模型选择、欠拟合和过拟合

在前几节基于Fashion-MNIST数据集的实验中，我们评价了机器学习模型在训练数据集和测试数据集上的表现。如果你改变过实验中的模型结构或者超参数，你也许发现了：当模型在训练数据集上更准确时，它在测试数据集上却不一定更准确。这是为什么呢？

训练误差和泛化误差

在解释上述现象之前，我们需要区分训练误差（training error）和泛化误差（generalization error）。通俗来讲，前者指模型在训练数据集上表现出的误差，后者指模型在任意一个测试数据样本上表现出的误差的期望，并常常通过测试数据集上的误差来近似。计算训练误差和泛化误差可以使用之前介绍过的损失函数，例如线性回归用到的平方损失函数和softmax回归用到的交叉熵损失函数。

让我们以高考为例来直观地解释训练误差和泛化误差这两个概念。训练误差可以认为是做往年高考试题（训练题）时的错误率，泛化误差则可以通过真正参加高考（测试题）时的答题错误率来近似。假设训练题和测试题都随机采样于一个未知的依照相同考纲的巨大试题库。如果让一名未学习中学知识的小学生去答题，那么测试题和训练题的答题错误率可能很相近。但如果换成一名反复练习训练题的高三备考生答题，即使在训练题上做到了错误率为0，也不代表真实的高考成绩会如此。

在机器学习里，我们通常假设训练数据集（训练题）和测试数据集（测试题）里的每一个样本都是从同一个概率分布中相互独立地生成的。基于该独立同分布假设，给定任意一个机器学习模型（含参数），它的训练误差的期望和泛化误差都是一样的。例如，如果我们将模型参数设成随机值（小学生），那么训练误差和泛化误差会非常相近。但我们从前面几节中已经了解到，模型的参数是通过在训练数据集上训练模型而学习出的，参数的选择依据了最小化训练误差（高三备考生）。所以，训练误差的期望小于或等于泛化误差。也就是说，一般情况下，由训练数据集学到的模型参数会使模型在训练数据集上的表现优于或等于在测试数据集上的表现。由于无法从训练误差估计泛化误差，一味地降低训练误差并不意味着泛化误差一定会降低。

机器学习模型应关注降低泛化误差。

模型选择

在机器学习中，通常需要评估若干候选模型的表现并从中选择模型。这一过程称为模型选择（model selection）。可供选择的候选模型可以是有着不同超参数的同类模型。以多层感知机为例，我们可以选择隐藏层的个数，以及每个隐藏层中隐藏单元个数和激活函数。为了得到有效的模型，我们通常要在模型选择上下一番功夫。下面，我们来描述模型选择中经常使用的验证数据集（validation data set）。

验证数据集

从严格意义上讲，测试集只能在所有超参数和模型参数选定后使用一次。不可以使用测试数据选择模型，如调参。由于无法从训练误差估计泛化误差，因此也不应只依赖训练数据选择模型。鉴于此，我们可以预留一部分在训练数据集和测试数据集以外的数据来进行模型选择。这部分数据被称为验证数据集，简称验证集（validation set）。例如，我们可以从给定的训练集中随机选取一小部分作为验证集，而将剩余部分作为真正的训练集。

然而在实际应用中，由于数据不容易获取，测试数据极少只使用一次就丢弃。因此，实践中验证数据集和测试数据集的界限可能比较模糊。从严格意义上讲，除非明确说明，否则本书中实验所使用的测试集应为验证集，实验报告的测试结果（如测试准确率）应为验证结果（如验证准确率）。

k 折交叉验证

由于验证数据集不参与模型训练，当训练数据不够用时，预留大量的验证数据显得太奢侈。一种改善的方法是k折交叉验证（k-fold cross-validation）。在k折交叉验证中，我们把原始训练数据集分割成k个不重合的子数据集，然后我们做k次模型训练和验证。每一次，我们使用一个子数据集验证模型，并使用其他k−1个子数据集来训练模型。在这k次训练和验证中，每次用来验证模型的子数据集都不同。最后，我们对这k次训练误差和验证误差分别求平均。

