

研究生《深度学习》课程

实验报告

|  |  |
| --- | --- |
| **实验名称：** | **卷积神经网络实验** |
| **姓 名：** |  |
| **学 号：** |  |
| **日 期：** |  |

## 一、实验内容

#### 1 二维卷积实验

1.1 手写二维卷积的实现，并在至少一个数据集上进行实验，从训练时间、预测精度、Loss变化等角度分析实验结果（最好使用图表展示）

1.2 使用torch.nn实现二维卷积，并在至少一个数据集上进行实验，从训练时间、预测精度、Loss变化等角度分析实验结果（最好使用图表展示）

1.3 不同超参数的对比分析（包括卷积层数、卷积核大小、batchsize、lr等）选其中至少1-2个进行分析

1.4 使用PyTorch实现经典模型AlexNet并在至少一个数据集进行试验分析 （无GPU环境则至少实现模型）

1.5 使用实验2中的前馈神经网络模型来进行实验，并将实验结果与卷积模型结果进行对比分析（选作）

2 空洞卷积实验

2.1 使用torch.nn实现空洞卷积，要求dilation满足HDC条件（如1,2,5）且要堆叠多层并在至少一个数据集上进行实验，从训练时间、预测精度、Loss变化等角度分析实验结果（最好使用图表展示）

