《模式识别与机器学习A》实验报告

实验题目： 卷积神经网络实验

学号： 2021110683

姓名： 徐柯炎

# 实验报告内容

## 实验目的

采用任意一种课程中介绍过的或者其它卷积神经网络模型（如Lenet-5、AlexNet等）用于解决某种媒体类型的模式识别问题。

## 实验内容

1. 卷积神经网络可以基于现有框架如TensorFlow、Pytorch或者Mindspore等构建，也可以自行设计实现。
2. 数据集可以使用手写体数字图像标准数据集，也可以自行构建。预测问题可以包括分类或者回归等。实验工作还需要对激活函数的选择、dropout等技巧的使用做实验分析。必要时上网查找有关参考文献。
3. 用不同数据量，不同超参数，比较实验效果，并给出截图和分析

## 实验环境

Windows10; python3.9;PyCharm 2021.2.2

## 实验过程、结果及分析（包括代码截图、运行结果截图及必要的理论支撑等）

### 卷积神经网络的原理

卷积神经网络（Convolutional Neural Networks，简称CNN）是一种具有局部连接、权值共享等特点的深层前馈神经网络（Feedforward Neural Networks），是深度学习（deep learning）的代表算法之一，擅长处理图像特别是图像识别等相关机器学习问题，比如图像分类、目标检测、图像分割等各种视觉任务中都有显著的提升效果，是目前应用最广泛的模型之一。

卷积神经网络由以下几个部分构成：

卷积层（Convolutional Layer）：

卷积操作是CNN的核心部分，它通过滤波器（卷积核）对输入图像进行卷积运算，以提取特征。每个卷积操作都会产生一个特征映射（Feature Map），如下所示：

其中， 是特征映射的像素值， 是输入图像的像素值，是滤波器的权重。卷积操作是通过滑动滤波器在输入图像上进行的。

池化层（Pooling Layer）：

池化操作用于降低特征映射的空间维度，减少计算复杂度，并增强特征的不变性。最常见的池化操作是最大池化（Max Pooling）和平均池化（Average Pooling）。

全连接层（Fully Connected Layer）：

全连接层将卷积和池化层提取的特征映射连接起来，然后通过激活函数进行非线性变换，最终输出网络的预测结果。全连接层的公式如下：

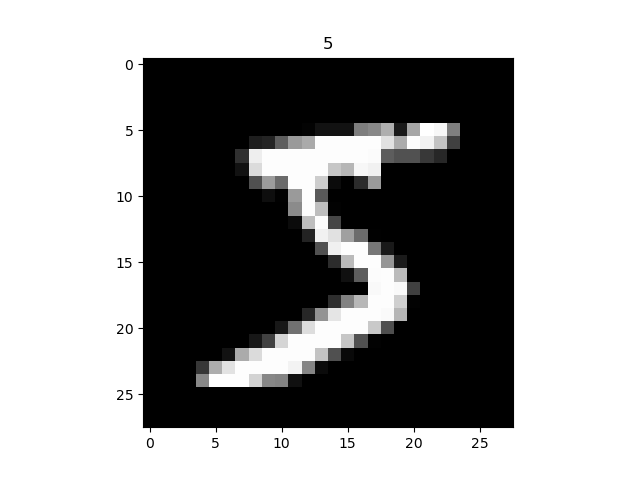
​

其中，是网络的第k个输出， 是全连接层的权重，是特征映射的像素值， 是偏置项，是激活函数，通常是ReLU（Rectified Linear Unit）或Sigmoid函数。

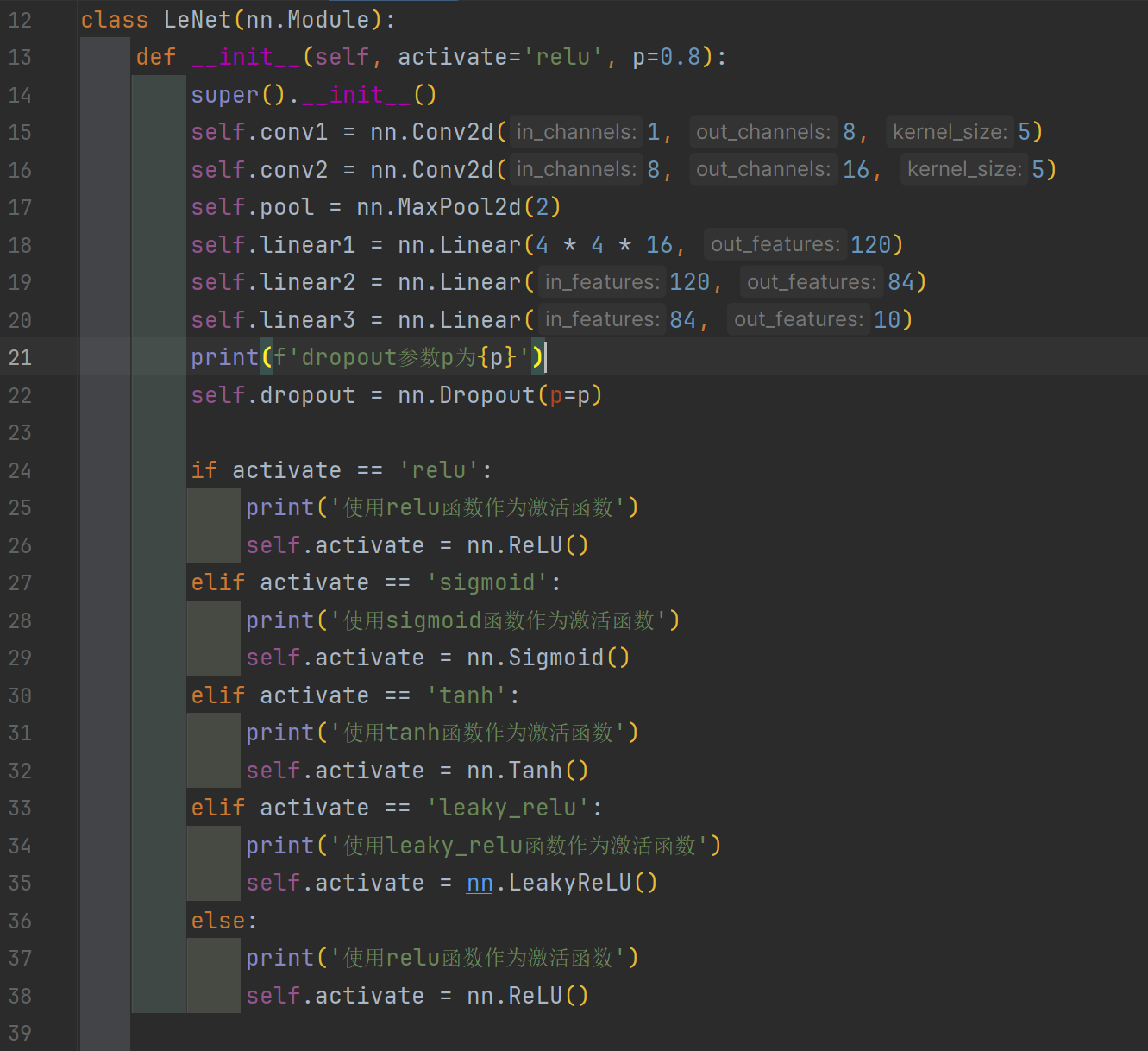
这些是卷积神经网络的基本原理，它们允许CNN自动学习图像中的特征，并在图像分类、目标检测和其他计算机视觉任务中取得出色的性能。

