1. 模型选择、欠拟合和过拟合

模型复杂性

重点介绍几个倾向于影响模型泛化的因素：

1. 可调整参数的数量。当可调整参数的数量（有时称为*自由度*）很大时，模型往往更容易过拟合。
2. 参数采用的值。当权重的取值范围较大时，模型可能更容易过拟合。
3. 训练样本的数量。即使你的模型很简单，也很容易过拟合只包含一两个样本的数据集。而过拟合一个有数百万个样本的数据集则需要一个极其灵活的模型。

模型选择

为了确定候选模型中的最佳模型，通常会使用验证集。

验证集

原则上，在确定所有的超参数之前，不希望用到测试集。 如果在模型选择过程中使用测试数据，可能会有过拟合测试数据的风险，那就麻烦大了。 如果训练数据过拟合，还可以在测试数据上的评估来判断过拟合。 但是如果测试数据过拟合又该怎么知道呢？

因此，决不能依靠测试数据进行模型选择。 然而，也不能仅仅依靠训练数据来选择模型，因为我们无法估计训练数据的泛化误差。

在实际应用中，情况变得更加复杂。 虽然理想情况下只会使用测试数据一次， 以评估最好的模型或比较一些模型效果，但现实是测试数据很少在使用一次后被丢弃。 很少能有充足的数据来对每一轮实验采用全新测试集。

解决此问题的常见做法是将数据分成三份， 除了训练和测试数据集之外，还增加一个*验证数据集*（validation dataset）， 也叫*验证集*（validation set）。 但现实是验证数据和测试数据之间的边界模糊得令人担忧。

K折交叉验证

当训练数据稀缺时，我们甚至可能无法提供足够的数据来构成一个合适的验证集。 这个问题的一个流行的解决方案是采用K*折交叉验证*。 这里，原始训练数据被分成K个不重叠的子集。 然后执行K次模型训练和验证，每次在$K-1$个子集上进行训练， 并在剩余的一个子集（在该轮中没有用于训练的子集）上进行验证。 最后，通过对K次实验的结果取平均来估计训练和验证误差。

欠拟合还是过拟合？

当比较训练和验证误差时，要注意两种常见的情况。

注意这样的情况：训练误差和验证误差都高得很严重， 但它们之间仅有一点差距。 如果模型不能降低训练误差，这可能意味着模型过于简单（即表达能力不足）， 无法捕获试图学习的模式。 此外，由于训练和验证误差之间的*泛化误差*很小， 我们有理由相信可以用一个更复杂的模型降低训练误差。 这种现象被称为*欠拟合*（underfitting）。

另一方面，当训练误差明显低于验证误差时要小心， 这表明严重的*过拟合*（overfitting）。 注意，*过拟合*并不总是一件坏事。 特别是在深度学习领域，众所周知， 最好的预测模型在训练数据上的表现往往比在保留（验证）数据上好得多。 最终，我们通常更关心验证误差，而不是训练误差和验证误差之间的差距。

2. 多项式回归

在多项式回归问题中，给定由单个特征x和对应实数标签y组成的训练数据， 试图找到下面的d阶多项式来估计标签y。

$$\hat{y}=\sum_{i=0}^{d} x^i w_i$$

这只是一个线性回归问题，因为特征是x的幂。

由于这只是一个线性回归问题， 可以使用平方误差作为我们的损失函数。

高阶多项式函数比低阶多项式函数复杂得多。 高阶多项式的参数较多，模型函数的选择范围较广。 因此在固定训练数据集的情况下， 高阶多项式函数相对于低阶多项式的训练误差应该始终更低（最坏也是相等）。 事实上，当数据样本包含了x的不同值时， 函数阶数等于数据样本数量的多项式函数可以完美拟合训练集。

现在通过**多项式拟合来探索过拟合和欠拟合。**

In [1]:
```
%matplotlib inline
import math
import numpy as np
import torch
from torch import nn
```

```
from d2l import torch as d2l
from utils import Accumulator
from torch.utils import data
```

生成数据集

给定x，我们将**使用以下三阶多项式来生成训练和测试数据的标签：**

$$y = 5 + 1.2x - 3.4\frac{x^2}{2!} + 5.6\frac{x^3}{3!} + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.1^2).$$

噪声项ϵ服从均值为0且标准差为0.1的正态分布。 在优化的过程中，为避免非常大的梯度值或损失值， 将特征从x^i调整为$\frac{x^i}{i!}$。 为训练集和测试集各生成100个样本。

In [2]:
```
max_degree = 20  # 多项式的最大阶数
n_train, n_test = 100, 100  # 训练和测试数据集大小
true_w = np.zeros(max_degree)  # 分配大量的空间
true_w[0:4] = np.array([5, 1.2, -3.4, 5.6])

def synthetic_data(true_w, num_data, max_degree):  # 定义一个函数生成多项式数据集
    features = np.random.normal(size=(n_train + n_test, 1))
    np.random.shuffle(features)
    poly_features = np.power(features, np.arange(max_degree).reshape(1, -1))
    for i in range(max_degree):
        poly_features[:, i] /= math.gamma(i + 1)  # gamma(n)=(n-1)!
    # labels的维度:(n_train+n_test,)
    labels = np.dot(poly_features, true_w)
    labels += np.random.normal(scale=0.1, size=labels.shape)
    x = torch.tensor(poly_features, dtype = torch.float32)
    y = torch.tensor(labels, dtype = torch.float32)
    return x, y
```

