第5章

循环神经网络

CHAPTER 5

任务导入：

目前处理的数据主要分为表格数据和图像数据。针对图像数据，可以采用卷积神经网络进行建模，需要充分考虑像素位置的影响。然而，大部分数据并非独立同分布的，而是具有一定的顺序性，因此需要针对性的模型进行处理。循环神经网络在这种情况下尤为有效，特别是在处理文本等序列数据时。为了解决数值不稳定性等问题，需要引入门控循环单元和长短期记忆网络。除此之外还将探讨编码器-解码器架构等技术，以应对各种序列学习问题。在此基础上本章最后会完成情感分析的任务。

知识目标：

（1）了解循环神经网络（RNN）及其现代变体 LSTM 与 GRU。

（2）了解编码器-解码器（Encoder-Decoder）方法。

能力目标：

（1）能自主搭建循环神经网络（RNN）。

（2）能自主搭建现代循环神经网络。

（3）能使用循环神经网络与卷积神经网络完成情感分析。

5.1　任务导学：基于深度学习方法的文本情感分析


在线视频

为了完成情感分析的任务，需要使模型能够理解语言的语法和语义，并且具备分析能力。本章将探讨如何使用循环神经网络来处理这一任务，重点解决序列数据处理的难点。在实践中将讨论数据预处理技术、模型架构设计以及训练和评估策略。通过本章的学习，读者将掌握情感分析任务中循环神经网络的应用技巧，为解决类似序列生成问题奠定基础。

5.2　任务知识


在线视频

5.2.1　循环神经网络

在观众欣赏影视作品的过程中，一位热情的影迷往往会对每一部作品进行评价。毕竟，优秀的电影值得更多的赞誉和认同。但实际情况远比这要复杂。随着时间的推移，观众对电影的评价可能会有显著的波动。心理学家为这些现象赋予了专业的术语。

(1) 锚定(anchoring)效应：观众的评价可能会受到他人观点的影响。例如，一部电影在获得重要奖项后，其评分可能会上升，尽管电影本身并未改变。这种效应可能会持续数月，直到人们逐渐忘记该电影的荣誉。研究表明，这种效应足以使评分上升超过 0.5 个百分点。

(2) 享乐适应(hedonic adaption)：观众很快会对较好的或较差的观影体验产生适应，并将其视为新的标准。例如，在连续观看了几部佳片后，观众可能会对下一部影片抱有更高的期望。因此，一部普通的电影在众多精彩的电影之后可能会被认为不尽如人意。

(3) 季节性(seasonality)：某些电影在特定季节更受欢迎。例如，圣诞节主题的电影在 8 月可能不会吸引太多观众。

(4) 有时，电影可能会因为制作人员的不当行为而变得不受待见。

(5) 一些电影因其质量欠佳而只能吸引少数观众。

简言之，电影评分并非一成不变。因此，运用时间序列分析可以更准确地推荐电影。此外，序列数据的应用远不止于电影评分。以下是一些其他场景的例子。

(1) 用户在使用软件时往往形成特定的习惯。例如，放学后学生更倾向于使用社交媒体应用，而交易时间内股市交易软件的使用频率会上升。

(2) 预测未来的股价比预测过去的股价更具挑战性，尽管两者都是对数字的预测。毕竟，预测未知比回顾已知要困难得多。在统计学中，前者称为外推法，后者称为内插法。

(3) 音乐、语言、文本和视频本质上是连续的序列。如果它们的顺序被打乱，原有的意义就会丧失。例如，文本标题“狗咬人”与“人咬狗”虽然由相同的字组成，但给人的冲击完全不同。

(4) 地震之间存在强烈的相关性。大规模地震发生后，通常会伴随几次强度较大的余震，这些余震的强度通常比没有大地震时的余震要大。实际上，地震在时间和空间上都具有相关性，余震往往在短时间内和近距离内发生。

(5) 人与人之间的互动也是连续的，这一点可以从社交媒体上的争论和讨论中得到体现。

处理这类序列数据需要运用统计工具和创新的深度学习网络架构。为了简单起见，以图 5-1 所示的股票价格(富时 100 指数)为例。

图 5-1　1984—2014 年富时 100 指数

其中,用 x_t 表示价格,当时间处于时间步(time step)$t \in \mathbf{Z}^+$ 时,价格为 x_t。需要注意的是,时间 t 对于上述序列来说往往是离散的,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:

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

首先,我们来探讨回归分析模型。交易员为了进行市场预测,可以采用回归分析方法。面临的主要挑战是如何确定输入数据的规模,因为随着时间的推移,输入数据集 $x_{t-1}, \cdots, x_1$ 会不断扩展。简言之,随着新数据的加入,输入集的规模也在不断增长,这就要求我们采用一种简化计算的方法。后续的讨论将主要集中于如何高效地预测 $P(x_t \mid x_{t-1}, \cdots, x_1)$ 的概率分布。简言之,我们可以采取两种主要策略。

第一种策略基于这样的假设:在实际情况中,可能不需要考虑整个历史序列 $x_{t-1}, \cdots, x_1$,而只需考虑最近 τ 期的数据,即序列 $x_{t-1}, \cdots, x_{t-\tau}$。这样做的直接优势是,一旦确定了 τ,模型参数的数量就保持不变,至少在 $t > \tau$ 的情况下是这样,这为训练深度学习网络提供了可能。这种模型称为自回归模型,因为它本质上是对自身进行回归分析。

第二种策略,如图 5-2 所示,涉及保留对过往观测的概括性描述 h_t,并同时更新预测结果 $\hat{x}_t$ 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 本身并不是直接观测到的数据,这类模型也称作隐变量自回归模型(latent autoregressive models)。

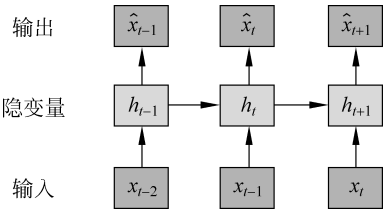

图 5-2　隐变量自回归模型

接下来,我们讨论马尔可夫模型。回想一下,在自回归模型的近似过程中,我们使用了 $x_{t-1}, \cdots, x_{t-\tau}$ 而不是 $x_{t-1}, \cdots, x_1$ 来估计 x_t。如果这种近似足够精确,那么我们可以说这个序列满足马尔可夫性质。特别是,如果 $\tau = 1$,得到一个一阶马尔可夫模型(first-order Markov model):

$$P(x_1, x_2, \cdots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}) \quad \text{当 } P(x_1 \mid x_0) = P(x_1)$$

当假设 x_t 仅是离散值时,这样的模型非常好,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,此处可以高效地计算 $P(x_{t+1} \mid x_{t-1})$:

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

利用这一事实，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此处学习知识点的范围，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。

原则上，将 $P(x_1, x_2, \cdots, x_T)$ 倒序展开也没什么问题。毕竟，基于条件概率公式总是可以写出：

$$P(x_1, x_2, \cdots, x_T) = \prod_{t=T}^{1} P(x_t \mid x_{t+1}, \cdots, x_T)$$

实际上，马尔可夫模型还能够推导出一个逆向的条件概率分布。但是，在许多实际情况中，数据遵循一个明确的时间方向，即时间是单向前进的。这是显而易见的，因为未来的事件不可能对过去产生影响。因此，如果改变 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，对于某些类型的可加性噪声 ε，我们可以明确地确定 $x_{t+1} = f(x_t) + \varepsilon$ 的关系，而反过来则不成立。而这个正向的时间推进方向，也正是我们通常关注的焦点。

现在，让我们开始进行模型训练的实践。在掌握了前述的统计工具之后，我们可以将理论应用于实际操作中。首先，我们可以创建一些数据样本：利用正弦波函数和一些随机的可加性噪声生成一个时间序列，时间点从 1 延伸到 1000，数据可视化结果如图 5-3 所示。

```
%matplotlib inline
import torch
from torch import nn
from d2l import torch as d2l
T = 1000                                 # 总共产生 1000 个点
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, 1000], figsize=(6, 3))
```

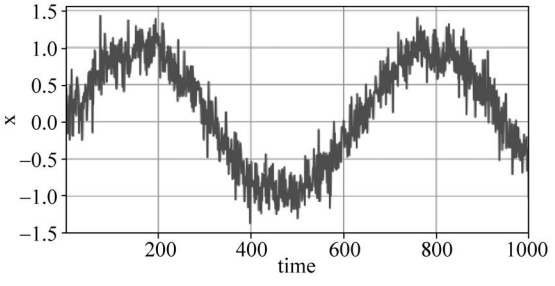

图 5-3　生成的数据

随后,我们需要将生成的时间序列数据转换为适合模型学习的格式,即特征-标签(feature-label)配对。这一过程涉及根据时间嵌入维度 τ,将序列数据重新构造为 $y_t = x_t$ 和 $x_t = [x_{t-\tau}, \cdots, x_{t-1}]$。由于序列的初始部分缺乏足够的历史数据来构建完整的 τ 个特征,这将导致生成的特征-标签对数量少于原始序列的长度。为了解决这个问题,我们可以选择两种方法:一是如果序列足够长,就忽略前 τ 个数据点;二是用零来填充这些缺失的数据点。

```
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))
    batch_size, n_train = 16, 600
    # 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,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。

