> ⚠ **Warning**
>
> 未经授权禁止传播。

源仓库

HW1

全程干货！

部分参考代码有给注释，这里便不赘述

Part 0 如何善用搜索自学

> **Important**
>
> 在大学，很多东西没人指导你学什么、怎么学，这时候，我们需要一些自学的方法可途径。

1. **问AI。**在知识储备方面，ai是最好的老师。注意，问不是照抄。
2. **捞**相应的方面的有经验同学帮助。
3. **网上搜索。**关于体系化的知识，比如“Python学习”，可以找网课、文档，比如上b站找，就有很好的网课。文档方面有例如廖雪峰博客、菜鸟教程，都很不错。
4. **看书。**在这个时代，这个方法已经不太推荐了，只有当你学的赛道比较新才可能用到。

Part 1 Python简介

编程语言

通过指令让计算机实现一定操作，python比C更符合人的语言直觉。大多编程语言从语法上是互通的，当你了解其一后，可善用搜索去学习另一语言的语法。

Python语法

建议自学廖雪峰博客。我这里只介绍我们可能用到的比较基础的，关于类、迭代器等特性，我就不说了，可以自学，但在这个作业中没必要。

代码块表示

C语言中是通过花括号{}来表示一块代码的，而python里面区别代码块的机制很特殊，是通过段前缩进来实现的，连续一段代码缩进相同的，就代表是一个代码块。

数据类型List（列表）：

> **Note**
>
> python变量声明极其随意，不需要指明数据类型，编译器自己判断这个变量有没有存在过、需要什么数据类型

用于储存一系列变量，变量类型随便。示例：

```
myLossList = [0.1,0.2,0.24]
```

循环

循环写法

```
for <variable> in <sequence>:
    <statements>
else:
    <statements>
#示例
for i in [1,2,3,4,5]
    print(i)
else:
    print(0)
```

函数

用于执行某种功能的代码块，定义为

```
def 函数名（参数列表）:
    函数体
# 例如
def hello() :
    print("Hello World!")

hello()
```

传入参数不多讲，注意：如果传入采用比如`f(x = 2)`就是指定了 x这个形参对应的实参值为2

类

用来描述具有相同的属性和函数的对象的集合。它定义了该集合中每个对象所共有的属性和函数。对象是类的实例。

可以把类作“类别”理解

类中的函数，我们叫方法

类是抽象、不具体的，因此我们在使用时，需要实例化。

调用类中的属性或者方法，用句点，例如：实例.属性/方法

```
class Student:
    """一个简单的类实例"""
    #id是类的属性，属性就是类中的变量，比如学号、姓名
    id = 12345
    name = 'Mary'
    #f是类中的函数，即方法，要求类中的方法必须传入self参数。
    #self参数代表传入的实例，在这里就是x，这样函数才知道是那个实例调用了它（这里体现不出作用，实际会有用）
    def say(self):
        return 'hello world'

# 实例化类，就是搞一个实在实在的变量储存它
```

```
Student1 = Student()

# 访问类的属性和方法
print("Student 类的属性 id 为：", Student1.id)
print("Student 类的方法 say 输出为：", Student1.say())
```

类还可以有初始化函数，这个函数在类实例的时候会被调用

```
#复数类
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart #实际上，这里定义了一个变量，表示实部
        self.i = imagpart
x = Complex(3.0, -4.5)
print(x.r) #获取实部，输出3.0
print(x.i) #获取虚部，输出-4.5
```

父类

我们知道，分类经常用二叉法或多叉法，一类之下还有很多细分，比如界门纲目科属种。

如果一个类B属于更泛的一种类A，我就说B继承了A，A是B的父类。子类具有父类的特性，又有自己单独的特性

```
#类定义
class people:
    #定义基本属性
    name = ''
    age = 0
    #定义构造方法，构造方法可以传入参数
    def __init__(self,n,a,w):
        self.name = n
        self.age = a
    def speak(self):
        print("%s 说: 我 %d 岁。" %(self.name,self.age))
    def say(self):
        print('说话! ')

#继承示例
class student(people):
    grade = ''
    def __init__(self,n,a,w,g):
        #调用父类的构函,即初始化函数
        people.__init__(self,n,a,w)
        self.grade = g
    #覆写父类的方法，即重写了这个方法，使他跟符合当前类
    def speak(self):
        print("%s 说: 我 %d 岁了，我在读 %d 年级"%(self.name,self.age,self.grade))

s = student('ken',10,60,3)
s.speak()
s.say() #可以调用父类的方法，这是没覆写的
```

