这是一篇关于图像分割损失函数的总结，具体包括：

1. Binary Cross Entropy
2. Weighted Cross Entropy
3. Balanced Cross Entropy
4. Dice Loss
5. Focal loss
6. Tversky loss
7. Focal Tversky loss
8. log-cosh dice loss (本文提出的新损失函数)

论文地址：

https://arxiv.org/pdf/2006.14822.pdf

代码地址：

https://github.com/shruti-jadon/Semantic-Segmentation-Loss-Functions

项目推荐：

https://github.com/JunMa11/SegLoss

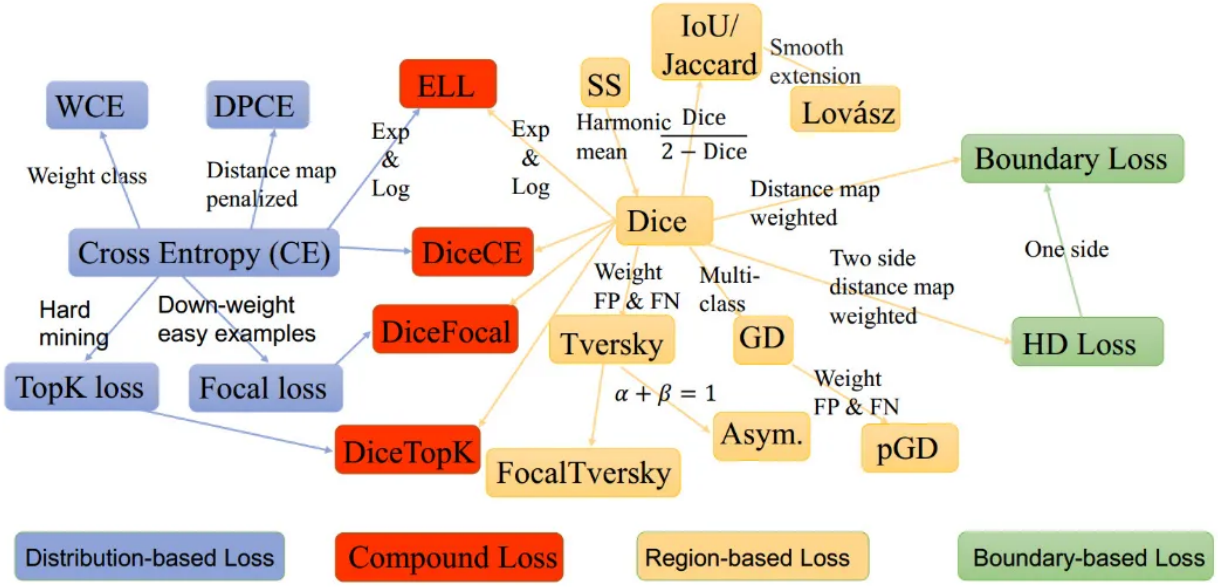
图像分割一直是一个活跃的研究领域，因为它有可能修复医疗领域的漏洞，并帮助大众。在过去的5年里，各种论文提出了不同的目标损失函数，用于不同的情况下，如偏差数据，稀疏分割等。在本文中，总结了大多数广泛用于图像分割的损失函数，并列出了它们可以帮助模型更快速、更好的收敛模型的情况。此外，本文还介绍了一种新的log-cosh dice损失函数，并将其在NBFS skull-stripping数据集上与广泛使用的损失函数进行了性能比较。某些损失函数在所有数据集上都表现良好，在未知分布数据集上可以作为一个很好的选择。

1 简介

深度学习彻底改变了从软件到制造业的各个行业。深度学习在医学界的应用也十分广泛，例如使用U-Net进行肿瘤分割、使用SegNet进行癌症检测等。在这些应用中，图像分割是

至关重要的，分割后的图像除了告诉我们存在某种疾病外，还展示了它到底存在于何处，这为实现自动检测CT扫描中的病变等功能提供基础保障。

图像分割可以定义为像素级别的分类任务。图像由各种像素组成，这些像素组合在一起定义了图像中的不同元素，因此将这些像素分类为一类元素的方法称为语义图像分割。在设计基于复杂图像分割的深度学习架构时，通常会遇到了一个至关重要的选择，即选择哪个损失/目标函数，因为它们会激发算法的学习过程。损失函数的选择对于任何架构学习正确的目标都是至关重要的，因此自2012年以来，各种研究人员开始设计针对特定领域的损失函数，以为其数据集获得更好的结果。



在本文中，总结了15种基于图像分割的损失函数。被证明可以在不同领域提供最新技术成果。这些损失函数可大致分为4类：基于分布的损失函数，基于区域的损失函数，基于边界的损失函数和基于复合的损失函数（ Distribution-based,Region-based，Boundary-based，and Compounded）。

TABLE I
TYPES OF SEMANTIC SEGMENTATION LOSS FUNCTIONS

Type	Loss Function
Distribution-based Loss	Binary Cross-Entropy Weighted Cross-Entropy Balanced Cross-Entropy Focal Loss Distance map derived loss penalty term
Region-based Loss	Dice Loss Sensitivity-Specificity Loss Tversky Loss Focal Tversky Loss **Log-Cosh Dice Loss**(ours)

