序列模型

统计工具

处理序列数据需要统计工具和新的深度神经网络架构。

用x_t表示价格，即在*时间步*（time step） $t \in \mathbb{Z}^+$时，观察到的价格x_t。 请注意，t对于本文中的序列通常是离散的，并在整数或其子集上变化。 假设一个交易员想在t日的股市中表现良好，于是通过以下途径预测x_t：

$$x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1).$$

自回归模型

为了实现这个预测，交易员可以使用回归模型。 仅有一个主要问题：输入数据的数量， 输入$x_{t-1}, \ldots, x_1$本身因t而异。 也就是说，输入数据的数量这个数字将会随着我们遇到的数据量的增加而增加， 因此需要一个近似方法来使这个计算变得容易处理。 本章后面的大部分内容将围绕着如何有效估计$P(x_t \mid x_{t-1}, \ldots, x_1)$展开。 简单地说，它归结为以下两种策略。

第一种策略，假设在现实情况下相当长的序列 $x_{t-1}, \ldots, x_1$可能是不必要的， 因此我们只需要满足某个长度为τ的时间跨度， 即使用观测序列$x_{t-1}, \ldots, x_{t-\tau}$。 当下获得的最直接的好处就是参数的数量总是不变的， 至少在$t > \tau$时如此，这就使我们能够训练一个上面提及的深度网络。 这种模型被称为*自回归模型*（autoregressive models）， 因为它们是对自己执行回归。

第二种策略，如 :numref: `fig_sequence-model` 所示， 是保留一些对过去观测的总结h_t， 并且同时更新预测$\hat{x}_t$和总结h_t。 这就产生了基于$\hat{x}_t = P(x_t \mid h_t)$估计$x_t$， 以及公式$h_t = g(h_{t-1}, x_{t-1})$更新的模型。 由于$h_t$从未被观测到，这类模型也被称为 *隐变量自回归模型*（latent autoregressive models）。

隐变量自回归模型
:label: `fig_sequence-model`

这两种情况都有一个显而易见的问题：如何生成训练数据？ 一个经典方法是使用历史观测来预测下一个未来观测。 显然，我们并不指望时间会停滞不前。 然而，一个常见的假设是虽然特定值x_t可能会改变， 但是序列本身的动力学不会改变。 这样的假设是合理的，因为新的动力学一定受新的数据影响， 而我们不可能用目前所掌握的数据来预测新的动力学。 统计学家称不变的动力学为*静止的*（stationary）。 因此，整个序列的估计值都将通过以下的方式获得：

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}, \ldots, x_1).$$

注意，如果我们处理的是离散的对象（如单词）， 而不是连续的数字，则上述的考虑仍然有效。 唯一的差别是，对于离散的对象， 我们需要使用分类器而不是回归模型来估计$P(x_t \mid x_{t-1}, \ldots, x_1)$。

马尔可夫模型

回想一下，在自回归模型的近似法中， 我们使用$x_{t-1}, \ldots, x_{t-\tau}$而不是$x_{t-1}, \ldots, x_1$来估计$x_t$。 只要这种是近似精确的，我们就说序列满足*马尔可夫条件*（Markov condition）。 特别是，如果$\tau = 1$，得到一个 *一阶马尔可夫模型*（first-order Markov model）， $P(x)$由下式给出：

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}) \text{ 当 } P(x_1 \mid x_0) = P(x_1).$$

当假设x_t仅是离散值时，这样的模型特别棒， 因为在这种情况下，使用动态规划可以沿着马尔可夫链精确地计算结果。 例如，我们可以高效地计算$P(x_{t+1} \mid x_{t-1})$：

$$\begin{aligned}
P(x_{t+1} \mid x_{t-1}) &= \frac{\sum_{x_t} P(x_{t+1}, x_t, x_{t-1})}{P(x_{t-1})} \\
&= \frac{\sum_{x_t} P(x_{t+1} \mid x_t, x_{t-1}) P(x_t, x_{t-1})}{P(x_{t-1})} \\
&= \sum_{x_t} P(x_{t+1} \mid x_t) P(x_t \mid x_{t-1})
\end{aligned}$$