数据类型的方法

python中数据类型底层就是个类，我们定义变量，实际就是实例化类。所以，我们可以调用类中的方法很方便的进行一些操作（python官方给我们写好的函数）

以列表为例：

序号	方法
1	list.append(obj) 在列表末尾添加新的对象
2	list.count(obj) 统计某个元素在列表中出现的次数
3	list.extend(seq) 在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）
4	list.index(obj) 从列表中找出某个值第一个匹配项的索引位置
5	list.insert(index, obj) 将对象插入列表
6	list.pop(index=-1) 移除列表中的一个元素（不传参数则默认最后一个元素），并且返回该元素的值
7	list.remove(obj) 移除列表中某个值的第一个匹配项
8	list.reverse() 反向列表中元素
9	list.sort(key=None, reverse=False) 对原列表进行排序
10	list.clear() 清空列表
11	list.copy() 复制列表

模块

如何将写好的代码在其他文件里面使用而不直接复制呢?

为此 Python 提供了一个办法，把这些定义存放在文件中，为一些脚本或者交互式的解释器实例使用，这个文件被称为模块。

Python 中的模块（Module）是一个包含 Python 定义和语句的文件，文件名就是模块名加上 **.py** 后缀。

模块可以包含函数、类、变量以及可执行的代码。通过模块，我们可以将代码组织成可重用的单元，便于管理和维护。

你在电脑上装好python3后，会自动有官方给你的一系列模块，你直接用就行了（云平台以及搭建了python3，你更不用操心），下面来讲怎么用。

import 语句

导入相应模块整个文件，例如`import torch`

from .. import语句

导入文件内特定部分，比如特定的函数（下面的modname是模块名，name1是你想导入的东西，比如函数或变量）

```
from modname import name1,name2...
```

特别的

```
from modname import *
```

表示导入所有东西

此外

```
from modname import name1 as myname
```

就是给name1起了个别名myname

Part 2 开始写作业

> **Note**
>
> 所有神经网络的实现类、方法官方都给我们写好了，我们只要会用或者看懂怎么用就行

导入数据

你只要知道导入了一些东西就行，具体是什么不需要了解

```
import torch
import torchvision
import torchvision.transforms as transforms
from torchvision.transforms import ToPILImage
show = ToPILImage()
```

划分数据

这一部分同样只要知道在干什么，不需要深究什么细节

```
# 设定对图片的归一化处理方式，并且下载数据集
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
     ])
#定义单次训练图片数量
batch_size = 4
#设置训练集 root是数据所在文件目录，train=True表示这是训练集
trainset = torchvision.datasets.CIFAR10(root='./dataset', train=True,
                                        download=True, transform=transform)
#训练集加载器
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
```

```
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./dataset', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)
```

查看数据

这个简单看一下就行了

```
# 观察一下数据集的内容
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck') # 类别名称
print(len(trainset)) # 训练集大小,用的是len()函数
print(trainset[0][0].size()) # 第 1 条数据的图像大小，用的是类的方法
print(trainset[0][1]) # 第 1 条数据的标签，数组类型
print(classes[trainset[0][1]]) # 第 1 条数据的文本标签
```