```
    # 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 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循环训练基本相同。因此,此处不会深入探讨太多细节。

```
    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)
```

训练过程如图 5-4 所示。

训练完成后,模型将进入预测阶段。鉴于训练过程中的损失值较低,预期模型在实际应用中将表现出色。为了验证这一点,首先需要测试模型对下一个时间点的预测能力,即进行单步预测。这涉及使用模型基于当前和历史数据来预测紧接着的下一个数据点。通过比较

```
epoch 1, loss: 0.076846
epoch 2, loss: 0.056340
epoch 3, loss: 0.053779
epoch 4, loss: 0.056320
epoch 5, loss: 0.051650
```

图 5-4　序列模型训练过程

模型的预测结果与实际观测值，可以评估模型的准确性和可靠性，如图 5-5 所示。

```
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, 1000],
        figsize = (6, 3))
```

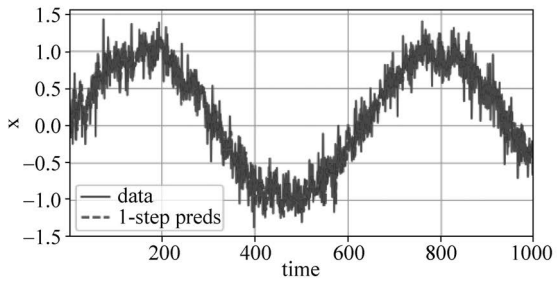

图 5-5　单步预测结果

正如预期，单步预测的性能表现良好。即便在预测超过训练数据范围的时间步 600＋4（n_train ＋ tau），预测结果依然显得合理。但是，如果观察到的数据序列仅到时间步 604，那么模型需要通过逐步预测的方式来进行更长时间的预测，即利用时间步 605 的预测值作为基础，逐步向后进行预测。这种方法称为递归预测或滚动预测，它允许模型在没有未来数据的情况下，逐步生成后续时间步的预测。

通常，对于直到 x_t 的观测序列，其在时间步 $t+k$ 处的预测输出称为 k 步提前预测（k-step-ahead-prediction）。由于观察点已经到了 x_{604}，它的 k 步预测是 $\hat{x}_{604+k}$。换句话说，必须使用自己的预测（而不是原始数据）来进行多步预测，预测结果如图 5-6 所示。

```
multistep_preds = torch.zeros(T)
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)))
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, 1000], figsize = (6, 3))
```

在上述示例中，多步预测的线条代表的预测表现并不尽如人意。随着预测步骤的增加，预测结果很快就会趋于一个常数值，显示出明显的衰减趋势。这种现象背后的原因可能是预测错误的累积效应：假设在步骤 1 之后积累了一些错误 $\epsilon_1=\bar{\epsilon}$。于是，步骤 2 的输入被扰动了 ϵ_1，结果积累的误差是依照次序的 $\epsilon_2=\bar{\epsilon}+c+c\epsilon_1$，其中 c 为某个常数，后面的预测误差以此类推。这种扰动会导致后续步骤的误差累积，形成一个递增的误差序列。随着时

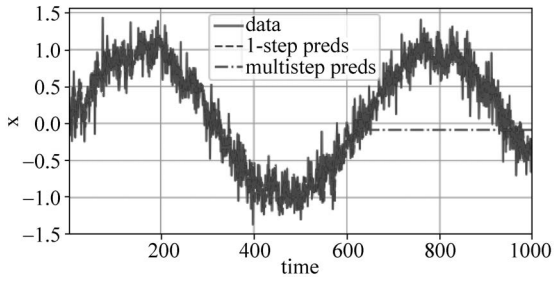

图 5-6　多步预测结果

间的推移,这种累积的误差可能会导致预测值迅速偏离实际观测值。

例如,短期天气预报,如未来 24 小时内的预测,通常相当准确。但是,一旦预测的时间范围超出这个短期,预测的准确性就会迅速降低。为了更深入地理解这种累积误差的影响,可以通过计算基于 $k=1,4,16,64$ 的 k 步预测来仔细观察预测的挑战。这种方法允许我们评估模型在不同预测范围下的稳定性和可靠性,同样的预测结果如图 5-7 所示。

```
    max_steps = 64
    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)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)到(i+T-tau-max_steps+
#1)for i in range(tau, tau + max_steps):
        features[:, i] = net(features[:, i - tau:i]).reshape(-1)
    steps = (1, 4, 16, 64)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, 1000],
             figsize=(6, 3))
```

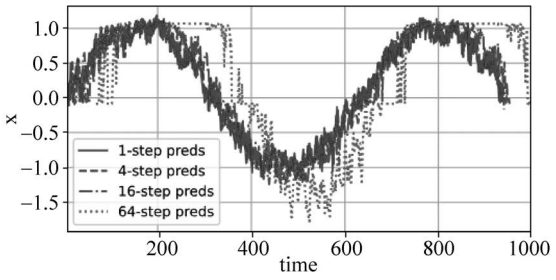

图 5-7　不同步数多步预测结果

以上例子清楚地说明了当试图预测更远的未来时,预测的质量是如何变化的。虽然“4 步预测”看起来仍然不错,但超过这个跨度的任何预测几乎都是无用的。

在深入了解循环神经网络之前,我们必须掌握文本数据的预处理方法。这是因为只有当文本被转换成适合语言模型处理的格式时,我们才能继续后续的工作。文本数据通常以一系列单词或字符的形式出现。本节将首先介绍文本预处理的常规步骤:

(1) 将文本内容加载为内存中的字符串。

(2) 将字符串分割成基本的语言单位,如单词或字符。

(3) 建立一个词汇表,将这些语言单位映射到对应的数字索引。

(4) 将文本转换成数字索引序列,以便模型能够处理。

首先是数据集的读取:从 H. G. Wells 的 *The Time Machine* 中提取文本数据。这个数据集相对较小,仅包含大约 30000 个单词,但它足以用于初步的实验。在实际应用中,文档集合可能包含数十亿个单词。下面的函数将文本数据读取为一个由多行文本组成的列表,每行文本作为一个单独的字符串。为了简化处理,在此忽略了标点符号和大小写的区别。

接下来是词元化过程:输入是一个文本行列表,列表中的每个元素代表一个文本序列,如一行文本。每个文本序列进一步被分割成一个词元列表,其中词元是文本的基本构成元素。最终,我们得到一个由词元列表构成的列表,其中的每个词元都是一个字符串。

紧接着是构建词汇表的步骤:由于词元的类型是字符串,而模型需要的是数字输入,因此我们需要将这些字符串类型的词元转换为模型可用的形式。我们首先将训练集中的所有文档合并,统计其中出现的唯一词元,这一统计结果被称为语料库。然后,根据每个唯一词元在语料库中的出现频率,为其分配一个从 0 开始的数字索引。为了降低模型的复杂性,通常会移除那些出现频率较低的词元。此外,对于那些在语料库中不存在或已被移除的词元,我们会将它们映射到一个特定的未知词元标记"< unk >"。我们还可以添加一些特殊的词元,如用于填充的"< pad >",表示序列开始的"< bos >"和表示序列结束的"< eos >",以完善词汇表的构建。

下面介绍 RNN 原理的相关内容。在 n 元语法模型中,其单词 x_t 在时间步 t 的条件概率仅取决于前面 $n-1$ 个单词。对于时间步 $t-(n-1)$之前的单词,如果想将其可能产生的影响合并到 x_t 上,需要增加 n,然而模型参数的数量也会随之呈指数增长,因为词表 γ 需要存储 $|\gamma|n$ 个数字,因此与其将 $P(x_t|x_{t-1},\cdots,x_{t-n+1})$模型化,不如使用隐变量模型:

$$P(x_t \mid x_{t-1},\cdots,x_1) \approx P(x_t \mid h_{t-1})$$

其中,h_{t-1} 是隐变量(hidden variable),也称为隐状态(hidden state),它存储了到时间步 $t-1$ 的序列信息。通常,可以基于当前输入 x_t 和先前隐状态 h_{t-1} 来计算时间步 t 处的隐状态:

$$h_t = f(x_t, h_{t-1})$$

在讨论的数学模型中,函数 f 代表的是一个精确的隐状态表达,而不是一个近似值。隐状态 h_t 的作用是保存到目前为止所有的观测数据,尽管这种做法可能会导致计算资源和存储空间的大量消耗。需要明确的是,隐藏层和隐状态是两个完全不同的概念。如前文所述,隐藏层是指在输入与输出之间不直接暴露的层级,而隐状态是指在特定时间点,基于之前的数据计算出的当前状态,这是技术定义下的输入,并且这一状态的确定完全依赖之前的时间步数据。