同样，存储在`poly_features`中的单项式由gamma函数重新缩放， gamma函数为$\Gamma(n) = (n-1)!$。

从生成的数据集中**查看一下前2个样本**， 第一个值是与偏置相对应的常量特征。

In [3]:
```
# NumPy ndarray转换为tensor
poly_features, labels = synthetic_data(true_w, n_train + n_test, max_degree)
```

In [4]:
```
poly_features[:2, :2], labels[:2]
```

Out[4]:
```
(tensor([[1.0000, 1.5883],
         [1.0000, 1.8318]]),
 tensor([6.3549, 7.2484]))
```

对模型进行训练和测试

首先**实现一个函数来评估模型在给定数据集上的损失。**

In [5]:
```
def evaluate_loss(net, data_iter, loss):  #@save
    """评估给定数据集上模型的损失"""
    metric = Accumulator(2)  # 损失的总和,样本数量
    for X, y in data_iter:
        out = net(X)
        y = y.reshape(out.shape)
        l = loss(out, y)
        metric.add(l.sum(), l.numel())
    return metric[0] / metric[1]
```

现在**定义训练函数。**

In [6]:
```
def train(train_features, test_features, train_labels, test_labels,
          num_epochs=500):
    loss = nn.MSELoss(reduction='none')
    input_shape = train_features.shape[-1]
    # 不设置偏置，因为我们已经在多项式中实现了它
    net = nn.Linear(input_shape, 1, bias=False)
    batch_size = min(10, train_labels.shape[0])
    train_iter = data.DataLoader(data.TensorDataset(train_features, train_labels.reshape(-1,1)),
                                batch_size, shuffle=True)
    test_iter = data.DataLoader(data.TensorDataset(test_features, test_labels.reshape(-1,1)),
                               batch_size, shuffle=False)
    trainer = torch.optim.SGD(net.parameters(), lr=0.01)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], ylim=[1e-3, 1e2],
                            legend=['train', 'test'])
    for epoch in range(num_epochs):
        d2l.train_epoch_ch3(net, train_iter, loss, trainer)
        if epoch == 0 or (epoch + 1) % 20 == 0:
            animator.add(epoch + 1, (evaluate_loss(net, train_iter, loss),
                                     evaluate_loss(net, test_iter, loss)))
    print(f'train loss: %.4f' % evaluate_loss(net, train_iter, loss))
```

```
    print(f'test loss: %.4f' % evaluate_loss(net, test_iter, loss))
    print('weight:', net.weight.data.numpy())
```

三阶多项式函数拟合(正常)

