第5章　自然语言处理

5.1　自然语言处理

5.1.1　概述

自然语言处理(Natural Language Processing,简称 NLP)被誉为人工智能皇冠上的明珠,是计算机科学和人工智能领域的一个重要方向。它主要研究人与计算机之间,使用自然语言进行有效通信的各种理论和方法。简单来说,计算机以用户的自然语言形式作为输入,在其内部通过定义的算法进行加工、计算等系列操作后(用以模拟人类对自然语言的理解),再返回用户所期望的结果,如图 5.1 所示。

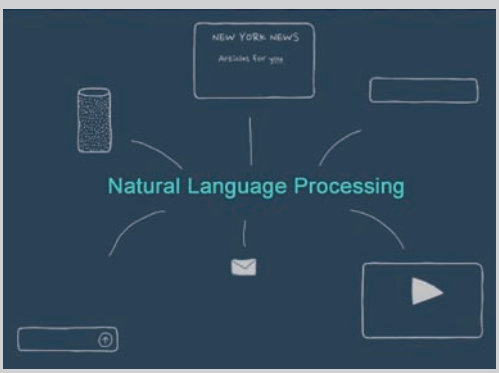


图 5.1　自然语言处理示意图

自然语言处理是一门融语言学、计算机科学和数学于一体的科学。它不仅限于研究语言学,还是研究能高效实现自然语言理解和自然语言生成的计算机系统,特别是其中的软件系统,因此它是计算机科学的一部分。

随着计算机和互联网技术的发展,自然语言处理技术在各领域广泛应用,如图 5.2 所示。在过去的几个世纪,工业革命用机械解放了人类的双手,在当今的人工智能革命中,计算机将代替人工,处理大规模的自然

■图 5.2 自然语言处理技术在各领域的应用

语言信息。我们平时常用的搜索引擎，新闻推荐，智能音箱等产品，都是以自然语言处理技术为核心的互联网和人工智能产品。

此外，自然语言处理技术的研究也在日新月异变化，每年投向计算语言学年会（Association for Computational Linguistics，ACL，自然语言处理领域的顶级会议）的论文数成倍增长，自然语言处理的应用效果被不断刷新，有趣的任务和算法更是层出不穷。

本节将简要介绍自然语言处理的发展历程、主要挑战，以及如何使用飞桨快速完成各项常见的自然语言处理任务。

致命密码：一场关于语言的较量

事实上，人们并非只在近代才开始研究和处理自然语言，在漫长的历史长河中，对自然语言妥当处理往往决定了战争的胜利或是政权的更迭。

16 世纪的英国大陆，英格兰和苏格兰刚刚完成统一，统治者为英格兰女王伊丽莎白一世，苏格兰女王玛丽因被视为威胁而遭到囚禁。玛丽女王和其他苏格兰贵族谋反，这些贵族们通过信件同被囚禁的玛丽女王联络，商量如何营救玛丽女王并推翻伊丽莎白女王的统治。为了能更安全地跟同伙沟通，玛丽使用了一种传统的文字加密形式——凯撒密码对他们之间的信件进行加密，如图 5.3 所示。

这种密码通过把原文中的每个字母替换成另外一个字符的形式，达到加密手段。然而他们的阴谋活动早在英格兰贵族监控之下，英格兰国务大臣弗朗西斯·沃尔辛厄姆爵士通过统计英文字母的出现频率和玛丽女王密函中的字母频率，找到了破解密码的规律。最终，玛丽和其他贵族在举兵谋反前夕被捕。这是近代西方第一次破译密码，开启了近现代密码学的先河。

图 5.3 凯撒密码

5.1.2 自然语言处理的发展历程

自然语言处理有着悠久的发展史，可粗略地分为兴起、符号主义、连接主义和深度学习四个阶段，如图 5.4 所示。

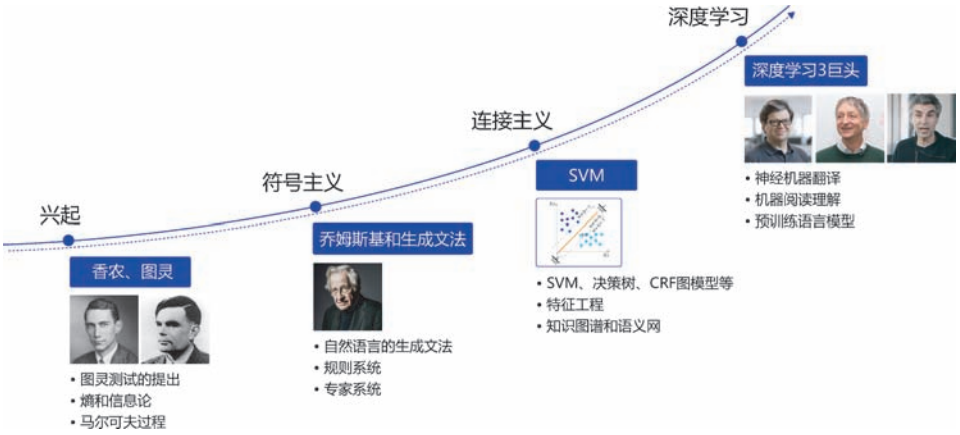


图 5.4 自然语言处理的发展历程

1. 兴起时期

大多数人认为，自然语言处理的研究兴起于 1950 年前后。在二战中，破解纳粹德国的恩尼格玛密码成为盟军对抗纳粹的重要战场，如图 5.5 所示。经过二战的洗礼，曾经参与过密码破译的香农和图灵等科学家开始思考自然语言处理和计算之间的关系。

1948 年香农把马尔可夫过程模型（Markov Progress）应用于建模自然语言，并提出把热力学中“熵”（Entropy）的概念扩展到自然语言建模领域。香农相信，自然语言跟其他物理世界的信号一样，是具有统计学规律的，通过统计分析可以帮助我们更好地理解自然语言。

1950年，艾伦·图灵提出著名的图灵测试，标志着人工智能领域的开端。二战后，受到美苏冷战的影响，美国政府开始重视机器自动翻译的研究工作，以便于随时监视苏联最新的科技进展。1954年美国乔治城大学在一项实验中，成功将约60句俄文自动翻译成英文，被视为机器翻译可行的开端。自此开始的十年间，政府与企业相继投入大量的资金，用于机器翻译的研究。

图5.5 恩尼格玛密码机

1956年，乔姆斯基(Chomsky)提出了“生成式文法”这一大胆猜想，他假设在客观世界存在一套完备的自然语言生成规律，每一句话都遵守这套规律而生成。总结出这个客观规律，人们就掌握了自然语言的奥秘。从此，自然语言的研究就被分为了以语言学为基础的符号主义学派，以及以概率统计为基础的连接主义学派。

2. 符号主义时期

在自然语言处理发展的初期阶段，大量的自然语言研究工作都聚焦从语言学角度，分析自然语言的词法、句法等结构信息，并通过总结这些结构之间的规则，达到处理和使用自然语言的目的。这一时期的代表人物就是乔姆斯基和他提出的“生成式文法”。1966年，完全基于规则的对话机器人ELIZA在MIT人工智能实验室诞生了，如图5.6所示。

