

PyTorch 自然语言处理

译者：[Yif Du](#)

协议：[CC BY-NC-ND 4.0](#)

所有模型都是错的,但其中一些是有用的。

本书旨在为新人提供自然语言处理（NLP）和深度学习，以涵盖这两个领域的重要主题。这两个主题领域都呈指数级增长。对于一本介绍深度学习和强调实施的 NLP 的书，本书占据了重要的中间地带。在写这本书时，我们不得不对哪些材料遗漏做出艰难的，有时甚至是不舒服的选择。对于初学者，我们希望本书能够为基础知识提供强有力的基础，并可以瞥见可能的内容。特别是机器学习和深度学习是一种经验学科，而不是智力科学。我们希望每章中慷慨的端到端代码示例邀请您参与这一经历。当我们开始编写本书时，我们从 PyTorch 0.2 开始。每个 PyTorch 更新从 0.2 到 0.4 修改了示例。PyTorch 1.0 将于本书出版时发布。本书中的代码示例符合 PyTorch 0.4，它应该与即将发布的 PyTorch 1.0 版本一样工作。关于本书风格的注释。我们在大多数地方都故意避免使用数学；并不是因为深度学习数学特别困难（事实并非如此），而是因为它在许多情况下分散了本书主要目标的注意力——增强初学者的能力。在许多情况下，无论是在代码还是文本方面，我们都有类似的动机，我们倾向于对简洁性进行阐述。高级读者和有经验的程序员可以找到方法来收紧代码等等，但我们的选择是尽可能明确，以便覆盖我们想要达到的大多数受众。

- [在线阅读](#)
- [在线阅读 \(Gitee\)](#)
- [ApacheCN 学习资源](#)
- [ApacheCN 面试求职群 724187166](#)
- [代码地址](#)

贡献指南

本项目需要校对，欢迎大家提交 Pull Request。

请您勇敢地去翻译和改进翻译。虽然我们追求卓越，但我们并不要求您做到十全十美，因此请不要担心因为翻译上犯错——在大部分情况下，我们的服务器已经记录所有的翻译，因此您不必担心会因为您的失误遭到无法挽回的破坏。（改编自维基百科）

联系方式

负责人

- [飞龙](#): 562826179

其他

- 在我们的 [apacheecn/nlp-pytorch-zh](#) github 上提 issue.
- 发邮件到 Email: apacheecn@163.com.
- 在我们的 [组织学习交流](#)群 中联系群主/管理员即可.

下载

Docker

```
docker pull apacheecn0/nlp-pytorch-zh
docker run -tid -p <port>:80 apacheecn0/nlp-pytorch-zh
# 访问 http://localhost:{port} 查看文档
```

PYPI

```
pip install nlp-pytorch-zh
nlp-pytorch-zh <port>
# 访问 http://localhost:{port} 查看文档
```

NPM

```
npm install -g nlp-pytorch-zh
nlp-pytorch-zh <port>
# 访问 http://localhost:{port} 查看文档
```

赞助我们

微信	支付宝
	

一、基础介绍

本文标题: [Natural-Language-Processing-with-PyTorch \(一\)](#)

文章作者: [Yif Du](#)

发布时间: 2018 年 12 月 17 日 - 09:12

最后更新: 2019 年 02 月 16 日 - 21:02

原始链接: [http://yifdu.github.io/2018/12/17/Natural-Language-Processing-with-PyTorch \(一\) /](http://yifdu.github.io/2018/12/17/Natural-Language-Processing-with-PyTorch (一) /)

许可协议: [署名-非商业性使用-禁止演绎 4.0 国际](#) 转载请保留原文链接及作者。

像 Echo (Alexa)、Siri 和谷歌翻译这样的家喻户晓的产品名称至少有一个共同点。它们都是自然语言处理 (NLP) 应用的产物, NLP 是本书的两个主要主题之一。NLP 是一套运用统计方法的技术, 无论是否有语言学的洞见, 为了解决现实世界的任务而理解文本。这种对文本的“理解”主要是通过将文本转换为可用的计算表示, 这些计算表示是离散或连续的组合结构, 如向量或张量、图形和树。

从数据 (本例中为文本) 中学习适合于任务的表示形式是机器学习的主题。应用机器学习的文本数据有超过三十年的历史,但最近 (2008 年至 2010 年开始) [1] 一组机器学习技术,被称为深度学习,继续发展和证明非常有效的各种人工智能 (AI) 在 NLP 中的任务,演讲,和计算机视觉。深度学习是我们要讲的另一个主题;因此,本书是关于 NLP 和深度学习的研究。

简单地说,深度学习使人们能够使用一种称为计算图和数字优化技术的抽象概念有效地从数据中学习表示。这就是深度学习和计算图的成功之处,像谷歌、Facebook 和 Amazon 这样的大型技术公司已经发布了基于它们的计算图框架和库的实现,以捕捉研究人员和工程师的思维。在本书中,我们考虑 PyTorch,一个越来越流行的基于 python 的计算图框架库来实现深度学习算法。在本章中,我们将解释什么是计算图,以及我们选择使用 PyTorch 作为框架。机器学习和深度学习的领域是广阔的。在这一章,在本书的大部分时间里,我们主要考虑的是所谓的监督学习;也就是说,使用标记的训练示例进行学习。我们解释了监督学习范式,这将成为本书的基础。如果到目前为止您还不熟悉其中的许多术语,那么您是对的。这一章,以及未来的章节,不仅澄清了这一点,而且深入研究了它们。如果您已经熟悉这里提到的一些术语和概念,我们仍然鼓励您遵循以下两个原因:为本书其余部分建立一个共享的词汇表,以及填补理解未来章节所需的任何空白。

本章的目标是:

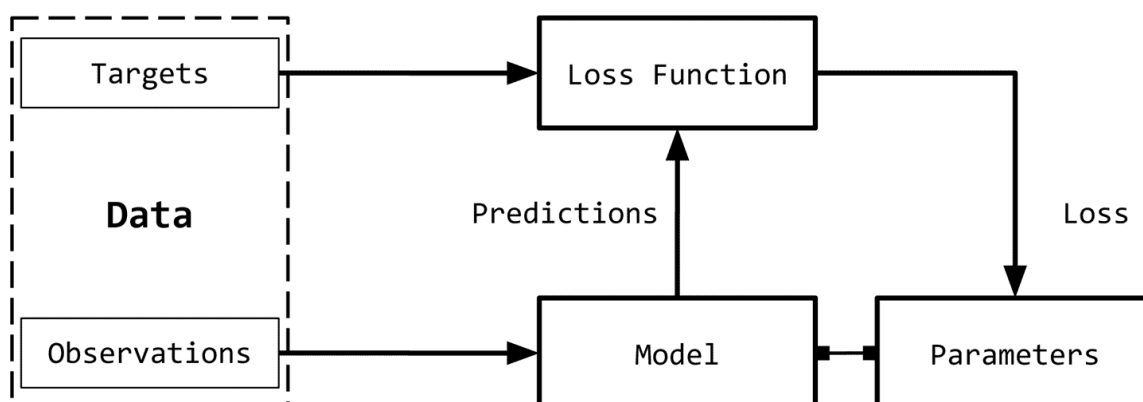
- 发展对监督学习范式的清晰理解,理解术语,并发展一个概念框架来处理未来章节的学习任务

- 学习如何为学习任务的输入编码
- 理解什么是计算图
- 掌握 PyTorch 的基本知识

让我们开始吧!

监督学习范式

机器学习中的监督，或者简单的监督学习，是指将目标（被预测的内容）的真实情况用于观察（输入）的情况。例如，在文档分类中，目标是一个分类标签，观察（输入）是一个文档。例如，在机器翻译中，观察（输入）是一种语言的句子，目标是另一种语言的句子。通过对输入数据的理解，我们在图 1-1 中演示了监督学习范式。



我们可以将监督学习范式分解为六个主要概念，如图 1-1 所示：观察：观察是我们想要预测的东西。我们用 x 表示观察值。我们有时把观察值称为“输入”。目标：目标是与观察相对应的标签。它通常是被预言的事情。按照机器学习/深度学习中的标准符号，我们用 y 表示这些。有时，这被称为真实情况。模型：模型是一个数学表达式或函数，它接受一个观察值 x ，并预测其目标标签的值。参数：有时也称为权重，这些参数化模型。标准使用的符号 w （权重）或 w_{hat} 。预测：预测，也称为估计，是模型在给定观测值的情况下所猜测目标的值。我们用一个 hat 表示这些。所以，目标 y 的预测用 y_{hat} 来表示。损失函数：损失函数是比较预测与训练数据中观测目标之间的距离的函数。给定一个目标及其预测，损失函数将分配一个称为损失的标量实值。损失值越低，模型对目标的预测效果越好。我们用 L 表示损失函数。

虽然在 NLP /深度学习建模或编写本书时，这在数学上并不是正式有效，但我们将正式重述监督学习范例，以便为该领域的新读者提供标准术语，以便他们拥有熟悉 arXiv 研究论文中的符号和写作风格。

考虑一个数据集 $D=\{X[i],y[i]\}$, $i=1..n$ ，有 n 个例子。给定这个数据集，我们想要学习一个由权重 w 参数化的函数（模型） f ，也就是说，我们对 f 的结构做一个假设，给定这个结构，权重 w 的学习值将充分表征模型。对于一个给定的输入 x ，模型预测 y_{hat} 作为目标： $y_{\text{hat}} = f(x;w)$ 在监督学习中，对于训练例子，我们知道观察的真正目标 y 。这个实例的损失将为

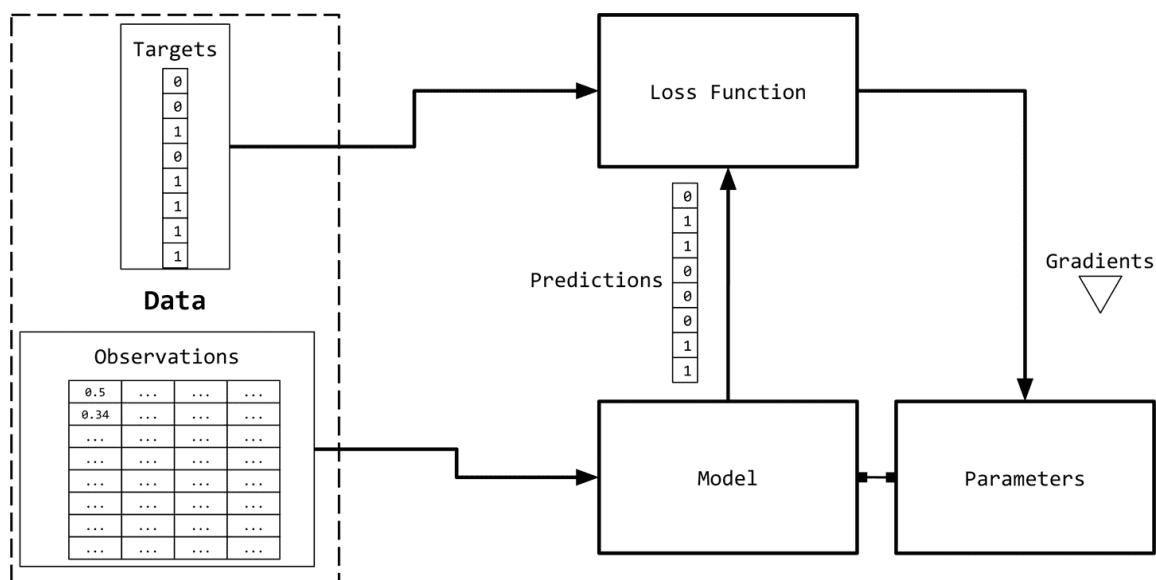
$L(y, \hat{y})$ 。然后，监督学习就变成了一个寻找最优参数/权值 w 的过程，从而使所有 n 个例子的累积损失最小化。

利用（随机）梯度下降法进行训练 监督学习的目标是为给定的数据集选择参数值，使损失函数最小化。换句话说，这等价于在方程中求根。我们知道梯度下降法是一种常见的求方程根的方法。回忆一下，在传统的梯度下降法中，我们对根（参数）的一些初值进行猜测，并迭代更新这些参数，直到目标函数（损失函数）的计算值低于可接受阈值（即收敛准则）。对于大型数据集，由于内存限制，在整个数据集上实现传统的梯度下降通常是不可能的，而且由于计算开销，速度非常慢。相反，通常采用一种近似的梯度下降称为随机梯度下降（SGD）。在随机情况下，数据点或数据点的子集是随机选择的，并计算该子集的梯度。当使用单个数据点时，这种方法称为纯 SGD，当使用（多个）数据点的子集时，我们将其称为小型批量 SGD。通常情况下，“纯”和“小型批量”这两个词在根据上下文变得清晰时就会被删除。在实际应用中，很少使用纯 SGD，因为它会由于有噪声的更新而导致非常慢的收敛。一般 SGD 算法有不同的变体，都是为了更快的收敛。在后面的章节中，我们将探讨这些变体中的一些，以及如何使用渐变来更新参数。这种迭代更新参数的过程称为反向传播。反向传播的每个步骤（又名周期）由向前传递和向后传递组成。向前传递用参数的当前值计算输入并计算损失函数。反向传递使用损失梯度更新参数。

请注意，到目前为止，这里没有什么是专门针对深度学习或神经网络的。图 1-1 中箭头的方向表示训练系统时数据的“流”。关于训练和“计算图”中“流”的概念，我们还有更多要说的，但首先，让我们看看如何用数字表示 NLP 问题中的输入和目标，这样我们就可以训练模型并预测结果。

观测和目标编码

我们需要用数字表示观测值（文本），以便与机器学习算法一起使用。图 1-2 给出了一个可视化的描述。



表示文本的一种简单方法是用数字向量表示。有无数种方法可以执行这种映射/表示。事实上，本书的大部分内容都致力于从数据中学习此类任务表示。然而，我们从基于启发式的一些简单的基于计数的表示开始。虽然简单，但是它们非常强大，或者可以作为更丰富的表示学习的起点。所有这些基于计数的表示都是从一个固定维数的向量开始的。

单热表示

顾名思义，单热表示从一个零向量开始，如果单词出现在句子或文档中，则将向量中的相应条目设置为 1。考虑下面两句话。

```
Time flies like an arrow.
Fruit flies like a banana.
```

对句子进行标记，忽略标点符号，并将所有的单词都用小写字母表示，就会得到一个大小为 8 的词汇表: {time, fruit, flies, like, a, an, arrow, banana}。所以，我们可以用一个八维的单热向量来表示每个单词。在本书中，我们使用 $1[w]$ 表示标记/单词 w 的单热表示。

对于短语、句子或文档，压缩的单热表示仅仅是其组成词的逻辑或的单热表示。使用图 1-3 所示的编码，短语 like a banana 的单热表示将是一个 3×8 矩阵，其中的列是 8 维的单热向量。通常还会看到“折叠”或二进制编码，其中文本/短语由词汇表长度的向量表示，用 0 和 1 表示单词的缺失或存在。like a banana 的二进制编码是: [0,0,0,1,1,0,0,1]。

	time	fruit	flies	like	a	an	arrow	banana
1 time	1	0	0	0	0	0	0	0
1 fruit	0	1	0	0	0	0	0	0
1 flies	0	0	1	0	0	0	0	0
1 like	0	0	0	1	0	0	0	0
1 a	0	0	0	0	1	0	0	0
1 an	0	0	0	0	0	1	0	0
1 arrow	0	0	0	0	0	0	1	0
1 banana	0	0	0	0	0	0	0	1

注意：在这一点上，如果你觉得我们把 `flies` 的两种不同的意思（或感觉）搞混了，恭喜你，聪明的读者！语言充满了歧义，但是我们仍然可以通过极其简化的假设来构建有用的解决方案。学习特定于意义的表示是可能的，但是我们现在做得有些超前了。

尽管对于本书中的输入，我们很少使用除了单热表示之外的其他表示，但是由于 NLP 中受欢迎、历史原因和完成目的，我们现在介绍术语频率（TF）和术语频率反转文档频率（TF-idf）表示。这些表示在信息检索（IR）中有着悠久的历史，甚至在今天的生产 NLP 系统中也得到了广泛的应用。（翻译有不足）

TF 表示

短语、句子或文档的 TF 表示仅仅是构成词的单热的总和。为了继续我们愚蠢的示例，使用前面提到的单热编码，`Fruit flies like time flies a fruit` 这句话具有以下 TF 表示：

`[1,2,2,1,1,1,0,0]`。注意，每个条目是句子（语料库）中出现相应单词的次数的计数。我们用 $TF(w)$ 表示一个单词的 TF。

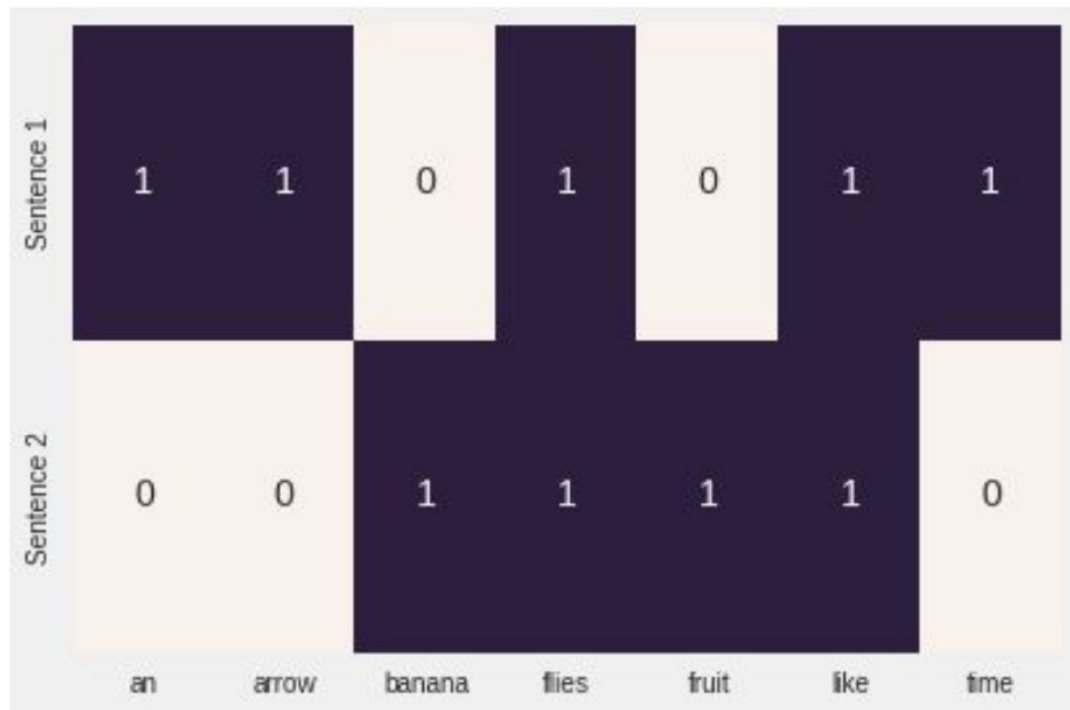
示例 1-1：使用 sklearn 生成“塌陷的”单热或二进制表示

```

from sklearn.feature_extraction.text import CountVectorizer
import seaborn as sns

corpus = ['Time flies flies like an arrow.',
          'Fruit flies like a banana.']
one_hot_vectorizer = CountVectorizer(binary=True)
one_hot = one_hot_vectorizer.fit_transform(corpus).toarray()
sns.heatmap(one_hot, annot=True,
            cbar=False, xticklabels=vocab,
            yticklabels=['Sentence 1', 'Sentence 2'])

```



折叠的单热是一个向量中有多个 1 的单热

TF-IDF 表示

考虑一组专利文件。您可能希望它们中的大多数都有诸如 `claim`、`system`、`method`、`procedure` 等单词，并且经常重复多次。TF 表示对更频繁的单词进行加权。然而，像 `claim` 这样的常用词并不能增加我们对具体专利的理解。相反，如果 `tetrafluoroethylene` 这样罕见的词出现的频率较低，但很可能表明专利文件的性质，我们希望在我们的表述中赋予它更大的权重。反文档频率（IDF）是一种启发式算法，可以精确地做到这一点。

IDF 表示惩罚常见的符号，并奖励向量表示中的罕见符号。 符号 w 的 $IDF(w)$ 对语料库的定义为

其中 $n[w]$ 是包含单词 w 的文档数量， N 是文档总数。TF-IDF 分数就是 $TF(w) * IDF(w)$ 的乘积。首先，请注意在所有文档（例如， $n[w] = N$ ）， $IDF(w)$ 为 0，TF-IDF 得分为 0，完全惩罚了这一项。其次，如果一个术语很少出现（可能只出现在一个文档中），那么 IDF 就是 $\log n$ 的最大值。

示例 1-2：使用 sklearn 生产 TF-IDF 表示

```
from sklearn.feature_extraction.text import TfidfVectorizer
import seaborn as sns

tfidf_vectorizer = TfidfVectorizer()
tfidf = tfidf_vectorizer.fit_transform(corpus).toarray()
sns.heatmap(tfidf, annot=True, cbar=False, xticklabels=vocab,
            yticklabels= ['Sentence 1', 'Sentence 2'])
```



在深度学习中，很少看到使用像 TF-IDF 这样的启发式表示对输入进行编码，因为目标是学习一种表示。通常，我们从一个使用整数索引的单热编码和一个特殊的“嵌入查找”层开始构建神经网络的输入。在后面的章节中，我们将给出几个这样做的例子。

目标编码

正如“监督学习范式”所指出的，目标变量的确切性质取决于所解决的 NLP 任务。例如，在机器翻译、摘要和回答问题的情况下，目标也是文本，并使用前面描述的单热编码方法进行编码。

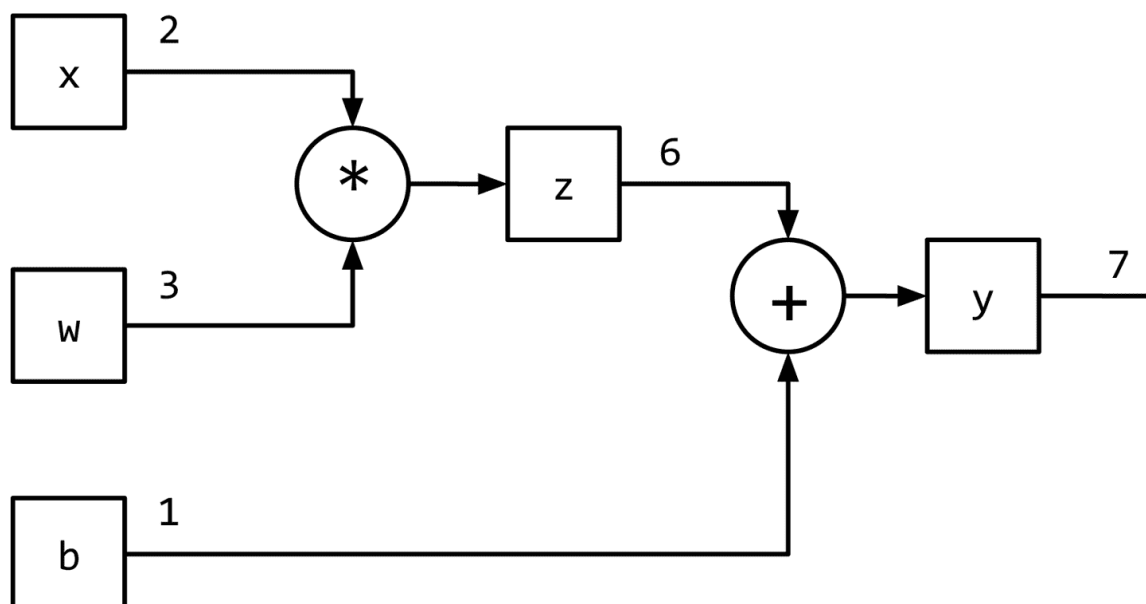
许多 NLP 任务实际上使用分类标签，其中模型必须预测一组固定标签中的一个。对此进行编码的一种常见方法是对每个标签使用惟一索引。当输出标签的数量太大时，这种简单的表示可能会出现问題。这方面的一个例子是语言建模问题，在这个问题中，任务是预测下一个单词，给定过去看到的单词。标签空间是一种语言的全部词汇，它可以很容易地增长到几十万，包括特殊字符、名称等等。我们将在后面的章节中重新讨论这个问题以及如何解决这个问题。

一些 NLP 问题涉及从给定文本中预测一个数值。例如，给定一篇英语文章，我们可能需要分配一个数字评分或可读性评分。给定一个餐馆评论片段，我们可能需要预测直到小数点后第一位的星级。给定用户的推文，我们可能需要预测用户的年龄群。有几种方法可以对数字目标进行编码，但是将目标简单地绑定到分类“容器”中（例如，“0-18”、“19-25”、“25-30”等等），并将其视为有序分类问题是一种合理的方法。绑定可以是均匀的，也可以是非均匀的，数据驱动的。虽然关于这一点的详细讨论超出了本书的范围，但是我们提请您注意这些问题，因为在这种情况下，目标编码会显著影响性能，我们鼓励您参阅 Dougherty 等人（1995）及其引用。

计算图

图 1-1 将监督学习（训练）范式概括为数据流架构，模型（数学表达式）对输入进行转换以获得预测，损失函数（另一个表达式）提供反馈信号来调整模型的参数。利用计算图数据结构可以方便地实现该数据流。从技术上讲，计算图是对数学表达式建模的抽象。在深度学习的上下文中，计算图的实现（如 Theano、TensorFlow 和 PyTorch）进行了额外的记录（bookkeeping），以实现在监督学习范式中训练期间获取参数梯度所需的自动微分。我们将在“PyTorch 基础知识”中进一步探讨这一点。推理（或预测）就是简单的表达式求值（计算图上的正向流）。让我们看看计算图如何建模表达式。考虑表达式： $y=wx+b$

这可以写成两个子表达式 $z = wx$ 和 $y = z + b$ ，然后我们可以用一个有向无环图（DAG）表示原始表达式，其中的节点是乘法和加法等数学运算。操作的输入是节点的传入边，操作的输出是传出边。因此，对于表达式 $y = wx + b$ ，计算图如图 1-6 所示。在下一节中，我们将看到 PyTorch 如何让我们以一种直观的方式创建计算图形，以及它如何让我们计算梯度，而无需考虑任何记录（bookkeeping）。



PyTorch 基础

在本书中，我们广泛地使用 PyTorch 来实现我们的深度学习模型。PyTorch 是一个开源、社区驱动的深度学习框架。与 Theano、Caffe 和 TensorFlow 不同，PyTorch 实现了一种“基于磁带的自动微分”方法，允许我们动态定义和执行计算图形。这对于调试和用最少的努力构建复杂的模型非常有帮助。

