浙江大学

计算机视觉(本科)作业报告

作业名称： Learning CNN

姓　　名：

学　　号：

电子邮箱：

联系电话：

导　　师：

2020 年 1 月 1 日

Learning CNN

一、 作业已实现的功能简述及运行简要说明

1.功能简述

本次实验的要求是实现最基本的卷积神经网络（CNN）LeNet-5 以及一个物体分类的 CNN，具体要求为：①自己用 MNIST 手写数字数据集（0-9 一共十个数字）6 万样本实现对 LeNet-5 的训练，对 MNIST 的 1 万测试样本进行测试，获得识别率是多少；②自己用 CIFAR-10 数据库实现 CNN 物体分类功能的训练与测试。

其中，在本实验过程中，我使用的是 Pytorch，没有直接读取各种深度学习开发工具已训练好的 CNN 网络结构与参数。

在本次实验中，可以得到训练好的网络和测试出的识别率。

2.运行简要说明：

打开 Pycharm 运行程序，调用../code/MNIST/train.py 和../code/CIFAR10/train.py 后可以得到训练好的网络，调用../code/MNIST/test.py 和../code/CIFAR10/test.py 后会得到识别率，准确度。

二、 作业的开发与运行环境

Windows 10

Python 3.7.4

Pytorch 1.2.0

三、 系统或算法的基本思路、原理、及流程或步骤等

程序分为以下五个步骤：

1. **数据加载和处理**

   加载和归一化训练数据和测试数据

2. **定义卷积神经网络**

   根据网络结构定义一个包含可训练参数的神经网络。

3. **定义损失函数和优化器**

   使用分类交叉熵 Cross-Entropy 作损失函数，动量 SGD 做优化器。

4. **训练**

   得到输入数据，将数据输入网络，计算损失值，执行反向传播之后将优化器的参数进行更新，以便进行下一轮。

5. **测试**

   用测试集对训练好的网络进行测试，测试其准确度、识别率。

四、 具体如何实现，例如关键（伪）代码、主要用到函数与算法等

（一）MNIST

1. 数据加载和处理

首先创建一个转换器，将数据转换为 Tensor，并归一化至[0, 1]，创建训练集和测试集（测试集为后面测试用），这里用到的函数为 torchvision.datasets.MNIST (self, root, train=True, transform=None,target_transform=None,download=False)其中 root 指数据存放的目录，train 明确是否是训练集，download 表示是否需要下载，transform 表示转换器，将数据进行转换。

然后创造数据加载器，这里用到的函数为 DataLoader(self, dataset, batch_size=1, shuffle=False)，其中 dataset 为上面创建的数据集，shuffle 表示是否打乱顺序。

最后建立类别标签。

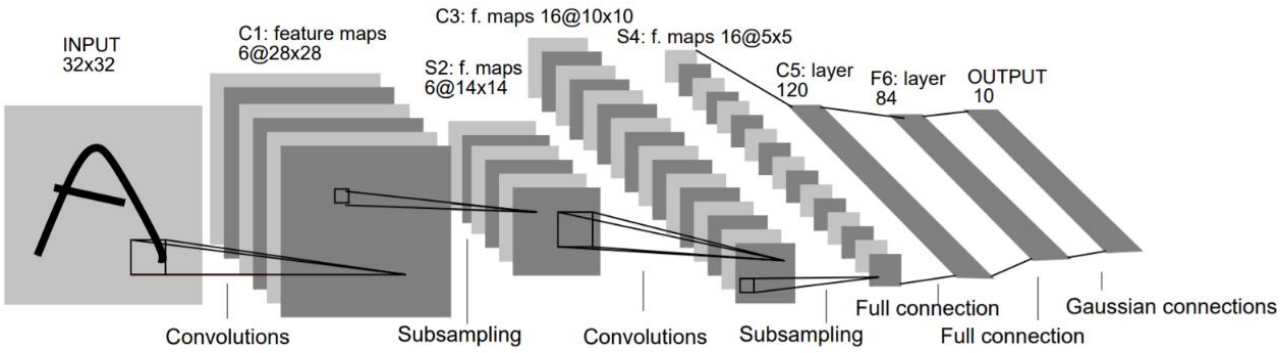
这里所涉及到的代码实例(创建数据集以创建训练集为例)如下:

```
# 训练集
trainset = torchvision.datasets.MNIST(
    root='data/',
    train=True,
    download=True,
    transform=transform
)
trainloader = DataLoader(
    dataset=trainset,
    batch_size=4,
    shuffle=True
)

# MNIST 数据集中的十种标签
classes = ('0', '1', '2', '3', '4',
           '5', '6', '7', '8', '9')
```

