多层感知机

1.多层感知机讲解

- 之前的线性回归和softmax回归，都是 单个仿射变换 ，该模型通过单个仿射变换将我们的输入直接映射到输出。有一个很大的问题是：线性意味着单调假设。
  很容易就能够举出来不是线性的例子，为了解决这类问题，我们加入 隐藏层 。
- **加入隐藏层**：

  我们可以通过在网络中加入一个或多个隐藏层来克服线性模型的限制，使其能处理更普遍的函数关系类型。要做到这一点，最简单的方法是将许多全连接层堆叠在一起。每一层都输出到上面的层，直到生成最后的输出。我们可以把前$L-1$层看作表示，把最后一层看作线性预测器。这种架构通常称为多层感知机（multilayer perceptron），通常缩写为MLP。下面，我们以图的方式描述了多层感知机（图4.1.1）。

  一个输入、隐藏、输出层的结构图例：

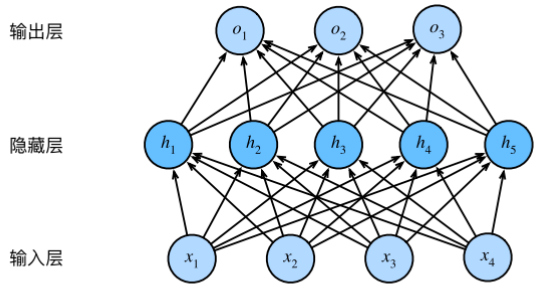


  图4.1.1: 一个单隐藏层的多层感知机，具有5个隐藏单元

- 激活函数的添加：
  为了发挥多层架构的潜力，我们还需要一个额外的关键要素：**在仿射变换之后对每个隐藏单元应用非线性的激活函数（activation function）σ** 。一般来说，有了激活函数，就不可能再将我们的多层感知机退化成线性模型：
- 公式化表达：
  - 没有引入激活函数:

    同之前的章节一样，我们通过矩阵$\mathbf{X} \in \mathbb{R}^{n \times d}$来表示$n$个样本的小批量，其中每个样本具有$d$个输入特征。对于具有$h$个隐藏单元的单隐藏层多层感知机，用$\mathbf{H} \in \mathbb{R}^{n \times h}$表示隐藏层的输出，称为隐藏表示（hidden representations）。在数学或代码中，$\mathbf{H}$也被称为隐藏层变量（hidden-layer variable）或隐藏变量（hidden variable）。因为隐藏层和输出层都是全连接的，所以我们有隐藏层权重$\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$和隐藏层偏置$\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$以及输出层权重$\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$和输出层偏置$\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$。形式上，我们按如下方式计算单隐藏层多层感知机的输出$\mathbf{O} \in \mathbb{R}^{n \times q}$：

$$\begin{aligned} \mathbf{H} &= \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{4.1.1}$$

    $H = XW^{(1)} + b^{(1)}$是第一个输入层到隐藏层的过程。

    $O = HW^{(2)} + b^{(2)}$是隐藏层到输出层的过程。

- 引入激活函数之后：
  在隐藏层次的每一个结点上面添加一个激活函数，也就是上面公式里面的H，往下进行传递之前首先经过激活函数的处理。

$$\mathbf{H} = \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}),$$

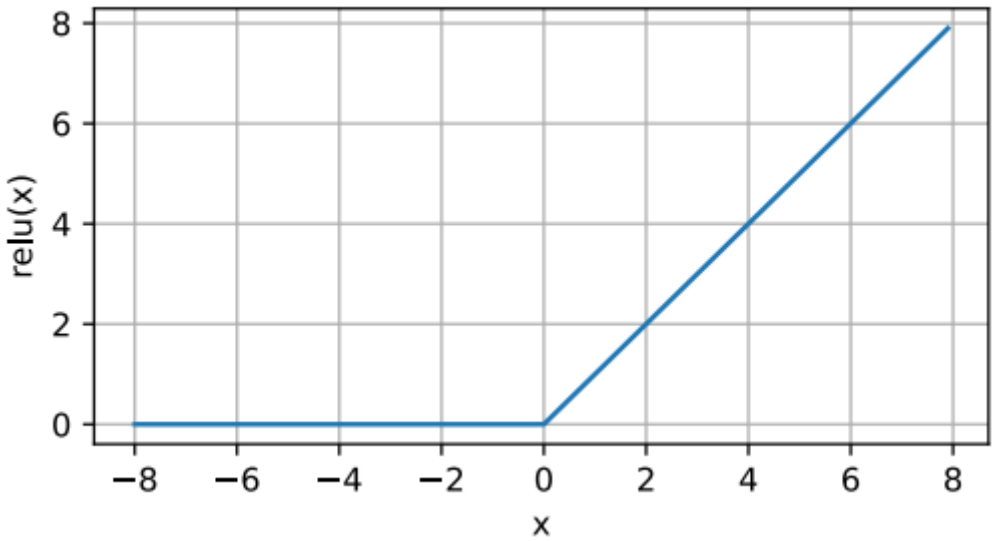
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.$$

