单层感知机

感知机是一种二元线性分类模型，其主要思想是通过一个线性分类器来将数据集分成两类。感知机的基本原理是通过输入数据的特征向量x和权重向量w的内积来进行分类。对于一个给定的输入向量x，感知机的输出y可表示为：

$$y = \begin{cases} 1, & \text{if } w \cdot x + b > 0 \\ -1, & \text{otherwise} \end{cases}$$

其中，w是权重向量，b是偏置，$w \cdot x$表示w和x的内积，y表示分类结果，若$y = 1$则表示输入x属于第一类，若$y = -1$则表示输入x属于第二类。

感知机的训练过程是通过不断调整权重向量和偏置，使得分类结果与实际结果相符。具体来说，对于每个训练样本(x_i, y_i)，如果感知机将其误分类，则需要更新权重向量和偏置。假设误分类的样本是x_k，则感知机的更新公式为：

$$w \leftarrow w + \eta y_k x_k$$

$$b \leftarrow b + \eta y_k$$

其中，η是学习率，控制每次更新的步长。

用一个简单的例子来说明感知机的原理。假设有一个二维平面上的数据集，其中点$(1, 3)$、$(2, 4)$、$(3, 1)$属于第一类，点$(1, 1)$、$(4, 2)$、$(3, 6)$属于第二类。我们希望用感知机对这个数据集进行分类。

首先，我们需要选择一个权重向量和一个偏置来进行分类。假设我们选择$w = [0, 0]$和$b = 0$，对于每个训练样本(x_i, y_i)，我们可以将其输入到感知机中，计算感知机的输出y，并与实际结果y_i进行比较。如果感知机将其误分类，则需要更新权重向量和偏置。

假设我们选择学习率$\eta = 0.1$，并按照样本顺序对感知机进行更新。我们先对第一个样本$(1, 3)$进行分类，计算感知机的输出为：

$$y = w \cdot x + b = [0, 0] \cdot [1, 3] + 0 = 0$$

由于$y = 0$，而实际结果为$y_i = 1$，因此感知机将其误分类。根据更新公式，我们需要将权重向量和偏置进行更新：

$$w \leftarrow w + \eta y_i x_i = [0, 0] + 0.1 \cdot 1 \cdot [1, 3] = [0.1, 0.3]$$

$$b \leftarrow b + \eta y_i = 0 + 0.1 \cdot 1 = 0.1$$

我们可以继续按照样本顺序对感知机进行更新。对于第二个样本$(2, 4)$，感知机的输出为：

$$y = [0.1, 0.3] \cdot [2, 4] + 0.1 = 1.1$$

由于$y > 0$，而实际结果为$y_i = 1$，因此感知机分类正确，不需要进行更新。同样地，对于第三个样本$(3, 1)$，感知机的输出为：

$$y = [0.1, 0.3] \cdot [3, 1] + 0.1 = 0.4$$

由于$y > 0$，而实际结果为$y_i = 1$，因此感知机分类正确，不需要进行更新。

接下来，我们对第四个样本$(1, 1)$进行分类，感知机的输出为：

$$y = [0.1, 0.3] \cdot [1, 1] + 0.1 = 0.5$$

由于$y > 0$，而实际结果为$y_i = -1$，因此感知机将其误分类。根据更新公式，我们需要将权重向量和偏置进行更新：

$$w \leftarrow w + \eta y_i x_i = [0.1, 0.3] + 0.1 \cdot (-1) \cdot [1, 1] = [0, 0.2]$$

$$b \leftarrow b + \eta y_i = 0.1 + 0.1 \cdot (-1) = 0$$

接下来，我们对第五个样本$(4, 2)$进行分类，感知机的输出为：

$$y = [0, 0.2] \cdot [4, 2] + 0 = 0.04$$

由于$y > 0$，而实际结果为$y_i = -1$，因此感知机将其误分类。根据更新公式，我们需要将权重向量和偏置进行更新：

$$w \leftarrow w + \eta y_i x_i = [0, 0.2] + 0.1 \cdot (-1) \cdot [4, 2] = [-0.4, 0]$$

$$b \leftarrow b + \eta y_i = 0 + 0.1 \cdot (-1) = -0.1$$

接下来，我们对第六个样本$(3,6)$进行分类，感知机的输出为：

$y=[-0.4,0]\cdot[3,6]+0=-1.2$

由于$y<-1$，而实际结果为$y_i=-1.2$，因此感知机分类正确，不需要进行更新。

最后的结果就是$w=[-0.4,0]$，$b=-0.1$

通过这个简单的例子，我们可以看到单层感知机的基本原理和训练过程。可以看到在上述的例子中，线性并不能将上述分类很好的实现。因此在实际应用中，感知机可以通过多层神经网络来扩展，从而处理更加复杂的分类问题。

