浙江大学

本科实验报告

Learning CNN

课程名称：计算机视觉

姓　　名：秦雨扬

学　　院：竺可桢学院

专　　业：自动化（控制）

学　　号：3210104591

电　　话：13588789194

邮　　件：qinyuyang2003@zju.edu.cn

指导老师：潘纲

2023 年 12 月 28 日

Contents

List of Figures

浙江大学实验报告

专业：自动化（控制）
姓名：秦雨扬
学号：3210104591
日期：2023 年 12 月 28 日
地点：玉泉曹光彪西-103

课程名称：计算机视觉　指导老师：潘纲　助教：周健均
实验名称：Learning CNN　实验类型：综合　成绩：

I 功能简述及运行说明

1. 功能简述

在本实验中，主要实现了以下几个功能

- LeNet-5 的训练，应用于 MNIST 数据集上的手写数字识别任务（图像分类）
- U-Net 的网络补全与测试，应用于 Carvana 数据集上的掩码 (Mask) 预测任务（语义分割）

2. 运行说明

- LeNet-5 的模型和训练程序为 *train.py*，需要建立同级 mnist 文件夹用于存放数据集，若无数据会自动下载，训练时每个 epoch 会分别记录 train 和 validation 的 accuarcy 和 loss, 将会打印并存在 runs 文件夹下，可使用 tensorboard logdir=runs 可视化查看。
- U-Net 的网络在 *unet.py* 中，测试程序 *unet_test.py*, 其中会载入预训练模型并对 *infer.jpg* 做推理并展示结果。可通过model 传入预训练模型地址。

II 开发与运行环境

实验使用 python 语言，测试系统为 *Ubuntu 20.04.6 LTS*, 测试环境为 *Anaconda 23.7.3* 下的 *python 3.8.18*，其它使用的包分别为

1) tensorboard 2.14.0
2) torch 2.0.1+cu118
3) torchvision 0.15.2+cu118
4) matplotlib 3.7.4

III 算法原理与具体实现

1. LeNet-5 训练

1.1 网络结构

LeNet-5 的网络构架如下图所示，在本实验中，在每一层后面添加了 ReLU(也尝试了 ELU 但是效果没什么区别，Sigmoid 效果差不多但是速度太慢了，最后就还是只用 ReLU)。此外，最后一层的 Gaussian connections 在论文 [1] 中的公式如下