2.常见激活函数简介

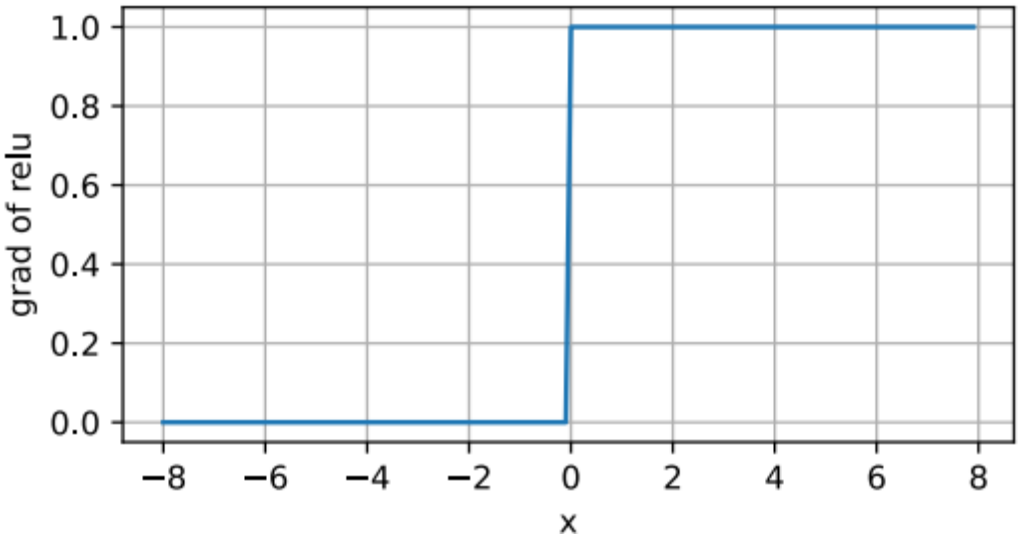
- RELU函数
  修正线性单元（Rectified linear unit， ReLU ）是最受欢迎的激活函数。
  ReLU函数被定义为该元素与0的最大值：

$$ReLU(x) = max(x, 0) \tag{1}$$

  RELU函数 简单地说就是保留正元素丢弃所有负元素直接为0 。
  - $y = x$的ReLU函数：



  - $y = x$的ReLU函数求导结果：



  rule函数表现好的原因：它求导表现得特别好：要么让参数消失，要么让参数通过。这使得优化表现得更好，并且ReLU减轻了困扰以往神经网络的梯度消失问题

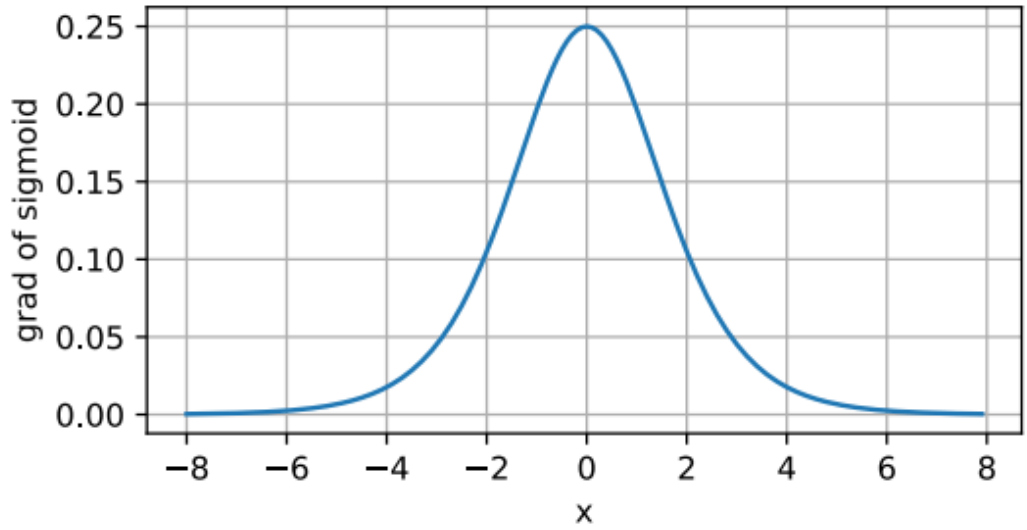
- sigmoid函数
  对于一个定义域在R中的输入， sigmoid 函数将输入变换为区间(0, 1)上的输出。

