

浙江大学



课程综合实践 实验报告

实验名称 简单 CNN 网络训练

姓名学号 陶泓宇 3200103929

实验日期 2021 年 8 月 4 日

目录

1	实验目的和要求	1
2	卷积神经网络基本构成	1
2.1	卷积层	1
2.2	池化层	2
2.3	全连接层	3
3	实验内容和原理	4
3.1	LeNet5 神经网络	4
3.2	ResNet18 网络	7
4	实验数据记录和处理	11
4.1	MNIST	11
4.2	CIFAR10	13
5	讨论、心得	14
6	参考资料	15
7	容器基本信息	15

插图

1	卷积运算示意图	1
2	最大值池化示意图	2
3	全连接层示意图	3
4	LeNet5 实现代码截图	4

5	训练过程代码截图	5
6	训练过程代码截图	7
7	ResNet18 示意图	8
8	ResNet18 结构图	9
9	ResNet18 代码实现截图	9
10	ResNet18 代码实现截图	10
11	ResNet18 代码实现截图	10
12	main_cifar10 文件截图	11
13	mnist 损失函数	12
14	GPU 利用情况截图	12
15	mnist 识别正确率	13
16	cifar10GPU 利用情况	13
17	cifar10 识别率曲线	14
18	cifar10-loss 曲线	14
19	容器信息	16

1 实验目的和要求

本次实验我们将完成两个简单的 CNN 网络，并在 GPU 上加速它的训练，体会基本的网络设计、训练流程。

2 卷积神经网络基本构成

2.1 卷积层

卷积层主要用于提取图像特征。

卷积层进行的处理就是卷积运算。卷积运算相当于图像处理中的“滤波器运算”。对于输入数据，卷积运算以一定间隔滑动滤波器的窗口并应用：将各个位置上滤波器的元素和输入的对应该元素相乘，然后再求和。然后，将这个结果保存到输出的对应位置。将这个过程在所有位置都进行一遍，就可以得到卷积运算的输出。

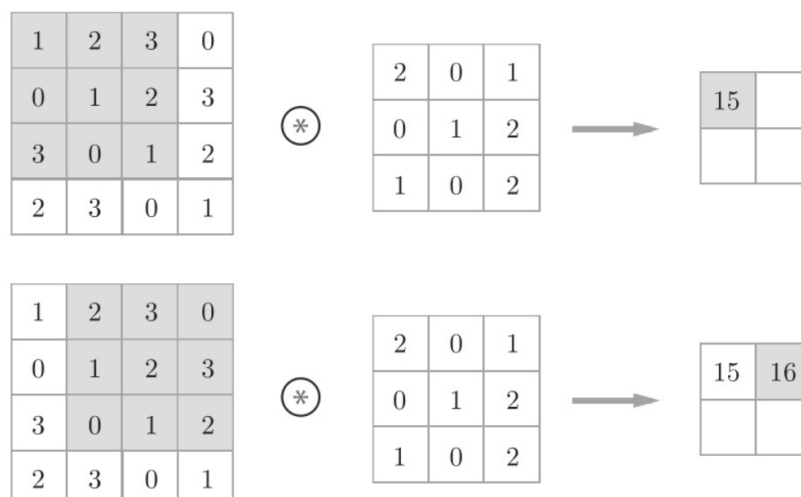


图 1: 卷积运算示意图

在 pytorch 中搭建卷积层采用以下代码：

```
1 nn.Conv2d(1,6,kernel_size=5,stride=1,padding=2)
```

第一个参数是输入通道数，第二个参数是输出通道数，第三个参数是卷积核尺寸，第四个参数是步长，第五个参数是填充层数；设原图尺寸为 $n \times n$ ；卷积核尺寸为 $f \times f$ ；padding 为 p ，stride 为 s ，输出图像像素表达式： $[(n+2p-f)/s + 1] \times [(n+2p-f)/s + 1]$