### 实验过程

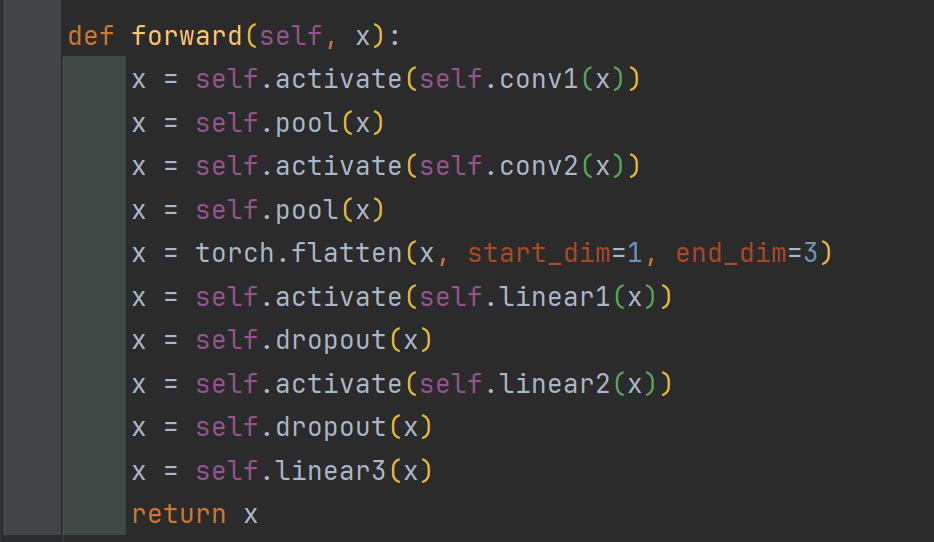
本实验采用pytorch库，数据集采用mnist数据集，mnist数据集的图像如下图所示：