2.2 将空洞卷积模型的实验结果与卷积模型的结果进行分析比对，训练时间、 预测精度、Loss变化等角度分析

2.3 不同超参数的对比分析（包括卷积层数、卷积核大小、不同dilation的选择，batchsize、lr等）选其中至少1-2个进行分析（选做）

#### 3 残差网络实验

3.1实现给定结构的残差网络，在至少一个数据集上进行实验，从训练时间、预测精度、Loss变化等角度分析实验结果（最好使用图表展示）

3.2 将残差网络与空洞卷积相结合，在至少一个数据集上进行实验，从训练时间、预测精度、Loss变化等角度分析实验结果（最好使用图表展示） （选做）

## 二、实验设计

（略）

## 三、实验环境及实验数据集

实验环境：Python3.7+Anaconda+PyTorch+Jupyter Notebook

涉及的数据集：车辆分类数据集、照片去雾数据集

## 四、实验过程

#### 4.1 二维卷积实验

1 手写二维卷积的实现

1. 自定义多通道卷积操作
2. def conv2d(X, K):
3. '''
4. :param X: 样本输入，shape(batch\_size,H,W)
5. :param K: 卷积核，shape(k\_h,k\_w)
6. :return: Y 卷积结果，shape(batch\_size, H-k\_h+1, W-k\_w+1)
7. '''
8. batch\_size, H, W = X.shape
9. k\_h, k\_w = K.shape
10. *# 初始化 Y*
11. Y = torch.zeros((batch\_size, H - k\_h + 1, W - k\_w + 1)).to(device)
12. for i in range(Y.shape[1]):
13. for j in range(Y.shape[2]):
14. Y[:, i, j] = (X[:, i: i + k\_h, j:j + k\_w] \* K).sum(dim=2).sum(dim=1)
15. return Y
16. def conv2d\_multi\_in(X, K):
17. '''
18. :param X: (batch\_size, C\_in，H，W)代表有C个输入通道
19. :param K: (C\_in, k\_h, k\_w)
20. :return: (batch\_size, H\_out, W\_out)
21. '''
22. res = conv2d(X[:, 0, :, :], K[0, :, :])
23. for i in range(1, X.shape[1]):  *# 多个通道的结果相加*
24. res += conv2d(X[:, i, :, :], K[i, :, :])
25. return res
26. *# 实现多输出通道*
27. *# 输出通道数 = 卷积核个数*
28. def conv2d\_multi\_in\_out(X, K):
29. '''
30. :param X: (batch\_size, C\_in，H，W)代表有C个输入通道
31. :param K: (K\_num, C\_in, k\_h, k\_w) k\_num表示卷积核的个数
32. :return: (batch\_size, K\_num, H\_out, W\_out)
33. '''
34. return torch.stack([conv2d\_multi\_in(X, k) for k in K], dim=1)
35. 自定义卷积层
36. class MyConv2D(torch.nn.Module):
37. def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size):
38. super(MyConv2D, self).\_\_init\_\_()
39. *# 初始化卷积层2个参数：卷积核、偏差*
40. if isinstance(kernel\_size, int):  *# 如果kernel size是一个数*
41. kernel\_size = (kernel\_size, kernel\_size)
42. *# weight的shape:(卷积核个数/输出通道数，输入通道数，卷积核高，卷积核宽)*
43. *# torch.randn:返回一个符合均值为0，方差为1的正态分布（标准正态分布）中填充随机数的张量*
44. self.weight = torch.nn.Parameter(torch.randn((out\_channels, in\_channels) + kernel\_size))
45. self.bias = torch.nn.Parameter(torch.randn(out\_channels, 1, 1))
46. def forward(self, x):
47. '''
48. :param x:
49. :return:
50. '''
51. return conv2d\_multi\_in\_out(x, self.weight) + self.bias
52. class MyConvModule(torch.nn.Module):
53. def \_\_init\_\_(self):
54. super(MyConvModule, self).\_\_init\_\_()
55. *# 定义一层卷积*
56. self.conv = torch.nn.Sequential(
57. MyConv2D(in\_channels=3, out\_channels=32, kernel\_size=3),
58. torch.nn.BatchNorm2d(32),
59. torch.nn.ReLU(inplace=True)  *# inplace=True表示计算出来的结果会替换掉原来的Tensor*
60. )
61. *# 输出层，将输出通道数变为分类数量*
62. self.fc = torch.nn.Linear(32, num\_classes)
63. def forward(self, X):
64. *# 图片经过一层卷积，输出（batch\_size，C\_out, H, W)*
65. out = self.conv(X)
66. *# 使用平均池化层将图片大小变为1\*1(图片原大小32\*32,卷积后为30\*30*
67. out = F.avg\_pool2d(out, 30)
68. *# 将out从shape batch\_size\*32\*1\*1变为batch\_size\*32*
69. out = out.squeeze()  *# squeeze的用法：*
70. *# 输入到全连接层*
71. out = self.fc(out)
72. return out
73. 训练函数
74. def train(net, data\_loader, device):
75. net.train()  # 指定为训练模式
76. train\_batch\_num = len(data\_loader)
77. total\_loss = 0.0
78. correct = 0  # 记录共有多少个样本被正确分类
79. sample\_num = 0
80. # 遍历每个batch进行训练
81. for data, target in data\_loader:
82. *# 将图片和标签放入指定的device中*
83. data = data.to(device)
84. target = target.to(device)
85. *# 将当前梯度清零*
86. optimizer.zero\_grad()
87. *# 使用模型计算出结果*
88. y\_hat = net(data)
89. *# 计算损失*
90. loss\_ = loss(y\_hat, target)
91. *# 进行反向传播*
92. loss\_.backward()
93. optimizer.step()
94. total\_loss += loss\_.item()
95. cor = (torch.argmax(y\_hat, 1) == target).sum().item()
96. correct += cor
97. *# 累加当前的样本总数*
98. sample\_num += target.shape[0]
99. print('loss: %.4f  acc: %.4f' % (loss\_.item(), cor/target.shape[0]))
100. *# 平均loss和准确率*
101. loss\_ = total\_loss / train\_batch\_num
102. acc = correct / sample\_num
103. return loss\_, acc
104. 测试函数
105. def test(net, data\_loader, device):
106. net.eval()  # 指定当前模式为测试模式（针对BN层和dropout层）
107. test\_batch\_num = len(data\_loader)
108. total\_loss = 0
109. correct = 0
110. sample\_num = 0
111. *# 指定不进行梯度计算（没有反向传播也会计算梯度，增大GPU开销*
112. with torch.no\_grad():
113. for data, target in data\_loader:
114. data = data.to(device)
115. target = target.to(device)
116. output = net(data)
117. loss\_ = loss(output, target)
118. total\_loss += loss\_.item()
119. correct += (torch.argmax(output, 1) == target).sum().item()
120. sample\_num += target.shape[0]
121. loss\_ = total\_loss / test\_batch\_num
122. acc = correct / sample\_num
123. return loss\_, acc
124. 模型训练并验证
125. # 模型训练与测试
126. train\_loss\_list = []
127. train\_acc\_list = []
128. test\_loss\_list = []
129. test\_acc\_list = []
130. for epoch in range(num\_epochs):
131. # 在训练集上训练
132. train\_loss, train\_acc = train(net, data\_loader=train\_iter, device=device)
133. # 测试集上验证
134. test\_loss, test\_acc = test(net, data\_loader=test\_iter, device=device)
135. train\_loss\_list.append(train\_loss)
136. train\_acc\_list.append(train\_acc)
137. test\_loss\_list.append(test\_loss)
138. test\_acc\_list.append(test\_acc)
139. print('epoch %d, train loss: %.4f, train acc: %.3f' % (epoch+1, train\_loss, train\_acc))
140. print('test loss: %.4f, test acc: %.3f' % (test\_loss, test\_acc))