2.2 池化层

池化用于降低卷积层输出的特征维度，在减少网络参数的同时还能防止过拟合；最大池化示意图如下：

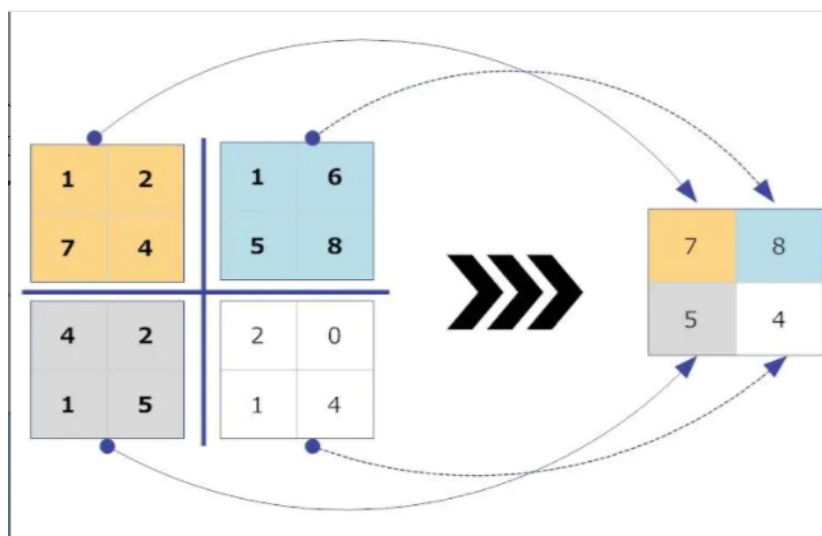


图 2: 最大值池化示意图

在 pytorch 中搭建池化层采用以下代码：

```
1 nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
```

第一个参数是卷积核的尺寸，第二个参数是步长，第三个参数是填充层数。

2.3 全连接层

全连接层用于组合之前提取到的所有特征，为了提升 CNN 网络性能，全连接层每个神经元的激励函数一般采用 ReLU 函数。全连接层示意图如下：

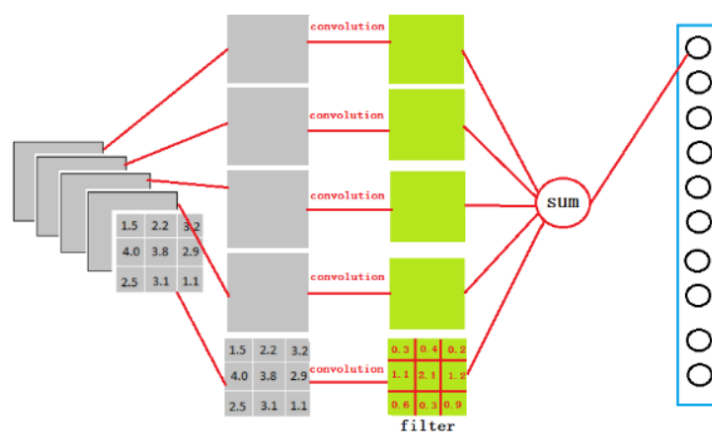


图 3: 全连接层示意图

在 pytorch 中搭建全连接层采用以下代码：

```
1 self.fc1 = nn.Sequential(  
2     nn.Linear(16*5*5,120),  
3     nn.ReLU()  
4 )  
5 self.fc2 = nn.Sequential(  
6     nn.Linear(120,84),  
7     nn.ReLU()  
8 )  
9 self.fc3 = nn.Linear(84,19)
```

`nn.linear()` 函数中第一个参数是输入的二维张量的大小，第二个参数是输出的二维张量的大小

3 实验内容和原理

3.1 LeNet5 神经网络

LeNet5 神经网络的 python 代码截图如下：

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        #卷积层与池化层
        self.conv1 = nn.Sequential(
            #卷积层，输入通道数为1，输出通道数为6，卷积核大小为5x5，步长为1，填充两层
            nn.Conv2d(1,6,kernel_size=5,stride=1,padding=2),#输出仍为nxn大小的矩阵
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,stride=2,padding=0)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(6,16,kernel_size=5,stride=1,padding=0),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,stride=2,padding=0)
        )

        #全连接层
        self.fc1 = nn.Sequential(
            nn.Linear(16*5*5,120),
            nn.ReLU()
        )
        self.fc2 = nn.Sequential(
            nn.Linear(120,84),
            nn.ReLU()
        )
        self.fc3 = nn.Linear(84,10)
    def forward(self,x):
        x = self.conv1(x)
        x = self.conv2(x)

        x = x.view(x.size(0), -1) #全连接层均使用的nn.Linear()线性结构，输入输出维度均为一维，故需要把数据拉为一维
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x
```

图 4: LeNet5 实现代码截图

self.conv1 与 self.conv2 分别定义了两个卷积层 + 池化层的类。

