课内实验2  
建立全连接神经网络

指导教师：刘胥影 助教：田秋雨

Due：2024年4月24日

## 任务描述

通过两种方式实现全连接神经网络，并对图片分类任务行进行测试与实验。

1. 手动实现简单的全连接神经网络
2. 使用Pytorch库简洁实现全连接神经网络

Fashion-MNIST图片分类数据集 [1] 包含10个类别的时装图像，训练集有60,000张图片，测试集中有10,000张图片。图片为灰度图片，高度（）和宽度（）均为28像素，通道数（channel）为1。10个类别分别为：t-shirt(T恤), trouser（裤子）, pullover（套衫）, dress（连衣裙）, coat（外套）， sandal（凉鞋），shirt（衬衫），sneaker（运动鞋），bag（包），ankle boot（短靴）。使用训练集数据进行训练，测试集数据进行测试。

## 教学要求

1. 掌握多层前馈神经网络及BP算法的原理与构建
2. 了解PyTorch库，掌握本实验涉及的相关部分
3. 进行参数分析实验，理解学习率等参数的影响

## 实验要求

### 使用Python编程构建手动实现单隐层全连接神经网络

**模型架构**

输入层28\*28=784个节点，输出层10个节点，隐藏层256个节点。  
注意，可以将这两个变量都视为超参数。 通常选择2的若干次幂作为层的宽度。 因为内存在硬件中的分配和寻址方式，这么做往往可以在计算上更高效。

激活函数：ReLU函数

损失函数：Cross entropy

性能指标：准确率

优化算法：实现标准BP或小批量梯度下降算法均可

**实现内容**

1. 初始化模型参数  
   对于每一层都要记录一个权重矩阵和一个偏置向量。
2. 设置激活函数  
   使用ReLU函数作为激活函数，要求手动实现该函数。
3. 前向计算  
   实现该函数。

注意：需要将每个二维图像转化为向量进行操作。

1. 设置损失函数  
   使用cross entropy作为损失函数。  
   可以自己手动实现，也可以直接调用nn.CrossEntropyLoss函数。
2. 训练模型
   1. 实现训练函数  
      该训练函数将会运行多个迭代周期（由num\_epochs指定）。 在每个迭代周期结束时，利用test\_iter访问到的测试数据集对模型进行评估。利用后面给出的Animator类来可视化训练进度。
   2. 可以使用PyTorch内置的优化器（torch.optim.SGD），也可以使用自己定制的优化器。
   3. 可以调用torch.optim.SGD函数进行参数更新
   4. 迭代周期数epoch设置为10，学习率设置为0.1，训练模型
3. 设置性能函数  
   使用准确率accuracy作为性能指标。  
   实现该函数。
4. 模型评估
   1. 对测试集数据进行测试
   2. 进行性能评估
5. 参数分析实验
   1. 在所有其他参数保持不变的情况下，更改超参数num\_hiddens的值，并查看此超参数的变化对结果有何影响。确定此超参数的最佳值。
   2. 改变学习速率会如何影响结果？保持模型架构和其他超参数（包括轮数）不变，学习率设置为多少会带来最好的结果？

### 使用PyTorch库简洁实现全连接神经网络

手动实现一个简单的多层神经网络是很容易的。然而如果网络有很多层，从零开始实现会变得很麻烦。可以使用高级API如PyTorch库简洁实现。

1. 请使用PyTorch库简洁实现前述的全连接神经网络，并进行模型评估。
   1. 优化器：使用torch.optim.SGD
   2. 小批量数据载入函数参见提供的代码。
2. 参数分析实验
3. 尝试添加不同数量的隐藏层（也可以修改学习率），怎么样设置效果最好？
4. 尝试不同的激活函数，哪个效果最好？
5. 尝试不同的方案来初始化权重，什么方法效果最好？

### 提交代码与报告

其中报告内容包括以下几个部分：

1. 手动实现单隐层全连接神经网络
2. 训练过程中，训练集与验证集误差随epoch变化的曲线图
3. 性能评估结果
4. 参数分析实验：包括实验设置与结果分析
5. 使用PyTorch库简洁实现全连接神经网络
6. 性能评估结果
7. 参数分析实验：包括实验设置与结果分析

## 提交要求

1. 超参数设置单独新建一个config.py，使用argparse库。
2. 模型文件为.pth。预测函数另外写在predict.py中。
3. 可视化结果保存在一个文件夹中。
4. 提交文件为代码、模型、可视化结果所在文件夹的压缩文件。

## 代码参考

1. 初始化模型参数

num\_inputs, num\_outputs, num\_hiddens = 784, 10, 256

W1 = nn.Parameter(torch.randn(

num\_inputs, num\_hiddens, requires\_grad=**True**) \* 0.01)

b1 = nn.Parameter(torch.zeros(num\_hiddens, requires\_grad=**True**))