多层感知机

前向传播

多层感知机（MLP）是一种前馈神经网络，其结构包含输入层、一个或多个隐藏层以及输出层。为了便于解释，我们将使用一个简单的多层感知机，它具有两个输入节点、一个包含三个节点的隐藏层和两个输出节点。这个MLP可以用于解决一个简单的分类问题。

假设我们有一个数据集，其中每个样本有两个特征值（x1，x2）。我们想要训练这个多层感知机来根据这些特征值对样本进行分类。

多层感知机的计算过程可以分为以下步骤：

1. 初始化权重和偏置 对于每一层之间的连接，我们需要初始化权重（W）和偏置（b）。在这个例子中，我们有两个权重矩阵（W1，W2）和两个偏置向量（b1，b2）。我们可以使用随机的小数值初始化它们。

   $W1=[[0.1,0.3,-0.2],[-0.4,0.5,0.2]]b1=[0.1,0.2,-0.1]$

   $W2=[[0.3,-0.2],[-0.1,0.4],[0.2,0.3]]b2=[0.05,-0.05]$

2. 输入层到隐藏层的计算 以一个样本 (x1 = 0.5, x2 = 0.8) 为例，我们首先将输入向量与权重矩阵相乘，然后加上偏置向量。

   $[0.1*0.5+(-0.4)*0.8+0.1,0.3*0.5+0.5*0.8+0.2,-0.2*0.5+0.2*0.8-0.1]=[-0.17,0.75,-0.04]$

3. 激活函数 在隐藏层和输出层，我们需要对加权和应用激活函数。常用的激活函数有 ReLU、Sigmoid 和 Tanh。在本例中，我们将使用 ReLU 函数。ReLU(x) = max(0, x)。

   ReLU后的隐藏层输出:

   $[ReLU(-0.17),ReLU(0.75),ReLU(0.04)]=[0,0.74,0]$

4.隐藏层到输出层的计算（续） 将隐藏层的输出与第二个权重矩阵相乘，然后加上第二个偏置向量。

$[0*0.3+0.75*-0.1+0*0.2+0.05,0*-0.2+0.75*0.4+0*0.3-0.05]=[-0.025,0.25]$

5.输出层的激活函数 在输出层，我们通常使用激活函数将输出值转换为概率分布。对于分类问题，通常使用 Softmax 函数。

$Softmax(x)=exp(x)/sum(exp(x))$

将 Softmax 应用于输出层的加权和：

$exp(-0.025)=0.97531$

$exp(0.25)=1.28403$

$sum(exp)=0.97531+1.28403=2.25934$

Softmax 输出:

$[1.01714/2.3496,1.33246/2.3496]=[0.431,0.568]$

由于输出层的节点数为 2，这意味着我们有两个类别。在这个例子中，第一个类别的概率是 0.431，第二个类别的概率是 0.568。因此，根据这个简单的多层感知机，输入样本 (x1 = 0.5, x2 = 0.8) 属于第二个类别的概率更高。

反向传播

为了解释多层感知机（MLP）反向传播的计算过程，我们将沿用上面的例子。我们有一个包含两个输入节点、一个包含三个节点的隐藏层和两个输出节点的 MLP。

反向传播算法用于更新网络中的权重和偏置，以便减小预测误差。在这个例子中，我们将假设训练样本 (x1 = 0.5, x2 = 0.8) 的正确类别是第一个类别，即目标输出向量为 [1, 0]。我们将使用均方误差（MSE）作为损失函数。计算过程如下：

1. 计算输出层的误差 输出层的预测值是 Softmax 输出：[0.433, 0.567]。我们首先计算输出层的误差。

   输出误差 = 预测值 - 目标值 :

   $$[0.431 - 1, 0.568 - 0] = [-0.569, 0.568]$$

2.计算隐藏层的误差 我们需要将输出层的误差传播到隐藏层。为此，我们需要计算隐藏层的加权误差。

   隐藏层误差 = 输出误差 * W2的转置 :

   $$[-0.569, 0.568] * [[0.3, -0.2], [-0.1, 0.4], [0.2, 0.3]] =$$

$$[-0.569 * 0.3 + 0.568 * -0.2, -0.569 * -0.1 + 0.568 * 0.4, -0.569 * 0.2 + 0.568 * 0.3] = [-0.2843, 0.2841, 0.0566]$$

