# 《人工智能》实验四报告

|  |  |  |  |
| --- | --- | --- | --- |
| 姓名 | 孙潇桐 | 学号 | 2021117405 |
| 实验地点 | 机房321 | 实验日期 | 2023-11-06 |

**一、实验内容**

# Build a backpropngntion network in a language of your choice and run it on MNIST

Dataset。MNIST Dataset can be download from [***http://yann.lecun.com/exdb/mnist/.***](http://yann.lecun.com/exdb/mnist/.)

**（使用你喜欢的语言实现BP神经网络，并在MNIST手写数据集上进行训练和测试）**

1. **实验原理**

我在学习了神经网络之后，就对多分类问题有了更加清晰的认识，简单的全连接神经网络就能完成一些多分类问题，这次的实验正好也是个多分类问题。我选用了当下最热门且最成熟的深度学习框架 PyTorch，因为这个框架支持我电脑上的英伟达**显卡加速**。

BP神经网络（Backpropagation Neural Network）：一种常见的人工神经网络模型，用于解决分类和回归问题。它由输入层、隐藏层和输出层组成，其中**隐藏层和输出层之间存在权重连接**。BP神经网络的训练过程通过**反向传播算法来更新网络的权重，以最小化预测输出与目标输出之间的差距**。

激活函数（Activation Function）：神经网络中的一种非线性函数，它被应用于神经元的输出，将输入信号转换为神经元的激活状态。激活函数引入非线性特性，使得神经网络能够学习和表示更加复杂的函数关系。**在这里，我使用ReLU函数。**

前向传播：在前向传播过程中，将输入数据通过神经网络，逐层计算并传递给下一层。每个节点都与上一层的所有节点连接，并根据连接权重进行加权求和。对于每个节点，使用**激活函数**对加权求和结果进行非线性转换，得到该节点的输出值。重复以上步骤，直到达到输出层，得到网络的输出结果。**我有一层输入层，两层隐藏层，一层输出层，下面是代码：**

class Net(nn.Module):  
 def \_\_init\_\_(self):  
 super(Net, self).\_\_init\_\_()  
 self.flatten = nn.Flatten() *# 展平图像* self.fc1 = nn.Linear(28 \* 28, 45)  
 self.relu1 = nn.ReLU()   
 self.fc2 = nn.Linear(45, 20) *# 隐藏层1* self.relu2 = nn.ReLU()  
 self.out = nn.Linear(20, 10) *# 隐藏层2*  
  
 def forward(self, x): *# 前向传播过程*  
  
 x = self.flatten(x)  
 x = self.fc1(x)  
 x = self.relu1(x)  
 x = self.fc2(x)  
 x = self.relu2(x)  
 x = self.out(x)  
 return x

损失函数（Loss Function）：于衡量神经网络预测输出与目标输出之间的差距。它是神经网络训练过程中的目标函数，通过**最小化**损失函数来调整网络的权重和偏置，以提高预测的准确性。**在这里，我使用交叉熵损失函数。**

反向传播算法（Backpropagation Algorithm）：用于训练神经网络的一种常见算法。它通过计算损失函数对网络权重的梯度，然后沿着梯度的反方向更新权重，从而实现网络的训练。反向传播算法基于**链式法则**，通过逐层计算梯度并反向传播，将输出误差分配给各层的权重，从而实现误差的反向传播和权重的更新。**在这里我使用torch损失函数提供的loss.backward()进行反向传播。**

梯度下降法（Gradient Descent）：一种优化算法，用于最小化损失函数。在反向传播算法中，梯度下降法被用来更新神经网络的权重和偏置，以逐步降低损失函数的值。梯度下降法的基本思想是通过沿着梯度的反方向迭代地调整参数，使得损失函数逐渐减小，达到最优解。**在这里，我使用optimizer.step()来进行梯度下降。**

完整代码：

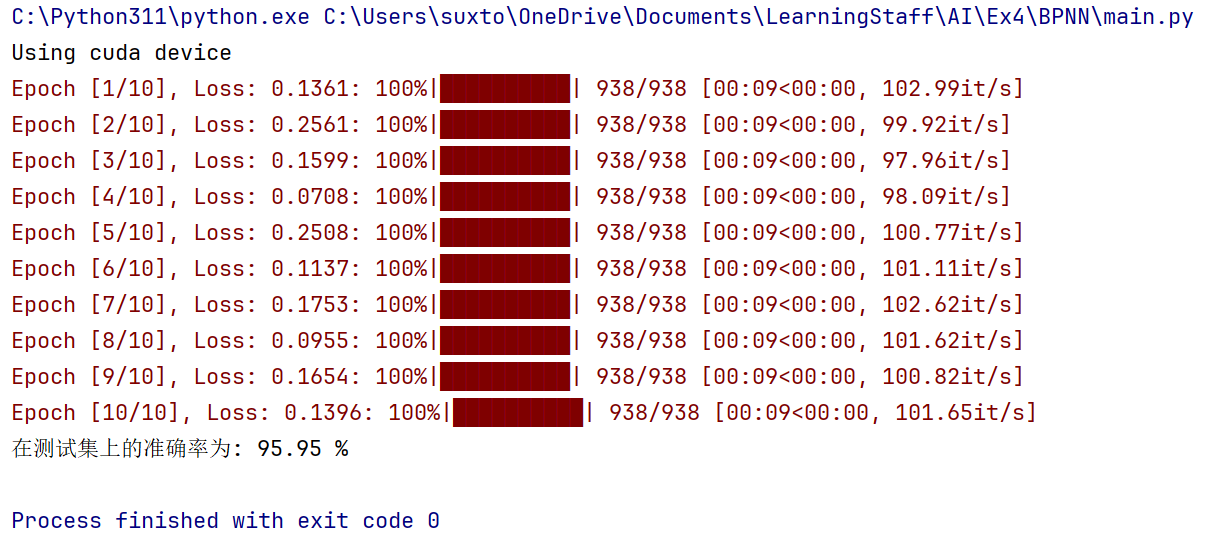
import torch  
import torch.nn as nn  
import torch.optim as optim  
import torchvision  
import torchvision.transforms as transforms  
from torch.utils.data import DataLoader  
  
