卷积神经网络-MNIST实战(基于pytorch)

目录

前言

随着深度学习的不断发展，神经网络也变得越来越热门。众所周知，cnn在图像分类问题上效果优越，本文将展示一个简单的卷积神经网络模型，使用mnist数据集进行测试。在本文后半段会给出本次实践的心得体会。

一、准备工作

首先我们要import我们需要用到的包，并进行必要的参数设置。

代码如下：

Python

```python
import time
import numpy as np
from torchvision import transforms
from torchvision.datasets import mnist
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.optim as optim

# 定义batch，即一次训练的样本量大小
train_batch_size = 128
test_batch_size = 128
```

二、导入数据集

1.下载数据集

代码如下：

Python

```python
# 定义图像数据转换操作
# mnist是灰度图，应设置为单通道
# ToTensor():[0,255]->[C,H,W];Normalize: 标准化(均值+标准差)
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize([0.5],[0.5])])

# 下载mnist数据集,若已下载，可将download定义为False
data_train = mnist.MNIST('./data', train=True, transform=transform,
target_transform=None, download=False)
data_test = mnist.MNIST('./data', train=False, transform=transform,
target_transform=None, download=False)
```

2.装载数据集

代码如下：

Plain Text

```
# 对数据进行装载，利用batch _size来确认每个包的大小，用Shuffle来确认打乱数据集的顺序。
train_loader = DataLoader(data_train, batch_size=train_batch_size,
shuffle=True)
test_loader = DataLoader(data_test,batch_size=test_batch_size,shuffle=True)
```

三、数据可视化

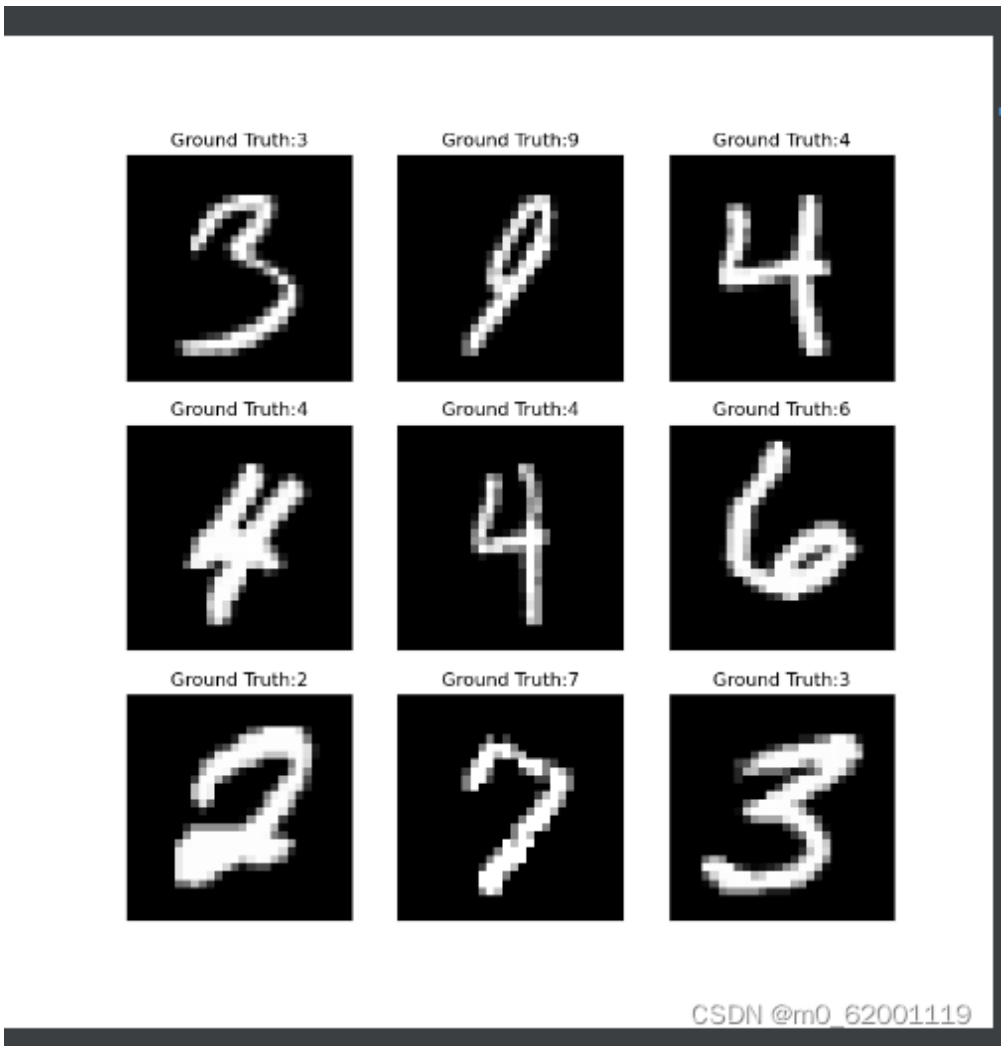
注意这一步不是必要的操作，但是可以检查数据集是否成功导入，能直观反映数据集的内容。