利用这一事实，我们只需要考虑过去观察中的一个非常短的历史： $P(x_{t+1} \mid x_t, x_{t-1}) = P(x_{t+1} \mid x_t)$。 隐马尔可夫模型中的动态规划超出了本节的范围 （我们将在 :numref: `sec_bi_rnn` 再次遇到）， 而动态规划这些计算工具已经在控制算法和强化学习算法广泛使用。

因果关系

原则上，将$P(x_1, \ldots, x_T)$倒序展开也没什么问题。 毕竟，基于条件概率公式，我们总是可以写出：

$$P(x_1, \ldots, x_T) = \prod_{t=T}^{1} P(x_t \mid x_{t+1}, \ldots, x_T).$$

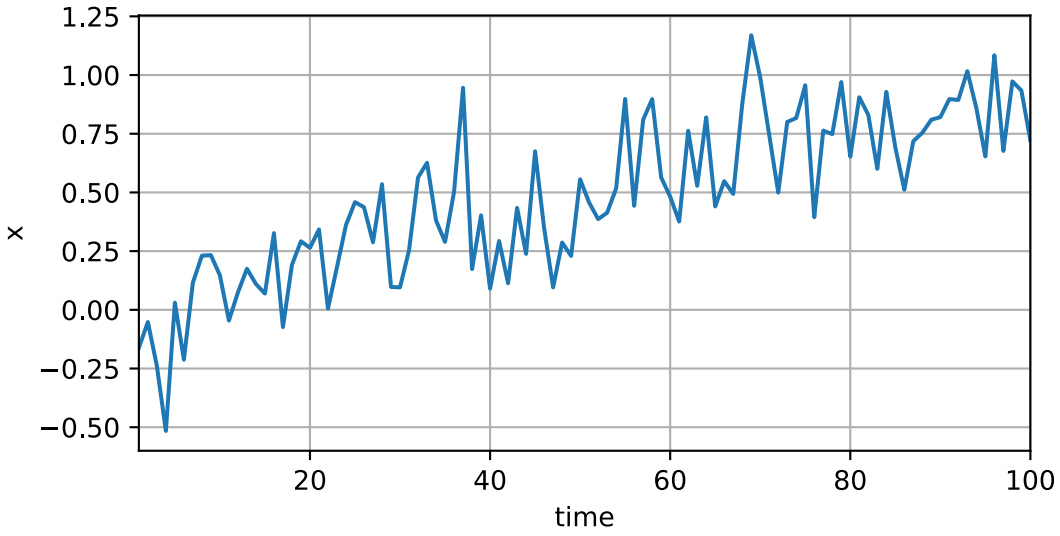
事实上，如果基于一个马尔可夫模型， 我们还可以得到一个反向的条件概率分布。 然而，在许多情况下，数据存在一个自然的方向，即在时间上是前进的。 很明显，未来的事件不能影响过去。 因此，如果我们改变x_t，可能会影响未来发生的事情x_{t+1}，但不能反过来。 也就是说，如果我们改变x_t，基于过去事件得到的分布不会改变。 因此，解释$P(x_{t+1} \mid x_t)$应该比解释$P(x_t \mid x_{t+1})$更容易。 例如，在某些情况下，对于某些可加性噪声ϵ， 显然我们可以找到$x_{t+1} = f(x_t) + \epsilon$， 而反之则不行 :cite: `Hoyer.Janzing.Mooij.ea.2009` 。 这是个好消息，因为这个前进方向通常也是我们感兴趣的方向。 彼得斯等人写的这本书 :cite: `Peters.Janzing.Scholkopf.2017` 已经解释了关于这个主题的更多内容，而我们仅仅触及了它的皮毛。