欠拟合和过拟合

接下来，我们将探究模型训练中经常出现的两类典型问题：一类是模型无法得到较低的训练误差，我们将这一现象称作欠拟合（underfitting）；另一类是模型的训练误差远小于它在测试数据集上的误差，我们称该现象为过拟合（overfitting）。在实践中，我们要尽可能同时应对欠拟合和过拟合。虽然有很多因素可能导致这两种拟合问题，在这里我们重点讨论两个因素：模型复杂度和训练数据集大小。

模型复杂度

为了解释模型复杂度，我们以多项式函数拟合为例。给定一个由标量数据特征x和对应的标量标签y组成的训练数据集，多项式函数拟合的目标是找一个K阶多项式函数

$$\hat{y} = b + \sum_{k=1}^{K} x^k w_k$$

来近似y。在上式中，wk是模型的权重参数，b是偏差参数。与线性回归相同，多项式函数拟合也使用平方损失函数。特别地，一阶多项式函数拟合又叫线性函数拟合。

因为高阶多项式函数模型参数更多，模型函数的选择空间更大，所以高阶多项式函数比低阶多项式函数的复杂度更高。因此，高阶多项式函数比低阶多项式函数更容易在相同的训练数据集上得到更低的训练误差。给定训练数据集，模型复杂度和误差之间的关系通常如图3.4所示。给定训练数据集，如果模型的复杂度过低，很容易出现欠拟合；如果模型复杂度过高，很容易出现过拟合。应对欠拟合和过拟合的一个办法是针对数据集选择合适复杂度的模型。

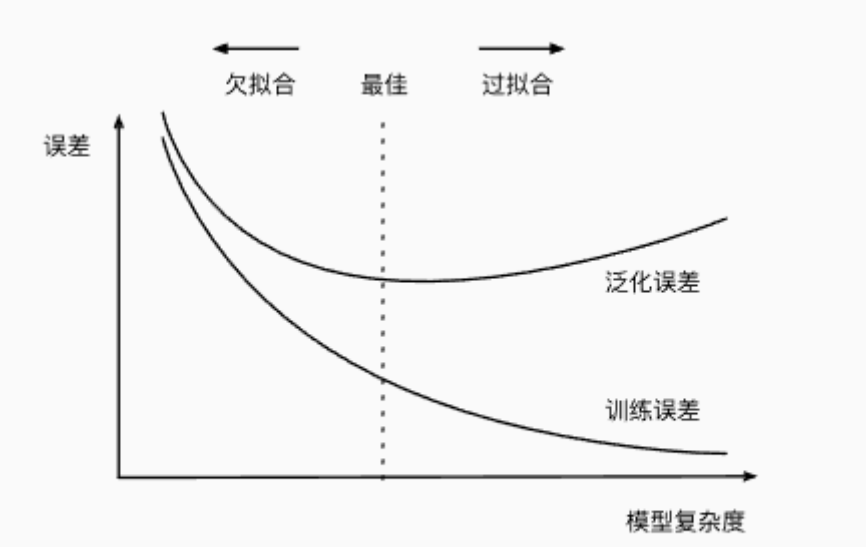


图 3.4 模型复杂度对欠拟合和过拟合的影响

训练数据集大小

影响欠拟合和过拟合的另一个重要因素是训练数据集的大小。一般来说，如果训练数据集中样本数过少，特别是比模型参数数量（按元素计）更少时，过拟合更容易发生。此外，泛化误差不会随训练数据集里样本数量增加而增大。因此，在计算资源允许的范围之内，我们通常希望训练数据集大一些，特别是在模型复杂度较高时，如层数较多的深度学习模型。