   接下来，我们需要将加权误差乘以隐藏层输出的梯度。在这个例子中，我们使用了 ReLU 激活函数。ReLU 的梯度为：如果输入值大　于 0，则梯度为 1；如果输入值小于等于 0，则梯度为 0。

   ReLU 梯度 = [0, 1, 1] 修正后的隐藏层误差 = 隐藏层误差 * ReLU 梯度 :

   $$[-0.2843 * 0, 0.2841 * 1, 0.0566 * 1] = [0, 0.2841, 0.0566]$$

3.更新权重和偏置 为了更新权重矩阵 W1、W2 和偏置向量 b1、b2，我们需要计算它们的梯度。梯度是损失函数关于权重和偏置的偏　导数。然后，我们使用梯度下降法更新权重和偏置。

   学习率 = 0.1

   W1_grad = 输入向量的转置 * 修正后的隐藏层误差 :

   $$[0.5, 0.8] * [0, 0.2841, 0.0566] =$$

   $$[[0.5 * 0, 0.5 * 0.2841, 0.5 * 0.0566], [0.8 * 0, 0.8 * 0.2841, 0.8 * 0.0566]]$$

   W1_grad:

   $$[[0, 0.14205, 0.0285], [0, 0.22728, 0.04528]]$$

   b1_grad = 修正后的隐藏层误差:

   $$[0, 0.2841, 0.0566]$$

   W2_grad = 隐藏层输出的转置 * 输出误差 :

   $$[0, 0.2841, 0.0566] * [-0.569, 0.568] =$$

   $$[[0 * -0.567, 0 * 0.567], [0.2841 * -0.569, 0.2841 * 0.568], [0.0566 * -0.569, 0.0566 * 0.568]]$$

   W2_grad:

   $$[[0, 0], [-0.1617, 0.1614], [-0.0322, 0.0321]]$$

   b2_grad = 输出误差 :

   $$[-0.569, 0.568]$$

接下来，我们根据梯度和学习率更新权重和偏置：

W1_new = W1 - 学习率 * W1_grad :

$$[[0.1, 0.3, -0.2], [-0.4, 0.5, 0.2]] - 0.1 * [[0, 0.14205, 0.0285], [0, 0.22728, 0.04528]]$$

$$= [[0.1, 0.2858, -0.20285], [-0.4, 0.4773, 0.1955]]$$

b1_new = b1 - 学习率 * b1_grad :

$$[0.1, 0.2, -0.1] - 0.1 * [0, 0.2841, 0.0566] = [0.1, 0.1716, -0.1057]$$

W2_new = W2 - 学习率 * W2_grad:

$$[[0.3,-0.2],[-0.1,0.4],[0.2,0.3]]-0.1*[[0,0],[-0.1617,0.1614],[-0.0322,0.0321]]$$

$$=[[0.3,-0.2],[-0.08383,0.38386],[0.20322,0.29679]]$$

b2_new = b2 - 学习率 * b2_grad:

$$[0.05,-0.05]-0.1*[-0.569,0.568]=[0.1069,-0.1068]$$

代码实现

定义一个隐藏层为两层的感知机：此代码表示输入特征有784个，而第一层隐藏层的w是一个784行，256列的矩阵，而第二层隐藏层的w是一个256行，10列的矩阵

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

初始化权重

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

定义batchsize,学习率以及训练次数

```
batch_size, lr, num_epochs = 256, 0.1, 10
loss = nn.CrossEntropyLoss()
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)
```

