第 3 章　自然语言处理

自然语言处理（Natural Language Processing，NLP）以语言为对象，通过词法、句法和语义分析等手段，使得模型更深入地理解文本的结构和语义。这种深度理解能力是大模型成功的基石，能够使其在多种任务中取得优越性能，如问答系统、机器翻译、文本生成等。

本章介绍自然语言处理的基本概念、词嵌入、循环神经网络、长短期记忆网络、门控循环单元等基础内容，同时列举部分应用实例，希望读者能够应用自然语言处理技术解决实际问题，为更好地探讨大模型原理及技术打下基础。

3.1　自然语言处理概述

在现代社会，信息交流无时无刻不在发生。出于交流的基本需求，每天都有大量包含自然语言的数据产生，如新闻文章、电子邮件、学术论文、语音视频等。自然语言处理技术的出现使计算机能够处理和理解这些数据，为人类交流提供帮助和做出决策，为促进计算机与人类之间的无缝交流和合作做出重要贡献。

3.1.1　基本任务

语言是借助语义线索（如文字、符号或图像）传递信息和含义的一种直观行为。自然语言是指汉语、英语等人类日常使用的语言，有别于程序设计语言等人造语言。自然语言处理是一种机器学习技术，使得计算机能够解读、处理和理解人类语言，成为人类和计算机之间沟通的桥梁。自然语言处理的基本任务主要分为两大类（如图 3.1 所示）：**自然语言理解**（Natural Language Understanding，NLU）和**自然语言生成**（Natural Language Generation，NLG）。

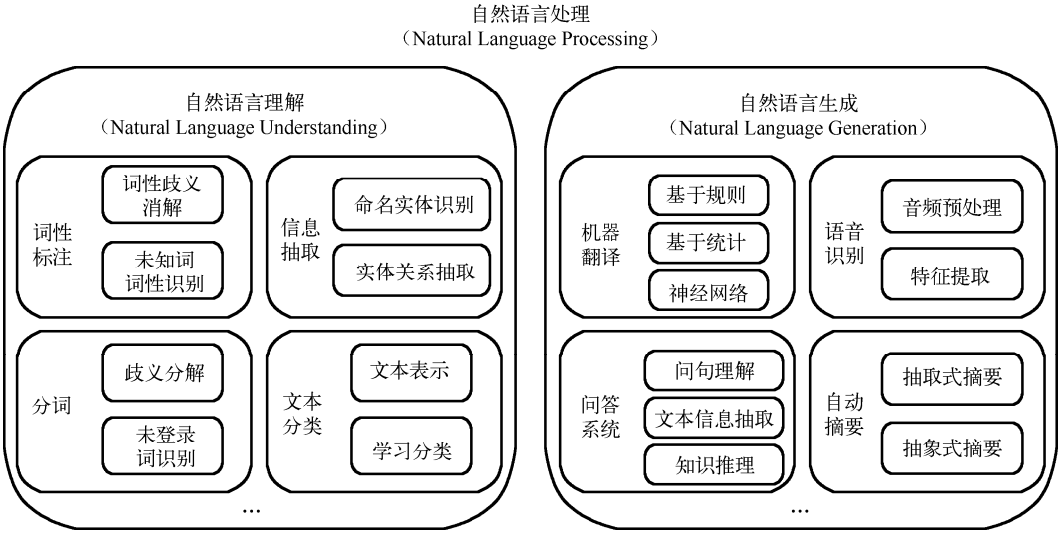


图 3.1　自然语言处理的基本任务

1. 自然语言理解

人与计算机交流的第一步就是让计算机理解人类输入给它的信息，比如当向手机中的语音助手询问“明天的天气怎么样？”时，它要能理解“明天”“天气”这两个重点内容。这类任务的研究目的是使计算机能够理解自然语言，从自然语言中提取有用的信息输出或用于下游的任务。自然语言理解类的任务包括但不限于以下几种。

- **词性标注：**指针对给定句子判断每个词的词性并加以标注的任务。在中文词性标注任务中，汉语缺乏词形态上的变化，因此不能从词的形态来识别词性。此外，汉语常用词兼类现象严重、覆盖范围广，这些都给中文词性标注研究带来了巨大挑战。
- **信息抽取：**指从自然语言文本中抽取出特定信息的任务，主要包括命名实体识别、实体关系抽取等子任务。命名实体识别是指从文本中识别出命名实体，如人名、地名、组织机构名等。实体关系抽取是指识别出文本实体中的目标关系。
- **分词：**旨在将句子、段落、文章这种长文本，分解为以字词为单位的数据结构，便于后续的处理工作。在中文文本处理中，分词指的是将连续的汉字序列切分成具有语义的词或词语的过程。相较于中文分词，英文分词相对简单，因为英文单词之间通常以空格分隔。
- **文本分类：**旨在将给定的文本数据分成不同的预定义类别。这些类别可以是任何类型，如新闻文章分类、电子邮件分类、评论分类等。该任务的主要目标是根据文本的内容和特征，自动将文本归类到正确的类别中。

常用于词性标注、命名实体识别、分词任务的模型有 BiLSTM-CRF 等。常用于文本分类任务的模型有文本卷积神经网络（TextCNN）、长短期记忆网络（LSTM）等。

2. 自然语言生成

计算机理解人类的输入后，我们还希望计算机能够生成满足人类目的、人类可以理解的自然语言形式的输出，从而实现真正的交流。就像询问语音助手“明天的天气怎么样？”时，它可以输出“明天阴转多云，气温零下六摄氏度到三摄氏度”，就像一个真人告诉你一样。所以，这类任务的侧重点在于生成，如从文本生成文本、从图片生成文本等。自然语言生成类的任务包括但不限于以下几种。

- **机器翻译：**指利用计算机和自然语言处理技术将一种自然语言的文本翻译成另一种自然语言的文本。机器翻译旨在解决不同语言之间的翻译问题，能够帮助人们跨越语言障碍进行交流和理解。
- **语音识别：**指利用计算机和自然语言处理技术将语音信号转换成文本的过程。语音识别技术可以帮助人们实现语音到文本的转换，从而方便人机交互，实现自动语音识别、语音搜索和智能语音助手等功能。
- **问答系统：**旨在使计算机能够理解自然语言问题，并从各种数据源中获取信息，给出准确的答案。其包含问句理解、文本信息提取、知识推理等重要步骤。该任务也被广泛应用于智能助手、搜索引擎、自动客服等领域。
- **自动摘要：**旨在从大量的文本中自动提取关键信息，生成简洁准确的摘要。其主要有抽取式摘要和抽象式摘要两种方法，前者直接从原文中抽取句子组成摘要，后者则

根据文本内容进行概括和重新表达。

常用于自然语言生成类的任务的模型有 Seq2Seq 模型、GPT（Generative Pretrained Transformer）、T5（Text-to-Text Transfer Transformer）等。

3.1.2 发展历程

自然语言处理技术的发展历程从 20 世纪 50 年代开始，经过了多个阶段，并不断地迭代发展，如今已经成为信息技术领域中的重要一环，如图 3.2 所示。20 世纪 50 年代—70 年代自然语言处理主要采用基于规则的方法。这种方法依赖于语言学家和开发者预先定义的规则系统，以便解析和理解语言。举一个过滤垃圾电子邮件的例子，为了过滤垃圾邮件需要制定一些规则：如果邮件正文中出现了“转账”“中奖”等词汇，就标记为垃圾邮件，诸如此类。很明显，这样的方法需要精心设计并且要不断维护以适应语言的变化和复杂性。此外，由于语言的多样性和歧义性，制定全面准确的规则集非常具有挑战性。

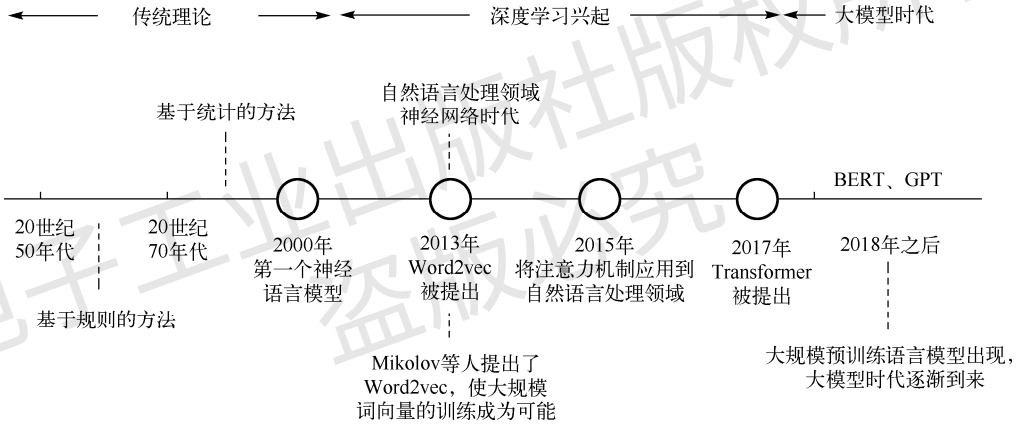


图 3.2 自然语言处理技术的发展历程

20 世纪70 年代以后，随着互联网的高速发展，丰富的语料库成为现实，硬件不断更新完善，基于统计的方法逐渐代替了基于规则的方法。这种方法通常依靠大量的语言数据来学习，得到数据中词、短语、句子的概率分布，从而实现对语言的处理和分析。

2000 年，第一个神经语言模型由 Bengio 等人提出，第一次用神经网络来解决语言模型的问题。这个模型将某词语之前出现的 n 个词语作为输入，预测下一个单词输出。模型一共有三层，第一层是映射层，将 n 个单词映射为对应的词嵌入；第二层是隐藏层；第三层是输出层，使用 Softmax 函数输出单词的概率分布，是一个多分类器。

2013 年，Mikolov 等人提出了 Word2vec，使大规模词向量的训练成为可能，读者可以在 3.2.2 节学习到更详细的 Word2vec 的相关知识。同时，自然语言处理领域的神经网络时代也逐渐开始，循环神经网络、卷积神经网络开始被广泛应用于自然语言处理领域。

2015 年，Bahdanau 等人使用注意力机制在机器翻译任务上将翻译和对齐同时进行，首次将注意力机制应用到自然语言处理领域。

2017 年，Transformer 被提出，可以说它是自然语言处理的颠覆者，它创造性地用非序列模型来处理序列化的数据，并且大获成功。2018 年之后，BERT、GPT 这两类大规模预训练

语言模型也随之出现，大模型时代逐渐到来。本书第 4 章将详细介绍 Transformer、BERT、GPT 的相关知识，读者可以继续学习。

3.1.3　应用领域

自然语言处理的应用非常广泛，包括翻译软件、聊天机器人、语音助手、搜索引擎等。这些应用领域均涉及自然语言生成和自然语言理解，但侧重点不同。例如，翻译软件和聊天机器人更侧重于自然语言生成，而搜索引擎则更侧重于自然语言理解。