```
1 nn.Conv2d(1,6,kernel_size=5,stride=1,padding=2),
2 #输出仍为nxn大小的矩阵
3 nn.ReLU(),
4 nn.MaxPool2d(kernel_size=2,stride=2,padding=0)
```

上述代码定义了第一个卷积层与第一个池化层，卷积层的输入通道为 1（灰度图像），由于卷积核为 5x5 且步长为 1，输出通道数为 6；nn.ReLU() 采用 ReLu 作为激活函数；池化层则采用最大池化，取 2x2 矩阵中的最大值合成得到新矩阵。

```

1 nn.Conv2d(6,16,kernel_size=5,stride=1,padding=0),
2 nn.ReLU(),
3 nn.MaxPool2d(kernel_size=2,stride=2,padding=0)

```

第二个卷积层与池化层也类似

self.fc 定义了全连接层，采用 ReLu 激活函数来获取相应的输出矩阵

```

1     def forward(self,x):
2         x = self.conv1(x)
3         x = self.conv2(x)
4
5         x = x.view(x.size(0), -1)
6 # 全连接层均使用的 nn.Linear() 线性结构，
7 # 输入输出维度均为一维，故需要把数据拉为一维
8         x = self.fc1(x)
9         x = self.fc2(x)
10        x = self.fc3(x)

```

上述代码则定义了前向传播的过程。

下面是 main.py 文件中部分代码截图：

```

#load data
transform = torchvision.transforms.ToTensor() #定义数据预处理方式：转换 PIL.Image 成 torch.FloatTensor

train_data = torchvision.datasets.MNIST(root="F:\\桌面\\data", #数据目录，这里目录结构要注意。
                                       train=True, #是否为训练集
                                       transform=transform, #加载数据预处理
                                       download=False) #是否下载
test_data = torchvision.datasets.MNIST(root="F:\\桌面\\data",
                                       train=False,
                                       transform=transform,
                                       download=False)

#数据加载器：组合数据集和采样器
train_loader = torch.utils.data.DataLoader(dataset = train_data,batch_size = 64,shuffle = True)
test_loader = torch.utils.data.DataLoader(dataset = test_data,batch_size = 64,shuffle = False)

#define loss
device = torch.device("cuda" if torch.cuda.is_available() else "cpu") #若检测到GPU环境则使用GPU，否则使用CPU
net = LeNet().to(device) #实例化网络，有GPU则将网络放入GPU加速
loss_fuc = nn.CrossEntropyLoss() #多分类问题，选择交叉熵损失函数
optimizer = optim.SGD(net.parameters(),lr = 0.001,momentum = 0.9) #选择SGD，学习率取0.001

```

图 5: 训练过程代码截图

采用 torchvision.datasets.MNIST 来读取数据与划分测试集与训练集。


```

1 train_data = torchvision.datasets.MNIST
2 (root="F:\\桌面\\data",
3 train=True,
4 transform=transform,
5 download=False)
6 test_data = torchvision.datasets.MNIST
7 (root="F:\\桌面\\data",
8 train=False,
9 transform=transform,
10 download=False)

```

采用 `torch.utils.data.DataLoader` 来读取数据，`dataset` 参数指定读取的数据集，`batch size` 指定一次读取的数据数量，`shuffle=True` 时随机打乱数据。

```

1 train_loader = torch.utils.data.DataLoader
2 (dataset = train_data,batch_size = 64,shuffle = True)
3 test_loader = torch.utils.data.DataLoader
4 (dataset = test_data,batch_size = 64,shuffle = False)

```

若检测到 GPU 环境则使用 GPU，否则使用 CPU

```

1 device = torch.device
2 ("cuda" if torch.cuda.is_available() else "cpu")
3 net = LeNet().to(device)

```

选择交叉熵损失函数作为损失函数

```

1 device = torch.device
2 loss_fuc = nn.CrossEntropyLoss()

```

选择 SGD（随机梯度下降）作为优化器，`net.parameters` 获取 `net` 网络的参数，`lr` 指定学习率，`momentum` 表示冲量