$$y_i = \sum_j (x_j - \omega_{i,j})^2$$

由于 pytorch 没有对应的 RBF unit，本实验中仍然采用 nn.Linear 代替。而最后模型的输出类别，是 OUTPUT 中最大的值最大的那个，即

$$\theta = argmax_{\theta} OUTPUT[\theta]$$

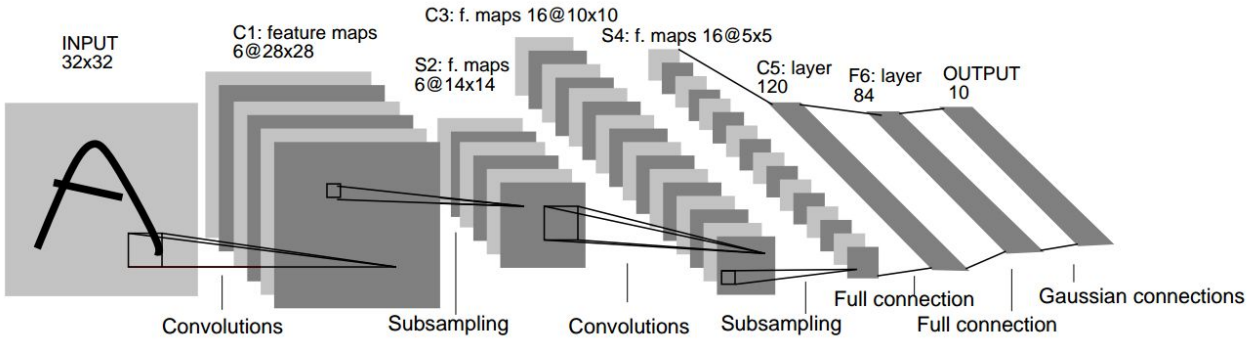


Figure 1: LeNet-5

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__() # 利用参数初始化父类
        self.backbone= nn.Sequential(
            nn.Conv2d(1, 6, 5),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(6, 16, 5),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2)
        )
        self.classifier=nn.Sequential(
            nn.Linear(16 * 4 * 4, 120),
            nn.ReLU(inplace=True),
            nn.Linear(120, 84),
            nn.ReLU(inplace=True),
            nn.Linear(84, 10)
        )

    def forward(self, x):
        x=self.backbone(x)
        x=x.view(x.size(0), -1)
        x=self.classifier(x)
        return x
```

1.2 数据集加载

使用 torchvision.datasets.MNIST 下载与载入数据集，但由于其为 PIL 图片，故需要进行 ToTensor() 变换，接着使用 torch.utils.data.DataLoader 对数据集进行分 batch。

```
# MNIST
train_dataset=torchvision.datasets.MNIST("mnist", train=False,
 ↪  transform=torchvision.transforms.ToTensor(), target_transform=None, download=True)
test_dataset=torchvision.datasets.MNIST("mnist", train=False,
 ↪  transform=torchvision.transforms.ToTensor(), target_transform=None, download=True)
print(len(train_dataset), len(test_dataset))
train_loader=torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader=torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=True)
```

1.3 模型训练与评估

在训练前，将 model.train 开启，同时将模型与数据均转移到 device 上，先清空 optimizer，再计算 loss 并 backward，再使用 optimizer.step() 更新梯度。

在评估方面，使用 torch.max 得到对应的类，与 label 进行比较后得到正确的个数。对整个 epoch 的 loss 和 correct 进行累计，最终得到 loss 与 accuracy。

在 validation 阶段，与 train 阶段基本相同，除了需要计算和反向传播梯度，故采用 model.eval 与 with torch.no_grad 进行保障。

optimizer 能够保持当前参数状态并基于计算得到的梯度进行参数更新。本实验中尝试了 SGD 与 Adam，同时采用 StepLR 的 scheduler 对模型学习率进行了调整。其参数具体影响见实验结果分析部分1.。

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
model = LeNet().to(device)
model=model.to(device)
```

```
# optimizer
optimizer=torch.optim.Adam(model.parameters(), lr=0.01)
# optimizer=torch.optim.SGD(model.parameters(), lr=0.01,momentum=0.8)
scheduler=torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.5) #gamma=0.5
# optimizer = torch.optim.SGD(model.parameters(), lr=1e-8, momentum=0.9)

# loss
criterion=nn.CrossEntropyLoss()
for epoch in range(200):
    #train
    model.train()
    correct=0
    train_loss=0
    total=0
    for i, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
```

```
        data=data.to(device)
        target=target.to(device)
        output=model(data)
        loss=criterion(output, target)
        train_loss+=loss.item()
        _, pred= torch.max(output.data, 1)
        total+=target.size(0)
        correct+=(pred==target).sum().item()
        loss.backward()
        optimizer.step()
    writer.add_scalar("Loss/train", train_loss, epoch)
    writer.add_scalar("Acc/train", correct/total, epoch)
    print("Train Epoch",epoch," | total loss=",train_loss,"acc=", correct/total*100,"%")
    # test
    scheduler.step()
    test_loss=0
    correct=0
    total=0
    model.eval()
    with torch.no_grad():
        for data, target in test_loader:
            data=data.to(device)
            target=target.to(device)
            output=model(data)
            test_loss+=criterion(output, target).item()
            total += target.size(0)
            _, pred= torch.max(output.data, 1)
            correct+=(pred==target).sum().item()
    writer.add_scalar("Loss/validation", test_loss, epoch)
    writer.add_scalar("Acc/validation",  correct/total, epoch)
    print("Eval Epoch",epoch,"| total loss=",test_loss,"acc=", correct/total*100,"%")

writer.flush()
writer.close()
```

