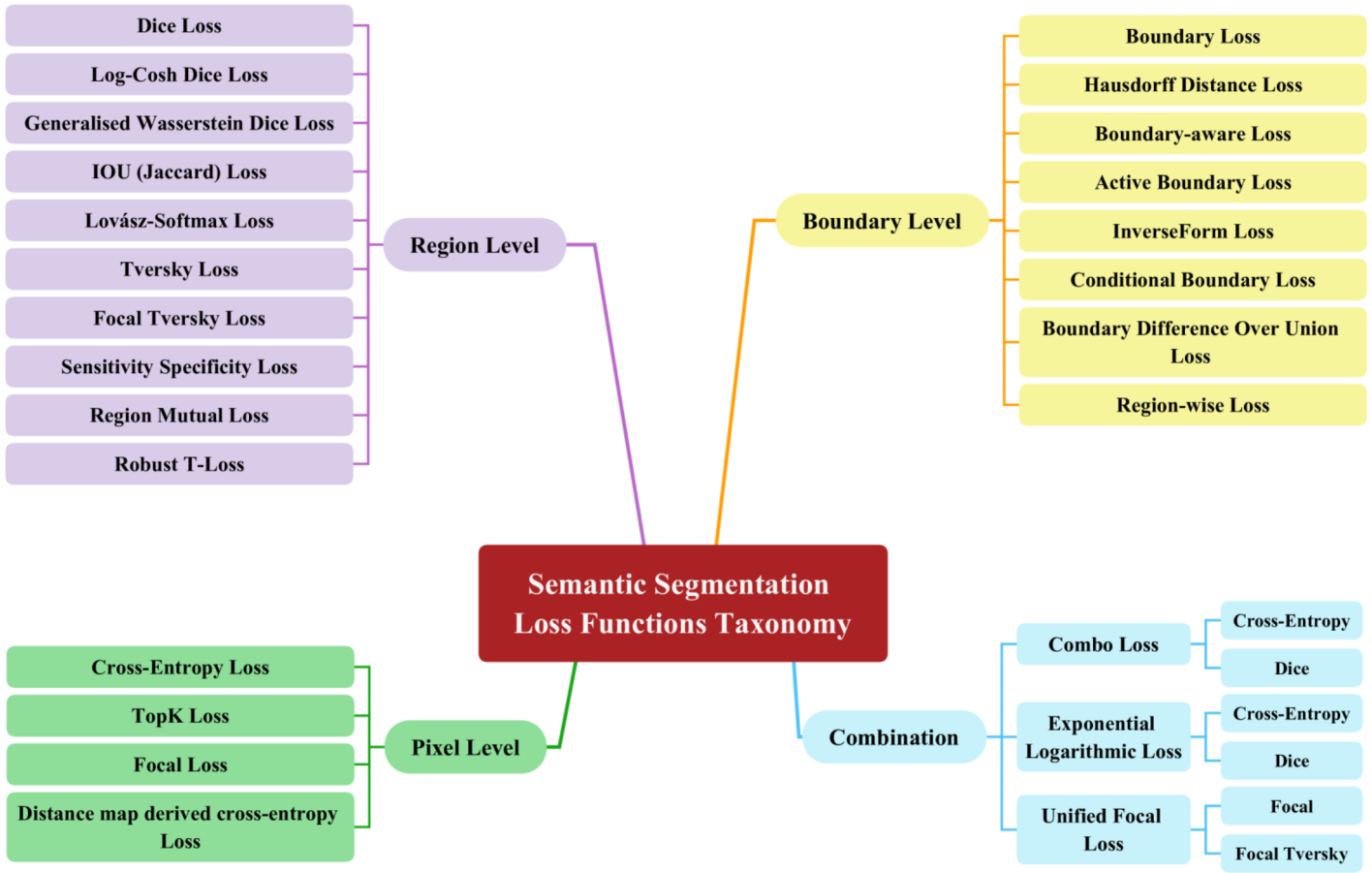


Fig 2. *The loss taxonomy subsections*

Loss Choice

Pixel Level losses

Considering individual pixels to achieve high accuracy

Boundary Level losses

Specialized in the of object boundaries with sharpen object separate overlapping objects aims

Region Level losses

Focused on the overall accuracy of the object segmentation itself, by capturing the essence of the object's shape and layout

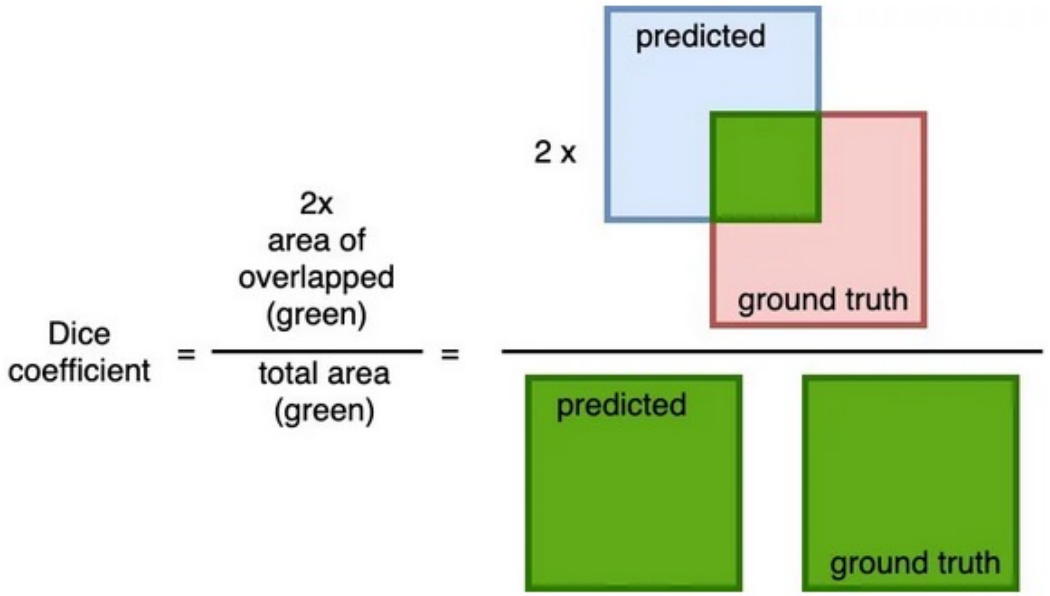
Combination losses

the idea is to harmonize elements from the previous ones and optimize the segmentation performance.

Loss Choice

The focus will be on **Dice** and **Combo** loss, respectively from pixel level and combination losses

- $Dice\ Loss = 1\ - Dice\ coefficient$



Loss Choice

The focus will be on **Dice** and **Combo** loss, respectively from pixel level and combination losses

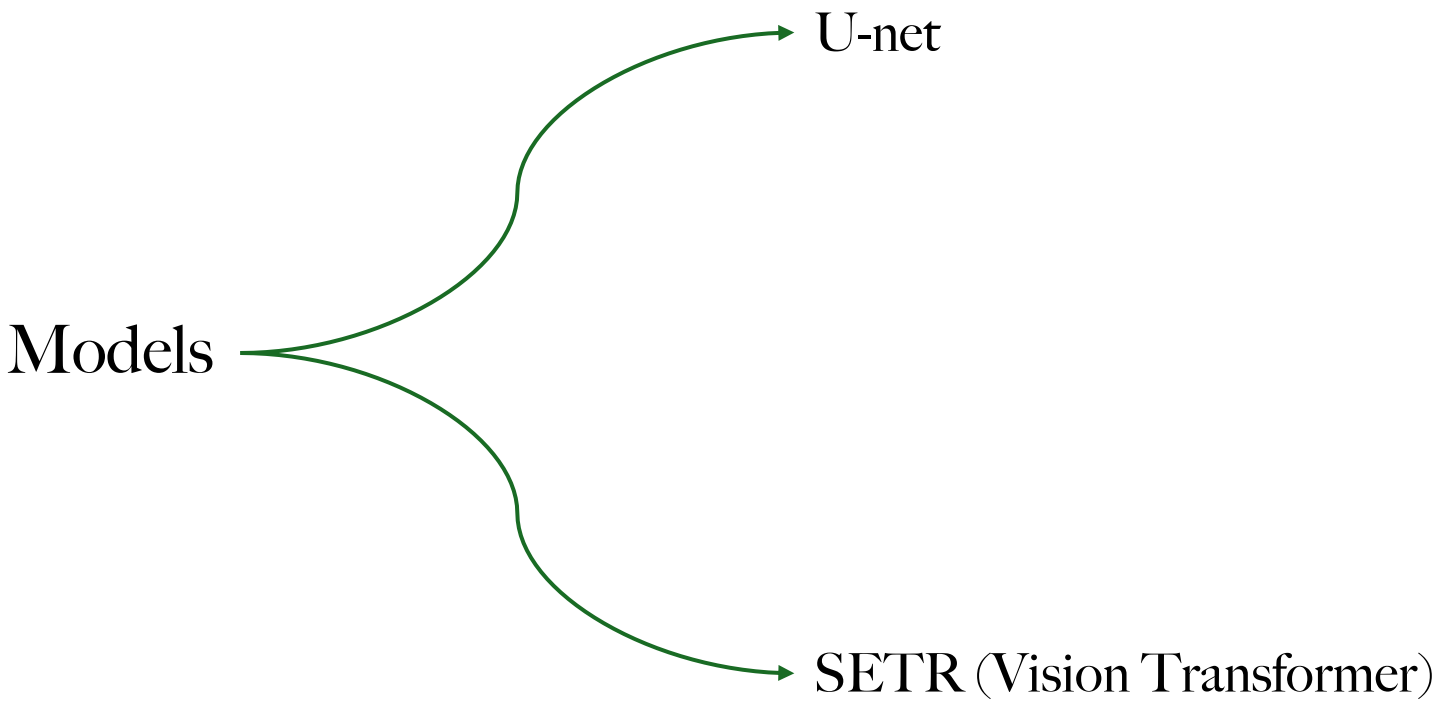
- Combo Loss as *weighted average* of BCE loss and dice

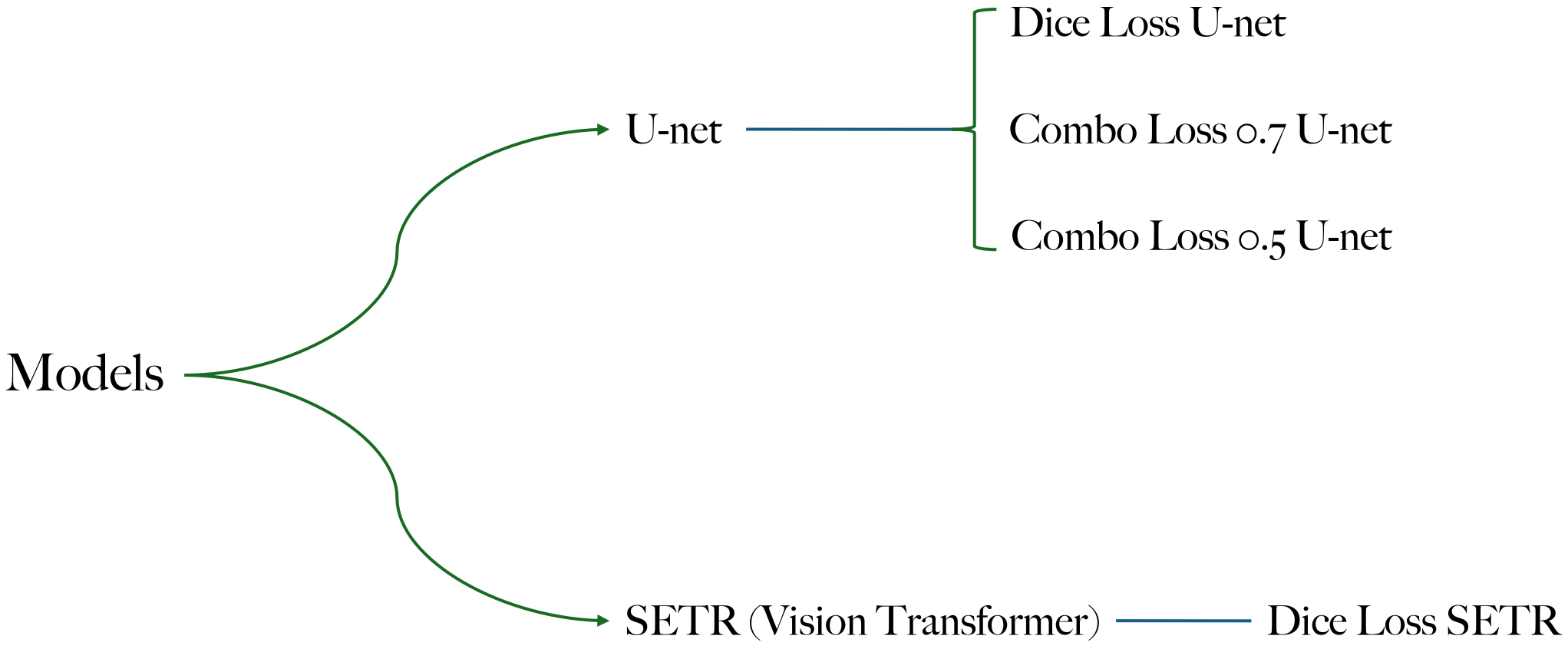
For $y_i, \hat{y}$ generic ground truth and predicted output

$$Combo\ loss(y, \hat{y}) = \alpha \times BCE + (1 - \alpha) \times DiceLoss$$

$$BCE(y, \hat{y}) = -\frac{1}{N}\sum_{i}^{N} y_i \log(\hat{y_i}) \ - (1 - y_i) \log(1 - \hat{y_i})$$

Models


Models
U-net
SETR (Vision Transformer)


Models
U-net
Dice Loss U-net
Combo Loss 0.7 U-net
Combo Loss 0.5 U-net
SETR (Vision Transformer)
Dice Loss SETR

Models: U-net

The deep convolutional networks excels in **visual recognition tasks**

Those were originally devoted to the classification task, with the prediction of a $\mathbb{Z}_k$ **output**

Now we need to predict a binary mask, i.e. a 2D image

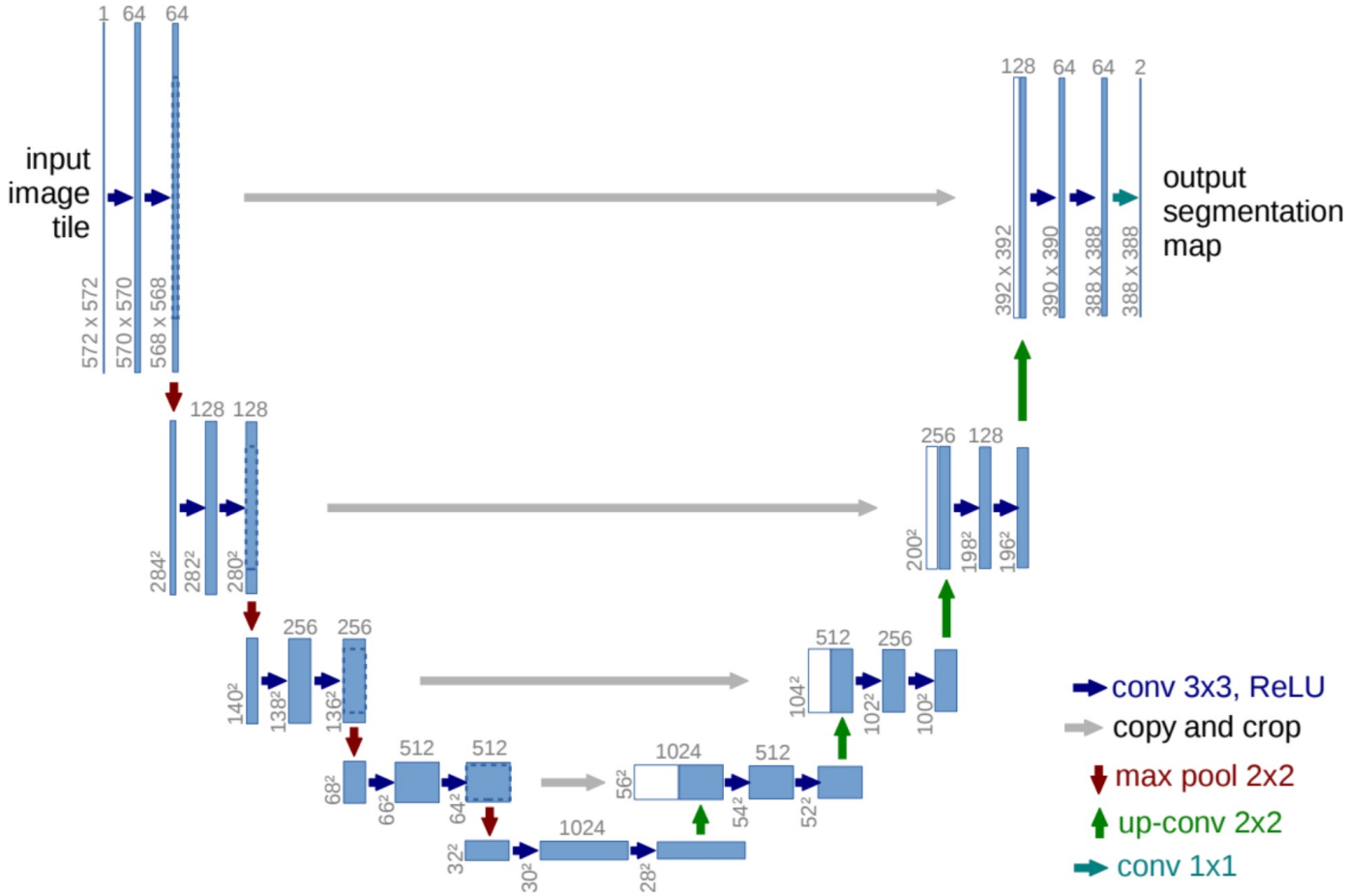
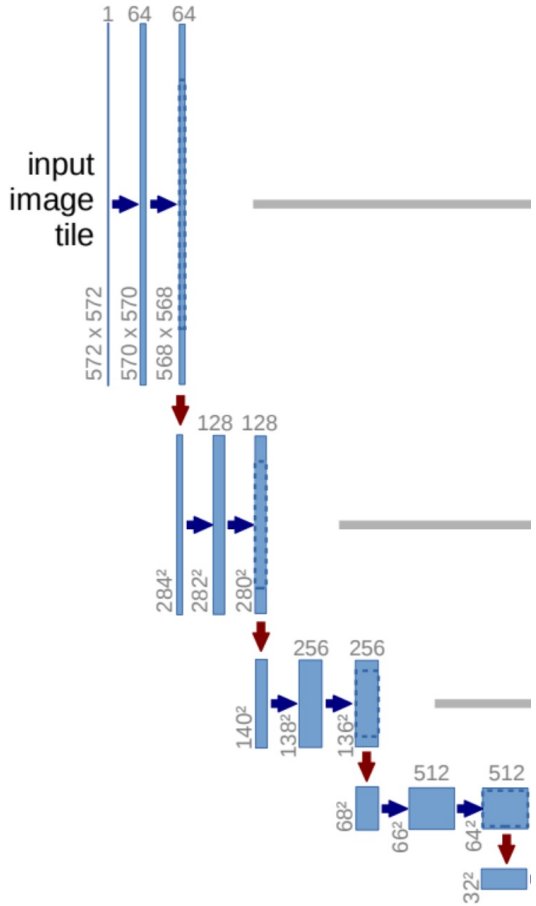


Fig. 6 U-net architecture [6]

Models: U-net

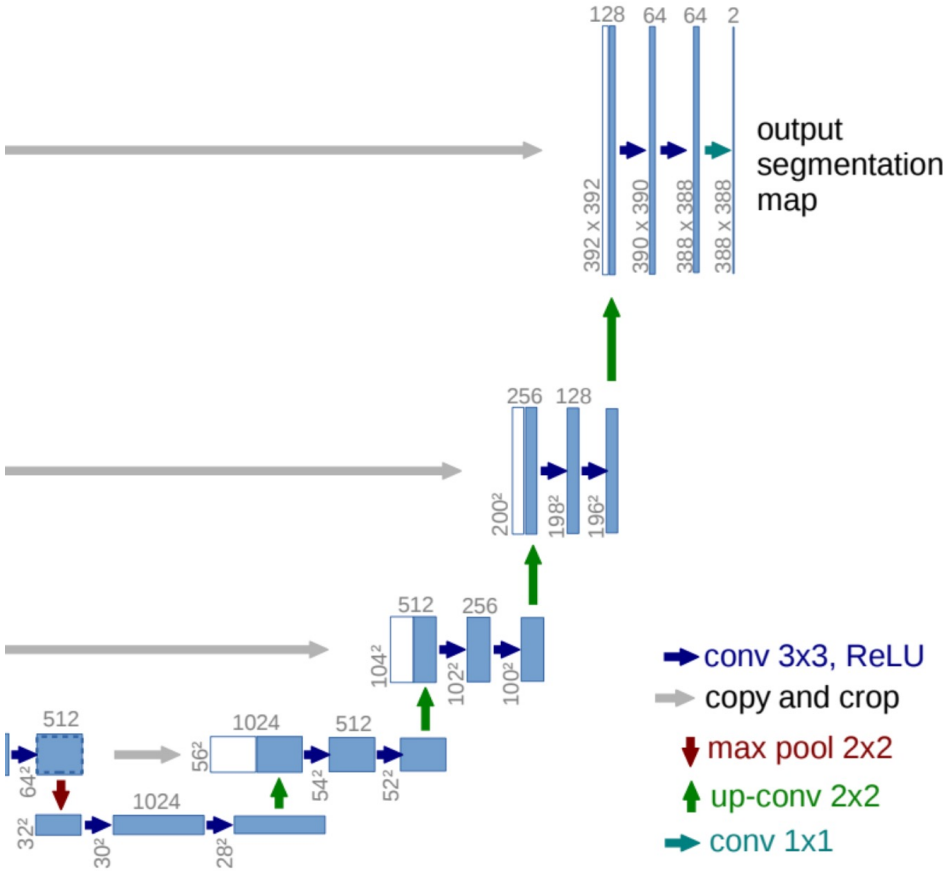


The network does a convolution for some layers

It reduces and modifies the original input dimensions and channel number

Results of each layer are **saved** for a later use

Models: U-net



To reconstruct the output mask the model does some deconvolution steps

Notice that the U shape comes due to the **reuse of previous layer output**: they are now attached to each new layer input

```
class UNet(nn.Module):
def __init__(self, in_channels=3, out_channels=1):
        super().__init__()
        # We need to divide the model in different sequential parts, since we do remember
        # that the U-net implements the copy-and-crop skip connections.
        ## ENCODING (down)
        self.enc1 = nn.Sequential(
        nn.Conv2d(in_channels, 64, 3, padding=1), nn.ReLU(),
        nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())
        self.pool1 = nn.MaxPool2d(2)        self.enc2 = nn.Sequential(
        nn.Conv2d(64, 128, 3, padding=1), nn.ReLU(),
        nn.Conv2d(128, 128, 3, padding=1), nn.ReLU())
        self.pool2 = nn.MaxPool2d(2)

        ## We reach the end of the U
        self.uend = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.ReLU(), nn.Conv2d(256, 256, 3, padding=1), nn.ReLU())

        ## DECODING (up)
        # It's the opposite of the previous part: remember that the mask
        # is needed of the same size of the input image!
        self.up2 = nn.ConvTranspose2d(256, 128, 2, stride=2)
        self.dec2 = nn.Sequential(
        nn.Conv2d(256, 128, 3, padding=1), nn.ReLU(),
        nn.Conv2d(128, 128, 3, padding=1), nn.ReLU())
        self.up1 = nn.ConvTranspose2d(128, 64, 2, stride=2)
        self.dec1 = nn.Sequential(
        nn.Conv2d(128, 64, 3, padding=1), nn.ReLU(),
        nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())
        self.out_conv = nn.Conv2d(64, out_channels, 1)

def forward(self, x):
        # The step is to do encoding and pooling up to the end of the U
        e1 = self.enc1(x)
        p1 = self.pool1(e1)
        e2 = self.enc2(p1)
        p2 = self.pool2(e2)
        b = self.uend(p2)
        # Then we need to do the up part, but with the skip connections.
        # So, previous inputs are attached to the current ones ()
        u2 = self.up2(b)
        c2 = torch.cat([e2, u2], dim=1)
        d2 = self.dec2(c2)
        u1 = self.up1(d2)
        c1 = torch.cat([e1, u1], dim=1)
        d1 = self.dec1(c1)
        out = self.out_conv(d1)
        return torch.sigmoid(out)
```

```
class UNet(nn.Module):
def __init__(self, in_channels=3, out_channels=1):
       super().__init__()
       # We need to divide the model in different sequential parts, since we do remember
       # that the U-net implements the copy-and-crop skip connections.
       ## ENCODING (down)
       self.enc1 = nn.Sequential(
       nn.Conv2d(in_channels, 64, 3, padding=1), nn.ReLU(),
       nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())
       self.pool1 = nn.MaxPool2d(2)        self.enc2 = nn.Sequential(
       nn.Conv2d(64, 128, 3, padding=1), nn.ReLU(),
       nn.Conv2d(128, 128, 3, padding=1), nn.ReLU())
       self.pool2 = nn.MaxPool2d(2)
```

Definition of the convolution phase

Sequence of convolution with 64 in 64 channels layers with ReLU activation and max pooling

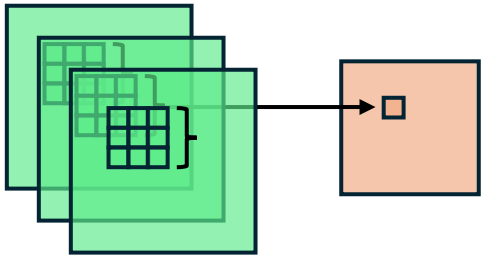
```
class UNet(nn.Module):
def __init__(self, in_channels=3, out_channels=1):
       super().__init__()
       # We need to divide the model in different sequential parts, since we do remember
       # that the U-net implements the copy-and-crop skip connections.
       ## ENCODING (down)
       self.enc1 = nn.Sequential(
       nn.Conv2d(in_channels, 64, 3, padding=1), nn.ReLU(),
       nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())
       self.pool1 = nn.MaxPool2d(2)        self.enc2 = nn.Sequential(
       nn.Conv2d(64, 128, 3, padding=1), nn.ReLU(),
       nn.Conv2d(128, 128, 3, padding=1), nn.ReLU())
       self.pool2 = nn.MaxPool2d(2)
```

Definition of the convolution phase

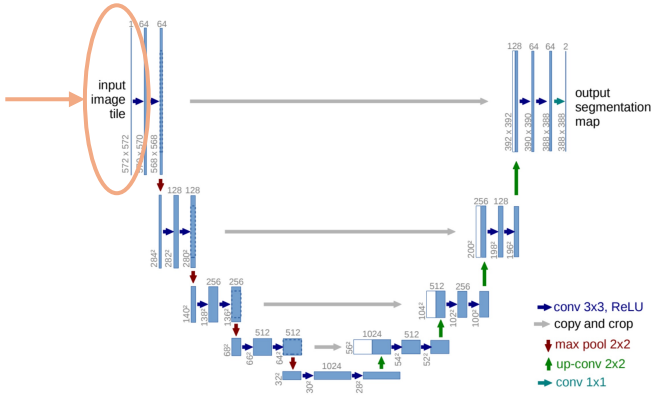
The path is divided into same-channel and doubled + maxpooling steps

Definition of the convolution phase

3 x 3 x 3 kernels applied with same spatial dimension due to padding 1

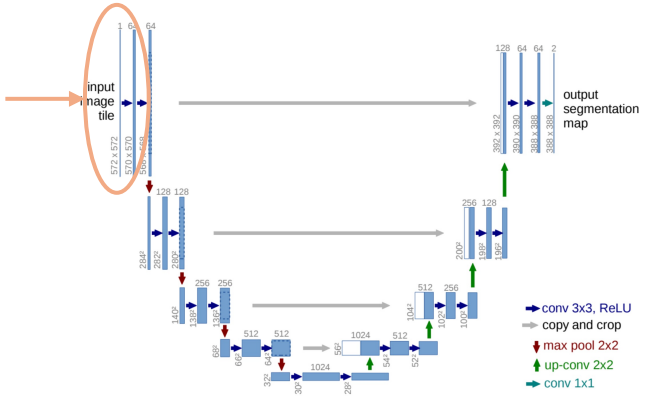


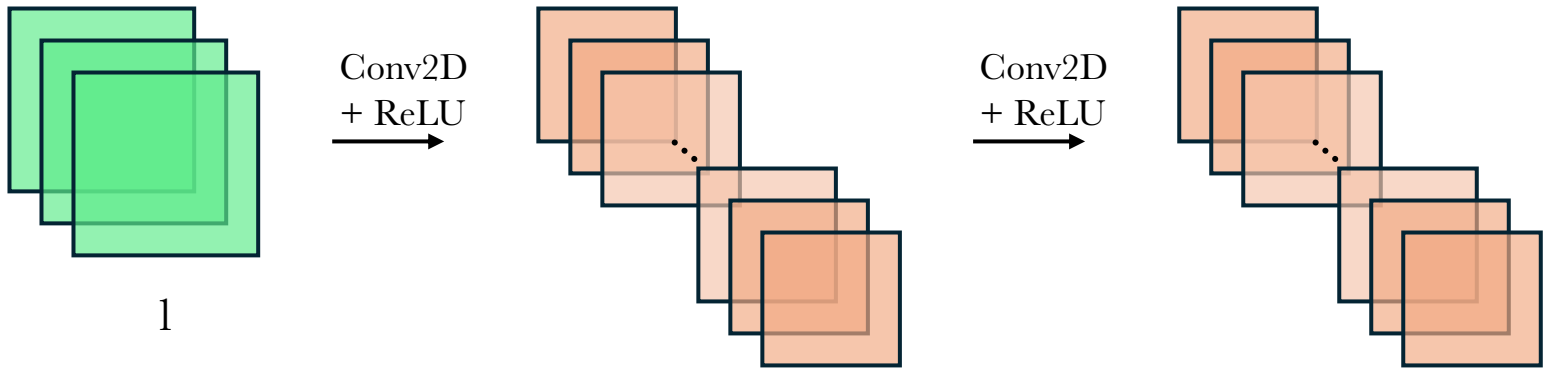
64 times

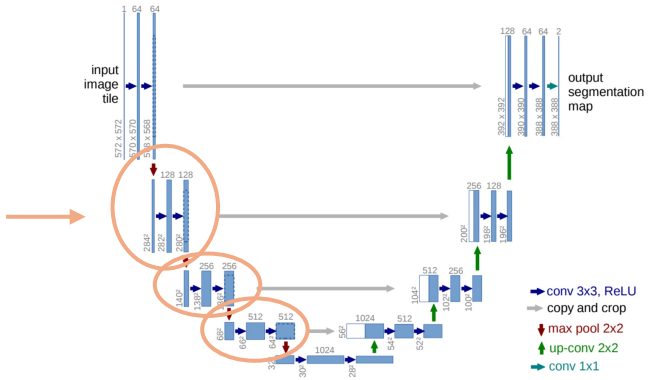


Definition of the convolution phase

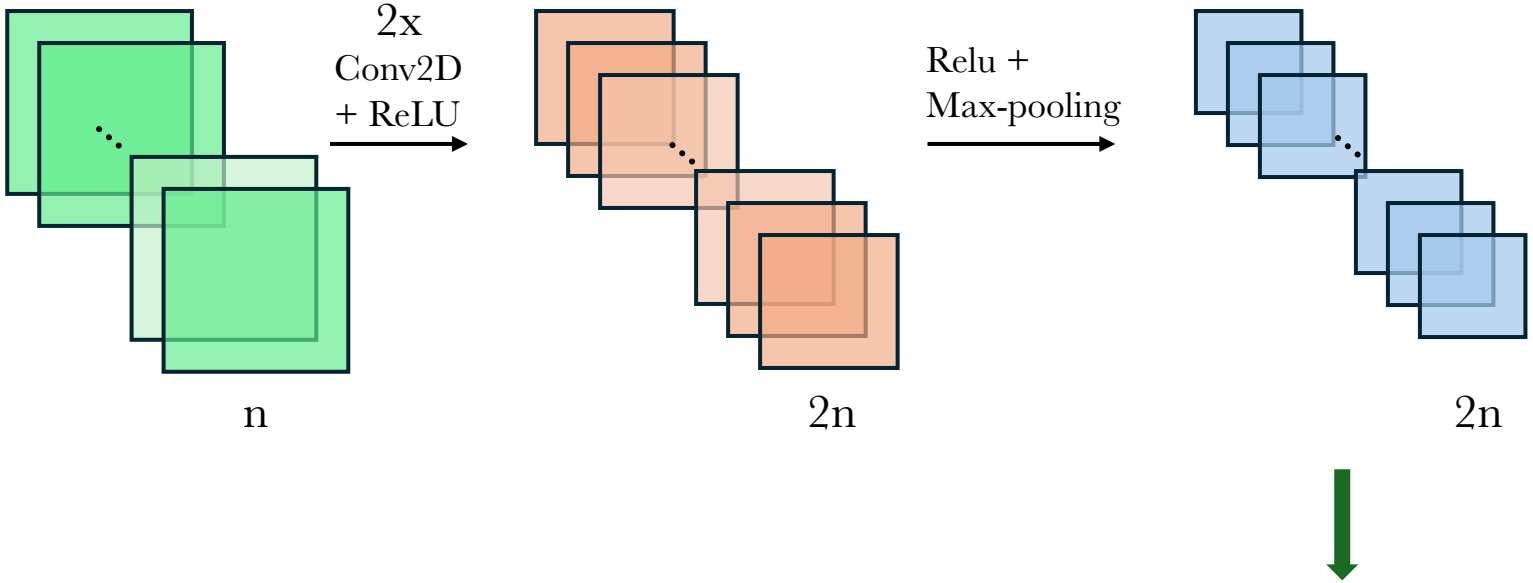
At the beginning we obtain 64 output channels
but at each depth level we double the number of channels



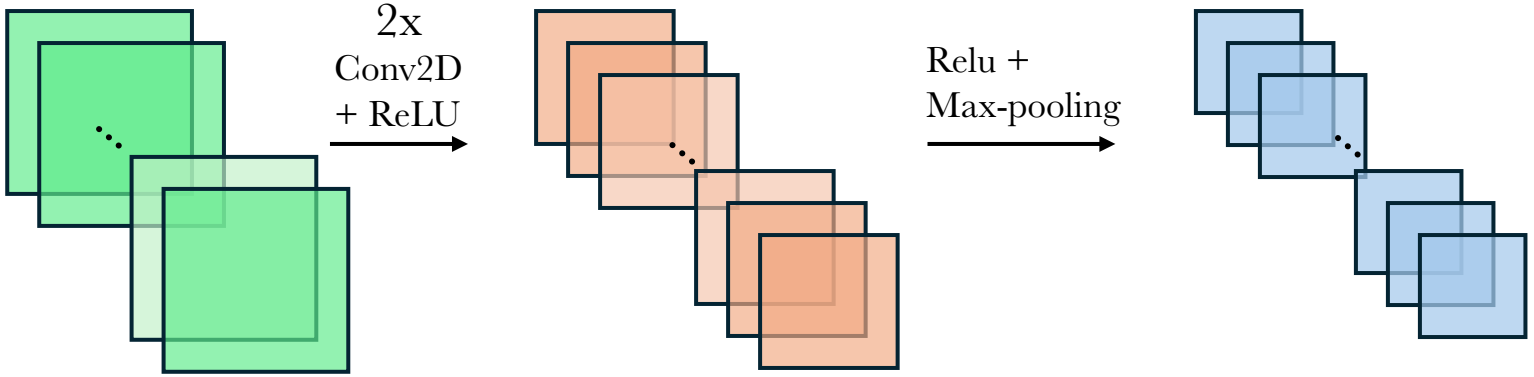




U-Net
Convolution
«depth level»



U-Net
Convolution
«depth level»



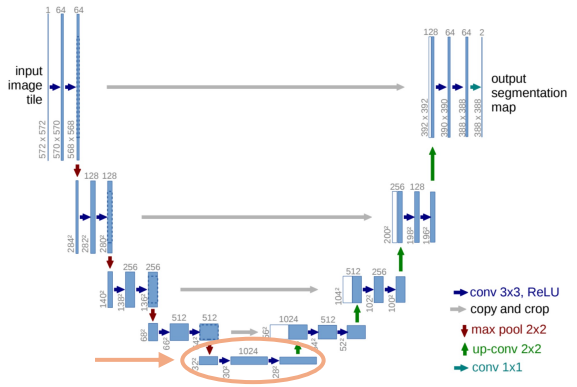
Remember that this will be pre-appended to the same level first deconvolution input

```
class UNet(nn.Module):
def __init__(self, in_channels=3, out_channels=1):
        super().__init__()
        # We need to divide the model in different sequential parts, since we do remember
        # that the U-net implements the copy-and-crop skip connections.
        ## ENCODING (down)
        self.enc1 = nn.Sequential(
        nn.Conv2d(in_channels, 64, 3, padding=1), nn.ReLU(),
        nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())
        self.pool1 = nn.MaxPool2d(2)        self.enc2 = nn.Sequential(
        nn.Conv2d(64, 128, 3, padding=1), nn.ReLU(),
        nn.Conv2d(128, 128, 3, padding=1), nn.ReLU())
        self.pool2 = nn.MaxPool2d(2)

        ## We reach the end of the U
        self.uend = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.ReLU(), nn.Conv2d(256, 256, 3, padding=1), nn.ReLU())

        ## DECODING (up)
        # It's the opposite of the previous part: remember that the mask
        # is needed of the same size of the input image!
        self.up2 = nn.ConvTranspose2d(256, 128, 2, stride=2)
        self.dec2 = nn.Sequential(
        nn.Conv2d(256, 128, 3, padding=1), nn.ReLU(),
        nn.Conv2d(128, 128, 3, padding=1), nn.ReLU())
        self.up1 = nn.ConvTranspose2d(128, 64, 2, stride=2)
        self.dec1 = nn.Sequential(
        nn.Conv2d(128, 64, 3, padding=1), nn.ReLU(),
        nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())
        self.out_conv = nn.Conv2d(64, out_channels, 1)

def forward(self, x):
        # The step is to do encoding and pooling up to the end of the U
        e1 = self.enc1(x)
        p1 = self.pool1(e1)
        e2 = self.enc2(p1)
        p2 = self.pool2(e2)
        b = self.uend(p2)
        # Then we need to do the up part, but with the skip connections.
        # So, previous inputs are attached to the current ones ()
        u2 = self.up2(b)
        c2 = torch.cat([e2, u2], dim=1)
        d2 = self.dec2(c2)
        u1 = self.up1(d2)
        c1 = torch.cat([e1, u1], dim=1)
        d1 = self.dec1(c1)
        out = self.out_conv(d1)
        return torch.sigmoid(out)
```

```
## We reach the end of the U
self.uend = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.ReLU(), nn.Conv2d(256, 256, 3, padding=1), nn.ReLU())
```