代码如下：

Plain Text

```
# 可视化数据
examples = enumerate(test_loader)
batch_idx, (example_data, example_targets) = next(examples)
plt.figure(figsize=(9, 9))
for i in range(9):
plt.subplot(3, 3, i+1)
plt.title("Ground Truth:{}".format(example_targets[i]))
plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
plt.xticks([])
plt.yticks([])
plt.show()
```

运行结果展示：



可以看到，输出了九张手写数字的灰度图。因为它是灰度图，所以这个模型将使用单通道输入。

四、搭建模型

1.cnn框架构建

可以看到，这是一个2+2的cnn模型，包括两个卷积层和两个全连接层。

ps: feature map size 计算公式 -> [(W-F+2P)/S + 1] * [(W-F+2P)/S + 1] * M

Plain Text

```
# 搭建CNN网络
class CNN(nn.Module):
def __init__(self):
super(CNN, self).__init__()

# 卷积层
self.conv1 = nn.Sequential(
# [b,28,28,1]->[b,28,28,16]->[b,14,14,16]
nn.Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1), padding=1),
nn.ReLU(inplace=True),
nn.MaxPool2d(kernel_size=2, stride=2),
)

self.conv2 = nn.Sequential(
# [b,14,14,16]->[b,14,14,32]->[b,7,7,32]
nn.Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=1),
nn.ReLU(inplace=True),
nn.MaxPool2d(kernel_size=2, stride=2),
)

# 全连接层
self.dense = nn.Sequential(
# 线性分类器
# []
nn.Linear(7 * 7 * 32, 128),
nn.ReLU(),
nn.Dropout(p=0.5),  # 缓解过拟合，一定程度上正则化
nn.Linear(128, 10),
)

# 前向计算
def forward(self, x):
x = self.conv1(x)
x = self.conv2(x)
x = x.view(x.size(0), -1)  # flatten张量铺平，便于全连接层的接收
return self.dense(x)
```

2.模型实例化

代码如下：

Plain Text

```
model = CNN()    # 实例化模型
print(model)     # 打印模型
```

结果展示：

```
CNN(
  (conv1): Sequential(
    (0): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (dense): Sequential(
    (0): Linear(in_features=1568, out_features=128, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=128, out_features=10, bias=True)
  )
)
```

五、训练模型

代码如下：

Plain Text

```
# 设置训练次数
num_epochs = 10
# 定义损失函数
criterion = nn.CrossEntropyLoss()
```

```
# 定义学习率及优化方法
LR = 0.01
optimizer = optim.Adam(model.parameters(), LR)

# 开始训练 先定义存储损失函数和准确率的数组
train_losses = []
train_acces = []
# 测试
eval_losses = []
eval_acces = []

print("start training...")
# 记录训练开始时刻
start_time = time.time()

# 训练模型
for epoch in range(num_epochs):

# 训练集:
train_loss = 0
train_acc = 0

# 将模型设置为训练模式
model.train()

for img, label in train_loader:
out = model(img)    # 返回每个类别的概率
loss = criterion(out, label)   # 对比实际label得到损失

optimizer.zero_grad()   # 模型参数梯度清零
loss.backward()    # 误差反向传递
optimizer.step()    # 更新参数

train_loss += loss    # 累计误差

_, pred = out.max(1)    # 返回最大概率的数字
num_correct = (pred == label).sum().item()  # 记录标签正确的个数
acc = num_correct / img.shape[0]
train_acc += acc

# 取平均存入
train_losses.append(train_loss / len(train_loader))
train_acces.append(train_acc / len(train_loader))

# 测试集:
eval_loss = 0
eval_acc = 0