2. U-Net 补全与测试

2.1 U-Net 原理

在 U-Net 原论文 [2] 中：

1) 左侧向下的结构被称为 Contracting Path，由通道数不断增加的卷积层和池化层组成
2) 右侧向上的结构被称为 Expanding Path，由通道数不断减少的卷积层和上采样层（反卷积层）组成
3) 在 Expanding Path 中，每次上采样层都会将 Contracting Path 中对应的特征图与自身的特征图进行拼接，这样可以保证 Expanding Path 中的每一层都能够利用 Contracting Path 中的信息

个人理解的话，这个 U-Net 通过降采样获得的全局的特征，通过拼接将局部信息与全局信息融合起来，所以在图像分割上能够取得不错的效果。（后面很多网络的 Backbone 都开始采用这样有点点金字塔的形状）

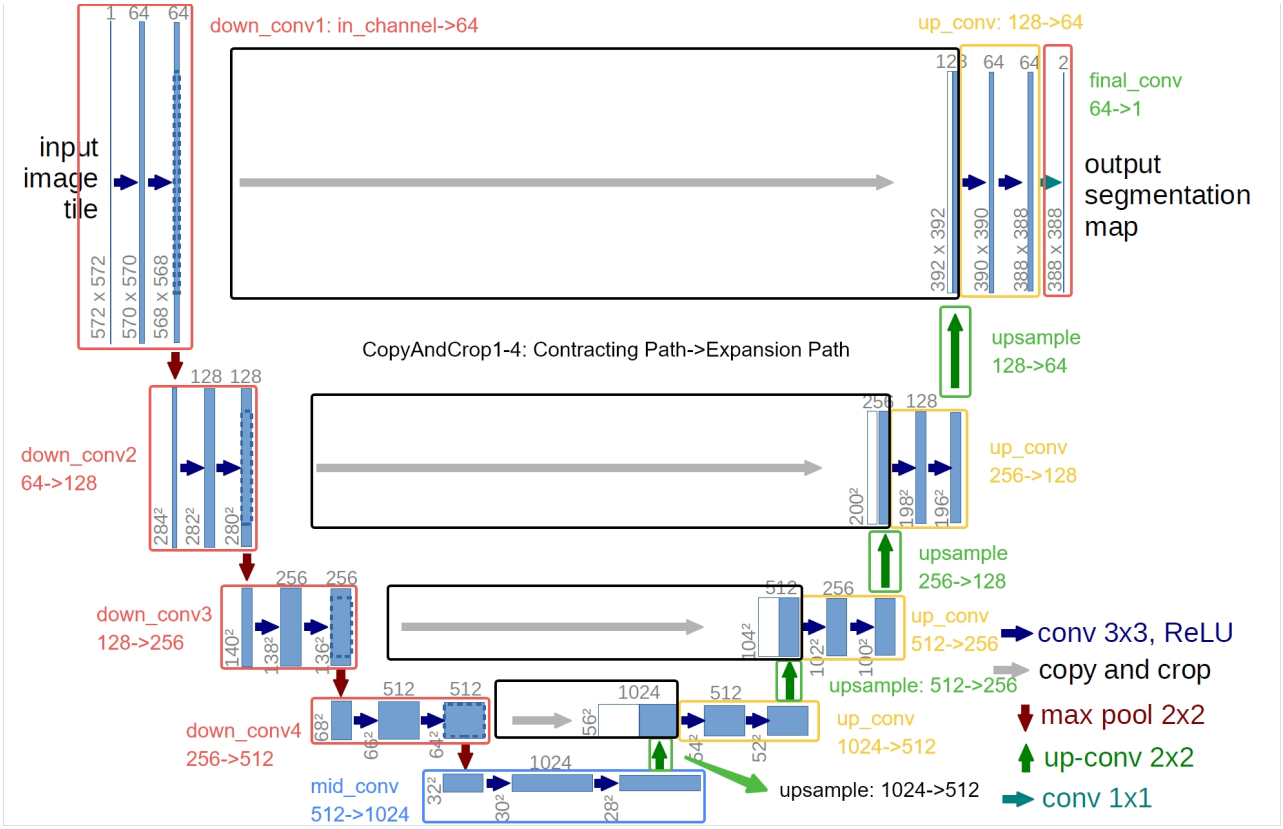


Figure 2: U-Net

2.2 *U-Net* 实现

根据图2所示，即为 U-Net 的结构图。本实验中，通过生成一个随机的张量进行输入，不断查看在 forward 中的 shape, 并对照图中给出的大小，就可以顺利地搭建起整个网络。在 CropAndConcat 部分，图中也有虚框示意，即适应右侧上采样的矩阵大小即可。

根据题目中给出的 adaptation 在进行调整，即

1) 通道数的变化已经在前面的图中进行了标注。和原论文不同，训练时 final_conv 的输出通道改成了 1，但是在通用的网络结构中就是 out_channels

2) down_conv, mid_conv 和 up_conv 都是由两个卷积层组成，每个卷积层都是 3×3 的卷积核，padding 为 1，stride 为 1。每个卷积层后都有一个 ReLU 激活函数，整体顺序为 Conv2d-Relu-Conv2d-Relu

3) final_conv 是一个 1×1 的卷积层，padding 为 0，stride 为 1，没有激活函数

虽然看起来顺利，但是在最终测试中出现了一些奇怪的情况，如图3所示。

最后发现，在图中 concat 的时候是左边是“短接”的特征图，右边是上采样的特征图，而在助教的实际代码中，是左边是上采样的特征图，右边是“短接”的特征图，最终得到正确的推理