Definition of the bottleneck

Sequence of convolution layers and Relu activations

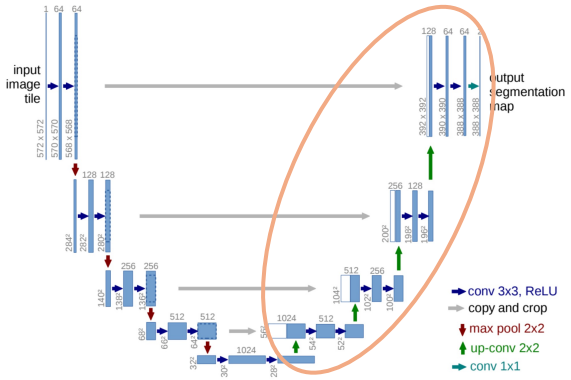


```
class UNet(nn.Module):
def __init__(self, in_channels=3, out_channels=1):
        super().__init__()
        # We need to divide the model in different sequential parts, since we do remember
        # that the U-net implements the copy-and-crop skip connections.
        ## ENCODING (down)
        self.enc1 = nn.Sequential(
        nn.Conv2d(in_channels, 64, 3, padding=1), nn.ReLU(),
        nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())
        self.pool1 = nn.MaxPool2d(2)        self.enc2 = nn.Sequential(
        nn.Conv2d(64, 128, 3, padding=1), nn.ReLU(),
        nn.Conv2d(128, 128, 3, padding=1), nn.ReLU())
        self.pool2 = nn.MaxPool2d(2)

        ## We reach the end of the U
        self.uend = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.ReLU(), nn.Conv2d(256, 256, 3, padding=1), nn.ReLU())

        ## DECODING (up)
        # It's the opposite of the previous part: remember that the mask
        # is needed of the same size of the input image!
        self.up2 = nn.ConvTranspose2d(256, 128, 2, stride=2)
        self.dec2 = nn.Sequential(
        nn.Conv2d(256, 128, 3, padding=1), nn.ReLU(),
        nn.Conv2d(128, 128, 3, padding=1), nn.ReLU())
        self.up1 = nn.ConvTranspose2d(128, 64, 2, stride=2)
        self.dec1 = nn.Sequential(
        nn.Conv2d(128, 64, 3, padding=1), nn.ReLU(),
        nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())
        self.out_conv = nn.Conv2d(64, out_channels, 1)

def forward(self, x):
        # The step is to do encoding and pooling up to the end of the U
        e1 = self.enc1(x)
        p1 = self.pool1(e1)
        e2 = self.enc2(p1)
        p2 = self.pool2(e2)
        b = self.uend(p2)
        # Then we need to do the up part, but with the skip connections.
        # So, previous inputs are attached to the current ones ()
        u2 = self.up2(b)
        c2 = torch.cat([e2, u2], dim=1)
        d2 = self.dec2(c2)
        u1 = self.up1(d2)
        c1 = torch.cat([e1, u1], dim=1)
        d1 = self.dec1(c1)
        out = self.out_conv(d1)
        return torch.sigmoid(out)
```

Definition of the deconvolution phase

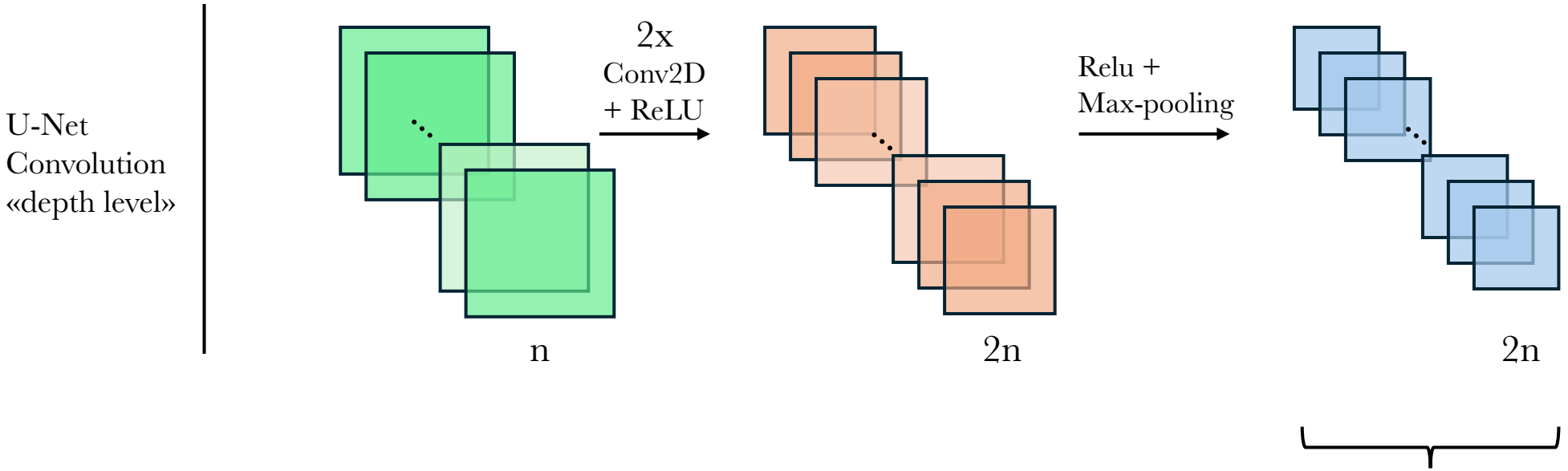
Analogue to the convolution phase with doubled in-channels, because...

```
## DECODING (up)
# It's the opposite of the previous part: remember that the mask
# is needed of the same size of the input image!
self.up2 = nn.ConvTranspose2d(256, 128, 2, stride=2)
self.dec2 = nn.Sequential(
nn.Conv2d(256, 128, 3, padding=1), nn.ReLU(),
nn.Conv2d(128, 128, 3, padding=1), nn.ReLU())
self.up1 = nn.ConvTranspose2d(128, 64, 2, stride=2)
self.dec1 = nn.Sequential(
nn.Conv2d(128, 64, 3, padding=1), nn.ReLU(),
nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())
self.out_conv = nn.Conv2d(64, out_channels, 1)
```



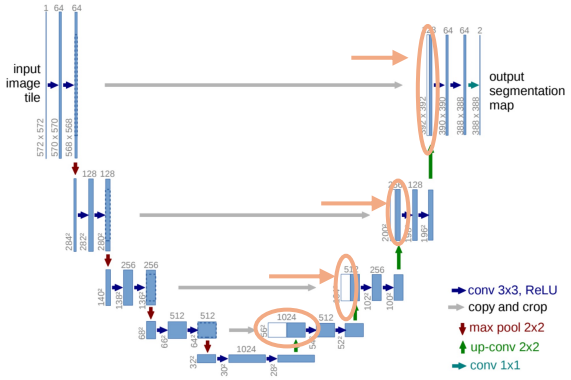
Definition of the deconvolution phase

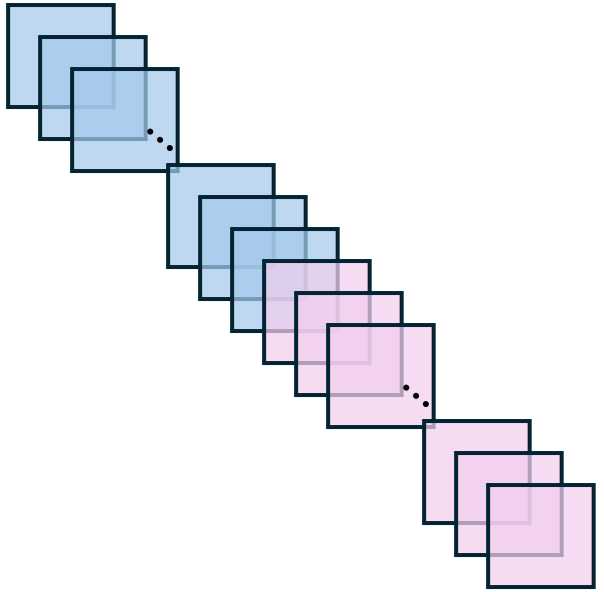
…the previous convolution output will be pre-appended



Definition of the deconvolution phase

The up deconvolution 2x2 with the pre-appending makes the next depth input of the same shape of the final output of the previous depth





```
class UNet(nn.Module):
def __init__(self, in_channels=3, out_channels=1):
        super().__init__()
        # We need to divide the model in different sequential parts, since we do remember
        # that the U-net implements the copy-and-crop skip connections.
        ## ENCODING (down)
        self.enc1 = nn.Sequential(
        nn.Conv2d(in_channels, 64, 3, padding=1), nn.ReLU(),
        nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())
        self.pool1 = nn.MaxPool2d(2)        self.enc2 = nn.Sequential(
        nn.Conv2d(64, 128, 3, padding=1), nn.ReLU(),
        nn.Conv2d(128, 128, 3, padding=1), nn.ReLU())
        self.pool2 = nn.MaxPool2d(2)

        ## We reach the end of the U
        self.uend = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.ReLU(), nn.Conv2d(256, 256, 3, padding=1), nn.ReLU())

        ## DECODING (up)
        # It's the opposite of the previous part: remember that the mask
        # is needed of the same size of the input image!
        self.up2 = nn.ConvTranspose2d(256, 128, 2, stride=2)
        self.dec2 = nn.Sequential(
        nn.Conv2d(256, 128, 3, padding=1), nn.ReLU(),
        nn.Conv2d(128, 128, 3, padding=1), nn.ReLU())
        self.up1 = nn.ConvTranspose2d(128, 64, 2, stride=2)
        self.dec1 = nn.Sequential(
        nn.Conv2d(128, 64, 3, padding=1), nn.ReLU(),
        nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())
        self.out_conv = nn.Conv2d(64, out_channels, 1)

def forward(self, x):
        # The step is to do encoding and pooling up to the end of the U
        e1 = self.enc1(x)
        p1 = self.pool1(e1)
        e2 = self.enc2(p1)
        p2 = self.pool2(e2)
        b = self.uend(p2)
        # Then we need to do the up part, but with the skip connections.
        # So, previous inputs are attached to the current ones ()
        u2 = self.up2(b)
        c2 = torch.cat([e2, u2], dim=1)
        d2 = self.dec2(c2)
        u1 = self.up1(d2)
        c1 = torch.cat([e1, u1], dim=1)
        d1 = self.dec1(c1)
        out = self.out_conv(d1)
        return torch.sigmoid(out)
```

Forward of the model

Model behavior with previous definitions and sigmoid return.
Here we can see the pre-appending in the deconvolution phase

```
def forward(self, x):
       # The step is to do encoding and pooling up to the end of the U
       e1 = self.enc1(x)
       p1 = self.pool1(e1)
       e2 = self.enc2(p1)
       p2 = self.pool2(e2)
       b = self.uend(p2)
       # Then we need to do the up part, but with the skip connections.
       # So, previous inputs are attached to the current ones ()
       u2 = self.up2(b)
       c2 = torch.cat([e2, u2], dim=1)
       d2 = self.dec2(c2)
       u1 = self.up1(d2)
       c1 = torch.cat([e1, u1], dim=1)
       d1 = self.dec1(c1)
       out = self.out_conv(d1)
       return torch.sigmoid(out)
```

U-Net with Dice Loss
Training Informations

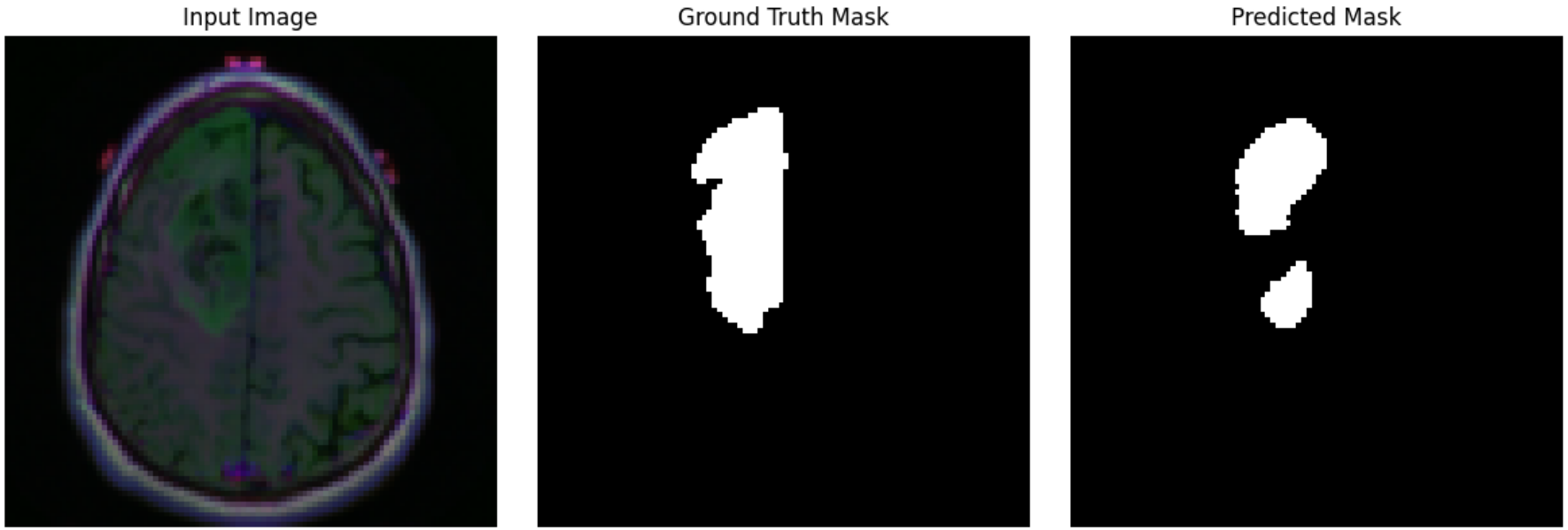
Adam optimizer with 10^{-4} learning rate

~ 20 training epochs

Saved an image and mask prediction, so to make available the training region evolution visualization…

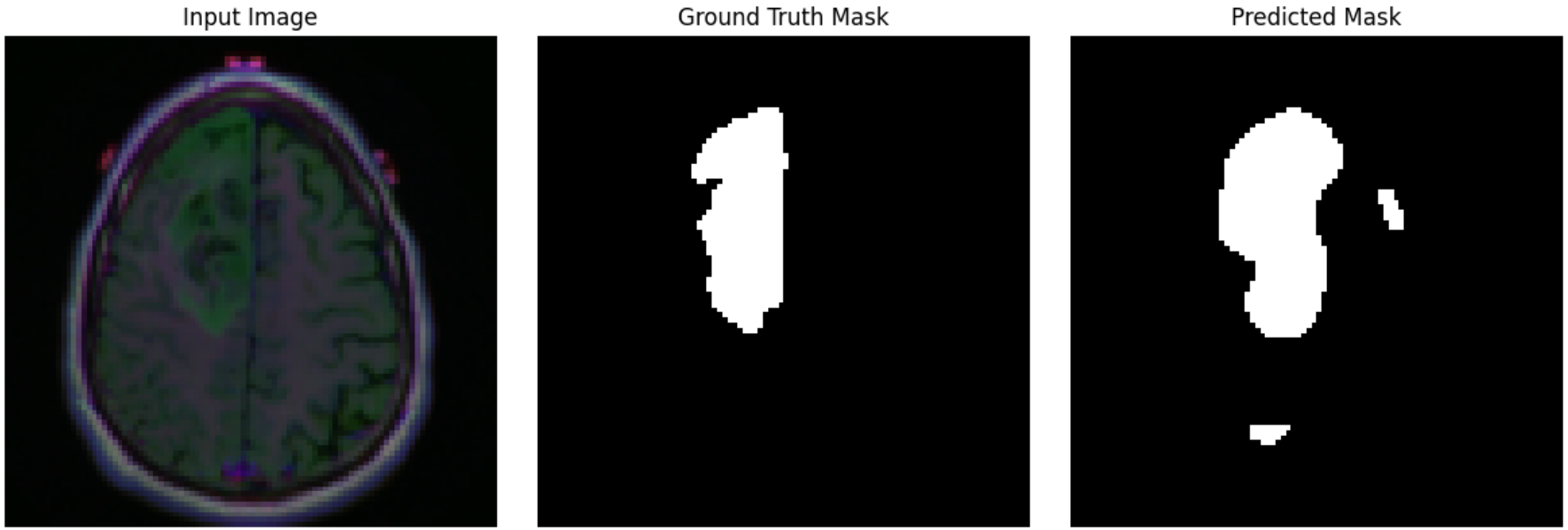
U-Net with Dice Loss

Visualizing the training region evolution



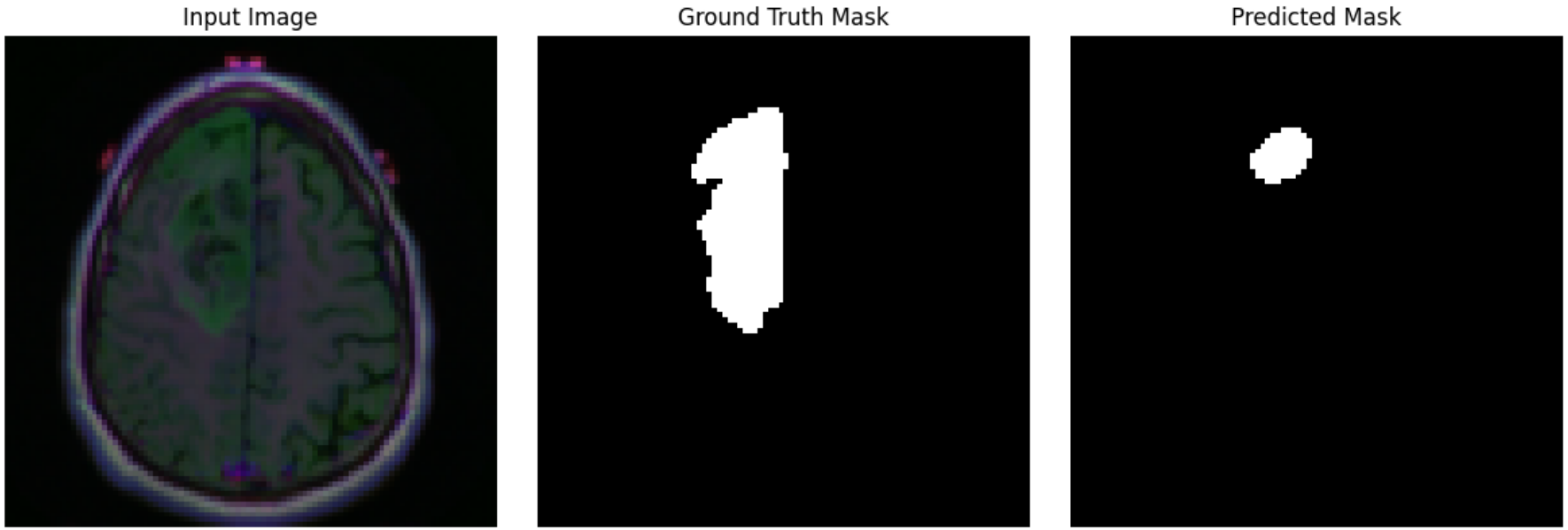
U-Net with Dice Loss

Visualizing the training region evolution



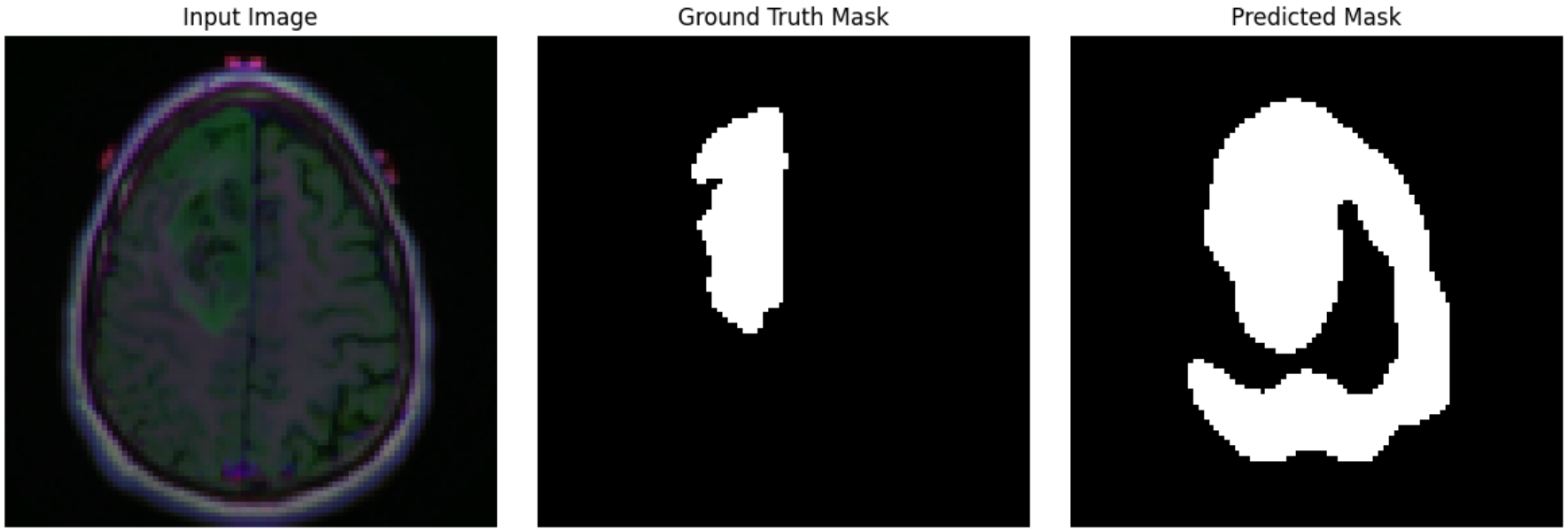
U-Net with Dice Loss

Visualizing the training region evolution



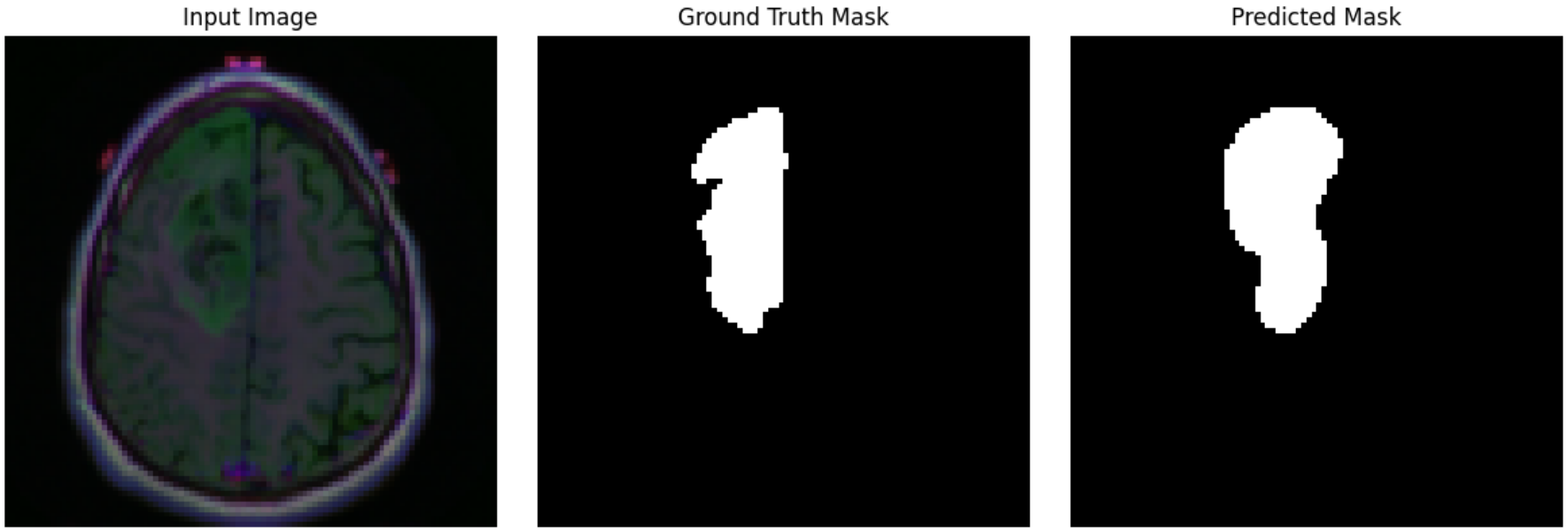
U-Net with Dice Loss

Visualizing the training region evolution



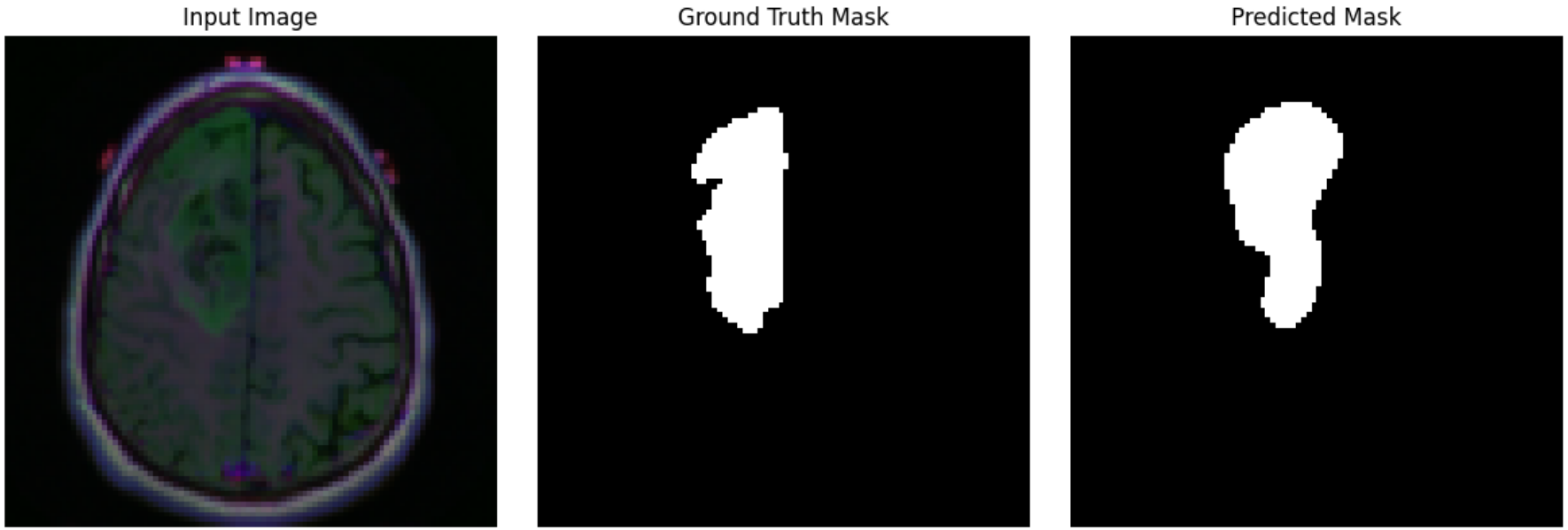
U-Net with Dice Loss

Visualizing the training region evolution



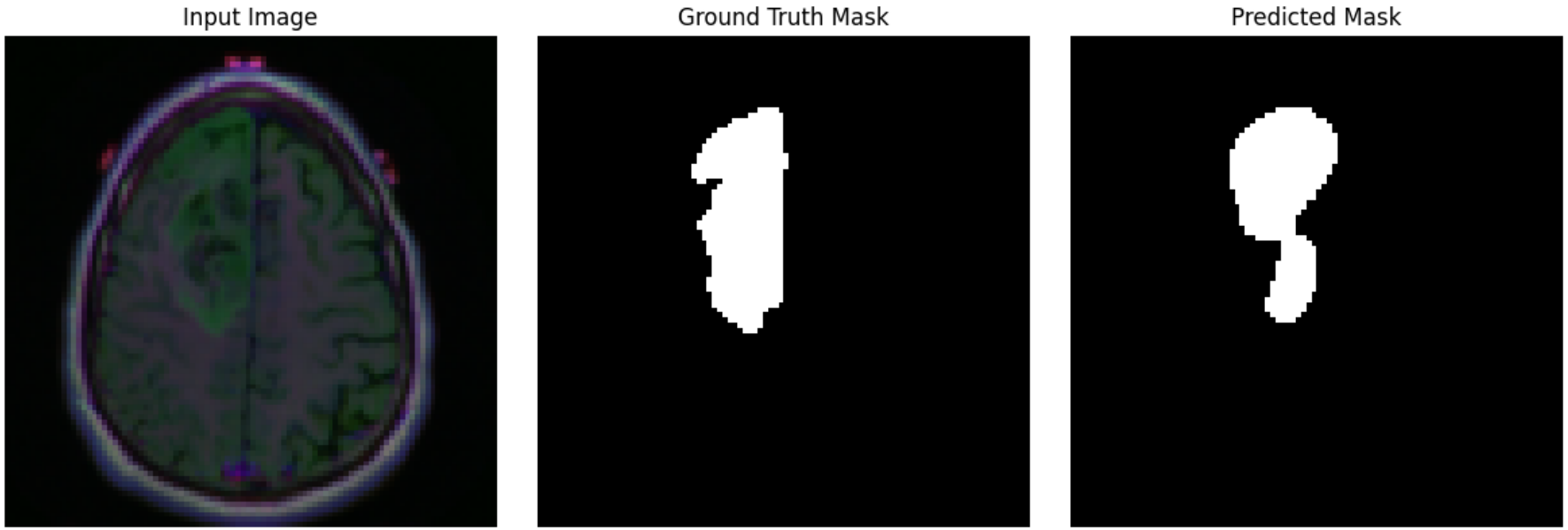
U-Net with Dice Loss

Visualizing the training region evolution



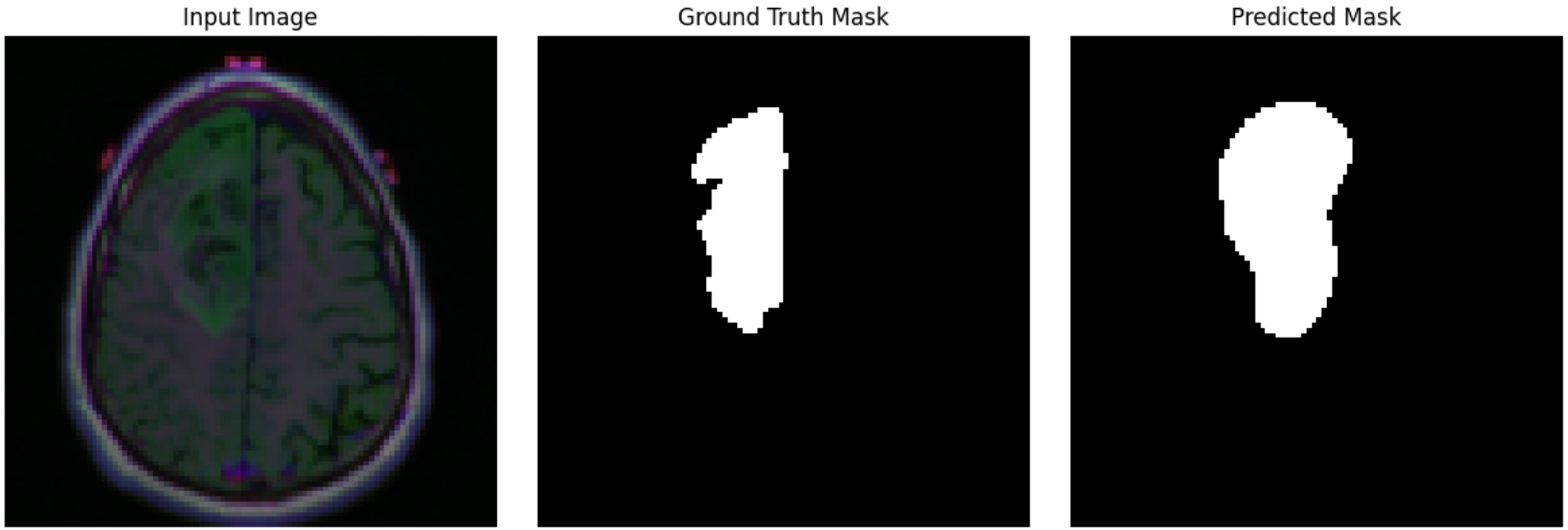
U-Net with Dice Loss

Visualizing the training region evolution



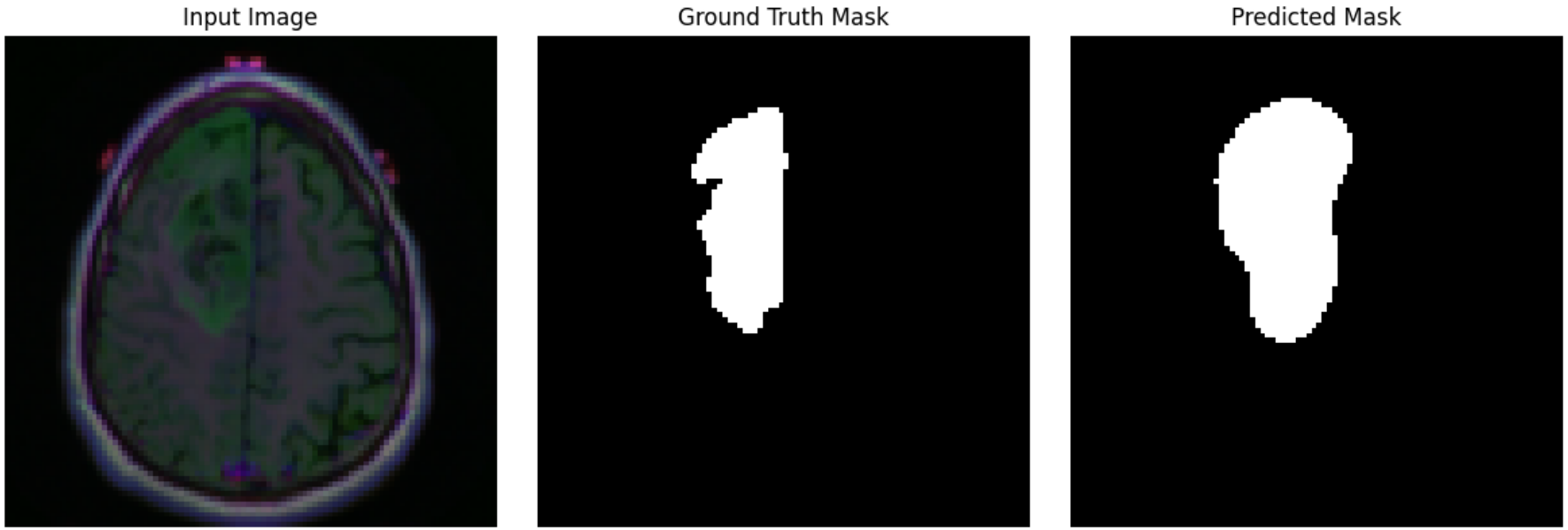
U-Net with Dice Loss

Visualizing the training region evolution



U-Net with Dice Loss

Visualizing the training region evolution



U-Net with Dice Loss

Accuracy results on the test set

Since we'll consider different losses, let's introduce different metrics in order to compare results

1. Pixel Accuracy $\frac{Correct\ pixel\ prediction}{Total\ pixels}$

2. Mean Accuracy $\frac{1}{Total\ Pixel} \times \left(\frac{TP\ class\ 0}{Total\ pixel\ class\ 0} + \frac{TP\ class\ 1}{Total\ pixel\ class\ 1}\right)$

3. IoU $\frac{TP}{TP + FP + FN}$

U-Net with Dice Loss

Accuracy results on the test set

1. Pixel Accuracy

$$\frac{Correct\ pixel\ prediction}{Total\ pixels}$$

2. Mean Accuracy

$$\frac{1}{Total\ Pixel} \times \left(\frac{TP\ class\ 0}{Total\ pixel\ class\ 0} + \frac{TP\ class\ 1}{Total\ pixel\ class\ 1}\right)$$

- $\in [0,1]$
- Random model: 50%
- Zero predictor: ~4% acc. error

3. IoU

$$\frac{TP}{TP + FP + FN}$$

- $\in [0,1]$
- Random model: ~ 10%
- Zero predictor: 0% acc.