1．翻译软件

翻译软件是一类将一种语言翻译成另一种语言的应用软件，是日常生活中最常见的自然语言处理应用之一。它帮助用户更快速、高效地完成翻译任务，被广泛应用于全球化企业的业务沟通、跨文化交流等场景。常见的翻译软件有谷歌翻译、百度翻译、DeepL 翻译等。

2．聊天机器人

聊天机器人是能与人类进行自然语言交互的智能机器人。它可以理解人类的输入，并基于事先设计好的对话规则、知识库和机器学习模型来生成回复。聊天机器人可以应用于多个领域，如在线客服等。常见的聊天机器人有微软的小冰、阿里巴巴的天猫精灵等。

3．语音助手

语音助手通常集成在智能手机、智能音箱、车载娱乐系统等设备上。它可以通过语音交互的方式帮助用户实现各种任务，如发送短信、播放音乐、查询天气、调节家居设备等。语音助手的发展靠近人类语言的自然交流方式，使得人们更加便捷地与智能设备进行交互。常见的语音助手有 Apple 的 Siri、小米的小爱同学等。

4．搜索引擎

搜索引擎是帮助用户在互联网上查找和获取信息的工具。搜索引擎利用自然语言处理技术对互联网上的文本信息进行索引和搜索，使得用户能够通过关键词搜索获得相关的网页、新闻、图片、视频等资源。搜索引擎通常使用自然语言处理技术完成查询解析、信息检索和排序等关键任务，从而提供更准确和丰富的搜索结果。一些著名的搜索引擎如谷歌、百度、必应等在全球范围内被广泛使用。

在自然语言处理的实际项目中，数据是必不可少的东西，在数据的支持下才能完成各种自然语言处理任务，本书 5.1.5 节介绍了部分常用的开源文本数据集。在文本数据被转换为数值形式之前，通常需要进行一定的预处理，包括低质去除、冗余去除、隐私去除及词元划分等步骤，具体内容请见本书 5.1.2 节数据预处理的文本数据预处理部分，这里不再赘述。

在理解了自然语言处理的概念后，接下来需要思考怎么样才能让计算机“懂得”自然语言，也就是如何处理文本数据才能输入后续算法。众所周知，计算机无法直接读懂非数值的自然语言，只有将其转化为数值形式才能被计算机处理。接下来的小节将介绍词嵌入，即将词语映射为数值的一种方式，是自然语言处理领域下游任务实现的重要基础。

使用词嵌入技术获得了词语的数值表示形式之后，再将其输入后续的网络模型中（如循环神经网络、Transformer 等）完成下游任务。图 3.3 展现了自然语言处理的一般流程。

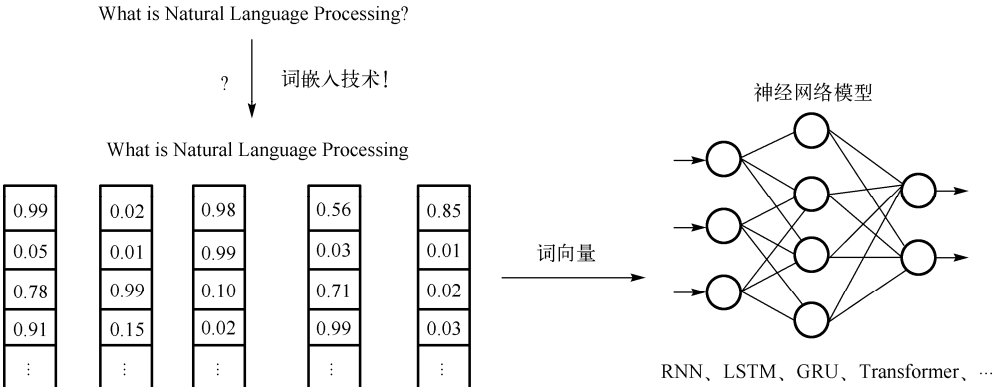


图 3.3 自然语言处理的一般流程

3.2 词 嵌 入

词嵌入（Word Embedding）可以认为是自然语言处理任务的第一步。它将输入的自然语言文本转换为模型可以处理的数值形式，即词向量，也可以认为是单词的特征向量。之后，这些向量就可以作为后续任务中神经网络模型的输入，如图 3.3 所示。但词向量并不能只是冰冷的数字，它还要包含一定的信息，使具有相似语义或语法特征的词汇在向量空间中彼此接近。词嵌入的过程就是将文本转化为有意义的数值表示形式的过程。

在最初自然语言处理任务中，文本数据转换成可供计算机识别的数据形式使用的是独热向量。它将文本转化为非常简单的向量表示形式，并且不局限于语言种类，是最简单的词嵌入的方式。

3.2.1 独热向量

独热向量是指使用 N 位 0 或 1 对 N 个单词进行编码，每个状态都有独立的表示形式。其分量和类别数一样多，类别对应的分量设置为 1（one-hot），其余分量设置为 0。例如，要编码 apple、bag、cat、dog、elephant 5 个单词，若用 5 位向量进行编码，则

$$\text{apple} = [1\ 0\ 0\ 0\ 0]$$

$$\text{bag} = [0\ 1\ 0\ 0\ 0]$$

$$\text{cat} = [0\ 0\ 1\ 0\ 0]$$

$$\text{dog} = [0\ 0\ 0\ 1\ 0]$$

$$\text{elephant} = [0\ 0\ 0\ 0\ 1]$$

具体而言，假设词典中不同词的数量为 N，每个词对应一个从 0 到 N 的不同整数（索引）。对于索引为 i 的任意词的独热向量，创建一个全为 0 的长度为 N 的向量，并将位置 i 处的元素设置为 1。这样，每个词都被表示为一个长度为 N 的向量，可以供后续神经网络直接使用。

独热向量容易构建，能够满足对各种内容进行编码的要求。但是这种表示方式存在明显的缺陷，即不能体现出编码内容之间的关联，在上述例子中，cat、dog 和elephant 都属于动物类，

而 apple 与这三个词属性关联度较低，但上述编码不能体现出这些类别关系。通常，可以用余弦相似度来准确地衡量向量之间的相似性，对于向量 $\boldsymbol{x},\boldsymbol{y}\in\mathbb{R}^d$，它们的余弦相似度是它们之间角度的余弦：

$$\cos(\boldsymbol{x},\boldsymbol{y})=\frac{\boldsymbol{x}^{\mathrm{T}}\boldsymbol{y}}{\|\boldsymbol{x}\|\|\boldsymbol{y}\|}\in[-1,1] \tag{3.1}$$

由于任意两个不同词的独热向量之间的余弦相似度为 0，所以独热向量不能编码词之间的相似性。此外，使用独热向量形成的特征矩阵非常稀疏，占用空间很大。可见，使用独热向量编码单词并不是一种很好的选择。因此，需要找到一种更好的表示方法，这种方法需要满足以下两点要求。

- 携带上下文信息，即词与词之间的联系能在词的向量表示中体现。
- 词的表示是稠密的，能用更小的空间、更低的维数表示更多的信息。

Word2vec 技术就满足了这两点要求，使词向量从高维且稀疏的表示变为低维且密集的表示，同时包含词的语义信息和词与词之间的相似性等信息。

3.2.2　Word2vec

Word2vec 是一种词嵌入技术，也可以看作一种神经网络模型，它通过对上下文中的词进行预测的方式来学习词向量。训练后的 Word2vec 将每个词映射到一个固定长度、低维度的词向量，这些向量在训练后会包含单词的语义信息，同时能更好地表达不同词之间的相似性和类比关系，图 3.4 所示为降维后的词向量表示，可以看到相似概念的词是聚集在一起的。这很好地解决了独热向量带来的问题。

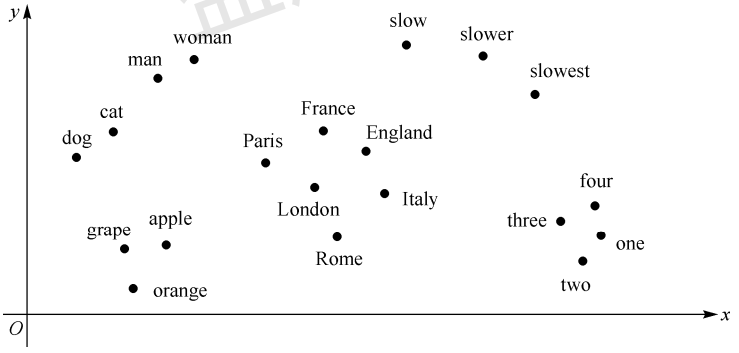


图 3.4　降维后的词向量表示

此外，训练后的词向量还可以更好地捕捉许多语言规律。例如，向量运算 vector（巴黎）–vector（法国）+vector（意大利）产生的向量非常接近 vector（罗马），并且 vector（king）–vector（man）+vector（woman）接近 vector（queen）等。和独热向量相比，Word2vec 生成的词向量具有以下优点。

- 训练时利用上下文信息，词向量包含词的语义信息和词与词之间的联系。
- 维度更少，所以占用空间更小、计算成本更低。
- 通用性强，可用于各种下游自然语言处理任务。

训练 Word2vec 的常用方法有两种：跳元（Skip-gram）模型和连续词袋（Continuous Bags

Of Words，CBOW）模型。它们的区别在于，跳元模型根据中心词预测上下文词，连续词袋模型根据上下文词预测中心词，两种方法的示意图如图 3.5 所示。下面将分别介绍这两种方法。

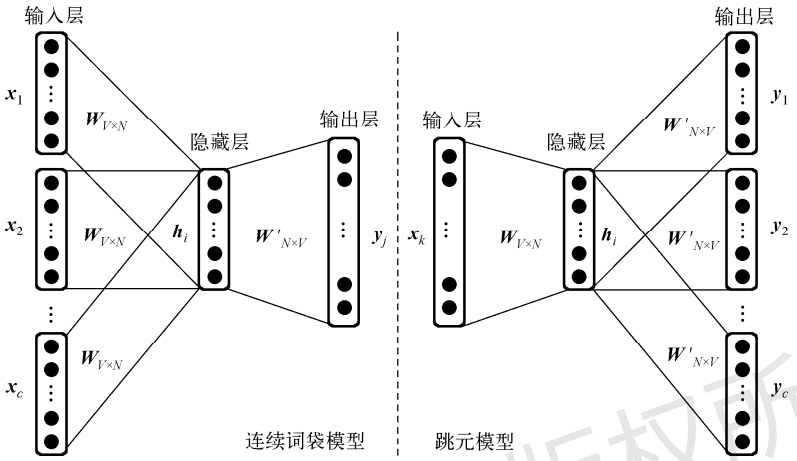


图 3.5 连续词袋模型和跳元模型示意图

1．跳元模型

跳元模型通过一个词来生成文本序列中该词周围的单词。以文本序列“the woman loves her daughter”为例，假设中心词选择“loves”，并将上下文窗口设置为 2，跳元模型生成与中心词距离不超过 2 个词的上下文词“the”“woman”“her”“daughter”的条件概率可由如下公式表示：

$$P(\text{"the","woman","her","daughter"}|\text{"loves"}) \tag{3.2}$$

由于上下文词是在给定中心词的情况下独立生成的，上述条件概率可以依照独立性改写为

$$P(\text{"the"}|\text{"loves"})\cdot P(\text{"woman"}|\text{"loves"})\cdot(\text{"her"}|\text{"loves"})\cdot P(\text{"daughter"}|\text{"loves"}) \tag{3.3}$$