$$sigmoid(x) = \frac{1}{1 + exp(-x)} \tag{2}$$

  - 当人们逐渐关注到到基于梯度的学习时，sigmoid函数是一个自然的选择，因为它是一个平滑的、可微的阈值单元近似。
    但是目前，sigmoid在隐藏层里面，更经常被 实现简单，更容易训练的 RELU函数取代。
  - sigmoid函数的导数和图像：
    导数：

$$\frac{d}{dx}\operatorname{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \operatorname{sigmoid}(x)\left(1-\operatorname{sigmoid}(x)\right).$$

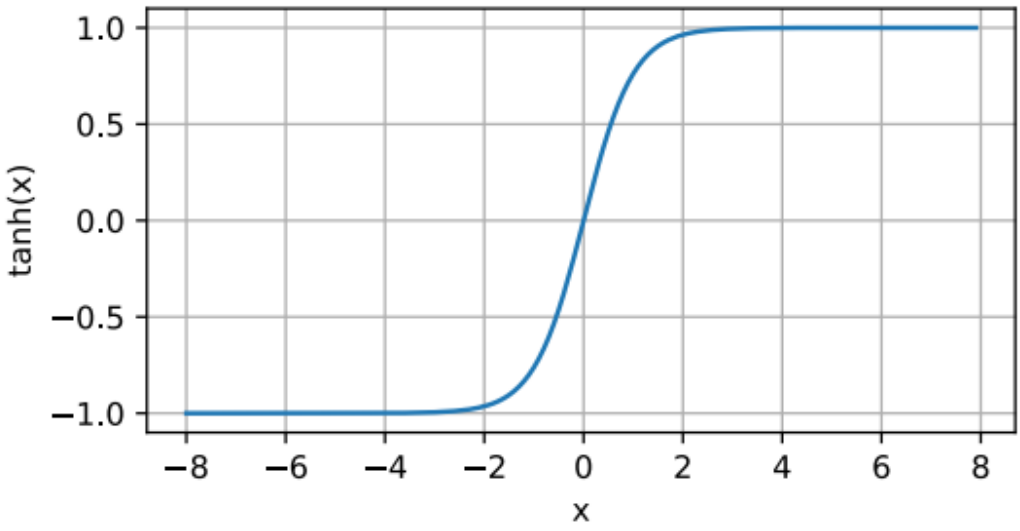
    图像：



  - 输入为0时，导数值最大为0.25。
    输入离0越远0，导数值越接近0。
- tanh函数：
  tanh(双曲正切)函数也能将其输入压缩转换到区间(-1, 1)上。
  公式：