Boundary-based Loss	Hausdorff Distance loss Shape aware loss
Compounded Loss	Combo Loss Exponential Logarithmic Loss

本文还讨论了确定哪种目标/损失函数在场景中可能有用的条件。除此之外，还提出了一种新的log-cosh dice损失函数用于图像语义分割。为了展示其效率，还比较了NBFS头骨剥离数据集上所有损失函数的性能。

2 Distribution-based loss

1. Binary Cross-Entropy
二进制交叉熵损失函数

交叉熵定义为对给定随机变量或事件集的两个概率分布之间的差异的度量。它被广泛用于分类任务，并且由于分割是像素级分类，因此效果很好。在多分类任务中，经常采用 soft max 激活函数+交叉熵损失函数，因为交叉熵描述了两个概率分布的差异，然而神经网络输出的是向量，并不是概率分布的形式。所以需要 softmax激活函数将一个向量进行“归一化”成概率分布的形式，再采用交叉熵损失函数计算 loss。

交叉熵损失函数的具体表达为：

$$L = -[y \log \hat{y} + (1-y) \log(1-\hat{y})] \tag{1}$$

其中，y_i 表示样本i的label，正类为1，负类为 0。 $P(y=1 \mid x) = \hat{y}$ 表示预测值。如果是计算 N个样本的总的损失函数，只要将 N 个Loss叠加起来就可以了:

$$L = -\sum_{i=1}^{N} y^{(i)} \log \hat{y}^{(i)} + \left(1-y^{(i)}\right) \log\left(1-\hat{y}^{(i)}\right)$$

交叉熵损失函数可以用在大多数语义分割场景中，但它有一个明显的缺点：当图像分割任务只需要分割前景和背景两种情况。当前景像素的数量远远小于背景像素的数量时，y=0的数量远大于y=1的数量，损失函数y=0的成分就会占据主导，使得模型严重偏向背景，导致效果不好。

#二值交叉熵，这里输入要经过sigmoid处理

```
import torch
import torch.nn as nn
```

```
import torch.nn as nn
import torch.nn.functional as F
nn.BCELoss\(F.sigmoid\(input\), target\)
```

#多分类交叉熵，用这个 loss 前面不需要加 `Softmax` 层 `nn.CrossEntropyLoss\(input, target\)`

2. Weighted Binary Cross-Entropy
加权交叉熵损失函数

$$\mathrm{WCE}(p,\hat{p}) = -(\beta p \log(\hat{p}) + (1-p)\log(1-\hat{p}))$$

加权交叉熵损失函数只是在交叉熵Loss的基础上为每一个类别添加了一个权重参数为正样本加权。设置 $\beta > 1$, 减少假阴性; 设置 $\beta < 1$, 减少假阳性。这样相比于原始的交叉熵Loss, 在样本数量不均衡的情况下可以获得更好的效果。

```
class WeightedCrossEntropyLoss\(torch.nn.CrossEntropyLoss\):
   """
   Network has to have NO NONLINEARITY\!
   """
   def \_\_init\_\_\(self, weight=None\):
       super\(WeightedCrossEntropyLoss, self\).\_\_init\_\_\(\)
       self.weight = weight

   def forward\(self, inp, target\):
       target = target.long\(\)
       num\_classes = inp.size\(\)\[1\]

       i0 = 1
       i1 = 2

       while i1 \< len\(inp.shape\): # this is ugly but torch only allows to transpo
           inp = inp.transpose\(i0, i1\)
           i0 += 1
           i1 += 1

       inp = inp.contiguous\(\)
       inp = inp.view\(-1, num\_classes\)

       target = target.view\(-1,\)
       wce\_loss = torch.nn.CrossEntropyLoss\(weight=self.weight\)

       return wce\_loss\(inp, target\)
```

3. Balanced Cross-Entropy
平衡交叉熵损失函数

$$\mathrm{BCE}(p,\hat{p}) = -(\beta p \log(\hat{p}) + (1-\beta)(1-p)\log(1-\hat{p})) \quad (2)$$

与加权交叉熵损失函数类似，但平衡交叉熵损失函数对负样本也进行加权。

4. Focal Loss

$$\mathrm{FL}(p,\hat{p}) = -(\alpha(1-\hat{p})^{\gamma} p \log(\hat{p}) + (1-\alpha)\hat{p}^{\gamma}(1-p)\log(1-\hat{p})) \quad (3$$

