SRNet隐写分析网络模型 (pytorch实现)

原创

DRZ_2000 于 2021-07-11 17:40:29 发布 1157 收藏 9

分类专栏： 本科毕业设计总结 机器学习 文章标签： 隐写分析

本文链接：https://blog.csdn.net/DRZ_2000/article/details/118655021

文章目录

一 SRNet隐写分析模型介绍

二 SRNet网络概述

三 训练结果展示

一 SRNet隐写分析模型介绍

SRNet模型是宾汉姆顿大学(Binghamton University)Jessica教授团队于2018年提出的图像隐写分析网络模型，应该说是当时SOTA(state-of-the-art)的隐写分析网络模型了，实验证明不论对空域隐写算法还是JEPG频域隐写算法，SRNet都有较好的检测性能。关于SRNet论文、官方代码和中文翻译分别如下所示：

1. SRNet网络论文地址: https://ieeexplore.ieee.org/document/8470101
2. SRNet模型官方代码实现(tensorflow): http://dde.binghamton.edu/download/feature_extractors/
3. SRNet知乎中文翻译: https://zhuanlan.zhihu.com/p/362127299

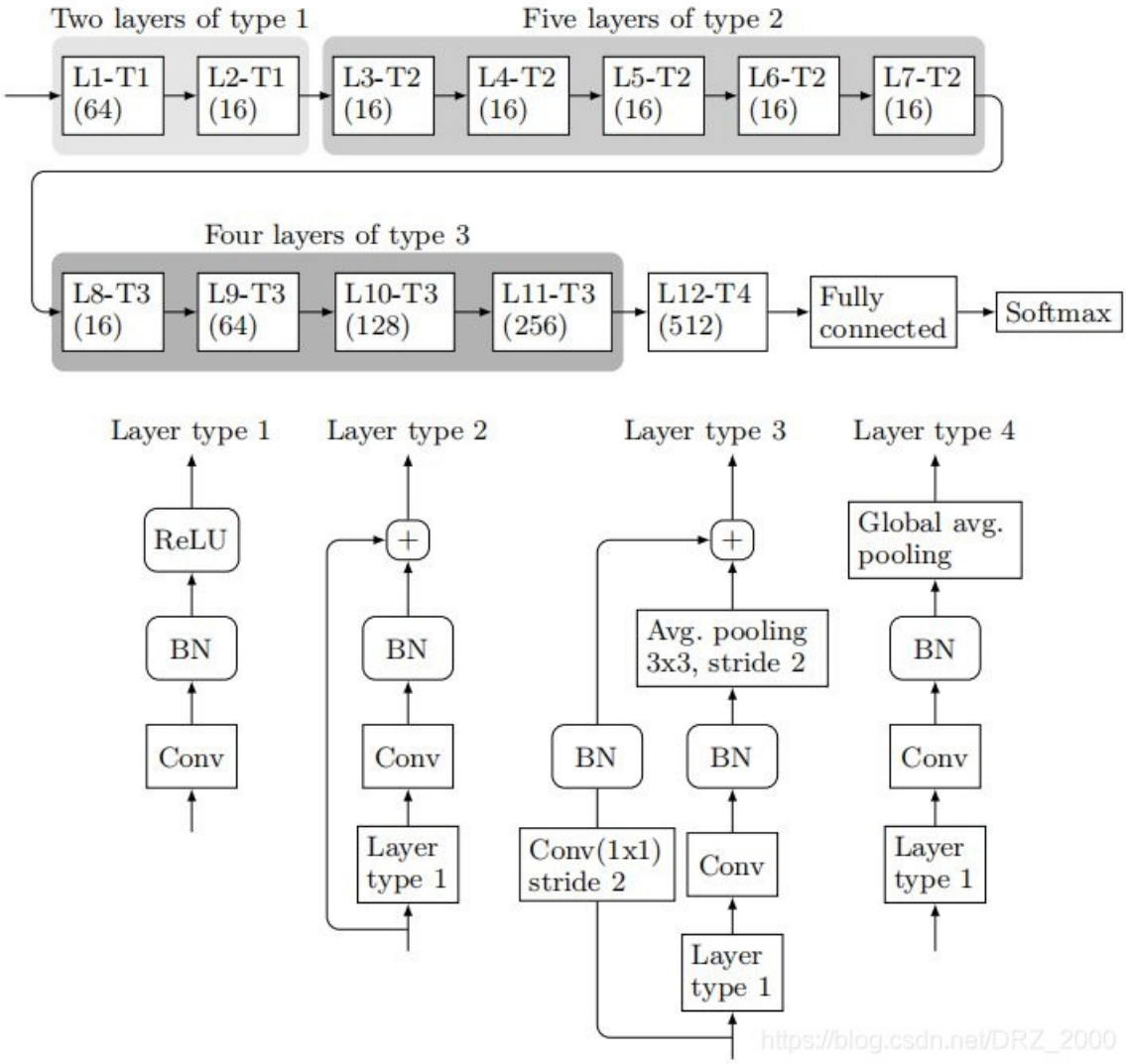
Jessica团队给出了tensorflow版代码，但是没有pytorch实现，小编斗胆pytorch复现了一版 (没办法，总不能厚脸皮直接用源码吧，好歹这是毕设啊)，嘿嘿，其实最终还是借鉴了源码，毕竟自己复现的效果比不上源码，话说小编的脸皮好像也不是很厚吧，像我这样脸皮薄的很少见了：）