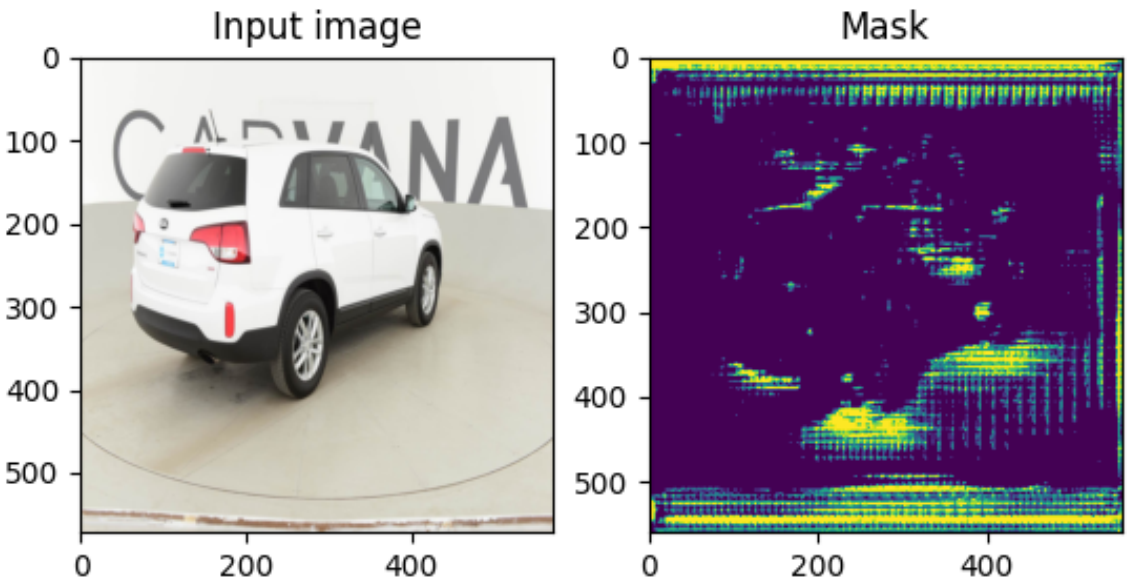


Figure 3: U-Net Fail

```python
class CropAndConcat(nn.Module):
    """
    ### Crop and Concatenate the feature map

    Crop the feature map from the contracting path to the size of the current feature map
    """

    def forward(self, x: torch.Tensor, contracting_x: torch.Tensor):
        """
        :param x: current feature map in the expansive path
        :param contracting_x: corresponding feature map from the contracting path
        """

        b, c, h, w = x.shape

        # TODO: Concatenate the feature maps
        # use torchvision.transforms.functional.center_crop(...)
        x = torch.cat(
            (x,center_crop(contracting_x, (h,w))),1
        )

        return x


class UNet(nn.Module):
    """
    ## U-Net
    """

```

```
    def __init__(self, in_channels: int, out_channels: int):
        """
        :param in_channels: number of channels in the input image
        :param out_channels: number of channels in the result feature map
        """
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels

        # TODO: Double convolution layers for the contracting path.
        # Number of features gets doubled at each step starting from 64.
        num = 64
        self.down_conv1 = nn.Sequential(
            nn.Conv2d(in_channels, num, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(num, num, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True)
        )

        self.down_conv2 = nn.Sequential(
            nn.Conv2d(num, num*2, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(num*2, num*2, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True)
        )

        self.down_conv3 = nn.Sequential(
            nn.Conv2d(num*2, num*4, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(num*4, num*4, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True)
        )

        self.down_conv4 = nn.Sequential(
            nn.Conv2d(num*4, num*8, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(num*8, num*8, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True)
        )

        # Down sampling layers for the contracting path
        self.down_sample1 = nn.MaxPool2d(2)
        self.down_sample2 = nn.MaxPool2d(2)
        self.down_sample3 = nn.MaxPool2d(2)
        self.down_sample4 = nn.MaxPool2d(2)

        # TODO: The two convolution layers at the lowest resolution (the bottom of the U).
        self.middle_conv = nn.Sequential(
            nn.Conv2d(num*8, num*16, kernel_size=3,padding=1,stride=1),
```

```
            nn.ReLU(inplace=True),
            nn.Conv2d(num*16, num*16, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True)
        )

        # Up sampling layers for the expansive path.
        # The number of features is halved with up-sampling.
        num = 1024
        self.up_sample1 = nn.ConvTranspose2d(num, num // 2, kernel_size=2, stride=2)