2 使用torch.nn实现二维卷积

1. PyTorch封装卷积层
2. class ConvModule(torch.nn.Module):
3. def \_\_init\_\_(self):
4. super(ConvModule, self).\_\_init\_\_()
5. *# 定义一个三层卷积（卷积层越少曲线越平滑，卷积层越多acc越高）*
6. self.conv = torch.nn.Sequential(
7. *# 第一层卷积*
8. *# stride步长,padding填充*
9. torch.nn.Conv2d(in\_channels=3, out\_channels=32, kernel\_size=3, stride=1, padding=0),
10. torch.nn.BatchNorm2d(32),
11. torch.nn.ReLU(inplace=True),
12. *# 第二层卷积*
13. torch.nn.Conv2d(in\_channels=32, out\_channels=64, kernel\_size=3, stride=1, padding=0),
14. torch.nn.BatchNorm2d(64),
15. torch.nn.ReLU(inplace=True),
16. *# 第三层卷积*
17. torch.nn.Conv2d(in\_channels=64, out\_channels=128, kernel\_size=3, stride=1, padding=0),
18. torch.nn.BatchNorm2d(128),
19. torch.nn.ReLU(inplace=True)
20. )
21. *# 输出层，将输出通道数变为分类数*
22. self.fc = torch.nn.Linear(128, num\_classes)
23. def forward(self, X):
24. out = self.conv(X)  *# 输出维度(batch\_size, C\_out, H, W)*
25. *# 平均池化*
26. out = F.avg\_pool2d(out, 26)  *# 池化后图片大小1\*1*
27. *# 将out从batch size\*128\*1\*1变为batch size\*128*
28. out = out.squeeze()
29. out = self.fc(out)
30. return out

3 不同超参数的对比分析

1. torch.nn构造5层卷积模型
2. class ConvModule\_5(nn.Module):
3. """5层卷积"""
4. def \_\_init\_\_(self):
5. super(ConvModule\_5, self).\_\_init\_\_()
6. self.conv = nn.Sequential(
7. nn.Conv2d(in\_channels=3, out\_channels=32, kernel\_size=3, stride=1, padding=0),
8. nn.BatchNorm2d(32),
9. nn.ReLU(inplace=True),
10. nn.Conv2d(in\_channels=32, out\_channels=64, kernel\_size=3, stride=1, padding=0),
11. nn.BatchNorm2d(64),
12. nn.ReLU(inplace=True),
13. nn.Conv2d(in\_channels=64, out\_channels=128, kernel\_size=3, stride=1, padding=0),
14. nn.BatchNorm2d(128),
15. nn.ReLU(inplace=True),
16. nn.Conv2d(in\_channels=128, out\_channels=128, kernel\_size=3, stride=1, padding=0),
17. nn.BatchNorm2d(128),
18. nn.ReLU(inplace=True),
19. nn.Conv2d(in\_channels=128, out\_channels=128, kernel\_size=3, stride=1, padding=0),
20. nn.BatchNorm2d(128),
21. nn.ReLU(inplace=True),
22. )
23. self.fc = nn.Linear(128, num\_classes)  *# 因为上一层(avg\_pool2d)输出为(16,128,3,2)*
24. *# self.fc = nn.Linear(128\*3\*2, num\_classes)  # 因为上一层(avg\_pool2d)输出为(16,128,3,2)*
25. def forward(self, X):
26. *# print(X.shape)*
27. out = self.conv(X)
28. *# print(out.shape)*
29. out = F.avg\_pool2d(out, kernel\_size=30)  *# 平均池化层将图片的大小变为1\*1*
30. *# print(out.shape)*
31. out = out.squeeze()  *# squeeze 去除所有item数量为1的维度*
32. *# out = out.flatten(start\_dim=1)  # flatten 将高维向量展平为 2 维向量*
33. *# out = out.view(-1, 128)*
34. *# print(out.shape)*
35. out = self.fc(out)
36. return out