训练

在了解了上述统计工具后，让我们在实践中尝试一下！ 首先，我们生成一些数据：(**使用正弦函数和一些可加性噪声来生成序列数据， 时间步为$1, 2, \ldots, 100$。**)

In [20]:
```
%matplotlib inline
import torch
from torch import nn
from d2l import torch as d2l
```

In [21]:
```
T = 100  # Time Series of length 500
time = torch.arange(1, T + 1, dtype=torch.float32)
x = torch.sin(0.01 * time) + torch.normal(0, 0.2, (T,))
d2l.plot(time, [x], 'time', 'x', xlim=[1, 100], figsize=(6, 3))
```



接下来，我们将这个序列转换为模型的“特征 - 标签”（feature-label）对。 基于嵌入维度τ，我们[**将数据映射为数据对**$y_t = x_t$ **和**$\mathbf{x}_t = [x_{t-\tau}, \ldots, x_{t-1}]$。] 你可能已经注意到，这比我们提供的数据样本少了τ个， 因为我们没有足够的历史记录来描述前τ个数据样本。 一个简单的解决办法是：如果拥有足够长的序列就丢弃这几项； 另一个方法是用零填充序列。 在这里，我们仅使用前600个“特征 - 标签”对进行训练。

In [22]:
```
tau = 4
features = torch.zeros((T - tau, tau))
for i in range(tau):
    features[:, i] = x[i: T - tau + i]
labels = x[tau:].reshape((-1, 1))
```

In [23]:
```
batch_size, n_train = 16, 60
# 只有前n_train个样本用于训练
train_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                            batch_size, is_train=True)
```

在这里，我们[**使用一个相当简单的架构训练模型： 一个拥有两个全连接层的多层感知机**]，ReLU激活函数和平方损失。

In [24]:
```
# 初始化网络权重的函数
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)

# 一个简单的多层感知机
# 5*10 + 11*1 = 61 param.
def get_net():
    net = nn.Sequential(nn.Linear(4, 10),
                        nn.ReLU(),
                        nn.Linear(10, 1))
    net.apply(init_weights)
    return net

# 平方损失。注意：MSELoss计算平方误差时不带系数1/2
loss = nn.MSELoss(reduction='none')
```

现在，准备[**训练模型**]了。实现下面的训练代码的方式与前面几节（如 :numref: `sec_linear_concise` ）中的循环训练基本相同。因此，我们不会深入探讨太多细节。

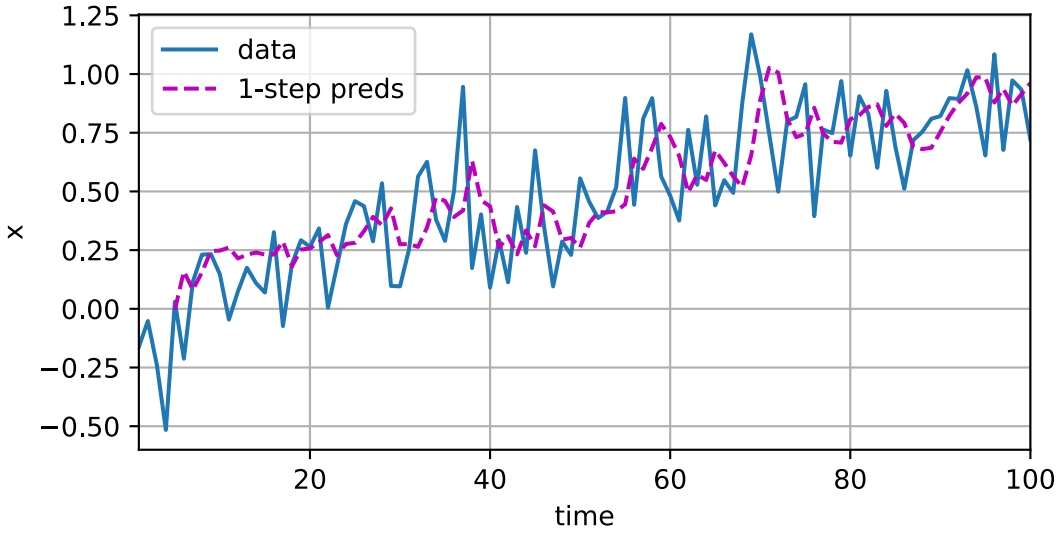
In [29]:
```
def train(net, train_iter, loss, epochs, lr):
    trainer = torch.optim.Adam(net.parameters(), lr)
    for epoch in range(epochs):
        for X, y in train_iter:
            trainer.zero_grad()
            l = loss(net(X), y)
            l.sum().backward()
            trainer.step()
        print(f'epoch {epoch + 1}, '
              f'loss: {d2l.evaluate_loss(net, train_iter, loss):f}')

net = get_net()
train(net, train_iter, loss, 5, 0.01)
```

```
epoch 1, loss: 0.071353
epoch 2, loss: 0.049345
epoch 3, loss: 0.060827
epoch 4, loss: 0.052482
epoch 5, loss: 0.046144
```