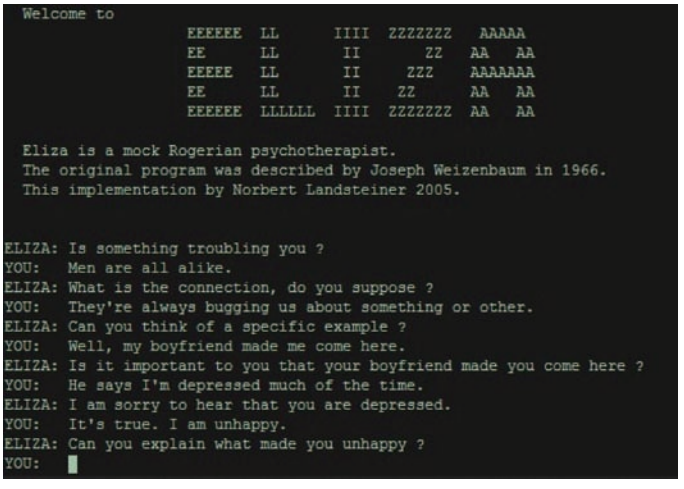


图5.6 基于规则的聊天机器人ELIZA

然而同年，自动语言处理顾问委员会(Automatic Language Processing Advisory Committee，ALPAC)的一项报告中提出，十年来的机器翻译研究进度缓慢、未达预期。该项报告发布后，机器翻译和自然语言的研究资金大为减缩，自然语言处理和人工智能的研究进入寒冰期。

3. 连接主义时期

1980年，由于计算机技术的发展和算力的提升，个人计算机可以处理更加复杂的计算任务，自然语言处理研究得以复苏，研究人员开始使用统计机器学习方法处理自然语言任务。

起初研究人员尝试使用浅层神经网络，结合少量标注数据的方式训练模型，虽然取得了一定的效果，但是仍然无法让大部分人满意。后来研究者开始使用人工提取自然语言特征的方式，结合简单的统计机器学习算法解决自然语言问题。其实现方式是基于研究者在不同领域总结的经验，将自然语言抽象成一组特征，使用这组特征结合少量标注样本，训练各种统计机器学习模型（如支持向量机、决策树、随机森林、概率图模型等），完成不同的自然语言任务。

由于这种方式基于大量领域专家经验积累（如解决一个情感分析任务，那么一个很重要的特征就是是否有命中情感词表），以及传统机器学习简单、鲁棒性强的特点，这个时期神经网络技术被大部分人所遗忘。

4. 深度学习时期

从2006年深度神经网络反向传播算法的提出开始，伴随着互联网的爆炸式发展和计算机（特别是GPU）算力的进一步提高，人们不再依赖语言学知识和有限的标注数据，自然语言处理领域迈入了深度学习时代。

基于互联海量数据，并结合深度神经网络的强大拟合能力，人们可以非常轻松地应对各种自然语言处理问题。越来越多的自然语言处理技术趋于成熟并显现出巨大的商业价值，自然语言处理和人工智能领域的发展进入了鼎盛时期。

自然语言处理的发展经历了多个历史阶段的演进，不同学派之间相互补充促进，共同推动了自然语言处理技术的快速发展。

5.1.3 自然语言处理技术面临的挑战

如何让机器像人一样，能够准确理解和使用自然语言？这是当前自然语言处理领域面临的最大挑战。为了解决这一问题，我们需要从语言学和计算两个角度思考。

1. 语言学角度

自然语言数量多、形态各异，理解自然语言对人来说本身也是一件复杂的事情，如同义词、情感倾向、歧义性、长文本处理、语言惯性表达等。通过如下几个例子，我们一同感受一下。

1）同义词问题

请问下列词语是否为同义词？（题目来源：四川话和东北话6级模拟考试）

瓜兮兮 和 铁憨憨

嘎嘎 和 肉（you）

磕碜 和 难看

吭吡瘪肚 和 速度慢

2）情感倾向问题

请问如何正确理解下面两个场景？

场景一：女朋友生气了，男朋友电话道歉。

女生：就算你买包我也不会原谅你！

男生：宝贝，放心，我不买，你别生气了？

问：女生会不会生气。

场景二：两个同宿舍的室友，甲和乙对话

甲：钥匙好像没了，你把锁别别。

乙：到底没没没？

甲：我也不道没没没。

乙：要没没你让我别，别别了，别秃鲁了咋整。

问：到底别不别？

3）歧义性问题

请问如何理解下面三句话？

一行行行行行，一行不行行行不行

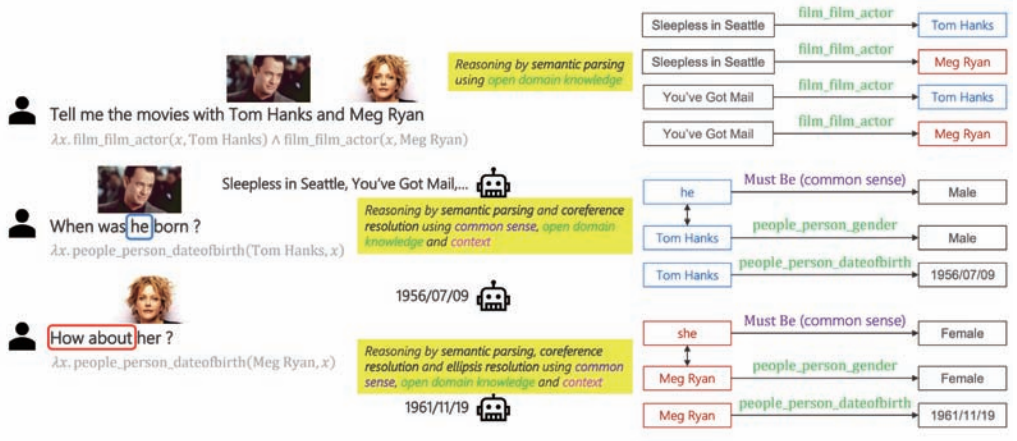
来到杨过曾经生活过的地方，小龙女说："我也想过过过儿过过的生活"

来到儿子等校车的地方，邓超对孙俪说："我也想等等等等等过的那辆车"

相信大多数人都需要花点脑筋去理解上面的句子，在不同的上下文中，相同的单词可以具有不同的含义，这种问题我们称之为歧义性问题。

4）对话/篇章等长文本处理问题

在处理长文本（如一篇新闻报道，一段多人对话，甚至于一篇长篇小说）时，需要经常处理各种省略、指代、话题转折和切换等语言学现象，如图 5.7 所示，这些都给机器理解自然语言带来了挑战。



■图 5.7　多轮对话中的指代和省略

5）探索自然语言理解的本质问题

> 研表究明，汉字的顺序并不定一能影阅响读，比如当你看完这句话后，才发这现里的字全是都乱的。

上面这句话从语法角度来说完全是错的，但是对大部分人来说完全不影响理解，甚至很多人都不会意识到这句话的语法是错的。

2. 计算角度

自然语言技术的发展除了受语言学的制约外，在计算角度也天然存在局限。顾名思义，

计算机是计算的机器，现有的计算机都以浮点数为输入和输出，擅长执行加减乘除类计算。自然语言本身并不是浮点数，计算机为了能存储和显示自然语言，需要把自然语言中的字符转换为一个固定长度（或者变长）的二进制编码，如图 5.8 所示。

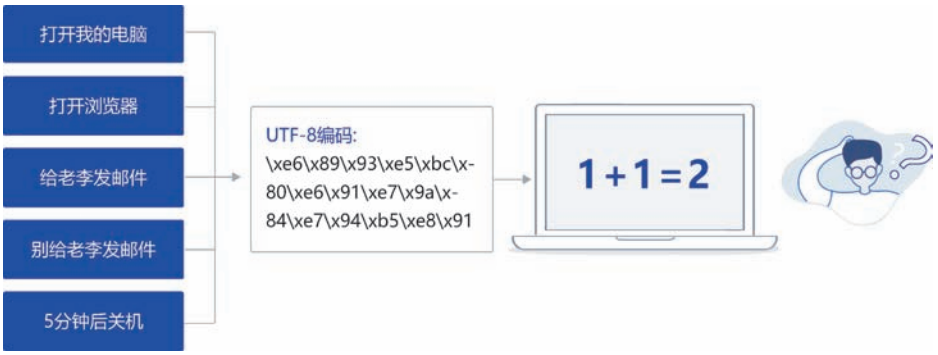


图 5.8 计算机计算自然语言流程

由于这个编码本身不是数字，对这个编码的计算往往不具备数学和物理含义。例如：把“法国”和“首都”放在一起，大多数人首先联想到的内容是“巴黎”。但是如果我们使用“法国”和“首都”的 UTF-8 编码去做加减乘除等运算，是无法轻易获取到“巴黎”的 UTF-8 编码，甚至无法获得一个有效的 UTF-8 编码。因此，如何让计算机可以有效地计算自然语言，是计算机科学家和工程师面临的巨大挑战。