Focal loss是在目标检测领域提出来的。其目的是关注难例（也就是给难分类的样本较大的权重）。对于正样本，使预测概率大的样本（简单样本）得到的loss变小，而预测概率小的样本（难例）loss变得大，从而加强对难例的关注度。但引入了额外参数，增加了调参难度。

```
class FocalLoss\(nn.Module\):
    """
    copy from: https://github.com/Hsuxu/Loss\_ToolBox-PyTorch/blob/master/FocalLoss/F
    This is a implementation of Focal Loss with smooth label cross entropy supported
    'Focal Loss for Dense Object Detection. \(https://arxiv.org/abs/1708.02002\)'
        Focal\_Loss= \-1\*alpha\*\(1-pt\)\*log\(pt\)
    :param num\_class:
    :param alpha: \(tensor\) 3D or 4D the scalar factor for this criterion
    :param gamma: \(float,double\) gamma > 0 reduces the relative loss for well-class
                    focus on hard misclassified example
    :param smooth: \(float,double\) smooth value when cross entropy
    :param balance\_index: \(int\) balance class index, should be specific when alpha
    :param size\_average: \(bool, optional\) By default, the losses are averaged over
    """

    def \_\_init\_\_\(self, apply\_nonlin=None, alpha=None, gamma=2, balance\_index=0
        super\(FocalLoss, self\).\_\_init\_\_\(\)
        self.apply\_nonlin = apply\_nonlin
        self.alpha = alpha
        self.gamma = gamma
        self.balance\_index = balance\_index
        self.smooth = smooth
        self.size\_average = size\_average

        if self.smooth is not None:
            if self.smooth \< 0 or self.smooth > 1.0:
                raise ValueError\('smooth value should be in \[0,1\]'\)
```

```
def forward\(self, logit, target\):
    if self.apply\_nonlin is not None:
        logit = self.apply\_nonlin\(logit\)
    num\_class = logit.shape\[1\]

    if logit.dim\(\) > 2:
        # N,C,d1,d2 \-> N,C,m \(m=d1\*d2\*...\)
        logit = logit.view\(logit.size\(0\), logit.size\(1\), \-1\)
        logit = logit.permute\(0, 2, 1\).contiguous\(\)
        logit = logit.view\(-1, logit.size\(-1\)\)
    target = torch.squeeze\(target, 1\)
    target = target.view\(-1, 1\)
    # print\(logit.shape, target.shape\)
    #
    alpha = self.alpha

    if alpha is None:
        alpha = torch.ones\(num\_class, 1\)
    elif isinstance\(alpha, \(list, np.ndarray\)\):
        assert len\(alpha\) == num\_class
        alpha = torch.FloatTensor\(alpha\).view\(num\_class, 1\)
        alpha = alpha / alpha.sum\(\)
    elif isinstance\(alpha, float\):
        alpha = torch.ones\(num\_class, 1\)
        alpha = alpha \* \(1 \- self.alpha\)
        alpha\[self.balance\_index\] = self.alpha

    else:
        raise TypeError\('Not support alpha type'\)

    if alpha.device \!= logit.device:
        alpha = alpha.to\(logit.device\)

    idx = target.cpu\(\).long\(\)

    one\_hot\_key = torch.FloatTensor\(target.size\(0\), num\_class\).zero\_\(\)
    one\_hot\_key = one\_hot\_key.scatter\_\(1, idx, 1\)
    if one\_hot\_key.device \!= logit.device:
        one\_hot\_key = one\_hot\_key.to\(logit.device\)

    if self.smooth:
        one\_hot\_key = torch.clamp\(
            one\_hot\_key, self.smooth/\(num\_class-1\), 1.0 \- self.smooth\)
    pt = \(one\_hot\_key \* logit\).sum\(1\) + self.smooth
    logpt = pt.log\(\)

    gamma = self.gamma

    alpha = alpha\[idx\]
    alpha = torch.squeeze\(alpha\)
    loss = \-1 \* alpha \* torch.pow\(\(1 \- pt\), gamma\) \* logpt

    if self.size\_average:
        loss = loss.mean\(\)
```

```
        else:
            loss = loss.sum\(\)
        return loss
```

5. Distance map derived loss penalty term
距离图得出的损失惩罚项

可以将距离图定义为ground truth与预测图之间的距离（欧几里得距离、绝对距离等）。合并映射的方法有2种，一种是创建神经网络架构，在该算法中有一个用于分割的重建head，或者将其引入损失函数。遵循相同的理论，可以从GT mask得出的距离图，并创建了一个基于惩罚的自定义损失函数。使用这种方法，可以很容易地将网络引导到难以分割的边界区域。损失函数定义为：

$$L(y,p) = \frac{1}{N}\sum_{i=1}^{N}(1+\phi)(\odot)L_{CE}(y,p) \tag{4}$$

```
class DisPenalizedCE\(torch.nn.Module\):
   """
   Only for binary 3D segmentation
   Network has to have NO NONLINEARITY\!
   """

   def forward\(self, inp, target\):
       # print\(inp.shape, target.shape\) # \(batch, 2, xyz\), \(batch, 2, xyz\)
       # compute distance map of ground truth
       with torch.no\_grad\(\):
           dist = compute\_edts\_forPenalizedLoss\(target.cpu\(\).numpy\(\)>0.5\) +

       dist = torch.from\_numpy\(dist\)
       if dist.device \!= inp.device:
           dist = dist.to\(inp.device\).type\(torch.float32\)
       dist = dist.view\(-1,\)

       target = target.long\(\)
       num\_classes = inp.size\(\)\[1\]

       i0 = 1
       i1 = 2

       while i1 \< len\(inp.shape\): # this is ugly but torch only allows to transpo
           inp = inp.transpose\(i0, i1\)
           i0 += 1
           i1 += 1
```

```
        inp = inp.contiguous\(\)
        inp = inp.view\(-1, num\_classes\)
        log\_sm = torch.nn.LogSoftmax\(dim=1\)
        inp\_logs = log\_sm\(inp\)

        target = target.view\(-1,\)
        # loss = nll\_loss\(inp\_logs, target\)
        loss = \-inp\_logs\[range\(target.shape\[0\]\), target\]
        # print\(loss.type\(\), dist.type\(\)\)
        weighted\_loss = loss\*dist

        return loss.mean\(\)
```