在跳元模型中，对于索引为 i 的词 w_i 都有两个 d 维向量：$\boldsymbol{v}_i\in\mathbb{R}^d$、$\boldsymbol{u}_i\in\mathbb{R}^d$ 分别表示该词为中心词和上下文词时的向量表示。假设中心词 w_c 在词典中的索引为 c，上下文词 w_o 在词典中的索引为 o，则生成上下文词的条件概率可以通过对向量点积后经过 Softmax 运算得到。

$$P(w_o|\ w_c)=\frac{\exp(\boldsymbol{u}_o^{\mathrm{T}}\boldsymbol{v}_c)}{\sum_{i\in\mathcal{V}}\exp(\boldsymbol{u}_i^{\mathrm{T}}\boldsymbol{v}_c)} \tag{3.4}$$

其中，$\mathcal{V}=0,1,2,\cdots,|\mathcal{V}|-1$ 为索引集。给定长度为 T 的文本序列，时刻 t 处的词表示为 $w^{(t)}$。假设上下文词是在给定任何中心词的情况下独立生成的，对于上下文窗口 m，跳元模型的似然函数是在给定任何中心词的情况下生成所有上下文词的概率，即

$$\prod_{t=1}^{T}\prod_{-m\leqslant j\leqslant m,j\neq 0}P(w^{(t+j)}|\ w^{(t)}) \tag{3.5}$$

训练跳元模型时，通过最大化该似然函数来学习模型参数，即最小化以下损失函数：

$$L=-\sum_{t=1}^{T}\sum_{-m\leqslant j\leqslant m,j\neq 0}P(w^{(t+j)}|\ w^{(t)}) \tag{3.6}$$

2．连续词袋模型

连续词袋模型根据文本序列中的上下文词推理得到中心词。以文本序列“the woman loves her daughter”为例，假设中心词选“loves”且上下文窗口为 2，连续词袋模型基于上下文词“the”“woman”“her”“daughter”生成中心词“loves”的条件概率可以表示为

$$P(\text{"loves"}\mid\text{"the"},\text{"woman"},\text{"her"},\text{"daughter"}) \tag{3.7}$$

由于连续词袋模型中存在多个上下文词，因此在计算条件概率时需要对这些上下文词向量进行平均。具体地说，对于字典中索引为 i 的任意词，分别用 $\boldsymbol{v}_i\in\mathbb{R}^d$、$\boldsymbol{u}_i\in\mathbb{R}^d$ 表示上下文词和中心词的两个向量。给定上下文词 $w_{o_1},w_{o_2},\cdots,w_{o_{2m}}$（在词表中索引是 $o_1,o_2,\cdots,o_{2m}$）生成任意中心词 w_c（在词表中索引是 c）的条件概率可以由以下公式建模：

$$P(w_c|\ w_{o_1},w_{o_2},\cdots,w_{o_{2m}})=\frac{\exp\left(\frac{1}{2m}\boldsymbol{u}_c^{\mathrm{T}}(\boldsymbol{v}_{o_1}+\boldsymbol{v}_{o_2}+\cdots+\boldsymbol{v}_{o_{2m}})\right)}{\sum_{i\in\mathcal{V}}\exp\left(\frac{1}{2m}\boldsymbol{u}_i^{\mathrm{T}}(\boldsymbol{v}_{o_1}+\boldsymbol{v}_{o_2}+\cdots+\boldsymbol{v}_{o_{2m}})\right)} \tag{3.8}$$

为了简洁，这里设为 $\mathcal{W}_o=w_{o_1},w_{o_2},\cdots,w_{o_{2m}}$ 和 $\bar{\boldsymbol{v}}_o=(\boldsymbol{v}_{o_1}+\boldsymbol{v}_{o_2}+\cdots+\boldsymbol{v}_{o_{2m}})/(2m)$，则上述公式可以简化为

$$P(w_c|\ \mathcal{W}_o)=\frac{\exp(\boldsymbol{u}_c^{\mathrm{T}}\bar{\boldsymbol{v}}_o)}{\sum_{i\in\mathcal{V}}\exp(\boldsymbol{u}_i^{\mathrm{T}}\bar{\boldsymbol{v}}_o)} \tag{3.9}$$

给定长度为 T 的文本序列，其中时刻 t 处的词表示为 $w^{(t)}$。对于上下文窗口 m，连续词袋模型的似然函数是在给定其上下文词的情况下生成所有中心词的概率：

$$\prod_{t=1}^{T}P(w^{(t)}|\ w^{(t-m)},\cdots,w^{(t-1)},w^{(t+1)},\cdots,w^{(t+m)}) \tag{3.10}$$

同样，训练连续词袋模型时也通过最大化该似然函数来学习模型参数，即最小化以下损失函数：

$$L=-\sum_{t=1}^{T}P(w^{(t)}|\ w^{(t-m)},\cdots,w^{(t-1)},w^{(t+1)},\cdots,w^{(t+m)}) \tag{3.11}$$

3．连续词袋模型工作流程举例

为了更清楚地理解两个模型的工作方式，这里介绍一个具体的例子来展示连续词袋模型的工作流程。仍以文本序列“the woman loves her daughter”为例，假设选择“loves”为中心词，上下文窗口设为 2，则此时的目标就是根据单词“the”“woman”“her”“daughter”来预测单词“loves”。

使用单词的独热编码作为输入：“the”=[1　0　0　0　0]、“woman”=[0　1　0　0　0]、

“loves”=[00 1 0 0]、“her”=[0 0 0 1 0]、“daughter”=[0 0 0 0 1]。图 3.5 所示的权重矩阵 $\boldsymbol{W}_{V\times N}$ 是模型通过训练得到的，假设该矩阵为

$$\begin{bmatrix} 1 & 2 & 3 & 0 & 1 \\ 1 & 2 & 1 & 2 & 2 \\ -1 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{3.12}$$

其中，N 表示单词数量，也是输入层单词的维数，在本例中为 5；V 表示最终希望得到的词向量维数。现在，输入单词“the”，即将单词的独热编码与权重矩阵相乘：

$$\begin{bmatrix} 1 & 2 & 3 & 0 & 1 \\ 1 & 2 & 1 & 2 & 2 \\ -1 & 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \tag{3.13}$$

得到的为单词“the”的词向量，同理可以得到每个单词的词向量为

$$\text{woman} = \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix} \quad \text{her} = \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix} \quad \text{daughter} = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} \tag{3.14}$$

将得到的 4 个向量相加求平均作为输出层的输入：

$$\frac{1}{4}\left(\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} 1.00 \\ 1.75 \\ 0.25 \end{bmatrix} \tag{3.15}$$

将该向量与权重矩阵 $\boldsymbol{W}'_{N\times V}$ 相乘得到输出向量（该权重矩阵也是网络训练的结果）。最后，将 Softmax 函数作用在输出向量上，得到每个词的概率分布，自此模型完成预测中心词任务。

$$\text{Softmax}\left(\begin{bmatrix} 1 & 2 & -1 \\ -1 & 2 & -1 \\ 1 & 2 & 2 \\ 0 & 2 & 0 \\ 1 & -1 & 2 \end{bmatrix} \begin{bmatrix} 1.00 \\ 1.75 \\ 0.25 \end{bmatrix} \right) = \text{Softmax}\left(\begin{bmatrix} 4.250 \\ 2.250 \\ 5.000 \\ 3.000 \\ -0.250 \end{bmatrix} \right) = \begin{bmatrix} 0.268 \\ 0.036 \\ 0.567 \\ 0.126 \\ 0.003 \end{bmatrix} \tag{3.16}$$

当然，还要通过损失函数计算该输出概率分布与预测单词“loves”的独热向量之间的损失，反向传播，更新网络参数。

需要注意的是，预测目标单词是训练网络的方式，获得网络的中间产物——权重矩阵 $\boldsymbol{W}_{V\times N}$ 才是期望得到的，因为任意一个单词的独热向量乘以该矩阵就能得到自己的词向量。

3.2.3 代码示例

这里将展示一段使用 Gensim 库来完成词向量操作的代码示例。Gensim 是一个用于对大规模文本语料进行语义建模的 Python 库。它专注于主题建模、文档相似度分析和词向量表示等自然语言处理任务。

1. 准备输入

Gensim 的 Word2vec 需要一系列句子作为输入，每个句子有一个单词列表（经过分词处理）。

```
# 导入模块并设置日志记录
import gensim,logging
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)
 s',level=logging.INFO)

sentences = [['first','sentence'],['second','sentence']]
```

2. 创建和训练模型

Word2vec 通过以下方式进行训练，其中有几个影响训练速度和质量的参数。size 表示 Word2vec 将词汇映射到的 *N* 维空间的维数，虽然更大的 size 值需要更多的训练数据，但可以产生更好的模型。workers 用于训练并行化，以加快训练速度。iter 是模型训练时在整个训练语料库上的迭代次数。sg 是模型训练所采用的算法类型，1 代表 skip-gram，0 代表 CBOW。

```
model = gensim.models.Word2Vec(sentences,min_count=3,size=50,
 workers=7,iter=20,sg=1,window=8)

# 以另一种方式训练模型
new_model = gensim.models.Word2Vec(iter=1)  # 一个还没有训练的空模型
new_model.build_vocab(sentences) # 遍历一次语句生成器
new_model.train(sentences)
```

3. 模型使用

Word2vec 支持一些具体的应用，包括单词相似性任务、找出与其他词差异最大的词汇等。

```
model.most_similar(positive=['woman','king'],negative=['man'],topn=1)
[('queen',0.50882536)]

model.doesnt_match("breakfast cereal dinner lunch";.split())
'cereal'

model.similarity('woman','man')
0.73723527

model.wv.doesnt_match(" 舆情 互联网 媒体 商业 场景 咨询 ".split())
'媒体'
```

然而，通过词嵌入得到的词向量仅仅包含最基础的信息，如词的语义信息、词与词之间的相似关系等，这显然不能直接应用于下游复杂任务。因此，词嵌入主要应用在数据的初步处理上，对于数据的进一步分析往往需要通过后续特定的模型来实现，常用的模型有循环神经网络及其两个变体（长短期记忆网络和门控循环单元）、Transformer 等。本章接下来会继续介绍循环神经网络、长短期记忆网络和门控循环单元的相关内容。第 4 章将介绍 Transformer 的相关内容。