此外，目前也有研究人员正在关注自然语言处理方法中的社会问题：包括自然语言处理模型中的偏见和歧视、大规模计算对环境和气候带来的影响、传统工作被取代后，人的失业和再就业问题等。

5.1.4 自然语言处理的常见任务

自然语言处理是非常复杂的领域，是人工智能中最为困难的问题之一，常见的任务如图 5.9 所示。

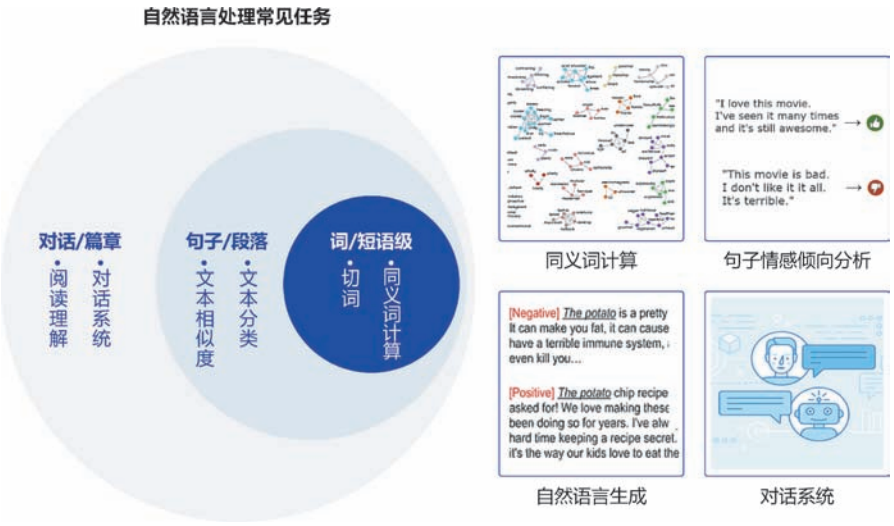


图 5.9 自然语言处理常见任务

(1) 词和短语级任务：包括切词、词性标注、命名实体识别(如“苹果很好吃”和“苹果很伟大”中的“苹果”哪个是苹果公司?)、同义词计算(如“好吃”的同义词是什么?)等以词为研究对象的任务。

(2) 句子和段落级任务：包括文本倾向性分析(如客户说：“你们公司的产品真好用!”是在夸赞还是在讽刺?)、文本相似度计算(如“我坐高铁去广州”和“我坐火车去广州”是一个意思吗?)等以句子为研究对象的任务。

(3) 对话和篇章级任务：包括机器阅读理解(如使用医药说明书回答患者的咨询问题)、对话系统(如打造一个 24 小时在线的 AI 话务员)等复杂的自然语言处理系统等。

(4) 自然语言生成：如机器翻译(如“我爱飞桨”的英文是什么?)、机器写作(以 AI 为题目写一首诗)等自然语言生成任务。

5.1.5 使用深度学习解决自然语言处理任务的套路

使用深度学习解决自然语言处理任务一般需要经历如下几个步骤，如图 5.10 所示。

前提是学习基本知识。在学习相关的知识后才能对任务有一定的了解，例如了解模型的网络结构、数据集的构成等，为后续解决任务打好基础。

(1) 处理数据。确认网络能够接收的数据形式，然后对数据进行处理。

(2) 实现网络。搭建网络的过程。

(3) 模型训练。训练模型调整参数的过程。

(4) 评估 & 上线。对训练出的模型效果进行评估，确认模型性能。

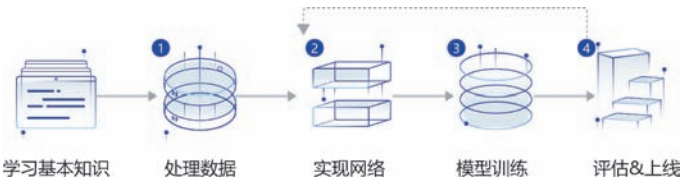


图 5.10 使用飞桨框架构建神经网络过程

5.1.6 使用飞桨探索自然语言处理

接下来，让我们一起探索几个经典的自然语言处理任务，包括：

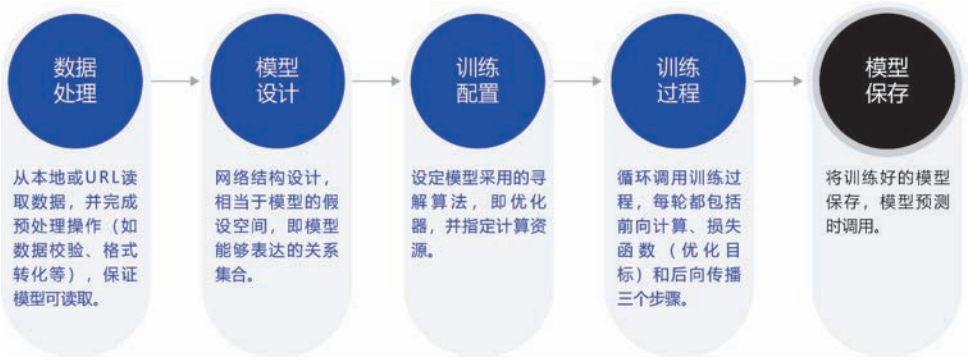
- 计算词语之间的关系(如同义词)：word2vec。
- 理解一个自然语言句子：文本分类和相似度计算。

一般来说，使用飞桨完成自然语言处理任务时，都可以遵守一个相似的套路，如图 5.11 所示。

5.1.7 作业

(1) 生活中有哪些地方使用了自然语言处理?

(2) 你希望如何应用自然语言处理?



■图 5.11 使用飞桨框架构建神经网络过程

作业提交方式

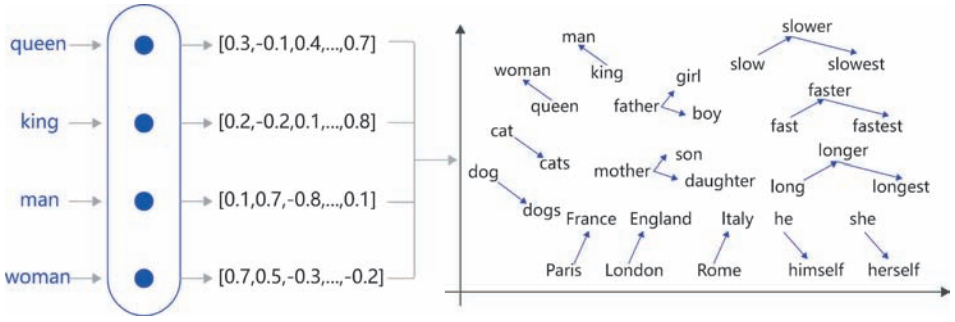
请读者扫描图书封底的二维码，在 AI Studio“零基础实践深度学习”课程中的“作业”节点下提交相关作业。

5.2 词向量 Word Embedding

5.2.1 概述

在自然语言处理任务中，词向量（Word Embedding）是表示自然语言里单词的一种方法，即把每个词都表示为一个 N 维空间内的点，即一个高维空间内的向量。通过这种方法，可实现将自然语言计算转换为向量计算。

