

计算机视觉

课程实验报告

|  |  |
| --- | --- |
| 院系 | 计算机科学与技术 |
| 专业班级 | 本硕博2101 |
| 姓名 | 姚晨炫 |
| 学号 | U202115674 |
| 指导老师 | 杨卫 |

目录

[1 网络架构 3](#_Toc152925588)

[2 数据集 4](#_Toc152925589)

[3 训练 5](#_Toc152925590)

[4 实验 6](#_Toc152925591)

[5 探究dropout的作用 7](#_Toc152925592)

# 网络架构

常规的识别手写数字的网络ResNet由两个卷积层和末尾的全连接层构成，如下图所示

图示

描述已自动生成

我尝试在网络中加入了两个dropout层来减轻网络的过拟合，加强网络的泛化能力

class Net(nn.Module):  
 def \_\_init\_\_(self):  
 super(Net, self).\_\_init\_\_()  
 self.conv1 = nn.Conv2d(1, 10, kernel\_size=5)   
 self.conv2 = nn.Conv2d(10, 20, kernel\_size=5)   
 self.conv2\_drop = nn.Dropout2d()  
 self.fc1 = nn.Linear(320, 50)   
 self.fc2 = nn.Linear(50, 10)   
  
 def forward(self, x):  
 x = F.relu(F.max\_pool2d(self.conv1(x), 2))   
 x = F.relu(F.max\_pool2d(self.conv2\_drop(self.conv2(x)), 2))   
 x = x.view(x.size(0), -1)   
 x = F.relu(self.fc1(x))   
 x = F.dropout(x, training=self.training)  
 x = self.fc2(x)  
 return F.softmax(x, dim=1)   
  
network = Net()

损失函数使用交叉熵，优化器采用SGD

#梯度优化方式采用SGD，带有momentum  
optimizer = optim.SGD(network.parameters(), lr=learning\_rate, momentum=momentum)  
loss\_f = nn.CrossEntropyLoss() #使用交叉熵损失函数

# 数据集

采用torch自带的minist数据集合，导入方法如下

#设置训练集和测试集  
train\_loader = torch.utils.data.DataLoader(  
 torchvision.datasets.MNIST('./data/', train=True, download=True,  
 transform=torchvision.transforms.ToTensor(),),  
 batch\_size=batch\_size, shuffle=True)  
  
test\_loader = torch.utils.data.DataLoader(  
 torchvision.datasets.MNIST('./data/', train=False, download=True,  
 transform=torchvision.transforms.ToTensor(),),  
 batch\_size=batch\_size, shuffle=True)

# 训练

采用标准的训练方式，代码如下，测试代码与训练相似

#训练模型的函数  
def train(epoch):  
 loop = tqdm(train\_loader, leave=True)  
 loss\_all = 0  
 network.train() #表示进入了测试模式  
 for batch\_idx, (data, target) in enumerate(loop):  
 optimizer.zero\_grad() #旧的梯度清零  
 output = network(data) #正向传播  
 loss = loss\_f(output, target) #计算损失  
 loss.backward() #计算梯度  
 optimizer.step() #梯度下降  
 loss\_all += loss #计算dataset的总loss  
 loop.set\_postfix(loss=loss.item()) #画图相关  
  
 #每隔一段时间记录下当前的loss以便画图  
 if batch\_idx % log\_interval == 0:  
 train\_losses.append(loss.item())  
 train\_counter.append((batch\_idx \* 64) + ((epoch - 1) \* len(train\_loader.dataset)))  
   
#测试模型的函数  
def test(epoch):  
 network.eval() #表明进入测试模式  
 test\_loss = 0 #测试的总损失  
 correct = 0 #测试的总正确数量  
 with torch.no\_grad(): #在测试的时候需要取消模型的梯度  
 for data, target in test\_loader:  
 output = network(data)  
 test\_loss += loss\_f(output, target)  
 pred = output.data.max(1, keepdim=True)[1]  
 correct += pred.eq(target.data.view\_as(pred)).sum()  
   
 test\_loss /= (len(test\_loader.dataset)/batch\_size)   
 test\_losses.append(test\_loss.item())  
 print('Epoch: {},Test set: Avg. loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(epoch,  
 test\_loss, correct, len(test\_loader.dataset),  
 100. \* correct / len(test\_loader.dataset)))