循环神经网络(Recurrent Neural Networks,RNN)是具有隐状态的神经网络。在介绍循环神经网络模型之前,先回顾一下多层感知机模型。

对于只有单隐藏层的多层感知机,设隐藏层的激活函数为 φ,给定一个小批量样本 $\boldsymbol{X} \in \mathbf{R}^{n\times d}$,其中批量大小为 n,输入维度为 d,则隐藏层的输出 $\boldsymbol{H} \in \mathbf{R}^{n\times h}$ 通过下式计算:

$$\boldsymbol{H} = \varphi(\boldsymbol{X}\boldsymbol{W}_{xh} + \boldsymbol{b}_h)$$

上式中,拥有的隐藏层权重参数为 $\boldsymbol{W}_{xh} \in \mathbf{R}^{d\times h}$,偏置参数为 $\boldsymbol{b}_h \in \mathbf{R}^{1\times h}$,以及隐藏单元的数目为 h。因此求和时可以应用广播机制。接下来,将隐状态 $\boldsymbol{H}$ 用作输出层的输入。输

出层由下式给出：

$$\boldsymbol{O}=\boldsymbol{H}\boldsymbol{W}_{hq}+\boldsymbol{b}_q$$

其中，$\boldsymbol{O}\in\mathbf{R}^{n\times q}$ 是输出变量；$\boldsymbol{W}_{hq}\in\mathbf{R}^{h\times q}$ 是权重参数；$\boldsymbol{b}_q\in\mathbf{R}^{1\times q}$ 是输出层的偏置参数。如果是分类问题，可以用 Softmax($\boldsymbol{O}$)来计算输出类别的概率分布。

这完全类似于解决回归问题，只要可以随机选择“特征-标签”对，并且通过自动微分和随机梯度下降能够学习网络参数就可以了。

有了隐状态后，情况就完全不同了。假设在时间步 t 有小批量输入 $\boldsymbol{X}_t\in\mathbf{R}^{n\times d}$。换言之，对于 n 个序列样本的小批量，$\boldsymbol{X}_t$ 的每一行对应于来自该序列的时间步 t 处的一个样本。接下来，用 $\boldsymbol{H}_t\in\mathbf{R}^{h\times h}$ 表示时间步 t 的隐状态。与多层感知机不同的是，在这里保存了前一个时间步的隐状态 $\boldsymbol{H}_{t-1}$，并引入了一个新的权重参数 $\boldsymbol{W}_{hh}\in\mathbf{R}^{h\times h}$ 来描述如何在当前时间步中使用前一个时间步的隐状态。具体来说，当前时间步隐状态由当前时间步的输入与前一个时间步的隐状态一起计算得出：

$$\boldsymbol{H}_t=\varphi(\boldsymbol{X}_t\boldsymbol{W}_{xh}+\boldsymbol{H}_{t-1}\boldsymbol{W}_{hh}+\boldsymbol{b}_h)$$

相较于之前的表达式，当前公式额外包含了一项 $\boldsymbol{H}_{t-1}\boldsymbol{W}_{hh}$，这一改变使得 h_t 的定义得以具体化。通过分析相邻时间步的隐状态 $\boldsymbol{H}_t$ 和 $\boldsymbol{H}_{t-1}$ 之间的相互作用，我们可以了解到这些状态是如何捕获并保持序列信息直至当前时间点的。它们就像是神经网络在当前时间步的状态记录或记忆，因此被称为隐状态。由于隐状态在当前时间步的定义与其在前一时间步的定义保持一致，上述公式中的计算过程形成了循环。正是基于这种循环计算的隐状态，神经网络得名循环神经网络。在循环神经网络中，执行这种计算的层被称为循环层。

循环神经网络的构建方法多种多样，上述公式定义的隐状态 RNN 是其中一种非常典型的架构。在时间步 t，输出层产生的输出与多层感知机中的计算过程相似：

$$\boldsymbol{O}_t=\boldsymbol{H}_t\boldsymbol{W}_{hq}+\boldsymbol{b}_q$$

循环神经网络的参数包括隐藏层的权重 $\boldsymbol{W}_{xh}\in\mathbf{R}^{d\times h}$，$\boldsymbol{W}_{hh}\in\mathbf{R}^{h\times h}$ 和偏置 $\boldsymbol{b}_h\in\mathbf{R}^{1\times h}$，以及输出层的权重 $\boldsymbol{W}_{hq}\in\mathbf{R}^{h\times q}$ 和偏置 $\boldsymbol{b}_q\in\mathbf{R}^{1\times q}$。值得一提的是，即使在不同的时间步，循环神经网络也总是使用这些模型参数。因此，循环神经网络的参数开销不会随着时间步的增加而增加。

图 5-8 展示了循环神经网络在 3 个相邻时间步的计算逻辑。在任意时间步 t，隐状态的计算可以被视为：

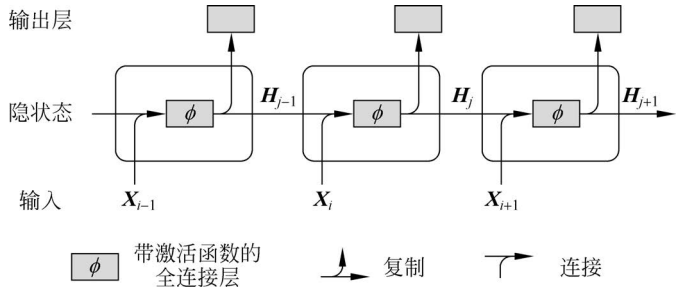

图 5-8　具有隐状态的循环神经网络

(1) 拼接当前时间步 t 的输入 $\boldsymbol{X}_t$ 和前一时间步 $t-1$ 的隐状态 $\boldsymbol{H}_{t-1}$；

(2) 将拼接的结果送入带有激活函数 φ 的全连接层。全连接层的输出是当前时间步 t

的隐状态 $\boldsymbol{H}_t$。

在本例中，模型参数是 $\boldsymbol{W}_{xh}$ 和 $\boldsymbol{W}_{hh}$ 的拼接，以及 $\boldsymbol{b}_h$ 的偏置，所有这些参数都来自定义 $\boldsymbol{H}_t$ 的公式。当前时间步 t 的隐状态 $\boldsymbol{H}_t$ 将参与计算下一时间步 $t+1$ 的隐状态 $\boldsymbol{H}_{t+1}$。而且 $\boldsymbol{H}_t$ 还将送入全连接输出层，用于计算当前时间步 t 的输出 $\boldsymbol{O}_t$。

基于循环神经网络的字符级语言模型的目标是通过考虑之前的词元以及当前的词元来预测接下来的词元，为此，原始的序列会向前移动一个词元位置作为目标标签。Bengio 及其同事首次提出利用神经网络进行语言建模的方法。现在，我们将探讨如何运用循环神经网络来构建这样一个语言模型。假设我们使用的小批量大小为 1，并且我们处理的文本序列是“machine”。为了简化模型训练的过程，可以考虑构建一个基于字符级的语言模型，这意味着我们将文本分解为字符而不是单词。图 5-9 展示了如何利用基于字符级的语言模型通过循环神经网络来预测下一个字符，其中输入序列为“machin”，而标签序列为“achine”。这种方法允许模型学习字符之间的模式，并据此预测文本序列中的下一个字符。

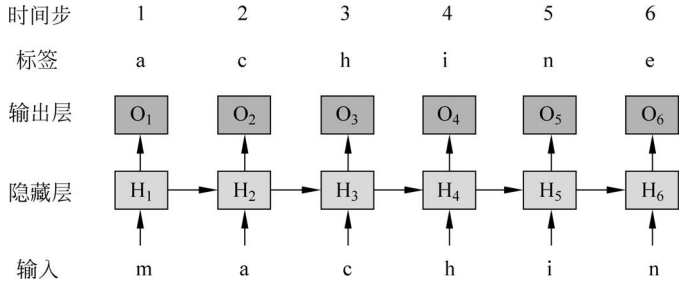

图 5-9　基于循环神经网络的字符级语言模型

在训练循环神经网络时，对每个时间步的输出层的输出应用 Softmax 函数，以便将输出转换为概率分布。接着，通过交叉熵损失函数来衡量模型输出与实际标签之间的差异。由于隐藏层中的隐状态是通过循环连接进行计算的，因此在图 5-9 所示的第 3 个时间步中，输出 O_3 是由前 3 个字符 m、a 和 c 共同决定的。由于在训练数据中，紧跟着 m、a、c 的下一个字符是 h，因此第 3 个时间步的损失将基于下一个字符的概率分布来计算，这个概率分布是由特征序列 m、a、c 以及当前时间步的标签 h 共同生成的。