多项式函数拟合实验

为了理解模型复杂度和训练数据集大小对欠拟合和过拟合的影响，下面我们以多项式函数拟合为例来实验。首先导入实验需要的包或模块
我们将生成一个人工数据集。在训练数据集和测试数据集中，给定样本特征x，我们使用如下的三阶多项式函数来生成该样本的标签：

xxxxxxxxxx python

```
import d2l as d2l
from mxnet import autograd, gluon, nd
from mxnet.gluon import data as gdata, loss as gloss, nn

n_train, n_test, true_w, true_b = 100, 100, [1.2, -3.4, 5.6], 5
features = nd.random.normal(shape=(n_train + n_test, 1))
poly_features = nd.concat(features, nd.power(features, 2),
                          nd.power(features, 3))
labels = (true_w[0] * poly_features[:, 0] + true_w[1] * poly_features[:, 1]
          + true_w[2] * poly_features[:, 2] + true_b)
labels += nd.random.normal(scale=0.1, shape=labels.shape)

features[:2], poly_features[:2], labels[:2]
```

```
F:\minconda\envs\d2l\lib\site-packages\mxnet\numpy\utils.py:37: FutureWarning: In the future `np.bool` will be defined as the corresponding NumPy scalar.
  bool = onp.bool
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

Cell In[1], line 2
      1 import d2l as d2l
----> 2 from mxnet import autograd, gluon, nd
      3 from mxnet.gluon import data as gdata, loss as gloss, nn
      5 n_train, n_test, true_w, true_b = 100, 100, [1.2, -3.4, 5.6], 5
```

```
File F:\minconda\envs\d2l\lib\site-packages\mxnet\__init__.py:33
     30 # version info
     31 __version__ = base.__version__
---> 33 from . import contrib
     34 from . import ndarray
     35 from . import ndarray as nd
```

```
File F:\minconda\envs\d2l\lib\site-packages\mxnet\contrib\__init__.py:30
     27 from . import autograd
     28 from . import tensorboard
---> 30 from . import text
     31 from . import onnx
     32 from . import io
```

```
File F:\minconda\envs\d2l\lib\site-packages\mxnet\contrib\text\__init__.py:23
     21 from . import utils
     22 from . import vocab
---> 23 from . import embedding
```

```
File F:\minconda\envs\d2l\lib\site-packages\mxnet\contrib\text\embedding.py:36
     34 from ... import base
     35 from ...util import is_np_array
---> 36 from ... import numpy as _mx_np
     37 from ... import numpy_extension as _mx_npx
     40 def register(embedding_cls):
```

```
File F:\minconda\envs\d2l\lib\site-packages\mxnet\numpy\__init__.py:23
     21 from . import random
     22 from . import linalg
---> 23 from .multiarray import *  # pylint: disable=wildcard-import
     24 from . import _op
     25 from . import _register
```

```
File F:\minconda\envs\d2l\lib\site-packages\mxnet\numpy\multiarray.py:47
     45 from ..ndarray.numpy import _internal as _npi
     46 from ..ndarray.ndarray import _storage_type, from_numpy
---> 47 from .utils import _get_np_op
     48 from .fallback import *  # pylint: disable=wildcard-import,unused-
wildcard-import
     49 from . import fallback
```

```
File F:\minconda\envs\d2l\lib\site-packages\mxnet\numpy\utils.py:37
     35 int64 = onp.int64
     36 bool_ = onp.bool_
---> 37 bool = onp.bool
     39 pi = onp.pi
     40 inf = onp.inf
```

```
File F:\minconda\envs\d2l\lib\site-packages\numpy\__init__.py:353, in
__getattr__(attr)
    348     warnings.warn(
    349         f"In the future `np.{attr}` will be defined as the "
    350         "corresponding NumPy scalar.", FutureWarning, stacklevel=2)
    352 if attr in __former_attrs__:
--> 353     raise AttributeError(__former_attrs__[attr])
    355 if attr == 'testing':
    356     import numpy.testing as testing
```

```
AttributeError: module 'numpy' has no attribute 'bool'.
`np.bool` was a deprecated alias for the builtin `bool`. To avoid this error in
existing code, use `bool` by itself. Doing this will not modify any behavior and
is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
The aliases was originally deprecated in NumPy 1.20; for more details and
guidance see the original release note at:
    https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
```