4 使用PyTorch实现经典模型AlexNet

1. class AlexNet(nn.Module):
2. def \_\_init\_\_(self, num\_classes=3):
3. super(AlexNet, self).\_\_init\_\_()
4. self.block1 = nn.Sequential(
5. nn.Conv2d(in\_channels=3, out\_channels=96, kernel\_size=11, stride=4, padding=2),
6. nn.ReLU(inplace=True),
7. nn.MaxPool2d(kernel\_size=3, stride=2),
8. )
9. self.block2 = nn.Sequential(
10. nn.Conv2d(in\_channels=96, out\_channels=256, kernel\_size=5, stride=1, padding=2),
11. nn.ReLU(inplace=True),
12. nn.MaxPool2d(kernel\_size=3, stride=2),
13. )
14. self.block3 = nn.Sequential(
15. nn.Conv2d(in\_channels=256, out\_channels=384, kernel\_size=3, stride=1, padding=1),
16. nn.ReLU(inplace=True),
17. )
18. self.block4 = nn.Sequential(
19. nn.Conv2d(in\_channels=384, out\_channels=384, kernel\_size=3, stride=1, padding=1),
20. nn.ReLU(inplace=True),
21. )
22. self.block5 = nn.Sequential(
23. nn.Dropout(p=0.5),
24. nn.Linear(in\_features=384\*3\*3, out\_features=4096),
25. nn.ReLU(inplace=True)
26. )
27. self.block6 = nn.Sequential(
28. nn.Dropout(p=0.5),
29. nn.Linear(in\_features=4096, out\_features=4096),
30. nn.ReLU(inplace=True)
31. )
32. self.block7 = nn.Sequential(
33. nn.Linear(in\_features=4096, out\_features=num\_classes)
34. )
35. def forward(self,x):
36. x = self.block1(x)
37. x = self.block2(x)
38. x = self.block3(x)
39. x = self.block4(x)
40. *#         print(x.size())*
41. x = x.view(x.size(0), 384\*3\*3)
42. x = self.block5(x)
43. x = self.block6(x)
44. x = self.block7(x)
45. return x

#### 4.2 空洞卷积实验

1 使用torch.nn实现空洞卷积，要求dilation满足HDC条件（如1,2,5）且要堆叠多层

1. 空洞卷积的实现
2. class ConvModule\_dilation(nn.Module):
3. def \_\_init\_\_(self):
4. super(ConvModule\_dilation, self).\_\_init\_\_()
5. self.conv = nn.Sequential(
6. nn.Conv2d(in\_channels=3, out\_channels=32, kernel\_size=3, stride=1, padding=0, dilation = 1),
7. nn.BatchNorm2d(32),
8. nn.ReLU(inplace=True),
9. nn.Conv2d(in\_channels=32, out\_channels=64, kernel\_size=3, stride=1, padding=0, dilation = 2),
10. nn.BatchNorm2d(64),
11. nn.ReLU(inplace=True),
12. nn.Conv2d(in\_channels=64, out\_channels=128, kernel\_size=3, stride=1, padding=0, dilation = 5),
13. nn.BatchNorm2d(128),
14. nn.ReLU(inplace=True)
15. )
16. self.fc = nn.Linear(128, num\_classes)  *# 上一层(avg\_pool2d)输出为(16,128,3,2)*
17. *# self.fc = nn.Linear(128\*3\*2, num\_classes)  # 上一层(avg\_pool2d)输出为(16,128,3,2)*
18. def forward(self, X):
19. *# print(X.shape)*
20. out = self.conv(X)
21. *# print(out.shape)*
22. out = F.avg\_pool2d(out, kernel\_size=30)  *# 平均池化层将图片的大小变为1\*1*
23. *# print(out.shape)*
24. out = out.squeeze()  *# squeeze 去除所有item数量为1的维度*
25. *# out = out.flatten(start\_dim=1)  # flatten 将高维向量展平为 2 维向量*
26. *# out = out.view(-1, 128)*
27. *# print(out.shape)*
28. out = self.fc(out)
29. return out