```

```
# 将模型设置为测试模式
model.eval()

# 处理方法同上
for img, label in test_loader:
out = model(img)
loss = criterion(out, label)

optimizer.zero_grad()
loss.backward()
optimizer.step()

eval_loss += loss

_, pred = out.max(1)
num_correct = (pred == label).sum().item()  # 记录标签正确的个数
acc = num_correct / img.shape[0]
eval_acc += acc

eval_losses.append(eval_loss / len(test_loader))
eval_acces.append(eval_acc / len(test_loader))

# 输出效果
print('epoch:{},Train Loss:{:.4f},Train Acc:{:.4f},'
'Test Loss:{:.4f},Test Acc:{:.4f}'
.format(epoch, train_loss / len(train_loader),
train_acc / len(train_loader),
eval_loss / len(test_loader),
eval_acc / len(test_loader)))
# 输出时长
stop_time = time.time()
print("time is:{:.4f}s".format(stop_time-start_time))
print("end training.")
```

结果展示：

```
start training...
epoch:0,Train Loss:0.2493,Train Acc:0.9253,Test Acc:0.9808
time is:51.9193s
epoch:1,Train Loss:0.0630,Train Acc:0.9806,Test Acc:0.9803
time is:51.8841s
epoch:2,Train Loss:0.0544,Train Acc:0.9827,Test Acc:0.9861
time is:56.7365s
```

```
epoch:3,Train Loss:0.0394,Train Acc:0.9875,Test Acc:0.9878
time is:60.0540s
epoch:4,Train Loss:0.0388,Train Acc:0.9876,Test Acc:0.9875
time is:58.4582s
epoch:5,Train Loss:0.0340,Train Acc:0.9892,Test Acc:0.9875
time is:51.3471s
epoch:6,Train Loss:0.0307,Train Acc:0.9902,Test Acc:0.9874
time is:54.1184s
epoch:7,Train Loss:0.0284,Train Acc:0.9908,Test Acc:0.9876
time is:55.6760s
epoch:8,Train Loss:0.0262,Train Acc:0.9911,Test Acc:0.9859
time is:51.9431s
epoch:9,Train Loss:0.0291,Train Acc:0.9910,Test Acc:0.9871
time is:51.0636s
end training.                                    CSDN @m0_62001119dy
```