3 Region-based loss

1. Dice Loss

Dice系数是计算机视觉界广泛使用的度量标准，用于计算两个图像之间的相似度。在2016年的时候，它也被改编为损失函数，称为Dice损失。

Dice系数：是用来度量集合相似度的度量函数，通常用于计算两个样本之间的像素之间的相似度，公式如下：

$$s = \frac{2|X \bigcap Y|}{|X| + |Y|} \text{ 或 } s = \frac{2TP}{2TP + FN + FP}$$

分子中之所以有一个系数2是因为分母中有重复计x和y的原因，取值范围是[0.1]。而针对分割任务来说x表示的就是Ground Truth分割图像，而y代表的就是预测的分割图像。

Dice Loss：

$$s = 1 - \frac{2|X \cap Y|}{|X| + |Y|}$$

此处，在分子和分母中添加1以确保函数在诸如 的极端情况下的确定性。Dice Loss使用与样本极度不均衡的情况，如果一般情况下使用Dice Loss会回反向传播有不利的影响，使得训练不稳定。

```
def get\_tp\_fp\_fn\(net\_output, gt, axes=None, mask=None, square=False\):
    """
    net\_output must be \(b, c, x, y\(, z\)\)\)
    gt must be a label map \(shape \(b, 1, x, y\(, z\)\) OR shape \(b, x, y\(, z\)\)\
    if mask is provided it must have shape \(b, 1, x, y\(, z\)\)\)
    :param net\_output:
    :param gt:
    :param axes:
    :param mask: mask must be 1 for valid pixels and 0 for invalid pixels
```

```
    :param square: if True then fp, tp and fn will be squared before summation
    :return:
    """
    if axes is None:
        axes = tuple\(range\(2, len\(net\_output.size\(\)\)\)\)

    shp\_x = net\_output.shape
    shp\_y = gt.shape

    with torch.no\_grad\(\):
        if len\(shp\_x\) \!= len\(shp\_y\):
            gt = gt.view\(\(shp\_y\[0\], 1, \*shp\_y\[1:\]\)\)

        if all\(\[i == j for i, j in zip\(net\_output.shape, gt.shape\)\]\):
            # if this is the case then gt is probably already a one hot encoding
            y\_onehot = gt
        else:
            gt = gt.long\(\)
            y\_onehot = torch.zeros\(shp\_x\)
            if net\_output.device.type == "cuda":
                y\_onehot = y\_onehot.cuda\(net\_output.device.index\)
            y\_onehot.scatter\_\(1, gt, 1\)

    tp = net\_output \* y\_onehot
    fp = net\_output \* \(1 \- y\_onehot\)
    fn = \(1 \- net\_output\) \* y\_onehot

    if mask is not None:
        tp = torch.stack\(tuple\(x\_i \* mask\[:, 0\] for x\_i in torch.unbind\(tp, d
        fp = torch.stack\(tuple\(x\_i \* mask\[:, 0\] for x\_i in torch.unbind\(fp, d
        fn = torch.stack\(tuple\(x\_i \* mask\[:, 0\] for x\_i in torch.unbind\(fn, d

    if square:
        tp = tp \*\* 2
        fp = fp \*\* 2
        fn = fn \*\* 2

    tp = sum\_tensor\(tp, axes, keepdim=False\)
    fp = sum\_tensor\(fp, axes, keepdim=False\)
    fn = sum\_tensor\(fn, axes, keepdim=False\)

    return tp, fp, fn


class SoftDiceLoss\(nn.Module\):
    def \_\_init\_\_\(self, apply\_nonlin=None, batch\_dice=False, do\_bg=True, smoot
                 square=False\):
        """
        paper: https://arxiv.org/pdf/1606.04797.pdf
        """
        super\(SoftDiceLoss, self\).\_\_init\_\_\(\)

        self.square = square
        self.do\_bg = do\_bg
        self.batch\_dice = batch\_dice
```

```
        self.apply\_nonlin = apply\_nonlin
        self.smooth = smooth

    def forward\(self, x, y, loss\_mask=None\):
        shp\_x = x.shape

        if self.batch\_dice:
            axes = \[0\] + list\(range\(2, len\(shp\_x\)\)\)
        else:
            axes = list\(range\(2, len\(shp\_x\)\)\)

        if self.apply\_nonlin is not None:
            x = self.apply\_nonlin\(x\)

        tp, fp, fn = get\_tp\_fp\_fn\(x, y, axes, loss\_mask, self.square\)

        dc = \(2 \* tp + self.smooth\) / \(2 \* tp + fp + fn + self.smooth\)

        if not self.do\_bg:
            if self.batch\_dice:
                dc = dc\[1:\]
            else:
                dc = dc\[:, 1:\]
        dc = dc.mean\(\)

        return \-dc
```