模型选择，过拟合和欠拟合

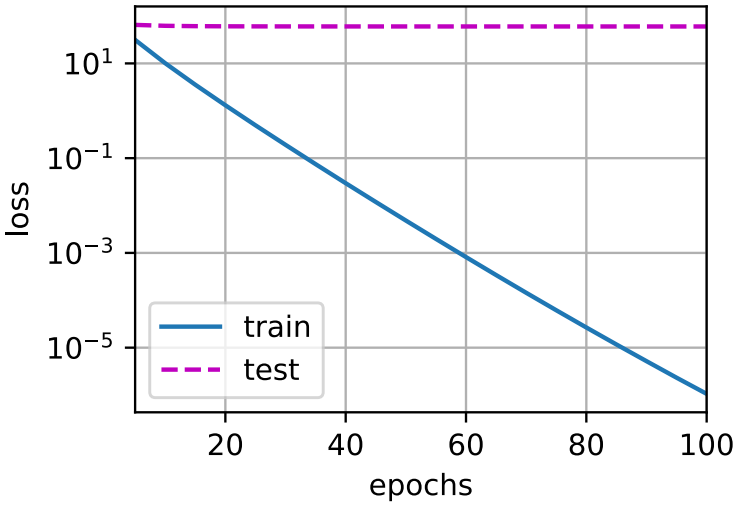
模型选择

模型选择： 模型选择是指在机器学习中从多个候选模型中选择一个最佳模型的过程。选择的标准通常基于模型在训练数据和验证数据上的性能。为了避免过拟合和欠拟合，我们需要在模型复杂度和泛化能力之间找到平衡。常见的模型选择技术包括交叉验证、信息准则（如 AIC 和 BIC）和正则化方法（如 L1 和 L2 正则化）。

过拟合

过拟合： 过拟合是指机器学习模型在训练数据上表现很好，但在新数据（测试数据）上表现较差的现象。过拟合通常发生在模型过于复杂或训练数据过少的情况下。在这种情况下，模型可能捕捉到训练数据中的噪声，而不是真正的数据分布。

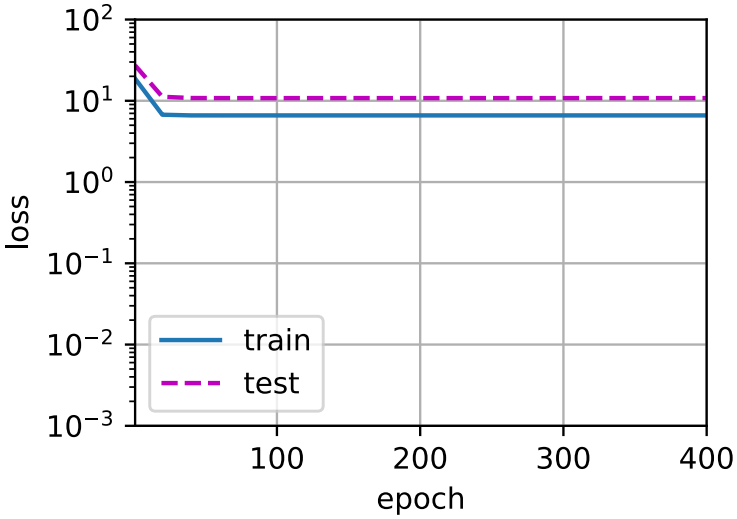
- 获取更多的训练数据
- 减小模型复杂度
- 使用正则化技术
- 使用模型平均或集成方法（如 bagging、boosting 和 stacking）
- 进行特征选择或降维



欠拟合

欠拟合： 欠拟合是指机器学习模型在训练数据和新数据（测试数据）上都表现不佳的现象。欠拟合通常发生在模型过于简单或者不能完全捕捉到数据分布的情况下。在这种情况下，模型可能缺乏足够的表达能力来描述数据的真实关系。为了解决欠拟合问题，我们可以采用以下方法：

- 增加模型复杂度
- 使用更多或更复杂的特征
- 调整模型超参数
- 尝试不同的机器学习算法



权重衰退

权重衰减是一种正则化技术，用于防止过拟合并改善模型的泛化能力。在机器学习中，正则化是一种通过对模型的参数施加约束来防止过拟合的方法。权重衰减特指在损失函数中添加一个与权重参数的平方值成正比的项，通常称为 L2 正则化。

在训练神经网络时，我们试图最小化损失函数（如均方误差损失或交叉熵损失）以获得最佳性能。但是，如果我们只关注训练数据上的损失，可能会导致过拟合，从而影响模型在新数据上的泛化能力。

为了防止过拟合，我们可以在损失函数中添加一个正则化项。权重衰减是指使用 L2 范数作为正则化项，具体形式为：

总损失 = 原始损失 + λ * Σ (权重参数^2)