$$tanh(x) = \frac{1 - exp(-2x)}{1 + exp(-2x)} \tag{3}$$

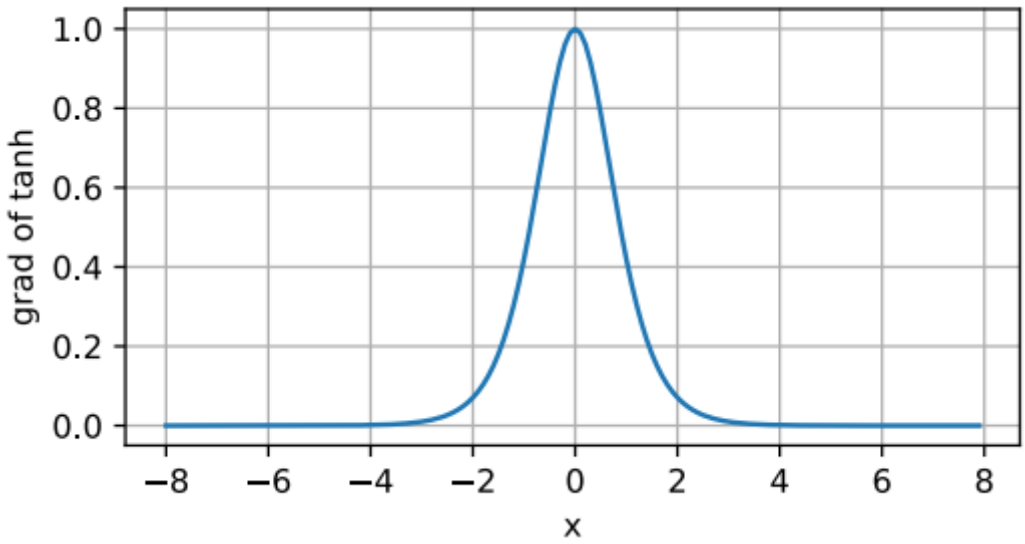
  - tanh函数图像：



- tanh函数的导数：

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x).$$

导数图像：



3.手动实现

> 继续使用之前在softmax里面的图像分类的题目背景，只不过借助MLP实现，一个输入层、隐藏层、输出层。

- 初始化的加载数据集：继续使用Fashion-MNIST图像分类数据集。

```
import torch
from torch import nn
from d2l import torch as d2l
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

每一个更新的批次大小是256。
用之前已经封装好的 `load_data_fashion_mnist()` 加载数据。 `train_iter` 和 `test_iter` 都是迭代器。

- **模型参数的初始化和解释**：
总共有四个参数张量：W_1, b_1, W_2, b_2。调用nn.Parameter进行初始化。
`W,b` 分别对应 `线性权重` 和 `偏置` 。每一层我们都要记录一个权重矩阵和一个偏置向量。

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256
W1 = nn.Parameter(torch.randn(
      num_inputs, num_hiddens, requires_grad=True) * 0.01)
# randn函数得到一个[784*256]的隐藏层权重的矩阵，乘以0.01是在缩放权重。
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]
```

说明一下：
隐藏层单元个数这里设置为256，是一个超参数。
输入层是784，因为图像size是$28 \times 28 = 784$。
输出层是10，因为最后分类的结果有10种。

数。通常，我们选择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。
对于W，b的初始化，权重选择使用randn函数，正态分布 随机生成 权重参数。偏置直接为0。

- **激活函数**：（直接手动写一个relu函数）
  也可以直接调用内置的relu函数。

```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

- 整个模型结构：
  - 模型结构：

$$\mathbf{H} = \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}),$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.$$

  - 手动实现：
    首先，使用reshape将每个二维图像转换为一个长度为num_inputs的向量。
    `H = relu(X@W1 + b1)` 就是输入层到隐藏层的过程。
    `(H@W2 + b2)` 就是隐藏层到输出层。
    直接输出 输出层 的结果。

```
def net(X):
    X = X.reshape((-1, num_inputs))
    H = relu(X@W1 + b1)  # 这里“@”代表矩阵乘法
    return (H@W2 + b2)
```

  - 损失函数的定义：(直接调库)
    `loss` 是一个 `nn.Module` 对象，具体是 `nn.CrossEntropyLoss` 的实例。

```
loss = nn.CrossEntropyLoss(reduction='none')
```

- 训练过程：
  设置updater为torch.optim.sgd

小批量随机梯度下降算法是一种优化神经网络的标准工具，PyTorch在optim模块中实现了该算法的许多变种。当我们实例化一个SGD实例时，我们要指定优化的参数（可通过net.parameters()从我们的模型中获得）以及优化算法所需的超参数字典。小批量随机梯度下降只需要设置lr值，这里设置为0.03。

```
trainer = torch.optim.SGD(net.parameters(), lr=0.03)
```

  - MLP中优化器的声明：

```
num_epochs, lr = 10, 0.1
updater = torch.optim.SGD(params, lr=lr)
```

    训练轮数是10，初始学习率是0.1。
  - 整个训练过程：

```
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, updater)
```

- 训练过程的详细代码：
  - 对于train_ch3函数，传入的参数为：**net**(整个模型正向传播的计算过程**和**计算中的所有权重(这个权重就是上面声明的W,b))。train_iter训练集迭代器、test_iter测试集迭代器、loss(一个计算损失函数的实例)、num_epochs(进行多少epoch)、updater(优化器)。

```
def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):
    """Train a model (defined in Chapter 3).