经过10次迭代，可以看到，模型的效果还算可观，测试的平均准确率能达到98%

六、损失可视化

注意这一步的操作同样可以省略，但我们想更直观的看出训练迭代的效果，因此将损失可视化。

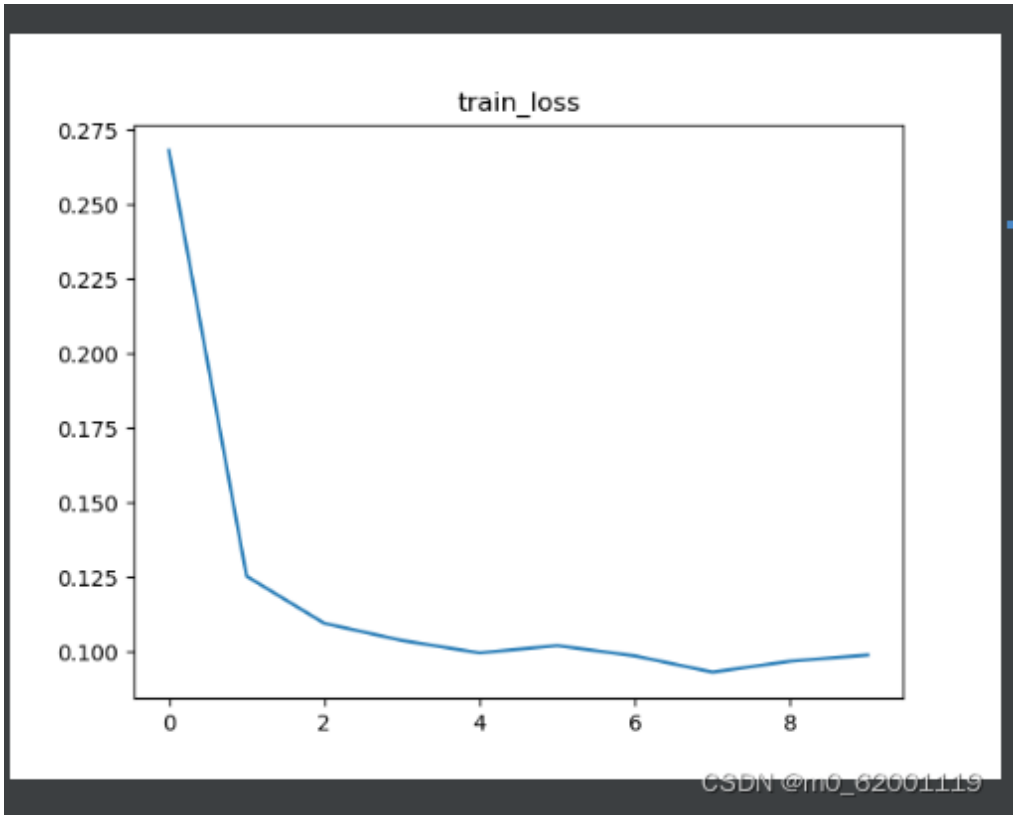
代码如下：

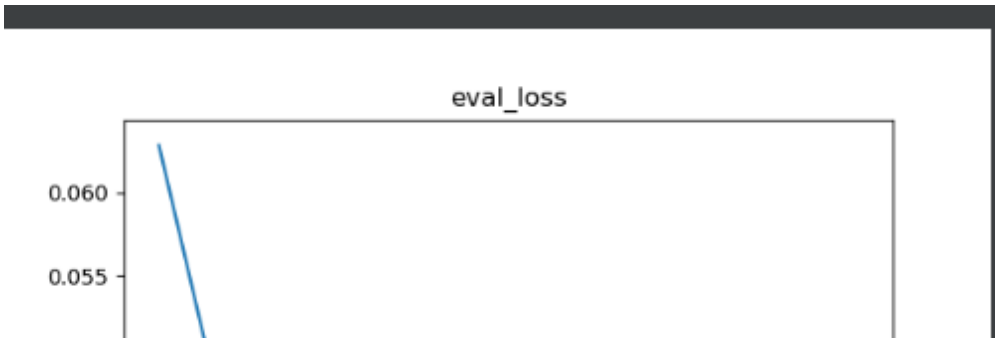
Plain Text

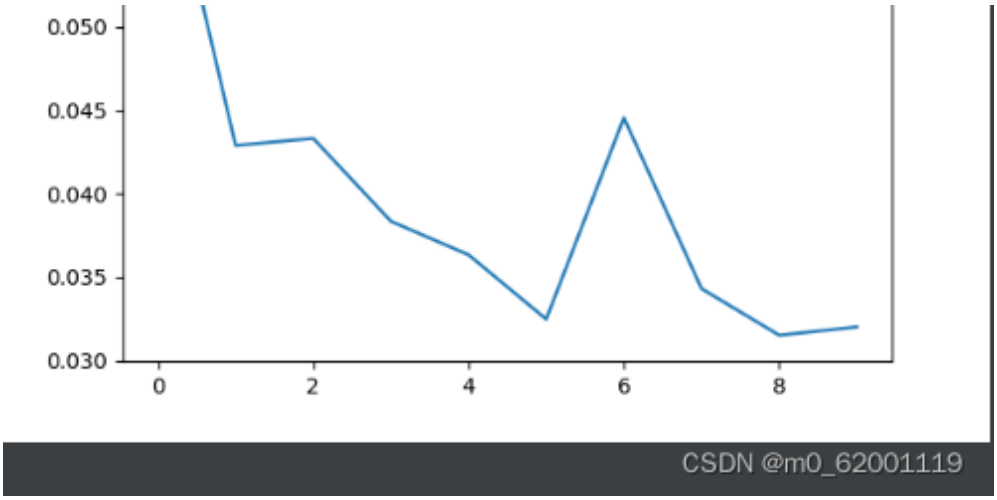
```
# 损失可视化
plt.title("train_loss")
plt.plot(np.arange(len(train_losses)), train_losses)
plt.show()

plt.title("eval_loss")
plt.plot(np.arange(len(eval_losses)), eval_losses)
plt.show()
```