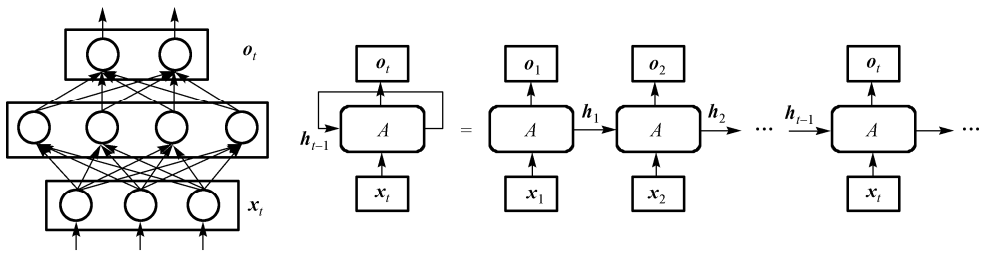
3.3 循环神经网络

时序是自然语言中十分重要的元素，是各类自然语言处理任务中都要获取的信息，简单的词序混乱就可以使整个句子不通顺。就像下面这句话我们并不能很好地理解：“working love learning we on deep”，正确的语序应该是“we love working on deep learning”。而传统的神经网络，如卷积神经网络，无法处理含时序信息的输入，从而导致其在自然语言处理任务上表现不佳。为了解决这类序列问题，循环神经网络被提出。循环神经网络（Recurrent Neural Network，RNN）是一类以序列数据为输入，在序列的演进方向进行递归且所有节点（循环单元）按链式连接的递归神经网络。就像卷积神经网络常用于处理图片等数据，循环神经网络主要用于处理序列数据，能够捕捉序列中的时序信息，因此它在自然语言处理领域被广泛应用。

3.3.1 循环神经网络介绍

回忆第 2 章介绍的全连接神经网络结构：它有若干输入神经元，其个数对应于输入数据或特征的维数；有若干隐藏层，每个隐藏层包含多个神经元；输出层有若干输出神经元，其个数对应于输出数据或特征的维数。对比图 3.6 中循环神经网络的结构，读者可能会产生疑问：为什么循环神经网络的输入层、隐藏层和输出层好像都只有一个神经元？实际上，如果忽略连接 A 的边，这里的输入 $\boldsymbol{x}_t$ 是一个词向量，对应的就是全连接神经网络中包含若干输入神经元的输入层，结构 A 就是隐藏层，如图 3.6（a）所示。

了解循环神经网络的主体后，下面介绍其核心要素，即隐藏层 A 的输入除了来自输入层 $\boldsymbol{x}_t$，还来自上一时刻的隐藏状态 $\boldsymbol{h}_{t-1}$。在每个时刻，循环神经网络的模块 A 在读取了输入 $\boldsymbol{x}_t$ 和 $\boldsymbol{h}_{t-1}$ 之后会生成新的隐藏状态 $\boldsymbol{h}_t$，并产生当前时刻的输出 $\boldsymbol{o}_t$。完整的序列可以用图 3.6（b）表示。



（a） （b）

图 3.6 循环神经网络

当前时刻的隐藏状态 $\boldsymbol{h}_t$ 由上一时刻的隐藏状态 $\boldsymbol{h}_{t-1}$ 和当前时刻的输入 $\boldsymbol{x}_t$ 共同计算得出，因此它捕获并保留了序列直到当前时刻的历史信息，如当前时刻神经网络的状态和记忆。具体地，可由如下公式表示：

$$\boldsymbol{h}_t = \phi(\boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{W}_h \boldsymbol{h}_{t-1} + \boldsymbol{b}_h) \tag{3.17}$$

当前时刻的输出可由如下公式计算，类似于线性层：

$$\boldsymbol{o}_t = \boldsymbol{W}_o \boldsymbol{h}_t + \boldsymbol{b}_o \tag{3.18}$$

图 3.7 展现了以“working love learning we on deep”为输入时循环神经网络的工作流程，清晰地展现出了隐藏状态 $\boldsymbol{h}_t$ 是如何由上一时刻的隐藏状态 $\boldsymbol{h}_{t-1}$ 和当前时刻的输入 $\boldsymbol{x}_t$ 共同得出的。

循环神经网络的参数包括隐藏层的权重 $\boldsymbol{W}_x$、$\boldsymbol{W}_h$ 和偏置 $\boldsymbol{b}_h$，以及输出层的权重 $\boldsymbol{W}_o$ 和偏置 $\boldsymbol{b}_o$。注意，隐藏层 A 输出的新的隐藏状态又重新输入 A 中。因此，在不同时刻使用的都是同样的权重，其参数开销不会随着时间步的增加而增加。

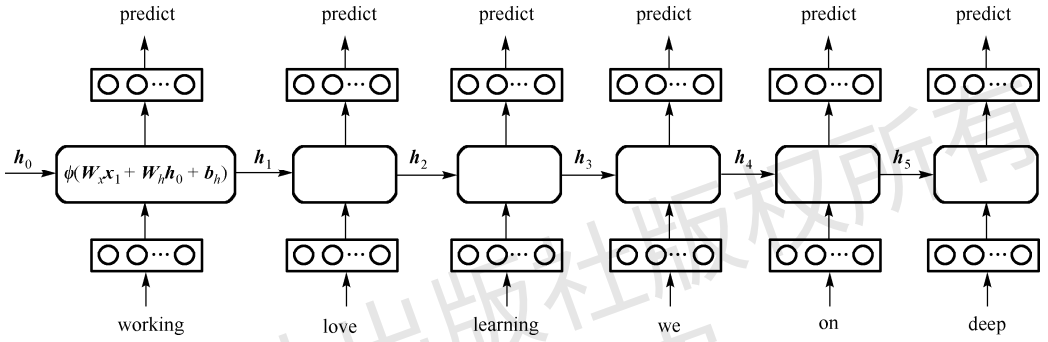


图 3.7　循环神经网络的工作流程举例

3.3.2　循环神经网络训练

循环神经网络也需要反向传播误差来调整自己的参数。循环神经网络在 BP 算法的基础上加入了时间序列，使用随时间反向传播的链式求导算法来反向传播误差，称为 BPTT（Back Propagation Through Time）算法。下面将介绍 BPTT 算法。

首先，根据 3.3.1 节的介绍，$\boldsymbol{h}_t$ 和 $\hat{\boldsymbol{y}}_t$ 分别为当前时刻的隐藏状态和当前时刻的输出：

$$\boldsymbol{h}_t = \phi(\boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{W}_h \boldsymbol{h}_{t-1}) \tag{3.19}$$

$$\hat{\boldsymbol{y}}_t = \text{Softmax}(\boldsymbol{W}_o \boldsymbol{h}_t) \tag{3.20}$$

若使用交叉熵来计算每个时刻的损失，则总误差可由各时刻交叉熵损失之和表示：

$$E_t(\boldsymbol{y}_t, \hat{\boldsymbol{y}}_t) = -\boldsymbol{y}_t \log \hat{\boldsymbol{y}}_t \tag{3.21}$$

$$E(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \sum E_t(\boldsymbol{y}_t, \hat{\boldsymbol{y}}_t) = -\sum \boldsymbol{y}_t \log \hat{\boldsymbol{y}}_t \tag{3.22}$$

根据以上定义，得到如图 3.8 所示的简化示意图。

以 $t=3$ 时刻的损失 E_3 为例，首先计算参数 $\boldsymbol{W}_o$ 的梯度：

$$\frac{\partial E_3}{\partial \boldsymbol{W}_o} = \frac{\partial E_3}{\partial \hat{\boldsymbol{y}}_3} \frac{\partial \hat{\boldsymbol{y}}_3}{\partial \boldsymbol{W}_o} \tag{3.23}$$

然后，根据图 3.9 所示的计算图，计算参数 $\boldsymbol{W}_h$ 和参数 $\boldsymbol{W}_x$ 的梯度：

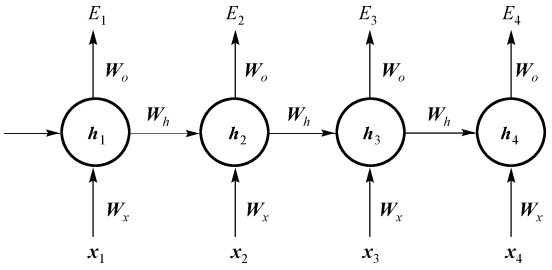


图 3.8　循环神经网络简化示意图

$$\frac{\partial E_3}{\partial \boldsymbol{W}_h}=\frac{\partial E_3}{\partial \hat{\boldsymbol{y}}_3}\frac{\partial \hat{\boldsymbol{y}}_3}{\partial \boldsymbol{h}_3}\frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{W}_h}+\frac{\partial E_3}{\partial \hat{\boldsymbol{y}}_3}\frac{\partial \hat{\boldsymbol{y}}_3}{\partial \boldsymbol{h}_3}\frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{h}_2}\frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{W}_h}+\frac{\partial E_3}{\partial \hat{\boldsymbol{y}}_3}\frac{\partial \hat{\boldsymbol{y}}_3}{\partial \boldsymbol{h}_3}\frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{h}_2}\frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{h}_1}\frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{W}_h} \tag{3.24}$$

$$\frac{\partial E_3}{\partial \boldsymbol{W}_x}=\frac{\partial E_3}{\partial \hat{\boldsymbol{y}}_3}\frac{\partial \hat{\boldsymbol{y}}_3}{\partial \boldsymbol{h}_3}\frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{W}_x}+\frac{\partial E_3}{\partial \hat{\boldsymbol{y}}_3}\frac{\partial \hat{\boldsymbol{y}}_3}{\partial \boldsymbol{h}_3}\frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{h}_2}\frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{W}_x}+\frac{\partial E_3}{\partial \hat{\boldsymbol{y}}_3}\frac{\partial \hat{\boldsymbol{y}}_3}{\partial \boldsymbol{h}_3}\frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{h}_2}\frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{h}_1}\frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{W}_x} \tag{3.25}$$

还可以将参数梯度表达式简化为以下形式：

$$\frac{\partial E_3}{\partial \boldsymbol{W}_h}=\sum_{k=0}^{3}\frac{\partial E_3}{\partial \hat{\boldsymbol{y}}_3}\frac{\partial \hat{\boldsymbol{y}}_3}{\partial \boldsymbol{h}_3}\left(\prod_{j=k+1}^{3}\frac{\partial \boldsymbol{h}_j}{\partial \boldsymbol{h}_{j-1}}\right)\frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{W}_h} \tag{3.26}$$

$$\frac{\partial E_3}{\partial \boldsymbol{W}_x}=\sum_{k=0}^{3}\frac{\partial E_3}{\partial \hat{\boldsymbol{y}}_3}\frac{\partial \hat{\boldsymbol{y}}_3}{\partial \boldsymbol{h}_3}\left(\prod_{j=k+1}^{3}\frac{\partial \boldsymbol{h}_j}{\partial \boldsymbol{h}_{j-1}}\right)\frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{W}_x} \tag{3.27}$$