如图 5.12 所示的词向量计算任务中，先把每个词（如 queen、king 等）转换成一个高维空间的向量，这些向量在一定意义上可以代表这个词的语义信息。再通过计算这些向量之间的距离，就可以计算出词语之间的关联关系，从而达到让计算机像计算数值一样去计算自然语言的目的。



■图 5.12 词向量计算示意图

因此，大部分词向量模型都需要回答两个问题：

（1）如何将词转换为向量？

自然语言单词是离散信号，比如“香蕉”“橘子”“水果”在我们看来就是 3 个离散的词。

我们应该如何将每个离散的单词转换为一个向量？

(2) 如何让向量具有语义信息？

比如，我们知道在很多情况下，“香蕉”和“橘子”更加相似，而“香蕉”和“句子”就没有那么相似，同时，“香蕉”和“食物”、“水果”的相似程度，可能介于“橘子”和“句子之间”。

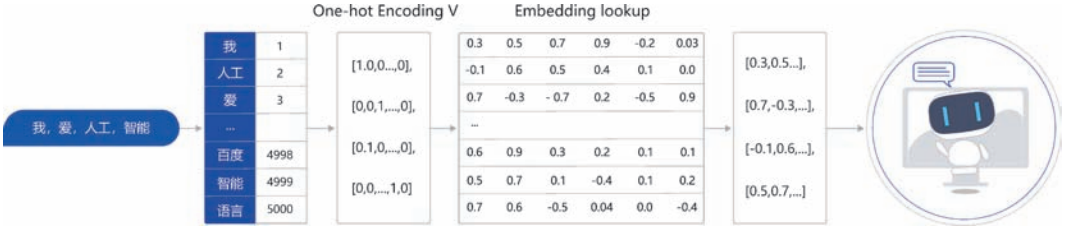
那么，我们该如何让词向量具备这样的语义信息？

5.2.2 如何将词转换为向量

自然语言单词是离散信号，比如“我”“ 爱”“人工智能”。如何将每个离散的单词转换为一个向量？通常情况下，我们可以维护一个如图 5.13 所示的查询表。表中每一行都存储了一个特定词语的向量值，每一列的第一个元素都代表着这个词本身，以便于我们进行词和向量的映射(如“我”对应的向量值为 [0.3,0.5,0.7,0.9,−0.2,0.03])。给定任何一个或者一组单词，我们都可以通过查询这个表格，实现将单词转换为向量的目的，这个查询和替换过程称之为 Embedding Lookup。

■图 5.13 词向量查询表

上述过程也可以使用一个字典数据结构实现。事实上，如果不考虑计算效率，使用字典实现上述功能是个不错的选择。然而在进行神经网络计算的过程中，需要大量的算力，常常要借助特定硬件(如 GPU)满足训练速度的需求。GPU 上所支持的计算都是以张量(Tensor)为单位展开的，因此在实际场景中，我们需要把 Embedding Lookup 的过程转换为张量计算，如图 5.14 所示。



■图 5.14 张量计算示意图

假设对于句子“我，爱，人工，智能”，把 Embedding Lookup 的过程转换为张量计算的流程如下：

(1) 通过查询字典，先将句子中的单词转换成一个 ID(通常是一个大于等于 0 的整数)，这个单词到 ID 的映射关系可以根据需求自定义(如图 5.14 中，我=>1，人工=>2，爱=>3，…)。

(2) 得到 ID 后,再将每个 ID 转换成一个固定长度的向量。假设字典的词表中有 5000 个词,那么,对于单词“我”,就可以用一个 5000 维的向量来表示。由于“我”的 ID 是 1,因此这个向量的第一个元素是 1,其他元素都是 0([1,0,0,…,0]);同样对于单词“人工”,第二个元素是 1,其他元素都是 0。用这种方式就实现了用一个向量表示一个单词。由于每个单词的向量表示都只有一个元素为 1,而其他元素为 0,因此我们称上述过程为 One-Hot Encoding。

(3) 经过 One-Hot Encoding 后,句子“我,爱,人工,智能”就被转换成了一个形状为 4×5000 的张量,记为 $\boldsymbol{V}$。在这个张量里共有 4 行、5000 列,从上到下,每一行分别代表了“我”“爱”“人工”“智能”四个单词的 One-Hot Encoding。最后,我们把这个张量 $\boldsymbol{V}$ 和另外一个稠密张量 $\boldsymbol{W}$ 相乘,其中 $\boldsymbol{W}$ 张量的形状为 5000×128(5000 表示词表大小,128 表示每个词的向量大小)。经过张量乘法,我们就得到了一个 4×128 的张量,从而完成了把单词表示成向量的目的。

5.2.3 如何让向量具有语义信息

得到每个单词的向量表示后,我们需要思考下一个问题:比如在多数情况下,“香蕉”和“橘子”更加相似,而“香蕉”和“句子”就没有那么相似;同时,“香蕉”和“食物”“水果”的相似程度可能介于“橘子”和“句子”之间。那么我们该如何让存储的词向量具备这样的语义信息呢?

我们先学习自然语言处理领域的一个小技巧。在自然语言处理研究中,科研人员通常有一个共识:使用一个单词的上下文来了解这个单词的语义,比如:

(1) “苹果手机质量不错,就是价格有点贵。”

(2) “这个苹果很好吃,非常脆。”

(3) “菠萝质量也还行,但是不如苹果支持的 APP 多。”

在上面的句子中,我们通过上下文可以推断出第一个“苹果”指的是苹果手机,第二个“苹果”指的是水果苹果,而第三个“菠萝”指的应该也是一个手机。事实上,在自然语言处理领域,使用上下文描述一个词语或者元素的语义是一个常见且有效的做法。我们可以使用同样的方式训练词向量,让这些词向量具备表示语义信息的能力。

2013 年,Mikolov 提出的经典 word2vec 算法就是通过上下文来学习语义信息。word2vec 包含两个经典模型,CBOW(Continuous Bag-of-Words)和 Skip-gram,如图 5.15 所示。

(1) CBOW:通过上下文的词向量推理中心词。

(2) Skip-gram:根据中心词推理上下文。

假设有一个句子 Pineapples are spikey and yellow,两个模型的推理方式如下:

(1) 在 CBOW 中,先在句子中选定一个中心词,并把其他词作为这个中心词的上下文。如图 5.15 CBOW 所示,把 spiked 作为中心词,把 Pineapples are and yellow 作为中心词的上下文。在学习过程中,使用上下文的词向量推理中心词,这样中心词的语义就被传递到上下文的词向量中,如 spikey=> pineapple,从而达到学习语义信息的目的。

(2) 在 Skip-gram 中,同样先选定一个中心词,并把其他词作为这个中心词的上下文。如图 5.15 Skip-gram 所示,把 spikey 作为中心词,把 Pineapples are and yellow 作为中心词的上下文。不同的是,在学习过程中,使用中心词的词向量去推理上下文,这样上下文定义