        self.up_sample2 = nn.ConvTranspose2d(num // 2, num // 4, kernel_size=2, stride=2)
        self.up_sample3 = nn.ConvTranspose2d(num // 4, num // 8, kernel_size=2, stride=2)
        self.up_sample4 = nn.ConvTranspose2d(num // 8, num // 16, kernel_size=2, stride=2)

        # TODO: Double convolution layers for the expansive path.
        # Their input is the concatenation of the current feature map and the feature map from the
        ↪ contracting path.
        # Therefore, the number of input features is double the number of features from up-sampling.
        self.up_conv1 = nn.Sequential(
            nn.Conv2d(num, num // 2, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(num // 2, num // 2, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True)
        )
        num=num//2
        self.up_conv2 = nn.Sequential(
            nn.Conv2d(num, num // 2, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(num // 2, num // 2, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True)
        )
        num=num//2

        self.up_conv3 = nn.Sequential(
            nn.Conv2d(num, num // 2, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(num // 2, num // 2, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True)
        )
        num=num//2

        self.up_conv4 = nn.Sequential(
            nn.Conv2d(num, num // 2, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(num // 2, num // 2, kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True)
        )
        num=num//2

        # Crop and concatenate layers for the expansive path.
```

```
        # TODO: Implement class CropAndConcat starting from line 6
        self.concat1 = CropAndConcat()
        self.concat2 = CropAndConcat()
        self.concat3 = CropAndConcat()
        self.concat4 = CropAndConcat()

        # TODO: Final 1*1 convolution layer to produce the output
        self.final_conv = nn.Conv2d(num, out_channels, kernel_size=1,padding=0,stride=1)

    def forward(self, x: torch.Tensor):
        """
        :param x: input image
        """
        # TODO: Contracting path
        # Remember to pass middle result to the expansive path
        x1 = self.down_conv1(x)
        # print(x1.shape)
        x2=self.down_sample1(x1)

        x2 = self.down_conv2(x2)
        x3=self.down_sample2(x2)

        x3 = self.down_conv3(x3)
        x4=self.down_sample3(x3)

        x4 = self.down_conv4(x4)
        x5=self.down_sample4(x4)

        x5=self.middle_conv(x5)

        x5=self.up_sample1(x5)

        x=self.concat1(x5,x4)
        x=self.up_conv1(x)
        x=self.up_sample2(x)

        x=self.concat2(x,x3)
        x=self.up_conv2(x)
        x=self.up_sample3(x)

        x=self.concat3(x,x2)
        x=self.up_conv3(x)
        x=self.up_sample4(x)

        x=self.concat4(x,x1)
        x=self.up_conv4(x)
        x=self.final_conv(x)
        # print(x.shape)
        return x
```