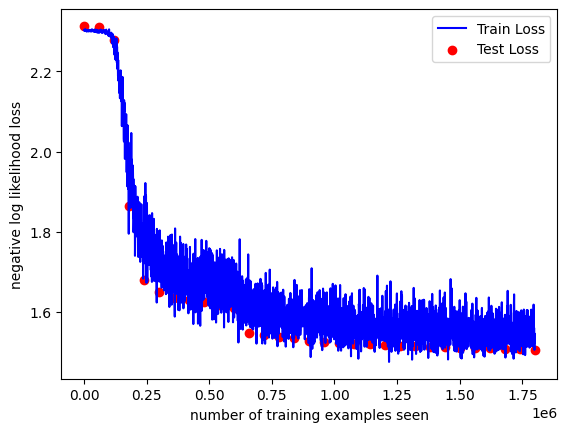
# 实验

实验环境为MacOS Apple Silicon M1 Pro，Python版本3.9，Pytorch版本2.0.0(MacOS版)

相关实验setting如下

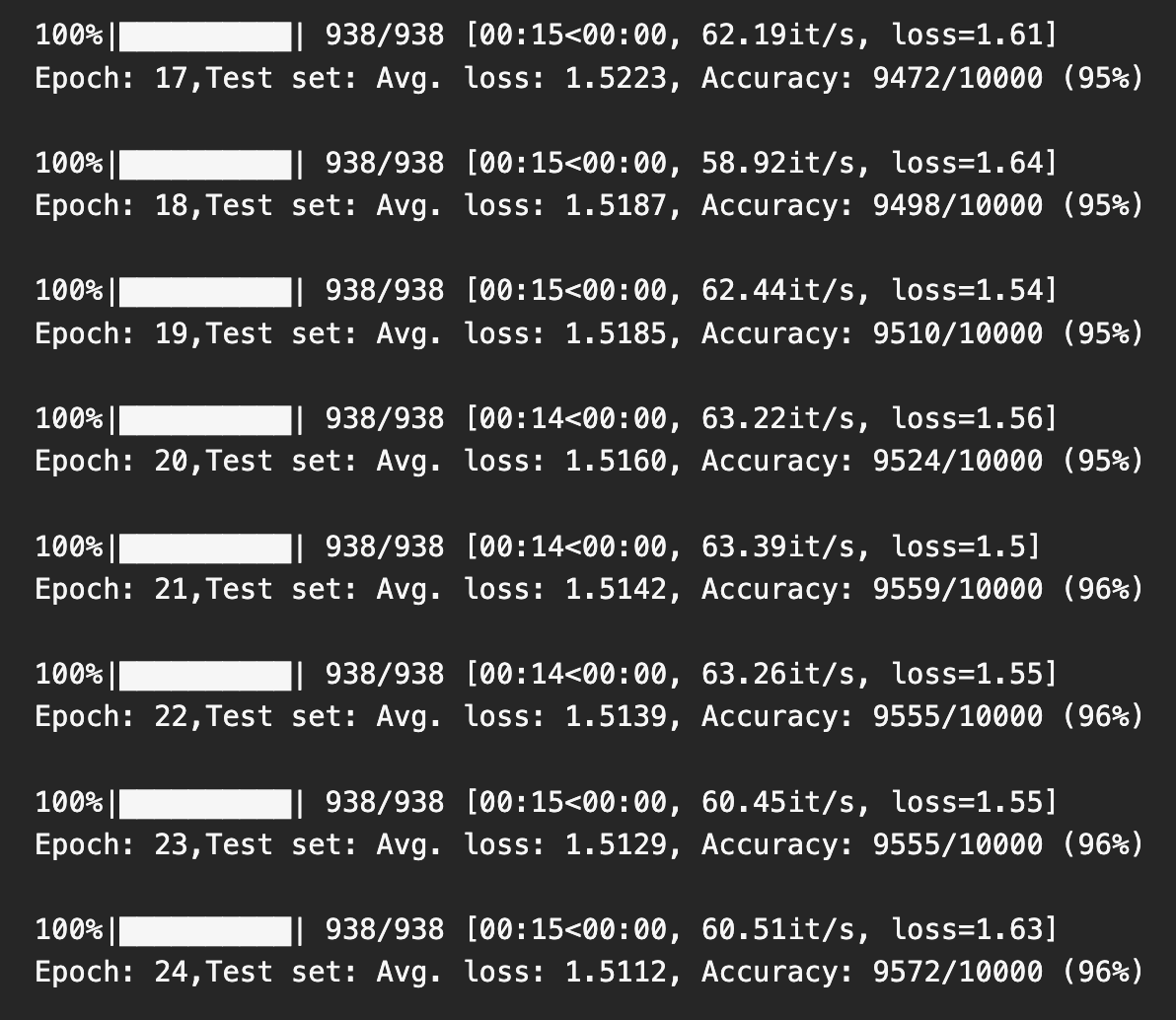
#相关参数  
n\_epochs = 30 #训练的轮次  
batch\_size = 64 #batch\_size  
learning\_rate = 0.01 #学习率  
momentum = 0.5 #momentum梯度下降时的超参数  
log\_interval = 10 #画图的记录点