2. Tversky Loss

$$T(A,B) = \frac{|A \cap B|}{|A \bigcap B| + \alpha|A - B| + \beta|B - A|}$$

$$TL(p,\hat{p}) = 1 - \frac{1 + p\hat{p}}{1 + p\hat{p} + \beta(1-p)\hat{p} + (1-\beta)p(1-\hat{p})}$$

论文地址为：https://arxiv.org/pdf/1706.05721.pdf 。

Tversky系数是Dice系数和 Jaccard 系数的一种推广。当设置 $\alpha = \beta = 0.5$, 此时Tversky系数就是Dice系数。而当设置$\alpha = \beta = 1$时，此时Tversky系数就是Jaccard系数。a和β分别控制假阴性和假阳性。通过调整a和β, 可以控制假阳性和假阴性之间的平衡。

```
class TverskyLoss\(nn.Module\):
    def \_\_init\_\_\(self, apply\_nonlin=None, batch\_dice=False, do\_bg=True, smoot
                 square=False\):
        """
        paper: https://arxiv.org/pdf/1706.05721.pdf
        """
        super\(TverskyLoss, self\).\_\_init\_\_\(\)

        self.square = square
```

```
        self.do\_bg = do\_bg
        self.batch\_dice = batch\_dice
        self.apply\_nonlin = apply\_nonlin
        self.smooth = smooth
        self.alpha = 0.3
        self.beta = 0.7

    def forward\(self, x, y, loss\_mask=None\):
        shp\_x = x.shape

        if self.batch\_dice:
            axes = \[0\] + list\(range\(2, len\(shp\_x\)\)\)
        else:
            axes = list\(range\(2, len\(shp\_x\)\)\)

        if self.apply\_nonlin is not None:
            x = self.apply\_nonlin\(x\)

        tp, fp, fn = get\_tp\_fp\_fn\(x, y, axes, loss\_mask, self.square\)


        tversky = \(tp + self.smooth\) / \(tp + self.alpha\*fp + self.beta\*fn + self

        if not self.do\_bg:
            if self.batch\_dice:
                tversky = tversky\[1:\]
            else:
                tversky = tversky\[:, 1:\]
        tversky = tversky.mean\(\)

        return \-tversky
```

3. Focal Tversky Loss

与“Focal loss”相似，后者着重于通过降低易用/常见损失的权重来说明困难的例子。Focal Tversky Loss还尝试借助γ系数来学习诸如在ROI（感兴趣区域）较小的情况下的困难示例，如下所示：

$$FTL = \sum_{c} (1 - TI_c)^{\gamma}$$

```
class FocalTversky\_loss\(nn.Module\):
    """
    paper: https://arxiv.org/pdf/1810.07842.pdf
    author code: https://github.com/nabsabraham/focal-tversky-unet/blob/347d39117c245
    """
    def \_\_init\_\_\(self, tversky\_kwargs, gamma=0.75\):
        super\(FocalTversky\_loss, self\).\_\_init\_\_\(\)
        self.gamma = gamma
        self.tversky = TverskyLoss\(\*\*tversky\_kwargs\)
```

```
    def forward\(self, net\_output, target\):
        tversky\_loss = 1 + self.tversky\(net\_output, target\) # = 1-tversky\(net\_o
        focal\_tversky = torch.pow\(tversky\_loss, self.gamma\)
        return focal\_tversky
```

4. Sensitivity Specificity Loss

首先敏感性就是召回率，检测出确实有病的能力：

$$\text{Sensitivity} = \frac{TP}{TP+FN}$$

特异性, 检测出确实没病的能力:

$$\text{Specificity} = \frac{TN}{TN+FP}$$

而Sensitivity Specificity Loss为:

$$SS = \lambda \frac{\sum_{n=1}^{N}(r_n - p_n)^2 r_n}{\sum_{n=1}^{N} r_n + \epsilon} + (1-\lambda)\frac{\sum_{n=1}^{N}(r_n - p_n)^2(1-r_n)}{\sum_{n=1}^{N}(1-r_n) + \epsilon}$$