本实验采用的神经网络为lenet模型，代码如下图所示：



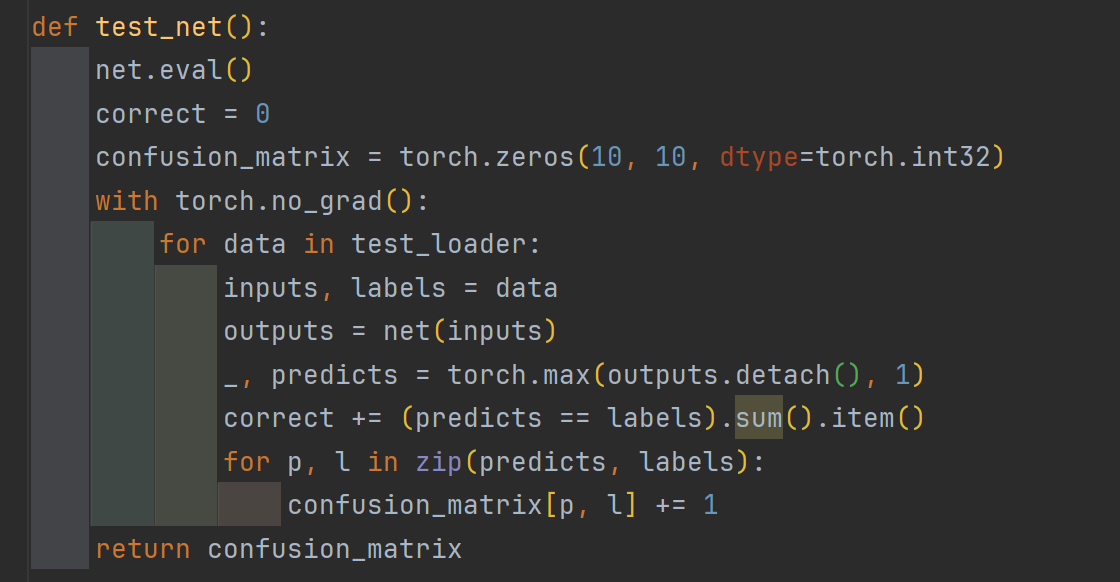
Lenet的前向传播函数如图所示：



接着用train函数来训练模型：

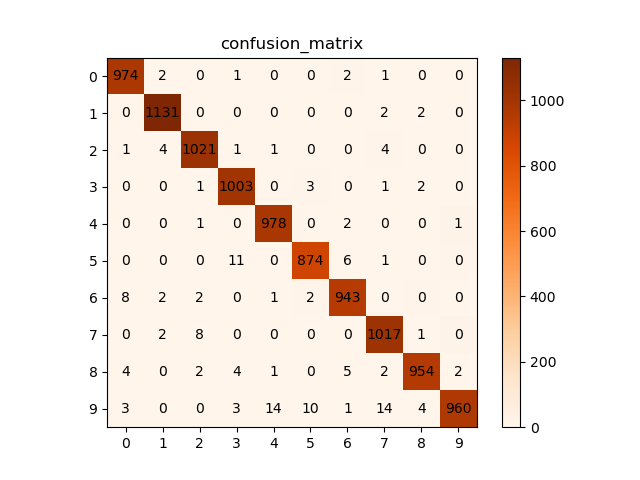


最后用test函数来测试模型：



### 实验测试

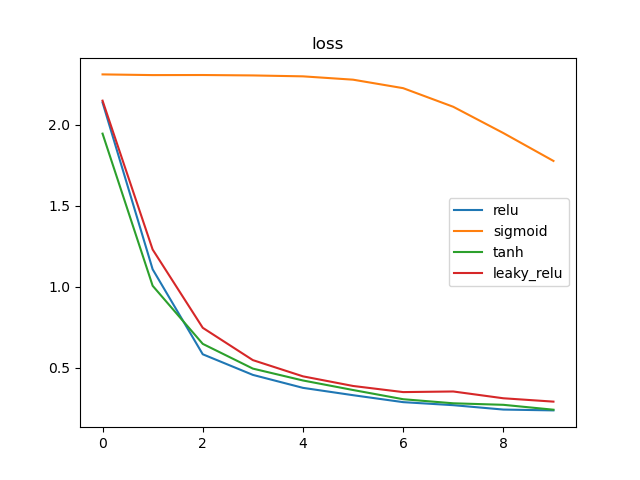
本实验从以下几个方面来测试：loss的下降速率、准确率（accuracy）、精确率（precision）、召回率（recall）、f1分数，训练用时，同时采用混淆矩阵绘制图像（由于太占空间这里就放一张，如下图所示）。



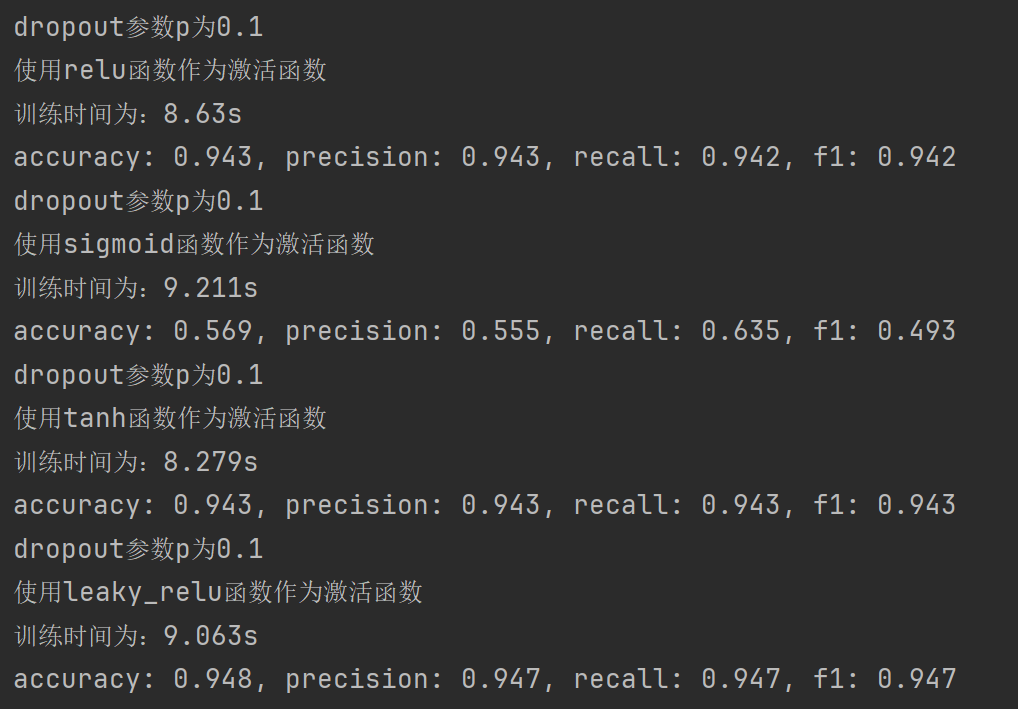
1. 激活函数的选择：

本实验对常用的激活函数进行测试，具体包括以下几个函数：sigmoid函数、relu函数，tanh函数、leaky relu函数。测试结果如下：

下图为loss下降的过程：



下图对模型进行评价：

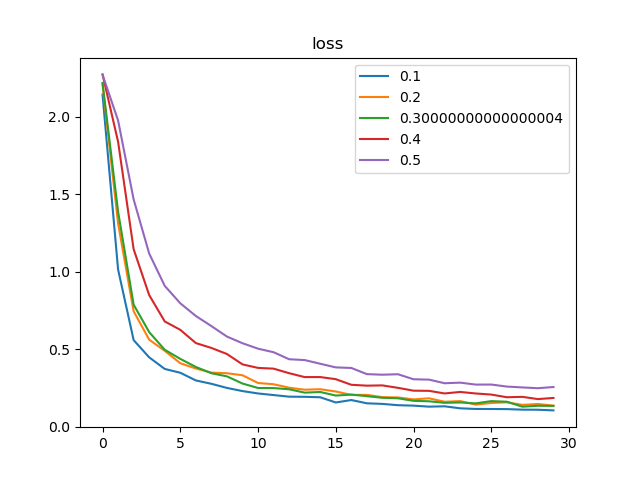


从上图看出，这几个函数相比而言sigmoid函数的训练效果最差，在所有指标上面都远差于其它几个函数。Relu和tanh收敛速度较快，但是从测试结果来看，效果要比leaky relu函数差一点。由于一般我们采用relu函数作为激活函数，因此下面的测试均采用relu函数。