在实际操作中，通常会使用大于 1 的小批量大小 n，这样可以通过同时处理多个数据样本来提高计算效率和稳定性。此外，每个词元都由一个 d 维的向量表示，这样的向量称为词元的嵌入(embedding)，它能够捕捉词元之间的语义关系。通过这种方式，循环神经网络能够学习如何根据给定的序列特征来预测下一个词元，从而逐步优化模型的性能。因此，在时间步 t 输入 $\boldsymbol{X}_t$ 将是一个 $n \times d$ 的矩阵。

评估模型性能的一个方法是使用序列的似然概率，但这个指标并不直观且难以直接比较。原因在于，较短的序列天生出现的概率更高，而像《战争与和平》这样的长文本出现的概率则相对较低。这种概率上的差异使得直接比较不同长度的文本序列变得不公平。

为了解决这个问题，我们可以借助信息论的概念。在 Softmax 回归的背景下，我们引入熵、惊异(surprise)和交叉熵等概念。如果我们的目标是压缩文本，那么我们可以根据当前的词元集合来预测下一个词元。一个优秀的语言模型应该能够更准确地预测下一个词元，因此在压缩序列时应该需要更少的信息(比特数)。因此，我们可以通过计算整个序列中

所有 n 个词元的交叉熵损失的平均值来评估模型的质量。这个平均交叉熵损失越低,表示模型对序列的预测越准确,从而也意味着模型的质量越高:

$$\frac{1}{n}\sum_{t=1}^{n}-\ln P(X_t \mid X_{t-1},\cdots,X_1)$$

在这个上下文中,P 代表语言模型给出的概率,而 X_t 是在时间步 t 观察到的具体词元。这种评估方式允许我们比较不同长度文档的性能,因为它考虑了模型预测下一个词元的准确性。在自然语言处理领域,科学家们通常更倾向于使用一个称为困惑度(perplexity)的指标来衡量模型的性能。

困惑度是一个衡量模型预测能力的指标,它反映了模型在预测下一个词元时的不确定性。一个较低的困惑度表示模型具有较高的预测准确性,因此是一个更好的语言模型。

现在,让我们开始从基础着手实现一个循环神经网络。这个网络将在 H. G. Wells 的《时光机器》文本数据集上进行训练。与之前的文本预处理步骤相同,我们首先需要读取并准备数据集。这包括将文本转换为模型可以理解的格式,通常是通过分词、建立词汇表,以及将文本转换为数字索引序列等步骤。完成这些预处理工作后,我们就可以开始构建和训练循环神经网络模型了。和前面文本预处理中介绍过的一样,先来读取数据集。

```
1.  %matplotlib inline                    #在 Jupyter Notebook 中使用
2.  import math
3.  import torch
4.  from torch import nn
5.  from torch.nn import functional as F
6.  from d2l import torch as d2l
7.  batch_size, num_steps = 32, 35
8.  train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

在处理 train_iter 时,每个词元都被转换为一个数字索引。如果直接将这些索引输入神经网络中,可能会对学习过程造成困难。为了提高模型的表现力,通常会将每个词元表示为一个特征向量,这种表示方法称为独热编码(one-hot encoding)。独热编码的基本思想是,对于给定的词表大小 N[词汇表中不同词元的数量,记为 len(vocab)],将每个词元的索引映射到一个长度为 N 的向量中,这个向量中除了与索引对应的位置为 1 外,其余位置都为 0。这样的向量称为该词元的独热向量。

在实际操作中,小批量数据的形状是一个二维张量,形式为(批量大小,时间步数)。使用 one_hot 函数可以将这样的二维张量转换为三维张量,其中最后一个维度的大小等于词表的大小[len(vocab)]。为了便于处理,通常需要调整输入张量的维度,以便得到一个形状为(时间步数,批量大小,词表大小)的张量。这样的形状调整使得我们可以更容易地沿着最外层维度(时间步)逐步更新小批量数据的隐状态。

接下来,我们需要初始化循环神经网络模型的参数。隐藏单元的数量 num_hiddens 是一个可以根据需要调整的超参数。在训练语言模型时,输入和输出都源自同一个词表,因此它们具有相同的维度,即词表的大小。这确保了模型的输入层和输出层能够正确地对接,使得模型能够有效地学习从输入序列到输出序列的映射。

```
9.  def get_params(vocab_size, num_hiddens, device):
10.         num_inputs = num_outputs = vocab_size
11.
```

```
        def normal(shape):
            return torch.randn(size = shape, device = device) * 0.01

        # 隐藏层参数
        W_xh = normal((num_inputs, num_hiddens))
        W_hh = normal((num_hiddens, num_hiddens))
        b_h = torch.zeros(num_hiddens, device = device)
        # 输出层参数
        W_hq = normal((num_hiddens, num_outputs))
        b_q = torch.zeros(num_outputs, device = device)
        # 附加梯度
        params = [W_xh, W_hh, b_h, W_hq, b_q]
        for param in params:
            param.requires_grad_(True)
        return params
```

为了定义循环神经网络模型，首先需要一个 init_rnn_state 函数在初始化时返回隐状态。这个函数的返回是一个张量，张量全用 0 填充，形状为(批量大小，隐藏单元数)。同时还会遇到隐状态包含多个变量的情况，而使用元组可以更容易处理一些。

```
    def init_rnn_state(batch_size, num_hiddens, device):
        return (torch.zeros((batch_size, num_hiddens), device = device), )
```

下面的 rnn 函数定义了如何在一个时间步内计算隐状态和输出。循环神经网络模型通过 inputs 最外层的维度实现循环，以便逐时间步更新小批量数据的隐状态 $\boldsymbol{H}$。此外，这里使用 tanh 函数作为激活函数。

```
    def rnn(inputs, state, params):
        # inputs 的形状:(时间步数,批量大小,词表大小)
        W_xh, W_hh, b_h, W_hq, b_q = params
        H, = state
        outputs = []
        # X 的形状:(批量大小,词表大小)
        for X in inputs:
            H = torch.tanh(torch.mm(X, W_xh) + torch.mm(H, W_hh) + b_h)
            Y = torch.mm(H, W_hq) + b_q
            outputs.append(Y)
        return torch.cat(outputs, dim = 0), (H,)
```

定义了所有需要的函数之后，接下来创建一个类来包装这些函数，并存储从零开始实现的循环神经网络模型的参数。

```
    class RNNModelScratch: #@save    """从零开始实现的循环神经网络模型"""
        def __init__(self, vocab_size, num_hiddens, device,
                     get_params, init_state, forward_fn):
            self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
            self.params = get_params(vocab_size, num_hiddens, device)
            self.init_state, self.forward_fn = init_state, forward_fn

        def __call__(self, X, state):
            X = F.one_hot(X.T, self.vocab_size).type(torch.float32)
            return self.forward_fn(X, state, self.params)

        def begin_state(self, batch_size, device):
            return self.init_state(batch_size, self.num_hiddens, device)
```

进入预测阶段,此处定义了一个函数,它的作用是基于用户输入的字符序列前缀来生成后续的字符。这个前缀是一个由多个字符组成的字符串。在处理前缀的每个字符时,模型会逐步传递并更新隐状态,但在开始的若干步骤中不产生输出。这一初始化阶段被称为预热期,模型在此时调整自身状态以适应上下文,而不进行预测。一旦预热期完成,模型的隐状态将更好地反映输入的上下文,此时模型开始进行预测并输出字符。

```
53.     def predict_ch8(prefix, num_preds, net, vocab, device):  #@save     """在 prefix 后
        面生成新字符"""
54.         state = net.begin_state(batch_size=1, device=device)
55.         outputs = [vocab[prefix[0]]]
56.         get_input = lambda: torch.tensor([outputs[-1]], device=device).reshape((1, 1))
57.         for y in prefix[1:]:                  # 预热期
58.             _, state = net(get_input(), state)
59.             outputs.append(vocab[y])
60.         for _ in range(num_preds):            # 预测 num_preds 步
61.             y, state = net(get_input(), state)
62.             outputs.append(int(y.argmax(dim=1).reshape(1)))
63.         return ''.join([vocab.idx_to_token[i] for i in outputs])
```

对于一个长度为 T 的序列,模型在迭代过程中会分别计算每个时间步的梯度,这会在反向传播中形成一条 $O(T)$ 长度的矩阵乘法操作链。当序列长度 T 较大时,这种方法可能会导致数值计算上的不稳定,如梯度爆炸或梯度消失的问题。为了维持循环神经网络模型的训练稳定性,通常会采用一些特殊的技术,其中之一就是梯度裁剪。