W2 = nn.Parameter(torch.randn(

num\_hiddens, num\_outputs, requires\_grad=**True**) \* 0.01)

b2 = nn.Parameter(torch.zeros(num\_outputs, requires\_grad=**True**))

params = [W1, b1, W2, b2]

1. 训练进度可视化类Animator

**class** **Animator**: *#@save*

*"""在动画中绘制数据"""*

**def** \_\_init\_\_(self, xlabel=**None**, ylabel=**None**, legend=**None**, xlim=**None**,

ylim=**None**, xscale='linear', yscale='linear',

fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,

figsize=(3.5, 2.5)):

*# 增量地绘制多条线*

**if** legend **is** **None**:

legend = []

d2l.use\_svg\_display()

self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)

**if** nrows \* ncols == 1:

self.axes = [self.axes, ]

*# 使用lambda函数捕获参数*

self.config\_axes = **lambda**: d2l.set\_axes(

self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)

self.X, self.Y, self.fmts = **None**, **None**, fmts

**def** add(self, x, y):

*# 向图表中添加多个数据点*

**if** **not** hasattr(y, "\_\_len\_\_"):

y = [y]

n = len(y)

**if** **not** hasattr(x, "\_\_len\_\_"):

x = [x] \* n

**if** **not** self.X:

self.X = [[] **for** \_ **in** range(n)]

**if** **not** self.Y:

self.Y = [[] **for** \_ **in** range(n)]

**for** i, (a, b) **in** enumerate(zip(x, y)):

**if** a **is** **not** **None** **and** b **is** **not** **None**:

self.X[i].append(a)

self.Y[i].append(b)

self.axes[0].cla()

**for** x, y, fmt **in** zip(self.X, self.Y, self.fmts):

self.axes[0].plot(x, y, fmt)

self.config\_axes()

display.display(self.fig)

display.clear\_output(wait=**True**)

1. argparse解析超参数

**config.py**

import argparse

parser = argparse.ArgumentParser(description='Hyper-parameters management')

# Hardware options

parser.add\_argument('--n\_threads', type=int, default=6,

help='number of threads for data loading')

parser.add\_argument('--cpu', action='store\_true',

help='use cpu only')

parser.add\_argument('--seed', type=int, default=1, help='random seed')

# data in/out and dataset

parser.add\_argument('--dataset\_path',default = r'./fixed\_data/',help='fixed trainset root path')

parser.add\_argument('--save',default=’model’,help='save path of trained model')

parser.add\_argument('--predict',default=’model’,help='save path of predict model')

parser.add\_argument('--batch\_size', type=list, default=4,help='batch size of trainset')

# train

parser.add\_argument('--epochs', type=int, default=500, metavar='N',help='number of epochs to train (default: 10)')

parser.add\_argument('--lr', type=float, default=0.01, metavar='LR',help='learning rate (default: 0.01)')

parser.add\_argument('--momentum', type=float, default=0.99, metavar='M',help='SGD momentum (default: 0.5)')

parser.add\_argument('--weight\_decay', type=float, default=3e-5, metavar='W',help='SGD weight\_decay (default: 3e-5)')

parser.add\_argument('--nesterov', type=bool, default=True, help='SGD nesterov (default: True)')

parser.add\_argument('--early-stop', default=20, type=int, help='early stopping (default: 20)')

#args = parser.parse\_args()

args, unknown = parser.parse\_known\_args()

**train.py**

import config

import torch

args=config.args

device=torch.device(‘cpu’ if args.cpu else ‘cuda’)

1. 读入数据需要重写torch中的dataset类

class my\_dataset(torch.utils.data.Dataset):

def \_\_init\_\_(self, x, y, transform=None, target\_transform=None):

# pytorh的训练集必须是tensor形式，可以直接在dataset类中转换，省去了定义transform

# 转换Y数据类型为长整型

self.point = torch.from\_numpy(x).type(torch.FloatTensor)

self.label = torch.from\_numpy(y).type(torch.FloatTensor)

self.transform = transform

self.target\_transform = target\_transform

def \_\_getitem\_\_(self, index):

x = self.point[index]

label = self.label[index]

# print(x)

return x, label

def \_\_len\_\_(self):

return len(self.label)

train\_data = my\_dataset(X, Y)

train\_loader = data.DataLoader(train\_data, batch\_size=args.bacth\_size, shuffle=True)

5.模型需要继承nn.Module

class Classifier(nn.Module):

# 初始化函数，对网路的输入层、隐含层、输出层的大小和使用的函数进行了规定。

def \_\_init\_\_(self, input\_size=2, hidden\_layersize=4):

super(Classifier, self).\_\_init\_\_()

#yourconde

def forward(self, x):

#your code

return x