动态 VS 静态计算图 像 Theano、Caffe 和 TensorFlow 这样的静态框架需要首先声明、编译和执行计算图。虽然这会导致非常高效的实现（在生产和移动设置中非常有用），但在研究和开发过程中可能会变得非常麻烦。像 Chainer、DyNet 和 PyTorch 这样的现代框架实现了动态计算图，从而支持更灵活的命令式开发风格，而不需要在每次执行之前编译模型。动态计算图在建模 NLP 任务时特别有用，每个输入可能导致不同的图结构。

PyTorch 是一个优化的张量操作库，它提供了一系列用于深度学习的包。这个库的核心是张量，它是一个包含一些多维数据的数学对象。0 阶张量就是一个数字，或者标量。一阶张量（一阶张量）是一个数字数组，或者说是一个向量。类似地，二阶张量是一个向量数组，或者说是一个矩阵。因此，张量可以推广为标量的 n 维数组，如图 1-7 所示

图 1-7 还未给出

在以下部分中，我们将使用 PyTorch 学习以下内容：

- 创建张量
- 操作与张量
- 索引、切片和与张量连接
- 用张量计算梯度
- 使用带有 gpu 的 CUDA 张量

在本节的其余部分中，我们将首先使用 PyTorch 来熟悉各种 PyTorch 操作。我们建议您现在已经安装了 PyTorch 并准备好了 Python 3.5+ 笔记本，并按照本节中的示例进行操作。我们还建议您完成本节后面的练习。

安装 PyTorch

第一步是通过在 pytorch.org 上选择您的系统首选项在您的机器上安装 PyTorch。选择您的操作系统，然后选择包管理器（我们推荐 `conda/pip`），然后选择您正在使用的 Python 版本（我们推荐 3.5+）。这将生成命令，以便您执行安装 PyTorch。在撰写本文时，conda 环境的安装命令如下：

```
conda install pytorch torchvision -c pytorch
```

注意：如果您有一个支持 CUDA 的图形处理器单元（GPU），您还应该选择合适的 CUDA 版本。要了解更多细节，请参考 pytorch.org 上的安装说明。

创建张量

首先，我们定义一个辅助函数，描述（x），它总结了张量 x 的各种性质，例如张量的类型、张量的维数和张量的内容：

```
Input[0]:
def describe(x):
    print("Type: {}".format(x.type()))
    print("Shape/size: {}".format(x.shape))
    print("Values: \n{}".format(x))
```

PyTorch 允许我们使用 torch 包以许多不同的方式创建张量。创建张量的一种方法是通过指定一个随机张量的维数来初始化它，如例 1-3 所示。

示例 1-3：在 PyTorch 中使用 torch.Tensor 创建张量

```
Input[0]:
import torch
describe(torch.Tensor(2, 3))
Output[0]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 3.2018e-05,  4.5747e-41,  2.5058e+25],
        [ 3.0813e-41,  4.4842e-44,  0.0000e+00]])
```

我们还可以创建一个张量通过随机初始化值区间上的均匀分布（0,1）或标准正态分布（从均匀分布随机初始化张量,说是很重要的,正如您将看到的在第三章和第四章）,见示例 1-4。

示例 1-4：创建随机初始化的张量

```
Input[0]:
import torch
describe(torch.rand(2, 3)) # uniform random
describe(torch.randn(2, 3)) # random normal
Output[0]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.0242,  0.6630,  0.9787],
        [ 0.1037,  0.3920,  0.6084]])
```

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ -0.1330, -2.9222, -1.3649],
        [ 2.3648,  1.1561,  1.5042]])
```

我们还可以创建张量，所有张量都用相同的标量填充。对于创建 0 或 1 张量，我们有内置函数，对于填充特定值，我们可以使用 `fill_()` 方法。任何带下划线（`_`）的 PyTorch 方法都是指就地（in place）操作；也就是说，它在不创建新对象的情况下就地修改内容，如示例 1-5 所示。

示例 1-5：创建填充的张量

```
Input[0]:
import torch
describe(torch.zeros(2, 3))
x = torch.ones(2, 3)
describe(x)
x.fill_(5)
describe(x)
Output[0]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 5.,  5.,  5.],
        [ 5.,  5.,  5.]])
```

示例 1-6 演示了如何通过使用 Python 列表以声明的方式创建张量。

示例 1-6：从列表创建和初始化张量

```
Input[0]:
x = torch.Tensor([[1, 2, 3],
                  [4, 5, 6]])
describe(x)
Output[0]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
```



```
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.]])
```

值可以来自列表（如前面的示例），也可以来自 NumPy 数组。当然，我们也可以从 PyTorch 张量变换到 NumPy 数组。注意，这个张量的类型是一个 `double` 张量，而不是默认的 `FloatTensor`。这对应于 NumPy 随机矩阵的数据类型 `float64`，如示例 1-7 所示。

示例 1-7：从 NumPy 创建和初始化张量

```
Input[0]:
import torch
import numpy as np
np.random.rand(2, 3)
describe(torch.from_numpy(np.random.rand(2, 3)))
Output[0]:
Type: torch.DoubleTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.8360,  0.8836,  0.0545],
        [ 0.6928,  0.2333,  0.7984]], dtype=torch.float64)
```

在处理使用 Numpy 格式数值的遗留库（legacy libraries）时，在 NumPy 和 PyTorch 张量之间切换的能力变得非常重要。

张量类型和大小

每个张量都有一个相关的类型和大小。使用 `torch` 时的默认张量类型。张量构造函数是 `torch.FloatTensor`。但是，可以在初始化时指定张量，也可以在以后使用类型转换方法将张量转换为另一种类型（`float`、`long`、`double` 等）。有两种方法可以指定初始化类型，一种是直接调用特定张量类型（如 `FloatTensor` 和 `LongTensor`）的构造函数，另一种是使用特殊的方法 `torch.tensor`，并提供 `dtype`，如例 1-8 所示。

示例 1-8：张量属性

```
Input[0]:
x = torch.FloatTensor([[1, 2, 3],
                       [4, 5, 6]])
describe(x)
Output[0]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.]])
Input[1]:
x = x.long()
describe(x)
Output[1]:
Type: torch.LongTensor
Shape/size: torch.Size([2, 3])
```

```

Values:
tensor([[ 1,  2,  3],
        [ 4,  5,  6]])
Input[2]:
x = torch.tensor([[1, 2, 3],
                  [4, 5, 6]], dtype=torch.int64)
describe(x)
Output[2]:
Type: torch.LongTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1,  2,  3],
        [ 4,  5,  6]])
Input[3]:
x = x.float()
describe(x)
Output[3]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.]])

```

我们利用张量对象的形状特性和尺寸方法来获取其尺寸的测量值。访问这些度量的两种方法基本上是相同的。在调试 PyTorch 代码时，检查张量的形状成为必不可少的工具。

张量操作

在创建了张量之后，可以像处理传统编程语言类型（如 `+`、`-`、`*` 和 `/`）那样对它们进行操作。除了操作符，我们还可以使用 `.add()` 之类的函数，如示例 1-9 所示，这些函数对应于符号操作符。

示例 1-9：张量操作：加法

```

Input[0]:
import torch
x = torch.randn(2, 3)
describe(x)
Output[0]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.0461,  0.4024, -1.0115],
        [ 0.2167, -0.6123,  0.5036]])
Input[1]:
describe(torch.add(x, x))
Output[1]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.0923,  0.8048, -2.0231],
        [ 0.4335, -1.2245,  1.0072]])
Input[2]:

```

```

describe(x + x)
Output[2]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.0923,  0.8048, -2.0231],
        [ 0.4335, -1.2245,  1.0072]])

```

还有一些运算可以应用到张量的特定维数上。正如您可能已经注意到的，对于 2D 张量，我们将行表示为维度 0，列表示为维度 1，如示例 1-10 所示。

示例 1-10：基于维度的张量操作

```

Input[0]:
import torch
x = torch.arange(6)
describe(x)
Output[0]:
Type: torch.FloatTensor
Shape/size: torch.Size([6])
Values:
tensor([ 0.,  1.,  2.,  3.,  4.,  5.])
Input[1]:
x = x.view(2, 3)
describe(x)
Output[1]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
Input[2]:
describe(torch.sum(x, dim=0))
Output[2]:
Type: torch.FloatTensor
Shape/size: torch.Size([3])
Values:
tensor([ 3.,  5.,  7.])
Input[3]:
describe(torch.sum(x, dim=1))
Output[3]:
Type: torch.FloatTensor
Shape/size: torch.Size([2])
Values:
tensor([ 3., 12.])
Input[4]:
describe(torch.transpose(x, 0, 1))
Output[4]:
Type: torch.FloatTensor
Shape/size: torch.Size([3, 2])
Values:
tensor([[ 0.,  3.],
        [ 1.,  4.],
        [ 2.,  5.]])

```

通常，我们需要执行更复杂的操作，包括索引、切片、连接和突变（indexing, slicing, joining and mutation）的组合。与 NumPy 和其他数字库一样，PyTorch 也有内置函数，可以使此类张量操作非常简单。

索引，切片和连接

如果您是一个 NumPy 用户，那么您可能非常熟悉示例 1-11 中所示的 PyTorch 的索引和切片方案。

示例 1-11：切片和索引张量

```
Input[0]:
import torch
x = torch.arange(6).view(2, 3)
describe(x)
Output[0]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
Input[1]:
describe(x[:1, :2])
Output[1]:
Type: torch.FloatTensor
Shape/size: torch.Size([1, 2])
Values:
tensor([[ 0.,  1.]])
Input[2]:
describe(x[0, 1])
Output[2]:
Type: torch.FloatTensor
Shape/size: torch.Size([])
Values:
1.0
```

示例 1-12 演示了 PyTorch 还具有用于复杂索引和切片操作的函数，您可能对有效地访问张量的非连续位置感兴趣。

示例 1-12：复杂索引：张量的非连续索引

```
Input[0]:
indices = torch.LongTensor([0, 2])
describe(torch.index_select(x, dim=1, index=indices))
Output[0]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 2])
Values:
tensor([[ 0.,  2.],
        [ 3.,  5.]])
Input[1]:
```

```

indices = torch.LongTensor([0, 0])
describe(torch.index_select(x, dim=0, index=indices))
Output[1]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.,  1.,  2.],
        [ 0.,  1.,  2.]])
Input[2]:
row_indices = torch.arange(2).long()
col_indices = torch.LongTensor([0, 1])
describe(x[row_indices, col_indices])
Output[2]:
Type: torch.FloatTensor
Shape/size: torch.Size([2])
Values:
tensor([ 0.,  4.])

```

注意索引（indices）是一个长张量;这是使用 PyTorch 函数进行索引的要求。我们还可以使用内置的连接函数连接张量，如示例 1-13 所示，通过指定张量和维度。

示例 1-13：连接张量

```

Input[0]:
import torch
x = torch.arange(6).view(2,3)
describe(x)
Output[0]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
Input[1]:
describe(torch.cat([x, x], dim=0))
Output[1]:
Type: torch.FloatTensor
Shape/size: torch.Size([4, 3])
Values:
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.],
        [ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
Input[2]:
describe(torch.cat([x, x], dim=1))
Output[2]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 6])
Values:
tensor([[ 0.,  1.,  2.,  0.,  1.,  2.],
        [ 3.,  4.,  5.,  3.,  4.,  5.]])
Input[3]:
describe(torch.stack([x, x]))
Output[3]:
Type: torch.FloatTensor

```

```
Shape/size: torch.Size([2, 2, 3])
Values:
tensor([[[ 0.,  1.,  2.],
          [ 3.,  4.,  5.]],

        [[ 0.,  1.,  2.],
          [ 3.,  4.,  5.]])
```

PyTorch 还在张量上实现了高效的线性代数操作，如乘法、逆和迹，如示例 1-14 所示。

示例 1-14：张量上的线性代数：乘法

```
Input[0]:
import torch
x1 = torch.arange(6).view(2, 3)
describe(x1)
Output[0]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[[ 0.,  1.,  2.],
          [ 3.,  4.,  5.]]])
Input[1]:
x2 = torch.ones(3, 2)
x2[:, 1] += 1
describe(x2)
Output[1]:
Type: torch.FloatTensor
Shape/size: torch.Size([3, 2])
Values:
tensor([[[ 1.,  2.],
          [ 1.,  2.],
          [ 1.,  2.]])
Input[2]:
describe(torch.mm(x1, x2))
Output[2]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 2])
Values:
tensor([[[ 3.,  6.],
          [12., 24.]])
```

到目前为止，我们已经研究了创建和操作恒定 PyTorch 张量对象的方法。就像编程语言（如 Python）变量封装一块数据,关于数据的额外信息（如内存地址存储,例如),PyTorch 张量处理构建计算图时所需的记账（bookkeeping）所需构建计算图对机器学习只是在实例化时通过启用一个布尔标志。

张量和计算图

PyTorch 张量类封装了数据（张量本身）和一系列操作，如代数操作、索引操作和整形操作。然而，1-15 所示的例子，当 `requires_grad` 布尔标志被设置为 `True` 的张量，记账操作启用，可以追踪的梯度张量以及梯度函数，这两个需要基于促进梯度学习讨论“监督学习范式”。

示例 1-15：为梯度记录创建张量

```
Input[0]:
import torch
x = torch.ones(2, 2, requires_grad=True)
describe(x)
print(x.grad is None)
Output[0]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 2])
Values:
tensor([[ 1.,  1.],
         [ 1.,  1.]])
True
Input[1]:
y = (x + 2) * (x + 5) + 3
describe(y)
print(x.grad is None)
Output[1]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 2])
Values:
tensor([[ 21.,  21.],
         [ 21.,  21.]])
True
Input[2]:
z = y.mean()
describe(z)
z.backward()
print(x.grad is None)
Output[2]:
Type: torch.FloatTensor
Shape/size: torch.Size([2, 2])
Values:
21.0
False
```

当您使用 `requires_grad=True` 创建张量时，您需要 PyTorch 来管理计算梯度的 bookkeeping 信息。首先，PyTorch 将跟踪向前传递的值。然后，在计算结束时，使用单个标量来计算向后传递。反向传递是通过对一个张量使用 `backward()` 方法来初始化的，这个张量是由一个损失函数的求值得到的。向后传递为参与向前传递的张量对象计算梯度值。

一般来说，梯度是一个值，它表示函数输出相对于函数输入的斜率。在计算图形设置中，模型中的每个参数都存在梯度，可以认为是该参数对误差信号的贡献。在 PyTorch 中，可以使用 `.grad` 成员变量访问计算图中节点的梯度。优化器使用 `.grad` 变量更新参数的值。

到目前为止，我们一直在 CPU 内存上分配张量。在做线性代数运算时，如果你有一个 GPU，那么利用它可能是有意义的。要利用 GPU，首先需要分配 GPU 内存上的张量。对 gpu 的访问是通过一个名为 CUDA 的专门 API 进行的。CUDA API 是由 NVIDIA 创建的，并且仅限于在 NVIDIA gpu 上使用。PyTorch 提供的 CUDA 张量对象在使用中与常规 cpu 绑定张量没有区别，除了内部分配的方式不同。

CUDA 张量

PyTorch 使创建这些 CUDA 张量变得非常容易（示例 1-16），它将张量从 CPU 传输到 GPU，同时维护其底层类型。PyTorch 中的首选方法是与设备无关，并编写在 GPU 或 CPU 上都能工作的代码。在下面的代码片段中，我们首先使用 `torch.cuda.is_available()` 检查 GPU 是否可用，然后使用 `torch.device` 检索设备名。然后，将实例化所有未来的张量，并使用 `.to(device)` 方法将其移动到目标设备。

示例 1-16：创建 CUDA 张量

```
Input[0]:
import torch
print (torch.cuda.is_available())
Output[0]:
True
Input[1]:
# preferred method: device agnostic tensor instantiation
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print (device)
Output[1]:
cuda
Input[2]:
x = torch.rand(3, 3).to(device)
describe(x)
Output[2]:
Type: torch.cuda.FloatTensor
Shape/size: torch.Size([3, 3])
Values:
tensor([[ 0.9149,  0.3993,  0.1100],
        [ 0.2541,  0.4333,  0.4451],
        [ 0.4966,  0.7865,  0.6604]], device='cuda:0')
```

要对 CUDA 和非 CUDA 对象进行操作，我们需要确保它们在同一设备上。如果我们不这样做，计算就会中断，如下面的代码片段所示。例如，在计算不属于计算图的监视指标时，就会出现这种情况。当操作两个张量对象时，确保它们在同一个设备上。例子 1-17 所示。

示例 1-17：混合 CUDA 张量和 CPU 绑定的张量

```
Input[0]
y = torch.rand(3, 3)
x + y
Output[0]
-----
```



```

RuntimeError                                Traceback (most recent call last)
      1 y = torch.rand(3, 3)
----> 2 x + y

RuntimeError: Expected object of type torch.cuda.FloatTensor but found type
torch.FloatTensor for argument #3 'other'
Input[1]
cpu_device = torch.device("cpu")
y = y.to(cpu_device)
x = x.to(cpu_device)
x + y
Output[1]
tensor([[ 0.7159,  1.0685,  1.3509],
        [ 0.3912,  0.2838,  1.3202],
        [ 0.2967,  0.0420,  0.6559]])

```

请记住，将数据从 GPU 来回移动是非常昂贵的。因此，典型的过程包括在 GPU 上执行许多并行计算，然后将最终结果传输回 CPU。这将允许您充分利用 gpu。如果您有几个 CUDA 可见的设备（即，最佳实践是在执行程序时使用 `CUDA_VISIBLE_DEVICES` 环境变量，如下图所示：

```
CUDA_VISIBLE_DEVICES=0,1,2,3 python main.py
```

在本书中我们不涉及并行性和多 gpu 训练，但是它们在缩放实验中是必不可少的，有时甚至在训练大型模型时也是如此。我们建议您参考 PyTorch 文档和讨论论坛，以获得关于这个主题的更多帮助和支持。

练习

掌握一个主题的最好方法是解决问题。这里有一些热身运动。许多问题将涉及到查阅官方文件[1]和寻找有用的功能。 .

1. Create a 2D tensor and then add a dimension of size 1 inserted at dimension 0.
2. Remove the extra dimension you just added to the previous tensor.
3. Create a random tensor of shape 5x3 in the interval [3, 7)
4. Create a tensor with values from a normal distribution (mean=0, std=1).
5. Retrieve the indexes of all the nonzero elements in the tensor `torch.Tensor([1, 1, 1, 0, 1])`.
6. Create a random tensor of size (3,1) and then horizontally stack 4 copies together.
7. Return the batch matrix-matrix product of two 3-dimensional matrices (`a=torch.rand(3,4,5)`, `b=torch.rand(3,5,4)`).
8. Return the batch matrix-matrix product of a 3D matrix and a 2D matrix (`a=torch.rand(3,4,5)`, `b=torch.rand(5,4)`).

Solutions

9. `a = torch.rand(3, 3) a.unsqueeze(0)`
10. `a.squeeze(0)`
11. `3 + torch.rand(5, 3) * (7 - 3)`
12. `a = torch.rand(3, 3) a.normal_()`
13. `a = torch.Tensor([1, 1, 1, 0, 1]) torch.nonzero(a)`
14. `a = torch.rand(3, 1) a.expand(3, 4)`
15. `a = torch.rand(3, 4, 5) b = torch.rand(3, 5, 4) torch.bmm(a, b)`
16. `a = torch.rand(3, 4, 5) b = torch.rand(5, 4) torch.bmm(a, b.unsqueeze(0).expand(a.size(0), *
b.size()))`

总结

在这一章中，我们介绍了本书的目标——自然语言处理（NLP）和深度学习——并对监督学习范式进行了详细的理解。在本章的最后，您现在应该熟悉或至少了解各种术语，例如观察、目标、模型、参数、预测、损失函数、表示、学习/训练和推理。您还了解了如何使用单热编码对学习任务的输入（观察和目标）进行编码。我们还研究了基于计数的表示，如 TF 和 TF-IDF。我们首先了解了什么是计算图，静态和动态计算图，以及 PyTorch 张量操纵操作。在第二章中，我们对传统的 NLP 进行了概述。第二章，这一章应该为你奠定必要的基础，如果你对这本书的主题是新的，并为你的书的其余部分做准备。

重点是 TF-IDF

词频 (TF) = 某个词在文章中出现的次数 / 文章中的总词数

逆文档频率 (IDF) = $\log(\text{语料库的文档总数} / (\text{包含该词的文档数} + 1))$

TF 应该很容易理解就是计算词频,IDF 衡量词的常见程度.为了计算 IDF 我们需要事先准备一个语料库用来模拟语言的使用环境,如果一个词越是常见,那么式子中分母越大,逆文档频率越接近 0.这里分母 +1 是为了避免分母为 0 的情况出现

TF-IDF = 词频 (TF) × 逆文档频率 (IDF)

TF-IDF 可以很好的实现提取文章中关键词的目的.

二、传统 NLP 快速回顾

本文标题: [Natural-Language-Processing-with-PyTorch \(二\)](#)

文章作者: [Yif Du](#)

发布时间: 2018 年 12 月 18 日 - 13:12

最后更新: 2019 年 02 月 16 日 - 23:02

原始链接: [http://yifdu.github.io/2018/12/18/Natural-Language-Processing-with-PyTorch \(二\) /](http://yifdu.github.io/2018/12/18/Natural-Language-Processing-with-PyTorch (二) /)

许可协议: [署名-非商业性使用-禁止演绎 4.0 国际](#) 转载请保留原文链接及作者。

自然语言处理 (NLP) 和计算语言学 (CL) 是人类语言计算研究的两个领域。NLP 旨在开发解决涉及语言的实际问题的方法, 如信息提取、自动语音识别、机器翻译、情绪分析、问答和总结。另一方面, CL 使用计算方法来理解人类语言的特性。我们如何理解语言? 我们如何产生语言? 我们如何学习语言? 语言之间有什么关系?

在文献中, 我们经常看到方法和研究人员的交叉, 从 CL 到 NLP, 反之亦然。来自语言学习的课程内容可以用来告知 NLP 中的先验, 统计和机器学习的方法可以用来回答 CL 想要回答的问题。事实上, 这些问题中的一些已经扩展到它们自己的学科, 如音位学、形态学、句法学、语义学和语用学。

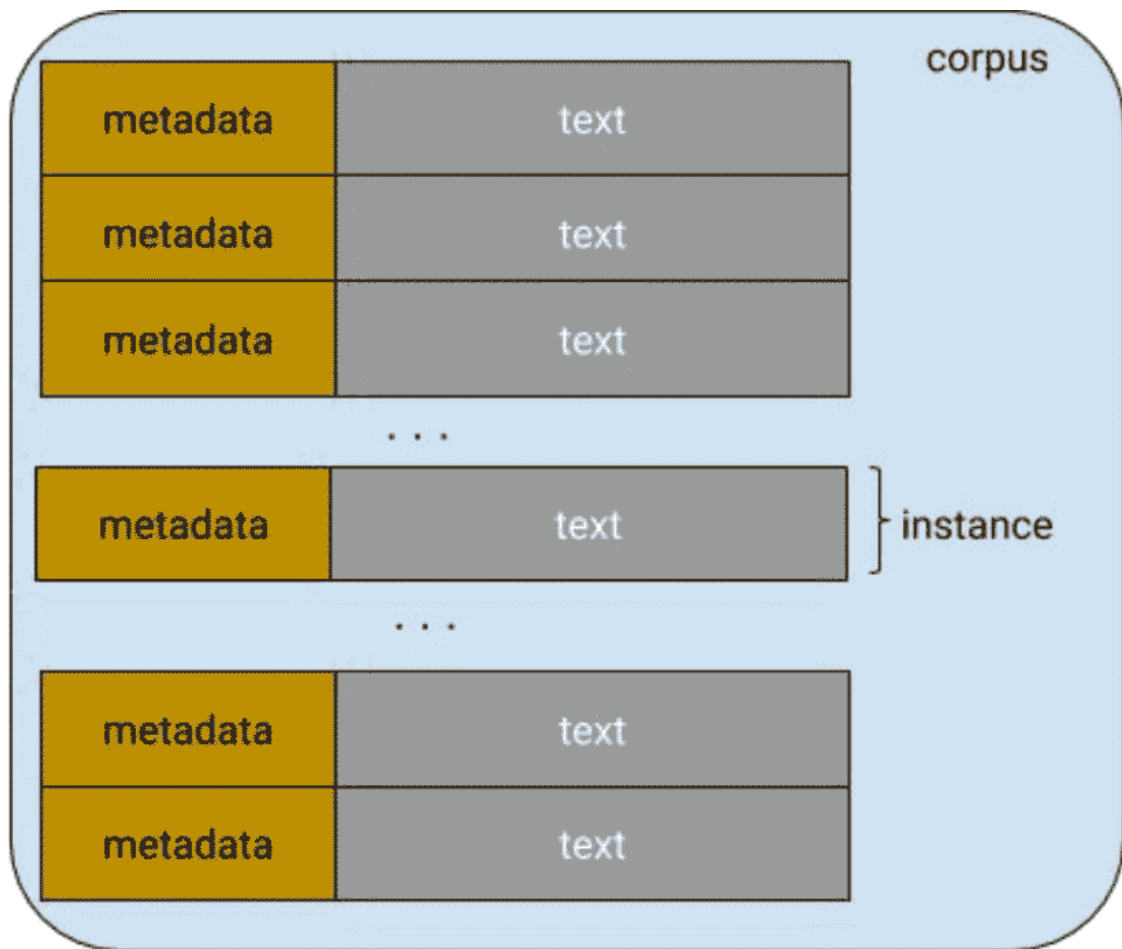
在本书中, 我们只关注 NLP, 但是我们经常根据需要从 CL 中借鉴思想。在我们将自己完全归属于 NLP 的神经网络方法之前, 有必要回顾一下一些传统的 NLP 概念和方法。这就是本章的目标。

如果你有一些 NLP 的背景, 你可以跳过这一章, 但你也可以留下来怀旧, 为未来建立一个共享的词汇表。

语料库, 标记和类型

所有的 NLP 方法, 无论是经典的还是现代的, 都以文本数据集开始, 也称为语料库 (复数: corpora)。语料库通常有原始文本 (ASCII 或 UTF-8 格式) 和与文本相关的任何元数据。原始文本是字符 (字节) 序列, 但是大多数时候将字符分组成连续的称为标记 (Tokens) 的连续单元是有用的。在英语中, 标记 (Tokens) 对应由空格字符或标点分隔的单词和数字序列。

元数据可以是与文本相关联的任何辅助信息, 例如标识符, 标签和时间戳。在机器学习术语中, 文本及其元数据称为实例或数据点。语料库 (图 2-1) 是一组实例, 也称为数据集。鉴于本书重点关注机器学习, 我们可以自由地交换术语语料库和数据集。



将文本分解为标记（Tokens）的过程称为分词（tokenization）。世界语的句子，`Maria frapis la verda sorĉistino` 有六个标记。分词可能比简单地基于非字母数字字符拆分文本更加复杂，如图 2-2 所示。对于像土耳其语这样的粘合语言来说，分隔空格和标点符号可能是不够的，因此可能需要更专业的技术。正如您将在第 4 章和第 6 章中看到的，通过将文本表示为字节流，我们可能能够在某些神经网络模型中完全规避分词问题;这对于粘合语言来说变得非常重要。

Turkish	English
kork(-mak)	(to) fear
korku	fear
korkusuz	fearless
korkusuzlaş (-mak)	(to) become fearless
korkusuzlaşmış	One who has become fearless
korkusuzlaştır(-mak)	(to) make one fearless
korkusuzlaştırıl(-mak)	(to) be made fearless
korkusuzlaştırılmış	One who has been made fearless
korkusuzlaştırılabil(-mek)	(to) be able to be made fearless
korkusuzlaştırılabilecek	One who will be able to be made fearless
korkusuzlaştırılabileceklerimiz	Ones who we can make fearless
korkusuzlaştırılabileceklerimizden	From the ones who we can make fearless
korkusuzlaştırılabileceklerimizdenmiş	I gather that one is one of those we can make fearless
korkusuzlaştırılabileceklerimizdenmişçesine	As if that one is one of those we can make fearless
korkusuzlaştırılabileceklerimizdenmişçesineyken	when it seems like that one is one of those we can make fearless

最后，看看下面这条推文：



分词 tweets 涉及到保存话题标签和 @handle，将表情符号（如 :-）和 urls 分割为一个单元。
#MakeAMovieCold 标签应该是 1 个标记还是 4 个？虽然大多数研究论文对这一问题并没有给予太多的关注，而且事实上，许多分词决策往往是任意的，但是这些决策在实践中对准确性的影响要比公认的要大得多。通常被认为是预处理的繁琐工作，大多数开放源码 NLP 包为分词提供了合理的支持。示例 2-1 展示了来自 NLTK 和 SpaCy 的示例，这是两个用于文本处理的常用包。

示例 2-1：对文本分词