```
64.     def grad_clipping(net, theta):  #@save     """梯度裁剪"""
65.         if isinstance(net, nn.Module):
66.             params = [p for p in net.parameters() if p.requires_grad]
67.         else:
68.             params = net.params
69.         norm = torch.sqrt(sum(torch.sum((p.grad ** 2)) for p in params))
70.         if norm > theta:
71.             for param in params:
72.                 param.grad[:] *= theta / norm
```

在开始模型训练之前,我们定义了一个特定的函数,这个函数负责在一个训练周期内对模型进行迭代训练。这种训练方式与常规的训练方式主要有以下三点区别。

(1) 由于不同的序列数据采样策略(如随机采样和顺序分区),隐状态的初始化方式会有所不同。

(2) 在对模型参数进行更新之前,会先对梯度进行裁剪,这样做是为了在梯度爆炸的情况下保持模型的稳定性。

(3) 使用困惑度作为评价模型性能的指标,这样可以确保不同长度的序列之间具有可比性。

具体来说,当采用顺序分区时,隐状态仅在每个周期的开始时进行初始化。由于小批量数据中的连续子序列样本是相邻的,前一个小批量数据中最后一个样本的隐状态会被用来初始化下一个小批量数据中的第一个样本的隐状态。这样的设计允许序列的历史信息在一个训练周期内传递。然而,由于隐状态的计算依赖整个周期内的所有小批量数据,这使得梯度计算变得复杂。为了简化计算,我们在处理每个小批量数据之前先进行梯度分离,确保隐状态的梯度计算仅局限于单个小批量数据的时间步内。

在随机抽样的情况下，由于样本是在随机位置抽取的，因此每个训练周期都需要重新初始化隐状态。模型参数的更新是通过一个更新函数来完成的，这个更新函数可以是自定义的，如 d2l.sgd，也可以是深度学习框架提供的优化算法。

```
#@save
def train_epoch_ch8(net, train_iter, loss, updater, device, use_random_iter):
"""训练网络一个迭代周期"""
    state, timer = None, d2l.Timer()
    metric = d2l.Accumulator(2)       # 训练损失之和,词元数量
    for X, Y in train_iter:
        if state is None or use_random_iter:
            # 在第一次迭代或使用随机抽样时初始化 state
            state = net.begin_state(batch_size=X.shape[0], device=device)
        else:
            if isinstance(net, nn.Module) and not isinstance(state, tuple):
                # state 对于 nn.GRU 是个张量
                state.detach_()
            else:
                # state 对于 nn.LSTM 或对于我们从零开始实现的模型是个张量
                for s in state:
                    s.detach_()
        y = Y.T.reshape(-1)
        X, y = X.to(device), y.to(device)
        y_hat, state = net(X, state)
        l = loss(y_hat, y.long()).mean()
        if isinstance(updater, torch.optim.Optimizer):
            updater.zero_grad()
            l.backward()
            grad_clipping(net, 1)
            updater.step()
        else:
            l.backward()
            grad_clipping(net, 1)
            # 因为已经调用了 mean 函数
            updater(batch_size=1)
        metric.add(l * y.numel(), y.numel())
    return math.exp(metric[0] / metric[1]), metric[1] / timer.stop()
```

循环神经网络模型的训练函数既支持从零开始实现，也可以使用高级 API 来实现。

```
#@save
def train_ch8(net, train_iter, vocab, lr, num_epochs, device,
              use_random_iter=False):    """训练模型"""
    loss = nn.CrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', ylabel='perplexity',
                            legend=['train'], xlim=[10, num_epochs])
    # 初始化
    if isinstance(net, nn.Module):
        updater = torch.optim.SGD(net.parameters(), lr)
    else:
        updater = lambda batch_size: d2l.sgd(net.params, lr, batch_size)
    predict = lambda prefix: predict_ch8(prefix, 50, net, vocab, device)
    # 训练和预测
    for epoch in range(num_epochs):
        ppl, speed = train_epoch_ch8(
```

```
                net, train_iter, loss, updater, device, use_random_iter)
            if (epoch + 1) % 10 == 0:
                print(predict('time traveller'))
                animator.add(epoch + 1, [ppl])
        print(f'困惑度 {ppl:.1f}, {speed:.1f} 词元/秒 {str(device)}')
        print(predict('time traveller'))
        print(predict('traveller'))
```

现在训练循环神经网络模型。因为在数据集中只使用了 10000 个词元,所以模型需要更多的迭代周期来更好地收敛。

```
num_epochs, lr = 500, 1
train_ch8(net, train_iter, vocab, lr, num_epochs, d2l.try_gpu())
```

最后检查一下使用随机抽样方法的结果。

```
net = RNNModelScratch(len(vocab), num_hiddens, d2l.try_gpu(), get_params, init_rnn_
    state, rnn)train_ch8(net, train_iter, vocab, lr, num_epochs, d2l.try_gpu(),
                use_random_iter=True)
```

虽然上述从零实现的循环神经网络对了解知识内容具有指导意义,但并不方便。接下来将展示如何使用深度学习框架的高级 API 提供的函数更有效地实现相同的语言模型。仍然从读取时光机器数据集开始。

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

高级 API 提供了循环神经网络的实现。首先构造一个具有 256 个隐藏单元的单隐藏层的循环神经网络层,命名为 rnn_layer。目前还没有讨论多层循环神经网络的意义。在这个阶段,只需将多层理解为一层循环神经网络的输出被用作下一层循环神经网络的输入就足够了。同时使用张量来初始化隐状态,其形状为(隐藏层数,批量大小,隐藏单元数)。通过一个隐状态和一个输入就可以使用更新后的隐状态计算输出。需要强调的是,rnn_layer 的“输出”(Y)不涉及输出层的计算:它是指每个时间步的隐状态,这些隐状态可以用作后续输出层的输入。

```
num_hiddens = 256
rnn_layer = nn.RNN(len(vocab), num_hiddens)
state = torch.zeros((1, batch_size, num_hiddens))
X = torch.rand(size=(num_steps, batch_size, len(vocab)))
Y, state_new = rnn_layer(X, state)
```

接着为一个完整的循环神经网络模型定义了一个 RNNModel 类。注意,rnn_layer 只包含隐藏的循环层,还需要创建一个单独的输出层。

```
#@save
class RNNModel(nn.Module):    """循环神经网络模型"""
    def __init__(self, rnn_layer, vocab_size, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        self.rnn = rnn_layer
```

```
147.            self.vocab_size = vocab_size
148.            self.num_hiddens = self.rnn.hidden_size
149.            # 如果 RNN 是双向的(之后将介绍),则 num_directions 应该是 2,否则应该是 1
150.            if not self.rnn.bidirectional:
151.                self.num_directions = 1
152.                self.linear = nn.Linear(self.num_hiddens, self.vocab_size)
153.            else:
154.                self.num_directions = 2
155.                self.linear = nn.Linear(self.num_hiddens * 2, self.vocab_size)
156.
157.        def forward(self, inputs, state):
158.            X = F.one_hot(inputs.T.long(), self.vocab_size)
159.            X = X.to(torch.float32)
160.            Y, state = self.rnn(X, state)
161.            # 全连接层首先将 Y 的形状改为(时间步数 * 批量大小,隐藏单元数)
162.            # 它的输出形状是(时间步数 * 批量大小,词表大小).
163.            output = self.linear(Y.reshape((-1, Y.shape[-1])))
164.            return output, state
165.
166.        def begin_state(self, device, batch_size=1):
167.            if not isinstance(self.rnn, nn.LSTM):
168.                # nn.GRU 以张量作为隐状态
169.                return  torch.zeros((self.num_directions * self.rnn.num_layers,
170.                                     batch_size, self.num_hiddens),
171.                                     device=device)
172.            else:
173.                # nn.LSTM 以元组作为隐状态
174.                return (torch.zeros((
175.                   self.num_directions * self.rnn.num_layers,
176.                   batch_size, self.num_hiddens), device=device),
177.                       torch.zeros((
178.                          self.num_directions * self.rnn.num_layers,
179.                          batch_size, self.num_hiddens), device=device))
```

使用前面代码中定义的超参数调用 train_ch8,并且使用高级 API 训练模型。

```
180.    num_epochs, lr = 500, 1
181.    d2l.train_ch8(net, train_iter, vocab, lr, num_epochs, device)
```

与从零开始的实现方式相比,由于深度学习框架的高级 API 对代码进行了更多的优化,该模型在较短的时间内达到了较低的困惑度。

5.2.2　现代循环神经网络

在之前的讨论中,通过循环神经网络构建了一个语言模型来处理文本数据。但对于序列学习任务,尤其是当序列较长时,循环神经网络可能不足以应对挑战。循环神经网络面临的常见问题包括数值稳定性问题,即便采用了梯度裁剪等策略,仍然可能需要更高级的模型结构。为了解决这些问题,门控循环单元(Gate Recurrent Unit,GRU)和长短期记忆网络(Long Short-Term Memory,LSTM)被提出并广泛应用。

此外,语言建模仅代表了序列学习任务的一方面。在更广泛的序列学习场景中,输入和输出通常都是可变长度的序列。为了处理这类问题,我们介绍了基于循环神经网络的“编码器-解码器”架构,这种架构能够有效地将输入序列编码为固定长度的表示,然后将这个表示

解码为输出序列,从而适用多种序列到序列的任务。

GRU 与标准循环神经网络的核心差异在于:GRU 具备控制隐状态更新的门控机制。这使得模型能够决定何时保留当前隐状态,何时忽略新信息。这些决策是由模型学习得到的,有效解决了数值稳定性的问题。例如,模型可能在初始观测后选择不改变隐状态,或者忽略无关观测。此外,模型还能在必要时重置隐状态。

接下来,将从零开始实现 GRU 模型,并使用相同的时间机器数据集。

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

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

接下来初始化模型参数。从标准差为 0.01 的高斯分布中提取权重,并将偏置项设为 0,超参数 num_hiddens 定义隐藏单元的数量,实例化与更新门、重置门、候选隐状态和输出层相关的所有权重和偏置。