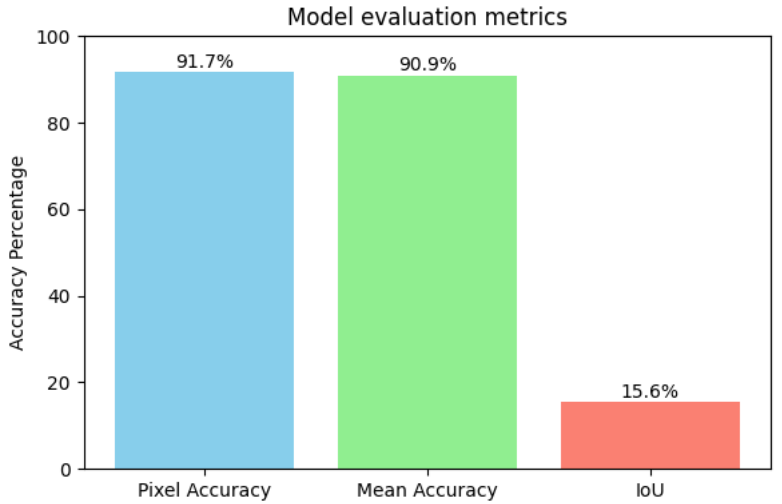
U-Net with Dice Loss

Accuracy results on the test set

1. Pixel Accuracy

2. Mean Accuracy

3. IoU



U-Net with 0.7 CE Combo Loss

Training Informations

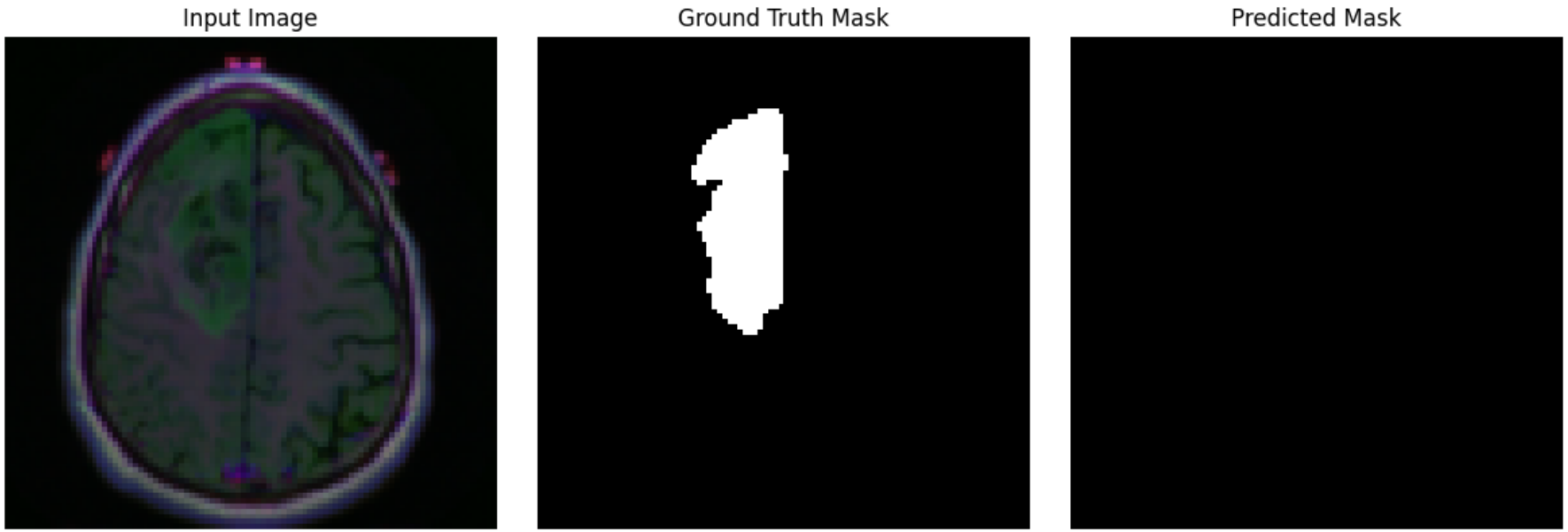
Adam optimizer with 10^{-4} learning rate

~ 20 training epochs

Saved an image and mask prediction, so to make available the training region evolution visualization…

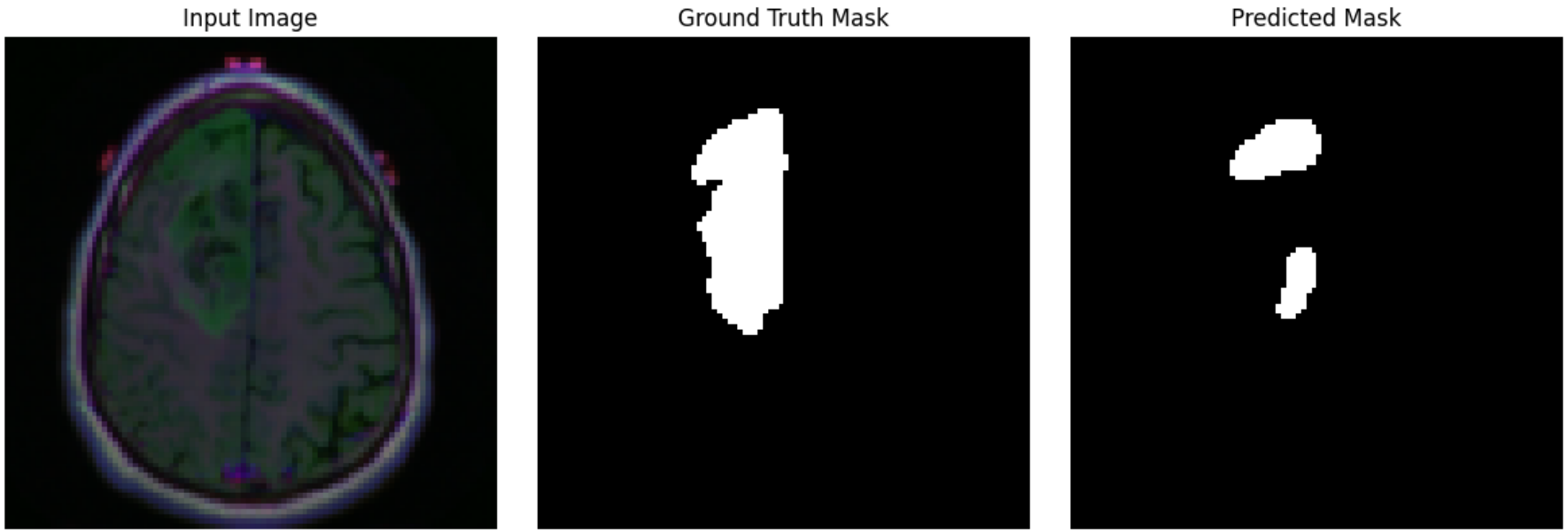
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



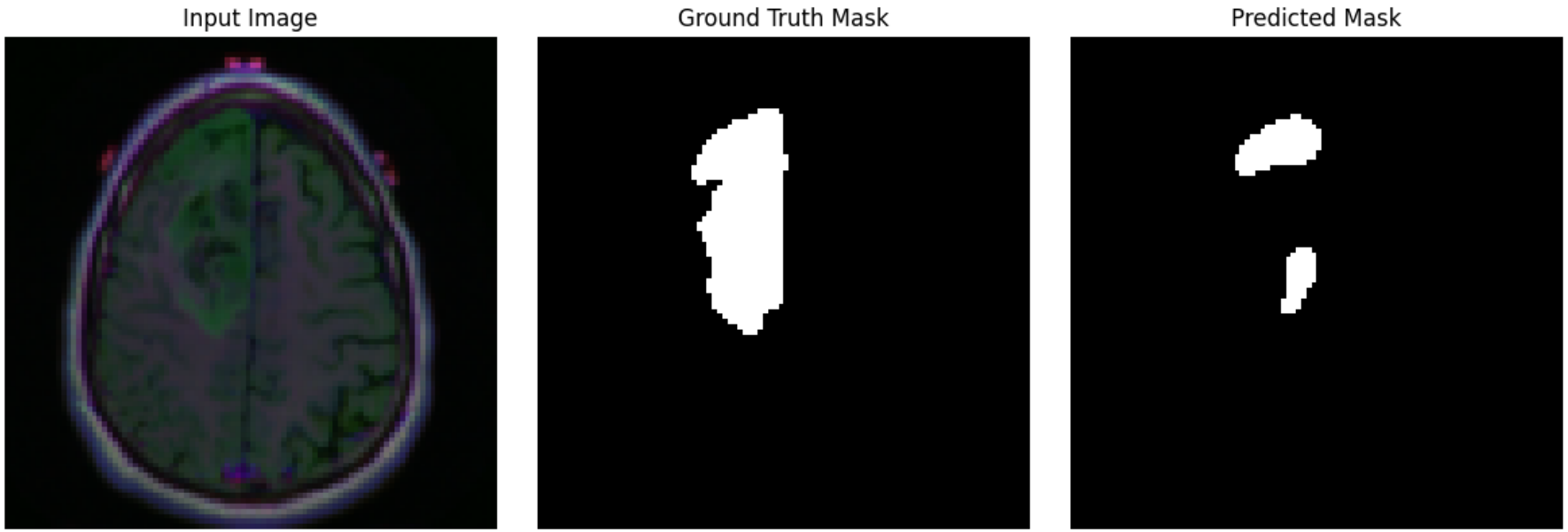
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



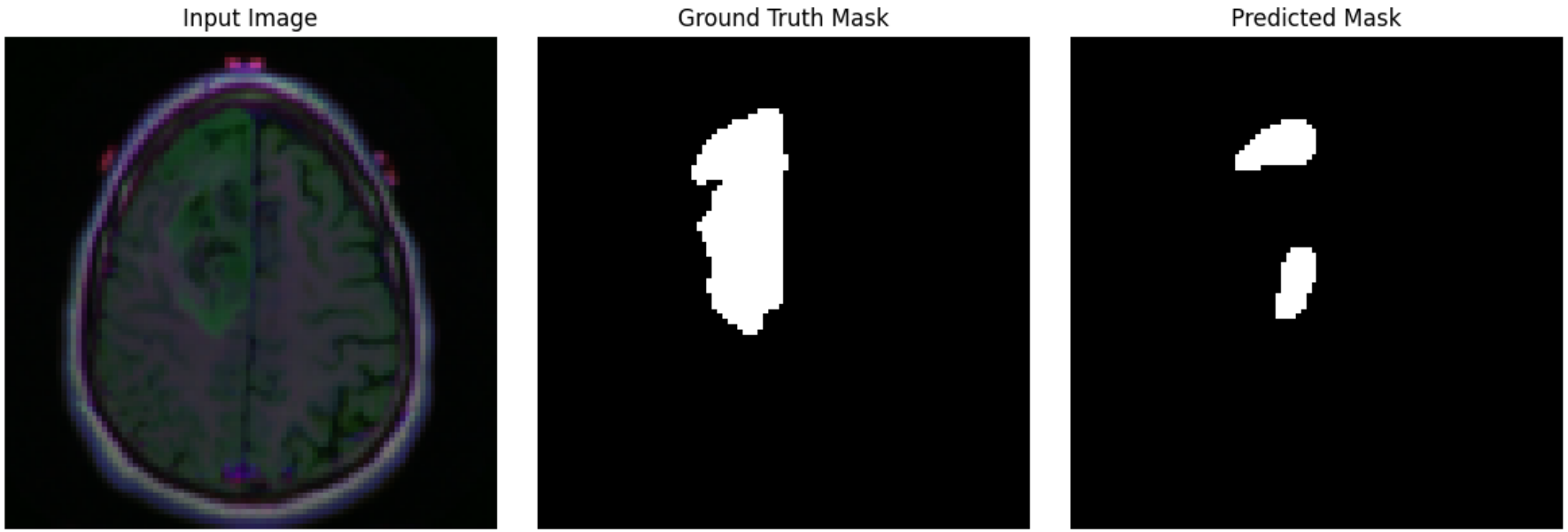
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



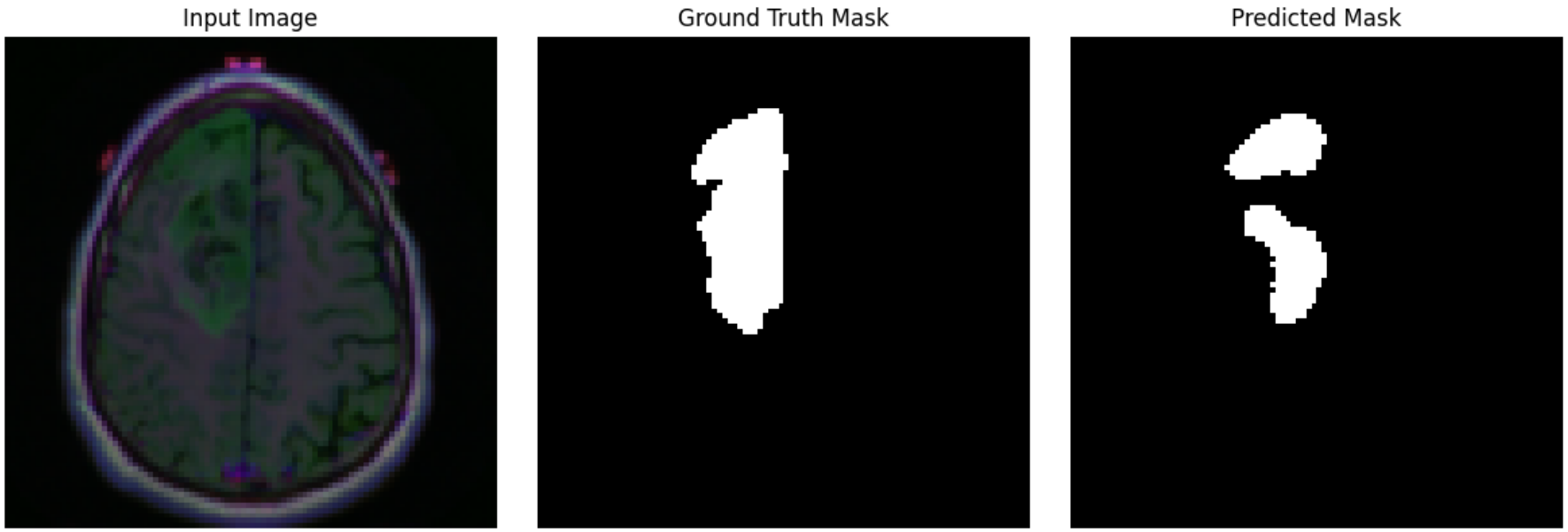
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



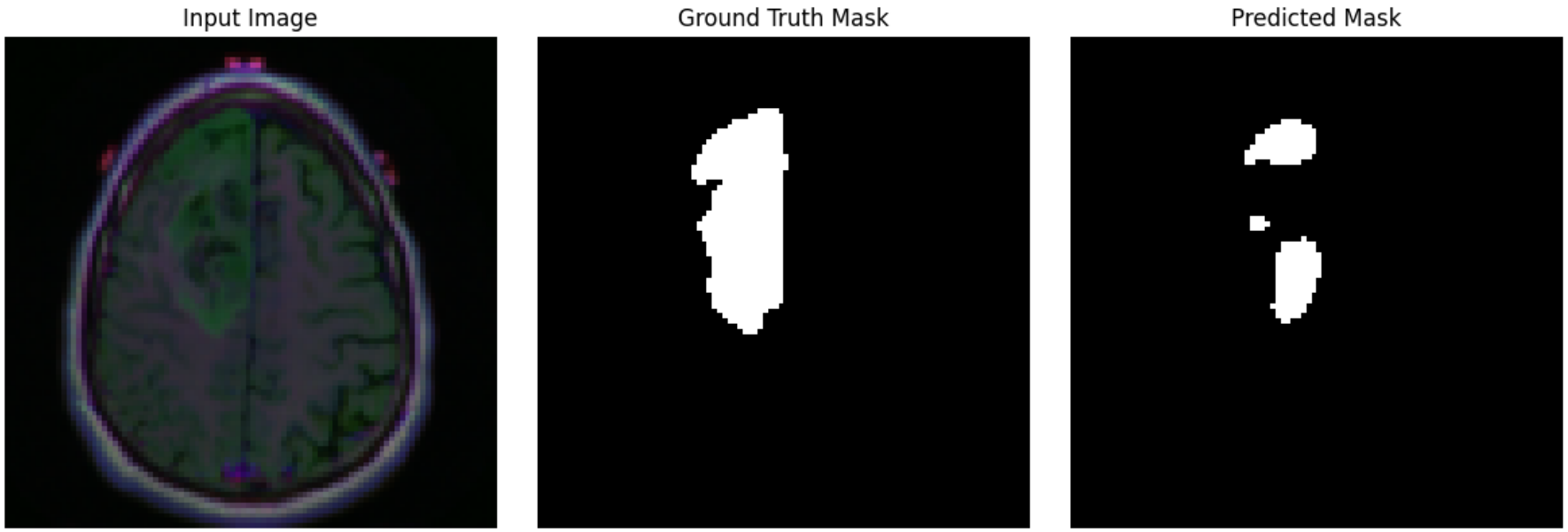
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



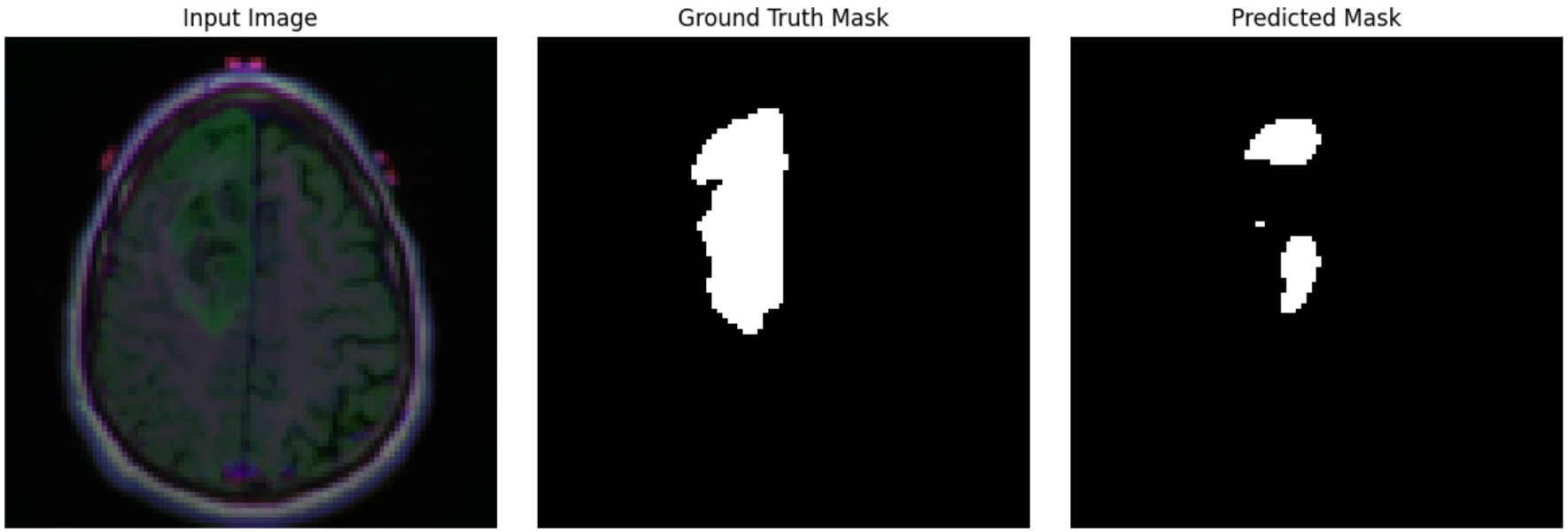
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



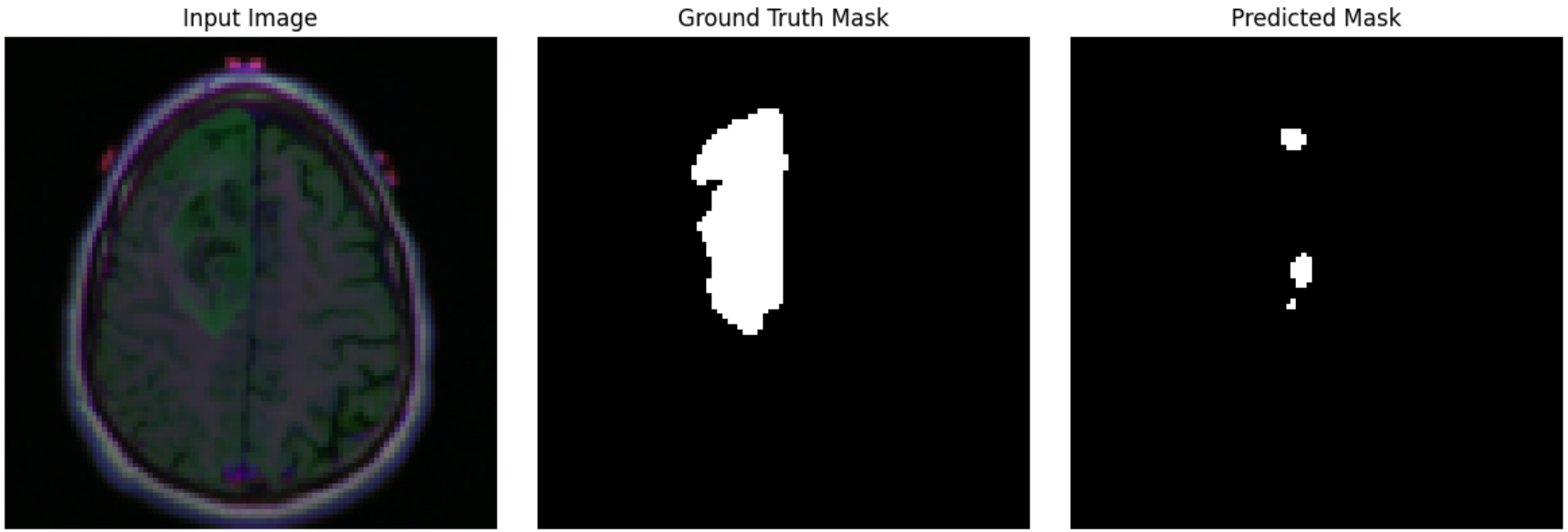
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



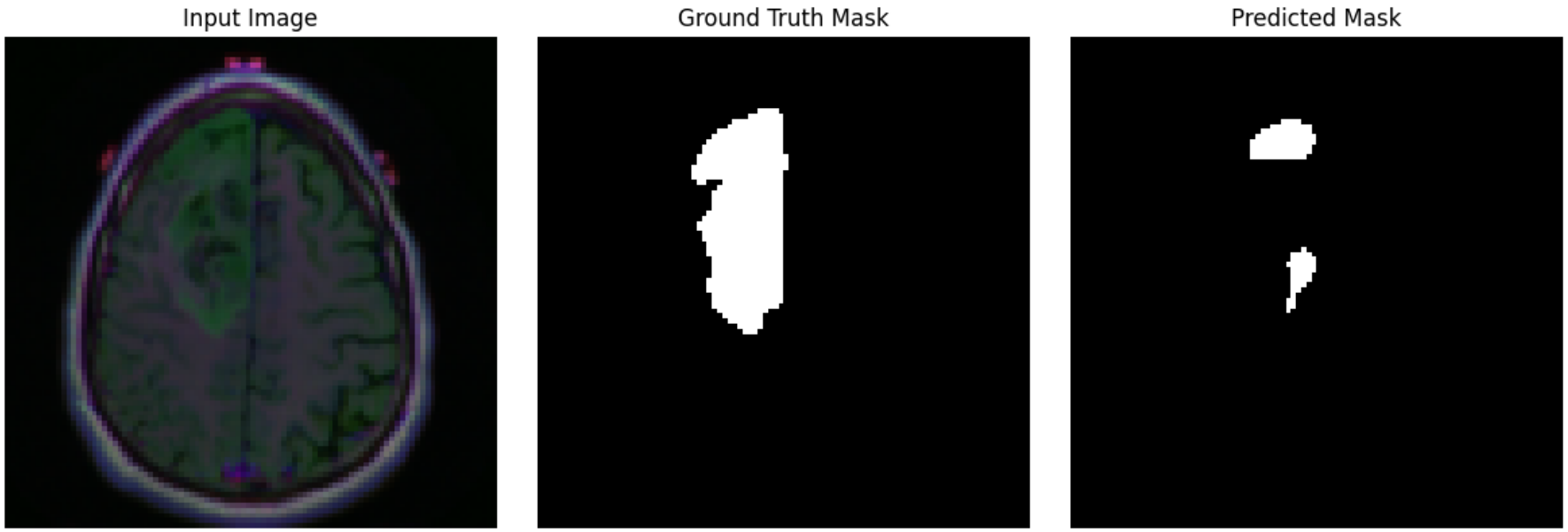
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



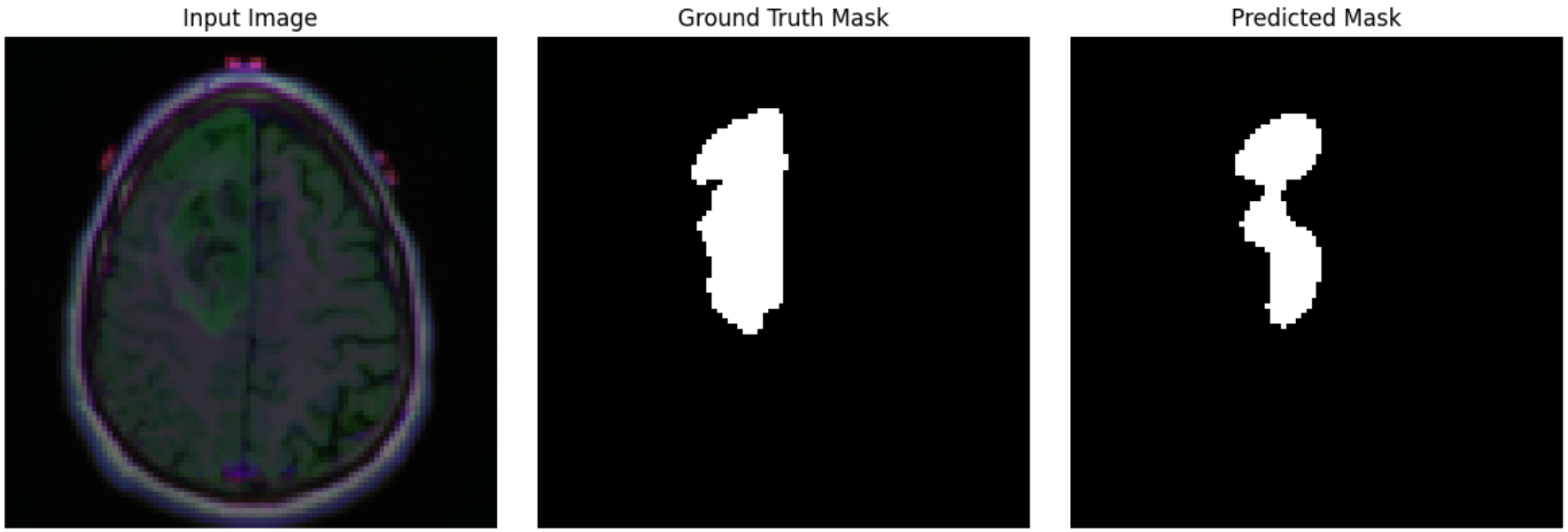
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



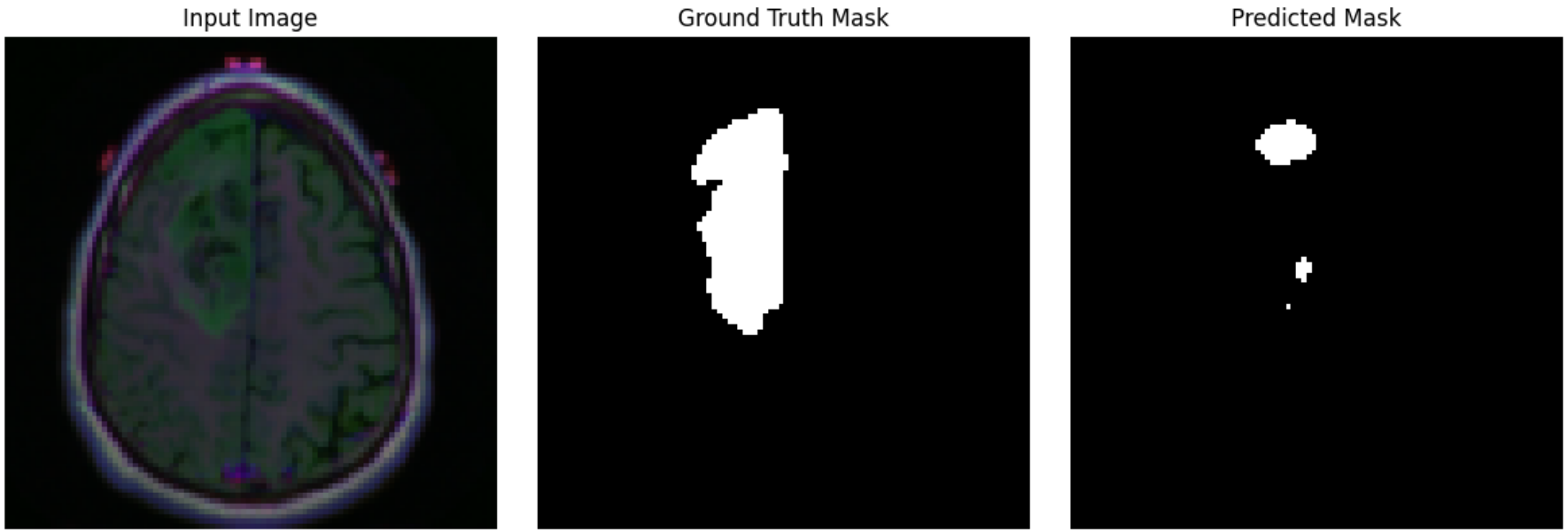
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



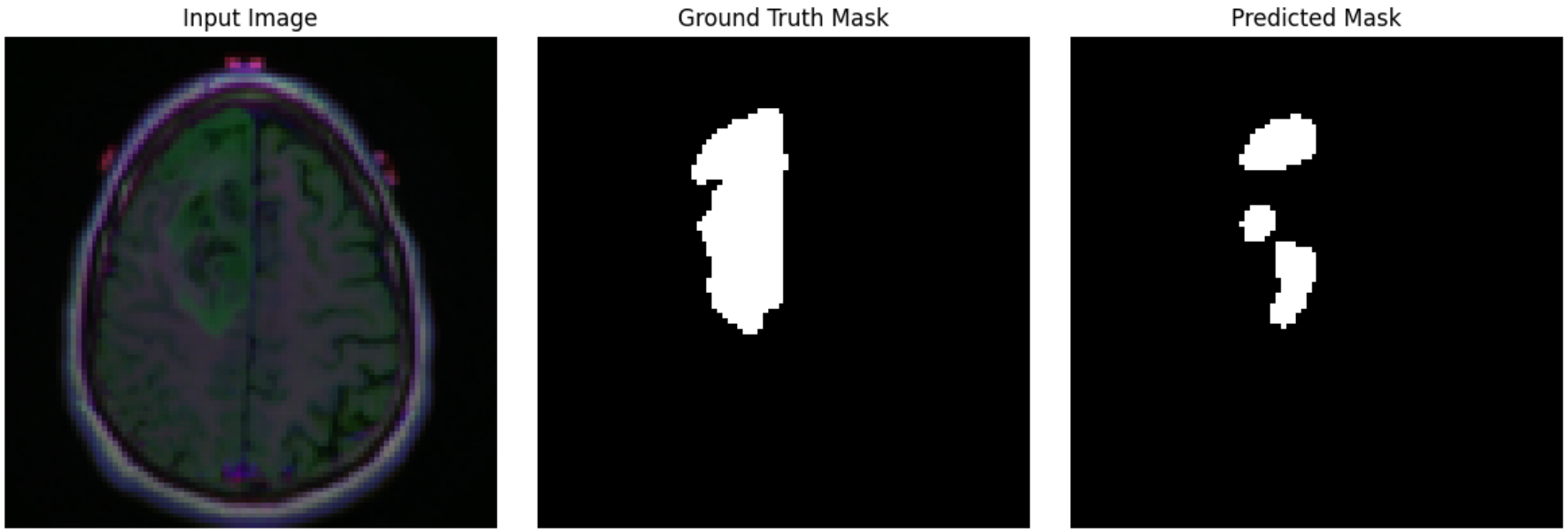
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



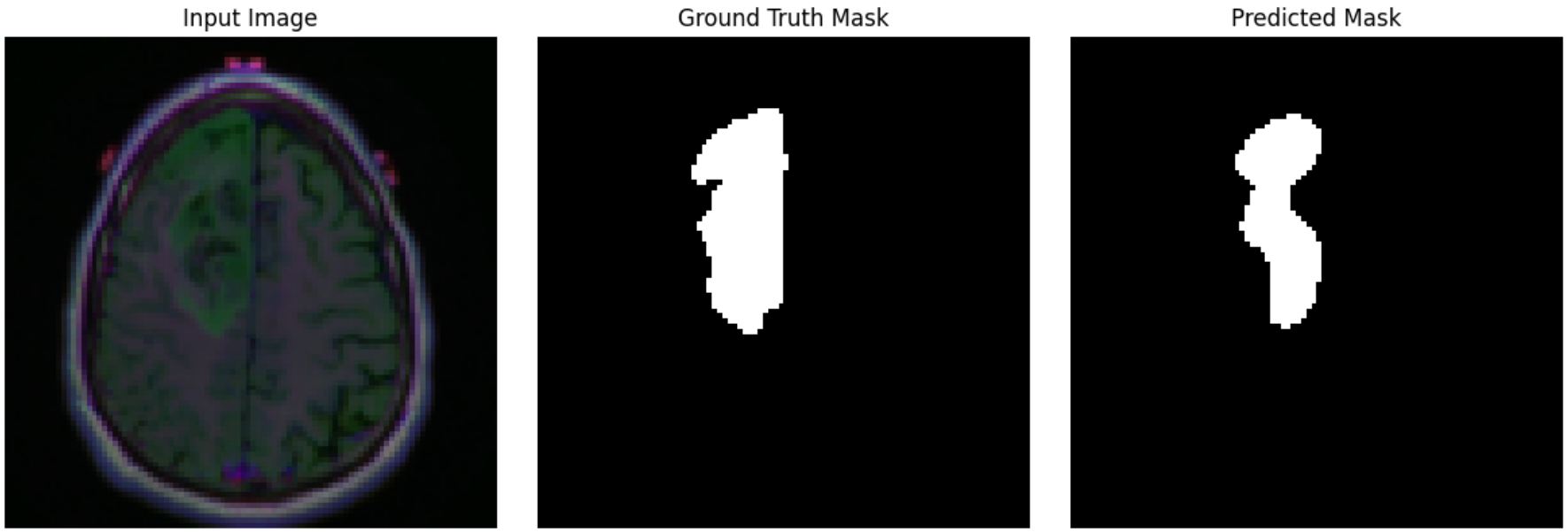
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



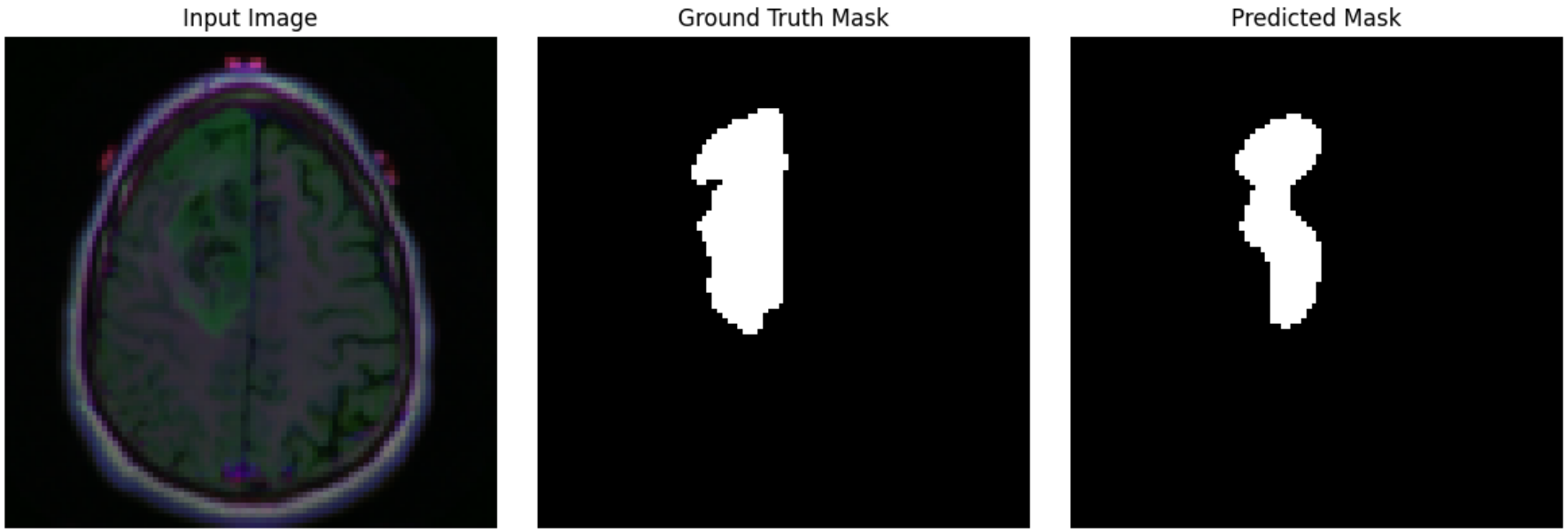
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



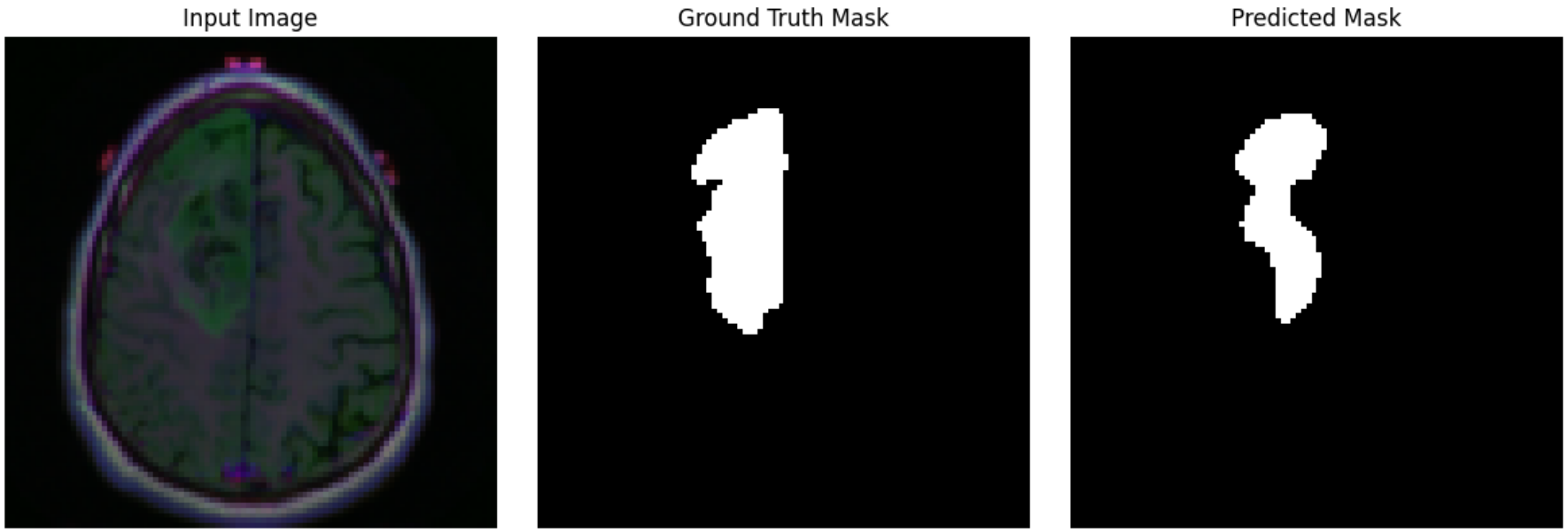
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution



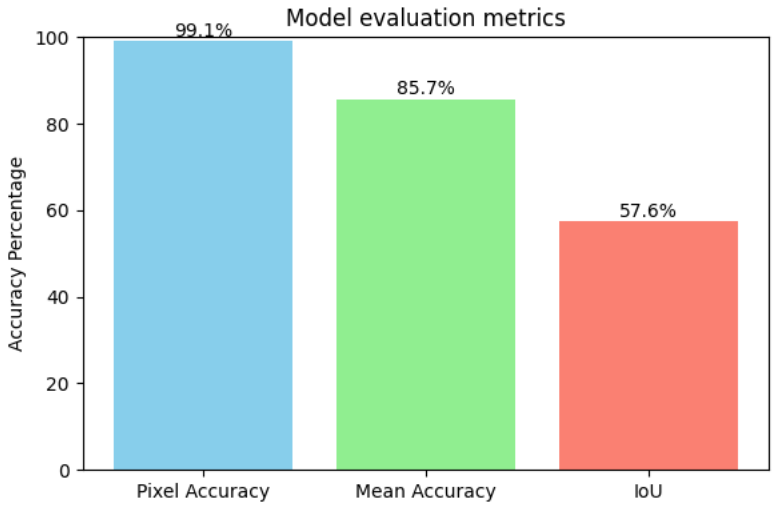
U-Net with 0.7 CE Combo Loss

visualizing the training region evolution

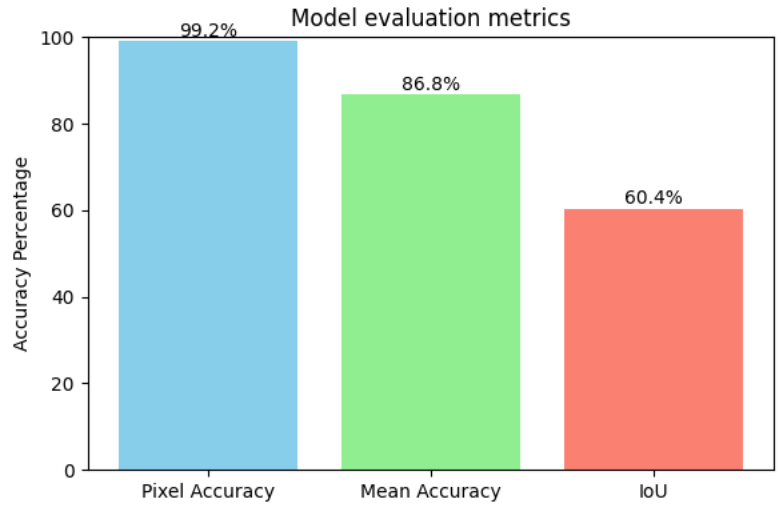


U-Net with 0.7 CE Combo Loss

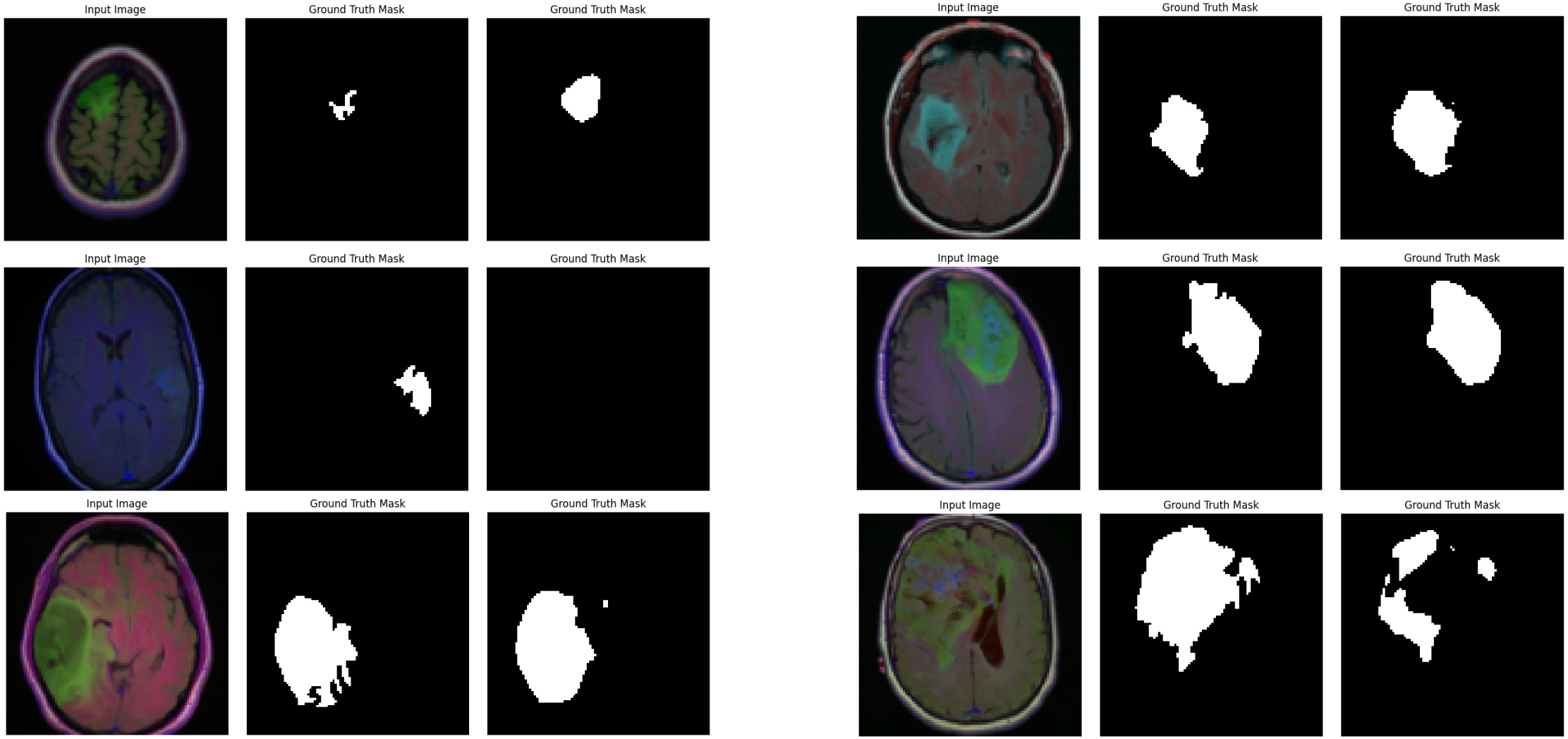
Accuracy results on the test set



And with epochs increase…



U-Net with 0.7 CE-R Combo Loss : What about other samples?