```
Input[0]:
import spacy
nlp = spacy.load('en')
text = "Mary, don't slap the green witch"
print([str(token) for token in nlp(text.lower())])
Output[0]:
['mary', ',', 'do', "n't", 'slap', 'the', 'green', 'witch', '.']
Input[1]:
from nltk.tokenize import TweetTokenizer
tweet=u"Snow White and the Seven Degrees
#MakeAMovieCold@midnight:-)"
```

```
tokenizer = TweetTokenizer()
print(tokenizer.tokenize(tweet.lower()))
Output[1]:
['snow', 'white', 'and', 'the', 'seven', 'degrees', '#makeamoviecold',
 '@midnight', ':-)']
```

类型是语料库中唯一的标记。语料库中所有类型的集合就是它的词汇表或词典。词可以区分为内容词和停止词。像冠词和介词这样的限定词主要是为了达到语法目的，就像填充物承载着内容词一样。

注：这种理解语言的语言学并将其应用于解决自然语言处理问题的过程称为特征工程。为了模型在不同语言之间的方便和可移植性，我们将这一点保持在最低限度。但是在构建和部署真实的生产系统时，特性工程是必不可少的，尽管最近的说法与此相反。对于特征工程的介绍，一般来说，可以阅读 Zheng（2016）的书。

一元组，二元组，三元组，...，N 元组

N 元组是文本中出现的固定长度（n）的连续标记序列。二元组有两个标记，一元组只有一个标记。从文本生成 N 元组非常简单，如示例 2-2 所示，但是 SpaCy 和 NLTK 等包提供了方便的方法。

示例 2-2：生成 N 元组

```
Input[0]:
def n_grams(text, n):
    ...
    takes tokens or text, returns a list of n grams
    ...
    return [text[i:i+n] for i in range(len(text)-n+1)]

cleaned = ['mary', ',', 'n't', 'slap', green', 'witch', '.']
print(n_grams(cleaned, 3))
Output[0]:
[['mary', ',', 'n't'],
 [' ', 'n't', 'slap'],
 ['n't', 'slap', 'green'],
 ['slap', 'green', 'witch'],
 ['green', 'witch', '.']]
```

对于子词（subword）信息本身携带有用信息的某些情况，可能需要生成字符 N 元组。例如，methanol 中的后缀 -ol 表示它是一种醇；如果您的任务涉及到对有机化合物名称进行分类，那么您可以看到 N 元组捕获的子单词（subword）信息是如何有用的。在这种情况下，您可以重用相同的代码，除了将每个字符 N 元组视为标记。（这里的子单词应该是值类似前缀后缀这种完整单词中的一部分）

词形和词干

词形是单词的词根形式。考虑动词 `fly`。它可以被屈折成许多不同的单词——`flow`、`fly`、`flies`、`flying`、`flow` 等等——而 `fly` 是所有这些看似不同的单词的词形。有时，为了保持向量表示的维数较低，将标记减少到它们的词形可能是有用的。这种简化称为词形还原（`lemmatization`），您可以在示例 2-3 中看到它的作用。

示例 2-3：词形还原

```
Input[0]
import spacy
nlp = spacy.load('en')
doc = nlp(u"he was running late")
for token in doc:
    print('{} --> {}'.format(token, token.lemma_))
Output[0]
he --> he
was --> be
running --> run
late --> late
```

例如，SpaCy 使用一个预定义的字典 WordNet 来提取词形，但是词形还原可以构建为一个机器学习问题，需要理解语言的形态学。

词干是最普通的词形还原。它涉及到使用手工制定的规则来去掉单词的结尾，从而将它们简化为一种叫做词干的常见形式。通常在开源包中实现的流行的词干分析器是 Porter 词干提取器和 Snowball 词干提取器。我们留给您去寻找合适的 SpaCy/NLTK api 来执行词干提取。

分类句子和文档

对文档进行归类或分类可能是 NLP 最早的应用之一。我们在第 1 章中描述的表示（词频（TF）和词频-逆文档频率（TF-idf））对于对较长的文本块（如文档或句子）进行分类和分类非常有用。主题标签的分配、评论情绪的预测、垃圾邮件的过滤、语言识别和邮件分类等问题可以被定义为受监督的文档分类问题。（半监督版本，其中只使用了一个小的标记数据集，非常有用，但超出了本书的范围。）

分类单词：词性标注

我们可以将标记的概念从文档扩展到单个单词或标记。分类单词的一个常见示例是词性标注，如示例 2-4 所示。

示例 2-4：词性

```
Input[0]
import spacy
```



```

nlp = spacy.load('en')
doc = nlp(u"Mary slapped the green witch.")
for token in doc:
    print('{} - {}'.format(token, token.pos_))
Output[0]
Mary - PROPN
slapped - VERB
the - DET
green - ADJ
witch - NOUN
. - PUNCT

```

分类短语：分块和命名实体识别

通常，我们需要标记文本的范围;即，一个连续的多标记边界。例如，

Mary slapped the green witch. 我们可能需要识别其中的名词短语（NP）和动词短语（VP），如下图所示：

[NP Mary] [VP slapped] [NP the green witch] .

这称为分块（Chunking）或浅解析（Shallow parsing）。浅解析的目的是推导出由名词、动词、形容词等语法原子组成的高阶单位。如果没有训练浅解析模型的数据，可以在词性标记上编写正则表达式来近似浅解析。幸运的是，对于英语和最广泛使用的语言来说，这样的数据和预先训练的模型是存在的。示例 2-5 给出了一个使用 SpaCy 的浅解析示例。

示例 2-5：名词块

```

Input[0]:
import spacy
nlp = spacy.load('en')
doc = nlp(u"Mary slapped the green witch.")
for chunk in doc.noun_chunks:
    print '{} - {}'.format(chunk, chunk.label_)
Output[0]:
Mary - NP
the green witch - NP

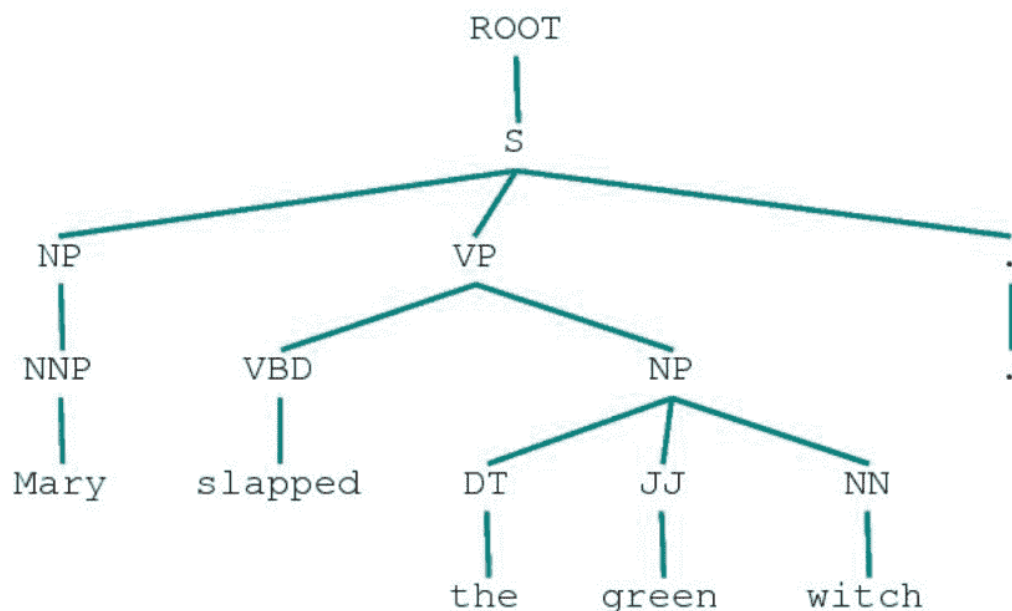
```

另一种有用的短语类型是命名实体。命名实体是一个字符串，它提到了一个真实世界的概念，如人员、位置、组织、药品名称等等。这里有一个例子：

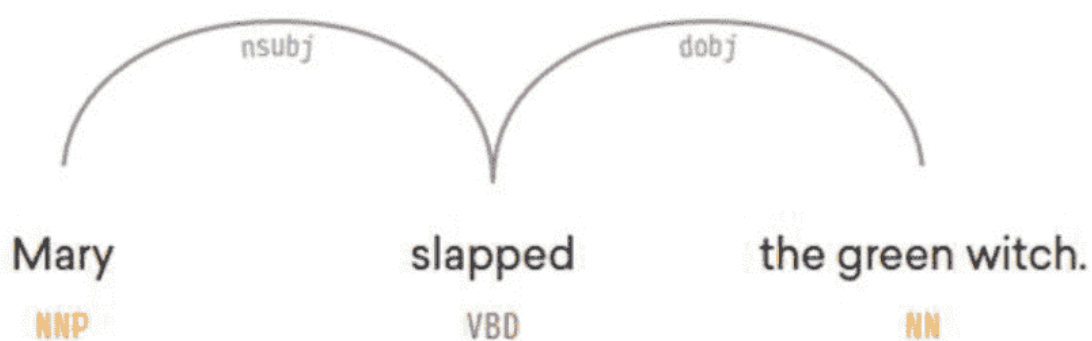
John **PERSON** was born in Chicken **GPE**, Alaska **GPE**, and studies at Cranberry Lemon University **ORG**.

句子结构

浅层解析识别短语单位，而识别它们之间关系的任务称为解析（parsing）。您可能还记得，在初级英语课上，用图表表示句子，如图 2-6 所示。



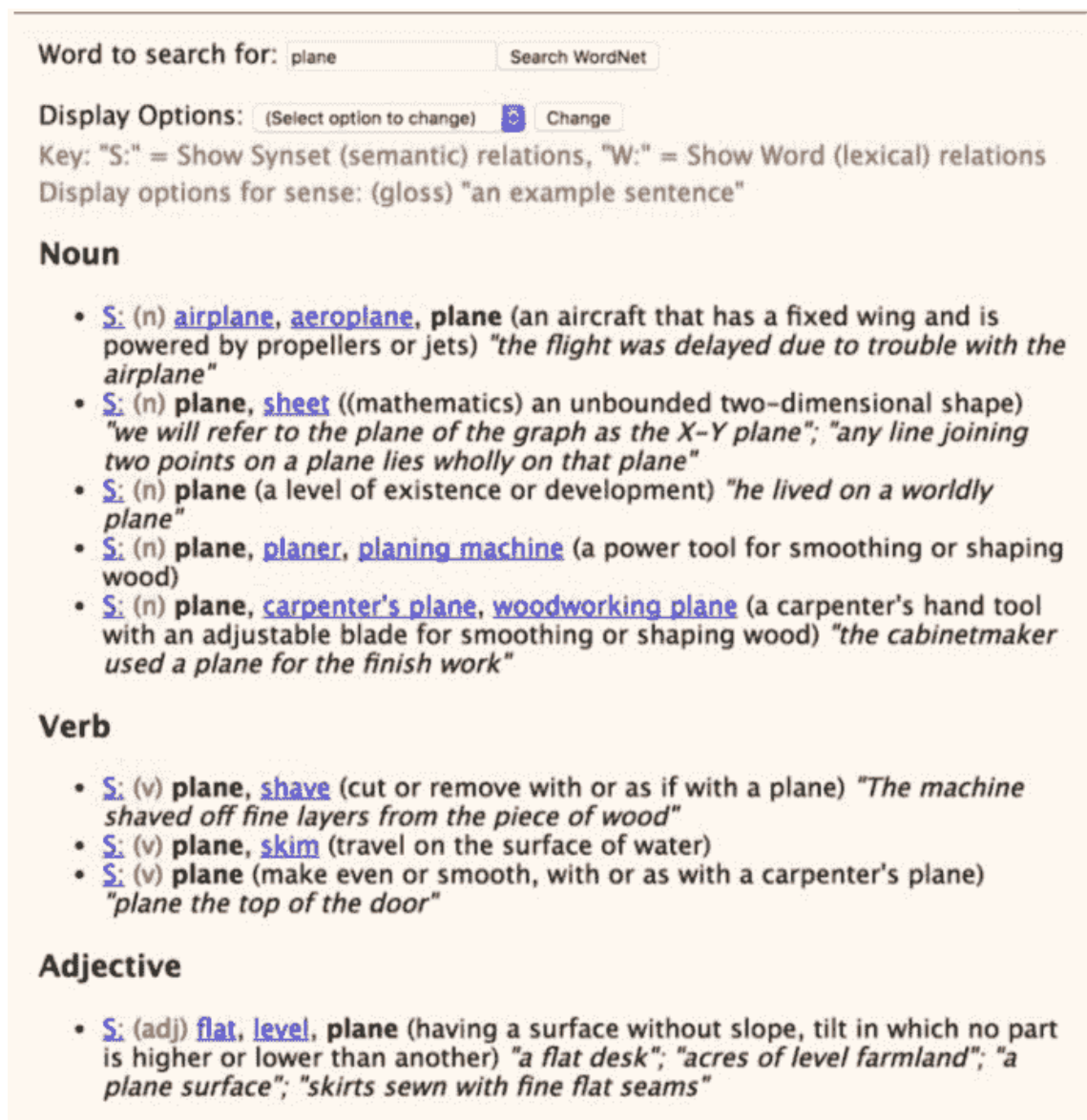
解析树（Parse tree）表示句子中不同的语法单元在层次上是如何相关的。图 2-6 中的解析树显示了所谓的成分解析。另一种可能更有用的显示关系的方法是使用依赖项解析（dependency parsing），如图 2-7 所示。



要了解更多关于传统解析的信息，请参阅本章末尾的参考资料部分。

单词意义和情感

单词有意义，而且通常不止一个。一个词的不同含义称为它的意义（senses）。WordNet 是一个长期运行的词汇资源项目，它来自普林斯顿大学，旨在对所有英语单词（嗯，大部分）的含义以及其他词汇关系进行分类。例如，考虑像 `plane` 这样的单词。图 2-8 显示了 `plane` 一词的不同用法。



Word to search for:

Display Options:

Key: "S:" = Show Synset (semantic) relations, "W:" = Show Word (lexical) relations
Display options for sense: (gloss) "an example sentence"

Noun

- **S:** (n) [airplane](#), [aeroplane](#), **plane** (an aircraft that has a fixed wing and is powered by propellers or jets) *"the flight was delayed due to trouble with the airplane"*
- **S:** (n) **plane**, [sheet](#) ((mathematics) an unbounded two-dimensional shape) *"we will refer to the plane of the graph as the X-Y plane"; "any line joining two points on a plane lies wholly on that plane"*
- **S:** (n) **plane** (a level of existence or development) *"he lived on a worldly plane"*
- **S:** (n) **plane**, [planer](#), [planing machine](#) (a power tool for smoothing or shaping wood)
- **S:** (n) **plane**, [carpenter's plane](#), [woodworking plane](#) (a carpenter's hand tool with an adjustable blade for smoothing or shaping wood) *"the cabinetmaker used a plane for the finish work"*

Verb

- **S:** (v) **plane**, [shave](#) (cut or remove with or as if with a plane) *"The machine shaved off fine layers from the piece of wood"*
- **S:** (v) **plane**, [skim](#) (travel on the surface of water)
- **S:** (v) **plane** (make even or smooth, with or as with a carpenter's plane) *"plane the top of the door"*

Adjective

- **S:** (adj) [flat](#), [level](#), **plane** (having a surface without slope, tilt in which no part is higher or lower than another) *"a flat desk"; "acres of level farmland"; "a plane surface"; "skirts sewn with fine flat seams"*

在 WordNet 这样的项目中数十年的努力是值得的，即使是在有现代方法的情况下。本书后面的章节给出了在神经网络和深度学习方法的背景下使用现有语言资源的例子。词的意义也可以从上下文中归纳出来。从文本中自动发现词义实际上是半监督学习在自然语言处理中的第一个应用。尽管我们在本书中没有涉及到这一点，但我们鼓励您阅读 Jurasky and Martin (2014)，第 17 章，Manning and Schutze (1999)，第 7 章。

总结

在这一章中，我们回顾了 NLP 中的一些基本术语和思想，这些在以后的章节中会很有用。本章只涉及了传统 NLP 所能提供的部分内容。我们忽略了传统 NLP 的一些重要方面，因为我们想将本书的大部分内容用于 NLP 的深度学习。然而，重要的是要知道，有大量的 NLP 研究工作不使用神经网络，但仍具有很高的影响力（即，广泛用于建筑生产系统）。在许多情况下，基于神经网络的方法应该被看作是传统方法的补充而不是替代。有经验的实践者经常使用这两个世界的优点来构建最先进的系统。为了更多地了解 NLP 的传统方法，我们推荐以下参考资料部分中的标题。

参考文献

1. Manning, Christopher D., and Hinrich Schütze. (1999). Foundations of statistical natural language processing. MIT press.
2. Bird, Steven, Ewan Klein, and Edward Loper. (2009). Natural language processing with Python: analyzing text with the natural language toolkit. O'Reilly Media.
3. Smith, Noah A. (2011). "Linguistic structure prediction." Synthesis lectures on human language technologies.
4. Jurafsky, Dan, and James H. Martin. (2014). Speech and language processing. Vol. 3. London: Pearson.
5. Russell, Stuart J., and Peter Norvig. (2016). Artificial intelligence: a modern approach. Malaysia: Pearson Education Limited.
6. Zheng, Alice, and Casari, Amanda. (2018). Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists. O'Reilly Media, Inc.

三、神经网络基础组件

本文标题: [Natural-Language-Processing-with-PyTorch \(三\)](#)

文章作者: [Yif Du](#)

发布时间: 2018 年 12 月 19 日 - 14:12

最后更新: 2018 年 12 月 28 日 - 11:12

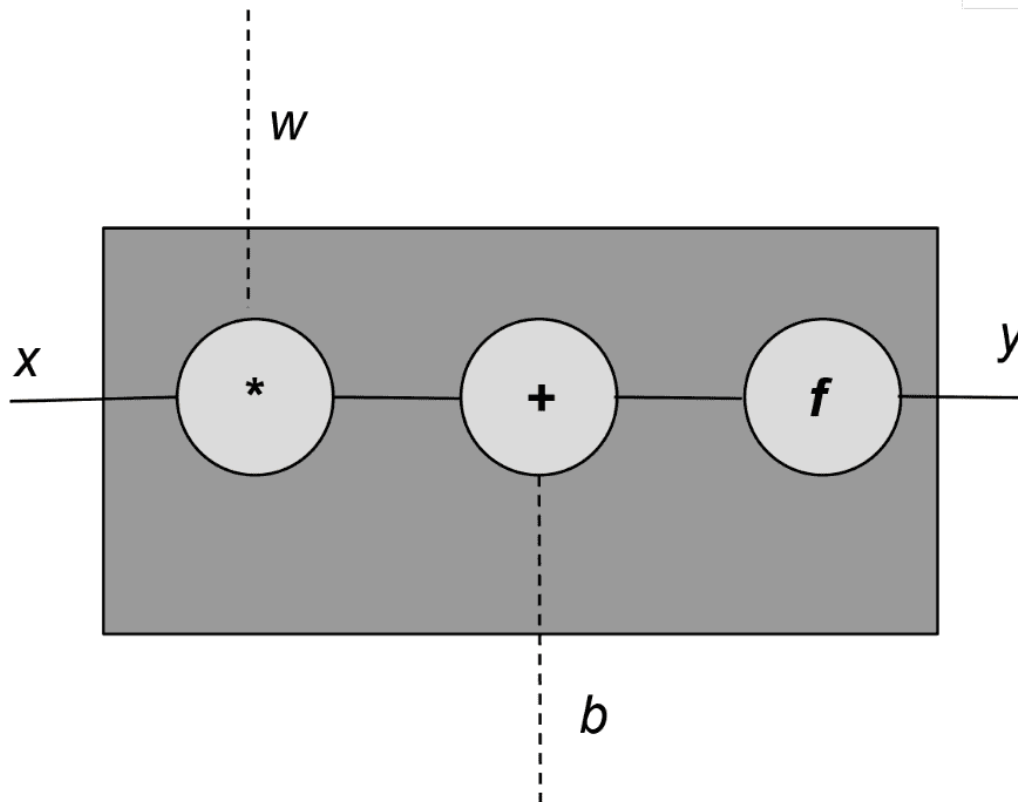
原始链接: <http://yifdu.github.io/2018/12/19/Natural-Language-Processing-with-PyTorch> (三) /

许可协议: [署名-非商业性使用-禁止演绎 4.0 国际](#) 转载请保留原文链接及作者。

本章通过介绍构建神经网络的基本思想，如激活函数、损失函数、优化器和监督训练设置，为后面的章节奠定了基础。我们从感知器开始，这是一个将不同概念联系在一起的一个单元的神经网络。感知器本身是更复杂的神经网络的组成部分。这是一种贯穿全书的常见模式，我们讨论的每个架构或网络都可以单独使用，也可以在其他复杂的网络中组合使用。当我们讨论计算图形和本书的其余部分时，这种组合性将变得清晰起来。

感知机：最简单的神经网络

最简单的神经网络单元是感知器。感知器在历史上是非常松散地模仿生物神经元的。就像生物神经元一样，有输入和输出，“信号”从输入流向输出，如图 3-1 所示。



每个感知器单元有一个输入 (x)，一个输出 (y)，和三个“旋钮” (knobs)：一组权重 (w)，偏量 (b)，和一个激活函数 (f)。权重和偏量都从数据学习,激活函数是精心挑选的取决于网络的网络设计师的直觉和目标输出。数学上，我们可以这样表示：

通常情况下感知器有不止一个输入。我们可以用向量表示这个一般情况;即， x 和 w 是向量， w 和 x 的乘积替换为点积：

激活函数，这里用 f 表示，通常是一个非线性函数。示例 3-1 展示了 PyTorch 中的感知器实现，它接受任意数量的输入、执行仿射转换、应用激活函数并生成单个输出。

示例 3-1：使用 PyTorch 实现感知机

```
import torch
import torch.nn as nn

class Perceptron(nn.Module):
    """ A Perceptron is one Linear layer """
    def __init__(self, input_dim):
        """
        Args:
            input_dim (int): size of the input features
        """
        super(Perceptron, self).__init__()
        self.fc1 = nn.Linear(input_dim, 1)

    def forward(self, x_in):
        """The forward pass of the Perceptron"""
```

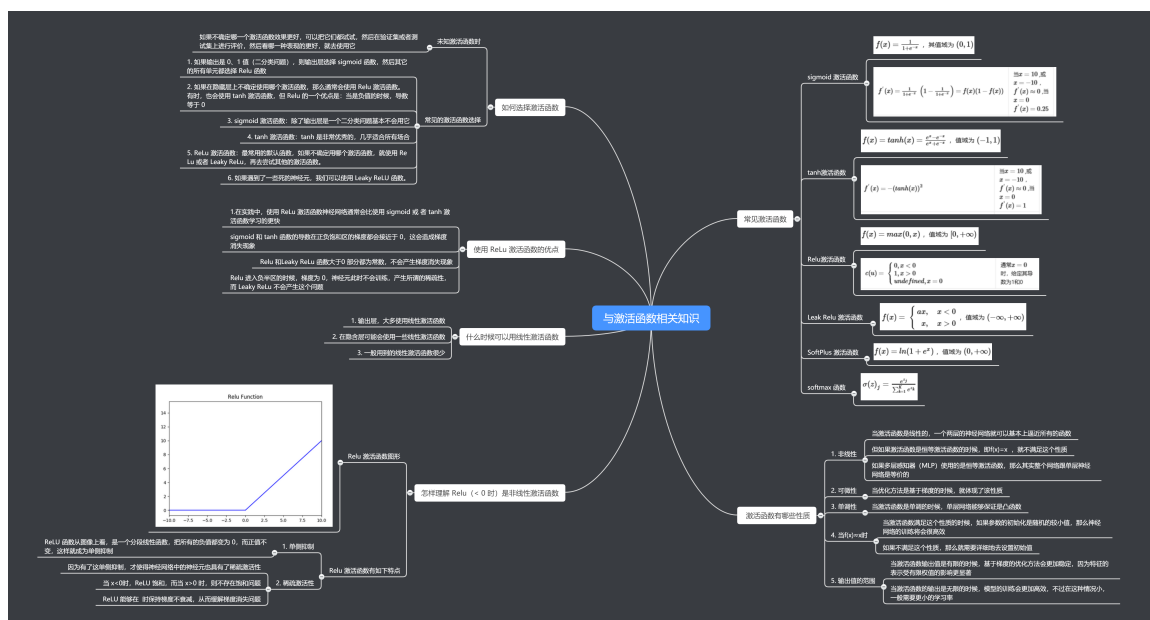
```
Args:
  x_in (torch.Tensor): an input data tensor.
  x_in.shape should be (batch, num_features)