其中左边为病烃像素的错误率即，1-Sensitivity，而不是正确率，所以设置 λ 为 0.05 。其中 $(r_n - p_n)^2$ 是为了得到平滑的梯度。

```
class SSLoss\(nn.Module\):
    def \_\_init\_\_\(self, apply\_nonlin=None, batch\_dice=False, do\_bg=True, smoot
                 square=False\):
        """
        Sensitivity-Specifity loss
        paper: http://www.rogertam.ca/Brosch\_MICCAI\_2015.pdf
        tf code: https://github.com/NifTK/NiftyNet/blob/df0f86733357fdc92bbc191c8fec0
        """
        super\(SSLoss, self\).\_\_init\_\_\(\)

        self.square = square
        self.do\_bg = do\_bg
        self.batch\_dice = batch\_dice
        self.apply\_nonlin = apply\_nonlin
        self.smooth = smooth
        self.r = 0.1 # weight parameter in SS paper

    def forward\(self, net\_output, gt, loss\_mask=None\):
        shp\_x = net\_output.shape
        shp\_y = gt.shape
        # class\_num = shp\_x\[1\]

        with torch.no\_grad\(\):
            if len\(shp\_x\) \!= len\(shp\_y\):
```

```
            gt = gt.view\(\(shp\_y\[0\], 1, \*shp\_y\[1:\]\)\)

        if all\(\[i == j for i, j in zip\(net\_output.shape, gt.shape\)\]\):
            # if this is the case then gt is probably already a one hot encoding
            y\_onehot = gt
        else:
            gt = gt.long\(\)
            y\_onehot = torch.zeros\(shp\_x\)
            if net\_output.device.type == "cuda":
                y\_onehot = y\_onehot.cuda\(net\_output.device.index\)
            y\_onehot.scatter\_\(1, gt, 1\)

    if self.batch\_dice:
        axes = \[0\] + list\(range\(2, len\(shp\_x\)\)\)
    else:
        axes = list\(range\(2, len\(shp\_x\)\)\)

    if self.apply\_nonlin is not None:
        softmax\_output = self.apply\_nonlin\(net\_output\)

    # no object value
    bg\_onehot = 1 \- y\_onehot
    squared\_error = \(y\_onehot \- softmax\_output\)\*\*2
    specificity\_part = sum\_tensor\(squared\_error\*y\_onehot, axes\)/\(sum\_ten
    sensitivity\_part = sum\_tensor\(squared\_error\*bg\_onehot, axes\)/\(sum\_te

    ss = self.r \* specificity\_part + \(1-self.r\) \* sensitivity\_part

    if not self.do\_bg:
        if self.batch\_dice:
            ss = ss\[1:\]
        else:
            ss = ss\[:, 1:\]
    ss = ss.mean\(\)

    return ss
```

5. Log-Cosh Dice Loss
（本文提出的损失函数）

Dice系数是一种用于评估分割输出的度量标准。它也已修改为损失函数，因为它可以实现分割目标的数学表示。但是由于其非凸性，它多次都无法获得最佳结果。Lovsz-softmax损失旨在通过添加使用Lovsz扩展的平滑来解决非凸损失函数的问题。同时，Log-Cosh方法已广泛用于基于回归的问题中，以平滑曲线。