```
def get_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

        def normal(shape):
            return torch.randn(size = shape, device = device) * 0.01

        def three():
            return (normal((num_inputs, num_hiddens)),
                    normal((num_hiddens, num_hiddens)),
                    torch.zeros(num_hiddens, device = device))

        W_xz, W_hz, b_z = three()          # 更新门参数
        W_xr, W_hr, b_r = three()          # 重置门参数
        W_xh, W_hh, b_h = three()          # 候选隐状态参数
        # 输出层参数
        W_hq = normal((num_hiddens, num_outputs))
        b_q = torch.zeros(num_outputs, device = device)
        # 附加梯度
        params = [W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q]
        for param in params:
            param.requires_grad_(True)
        return params
```

然后定义模型。在实现中定义隐状态的初始化函数 init_gru_state。此函数返回一个形状为(批量大小,隐藏单元个数)的张量,张量的值全部为零。

```
    def gru(inputs, state, params):
        W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q = params
        H, = state
        outputs = []
        for X in inputs:
            Z = torch.sigmoid((X @ W_xz) + (H @ W_hz) + b_z)
            R = torch.sigmoid((X @ W_xr) + (H @ W_hr) + b_r)
            H_tilda = torch.tanh((X @ W_xh) + ((R * H) @ W_hh) + b_h)
            H = Z * H + (1 - Z) * H_tilda
```

```
            Y = H @ W_hq + b_q
            outputs.append(Y)
        return torch.cat(outputs, dim = 0), (H,)
```

最后训练与预测的工作方式与门控循环单元中的实现方式完全相同。训练结束后，分别打印输出训练集的困惑度，以及前缀“time traveler”和“traveler”的预测序列上的困惑度。

```
    vocab_size, num_hiddens, device = len(vocab), 256, d2l.try_gpu()
    num_epochs, lr = 500, 1
    model = d2l.RNNModelScratch(len(vocab), num_hiddens, device, get_params, init_gru_
state, gru)
    d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

还可以调用高级 API 来更简洁地实现 GRU。

```
    num_inputs = vocab_size
    gru_layer = nn.GRU(num_inputs, num_hiddens)
    model = d2l.RNNModel(gru_layer, len(vocab))
    model = model.to(device)
    d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

潜变量模型在处理长期依赖和短期信息丢失方面面临挑战。为了解决这些问题，长短期记忆网络被开发出来，它具备与门控循环单元类似的功能。

LSTM 的设计灵感来源于计算机逻辑门的概念。它引入了记忆单元，也称为单元，这是一种特殊的隐状态，用于存储额外信息。记忆单元由多个门控制。输出门负责决定何时从单元中输出信息，输入门决定何时向单元添加新数据，而遗忘门则管理着何时清除单元中的旧信息。这种设计模仿了 GRU 中的机制，通过专门的门控结构来决定何时保留或忽略输入信息。

接下来，从零开始实现长短期记忆网络。同样地，从加载时光机器数据集开始。

```
import torch
from torch import nn
from d2l import torch as d2l
batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

接下来是模型参数的初始化。如前所述，超参数 num_hiddens 定义隐藏单元的数量。按照标准差 0.01 的高斯分布初始化权重，并将偏置项设为 0。

```
def get_lstm_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size = shape, device = device) * 0.01

    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                torch.zeros(num_hiddens, device = device))

    W_xi, W_hi, b_i = three()          # 输入门参数
    W_xf, W_hf, b_f = three()          # 遗忘门参数
    W_xo, W_ho, b_o = three()          # 输出门参数
    W_xc, W_hc, b_c = three()          # 候选记忆元参数
```

```
        # 输出层参数
        W_hq = normal((num_hiddens, num_outputs))
        b_q = torch.zeros(num_outputs, device=device)
        # 附加梯度
        params = [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc,
                  b_c, W_hq, b_q]
        for param in params:
            param.requires_grad_(True)
        return params
```

然后是定义模型。在初始化函数中,长短期记忆网络的隐状态需要返回一个额外的记忆元,单元的值为0,形状为(批量大小,隐藏单元数)。因此得到以下的状态初始化。

```
    def init_lstm_state(batch_size, num_hiddens, device):
        return (torch.zeros((batch_size, num_hiddens), device=device),
                torch.zeros((batch_size, num_hiddens), device=device))
```

实际模型的定义与前面讨论的一样:提供3个门和1个额外的记忆元。请注意,只有隐状态才会传递到输出层,而记忆元 C_t 不直接参与输出计算。

```
    def lstm(inputs, state, params):
        [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c,
        W_hq, b_q] = params
        (H, C) = state
        outputs = []
        for X in inputs:
            I = torch.sigmoid((X @ W_xi) + (H @ W_hi) + b_i)
            F = torch.sigmoid((X @ W_xf) + (H @ W_hf) + b_f)
            O = torch.sigmoid((X @ W_xo) + (H @ W_ho) + b_o)
            C_tilda = torch.tanh((X @ W_xc) + (H @ W_hc) + b_c)
            C = F * C + I * C_tilda
            H = O * torch.tanh(C)
            Y = (H @ W_hq) + b_q
            outputs.append(Y)
        return torch.cat(outputs, dim=0), (H, C)
```

最后是训练与预测。通过实例化RNNModelScratch类来训练一个长短期记忆网络。

```
    vocab_size, num_hiddens, device = len(vocab), 256, d2l.try_gpu()
    num_epochs, lr = 500, 1
    model = d2l.RNNModelScratch(len(vocab), num_hiddens, device, get_lstm_params, init_
lstm_state, lstm)
    d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

同样地,也可以利用高级API来简化LSTM模型。高级API封装了前文介绍的所有配置细节。这段代码的运行速度要快得多,因为它使用编译好的运算符,而不是Python来处理之前讨论的许多细节。

```
    num_inputs = vocab_size
    lstm_layer = nn.LSTM(num_inputs, num_hiddens)
    model = d2l.RNNModel(lstm_layer, len(vocab))
    model = model.to(device)
    d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

长短期记忆网络是一种标准的隐变量自回归模型,它通过关键的状态控制来处理序列数据。多年来,出现了多种LSTM的衍生版本,包括多层结构、残差连接以及各种正则化技

术。尽管如此，由于需要处理序列数据的长距离依赖问题，训练 LSTM 和其他类似的序列模型（如 GRU）通常需要较高的计算成本。在后续章节中，我们将探讨一些先进的替代模型，如 Transformer，它们在处理序列数据方面提供了新的方法。

语言模型在自然语言处理领域扮演着基础角色，而机器翻译则被视为语言模型的最重要测试场景。机器翻译问题实质上是序列转换问题的核心，它涉及将输入序列转换为输出序列。

在神经网络用于端到端学习变得流行之前，统计方法在机器翻译领域占据主导地位。统计机器翻译依赖对翻译模型和语言模型等组件的统计分析，因此，基于神经网络的方法被特别称为神经机器翻译，以区分这两种不同的方法。

机器翻译任务涉及处理可变长度的输入和输出序列。为了解决这一问题，设计了一种包含编码器和解码器两个主要部分的架构。编码器负责接收一个可变长度的输入序列，并将其转换成一个固定形状的编码状态；解码器则将这个固定形状的编码状态转换为一个可变长度的输出序列。这被称为编码器-解码器（encoder-decoder）架构，如图 5-10 所示。

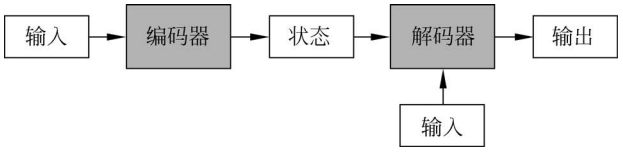

图 5-10　编码器-解码器架构

以英文到法文的机器翻译为例，考虑一个英文输入序列："They" "are" "watching" "."。在这个编码器-解码器架构中，首先将这个可变长度的输入序列编码成一个固定的状态，然后逐步解码这个状态，逐个生成翻译后的输出序列："Ils" "regardent" "."。

由于编码器-解码器架构是构建后续章节中各种序列转换模型的基础，我们将其设计为一个易于后续代码实现的接口。

编码器部分的接口定义如下：编码器接收一个可变长度的序列作为输入 X。任何继承自 encoder 基类的模型都需要完成相应的代码实现，以确保能够将输入序列编码为一个固定形状的状态，供解码器使用。

```
from torch import nn

#@save
class Encoder(nn.Module):     """编码器 - 解码器架构的基本编码器接口"""
    def __init__(self, * * kwargs):
        super(Encoder, self).__init__( * * kwargs)

    def forward(self, X, * args):
        raise NotImplementedError
```