Returns:
  the resulting tensor. tensor.shape should be (batch,)
"""
return torch.sigmoid(self.fc1(x_in)).squeeze()
```

线性运算 $w_vec^T \cdot x_vec + b$ 称为仿射变换。PyTorch 方便地在 `torch` 中提供了一个 `Linear()` 类。 `nn` 模块，它做权值和偏差所需的簿记，并做所需的仿射变换。在“深入到有监督的训练”中，您将看到如何从数据中“学习”权重 w 和 b 的值。前面示例中使用的激活函数是 `sigmoid` 函数。在下一节中，我们将回顾一些常见的激活函数，包括 `sigmoid` 函数。

激活函数

激活函数是神经网络中引入的非线性函数，用于捕获数据中的复杂关系。在“深入到有监督的训练”和“多层感知器”中，我们深入研究了为什么学习中需要非线性，但首先，让我们看看一些常用的激活函数。



Sigmoid

sigmoid 是神经网络历史上最早使用的激活函数之一。它取任何实值并将其压缩在 0 和 1 之间。数学上，sigmoid 的表达式如下：

从表达式中很容易看出，sigmoid 是一个光滑的、可微的函数。Torch 将 sigmoid 实现为 `Torch.sigmoid()`，如示例 3-2 所示。

示例 3-2: Sigmoid 激活


```
import torch
import matplotlib.pyplot as plt

x = torch.range(-5., 5., 0.1)
y = torch.sigmoid(x)
plt.plot(x.numpy(), y.numpy())
plt.show()
```

从图中可以看出，sigmoid 函数饱和（即，产生极值输出）非常快，对于大多数输入。这可能成为一个问题，因为它可能导致梯度变为零或发散到溢出的浮点值。这些现象分别被称为消失梯度问题和爆炸梯度问题。因此，在神经网络中，除了在输出端使用 sigmoid 单元外，很少看到其他使用 sigmoid 单元的情况，在输出端，压缩属性允许将输出解释为概率。

Tanh

如例 3-3 所示，tanh 激活函数是 sigmoid 在外观上的不同变体。当你写下 tanh 的表达式时，这就变得很清楚了：

通过一些争论（我们留作练习），您可以确信 tanh 只是 sigmoid 的一个线性变换。当您为 `tanh()` 写下 PyTorch 代码并绘制曲线时，这一点也很明显。注意双曲正切，像 sigmoid，也是一个“压缩”函数，除了它映射一个实值集合从 $(-\infty, +\infty)$ 到 $(-1, +1)$ 范围。

示例 3-3：Tanh 激活

```
import torch
import matplotlib.pyplot as plt

x = torch.range(-5., 5., 0.1)
y = torch.tanh(x)
plt.plot(x.numpy(), y.numpy())
plt.show()
```

ReLU

ReLU（发音为 ray-luh）代表线性整流单元。这可以说是最重要的激活函数。事实上，我们可以大胆地说，如果没有使用 ReLU，许多最近在深度学习方面的创新都是不可能实现的。对于一些如此基础的东西来说，神经网络激活函数的出现也是令人惊讶的。它的形式也出奇的简单：

$f(x) = \max(0, x)$ 因此，ReLU 单元所做的就是将负值裁剪为零，如示例 3-4 所示。

示例 3-4：ReLU 激活

```
import torch
import matplotlib.pyplot as plt

relu = torch.nn.ReLU()
x = torch.range(-5., 5., 0.1)
y = relu(x)
```

```
plt.plot(x.numpy(), y.numpy())
plt.show()
```

ReLU 的裁剪效果有助于消除梯度问题，随着时间的推移，网络中的某些输出可能会变成零，再也不会恢复。这就是所谓的“ReLU 死亡”问题。为了减轻这种影响，提出了 Leaky ReLU 或 Parametric ReLU (PReLU) 等变体，其中泄漏系数 a 是一个可学习参数: $f(x)=\max(x, ax)$

```
import torch
import matplotlib.pyplot as plt

prelu = torch.nn.PReLU(num_parameters=1)
x = torch.range(-5., 5., 0.1)
y = prelu(x)

plt.plot(x.numpy(), y.numpy())
plt.show()
```

Softmax

激活函数的另一个选择是 softmax。与 sigmoid 函数类似，softmax 函数将每个单元的输出压缩为 0 到 1 之间。然而，softmax 操作还将每个输出除以所有输出的和，从而得到一个离散概率分布，除以 k 个可能的类。结果分布中的概率总和为 1。这对于解释分类任务的输出非常有用，因此这种转换通常与概率训练目标配对，例如分类交叉熵，它在“深入研究监督训练”中介绍

```
Input[0]
import torch.nn as nn
import torch

softmax = nn.Softmax(dim=1)
x_input = torch.randn(1, 3)
y_output = softmax(x_input)
print(x_input)
print(y_output)
print(torch.sum(y_output, dim=1))
Output[0]
tensor([[ 0.5836, -1.3749, -1.1229]])
tensor([[ 0.7561,  0.1067,  0.1372]])
tensor([ 1.])
```

在本节中，我们研究了四个重要的激活函数:Sigmoid、Tanh、ReLU 和 softmax。这些只是你在构建神经网络时可能用到的四种激活方式。随着本书的深入，我们将会清楚地看到应该使用哪些激活函数以及在哪里使用，但是一般的指南只是简单地遵循过去的工作原理。

损失函数

在第 1 章中，我们看到了通用的监督机器学习架构，以及损失函数或目标函数如何通过查看数据来帮助指导训练算法选择正确的参数。回想一下，一个损失函数将真相（ y ）和预测（ \hat{y} ）作为输入，产生一个实值的分数。这个分数越高，模型的预测就越差。PyTorch 在它的 `nn` 包中实现了许多损失函数，这些函数太过全面，这里就不介绍了，但是我们将介绍一些常用的损失函数。

均方误差损失

回归问题的网络的输出（ \hat{y} ）和目标（ y ）是连续值，一个常用的损失函数的均方误差（MSE）。

MSE 就是预测值与目标值之差的平方的平均值。还有一些其他的损失函数可以用于回归问题，例如平均绝对误差（MAE）和均方根误差（RMSE），但是它们都涉及到计算输出和目标之间的实值距离。示例 3-6 展示了如何使用 PyTorch 实现 MSE 损失。

示例 3-6：MSE 损失

```
Input[0]
import torch
import torch.nn as nn

mse_loss = nn.MSELoss()
outputs = torch.randn(3, 5, requires_grad=True)
targets = torch.randn(3, 5)
loss = mse_loss(outputs, targets)
print(loss)
Output[0]
tensor(3.8618)
```

类别交叉熵损失

分类交叉熵损失（categorical cross-entropy loss）通常用于多类分类设置，其中输出被解释为类隶属度概率的预测。目标（ y ）是 n 个元素的向量，表示所有类的真正多项分布。如果只有一个类是正确的，那么这个向量就是单热向量。网络的输出（ \hat{y} ）也是一个向量 n 个元素，但代表了网络的多项分布的预测。分类交叉熵将比较这两个向量（ y , \hat{y} ）来衡量损失：

交叉熵和它的表达式起源于信息论，但是为了本节的目的，把它看作一种计算两个分布有多不同的方法是有帮助的。我们希望正确的类的概率接近 1，而其他类的概率接近 0。

为了正确地使用 PyTorch 的交叉熵损失，一定程度上理解网络输出、损失函数的计算方法和来自真正表示浮点数的各种计算约束之间的关系是很重要的。具体来说，有四条信息决定了网络输出和损失函数之间微妙的关系。首先，一个数字的大小是有限制的。其次，如果 softmax 公式中使用的指数函数的输入是负数，则结果是一个指数小的数，如果是正数，则结果是一个指数大的数。接下来，假定网络的输出是应用 softmax 函数之前的向量。最后，对数函数是指数函数的

倒数,和 $\log(\exp(x))$ 就等于 x 。因这四个信息,数学简化假设 指数函数和对数函数是为了更稳定的数值计算和避免很小或很大的数字。这些简化的结果是,不使用 softmax 函数的网络输出可以与 PyTorch 的交叉熵损失一起使用,从而优化概率分布。然后,当网络经过训练后,可以使用 softmax 函数创建概率分布,如例 3-7 所示。

示例 3-7: 交叉熵损失

```
Input[0]
import torch
import torch.nn as nn

ce_loss = nn.CrossEntropyLoss()
outputs = torch.randn(3, 5, requires_grad=True)
targets = torch.tensor([1, 0, 3], dtype=torch.int64)
loss = ce_loss(outputs, targets)
print(loss)
Output[0]
tensor(2.7256)
```

二元交叉熵

我们在上一节看到的分类交叉熵损失函数在 we 有多个类的分类问题中非常有用。有时,我们的任务包括区分两个类——也称为二元分类。在这种情况下,利用二元交叉熵损失是有效的。我们将在示例任务的“示例:对餐馆评论的情绪进行分类”中研究这个损失函数。

在示例 3-8 中,我们使用表示网络输出的随机向量上的 sigmoid 激活函数创建二进制概率输出向量。接下来,真实情况被实例化为一个 0 和 1 的向量。最后,利用二元概率向量和基真值向量计算二元交叉熵损失。

示例 3-8: 二元交叉熵损失

```
Input[0]
bce_loss = nn.BCELoss()
sigmoid = nn.Sigmoid()
probabilities = sigmoid(torch.randn(4, 1, requires_grad=True))
targets = torch.tensor([1, 0, 1, 0], dtype=torch.float32).view(4, 1)
loss = bce_loss(probabilities, targets)
print(probabilities)
print(loss)
Output[0]
tensor([[ 0.1625],
        [ 0.5546],
        [ 0.6596],
        [ 0.4284]])
tensor(0.9003)
```

深入监督学习

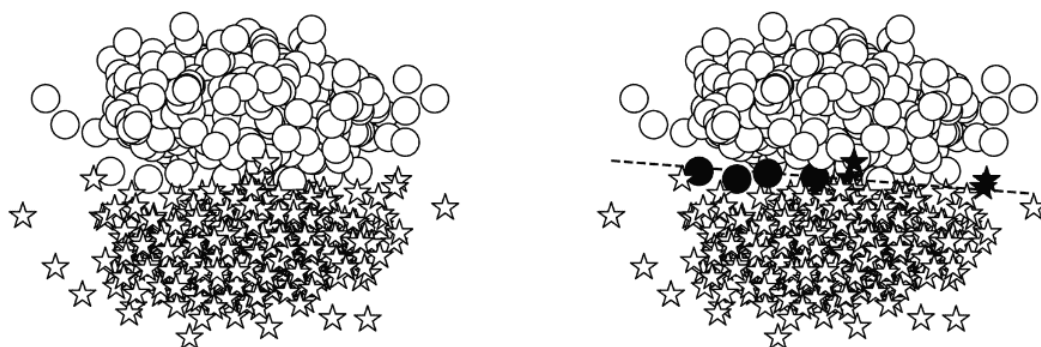
监督学习是学习如何将观察结果映射到特定目标的问题。在这一节中，我们将更详细地讨论。具体地说，我们明确地描述了如何使用模型预测和损失函数对模型参数进行基于梯度的优化。这是一个重要的部分，因为本书的其余部分都依赖于它，所以即使您对监督学习有些熟悉，也值得仔细阅读它。

回顾第 1 章，有监督学习需要以下内容：模型、损失函数、训练数据和优化算法。监督学习的训练数据是观察和目标对，模型从观察中计算预测，损失衡量预测相对于目标的误差。训练的目的是利用基于梯度的优化算法来调整模型的参数，使损失尽可能小。

在本节的其余部分中，我们将讨论一个经典的玩具问题：将二维点划分为两个类中的一个。直观上，这意味着学习一条直线（称为决策边界或超平面）来区分类之间的点。我们一步一步地描述数据结构，选择模型，选择一个损失，建立优化算法，最后，一起运行它。

构造玩具数据

在机器学习中，当试图理解一个算法时，创建具有易于理解的属性的合成数据是一种常见的实践。在本节中，我们使用“玩具”任务的合成数据——将二维点分类为两个类中的一个。为了构建数据，我们从 xy 平面的两个不同部分采样点，为模型创建了一个易于学习的环境。示例如图 3-2 所示。模型的目标是将星星（*）作为一个类，圆圈（○）作为另一个类。这可以在图的右边看到，线上面的东西和线下面的东西分类不同。生成数据的代码位于本章附带的 Python 笔记本中名为 `get_toy_data()` 的函数中。



选择模型

我们在这里使用的模型是在本章开头介绍的：感知器。感知器是灵活的，因为它允许任何大小的输入。在典型的建模情况下，输入大小由任务和数据决定。在这个玩具示例中，输入大小为 2，因为我们显式地将数据构造为二维平面。对于这个两类问题，我们为类指定一个数字索引：0 和 1。字符串的映射标签 * 和 ○ 类指数是任意的，只要它在数据预处理是一致的，训练、评估和测试。该模型的另一个重要属性是其输出的性质。由于感知器的激活函数是一个 sigmoid，感知器的输出为数据点（x）为类 1 的概率；即 $P(y = 1 \mid x)$ 。

转换概率到具体类

对于二元分类问题,我们可以输出概率转换成两个离散类通过利用决策边界 δ 。如果预测的概率 $P(y = 1 \mid x) > \delta$,预测类是 1,其它类是 0。通常,这个决策边界被设置为 0.5,但是在实践中,您可能需要优化这个超参数(使用一个评估数据集),以便在分类中获得所需的精度。

选择损失函数

在准备好数据并选择了模型体系结构之后,在有监督的训练中还可以选择另外两个重要组件:损失函数和优化器。在模型输出为概率的情况下,最合适的损失函数是基于熵的交叉损失。对于这个玩具数据示例,由于模型产生二进制结果,我们特别使用 BCE 损失。

选择优化器

在这个简化的监督训练示例中,最后的选择点是优化器。当模型产生预测,损失函数测量预测和目标之间的误差时,优化器使用错误信号更新模型的权重。最简单的形式是,有一个超参数控制优化器的更新行为。这个超参数称为学习率,它控制错误信号对更新权重的影响。学习速率是一个关键的超参数,你应该尝试几种不同的学习速率并进行比较。较大的学习率会对参数产生较大的变化,并会影响收敛性。学习率过低会导致在训练过程中进展甚微。

PyTorch 库为优化器提供了几种选择。随机梯度下降法(SGD)是一种经典的选择算法,但对于复杂的优化问题,SGD 存在收敛性问题,往往导致模型较差。当前首选的替代方案是自适应优化器,例如 Adagrad 或 Adam,它们使用关于更新的信息。在下面的例子中,我们使用 Adam,但是它总是值得查看几个优化器。对于 Adam,默认的学习率是 0.001。对于学习率之类的超参数,总是建议首先使用默认值,除非您从论文中获得了需要特定值的秘诀。

示例 3-9: 实例化 Adam 优化器

```
Input[0]
import torch.nn as nn
import torch.optim as optim

input_dim = 2
lr = 0.001

perceptron = Perceptron(input_dim=input_dim)
bce_loss = nn.BCELoss()
optimizer = optim.Adam(params=perceptron.parameters(), lr=lr)
```

放到一起：基于梯度的监督学习

学习从计算损失开始;也就是说,模型预测离目标有多远。损失函数的梯度,反过来,是参数应该改变多少的信号。每个参数的梯度表示给定参数的损失值的瞬时变化率。实际上,这意味着您

可以知道每个参数对损失函数的贡献有多大。直观上，这是一个斜率，你可以想象每个参数都站在它自己的山上，想要向上或向下移动一步。基于梯度的模型训练所涉及的最简单的形式就是迭代地更新每个参数，并使用与该参数相关的损失函数的梯度。

让我们看看这个梯度步进（gradient-steeping）算法是什么样子的。首先，使用名为 `zero_grad()` 的函数清除当前存储在模型（感知器）对象中的所有记帐信息，例如梯度。然后，模型计算给定输入数据（`x_data`）的输出（`y_pred`）。接下来，通过比较模型输出（`y_pred`）和预期目标（`y_target`）来计算损失。这正是有监督训练信号的有监督部分。PyTorch 损失对象（`criteria`）具有一个名为 `backward()` 的函数，该函数迭代地通过计算图向后传播损失，并将其梯度通知每个参数。最后，优化器（`opt`）用一个名为 `step()` 的函数指示参数如何在知道梯度的情况下更新它们的值。

整个训练数据集被划分成多个批（batch）。在文献和本书中，术语小批量也可以互换使用，而不是“批量”来强调每个批量都明显小于训练数据的大小；例如，训练数据可能有数百万个，而小批数据可能只有几百个。梯度步骤的每一次迭代都在一批数据上执行。名为 `batch_size` 的超参数指定批次的大小。由于训练数据集是固定的，增加批大小会减少批的数量。在多个批量（通常是有限大小数据集中的批量数量）之后，训练循环完成了一个周期。周期一个完整的训练迭代。如果每个周期的批数量与数据集中的批数量相同，那么周期就是对数据集的完整迭代。模型是为一定数量的周期而训练的。要训练的周期的数量对于选择来说不是复杂的，但是有一些方法可以决定什么时候停止，我们稍后将讨论这些方法。如示例 3-10 所示，受监督的训练循环因此是一个嵌套循环：数据集或批量集合上的内部循环，以及外部循环，后者在固定数量的周期或其他终止条件上重复内部循环。

示例 3-10：感知机和二分类的监督训练循环

```
# each epoch is a complete pass over the training data
for epoch_i in range(n_epochs):
    # the inner loop is over the batches in the dataset
    for batch_i in range(n_batches):

        # Step 0: Get the data
        x_data, y_target = get_toy_data(batch_size)

        # Step 1: Clear the gradients
        perceptron.zero_grad()

        # Step 2: Compute the forward pass of the model
        y_pred = perceptron(x_data, apply_sigmoid=True)

        # Step 3: Compute the loss value that we wish to optimize
        loss = bce_loss(y_pred, y_target)

        # Step 4: Propagate the loss signal backward
        loss.backward()

        # Step 5: Trigger the optimizer to perform one update
        optimizer.step()
```


辅助训练概念

基于梯度监督学习的核心概念很简单:定义模型, 计算输出, 使用损失函数计算梯度, 应用优化算法用梯度更新模型参数。然而, 在训练过程中有几个重要但辅助的概念。我们将在本节介绍其中的一些。

正确度量模型表现: 评估度量

核心监督训练循环之外最重要的部分是使用模型从未训练过的数据来客观衡量性能。模型使用一个或多个评估指标进行评估。在自然语言处理 (NLP) 中, 存在多种评价指标。最常见的, 也是我们将在本章使用的, 是准确性。准确性仅仅是在训练过程中未见的数据集上预测正确的部分。

正确度量模型表现: 分割数据集

一定要记住, 最终的目标是很好地概括数据的真实分布。这是什么意思? 假设我们能够看到无限数量的数据 (“真实/不可见的分布”), 那么存在一个全局的数据分布。显然, 我们不能那样做。相反, 我们用有限的样本作为训练数据。我们观察有限样本中的数据分布这是真实分布的近似或不完全图像。如果一个模型不仅减少了训练数据中样本的误差, 而且减少了来自不可见分布的样本的误差, 那么这个模型就比另一个模型具有更好的通用性。当模型致力于降低它在训练数据上的损失时, 它可以过度适应并适应那些实际上不是真实数据分布一部分的特性。

要实现这一点, 标准实践是将数据集分割为三个随机采样的分区, 称为训练、验证和测试数据集, 或者进行 k 折交叉验证。分成三个分区是两种方法中比较简单的一种, 因为它只需要一次计算。您应该采取预防措施, 确保在三个分支之间的类分布保持相同。换句话说, 通过类标签聚合数据集, 然后将每个由类标签分隔的集合随机拆分为训练、验证和测试数据集, 这是一种很好的实践。一个常见的分割百分比是预留 70% 用于训练, 15% 用于验证, 15% 用于测试。不过, 这不是一个硬编码的约定。

在某些情况下, 可能存在预定义的训练、验证和测试分离; 这在用于基准测试任务的数据集中很常见。在这种情况下, 重要的是只使用训练数据更新模型参数, 在每个周期结束时使用验证数据测量模型性能, 在所有的建模选择被探索并需要报告最终结果之后, 只使用测试数据一次。这最后一部分是极其重要的, 因为更多的机器学习工程师在玩模型的性能测试数据集, 他们是偏向选择测试集上表现得更好。当这种情况发生时, 它是不可能知道该模型性能上看不见的数据没有收集更多的数据。

使用 k 折交叉验证的模型评估与使用预定义分割的评估非常相似, 但是在此之前还有一个额外的步骤, 将整个数据集分割为 k 个大小相同的折。其中一折保留用于评估, 剩下的 $k-1$ 折用于训练。通过交换出计算中的哪些折, 可以重复执行此操作。因为有 k 折, 每一折都有机会成为一个评价折, 并产生一个特定于折的精度, 从而产生 k 个精度值。最终报告的准确性只是具有标准差的平均值。k 折评估在计算上是昂贵的, 但是对于较小的数据集来说是非常必要的, 对于较小的数据集来说, 错误的分割可能导致过于乐观 (因为测试数据太容易了) 或过于悲观 (因为测试数据太困难了)。

了解什么时候停止训练

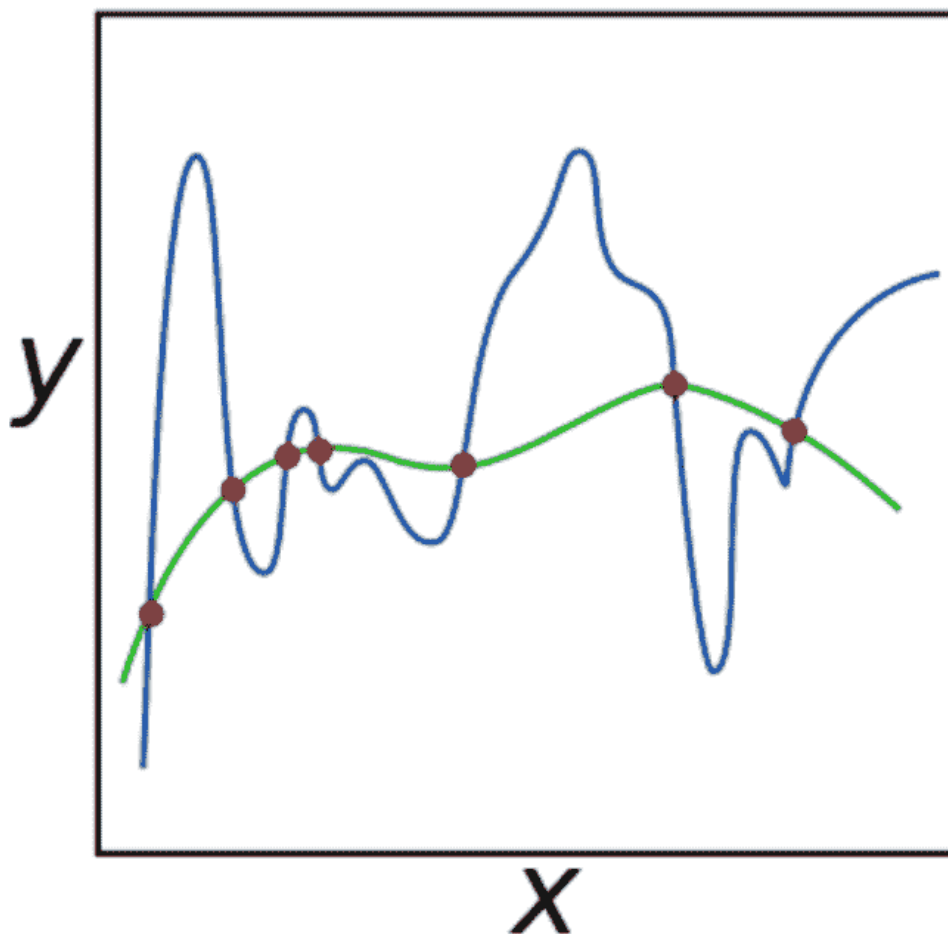
之前的例子训练了固定次数的模型。虽然这是最简单的方法，但它是任意的和不必要的。正确度量模型性能的一个关键功能是使用该度量来知道何时应该停止训练。最常用的方法是使用启发式方法，称为早期停止（early stopping）。早期停止通过跟踪验证数据集上从一个周期到另一个周期的性能并注意性能何时不再改进来的工作。然后，如果业绩继续没有改善，训练将终止。在结束训练之前需要等待的时间称为耐心。一般来说，模型停止改进某些数据集的时间点称为模型收敛的时间点。在实际应用中，我们很少等待模型完全收敛，因为收敛是耗时的，而且会导致过拟合。

查找正确的超参数

我们在前面了解到，参数（或权重）采用优化器针对称为小批量的固定训练数据子集调整的实际值。超参数是影响模型中参数数量和参数所取值的任何模型设置。有许多不同的选择来决定如何训练模型。这些选择包括选择一个损失函数;优化器;优化器的学习率，如层大小（在第 4 章中介绍);有等到早停止（early stopping）的耐心;和各种正规化决策（也在第 4 章讨论）。需要注意的是，这些决策会对模型是否收敛及其性能产生很大影响，你应该系统地探索各种选择点。

正则化

深度学习（以及机器学习）中最重要的概念之一是正则化。正则化的概念来源于数值优化理论。回想一下，大多数机器学习算法都在优化损失函数，以找到最可能解释观测结果（即，产生的损失最少）。对于大多数数据集和任务，这个优化问题可能有多个解决方案（可能的模型）。那么我们（或优化器）应该选择哪一个呢?为了形成直观的理解，请考虑图 3-3 通过一组点拟合曲线的任务。



两条曲线都“拟合”这些点，但哪一条是不太可能的解释呢？通过求助于奥卡姆剃刀，我们凭直觉知道一个简单的解释比复杂的解释更好。这种机器学习中的平滑约束称为 L2 正则化。在 PyTorch 中，您可以通过在优化器中设置 `weight_decay` 参数来控制这一点。`weight_decay` 值越大，优化器选择的解释就越流畅；也就是说，L2 正则化越强。

除了 L2，另一种流行的正则化是 L1 正则化。L1 通常用来鼓励稀疏解；换句话说，大多数模型参数值都接近于零。在第 4 章中，您将看到一种结构正则化技术，称为“丢弃”。模型正则化是一个活跃的研究领域，PyTorch 是实现自定义正则化的灵活框架。

示例：分类餐馆评论的情感

在上一节中，我们通过一个玩具示例深入研究了有监督的训练，并阐述了许多基本概念。在本节中，我们将重复上述练习，但这次使用的是一个真实的任务和数据集：使用感知器和监督训练对 Yelp 上的餐馆评论进行分类，判断它们是正面的还是负面的。因为这是本书中第一个完整的 NLP 示例，所以我们将极其详细地描述辅助数据结构和训练例程。后面几章中的示例将遵循非常相似的模式，因此我们鼓励您仔细遵循本节，并在需要复习时参考它。

在本书的每个示例的开头，我们将描述正在使用的数据集和任务。在这个例子中，我们使用 Yelp 数据集，它将评论与它们的情感标签（正面或负面）配对。此外，我们还描述了一些数据集操作步骤，这些步骤用于清理数据集并将其划分为训练、验证和测试集。

在理解数据集之后，您将看到定义三个辅助类的模式，这三个类在本书中反复出现，用于将文本数据转换为向量化形式：词汇表（the Vocabulary）、向量化器（Vectorizer）和 PyTorch 的 `DataLoader`。词汇表协调我们在“观察和目标编码”中讨论的整数到标记（token）映射。我们使用一个词汇表将文本标记（text tokens）映射到整数，并将类标签映射到整数。接下来，向量化器（vectorizer）封装词汇表，并负责接收字符串数据，如审阅文本，并将其转换为将在训练例程中使用的数字向量。我们使用最后一个辅助类，PyTorch 的 `DataLoader`，将单个向量化数据点分组并整理成小批量。

在描述了构成文本向量化小批量管道（text-to-vectorized-minibatch pipeline）的数据集和辅助类之后，概述了感知器分类器及其训练例程。需要注意的重要一点是，本书中的每个示例的训练例程基本保持不变。我们会在这个例子中更详细地讨论它，因此，我们再次鼓励您使用这个例子作为未来训练例程的参考。我们通过讨论结果来总结这个例子，并深入了解模型学习到了什么。

Yelp 评论数据集

2015 年，Yelp 举办了一场竞赛，要求参与者根据点评预测一家餐厅的评级。同年，Zhang, Zhao, 和 Lecun（2015）将 1 星和 2 星评级转换为“消极”情绪类，将 3 星和 4 星评级转换为“积极”情绪类，从而简化了数据集。该数据集分为 56 万个训练样本和 3.8 万个测试样本。在这个数据集部分的其余部分中，我们将描述最小化清理数据并导出最终数据集的过程。然后，我们概述了利用 PyTorch 的数据集类的实现。

在这个例子中，我们使用了简化的 Yelp 数据集，但是有两个细微的区别。第一个区别是我们使用数据集的“轻量级”版本，它是通过选择 10% 的训练样本作为完整数据集而派生出来的。这两个结果：首先，使用一个小数据集可以使训练测试循环快速，因此我们可以快速地进行实验。其次，它生成的模型精度低于使用所有数据。这种低精度通常不是主要问题，因为您可以使用从较小数据集子集中获得的知识对整个数据集进行重新训练。在训练深度学习模型时，这是一个非常实用的技巧，因为在许多情况下，训练数据的数量是巨大的。

从这个较小的子集中，我们将数据集分成三个分区：一个用于训练，一个用于验证，一个用于测试。虽然原始数据集只有两个部分，但是有一个验证集是很重要的。在机器学习中，您经常在数据集的训练部分上训练模型，并且需要一个保留部分来评估模型的性能。如果模型决策基于保留部分，那么模型现在不可避免地偏向于更好地执行保留部分。因为度量增量进度是至关重要的，所以这个问题的解决方案是使用第三个部分，它尽可能少地用于评估。

综上所述，您应该使用数据集的训练部分来派生模型参数，使用数据集的验证部分在超参数之间进行选择（进行建模决策），使用数据集的测试分区进行最终评估和报告。在例 3-11 中，我们展示了如何分割数据集。注意，随机种子被设置为一个静态数字，我们首先通过类标签聚合以确保类分布保持不变。

示例 3-11：创建训练，验证和测试分割