定义、训练和测试模型

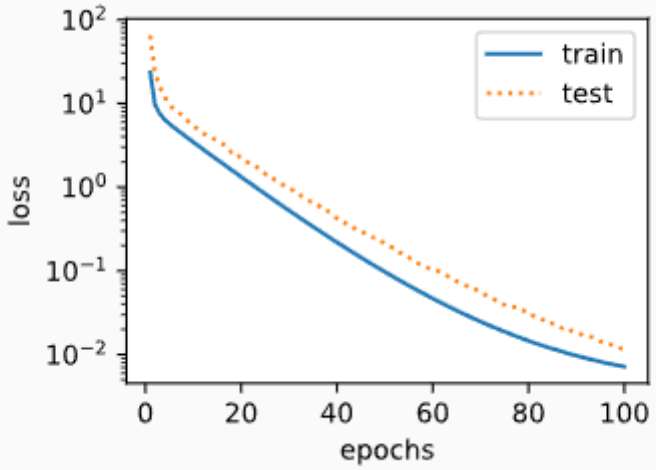
```
# 本函数已保存在d2lzh包中方便以后使用
def semilogy(x_vals, y_vals, x_label, y_label, x2_vals=None, y2_vals=None,
             legend=None, figsize=(3.5, 2.5)):
    d2l.set_figsize(figsize)
    d2l.plt.xlabel(x_label)
    d2l.plt.ylabel(y_label)
    d2l.plt.semilogy(x_vals, y_vals)
    if x2_vals and y2_vals:
        d2l.plt.semilogy(x2_vals, y2_vals, linestyle=':')
        d2l.plt.legend(legend)
```

和线性回归一样，多项式函数拟合也使用平方损失函数。因为我们将尝试使用不同复杂度的模型来拟合生成的数据集，所以我们把模型定义部分放在fit_and_plot函数中。多项式函数拟合的训练和测试步骤与“softmax回归的从零开始实现”一节介绍的softmax回归中的相关步骤类似。

```
num_epochs, loss = 100, gloss.L2Loss()

def fit_and_plot(train_features, test_features, train_labels, test_labels):
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize()
    batch_size = min(10, train_labels.shape[0])
    train_iter = gdata.DataLoader(gdata.ArrayDataset(
        train_features, train_labels), batch_size, shuffle=True)
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': 0.01})
    train_ls, test_ls = [], []
    for _ in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        train_ls.append(loss(net(train_features),
                             train_labels).mean().asscalar())
        test_ls.append(loss(net(test_features),
                            test_labels).mean().asscalar())
    print('final epoch: train loss', train_ls[-1], 'test loss', test_ls[-1])
    semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
             range(1, num_epochs + 1), test_ls, ['train', 'test'])
    print('weight:', net[0].weight.data().asnumpy(),
          '\nbias:', net[0].bias.data().asnumpy())
```

三阶多项式函数拟合（正常）

我们先使用与数据生成函数同阶的三阶多项式函数拟合。实验表明，这个模型的训练误差和在测试数据集的误差都较低。训练出的模型参数也接近真实值：w1=1.2,w2=−3.4,w3=5.6,b=5。