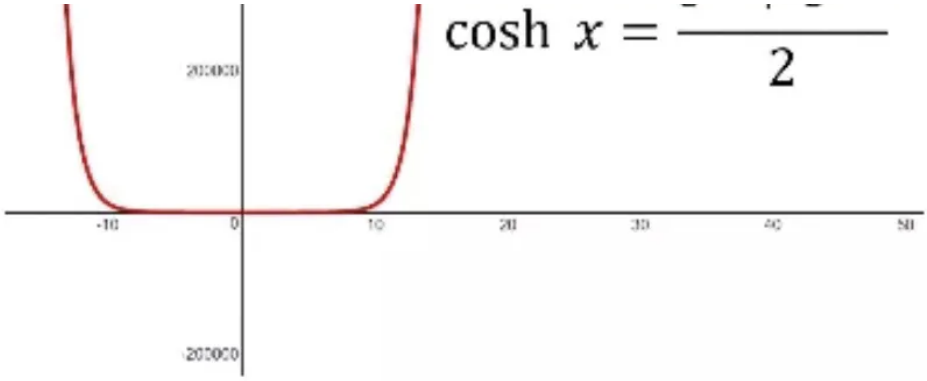


Fig. 3. Cosh(x) function is the average of e^x and e^-x

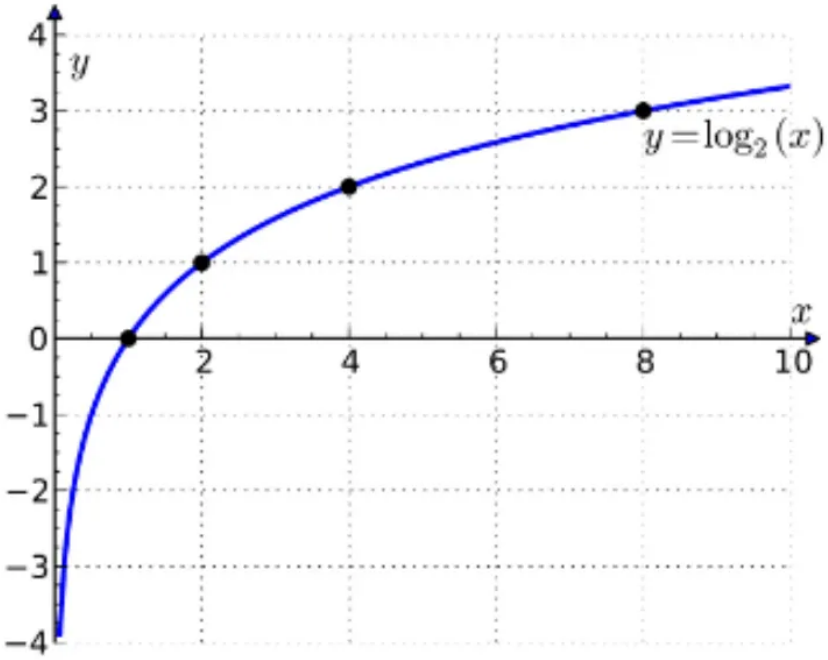


Fig. 4. Log(x) function with range from $(-\infty, C)$

将Cosh(x)函数和Log(x)函数合并，可以得到Log-Cosh Dice Loss：

$$L_{lc-dce} = \log(\cosh(\text{ DiceLoss }))$$

```
def log\_cosh\_dice\_loss\(self, y\_true, y\_pred\):
    x = self.dice\_loss\(y\_true, y\_pred\)
    return tf.math.log\(\(torch.exp\(x\) + torch.exp\(-x\)\) / 2.0\)
```

4 Boundary-based loss

1. Shape-aware Loss

顾名思义，Shape-aware Loss考虑了形状。通常，所有损失函数都在像素级起作用，Shape-aware Loss会计算平均点到曲线的欧几里得距离，即预测分割到ground truth的曲线

周围点之间的欧式距离，并将其用作交叉熵损失函数的系数，具体定义如下：（CE指交叉熵损失函数）

$$E_i = D\left(\hat{C}, C_{GT}\right)$$

$$L_{\text{shape}-\text{aware}} = -\sum_i CE(y,\hat{y}) - \sum iE_iCE(y,\hat{y})$$

```
class DistBinaryDiceLoss\(nn.Module\):
   """
   Distance map penalized Dice loss
   Motivated by: https://openreview.net/forum\?id=B1eIcvS45V
   Distance Map Loss Penalty Term for Semantic Segmentation
   """
   def \_\_init\_\_\(self, smooth=1e-5\):
       super\(DistBinaryDiceLoss, self\).\_\_init\_\_\(\)
       self.smooth = smooth

   def forward\(self, net\_output, gt\):
       """
       net\_output: \(batch\_size, 2, x,y,z\)
       target: ground truth, shape: \(batch\_size, 1, x,y,z\)
       """
       net\_output = softmax\_helper\(net\_output\)
       # one hot code for gt
       with torch.no\_grad\(\):
           if len\(net\_output.shape\) \!= len\(gt.shape\):
               gt = gt.view\(\(gt.shape\[0\], 1, \*gt.shape\[1:\]\)\)

           if all\(\[i == j for i, j in zip\(net\_output.shape, gt.shape\)\]\):
               # if this is the case then gt is probably already a one hot encoding
               y\_onehot = gt
           else:
               gt = gt.long\(\)
               y\_onehot = torch.zeros\(net\_output.shape\)
               if net\_output.device.type == "cuda":
                   y\_onehot = y\_onehot.cuda\(net\_output.device.index\)
               y\_onehot.scatter\_\(1, gt, 1\)

       gt\_temp = gt\[:,0, ...\].type\(torch.float32\)
       with torch.no\_grad\(\):
           dist = compute\_edts\_forPenalizedLoss\(gt\_temp.cpu\(\).numpy\(\)>0.5\)
       # print\('dist.shape: ', dist.shape\)
       dist = torch.from\_numpy\(dist\)

       if dist.device \!= net\_output.device:
           dist = dist.to\(net\_output.device\).type\(torch.float32\)

       tp = net\_output \* y\_onehot
       tp = torch.sum\(tp\[:,1,...\] \* dist, \(1,2,3\)\)

       dc = \(2 \* tp + self.smooth\) / \(torch.sum\(net\_output\[:,1,...\], \(1,2,3
```

```
        dc = dc.mean\(\)

        return \-dc
```

2. Hausdorff Distance Loss

Hausdorff Distance Loss（HD）是分割方法用来跟踪模型性能的度量。它定义为：

$$d(X,Y) = \max_{x \epsilon X} \min_{y \epsilon Y} ||x - y||_2$$

任何分割模型的目的都是为了最大化Hausdorff距离，但是由于其非凸性，因此并未广泛用作损失函数。有研究者提出了基于Hausdorff距离的损失函数的3个变量，它们都结合了度量用例，并确保损失函数易于处理。

```
class HDDTBinaryLoss\(nn.Module\):
   def \_\_init\_\_\(self\):
       """
       compute haudorff loss for binary segmentation
       https://arxiv.org/pdf/1904.10030v1.pdf
       """
       super\(HDDTBinaryLoss, self\).\_\_init\_\_\(\)


   def forward\(self, net\_output, target\):
       """
       net\_output: \(batch\_size, 2, x,y,z\)
       target: ground truth, shape: \(batch\_size, 1, x,y,z\)
       """
       net\_output = softmax\_helper\(net\_output\)
       pc = net\_output\[:, 1, ...\].type\(torch.float32\)
       gt = target\[:,0, ...\].type\(torch.float32\)
       with torch.no\_grad\(\):
           pc\_dist = compute\_edts\_forhdloss\(pc.cpu\(\).numpy\(\)>0.5\)
           gt\_dist = compute\_edts\_forhdloss\(gt.cpu\(\).numpy\(\)>0.5\)
       # print\('pc\_dist.shape: ', pc\_dist.shape\)

       pred\_error = \(gt \- pc\)\*\*2
       dist = pc\_dist\*\*2 + gt\_dist\*\*2 # \\alpha=2 in eq\(8\)

       dist = torch.from\_numpy\(dist\)
       if dist.device \!= pred\_error.device:
           dist = dist.to\(pred\_error.device\).type\(torch.float32\)

       multipled = torch.einsum\("bxyz,bxyz->bxyz", pred\_error, dist\)
       hd\_loss = multipled.mean\(\)

       return hd\_loss
```