```
# Splitting the subset by rating to create new train, val, and test splits
by_rating = collections.defaultdict(list)
for _, row in review_subset.iterrows():
    by_rating[row.rating].append(row.to_dict())

# Create split data
final_list = []
np.random.seed(args.seed)

for _, item_list in sorted(by_rating.items()):
    np.random.shuffle(item_list)

    n_total = len(item_list)
    n_train = int(args.train_proportion * n_total)
    n_val = int(args.val_proportion * n_total)
    n_test = int(args.test_proportion * n_total)

    # Give data point a split attribute
    for item in item_list[:n_train]:
        item['split'] = 'train'

    for item in item_list[n_train:n_train+n_val]:
        item['split'] = 'val'

    for item in item_list[n_train+n_val:n_train+n_val+n_test]:
        item['split'] = 'test'

    # Add to final list
    final_list.extend(item_list)

final_reviews = pd.DataFrame(final_list)
```

除了创建一个子集，该子集有三个分区用于训练、验证和测试之外，我们还通过在标点符号周围添加空格和删除并非所有分割都使用标点符号的无关符号来最低限度地清理数据，如示例 3-12 所示。

示例 3-12：最小程度清理数据

```
def preprocess_text(text):
    text = text.lower()
    text = re.sub(r"([.,!?])", r" \1 ", text)
    text = re.sub(r"^[a-zA-Z.,!?]+", r" ", text)
    return text

final_reviews.review = final_reviews.review.apply(preprocess_text)
```

理解 PyTorch 的数据集表示

示例 3-13 中给出的 `ReviewDataset` 类假设数据集已被最少地清理并分割为三个部分。特别是，数据集假定它可以基于空白分隔评论，以便获得评论中的标记（tokens）列表。此外，它假定数

据有一个注释，该注释将数据拆分为它所属的部分。需要注意的是，我们使用 Python 的类方法为这个数据集类指定了入口点方法。我们在整本书中都遵循这个模式。

PyTorch 通过提供数据集类为数据集提供了一个抽象。数据集类是一个抽象迭代器。在对新数据集使用 PyTorch 时，必须首先从数据集类继承子类（或继承），并实现 `__getitem__` 和 `__len__` 方法。对于这个例子，我们创建了一个 `ReviewDataset` 类，它继承自 PyTorch 的 `Dataset` 类，并实现了两个方法: `__getitem__` 和 `__len__`。通过实现这两种方法，有一个概念上的约定，允许各种 PyTorch 实用程序使用我们的数据集。在下一节中，我们将介绍其中一个实用程序，特别是 `DataLoader`。下面的实现严重依赖于一个名为 `ReviewVectorizer` 的类。在下一节中，我们将描述 `ReviewVectorizer`，但是您可以直观地将其描述为处理从评审文本到表示评审的数字向量的转换的类。神经网络只有通过一定的向量化步骤才能与文本数据进行交互。总体设计模式是实现一个数据集类，它处理一个数据点的向量化逻辑。然后，PyTorch 的 `DataLoader`（下一节也将介绍）将通过对数据集进行采样和整理来创建小批数据（minibatch）。

示例 3-13: `ReviewDataset` 类

```
from torch.utils.data import Dataset

class ReviewDataset(Dataset):
    def __init__(self, review_df, vectorizer):
        """
        Args:
            review_df (pandas.DataFrame): the dataset
            vectorizer (ReviewVectorizer): vectorizer instantiated from
dataset
        """
        self.review_df = review_df
        self._vectorizer = vectorizer

        self.train_df = self.review_df[self.review_df.split=='train']
        self.train_size = len(self.train_df)

        self.val_df = self.review_df[self.review_df.split=='val']
        self.validation_size = len(self.val_df)

        self.test_df = self.review_df[self.review_df.split=='test']
        self.test_size = len(self.test_df)

        self._lookup_dict = {'train': (self.train_df, self.train_size),
                              'val': (self.val_df, self.validation_size),
                              'test': (self.test_df, self.test_size)}

        self.set_split('train')

    @classmethod
    def load_dataset_and_make_vectorizer(cls, review_csv):
        """Load dataset and make a new vectorizer from scratch

        Args:
            review_csv (str): location of the dataset
```

```

Returns:
    an instance of ReviewDataset
"""

review_df = pd.read_csv(review_csv)
return cls(review_df, ReviewVectorizer.from_dataframe(review_df))

def get_vectorizer(self):
    """ returns the vectorizer """
    return self._vectorizer

def set_split(self, split="train"):
    """ selects the splits in the dataset using a column in the dataframe

    Args:
        split (str): one of "train", "val", or "test"
    """
    self._target_split = split
    self._target_df, self._target_size = self._lookup_dict[split]

def __len__(self):
    return self._target_size

def __getitem__(self, index):
    """the primary entry point method for PyTorch datasets

    Args:
        index (int): the index to the data point
    Returns:
        a dict of the data point's features (x_data) and label (y_target)
    """
    row = self._target_df.iloc[index]

    review_vector = \
        self._vectorizer.vectorize(row.review)

    rating_index = \
        self._vectorizer.rating_vocab.lookup_token(row.rating)

    return {'x_data': review_vector,
            'y_target': rating_index}

def get_num_batches(self, batch_size):
    """Given a batch size, return the number of batches in the dataset

    Args:
        batch_size (int)
    Returns:
        number of batches in the dataset
    """
    return len(self) // batch_size

```

Vocabulary, Vectorizer和DataLoader

Vocabulary, Vectorizer 和 DataLoader 是三个类，我们几乎在本书的每个示例中都使用它们来执行一个关键的管道:将文本输入转换为向量化的小批（minibatch）。管道从预处理文本开始;

每个数据点都是标记的集合。在本例中，标记碰巧是单词，但是正如您将在第 4 章和第 6 章中看到的，标记也可以是字符。以下小节中提供的三个类负责将每个标记映射到一个整数，将此映射应用到每个数据点，以创建一个向量化表单，然后将向量化数据点分组到模型的一个小批量中。

词汇表

从文本到向量化的小批量处理的第一步是将每个标记（tokens）映射到其自身的数字版本。标准的方法是在标记（tokens）和整数之间有一个双向映射（可以反向映射）。在 Python 中，这只是两个字典。我们将这个词封装到词汇表类中，如示例 3-14 所示。词汇表类不仅管理这个双射—允许用户添加新的标记并使索引自动递增—而且还处理一个名为 `UNK.UNK` 的特殊标记，它代表“未知”标记。通过使用 `UNK` 标记，我们可以在测试期间处理训练中从未见过的标记；例如，您可能会遇到一个以前从未见过的单词。正如我们将在接下来的向量化器中看到的，我们甚至将显式地限制词汇表中不经常出现的标记，以便在我们的训练例程中有 `UNK` 标记。这对于限制词汇表类使用的内存非常重要。预期的行为是调用 `add_token` 向词汇表中添加新的标记，检索标记索引时调用 `lookup_token`，检索特定索引对应的标记时调用 `lookup_index`。

示例 3-14: `Vocabulary` 类

```
class Vocabulary(object):
    """Class to process text and extract vocabulary for mapping"""

    def __init__(self, token_to_idx=None, add_unk=True, unk_token="<UNK>"):
        """
        Args:
            token_to_idx (dict): a pre-existing map of tokens to indices
            add_unk (bool): a flag that indicates whether to add the UNK token
            unk_token (str): the UNK token to add into the Vocabulary
        """

        if token_to_idx is None:
            token_to_idx = {}
        self._token_to_idx = token_to_idx

        self._idx_to_token = {idx: token
                              for token, idx in self._token_to_idx.items()}

        self._add_unk = add_unk
        self._unk_token = unk_token

        self.unk_index = -1
        if add_unk:
            self.unk_index = self.add_token(unk_token)

    def to_serializable(self):
        """ returns a dictionary that can be serialized """
        return {'token_to_idx': self._token_to_idx,
                'add_unk': self._add_unk,
                'unk_token': self._unk_token}
```

```

@classmethod
def from_serializable(cls, contents):
    """ instantiates the Vocabulary from a serialized dictionary """
    return cls(**contents)

def add_token(self, token):
    """Update mapping dicts based on the token.

    Args:
        token (str): the item to add into the Vocabulary
    Returns:
        index (int): the integer corresponding to the token
    """
    if token in self._token_to_idx:
        index = self._token_to_idx[token]
    else:
        index = len(self._token_to_idx)
        self._token_to_idx[token] = index
        self._idx_to_token[index] = token
    return index

def lookup_token(self, token):
    """Retrieve the index associated with the token
    or the UNK index if token isn't present.

    Args:
        token (str): the token to look up
    Returns:
        index (int): the index corresponding to the token
    Notes:
        `unk_index` needs to be >=0 (having been added into the
Vocabulary)
        for the UNK functionality
    """
    if self.add_unk:
        return self._token_to_idx.get(token, self.unk_index)
    else:
        return self._token_to_idx[token]

def lookup_index(self, index):
    """Return the token associated with the index

    Args:
        index (int): the index to look up
    Returns:
        token (str): the token corresponding to the index
    Raises:
        KeyError: if the index is not in the Vocabulary
    """
    if index not in self._idx_to_token:
        raise KeyError("the index (%d) is not in the Vocabulary" % index)
    return self._idx_to_token[index]

def __str__(self):
    return "<Vocabulary(size=%d)>" % len(self)

```



```
def __len__(self):
    return len(self._token_to_idx)
```

向量化

从文本数据集到向量化的小批量的第二个阶段是迭代输入数据点的标记，并将每个标记转换为其整数形式。这个迭代的结果应该是一个向量。由于这个向量将与来自其他数据点的向量组合，因此有一个约束条件，即由向量化器生成的向量应该始终具有相同的长度。

为了实现这些目标，`Vectorizer` 类封装了评审词汇表，它将评审中的单词映射到整数。在示例 3-15 中，向量化器为 `from_dataframe` 方法使用 Python 的 `classmethod` 装饰器来指示实例化向量化器的入口点。`from_dataframe` 方法在 pandas `dataframe` 的行上迭代，有两个目标。第一个目标是计算数据集中出现的所有标记的频率。第二个目标是创建一个词汇表，该词汇表只使用与 `from_dataframe` 方法截止提供的关键字参数一样频繁的标记。有效地，这种方法是找到所有至少出现截止时间的单词，并将它们添加到词汇表中。由于还将 `UNK` 标记添加到词汇表中，因此在调用词汇表的 `lookup_` 标记方法时，未添加的任何单词都将具有 `unk_index`。

方法向量化封装了向量化器的核心功能。它以表示评审的字符串作为参数，并返回评审的向量化表示。在这个例子中，我们使用在第 1 章中介绍的折叠的单热表示。这种表示方式创建了一个二进制向量——一个包含 1 和 0 的向量——它的长度等于词汇表的大小。二进制向量在与复习中的单词对应的位置有 1。注意，这种表示有一些限制。首先，它是稀疏的——复习中惟一单词的数量总是远远少于词汇表中惟一单词的数量。第二，它抛弃了单词在评论中出现的顺序（词袋，“bag of words”）。在后面的章节中，您将看到其他没有这些限制的方法。

示例 3-15: `Vectorizer` 类

```
class ReviewDataset(Dataset):
    def __init__(self, review_df, vectorizer):
        """
        Args:
            review_df (pandas.DataFrame): the dataset
            vectorizer (ReviewVectorizer): vectorizer instantiated from
dataset
        """
        self.review_df = review_df
        self._vectorizer = vectorizer

        self.train_df = self.review_df[self.review_df.split=='train']
        self.train_size = len(self.train_df)

        self.val_df = self.review_df[self.review_df.split=='val']
        self.validation_size = len(self.val_df)

        self.test_df = self.review_df[self.review_df.split=='test']
        self.test_size = len(self.test_df)

        self._lookup_dict = {'train': (self.train_df, self.train_size),
                              'val': (self.val_df, self.validation_size),
```

```

        'test': (self.test_df, self.test_size)}

        self.set_split('train')

    @classmethod
    def load_dataset_and_make_vectorizer(cls, review_csv):
        """Load dataset and make a new vectorizer from scratch

        Args:
            review_csv (str): location of the dataset
        Returns:
            an instance of ReviewDataset
        """
        review_df = pd.read_csv(review_csv)
        train_review_df = review_df[review_df.split=='train']
        return cls(review_df,
ReviewVectorizer.from_dataframe(train_review_df))

    @classmethod
    def load_dataset_and_load_vectorizer(cls, review_csv,
vectorizer_filepath):
        """Load dataset and the corresponding vectorizer.
        Used in the case in the vectorizer has been cached for re-use

        Args:
            review_csv (str): location of the dataset
            vectorizer_filepath (str): location of the saved vectorizer
        Returns:
            an instance of ReviewDataset
        """
        review_df = pd.read_csv(review_csv)
        vectorizer = cls.load_vectorizer_only(vectorizer_filepath)
        return cls(review_df, vectorizer)

    @staticmethod
    def load_vectorizer_only(vectorizer_filepath):
        """a static method for loading the vectorizer from file

        Args:
            vectorizer_filepath (str): the location of the serialized
vectorizer
        Returns:
            an instance of ReviewVectorizer
        """
        with open(vectorizer_filepath) as fp:
            return ReviewVectorizer.from_serializable(json.load(fp))

    def save_vectorizer(self, vectorizer_filepath):
        """saves the vectorizer to disk using json

        Args:
            vectorizer_filepath (str): the location to save the vectorizer
        """
        with open(vectorizer_filepath, "w") as fp:
            json.dump(self._vectorizer.to_serializable(), fp)

    def get_vectorizer(self):

```

```

        """ returns the vectorizer """
        return self._vectorizer

    def set_split(self, split="train"):
        """selects the splits in the dataset using a column in the
        dataframe"""
        self._target_split = split
        self._target_df, self._target_size = self._lookup_dict[split]

    def __len__(self):
        return self._target_size

    def __getitem__(self, index):
        """the primary entry point method for PyTorch datasets

        Args:
            index (int): the index to the data point
        Returns:
            a dict of the data point's features (x_data) and label (y_target)
        """
        row = self._target_df.iloc[index]

        review_vector = \
            self._vectorizer.vectorize(row.review)

        rating_index = \
            self._vectorizer.rating_vocab.lookup_token(row.rating)

        return {'x_data': review_vector,
                'y_target': rating_index}

    def get_num_batches(self, batch_size):
        """Given a batch size, return the number of batches in the dataset

        Args:
            batch_size (int)
        Returns:
            number of batches in the dataset
        """
        return len(self) // batch_size

```

DataLoader

文本向量化的小批量的最后一个阶段是对向量化的数据点进行分组。因为分组成小批是训练神经网络的重要部分，所以 PyTorch 提供了一个名为 `DataLoader` 的内置类来协调这个过程。

`DataLoader` 类通过提供一个 PyTorch 数据集（例如为本例定义的 `ReviewDataset`）、一个 `batch_size` 和一些其他关键字参数来实例化。得到的对象是一个 Python 迭代器，它对数据集 19 中提供的数据点进行分组和整理。在示例 3-16 中，我们将 `DataLoader` 包装在 `generate_batch()` 函数中，该函数是一个生成器，用于方便地在 CPU 和 GPU 之间切换数据。

示例 3-16: `generate_batches` 函数

```
def generate_batches(dataset, batch_size, shuffle=True,
                    drop_last=True, device="cpu"):
    """
    A generator function which wraps the PyTorch DataLoader. It will
    ensure each tensor is on the write device location.
    """
    dataloader = DataLoader(dataset=dataset, batch_size=batch_size,
                            shuffle=shuffle, drop_last=drop_last)

    for data_dict in dataloader:
        out_data_dict = {}
        for name, tensor in data_dict.items():
            out_data_dict[name] = data_dict[name].to(device)
        yield out_data_dict
```

感知机分类器

我们在这里使用的模型是我们在本章开头展示的感知器的重新实现。`ReviewClassifier` 继承自 PyTorch 的模块，并创建具有单个输出的单个线性层。因为这是一个二元分类设置（消极或积极的审查），所以这是一个适当的设置。最终的非线性函数为 sigmoid 函数。

我们对 `forward` 方法进行参数化，以允许可选地应用 sigmoid 函数。要理解其中的原因，首先需要指出的是，在二元分类任务中，二元交叉熵损失（`torch.nn.BCELoss`）是最合适的损失函数。它是用数学公式表示二进制概率的。然而，应用一个 Sigmoid 然后使用这个损失函数存在数值稳定性问题。为了给用户提供更稳定的快捷方式，PyTorch 提供了 `BCEWithLogitsLoss`。要使用这个损失函数，输出不应该应用 sigmoid 函数。因此，在默认情况下，我们不应用 sigmoid。但是，如果分类器的用户希望得到一个概率值，则需要使用 sigmoid，并将其作为选项保留。在示例 3-17 的结果部分中，我们看到了以这种方式使用它的示例。

示例 3-17：感知机分类器

```
import torch.nn as nn
import torch.nn.functional as F

class ReviewClassifier(nn.Module):
    """ a simple perceptron based classifier """
    def __init__(self, num_features):
        """
        Args:
            num_features (int): the size of the input feature vector
        """
        super(ReviewClassifier, self).__init__()
        self.fc1 = nn.Linear(in_features=num_features,
                              out_features=1)

    def forward(self, x_in, apply_sigmoid=False):
        """The forward pass of the classifier

        Args:
            x_in (torch.Tensor): an input data tensor.
```

```

        x_in.shape should be (batch, num_features)
        apply_sigmoid (bool): a flag for the sigmoid activation
            should be false if used with the Cross Entropy losses
    Returns:
        the resulting tensor. tensor.shape should be (batch,)
    """
    y_out = self.fc1(x_in).squeeze()
    if apply_sigmoid:
        y_out = F.sigmoid(y_out)
    return y_out

```

训练例程

在本节中，我们将概述训练例程的组件，以及它们如何与数据集和模型结合来调整模型参数并提高其性能。在其核心，训练例程负责实例化模型，在数据集上迭代，在给定数据作为输入时计算模型的输出，计算损失（模型的错误程度），并根据损失比例更新模型。虽然这可能看起来有很多细节需要管理，但是改变训练常规的地方并不多，因此，在您的深度学习开发过程中，这将成为一种习惯。为了帮助管理高层决策，我们使用 `args` 对象集中协调所有决策点，您可以在示例 3-18 中看到。

示例 3-18：用于分类 Yelp 评论的参数

```

from argparse import Namespace

args = Namespace(
    # Data and Path information
    frequency_cutoff=25,
    model_state_file='model.pth',
    review_csv='data/yelp/reviews_with_splits_lite.csv',
    save_dir='model_storage/ch3/yelp/',
    vectorizer_file='vectorizer.json',
    # No Model hyper parameters
    # Training hyper parameters
    batch_size=128,
    early_stopping_criteria=5,
    learning_rate=0.001,
    num_epochs=100,
    seed=1337,
    # ... runtime options omitted for space
)

```

在本节的其余部分中，我们首先描述训练状态，这是一个用于跟踪关于训练过程的信息的小字典。当您跟踪关于训练例程的更多细节时，这个字典将会增长，如果您选择这样做，您可以系统化它，但是在下一个示例中给出的字典是您将在模型训练期间跟踪的基本信息集。在描述了训练状态之后，我们将概述为要执行的模型训练实例化的对象集。这包括模型本身、数据集、优化器和损失函数。在其他示例和补充材料中，我们包含了其他组件，但为了简单起见，我们不在文本中列出它们。最后，我们用训练循环本身结束本节，并演示标准 PyTorch 优化模式。

设置阶段来启动训练

示例 3-19 展示了我们为这个示例实例化的训练组件。第一项是初始训练状态。该函数接受 `args` 对象作为参数，以便训练状态能够处理复杂的信息，但是在本书的文本中，我们没有展示这些复杂性。我们建议您参考补充材料，看看您在训练状态下还可以使用哪些额外的东西。这里显示的最小集包括训练损失、训练精度、验证损失和验证精度的周期索引和列表。它还包括测试损失和测试精度两个字段。

接下来要实例化的两个项目是数据集和模型。在本例中，以及本书其余部分的示例中，我们将数据集设计为负责实例化向量化器。在补充材料中，数据集实例化嵌套在一个 `if` 语句中，该 `if` 语句允许加载以前实例化的向量化器，或者一个新的实例化，该实例化器也将保存到磁盘。重要的是，通过协调用户的意愿（通过 `args.cuda`）和检查 GPU 设备是否确实可用的条件，将模型移动到正确的设备。目标设备用于核心训练循环中的 `generate_batch` 函数调用，以便数据和模型将位于相同的设备位置。

初始实例化中的最后两项是损失函数和优化器。本例中使用的损失函数是 `bcewithlogits` 损失。要更详细地解释为什么使用这种损失，请参考“感知器分类器”，它描述了模型。简而言之，最合适的损失函数的二元分类是二进制交叉熵（BCE）损失和更数值稳定对 `BCEWithLogitsLoss` 的模型不适用 Sigmoid 函数输出比一对 `BCELoss` 模型，并应用 sigmoid 函数的输出。我们使用的优化器是 Adam 优化器。一般来说，Adam 与其他优化器相比具有很强的竞争力，在撰写本文时，还没有令人信服的证据表明可以使用任何其他优化器来替代 Adam。我们鼓励您通过尝试其他优化器并注意性能来验证这一点。

示例 3-19：实例化数据集，模型，损失，优化器和训练状态

```
import torch.optim as optim

def make_train_state(args):
    return {'epoch_index': 0,
            'train_loss': [],
            'train_acc': [],
            'val_loss': [],
            'val_acc': [],
            'test_loss': -1,
            'test_acc': -1}
train_state = make_train_state(args)

if not torch.cuda.is_available():
    args.cuda = False
args.device = torch.device("cuda" if args.cuda else "cpu")

# dataset and vectorizer
dataset = ReviewDataset.load_dataset_and_make_vectorizer(args.review_csv)
vectorizer = dataset.get_vectorizer()

# model
classifier = ReviewClassifier(num_features=len(vectorizer.review_vocab))
classifier = classifier.to(args.device)
```