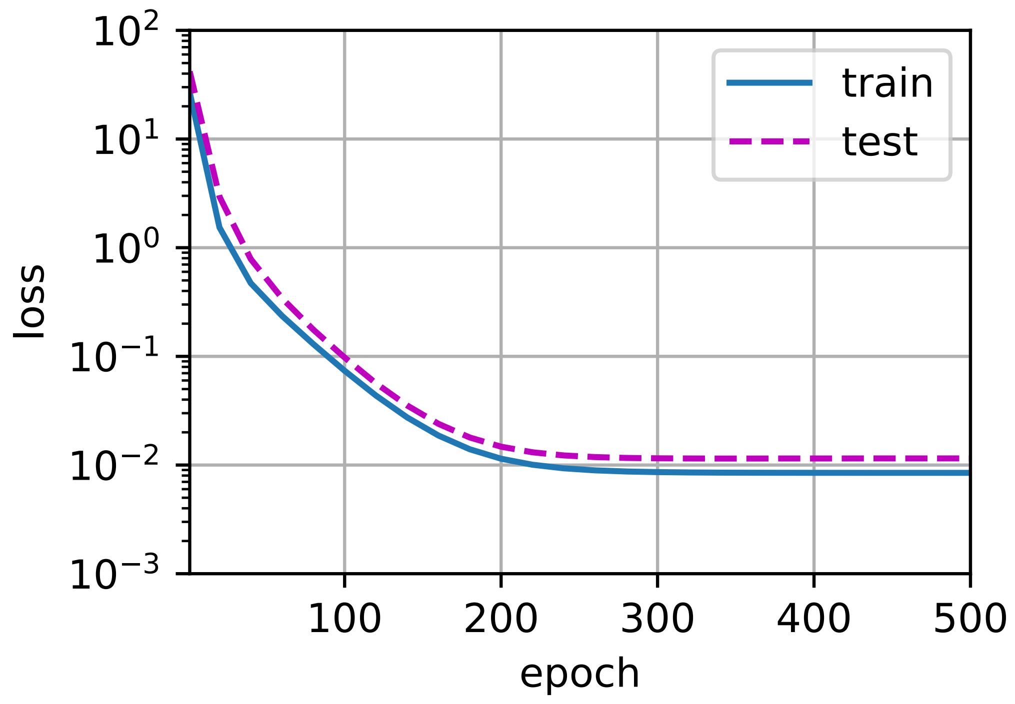
我们将首先使用三阶多项式函数，它与数据生成函数的阶数相同。

结果表明，该模型能有效降低训练损失和测试损失。

学习到的模型参数也接近真实值$w = [5, 1.2, -3.4, 5.6]$。

In [7]:
```
# 从多项式特征中选择前4个维度，即1,x,x^2/2!,x^3/3!
train(poly_features[:n_train, :4], poly_features[n_train:, :4],
      labels[:n_train], labels[n_train:])
```

```
train loss: 0.0085
test loss: 0.0115
weight: [[ 4.9993577   1.1955613 -3.4087014   5.622792 ]]
```

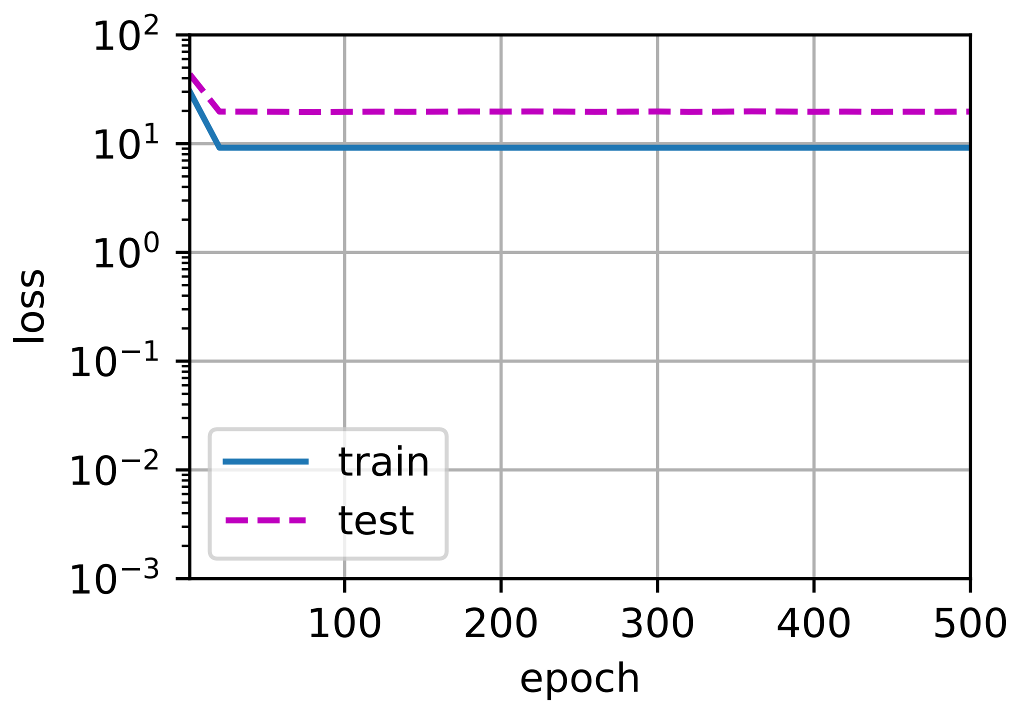


线性函数拟合(欠拟合)

让我们再看看线性函数拟合，减少该模型的训练损失相对困难。 在最后一个迭代周期完成后，训练损失仍然很高。 当用来拟合非线性模式（如这里的三阶多项式函数）时，线性模型容易欠拟合。

In [8]:
```
# 从多项式特征中选择前2个维度，即1和x
train(poly_features[:n_train, :2], poly_features[n_train:, :2],
      labels[:n_train], labels[n_train:])
```

```
train loss: 9.1738
test loss: 19.7053
weight: [[3.1846519 3.5910935]]
```



高阶多项式函数拟合(过拟合)