U-Net with 0.5 CE Combo Loss

Training Informations

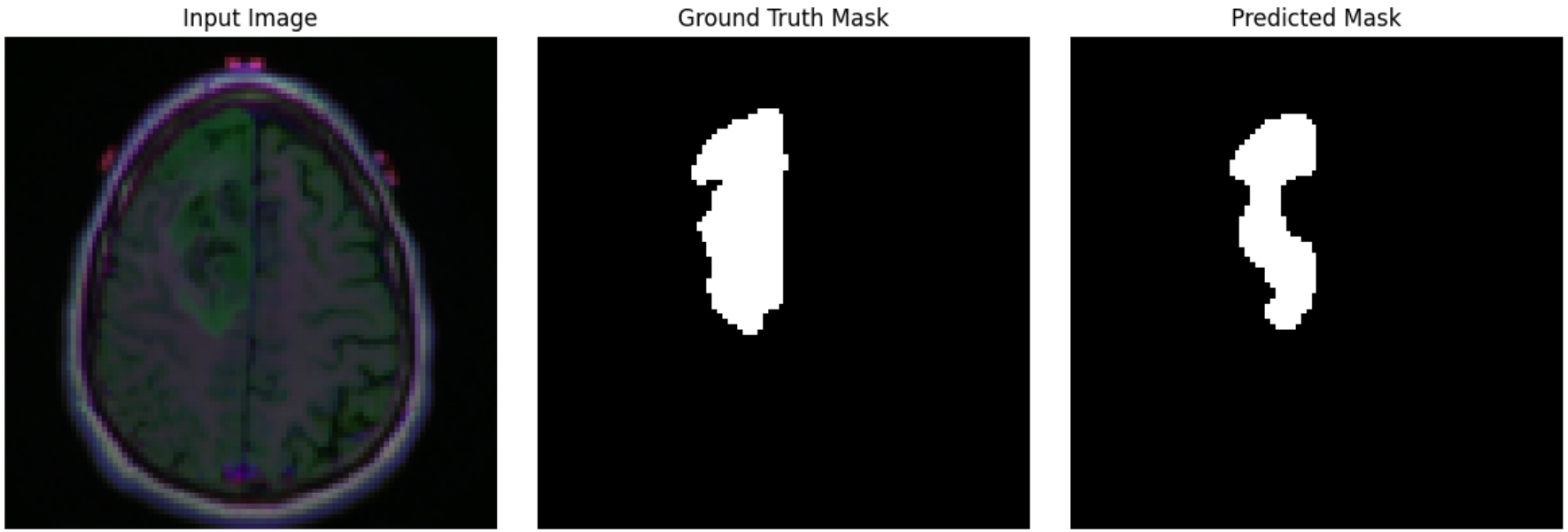
Adam optimizer with 10^{-4} learning rate

~ 40 training epochs

Saved an image and mask prediction, so to make available the training region evolution visualization…

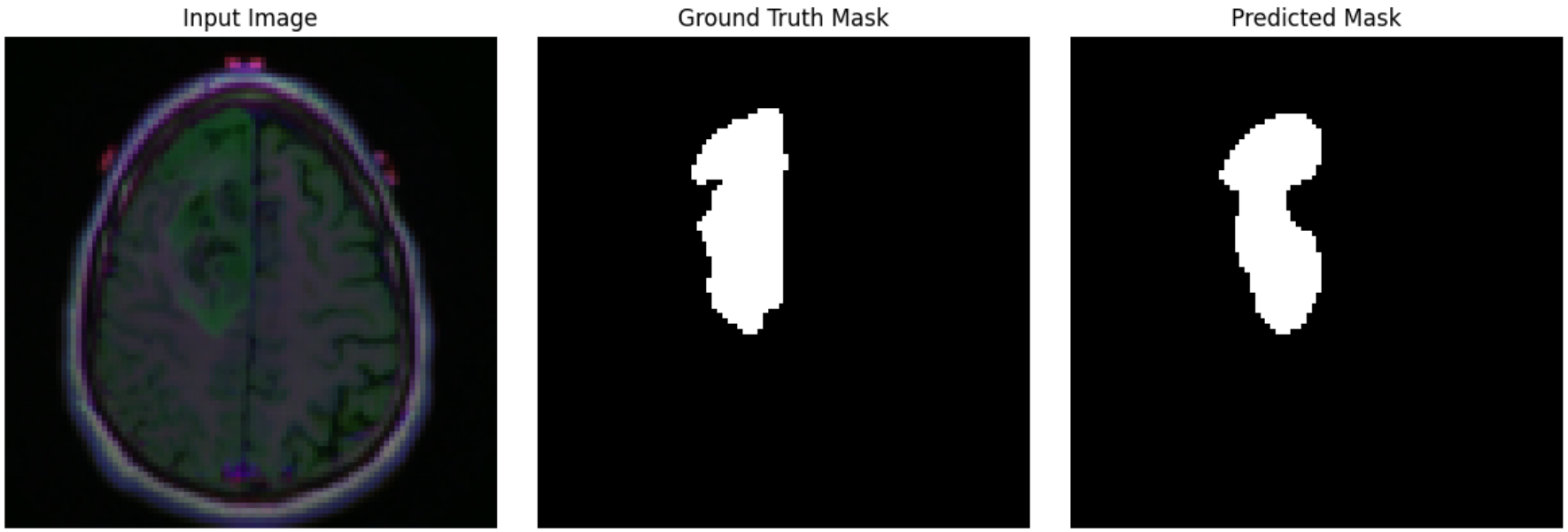
U-Net with 0.5 CE-R Combo Loss

visualizing the training region evolution



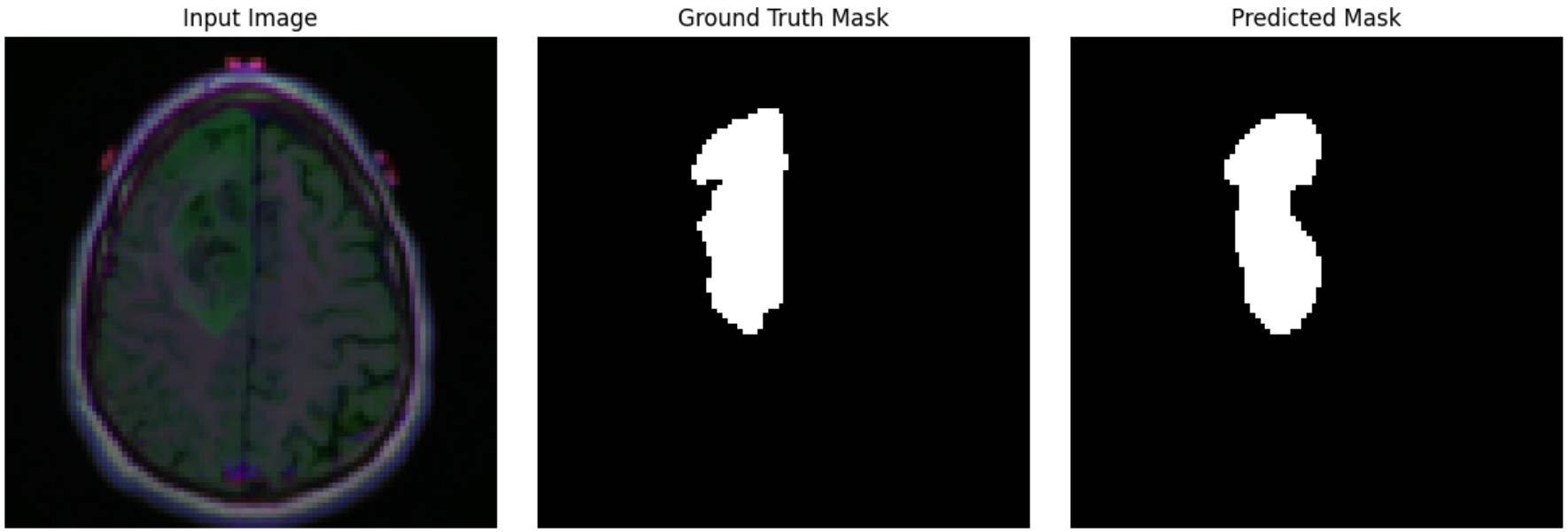
U-Net with 0.5 CE-R Combo Loss

visualizing the training region evolution



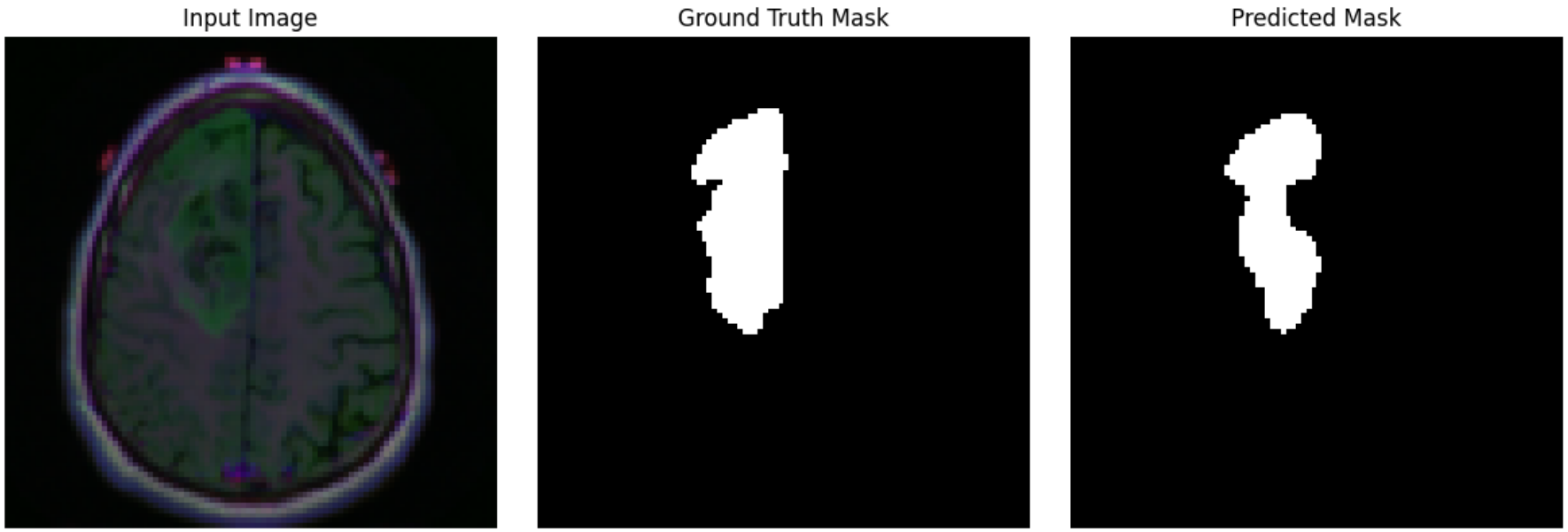
U-Net with 0.5 CE-R Combo Loss

visualizing the training region evolution



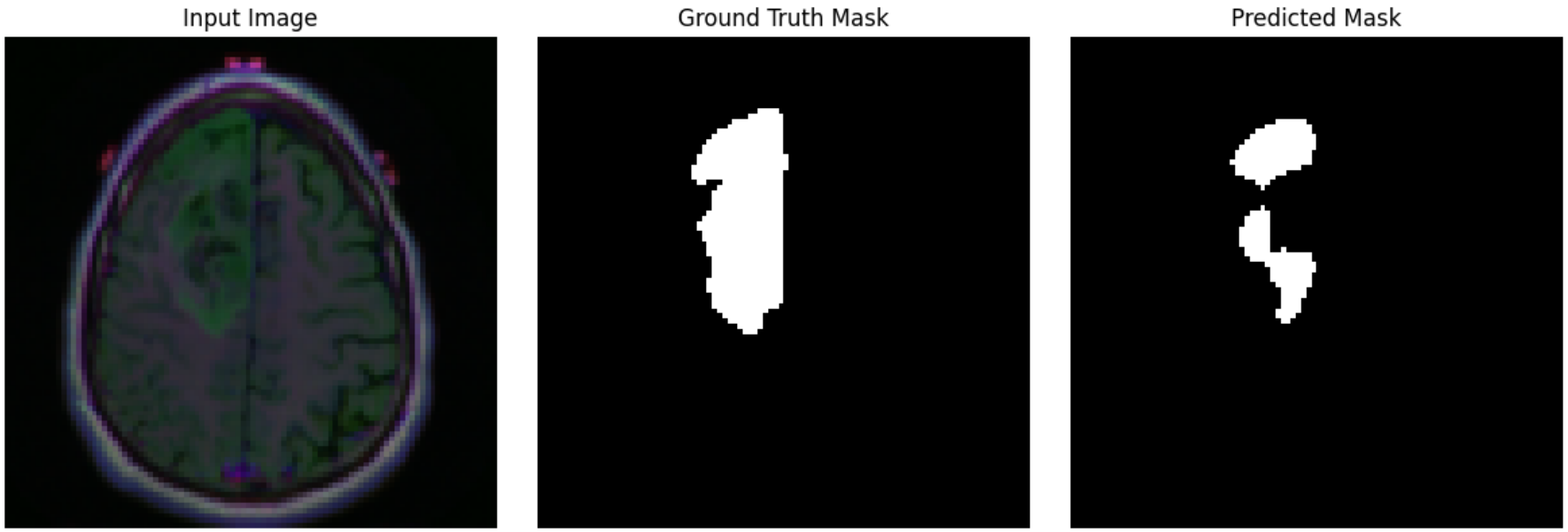
U-Net with 0.5 CE-R Combo Loss

visualizing the training region evolution



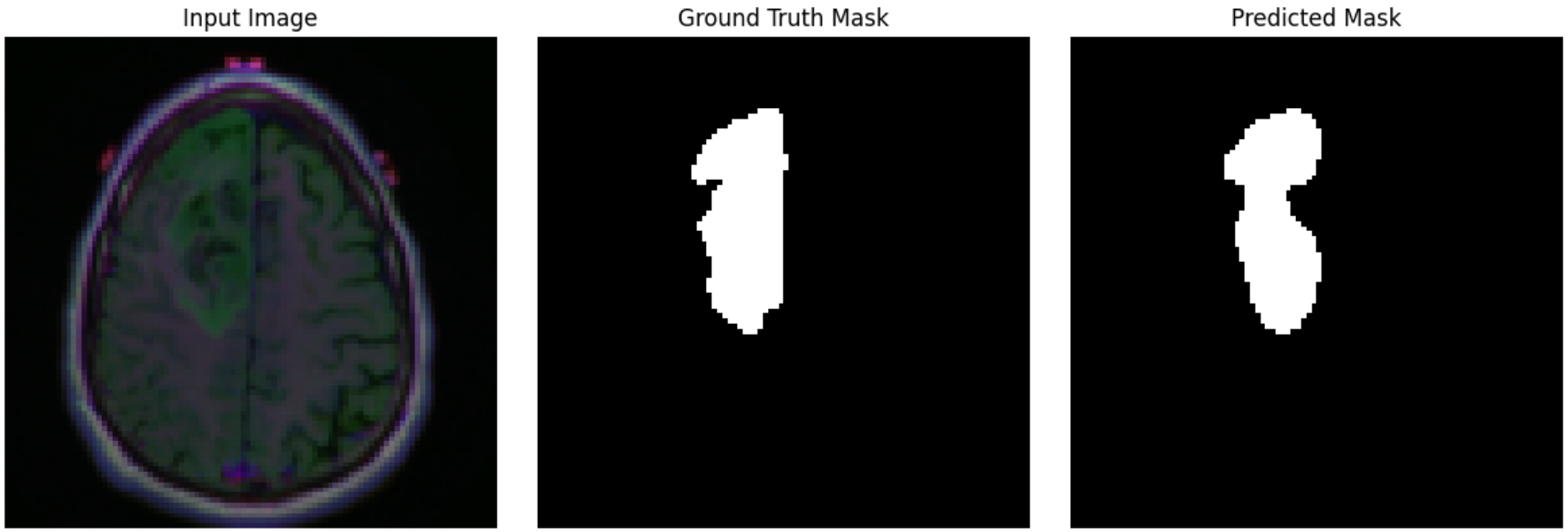
U-Net with 0.5 CE-R Combo Loss

visualizing the training region evolution



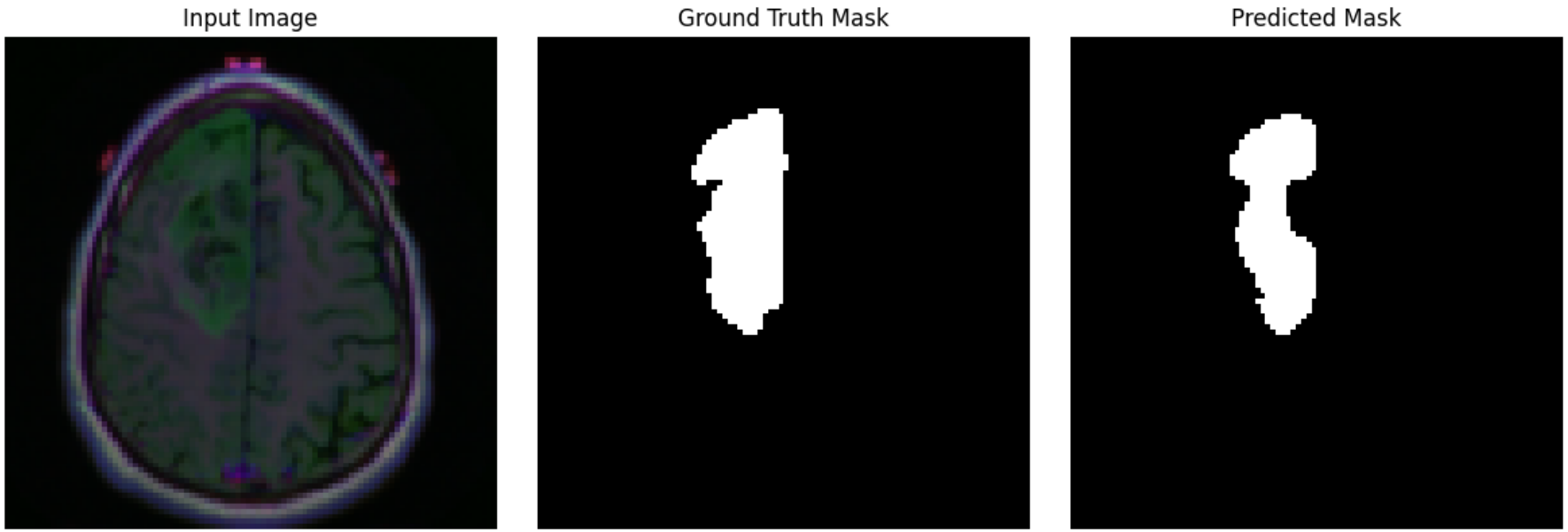
U-Net with 0.5 CE-R Combo Loss

visualizing the training region evolution



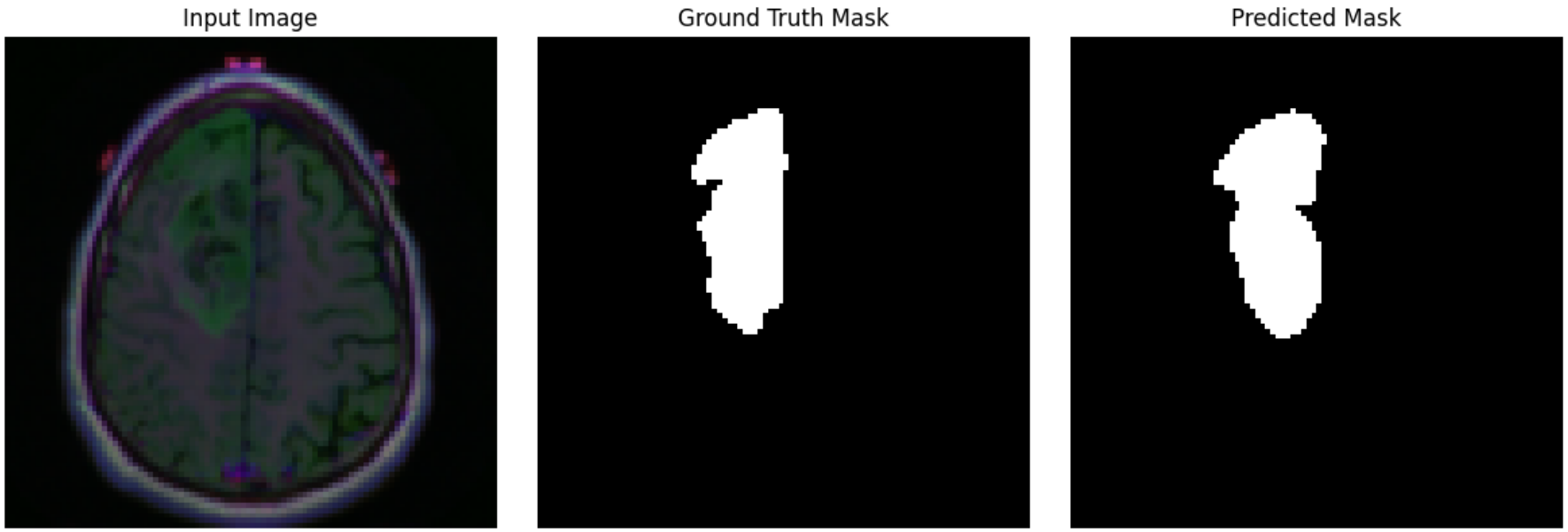
U-Net with 0.5 CE-R Combo Loss

visualizing the training region evolution



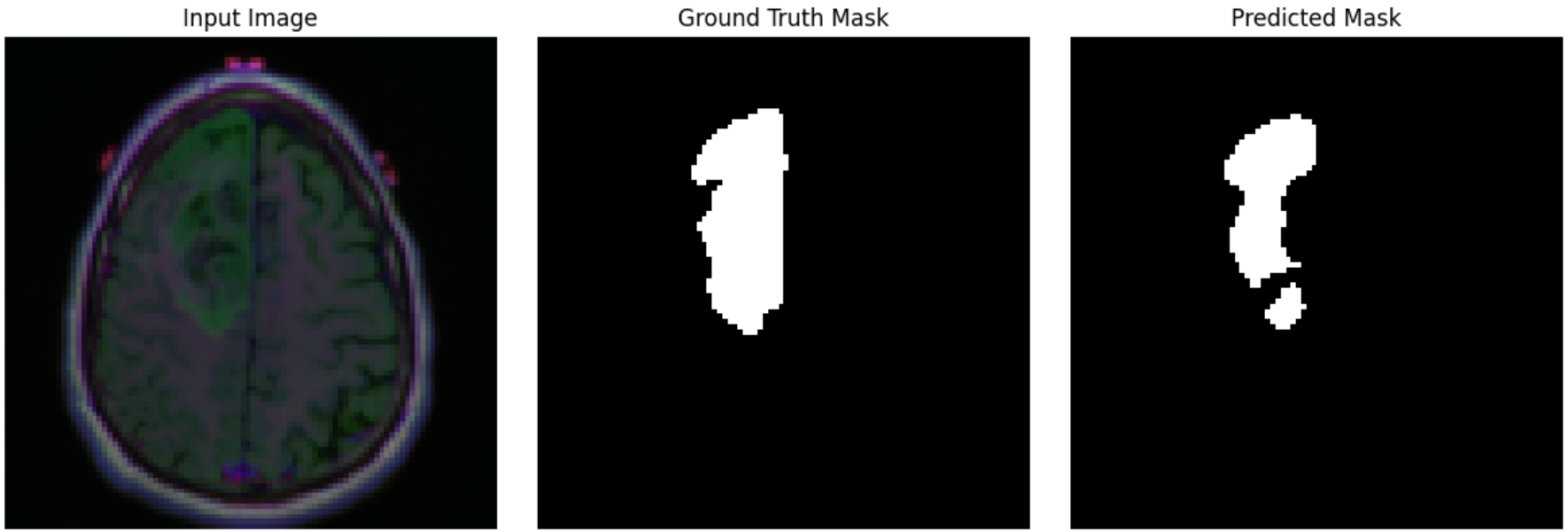
U-Net with 0.5 CE-R Combo Loss

visualizing the training region evolution



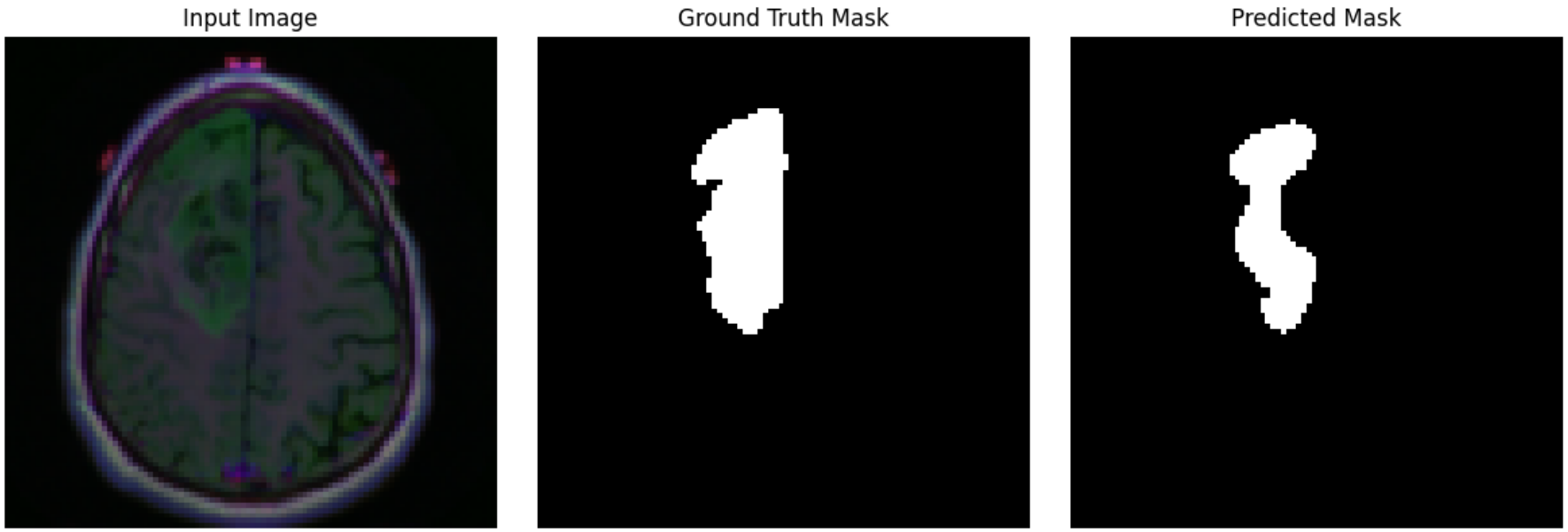
U-Net with 0.5 CE-R Combo Loss

visualizing the training region evolution



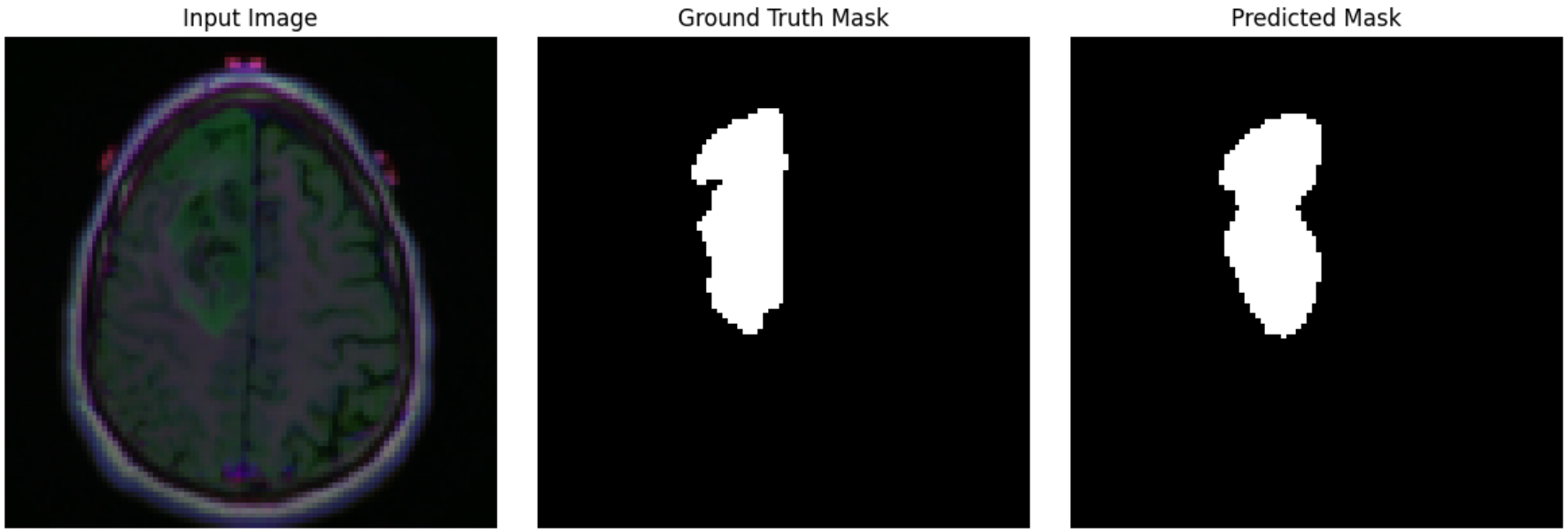
U-Net with 0.5 CE-R Combo Loss

visualizing the training region evolution

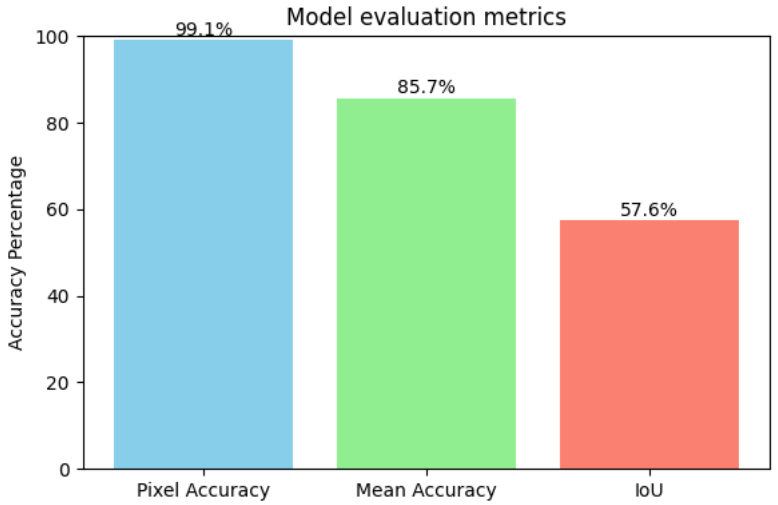


U-Net with 0.5 CE-R Combo Loss

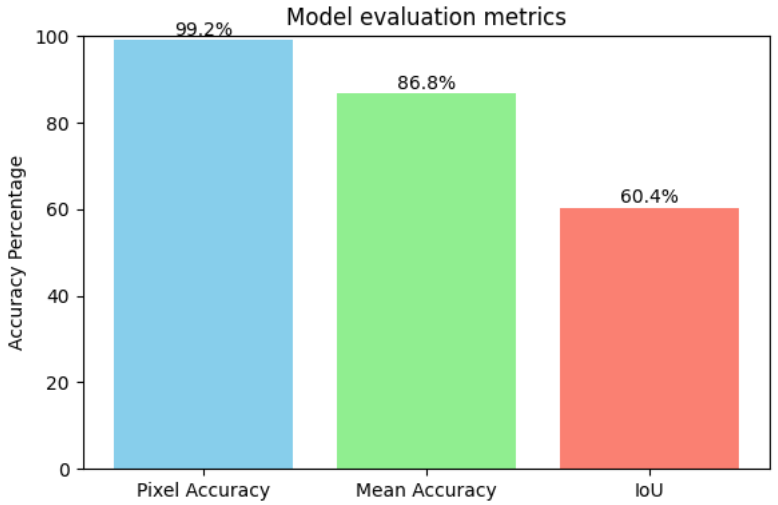
visualizing the training region evolution



U-Net with Combo Loss
Model Comparison

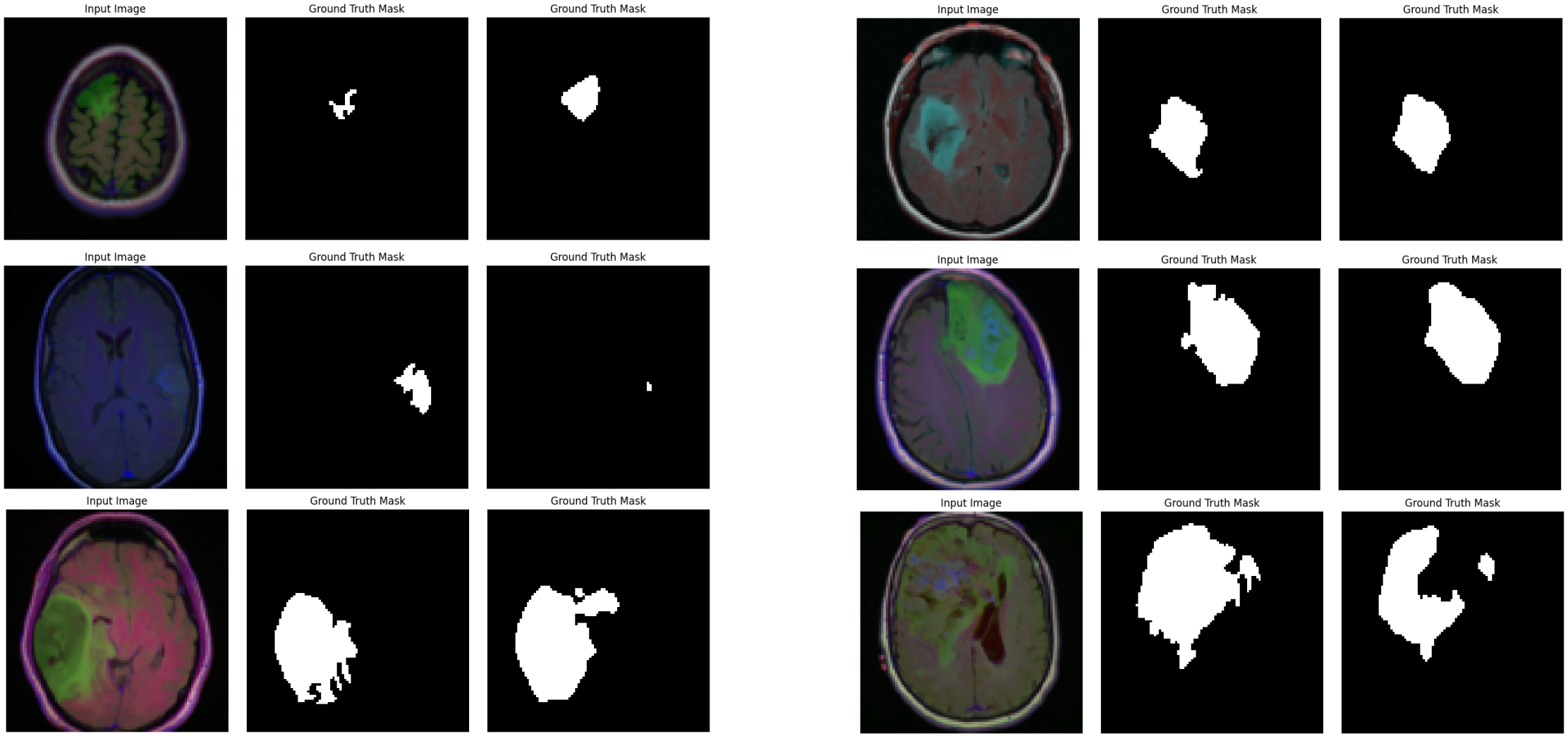


Combo ratio 0.7
Approx 20 epochs learning



Combo ratio 0.5
Approx 40 epochs learning

U-Net with 0.7 CE-R Combo Loss : What about other samples?



Vision Transformers

U-net is not the only approach we can try for this specific task, since the recent advances with vision transformed has shown great results [9]

The idea is to offer a similar approach to image handling than the one proposed to the Natural Language processing word one.

Transformers as architectures are originally designed for sequential data in Natural Language Processing and those are based on the self-attention mechanism; the visual transformer leverage the same architecture for the image handling.