```

1 optimizer = optim.SGD
2 (net.parameters(),lr = 0.001,momentum = 0.9)

```

训练时采用梯度清零的方式进行训练，设定训练的 epoch 为 64，每 100 个 batch 打印一次平均 loss。

```
#开始训练
EPOCH = 64 #迭代次数
for epoch in range(EPOCH):
    sum_loss = 0
    #数据读取
    for i,data in enumerate(train_loader):
        inputs,labels = data
        inputs, labels = inputs.to(device), labels.to(device) #有GPU则将数据置入GPU加速

        # 梯度清零
        optimizer.zero_grad()

        # 传递损失 + 更新参数
        output = net(inputs)
        loss = loss_fuc(output,labels)
        loss.backward()
        optimizer.step()

        # 每训练100个batch打印一次平均loss
        sum_loss += loss.item()
        if i % 100 == 99:
            print('[Epoch:%d, batch:%d] train loss: %.03f' % (epoch + 1, i + 1, sum_loss / 100))
            sum_loss = 0.0

    correct = 0
    total = 0

    for data in test_loader:
        test_inputs, labels = data
        test_inputs, labels = test_inputs.to(device), labels.to(device)
        outputs_test = net(test_inputs)
        _, predicted = torch.max(outputs_test.data, 1) #输出得分最高的类
        total += labels.size(0) #统计50个batch 图片的总个数
        correct += (predicted == labels).sum() #统计50个batch 正确分类的个数
    print('第{}个epoch的识别准确率为: {}'.format (epoch + 1, 100*correct.item()/total))
```

图 6: 训练过程代码截图

3.2 ResNet18 网络

在构建训练 cifar10 的神经网络，我参考了 LesNet18 网络，根据简书上的一张图搭建了 ResNet18 网络，原图如下，但我发现输入的图像尺寸是 3x32x32，并不是图中所说的 3x224x224，因此，我去掉了最后的那个卷积核大小为 7 的平均池化层。

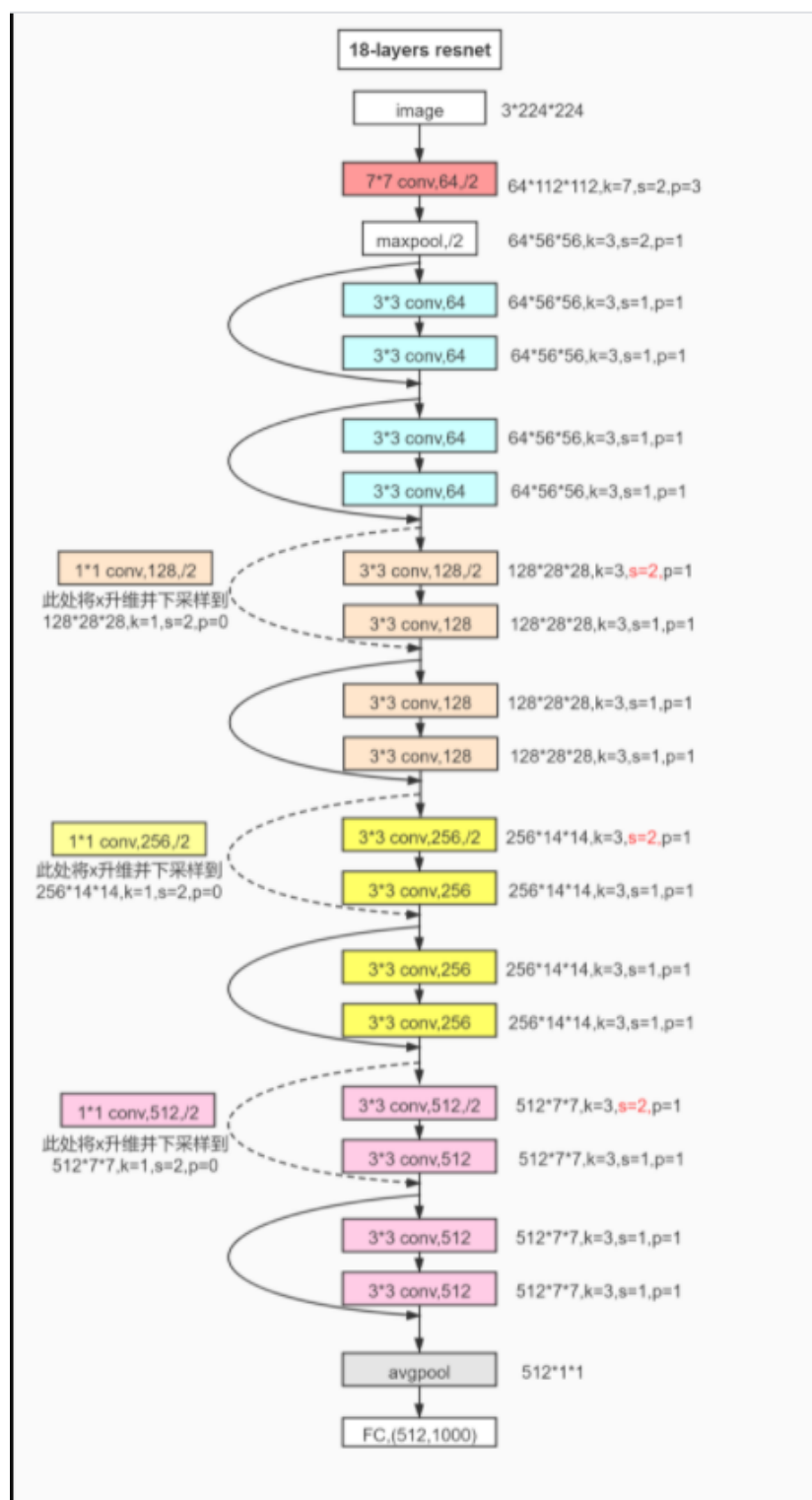


图 7: ResNet18 示意图

ResNet18 的作者提出的结构图如下图左侧 18-layers 对应的那一列：

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

图 8: ResNet18 结构图

该网路共 17 个卷积层，1 个全连接层，第一个卷积层没有 shortcut 机制，之后接的几个卷积层都有 shortcut 机制，最后再接一个全连接层。

ResNet18 的代码实现截图如下：