定义神经网络

卷积层、池化层、全连接层、展平等如果不清楚的话就理解成一种神经网络中操作数据的方式。

这是老师给的参考代码，只要知道每行干什么的就行，不是要你自己会完整写。

```
import torch.nn as nn
import torch.nn.functional as F

# torch是模块 nn是模块下的一个模块，注意

#网络就是一个类，它属于父类Module，就是神经网络模型，我们要更加具体得来实现
class Net(nn.Module):
    #初始化函数，实例化的时候会调用
    def __init__(self):
        # nn.Module子类的函数必须在构造函数中执行父类的构造函数
        super(Net, self).__init__()

        # 下面都出现了self.xx = nn.xx() 前面说了 nn是模块，nn.xx()就是在调用模块中的类，并传入参数初始化

        # 卷积层 '3'表示输入图片为3通道(RGB)， '6'表示输出通道数，'5'表示卷积核为5*5
        self.conv1 = nn.Conv2d(3, 6, 5)
        # 卷积层
        self.conv2 = nn.Conv2d(6, 16, 5)
        # 仿射层/全连接层，y = Wx + b
        self.fc1   = nn.Linear(16*5*5, 120)
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

        #x是传入的来训练的数据，下文train有讲
    def forward(self, x):
        # 卷积 -> 激活 -> 池化 (relu激活函数不改变输入的形状)
```

```
        # [batch size, 3, 32, 32] -- conv1 --> [batch size, 6, 28, 28] -- maxpool --> [batch
size, 6, 14, 14]
        #这里嵌套了函数，注意
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # [batch size, 6, 14, 14] -- conv2 --> [batch size, 16, 10, 10] --> maxpool -->
[batch size, 16, 5, 5]
        #这里嵌套了函数，注意
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        # 把 16 * 5 * 5 的特征图展平，变为 [batch size, 16 * 5 * 5]，以送入全连接层
        x = x.view(x.size()[0], -1)
        # 全连接层1
        # [batch size, 16 * 5 * 5] -- fc1 --> [batch size, 120]
        x = F.relu(self.fc1(x))
        # 全连接层2
        # [batch size, 120] -- fc2 --> [batch size, 84]
        x = F.relu(self.fc2(x))
        # 全连接层3
        # [batch size, 84] -- fc3 --> [batch size, 10]
        x = self.fc3(x)
        return x

#实例化你的网络，准备训练
net = Net()
print(net)
```

训练

训练一个网络，我们需要有用于计算损失的损失函数，用于决定怎么更新参数优化器，训练次数以及把以上过程组织起来的训练函数

优化器

用于在神经网络训练过程中**更新模型的参数**，决定了**如何根据损失函数的梯度来更新模型参数**（不给的话无从下手，往那个方向？），就是一种操作数据的方式，这里采用SGD(官方给的)。

选择损失函数

形如一元线性回归误差的定义，只不过我们用的定义比较复杂。总之就是真实值于理论值的偏差

准备阶段代码

```
from torch import optim
criterion = nn.CrossEntropyLoss() # 交叉熵损失函数，官方给好的
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9) # 使用SGD（随机梯度下降）优化，
不需要知道传入的第一个参数干嘛的。需要知道lr:learning rate 学习率，决定学习步长；momentum 动量大小，或者
理解为惯性大小，即保持变化一定时间内不变，可以帮助跳出局部最优解
num_epochs = 5 #定义训练 epoch 的数量，每个epoch 训练12000次
```

训练用的函数

```
#传入参数应该好理解吧，之前我们都讲到了（除了save_path，这个下文就是了）
def train(trainloader, net, num_epochs, criterion, optimizer, save_path):
    for epoch in range(num_epochs):
        #用来储存损失值
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):

            # 1. 取出数据
            inputs, labels = data

            # 梯度清零，就像写草稿，我们要擦掉上一次计算留下来的痕迹，不然就乱了
            optimizer.zero_grad()

            # 2. 前向计算和反向传播
            outputs = net(inputs) # 送入网络（正向传播）
            loss = criterion(outputs, labels) # 计算损失函数

            # 3. 反向传播，更新参数
            loss.backward() # 反向传播
            optimizer.step()

            # 下面的这段代码对于训练无实际作用，仅用于观察训练状态
            running_loss += loss.item() #迭代器取出损失值，只要知道它让我们得到了损失值就行

            if i % 2000 == 1999: # 每2000个batch打印一下训练状态
                # 格式化输出了一串字符串
                print('epoch %d: batch %5d loss: %.3f' \
                      % (epoch+1, i+1, running_loss / 2000))
                #清零方便下次循环统计
                running_loss = 0.0
        #储存权重文件
        torch.save(net.state_dict(), f"{save_path}/epoch_{epoch + 1}_model.pth")

    print('Finished Training')
```