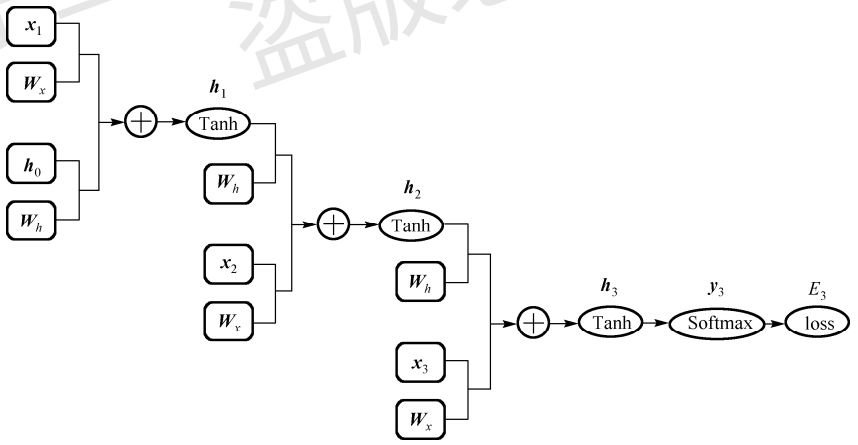


图 3.9　循环神经网络计算图

3.3.3　循环神经网络梯度问题

梯度消失和梯度爆炸是神经网络在训练过程中常出现的问题，梯度消失和梯度爆炸产生的原因主要有两种：深层网络的结构和不合适的损失函数，但本质上都是因为梯度反向传播中的连乘效应。因此，循环神经网络在训练时也会出现这两种问题。下面将简单介绍循环神经网络训练过程中的梯度消失和梯度爆炸问题。

1. 梯度消失问题

观察 3.3.2 节中 E_3 对参数 $\boldsymbol{W}_h$ 和参数 $\boldsymbol{W}_x$ 的偏导数，会发现其中均存在连乘情况，当激活函数为 Tanh 函数时，连乘部分可以表示为如下公式：

$$\prod_{j=k+1}^{3}\frac{\partial \boldsymbol{h}_j}{\partial \boldsymbol{h}_{j-1}}=\prod_{j=k+1}^{3}\text{Tanh}'\boldsymbol{W}_h \tag{3.28}$$

根据第 2 章介绍的 Tanh 函数，可以知道其导数形式为

$$\text{Tanh}'(x)=(1-\text{Tanh}^2(x)) \tag{3.29}$$

因此，$\text{Tanh}'(x)$ 值域为（0, 1]。如果 Tanh′ 与 $\boldsymbol{W}_h$ 的乘积小于 1，假设有 n 项连乘，则可以表示为（小于 1 的数）^{n}。随着时刻向前推移，梯度呈指数级下降，即出现梯度消失问题。

2. 梯度爆炸问题

如果 Tanh′ 与 $\boldsymbol{W}_h$ 的乘积大于 1，假设有 n 项连乘，则可以表示为（大于 1 的数）^{n}。随着时刻向前推移，梯度呈指数级上升，即出现梯度爆炸问题。

3. 梯度消失和梯度爆炸问题的缓解

无论是循环神经网络还是卷积神经网络等其他网络，它们出现梯度消失和梯度爆炸问题的本质原因是一样的，因此对于这些网络来说存在一些共性的缓解方案。

- **更换激活函数：**根据前面的分析，当使用 Tanh 函数时，若 Tanh′ 值过小，则会导致梯度消失问题，此时可以更换为 ReLU 函数。根据第 2 章介绍可以得到，ReLU 函数的导数在正数部分恒为 1，因此解决了由于激活函数导数的值过小导致的梯度消失问题，起到了缓解作用。
- **使用批归一化：**正如第 2 章所述，批归一化的思想是对每层的输入进行归一化，使其均值接近 0，标准差接近 1。这样，输入值就能落在激活函数的梯度非饱和区，也就是梯度较大的区域，从而缓解梯度消失问题。
- **梯度裁剪：**该方案主要针对梯度爆炸问题，其思想是对梯度设置一个裁剪阈值，在更新梯度时如果梯度超过这个阈值，则将其设置为阈值范围内的值，因此能够缓解梯度爆炸的问题。

3.3.4　双向循环神经网络

现在考虑下面三个文本填空任务：

我____

我____困，我刚起床

我____困，我想赶紧睡觉

可以分别填入“很高兴”“不”“非常”使句子意思表达流畅。很明显，短语的“下文”在填空任务中起到十分关键的作用，它传达的信息关乎选择什么词来填空。因此，如果无法

利用这一特性，普通的循环神经网络模型将在相关任务上表现不佳。而既可以学习正向特征，又可以学习反向特征的双向循环神经网络（Bidirectional Recurrent Neural Network，Bi-RNN）在完成该类任务时会有更高的拟合度。

双向循环神经网络采用了两个方向的循环神经网络：从第一个词元开始向后运行，以及从最后一个词元开始向前运行。其处理过程就是在正向传播的基础上再进行一次反向传播。正向传播和反向传播都连接着一个输出层。这个结构提供给输出层输入序列中每个点的完整的过去和未来的上下文信息。图 3.10 所示为双向循环神经网络架构。

对于时刻 t，$\boldsymbol{x}_t$ 表示输入，前向隐藏状态为 $\vec{\boldsymbol{h}}_t$，反向隐藏状态为 $\overleftarrow{\boldsymbol{h}}_t$，$\boldsymbol{o}_t$ 表示本时刻的输出，则前向隐藏状态和反向隐藏状态的更新可由如下公式表示：

$$\vec{\boldsymbol{h}}_t = \phi(\boldsymbol{W}_x^{(f)}\boldsymbol{x}_t + \boldsymbol{W}_h^{(f)}\vec{\boldsymbol{h}}_{t-1} + \boldsymbol{b}_h^{(f)}) \tag{3.30}$$

$$\overleftarrow{\boldsymbol{h}}_t = \phi(\boldsymbol{W}_x^{(b)}\boldsymbol{x}_t + \boldsymbol{W}_h^{(b)}\overleftarrow{\boldsymbol{h}}_{t-1} + \boldsymbol{b}_h^{(b)}) \tag{3.31}$$

其中，$\boldsymbol{W}_x^{(f)}$、$\boldsymbol{W}_x^{(b)}$、$\boldsymbol{W}_h^{(f)}$、$\boldsymbol{W}_h^{(b)}$ 为权重参数；$\boldsymbol{b}_h^{(f)}$、$\boldsymbol{b}_h^{(b)}$ 为偏置参数。

将前向隐藏状态 $\vec{\boldsymbol{h}}_t$ 和反向隐藏状态 $\boldsymbol{h}_t$ 连接起来，获得需要送入输入层的隐藏状态 $\boldsymbol{h}_t$。

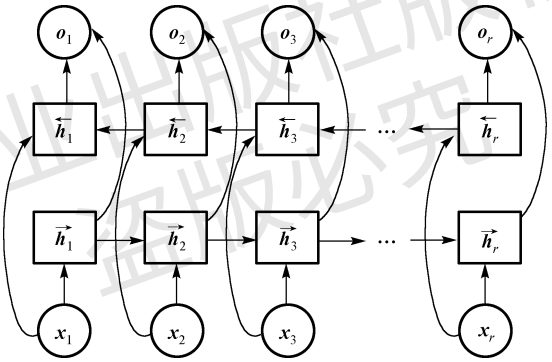


图 3.10　双向循环神经网络架构

输出可由如下公式计算：

$$\boldsymbol{o}_t = \boldsymbol{W}_h\boldsymbol{h}_t + \boldsymbol{b}_o \tag{3.32}$$

注意，若前向隐藏状态 $\vec{\boldsymbol{h}}_t$ 和反向隐藏状态 $\overleftarrow{\boldsymbol{h}}_t$ 的维度为 h，则连接后的隐藏状态 $\boldsymbol{h}_t$ 的维度将为 $2h$。

由于普通循环神经网络能够利用历史信息，因此其在处理序列数据时有显著优势，但存在一些问题：当训练深层网络时，循环神经网络面临梯度在反向传播过程中消失或爆炸的问题。正是由于梯度消失问题，普通循环神经网络难以学习和记忆过去很长时间内的输入信息，这个问题也在处理长序列和复杂序列模式时变得尤为明显。因此，长短期记忆网络和门控循环单元的出现缓解了这些问题。

3.4　长短期记忆网络

长短期记忆网络（Long Short-Term Memory Network，LSTM）是解决长期信息保存和短

期输入缺失问题最早的方法之一，出自 1997 年的论文“Long short-term memory”。它比门控循环单元更复杂，却比门控循环单元早近 20 年被提出。在解决长序列训练过程中的梯度消失和梯度爆炸问题上，长短期记忆网络也有不错的表现。

3.4.1　长短期记忆网络介绍

长短期记忆网络是循环神经网络的变体，和普通的循环神经网络相比，它主要改变了隐藏层 A 的结构，和3.5 节将要介绍的门控循环单元有许多相似之处，但是结构更复杂。长短期记忆网络引入了记忆元（Memory Cell）的概念，简称单元（Cell），它是除了输入 $\boldsymbol{x}_t$ 和隐藏状态 $\boldsymbol{h}_t$ 的另一个输入，其设计目的是用于记录附加的信息。还引入了门机制对当前的输入信息进行筛选，从而决定哪些信息可以传递到下一层。如图 3.11 所示，长短期记忆网络的主体结构由许多“门”构成，包括输入门、输出门和遗忘门。除了这几个重要的门，还包括候选单元状态、单元状态更新和隐藏状态更新等步骤。

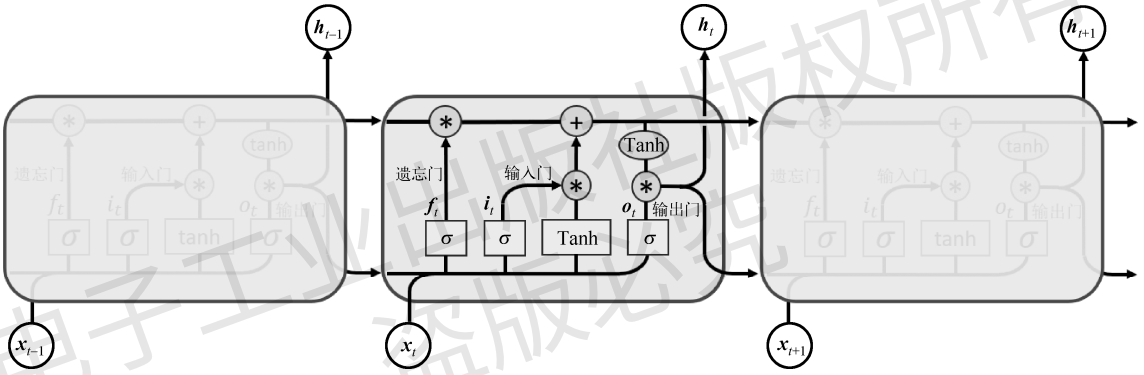


图 3.11　长短期记忆网络结构示意图

1. 遗忘门、输入门和输出门

简单来讲，遗忘门、输入门和输出门的功能也可以简要概括如下。

- 遗忘门：决定模型会从单元状态中丢弃什么信息。
- 输入门：决定模型要从候选单元状态中保存什么信息。
- 输出门：决定模型要将单元状态中的什么信息传递给隐藏状态。