现在，尝试使用一个阶数过高的多项式来训练模型。

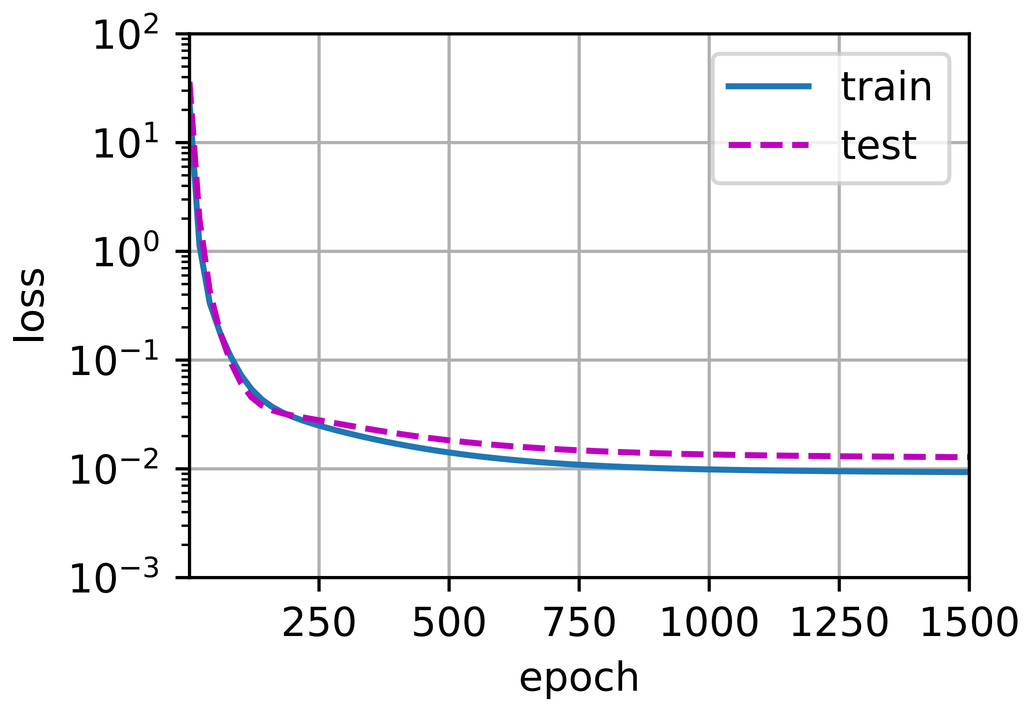
在这种情况下，没有足够的数据用于学到高阶系数应该具有接近于零的值。 因此，这个过于复杂的模型会轻易受到训练数据中噪声的影响。

虽然训练损失可以有效地降低，但测试损失仍然很高。

结果表明，**复杂模型对数据造成了过拟合。**

In [9]:
```
# 从多项式特征中选取所有维度
train(poly_features[:n_train, :], poly_features[n_train:, :],
      labels[:n_train], labels[n_train:], num_epochs=1500)
```

```
train loss: 0.0093
test loss: 0.0128
weight: [[ 4.996246     1.2937849   -3.4003422    5.141419     0.01160298   1.3533144
   0.17649329   0.21850222   0.01272785 -0.19861323   0.12319089 -0.06435297
  -0.13202149 -0.02369711 -0.22132273 -0.05024936 -0.16215695   0.12477634
   0.13711424 -0.11491285]]
```



权重衰减

范数与权重衰减

L_2范数和L_1范数，是更为一般的L_p范数的特殊情况。

在训练参数化机器学习模型时， *权重衰减*（weight decay）是最广泛使用的正则化的技术之一，

它通常也被称为L_2*正则化*。

这项技术通过函数与零的距离来衡量函数的复杂度， 因为在所有函数f中，函数$f = 0$（所有输入都得到值0） 在某种意义上是最简单的。

但是如何精确地测量一个函数和零之间的距离没有一个正确的答案。 事实上，函数分析和巴拿赫空间理论的研究，都在致力于回答这个问题。

一种简单的方法是通过线性函数 $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ 中的权重向量的某个范数来度量其复杂性，

例如$\|\mathbf{w}\|^2$。 要保证权重向量比较小，

最常用方法是将其范数作为惩罚项加到最小化损失的问题中。 将原来的训练目标*最小化训练标签上的预测损失*， 调整为*最小化预测损失和惩罚项之和*。

现在，如果权重向量增长的太大， 学习算法可能会更集中于最小化权重范数$\|\mathbf{w}\|^2$。

对于线性回归的损失：

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^n \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2.$$

为了惩罚权重向量的大小，通过*正则化常数*λ作为权衡将$\|\mathbf{w}\|^2$加入损失中， 这是一个非负超参数，使用验证数据拟合：

$$L(\mathbf{w}, b) + \frac{\lambda}{2} \|\mathbf{w}\|^2,$$

$$L(\mathbf{w}, b) + \frac{\lambda}{2} \|\mathbf{w}\|^2,$$

如果$\lambda = 0$，即为原来的损失函数。 如果$\lambda > 0$，$\|\mathbf{w}\|$的大小则被限制。

使用平方范数而不是标准范数（即欧几里得距离）是为了便于计算。 通过平方去掉平方根，留下权重向量每个分量的平方和，使得惩罚的导数很容易计算：导数的和等于和的导数。

L_2正则化线性模型构成经典的*岭回归*（ridge regression）算法， L_1正则化线性回归是统计学中类似的基本模型， 通常被称为*套索回归*（lasso regression）。

使用L_2范数的一个原因是它对权重向量的大分量施加了巨大的惩罚。 这使得学习算法偏向于在大量特征上均匀分布权重的模型。 在实践中，这可能使它们对单个变量中的观测误差更为稳定。 相比之下，L_1惩罚会导致模型将权重集中在一小部分特征上， 而将其他权重清除为零。 这称为*特征选择*（feature selection），这可能是其他场景下需要的。

L_2正则化回归的小批量随机梯度下降更新如下式：

$$\mathbf{w} \leftarrow \left(1- \eta\lambda \right) \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right).$$