1. 小编自己pytorch版SRNet模型复现: https://github.com/Uranium-Deng/Steganalysis-StegoRemoval

---

二 SRNet网络概述

SRNet网络结构图如下所示：



SRNet网络可以分成3段4类（从功能模块上分可以分成3段，从网路结构上可以分成4类），其中4类就是上图中下半部分的Layer Type 1 -Layer Type 4四类，3段对应功能模块，其中第一段是前两个阴影块也就是从第一层到第七层，该部分主要作用是提取之前隐写嵌入时引入的高频噪声；第二段是第三个阴影块，也就是第8层到第11层，该段主要作用是缩小特征图的维度；第三段是从第12层及其之后的所有网络层，主要扮演一个分类器的角色classifier。

关于SRNet网络模型在训练时一些超参数的设置，此处不会赘述。一开始自己按照论文中说明的超参数设置进行训练，得到的结果并不好，虽然在训练集上的accuracy一直在上升，但是在验证集上accuracy震荡的很厉害，得到的结果很不好，难道是自己用的电脑不行？鬼知道！于是小编只能慢慢摸索合适自己的网络参数。

论文中评判网络模型性能时采用的是错误率 Error Rate，计算公式为：$P(E)=\min\{P \quad +P \quad \}$，小编这里没有 pytorch版本SRNet

```
import torch
import torch.nn as nn


class Block1(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Block1, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels

        self.block = nn.Sequential(
            nn.Conv2d(in_channels=self.in_channels, out_channels=self.out_channels, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=self.out_channels),
            nn.ReLU(),
        )
```

```
    def forward(self, inputs):
        ans = self.block(inputs)
        # print('ans shape: ', ans.shape)
        return ans


class Block2(nn.Module):
    def __init__(self):
        super(Block2, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels=16, out_channels=16, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=16),
            nn.ReLU(),
            nn.Conv2d(in_channels=16, out_channels=16, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=16),
        )

    def forward(self, inputs):
        ans = torch.add(inputs, self.block(inputs))
        # print('ans shape: ', ans.shape)
        return inputs + ans


class Block3(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Block3, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels

        self.branch1 = nn.Sequential(
            nn.Conv2d(in_channels=self.in_channels, out_channels=self.out_channels, kernel_size=1, stride=2),
            nn.BatchNorm2d(num_features=self.out_channels),
        )
        self.branch2 = nn.Sequential(
            nn.Conv2d(in_channels=self.in_channels, out_channels=self.out_channels, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=self.out_channels),
            nn.ReLU(),
            nn.Conv2d(in_channels=self.out_channels, out_channels=self.out_channels, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=self.out_channels),
            nn.AvgPool2d(kernel_size=3, stride=2, padding=1),
        )

    def forward(self, inputs):
        ans = torch.add(self.branch1(inputs), self.branch2(inputs))
        # print('ans shape: ', ans.shape)
        return ans


class Block4(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Block4, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels

        self.block = nn.Sequential(
            nn.Conv2d(in_channels=self.in_channels, out_channels=self.out_channels, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=self.out_channels),
            nn.ReLU(),
            nn.Conv2d(in_channels=self.out_channels, out_channels=self.out_channels, kernel_size=3, stride=1, padding=1)
```