    Defined in :numref:`sec_softmax_scratch`"""
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0.3, 0.9],
                        legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch + 1, train_metrics + (test_acc,))
    train_loss, train_acc = train_metrics
    assert train_loss < 0.5, train_loss
    assert train_acc <= 1 and train_acc > 0.7, train_acc
    assert test_acc <= 1 and test_acc > 0.7, test_acc
```

    - animator函数只是帮助可视化这个过程。
    - train_epoch_ch3中定义了每一个epoch做的事情。
      函数接受net，训练集、计算损失的实例、优化器

```
def train_epoch_ch3(net, train_iter, loss, updater):
    """The training loop defined in Chapter 3.

    Defined in :numref:`sec_softmax_scratch`"""
    # Set the model to training mode
    if isinstance(net, torch.nn.Module):
        net.train()
    # Sum of training loss, sum of training accuracy, no. of examples
    metric = Accumulator(3)
    for X, y in train_iter:
        # Compute gradients and update parameters
        y_hat = net(X)
        l = loss(y_hat, y)
        if isinstance(updater, torch.optim.Optimizer):
            # Using PyTorch in-built optimizer & loss criterion
            updater.zero_grad()
            l.mean().backward()
            updater.step()
        else:
            # Using custom built optimizer & loss criterion
            l.sum().backward()
            updater(X.shape[0])
        metric.add(float(l.sum()), accuracy(y_hat, y), y.numel())
    # Return training loss and training accuracy
    return metric[0] / metric[2], metric[1] / metric[2]
```

    - evaluate_accuracy函数计算当前的权重对于测试集合的**精度**，公式为$\frac{\text{正确分类样本}}{\text{所有待分类样本总数}}$。
      具体的计算过程在soft_max章节中已经说明过。

> 以上就是 多层感知机(一个隐藏层)的手动全部实现过程，net的结构完全是自己定义，net里面的relu激活函数也是自己定义的。loss和updater是直接借助调库实现的。

4.简洁实现MLP

- 引入部分：调库

```
import torch
from torch import nn
from d2l import torch as d2l
```

- **模型解释、声明、初始化**
  第一层是一个平展层，用来将输入转换为shape为[batch_size,756]的二维张量。

  第二层是全连接层，线性层。(接受756个输入，256个输出)。
  之后对线性层的输出使用`relu激活函数`。
  最后再对激活之后的结果再进行一次线性层。(256个输入，10个输出)。
  最后10个输出是因为最后的分类类别个数就是10。

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

def init_weights(m):
      if type(m) == nn.Linear:
            nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights);
```

  对于init_weights 和 apply的解释:
  `apply()` 方法会递归地遍历网络中的每一层，并对每一层应用传入的函数 `init_weights`。
  然后对于每一层:
  这部分代码检查传入的模型层 `m` 是否是 `nn.Linear` （全连接层）
  如果是：使用nn.init.normal_(m.weight, std=0.01)按照均值为0，标准差为0.01的正态分布来随机生成值。
  总结就是 通过def init_weights和apply实现了整个网络权重值的初始化。

- 之后代码:

```
batch_size, lr, num_epochs = 256, 0.1, 10
loss = nn.CrossEntropyLoss(reduction='none')
trainer = torch.optim.SGD(net.parameters(), lr=lr)
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```

  比较一下这个和上面的手动实现:

  - 比较两个net:
    手动实现里面:

```
def net(X):
    X = X.reshape((-1, num_inputs))
    H = relu(X@W1 + b1)  # 这里“@”代表矩阵乘法
    return (H@W2 + b2)
```

    同时还自己写了relu，自己写了w,b的初始化。

简介实现里面：

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

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights);
```

借助nn.Sequential类来直接很方便进行初始化。

- train_iter,testiter,loss都一样。
- 上面函数里面的trainer参数，其实就等同于`手动实现mlp里面的updater`，本质上都是直接调用了torch.optim.SGD然后传进去整个net的parameters和lr。

> 在手动实现了MLP之后，更容易理解调库的实现原理、调库之后做的事情。重点在于net的初始化。