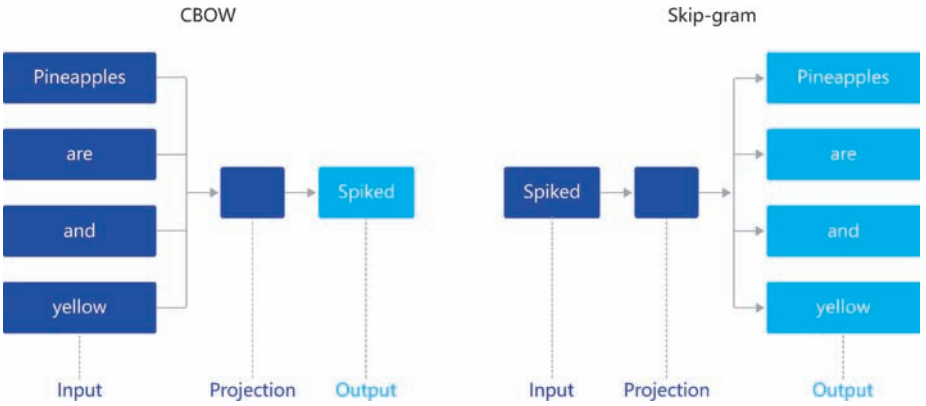


图 5.15 CBOW 和 Skip-gram 语义学习示意图

的语义被传入中心词的表示中，如"pineapple→spiked"，从而达到学习语义信息的目的。

说明：

一般来说，CBOW 比 Skip-gram 训练速度快，训练过程更加稳定，原因是 CBOW 使用上下文 average 的方式进行训练，每个训练 step 会见到更多样本。而在生僻字（出现频率低的字）处理上，skip-gram 比 CBOW 效果更好，原因是 skip-gram 不会刻意回避生僻字。

CBOW 和 Skip-gram 的算法实现

我们以这句话：Pineapples are spikey and yellow 为例分别介绍 CBOW 和 Skip-gram 的算法实现。

如图 5.16 所示，CBOW 是一个具有 3 层结构的神经网络，分别是：

（1）输入层：一个形状为 $\boldsymbol{C}\times\boldsymbol{V}$ 的 one-hot 张量，其中 C 代表上线文中词的个数，通常是一个偶数，我们假设为 4；V 表示词表大小，我们假设为 5000，该张量的每一行都是一个上下文词的 one-hot 向量表示，比如 Pineapples，are，and，yellow。

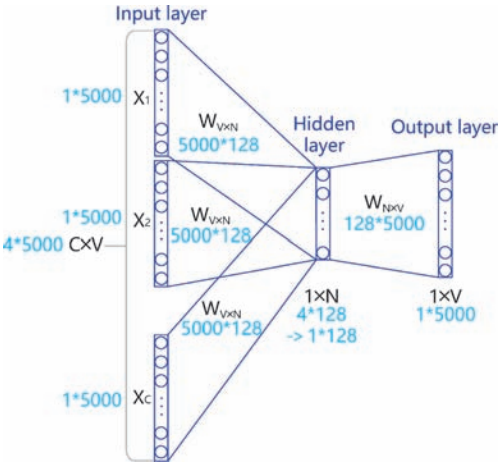
（2）隐藏层：一个形状为 $\boldsymbol{V}\times\boldsymbol{N}$ 的参数张量 $W1$，一般称为 word embedding，N 表示每个词的词向量长度，我们假设为 128。输入张量和 word embedding W1 进行矩阵乘法，就会得到一个形状为 $C\times N$ 的张量。综合考虑上下文中所有词的信息去推理中心词，因此将上下文中 C 个词相加得一个 $1\times N$ 的向量，是整个上下文的一个隐含表示。

（3）输出层：创建另一个形状为 $N\times V$ 的参数张量，将隐藏层得到的 $1\times N$ 的向量乘以该 $N\times V$ 的参数张量，得到了一个形状为 $1\times V$ 的向量。最终，$1\times V$ 的向量代表了使用上下文去推理中心词，每个候选词的打分，再经过 Softmax 函数的归一化，即得到了对中心词的推理概率：

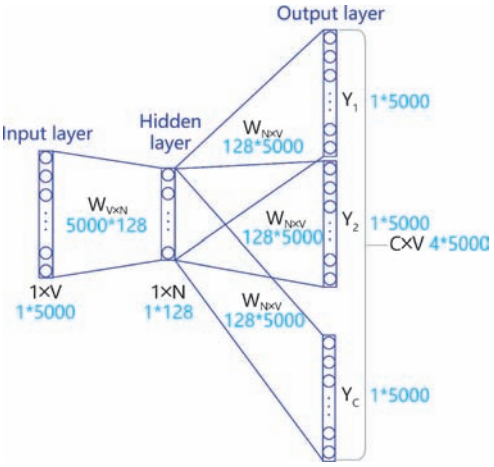
$$\text{softmax}(O_i)=\frac{\exp(O_i)}{\sum_j \exp(O_j)}$$

如图 5.17 所示，Skip-gram 是一个具有 3 层结构的神经网络，分别是：

（1）Input Layer（输入层）：接收一个 one-hot 张量 $V\in R^{1\times \text{vocab\_size}}$ 作为网络的输入，里面存储着当前句子中心词的 one-hot 表示。



■图 5.16 CBOW 的算法实现



■图 5.17 Skip-gram 算法实现

(2) Hidden Layer(隐藏层):将张量 V 乘以一个 word embedding 张量 $W_1 \in R^{\text{vocab\_size} \times \text{embed\_size}}$,并把结果作为隐藏层的输出,得到一个形状为 $R^{1 \times \text{embed\_size}}$ 的张量,里面存储着当前句子中心词的词向量。

(3) Output Layer(输出层):将隐藏层的结果乘以另一个 word embedding 张量 $W_2 \in R^{\text{embed\_size} \times \text{vocab\_size}}$,得到一个形状为 $R^{1 \times \text{vocab\_size}}$ 的张量。这个张量经过 Softmax 变换后,就得到了使用当前中心词对上下文的预测结果。根据这个 Softmax 的结果,我们就可以去训练词向量模型。

在实际操作中,使用一个滑动窗口(一般情况下,长度是奇数),从左到右开始扫描当前句子。每个扫描出来的片段被当成一个小句子,每个小句子中间的词被认为是中心词,其余的词被认为是这个中心词的上下文。

1) Skip-gram 的理想实现

使用神经网络实现 Skip-gram 中,模型接收的输入应该有两个不同的张量:

(1) 代表中心词的张量:假设我们称之为 center_words V,一般来说,这个张量是一个形状为[batch_size,vocab_size]的 one-hot 张量,表示在一个 mini-batch 中,每个中心词的 ID,对应位置为 1,其余为 0。

(2) 代表目标词的张量:目标词是指需要推理出来的上下文词,假设我们称之为 target_words T,一般来说,这个张量是一个形状为[batch_size,1]的整型张量,这个张量中的每个元素是一个[0,vocab_size-1]的值,代表目标词的 ID。

在理想情况下,我们可以使用一个简单的方式实现 Skip-gram。即把需要推理的每个目标词都当成一个标签,把 Skip-gram 当成一个大规模分类任务进行网络构建,过程如下:

(1) 声明一个形状为[vocab_size,embedding_size]的张量,作为需要学习的词向量,记为 W_0。对于给定的输入 V,使用向量乘法,将 V 乘以 W_0,这样就得到了一个形状为[batch_size,embedding_size]的张量,记为 $H = V \times W_0$。这个张量 H 就可以看成是经过词向量查表后的结果。

(2) 声明另外一个需要学习的参数 W_1,这个参数的形状为[embedding_size, vocab_

size]。将上一步得到的 H 去乘以 W_1，得到一个新的张量 $O=H\times W_1$，此时的 O 是一个形状为[batch_size，vocab_size]的张量，表示当前这个 mini-batch 中的每个中心词预测出的目标词的概率。

(3) 使用 Softmax 函数对 mini-batch 中每个中心词的预测结果做归一化，即可完成网络构建。

2) Skip-gram 的实际实现

然而在实际情况中，vocab_size 通常很大(几十万甚至几百万)，导致 W_0 和 W_1 也会非常大。对于 W_0 而言，所参与的矩阵运算并不是通过一个矩阵乘法实现，而是通过指定 ID，对参数 W_0 进行访存的方式获取。然而对 W_1 而言，仍要处理一个非常大的矩阵运算(计算过程非常缓慢，需要消耗大量的内存/显存)。为了缓解这个问题，通常采取负采样(negative_sampling)的方式来近似模拟多分类任务。此时新定义的 W_0 和 W_1 均为形状为[vocab_size，embedding_size]的张量。