网络输出层的偏置项不会被正则化。

高维线性回归

我们通过一个简单的例子来演示权重衰减。

首先，**像之前一样生成一些数据**，生成公式如下：

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01^2).$$

标签是关于输入的线性函数。 标签同时被均值为0，标准差为0.01高斯噪声破坏。 为了使过拟合的效果更加明显，可以将问题的维数增加到$d = 200$， 并使用一个只包含20个样本的小训练集。

In [10]:
```
n_train, n_test, num_inputs, batch_size = 20, 100, 200, 5
true_w, true_b = torch.ones((num_inputs, 1)) * 0.01, 0.05

def synthetic_data2(w, b, num_examples):
    """生成y=Xw+b+噪声"""
    X = torch.normal(0, 1, (num_examples, len(w)))
    y = torch.matmul(X, w) + b
    y += torch.normal(0, 0.01, y.shape)
    return X, y.reshape((-1, 1))

train_data, train_labels = synthetic_data2(true_w, true_b, n_train)
train_iter = data.DataLoader(data.TensorDataset(train_data, train_labels.reshape(-1,1)), batch_size, shuffle=True)
test_data, test_labels = synthetic_data2(true_w, true_b, n_test)
test_iter = data.DataLoader(data.TensorDataset(test_data, test_labels.reshape(-1,1)), batch_size, shuffle=False)
```

从零开始实现

下面我们将从头开始实现权重衰减，只需将L_2的平方惩罚添加到原始目标函数中。

定义L_2范数惩罚

实现这一惩罚最方便的方法是对所有项求平方后并将它们求和。

In [11]:
```
def l2_penalty(w):
    return torch.sum(w.pow(2)) / 2
```

定义训练代码实现

下面的代码将模型拟合训练数据集，并在测试数据集上进行评估。

和线性回归相比，唯一的变化是损失现在包括了惩罚项。

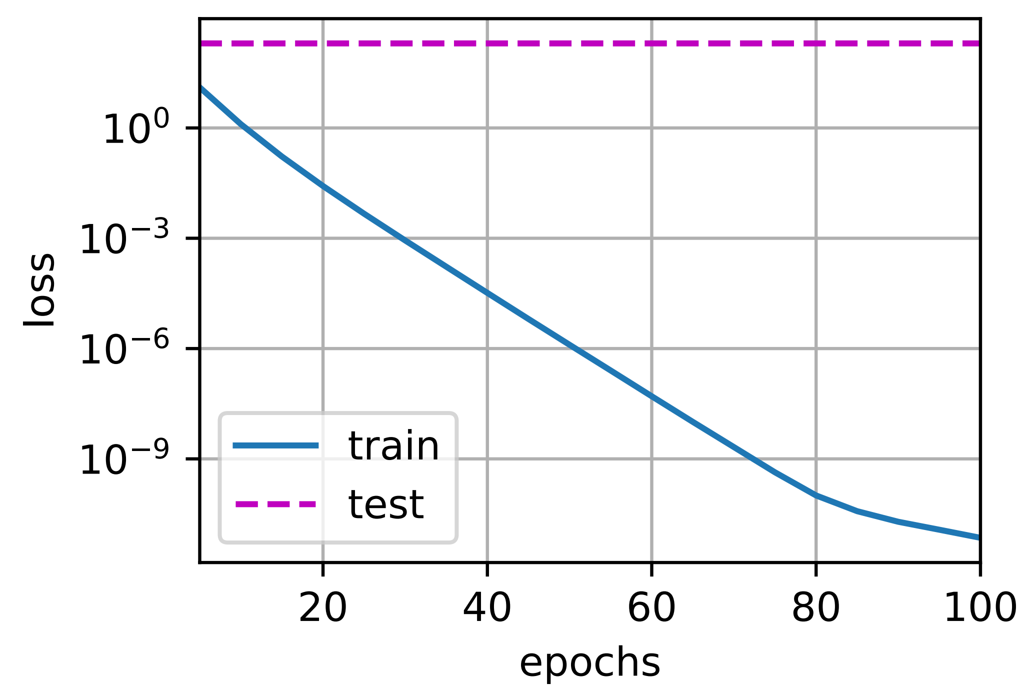
In [12]:
```
def train(lambd):
    net = nn.Linear(num_inputs, 1)
    for param in net.parameters():
        param.data.normal_()
    loss = nn.MSELoss(reduction='none')
    num_epochs, lr = 100, 0.003
    trainer = torch.optim.SGD(net.parameters(), lr)
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            # 增加了L2范数惩罚项,
            # 广播机制使l2_penalty(w)成为一个长度为batch_size的向量
            trainer.zero_grad()
            l = loss(net(X), y) + lambd * l2_penalty(net.weight)
            l.mean().backward()
            trainer.step()
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (evaluate_loss(net, train_iter, loss),
                                     evaluate_loss(net, test_iter, loss)))
    print('w的L2范数是：', torch.norm(net.weight).item())
```