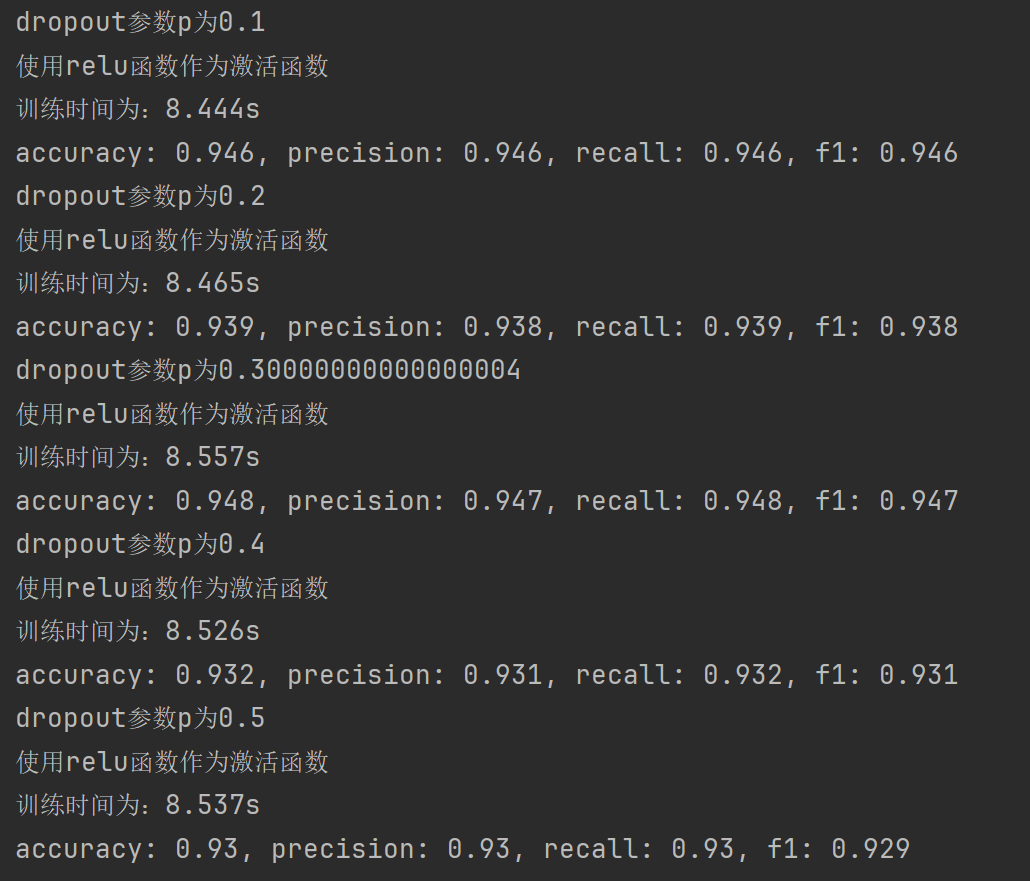
1. Dropout测试：

本实验对dropout的取值进行测试。测试结果如下：

下图为loss下降的过程：



下图对模型进行评价：

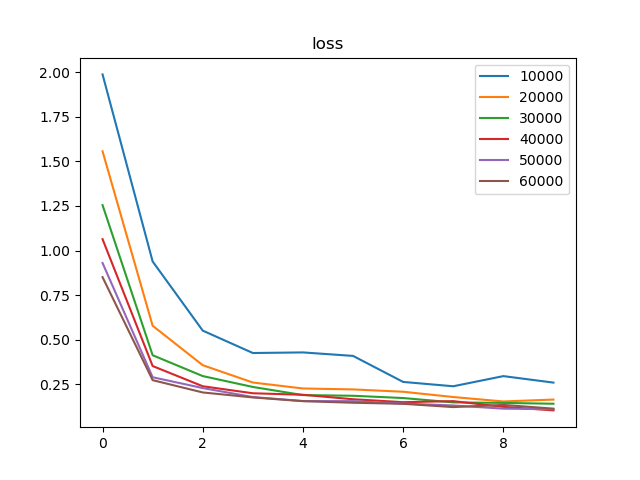


从上图看出，当dropout参数为0.3时，也就是说每次训练屏蔽30%的节点时，测试效果最好，尽管loss不是最低，这说明当loss下降到一定程度的时候会出现过拟合的现象，而dropout刚好缓解了这种现象，并且我们经过实验可知，dropout参数为0.3的时候效果最好。