```
# loss and optimizer
loss_func = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(classifier.parameters(), lr=args.learning_rate)
```

训练循环

训练循环使用来自初始实例化的对象来更新模型参数，以便随着时间的推移进行改进。更具体地说，训练循环由两个循环组成：一个内循环覆盖数据集中的小批量，另一个外循环重复内循环若干次。在内部循环中，计算每个小批量的损失，并使用优化器更新模型参数。

为了更全面地介绍所发生的事情，让我们从下面的代码片段顶部开始。在第一行中，我们使用 `for` 循环，它的范围跨越各个周期。周期的数量是一个可以设置的超参数。它控制训练例程应该对数据集进行多少次传递。在实践中，您应该使用类似于早期停止标准的东西来在循环结束之前终止它。在补充资料中，我们向您展示了如何做到这一点。

在 `for` 循环的顶部，有几个例程定义和实例化。首先设置训练状态的周期索引，然后设置数据集的分割（首先是 `"train"`，然后是 `"val"`，当我们想在周期结束时测量模型性能，最后是 `"test"`，当我们想评估模型的最终性能）。考虑到我们是如何构造数据集的，应该总是在调用 `generate_batch()` 之前设置拆分。创建 `batch_generator` 之后，将实例化两个浮动，以跟踪批量之间的损失和准确性。有关这里使用的“运行平均公式”的更多细节，请参阅 Wikipedia 的 `moving average` 页面。最后，调用分类器的 `.train()` 方法，表示模型处于“训练模式”，模型参数是可变的。这也支持像丢弃这样的正则化机制。

训练循环的下一部分是迭代 `batch_generator` 中的训练批，并执行更新模型参数的基本操作。在每个批量迭代中，首先使用 `optimizer.zero_grad()` 方法重置优化器的梯度。然后，从模型中计算输出。接下来，损失函数用于计算模型输出与监督目标（真正的类标签）之间的损失。在此之后，对损失对象（而不是损失函数对象）调用 `loss.backward()` 方法，导致梯度传播到每个参数。最后，优化器使用这些传播的梯度来使用 `optimizer.step()` 方法执行参数更新。这五个步骤是梯度下降的基本步骤。除此之外，还有一些用于记帐和跟踪的额外操作。具体来说，损失和精度值（作为常规 Python 变量存储）被计算出来，然后用于更新运行损失和运行精度变量。

在训练分割批量的内部循环之后，有两个簿记和实例化操作。具体来说，首先用最终的损失和精度值更新训练状态。然后，创建一个新的批量生成器、运行损失和运行精度。验证数据的循环几乎与训练数据相同，因此重用相同的变量。有一个主要的区别：调用分类器的 `.eval()` 方法，它执行与分类器的 `.train()` 方法相反的操作。`eval()` 方法使模型参数不可变，且不可丢失。求值模式还禁止计算梯度的损失并将其传播回参数。这很重要，因为我们不希望模型根据验证数据调整参数。相反，我们希望这些数据作为模型执行情况的度量。如果其测量性能之间存在着很大的差异在训练数据和验证数据的测量性能,很可能模型过度拟合训练数据,你应该调整模型或训练程序（比如设置阻止早期,我们使用补充笔记本对于这个例子）。

在对验证数据进行迭代并保存由此产生的验证损失和精度值之后，外部 `for` 循环就完成了。我们在本书中实现的每个训练例程都将遵循非常相似的设计模式。事实上，所有梯度下降算法都遵循相似的设计模式。在您习惯了从头开始编写这个循环之后，您将会学会如何使用它执行梯度下降！示例 3-20 给出了代码。

示例 3-20：粗糙的训练循环

[illegible]


```

device=args.device)

running_loss = 0.
running_acc = 0.
classifier.eval()

for batch_index, batch_dict in enumerate(batch_generator):

    # step 1\ . compute the output
    y_pred = classifier(x_in=batch_dict['x_data'].float())

    # step 2\ . compute the loss
    loss = loss_func(y_pred, batch_dict['y_target'].float())
    loss_batch = loss.item()
    running_loss += (loss_batch - running_loss) / (batch_index + 1)

    # step 3\ . compute the accuracy
    acc_batch = compute_accuracy(y_pred, batch_dict['y_target'])
    running_acc += (acc_batch - running_acc) / (batch_index + 1)

train_state['val_loss'].append(running_loss)
train_state['val_acc'].append(running_acc)

```

评估，推断和检查

在您有了一个经过训练的模型之后，接下来的步骤是要么评估它是如何处理一些保留下来的数据的，要么使用它对新数据进行推断，要么检查模型的权重，看看它学到了什么。在本节中，我们将向您展示所有三个步骤。

在测试数据上评估

为了评估外置测试集上的数据，代码与我们在上一个示例中看到的训练例程中的验证循环完全相同，但有一个细微的区别:分割设置为 "test" 而不是 "val"。数据集的两个分区之间的区别在于，测试集应该尽可能少地运行。每次您在测试集上运行一个训练过的模型，做出一个新的模型决策（例如改变层的大小），并在测试集上重新测量新的再训练模型时，您都是在向测试数据倾斜您的建模决策。换句话说，如果您足够频繁地重复这个过程，那么测试集作为真正交付数据的精确度量将变得毫无意义。示例 3-21 对此进行了更深入的研究。

示例 3-21：测试集评估

```

Input[0]
dataset.set_split('test')
batch_generator = generate_batches(dataset,
                                   batch_size=args.batch_size,
                                   device=args.device)

running_loss = 0.
running_acc = 0.
classifier.eval()

for batch_index, batch_dict in enumerate(batch_generator):

```

```

# compute the output
y_pred = classifier(x_in=batch_dict['x_data'].float())

# compute the loss
loss = loss_func(y_pred, batch_dict['y_target'].float())
loss_batch = loss.item()
running_loss += (loss_batch - running_loss) / (batch_index + 1)

# compute the accuracy
acc_batch = compute_accuracy(y_pred, batch_dict['y_target'])
running_acc += (acc_batch - running_acc) / (batch_index + 1)

train_state['test_loss'] = running_loss
train_state['test_acc'] = running_acc
Input[1]
print("Test loss: {:.3f}".format(train_state['test_loss']))
print("Test Accuracy: {:.2f}".format(train_state['test_acc']))
Output[1]
Test loss: 0.297
Test Accuracy: 90.55

```

推断和分类新的数据点

评价模型的另一种方法是对新数据进行推断，并对模型是否有效进行定性判断。我们可以在示例 3-22 中看到这一点。

示例 3-22：打印样本评论的预测

```

Input[0]
def predict_rating(review, classifier, vectorizer,
                  decision_threshold=0.5):
    """Predict the rating of a review

    Args:
        review (str): the text of the review
        classifier (ReviewClassifier): the trained model
        vectorizer (ReviewVectorizer): the corresponding vectorizer
        decision_threshold (float): The numerical boundary which
            separates the rating classes
    """

    review = preprocess_text(review)
    vectorized_review = torch.tensor(vectorizer.vectorize(review))
    result = classifier(vectorized_review.view(1, -1))

    probability_value = F.sigmoid(result).item()

    index = 1
    if probability_value < decision_threshold:
        index = 0

    return vectorizer.rating_vocab.lookup_index(index)

```

```

test_review = "this is a pretty awesome book"
prediction = predict_rating(test_review, classifier, vectorizer)
print("{} -> {}".format(test_review, prediction))
Output[0]
this is a pretty awesome book -> positive

```

检查模型权重

最后，了解模型在完成训练后是否表现良好的最后一种方法是检查权重，并对权重是否正确做出定性判断。如示例 3-23 所示，使用感知器和压缩的单热编码，这是一种相当简单的方法，因为每个模型的权重与词汇表中的单词完全对应。

示例 3-23：打印对应分类器权重的单词

```

Input[0]
# Sort weights
fc1_weights = classifier.fc1.weight.detach()[0]
_, indices = torch.sort(fc1_weights, dim=0, descending=True)
indices = indices.numpy().tolist()

# Top 20 words
print("Influential words in Positive Reviews:")
print("-----")
for i in range(20):
    print(vectorizer.review_vocab.lookup_index(indices[i]))
Output[0]
Influential words in Positive Reviews:
-----
great
awesome
amazing
love
friendly
delicious
best
excellent
definitely
perfect
fantastic
wonderful
vegas
favorite
loved
yummy
fresh
reasonable
always
recommend
Input[1]
# Top 20 negative words
print("Influential words in Negative Reviews:")
print("-----")
indices.reverse()

```

```
for i in range(20):
    print(vectorizer.review_vocab.lookup_index(indices[i]))
Output[1]
Influential words in Negative Reviews:
-----
worst
horrible
mediocre
terrible
not
rude
bland
disgusting
dirty
awful
poor
disappointing
ok
no
overpriced
sorry
nothing
meh
manager
gross
```

总结

在这一章中，你学习了监督神经网络训练的一些基本概念：

1. 最简单的神经网络模型，感知器
2. 基本概念如激活函数、损失函数及其不同种类
3. 在一个玩具示例的上下文中，训练循环、批大小和时间
4. 泛化是什么意思，以及使用训练/测试/验证分割来衡量泛化性能的良好实践
5. 早期停止等准则来确定训练算法的端点或收敛性 什么是超参数和他们的一些例子，如批大小，学习率等等
6. 如何使用 PyTorch 实现的感知器模型对英文 Yelp 餐厅评论进行分类，如何通过检验权重来解释该模型

在第 4 章中，我们介绍了前馈网络，首先在不起眼的感知器模型的基础上，通过纵向和横向叠加来建立前馈网络，从而得到多层感知器模型。我们还研究了一种新的基于卷积运算的前馈网络来捕获语言子结构。

四、自然语言处理的前馈网络

本文标题: [Natural-Language-Processing-with-PyTorch \(四\)](#)

文章作者: [Yif Du](#)

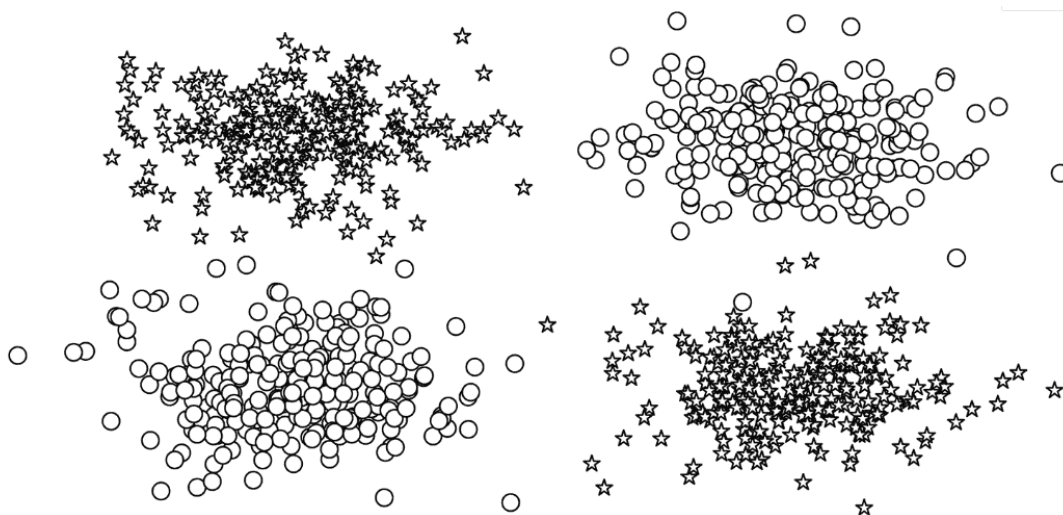
发布时间: 2018 年 12 月 20 日 - 09:12

最后更新: 2018 年 12 月 28 日 - 11:12

原始链接: <http://yifdu.github.io/2018/12/20/Natural-Language-Processing-with-PyTorch> (四) /

许可协议: [署名-非商业性使用-禁止演绎 4.0 国际](#) 转载请保留原文链接及作者。

在第 3 章中, 我们通过观察感知器来介绍神经网络的基础, 感知器是现存最简单的神经网络。感知器的一个历史性的缺点是它不能学习数据中存在的一些非常重要的模式。例如, 查看图 4-1 中绘制的数据点。这相当于非此即彼 (XOR) 的情况, 在这种情况下, 决策边界不能是一条直线 (也称为线性可分)。在这个例子中, 感知器失败了。



在这一章中, 我们将探索传统上称为前馈网络的神经网络模型, 以及两种前馈神经网络: 多层感知器和卷积神经网络。多层感知器在结构上扩展了我们在第 3 章中研究的简单感知器, 将多个感知器分组在一个单层, 并将多个层叠加在一起。我们稍后将介绍多层感知器, 并在“示例: 带有多层感知器的姓氏分类”中展示它们在多层分类中的应用。

本章研究的第二种前馈神经网络, 卷积神经网络, 在处理数字信号时深受窗口滤波器的启发。通过这种窗口特性, 卷积神经网络能够在输入中学习局部化模式, 这不仅使其成为计算机视觉的主轴, 而且是检测单词和句子等序列数据中的子结构的理想候选。我们在“卷积神经网络”中概述了卷积神经网络, 并在“示例: 使用 CNN 对姓氏进行分类”中演示了它们的使用。

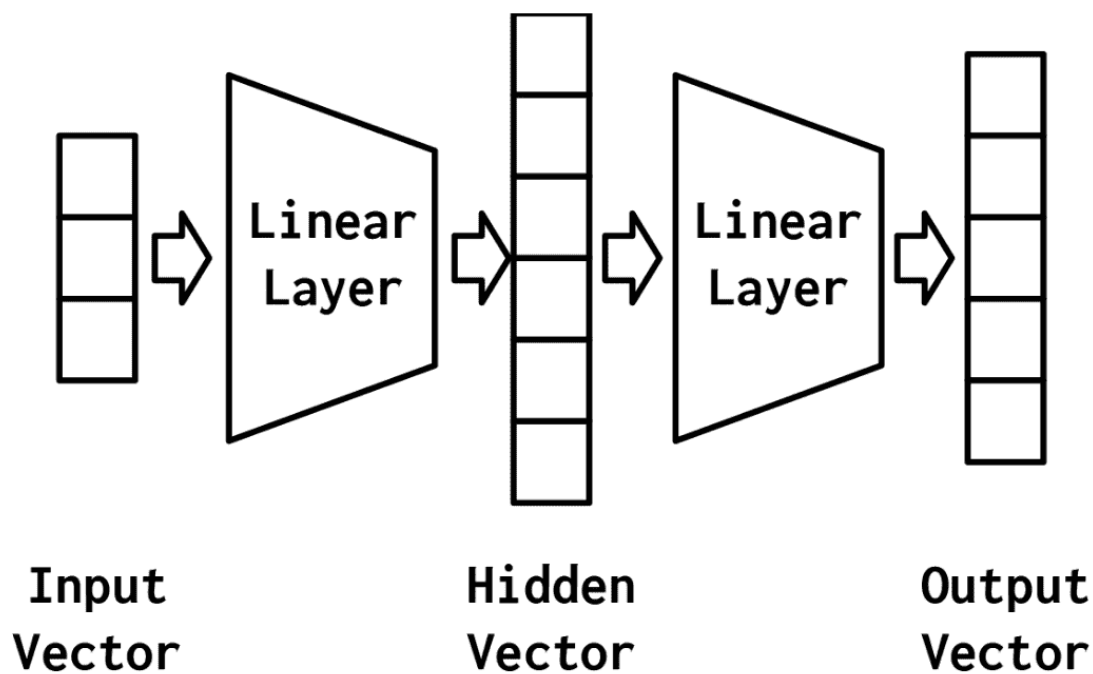
在本章中，多层感知器和卷积神经网络被分组在一起，因为它们都是前馈神经网络，并且与另一类神经网络——循环神经网络（RNNs）形成对比，循环神经网络（RNNs）允许反馈（或循环），这样每次计算都可以从之前的计算中获得信息。在第 6 章和第 7 章中，我们将介绍 RNNs 以及为什么允许网络结构中的循环是有益的。

在我们介绍这些不同的模型时，确保您理解事物如何工作的一个有用方法是在计算数据张量时注意它们的大小和形状。每种类型的神经网络层对它所计算的数据张量的大小和形状都有特定的影响，理解这种影响可以极大地有助于对这些模型的深入理解。

多层感知机

多层感知器（MLP）被认为是最基本的神经网络构建模块之一。最简单的 MLP 是对第 3 章感知器的扩展。感知器将数据向量作为输入，计算出一个输出值。在 MLP 中，许多感知器被分组，以便单个层的输出是一个新的向量，而不是单个输出值。在 PyTorch 中，正如您稍后将看到的，这只需设置线性层中的输出特性的数量即可完成。MLP 的另一个方面是，它将多个层与每个层之间的非线性结合在一起。

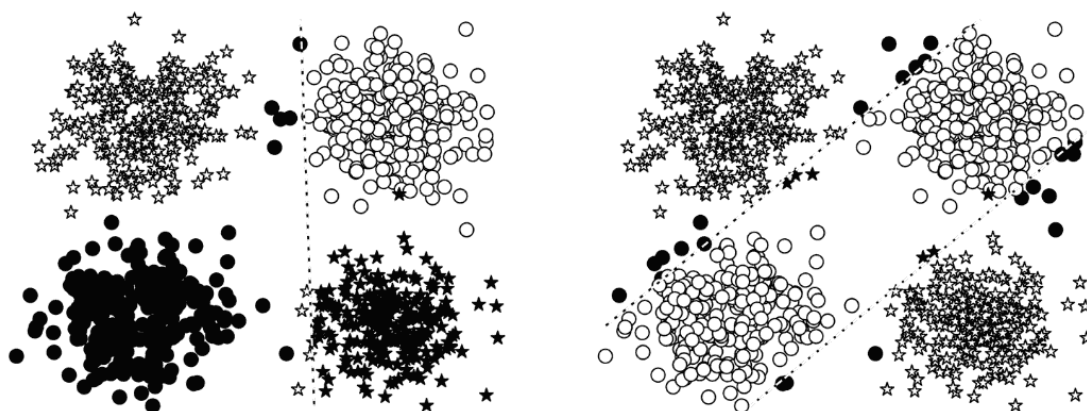
最简单的 MLP，如图 4-2 所示，由三个表示阶段和两个线性层组成。第一阶段是输入向量。这是给定给模型的向量。在“示例:对餐馆评论的情绪进行分类”中，输入向量是 Yelp 评论的一个收缩的单热表示。给定输入向量，第一个线性层计算一个隐藏向量——表示的第二阶段。隐藏向量之所以这样被调用，是因为它是位于输入和输出之间的层的输出。我们所说的“层的输出”是什么意思?理解这个的一种方法是隐藏向量中的值是组成该层的不同感知器的输出。使用这个隐藏的向量，第二个线性层计算一个输出向量。在像 Yelp 评论分类这样的二进制任务中，输出向量仍然可以是 1。在多类设置中，您将在本章后面的“示例:带有多层感知器的姓氏分类”一节中看到，输出向量是类数量的大小。虽然在这个例子中，我们只展示了一个隐藏的向量，但是有可能有多个中间阶段，每个阶段产生自己的隐藏向量。最终的隐藏向量总是通过线性层和非线性的组合映射到输出向量。



mlp 的力量来自于添加第二个线性层和允许模型学习一个线性分割的的中间表示——该属性的能表示一个直线（或更一般的,一个超平面）可以用来区分数据点落在线（或超平面）的哪一边的。学习具有特定属性的中间表示，如分类任务是线性可分的，这是使用神经网络的最深刻后果之一，也是其建模能力的精髓。在下一节中，我们将更深入地研究这意味着什么。

简单示例：XOR

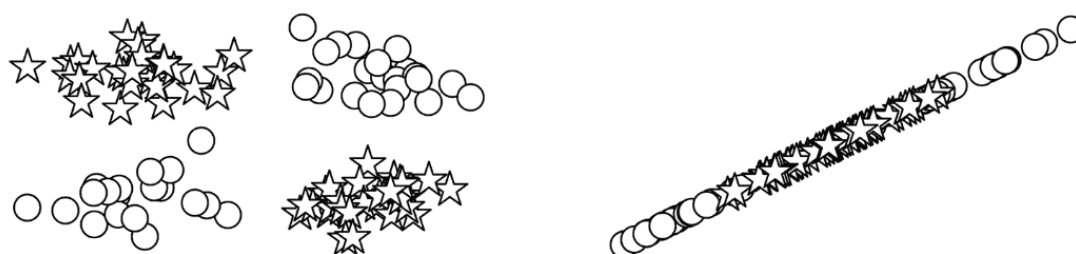
让我们看一下前面描述的 XOR 示例，看看感知器与 MLP 之间会发生什么。在这个例子中，我们 在一个二元分类任务中训练感知器和 MLP:星和圆。每个数据点是一个二维坐标。在不深入研究 实现细节的情况下，最终的模型预测如图 4-3 所示。在这个图中，错误分类的数据点用黑色填充，而正确分类的数据点没有填充。在左边的面板中，从填充的形状可以看出，感知器在学习一个可以将星星和圆分开的决策边界方面有困难。然而，MLP（右面板）学习了一个更精确地对恒星和圆进行分类的决策边界。



虽然在图中显示 MLP 有两个决策边界，这是它的优点，但它实际上只是一个决策边界!决策边界就是这样出现的，因为中间表示法改变了空间，使一个超平面同时出现在这两个位置上。在图 4-4 中，我们可以看到 MLP 计算的中间值。这些点的形状表示类（星形或圆形）。我们所看到的是，神经网络（本例中为 MLP）已经学会了“扭曲”数据所处的空间，以便在数据通过最后一层时，用一线来分割它们。MLP 的输入和中间表示是可视化的。从左到右：（1）网络的输入，（2）第一个线性模块的输出，（3）第一个非线性模块的输出，（4）第二个线性模块的输出。如您所见，第一个线性模块的输出将圆和星分组，而第二个线性模块的输出将数据点重新组织为线性可分的。

相反，如图 4-5 所示，感知器没有额外的一层来处理数据的形状，直到数据变成线性可分的。

The Perceptron's Input and Intermediate Representation



在 PyTorch 中实现 MLP

在上一节中，我们概述了 MLP 的核心思想。在本节中，我们将介绍 PyTorch 中的一个实现。如前所述，MLP 除了第 3 章中简单的感知器之外，还有一个额外的计算层。我们在例 4-1 中给出的实现中，我们用 PyTorch 的两个线性模块实例化了这个想法。线性对象被命名为 `fc1` 和 `fc2`，它们遵循一个通用约定，即将线性模块称为“完全连接层”，简称为“fc 层”。除了这两个线性层外，还有一个修正的线性单元（ReLU）非线性（在第 3 章“激活函数”一节中介绍），它在被输入到第二个线性层之前应用于第一个线性层的输出。由于层的顺序性，您必须确保层中的输出数量等于下一层的输入数量。使用两个线性层之间的非线性是必要的，因为没有它，两个线性层在数学上等价于一个线性层⁴，因此不能建模复杂的模式。MLP 的实现只实现反向传播的前向传递。这是因为 PyTorch 根据模型的定义和向前传递的实现，自动计算出如何进行向后传递和梯度更新。

示例 4-1：多层感知机

```
import torch.nn as nn
import torch.nn.functional as F

class MultilayerPerceptron(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        """
        Args:
            input_dim (int): the size of the input vectors
```

```

        hidden_dim (int): the output size of the first Linear layer
        output_dim (int): the output size of the second Linear layer
    """
    super(MultilayerPerceptron, self).__init__()
    self.fc1 = nn.Linear(input_dim, hidden_dim)
    self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x_in, apply_softmax=False):
        """The forward pass of the MLP

        Args:
            x_in (torch.Tensor): an input data tensor.
                x_in.shape should be (batch, input_dim)
            apply_softmax (bool): a flag for the softmax activation
                should be false if used with the Cross Entropy losses
        Returns:
            the resulting tensor. tensor.shape should be (batch, output_dim)
        """
        intermediate = F.relu(self.fc1(x_in))
        output = self.fc2(intermediate)

        if apply_softmax:
            output = F.softmax(output, dim=1)
        return output

```

在例 4-2 中，我们实例化了 MLP。由于 MLP 实现的通用性，我们可以为任何大小的输入建模。为了演示，我们使用大小为 3 的输入维度、大小为 4 的输出维度和大小为 100 的隐藏维度。请注意，在 `print` 语句的输出中，每个层中的单元数很好地排列在一起，以便为维度 3 的输入生成维度 4 的输出。

示例 4-2: NLP 的示例实例化

```

Input[0]
batch_size = 2 # number of samples input at once
input_dim = 3
hidden_dim = 100
output_dim = 4

# Initialize model
mlp = MultilayerPerceptron(input_dim, hidden_dim, output_dim)
print(mlp)
Output[0]
MultilayerPerceptron(
  (fc1): Linear(in_features=3, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=4, bias=True)
  (relu): ReLU()
)

```

我们可以通过传递一些随机输入来快速测试模型的“连接”，如示例 4-3 所示。因为模型还没有经过训练，所以输出是随机的。在花费时间训练模型之前，这样做是一个有用的完整性检查。请注意 PyTorch 的交互性是如何让我们在开发过程中实时完成所有这些工作的，这与使用 NumPy 或 panda 没有太大区别：

示例 4-3：使用随机输入测试 MLP

```
Input[0]
def describe(x):
    print("Type: {}".format(x.type()))
    print("Shape/size: {}".format(x.shape))
    print("Values: \n{}".format(x))

x_input = torch.rand(batch_size, input_dim)
describe(x_input)
Output[0]
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.8329,  0.4277,  0.4363],
        [ 0.9686,  0.6316,  0.8494]])
Input[1]
y_output = mlp(x_input, apply_softmax=False)
describe(y_output)
Output[1]
Type: torch.FloatTensor
Shape/size: torch.Size([2, 4])
Values:
tensor([[ -0.2456,  0.0723,  0.1589, -0.3294],
        [-0.3497,  0.0828,  0.3391, -0.4271]])
```

学习如何读取 PyTorch 模型的输入和输出非常重要。在前面的例子中，MLP 模型的输出是一个有两行四列的张量。这个张量中的行与批量维数对应，批量维数是小批量中的数据点的数量。列是每个数据点的最终特征向量。在某些情况下，例如在分类设置中，特征向量是一个预测向量。名称为“预测向量”表示它对应于一个概率分布。预测向量会发生什么取决于我们当前是在进行训练还是在执行推理。在训练期间，输出按原样使用，带有一个损失函数和目标类标签的表示。我们将在“示例:带有多层感知器的姓氏分类”中对此进行深入介绍。

但是，如果您想将预测向量转换为概率，则需要额外的步骤。具体来说，您需要 softmax 函数，它用于将一个值向量转换为概率。softmax 有许多根。在物理学中，它被称为玻尔兹曼或吉布斯分布;在统计学中，它是多项式逻辑回归;在自然语言处理（NLP）社区，它是最大熵（MaxEnt）分类器。不管叫什么名字，这个函数背后的直觉是，大的正值会导致更高的概率，小的负值会导致更小的概率。在示例 4-3 中，`apply_softmax` 参数应用了这个额外的步骤。在例 4-4 中，您可以看到相同的输出，但是这次将 `apply_softmax` 标志设置为 `True`：

示例 4-4：带有 `apply_softmax=True` 的 MLP