```

class basic_block1(nn.Module):
    """定义基础块，用于输出通道数和输入通道数相同。class参数个(out_channels)"""
    def __init__(self, in_channels):
        super(basic_block1, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, in_channels, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(in_channels)
        )
        self.conv2 = nn.Conv2d(in_channels, in_channels, kernel_size=3, stride=1, padding=1)
    def forward(self, x):
        y = self.conv1(x)
        y = self.conv2(y)
        return F.relu(x+y) #输出是输入x+经过两次卷积以后的和再relu一下

class basic_block2(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(basic_block2, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, padding=0)
        self.conv2 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
        )
        self.conv3 = nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, padding=0)
    def forward(self, x):
        z = self.conv1(x)
        y = self.conv2(x)
        y = self.conv3(y)
        return F.relu(y+z)

# 定义ResNet18
class ResNet18(nn.Module):
    """output_shape = (image_shape-filter_shape+padding)/stride + 1"""
    def __init__(self):
        super(ResNet18, self).__init__()
        self.conv1 = nn.Sequential(

```

图 9: ResNet18 代码实现截图

```

class basic_block(nn.Module):
    def __init__(self, inchannel, outchannel):
        super(basic_block, self).__init__()
        self.conv1 = nn.Conv2d(inchannel, outchannel, kernel_size=3, stride=2, padding=0)
        self.conv2 = nn.Sequential(
            nn.Conv2d(inchannel, outchannel, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(outchannel, outchannel, kernel_size=3, stride=1, padding=1)
        )
        self.conv3 = nn.Conv2d(inchannel, outchannel, kernel_size=3, stride=1, padding=0)
    def forward(self, x):
        z = self.conv1(x)
        y = self.conv2(z)
        w = self.conv3(y)
        return F.relu(z+y)