效果展示：







可以看到，随着新一轮的迭代，训练的loss值下降比较可观，测试的loss值有些起伏，但大体上是在逐渐降低的。

总结

一：可以看到，这个模型对于mnist数据集的测试效果还不错。但实际上，作者在初次搭建网络的时候，测试准确率只有百分之八十几。查阅资料后，我发现，对于一个cnn模型，某些参数的设置会对整个模型的泛化能力产生较大影响。下面我将列举出几个比较典型的在参数方面优化网络的方法。

1. layers_num:

适当增加隐含层数目以加深网络深度，会在一定程度上改善网络性能。但同时也提高了训练该网络的计算成本。当网络的单元数设置过少时，可能会导致欠拟合，而单元数设置过多时，就有可能产生过拟合现象。

2. batch:

在卷积神经网络的学习过程中，小批次相对来说会表现得更好，选取范围一般位于区间[16,128]内。作者一开始batch_size设置的是512，后来改为128，整体测试效果提升了2~5个百分点。

3. epochs:

若条件满足，训练次数可以尽可能设置大一点。在每批样例训练完成之后，比较测试误差和训练误差，如果它们的差距在缩小，那么就继续训练，直到准确率达到稳定峰值。另外，最好在每批训练之后，保存模型的参数，这样，在训练完之后可以从多个模型中选择最佳的模型。

4. Dropout：

如果有数百万的参数需要学习，正则化就是避免产生过拟合的必须手段。执行 Dropout 很容易，并且通常能带来更快地学习。0.5 的默认值是一个不错的选择，当然，这取决于具体任务。如果模型不太复杂，0.2 的 Dropout 值或许就够了。作者搭建的网络在全连接层的第一层输出后使用了Dropout进行正则化，效果确实要比之前好一些，计算速度则是大幅度提升。

二：另外，在这次卷积神经网络的学习中，我也总结了一些心得体会。

1. visualize可视化:

训练深度学习模型有上千种出差错的方式。很有可能模型已经训练了几个小时或者好几天，然而在训练完成之后，才发现到某个地方出问题了。为了避免这种情况，一定要对训练过程作可视化处理。比如保存或打印损失值、训练误差、测试误差等重要数据。

2. 配置：

如果大家有条件，建议尽量使用GPU来跑神经网络，作者这个模型是在CPU上跑的，其实网络结构并不算复杂，但是每次训练时长几乎达到一分钟，非常耗时！

3. 数据集：

如果大家搭建的网络模型比较复杂，但数据集比较单一，就很容易出现过拟合的现象，如果有条件，尽量选择足够大的样本数据集，以期达到更好的模型训练效果。

4. 激活函数，损失函数，学习率，优化函数：

这些函数及参数的设置也非常重要。作者在这个模型中选择的是relu函数,CrossEntropyLoss交叉熵损失函数，学习率是0.01，优化方法是Adam()。这是比较常见的设置方法。但往往有些模型要根据实际情况来选取不同的激活函数和优化方法，这里作者还了解得不够深入，想进一步了解可移步深度学习（RNN系列、CNN、 Attention系列 + 激活函数 + 损失函数 + 优化器 + BN + Transformer+Dropout）_MrWilliamVs的专栏-CSDN博客

参考

PyTorch学习笔记（1）nn.Sequential、nn.Conv2d、nn.BatchNorm2d、nn.ReLU和nn.MaxPool2d_张小波的博客-CSDN博客

积神经网络的参数优化方法——调整网络结构是关键！！！你只需不停增加层，直到测试误差不再减少. - bonelee - 博客园

聊一聊CNN中的感受野、优化函数、激活函数、Loss函数等_zshluckydogs的博客-CSDN博客

一文看懂卷积神经网络-CNN（基本原理+独特价值+实际应用）- 产品经理的人工智能学习库

尾声

以上就是今天要展示的全部内容，本文还有很多不足之处，敬请指正！

下一篇文章指路-> 卷积神经网络 实战CIFAR10-基于pytorch_m0_62001119的博客-CSDN博客