预测

由于训练损失很小，因此我们期望模型能有很好的工作效果。 让我们看看这在实践中意味着什么。 首先是检查[**模型预测下一个时间步**]的能力， 也就是*单步预测*（one-step-ahead prediction）。

In [30]:
```
onestep_preds = net(features)
d2l.plot([time, time[tau:]],
         [x.detach().numpy(), onestep_preds.detach().numpy()], 'time',
         'x', legend=['data', '1-step preds'], xlim=[1, 100],
         figsize=(6, 3))
```



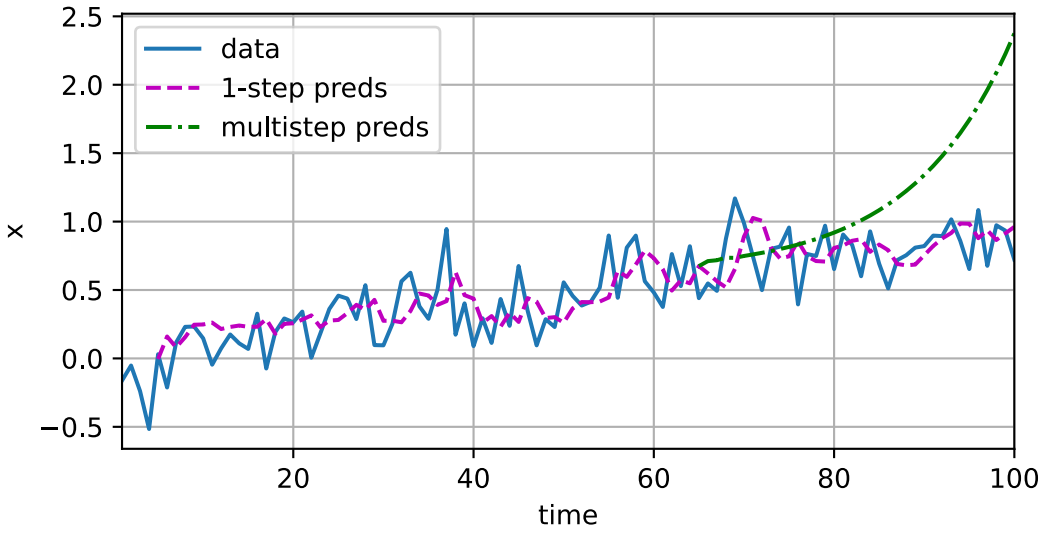
正如我们所料，单步预测效果不错。 即使这些预测的时间步超过了$60+4$（`n_train + tau`），其结果看起来仍然是可信的。 然而有一个小问题：如果数据观察序列的时间步只到64， 我们需要一步一步地向前迈进：