class ResNet18(nn.Module):
    def __init__(self):
        super(ResNet18, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=1)
        )
        self.conv2 = basic_block([inchannel=64])
        self.conv_insert = basic_block([inchannel=64])
        self.conv3 = basic_block([inchannel=64, outchannel=128])
        self.conv4 = basic_block([inchannel=128])
        self.conv5 = basic_block([inchannel=128, outchannel=256])
        self.conv6 = basic_block([inchannel=256])
        self.conv7 = basic_block([inchannel=256, outchannel=512])
        self.conv8 = basic_block([inchannel=512])
        self.fc1 = nn.Linear(512, 1000)
    def forward(self, x):
        self.fc1 = nn.Linear(512, 1000)
        print(x.shape)
        x = F.relu(F.max_pool2d(x, 2, 2))
        x = self.conv1(x)
        print(x.shape)
        x = self.conv2(x)
        print(x.shape)
        x = self.conv_insert(x)
        print(x.shape)
        x = self.conv3(x)
        print(x.shape)
        x = self.conv4(x)
        print(x.shape)
        x = self.conv5(x)
        print(x.shape)
        x = self.conv6(x)
        print(x.shape)
        x = self.conv7(x)
        print(x.shape)
        x = self.conv8(x)
        print(x.shape)
        x = x.view(x.size(0), -1) # 降维为二维数据
        print(x.shape)

```

图 10: ResNet18 代码实现截图

```

self.conv2 = basic_block([inchannel=64])
self.conv_insert = basic_block([inchannel=64])
self.conv3 = basic_block([inchannel=64, outchannel=128])
self.conv4 = basic_block([inchannel=128])
self.conv5 = basic_block([inchannel=128, outchannel=256])
self.conv6 = basic_block([inchannel=256])
self.conv7 = basic_block([inchannel=256, outchannel=512])
self.conv8 = basic_block([inchannel=512])
self.fc1 = nn.Linear(512, 1000)
    def forward(self, x):
        self.fc1 = nn.Linear(512, 1000)
        print(x.shape)
        x = F.relu(F.max_pool2d(x, 2, 2))
        x = self.conv1(x)
        print(x.shape)
        x = self.conv2(x)
        print(x.shape)
        x = self.conv_insert(x)
        print(x.shape)
        x = self.conv3(x)
        print(x.shape)
        x = self.conv4(x)
        print(x.shape)
        x = self.conv5(x)
        print(x.shape)
        x = self.conv6(x)
        print(x.shape)
        x = self.conv7(x)
        print(x.shape)
        x = self.conv8(x)
        print(x.shape)
        x = x.view(x.size(0), -1) # 降维为二维数据
        print(x.shape)

```

图 11: ResNet18 代码实现截图

main_cifar10.py 文件与 main_minist.py 文件类似:

```

39
40 #开始训练
41 EPOCH = 1 #迭代次数
42 for epoch in range(EPOCH):
43     sum_loss = 0
44     #数据读取
45     for i,data in enumerate(train_loader):
46         print(i)
47         inputs,labels = data
48         inputs, labels = inputs.to(device), labels.to(device) #有GPU则将数据置入GPU加速
49
50         # 梯度清零
51         optimizer.zero_grad()
52
53         # 传递损失 + 更新参数
54         output = net(inputs)
55         loss = loss_fuc(output,labels)
56         loss.backward()
57         optimizer.step()
58
59     # 每训练100个batch打印一次平均loss
60     sum_loss += loss.item()
61     if i % 100 == 99:
62         print('[Epoch:%d, batch:%d] train loss: %.03f' % (epoch + 1, i + 1, sum_loss / 100))
63         sum_loss = 0.0
64
65     correct = 0
66     total = 0
67
68     for data in test_loader:
69         test_inputs, labels = data
70         test_inputs, labels = test_inputs.to(device), labels.to(device)
71         outputs_test = net(test_inputs)
72         _, predicted = torch.max(outputs_test.data, 1) #输出得分最高的类
73         total += labels.size(0) #统计50个batch 图片的总个数
74         correct += (predicted == labels).sum() #统计50个batch 正确分类的个数
75     print('the {}epoch right: {}%'.format(epoch + 1, 100*correct.item()/total))
76

```

图 12: main_cifar10 文件截图

训练的损失函数取 `nn.CrossEntropyLoss()`，优化器取 SGD，学习率取 0.01

4 实验数据记录和处理

4.1 MNIST

100 个 epoch 后 LeNet 网络的损失函数如下图所示：

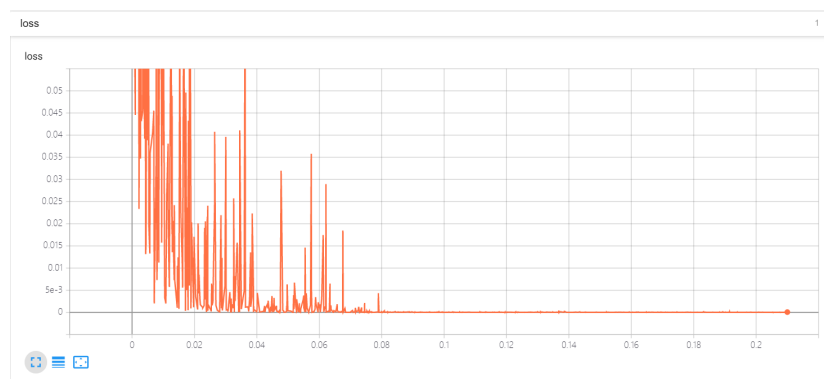


图 13: mnist 损失函数

GPU 利用情况截图如下：

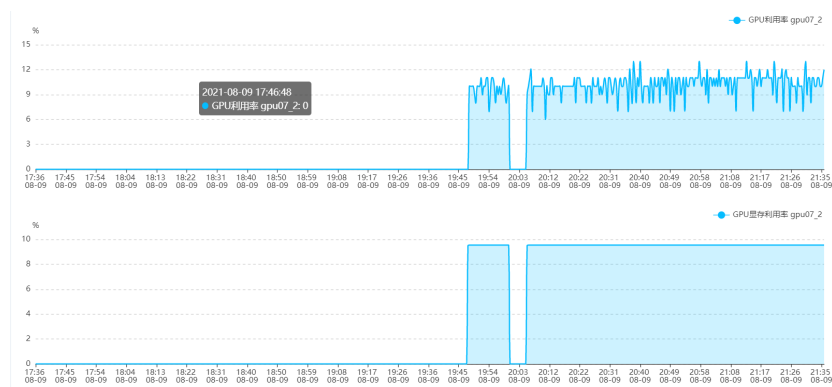


图 14: GPU 利用情况截图

识别正确率如下：