2.3 导入模型与推理

而在推理过程中，有如下过程

1) 读取图像并进行 resize 和 ToTensor
2) 为 img 增加 batch 维度
3) 将 img 转移到 device 上并进行推理
4) 转移到 cpu 上并去除 batch 维度和 channel 维度，得到二维图像
5) 进行 sigmoid 处理并通过是否大于 theshold 判断转化为二值的 mask
6) 转化为 numpy 并展示

```
21 if __name__ == "__main__":
22     parser = argparse.ArgumentParser(description='Predict masks from input images')
23     parser.add_argument('--model', '-m', default='model.pth',
24                         help='Specify the file in which the model is stored')
25     args = parser.parse_args()
26     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
27 
28     print(f'Loading model {args.model}')
29     print(f'Using device {device}')
30 
31     model = UNet(in_channels=3, out_channels=1).to(device)
32     # x=torch.rand((1,3,572,572)).to(device)
33     # y=model(x)
34     state_dict = torch.load(args.model, map_location=device)
35     model.load_state_dict(state_dict)
36 
37     print('Model loaded')
38 
39     img=Image.open("infer.jpg")
40     img=torchvision.transforms.Resize((572,
       ↪ 572),interpolation=torchvision.transforms.InterpolationMode.BILINEAR)(img)
41     img0=copy.deepcopy(img)
42     img=torchvision.transforms.ToTensor()(img)
43     # img=torch.tensor([img])
44     img.unsqueeze_(0)
45     img=img.to(device)
46     mask=model(img)
47     mask=mask.to("cpu")
48     mask=mask.squeeze(0)
49     mask = mask.squeeze(0)
50     mask=torch.nn.Sigmoid()(mask)
51     theshold=0.1
52     mask=(mask>theshold).numpy()
53     print(mask)
54     plot_img_and_mask(img0, mask)
```