#### 4.3 残差卷积实验

1实现给定结构的残差网络

1. 残差网络的实现
2. class ResidualBlock(nn.Module):
3. def \_\_init\_\_(self, inchannel, outchannel, stride = 1):
4. super(ResidualBlock, self).\_\_init\_\_()
5. self.left = nn.Sequential(
6. nn.Conv2d(inchannel, outchannel, kernel\_size=3, stride=stride, padding=1, bias=False),
7. nn.BatchNorm2d(outchannel),
8. nn.ReLU(inplace=True),
9. nn.Conv2d(outchannel, outchannel, kernel\_size=3, stride=1, padding=1, bias=False),
10. nn.BatchNorm2d(outchannel)
11. )
12. self.shortcut =  nn.Sequential()
13. if stride != 1 or inchannel != outchannel:
14. self.shortcut = nn.Sequential(
15. nn.Conv2d(inchannel, outchannel, kernel\_size=1, stride=stride, bias=False),
16. nn.BatchNorm2d(outchannel)
17. )
18. def forward(self,x):
19. out = self.left(x)
20. out += self.shortcut(x)
21. out = F.relu(out)
22. return out

## 五、实验结果

#### 5.1 二维卷积实验

1 手写二维卷积的实现

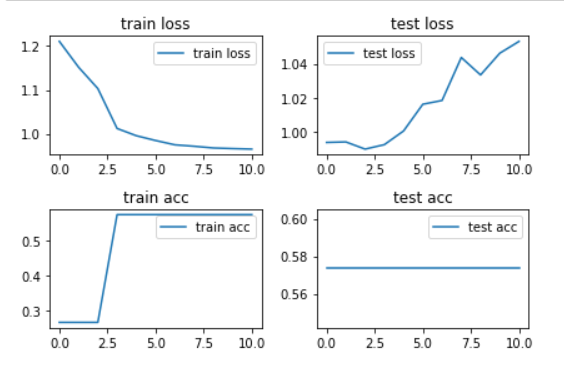


图5.1 手写二维卷积结果图

结果分析：手动实现二维卷积的实验，由于卷积操作使用for循环实现，所以每轮运行耗时长，一轮耗时几乎30mins。在训练集上 loss始终在下降，但测试集上的 loss 在短暂下降后出现了上升，说明出现了“过拟合”现象。说明模型具备了一定的学习能力却学习的不太好。

1. 使用torch.nn实现二维卷积

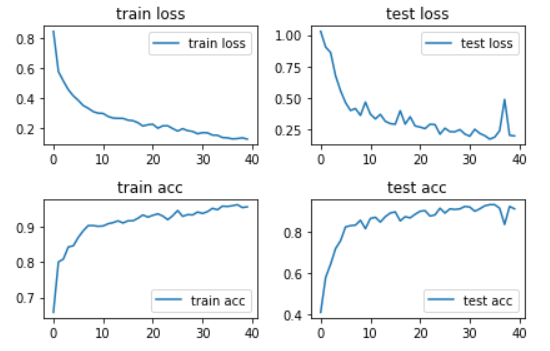


图5.2 torch.nn实现二维卷积结果图

结果分析：利用PyTorch自带的封装卷积层操作，减少了for循环时间，耗时比手写卷积实验短了很多。在训练集上 loss始终在下降，但测试集上的 loss 下降后又出现了上升，说明出现了“过拟合”现象，而且出现了测试集上的loss值，accuracy值比训练集上的抖动更大，更不光滑的结果，说明模型具备了一定的学习能力却学习的不太好。