假设有一个中心词 c 和一个上下文词正样本 t_p。在 Skip-gram 的理想实现里，需要最大化使用 c 推理 t_p 的概率。在使用 Softmax 学习时，需要最大化 t_p 的推理概率，同时最小化其他词表中词的推理概率。之所以计算缓慢，是因为需要对词表中的所有词都计算一遍。然而我们还可以使用另一种方法，就是随机从词表中选择几个代表词，通过最小化这几个代表词的概率，去近似最小化整体的预测概率。比如，先指定一个中心词(如“人工”)和一个目标词正样本(如“智能”)，再随机在词表中采样几个目标词负样本(如“日本”“喝茶”等)。有了这些内容，我们的 Skip-gram 模型就变成了一个二分类任务。对于目标词正样本，我们需要最大化它的预测概率；对于目标词负样本，我们需要最小化它的预测概率。通过这种方式，我们就可以完成计算加速。上述做法，我们称之为负采样。

在实现的过程中，通常会让模型接收 3 个张量输入：

(1) 代表中心词的张量：假设我们称之为 center_words V，一般来说，这个张量是一个形状为[batch_size，vocab_size]的 one-hot 张量，表示在一个 mini-batch 中每个中心词具体的 ID。

(2) 代表目标词的张量：假设我们称之为 target_words T，一般来说，这个张量同样是一个形状为[batch_size，vocab_size]的 one-hot 张量，表示在一个 mini-batch 中每个目标词具体的 ID。

(3) 代表目标词标签的张量：假设我们称之为 labels L，一般来说，这个张量是一个形状为[batch_size，1]的张量，每个元素不是 0 就是 1(0：负样本，1：正样本)。

模型训练过程如下：

(1) 用 V 去查询 W_0，用 T 去查询 W_1，分别得到两个形状为[batch_size，embedding_size]的张量，记为 H_1 和 H_2。

(2) 点乘这两个张量，最终得到一个形状为[batch_size]的张量 $O=\left[O_i=\sum_j H_0[i,j]\cdot H_1[i,j]\right]_{i=1}^{batch\_size}$。

(3) 使用 Sigmoid 函数作用在 O 上，将上述点乘的结果归一化为一个 0-1 的概率值，作为预测概率，根据标签信息 L 训练这个模型即可。

在结束模型训练之后，一般使用 W_0 作为最终要使用的词向量，可以用 W_0 提供的向量表示。通过向量点乘的方式，计算两个不同词之间的相似度。

5.3 使用飞桨实现 Skip-gram

5.3.1 概述

在飞桨中，不同深度学习模型的训练过程基本一致，流程如下：

（1）数据处理：选择需要使用的数据，并做好必要的预处理工作。

（2）网络定义：使用飞桨定义好网络结构，包括输入层，中间层，输出层，损失函数和优化算法。

（3）网络训练：将准备好的数据送入神经网络进行学习，并观察学习的过程是否正常，如损失函数值是否在降低，也可以打印一些中间步骤的结果出来等。

（4）网络评估：使用测试集合测试训练好的神经网络，看看训练效果如何。

在数据处理前，需要先加载飞桨平台（如果用户在本地使用，请确保已经安装飞桨）。

```
import io
import os
import sys
import requests
from collections import OrderedDict
import math
import random
import numpy as np
import paddle
import paddle.fluid as fluid

from paddle.fluid.dygraph.nn import Embedding
```

5.3.2 数据处理

首先，找到一个合适的语料用于训练 word2vec 模型。我们选择 text8 数据集，这个数据集里包含了大量从维基百科收集到的英文语料，我们可以通过如下代码下载数据集，下载后的文件被保存在当前目录的 text8.txt 文件内。

```
#下载语料用来训练 word2vec
def download():
    #可以从百度云服务器下载一些开源数据集(dataset.bj.bcebos.com)
    corpus_url = "https://dataset.bj.bcebos.com/word2vec/text8.txt"
    #使用 Python 的 requests 包下载数据集到本地
    web_request = requests.get(corpus_url)
    corpus = web_request.content
    #把下载后的文件存储在当前目录的 text8.txt 文件内
    with open("./text8.txt", "wb") as f:
        f.write(corpus)
    f.close()

download()
```

接下来，把下载的语料读取到程序里，并打印前 500 个字符看看语料的样子，代码如下：

```
#读取 text8 数据
def load_text8():
    with open("./text8.txt", "r") as f:
        corpus = f.read().strip("\n")
    f.close()
    return corpus

corpus = load_text8()
#打印前 500 个字符，简要看一下这个语料的样子
print(corpus[:500])
```

一般来说，在自然语言处理中，需要先对语料进行切词。对于英文来说，可以比较简单地直接使用空格进行切词，代码如下：

```
#对语料进行预处理(分词)
def data_preprocess(corpus):
    #由于英文单词出现在句首的时候经常要大写，所以我们把所有英文字符都转换为小写，
    #以便对语料进行归一化处理(Apple vs apple 等)
    corpus = corpus.strip().lower()
    corpus = corpus.split(" ")
    return corpus

corpus = data_preprocess(corpus)
print(corpus[:50])
```

在经过切词后，需要对语料进行统计，为每个词构造 ID。一般来说，可以根据每个词在语料中出现的频次构造 ID，频次越高，ID 越小，便于对词典进行管理。代码如下：

```
#构造词典，统计每个词的频率，并根据频率将每个词转换为一个整数 id
def build_dict(corpus):
    #首先统计每个不同词的频率(出现的次数)，使用一个词典记录
    word_freq_dict = dict()
    for word in corpus:
        if word not in word_freq_dict:
            word_freq_dict[word] = 0
        word_freq_dict[word] += 1

    #将这个词典中的词，按照出现次数排序，出现次数越高，排序越靠前
    #一般来说，出现频率高的高频词往往是：I，the，you 这种代词，而出现频率低的词，往往是一些
名词，如：nlp
    word_freq_dict = sorted(word_freq_dict.items(), key = lambda x:x[1], reverse = True)

    #构造 3 个不同的词典，分别存储，
    #每个词到 id 的映射关系：word2id_dict
    #每个 id 出现的频率：word2id_freq
    #每个 id 到词的映射关系：id2word_dict
    word2id_dict = dict()
    word2id_freq = dict()
    id2word_dict = dict()
```

```
        #按照频率,从高到低,开始遍历每个单词,并为这个单词构造一个独一无二的 id
        for word, freq in word_freq_dict:
            curr_id = len(word2id_dict)
            word2id_dict[word] = curr_id
            word2id_freq[word2id_dict[word]] = freq
            id2word_dict[curr_id] = word

        return word2id_freq, word2id_dict, id2word_dict

    word2id_freq, word2id_dict, id2word_dict = build_dict(corpus)
    vocab_size = len(word2id_freq)
    print("there are totoally %d different words in the corpus" % vocab_size)
    for _, (word, word_id) in zip(range(50), word2id_dict.items()):
        print("word %s, its id %d, its word freq %d" % (word, word_id, word2id_freq[word_id]))
```

得到 word2id 词典后,我们还需要进一步处理原始语料,把每个词替换成对应的 ID,便于神经网络进行处理,代码如下:

```
    #把语料转换为 id 序列
    def convert_corpus_to_id(corpus, word2id_dict):
        #使用一个循环,将语料中的每个词替换成对应的 id,以便于神经网络进行处理
        corpus = [word2id_dict[word] for word in corpus]
        return corpus

    corpus = convert_corpus_to_id(corpus, word2id_dict)
    print("%d tokens in the corpus" % len(corpus))
    print(corpus[:50])
```