其中 λ 是一个正的超参数，控制正则化的强度。权重衰减的直观解释是：通过在损失函数中增加权重参数的平方和，我们鼓励模型选择较小的权重参数。较小的权重参数意味着模型对输入特征的依赖性较低，从而降低了过拟合的风险。

在训练过程中，权重衰减会使权重参数向零收缩，从而减小模型复杂度。较小的 λ 值会导致较弱的正则化效果，而较大的 λ 值会导致较强的正则化效果。通过调整 λ 值，我们可以在模型复杂度和泛化能力之间找到平衡。

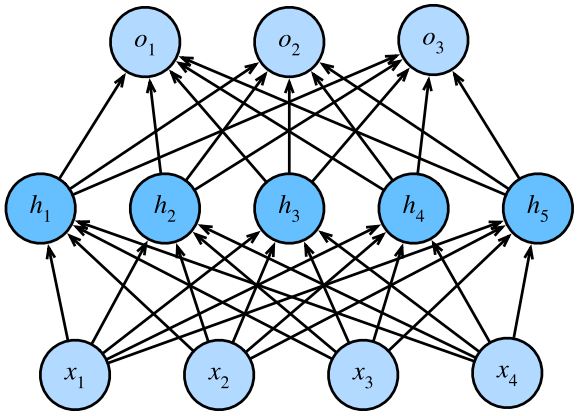
总之，权重衰减（L2 正则化）是一种防止过拟合的技术，通过在损失函数中添加权重参数的平方和来实现。通过使用权重衰减，我们可以平衡模型复杂度和泛化能力，从而在训练数据和新数据上都获得较好的性能。

丢弃法

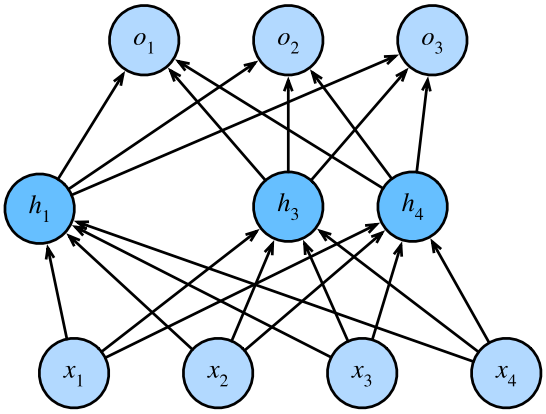
丢弃法（Dropout）是一种用于防止过拟合的机器学习技术，尤其在深度学习和神经网络领域应用广泛。过拟合是指模型在训练数据上表现优秀，但在新的、未见过的数据上泛化性能较差的现象。为了解决过拟合问题，丢弃法在训练过程中随机关闭一部分神经元，从而使模型更加稳定，提高泛化能力。

丢弃法的原理是通过随机关闭神经元，让神经网络在训练过程中学会不依赖于单个神经元的输出，从而降低神经元之间的相互依赖性，提高模型的泛化能力。在每次迭代训练时，丢弃法以一定概率p（通常为0.5）随机关闭一部分神经元，这些神经元在当前迭代中不参与计算和权重更新。这样做的好处是可以生成多个不同的子网络，这些子网络共同学习不同的特征，从而提高整体模型的泛化能力。

暂退前



暂退后



下面用一个简单的例子详细解释丢弃法：

假设我们有一个三层的神经网络，包括输入层、隐藏层和输出层。输入层有4个神经元，隐藏层有5个神经元，输出层有3个神经元。我们在隐藏层应用丢弃法，丢弃概率为0.5。

在训练过程中的某次迭代：

1. 首先，我们随机生成一个与隐藏层神经元个数相同的向量，例如：[0.3, 0.7, 0.1, 0.8, 0.4]。
2. 然后，将这个向量与丢弃概率p（0.5）进行比较，若向量中的值小于p，则对应的神经元被关闭。在本例中，我们得到一个掩码向量：[1, 0, 1, 0, 1]（1表示神经元保持激活，0表示神经元被关闭）。
3. 接着，将掩码向量应用到隐藏层的输出上，从而关闭相应的神经元。假设隐藏层的原始输出是：[0.9, 0.6, 0.8, 0.7, 0.4]，应用掩码向量后得到新的输出：[0.9, 0, 0.8, 0, 0.4]。
4. 最后，使用新的隐藏层输出继续
5. 计算输出层，并进行误差反向传播以更新权重。在这个例子中，我们已经关闭了隐藏层中的一些神经元，因此输出层计算时并没有使用所有神经元的信息。这样，在整个训练过程中，不同迭代步骤关闭不同的神经元，相当于在训练多个子网络。