忽略正则化直接训练

现在用 `lambd = 0` 禁用权重衰减后运行这个代码。 注意，这里训练误差有了减少，但测试误差没有减少， 这意味着出现了严重的过拟合。

In [13]:
```
train(lambd=0)
```

```
w的L2范数是： 13.843637466430664
```

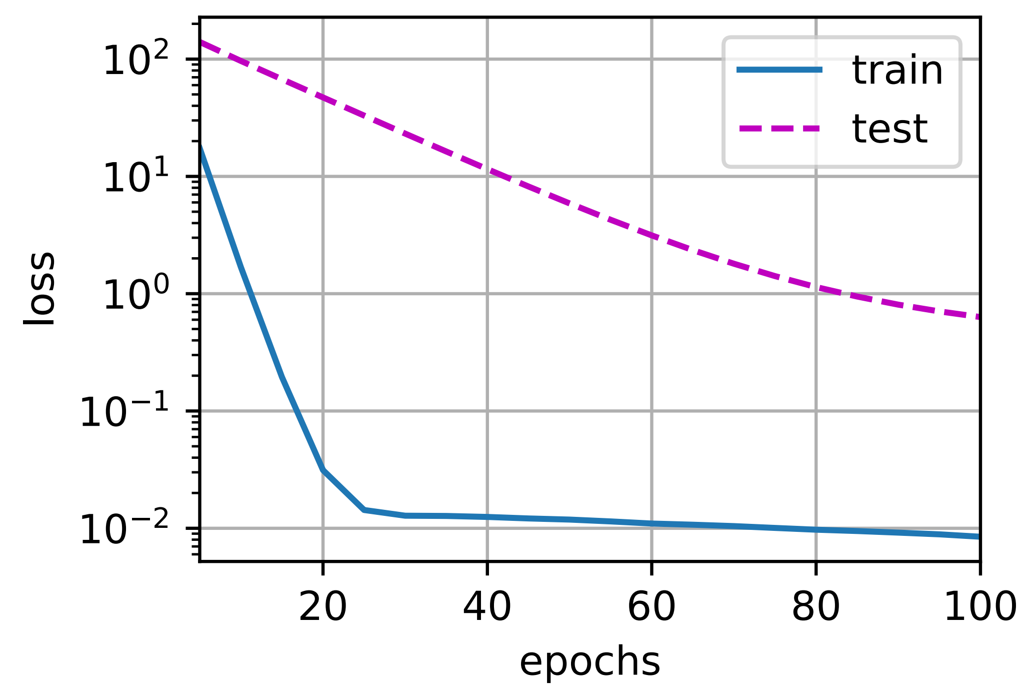


使用权重衰减

下面，使用权重衰减来运行代码。 注意，在这里训练误差增大，但测试误差减小。 这正是期望从正则化中得到的效果。

In [14]:
```
train(lambd=3)
```

w的L2范数是： 0.4153316020965576



简洁实现

由于权重衰减在神经网络优化中很常用， 深度学习框架为了便于使用权重衰减， 将权重衰减集成到优化算法中，以便与任何损失函数结合使用。 此外，这种集成还有计算上的好处， 允许在不增加任何额外的计算开销的情况下向算法中添加权重衰减。 由于更新的权重衰减部分仅依赖于每个参数的当前值， 因此优化器必须至少接触每个参数一次。

在pytorch中，实例化优化器时可以直接通过 `weight_decay` 指定weight decay超参数。

默认情况下，PyTorch同时衰减权重和偏移。 这里只为权重设置了 `weight_decay` ，所以偏置参数b不会衰减。

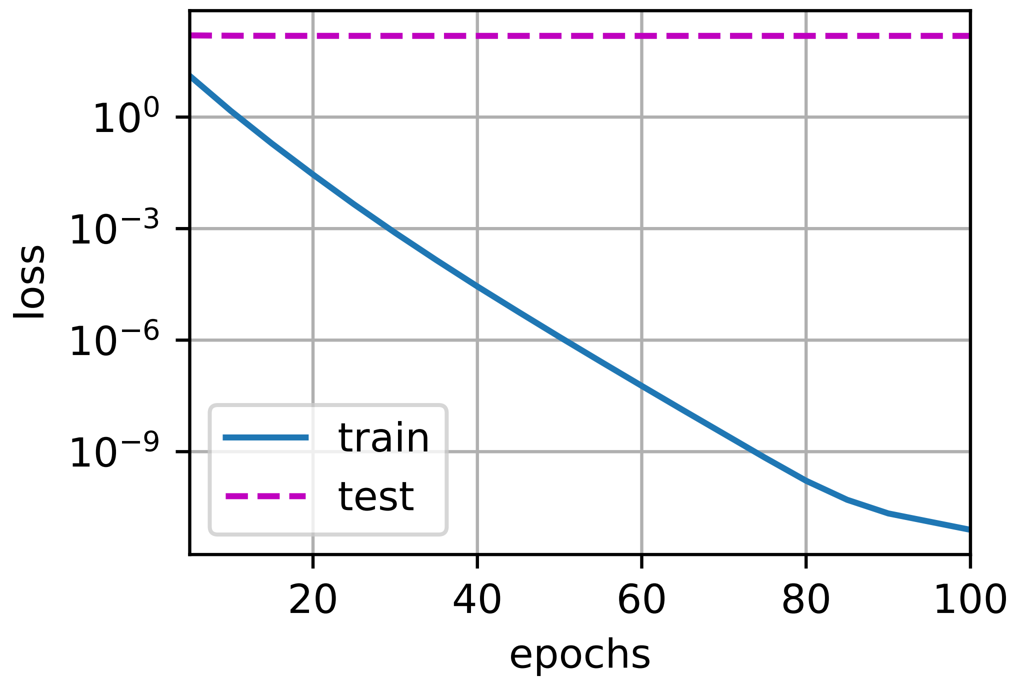
In [15]:
```
def train_concise(wd):
    net = nn.Linear(num_inputs, 1)
    for param in net.parameters():
        param.data.normal_()
    loss = nn.MSELoss(reduction='none')
    num_epochs, lr = 100, 0.003
    # 偏置参数没有衰减
    trainer = torch.optim.SGD([
        {"params":net.weight,'weight_decay': wd},
        {"params":net.bias}], lr=lr)
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            trainer.zero_grad()
            l = loss(net(X), y)
            l.mean().backward()
            trainer.step()
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1,
                         (d2l.evaluate_loss(net, train_iter, loss),
                          d2l.evaluate_loss(net, test_iter, loss)))
    print('w的L2范数：', net.weight.norm().item())
```