IV 实验结果与分析

1. LeNet-5 训练

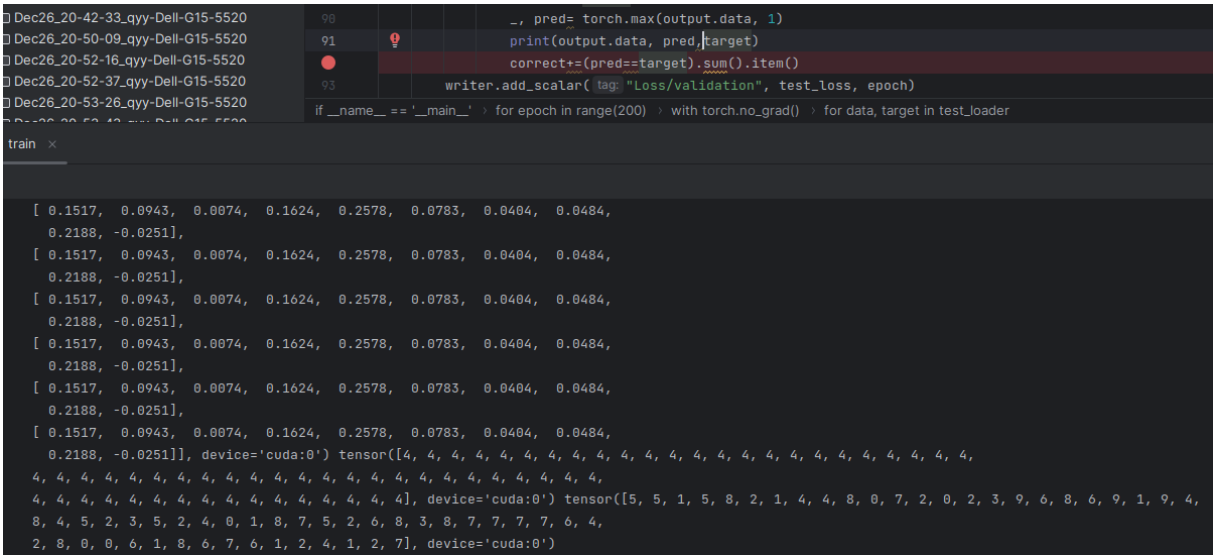
LeNet-5 训练如图4所示，可以看到绝大多数的情况训练集的精度能达到 99.5%，在验证集上的精度能达到 98.5% 精度。此前因为一不小心用验证集训练，train 和 val 精度都达到了 1，幸好意识到反常发现了问题。

本实验主要研究了 lr,scheduler 以及 optimizer 对训练的影响。在 MNIST 数据集上差别总体不大，除了 lr=0.001 时的曲线可以看到精度上升明显缓慢而且可能陷入了局部最小值。而 schedule 似乎没什么效果，除了抑制 Adam 在 lr=0.01 时 loss 的发散速度。

采用 Adam，若采用 0.01 的 lr，acc 无法收敛到，loss 侧也看到了明显的发现，在 epoch 较大的时候反而有时候验证集会精度下降到 96.87%，相对较不理想。

如图5而且当 lr 为 0.1/0.05 时，Adam 对应的 Acc 只有 11% 左右，且观察输出会发现其每个 batch 输出的最大值类是同一个，进而观察到不同图片其 output 也是同一个，推测可能是因为 Adam 在 lr 较大的时候出现了一些 bug??? 不太懂，不过 SGD 实测没有这个 bug。