```
Input[0]
y_output = mlp(x_input, apply_softmax=True)
describe(y_output)
Output[0]
Type: torch.FloatTensor
Shape/size: torch.Size([2, 4])
Values:
tensor([[ 0.2087,  0.2868,  0.3127,  0.1919],
        [ 0.1832,  0.2824,  0.3649,  0.1696]])
```

综上所述，mlp 是将张量映射到其他张量的线性层。在每一对线性层之间使用非线性来打破线性关系，并允许模型扭曲向量空间。在分类设置中，这种扭曲应该导致类之间的线性可分性。另外，您可以使用 softmax 函数将 MLP 输出解释为概率，但是不应该将 softmax 与特定的损失函数一起使用，因为底层实现可以利用高级数学/计算捷径。

Example: Surname Classification with a Multilayer Perceptron

在本节中，我们将 MLP 应用于将姓氏分类到其原籍国的任务。从公开观察到的数据推断人口统计信息（如国籍）具有从产品推荐到确保不同人口统计用户获得公平结果的应用。人口统计和其他自我识别信息统称为“受保护属性”。“在建模和产品中使用这些属性时，必须小心。”我们首先对每个姓氏的字符进行拆分，并像对待“示例:将餐馆评论的情绪分类”中的单词一样对待它们。除了数据上的差异，字符层模型在结构和实现上与基于单词的模型基本相似。

您应该从这个例子中吸取的一个重要教训是，MLP 的实现和训练是从我们在第 3 章中看到的感知器的实现和训练直接发展而来的。事实上，我们在本书的第 3 章中提到了这个例子，以便更全面地了解这些组件。此外，我们不包括您可以在“例子:餐馆评论的情绪分类”中看到的代码，如果你想在一个地方看到例子代码，我们强烈建议你跟随补充材料。

本节的其余部分将从姓氏数据集及其预处理步骤的描述开始。然后，我们使用 Vocabulary，Vectorizer 和 DataLoader 类逐步完成从姓氏字符串到向量化小批量的管道。如果您通读了第 3 章，您应该将这些辅助类视为老朋友，只是做了一些小小的修改。

我们将通过描述姓氏分类器模型及其设计背后的思想过程来继续本节。MLP 类似于我们在第 3 章中看到的感知器例子，但是除了模型的改变，我们在这个例子中引入了多类输出及其对应的损失函数。在描述了模型之后，我们完成了训练例程。训练程序与您在“示例:对餐馆评论的情绪进行分类”中看到的非常相似，因此为了简洁起见，我们在这里不像在该部分中那样深入。我们强烈建议您回顾这一节以获得更多的澄清。

姓氏数据集

在这个例子中，我们介绍了一个姓氏数据集，它收集了来自 18 个不同国家的 10,000 个姓氏，这些姓氏是作者从互联网上不同的姓名来源收集的。该数据集将在本书的几个示例中重用，并具

有一些使其有趣的属性。第一个性质是它是相当不平衡的。排名前三的课程占数据的 60% 以上：27% 是英语，21% 是俄语，14% 是阿拉伯语。剩下的 15 个民族的频率也在下降——这也是语言特有的特性。第二个特点是，在国籍和姓氏正字法（拼写）之间有一种有效和直观的关系。有些拼写变体与原籍国联系非常紧密（比如 O'Neill、Antonopoulos、Nagasawa 或 Zhu）。

为了创建最终的数据集，我们从一个比本书补充材料中包含的版本处理更少的版本开始，并执行了几个数据集修改操作。第一个目的是减少这种不平衡——原始数据集中 70% 以上是俄文，这可能是由于抽样偏差或俄文姓氏的增多。为此，我们通过选择标记为俄语的姓氏的随机子集对这个过度代表的类进行子样本。接下来，我们根据国籍对数据集进行分组，并将数据集分为三个部分：70% 到训练数据集，15% 到验证数据集，最后 15% 到测试数据集，以便跨这些部分的类标签分布具有可比性。

`SurnameDataset` 的实现与“示例：餐馆评论的情感分类”中的 `ReviewDataset` 几乎相同，只是在 `__getitem__` 方法的实现方式上略有不同。回想一下，本书中呈现的数据集类继承自 PyTorch 的数据集类，因此，我们需要实现两个函数：`__getitem__` 方法，它在给定索引时返回一个数据点；以及 `__len__` 方法，该方法返回数据集的长度。“示例：餐厅评论的情绪分类”中的示例与本示例的区别在 `__getitem__` 中，如示例 4-5 所示。它不像“示例：将餐馆评论的情绪分类”那样返回一个向量化的评论，而是返回一个向量化的姓氏和与其国籍相对应的索引：

示例 4-5：实现 `SurnameDataset.__getitem__()`

```
class SurnameDataset(Dataset):
    # Implementation is nearly identical to Section 3.5

    def __getitem__(self, index):
        row = self._target_df.iloc[index]
        surname_vector = \
            self._vectorizer.vectorize(row.surname)
        nationality_index = \
            self._vectorizer.nationality_vocab.lookup_token(row.nationality)

        return {'x_surname': surname_vector,
                'y_nationality': nationality_index}
```

Vocabulary, Vectorizer 和 DataLoader

为了使用字符对姓氏进行分类，我们使用 `Vocabulary`，`Vectorizer` 和 `DataLoader` 将姓氏字符串转换为向量化的的小批量。这些数据结构与“示例：餐馆评论的情感分类”中使用的数据结构相同，它们举例说明了一种多态性，这种多态性将姓氏的字符标记与 Yelp 评论的单词标记相同对待。数据不是通过将字标记映射到整数来向量化的，而是通过将字符映射到整数来向量化的。

词汇表的类

本例中使用的词汇类与“示例：餐馆评论的情感分类”中的词汇完全相同，该词汇类将 Yelp 评论中的单词映射到对应的整数。简要概述一下，词汇表是两个 Python 字典的协调，这两个字典在标记（在本例中是字符）和整数之间形成一个双射；也就是说，第一个字典将字符映射到整数索引，第二个字典将整数索引映射到字符。`add_token` 方法用于向词汇表中添加新的标记，`lookup_token` 方法用于检索索引，`lookup_index` 方法用于检索给定索引的标记（在推断阶段很有用）。与 Yelp 评论的词汇表不同，我们使用的是单热词汇表，不计算字符出现的频率，只对频繁出现的条目进行限制。这主要是因为数据集很小，而且大多数字符足够频繁。

SurnameVectorizer

虽然词汇表将单个标记（字符）转换为整数，但 `SurnameVectorizer` 负责应用词汇表并将姓氏转换为向量。实例化和使用非常类似于“示例：对餐馆评论的情绪进行分类”中的 `ReviewVectorizer`，但有一个关键区别：字符串没有在空格上分割。姓氏是字符的序列，每个字符在我们的词汇表中是一个单独的标记。然而，在“卷积神经网络”出现之前，我们将忽略序列信息，通过迭代字符串输入中的每个字符来创建输入的收缩单热向量表示。我们为以前未遇到的字符指定一个特殊的标记，即 `UNK`。由于我们仅从训练数据实例化词汇表，而且验证或测试数据中可能有惟一的字符，所以在字符词汇表中仍然使用 `UNK` 符号。

您应该注意，虽然我们在这个示例中使用了收缩的单热，但是在后面的章节中，您将了解其他向量化方法，它们是单热编码的替代方法，有时甚至更好。具体来说，在“示例：使用 CNN 对姓氏进行分类”中，您将看到一个热门矩阵，其中每个字符都是矩阵中的一个位置，并具有自己的热门向量。然后，在第 5 章中，您将学习嵌入层，返回整数向量的向量化，以及如何使用它们创建密集向量矩阵。但是现在，让我们看一下示例 4-6 中 `SurnameVectorizer` 的代码。

示例 4-6：实现 `SurnameVectorizer`

```
class SurnameVectorizer(object):
    """ The Vectorizer which coordinates the Vocabularies and puts them to
    use """
    def __init__(self, surname_vocab, nationality_vocab):
        self.surname_vocab = surname_vocab
        self.nationality_vocab = nationality_vocab

    def vectorize(self, surname):
        """Vectorize the provided surname

        Args:
            surname (str): the surname
        Returns:
            one_hot (np.ndarray): a collapsed onehot encoding
        """
        vocab = self.surname_vocab
        one_hot = np.zeros(len(vocab), dtype=np.float32)
        for token in surname:
```

```

        one_hot[vocab.lookup_token(token)] = 1
    return one_hot

@classmethod
def from_dataframe(cls, surname_df):
    """Instantiate the vectorizer from the dataset dataframe

    Args:
        surname_df (pandas.DataFrame): the surnames dataset
    Returns:
        an instance of the SurnameVectorizer
    """
    surname_vocab = Vocabulary(unk_token="@")
    nationality_vocab = Vocabulary(add_unk=False)

    for index, row in surname_df.iterrows():
        for letter in row.surname:
            surname_vocab.add_token(letter)
        nationality_vocab.add_token(row.nationality)

    return cls(surname_vocab, nationality_vocab)

```

姓氏分类器模型

`SurnameClassifier` 是本章前面介绍的 MLP 的实现（示例 4-7）。第一个线性层将输入向量映射到中间向量，并对该向量应用非线性。第二线性层将中间向量映射到预测向量。

在最后一步中，可选地应用 softmax 操作，以确保输出和为 1；这就是所谓的“概率”。它是可选的原因与我们使用的损失函数的数学公式有关——交叉熵损失。我们研究了“损失函数”中的交叉熵损失。回想一下，交叉熵损失对于多类分类是最理想的，但是在训练过程中软最大值的计算不仅浪费而且在很多情况下并不稳定。

示例 4-7：作为 MLP 的 `SurnameClassifier`

```

import torch.nn as nn
import torch.nn.functional as F

class SurnameClassifier(nn.Module):
    """ A 2-layer Multilayer Perceptron for classifying surnames """
    def __init__(self, input_dim, hidden_dim, output_dim):
        """
        Args:
            input_dim (int): the size of the input vectors
            hidden_dim (int): the output size of the first Linear layer
            output_dim (int): the output size of the second Linear layer
        """
        super(SurnameClassifier, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x_in, apply_softmax=False):
        """The forward pass of the classifier

```



```

    Args:
        x_in (torch.Tensor): an input data tensor.
            x_in.shape should be (batch, input_dim)
        apply_softmax (bool): a flag for the softmax activation
            should be false if used with the Cross Entropy losses
    Returns:
        the resulting tensor. tensor.shape should be (batch, output_dim)
    """
    intermediate_vector = F.relu(self.fc1(x_in))
    prediction_vector = self.fc2(intermediate_vector)

    if apply_softmax:
        prediction_vector = F.softmax(prediction_vector, dim=1)

    return prediction_vector

```

训练例程

虽然我们使用了不同的模型、数据集和损失函数，但是训练例程是相同的。因此，在例 4-8 中，我们只展示了 `args` 以及本例中的训练例程与“示例:餐厅评论情绪分类”中的示例之间的主要区别。

示例 4-8：用于使用 MLP 分类姓氏的参数

```

args = Namespace(
    # Data and path information
    surname_csv="data/surnames/surnames_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch4/surname_mlp",
    # Model hyper parameters
    hidden_dim=300
    # Training hyper parameters
    seed=1337,
    num_epochs=100,
    early_stopping_criteria=5,
    learning_rate=0.001,
    batch_size=64,
    # Runtime options omitted for space
)

```

训练中最显著的差异与模型中输出的种类和使用的损失函数有关。在这个例子中，输出是一个多类预测向量，可以转换为概率。正如在模型描述中所描述的，这种输出的损失类型仅限于 `CrossEntropyLoss` 和 `NLLLoss`。由于它的简化，我们使用了 `CrossEntropyLoss`。

在例 4-9 中，我们展示了数据集、模型、损失函数和优化器的实例化。这些实例应该看起来与“示例:将餐馆评论的情绪分类”中的实例几乎相同。事实上，在本书后面的章节中，这种模式将对每个示例进行重复。

示例 4-9：实例化数据集，模型，损失和优化器


```

dataset = SurnameDataset.load_dataset_and_make_vectorizer(args.surname_csv)
vectorizer = dataset.get_vectorizer()

classifier = SurnameClassifier(input_dim=len(vectorizer.surname_vocab),
                              hidden_dim=args.hidden_dim,
                              output_dim=len(vectorizer.nationality_vocab))

classifier = classifier.to(args.device)

loss_func = nn.CrossEntropyLoss(dataset.class_weights)
optimizer = optim.Adam(classifier.parameters(), lr=args.learning_rate)

```

训练循环

与“示例：餐馆评论的情感分类”中的训练循环相比，本例的训练循环除了变量名以外几乎是相同的。具体来说，示例 4-10 显示了使用不同的键从 `batch_dict` 中获取数据。除了外观上的差异，训练循环的功能保持不变。利用训练数据，计算模型输出、损失和梯度。然后，使用梯度来更新模型。

示例 4-10：训练循环的代码段

```

# the training routine is these 5 steps:

# -----
# step 1\. zero the gradients
optimizer.zero_grad()

# step 2\. compute the output
y_pred = classifier(batch_dict['x_surname'])

# step 3\. compute the loss
loss = loss_func(y_pred, batch_dict['y_nationality'])
loss_batch = loss.to("cpu").item()
running_loss += (loss_batch - running_loss) / (batch_index + 1)

# step 4\. use loss to produce gradients
loss.backward()

# step 5\. use optimizer to take gradient step
optimizer.step()

```

模型评估和预测

要理解模型的性能，您应该使用定量和定性方法分析模型。定量测量出的测试数据的误差，决定了分类器能否推广到不可见的例子。定性地说，您可以通过查看分类器的前 `k` 个预测来为一个新示例开发模型所了解的内容的直觉。

在测试数据集上评估

评价 `SurnameClassifier` 测试数据,我们执行文本分类的例子“餐馆评论的例子:分类情绪”的相同的常规例程:我们将数据集设置为遍历测试数据,调用 `classifier.eval()` 方法,并遍历测试数据以同样的方式与其他数据。在这个例子中,调用 `classifier.eval()` 可以防止 PyTorch 在使用测试/评估数据时更新模型参数。

该模型对测试数据的准确性达到 50% 左右。如果您在附带的笔记本中运行训练例程,您会注意到在训练数据上的性能更高。这是因为模型总是更适合它所训练的数据,所以训练数据的性能并不代表新数据的性能。如果您遵循代码,我们鼓励您尝试隐藏维度的不同大小。您应该注意到性能的提高。然而,这种增长不会很大(尤其是与“用 CNN 对姓氏进行分类的例子”中的模型相比)。其主要原因是收缩的单热向量化方法是一种弱表示。虽然它确实简洁地将每个姓氏表示为单个向量,但它丢弃了字符之间的顺序信息,这对于识别起源非常重要。

CLASSIFYING A NEW SURNAME

示例 4-11 显示了分类新姓氏的代码。给定一个姓氏作为字符串,该函数将首先应用向量化过程,然后获得模型预测。注意,我们包含了 `apply_softmax` 标志,所以结果包含概率。模型预测,在多项式的情况下,是类概率的列表。我们使用 PyTorch 张量最大函数来得到由最高预测概率表示的最优类。

示例 4-11: 执行国籍预测的函数

```
def predict_nationality(name, classifier, vectorizer):
    vectorized_name = vectorizer.vectorize(name)
    vectorized_name = torch.tensor(vectorized_name).view(1, -1)
    result = classifier(vectorized_name, apply_softmax=True)

    probability_values, indices = result.max(dim=1)
    index = indices.item()

    predicted_nationality = vectorizer.nationality_vocab.lookup_index(index)
    probability_value = probability_values.item()

    return {'nationality': predicted_nationality,
            'probability': probability_value}
```

为新的姓氏获取前k个预测

RETRIEVING THE TOP-K PREDICTIONS FOR A NEW SURNAME

不仅要看最好的预测,还要看更多的预测。例如, NLP 中的标准实践是采用 `k` 个最佳预测并使用另一个模型对它们重新排序。PyTorch 提供了一个 `torch.topk` 函数,它提供了一种方便的方法来获得这些预测,如示例 4-12 所示。

示例 4-12: 预测前 `k` 个国籍

```
def predict_topk_nationality(name, classifier, vectorizer, k=5):
    vectorized_name = vectorizer.vectorize(name)
    vectorized_name = torch.tensor(vectorized_name).view(1, -1)
    prediction_vector = classifier(vectorized_name, apply_softmax=True)
    probability_values, indices = torch.topk(prediction_vector, k=k)

    # returned size is 1,k
    probability_values = probability_values.detach().numpy()[0]
    indices = indices.detach().numpy()[0]

    results = []
    for prob_value, index in zip(probability_values, indices):
        nationality = vectorizer.nationality_vocab.lookup_index(index)
        results.append({'nationality': nationality,
                       'probability': prob_value})

    return results
```

MLPs 正则化：权重正则化和结构化正则化（或丢弃）

在第三章中，我们解释了正则化是如何解决过拟合问题的，并研究了两种重要的权重正则化类型——L1 和 L2。这些权值正则化方法也适用于 MLPs 和卷积神经网络，我们将在本章后面介绍。除权值正则化外，对于深度模型（即例如本章讨论的前馈网络，一种称为丢弃的结构正则化方法变得非常重要。

DROPOUT

简单地说，在训练过程中，丢弃有一定概率使属于两个相邻层的单元之间的连接减弱。这有什么用呢？我们从斯蒂芬·梅里蒂（Stephen Merity）的一段直观（且幽默）的解释开始：“丢弃，简单地说，是指如果你能在喝醉的时候反复学习如何做一件事，那么你应该能够在清醒的时候做得更好。这一见解产生了许多最先进的结果和一个新兴的领域。”神经网络——尤其是具有大量分层的深层网络——可以在单元之间创建有趣的相互适应。“共同适应”（Coadaptation）是神经科学中的一个术语，但在这里它只是指一种情况，即两个单元之间的联系变得过于紧密，而牺牲了其他单元之间的联系。这通常会导致模型与数据过拟合。通过概率地丢弃单元之间的连接，我们可以确保没有一个单元总是依赖于另一个单元，从而产生健壮的模式。丢弃不会向模型中添加额外的参数，但是需要一个超参数——“丢弃概率”。丢弃概率，正如你可能已经猜到的，是单位之间的连接被丢弃的概率。通常将下降概率设置为 0.5。例 4-13 给出了一个带丢弃的 MLP 的重新实现。

示例 4-13：带有丢弃的 MLP

```
import torch.nn as nn
import torch.nn.functional as F

class MultilayerPerceptron(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        """
        Args:
```

```

        input_dim (int): the size of the input vectors
        hidden_dim (int): the output size of the first Linear layer
        output_dim (int): the output size of the second Linear layer
    """
    super(MultilayerPerceptron, self).__init__()
    self.fc1 = nn.Linear(input_dim, hidden_dim)
    self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x_in, apply_softmax=False):
        """The forward pass of the MLP

        Args:
            x_in (torch.Tensor): an input data tensor.
                x_in.shape should be (batch, input_dim)
            apply_softmax (bool): a flag for the softmax activation
                should be false if used with the Cross Entropy losses
        Returns:
            the resulting tensor. tensor.shape should be (batch, output_dim)
        """
        intermediate = F.relu(self.fc1(x_in))
        output = self.fc2(F.dropout(intermediate, p=0.5))

        if apply_softmax:
            output = F.softmax(output, dim=1)
        return output

```

请注意，丢弃只适用于训练期间，不适用于评估期间。作为练习，我们鼓励您试验带有丢弃的 `SurnameClassifier` 模型，看看它如何更改结果。

卷积神经网络

在本章的第一部分中，我们深入研究了 MLPs、由一系列线性层和非线性函数构建的神经网络。mlp 不是利用顺序模式的最佳工具。例如，在姓氏数据集中，姓氏可以有（不同长度的）段，这些段可以显示出相当多关于其起源国家的信息（如 O'Neill 中的 O、Antonopoulos 中的 opoulos、Nagasawa 中的 sawa 或 Zhu 中的 Zh）。这些段的长度可以是可变的，挑战是在不显式编码的情况下捕获它们。

在本节中，我们将介绍卷积神经网络（CNN），这是一种非常适合检测空间子结构（并因此创建有意义的空间子结构）的神经网络。CNNs 通过使用少量的权重来扫描输入数据张量来实现这一点。通过这种扫描，它们产生表示子结构检测（或不检测）的输出张量。

在本节的其余部分中，我们首先描述 CNN 的工作方式，以及在设计 CNN 时应该考虑的问题。我们深入研究 CNN 超参数，目的是提供直观的行为和这些超参数对输出的影响。最后，我们通过几个简单的例子逐步说明 CNNs 的机制。在“示例:使用 CNN 对姓氏进行分类”中，我们将深入研究一个更广泛的示例。

历史背景

CNNs 的名称和基本功能源于经典的数学运算卷积。卷积已经应用于各种工程学科，包括数字信号处理和计算机图形学。一般来说，卷积使用程序员指定的参数。这些参数被指定来匹配一些功能设计，如突出边缘或抑制高频声音。事实上，许多 Photoshop 滤镜都是应用于图像的固定卷积运算。然而，在深度学习和本章中，我们从数据中学习卷积滤波器的参数，因此它对于解决当前的任务是最优的。

CNN 超参数

为了理解不同的设计决策对 CNN 意味着什么，我们在图 4-6 中展示了一个示例。在本例中，单个“核”应用于输入矩阵。卷积运算（线性算子）的精确数学表达式对于理解这一节并不重要，但是从这个图中可以直观地看出，核是一个小的方阵，它被系统地应用于输入矩阵的不同位置。

虽然经典卷积是通过指定核的具体值来设计的，但是 CNN 是通过指定控制 CNN 行为的超参数来设计的，然后使用梯度下降来为给定数据集找到最佳参数。两个主要的超参数控制卷积的形状（称为 `kernel_size`）和卷积将在输入数据张量（称为 `stride`）中相乘的位置。还有一些额外的超参数控制输入数据张量被 0 填充了多少（称为 `padding`），以及当应用到输入数据张量（称为 `dilation`）时，乘法应该相隔多远。在下面的小节中，我们将更详细地介绍这些超参数。

卷积操作的维度

首先要理解的概念是卷积运算的维数。在图 4-6 和本节的其他图中，我们使用二维卷积进行说明，但是根据数据的性质，还有更适合的其他维度的卷积。在 PyTorch 中，卷积可以是一维、二维或三维的，分别由 `Conv1d`、`Conv2d` 和 `Conv3d` 模块实现。一维卷积对于每个时间步都有一个特征向量的时间序列非常有用。在这种情况下，我们可以在序列维度上学习模式。NLP 中的卷积运算大多是一维的卷积。另一方面，二维卷积试图捕捉数据中沿两个方向的时空模式；例如，在图像中沿高度和宽度维度——为什么二维卷积在图像处理中很流行。类似地，在三维卷积中，模式是沿着数据中的三维捕获的。例如，在视频数据中，信息是三维的，二维表示图像的帧，时间维表示帧的序列。就本书而言，我们主要使用 `Conv1d`。

通道

非正式地，通道（channel）是指沿输入中的每个点的特征维度。例如，在图像中，对应于 RGB 组件的图像中的每个像素有三个通道。在使用卷积时，文本数据也可以采用类似的概念。从概念上讲，如果文本文档中的“像素”是单词，那么通道的数量就是词汇表的大小。如果我们更细粒度地考虑字符的卷积，通道的数量就是字符集的大小（在本例中刚好是词汇表）。在 PyTorch 卷积实现中，输入通道的数量是 `in_channels` 参数。卷积操作可以在输出（`out_channels`）中产生多个通道。您可以将其视为卷积运算符将输入特征维“映射”到输出特征维。图 4-7 和图 4-8 说明了这个概念。

很难立即知道有多少输出通道适合当前的问题。为了简化这个困难，我们假设边界是 1,024——我们可以有一个只有一个通道的卷积层，也可以有一个只有 1,024 个通道的卷积层。现在我们有边界，接下来要考虑的是有多少个输入通道。一种常见的设计模式是，从一个卷积层到下一个卷积层，通道数量的缩减不超过 2 倍。这不是一个硬性的规则，但是它应该让您了解适当数量的 `out_channels` 是什么样子的。

核大小

核矩阵的宽度称为核大小（PyTorch 中的 `kernel_size`）。在图 4-6 中，核大小为 2，而在图 4-9 中，我们显示了一个大小为 3 的内核。您应该形成的直觉是，卷积将输入中的空间（或时间）本地信息组合在一起，每个卷积的本地信息量由内核大小控制。然而，通过增加核的大小，也会减少输出的大小（Dumoulin 和 Visin, 2016）。这就是为什么当核大小为 3 时，输出矩阵是图 4-9 中的 2×2 ，而当核大小为 2 时，输出矩阵是图 4-6 中的 3×3 。

此外，您可以将 NLP 应用程序中核大小的行为看作类似于通过查看单词组捕获语言模式的 N 元组的行为。使用较小的核大小，可以捕获较小的频繁模式，而较大的核大小会导致较大的模式，这可能更有意义，但是发生的频率更低。较小的核大小会导致输出中的细粒度特性，而较大的核大小会导致粗粒度特性。

跨步

跨步控制卷积之间的步长。如果步长与核相同，则内核计算不会重叠。另一方面，如果跨度为 1，则内核重叠最大。输出张量可以通过增加步幅的方式被有意的压缩来总结信息，如图 4-10 所示。

填充

即使跨步和 `kernel_size` 允许控制每个计算出的特征值有多大范围，它们也有一个有害的、有时是无意的副作用，那就是缩小特征映射的总大小（卷积的输出）。为了抵消这一点，输入数据张量被人为地增加了长度（如果是一维、二维或三维）、高度（如果是二维或三维）和深度（如果

是三维），方法是在每个维度上附加和前置 0。这意味着 CNN 将执行更多的卷积，但是输出形状可以控制，而不会影响所需的核大小、步幅或扩展。图 4-11 展示了正在运行的填充。

膨胀

膨胀控制卷积核如何应用于输入矩阵。在图 4-12 中，我们显示，将膨胀从 1（默认值）增加到 2 意味着当应用于输入矩阵时，核的元素彼此之间是两个空格。另一种考虑这个问题的方法是在核中跨跃——在核中的元素或核的应用之间存在一个步长，即存在“洞”。这对于在不增加参数数量的情况下总结输入空间的更大区域是有用的。当卷积层被叠加时，扩张卷积被证明是非常有用的。连续扩张的卷积指数级地增大了“接受域”的大小，即网络在做出预测之前所看到的输入空间的大小。

在 PyTorch 实现 CNNs

在本节中，我们将通过端到端示例来利用上一节中介绍的概念。一般来说，神经网络设计的目标是找到一个能够完成任务的超参数组态。我们再次考虑在“示例:带有多层感知器的姓氏分类”中引入的现在很熟悉的姓氏分类任务，但是我们将使用 CNNs 而不是 MLP。我们仍然需要应用最后一个线性层，它将学会从一系列卷积层创建的特征向量创建预测向量。这意味着目标是确定卷积层的配置，从而得到所需的特征向量。所有 CNN 应用程序都是这样的:首先有一组卷积层，它们提取一个特征映射，然后将其作为上游处理的输入。在分类中，上游处理几乎总是应用线性（或 fc）层。

本节中的实现遍历设计决策，以构建一个特征向量。我们首先构造一个人工数据张量，以反映实际数据的形状。数据张量的大小是三维的——这是向量化文本数据的最小批大小。如果你对一个字符序列中的每个字符使用单热向量，那么单热向量序列就是一个矩阵，而单热矩阵的小批量就是一个三维张量。使用卷积的术语，每个单热向量（通常是词汇表的大小）的大小是输入通道的数量，字符序列的长度是宽度。

在例 4-14 中，构造特征向量的第一步是将 PyTorch 的 `Conv1d` 类的一个实例应用到三维数据张量。通过检查输出的大小，你可以知道张量减少了多少。我们建议您参考图 4-9 来直观地解释为什么输出张量在收缩。

示例 4-14：人造数据和使用 `Conv1d` 类