接下来,需要使用二次采样法处理原始文本。二次采样法的主要思想是降低高频词在语料中出现的频次,降低的方法是随机将高频的词抛弃,频率越高,被抛弃的概率就越高,频率越低,被抛弃的概率就越低,这样像标点符号或冠词这样的高频词就会被抛弃,从而优化整个词表的词向量训练效果,代码如下:

```
    #使用二次采样算法(subsampling)处理语料,强化训练效果
    def subsampling(corpus, word2id_freq):

        #这个 discard 函数决定了一个词会不会被替换,这个函数是具有随机性的,每次调用结果不同
        #如果一个词的频率很大,那么它被遗弃的概率就很大
        def discard(word_id):
            return random.uniform(0, 1) <1 - math.sqrt(
                1e-4 / word2id_freq[word_id] * len(corpus))

        corpus = [word for word in corpus if not discard(word)]
        return corpus

    corpus = subsampling(corpus, word2id_freq)
    print("%d tokens in the corpus" % len(corpus))
    print(corpus[:50])
```

在完成语料数据预处理之后,需要构造训练数据。根据上面的描述,我们需要使用一个

滑动窗口对语料从左到右扫描，在每个窗口内，中心词需要预测它的上下文，并形成训练数据。

在实际操作中，由于词表往往很大（50 000，100 000 等），对大词表的一些矩阵运算（如 Softmax）需要消耗巨大的资源，因此可以通过负采样的方式模拟 Softmax 的结果，代码实现如下。

（1）给定一个中心词和一个需要预测的上下文词，把这个上下文词作为正样本。

（2）通过词表随机采样的方式，选择若干个负样本。

（3）把一个大规模分类问题转化为一个 2 分类问题，通过这种方式优化计算速度。

```
#构造数据，准备模型训练
#max_window_size 代表了最大的 window_size 的大小，程序会根据 max_window_size 从左到右扫描整个语料
#negative_sample_num 代表了对于每个正样本，我们需要随机采样多少负样本用于训练，
#一般来说，negative_sample_num 的值越大，训练效果越稳定，但是训练速度越慢
def build_data(corpus, word2id_dict, word2id_freq, max_window_size = 3, negative_sample_num = 4):
    #使用一个 list 存储处理好的数据
    dataset = []
    #从左到右，开始枚举每个中心点的位置
    for center_word_idx in range(len(corpus)):
        #以 max_window_size 为上限，随机采样一个 window_size，这样会使得训练更加稳定
        window_size = random.randint(1, max_window_size)
        #当前的中心词就是 center_word_idx 所指向的词
        center_word = corpus[center_word_idx]

        #以当前中心词为中心，左右两侧在 window_size 内的词都可以看成是正样本
        positive_word_range = (max(0, center_word_idx - window_size), min(len(corpus) - 1, center_word_idx + window_size))
        positive_word_candidates = [corpus[idx] for idx in range(positive_word_range[0], positive_word_range[1] + 1) if idx != center_word_idx]

        #对于每个正样本来说，随机采样 negative_sample_num 个负样本，用于训练
        for positive_word in positive_word_candidates:
            #首先把（中心词，正样本，label = 1）的三元组数据放入 dataset 中，
            #这里 label = 1 表示这个样本是个正样本
            dataset.append((center_word, positive_word, 1))

            #开始负采样
            i = 0
            while i <negative_sample_num:
                negative_word_candidate = random.randint(0, vocab_size - 1)

                if negative_word_candidate not in positive_word_candidates:
                    #把（中心词，正样本，label = 0）的三元组数据放入 dataset 中，
                    #这里 label = 0 表示这个样本是个负样本
                    dataset.append((center_word, negative_word_candidate, 0))
                    i += 1
    return dataset

dataset = build_data(corpus, word2id_dict, word2id_freq)
for _, (center_word, target_word, label) in zip(range(50), dataset):
```

```
        print("center_word %s, target %s, label %d" % (id2word_dict[center_word],
                                                       id2word_dict[target_word], label))
```

训练数据准备好后，把训练数据都组装成 mini-batch，并准备输入到网络中进行训练，代码如下：

```
#构造 mini - batch,准备对模型进行训练
#我们将不同类型的数据放到不同的张量里,便于神经网络进行处理
#并通过 numpy 的 array 函数,构造出不同的张量来,并把这些张量送入神经网络中进行训练
def build_batch(dataset, batch_size, epoch_num):

    #center_word_batch 缓存 batch_size 个中心词
    center_word_batch = []
    #target_word_batch 缓存 batch_size 个目标词(可以是正样本或者负样本)
    target_word_batch = []
    #label_batch 缓存了 batch_size 个 0 或 1 的标签,用于模型训练
    label_batch = []

    for epoch in range(epoch_num):
        #每次开启一个新 epoch 之前,都对数据进行一次随机打乱,提高训练效果
        random.shuffle(dataset)

        for center_word, target_word, label in dataset:
            #遍历 dataset 中的每个样本,并将这些数据送到不同的张量里
            center_word_batch.append([center_word])
            target_word_batch.append([target_word])
            label_batch.append(label)

            #当样本积攒到一个 batch_size 后,我们把数据都返回回来
            #在这里我们使用 numpy 的 array 函数把 list 封装成张量
            #并使用 Python 的迭代器机制,将数据 yield 出来
            #使用迭代器的好处是可以节省内存
            if len(center_word_batch) == batch_size:
                yield np.array(center_word_batch).astype("int64"), \
                    np.array(target_word_batch).astype("int64"), \
                    np.array(label_batch).astype("float32")
                center_word_batch = []
                target_word_batch = []
                label_batch = []

    if len(center_word_batch) > 0:
        yield np.array(center_word_batch).astype("int64"), \
            np.array(target_word_batch).astype("int64"), \
            np.array(label_batch).astype("float32")

for _, batch in zip(range(10), build_batch(dataset, 128, 3)):
    print(batch)
```