$$\begin{aligned}\hat{x}_{65} &= f(x_{61}, x_{62}, x_{63}, x_{64}),\\ \hat{x}_{66} &= f(x_{62}, x_{63}, x_{64}, \hat{x}_{65}),\\ \hat{x}_{67} &= f(x_{63}, x_{64}, \hat{x}_{65}, \hat{x}_{66}),\\ \hat{x}_{68} &= f(x_{64}, \hat{x}_{65}, \hat{x}_{66}, \hat{x}_{67}),\\ \hat{x}_{69} &= f(\hat{x}_{65}, \hat{x}_{66}, \hat{x}_{67}, \hat{x}_{68}),\\ &\ldots\end{aligned}$$

通常，对于直到x_t的观测序列，其在时间步$t+k$处的预测输出$\hat{x}_{t+k}$称为*k步预测*（k-step-ahead-prediction）。 由于我们的观察已经到了x_{64}，它的k步预测是$\hat{x}_{64+k}$。 换句话说，我们必须使用我们自己的预测（而不是原始数据）来[**进行多步预测**]。 让我们看看效果如何。

In [49]:
```
multistep_preds = torch.zeros(T)
#multistep_preds = x.clone().detach()
multistep_preds[: n_train + tau] = x[: n_train + tau]

for i in range(n_train + tau, T):
    multistep_preds[i] = net(multistep_preds[i-tau:i].reshape((1,-1)))
        #torch.cat((x[i - tau:i-1], multistep_preds[i-1].reshape(1))).reshape((1, -1)))
```

In [50]:
```
d2l.plot([time, time[tau:], time[n_train + tau:]],
         [x.detach().numpy(), onestep_preds.detach().numpy(),
          multistep_preds[n_train + tau:].detach().numpy()], 'time',
         'x', legend=['data', '1-step preds', 'multistep preds'],
         xlim=[1, 100], figsize=(6, 3))
```



如上面的例子所示，绿线的预测显然并不理想。 经过几个预测步骤之后，预测的结果很快就会衰减到一个常数。 为什么这个算法效果这么差呢？事实是由于错误的累积： 假设在步骤1之后，我们积累了一些错误$\epsilon_1 = \bar{\epsilon}$。 于是，步骤2的输入被扰动了$\epsilon_1$， 结果积累的误差是依照次序的$\epsilon_2 = \bar{\epsilon} + c\epsilon_1$， 其中$c$为某个常数，后面的预测误差依此类推。 因此误差可能会相当快地偏离真实的观测结果。 例如，未来24小时的天气预报往往相当准确， 但超过这一点，精度就会迅速下降。 我们将在本章及后续章节中讨论如何改进这一点。

基于$k = 1, 4, 16$，通过对整个序列预测的计算， 让我们[**更仔细地看一下k步预测**]的困难。

In [55]:
```
max_steps = 16
```

In [56]:
```
features = torch.zeros((T - tau - max_steps + 1, tau + max_steps))
# 列i (i<tau) 是来自x的观测，其时间步从 (i+1) 到 (i+T-tau-max_steps+1)
for i in range(tau):
    features[:, i] = x[i: i + T - tau - max_steps + 1]

# 列i (i>=tau) 是来自 (i-tau+1) 步的预测，其时间步从 (i+1) 到 (i+T-tau-max_steps+1)
for i in range(tau, tau + max_steps):
    features[:, i] = net(features[:, i - tau:i]).reshape(-1)
```

In [59]:
```
steps = (0, 1, 4, 16)
d2l.plot([time[tau + i - 1: T - max_steps + i] for i in steps],
         [features[:, (tau + i - 1)].detach().numpy() for i in steps], 'time', 'x',
         legend=[f'{i}-step preds' for i in steps], xlim=[5, 100],
         figsize=(6, 3))
```