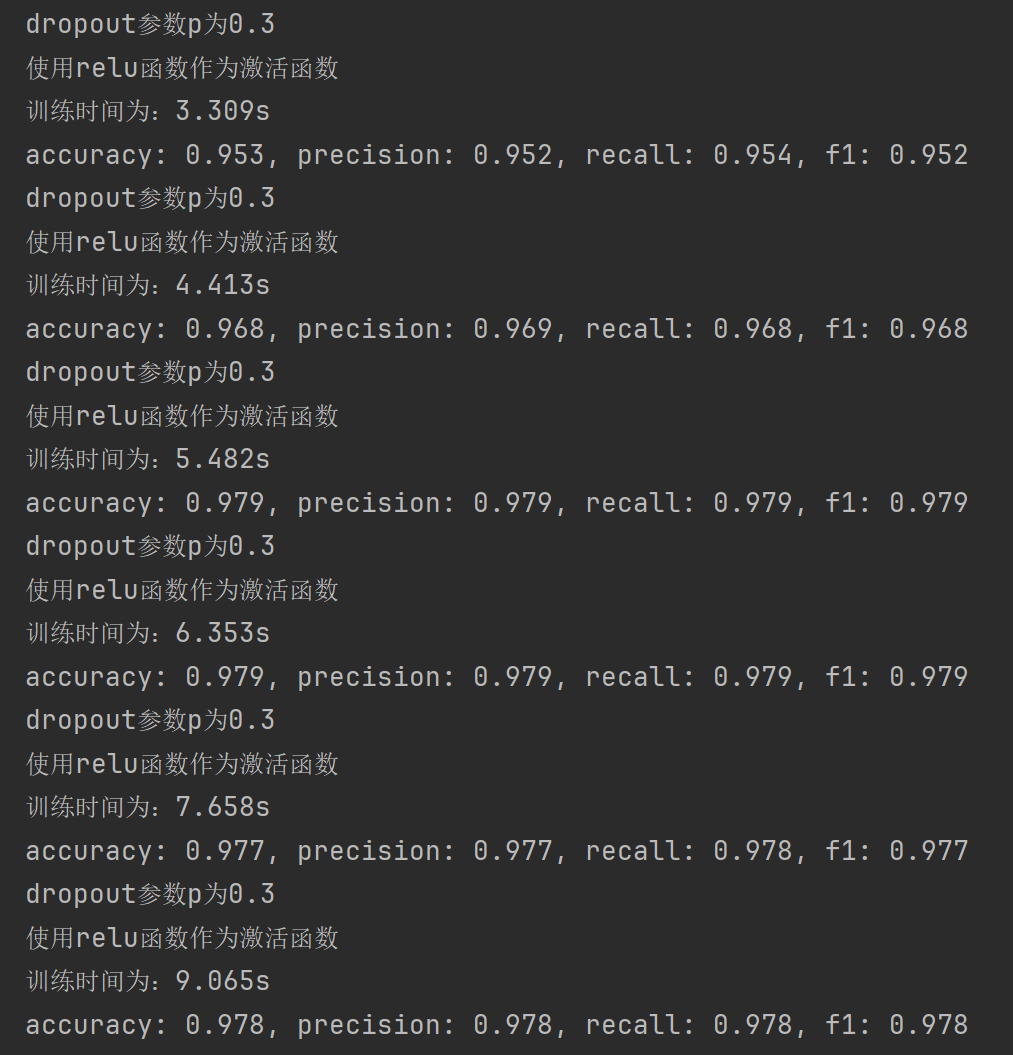
1. 样本数量测试：

本实验对训练轮数的取值进行测试。测试结果如下：

下图为loss下降的过程：



下图对模型进行评价：

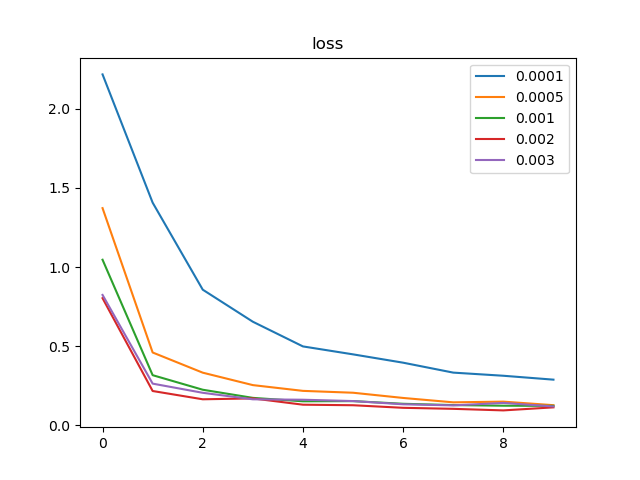


从上图看出，训练样本越多，训练时间越长，同时训练效果也越好，但是也需要防止数据过多时过拟合的发生，尽管在这里并没有出现。

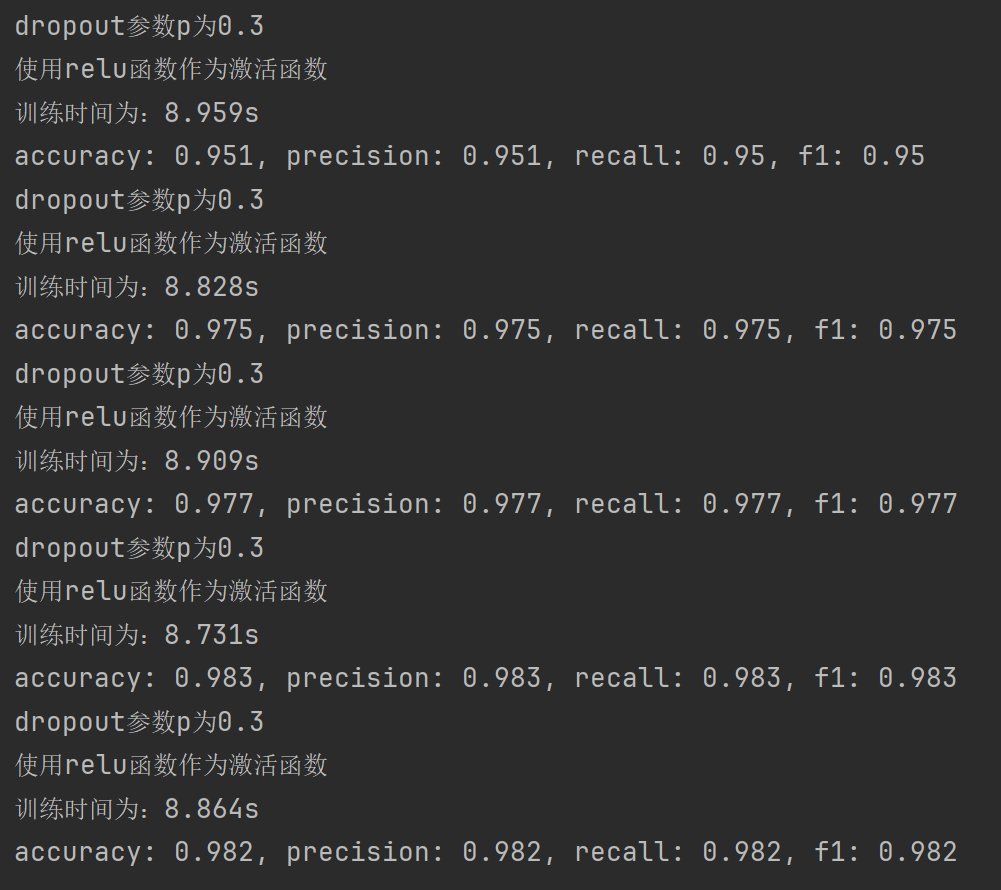
1. 初始学习率测试：

本实验对初始学习率的取值进行测试。测试结果如下：

下图为loss下降的过程：



下图对模型进行评价：

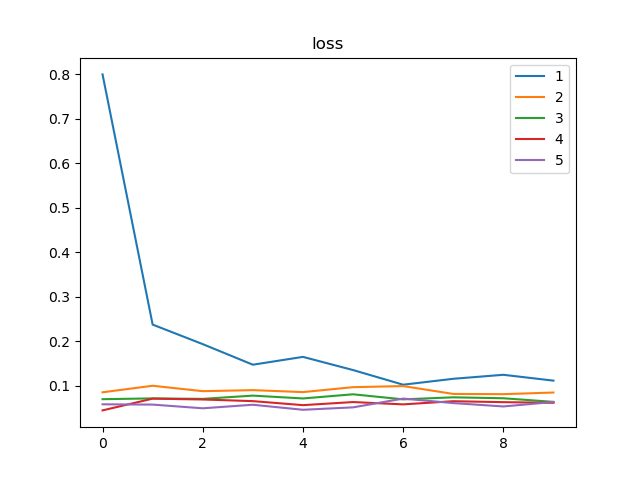


从上图可以看出，并不是学习率越高收敛速度越快，因为这里用的是adam算法，它会根据每个参数的变化情况针对性的调整学习率。在这里当学习率为0.002时收敛速度最快，同时测试效果最好。

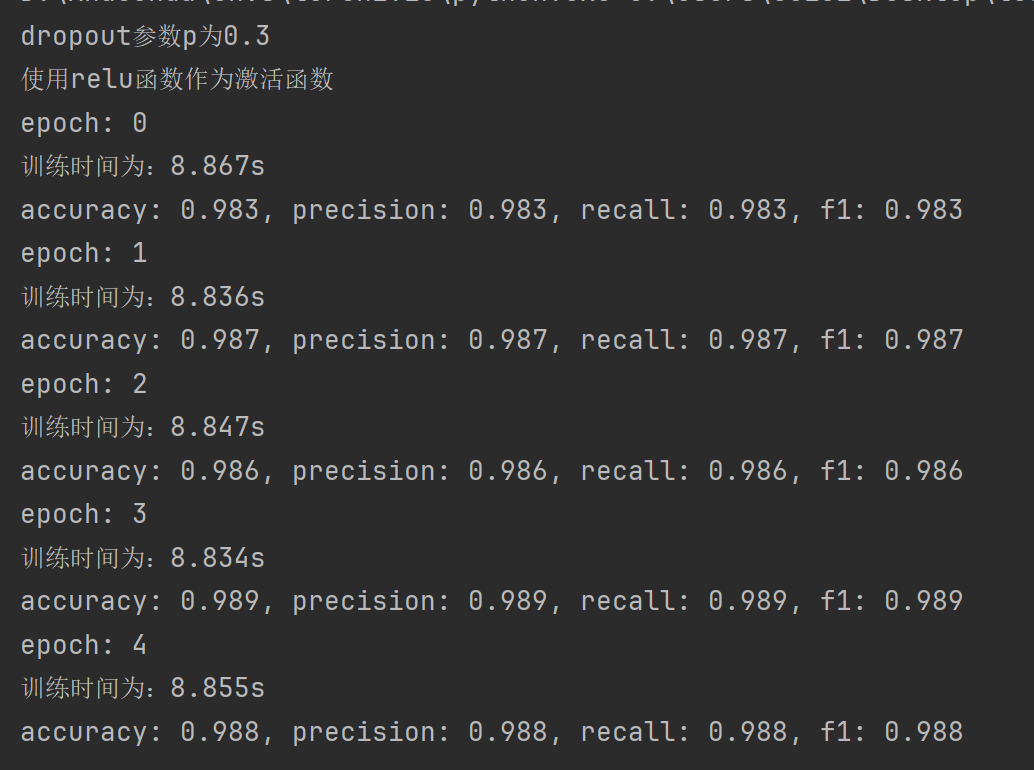
1. 训练轮数：