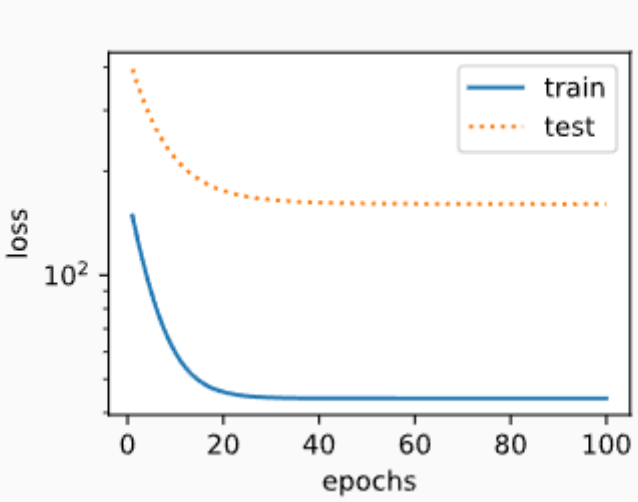

```
fit_and_plot(poly_features[:n_train, :], poly_features[n_train:, :],
             labels[:n_train], labels[n_train:])
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

Cell In[2], line 1
----> 1 fit_and_plot(poly_features[:n_train, :], poly_features[n_train:, :],
      2              labels[:n_train], labels[n_train:])
```

```
NameError: name 'fit_and_plot' is not defined
```

线性函数拟合（欠拟合）

我们再试试线性函数拟合。很明显，该模型的训练误差在迭代早期下降后便很难继续降低。在完成最后一次迭代周期后，训练误差依旧很高。线性模型在非线性模型（如三阶多项式函数）生成的数据集上容易欠拟合。



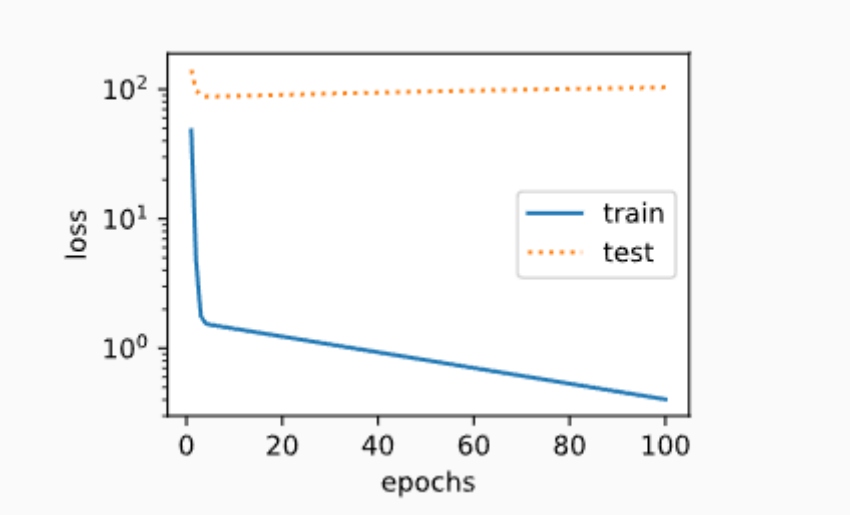
```
fit_and_plot(features[:n_train, :], features[n_train:, :], labels[:n_train],
             labels[n_train:])
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

Cell In[3], line 1
----> 1 fit_and_plot(features[:n_train, :], features[n_train:, :],
labels[:n_train],
      2              labels[n_train:])
```

```
NameError: name 'fit_and_plot' is not defined
```

训练样本不足（过拟合）

事实上，即便使用与数据生成模型同阶的三阶多项式函数模型，如果训练样本不足，该模型依然容易过拟合。让我们只使用两个样本来训练模型。显然，训练样本过少了，甚至少于模型参数的数量。这使模型显得过于复杂，以至于容易被训练数据中的噪声影响。在迭代过程中，尽管训练误差较低，但是测试数据集上的误差却很高。这是典型的过拟合现象。




```
fit_and_plot(poly_features[0:2, :], poly_features[n_train:, :], labels[0:2],
             labels[n_train:])
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

Cell In[4], line 1
----> 1 fit_and_plot(poly_features[0:2, :], poly_features[n_train:, :],
labels[0:2],
      2              labels[n_train:])
```

```
NameError: name 'fit_and_plot' is not defined
```