5.3.3 网络定义

定义 Skip-gram 的网络结构，用于模型训练。在飞桨动态图中，对于任意网络，都需要

定义一个继承自 fluid.dygraph.Layer 的类来搭建网络结构、参数等数据的声明。同时需要在 forward 函数中定义网络的计算逻辑。值得注意的是，我们仅需要定义网络的前向计算逻辑，飞桨会自动完成神经网络的反向计算，代码如下：

```
#定义 Skip-gram 训练网络结构
#这里我们使用的是 Paddlepaddle 的 1.8.0 版本
#一般来说，在使用 fluid 训练的时候，我们需要通过一个类来定义网络结构，这个类继承了 fluid.dygraph.Layer
class SkipGram(fluid.dygraph.Layer):
    def __init__(self, vocab_size, embedding_size, init_scale=0.1):
        #vocab_size 定义了这个 skipgram 模型的词表大小
        #embedding_size 定义了词向量的维度是多少
        #init_scale 定义了词向量初始化的范围，一般来说，比较小的初始化范围有助于模型训练
        super(SkipGram, self).__init__()
        self.vocab_size = vocab_size
        self.embedding_size = embedding_size

        #使用 paddle.fluid.dygraph 提供的 Embedding 函数，构造一个词向量参数
        #这个参数的大小为：[self.vocab_size, self.embedding_size]
        #数据类型为：float32
        #这个参数的名称为：embedding_para
        #这个参数的初始化方式为在[-init_scale, init_scale]区间进行均匀采样
        self.embedding = Embedding(
            size=[self.vocab_size, self.embedding_size],
            dtype='float32',
            param_attr=fluid.ParamAttr(
                name='embedding_para',
                initializer=fluid.initializer.UniformInitializer(
                    low=-0.5/embedding_size, high=0.5/embedding_size)))

        #使用 paddle.fluid.dygraph 提供的 Embedding 函数，构造另外一个词向量参数
        #这个参数的大小为：[self.vocab_size, self.embedding_size]
        #数据类型为：float32
        #这个参数的名称为：embedding_para_out
        #这个参数的初始化方式为在[-init_scale, init_scale]区间进行均匀采样
        #跟上面不同的是，这个参数的名称跟上面不同，因此，
        #embedding_para_out 和 embedding_para 虽然有相同的形状，但是权重不共享
        self.embedding_out = Embedding(
            size=[self.vocab_size, self.embedding_size],
            dtype='float32',
            param_attr=fluid.ParamAttr(
                name='embedding_out_para',
                initializer=fluid.initializer.UniformInitializer(
                    low=-0.5/embedding_size, high=0.5/embedding_size)))

    #定义网络的前向计算逻辑
    #center_words 是一个 tensor(mini-batch)，表示中心词
    #target_words 是一个 tensor(mini-batch)，表示目标词
    #label 是一个张量(mini-batch)，表示这个词是正样本还是负样本(用 0 或 1 表示)
    #用于在训练中计算这个张量中对应词的同义词，用于观察模型的训练效果
    def forward(self, center_words, target_words, label):
```

```
        #首先,通过 embedding_para(self.embedding)参数,将 mini-batch 中的词转换为词向量
        #这里 center_words 和 eval_words_emb 查询的是一个相同的参数
        #而 target_words_emb 查询的是另一个参数
        center_words_emb = self.embedding(center_words)
        target_words_emb = self.embedding_out(target_words)

        #center_words_emb = [batch_size, embedding_size]
        #target_words_emb = [batch_size, embedding_size]
        #我们通过点乘的方式计算中心词到目标词的输出概率,并通过 Sigmoid 函数估计这个词
是正样本还是负样本的概率.
        word_sim = fluid.layers.elementwise_mul(center_words_emb, target_words_emb)
        word_sim = fluid.layers.reduce_sum(word_sim, dim = -1)
        word_sim = fluid.layers.reshape(word_sim, shape=[-1])
        pred = fluid.layers.sigmoid(word_sim)

        #通过估计的输出概率定义损失函数,注意我们使用的是 sigmoid_cross_entropy_with_
logits 函数
        #将 Sigmoid 计算和 cross entropy 合并成一步计算可以更好的优化,所以输入的是 word_
sim,而不是 pred

        loss = fluid.layers.sigmoid_cross_entropy_with_logits(word_sim, label)
        loss = fluid.layers.reduce_mean(loss)

        #返回前向计算的结果,飞桨会通过 backward 函数自动计算出反向结果.
        return pred, loss
```

5.3.4 网络训练

完成网络定义后,就可以启动模型训练。我们定义每隔 100 步打印一次 *Loss* 值,以确保当前的网络是正常收敛的。同时,我们每隔 10 000 步观察一下 Skip-gram 计算出来的同义词(使用 embedding 的乘积),可视化网络训练效果,代码如下:

```
#开始训练,定义一些训练过程中需要使用的超参数
batch_size = 512
epoch_num = 3
embedding_size = 200
step = 0
learning_rate = 0.001

#定义一个使用 word-embedding 查询同义词的函数
#这个函数 query_token 是要查询的词,k 表示要返回多少个最相似的词,embed 是我们学习到的
word-embedding 参数
#我们通过计算不同词之间的 cosine 距离,来衡量词和词的相似度
#具体实现如下,x 代表要查询词的 Embedding,Embedding 参数矩阵 W 代表所有词的 Embedding
#两者计算 Cos 得出所有词对查询词的相似度得分向量,排序取 top_k 放入 indices 列表
def get_similar_tokens(query_token, k, embed):
    W = embed.numpy()
    x = W[word2id_dict[query_token]]
    cos = np.dot(W, x) / np.sqrt(np.sum(W * W, axis=1) * np.sum(x * x) + 1e-9)
    flat = cos.flatten()
```

```
        indices = np.argpartition(flat, -k)[-k:]
        indices = indices[np.argsort(-flat[indices])]
        for i in indices:
            print('for word %s, the similar word is %s' % (query_token, str(id2word_dict[i])))

    #将模型放到 GPU 上训练(fluid.CUDAPlace(0)),如果需要指定 CPU,则需要改为 fluid.CPUPlace()
    with fluid.dygraph.guard(fluid.CUDAPlace(0)):
        #通过我们定义的 SkipGram 类,来构造一个 Skip-gram 模型网络
        skip_gram_model = SkipGram(vocab_size, embedding_size)
        #构造训练这个网络的优化器
        adam = fluid.optimizer.AdamOptimizer(learning_rate=learning_rate, parameter_list =
    skip_gram_model.parameters())

        #使用 build_batch 函数,以 mini-batch 为单位,遍历训练数据,并训练网络
        for center_words, target_words, label in build_batch(
            dataset, batch_size, epoch_num):
            #使用 fluid.dygraph.to_variable 函数,将一个 numpy 的张量,转换为飞桨可计算的张量
            center_words_var = fluid.dygraph.to_variable(center_words)
            target_words_var = fluid.dygraph.to_variable(target_words)
            label_var = fluid.dygraph.to_variable(label)

            #将转换后的张量送入飞桨中,进行一次前向计算,并得到计算结果
            pred, loss = skip_gram_model(
                center_words_var, target_words_var, label_var)

            #通过 backward 函数,让程序自动完成反向计算
            loss.backward()
            #通过 minimize 函数,让程序根据 loss,完成一步对参数的优化更新
            adam.minimize(loss)
            #使用 clear_gradients 函数清空模型中的梯度,以便于下一个 mini-batch 进行更新
            skip_gram_model.clear_gradients()

            #每经过 100 个 mini-batch,打印一次当前的 loss,看看 loss 是否在稳定下降
            step += 1
            if step % 100 == 0:
                print("step %d, loss %.3f" % (step, loss.numpy()[0]))

            #经过 10000 个 mini-batch,打印一次模型对 eval_words 中的 10 个词计算的同义词
            #这里我们使用词和词之间的向量点积作为衡量相似度的方法
            #我们只打印了 5 个最相似的词
            if step % 10000 == 0:
                get_similar_tokens('one', 5, skip_gram_model.embedding.weight)
                get_similar_tokens('she', 5, skip_gram_model.embedding.weight)
                get_similar_tokens('chip', 5, skip_gram_model.embedding.weight)
```

从打印结果可以看到,经过一定步骤的训练,Loss 逐渐下降并趋于稳定。同时也可以发现 Skip-gram 模型可以学习到一些有趣的语言现象,比如:跟 who 比较接近的词是 whose、he、she、him、himself。

5.3.5 词向量的有趣使用

在使用 word2vec 模型的过程中,研究人员发现了一些有趣的现象。比如当得到整个词

表的 word embedding 之后，对任意词都可以基于向量乘法计算跟这个词最接近的词。我们会发现，word2vec 模型可以自动学习一些同义词关系，如：

```
Top 5 words closest to "beijing" are:
1. newyork
2. paris
3. tokyo
4. berlin
5. seoul

...

Top 5 words closest to "apple" are:
1. banana
2. pineapple
3. huawei
4. peach
5. orange
```

除此以外，研究人员还发现可以使用加减法完成一些基于语言的逻辑推理，如：

```
Top 1 words closest to "king - man + woman" are
1. queen

...

Top 1 words closest to "captial - china + america" are
1. washington
```

还有更多有趣的例子，赶快使用飞桨尝试实现一下吧。

5.3.6 作业

（1）如何使用飞桨实现 CBOW 算法？

（2）有些词天然具有歧义，比如“苹果”，在学习 word2vec 的时候，如何解决和区分歧义性词？

（3）如何构造一个自然语言句子的向量表示？

作业提交方式

请读者扫描图书封底的二维码，在 AI Studio“零基础实践深度学习”课程中的“作业”节点下提交相关作业。