本实验对训练轮数的取值进行测试。测试结果如下：

下图为loss下降的过程：



下图对模型进行评价：



从上图可以看出，当训练轮数增加以后，loss非但没有下降反而还上升了一点，但测试效果比先前好了许多，这说明当loss太低的时候发生了过拟合的现象，但是随着训练轮数的增加，损失函数逐渐趋于收敛，此时再进行训练也意义不大了。因此在这里训练轮数为4时最好。

## 实验总体结论

1. 模型的激活函数不再使用sigmoid函数，而应该选取relu，tanh等函数。
2. 使用dropout防止过拟合：当loss下降到一定程度的时候会出现过拟合的现象，而dropout刚好缓解了这种现象。
3. 训练样本越多，训练时间越长，同时训练效果也越好，但是也需要防止数据过多时过拟合的发生，尽管在这里并没有出现。
4. 学习率越小，迭代次数越多，训练时间越长，训练效果越好；学习率偏大的时候容易出现震荡的现象，虽然用时短了，但是训练效果偏差。
5. 当训练轮数增加以后，loss非但没有下降反而还上升了一点，但测试效果比先前好了许多，这说明当loss太低的时候发生了过拟合的现象，但是随着训练轮数的增加，损失函数逐渐趋于收敛，此时再进行训练也意义不大了。