权重文件与save_path

```
# 使用定义的网络进行训练
save_path = '<input your path>'
train(trainloader, net, num_epochs, criterion, optimizer, save_path)
```

save_path是一个字符串，储存要保存权重文件（python字符串可以单引号）

什么是权重文件

权重文件是保存神经网络模型**所有参数值**的文件，也称为模型文件或检查点文件，就是保存"学到的知识"的地方。用于助教检查，我们不负责操作它，只需要保存它。

要求每个训练的权重文件单独储存，因此每次调用train()前，我们需要更改save_path的值，第一次基本训练，我们就这样改:

`save_path = './model'`

代表存储权重文件在代码当前文件夹的子级文件夹model下

> **Caution**
>
> 需要保证存储的文件夹存在！！！因此我们要先创建model文件夹，不然会报错
>
> 当然，你可以在定义train函数里面第一行加上一句代码:
>
> `os.makedirs(save_path, exist_ok=True)`
>
> 并且在train函数定义前加上:
>
> `import os`
>
> 意思是利用os pack来创建文件夹

Task1 损失函数绘制

提供的代码

```
import matplotlib.pyplot as plt

def draw(values):
    plt.plot(values)
    plt.show()

draw([0.1, 0.2, 0.4, 0.8])
```

可见用到了matplotlib。自学matplotlib，你可以实现一图多线、图例、颜色改变等等可视化操作。

这里简单说明:

`plt.plot(x坐标列表,y坐标列表)`

draw()这个函数传入values参数，这是个List(列表)

所以我们只要传入每个epoch的损失函数平均值就行了

这就需要我们定义一个变量，在train中每个epoch结束后将损失函数平均值加入列表中，这里用到了list中的方法`append(待加入的loss值)`

注：batch表示单次训练；epoch表示12000次训练

示例

```
# 改写你的train函数
# 定义变量 =[]是为了告诉编译器这是个列表，不过目前内容空
```

```
trainloss = []
def train(trainloader, net, num_epochs, criterion, optimizer, save_path):
    trainloss.clear()#用clear方法清空一下列表
    for epoch in range(num_epochs):
        #用来储存损失值,这是每2000次的
        running_loss = 0.0
        #用来储存损失值,这是每12000次的
        myloss = 0.0
        for i, data in enumerate(trainloader, 0):

            # 1. 取出数据
            inputs, labels = data

            # 梯度清零，就像写草稿，我们要擦掉上一次计算留下来的痕迹，不然就乱了
            optimizer.zero_grad()

            # 2. 前向计算和反向传播
            outputs = net(inputs) # 送入网络（正向传播）
            loss = criterion(outputs, labels) # 计算损失函数

            # 3. 反向传播，更新参数
            loss.backward() # 反向传播
            optimizer.step()

            # 下面的这段代码对于训练无实际作用，仅用于观察训练状态
            running_loss += loss.item() #迭代器取出损失值，只要知道它让我们得到了损失值就行

            if i % 2000 == 1999: # 每2000个batch打印一下训练状态
                # 格式化输出了一串字符串
                print('epoch %d: batch %5d loss: %.3f' \
                      % (epoch+1, i+1, running_loss / 2000))
                myloss+=running_loss
                #清零方便下次循环统计
                running_loss = 0.0
        #每一个epoch就储存一下loss
        trainloss.append(myloss / 12000)
        #储存权重文件
        torch.save(net.state_dict(), f"{save_path}/epoch_{epoch + 1}_model.pth")

    print('Finished Training')
```

然后画图

```
import matplotlib.pyplot as plt

def draw():
    plt.plot([1,2,3,4,5],trainloss)
    plt.show()

draw()
```