根据代码中的绘图函数，我画出了梯度下降过程中的损失和准确率变化。



实验部分采用tqdm做了一个简单的进度条，刚开始的时候模型准确率为10%，跟乱猜没啥区别

大约在23个epoch左右的时候模型收敛，准确率为96%



# 探究dropout的作用

在机器学习的模型中，如果模型的参数太多，而训练样本又太少，训练出来的模型很容易产生过拟合的现象。在训练神经网络的时候经常会遇到过拟合的问题，过拟合具体表现在：模型在训练数据上损失函数较小，预测准确率较高；但是在测试数据上损失函数比较大，预测准确率较低。

过拟合是很多机器学习的通病。如果模型过拟合，那么得到的模型几乎不能用。为了解决过拟合问题，一般会采用模型集成的方法，即训练多个模型进行组合。此时，训练模型费时就成为一个很大的问题，不仅训练多个模型费时，测试多个模型也是很费时。

Dropout可以比较有效的缓解过拟合的发生，在一定程度上达到正则化的效果。具体如下

1. **取平均的作用：** 先回到标准的模型即没有dropout，我们用相同的训练数据去训练5个不同的神经网络，一般会得到5个不同的结果，此时我们可以采用 “5个结果取均值”或者“多数取胜的投票策略”去决定最终结果。例如3个网络判断结果为数字9,那么很有可能真正的结果就是数字9，其它两个网络给出了错误结果。这种“综合起来取平均”的策略通常可以有效防止过拟合问题。因为不同的网络可能产生不同的过拟合，取平均则有可能让一些“相反的”拟合互相抵消。dropout掉不同的隐藏神经元就类似在训练不同的网络，随机删掉一半隐藏神经元导致网络结构已经不同，整个dropout过程就相当于对很多个不同的神经网络取平均。而不同的网络产生不同的过拟合，一些互为“反向”的拟合相互抵消就可以达到整体上减少过拟合。
2. **减少神经元之间复杂的共适应关系：** 因为dropout程序导致两个神经元不一定每次都在一个dropout网络中出现。这样权值的更新不再依赖于有固定关系的隐含节点的共同作用，阻止了某些特征仅仅在其它特定特征下才有效果的情况 。迫使网络去学习更加鲁棒的特征 ，这些特征在其它的神经元的随机子集中也存在。换句话说假如我们的神经网络是在做出某种预测，它不应该对一些特定的线索片段太过敏感，即使丢失特定的线索，它也应该可以从众多其它线索中学习一些共同的特征。从这个角度看dropout就有点像L1，L2正则，减少权重使得网络对丢失特定神经元连接的鲁棒性提高。

我设置了一个没有dropout的网络

class Net2(nn.Module):  
 def \_\_init\_\_(self):  
 super(Net2, self).\_\_init\_\_()  
 self.conv1 = nn.Conv2d(1, 10, kernel\_size=5)   
 self.conv2 = nn.Conv2d(10, 20, kernel\_size=5)   
 self.fc1 = nn.Linear(320, 50)   
 self.fc2 = nn.Linear(50, 10)   
  
 def forward(self, x):  
 x = F.relu(F.max\_pool2d(self.conv1(x), 2))   
 x = F.relu(F.max\_pool2d(self.conv2(x), 2))   
 x = x.view(x.size(0), -1)   
 x = F.relu(self.fc1(x))   
 x = self.fc2(x)  
 return F.softmax(x, dim=1)   
  
network = Net2()

参数与上文一致，训练结果如下

在相同epoch时，（左图为dropout，右图为无dropout）

文本

中度可信度描述已自动生成文本

描述已自动生成

总体loss下降图,最后的准确率为90%

图表, 直方图

描述已自动生成

可以看到，训练loss的下降速度和精度都与使用了dropout有较大的区别，在测试集的表现不如加了dropout的版本