## 完整实验代码

main.py

1. **import** time
2. **import** matplotlib.pyplot as plt
3. **import** numpy as np
4. **import** torch
5. **import** torchvision
6. **from** torch.utils.data **import** DataLoader
7. **import** torchvision.transforms as transforms
8. **import** torch.optim as optim
9. **from** LeNet **import** LeNet
10. **from** torch **import** nn
12. batch\_size = 30
13. lr = 0.002
14. epoch = 5
15. activate\_list = ['relu', 'sigmoid', 'tanh', 'leaky\_relu']
16. activate\_test = False
17. dropout\_list = np.arange(0.1, 0.6, 0.1)
18. dropout\_test = False
19. lr\_list = [0.0001, 0.0005, 0.001, 0.002, 0.003]
20. lr\_test = False
21. bs\_list = np.arange(10, 100, 10)
22. bs\_test = False
23. num\_list = np.arange(10000, 60001, 10000)
24. num\_test = False

27. **def** test\_net():
28. net.eval()
29. correct = 0
30. confusion\_matrix = torch.zeros(10, 10, dtype=torch.int32)
31. with torch.no\_grad():
32. **for** data **in** test\_loader:
33. inputs, labels = data
34. outputs = net(inputs)
35. \_, predicts = torch.max(outputs.detach(), 1)
36. correct += (predicts == labels).sum().item()
37. **for** p, l **in** zip(predicts, labels):
38. confusion\_matrix[p, l] += 1
39. **return** confusion\_matrix

42. **def** train\_net():
43. **global** run\_loss, loss\_list
44. net.train()
45. **print**(f'epoch: {e}')
46. **for** i, data **in** enumerate(train\_loader):
47. inputs, labels = data
48. optimizer.zero\_grad()
49. outputs = net(inputs)
50. loss = criterion(outputs, labels)
51. loss.backward()
52. optimizer.step()
53. run\_loss += loss.item()
54. **if** i % print\_num == print\_num - 1:
55. run\_loss /= print\_num
56. loss\_list.append(run\_loss)
57. # print(f'[{e + 1}, {(i + 1) \* batch\_size}:{train\_num}] \tloss: {round(run\_loss, 5)}')
58. run\_loss = 0.0

61. **def** plot\_loss(loss\_lists, name):
62. plt.title("loss")
63. **for** loss\_list, label **in** zip(loss\_lists, name):
64. num = len(loss\_list)
65. x = np.arange(num)
66. plt.plot(x, loss\_list, label=label)
67. plt.legend()
68. plt.show()

71. **def** cal\_conmat(con\_mat):
72. p\_num = torch.sum(con\_mat, 0)
73. l\_num = torch.sum(con\_mat, 1)
74. tp = torch.diagonal(con\_mat)
76. precision = tp / p\_num
77. recall = tp / l\_num
78. accuracy = (torch.sum(tp) / torch.sum(con\_mat)).item()
79. f1 = 2 / (1 / precision + 1 / recall)
81. esti\_dic = {"accuracy": accuracy,
82. "precision": precision,
83. "recall": recall,
84. "f1": f1}
85. **return** esti\_dic

88. **def** plot\_conmat(con\_mat):
89. fig = plt.figure()
90. num = con\_mat.size(0)
91. plt.imshow(con\_mat, interpolation='nearest', cmap=plt.cm.Oranges)
92. plt.title('confusion\_matrix')
93. plt.colorbar()
94. tick\_marks = np.arange(num)
95. plt.xticks(tick\_marks, tick\_marks)
96. plt.yticks(tick\_marks, tick\_marks)
97. **for** i **in** range(num):
98. **for** j **in** range(num):
99. plt.text(i, j, format(con\_mat[i, j]), va='center', ha='center')
100. fig.show()

103. **def** print\_esti(estimate: dict):
104. accuracy = estimate['accuracy']
105. precision = torch.mean(estimate['precision']).item()
106. recall = torch.mean(estimate['recall']).item()
107. f1 = torch.mean(estimate['f1']).item()
108. **print**(f'accuracy: {round(accuracy, 3)}, precision: {round(precision, 3)}, '
109. f'recall: {round(recall, 3)}, f1: {round(f1, 3)}')

