LeNet [1]是早期用来识别手写数字图像的卷积神经网络，这个名字来源于 LeNet 论文的第一作者 Yann LeCun。LeNet 展示了通过梯度下降训练卷积神经网络可以达到手写数字识别在当时最先进的结果。这个奠基性的工作第一次将卷积神经网络推上舞台，为世人所知。

LeNet 网络结构比较简单，本篇的核心是借助 LeNet 这一最初的基本卷积神经网络介绍一下基于 pytorch 框架的深度学习网络代码框架与核心内容。

解决的核心问题就是前面介绍的卷积神经网络的理论知识如何用代码实现。

LeNet 网络结构

LeNet 的网络结构如下图所示。

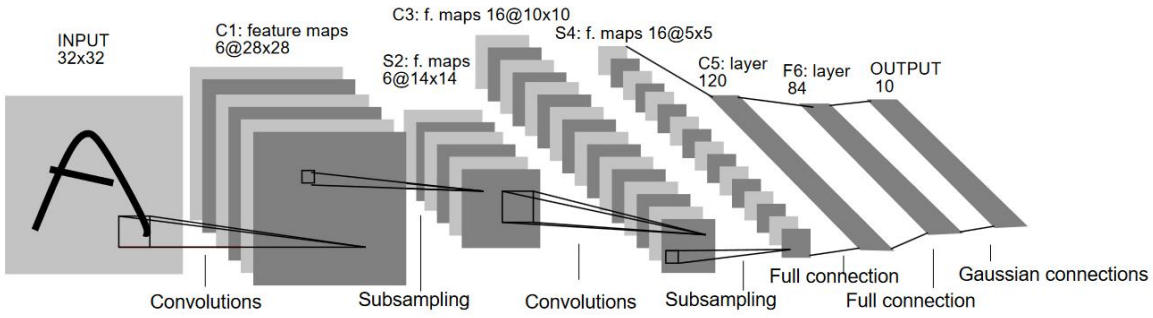


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

LeNet 仅包括 2 个连续的卷积层和池化层和最后的全连接层。

每个卷积层都使用 5*5 的卷积核，并在输出上使用 sigmoid 激活函数。第一个卷积层输出通道数为 6，第二个卷积层输出通道数则增加到 16。这是因为第二个卷积层比第一个卷积层的输入的高和宽要小，所以增加输出通道使两个卷积层的参数尺寸类似。

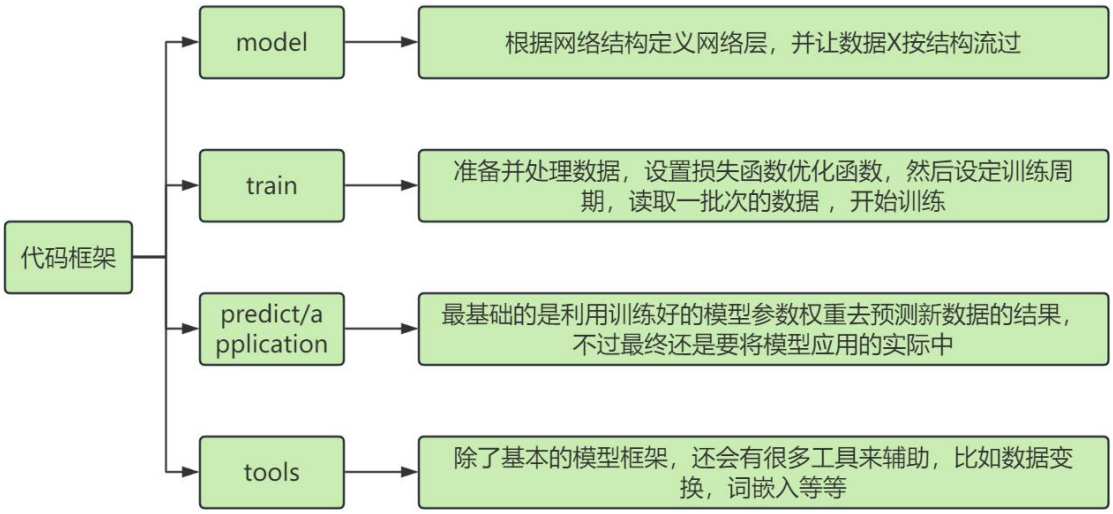
卷积层后的最大池化层的窗口形状均为 2*2，且步幅为 2。由于池化窗口与步幅形状相同，池化窗口在输入上每次滑动所覆盖的区域互不重叠。