2. 定义卷积神经网络

这里我用的是 LeNet-5 网络，其网络结构如下图（Fig-1）所示。



该网络包含 7 层（不算入输入层），由两个卷积层，两个池化层和三个全连接层组成。

第一层为卷积层，该层的输入就是原始图像的像素，在这里 LeNet 模型接受的输入层大小为 32 * 32 * 1，卷积层过滤器的尺寸为 5 * 5，深度为 6，步长为 1，所以这一层的输出尺寸为 32 - 5 + 1 = 28，同时采用 ReLu 函数作为激活函数。

第二层为池化层，该层的输入为第一层的输出，输入尺寸为 28 * 28 * 6，池化层采用的过滤器大小为 2 * 2，长和宽的步长均为 2 * 2，采用最大池化的方式，所以本层的输出尺寸为 14 * 14 * 6。

第三层为卷积层，该层输入尺寸为 14 * 14 * 6，过滤器尺寸为 5 * 5，深度为 16，步长为 1，所以该层的输出尺寸为 14 - 5 + 1 = 10，同时采用 ReLu 函数作为激活函数。

第四层为池化层，该层的输入尺寸为 10 * 10 * 16，其他参数与第二层保持一致，所以本层的输出尺寸为 5 * 5 * 16。

接下来第五层为全连接层，该层的输入尺寸为 5 * 5 * 16，在全连接层，首先将输入矩阵拉成一个向量，这里将把输入转换成一个 5 * 5 * 16 维的向量，输出向量的维数为 120，将输入向量与输出向量全连接，参数矩阵的尺寸为[5 * 5 * 16, 120]，本层输出节点个数：120，共有 5×5×16×120+120=48120 个参数。

第六层也是全连接层，该层输入向量维数为 120，输出向量维数为 84，共有 120×84+84=10164 个，将输入向量与输出向量全连接，参数矩阵的尺寸为[120, 84]。

第七层为全连接层，本层输入向量维数为 84，输出向量维数为 10，共有 84×10+10=850 个，将输入与输出进行全连接，参数矩阵的尺寸为[84, 10]。最终输出是一个范围为[0, 1]的 10 维向量，依次代表 0-9，取 10 个数中最大的值为预测结果。

在具体实现上，首先，torch.nn.nnModule 是所有神经网络的基类，定义神经网络需要继承 torch.nn.nnModule。

在定义卷积层和池化层以及进行之间的运算时，采用的几个函数如下：当定义卷积层时，采用的函数为 `torch.nn.Conv2d(self, in_channels, out_channels, kernel_size)`,其中 in_channels 表示输入维度，out_channels 表示输出维度，kernel_size 表示卷积核大小，在定义向前传播的方法时，关于卷积层与池化层之间进行运算中，用到了 `torch.nn.functional.relu(input, inplace=False)`函数，relu 函数为激活函数，用于增加神经网络各层之间的非线性关系，在进行池化时候，我用到了 `x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))`函数。

在定义全连接层时，用到了 `torch.nn.(self, in_features, out_features, bias=True)`，其中 in_features 为输入样本的大小，out_features 为输出样本的大小，之后运用 view()函数对向量的大小进行重新定义，其中参数-1 表示自适应，根据另一个参数的大小定义。之后采用 `torch.nn.functional.relu(input, inplace=False)`函数进行激活。

网络结构的代码定义如下：

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        # 卷积层
        #三个参数分别代表：input channels 输入维度，output channels 输出维度，kernel size 卷积核大小
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)

        # 全连接层
        #in_features: input vector dimensions 输入样本的大小
        #out_features: output vector dimensions 输出样本的大小
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    #定义前向传播
    def forward(self, x):
        # 卷积 --> ReLu --> 池化
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))

        # '-1'表示自适应
        # x = (n * 16 * 4 * 4)  其中 n 表示 input channels
        # x.size()[0]  即 n
        x = x.view(x.size()[0], -1)

        # 全连接层
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x
```

3. 定义损失函数和优化器

定义损失函数使用分类交叉熵 `torch.nn.CrossEntropyLoss()`,即定义了一个交叉熵损失函数（交叉熵描述了两个概率分布之间的距离，当交叉熵越小说明二者之间越接近）。

定义优化器则用动量 SGD 做优化器 `optim.SGD(self, params, lr=required, momentum=0)`，SGD 是最基础的优化方法。普通的训练方法,需要重复不断的把整套数据放入神经网络中训练，这样消耗的计算资源会很大。当我们使用 SGD 会把数据拆分后再分批不断放入神经网络中计算。每次使用批数据，虽然不能反映整体数据的情况，不过却很大程度上加速了训练过程，而且也不会丢失太多准确率。

实际代码如下：