112. **if** \_\_name\_\_ == '\_\_main\_\_':
113. transform = transforms.Compose([transforms.ToTensor(),
114. transforms.Normalize(0.5, 0.5)])
115. train\_data = torchvision.datasets.MNIST('mnist-data', train=True, transform=transform)
116. test\_data = torchvision.datasets.MNIST('mnist-data', train=False, transform=transform)
117. train\_loader = DataLoader(dataset=train\_data, batch\_size=batch\_size, shuffle=True, num\_workers=2)
118. test\_loader = DataLoader(dataset=test\_data, batch\_size=batch\_size, shuffle=True, num\_workers=2)
119. # print(type(train\_data[0][1]))
120. # print(train\_data)
121. # print(test\_data.data.size())
122. train\_num = len(train\_data)
123. test\_num = len(test\_data)
124. print\_num = int(train\_num / (batch\_size \* 10))
126. **if** activate\_test:
127. loss\_lists = []
128. **for** act **in** activate\_list:
129. net = LeNet(activate=act)
130. optimizer = optim.Adam(net.parameters(), lr=lr)
132. criterion = nn.CrossEntropyLoss()
133. run\_loss = 0.0
134. loss\_list = []
135. **for** e **in** range(epoch):
136. start = time.time()
137. train\_net()
138. end = time.time()
139. **print**(f'训练时间为：{round(end - start, 3)}s')
140. con\_matrix = test\_net()
141. loss\_lists.append(loss\_list)
142. estimate = cal\_conmat(con\_matrix)
143. print\_esti(estimate)
144. plot\_conmat(con\_matrix)
145. plot\_loss(loss\_lists, activate\_list)
147. **elif** dropout\_test:
148. loss\_lists = []
149. **for** dropout **in** dropout\_list:
150. net = LeNet(p=dropout)
151. optimizer = optim.Adam(net.parameters(), lr=lr)
153. criterion = nn.CrossEntropyLoss()
154. run\_loss = 0.0
155. loss\_list = []
156. **for** e **in** range(epoch):
157. start = time.time()
158. train\_net()
159. end = time.time()
160. **print**(f'训练时间为：{round(end - start, 3)}s')
161. con\_matrix = test\_net()
162. loss\_lists.append(loss\_list)
163. estimate = cal\_conmat(con\_matrix)
164. print\_esti(estimate)
165. plot\_conmat(con\_matrix)
166. plot\_loss(loss\_lists, dropout\_list)
168. **elif** lr\_test:
169. loss\_lists = []
170. **for** lr **in** lr\_list:
171. net = LeNet()
172. optimizer = optim.Adam(net.parameters(), lr=lr)
174. criterion = nn.CrossEntropyLoss()
175. run\_loss = 0.0
176. loss\_list = []
177. **for** e **in** range(epoch):
178. start = time.time()
179. train\_net()
180. end = time.time()
181. **print**(f'训练时间为：{round(end - start, 3)}s')
182. con\_matrix = test\_net()
183. loss\_lists.append(loss\_list)
184. estimate = cal\_conmat(con\_matrix)
185. print\_esti(estimate)
186. plot\_conmat(con\_matrix)
187. plot\_loss(loss\_lists, lr\_list)
189. **elif** num\_test:
190. loss\_lists = []
191. **for** num **in** num\_list:
192. train\_data = torchvision.datasets.MNIST('mnist-data', train=True, transform=transform)
193. train\_data.data = train\_data.data[:num]
194. train\_data.targets = train\_data.targets[:num]
195. train\_loader = DataLoader(dataset=train\_data, batch\_size=batch\_size, shuffle=True, num\_workers=2)
196. train\_num = len(train\_data)
197. test\_num = len(test\_data)
198. print\_num = int(train\_num / (batch\_size \* 10))
199. net = LeNet()
200. optimizer = optim.Adam(net.parameters(), lr=lr)
202. criterion = nn.CrossEntropyLoss()
203. run\_loss = 0.0
204. loss\_list = []
205. **for** e **in** range(epoch):
206. start = time.time()
207. train\_net()
208. end = time.time()
209. **print**(f'训练时间为：{round(end - start, 3)}s')
210. con\_matrix = test\_net()
211. loss\_lists.append(loss\_list)
212. estimate = cal\_conmat(con\_matrix)
213. print\_esti(estimate)
214. plot\_conmat(con\_matrix)
215. plot\_loss(loss\_lists, num\_list)
217. **else**:
218. loss\_lists = []
219. net = LeNet()
220. optimizer = optim.Adam(net.parameters(), lr=lr)
221. criterion = nn.CrossEntropyLoss()
222. run\_loss = 0.0
223. **for** e **in** range(epoch):
224. loss\_list = []
225. start = time.time()
226. train\_net()
227. end = time.time()
228. **print**(f'训练时间为：{round(end - start, 3)}s')
229. con\_matrix = test\_net()
230. loss\_lists.append(loss\_list)
231. estimate = cal\_conmat(con\_matrix)
232. print\_esti(estimate)
233. plot\_conmat(con\_matrix)
234. plot\_loss(loss\_lists, np.arange(1, epoch+1))

Lenet.py

1. # --\*-- coding:utf-8 --\*--
2. """
3. @Filename: LeNet.py
4. @Author: Keyan Xu
5. @Time: 2023-10-30
6. """
7. **import** torch
8. **from** torch **import** nn
9. **import** torch.nn.functional as F

12. **class** LeNet(nn.Module):
13. **def** \_\_init\_\_(self, activate='relu', p=0.3):
14. super().\_\_init\_\_()
15. self.conv1 = nn.Conv2d(1, 8, 5)
16. self.conv2 = nn.Conv2d(8, 16, 5)
17. self.pool = nn.MaxPool2d(2)
18. self.linear1 = nn.Linear(4 \* 4 \* 16, 120)
19. self.linear2 = nn.Linear(120, 84)
20. self.linear3 = nn.Linear(84, 10)
21. **print**(f'dropout参数p为{p}')
22. self.dropout = nn.Dropout(p=p)
24. **if** activate == 'relu':
25. **print**('使用relu函数作为激活函数')
26. self.activate = nn.ReLU()
27. **elif** activate == 'sigmoid':
28. **print**('使用sigmoid函数作为激活函数')
29. self.activate = nn.Sigmoid()
30. **elif** activate == 'tanh':
31. **print**('使用tanh函数作为激活函数')
32. self.activate = nn.Tanh()
33. **elif** activate == 'leaky\_relu':
34. **print**('使用leaky\_relu函数作为激活函数')
35. self.activate = nn.LeakyReLU()
36. **else**:
37. **print**('使用relu函数作为激活函数')
38. self.activate = nn.ReLU()
40. **def** forward(self, x):
41. x = self.activate(self.conv1(x))
42. x = self.pool(x)
43. x = self.activate(self.conv2(x))
44. x = self.pool(x)
45. x = torch.flatten(x, start\_dim=1, end\_dim=3)
46. x = self.activate(self.linear1(x))
47. x = self.dropout(x)
48. x = self.activate(self.linear2(x))
49. x = self.dropout(x)
50. x = self.linear3(x)
51. **return** x

## 参考文献

无