

实验2：神经网络进行手写数字识别

郭慕凡 201250205

实验2：神经网络进行手写数字识别

过程说明：

1. 加载数据集
2. 设计网络模型
3. 训练
 - 3.1 通过试验设置超参数：epoch, batch_size, learning_rate
 - 3.1.1 epoch_size的设定：
 - 3.1.2 learning_rate的设定
 - 3.2 选取loss和优化器
 - 3.3 训练
 - 3.4 保存参数文件，待预测部分代码使用
4. 测试
 - 4.1 加载测试数据集
 - 4.2 加载训练好的参数文件
 - 4.3 遍历，累加预测正确的数量，计算正确率

结果

指标1：正确率：99%+，测试中最佳可以达到99.5%左右

指标2：运行时间：

过程说明：

1. 加载数据集

利用torchvision的API进行MINIST数据集的下载及预处理，预处理的代码如下：

```
train_dataset = datasets.MNIST(root='./data', train=True,
                                transform=transforms.ToTensor(), download=True)
```

将下载数据存放在项目目录下的data子文件夹

2. 设计网络模型

整个项目将网络定义、训练和测试的代码分别放在三个py文件中。其中，网络的定义在 `Model.py`

网络结构定义如下：

myCNN类继承自 `torch.nn.Module` 类，其中用一个 `dim_hidden` 数组设定中间层的维数（亦即网络参数的shape）

为了得到较好的结果，设定了一个比较大的模型，参数约为 10^6 量级

由较大模型带来的性能问题、运算时间问题将在训练代码中启用cuda解决

```
# 定义模型
class myCNN(nn.Module):
    dim_hidden = []
    def __init__(self):
```

```

super(myCNN, self).__init__()
"""定义网络结果"""
# dim_hidden : 定义网络中间层参数shape
dim_hidden = [32, 128, 512, 1024, 256]
self.dim_hidden = dim_hidden
"""卷积层"""
self.conv1 = nn.Conv2d(in_channels=1, out_channels=dim_hidden[0],
kernel_size=3, stride=1, padding=1)
self.conv2 = nn.Conv2d(in_channels=dim_hidden[0],
out_channels=dim_hidden[1], kernel_size=3, stride=1, padding=1)
self.conv3 = nn.Conv2d(in_channels=dim_hidden[1],
out_channels=dim_hidden[2], kernel_size=3, stride=1, padding=1)
self.bn1 = nn.BatchNorm2d(dim_hidden[0])
self.bn2 = nn.BatchNorm2d(dim_hidden[1])
self.bn3 = nn.BatchNorm2d(dim_hidden[2])
"""全连接层"""
self.fc1 = nn.Linear(in_features=dim_hidden[2] * 7 * 7,
out_features=dim_hidden[3]) # 注意输入的dim: 28/2/2 = 7
self.fc2 = nn.Linear(in_features=dim_hidden[3],
out_features=dim_hidden[4])
self.fc3 = nn.Linear(in_features=dim_hidden[4], out_features=10)

def forward(self, x):
"""forward前向运算"""
x = F.relu(self.bn1(self.conv1(x)))
x = nn.functional.max_pool2d(x, kernel_size=2)
x = F.relu(self.bn2(self.conv2(x)))
x = nn.functional.max_pool2d(x, kernel_size=2)
x = F.relu(self.bn3(self.conv3(x)))
x = x.view(-1, self.dim_hidden[2] * 7 * 7)
x = F.relu(self.fc1(x))
x = F.relu(self.fc2(x))
x = self.fc3(x)
return x
"""backward自动计算"""

```

3. 训练

3.1 通过试验设置超参数: epoch, batch_size, learning_rate

batch_size设置为64; epoch(数据集遍历次数)和learning_rate的设置则是在实验中不断优化的。

3.1.1 epoch_size的设定:

实践中, epoch_size分别取10,20和40;

epoch_size取40时, 发现当epoch >20时, loss已经基本不变了 (非常小, 且基本不再降低)

而根据实验发现, 对于我设置的参数偏大的网络结构, epoch=10不足以得到稳定的且正确率大于99%的结果。

如图, 我分别保存了前20次epoch遍历结束后的网络参数, 并对其分别在test数据集上检测其预测正确率:

```
predicting ...
测试集在epoch:0准确率: 98.1900 %
测试集在epoch:1准确率: 98.3200 %
测试集在epoch:2准确率: 98.6300 %
测试集在epoch:3准确率: 98.5700 %
测试集在epoch:4准确率: 98.9500 %
测试集在epoch:5准确率: 99.2000 %
测试集在epoch:6准确率: 98.7800 %
测试集在epoch:7准确率: 99.1000 %
测试集在epoch:8准确率: 99.0300 %
测试集在epoch:9准确率: 99.1800 %
测试集在epoch:10准确率: 99.2600 %
测试集在epoch:11准确率: 99.1800 %
测试集在epoch:12准确率: 99.0300 %
测试集在epoch:13准确率: 99.2700 %
测试集在epoch:14准确率: 99.3500 %
测试集在epoch:15准确率: 99.2300 %
测试集在epoch:16准确率: 99.1300 %
测试集在epoch:17准确率: 99.2500 %
测试集在epoch:18准确率: 99.0900 %
测试集在epoch:19准确率: 99.2100 %
最大正确率:99.3500
```

```
Process finished with exit code 0
```