ViTs divide the image into patches and process each of them as token element of the sequence,

The key differences are patches usage, attention and positional embedding for spacial information

Vision Transformers: SETR

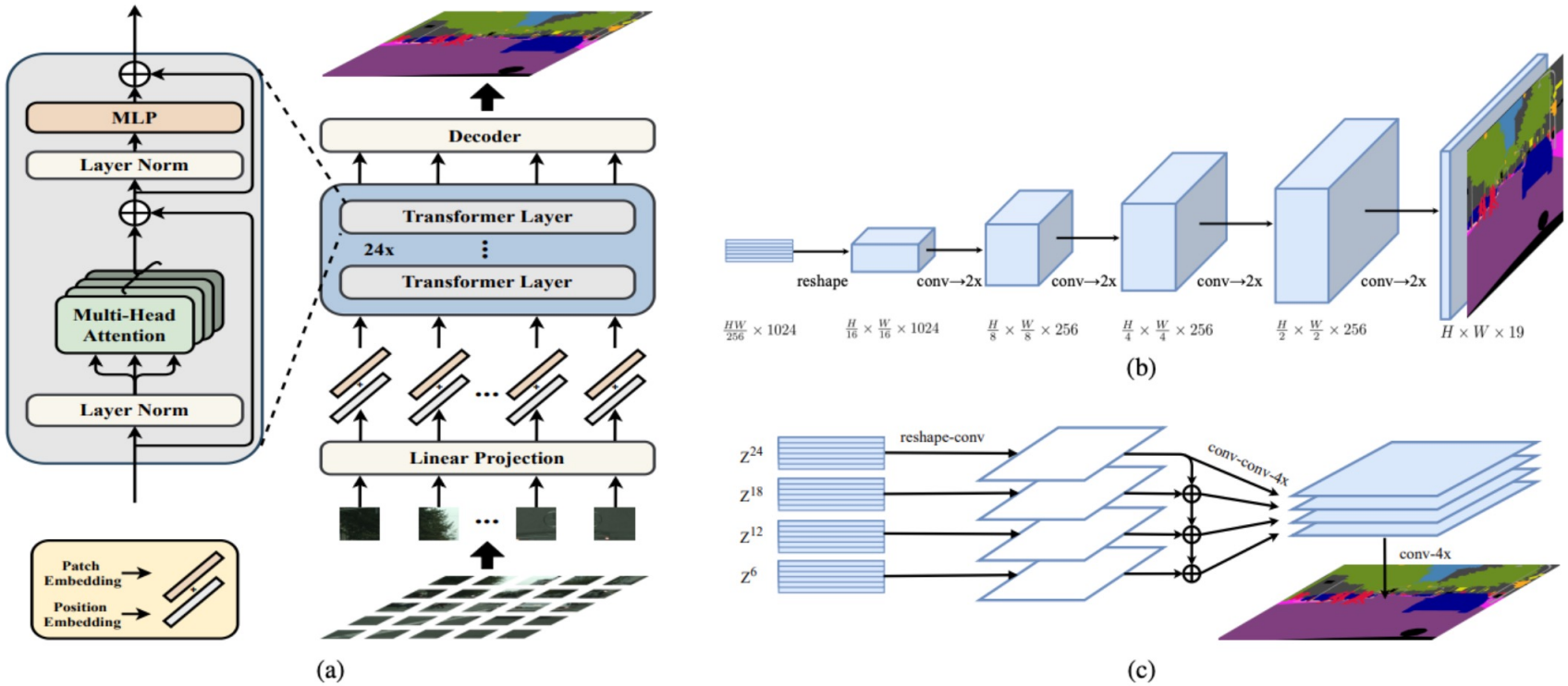


Fig 4. Schematic illustration of the proposed SEgmentation TRansformer (SETR). []

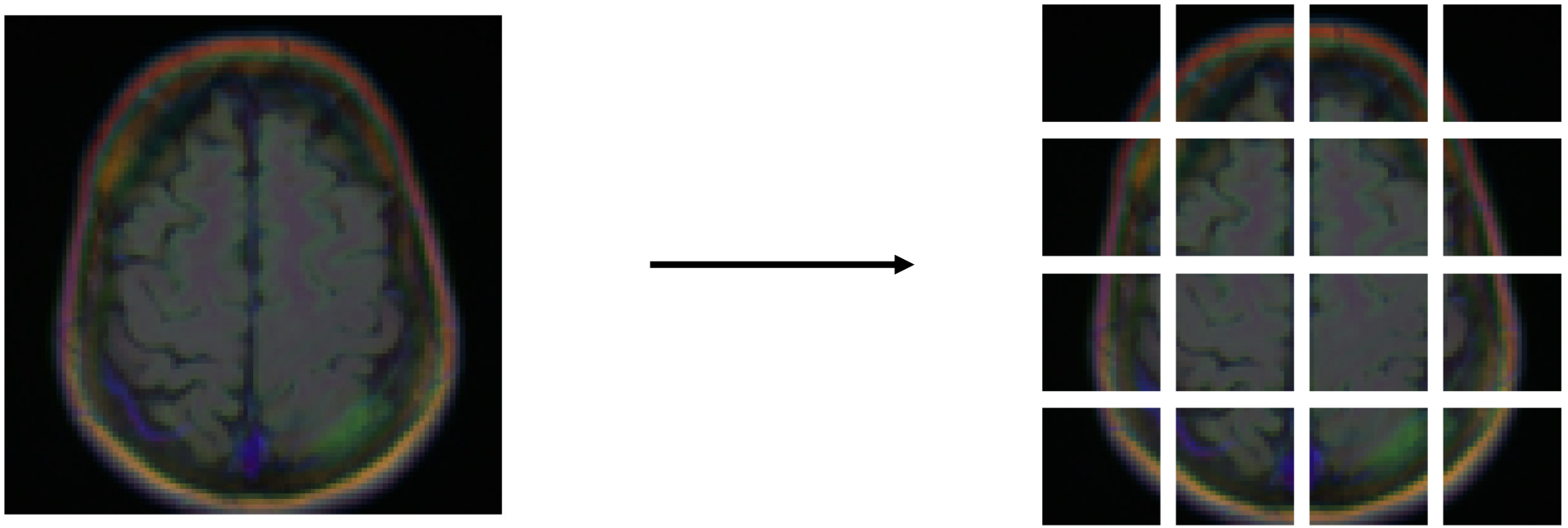
```
class SETR(nn.Module):

def __init__(self, img_size=128, num_cuts = 4,in_ch=3, num_classes=1, dim=1024, depth=8, heads=8, mlp_dim=2048):
        super().__init__()

        # Internal parameters definition:
        self.patch_size  = int(W / num_cuts)
        self.num_cuts    = num_cuts
        self.num_patches = self.num_cuts ** 2
        self.dim = dim
        self.in_ch = in_ch

        # Patch + Positional Embedding. We need to redefine this due to generalization...
        self.patch_embed = nn.Linear(self.patch_size * self.patch_size * in_ch, dim)
        self.pos_embed   = nn.Parameter(torch.randn(1, self.num_patches, dim))

        # Transformer Encoder. We refer to the previously defined class
        self.transformer = nn.Sequential(
        *[TransformerEncoderLayer(dim, heads, mlp_dim) for _ in range(depth)]
        )

        # Deconvolution decoder
        # Notice that is the same of before byt dimension here is generic
        self.decoder = nn.Sequential(
        nn.ConvTranspose2d(dim, 256, kernel_size=2, stride=2), nn.ReLU(),
        nn.ConvTranspose2d(256, 256, kernel_size=2, stride=2), nn.ReLU(),
        nn.ConvTranspose2d(256, 256, kernel_size=2, stride=2), nn.ReLU(),
        nn.ConvTranspose2d(256, 256, kernel_size=2, stride=2), nn.ReLU(),
        nn.Conv2d(256, num_classes, kernel_size=1))

def forward(self, x):

        # Split image into patches
        B = x.size(0)
        x = x.unfold(2, self.patch_size, self.patch_size)\
             .unfold(3, self.patch_size, self.patch_size)\
             .permute(0, 2, 3, 1, 4, 5)\
             .reshape(B, self.num_patches, self.in_ch * self.patch_size * self.patch_size)

        # Embedding + Positional encoding
        x = self.patch_embed(x) + self.pos_embed

        # Transformer encoder
        x = x.permute(1, 0, 2) # (N, B, dim) for nn.MultiheadAttention
        x = self.transformer(x)
        x = x.permute(1, 2, 0) # (B, dim, N)

        # Reshape to 2D feature map
        x = x.reshape(B, self.dim, self.num_cuts, self.num_cuts) # (B, dim, H/16, W/16)

        # Decoder
        x = self.decoder(x) # (B, num_classes, H, W)
        # Now we need values at zero or one

        return torch.sigmoid(x)
```

Forward of the model

```
def forward(self, x):

        # Split image into patches
        B = x.size(0)
        x = x.unfold(2, self.patch_size, self.patch_size)\
             .unfold(3, self.patch_size, self.patch_size)\
             .permute(0, 2, 3, 1, 4, 5)\
             .reshape(B, self.num_patches, self.in_ch * self.patch_size * self.patch_size)

        # Embedding + Positional encoding
        x = self.patch_embed(x) + self.pos_embed

        # Transformer encoder
        x = x.permute(1, 0, 2) # (N, B, dim) for nn.MultiheadAttention
        x = self.transformer(x)
        x = x.permute(1, 2, 0) # (B, dim, N)

        # Reshape to 2D feature map
        x = x.reshape(B, self.dim, self.num_cuts, self.num_cuts) # (B, dim, H/16, W/16)

        # Decoder
        x = self.decoder(x) # (B, num_classes, H, W)
        # Now we need values at zero or one

        return torch.sigmoid(x)
```

```python
def forward(self, x):

        # Split image into patches
        B = x.size(0)
        x = x.unfold(2, self.patch_size, self.patch_size)\
             .unfold(3, self.patch_size, self.patch_size)\
             .permute(0, 2, 3, 1, 4, 5)\
             .reshape(B, self.num_patches, self.in_ch * self.patch_size * self.patch_size)

        # Embedding + Positional encoding
        x = self.patch_embed(x) + self.pos_embed

        # Transformer encoder
        x = x.permute(1, 0, 2) # (N, B, dim) for nn.MultiheadAttention
        x = self.transformer(x)
        x = x.permute(1, 2, 0) # (B, dim, N)

        # Reshape to 2D feature map
        x = x.reshape(B, self.dim, self.num_cuts, self.num_cuts) # (B, dim, H/16, W/16)

        # Decoder
        x = self.decoder(x) # (B, num_classes, H, W)
        # Now we need values at zero or one

        return torch.sigmoid(x)
```

```python
# Split image into patches
B = x.size(0)
x = x.unfold(2, self.patch_size, self.patch_size)\
     .unfold(3, self.patch_size, self.patch_size)\
     .permute(0, 2, 3, 1, 4, 5)\
     .reshape(B, self.num_patches, self.in_ch * self.patch_size * self.patch_size)
```



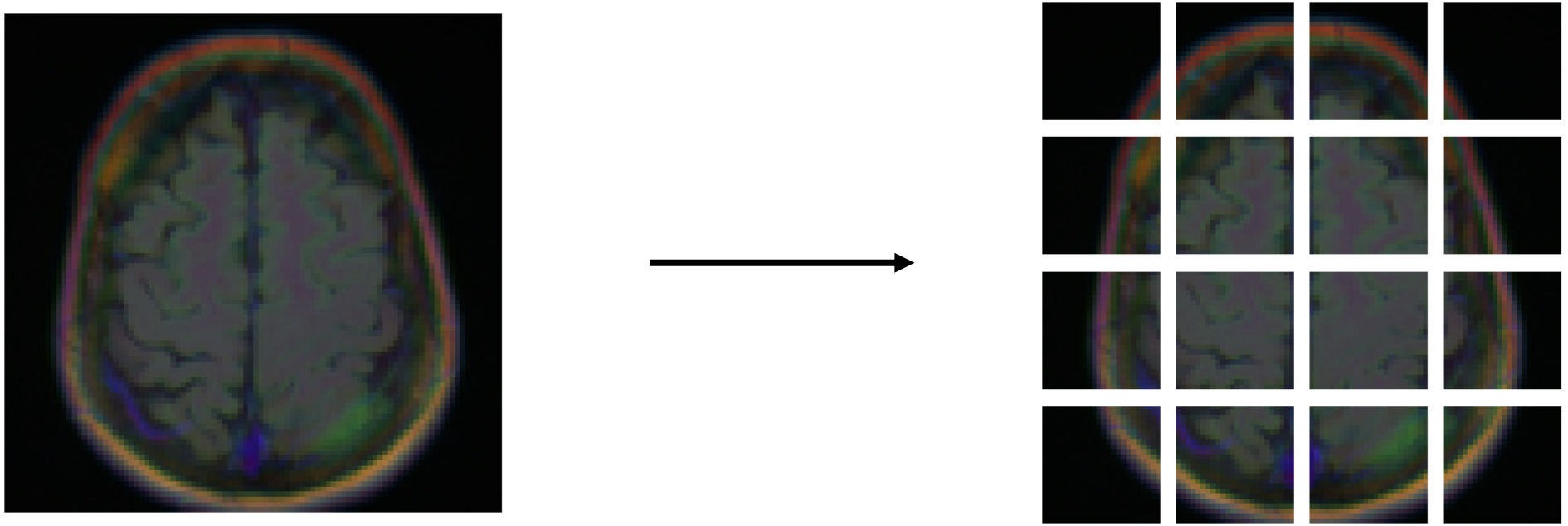
2 is the height dimension

Patch cutting size

As if the slices are in a collector tensor itself

Now we have something like $[B, 3, p.num\ H, p.num\ W, p.size\ H, p.size\ W]$

```
# Split image into patches
B = x.size(0)
x = x.unfold(2, self.patch_size, self.patch_size)\
     .unfold(3, self.patch_size, self.patch_size)\
     .permute(0, 2, 3, 1, 4, 5)\
     .reshape(B, self.num_patches, self.in_ch * self.patch_size * self.patch_size)
```

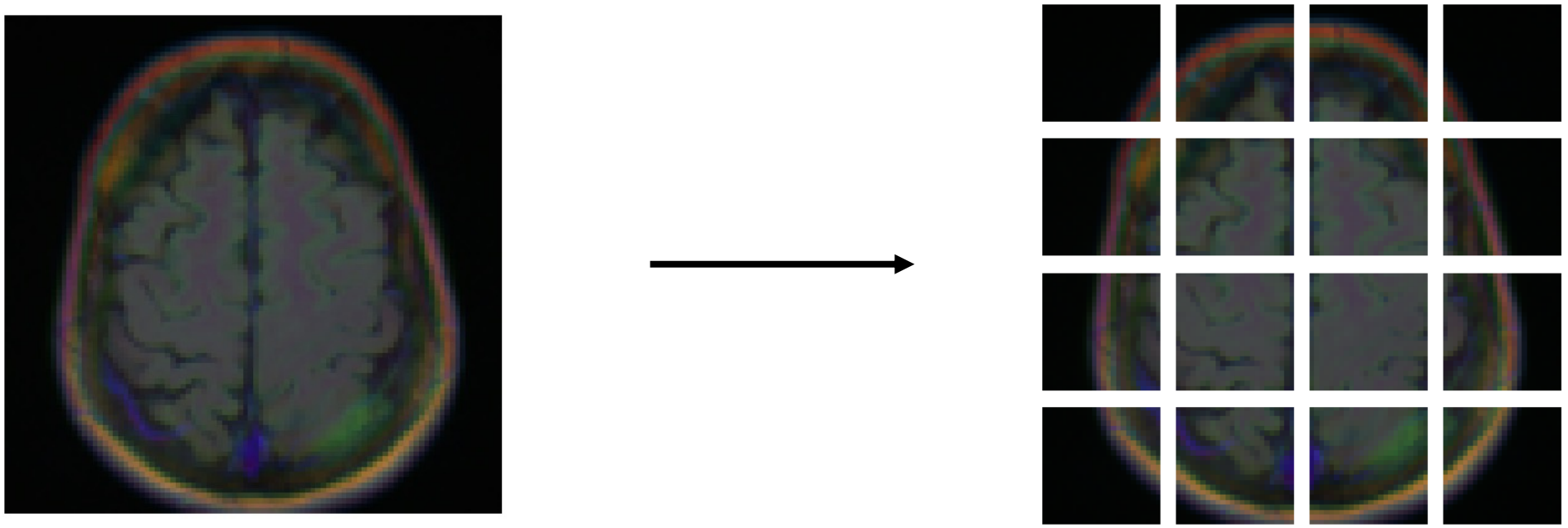


```python
# Split image into patches
B = x.size(0)
x = x.unfold(2, self.patch_size, self.patch_size)\
     .unfold(3, self.patch_size, self.patch_size)\
     .permute(0, 2, 3, 1, 4, 5)\
     .reshape(B, self.num_patches, self.in_ch * self.patch_size * self.patch_size)
```

Now we have something like $[B, 3, p.num\ H, p.num\ W, p.size\ H, p.size\ W]$

So we need to reorganize in $[B, p.num\ H, p.num\ W, 3, p.size\ H, p.size\ W]$

Each image patch is now a 1D vector



The number of patches should be elevated, even if this is a time-consuming obligation

```python
def forward(self, x):

        # Split image into patches
        B = x.size(0)
        x = x.unfold(2, self.patch_size, self.patch_size)\
             .unfold(3, self.patch_size, self.patch_size)\
             .permute(0, 2, 3, 1, 4, 5)\
             .reshape(B, self.num_patches, self.in_ch * self.patch_size * self.patch_size)

        # Embedding + Positional encoding
        x = self.patch_embed(x) + self.pos_embed

        # Transformer encoder
        x = x.permute(1, 0, 2) # (N, B, dim) for nn.MultiheadAttention
        x = self.transformer(x)
        x = x.permute(1, 2, 0) # (B, dim, N)

        # Reshape to 2D feature map
        x = x.reshape(B, self.dim, self.num_cuts, self.num_cuts) # (B, dim, H/16, W/16)

        # Decoder
        x = self.decoder(x) # (B, num_classes, H, W)
        # Now we need values at zero or one

        return torch.sigmoid(x)
```

```
# Embedding + Positional encoding
x = self.patch_embed(x) + self.pos_embed
```

with

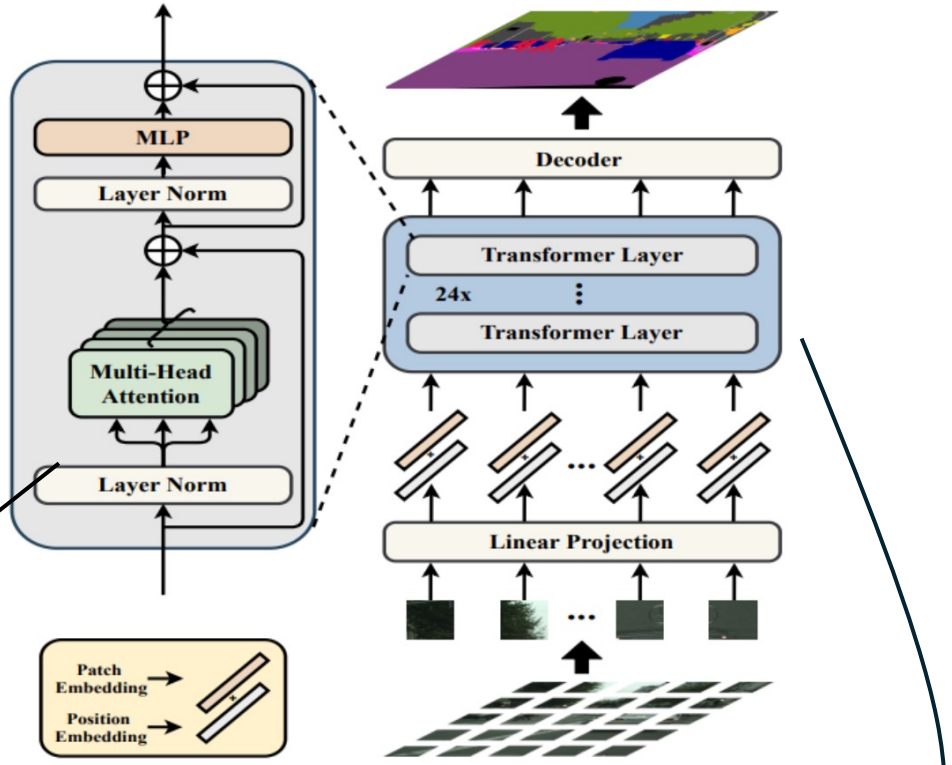
```
self.patch_embed = nn.Linear(self.patch_size * self.patch_size * in_ch, dim)
self.pos_embed = nn.Parameter(torch.randn(1, self.num_patches, dim))
```

After each patch transformation in a 1D vector
we transform it into a proper token of chosen dimension `dim`

```python
def forward(self, x):

        # Split image into patches
        B = x.size(0)
        x = x.unfold(2, self.patch_size, self.patch_size)\
             .unfold(3, self.patch_size, self.patch_size)\
             .permute(0, 2, 3, 1, 4, 5)\
             .reshape(B, self.num_patches, self.in_ch * self.patch_size * self.patch_size)

        # Embedding + Positional encoding
        x = self.patch_embed(x) + self.pos_embed

        # Transformer encoder
        x = x.permute(1, 0, 2) # (N, B, dim) for nn.MultiheadAttention
        x = self.transformer(x)
        x = x.permute(1, 2, 0) # (B, dim, N)

        # Reshape to 2D feature map
        x = x.reshape(B, self.dim, self.num_cuts, self.num_cuts) # (B, dim, H/16, W/16)

        # Decoder
        x = self.decoder(x) # (B, num_classes, H, W)
        # Now we need values at zero or one

        return torch.sigmoid(x)
```

```python
# Transformer encoder
x = x.permute(1, 0, 2) # (N, B, dim) for nn.MultiheadAttention
x = self.transformer(x)
x = x.permute(1, 2, 0) # (B, dim, N)
```

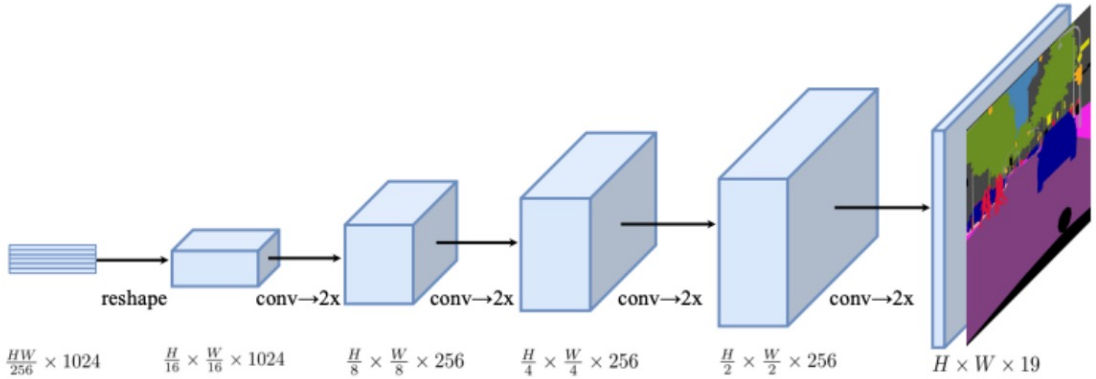
```python
class TransformerEncoderLayer(nn.Module):
def __init__(self, dim, heads, mlp_dim):
super().__init__()
self.norm1 = nn.LayerNorm(dim)
self.attn = nn.MultiheadAttention(embed_dim=dim, num_heads=heads)

self.norm2 = nn.LayerNorm(dim)
self.mlp = nn.Sequential(
            nn.Linear(dim, mlp_dim),
            nn.ReLU(),
            nn.Linear(mlp_dim, dim)
)

def forward(self, x):
x = x + self.attn(self.norm1(x), self.norm1(x), self.norm1(x))[0]
x = x + self.mlp(self.norm2(x))
return x

self.transformer = nn.Sequential(
*[TransformerEncoderLayer(dim, heads, mlp_dim) for i in range(depth)]
)
```



«depth» times transformer layer applied in sequence

```python
def forward(self, x):

        # Split image into patches
        B = x.size(0)
        x = x.unfold(2, self.patch_size, self.patch_size)\
             .unfold(3, self.patch_size, self.patch_size)\
             .permute(0, 2, 3, 1, 4, 5)\
             .reshape(B, self.num_patches, self.in_ch * self.patch_size * self.patch_size)

        # Embedding + Positional encoding
        x = self.patch_embed(x) + self.pos_embed

        # Transformer encoder
        x = x.permute(1, 0, 2) # (N, B, dim) for nn.MultiheadAttention
        x = self.transformer(x)
        x = x.permute(1, 2, 0) # (B, dim, N)

        # Reshape to 2D feature map
        x = x.reshape(B, self.dim, self.num_cuts, self.num_cuts) # (B, dim, H/16, W/16)

        # Decoder
        x = self.decoder(x) # (B, num_classes, H, W)
        # Now we need values at zero or one

        return torch.sigmoid(x)
```

```
# Decoder
x = self.decoder(x) # (B, num_classes, H, W)
# Now we need values at zero or one

return torch.sigmoid(x)
```




```
decoder = nn.Sequential(
nn.ConvTranspose2d(dim, 256, kernel_size=2, stride=2), nn.ReLU(),
nn.ConvTranspose2d(256, 256, kernel_size=2, stride=2), nn.ReLU(),
nn.ConvTranspose2d(256, 256, kernel_size=2, stride=2), nn.ReLU(),
nn.ConvTranspose2d(256, 256, kernel_size=2, stride=2), nn.ReLU(),
nn.Conv2d(256, 1, kernel_size=1))
```

```
class SETR(nn.Module):

def __init__(self, img_size=128, num_cuts = 4,in_ch=3, num_classes=1, dim=1024, depth=8, heads=8, mlp_dim=2048):
        super().__init__()

        # Internal parameters definition:
        self.patch_size  = int(W / num_cuts)
        self.num_cuts    = num_cuts
        self.num_patches = self.num_cuts ** 2
        self.dim = dim
        self.in_ch = in_ch

        # Patch + Positional Embedding. We need to redefine this due to generalization...
        self.patch_embed = nn.Linear(self.patch_size * self.patch_size * in_ch, dim)
        self.pos_embed   = nn.Parameter(torch.randn(1, self.num_patches, dim))

        # Transformer Encoder. We refer to the previously defined class
        self.transformer = nn.Sequential(
        *[TransformerEncoderLayer(dim, heads, mlp_dim) for _ in range(depth)]
        )

        # Deconvolution decoder
        # Notice that is the same of before byt dimension here is generic
        self.decoder = nn.Sequential(
        nn.ConvTranspose2d(dim, 256, kernel_size=2, stride=2), nn.ReLU(),
        nn.ConvTranspose2d(256, 256, kernel_size=2, stride=2), nn.ReLU(),
        nn.ConvTranspose2d(256, 256, kernel_size=2, stride=2), nn.ReLU(),
        nn.ConvTranspose2d(256, 256, kernel_size=2, stride=2), nn.ReLU(),
        nn.Conv2d(256, num_classes, kernel_size=1))

def forward(self, x):

        # Split image into patches
        B = x.size(0)
        x = x.unfold(2, self.patch_size, self.patch_size)\
             .unfold(3, self.patch_size, self.patch_size)\
             .permute(0, 2, 3, 1, 4, 5)\
             .reshape(B, self.num_patches, self.in_ch * self.patch_size * self.patch_size)

        # Embedding + Positional encoding
        x = self.patch_embed(x) + self.pos_embed

        # Transformer encoder
        x = x.permute(1, 0, 2) # (N, B, dim) for nn.MultiheadAttention
        x = self.transformer(x)
        x = x.permute(1, 2, 0) # (B, dim, N)

        # Reshape to 2D feature map
        x = x.reshape(B, self.dim, self.num_cuts, self.num_cuts) # (B, dim, H/16, W/16)

        # Decoder
        x = self.decoder(x) # (B, num_classes, H, W)
        # Now we need values at zero or one

        return torch.sigmoid(x)
```

SETR with Dice Loss
Training Informations

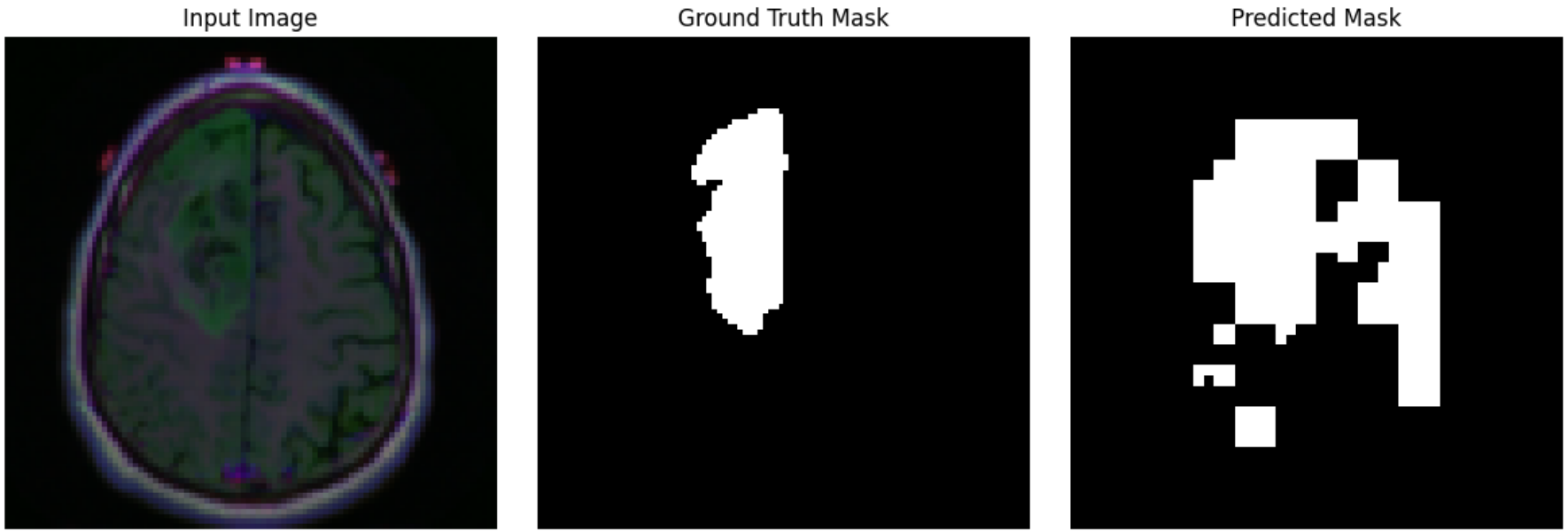
Adam optimizer with 10^{-4} learning rate

~ 350 training epochs

Saved an image and mask prediction, so to make available the training region evolution visualization…

SETR with Dice Loss

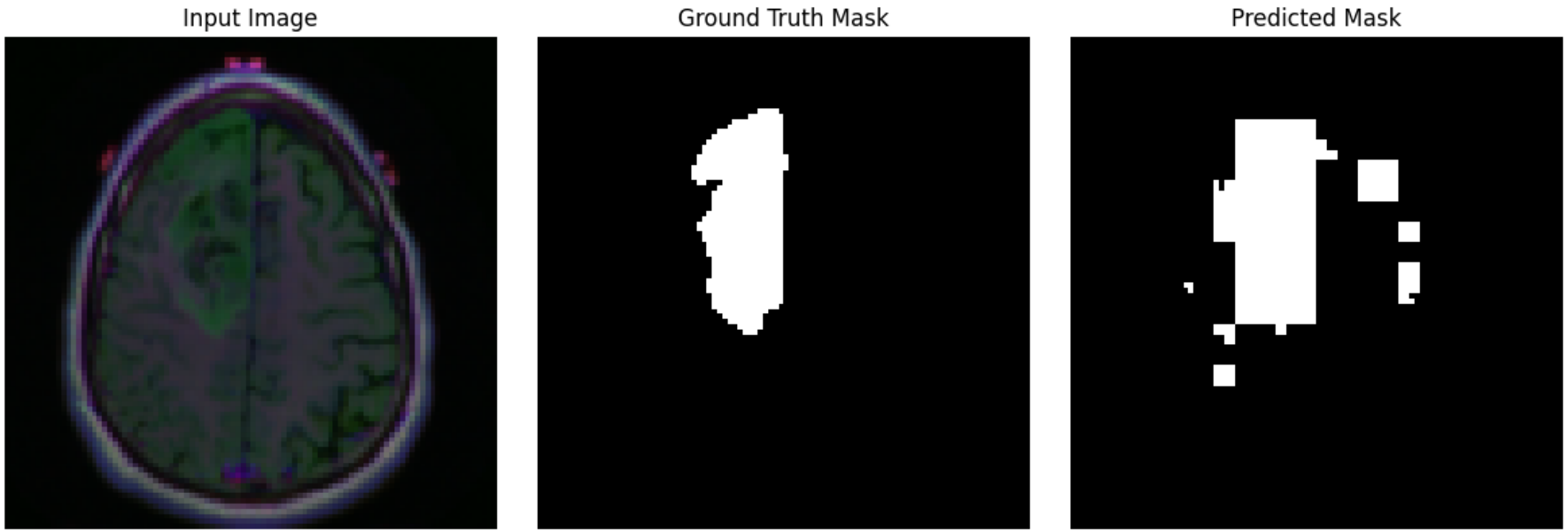
Visualizing the training region evolution