Task 2 正则化

> L2 正则化：请查阅Pytorch有关SGD优化器的文档或其它网络资料，修改`3. 模型训练与测试过程`中的代码，尝试为模型的损失函数加入一项L_2损失，并在报告中说明你所做的修改。
>
> Dropout正则化：请查阅Pytorch有关Dropout层的文档或其它网络资料，修改`2. 定义用于分类的网络结构`中的代码，在**第一个线性层和第二个线性层之间**加入一个Dropout层，并在报告中说明你所做的修改。

这个文档都给你了。简单来说:

L2正则化直接在SGD优化器中加参数weight_decay

```
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9,weight_decay = 1e-4)
```

Dropout则需要在新Net中添加一层，具体怎么实现不需要知道，因为官方给你封装好了函数`Dropout()`

```
# TODO: 在Dropout_Net中加入dropout层
class Dropout_Net(nn.Module):
    #实例化时调用的函数
    def __init__(self):
        # nn.Module子类的函数必须在构造函数中执行父类的构造函数
        super(Dropout_Net, self).__init__()

        # 卷积层 '1'表示输入图片为单通道, '6'表示输出通道数, '5'表示卷积核为5*5
        self.conv1 = nn.Conv2d(3, 6, 5)
        # 卷积层
        self.conv2 = nn.Conv2d(6, 16, 5)
        # 仿射层/全连接层，y = Wx + b
        self.fc1   = nn.Linear(16*5*5, 120)
        # 加载Dropout 失活概率p
        self.dropout = nn.Dropout(p=0.2)

        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    def forward(self, x):
        # 卷积 -> 激活 -> 池化
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        # reshape，‘-1’表示自适应
        x = x.view(x.size()[0], -1)
        x = F.relu(self.fc1(x))

        #fc1和fc2间加入Dropout层
        x = self.dropout(x)

        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
dropout_net = Dropout_Net()
```

> **Caution**

> dropout网络的实例化名为dropout_net，因此在优化器和调用train函数的相关参数时要把原来的net改成dropout_net

Task 3 调参

- 可调参数：lr, num_epoch, momentum, weight_decay, 甚至dropout里面的p

具体实现形式，就是用for(循环语句)，每次循环更改对应参数到不同值（num_epoch不用，直接改个大的值就行了），并且每次保存损失函数值，最后画图分析。

以lr为例

```
import matplotlib.pyplot as plt
from torch import optim

criterion = nn.CrossEntropyLoss()
save_path = 'lr_adjust'


for lr in [0.1,0.001,0.0001]:
    net = Net() # 重新实例化，重置网络记忆
    # lr赋值成对应循环的值
    optimizer = optim.SGD(net.parameters(), lr=lr, momentum=0.9)
    train(trainloader, net, num_epochs, criterion, optimizer, save_path)
    # 添加一条线，在show()方法使用前，每次plot都是绘制一条新的线条，颜色会自己改变的。
    # label参数可以设置图线的标签
    plt.plot([1,2,3,4,5],trainloss,label=f'lr={lr}')
    # f' '表示格式化输出字符串，和C中的printf("%s",str)有点像，不过python里用{}括起来为代码部分，其他部分为字符串。
    # 这里f'lr={lr}'表示输出字符串：lr=0.1或lr=0.001或lr=0.0001，输出哪个取决于当前循环中lr的值

plt.legend('best')  #让标签图例显示在最佳位置
plt.title('lr Adjustment')
plt.show()
```

Task 4 实现现代卷积神经网络

> 查阅资料（参考：动手学深度学习以及`torchvision`的模型源码），修改`2. 定义用于分类的网络结构`中的代码，实现一种现代卷积神经网络。

有AlexNet、ResNet等，它们在LeNet的基础上加入了一些新的特性，我们选择一个即可。建议用AlexNet，它仅仅基于LeNet做出一些小变化。

> 1. AlexNet比相对较小的LeNet5要深得多。AlexNet由八层组成：五个卷积层、两个全连接隐藏层和一个全连接输出层。
> 2. AlexNet使用ReLU而不是sigmoid作为其激活函数。(当然，在本实验中的Lenet使用ReLU)