如图 3.12 所示，当前时刻的输入为 $\boldsymbol{x}_t$，上一时刻的隐藏状态为 $\boldsymbol{h}_{t-1}$，则遗忘门 $\boldsymbol{f}_t$、输入门 $\boldsymbol{i}_t$、输出门 $\boldsymbol{o}_t$ 可由如下公式得到

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_x^{(f)}\boldsymbol{x}_t + \boldsymbol{W}_h^{(f)}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{(f)}) \tag{3.33}$$

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_x^{(i)}\boldsymbol{x}_t + \boldsymbol{W}_h^{(i)}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{(i)}) \tag{3.34}$$

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_x^{(o)}\boldsymbol{x}_t + \boldsymbol{W}_h^{(o)}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{(o)}) \tag{3.35}$$

其中，$\boldsymbol{W}_x^{(f)}$、$\boldsymbol{W}_x^{(i)}$、$\boldsymbol{W}_x^{(o)}$、$\boldsymbol{W}_h^{(f)}$、$\boldsymbol{W}_h^{(i)}$、$\boldsymbol{W}_h^{(o)}$ 是权重参数；$\boldsymbol{b}^{(f)}$、$\boldsymbol{b}^{(i)}$、$\boldsymbol{b}^{(o)}$ 是偏置参数；σ 是 Sigmod 函数。

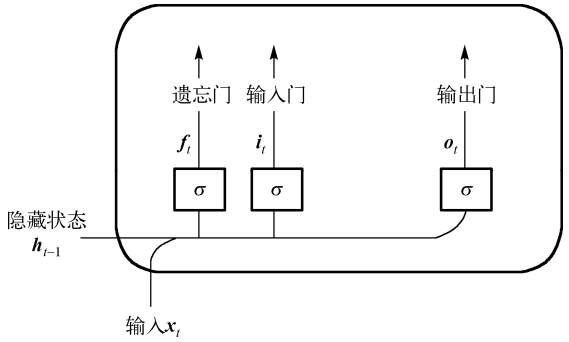


图 3.12 遗忘门、输入门和输出门

2. 候选单元状态

候选单元状态 $\tilde{\boldsymbol{c}}_t$ 的计算和上述三个门类似，可由如下公式得到

$$\tilde{\boldsymbol{c}}_t = \text{Tanh}(\boldsymbol{W}_x^{(c)}\boldsymbol{x}_t + \boldsymbol{W}_h^{(c)}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{(c)}) \tag{3.36}$$

其中，$\boldsymbol{W}_x^{(c)}$、$\boldsymbol{W}_h^{(c)}$ 是权重参数；$\boldsymbol{b}^{(c)}$ 是偏置参数。Tanh 函数使 $\tilde{\boldsymbol{c}}_t$ 中的值保持在区间(–1,1)。

3. 单元状态更新

如图 3.13 所示，类似于门控循环单元，长短期记忆网络利用输入门 $\boldsymbol{i}_t$ 和遗忘门 $\boldsymbol{f}_t$ 组合上一时刻的单元状态和候选单元状态，得到单元状态的更新公式：

$$\boldsymbol{c}_t = \boldsymbol{f}_t * \boldsymbol{c}_{t-1} + \boldsymbol{i}_t * \tilde{\boldsymbol{c}}_t \tag{3.37}$$

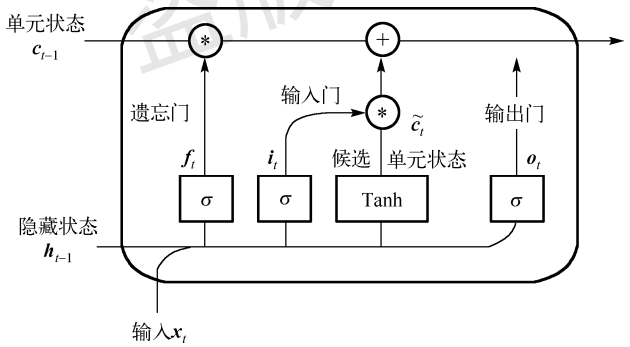


图 3.13 单元状态更新

若遗忘门 $\boldsymbol{f}_t$ 的值始终为 1 且输入门 $\boldsymbol{i}_t$ 的值始终为 0，则上一时刻的单元状态 $\boldsymbol{c}_{t-1}$ 将随时间被保存并传递到当前时刻。这种设计缓解了梯度消失问题，并更好地“捕获”序列中的长距离依赖关系。

4. 隐藏状态更新

利用输出门计算新的隐藏状态：

$$\boldsymbol{h}_t = \boldsymbol{o}_t * \text{Tanh}(\boldsymbol{c}_t) \tag{3.38}$$

Tanh 函数的使用确保了 $\boldsymbol{h}_t$ 中的值在区间(–1,1)中。当输出门 $\boldsymbol{o}_t$ 中的值接近 1 时，就能够将单元内的信息传递给隐藏状态，而当输出门 $\boldsymbol{o}_t$ 中的值接近 0 时，不需要更新隐藏状态。

3.4.2　长短期记忆网络应用

由于长短期记忆网络在解决长期信息保存等方面的优点，它比普通的循环神经网络应用更加广泛。下面将介绍一个使用长短期记忆网络完成的情感分析项目，实现对影评文本的分析。该项目来自极客时间《PyTorch 深度学习实战》。

1. 数据准备

这里利用 TorchText 工具包从 IMDB 中读取数据集。IMDB 是一个互联网电影数据库，其中包含了 50000 条严重两极分化的电影评论，该数据集被平均划分为训练集和测试集，各含有 25000 条评论，其中有 50%的正面评论和 50%的负面评论。

```
# 读取 IMDB 数据集
import torchtext
train_iter = torchtext.datasets.IMDB(root='./data',split='train')
next(train_iter)
```

2. 数据处理

读取数据集后，需要将文本和分类标签处理成向量。首先，创建一个英文分词器以完成英文的分词。然后，根据 IMDB 数据集的训练集迭代器 train_iter 构建词汇表 vocab。

```
# 创建分词器
tokenizer = torchtext.data.utils.get_tokenizer('basic_english')
print(tokenizer('here is the an example!'))
'''
输出：['here','is','the','an','example','!']
'''

# 构建词汇表
def yield_tokens(data_iter):
    for _,text in data_iter:
        yield tokenizer(text)

vocab = torchtext.vocab.build_vocab_from_iterator(yield_tokens
(train_iter),specials=["<pad>","<unk>"])
vocab.set_default_index(vocab["<unk>"])

print(vocab(tokenizer('here is the an example <pad> <pad>')))
'''
输出：[131,9,40,464,0,0]
'''
```

为了方便后续调用，创建了 text_pipeline 和label_pipeline。text_pipeline 用于给定一段文本，返回分词后的序号。label_pipeline 用于将情绪分类转化为数字，即将“neg”转化为 0，将“pos”转化为 1。

```
# 数据处理 pipelines
text_pipeline = lambda x: vocab(tokenizer(x))
```

```
label_pipeline = lambda x: 1 if x == 'pos' else 0

print(text_pipeline('here is the an example'))
print(label_pipeline('neg'))
'''
输出：[131,9,40,464,0,0 ,... , 0]
输出：0
'''
```

3. 生成训练数据

这里的 collate_batch 函数需要完成一系列的数据处理工作：生成文本的 tensor、生成标签的 tensor、生成句子长度的 tensor，以及对文本进行截断、补位操作等。

```
import torch
import torchtext
from torch.utils.data import DataLoader
from torch.utils.data.dataset import random_split
from torchtext.data.functional import to_map_style_dataset

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# 完成部分数据处理工作
def collate_batch(batch):
    max_length = 256
    pad = text_pipeline('<pad>')
    label_list,text_list,length_list = [],[],[]
    # 获取标签、文本
    for (_label,_text) in batch:
        label_list.append(label_pipeline(_label))
        processed_text = text_pipeline(_text)[:max_length]
        length_list.append(len(processed_text))
        text_list.append((processed_text+pad*max_length)[:max_length])
    # 将数据转为 tensor
    label_list = torch.tensor(label_list,dtype=torch.int64)
    text_list = torch.tensor(text_list,dtype=torch.int64)
    length_list = torch.tensor(length_list,dtype=torch.int64)
    return label_list.to(device),text_list.to(device),length_
    list.to(device)

train_iter = torchtext.datasets.IMDB(root='./data',split='train')
train_dataset = to_map_style_dataset(train_iter)
num_train = int(len(train_dataset) * 0.95)
split_train_,split_valid_ = random_split(train_dataset,[num_train,
len(train_dataset) - num_train])
train_dataloader = DataLoader(split_train_,batch_size=8,shuffle
=True,collate_fn=collate_batch)
valid_dataloader = DataLoader(split_valid_,batch_size=8,shuffle
 =False,collate_fn=collate_batch)
```

4．模型构建

这里使用长短期记忆网络进行情绪分类的预测。首先是一个 Embedding 层，用来接收文本序号的 tensor，然后是长短期记忆网络层，最后是一个全连接层用于分类。其中，当 bidirectional 为 True 时，表示网络为双向长短期记忆网络，当 bidirectional 为False 时，表示网络为单向长短期记忆网络。

```
class LSTM(torch.nn.Module):
    def _ _init_ _ (self,vocab_size,embedding_dim,hidden_dim,output_dim,n_layers,bidirectional,
            dropout_rate,pad_index=0):
        super()._ _init_ _ ()
        self.embedding = torch.nn.Embedding(vocab_size,embedding_dim,padding_idx=pad_index)
        self.lstm = torch.nn.LSTM(embedding_dim,hidden_dim,n_layers,bidirectional=bidirectional,
                dropout=dropout_rate,batch_first=True)
        self.fc = torch.nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim,output_dim)
        self.dropout = torch.nn.Dropout(dropout_rate)

def forward(self,ids,length):
    embedded = self.dropout(self.embedding(ids))
    packed_embedded = torch.nn.utils.rnn.pack_padded_sequence(embedded,length,batch_first=True,enforce_sorted=False)
    packed_output,(hidden,cell) = self.lstm(packed_embedded)
    output,output_length = torch.nn.utils.rnn.pad_packed_ sequence(packed_output)
    if self.lstm.bidirectional:
        hidden = self.dropout(torch.cat([hidden[-1],hidden[-2]],dim=-1))
    else:
        hidden = self.dropout(hidden[-1])
    prediction = self.fc(hidden)
    return prediction
```

5．模型训练和评估

现在可以进行模型训练。首先要实例化网络模型，然后定义损失函数和优化方法。其中，由于数据的情感共分为两类，因此这里的 output_dim 设置为 2。

```
# 实例化模型
vocab_size = len(vocab)
embedding_dim = 300
hidden_dim = 300
output_dim = 2
n_layers = 2
bidirectional = True
dropout_rate = 0.5
```

```
9
10  model = LSTM(vocab_size,embedding_dim,hidden_dim,output_dim,
    n_layers,bidirectional,dropout_rate)
11  model = model.to(device)
12
13  # 损失函数与优化方法
14  lr = 5e-4
15  criterion = torch.nn.CrossEntropyLoss()
16  criterion = criterion.to(device)
17
18  optimizer = torch.optim.Adam(model.parameters(),lr=lr)
```