5. Compounded loss

1.Exponential Logarithmic Loss

指数对数损失函数集中于使用骰子损失和交叉熵损失的组合公式来预测不那么精确的结构。对骰子损失和熵损失进行指数和对数转换，以合并更精细的分割边界和准确的数据分布的好处。它定义为：

$$
\begin{aligned}
L_{Exp} &= w_{\text{Dice}} L_{\text{Dice}} + w_{\text{cross}} L_{\text{cross}}, \text{ where} \\
L_{\text{Dice}} &= E\big(-\ln(DC)^{\gamma_{\text{Dice}}}\big) \\
L_{\text{cross}} &= E\big(w_l(-\ln(p_l))^{\gamma_{\text{cross}}}\big)
\end{aligned}
$$

2. Combo Loss

组合损失定义为 Dice loss和修正的交叉熵的加权和。它试图利用Dice损失解决类不平衡问题的灵活性，同时使用交叉熵进行曲线平滑。定义为：　　(DL指Dice Loss)

$$
\begin{aligned}
L_{m-bce} &= -\frac{1}{N}\sum_i \beta(y - \log(\hat{y})) + (1-\beta)(1-y)\log(1-\hat{y}) \\
CL(y,\hat{y}) &= \alpha L_{m-bce} - (1-\alpha) DL(y,\hat{y})
\end{aligned}
$$

6 实验与结果

数据集：NBFS Skull Stripping Dataset

实验细节：使用了简单的2D U-Net模型架构

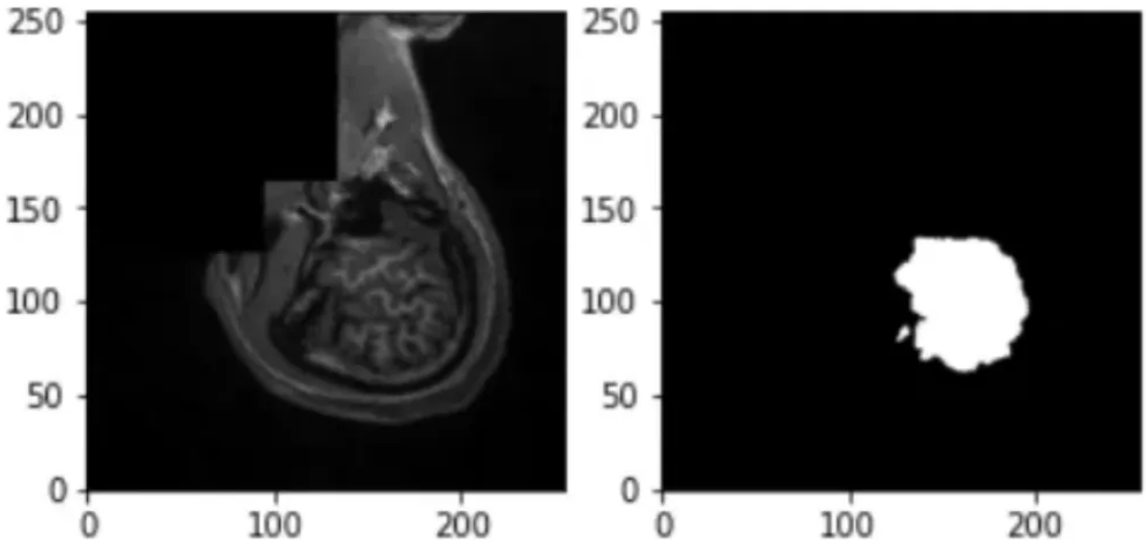


Fig. 5. Sample CT scan image from NBFS Skull Stripping Dataset

对比实验

TABLE II
COMPARISON OF SOME ABOVE MENTIONED LOSS FUNCTIONS ON BASIS OF DICE SCORES, SENSITIVITY AND SPECIFICITY FOR SKULL SEGMENTATION

Loss Functions	Evaluation Metrics		
	Dice Coefficient	*Sensitivity*	*Specificity*
Binary Cross-Entropy	0.968	0.976	0.998
Focal Loss	0.936	0.952	0.999
Dice Loss	0.970	0.981	0.998
Tversky Loss	0.965	0.979	0.996
Focal Tversky Loss	0.977	0.990	0.997
Log Cosh Dice Loss	0.975	0.975	0.997

参考文献

[1] https://blog.csdn.net/m0_37477175/article/details/83004746

[2] https://zhuanlan.zhihu.com/p/89194726