from tqdm import tqdm  
  
device = (  
 "cuda"  
 if torch.cuda.is\_available()  
 else "cpu"  
)  
print(f"Using {device} device")  
  
*# 超参数*batch\_size = 64  
num\_epochs = 10  
learning\_rate = 0.001  
train = True  
  
*# 加载MNIST数据集并进行预处理*transform = transforms.Compose([  
 transforms.ToTensor(),  
 transforms.Normalize((0.5,), (0.5,))  
])  
  
train\_dataset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)  
test\_dataset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)  
  
train\_loader = torch.utils.data.DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)  
test\_loader = torch.utils.data.DataLoader(test\_dataset, batch\_size=batch\_size, shuffle=False)  
  
  
class Net(nn.Module):  
 def \_\_init\_\_(self):  
 super(Net, self).\_\_init\_\_()  
 self.flatten = nn.Flatten() *# 展平图像* self.fc1 = nn.Linear(28 \* 28, 45)  
 self.relu1 = nn.ReLU()  
 self.fc2 = nn.Linear(45, 20)  
 self.relu2 = nn.ReLU()  
 self.out = nn.Linear(20, 10)  
  
 def forward(self, x):  
 x = self.flatten(x)  
 x = self.fc1(x)  
 x = self.relu1(x)  
 x = self.fc2(x)  
 x = self.relu2(x)  
 x = self.out(x)  
 return x  
  
  
*# 实例化模型，并将模型移动到显卡*model = Net().to(device)  
  
*# 定义损失函数和优化器*criterion = nn.CrossEntropyLoss()  
optimizer = optim.Adam(model.parameters(), lr=learning\_rate)  
  
data\_tier = DataLoader(train\_loader.dataset, batch\_size=train\_loader.batch\_size, shuffle=True)  
if train: *# 训练模型* total\_step = len(data\_tier)  
 for epoch in range(num\_epochs):  
 progress\_bar = tqdm(data\_tier, total=total\_step) *# 使用 tqdm 创建进度条* for i, (images, labels) in enumerate(progress\_bar):  
 images = images.to(device)  
 labels = labels.to(device)  
 model.zero\_grad()  
 *# 向前传播* outputs = model(images)  
 loss = criterion(outputs, labels)  
 *# 反向传播和优化* optimizer.zero\_grad()  
 loss.backward()  
 optimizer.step()  
  
 if (i + 1) % 100 == 0:  
 progress\_bar.set\_description('Epoch [{}/{}], Loss: {:.4f}'  
 .format(epoch + 1, num\_epochs, loss.item()))  
 *# 保存模型* torch.save(model.state\_dict(), 'model.pth')  
  
if not train: *# 如果不训练的话，载入模型* model = Net()  
 model.load\_state\_dict(torch.load('model.pth'))  
  
*# 在测试集上评估模型*model.eval()  
with torch.no\_grad(): *# 关闭模型的反向传播过程* correct = 0  
 total = 0  
 for images, labels in test\_loader:  
 images = images.to(device)  
 labels = labels.to(device)  
  
 outputs = model(images)  
 *# outputs = torch.softmax(outputs, dim=1) # 使用 softmax 函数优化输出* \_, predicted = torch.max(outputs.data, 1)  
 total += labels.size(0)  
 correct += (predicted == labels).sum().item()  
 print('在测试集上的准确率为: {} %'.format(100 \* correct / total))

1. **实验过程以及结果分析**

完成上面的编码之后，还需要设置超参数。我的超参数是：

分成64个batch，跑10个循环，学习率为0.001

直接运行代码，跑10个循环，最后得到模型的**准确率是95.95%**，感觉已经到**全连接网络的极限**了，我问了几个同学，基本上没有超过96%的。如果要更进一步，应该要使用CNN优化了。



1. **实验总结**

实验结果显示，经过10个循环的训练，该模型在MNIST测试数据集上达到了95.95%的准确率。这是一个相当不错的结果，说明BP神经网络在解决MNIST手写数字识别问题上具有一定的效果。

总的来说，通过这个实验，我进一步加深了对BP神经网络及其训练过程的理解。我也学会了如何使用pytorch构建和训练神经网络模型，并在MNIST数据集上进行评估。这为我在后续的深度学习项目中提供了一个良好的基础。然而，我也意识到在处理更复杂的图像数据集时，可能需要采用更高级的技术，如卷积神经网络（CNN），以进一步提高准确性和性能。

经过这次实验，我使用了pytorch框架，也学会了如何训练和评价一个模型，以及调整模型的参数。这个框架是真的方便，有了这个东西之后，我相信以后的人工智能实验也能轻松完成。