Epoche: 10 + 8 + 20

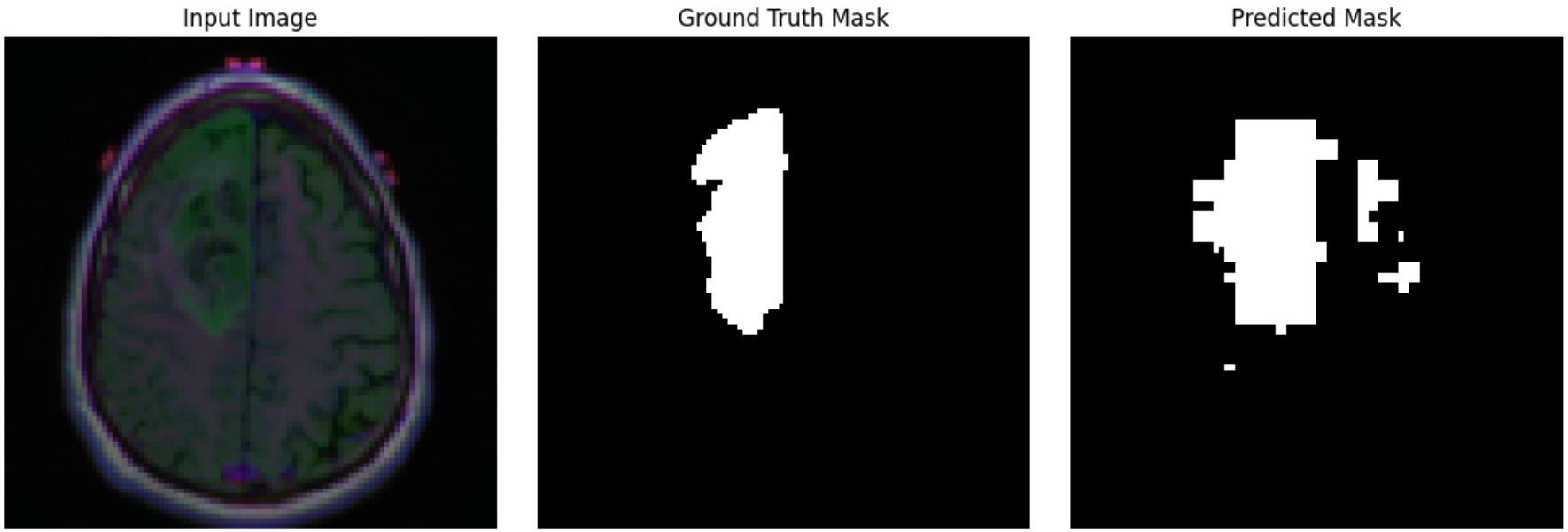
SETR with Dice Loss

Visualizing the training region evolution



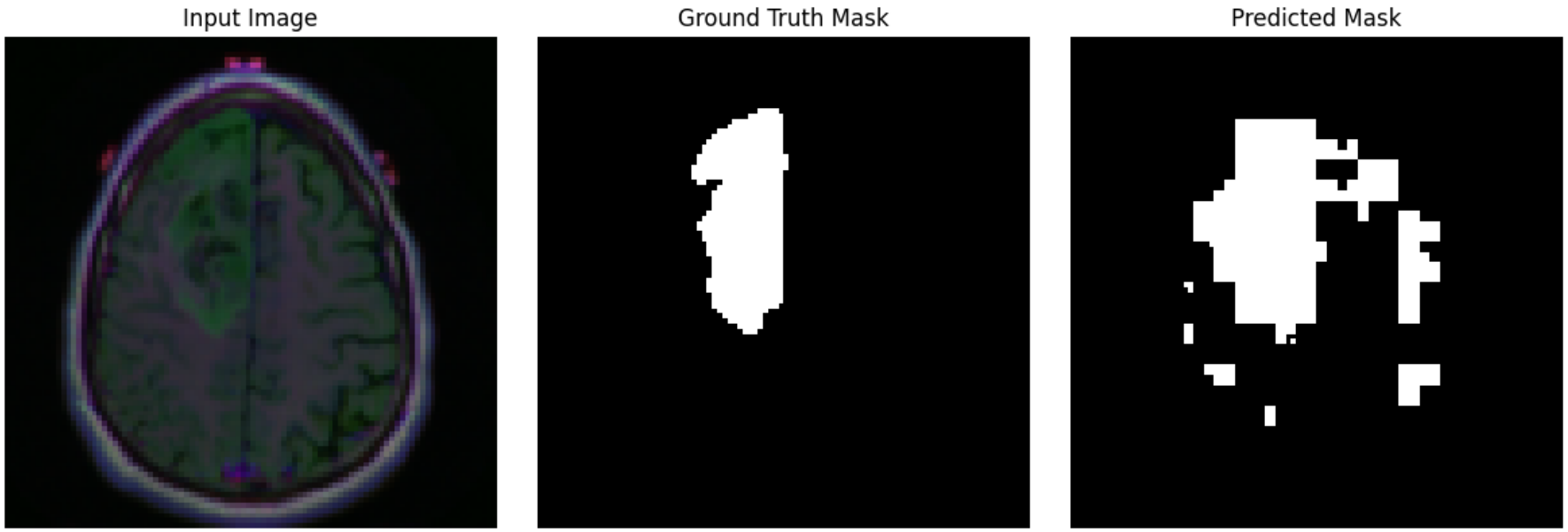
SETR with Dice Loss

Visualizing the training region evolution



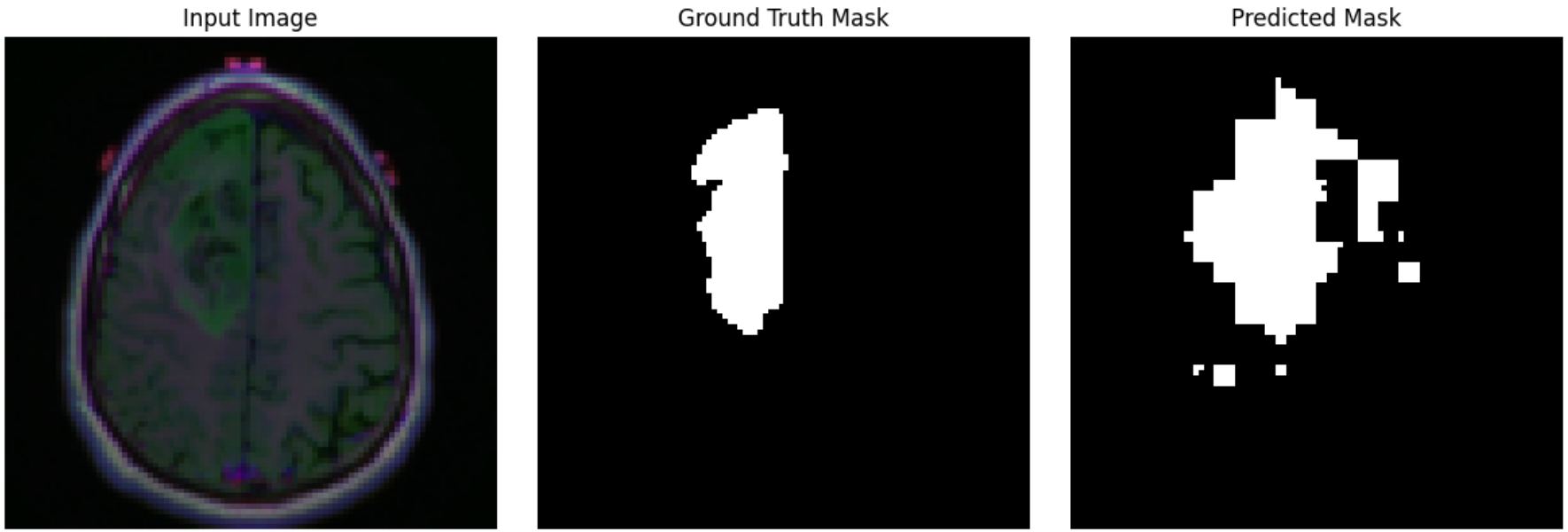
SETR with Dice Loss

Visualizing the training region evolution



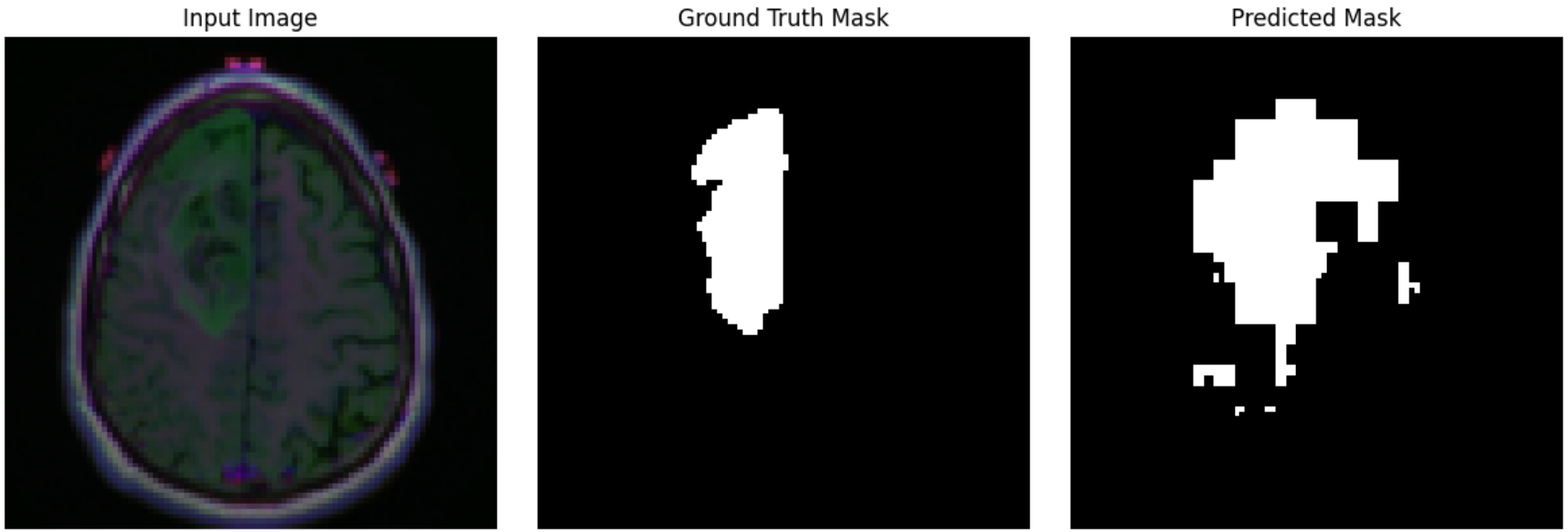
SETR with Dice Loss

Visualizing the training region evolution



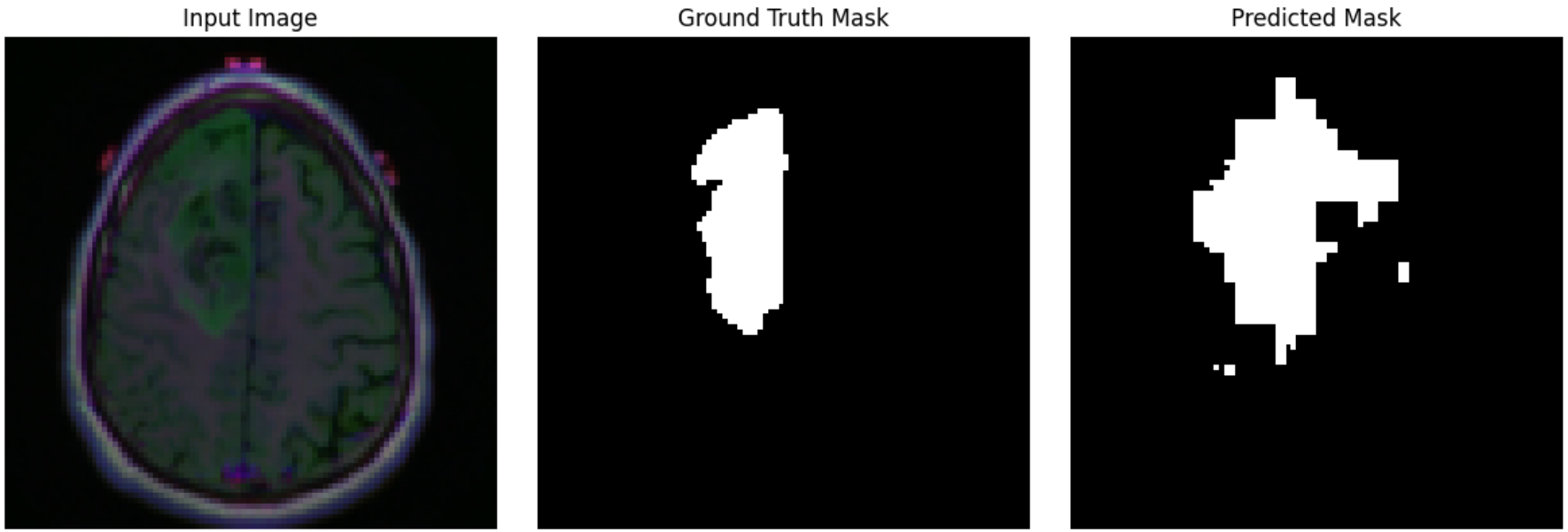
SETR with Dice Loss

Visualizing the training region evolution



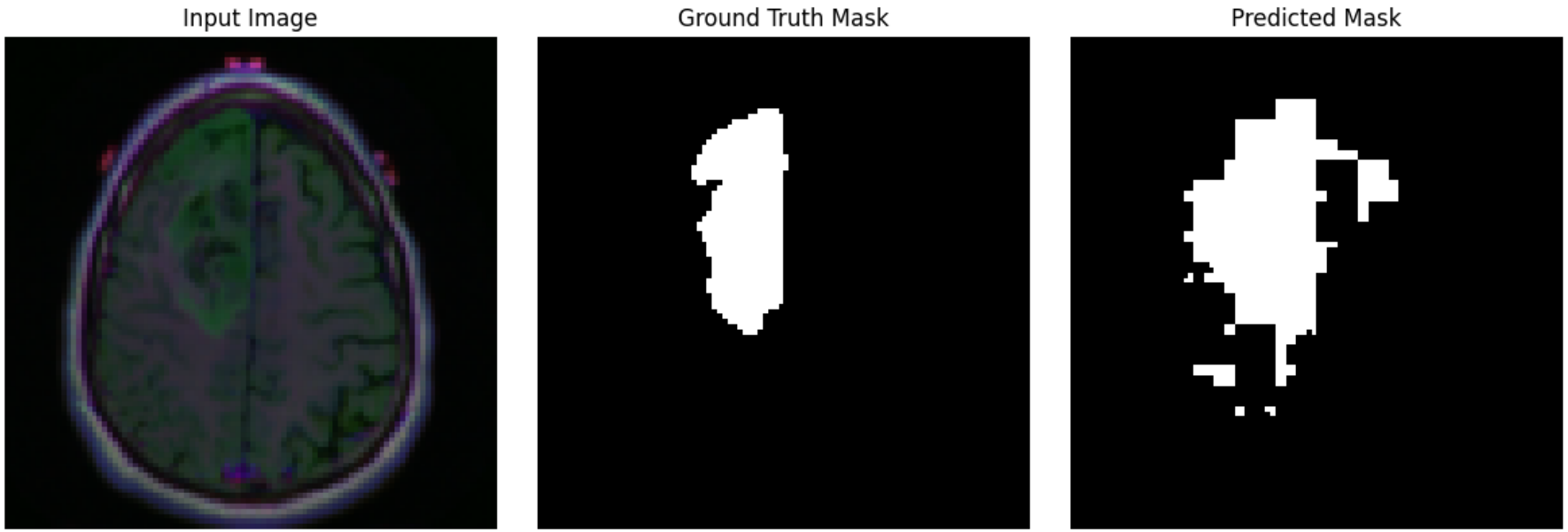
SETR with Dice Loss

Visualizing the training region evolution



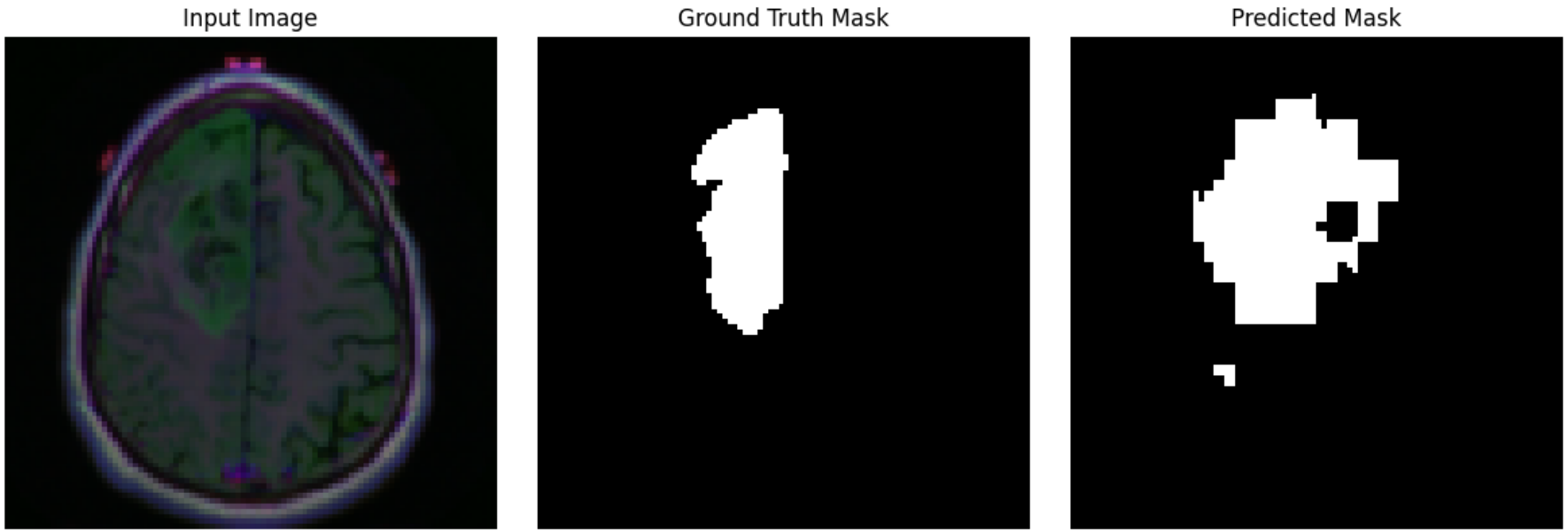
SETR with Dice Loss

Visualizing the training region evolution



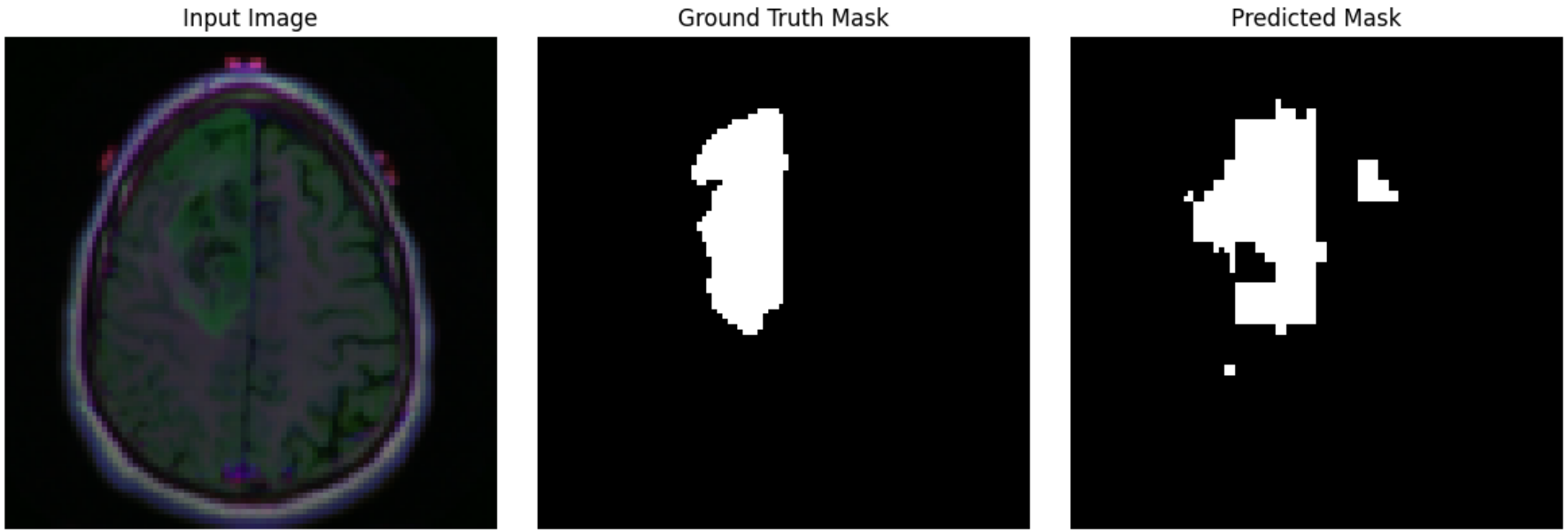
SETR with Dice Loss

Visualizing the training region evolution



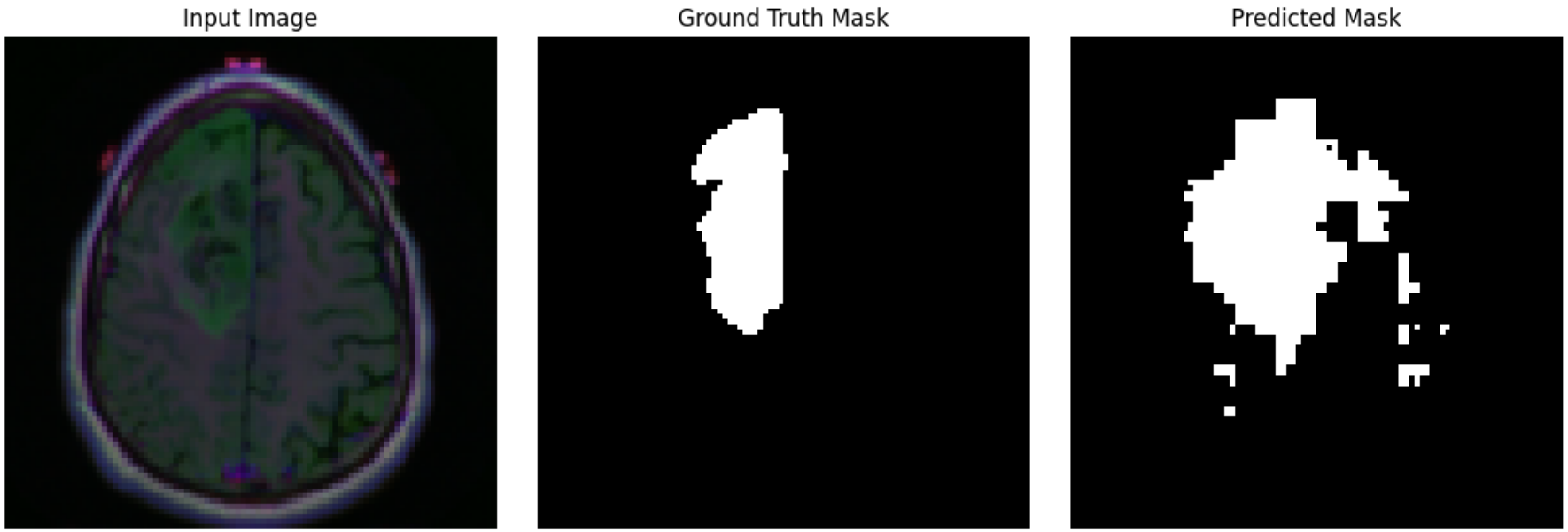
SETR with Dice Loss

Visualizing the training region evolution



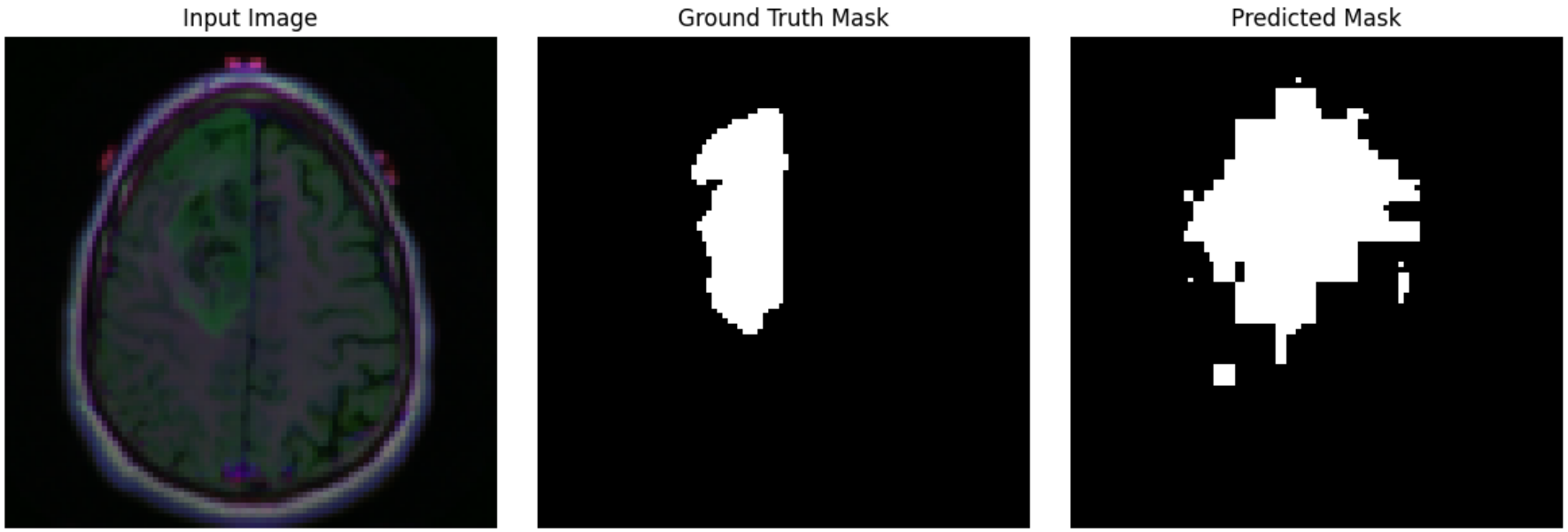
SETR with Dice Loss

Visualizing the training region evolution



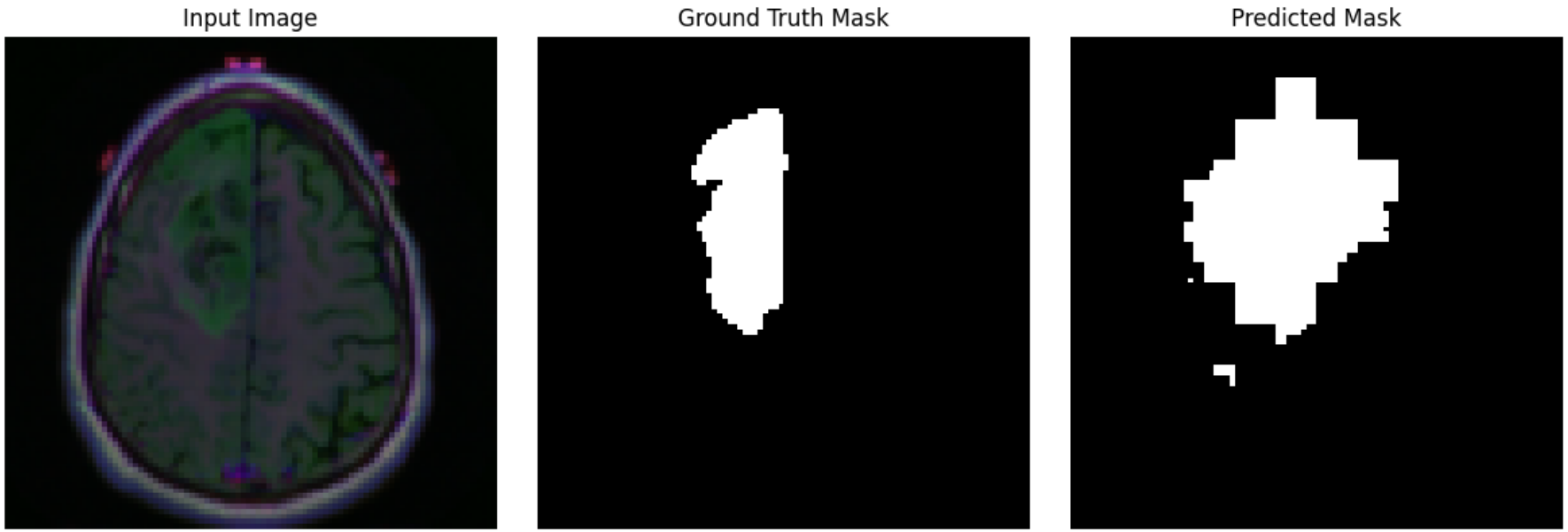
SETR with Dice Loss

Visualizing the training region evolution



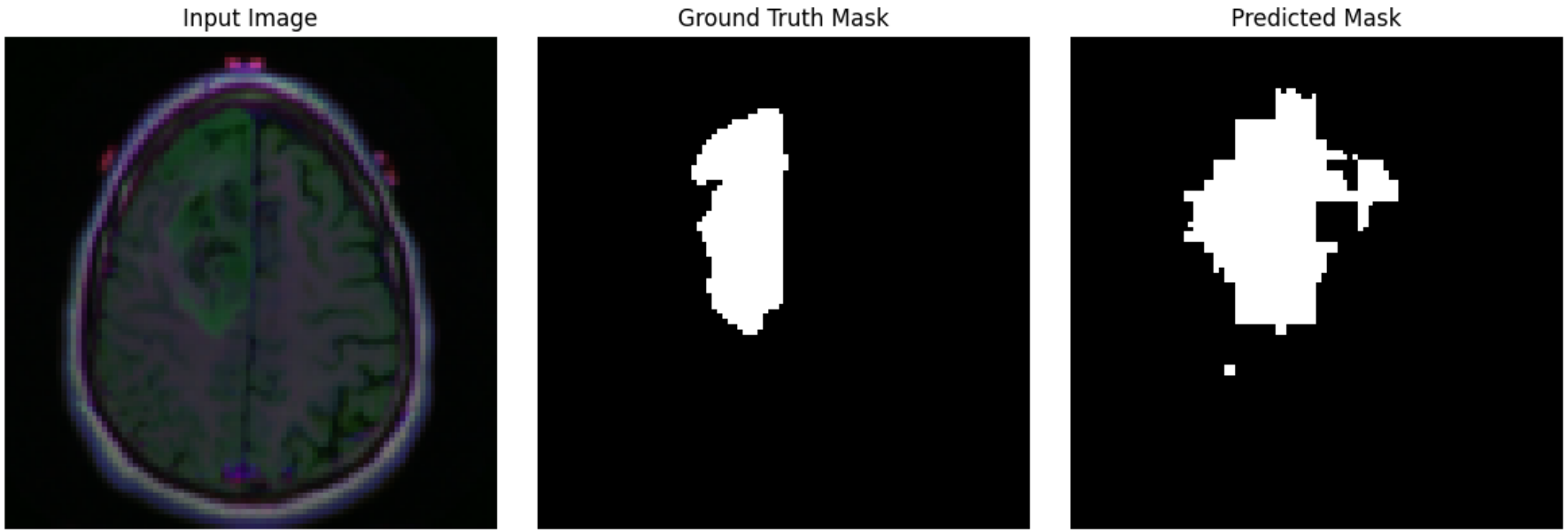
SETR with Dice Loss

Visualizing the training region evolution



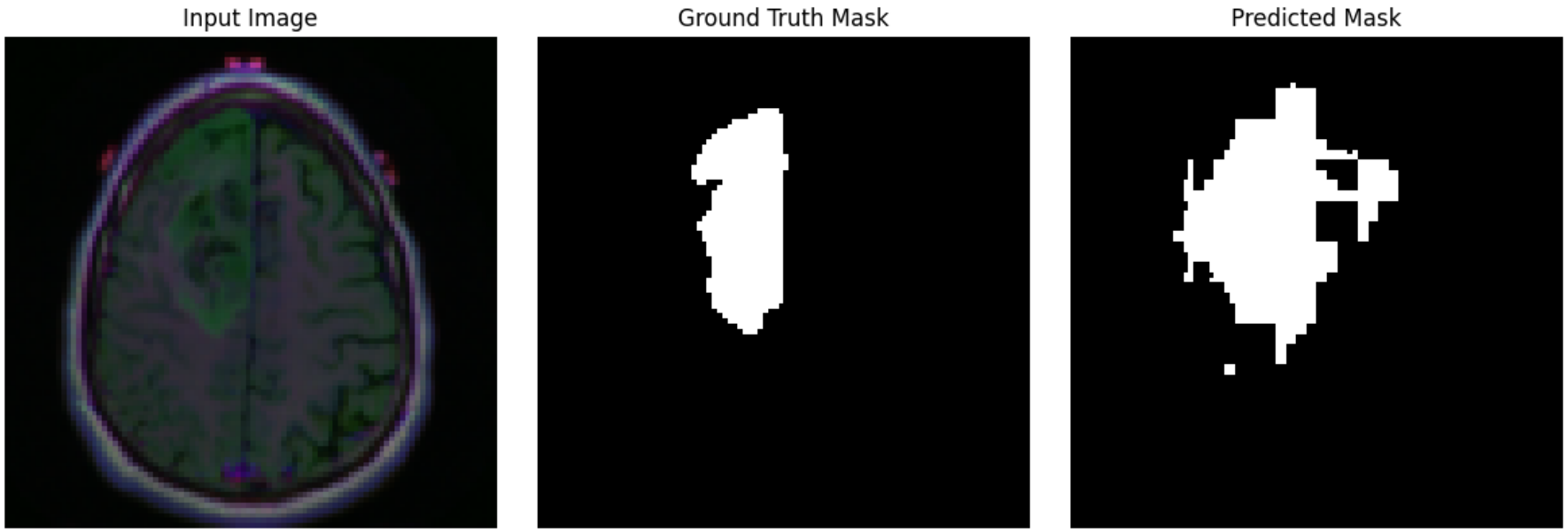
SETR with Dice Loss

Visualizing the training region evolution



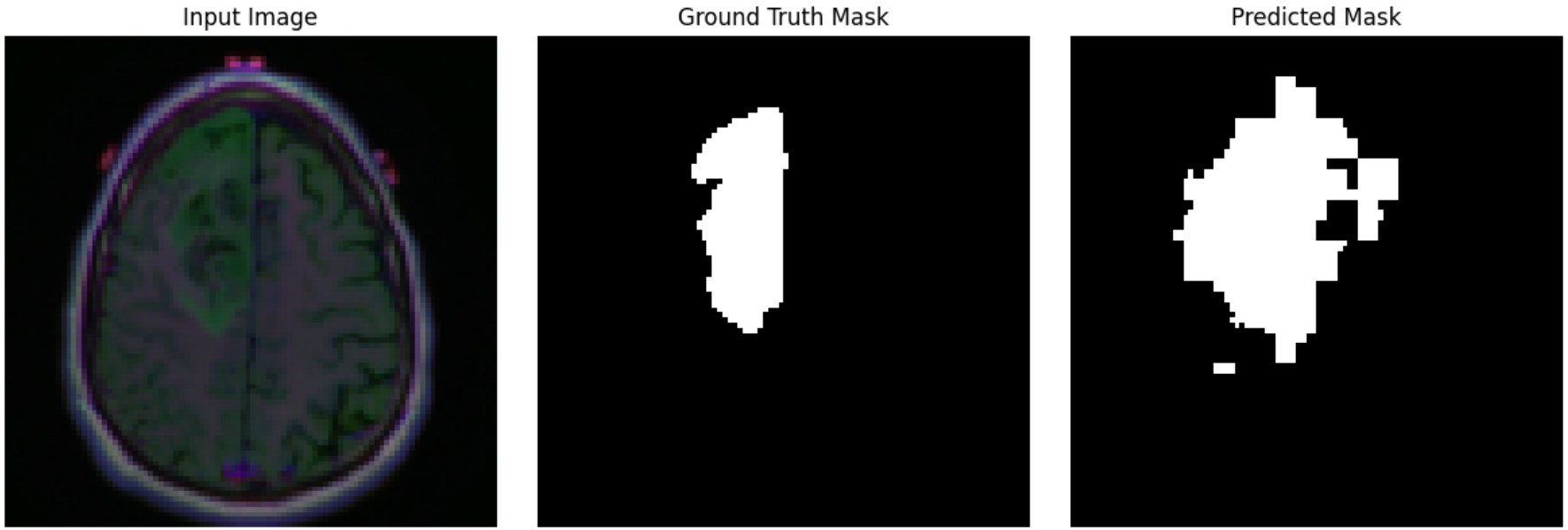
SETR with Dice Loss

Visualizing the training region evolution



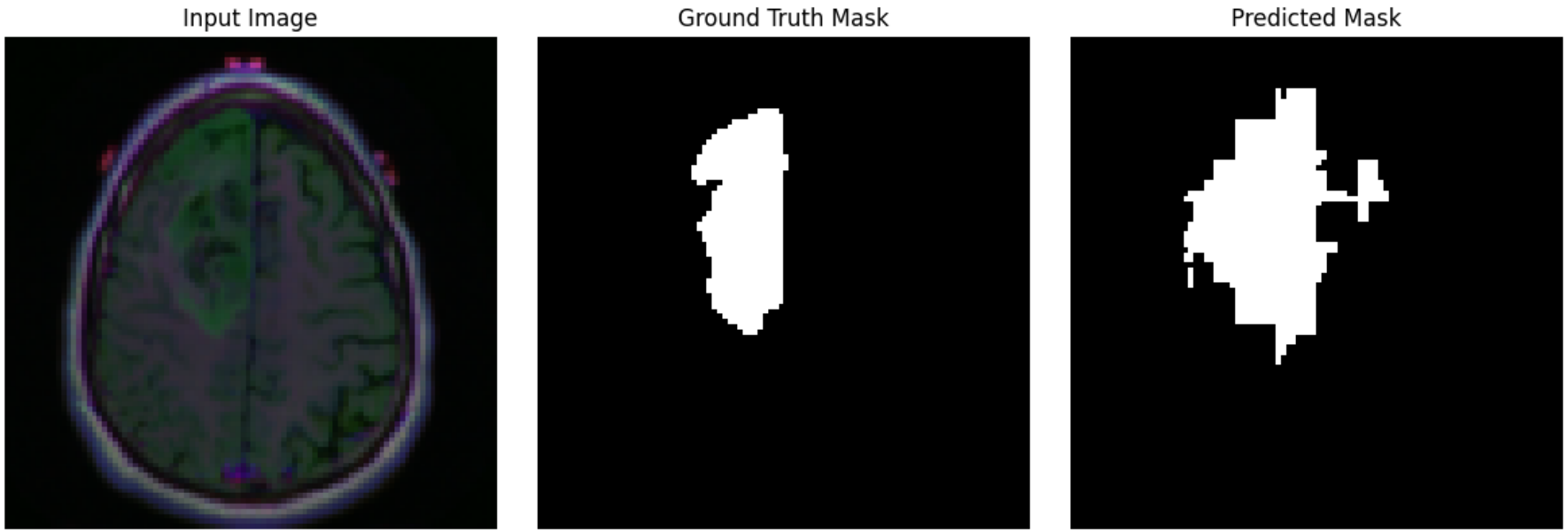
SETR with Dice Loss

Visualizing the training region evolution



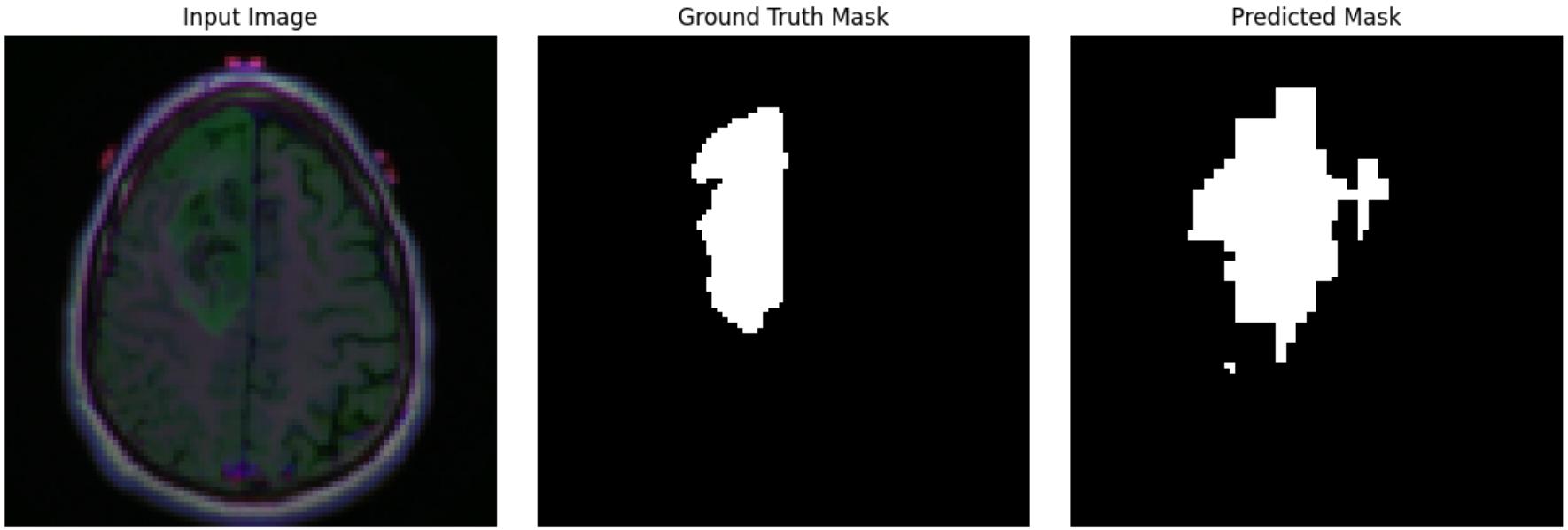
SETR with Dice Loss

Visualizing the training region evolution



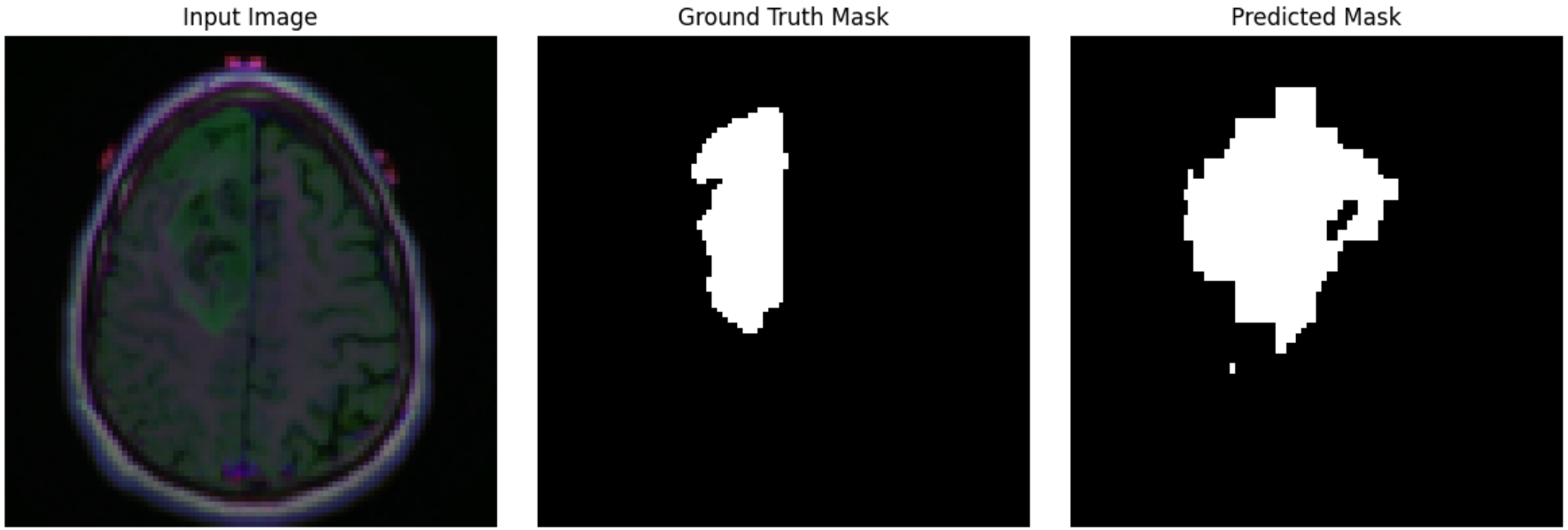
SETR with Dice Loss

Visualizing the training region evolution



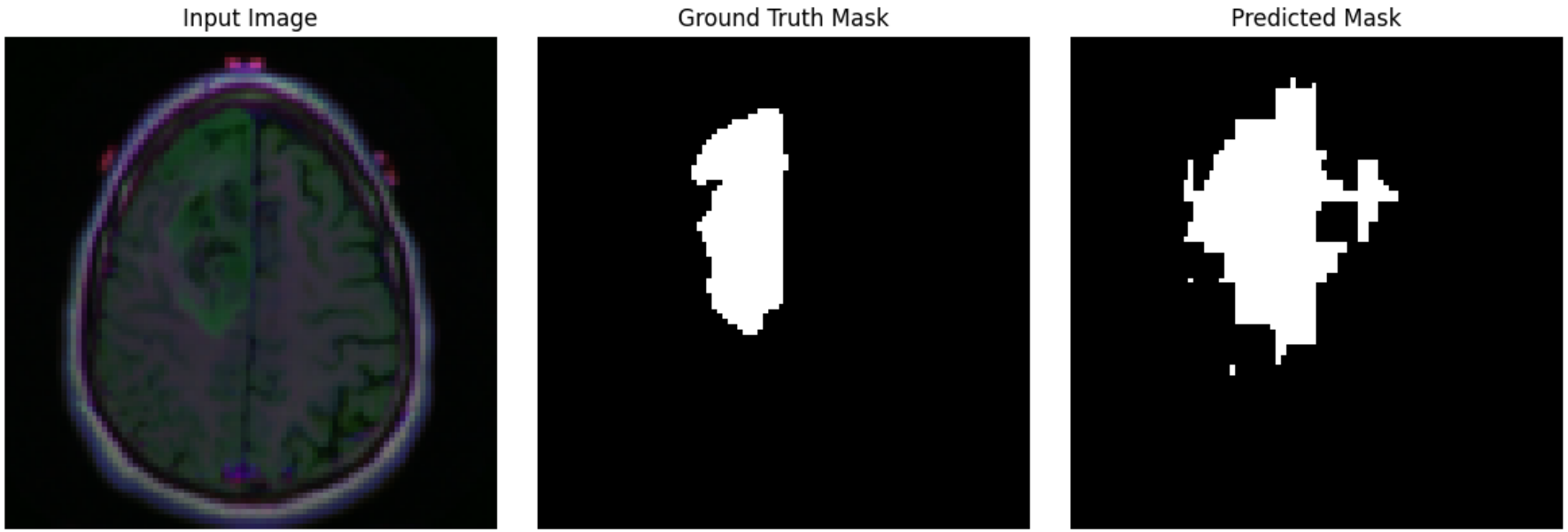
SETR with Dice Loss

Visualizing the training region evolution



SETR with Dice Loss

Visualizing the training region evolution

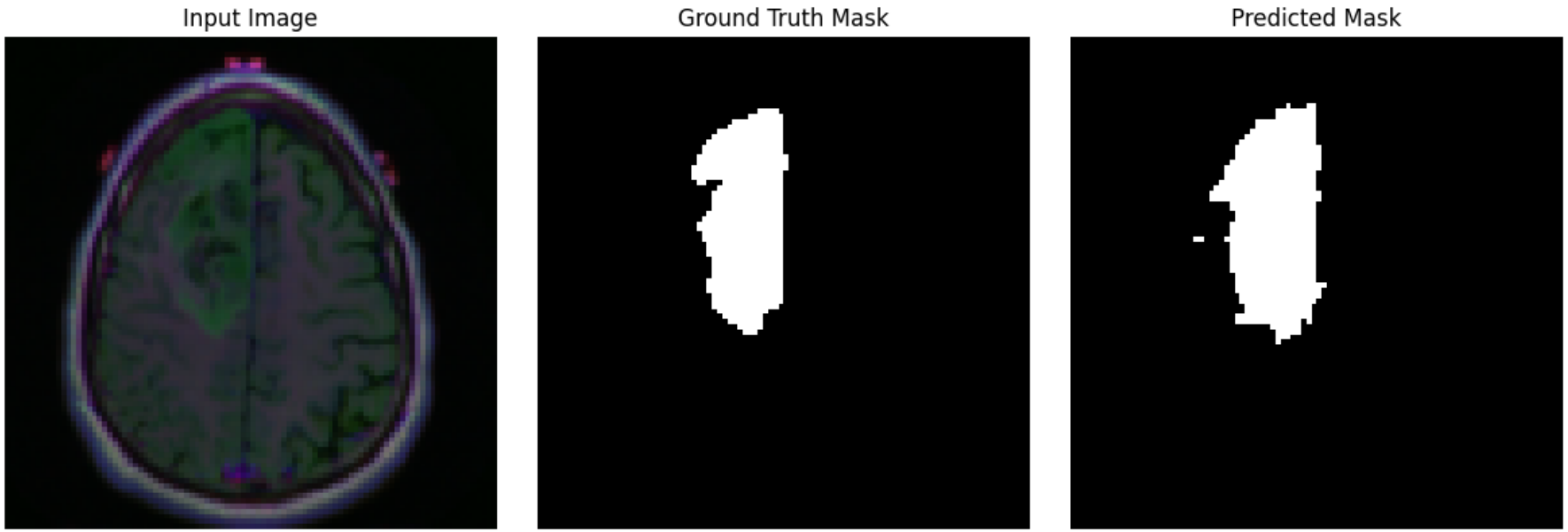


SETR with Dice Loss
Visualizing the training region evolution

But after 300 learning epochs, i.e. ~6 training hours

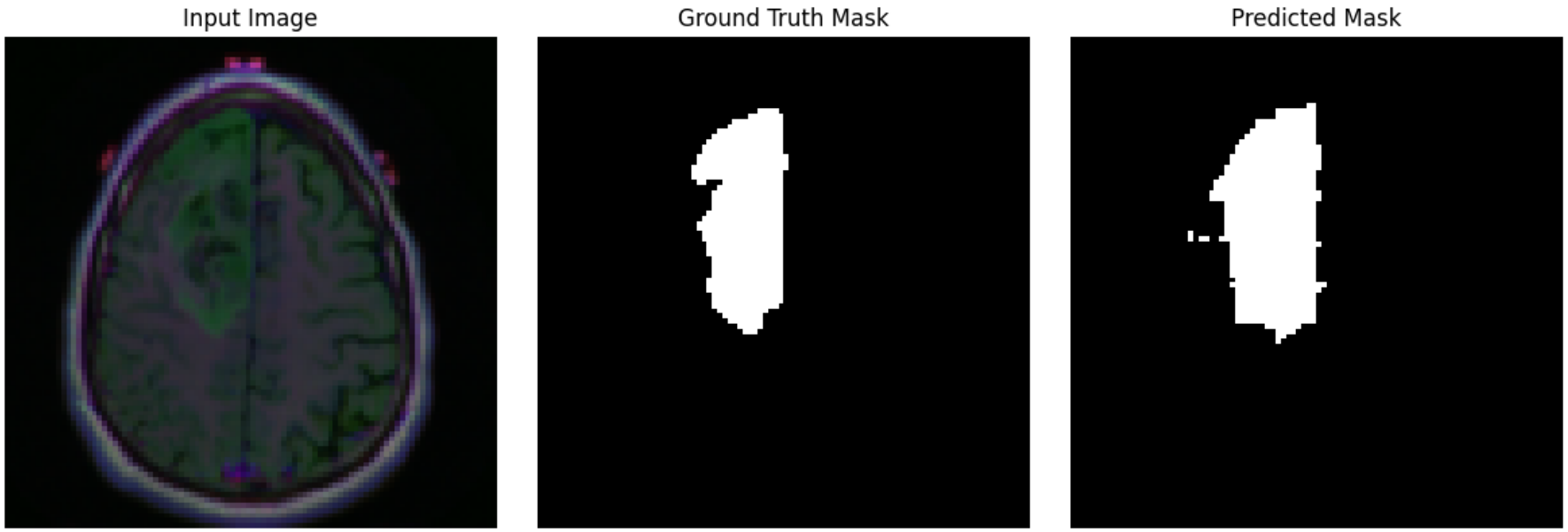
SETR with Dice Loss

Visualizing the training region evolution



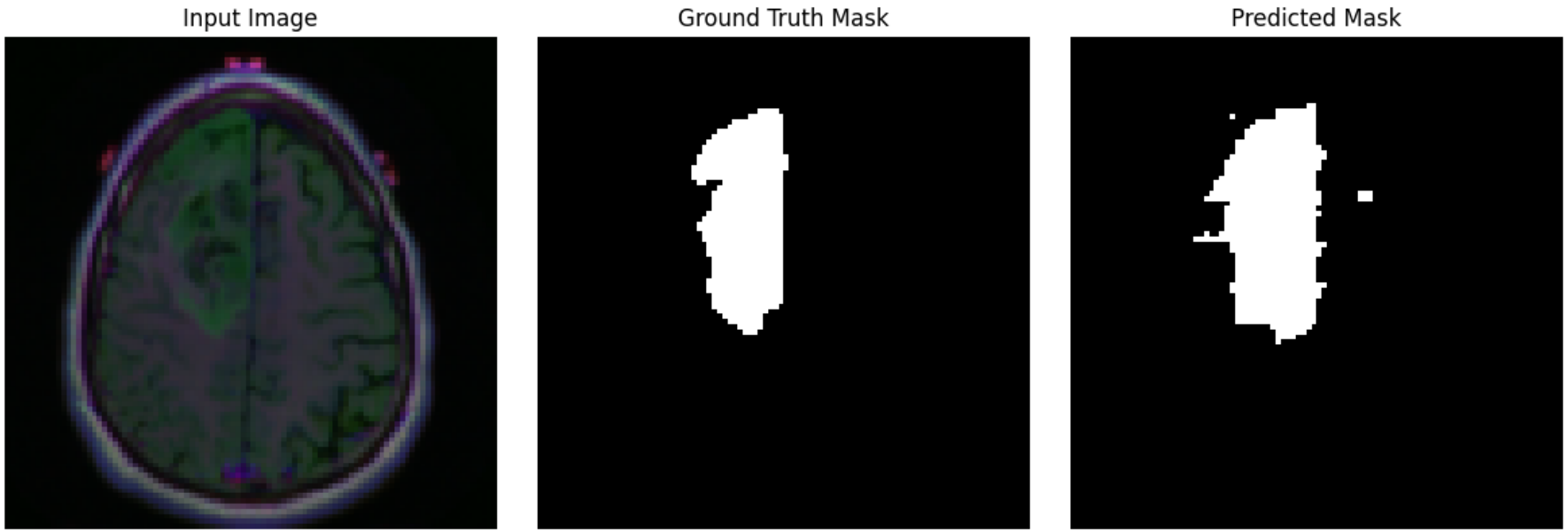
SETR with Dice Loss

Visualizing the training region evolution



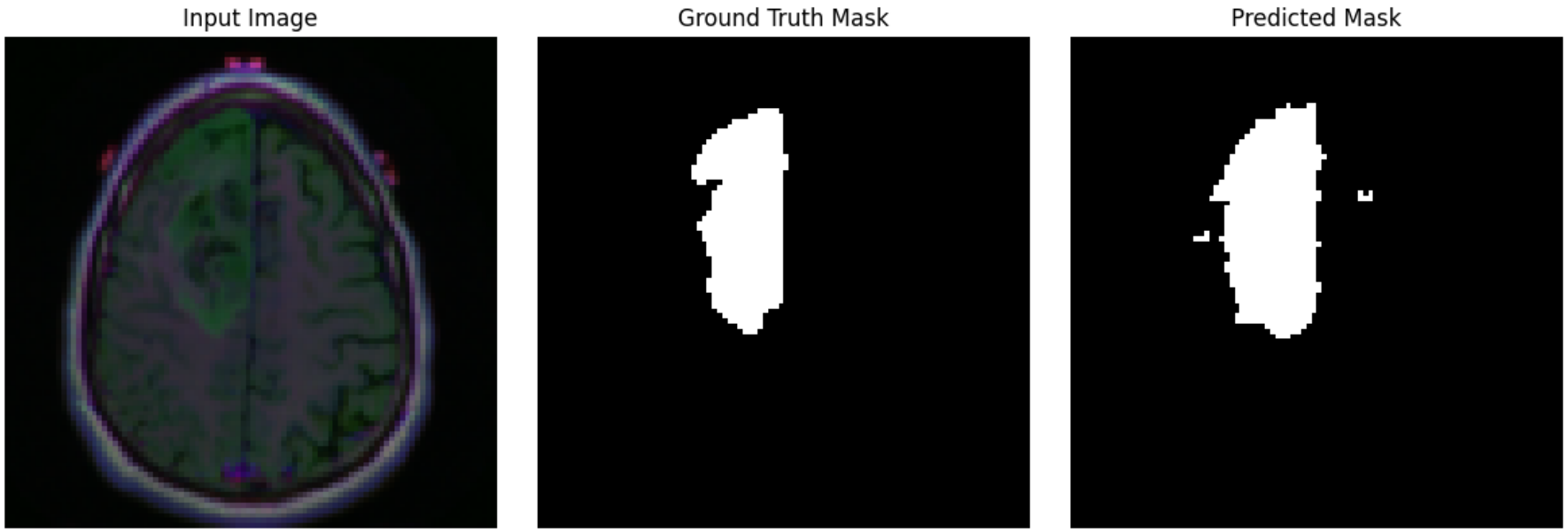
SETR with Dice Loss

Visualizing the training region evolution