卷积层和池化层的输出张量维度为(批量大小, 通道, 高, 宽)。当输出传入全连接层时，全连接层会将小批量中每个样本变平（flatten）。也就是说，全连接层的输入维度将变成二维，其中第一维是小批量中的样本总数 batch_size，第二维是每个样本变平后的向量表示，且向量长度为通道、高和宽的乘积。最后有 3 个全连接层，它们的输出特征尺寸分别是 120、84 和 10，其中 10 为输出的类别个数。

可以看到，LeNet 作为第一个应用到实际的卷积神经网络，结构较为简单，不需做过多的介绍，仅以此作为代码演练分析的第一个实例。

代码分析

在分析具体代码前，我们先来看一下整体的代码框架，一般来说，深度学习网络模型分为三个核心部分，一是模型定义，二是模型训练，三是模型预测应用。接下来我们详细介绍一下各模块。



模型定义

定义的模型类（继承子 pytorch 里面的基类）核心是两个函数，一个是 init 初始化函数，在这个函数里定义各个网络层的参数，比如输入输出通道数，卷积核池化层尺寸等；第二个是 forward 前向传递函数，这里是将数据的流动路线（经过每个层的先后顺序）按照网络结构写出来。

```
# 导入 pytorch 库
import torch
# 导入 torch.nn 模块 这是 pytorch 里面网络层的基类，定义的模型类就是继承 nn.Module。
from torch import nn
import random

# 定义 LeNet 网络模型
# LeNet（子类）继承 nn.Module（父类）
class LeNet(nn.Module):
    # 子类继承中重新定义 Module 类的__init__()和 forward()函数
    # init()函数：进行初始化，定义模型中各层的参数
    def __init__(self):
        # super：引入父类的初始化方法给子类进行初始化
        super(LeNet,self).__init__()
```