   在训练完成后，我们会用完整的神经网络（不使用丢弃法）对测试数据进行预测。为了获得整体模型的预测结果，需要对训练时应用丢弃法得到的子网络输出进行集成。这里的集成方法是通过将训练过程中丢弃神经元的概率p进行缩放。例如，假设最后一层隐藏层的输出是 [1.0, 0.8, 1.2, 0.6, 0.4]，我们需要将这个输出乘以 (1-p)（即0.5），得到新的输出 [0.5, 0.4, 0.6, 0.3, 0.2]，然后用这个结果进行预测。

   通过使用丢弃法，神经网络可以学习到更加稳定和鲁棒的特征表示，从而提高在新数据上的泛化能力。这种方法可以有效降低过拟合现象，提高模型的性能。

代码实现

定义一个函数，该函数功能为如果一个值大于p，则该值(x)等于x*p, 但是如果p=1,则x=x;如果p=0则x=0

```
def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    # 在本情况中，所有元素都被丢弃
    if dropout == 1:
        return torch.zeros_like(X)
    # 在本情况中，所有元素都被保留
    if dropout == 0:
        return X
    mask = (torch.rand(X.shape) > dropout).float()
    return mask * X / (1.0 - dropout)
```

随机生成X：

```
X= torch.arange(16, dtype = torch.float32).reshape((2, 8))
```

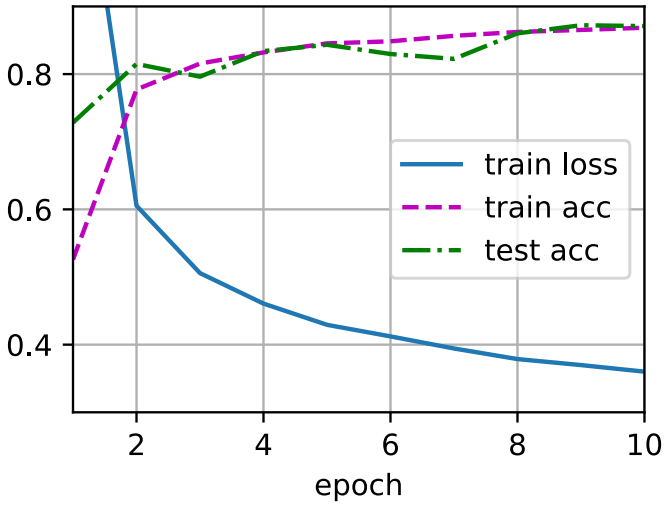
将该X值代入多层感知机进行训练，并在第一个隐藏层使用p=0.2的dropout rate, 再第二个隐藏层使用p=0.5的dropout rate

```
num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256
dropout1, dropout2 = 0.2, 0.5

class Net(nn.Module):
    def __init__(self, num_inputs, num_outputs, num_hiddens1, num_hiddens2,
                 is_training = True):
        super(Net, self).__init__()
        self.num_inputs = num_inputs
        self.training = is_training
        self.lin1 = nn.Linear(num_inputs, num_hiddens1)
        self.lin2 = nn.Linear(num_hiddens1, num_hiddens2)
        self.lin3 = nn.Linear(num_hiddens2, num_outputs)
        self.relu = nn.ReLU()

    def forward(self, X):
        H1 = self.relu(self.lin1(X.reshape((-1, self.num_inputs))))
        # 只有在训练模型时才使用dropout
        if self.training == True:
            # 在第一个全连接层之后添加一个dropout层
            H1 = dropout_layer(H1, dropout1)
        H2 = self.relu(self.lin2(H1))
        if self.training == True:
            # 在第二个全连接层之后添加一个dropout层
            H2 = dropout_layer(H2, dropout2)
        out = self.lin3(H2)
        return out
net = Net(num_inputs, num_outputs, num_hiddens1, num_hiddens2)
num_epochs, lr, batch_size = 10, 0.5, 256
loss = nn.CrossEntropyLoss()
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