这些图看起来和我们从零开始实现权重衰减时的图相同。

然而，它们运行得更快，更容易实现。 对于更复杂的问题，这一好处将变得更加明显。

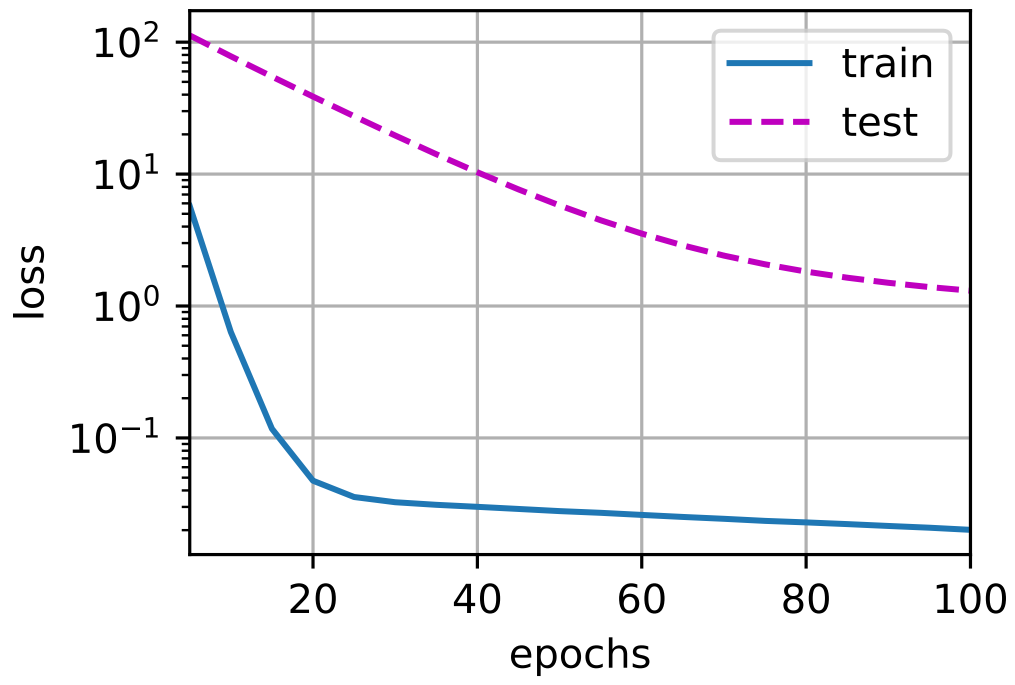
In [16]:
```
train_concise(0)
```

w的L2范数： 12.510650634765625



In [17]:
```
train_concise(3)
```

w的L2范数： 0.4902324378490448



3. 暂退法（Dropout）

在标准Dropout正则化中，通过按保留（未丢弃）的节点的分数进行规范化来消除每一层的偏差。 换言之，每个中间活性值h以*暂退概率*p由随机变量h'替换，如下所示：

$$h' = \begin{cases} 0 & \text{概率为 } p \\ \frac{h}{1-p} & \text{其他情况} \end{cases}$$

根据此模型的设计，其期望值保持不变，即$E[h'] = h$。

实践中的Dropout

对于一个带有1个隐藏层和5个隐藏单元的多层感知机。 当我们将Dropout应用到隐藏层，以p的概率将隐藏单元置为零时， 结果可以看作是一个只包含原始神经元子集的网络。