解码器接口在设计时引入了一个 init_state 函数，这个函数的目的是将编码器的输出（enc_outputs）转换为解码器可以使用的编码状态。这个过程可能需要考虑额外的输入信息，如输入序列的有效长度，以确保解码器能够准确地处理变长的输入。

在解码过程中，解码器会在每个时间步接收两个输入：一个是上一个时间步生成的词元（作为当前时间步的输入），另一个是编码后的状态。解码器将这两个输入映射到当前时间步的输出词元，从而逐步构建出一个长度可变的输出序列。这个过程会一直持续，直到生

成一个结束符号或达到某个预设的最大长度为止。

```
#@save
class Decoder(nn.Module):    """编码器-解码器架构的基本解码器接口"""
    def __init__(self, **kwargs):
        super(Decoder, self).__init__(**kwargs)

    def init_state(self, enc_outputs, *args):
        raise NotImplementedError

    def forward(self, X, state):
        raise NotImplementedError
```

总而言之,编码器-解码器架构包含了一个编码器和一个解码器,并且还拥有可选的额外的参数。在前向传播中,编码器的输出用于生成编码状态,这个状态又被解码器作为其输入的一部分。

```
#@save
class EncoderDecoder(nn.Module):    """编码器-解码器架构的基类"""
    def __init__(self, encoder, decoder, **kwargs):
        super(EncoderDecoder, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, enc_X, dec_X, *args):
        enc_outputs = self.encoder(enc_X, *args)
        dec_state = self.decoder.init_state(enc_outputs, *args)
        return self.decoder(dec_X, dec_state)
```


在线视频

5.3 文本情感分析

随着社交网络和在线评论平台的快速兴起,众多用户生成的评论信息被捕捉并存档。这些丰富的评论信息中潜藏着巨大的价值,能够为决策提供关键的支持。情绪分析致力于挖掘文本中所反映的情绪倾向,包括用户对商品、文章或在线论坛帖子的看法。这一技术在政治决策、金融市场分析和营销策略等多个行业得到了广泛应用,帮助分析大众对于特定政策、市场动态或产品的看法及情绪反应。

情绪分析通过将文本划分为明确的情绪类别,如正面或负面,从而简化了复杂的文本信息。这一过程可以被理解为一种文本的分类挑战,它涉及将不同长度的文本信息归纳为有限的、预定义的情绪类别。在接下来的章节中,我们将使用斯坦福大学提供的广泛电影评论数据集作为情绪分析的案例研究。该数据集涵盖了 25000 条来自 IMDb 的电影观众评论,这些评论被均匀地分配到训练和测试两个子集中。在这两个子集中,正面评论和负面评论的数量持平,从而确保了情绪极性的均衡代表性。

首先准备数据集,下载并将数据集替换为读者实际的 aclImdb 数据集放置位置。

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

```
5.
6.  data_dir = 'Your_Path_to_aclImdb'
```

接下来，读取训练和测试数据集。每个样本都是一个评论及其标签：1 表示“积极”，0 表示“消极”。

```
7.  #@save
8.  def read_imdb(data_dir, is_train):
9.      """读取 IMDb 评论数据集文本序列和标签"""
10.         data, labels = [], []
11.         for label in ('pos', 'neg'):
12.             folder_name = os.path.join(data_dir, 'train' if is_train else 'test',
13.                                        label)
14.             for file in os.listdir(folder_name):
15.                 with open(os.path.join(folder_name, file), 'rb') as f:
16.                     review = f.read().decode('utf-8').replace('\n', '')
17.                     data.append(review)
18.                     labels.append(1 if label == 'pos' else 0)
19.         return data, labels
20.
21.     train_data = read_imdb(data_dir, is_train=True)
22.     print('训练集数目:', len(train_data[0]))
23.     for x, y in zip(train_data[0][:3], train_data[1][:3]):
24.         print('标签:', y, 'review:', x[0:60])
```

然后预处理数据集，将每个单词作为一个词元，过滤掉出现不到 5 次的单词，从训练数据集中创建一个词表。词元化后绘制评论词元长度的直方图。结果显示评论的长度各不相同。为了每次处理一小批量这样的评论，通过截断和填充将每个评论的长度设置为 500。

```
25.     train_tokens = d2l.tokenize(train_data[0], token='word')
26.     vocab = d2l.Vocab(train_tokens, min_freq=5, reserved_tokens=['<pad>'])
27.
28.     d2l.set_figsize()
29.     d2l.plt.xlabel('# tokens per review')
30.     d2l.plt.ylabel('count')
31.     d2l.plt.hist([len(line) for line in train_tokens], bins=range(0, 1000, 50));
32.
33.     num_steps = 500  # 序列长度
34.     train_features = torch.tensor([d2l.truncate_pad(
35.         vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
36.     print(train_features.shape)
```

之后就可以创建数据迭代器。在每次迭代中，都会返回一小批量样本。

```
37.     train_iter = d2l.load_array((train_features,
38.         torch.tensor(train_data[1])), 64)
39.
40.     for X, y in train_iter:
41.         print('X:', X.shape, ', y:', y.shape)
42.         break
43.     print('小批量数目:', len(train_iter))
```

最后整合代码获得 load_data_imdb 函数。它返回训练和测试数据迭代器以及 IMDb 评论数据集的词表。

```
44.     #@save
```

```
def load_data_imdb(batch_size, num_steps = 500):
    """返回数据迭代器和 IMDb 评论数据集的词表"""
    data_dir = 'Your_Path_to_aclImdb'
    train_data = read_imdb(data_dir, True)
    test_data = read_imdb(data_dir, False)
    train_tokens = d2l.tokenize(train_data[0], token = 'word')
    test_tokens = d2l.tokenize(test_data[0], token = 'word')
    vocab = d2l.Vocab(train_tokens, min_freq = 5)
    train_features = torch.tensor([d2l.truncate_pad(
       vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
    test_features = torch.tensor([d2l.truncate_pad(
       vocab[line], num_steps, vocab['<pad>']) for line in test_tokens])
    train_iter = d2l.load_array((train_features, torch.tensor(train_data[1])),
                                batch_size)
    test_iter = d2l.load_array((test_features, torch.tensor(test_data[1])),
                               batch_size,
                               is_train = False)
    return train_iter, test_iter, vocab
```

接下来首先使用循环神经网络完成情感分析任务。

与词相似度和类比任务一样,可以将预先训练的词向量应用于情感分析,如图 5-11 所示。由于 IMDb 评论数据集不是很大,使用在大规模语料库上预训练的文本表示可以减少模型的过拟合。作为图 5-11 中所示的具体示例,此处将使用预训练的 GloVe 模型来表示每个词元,并将这些词元表示送入多层双向循环神经网络以获得文本序列表示,该文本序列表示将被转换为情感分析输出。

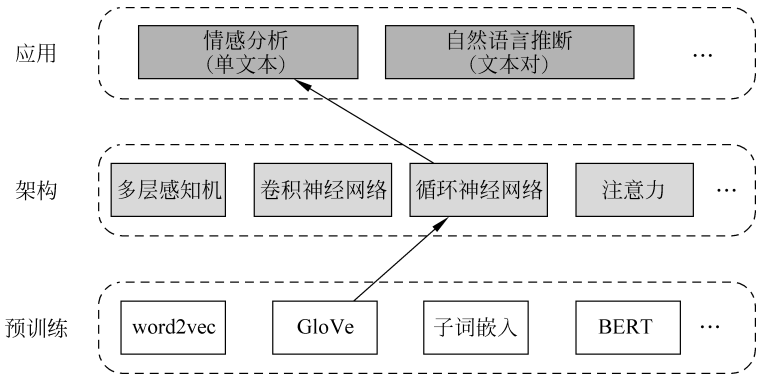

图 5-11　将 GloVe 送入基于循环神经网络的架构

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

batch_size = 32
train_iter, test_iter, vocab = load_data_imdb(batch_size)
```

在文本分类任务(如情感分析)中,可变长度的文本序列将被转换为固定长度的类别。在下面的 BiRNN 类中,虽然文本序列的每个词元经由嵌入层(self.embedding)获得其单独的预训练 GloVe 表示,但是整个序列由双向循环神经网络(self.encoder)编码。更具体地说,双向长短期记忆网络在初始和最终时间步的隐状态(在最后一层)被连接起来作为文本

序列的表示。然后，通过一个具有两个输出（“积极”和“消极”）的全连接层（self.decoder），将此单一文本表示转换为输出类别。