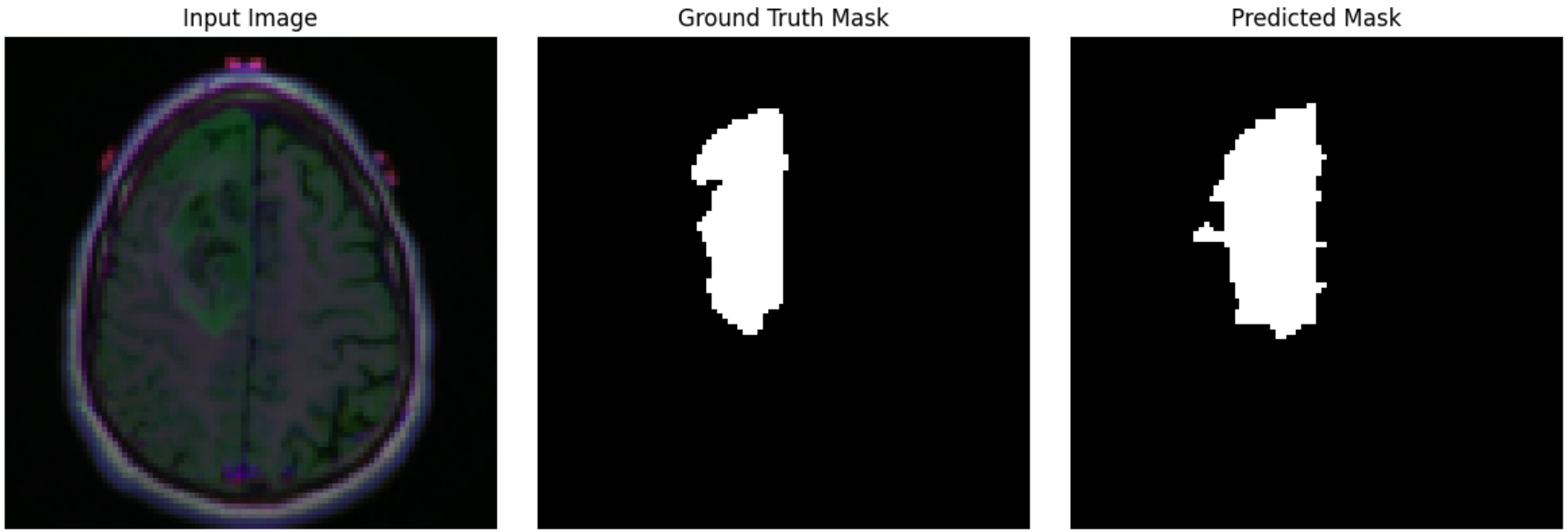
SETR with Dice Loss

Visualizing the training region evolution



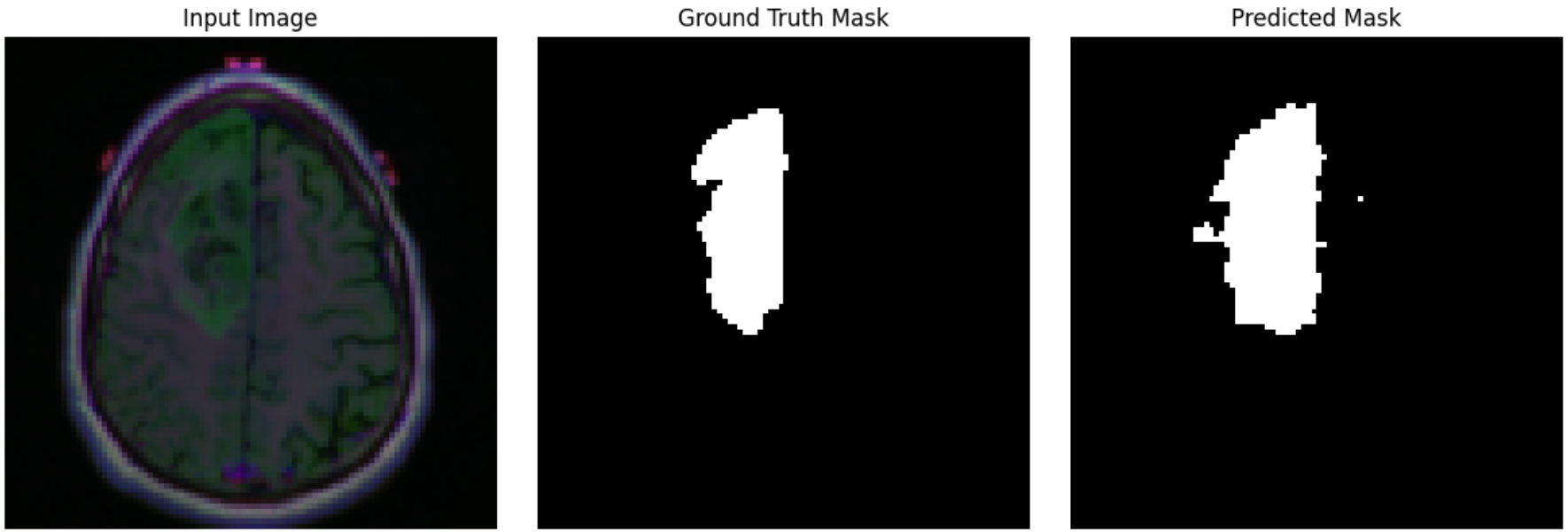
SETR with Dice Loss

Visualizing the training region evolution



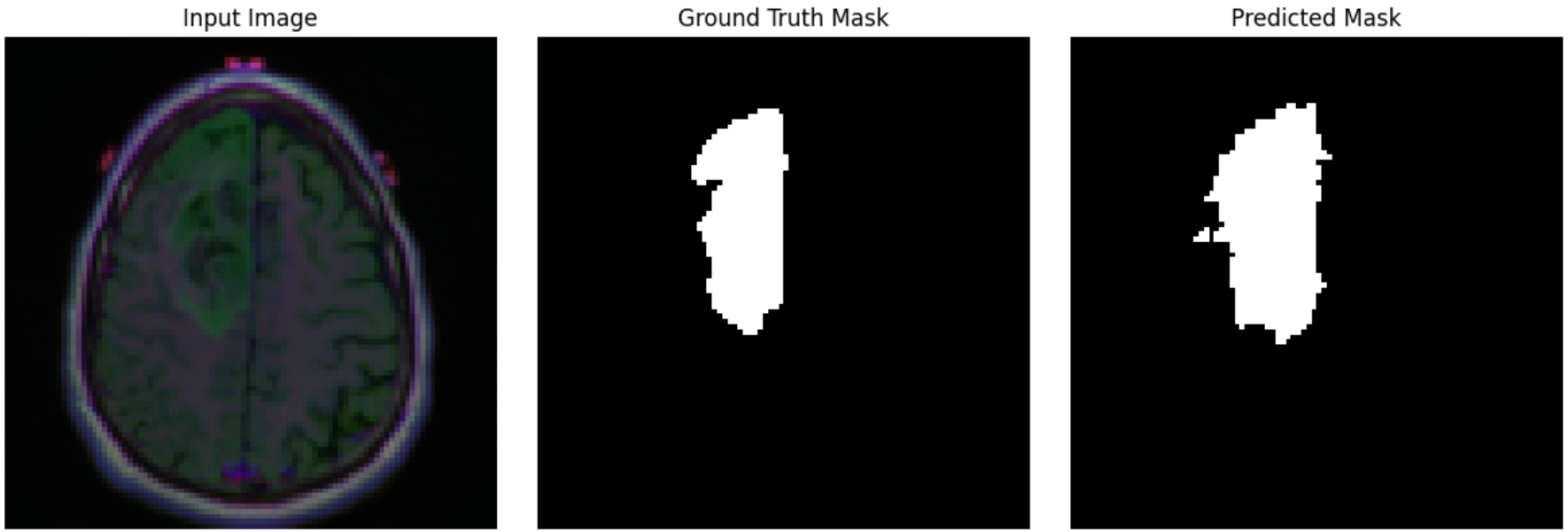
SETR with Dice Loss

Visualizing the training region evolution



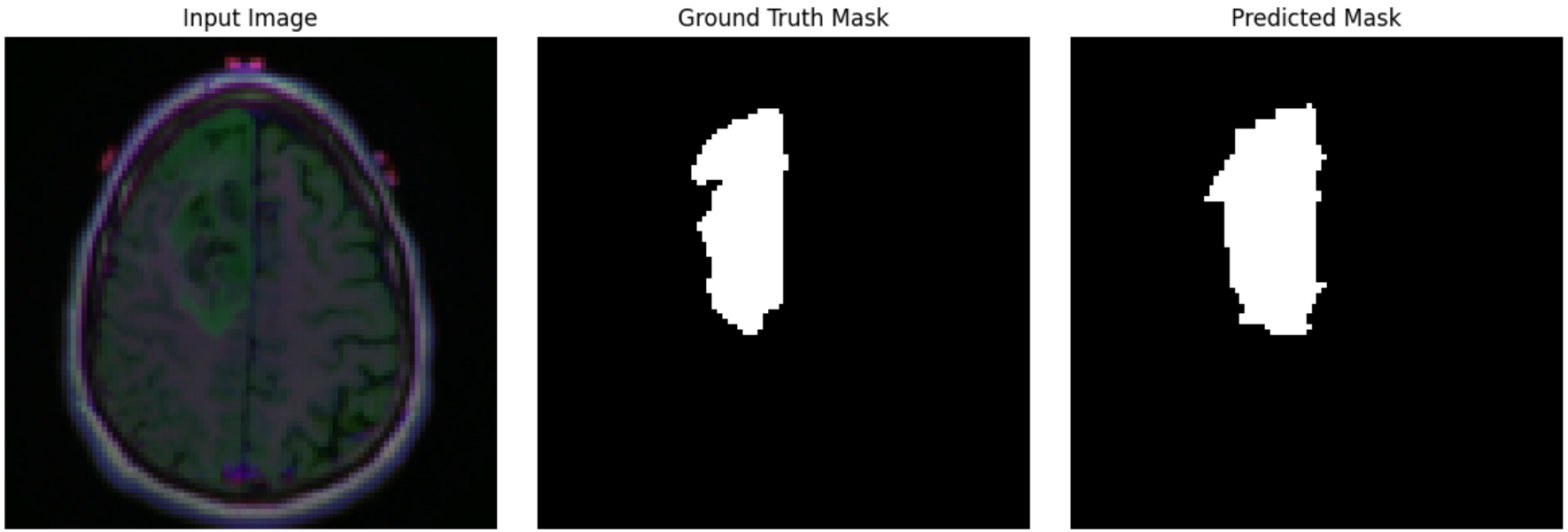
SETR with Dice Loss

Visualizing the training region evolution



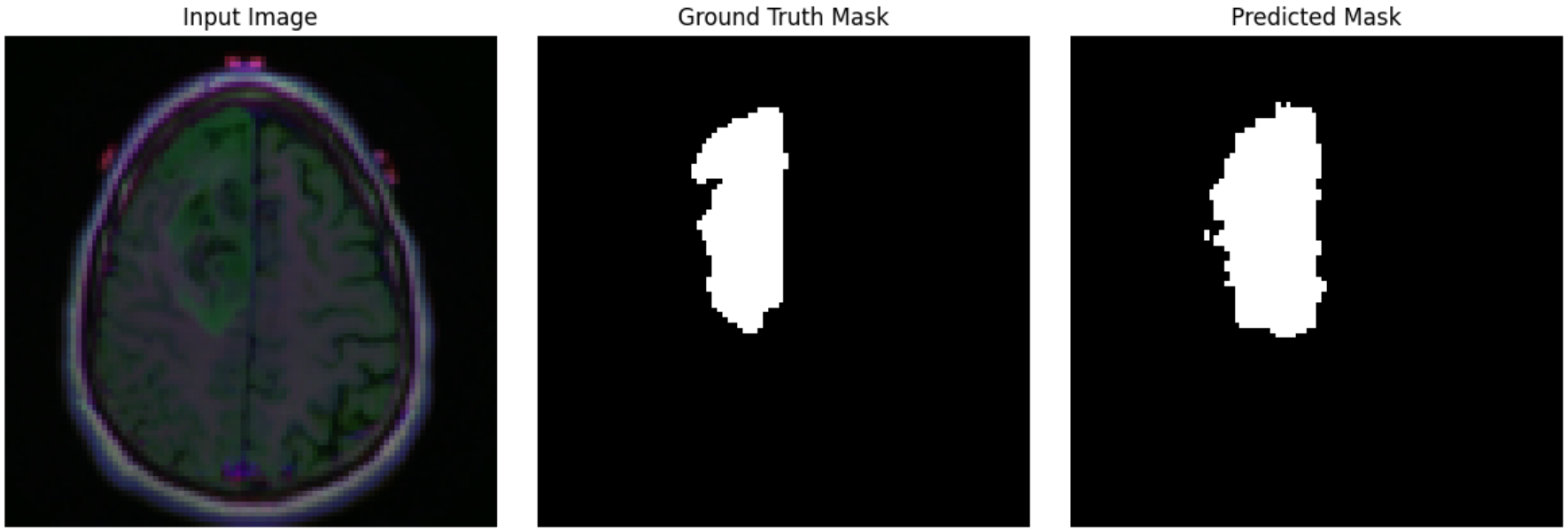
SETR with Dice Loss

Visualizing the training region evolution



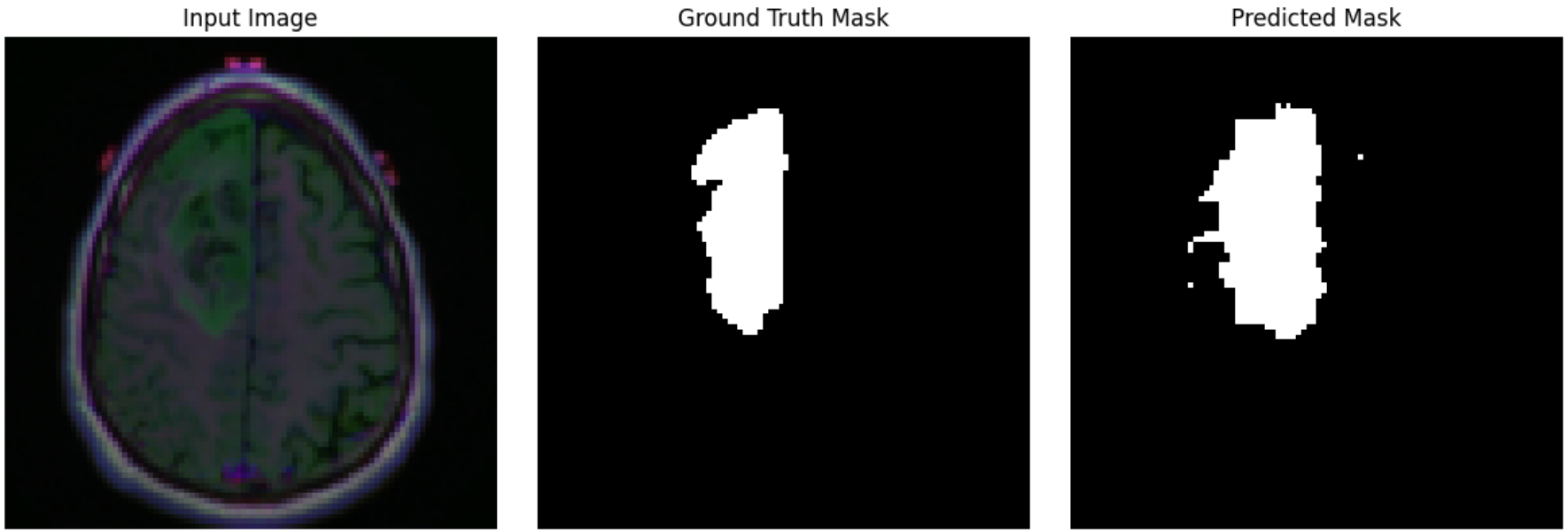
SETR with Dice Loss

Visualizing the training region evolution



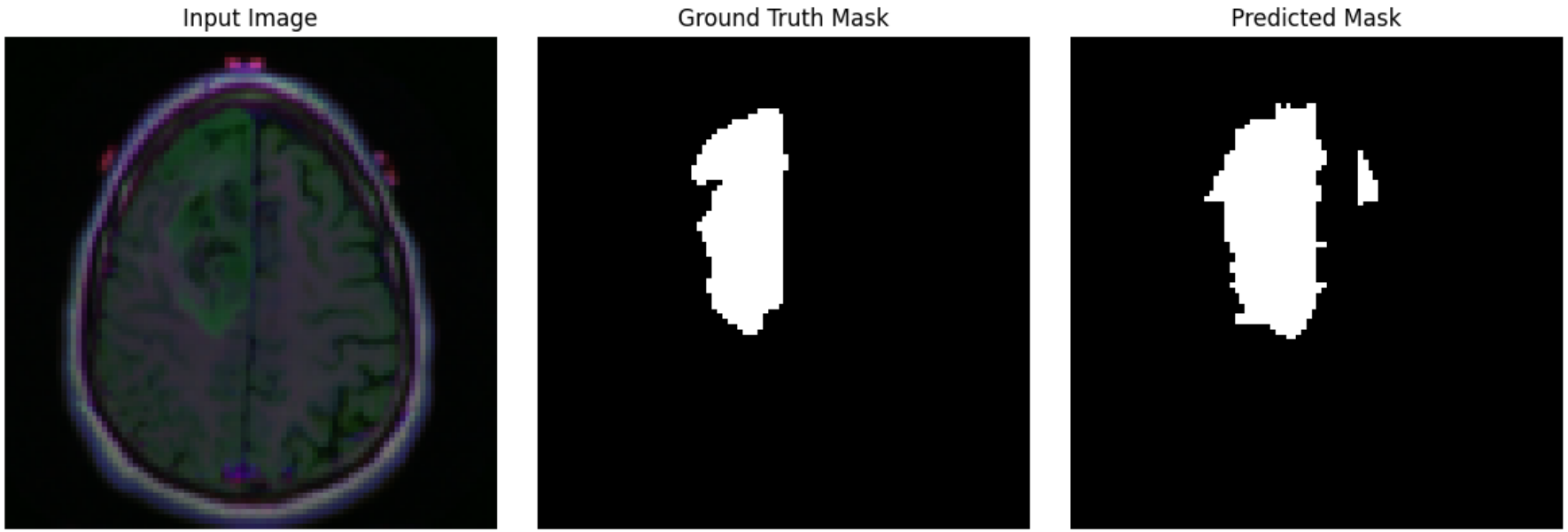
SETR with Dice Loss

Visualizing the training region evolution



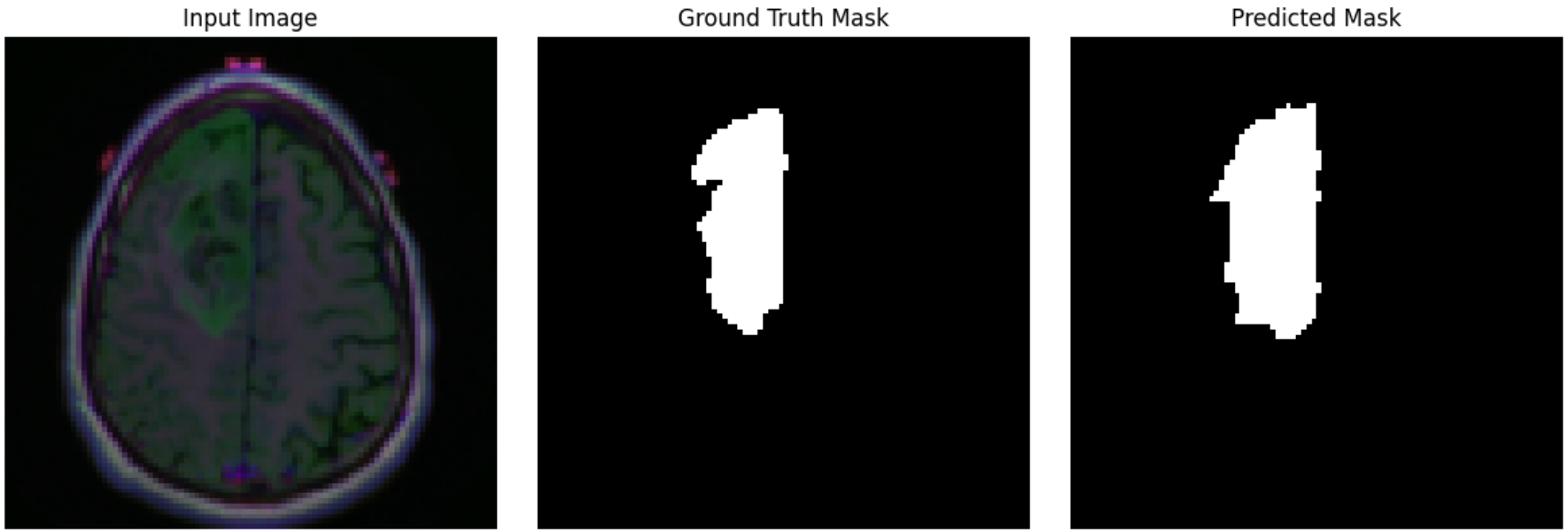
SETR with Dice Loss

Visualizing the training region evolution



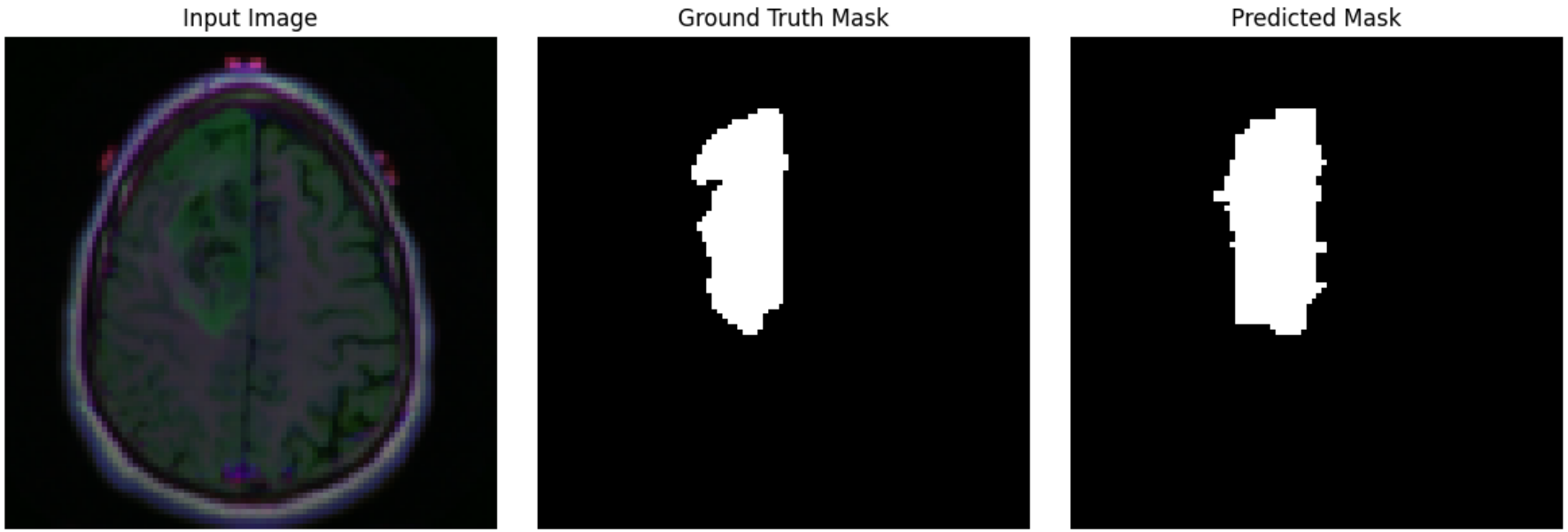
SETR with Dice Loss

Visualizing the training region evolution



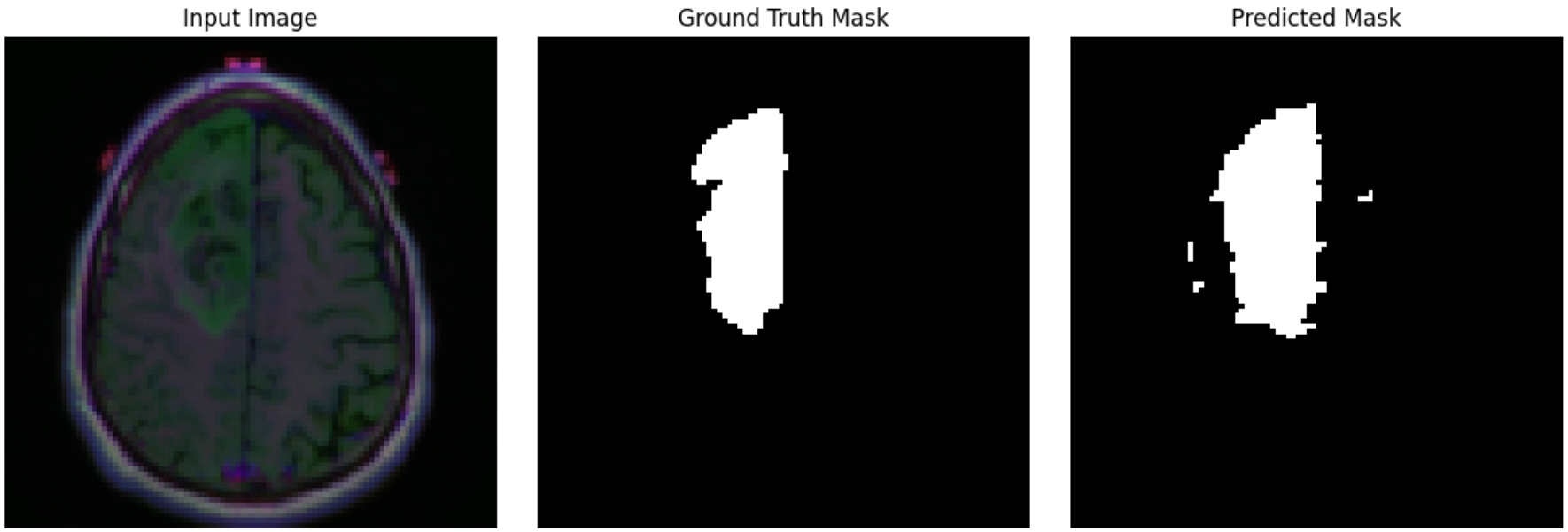
SETR with Dice Loss

Visualizing the training region evolution



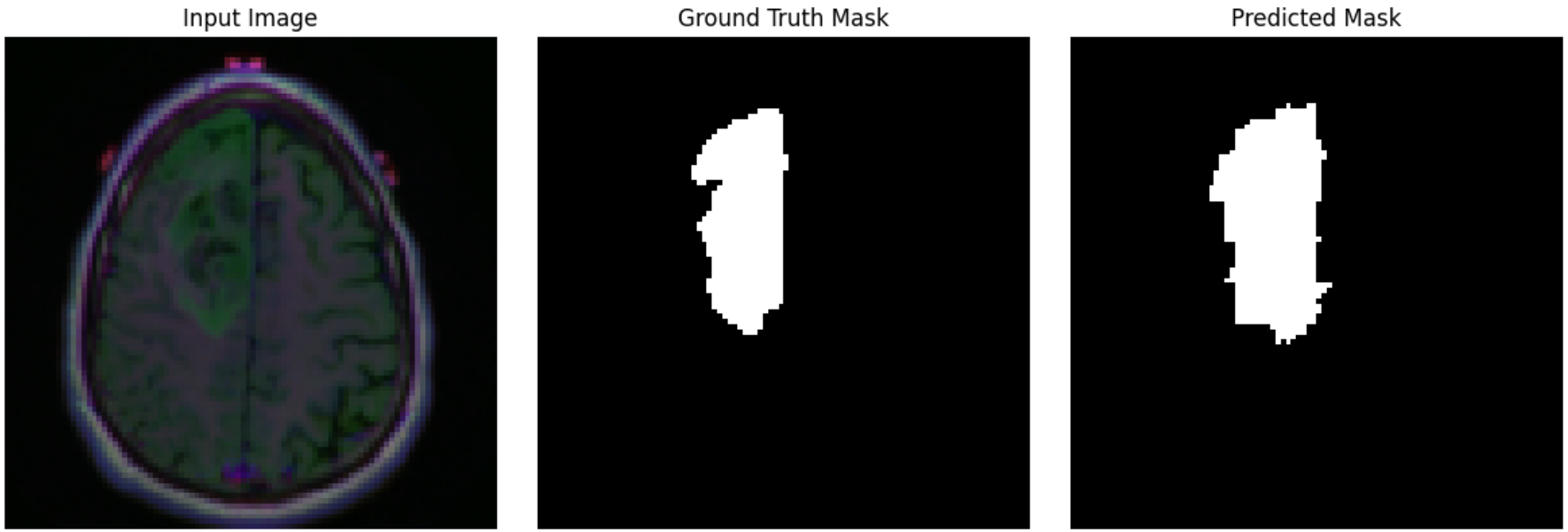
SETR with Dice Loss

Visualizing the training region evolution



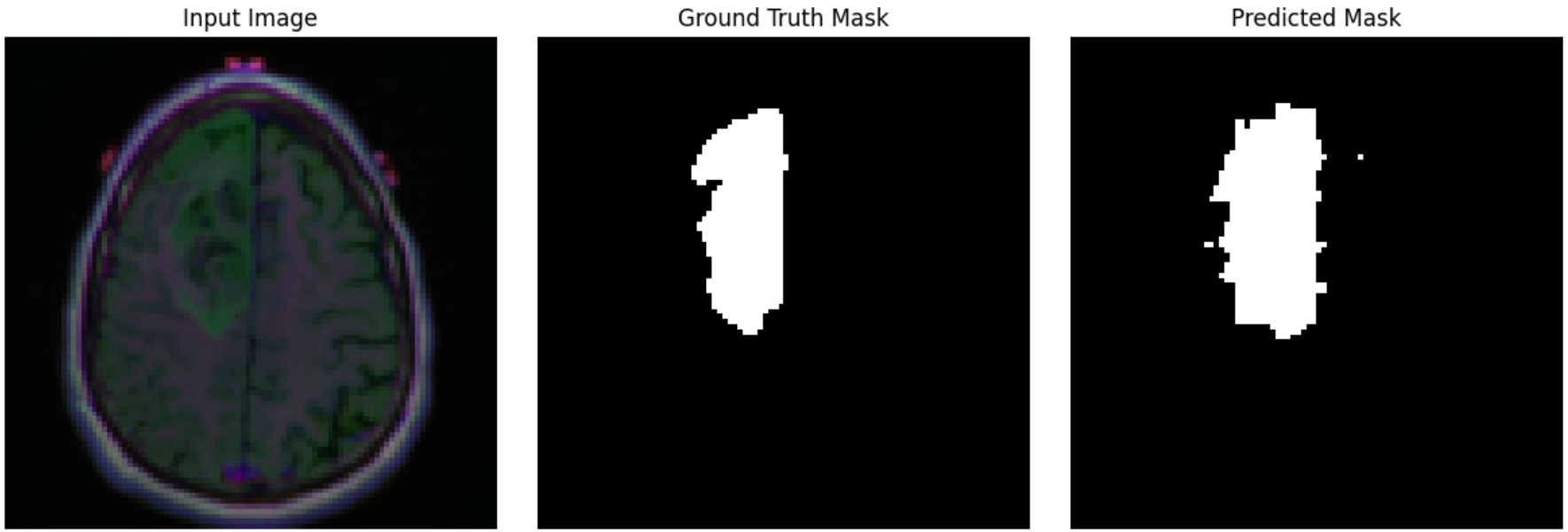
SETR with Dice Loss

Visualizing the training region evolution



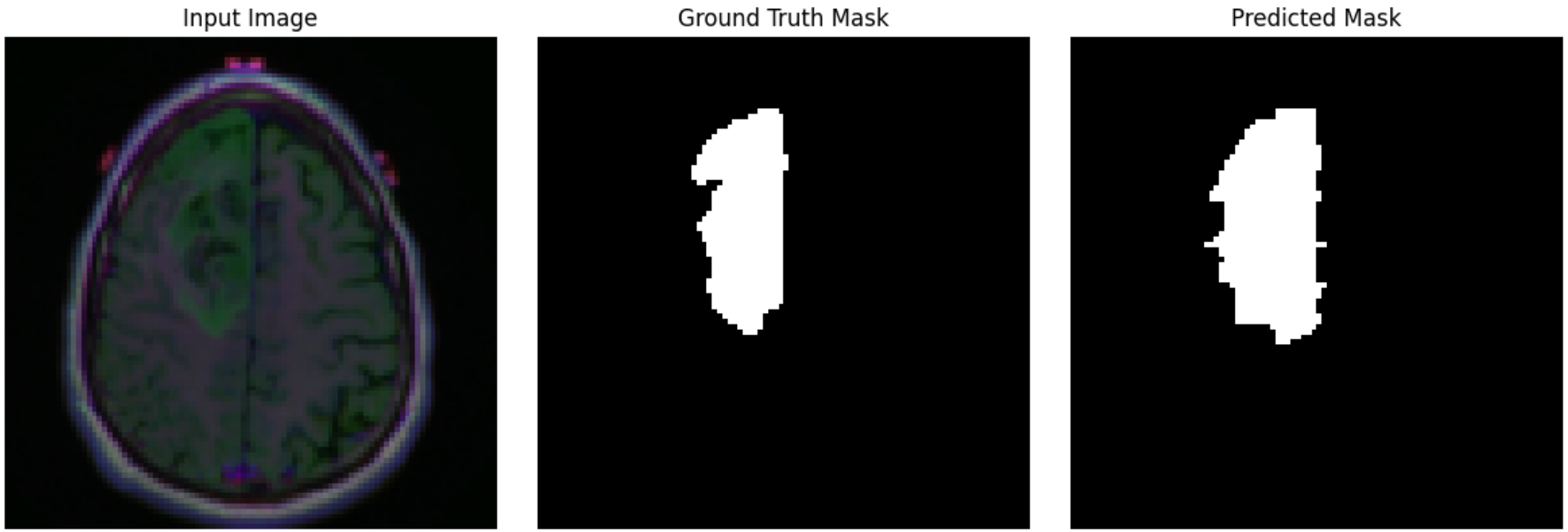
SETR with Dice Loss

Visualizing the training region evolution



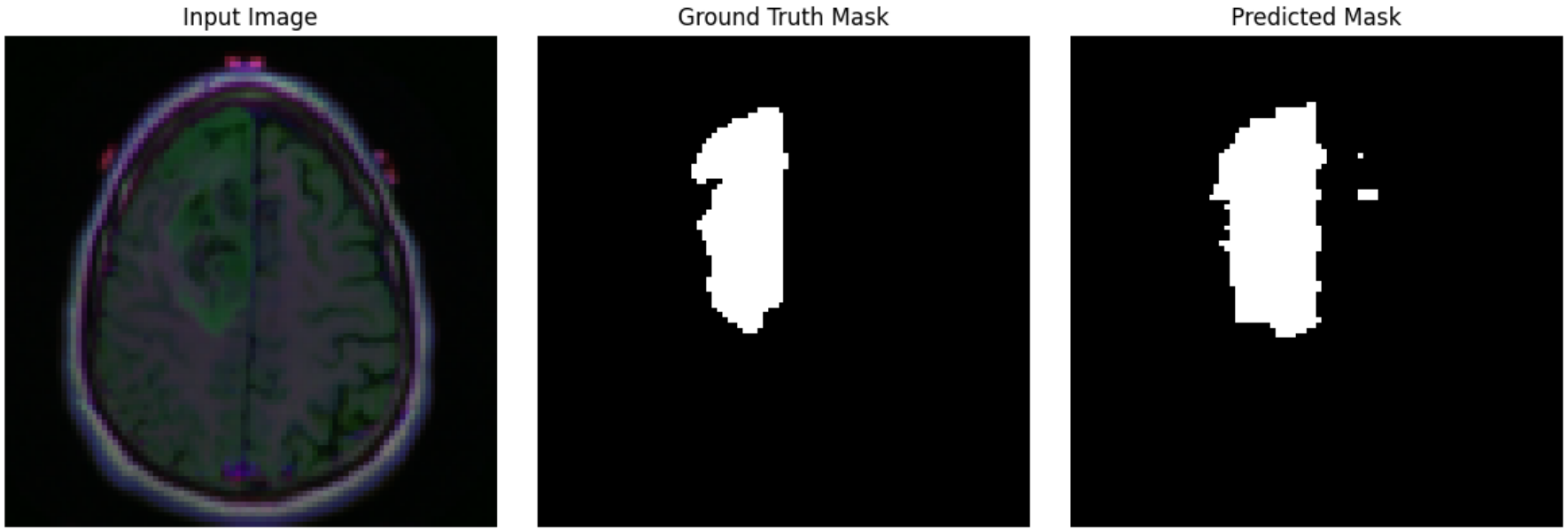
SETR with Dice Loss

Visualizing the training region evolution



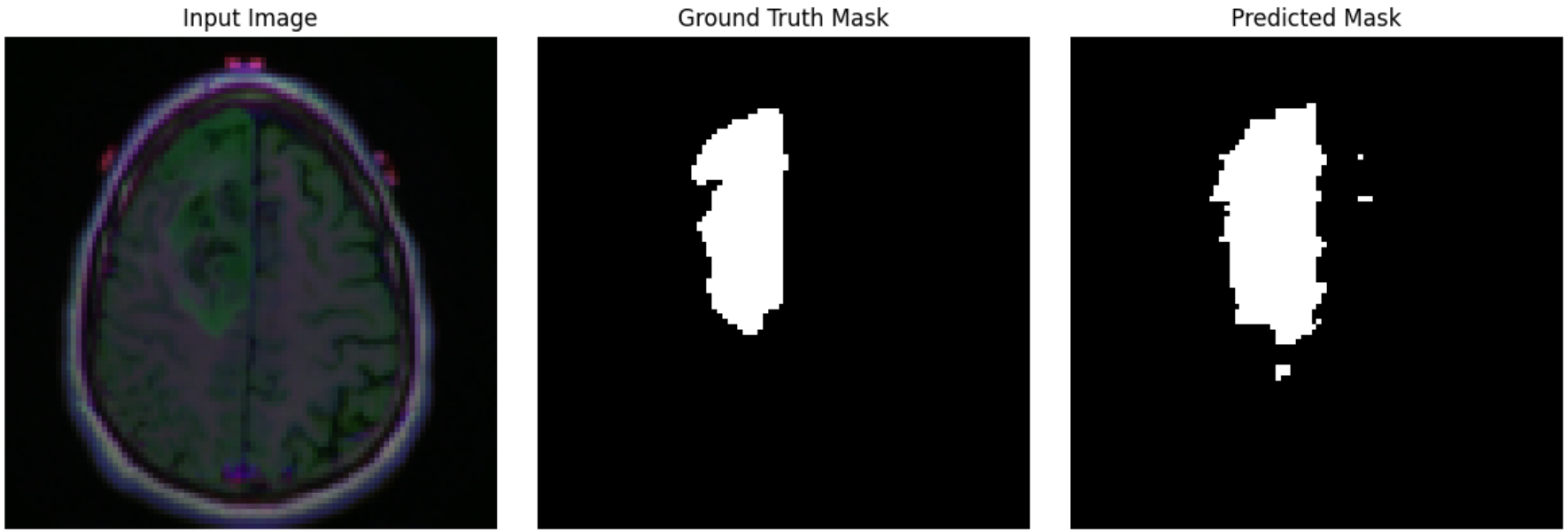
SETR with Dice Loss

Visualizing the training region evolution



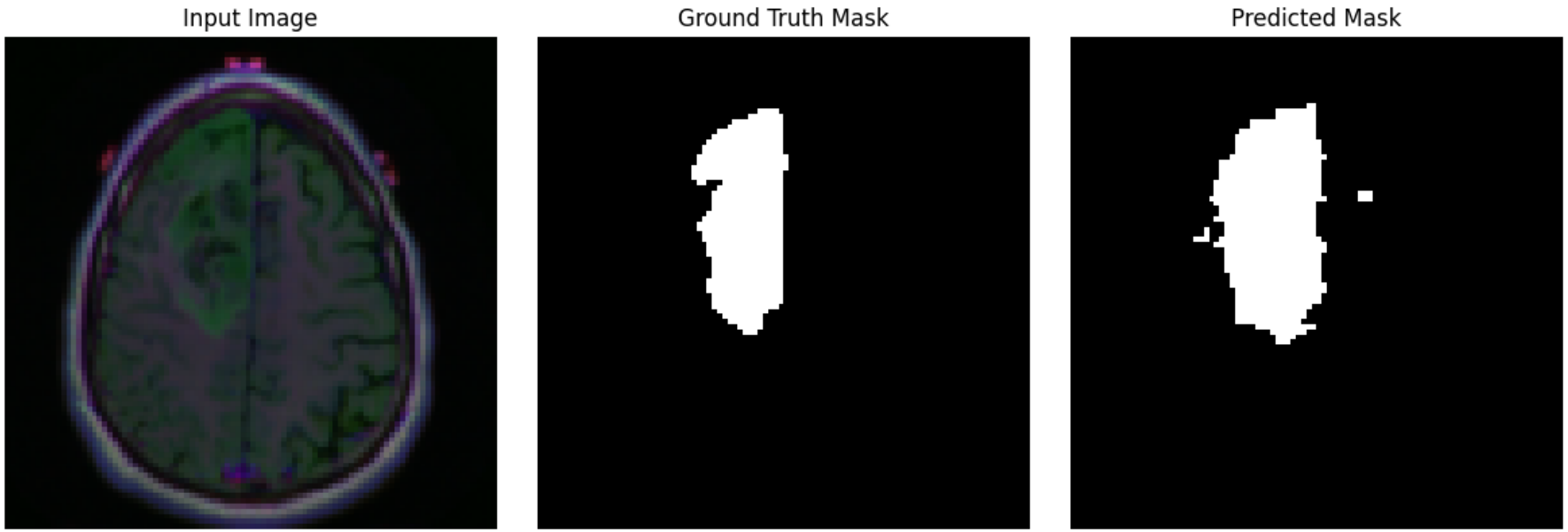
SETR with Dice Loss

Visualizing the training region evolution



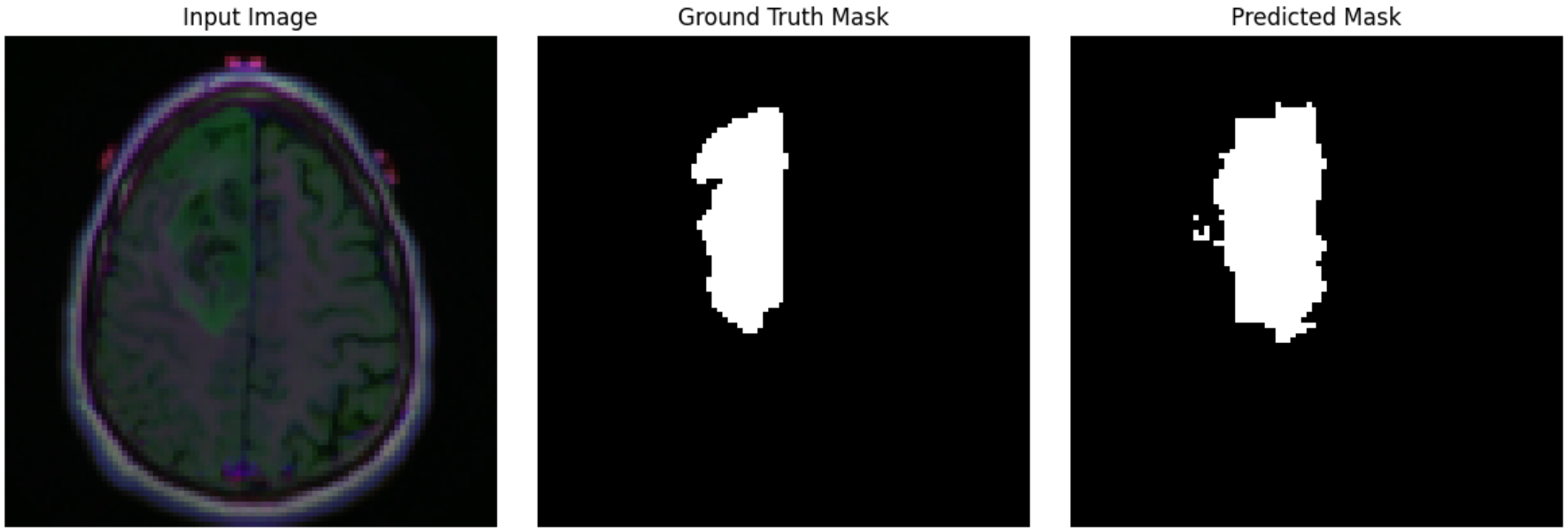
SETR with Dice Loss

Visualizing the training region evolution



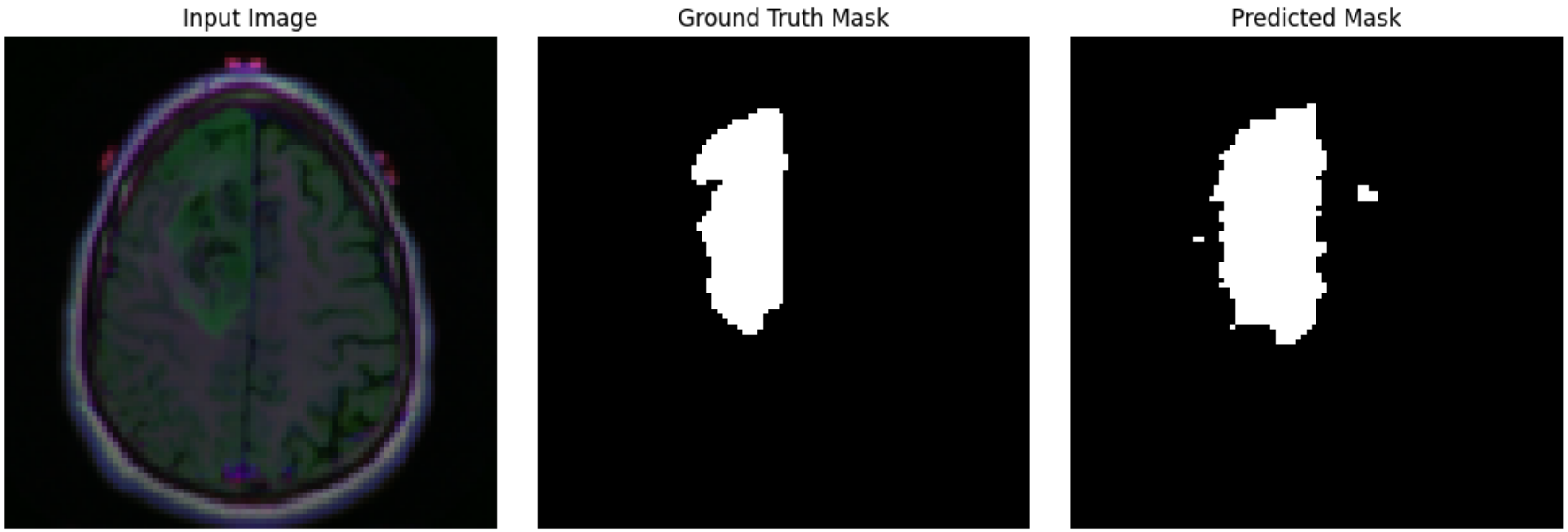
SETR with Dice Loss

Visualizing the training region evolution



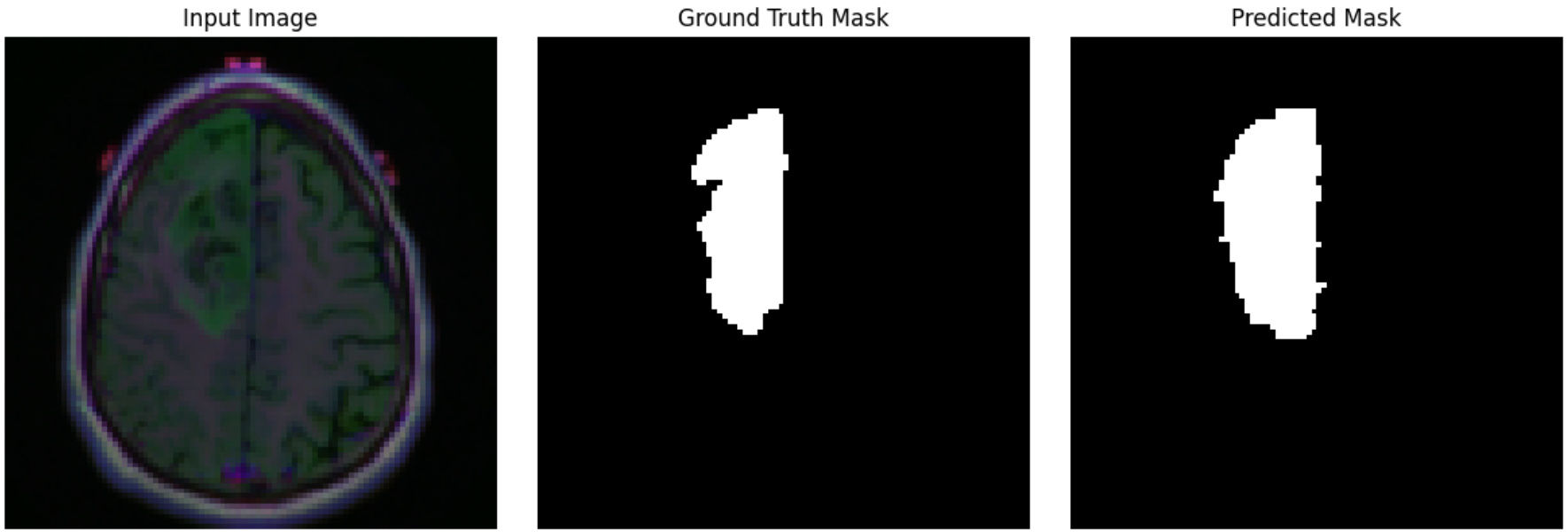
SETR with Dice Loss

Visualizing the training region evolution

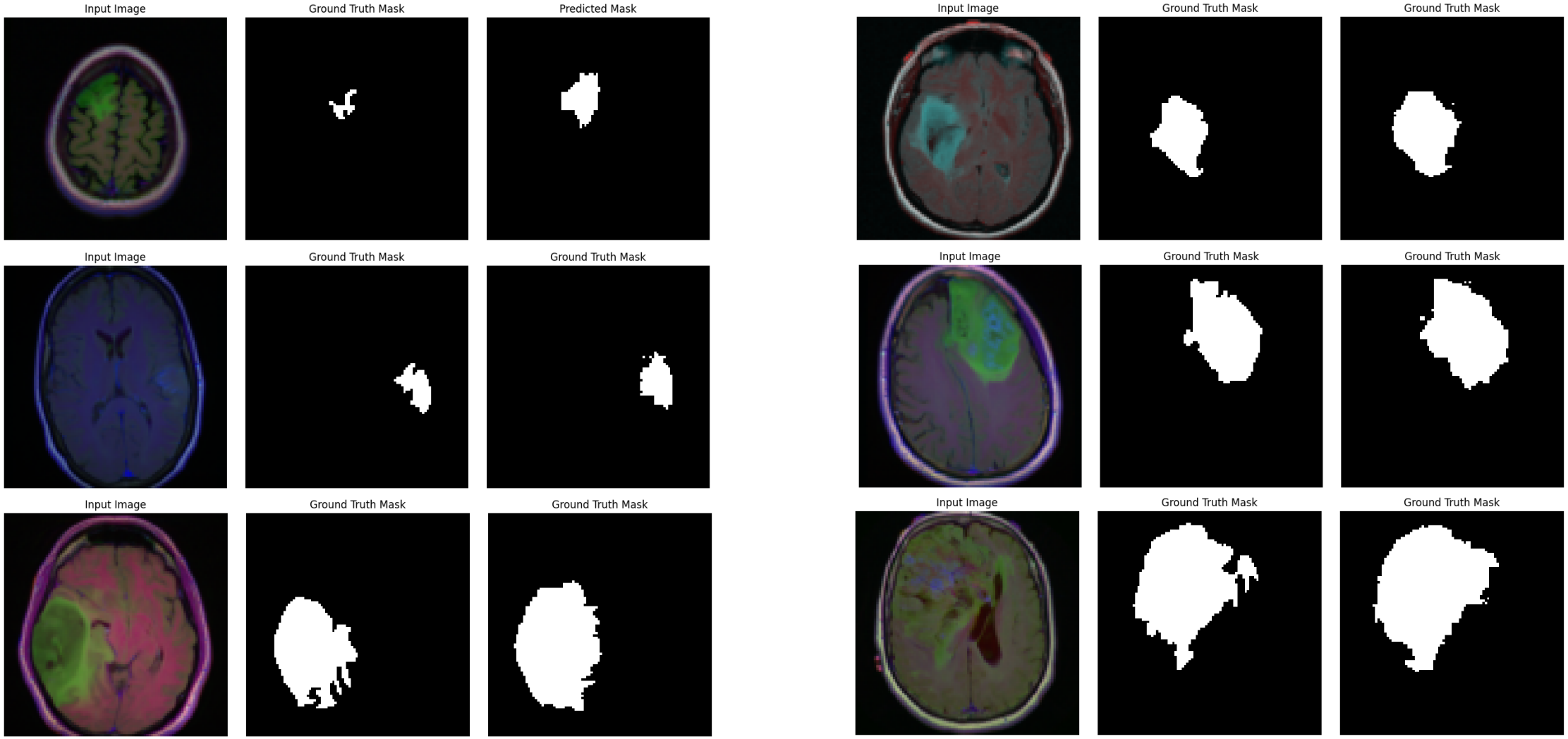


SETR with Dice Loss

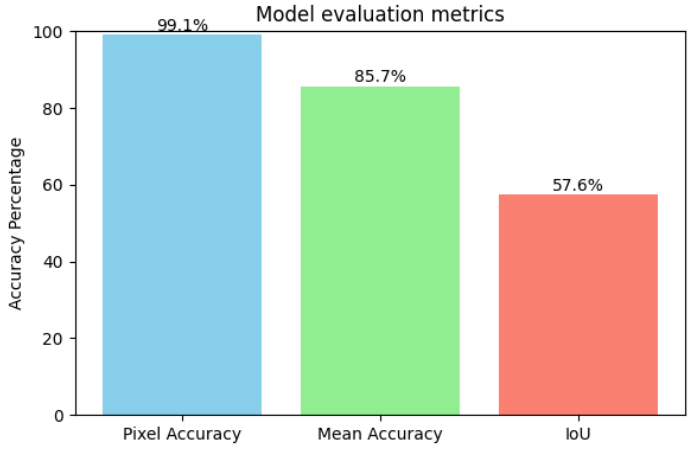
Visualizing the training region evolution



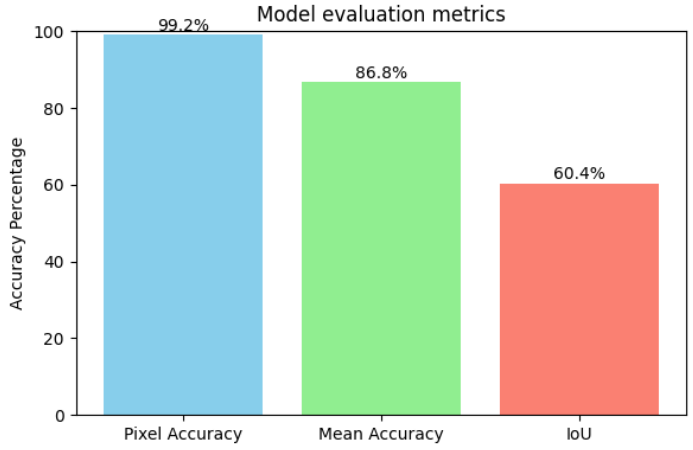
SETR with Dice Loss : What about other samples?



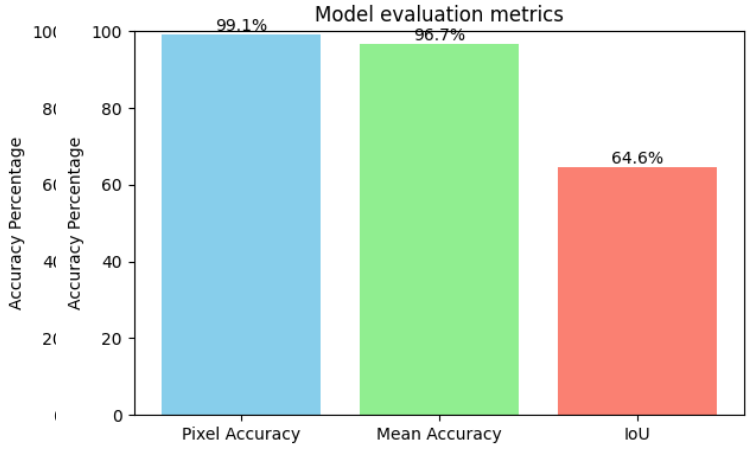
Final model comparison



Combo ratio 0.7
Approx 20 epochs learning



Combo ratio 0.5
Approx 40 epochs learning



SETR Dice
Approx 350 epochs learning

Conclusions