卷积层，可以看到 nn.Conv2d 函数的主要参数是输入输出通道和卷积核尺寸及 padding 值（输入通道为 1，输出为 6，卷积核为 5，扩充边缘为 2。），可以回过头再看看前面介绍卷积层的理论部分。每经过一个网络层，要重点关注数据的维度变化，这里数据输入维度 1*28*28，输出维度是 6*28*28。

```
self.c1=nn.Conv2d(in_channels=1,out_channels=6,kernel_size=5,padding=2)
        print(self.c1.weight.shape)
        self.Sigmoid=nn.Sigmoid()# 使用 sigmoid 作为激活函数
        # AvgPool2d：二维平均池化操作
        # 池化层，池化层主要是对每一通道的尺寸进行变化，不改变通道数。输入大小为 28*28，输出大小为 14*14，输入通道为 6，输出为 6，卷积核为 2，步长为 2
        self.s2=nn.AvgPool2d(kernel_size=2,stride=2)
        # 卷积层，输入大小为 14*14，输出大小为 10*10，输入通道为 6，输出为 16，卷积核为 5。这里一个约定俗成的点是所有网络层的总数是一直累加的，不是按照相同的网络层累加，前面用层的英文首字母表示，如第一层为卷积层，则命名为 c1，后面的第二层为池化层则命名为 s2，第三层为卷积层命名 c3。
        self.c3=nn.Conv2d(in_channels=6,out_channels=16,kernel_size=5)
        print(self.c3.weight.shape)
        # 池化层，输入大小为 10*10，输出大小为 5*5，输入通道为 16，输出为 16，卷积核为 2，步长为 2
        self.s4=nn.AvgPool2d(kernel_size=2,stride=2)
        # 卷积层，输入大小为 5*5，输出大小为 1*1，输入通道为 16，输出为 120，卷积核为 5
        self.c5=nn.Conv2d(in_channels=16,out_channels=120,kernel_size=5)
        print(self.c5.weight.shape)
        # Flatten()：将张量（多维数组）平坦化处理，张量的第 0 维表示的是 batch_size（数量），所以 Flatten()默认从第二维开始平坦化，也就是说将一个样本的 通道数*长*宽，乘起来之后作为第一个全连接层的输入。
        self.flatten=nn.Flatten()
        # Linear（in_features，out_features）
        # in_features 指的是[batch_size, size]中的 size,即样本的大小， 通道数*长*宽
        # out_features 指的是[batch_size，output_size]中的 output_size，样本输出的维度大小，也代表了该全连接层的神经元个数
        self.f6=nn.Linear(in_features=120,out_features=84)
# 最终的 output 层则是网络输出定义的数量，比如分为 10 类，则输出值为 10，以此类推。
        self.output=nn.Linear(in_features=84,out_features=10)

    # forward()：定义前向传播过程,描述了数据 X 在模型网络各层之间的流动顺序
    def forward(self,x):
        # x 输入为 28*28*1， 输出为 28*28*6
```

```
        x=self.Sigmoid(self.c1(x))
        # x 输入为 28*28*6，输出为 14*14*6
        x=self.s2(x)
        # x 输入为 14*14*6，输出为 10*10*16
        x=self.Sigmoid(self.c3(x))
        # x 输入为 10*10*16，输出为 5*5*16
        x=self.s4(x)
        # x 输入为 5*5*16，输出为 1*1*120
        x=self.c5(x)
        x=self.flatten(x)
        # x 输入为 120，输出为 84
        x=self.f6(x)
        # x 输入为 84，输出为 10
        x=self.output(x)
        return x

        # 测试代码
#一个 python 文件通常有两种使用方法，第一是作为脚本直接执行，第二是 import 到其他的 python 脚本中被调用（模块重用）执行。因此 if __name__ == '__main__': 的作用就是控制这两种情况执行代码的过程，在 if __name__ == '__main__': 下的代码只有在第一种情况下（即文件作为脚本直接执行）才会被执行，而 import 到其他脚本中是不会被执行的。
        # 每个 python 模块（python 文件）都包含内置的变量 __name__，当该模块被直接执行的时候，__name__ 等于文件名（包含后缀 .py ）
        # 如果该模块 import 到其他模块中，则该模块的 __name__ 等于模块名称（不包含后缀.py）
        #“__main__”这个变量名称始终指当前执行模块的名称（包含后缀.py）
        # if 确保只有单独运行该模块时，此表达式才成立，才可以进入此判断语法，执行其中的测试代码，反之不行
# if __name__ == "__main__":
#                 # rand:返回一个张量，包含了从区间[0, 1)的均匀分布中抽取的一组随机数，此处为四维张量
#         x = torch.rand([1, 1, 28, 28])
#                 # 模型实例化
#         model = LeNet()
#         y = model(x)
```