下面是训练过程、验证过程及模型评估的代码。

```
1   def train(dataloader,model,criterion,optimizer,device):
2       model.train()
3       epoch_losses,epoch_accs = [],[]
4       for batch in tqdm.tqdm(dataloader,desc='training...',file=
        sys.stdout):
5           (label,ids,length) = batch
6           label = label.to(device)
7           ids = ids.to(device)
8           length = length.to(device)
9           prediction = model(ids,length)
10          loss = criterion(prediction,label) # loss计算
11          accuracy = get_accuracy(prediction,label)
12          # 梯度更新
13          optimizer.zero_grad()
14          loss.backward()
15          optimizer.step()
16          epoch_losses.append(loss.item())
17          epoch_accs.append(accuracy.item())
18          return epoch_losses,epoch_accs
19
20  def evaluate(dataloader,model,criterion,device):
21      model.eval()
22      epoch_losses,epoch_accs = [],[]
23      with torch.no_grad():
24          for batch in tqdm.tqdm(dataloader,desc='evaluating...',
            file=sys.stdout):
25              (label,ids,length) = batch
26              label = label.to(device)
27              ids = ids.to(device)
28              length = length.to(device)
29              prediction = model(ids,length)
30              loss = criterion(prediction,label) # loss计算
31              accuracy = get_accuracy(prediction,label)
32              epoch_losses.append(loss.item())
```

```
            epoch_accs.append(accuracy.item())
    return epoch_losses,epoch_accs

def get_accuracy(prediction,label):
    batch_size,_ = prediction.shape
    predicted_classes = prediction.argmax(dim=-1)
    correct_predictions = predicted_classes.eq(label).sum()
    accuracy = correct_predictions / batch_size
    return accuracy
```

训练过程的具体代码如下，包括计算损失和准确度，保存损失列表和最优模型。

```
n_epochs = 10
best_valid_loss = float('inf')

train_losses,train_accs,valid_losses,valid_accs = [],[],[],[]
    for epoch in range(n_epochs):
    train_loss,train_acc = train(train_dataloader,model,criterion,
    optimizer,device)
    valid_loss,valid_acc = evaluate(valid_dataloader,model,criterion,
    device)
    train_losses.extend(train_loss)
    train_accs.extend(train_acc)
    valid_losses.extend(valid_loss)
    valid_accs.extend(valid_acc)
    epoch_train_loss = np.mean(train_loss)
    epoch_train_acc = np.mean(train_acc)
    epoch_valid_loss = np.mean(valid_loss)
    epoch_valid_acc = np.mean(valid_acc)
    if epoch_valid_loss < best_valid_loss:
        best_valid_loss = epoch_valid_loss
        torch.save(model.state_dict(),'lstm.pt')
    print(f'epoch: {epoch+1}')
    print(f'train_loss: {epoch_train_loss:.3f},train_acc: {epoch_
    train_acc:.3f}')
    print(f'valid_loss: {epoch_valid_loss:.3f},valid_acc: {epoch_
    valid_acc:.3f}')
```

3.5　门控循环单元

前面的章节中提到在循环神经网络计算梯度时会出现梯度消失和梯度爆炸问题，门控循环单元（Gated Recurrent Unit，GRU）的提出就是为了解决这类反向传播中的梯度问题及长期记忆问题。此外，相对于 3.4 节介绍的长短期记忆网络，门控循环单元能在提供同等效果的同时有更快的计算速度，因此常用于文本生成、情感分析等自然语言处理任务。

3.5.1 门控循环单元介绍

门控循环单元也是循环神经网络的一种变体，和长短期记忆网络类似且结构更简单，能在提供和长短期记忆网络同等效果的同时拥有更快的计算速度。门控循环单元的结构示意图如图 3.14 所示，主要包括重置门、更新门两个门结构，候选隐藏状态及隐藏状态更新两个主要步骤。

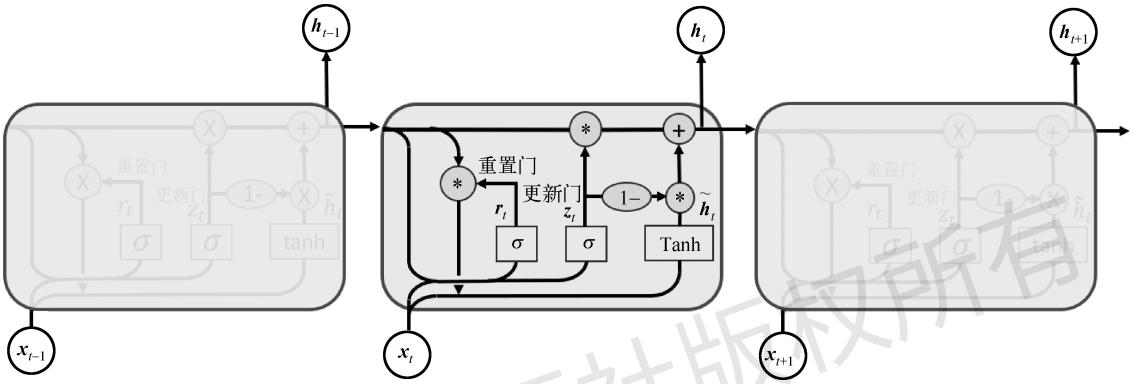


图 3.14 门控循环单元的结构示意图

1. 重置门和更新门

简单来讲，重置门和更新门的功能可以简要概括如下。

- 重置门：决定隐藏状态中的什么信息需要保存。
- 更新门：决定新的隐藏状态有多少来自候选隐藏状态，有多少来自旧隐藏状态。

如图 3.15 所示，当前时刻的输入为 $\boldsymbol{x}_t$，上一时刻的隐藏状态为 $\boldsymbol{h}_{t-1}$，重置门 $\boldsymbol{r}_t$ 和更新门 $\boldsymbol{z}_t$ 可由如下公式得到

$$\boldsymbol{r}_t = \sigma(\boldsymbol{W}_x^{(r)}\boldsymbol{x}_t + \boldsymbol{W}_h^{(r)}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{(r)}) \tag{3.39}$$

$$\boldsymbol{z}_t = \sigma(\boldsymbol{W}_x^{(z)}\boldsymbol{x}_t + \boldsymbol{W}_h^{(z)}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{(z)}) \tag{3.40}$$

其中，$\boldsymbol{W}_x^{(r)}$、$\boldsymbol{W}_h^{(r)}$、$\boldsymbol{W}_x^{(z)}$、$\boldsymbol{W}_h^{(z)}$ 为权重参数；$\boldsymbol{b}^{(r)}$、$\boldsymbol{b}^{(z)}$ 为偏置参数；σ 为 Sigmod 函数。

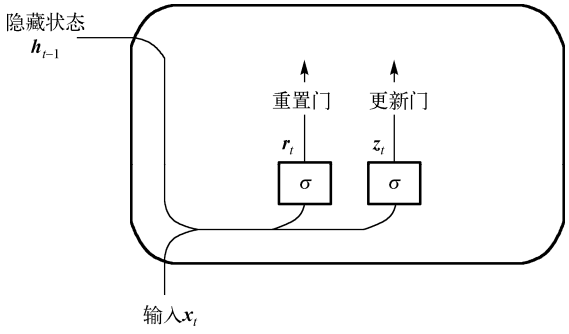


图 3.15 重置门和更新门

2．候选隐藏状态

如图 3.16 所示，根据输入 $\boldsymbol{x}_t$，以及上一时刻的隐藏状态 $\boldsymbol{h}_{t-1}$ 和重置门 $\boldsymbol{r}_t$，候选隐藏状态的计算可由如下公式得到

$$\tilde{\boldsymbol{h}}_t = \text{Tanh}(\boldsymbol{W}_x^{(h)}\boldsymbol{x}_t + \boldsymbol{W}_h^{(h)}(\boldsymbol{r}_t * \boldsymbol{h}_{t-1}) + \boldsymbol{b}^{(h)}) \tag{3.41}$$

其中，$\boldsymbol{W}_x^{(h)}$、$\boldsymbol{W}_h^{(h)}$ 为权重参数；$\boldsymbol{b}^{(h)}$ 为偏置参数；Tanh 函数使候选隐藏状态的值保持在区间 (−1, 1)中。可以看到，当 $\boldsymbol{r}_t$ 中的值接近 1 时，模型就接近一个普通的循环神经网络；当 $\boldsymbol{r}_t$ 中的值接近 0 时，上一时刻的隐藏状态接近被忽略，候选隐藏状态是将 $\boldsymbol{x}_t$ 输入线性层的结果。

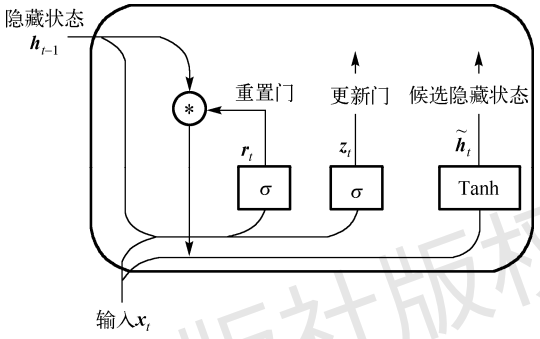


图 3.16　候选隐藏状态

3．隐藏状态更新

如图 3.17 所示，利用更新门 $\boldsymbol{z}_t$ 组合上一时刻的隐藏状态和候选隐藏状态，得到隐藏状态的更新公式：

$$\boldsymbol{h}_t = \boldsymbol{z}_t * \boldsymbol{h}_{t-1} + (1 - \boldsymbol{z}_t) * \tilde{\boldsymbol{h}}_t \tag{3.42}$$

其中，符号 * 表示 Hadamard 积，即按元素乘积。可以看到，当 $\boldsymbol{z}_t$ 中的值接近 1 时，$\boldsymbol{h}_t$ 就接近于 $\boldsymbol{h}_{t-1}$，模型就倾向于保留旧状态。相反，当 $\boldsymbol{z}_t$ 中的值接近 0 时，$\boldsymbol{h}_t$ 就接近于 $\tilde{\boldsymbol{h}}_t$。这可以帮助处理循环神经网络中梯度消失问题，并更好地“捕获”距离很长的序列的依赖关系。