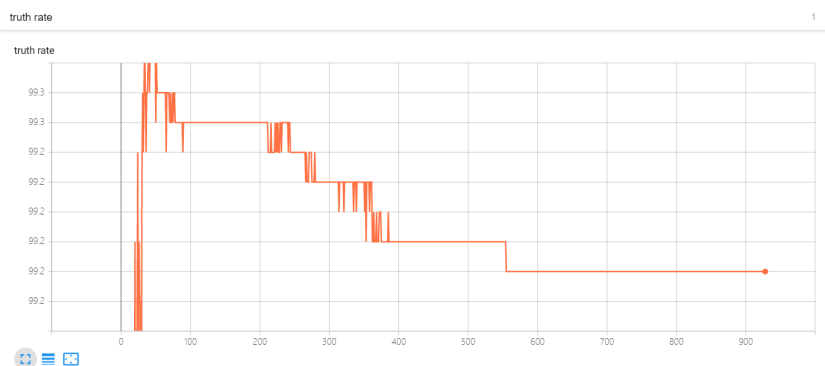


图 15: mnist 识别正确率

4.2 CIFAR10

训练时的 GPU 利用情况如下：

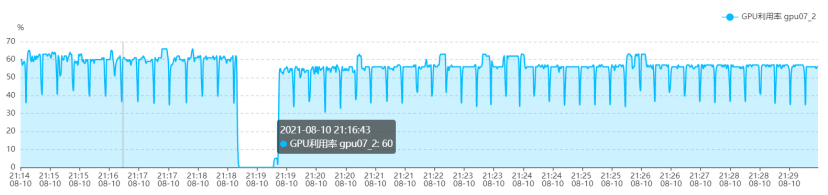


图 16: cifar10GPU 利用情况

识别正确率如下：

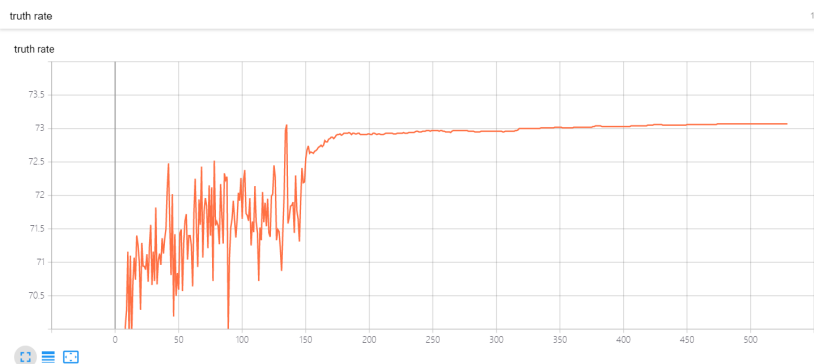


图 17: cifar10 识别率曲线

loss 函数曲线如下:

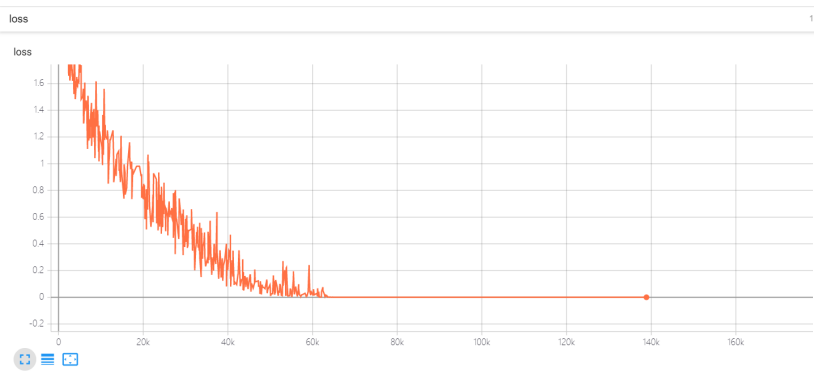


图 18: cifar10-loss 曲线

5 讨论、心得

Q: 池化层有很多种, 较常使用的是平均和最大, 在这个数据集上使用有什么的区别, 哪个效果好一些?

A: 平均值池化取平均值, 保留了背景特征, 但容易模糊图像, 最大值池化则保留纹理特征, 在 MNIST 中采取最大池化以保留文字边缘特征, 而在 CIFAR10 中则采取平均值池化以获取背景特征 (图像较为复杂)。

Q: 你采用了哪些策略来改进你的网络, 效果如何?

A: 在进行 cifar10 的识别的时候, 尝试了各种优化方法, 使用 Adamax 时, 最开始的识别准确率比 SGD 要高, 但最后收敛在百分之 57 的正确识别率上, 在使用 SGD 时, 还尝试了调整学习率, 在 30 个 epoch 后, 学习率取 0.01 的识别正确率在百分之 70 左右, 而取 0.02 时识别正确率在百分之 67 左右, 学习率取 0.001 时识别正确率可以收敛在百分之 75.

6 参考资料

[1] 卷积神经网络 LeNet-5 的 pytorch 代码实现 (LeNet5 与 main 函数的编写很大程度上参考了这篇博客)

https://blog.csdn.net/didi_ya/article/details/108317958

[2] 深度学习小白实现残差网络 resnet18 ——pytorch

https://blog.csdn.net/weixin_44331304/article/details/106127552

[3] Pytorch 实战 2: ResNet-18 实现 Cifar-10 图像分类

<https://blog.csdn.net/sunqiande88/article/details/80100891>

[4] resnet18 50 网络结构以及 pytorch 实现代码

<https://www.jianshu.com/p/085f4c8256f1>

[5] 通过 Pytorch 实现 ResNet18

<https://zhuanlan.zhihu.com/p/157134695>

7 容器基本信息

因为怕删掉原来 tensorflow 的容器以后再申请 pytorch 的容器会花很久, 我就直接在原始容器上面安装了 pytorch, 容器信息如下:

环境名称	20210723153248_thy	创建时间	2021-07-23 15:33:11
镜像	10.202.2.10:100:5000/tensorflow/tensorflow:1.14-cuda10-py36	状态	运行中
资源组	Tra_group_chenjh	GPU	GeForce RTX 2080 Ti:1
CPU	4	挂载路径	/chenjh08
副本数	1	shm_size	TotalMemory/2
数据挂载路径			

图 19: 容器信息