We saw different approaches for the segmentation task for FLAIR brain tumors, with a focus on loss choices and model precision evolution.

Better results could be achieved with higher image resolution and training epochs, together with higher computational power

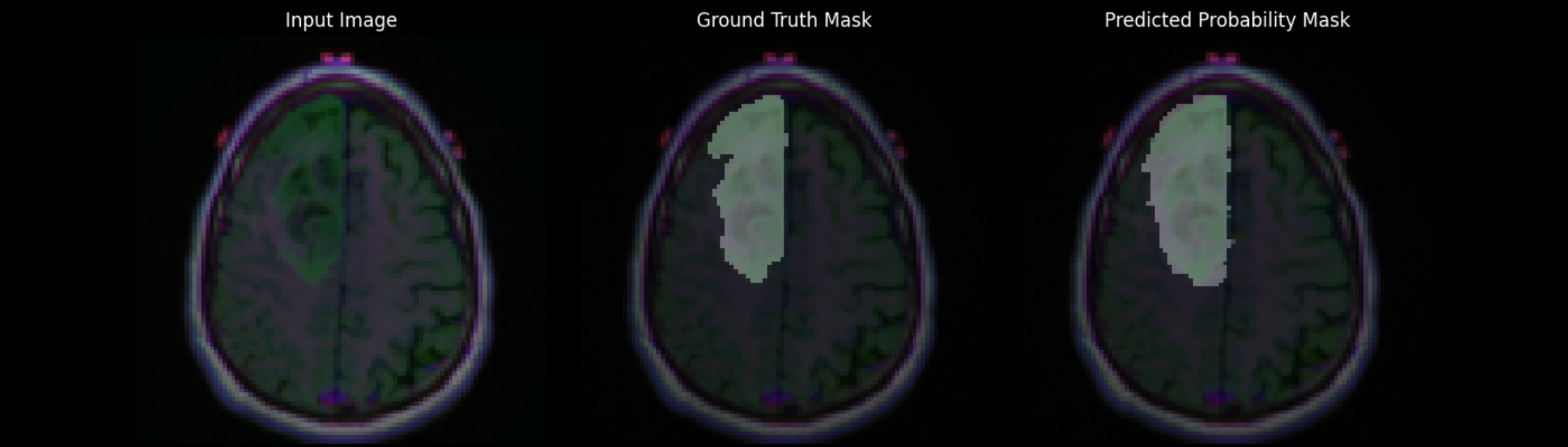
However, we can recover from the formal definition that each predicted mask is a **matrix of probabilities**: we can access the model certainty around the prediction…

$$with\ M = M_{NxNx1}([0,1]), \qquad m_{i,j} \sim p(M_{i,j} = tumoral)$$

…that we can visualize as an overlay.

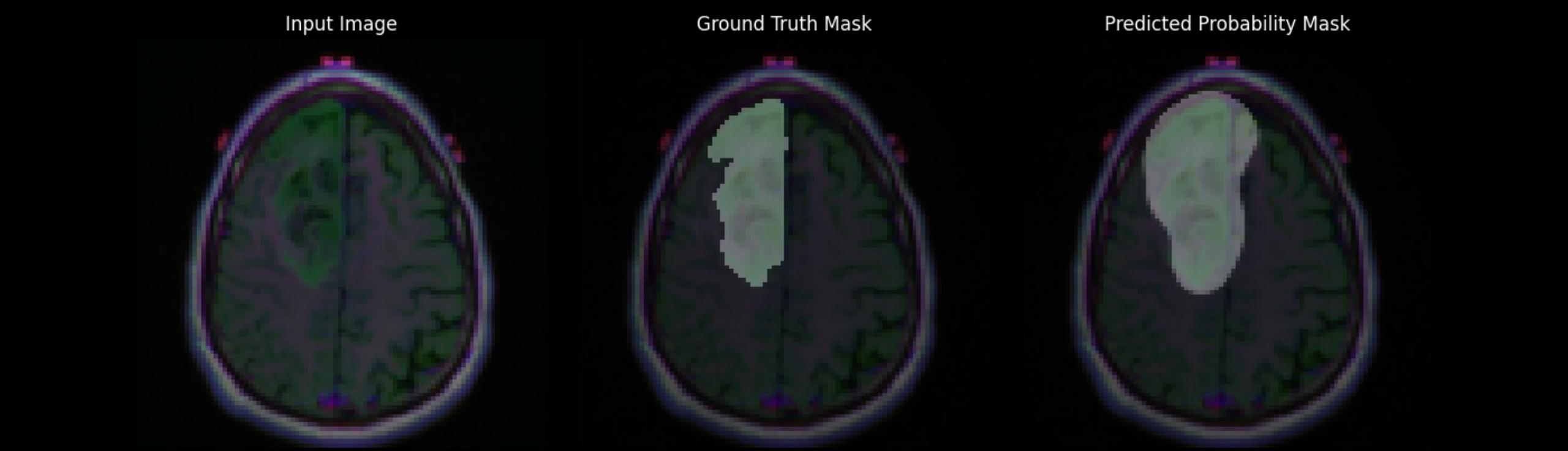
Probability Mask overlay

SETR model



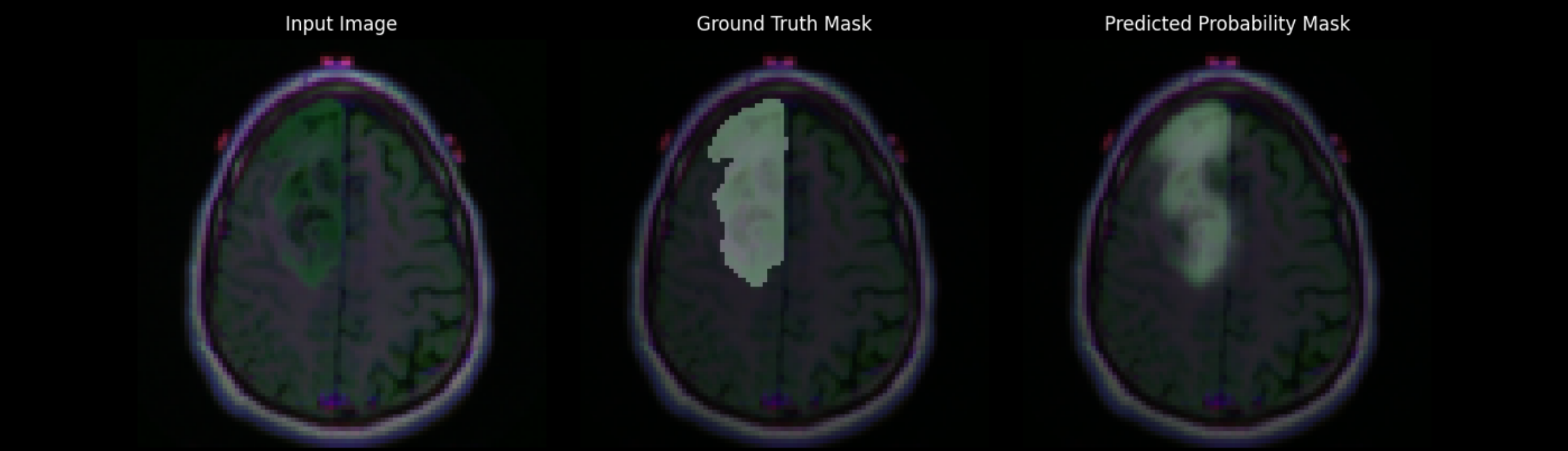
Probability Mask overlay

Dice U-Net model



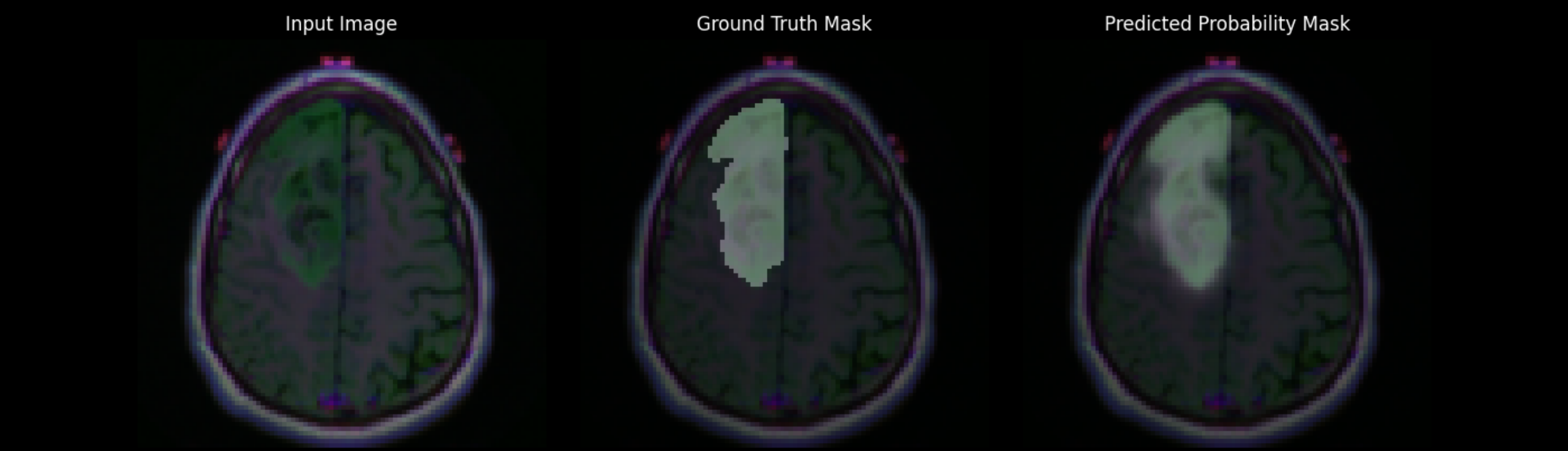
Probability Mask overlay

Combo 0.7 U-Net model



Probability Mask overlay

Combo 0.5 U-Net model



Project References

1. [1] EU cancer statistics: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cancer_statistics
2. [2] Brain tumor segmentation with Deep Neural Networks,Medical Image Analysis, Mohammad Havaei, Axel Davy, and others. https://doi.org/10.1016/j.media.2016.05.004.
3. [3] Multi-class glioma segmentation on real-world data with missing MRI sequences: comparison of three deep learning algorithms: https://www.nature.com/articles/s41598-023-44794-0
4. [4] Dataset: https://www.kaggle.com/datasets/mateuszbuda/lgg-mri-segmentation
5. [5] Azad, R., Heidary, M., Yilmaz, K., Hüttemann, M., Karimijafarbigloo, S., Wu, Y., Schmeink, A., & Merhof, D. (2023). Loss Functions in the Era of Semantic Segmentation: A Survey and Outlook. arXiv:2312.05391. https://arxiv.org/abs/2312.05391
6. [6] Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. arXiv preprint arXiv:1505.04597. https://arxiv.org/abs/1505.04597
7. [7] The U-net: A Complete Guide. https://medium.com/@alejandro.itoaramendia/decoding-the-u-net-a-complete-guide-810b1c6d56d8
8. [8] Loss Function library: https://www.kaggle.com/code/bigironsphere/loss-function-library-keras-pytorch#Combo-Loss
9. [9] Image Segmentation Using Vision Transformers (ViT): A Deep Dive with Cityscapes and CamVid Datasets https://medium.com/@ankitrajsh/image-segmentation-using-vision-transformers-vit-a-deep-dive-with-cityscapes-and-camvid-datasets-fc1ccdca295b
10. [10] Segmenter: Transformer for Semantic Segmentation, Robin Strudel and Ricardo Garcia and Ivan Laptev and Cordelia Schmid, https://arxiv.org/abs/2105.05633