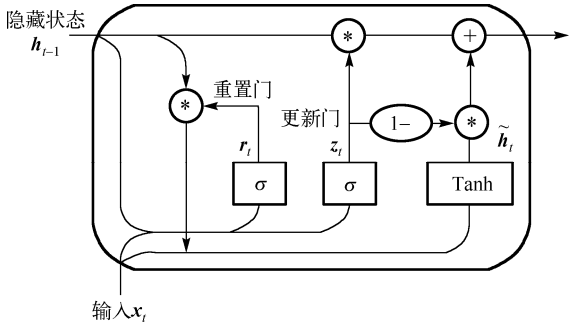


图 3.17　隐藏状态更新

3.5.2　门控循环单元应用

门控循环单元在处理序列数据时表现出色，能很好地应用于时间序列预测、文本生成等

任务。下面展示一段门控循环单元应用于时间序列预测任务的代码，以便读者更好地理解门控循环单元的作用。

1. 数据获取与处理

这里使用的是一个电力负荷数据集，包含有关电力系统的电力负荷、价格、天气情况等特征，常用于预测未来的电力需求或价格。首先从.csv 文件中读取数据，然后定义训练集和测试集的尺寸、预测数据长度及观测窗口。其中，观测窗口表示利用多少数据去预测。

```
true_data = pd.read_csv('ETTh1.csv')

target = 'OT' # 要预测的特征列
test_size = 0.15 #  测试集的尺寸划分
train_size = 0.85 #  训练集的尺寸划分
pre_len = 4 # 预测未来数据的长度
train_window = 32 # 观测窗口

true_data = np.array(true_data[target])
```

定义标准化优化器，根据定义的尺寸划分训练集和测试集，对数据进行标准化处理，将数据格式转化为 tensor。

```
# 定义标准化优化器
scaler_train = MinMaxScaler(feature_range=(0,1))
scaler_test = MinMaxScaler(feature_range=(0,1))
# 训练集和测试集划分
train_data = true_data[:int(train_size * len(true_data))]
test_data = true_data[-int(test_size * len(true_data)):]
print(" 训练集尺寸:",len(train_data))
print(" 测试集尺寸:",len(test_data))
# 进行标准化处理
train_data_normalized = scaler_train.fit_transform(train_data.
 reshape(-1,1))
test_data_normalized = scaler_test.fit_transform(test_data.
 reshape(-1,1))
# 转化为深度学习模型需要的类型 tensor
train_data_normalized = torch.FloatTensor(train_data_normalized)
test_data_normalized = torch.FloatTensor(test_data_normalized)
```

定义训练器的输入、创建数据集和 DataLoader 数据加载器。

```
class TimeSeriesDataset(Dataset):
    def _ _init_ _(self,sequences):
        self.sequences = sequences
    def _ _len_ _(self):
        return len(self.sequences)
    def _ _getitem_ _(self,index):
        sequence,label = self.sequences[index]
        return torch.Tensor(sequence),torch.Tensor(label)
```

```
9
10  def create_inout_sequences(input_data,tw,pre_len):
11      # 创建时间序列数据专用的数据分割器
12      inout_seq = []
13      L = len(input_data)
14      for i in range(L - tw):
15          train_seq = input_data[i:i + tw]
16          if (i + tw + 4) > len(input_data):
17              break
18          train_label = input_data[i + tw:i + tw + pre_len]
19          inout_seq.append((train_seq,train_label))
20      return inout_seq
21
22  # 定义训练器的输入
23  train_inout_seq = create_inout_sequences(train_data_normalized,train_window,pre_len)
24  test_inout_seq = create_inout_sequences(test_data_normalized,train_window,pre_len)
25  # 创建数据集
26  train_dataset = TimeSeriesDataset(train_inout_seq)
27  test_dataset = TimeSeriesDataset(test_inout_seq)
28  # 创建 DataLoader 数据加载器
29  batch_size = 32  # 根据需要调整批量大小
30  train_loader = DataLoader(train_dataset,batch_size=batch_size,shuffle=True,
     drop_last=True)
31  test_loader = DataLoader(test_dataset,batch_size=batch_size,shuffle=False,
     drop_last=True)
```

2. 模型构建

这里使用门控循环单元网络，Dropout 用于避免过拟合。实例化模型，损失函数为均方误差损失函数，优化器为 Adam，epochs 为训练轮次。

```
1   class GRU(nn.Module):
2       def _ _init_ _(self,input_dim=1,hidden_dim=32,num_layers=1,
        output_dim=1,pre_len= 4):
3           super(GRU,self). _ _init _ _ ()
4           self.pre_len = pre_len
5           self.num_layers = num_layers
6           self.hidden_dim = hidden_dim
7           self.gru = nn.GRU(input_dim,hidden_dim,num_layers =num_
            layers,batch_first=True)
8           self.fc = nn.Linear(hidden_dim,output_dim)
9           self.relu = nn.ReLU()
10          self.dropout = nn.Dropout(0.1)
11
12      def forward(self,x):
13          h0_gru = torch.zeros(self.num_layers,x.size(0),self.
             hidden_dim).to(x.device)
14          out,_ = self.gru(x,h0_gru)
```

```
        out = self.dropout(out)
        # 取最后 pre_len 时间步的输出
        out = out[:,-self.pre_len:,:]
        out = self.fc(out)
        out = self.relu(out)
        return out

model = GRU(input_dim=1,output_dim=1,num_layers=2,hidden_dim=
train_window,pre_len=pre_len)
loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(),lr=0.005)
epochs = 20
Train = True # 训练还是预测
```

3. 模型训练和预测

Train 用于确定进行模型训练还是预测，当 Train 为 True 时进行训练。

```
if Train:
    losses = []
    model.train() # 训练模式
    for i in range(epochs):
        start_time = time.time() # 计算起始时间
        for seq,labels in train_loader:
            model.train()
            optimizer.zero_grad()
            y_pred = model(seq)
            single_loss = loss_function(y_pred,labels)
            single_loss.backward()
            optimizer.step()
            print(f'epoch: {i:3} loss: {single_loss.item():10.8f}')
            losses.append(single_loss.detach().numpy())
        torch.save(model.state_dict(),'save_model.pth')
        print(f" 模型已保存，用时:{(time.time() - start_time) / 60:.4f} min")
```

当 Train 为 False 时使用测试集评估训练好的模型。

```
else:
    # 加载模型进行预测
    model.load_state_dict(torch.load('save_model.pth'))
    lstm_model.eval() # 评估模式
    results = []
    reals = []
    losses = []

    for seq,labels in test_loader:
        pred = model(seq)
        # 均方误差计算绝对值(预测值-真实值)
        mae = calculate_mae(pred.detach().numpy(),np.array(labels))
        losses.append(mae)
        for j in range(batch_size):
```

```
        for i in range(pre_len):
            reals.append(labels[j][i][0].detach().numpy())
            results.append(pred[j][i][0].detach().numpy())

    reals=scaler_test.inverse_transform(np.array(reals).reshape(1,-1))[0]
    results = scaler_test.inverse_transform(np.array(results). reshape(1,-1))[0]
    print(" 模型预测结果: ",results)
    print(" 预测误差 MAE:",losses)
```

3.6　思　　考

本章介绍了一些比较基础的自然语言处理知识，包括词嵌入、循环神经网络、门控循环单元、长短期记忆网络等。词嵌入将词汇转换为密集的、连续的词向量表示，捕捉语义和语法等语言信息，使非结构化的文本数据变成能被计算机识别的形式。循环神经网络能很好地处理包含时序信息的输入，在处理序列数据时具有天然的优势，但会出现梯度消失和梯度爆炸问题，门控循环单元和长短期记忆网络的出现很好地解决了这类梯度问题，同时解决了长期记忆问题。这些基础知识为自然语言处理打下了坚实的基础，在自然语言发展过程中具有里程碑式的意义。

随着技术的不断发展，自然语言处理方向也不断地有卓越的工作出现，如机器翻译模型 Transformer、基于 Transformer 的预训练语言表示模型 BERT、GPT 系列等。尽管这些工作的出现对传统的基于监督学习的自然语言处理模型产生了巨大的影响，但是自然语言处理是构成 AI 核心的关键环节，它标志着 AI 技术由简单的感知功能迈向更高层次的认知功能的转变。

自然语言处理基础知识中的词嵌入、循环神经网络、门控循环单元及长短期记忆网络等技术也可以在当前大模型的背景下从以下方面进一步思考。

- 词嵌入：词嵌入是将单词映射到高维实数向量空间的技术，通常用于将文本转换为计算机可处理的形式。在大模型的背景下，读者可以思考如何进一步改进词嵌入的质量和效率，以提高模型的性能和泛化能力。
- 循环神经网络：循环神经网络是一种能够处理序列数据的神经网络结构，在自然语言处理任务中被广泛应用。在大模型的背景下，读者可以思考如何通过设计更深、更复杂的循环神经网络结构来提高模型的性能，并探索如何解决循环神经网络中的梯度消失和梯度爆炸等问题。
- 门控循环单元：门控循环单元是一种改进的循环神经网络结构，具有更少的参数和更简单的计算过程。在大模型的背景下，研究人员可以思考如何利用门控循环单元的优势来设计更高效、更稳定的模型，并探索如何在大规模数据集上训练更大规模的门控循环单元模型。
- 长短期记忆网络：长短期记忆网络是一种具有长期记忆能力的循环神经网络变体，在处理长序列数据时表现出色。在大模型的背景下，读者可以思考如何进一步改进长短期记忆网络的结构和性能，以应对更复杂的自然语言处理任务，并探索如何将长短期记忆网络与其他技术结合，以提高模型的性能和泛化能力。

自然语言处理技术是大模型发展的主要驱动力之一，通过使用更多的数据和更复杂的模型结构，自然语言处理技术推动了大模型的发展，不断提升大模型的性能和泛化能力，为大模型的进一步发展提供更广阔的空间和更深层次的探索。

习 题 3

理论习题

1．请简述连续词袋模型和跳元模型的区别。

2．请根据 3.3.2 节介绍的 BPTT 算法，计算损失 E_2 分别对参数 $\boldsymbol{W}_o$、$\boldsymbol{W}_h$、$\boldsymbol{W}_x$ 的梯度。

3．什么是梯度消失问题？如何解决？

4．梯度消失问题是否可以通过增加学习率来缓解？

5．请简述长短期记忆网络和门控循环单元的区别。

6．梯度消失是训练深度学习网络时常见的问题之一，分析长短期记忆网络是如何避免梯度消失的。

实践习题

1. 下载在 Google News 数据集(大约有1000 亿个词)上预训练得到的词向量，找出与“Artificial Intelligence”余弦相似度最高的 10 个词，并输出其相似度的数值。

2. 在 3.2.3 节中介绍了使用 Gensim 库完成词向量操作的示例(如应用到单词相似性任务、找出最大差异词汇任务上)，请自行搜索其他可应用的任务并尝试学习如何使用。

3．训练一个机器翻译模型实现英文-中文翻译任务，尝试使用不同的网络（长短期记忆网络、门控循环单元），并对比其表现。

4．目前，我国中成药出口已遍及全球 130 个国家和地区。中成药均附有中成药说明书，但是一般为中文，需要翻译成英语，以便于患者按照说明书正确服用。尝试搜集不少于 1000 种中成药说明书，构建一个中成药说明书翻译语料库。使用预训练后的大模型在自建的中成药说明书翻译语料库上进行微调训练。测试经过“预训练-微调”后的大模型针对中成药说明书的翻译效果。