多层感知机

1. pytorch 网络的基石：torch.nn.Module

PyTorch 使用 torch.nn.Modules 来表示一个神经网络，它具有以下特点

- **是神经网络的基石** PyTorch 提供了一个健壮的模块库，使定义新的自定义模块变得简单，允许轻松地构建复杂的多层神经网络。
- **与 PyTorch 的 autograd 系统紧密集成。**模块使得为 PyTorch 的优化器指定要更新的可学习参数变得简单。
- **易于使用和转换。**模块可以直接保存和恢复，在CPU / GPU / TPU设备之间传输，修剪，量化，等等。

可以说几乎所有 PyTorch 的层和模块都是 torch.nn.Module 的子类。

nn.Module 常用方法

```
class Module(object):
    def __init__(self) -> None:
        # 初始化 Module 的状态，一般在这里定义和创建整个网络需要的层和模块
    def forward(self, *input):
        # 执行前向传播的函数，由 PyTorch 自动调用
    def apply(self: T, fn: Callable[['Module'], None]) -> T:
        # 递归的将 fn 函数用在所有的子模块还有自己上，通常用来初始化参数
    def cuda(self: T, device: Optional[Union[int, device]] = None) -> T:
        # 将所有网络参数挪到 GPU 上
```

```
    def cpu(self: T) -> T:
        # 将所有网络参数挪到 CPU 上
    def type(self: T, dst_type: Union[dtype, str]) -> T:
        # 强制类型转换
    def parameters(self, recurse: bool = True) -> Iterator[Parameter]:
        # 返回一个遍历所有参数的迭代器
    def train(self: T, mode: bool = True) -> T:
        # 将 Module 设置为训练模式（只对部分模块如 Batchnorm、Dropout 等等起作用）
    def eval(self: T) -> T:
        # 将 Module 设置为测试模式（只对部分模块如 Batchnorm、Dropout 等等起作用）
```

2. PyTorch的模块包： torch.nn

torch.nn 包含了绝大部分深度学习常用的**基础模块、优化器、损失函数**等等，且这些都是基于 torch.nn.Module 的，所以它们继承了 torch.nn.Module 的方法，可以用这些基础部件灵活的组装任意形式的深度学习网络。

一些常用的 torch.nn 模块

以下是本节课会用到的模块。

nn.Flatten

nn.Sequential

nn.Linear

nn.ReLU

3. 使用 torch.nn 搭建多层感知机

本章我们采用 PyTorch 内置的高级API简洁的实现多层感知机，并体会理解深度学习**模块化**的概念。

In [1]:

```
import torch
from torch import nn
from IPython import display
from d2l import torch as d2l

batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

模型

与softmax回归相比，唯一的区别是我们添加了2个全连接层（之前我们只添加了1个全连接层）。 第一层是**隐藏层**，它(**包含256个隐藏单元，并使用了ReLU激活函数**)。 第二层是输出层。

我们可以使用现成的 PyTorch 容器 nn.Sequential 来实现 MLP。

nn.Sequential 的输入是一系列 nn.Module ， 然后调用 nn.Sequential 会顺序执行输入的 nn.Module 。

In [2]:

```
net = nn.Sequential(nn.Flatten(),
                    nn.Linear(784, 256),
                    nn.ReLU(),
                    nn.Linear(256, 10))

```

或者可以自己继承 nn.Module ， 自己定义执行顺序，来实现一个多层感知机。

In [3]:

```python
class MLP(nn.Module):
    def __init__(self, input_size: int, hidden_size: int, output_size: int):
        super().__init__()                                    # 初始化父类
        self.flatten = nn.Flatten()
        self.hidden_layer = nn.Linear(input_size, hidden_size)   # 定义线性层
        self.activate = nn.ReLU()                             # 定义非线性激活层
        self.output_layer = nn.Linear(hidden_size, output_size)

    def forward(self, x):                                     # 定义前向传播函数
        x = self.flatten(x)
        x = self.hidden_layer(x)
        x = self.activate(x)
        x = self.output_layer(x)
        return x
```

初始化网络参数，这里使用 nn.Module 的 apply() 方法

In [4]:

```python
net = MLP(28 * 28, 256, 10)
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights);
```

训练过程的实现与我们实现softmax回归时完全相同， 这种模块化设计使我们能够将与和模型架构有关的内容独立出来。

在使用 PyTorch 内置模块实现网络的同时，还可以使用 PyTorch 现成**损失函数**和**优化器**。

分类问题，我们采用 PyTorch 自带的**交叉熵损失函数**，同时采用自带的 SGD **优化器**

In [5]:

```python
batch_size, lr, num_epochs = 256, 0.1, 10

loss = nn.CrossEntropyLoss(reduction='none')
trainer = torch.optim.SGD(net.parameters(), lr=lr)
```

将上一章节实现的各种函数封装在.py文件中调用

In [6]:

```python
from utils import Accumulator, accuracy, evaluate_accuracy, Animator
```

实现训练函数

In [19]:

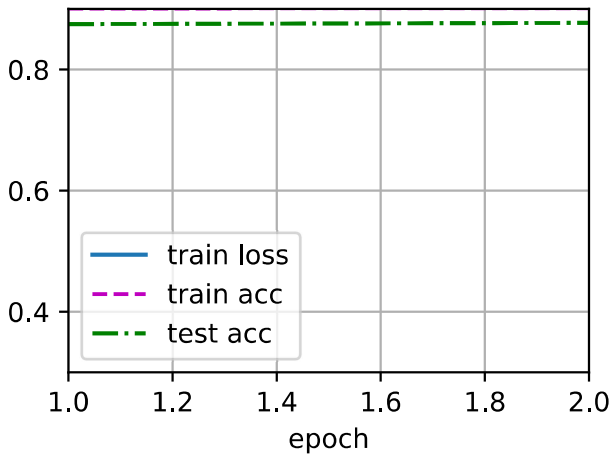
```python
def train(net, train_iter, test_iter, loss, num_epochs, updater):  #@save
    """训练模型"""
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0.3, 0.9],
                        legend=['train loss', 'train acc', 'test acc'])
    net.train()
    metric = Accumulator(3)
    for epoch in range(num_epochs):
        for X, y in train_iter:
            # 计算梯度并更新参数
            y_hat = net(X)
            l = loss(y_hat, y)
            updater.zero_grad()
            l.mean().backward()
            updater.step()
            metric.add(float(l.sum()), accuracy(y_hat, y), y.numel())

        # 保存并展示训练效果
        train_metrics = (metric[0] / metric[2], metric[1] / metric[2])
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch + 1, train_metrics + (test_acc,))
        metric.reset()
    train_loss, train_acc = train_metrics
    print(f'train_loss = {train_loss:.4f}, train_acc = {train_acc:.4f}, test_acc = {test_acc:.4
```

In [20]:

```python
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
train(net, train_iter, test_iter, loss, 2, trainer)
```

```
train_loss = 0.2750, train_acc = 0.9012, test_acc = 0.8771
```



小结

- 可以使用高级API更简洁地实现多层感知机。
- 对于相同的分类问题，多层感知机的实现与softmax回归的实现相同，只是多层感知机的实现里增加了带有激活函数的隐藏层。

练习

1. 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？
2. 尝试不同的激活函数，哪个效果最好？
3. 尝试不同的方案来初始化权重，什么方法效果最好？