模型训练

上图是模型训练的基本框架，其中红色字体的是最基础的必不可少的部分，其它的部分是为了优化训练和可视化附加的，写代码时只需要按照这个框架往里填充内容即可。

```
import torch
from torch import nn

#从自己创建的 models 库里导入 LeNet 模块

#import LeNet  仅仅是把 LeNet.py 导入进来,当我们创建 LeNet 的实例的时候需要通过指定 LeNet.py 中的具体类.

#例如:我的 LeNet.py 中的类名是 LeNet,则后面的模型实例化 LeNet 需要通过 **LeNet.LeNet()**来操作
#还可以通过  from  还可以通过  from LeNet import *  直接把 LeNet.py 中除了以 _ 开头的内容都导入

from models import LeNet
from LeNet import *
# lr_scheduler：提供一些根据 epoch 训练次数来调整学习率的方法
from torch.optim import lr_scheduler
# torchvision：服务于 PyTorch 深度学习框架的的一个图形库，主要用来构建计算机视觉模型
```

```
# transforms：主要是用于常见的一些图形变换
# datasets：包含加载数据的函数及常用的数据集接口
from torchvision import datasets,transforms
# os：operating system（操作系统），os 模块封装了常见的文件和目录操作
import os

# 设置数据转化方式，如数据切割，转化为 Tensor 格式等
# Compose()：将多个 transforms 的操作整合在一起
# ToTensor(): 将 numpy 的 ndarray 或 PIL.Image 读的图片转换成形状为(C,H, W)的 Tensor 格式，且归一化到[0,1.0]之间
data_transform=transforms.Compose([transforms.ToTensor()])


# 加载训练数据集的方法之经典数据集，下篇会介绍另两种数据集创建方法
# MNIST 数据集来自美国国家标准与技术研究所, 训练集 (training set)、测试集(test set)由分别由来自 250 个不同人手写的数字构成
# MNIST 数据集包含：Training set images、Training set images、Test set images、Test set labels
# train = true 是训练集，false 为测试集

train_dataset=datasets.MNIST(root='./data',train=True,transform=data_transform,download=True)
# DataLoader：将读取的数据按照 batch size 大小封装并行训练
# dataset (Dataset)：加载的数据集
# batch_size (int, optional)：每个 batch 加载多少个样本(默认: 1)
# shuffle (bool, optional)：设置为 True 时会在每个 epoch 重新打乱数据(默认: False)
train_dataloader=torch.utils.data.DataLoader(dataset=train_dataset,batch_size=16,shuffle=True)

# 加载测试数据集
test_dataset=datasets.MNIST(root='./data',train=False,transform=data_transform,download=True)
test_dataloader=torch.utils.data.DataLoader(dataset=test_dataset,batch_size=16,shuffle=True)
# 如果有 NVIDA 显卡，转到 GPU 训练，否则用 CPU
device='cuda' if torch.cuda.is_available() else 'cpu'


# 模型实例化，将模型转到 device
model=LeNet.LeNet()#.to(device)
# 定义损失函数（交叉熵损失）
loss_fn=nn.CrossEntropyLoss()
# 定义优化器(随机梯度下降法)
# params(iterable)：要训练的参数，一般传入的是 model.parameters()
# lr(float)：learning_rate 学习率，也就是步长
```

```
# momentum(float, 可选)：动量因子（默认：0），矫正优化率
optimizer=torch.optim.SGD(model.parameters(),lr=1e-3,momentum=0.9)
# 学习率，每隔 10 轮变为原来的 0.1
# StepLR：用于调整学习率，一般情况下会设置随着 epoch 的增大而逐渐减小学习率从而达到更好的训练效果
# optimizer （Optimizer）：需要更改学习率的优化器
# step_size（int）：每训练 step_size 个 epoch，更新一次参数
# gamma（float）：更新 lr 的乘法因子
lr_scheduler=lr_scheduler.StepLR(optimizer,step_size=10,gamma=0.1)

# 定义训练函数，参数包括数据集，模型，损失函数和优化函数
def train(dataloader,model,loss_fn,optimizer):
    loss,current,n=0.0,0.0,0
    # dataloader: 传入数据（数据包括：训练数据和标签）
    # enumerate()：用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，一般用在 for 循环当中
    # enumerate 返回值有两个：一个是序号，一个是数据（包含训练数据和标签）
    # x：训练数据（inputs）(tensor 类型的），y：标签（labels）(tensor 类型的）
    for batch,(x,y) in enumerate(dataloader):
        # 前向传播
        x,y=x.to(device),y.to(device)
        # 计算训练值
        output=model(x)
        # 计算标签值（label）与预测值的损失函数，这里需要注意一下根据损失函数的类别，预测值和标签值的维度。比如这里是交叉熵损失函数，则 output 是 16*10 张量，而 y 是 1*10 的张量
        cur_loss=loss_fn(output,y)
        # torch.max(input, dim)函数
        # input 是具体的 tensor，dim 是 max 函数索引的维度，0 是每列的最大值，1 是每行的最大值输出
        # 函数会返回两个 tensor，第一个 tensor 是每行的最大值；第二个 tensor 是每行最大值的索引
        _,pred=torch.max(output,axis=1)
        # 计算每批次的准确率
        # output.shape[0]一维长度为该批次的数量
        # torch.sum()对输入的 tensor 数据的某一维度求和
        cur_acc=torch.sum(y==pred)/output.shape[0]
        # 反向传播
        # 清空过往梯度
        optimizer.zero_grad()
        # 反向传播，计算当前梯度
        cur_loss.backward()
        # 根据梯度更新网络参数
```

```
            optimizer.step()
            # .item()：得到元素张量的元素值
            loss+=cur_loss.item()
            current+=cur_acc.item()
            n=n+1

        train_loss=loss/n
        train_acc=current/n
        # 计算训练的错误率
        print('train_loss'+str(train_loss))
        # 计算训练的准确率
        print('train_acc'+str(train_acc))

# 定义验证函数
def val(dataloader,model,loss_fn):
        # model.eval()：设置为验证模式，如果模型中有 Batch Normalization 或 Dropout，则不启用，以防改变权值
        model.eval()
        loss,current,n=0.0,0.0,.0
        # with torch.no_grad()：将 with 语句包裹起来的部分停止梯度的更新，从而节省了 GPU 算力和显存，但是并不会影响 dropout 和 BN 层的行为
        with torch.no_grad():
            for batch,(x,y) in enumerate(dataloader):
                # 前向传播
                x,y=x.to(device),y.to(device)
                output=model(x)
                cur_loss=loss_fn(output,y)
                _,pred=torch.max(output,axis=1)
                cur_acc=torch.sum(y==pred)/output.shape[0]
                loss+=cur_loss.item()
                current+=cur_acc.item()
                n=n+1
            # 计算验证的错误率
            print("val_loss:"+str(loss/n))
            # 计算验证的准确率
            print("val_acc：" + str(current / n))
            # 返回模型准确率
            return current / n
# 开始训练
# 训练次数
epoch=10
# 用于判断最佳模型
min_acc=0
for t in range(epoch):
```

```
        print(f'epoch {t+1}\n --------')
        # 训练模型
        train(train_dataloader,model,loss_fn,optimizer)
        # 验证模型
        a=val( test_dataloader,model,loss_fn)
        # 保存最好的模型权重
        if a>min_acc:
            folder='save_model'
            # path.exists：判断括号里的文件是否存在，存在为 True，括号内可以是文件路径
            if not os.path.exists(folder):
                # os.mkdir() ：用于以数字权限模式创建目录
                os.mkdir('save_model')
            min_acc=a
            print('save better model')
            # torch.save(state, dir)保存模型等相关参数，dir 表示保存文件的路径+保存文件名
            # model.state_dict()：返回的是一个 OrderedDict，存储了网络结构的名字和对应的参数

#pytorch 保存模型参数权重的核心句子。
            torch.save(model.state_dict(),'save_model/better_model.pth')

print('Done!')
```