```
# 定义损失函数和优化器,使用分类交叉熵 Cross-Entropy 作损失函数，动量 SGD 做优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.005, momentum=0.9)
```

4. 训练

首先用 Python 的内置函数 enumerate 获得索引和数据，数据赋值给 data，data 中包含数据和标签信息，分别赋值给 inputs 和 labels，之后将 inputs 和 labels 转换为 Variable。因为反向传播过程中梯度会累加上一次循环的梯度，所以在每次循环中我们都要把梯度重新归零，这里我们用到的函数为 optimizer.zero_grad()。之后我们把数据输进网络，计算损失值，loss 进行反向传播，当执行反向传播之后，将优化器的参数进行更新，以便进行下一轮操作。

其中代码实例如下所示：

```
#得到输入数据
inputs, labels = data
#使用 gpu
if torch.cuda.is_available():
    inputs = inputs.cuda()
    labels = labels.cuda()
inputs, labels = Variable(inputs), Variable(labels)

# 梯度清 0
optimizer.zero_grad()

# forward + backward + optimize
outputs = net(inputs)
loss = criterion(outputs, labels)
loss.backward()

# 更新参数
optimizer.step()
```

5. 测试

用测试集对训练好的网络进行测试，测试方法如下：首先取最大值作为预测结果，这里我们用到的函数为 torch.max(self: Tensor, *, out: Optional[Tensor]=None)，该函数返回的两个元素为最大的值和最大的值的索引，我们只需要最大值的索引，然后将预测值与标签值进行比较，如果预测值与标签值相等，则计入正确总数，最后用正确数量除以总数目得到预测准确率（即识别率）。

```
for k, data in enumerate(dataloader):
    inputs, labels = data
    if torch.cuda.is_available():
        inputs = inputs.cuda()
        labels = labels.cuda()
    inputs, labels = Variable(inputs), Variable(labels)

    outputs = model(inputs)

    # 取最大值为预测结果
    _, predicted = torch.max(outputs, 1)

    for j in range(len(predicted)):
        predicted_num = predicted[j].item()
        label_num = labels[j].item()
        # 预测值与标签值进行比较
        if predicted_num == label_num:
            correct_num += 1