trainer = torch.optim.SGD(net.parameters(), lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```

生成的结果为：



可以看到随着训练次数的不断迭代，测试集的精确度在不断提高，过拟合现象并未发生

数值稳定性和模型初始化

梯度爆炸

梯度爆炸是深度学习中的一个常见问题，梯度爆炸发生在梯度变得非常大的情况下，导致权重更新过大，使得模型参数更新变得不稳定。这可能会导致网络性能下降，甚至无法收敛。在训练过程中，我们需要计算损失函数相对于模型参数的梯度。由于深度学习具有循环结构，我们需要使用链式法则进行梯度计算。

在某些情况下，例如权重矩阵W的值很大，梯度可能会在反向传播过程中不断累积，导致梯度爆炸。当梯度变得非常大时，权重更新过大，模型变得不稳定。这可能导致模型无法收敛，学习的效果变差。

例如：

W = [[2, 0], [0, 2]]

h_0 = [1, 1]

h_1 = tanh(W * h_0) = tanh([2, 2]) = [0.96, 0.96]

h_2 = tanh(W * h_1) = tanh([2.96, 2.96]) = [0.999, 0.999]

h_3 = tanh(W * h_2) = tanh([3.96, 3.96]) = [0.9999, 0.9999]

h_4 = tanh(W * h_3) = tanh([4.95, 4,95]) = [0.99999, 0.99999]

tanh激活函数的导数可以表示为：

tanh'(x) = 1 - tanh^2(x)

从上述例子可以看出，不过才4层隐藏层，梯度已经爆炸，tanh'(x) 无限接近于0

梯度消失

梯度消失是指在反向传播过程中，梯度的值变得非常小，接近于0。这会导致模型参数更新缓慢或几乎不更新，从而使得网络训练变得困难，收敛速度变慢。

梯度消失通常是由于激活函数（如Sigmoid或Tanh）的导数值范围较小或者权重初始化不当导致的。在训练深度神经网络时，随着层数的增加，这个问题变得更加严重。

例子：

假设我们正在训练一个简单的前馈神经网络，该网络包含多个隐藏层，并使用Sigmoid激活函数。Sigmoid激活函数的导数值在0到0.25之间。在反向传播过程中，我们需要计算损失函数相对于模型参数的梯度。由于激活函数的导数值较小，当我们将这些导数值相乘时，梯度会变得越来越小。

假设我们有一个5层的神经网络，每一层都使用Sigmoid激活函数。在反向传播过程中，梯度可能需要乘以多个Sigmoid导数值。例如，如果每一层的导数值为0.2，那么在第五层，梯度将乘以0.2^5 = 0.00032。这会导致梯度值变得非常小，使得参数更新非常缓慢，从而导致网络训练困难，收敛速度变慢。

模型初始化

在线性激活函数中，我们可以使用类似的方法来保持每层的方差为常数。这里我们依然使用Xavier初始化方法。因为在线性激活函数中，激活函数对输入的变化是线性的，所以权重初始化可以直接从输入到输出保持方差一致。

假设我们有一个三层的全连接神经网络（输入层、隐藏层和输出层），并使用线性激活函数。输入层有10个神经元，隐藏层有5个神经元，输出层有2个神经元。现在我们将使用Xavier初始化来设置权重。

在Xavier初始化中，权重w的初始化方法是从一个均匀分布中随机抽取，其范围是[-limit, limit]，其中limit为：

limit = sqrt(6 / (n_input + n_output))

n_input表示该层输入神经元的数量，n_output表示该层输出神经元的数量。对于本例中的隐藏层，n_input=10（输入层神经元数量），n_output=5（隐藏层神经元数量）。计算得到的limit如下：

limit = sqrt(6 / (10 + 5)) limit ≈ 0.49

所以，我们可以从均匀分布U(-0.49, 0.49)中随机抽取权重值来初始化隐藏层的权重矩阵。

类似地，我们可以计算输出层的权重矩阵初始化范围。在这里，n_input=5（隐藏层神经元数量），n_output=2（输出层神经元数量）。计算得到的limit如下：

limit = sqrt(6 / (5 + 2)) limit ≈ 0.82

因此，我们可以从均匀分布U(-0.82, 0.82)中随机抽取权重值来初始化输出层的权重矩阵。

通过使用Xavier初始化，我们使得每层的权重矩阵具有适当的方差，从而使得各层的输出方差在传播过程中保持相对稳定。这有助于避免在训练过程中出现梯度消失或梯度爆炸的问题。在线性激活函数的情况下，Xavier初始化尤其适用，因为激活函数对输入的变化是线性的，所以权重初始化可以直接从输入到输出保持方差一致。