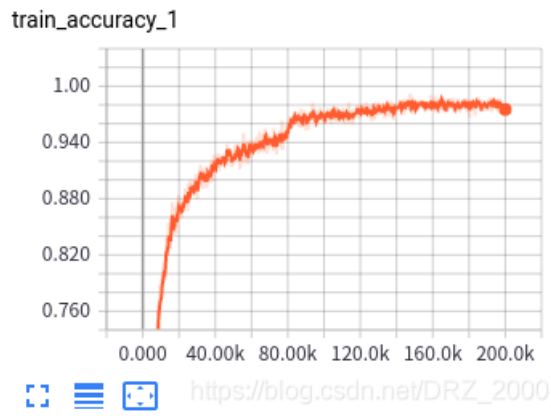
```
            nn.Conv2d(in_channels=self.out_channels, out_channels=self.out_channels, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=self.out_channels),
        )

    def forward(self, inputs):
        temp = self.block(inputs)
        ans = torch.mean(temp, dim=(2, 3))
        # print('ans shape: ', ans.shape)
        return ans


class SRNet(nn.Module):
    def __init__(self, data_format='NCHW', init_weights=True):
        super(SRNet, self).__init__()
        self.inputs = None
        self.outputs = None
        self.data_format = data_format

        # 第一种结构类型
        self.layer1 = Block1(1, 64)
        self.layer2 = Block1(64, 16)

        # 第二种结构类型
        self.layer3 = Block2()
        self.layer4 = Block2()
        self.layer5 = Block2()
        self.layer6 = Block2()
        self.layer7 = Block2()

        # 第三种类型
        self.layer8 = Block3(16, 16)
        self.layer9 = Block3(16, 64)
        self.layer10 = Block3(64, 128)
        self.layer11 = Block3(128, 256)

        # 第四种类型
        self.layer12 = Block4(256, 512)

        # 最后一层，全连接层
        self.layer13 = nn.Linear(512, 2)

        if init_weights:
            self._init_weights()

    def forward(self, inputs):
        inputs = inputs.permute(0, 3, 1, 2)  # NHWC -> NCHW
        self.inputs = inputs.float()
        # print('self.input.shape: ', self.inputs.shape)

        # 第一种结构类型
        x = self.layer1(self.inputs)
        x = self.layer2(x)

        # 第二种结构类型
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.layer5(x)
        x = self.layer6(x)
        x = self.layer7(x)
```