# 计算预测准确率
correct_rate = correct_num / data_len
print(str(i),'correct rate is {:.3f}%'.format(correct_rate * 100))
```

(二) CIFAR-10

用 CIFAR-10 实现 CNN 物体分类功能的训练与测试的过程中，其中数据集的加载和处理，定义损失函数和优化器,网络的训练和测试部分的步骤大致相同，主要针对定义卷积神经网络和定义损失函数和优化器方面进行介绍和总结。

1. 数据加载和处理

加载和归一化训练数据和测试数据，同加载和处理 MNIST 数据集大致相同，可参见上面所写的内容和步骤，具体详见代码。

2. 定义卷积神经网络

这里我用的是 VGG19 模型，VGG 网络结构如下图（Fig-2）所示，VGG19 的网络结构如图（Fig-3 所示）其中 conv 表示卷积层，FC 表示全连接层，conv3 表示卷积层使用 3x3 filters，conv3-64 表示深度 64，maxpool 表示最大池化，VGG19 包含了 19 个隐藏层（16 个卷积层和 3 个全连接层），如下图（Fig-2）中的 E 列所示。VGGNet 的结构非常简洁，整个网络都使用了同样大小的卷积核尺寸（3x3）和最大池化尺寸（2x2），并且几个小滤波器（3x3）卷积层的组合比一个大滤波器（5x5 或 7x7）卷积层更好，验证了通过不断加深网络结构可以提升性能。

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

Fig-2

VGG19		
序号	层结构	
1	conv1-1	1
2	relu1-1	
3	conv1-2	2
4	relu1-2	
5	pool1	
6	conv2-1	3
7	relu2-1	
8	conv2-2	4
9	relu2-2	
10	pool2	
11	conv3-1	5
12	relu3-1	
13	conv3-2	6
14	relu3-2	
15	conv3-3	7
16	relu3-3	
17	conv3-4	8
18	relu3-4	
19	pool3	
20	conv4-1	9
21	relu4-1	
22	conv4-2	10
23	relu4-2	
24	conv4-3	11
25	relu4-3	
26	conv4-4	12
27	relu4-4	
28	pool4	
29	conv5-1	13
30	relu5-1	
31	conv5-2	14
32	relu5-2	
33	conv5-3	15
34	relu5-3	
35	conv5-4	16
36	relu5-4	
37	pool5	
38	fc6(4096)	17
39	relu6	
40	fc7(4096)	18
41	relu7	
42	fc8(1000)	19
43	prob(softmax)	

Fig-3

具体的代码实现详见../code/CIFAR10/model.py

3. 定义损失函数和优化器

同对 MNIST 数据集进行训练前定义的损失函数和优化器步骤大致相同，可参见上面所写的内容，具体详见代码。

4. 训练

同对 MNIST 数据集进行的训练大致相同，可参见上面所写的内容，具体详见代码。

5. 测试

同对 MNIST 数据集进行的测试大致相同，可参见上面所写的内容，具体详见代码。

五、 实验结果与分析

1.MNIST

用 MNIST 手写数字数据集（0-9 一共十个数字）6 万样本实现对 LeNet-5 的训练，训练出 50 个网络，对 MNIST 的 1 万测试样本进行测试，获得识别率如下（%）：

98.000	98.120	98.080	98.640	98.590	98.670	98.460	98.680	98.750	98.060
98.390	98.560	98.870	98.690	98.430	98.020	97.990	98.700	98.330	98.520
98.290	98.530	97.880	96.160	97.120	97.300	97.890	96.400	96.160	96.760
97.760	97.450	97.900	97.030	96.330	95.200	91.530	98.070	97.940	96.640
98.070	98.500	98.380	97.940	97.900	98.620	98.300	98.810	98.770	98.630

从最终结果得到的准确率来看，训练出来的网络识别率比较稳定，都比较高，大部分达到了 95%以上，于是我选择保留第 22 次训练出来的网络在../code/MNIST/net 中。

2.CIFAR-10

用 CIFAR-10 数据库实现 CNN 物体分类功能的训练.同样训练出 50 个网路，对网络的测试结果如下（%）：

60.640	64.900	74.500	78.070	81.030	81.730	82.620	82.960	84.980	86.350
84.000	83.720	87.070	87.240	86.770	87.740	87.990	88.080	88.090	88.410
88.350	88.860	89.120	88.340	90.060	88.590	89.200	89.710	89.910	89.270
89.400	89.770	90.080	89.590	90.370	90.320	90.480	90.410	89.960	89.610
91.070	90.300	90.510	90.260	90.430	90.660	90.320	91.010	90.730	90.690

从最终结果得到的准确率来看，随着训练次数的增加，训练出来的网络识别率在逐渐提高，最后趋于稳定，我选择保留第 41 次训练出来的网络，由于网络太大， 可在以下链接中下载：分享了一个文件(夹)给你：【net41.pkl】，点击链接查看：https://pan.zju.edu.cn/share/6f6c218ff4c218d0e157d0348b

六、 结论与心得体会

在完成本次实验的过程中主要遇到了两个问题,接下来将主要介绍这两个问题及其解决方案。

第一个问题出现在神经网络的选取上,刚开始对于 CIFAR-10 数据库的神将网络训练时，我采取定义 CIFAR-10 的神经网络结构为 LeNet-5，结果训练后效果一直不是特别好，准确率在百分之六十左右，于是我采取 VGG 网络结构，准确率有了显著的提升。

第二个问题出现在神经网络的训练次数上，刚开始为了缩短训练时间将对 CIFAR-10 数据库的神经网络训练减少到了 10 次，但是出现的准确率只有百分之八十五左右，在增加了训练次数后，训练得到的识别率显著提高。

通过本次实验，我了解了神经网络的结构。之前对神经网络一直是一知半解，虽然已经听过了很多次对于神经网络结构的讲解，但是对于其具体的实施没有什么概念，通过这次实

验的上手操作，我懂得了如何用 Pytorch 构造一个简单的神经网络，对于神经网络有了更加深刻的了解，收获很多。

七、 参考文献

optimizer.zero_grad()： https://blog.csdn.net/scut_salmon/article/details/82414730

python torch.optim.SGD： https://blog.csdn.net/qq_34690929/article/details/79932416

CrossEntropyLoss： https://blog.csdn.net/m0_38133212/article/details/88087206

View： https://blog.csdn.net/york1996/article/details/81949843

Relu 激活函数： https://blog.csdn.net/htt789/article/details/80235908