这里以AlexNet为例

参照动手深度学习

1. AlexNet比相对较小的LeNet要深得多。AlexNet由八层组成：五个卷积层、两个全连接隐藏层和一个全连接输出层。所以我们要增多卷积层、全连接层

主要改动对比

```
# 原来的简单网络（处理32x32）
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)       # 输入3通道，输出6通道
        self.conv2 = nn.Conv2d(6, 16, 5)      # 输入6通道，输出16通道
        self.fc1 = nn.Linear(16*5*5, 120)     # 全连接层
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)          # 输出10个类别

# 改成AlexNet（处理32x32），注意，动手深度学习中使用的数据集图片大小不一样，这点需要我们改动
class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        # 5个卷积层，层数变多了
        self.conv1 = nn.Conv2d(3, 96, 3)      # 卷积核从5改成3
        self.conv2 = nn.Conv2d(96, 256, 3)
        self.conv3 = nn.Conv2d(256, 384, 3)
        self.conv4 = nn.Conv2d(384, 384, 3)
        self.conv5 = nn.Conv2d(384, 256, 3)
        # 3个全连接层
        self.fc1 = nn.Linear(256, 4096)       # 参数更多了
        self.fc2 = nn.Linear(4096, 4096)
        self.fc3 = nn.Linear(4096, 10)
        self.dropout = nn.Dropout(0.5)        # 新增功能
```

改动说明

1. 网络变深了但输入尺寸不变

```
# 原来只有2个卷积层
self.conv1 = nn.Conv2d(3, 6, 5)
self.conv2 = nn.Conv2d(6, 16, 5)

# 现在有5个卷积层
self.conv1 = nn.Conv2d(3, 96, 3)
self.conv2 = nn.Conv2d(96, 256, 3)
self.conv3 = nn.Conv2d(256, 384, 3)
self.conv4 = nn.Conv2d(384, 384, 3)
self.conv5 = nn.Conv2d(384, 256, 3)
```

2. 通道数大大增加

```
# 原来：6个通道 → 16个通道
self.conv1 = nn.Conv2d(3, 6, 5)    # 输出6个通道
self.conv2 = nn.Conv2d(6, 16, 5)   # 输出16个通道

# 现在：96个通道 → 256个通道 → 384个通道
self.conv1 = nn.Conv2d(3, 96, 3)   # 输出96个通道
self.conv2 = nn.Conv2d(96, 256, 3) # 输出256个通道
```

3. 卷积核变小了

```
# 原来用5×5的大窗口
self.conv1 = nn.Conv2d(3, 6, 5)

# 现在用3×3的小窗口
self.conv1 = nn.Conv2d(3, 96, 3)
```

因为输入图片小(32×32)，用大卷积核会丢失太多信息

4. 加了Dropout功能

```
self.dropout = nn.Dropout(0.5)
```

完整的前向传播函数

```
def forward(self, x):
    # 第一层卷积 -> 激活 -> 池化
    x = F.relu(self.conv1(x))
    x = F.max_pool2d(x, 2)

    # 第二层卷积 -> 激活 -> 池化
    x = F.relu(self.conv2(x))
    x = F.max_pool2d(x, 2)

    # 第三层卷积 -> 激活
    x = F.relu(self.conv3(x))

    # 第四层卷积 -> 激活
    x = F.relu(self.conv4(x))

    # 第五层卷积 -> 激活 -> 池化
    x = F.relu(self.conv5(x))
    x = F.max_pool2d(x, 2)

    # 全局平均池化，替代展平操作
    x = F.adaptive_avg_pool2d(x, (1, 1))
    x = torch.flatten(x, 1)

    # 全连接层部分
    x = F.relu(self.fc1(x))
    x = self.dropout(x)         # 随机丢弃一些神经元
```

```
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)             # 最后一层直接输出

        return x
```

Part 3 结语

关于神经网络还有很多需要我们探索，本文只是做个引子，引导完成作业需要往哪些方向学习。希望同学们可以通过自学来独立完成这个作业，而不是找人代做或者copy。

祝学业有成，前程似锦。