```
class BiRNN(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, **kwargs):
        super(BiRNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # 将 bidirectional 设置为 True 以获取双向循环神经网络
        self.encoder = nn.LSTM(embed_size, num_hiddens, num_layers=num_layers,
bidirectional=True)
        self.decoder = nn.Linear(4 * num_hiddens, 2)

    def forward(self, inputs):
        # inputs 的形状是(批量大小,时间步数)
        # 因为长短期记忆网络要求其输入的第一个维度是时间维
        # 所以在获得词元表示之前,输入会被转置
        # 输出形状为(时间步数,批量大小,词向量维度)
        embeddings = self.embedding(inputs.T)
        self.encoder.flatten_parameters()
        # 返回上一个隐藏层在不同时间步的隐状态,
        # outputs 的形状是(时间步数,批量大小,2*隐藏单元数)
        outputs, _ = self.encoder(embeddings)
        # 连接初始和最终时间步的隐状态,作为全连接层的输入,
        # 其形状为(批量大小,4*隐藏单元数)
        encoding = torch.cat((outputs[0], outputs[-1]), dim=1)
        outs = self.decoder(encoding)
        return outs
```

接着构造一个具有两个隐藏层的双向循环神经网络来表示单个文本以进行情感分析。

```
embed_size, num_hiddens, num_layers = 100, 100, 2
devices = d2l.try_all_gpus()
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
    if type(m) == nn.LSTM:
        for param in m._flat_weights_names:
            if "weight" in param:
                nn.init.xavier_uniform_(m._parameters[param])
net.apply(init_weights);
```

下面为词表中的单词加载预训练的 100 维（需要与 embed_size 一致）的 GloVe 嵌入。打印词表中所有词元向量的形状。使用这些预训练的词向量来表示评论中的词元，并且在训练期间不要更新这些向量。

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')

embeds = glove_embedding[vocab.idx_to_token]
embeds.shape

net.embedding.weight.data.copy_(embeds)
net.embedding.weight.requires_grad = False
```

现在可以训练双向循环神经网络进行情感分析。

```
lr, num_epochs = 0.01, 5
trainer = torch.optim.Adam(net.parameters(), lr = lr)
loss = nn.CrossEntropyLoss(reduction = "none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

定义以下函数来使用训练好的模型 net 预测文本序列的情感。

```
#@save
def predict_sentiment(net, vocab, sequence):
"""预测文本序列的情感"""
    sequence = torch.tensor(vocab[sequence.split()], device = d2l.try_gpu())
    label = torch.argmax(net(sequence.reshape(1, -1)), dim = 1)
    return 'positive' if label == 1 else 'negative'
```

最后用训练好的模型来对两个模型进行预测。

```
predict_sentiment(net, vocab, 'this movie is so great')
predict_sentiment(net, vocab, 'this movie is so bad')
```

除了使用循环神经网络,还可以使用卷积神经网络完成情感分析任务。之前探讨过使用二维卷积神经网络处理二维图像数据的机制,并将其应用于局部特征,如相邻像素。虽然卷积神经网络最初是为计算机视觉设计的,但它也被广泛用于自然语言处理。简单地说,只要将任何文本序列想象成一维图像即可。通过这种方式,一维卷积神经网络可以处理文本中的局部特征,如 n 元语法。

此处将使用 textCNN 模型来演示如何设计一个表示单个文本的卷积神经网络架构。与使用带有 GloVe 预训练的循环神经网络架构进行情感分析相比,唯一的区别在于架构的选择。

```
batch_size = 32
train_iter, test_iter, vocab = load_data_imdb(batch_size)
```

在介绍该模型之前,先看看一维卷积是如何工作的。请记住,这只是基于互相关运算的二维卷积的特例。如图 5-12 所示,在一维情况下,卷积窗口在输入张量上从左向右滑动。在滑动期间,卷积窗口中某个位置包含的输入子张量(例如,图 5-12 中的 0 和 1)和核张量(例如,图 5-12 中的 1 和 2)按元素相乘。这些乘法的总和在输出张量的相应位置给出单个标量值(例如,图 5-12 中的 $0\times1+1\times2=2$)。

图 5-12 一维卷积工作原理

此处在下面的 corr1d 函数中实现了一维互相关。给定输入张量 $\boldsymbol{X}$ 和核张量 $\boldsymbol{K}$,它返回输出张量 $\boldsymbol{Y}$。对于任何具有多个通道的一维输入,卷积核需要具有相同数量的输入通道。然后,对于每个通道,对输入的一维张量和卷积核的一维张量执行互相关运算,将所有通道上的结果相加以产生一维输出张量。可以实现多个输入通道的一维互相关运算。

```
def corr1d(X, K):
    w = K.shape[0]
    Y = torch.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
```

```
            Y[i] = (X[i: i + w] * K).sum()
        return Y

    def corr1d_multi_in(X, K):
        # 首先,遍历'X'和'K'的第 0 维(通道维)。然后,把它们加在一起
        return sum(corr1d(x, k) for x, k in zip(X, K))
```

类似地,可以使用汇聚层从序列表示中提取最大值,作为跨时间步的最重要特征。textCNN 中使用的最大时间汇聚层的工作原理类似于一维全局汇聚。对于每个通道在不同时间步存储值的多通道输入,每个通道的输出是该通道的最大值。请注意,最大时间汇聚允许在不同通道上使用不同数量的时间步。

使用一维卷积和最大时间汇聚,textCNN 模型将单个预训练的词元表示作为输入,然后获得并转换用于下游应用的序列表示。

对于具有由 d 维向量表示的 n 个词元的单个文本序列,输入张量的宽度、高度和通道数分别为 n、1 和 d。textCNN 模型将输入转换为输出,如下所示:

(1) 定义多个一维卷积核,并分别对输入执行卷积运算。具有不同宽度的卷积核可以捕获不同数目的相邻词元之间的局部特征。

(2) 在所有输出通道上执行最大时间汇聚层,然后将所有标量汇聚输出连接为向量。

(3) 使用全连接层将连接后的向量转换为输出类别。Dropout 可以用来减少过拟合。

据此定义模型,在下面的类中实现 textCNN 模型。与双向循环神经网络模型相比,除了用卷积层代替循环神经网络层外,还使用了两个嵌入层:一个是可训练权重,另一个是固定权重。

```
    class TextCNN(nn.Module):
        def __init__(self, vocab_size, embed_size, kernel_sizes, num_channels,
                     **kwargs):
            super(TextCNN, self).__init__(**kwargs)
            self.embedding = nn.Embedding(vocab_size, embed_size)
            # 这个嵌入层不需要训练
            self.constant_embedding = nn.Embedding(vocab_size, embed_size)
            self.dropout = nn.Dropout(0.5)
            self.decoder = nn.Linear(sum(num_channels), 2)
            # 最大时间汇聚层没有参数,因此可以共享此实例
            self.pool = nn.AdaptiveAvgPool1d(1)
            self.relu = nn.ReLU()
            # 创建多个一维卷积层
            self.convs = nn.ModuleList()
            for c, k in zip(num_channels, kernel_sizes):
                self.convs.append(nn.Conv1d(2 * embed_size, c, k))

        def forward(self, inputs):
            # 沿着向量维度将两个嵌入层连接起来
            # 每个嵌入层的输出形状都是(批量大小,词元数量,词元向量维度)连接起来
            embeddings = torch.cat((
                self.embedding(inputs), self.constant_embedding(inputs)), dim = 2)
            # 根据一维卷积层的输入格式,重新排列张量,以便通道作为第 2 维
            embeddings = embeddings.permute(0, 2, 1)
            # 每个一维卷积层在最大时间汇聚层合并后,获得的张量形状是(批量大小,通道数,1)
            # 删除最后一个维度并沿通道维度连接
```

```
        encoding = torch.cat([
            torch.squeeze(self.relu(self.pool(conv(embeddings))), dim = -1)
            for conv in self.convs], dim = 1)
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

接下来创建一个 textCNN 实例。它有 3 个卷积层,卷积核宽度分别为 3、4 和 5,均有 100 个输出通道。

```
embed_size, kernel_sizes, nums_channels = 100, [3, 4, 5], [100, 100, 100]
devices = d2l.try_all_gpus()
net = TextCNN(len(vocab), embed_size, kernel_sizes, nums_channels)
def init_weights(m):
    if type(m) in (nn.Linear, nn.Conv1d):
        nn.init.xavier_uniform_(m.weight)
net.apply(init_weights);
```

接下来加载预训练词向量。首先加载预训练的 100 维 GloVe 嵌入作为初始化的词元表示。这些词元表示(嵌入权重)在 embedding 中将被训练,在 constant_embedding 中将被固定。

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.requires_grad = False
```

之后可以训练 textCNN 模型进行情感分析。

```
lr, num_epochs = 0.001, 5
trainer = torch.optim.Adam(net.parameters(), lr = lr)
loss = nn.CrossEntropyLoss(reduction = "none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

同样地,可以使用训练好的模型对两个简单的句子进行预测。

```
predict_sentiment(net, vocab, 'this movie is so great')
predict_sentiment(net, vocab, 'this movie is so bad')
```