可见，当epoch>10时稳定在99%以上，且最大的正确率出现在14处，可见epoch_size应该取10以上，又可见当epoch增大时正确率不再稳定上升，可见更大的epoch不会带来泛化性能的提升，反而会造成性能浪费甚至出现过拟合的问题。因此，epoch_size设置在20较为合理

3.1.2 learning_rate的设定

1. 一开始，我将learning_rate设定为0.01，结果出现了灾难性的结果：第一次实验时，我定义的模型较小，训练量较少，结果正确率只有30%

对于这种结果的出现，我的理解是：当学习率过大时，在梯度下降时很可能就“错过”了极小值点。

在目前模型的测试如图，虽然不会出现正确率低至30%的结果，但是仍可以看出收敛较慢：

```
...\\users\\811\\anaconda\\pycharm\\exce...\\users\\
predicting ...
测试集在epoch:0准确率: 95.9800 %
测试集在epoch:1准确率: 97.6200 %
测试集在epoch:2准确率: 98.1200 %
测试集在epoch:3准确率: 97.6700 %
测试集在epoch:4准确率: 97.2700 %
测试集在epoch:5准确率: 98.7600 %
测试集在epoch:6准确率: 98.5600 %
测试集在epoch:7准确率: 98.9300 %
测试集在epoch:8准确率: 99.2100 %
```

2. 将learning_rate设定为0.001，得到较为良好的结果（最终采用的结果）

3. 继续降低learning_rate，在learning_rate = 0.0005和0.0001处实验，发现除了收敛时间更慢，没有明显的准确率的提升

3.2 选取loss和优化器

在选用误差函数时，一开始想到的是简单的平方差函数，但是通过上网学习了解，CrossEntropyLoss更为合适（交叉熵损失函数）。

什么是交叉熵？

交叉熵主要是用来判定实际的输出与期望的输出的接近程度，为什么这么说呢，举个例子：在做分类的训练的时候，如果一个样本属于第K类，那么这个类别所对应的的输出节点的输出值应该为1，而其他节点的输出都为0，即[0,0,1,0,...,0,0]，这个数组也就是样本的Label，是神经网络最期望的输出结果。也就是说用它来衡量网络的输出与标签的差异，利用这种差异经过反向传播去更新网络参数。

优化器选用常用的Adam优化器。

3.3 训练

训练流程概括为：将数据输入到模型 -> 得到预测结果 -> 与正确值比较，计算loss -> 反向传播更新参数值

为了加快训练速度，我在代码中加入了检测cuda是否存在，如果存在，则启用gpu加速的部分

```
for epoch in loop:  
    for i, (images, labels) in enumerate(train_loader):  
        # use gpu  
        if torch.cuda.is_available():  
            images = images.cuda()  
            labels = labels.cuda()  
        output = model(images)  
        loss = criterion(output, labels)  
        # output loss  
        loop.set_postfix(loss=loss)  
        # 清零grad  
        optimizer.zero_grad()  
        # 反向传播  
        loss.backward()  
        # 参数更新  
        optimizer.step()
```

3.4 保存参数文件，待预测部分代码使用

为了降低项目代码的耦合度，将预测和训练的代码分别存到两个文件中，这就需要保存和加载模型的参数文件。

利用 `np.save` 和 `np.load` 保存和加载参数文件

4. 测试

4.1 加载测试数据集

方法基本同加载训练数据集一致

4.2 加载训练好的参数文件

4.3 遍历，累加预测正确的数量，计算正确率

定义两个变量：

1. `correct`: 用于存储正确标记的数量
2. `total`: 用于记录数据集总的数据量

计算正确率：

$$correctness = \frac{correct}{total} * 100\%$$

结果

指标1：正确率：99%+，测试中最佳可以达到99.5%左右

```
predicting ...
测试集在epoch:0准确率: 98.1900 %
测试集在epoch:1准确率: 98.3200 %
测试集在epoch:2准确率: 98.6300 %
测试集在epoch:3准确率: 98.5700 %
测试集在epoch:4准确率: 98.9500 %
测试集在epoch:5准确率: 99.2000 %
测试集在epoch:6准确率: 98.7800 %
测试集在epoch:7准确率: 99.1000 %
测试集在epoch:8准确率: 99.0300 %
测试集在epoch:9准确率: 99.1800 %
测试集在epoch:10准确率: 99.2600 %
测试集在epoch:11准确率: 99.1800 %
测试集在epoch:12准确率: 99.0300 %
测试集在epoch:13准确率: 99.2700 %
测试集在epoch:14准确率: 99.3500 %
测试集在epoch:15准确率: 99.2300 %
测试集在epoch:16准确率: 99.1300 %
测试集在epoch:17准确率: 99.2500 %
测试集在epoch:18准确率: 99.0900 %
测试集在epoch:19准确率: 99.2100 %
最大正确率:99.3500

Process finished with exit code 0
```

正确率稳定在99%以上

最大正确率在实验中能够达到99.5左右。

指标2：运行时间：

利用tqdm库显示进度条：



利用CUDA加速后，一个epoch大概要运行20s左右