模型预测与应用

这里只是最简单的应用，模型训练完成之后最终的目的应当是为了在实际中应用，在介绍完基本模型之后，也将结合大模型和自动驾驶探讨如何具体的应用。

```
import torch
from torch import nn
from models import LeNet
from models.LeNet import *
from torch.autograd import Variable
from torchvision import datasets,transforms
from torchvision.transforms import ToPILImage

# Compose()：将多个 transforms 的操作整合在一起
data_transform=transforms.Compose([
    transforms.ToTensor()
])

# 加载训练数据集
train_dataset = datasets.MNIST(root='./data', train=True, transform=data_transform,
```

```
download=True)
# 给训练集创建一个数据加载器, shuffle=True 用于打乱数据集，每次都会以不同的顺序返回
train_dataloader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=16, shuffle=True)

# 加载测试数据集
test_dataset = datasets.MNIST(root='./data', train=False, transform=data_transform, download=True)
test_dataloader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=16, shuffle=True)

# 如果有 NVIDA 显卡，转到 GPU 训练，否则用 CPU
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# 模型实例化，将模型转到 device
model = LeNet().to(device)

# 加载 train.py 里训练好的模型
model.load_state_dict(torch.load("../train/save_model/better_model.pth"))

# 结果类型
classes = [
    "0",
    "1",
    "2",
    "3",
    "4",
    "5",
    "6",
    "7",
    "8",
    "9",
]

# 把 Tensor 转化为图片，方便可视化
show = ToPILImage()

# 进入验证阶段
for i in range(10):
    x, y = test_dataset[i][0], test_dataset[i][1]
    # show()：显示图片
    show(x).show()
    # unsqueeze(input, dim)，input(Tensor)：输入张量，dim (int)：插入维度的索
```

引，最终将张量维度扩展为 4 维

```
x = Variable(torch.unsqueeze(x, dim=0).float(), requires_grad=False).to(device)
with torch.no_grad():
    pred = model(x)
    # argmax(input)：返回指定维度最大值的序号
    # 得到验证类别中数值最高的那一类，再对应 classes 中的那一类
    predicted, actual = classes[torch.argmax(pred[0])], classes[y]
    # 输出预测值与真实值
    print(f'predicted: "{predicted}", actual:"{actual}"')
```

参考资料

https://zh.d2l.ai/index.html

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

不如语冰