比如在下图中，删除了h_2和h_5， 因此输出的计算不再依赖于h_2或h_5，并且它们各自的梯度在执行反向传播时也会消失。 这样，输出层的计算不能过度依赖于$h_1, \ldots, h_5$的任何一个元素。

dropout前后的多层感知机

通常，在测试时不用Dropout。 给定一个训练好的模型和一个新的样本，我们不会丢弃任何节点，因此不需要标准化。

然而也有一些例外：一些研究人员在测试时使用Dropout， 用于估计神经网络预测的"不确定性"：

> 如果通过许多不同的Dropout遮盖后得到的预测结果都是一致的，那么我们可以说网络发挥更稳定。

从零开始实现

要实现单层的Dropout函数， 从均匀分布$U[0, 1]$中抽取样本，样本数与这层神经网络的维度一致。 然后保留那些对应样本大于p的节点，把剩下的丢弃。

在下面的代码中，**我们实现 `dropout_layer` 函数**。

In [18]:
```
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)
```

通过下面几个例子来**测试 `dropout_layer` 函数**。

将输入 `X` 通过暂退法操作，暂退概率分别为0、0.5和1。

In [19]:
```
X= torch.arange(16, dtype = torch.float32).reshape((2, 8))
print(X)
print(dropout_layer(X, 0.))
print(dropout_layer(X, 0.5))
print(dropout_layer(X, 1.))
```

```
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  0.,  0.,  6.,  0., 10., 12.,  0.],
        [ 0.,  0., 20., 22.,  0., 26., 28., 30.]])
tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
```

定义模型参数

使用Fashion-MNIST数据集,并定义**具有两个隐藏层的多层感知机，每个隐藏层包含256个单元。**

定义模型

我们可以将Dropout应用于每个隐藏层的输出（在激活函数之后）， 并且可以为每一层分别设置暂退概率：常见的技巧是在靠近输入层的地方设置较低的暂退概率。

下面的模型将第一个和第二个隐藏层的暂退概率分别设置为0.2和0.5， 并且Dropout只在训练期间有效。

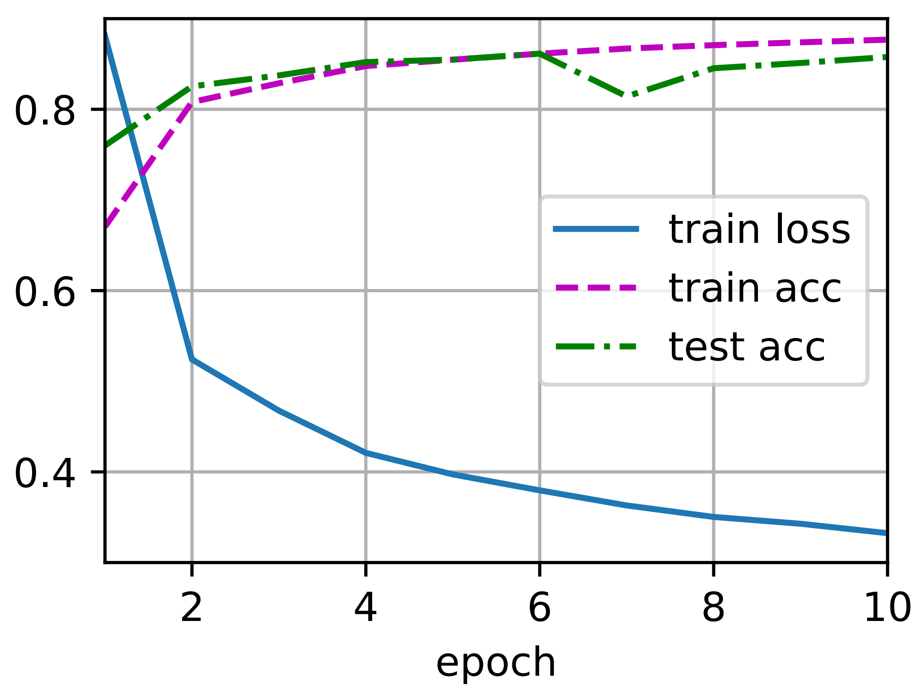
In [20]:
```
dropout1, dropout2 = 0.2, 0.5
num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256
class Net(nn.Module):
    def __init__(self, num_inputs, num_outputs, num_hiddens1, num_hiddens2,
                 is_training = True):
        super(Net, self).__init__()
        self.num_inputs = 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)
```

训练和测试

In [21]:
```
num_epochs, lr, batch_size = 10, 0.5, 256
loss = nn.CrossEntropyLoss(reduction='none')
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)
# d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, 'cuda:0')
```



简洁实现

如果使用深度学习框架的高级API，只需在每个全连接层之后添加一个`Dropout`层， 将暂退概率作为唯一的参数传递给它的构造函数。

在训练时，`Dropout`层将根据指定的暂退概率随机丢弃上一层的输出（相当于下一层的输入）。

在测试时，`Dropout`层仅传递数据。

In [22]:
```
net = nn.Sequential(nn.Flatten(),
        nn.Linear(784, 256),
        nn.ReLU(),
        # 在第一个全连接层之后添加一个dropout层
        nn.Dropout(dropout1),
        nn.Linear(256, 256),
        nn.ReLU(),
        # 在第二个全连接层之后添加一个dropout层
        nn.Dropout(dropout2),
        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);
```

训练和测试

In [23]:
```
trainer = torch.optim.SGD(net.parameters(), lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```