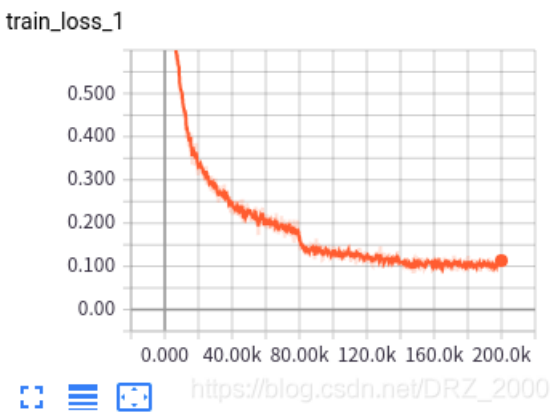
```
        # 第三种类型
        x = self.layer8(x)
        x = self.layer9(x)
        x = self.layer10(x)
        x = self.layer11(x)

        # 第四种类型
        x = self.layer12(x)

        # 最后一层全连接
        self.outputs = self.layer13(x)
        # print('self.outputs.shape: ', self.outputs.shape)
        return self.outputs

    def _init_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0.2)
            if isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            if isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0.001)


# 测试网络结构是否正确
x = torch.rand(size=(3, 256, 256, 1))
print(x.shape)

net = SRNet(data_format='NCHW', init_weights=True)
print(net)

output_Y = net(x)
print('output shape: ', output_Y.shape)
print('output: ', output_Y)
```

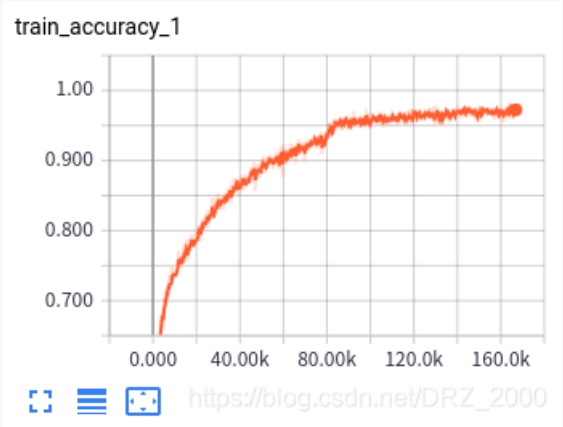
torch复现的版本中没有采用L2正则，所以这可能是效果不好的原因之一吧。

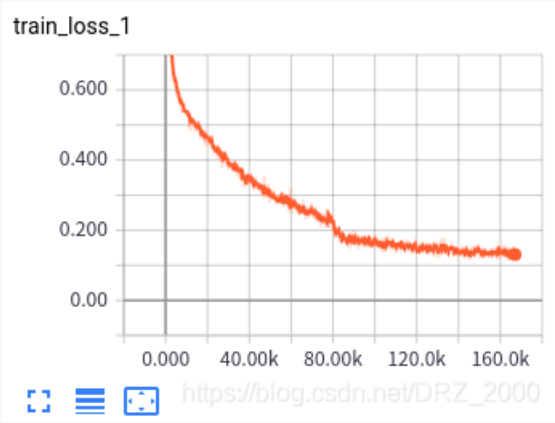
三 训练结果展示

原论文中使用了很多隐写术，其中空域和频域的隐写算法都有，在嵌入率方面选用的是0.1bpp、0.2bpp、至0.5bpp共5种不同的嵌入率。本文仅使用了S-UINWARD、WOW、HUGO三种空域隐写术和0.4bpp、0.7bpp、1.0bpp三种嵌入率（嵌入率太低，隐写分析难度较大，模型训练得到的结果太不好了，面子上很不好看，所以就适当的提高了嵌入率：）**部分**训练的结果如下图所示：









3个隐写术3个嵌入率，共9种，每一个都需要单独训练（直接训练嵌入率最小的模型，得到的结果很烂，只能先1.0bpp再0.7bpp最后0.4bpp这样逐渐递进）

为了提升SRNet网络隐写分析的性能，小编尝试在原始SRNet网络的基础上添加CBAM注意力机制，让模型更好的进行特征提取。**关于CBAM注意力机制的介绍以及如何将CBAM注意力模块添加到原始SRNet网络中，小编将在下一篇中进行介绍。**

路漫漫其修远兮，吾将上下而求索，本章内容对您有帮助的话，就请您不吝啬点赞吧，谢谢啦。