1. 不同超参数的对比分析

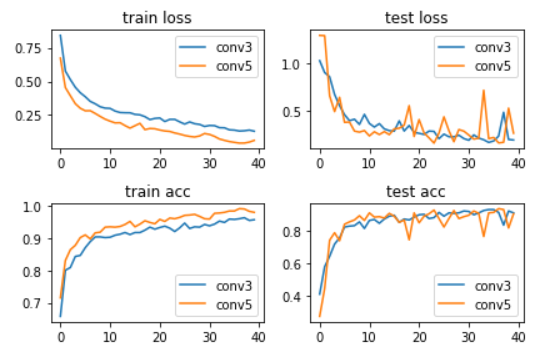


图5.3 3层卷积与5层卷积对比结果图

结果分析：在3层卷积与5层卷积的实验结果对比中，发现卷积层越少曲线越平滑，卷积层越多accuracy值越高。

1. PyTorch实现经典模型AlexNet

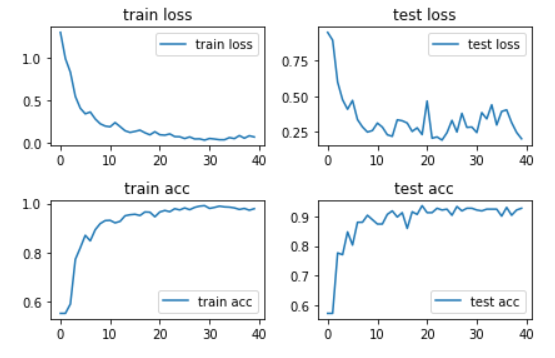


图5.4 PyTorch实现经典模型AlexNet结果图

结果分析：测试集上的结果值的抖动比训练集上值的抖动要大，结果曲线图也没训练集光滑。训练集上loss始终在下降，accuracy始终上升，但测试集上的 loss 下降后又出现了上升，说明出现了“过拟合”现象。

#### 5.2 空洞卷积实验

1.torch.nn实现空洞卷积及空洞卷积模型与卷积模型对比

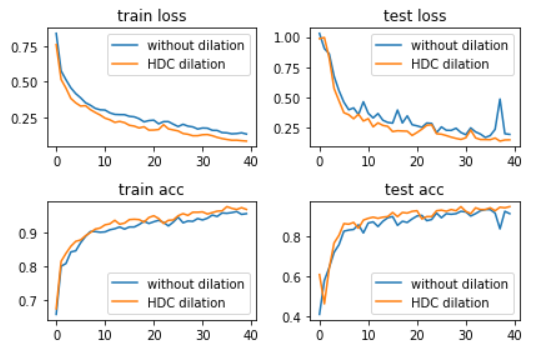


图5.5 torch.nn实现空洞卷积结果图

结果分析：与普通卷积模型对比，我们发现满足HDC空洞卷积的模型表现效果更好，模型在训练集上可以达到一个更低的loss和更高的准确率，且波动的幅度小。空洞卷积的训练速度也要强于普通卷积。

#### 5.3 残差卷积实验

1.实现给定结构的残差网络

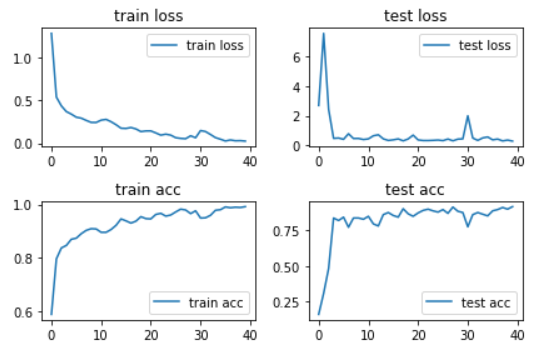


图5.6 残差网络结果图

结果分析：残差网络模型在训练集上loss始终在下降，accuracy始终上升，但测试集上的 loss 抖动太大，说明模型的学习能力不太好。

## 六、实验心得体会

实验是个将课堂知识与动手实践融会贯通的部分，在这次的实验过程中我对卷积网络的逻辑有了更清楚的认识，清楚了卷积网络通过卷积、池化、全连接一步一步搭建的过程。虽然在这次实验问题上碰上了一些问题，但最后在多次查看PPT及同学的帮助下解决了问题，完成了卷积网络、空洞卷积网络及残差卷积网络模型的实现，对于深度学习实验代码的编写也有了很大的提升。

## 七、参考文献

## 八、附录