Figure 4: TensorBoard Result



```
Dec26_20-42-33_qyy-Dell-G15-5520
Dec26_20-50-09_qyy-Dell-G15-5520
Dec26_20-52-16_qyy-Dell-G15-5520
Dec26_20-52-37_qyy-Dell-G15-5520
Dec26_20-53-26_qyy-Dell-G15-5520
```

```python
            _, pred= torch.max(output.data, 1)
            print(output.data, pred,target)
            correct+=(pred==target).sum().item()
        writer.add_scalar( tag: "Loss/validation", test_loss, epoch)
```

```
 [ 0.1517,  0.0943,  0.0074,  0.1624,  0.2578,  0.0783,  0.0404,  0.0484,
   0.2188, -0.0251],
 [ 0.1517,  0.0943,  0.0074,  0.1624,  0.2578,  0.0783,  0.0404,  0.0484,
   0.2188, -0.0251],
 [ 0.1517,  0.0943,  0.0074,  0.1624,  0.2578,  0.0783,  0.0404,  0.0484,
   0.2188, -0.0251],
 [ 0.1517,  0.0943,  0.0074,  0.1624,  0.2578,  0.0783,  0.0404,  0.0484,
   0.2188, -0.0251],
 [ 0.1517,  0.0943,  0.0074,  0.1624,  0.2578,  0.0783,  0.0404,  0.0484,
   0.2188, -0.0251],
 [ 0.1517,  0.0943,  0.0074,  0.1624,  0.2578,  0.0783,  0.0404,  0.0484,
   0.2188, -0.0251]], device='cuda:0') tensor([4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4], device='cuda:0') tensor([5, 5, 1, 5, 8, 2, 1, 4, 4, 8, 0, 7, 2, 0, 2, 3, 9, 6, 8, 6, 9, 1, 9, 4,
8, 4, 5, 2, 3, 5, 2, 4, 0, 1, 8, 7, 5, 2, 6, 8, 3, 8, 7, 7, 7, 7, 6, 4,
2, 8, 0, 0, 6, 1, 8, 6, 7, 6, 1, 2, 4, 1, 2, 7], device='cuda:0')
```

Figure 5: LeNet-5 Error

2. U-net 推理

实验推理结果如图6所示

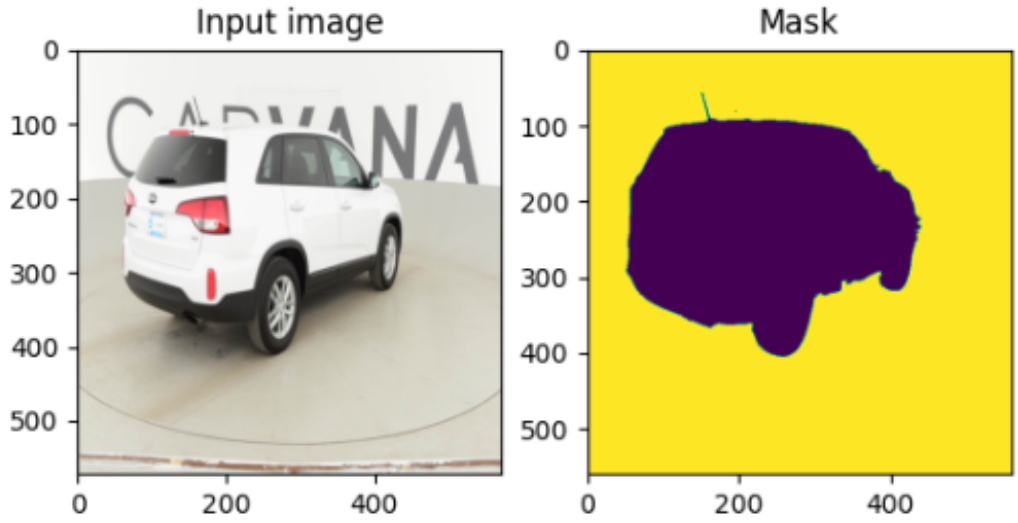


Figure 6: U-net 推理

V 心得体会

- 在本次实验中，对经典的 LeNet-5 进行了复现，取得了不错的效果，同时比较了 Adam 与 SGD 在不同参数条件下的性能。虽然说之前也做过类似的 project 但是从来没有比较过这个的差别，受益匪浅。但是也发现 Adam 在 lr 较大时发生了一些未知错误。
- U-Net 的实验感觉中规中矩，本实验中填空的方式其实很好地避免了机器学习中模型有错误但是精度根本看不出差别的情况。虽然填空也不可避免地遇到了理解歧义的问题，但是总体感觉还是很不错的。

VI Appendix

参考文献

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds. Cham: Springer International Publishing, 2015, pp. 234–241.