```
Input[0]
batch_size = 2
one_hot_size = 10
sequence_width = 7
data = torch.randn(batch_size, one_hot_size, sequence_width)
conv1 = Conv1d(in_channels=one_hot_size, out_channels=16,
               kernel_size=3)
intermediate1 = conv1(data)
print(data.size())
print(intermediate1.size())
```



```
Output[0]
torch.Size([2, 10, 7])
torch.Size([2, 16, 5])
```

进一步减小输出张量的主要方法有三种。第一种方法是创建额外的卷积并按顺序应用它们。最终，对应的 `sequence_width`（`dim=2`）维度的大小将为 1。我们在例 4-15 中展示了应用两个额外卷积的结果。一般来说，对输出张量的约简应用卷积的过程是迭代的，需要一些猜测工作。我们的示例是这样构造的：经过三次卷积之后，最终的输出在最终维度上的大小为 1。

示例 4-15：卷积在数据上的迭代应用

```
Input[0]
conv2 = nn.Conv1d(in_channels=16, out_channels=32, kernel_size=3)
conv3 = nn.Conv1d(in_channels=32, out_channels=64, kernel_size=3)

intermediate2 = conv2(intermediate1)
intermediate3 = conv3(intermediate2)

print(intermediate2.size())
print(intermediate3.size())
Output[0]
torch.Size([2, 32, 3])
torch.Size([2, 64, 1])
Input[1]
y_output = intermediate3.squeeze()
print(y_output.size())
Output[1]
torch.Size([2, 64])
```

在每次卷积中，通道维度的大小都会增加，因为通道维数是每个数据点的特征向量。张量实际上是一个特征向量的最后一步是去掉讨厌的尺寸为 1 维。您可以使用 `squeeze()` 方法来实现这一点。该方法将删除 `size=1` 的所有维度并返回结果。然后，得到的特征向量可以与其他神经网络组件（如线性层）一起使用来计算预测向量。

另外还有两种方法可以将张量简化为每个数据点的一个特征向量：将剩余的值压平为特征向量，并在额外维度上求平均值。这两种方法如示例 4-16 所示。使用第一种方法，只需使用 PyTorch 的 `view()` 方法将所有向量展成单个向量。第二种方法使用一些数学运算来总结向量中的信息。最常见的操作是算术平均值，但沿特征映射维数求和使用最大值也是常见的。每种方法都有其优点和缺点。扁平化保留了所有的信息，但会导致比预期（或计算上可行）更大的特征向量。平均变得与额外维度的大小无关，但可能会丢失信息。

示例 4-16：用于归约到特征向量的两个相加方法

```
Input[0]
# Method 2 of reducing to feature vectors
print(intermediate1.view(batch_size, -1).size())

# Method 3 of reducing to feature vectors
print(torch.mean(intermediate1, dim=2).size())
# print(torch.max(intermediate1, dim=2).size())
```



```
# print(torch.sum(intermediate1, dim=2).size())
Output[0]
torch.Size([2, 80])
torch.Size([2, 16])
```

这种设计一系列卷积的方法是基于经验的:从数据的预期大小开始,处理一系列卷积,最终得到适合您的特征向量。虽然这种方法在实践中效果很好,但在给定卷积的超参数和输入张量的情况下,还有另一种计算张量输出大小的方法,即使用从卷积运算本身推导出的数学公式。

Example: Classifying Surnames by Using a CNN

为了证明 CNN 的有效性,让我们应用一个简单的 CNN 模型来分类姓氏。这项任务的许多细节与前面的 MLP 示例相同,但真正发生变化的是模型的构造和向量化过程。模型的输入,而不是我们在上一个例子中看到的收缩的单热向量,将是一个单热的矩阵。这种设计将使 CNN 能够更好地“查看”字符的排列,并对在“示例:带有多层感知器的姓氏分类”中使用的收缩的单热编码中丢失的序列信息进行编码。

The SurnameDataset

虽然姓氏数据集之前在“示例:带有多层感知器的姓氏分类”中进行了描述,但是我们建议您参考“姓氏数据集”来了解它的描述。尽管我们使用了来自“示例:带有多层感知器的姓氏分类”中的相同数据集,但在实现上有一个不同之处:数据集由单热向量矩阵组成,而不是一个收缩的单热向量。为此,我们实现了一个数据集类,它跟踪最长的姓氏,并将其作为矩阵中包含的行数提供给向量化器。列的数量是单热向量的大小(词汇表的大小)。示例 4-17 显示了对

`SurnameDataset.__getitem__` 的更改;我们显示对 `SurnameVectorizer` 的更改。在下一小节向量化。

我们使用数据集中最长的姓氏来控制单热矩阵的大小有两个原因。首先,将每一小批姓氏矩阵组合成一个三维张量,要求它们的大小相同。其次,使用数据集中最长的姓氏意味着可以以相同的方式处理每个小批量。

示例 4-17: 为传递最大姓氏长度而修改的 `SurnameDataset`

```
class SurnameDataset(Dataset):
    # ... existing implementation from Section 4.2

    def __getitem__(self, index):
        row = self._target_df.iloc[index]

        surname_matrix = \
            self._vectorizer.vectorize(row.surname, self._max_seq_length)

        nationality_index = \
            self._vectorizer.nationality_vocab.lookup_token(row.nationality)
```

```

return {'x_surname': surname_matrix,
        'y_nationality': nationality_index}

```

Vocabulary, Vectorizer 和 DataLoader

在本例中，尽管词汇表和 `DataLoader` 的实现方式与“示例:带有多层感知器的姓氏分类”中的示例相同，但 `Vectorizer` 的 `vectorize()` 方法已经更改，以适应 CNN 模型的需要。具体来说，正如我们在示例 4-18 中的代码中所示，该函数将字符串中的每个字符映射到一个整数，然后使用该整数构造一个由单热向量组成的矩阵。重要的是，矩阵中的每一列都是不同的单热向量。主要原因是，我们将使用的 `Conv1d` 层要求数据张量在第 0 维上具有批量，在第 1 维上具有通道，在第 2 维上具有特性。

除了更改为使用单热矩阵之外，我们还修改了向量化器，以便计算姓氏的最大长度并将其保存为 `max_surname_length`

示例 4-18: 为 CNN 实现姓氏向量化器

```

class SurnameVectorizer(object):
    """ The Vectorizer which coordinates the Vocabularies and puts them to
    use """
    def vectorize(self, surname):
        """
        Args:
            surname (str): the surname
        Returns:
            one_hot_matrix (np.ndarray): a matrix of onehot vectors
        """

        one_hot_matrix_size = (len(self.character_vocab),
self.max_surname_length)
        one_hot_matrix = np.zeros(one_hot_matrix_size, dtype=np.float32)

        for position_index, character in enumerate(surname):
            character_index = self.character_vocab.lookup_token(character)
            one_hot_matrix[character_index][position_index] = 1

        return one_hot_matrix

    @classmethod
    def from_dataframe(cls, surname_df):
        """Instantiate the vectorizer from the dataset dataframe

        Args:
            surname_df (pandas.DataFrame): the surnames dataset
        Returns:
            an instance of the SurnameVectorizer
        """

        character_vocab = Vocabulary(unk_token="@")
        nationality_vocab = Vocabulary(add_unk=False)
        max_surname_length = 0

        for index, row in surname_df.iterrows():

```

```

max_surname_length = max(max_surname_length, len(row.surname))
for letter in row.surname:
    character_vocab.add_token(letter)
nationality_vocab.add_token(row.nationality)

return cls(character_vocab, nationality_vocab, max_surname_length)

```

使用 CNN 重新实现SurnameClassifier

我们在本例中使用的模型是使用我们在“卷积神经网络”中介绍的方法构建的。实际上，我们在该部分中创建的用于测试卷积层的“人工”数据与姓氏数据集中使用本例中的向量化器的数据张量的大小完全匹配。正如您在示例 4-19 中所看到的，它与我们在“卷积神经网络”中引入的 Conv1d 序列既有相似之处，也有需要解释的新添加内容。具体来说，该模型类似于“卷积神经网络”，它使用一系列一维卷积来增量地计算更多的特征，从而得到一个单特征向量。

然而，本例中的新内容是使用 sequence 和 ELU PyTorch 模块。序列模块是封装线性操作序列的方便包装器。在这种情况下，我们使用它来封装 Conv1d 序列的应用程序。ELU 是类似于第 3 章中介绍的 ReLU 的非线性函数，但是它不是将值裁剪到 0 以下，而是对它们求幂。ELU 已经被证明是卷积层之间使用的一种很有前途的非线性（Clevert et al., 2015）。

在本例中，我们将每个卷积的通道数与 num_channels 超参数绑定。我们可以选择不同数量的通道分别进行卷积运算。这样做需要优化更多的超参数。我们发现 256 足够大，可以使模型达到合理的性能。

示例 4-19：基于 CNN 的 SurnameClassifier

```

import torch.nn as nn
import torch.nn.functional as F

class SurnameClassifier(nn.Module):
    def __init__(self, initial_num_channels, num_classes, num_channels):
        """
        Args:
            initial_num_channels (int): size of the incoming feature vector
            num_classes (int): size of the output prediction vector
            num_channels (int): constant channel size to use throughout
        network
        """
        super(SurnameClassifier, self).__init__()

        self.convnet = nn.Sequential(
            nn.Conv1d(in_channels=initial_num_channels,
                      out_channels=num_channels, kernel_size=3),
            nn.ELU(),
            nn.Conv1d(in_channels=num_channels, out_channels=num_channels,
                      kernel_size=3, stride=2),
            nn.ELU(),
            nn.Conv1d(in_channels=num_channels, out_channels=num_channels,
                      kernel_size=3, stride=2),
            nn.ELU(),

```

```

        nn.Conv1d(in_channels=num_channels, out_channels=num_channels,
                  kernel_size=3),
        nn.ELU()
    )
    self.fc = nn.Linear(num_channels, num_classes)

def forward(self, x_surname, apply_softmax=False):
    """The forward pass of the classifier

    Args:
        x_surname (torch.Tensor): an input data tensor.
            x_surname.shape should be (batch, initial_num_channels,
                                      max_surname_length)
        apply_softmax (bool): a flag for the softmax activation
            should be false if used with the Cross Entropy losses
    Returns:
        the resulting tensor. tensor.shape should be (batch, num_classes)
    """
    features = self.convnet(x_surname).squeeze(dim=2)
    prediction_vector = self.fc(features)

    if apply_softmax:
        prediction_vector = F.softmax(prediction_vector, dim=1)

    return prediction_vector

```

训练例程

训练程序包括以下似曾相识的操作序列:实例化数据集,实例化模型,实例化损失函数,实例化优化器,遍历数据集的训练分区和更新模型参数,遍历数据集的验证分区和测量性能,然后重复数据集迭代一定次数。此时,这是本书到目前为止的第三个训练例程实现,应该将这个操作序列内部化。对于这个例子,我们将不再详细描述具体的训练例程,因为它与“示例:带有多层感知器的姓氏分类”中的例程完全相同。但是,输入参数是不同的,您可以在示例 4-20 中看到。

示例 4-20: CNN 姓氏分类器的输入参数

```

args = Namespace(
    # Data and Path information
    surname_csv="data/surnames/surnames_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch4/cnn",
    # Model hyper parameters
    hidden_dim=100,
    num_channels=256,
    # Training hyper parameters
    seed=1337,
    learning_rate=0.001,
    batch_size=128,
    num_epochs=100,
    early_stopping_criteria=5,
    dropout_p=0.1,

```

```
# Runtime omitted for space ...  
)
```

模型评估和预测

要理解模型的性能，需要对性能进行定量和定性的度量。下面将描述这两个度量的基本组件。我们鼓励您扩展它们，以探索该模型及其所学习到的内容。

在测试集上评估

正如“示例:带有多层感知器的姓氏分类”中的示例与本示例之间的训练例程没有变化一样，执行评估的代码也没有变化。总之，调用分类器的 `eval()` 方法来防止反向传播，并迭代测试数据集。与 MLP 约 50% 的性能相比，该模型的测试集性能准确率约为 56%。尽管这些性能数字绝不是这些特定架构的上限，但是通过一个相对简单的 CNN 模型获得的改进应该足以让您在文本数据上尝试 CNNs。

为新的姓氏分类或获取最佳预测

在本例中，`predict_nationality()` 函数的一部分发生了更改，如示例 4-21 所示:我们没有使用视图方法重塑新创建的数据张量以添加批量维度，而是使用 PyTorch 的 `unsqueeze()` 函数在批量应该在的位置添加大小为 1 的维度。相同的更改反映在 `predict_topk_nationality()` 函数中。

示例 4-21: 使用训练过的模型做出预测

```
def predict_nationality(surname, classifier, vectorizer):
    """Predict the nationality from a new surname

    Args:
        surname (str): the surname to classifier
        classifier (SurnameClassifier): an instance of the classifier
        vectorizer (SurnameVectorizer): the corresponding vectorizer

    Returns:
        a dictionary with the most likely nationality and its probability
    """
    vectorized_surname = vectorizer.vectorize(surname)
    vectorized_surname = torch.tensor(vectorized_surname).unsqueeze(0)
    result = classifier(vectorized_surname, apply_softmax=True)

    probability_values, indices = result.max(dim=1)
    index = indices.item()

    predicted_nationality = vectorizer.nationality_vocab.lookup_index(index)
    probability_value = probability_values.item()

    return {'nationality': predicted_nationality, 'probability':
            probability_value}
```

CNN 中的其他话题

为了结束我们的讨论，我们概述了几个其他的主题，这些主题是 CNNs 的核心，但在它们的共同使用中起着主要作用。特别是，您将看到池化操作、批量规范化、网络中的网络连接和残差连接

池化操作

池化是将高维特征映射总结为低维特征映射的操作。卷积的输出是一个特征映射。特征映射中的值总结了输入的一些区域。由于卷积计算的重叠性，许多计算出的特征可能是冗余的。池化是一种将高维（可能是冗余的）特征映射总结为低维特征映射的方法。在形式上，池是一种像求和，均值或最大值这样的算术运算符，系统地应用于特征映射中的局部区域，得到的池操作分别称为求和池化、平均池化和最大池化。池还可以作为一种方法，将较大但较弱的特征映射的统计强度改进为较小但较强的特征映射。图 4-13 说明了池化。

批量规范化 (BatchNorm)

批量标准化是设计网络时经常使用的一种工具。BatchNorm 对 CNN 的输出进行转换，方法是将激活量缩放为零均值和单位方差。它用于 Z 转换的平均值和方差值每批更新一次，这样任何单个批中的波动都不会太大地移动或影响它。BatchNorm 允许模型对参数的初始化不那么敏感，并且简化了学习速率的调整 (Ioffe and Szegedy, 2015)。在 PyTorch 中，批量规范是在 nn 模块中定义的。例 4-22 展示了如何用卷积和线性层实例化和使用批量规范。

示例 4-22：使用 Conv1D 层和批量规范化

```
# ...
self.conv1 = nn.Conv1d(in_channels=1, out_channels=10,
                        kernel_size=5,
                        stride=1)
self.conv1_bn = nn.BatchNorm1d(num_features=10)
# ...

def forward(self, x):
    # ...
    x = F.relu(self.conv1(x))
    x = self.conv1_bn(x)
    # ...
```

网络中的网络连接（1x1卷积）

网络中的网络连接（NiN）连接是具有 `kernel_size=1` 的卷积内核，具有一些有趣的特性。具体来说，`1x1` 卷积就像通道之间的一个完全连通的线性层。这在从多通道特征映射映射到更浅的特征映射时非常有用。在图 4-14 中，我们展示了一个应用于输入矩阵的 NiN 连接。如您所见，它将两个通道简化为一个通道。因此，NiN 或 `1x1` 卷积提供了一种廉价的方法来合并参数较少的额外非线性（Lin et al., 2013）。

残差连接/残差块

CNNs 中最重要的趋势之一是残差连接，它支持真正深层的网络（超过 100 层）。它也称为跳跃连接。如果将卷积函数表示为 `conv`，则残差块的输出如下：

总结

在本章中，您学习了两个基本的前馈架构：多层感知器（MLP；也称为“全连接”网络）和卷积神经网络（CNN）。我们看到了 MLPs 在近似任何非线性函数方面的威力，并展示了 MLPs 在 NLP 中的应用，以及从姓氏中对国籍进行分类的应用。我们研究了 mlp 的一个主要缺点/限制——缺乏参数共享——并介绍了卷积网络架构作为一种可能的解决方案。最初为计算机视觉开发的 CNNs，现已成为 NLP 的中流砥柱；主要是因为它们的高效实现和低内存需求。我们研究了卷积的不同变体——填充、扩展和扩展——以及它们如何转换输入空间。本章还专门讨论了卷积滤波器的输入和输出大小选择的实际问题。我们展示了卷积操作如何通过扩展姓氏分类示例来使用卷积网络来帮助捕获语言中的子结构信息。最后，我们讨论了与卷积网络设计相关的一些杂项但重要的主题：1) 池化，2) 批量规范化，3) `1x1` 卷积，4) 残差连接。在现代 CNN 的设计中，经常可以看到像 Inception 架构（Szegedy et al., 2015）那样同时使用许多这样的技巧。在 Inception 架构中，小心地使用这些技巧可以让卷积网络深入到数百层，不仅精确，而且训练速度快。在第 5 章中，我们探讨了学习和使用表示表示离散单元的主题，例如使用嵌入的单词、句子、文档和其他特征类型。

五、嵌入单词和类型

本文标题: [Natural-Language-Processing-with-PyTorch \(五\)](#)

文章作者: [Yif Du](#)

发布时间: 2018 年 12 月 21 日 - 12:12

最后更新: 2018 年 12 月 28 日 - 11:12

原始链接: <http://yifdu.github.io/2018/12/21/Natural-Language-Processing-with-PyTorch> (五) /

许可协议: [署名-非商业性使用-禁止演绎 4.0 国际](#) 转载请保留原文链接及作者。

在实现自然语言处理（NLP）任务时，我们需要处理不同类型的离散类型。最明显的例子是单词。单词来自一个有限的集合（也就是词汇表）。其他离散类型的示例包括字符、部分语言标记、命名实体、命名实体类型、解析特性、产品目录中的项等等。本质上，当任何输入特征来自有限（或可数无限）集时，它都是离散类型。

将离散类型（如单词）表示为密集向量是 NLP 中深度学习成功的核心。术语“表示学习”和“嵌入”是指学习从一种离散类型到向量空间中一点的映射。当离散类型为词时，密集向量表示称为词嵌入（word embedding）。我们在第 2 章中看到了基于计数的嵌入方法的例子，比如词频-逆文档频率（TF-IDF）。在本章中，我们主要研究基于学习或基于预测（Baroni et al., 2014）的嵌入方法，即通过最大化特定学习任务的目标来学习表征；例如，根据上下文预测一个单词。基于学习的嵌入方法由于其广泛的适用性和性能而在法理上受到限制。事实上，单词嵌入在 NLP 任务中的普遍性为它们赢得了“NLP 的 Sriracha”的称号，因为您可以在任何 NLP 任务中使用单词嵌入，并期望任务的性能得到改进。但是我们认为这种绰号是误导的，因为与 Sriracha 不同，嵌入（embeddings）通常不是作为事后添加到模型中的，而是模型本身的基本组成部分。

在这一章中，我们讨论与嵌入词相关的向量表示：嵌入词的方法，优化嵌入词的方法，用于监督和非监督语言任务，可视化嵌入词的方法，以及组合嵌入词的句子和文件的方法。但是，您必须记住，我们在这里描述的方法适用于任何离散类型。

为什么要学习嵌入？

在前几章中，您看到了创建单词向量表示的传统方法。具体来说，您了解到可以使用单热表示——与词汇表大小相同的长度的向量，除了单个位置以外，其他地方都是 0，值为 1 表示特定的单词。此外，您还看到了计数表示——向量的长度与模型中唯一单词的数量相同，但是在向量中与句子中单词的频率相对应的位置上有计数。基于计数的表示也称为分布表示，因为它们的重要内

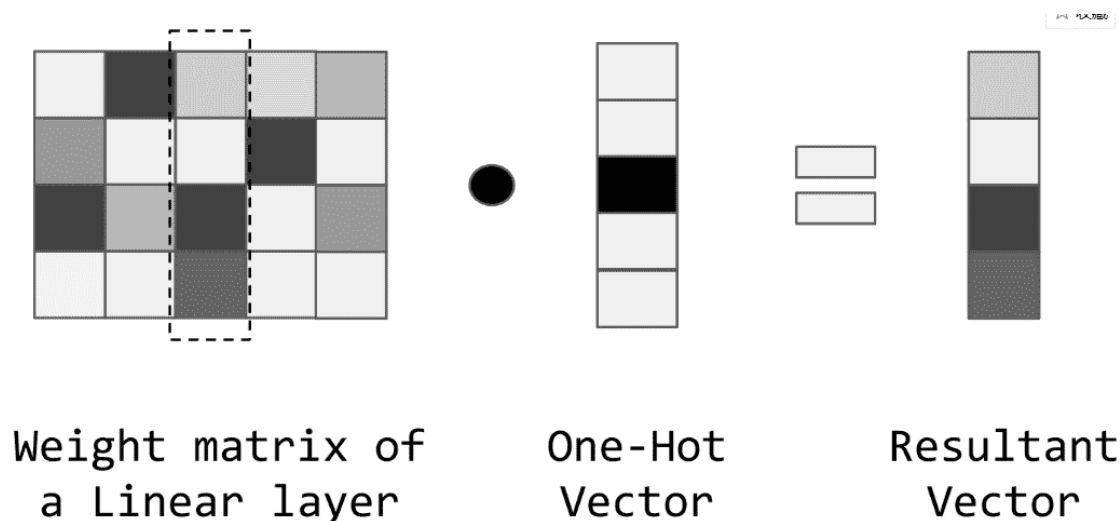
容或意义是由向量中的多个维度表示的。分布表示具有悠久的历史（Firth, 1935），可以很好地用于许多机器学习和神经网络模型。这些表示，不是从数据中学习的，而是启发式构建的。

分布式表示的名称来源于这样一个事实：由于单词现在由低得多的密集向量表示（比如 $d = 100$ ，而不是整个词汇的大小，可以是大约 10^5 到 10^6 或更高），并且一个单词的含义和其他属性分布在这个密集向量的不同维度上。

低维的学习密集表示比我们在前几章中看到的基于单热和计数的向量有几个优点。首先，降低维度是计算效率。其次，基于计数的表示导致高维向量在多个维度上冗余地编码相似的信息，并且不共享统计强度。第三，输入中维数过高会导致机器学习和优化中的实际问题，这通常被称为“维数灾难”。“传统上，为了处理这个维度问题，我们使用了像 PCA/SVD 这样的降维方法，但有点讽刺的是，当维度以百万的顺序排列时（典型的 NLP 情况），这些方法不能很好地进行缩放。”第四，从特定于任务的数据中学习（或微调）的表示形式最适合手头的任务。使用 TF-IDF 等启发式方法或 SVD 等低维方法，不清楚嵌入方法的优化目标是否与任务相关。

嵌入的有效性

为了理解嵌入是如何工作的，让我们看一个用单热向量乘以线性层中的权重矩阵的例子，如图 5-1 所示。在第 3 章和第 4 章中，单热向量的大小与词汇表相同。这个向量被称为“单热”，因为它在索引中有一个 1，表示存在一个特定的单词。



根据定义，接受这个单热向量作为输入的线性层的权值矩阵必须与这个单热向量的大小具有相同的行数。当您执行矩阵乘法时，如图 5-1 所示，结果向量实际上只是选择了由非零项指示的行。从这个观察结果中，我们可以跳过乘法步骤，直接使用整数作为检索所选行的索引。

关于嵌入效率的最后一点注意事项是，尽管图 5-1 中的示例显示的权重矩阵与传入的单热向量具有相同的维度，但情况并非总是如此。实际上，嵌入通常用于在低维空间中表示单词，而不是使用一个单热向量或基于计数的表示。在研究文献中，嵌入的典型尺寸从 25 维到 500 维不等，准确的选择可以归结为您必须节省的图形处理器单元（GPU）内存。

学习单词嵌入的方式

本章的目标不是教授特定的单词嵌入技术，而是理解什么是嵌入，如何以及在哪里使用它们，如何在模型中可靠地使用它们，以及理解它们的局限性。我们之所以这样做，是因为实践者很少发现自己处于需要编写嵌入训练算法的新单词的情况下。然而，在这一节中，我们将简要概述当前训练单词嵌入的方法。所有的单词嵌入方法都是用单词来训练的。但是以一种监督的方式。这可以通过构造辅助监督任务来实现，在这些任务中，数据被隐式地标记，直观地认为，优化后用于解决辅助任务的表示形式将捕获文本语料库的许多统计和语言属性，以便普遍使用。以下是一些辅助任务的例子：

- 给出一个单词序列，预测下一个单词。这也称为语言建模任务。
- 给定单词前后的顺序，预测缺失的单词。
- 给定一个单词，预测出现在窗口中的单词，独立于位置。

当然，这个列表是不完整的，辅助任务的选择取决于算法设计者的直觉和计算费用。例如 Glove、连续词袋、Skipgrams 等等。我们参考 Goldberg, 2017，第 10 章，但我们将简要研究 CBOW 模型。但是，对于大多数目的来说，使用预先训练好的单词嵌入并对它们进行微调就足够了。

预训练词嵌入的实践用法

本章的大部分内容，以及本书的后面部分，都涉及到使用经过预先训练的单词嵌入。使用前面描述的许多方法中的一种，可以免费下载和使用预先训练过的单词嵌入、在大型语料（类似于所有新闻、维基百科和通用爬行者）上进行训练的单词嵌入。本章的其余部分将展示如何有效地加载和查找这些嵌入，研究词嵌入的一些属性，并展示在 NLP 任务中使用预先训练的嵌入的示例。

加载嵌入

词嵌入已经变得非常流行和普及，您可以从原始的 Word2Vec、Stanford 的 GLoVe、Facebook 的 FastText 和许多其他版本下载许多不同的版本。通常，嵌入将以以下格式出现：每行以嵌入的单词/类型开始，然后是一系列数字（即，向量表示）。这个序列的长度就是表示的维数（也就是嵌入维数）。嵌入维数通常是数百。标记（token）类型的数量通常是词汇表的大小，以百万计。例如，这里是来自 GloVe 的狗和猫向量的前七个维度：dog: -1.242 -0.360 0.573 0.367 0.600 -0.189 1.273 ... cat: -0.964 -0.610 0.674 0.351 0.413 -0.212 1.380 ...

为了有效地加载和处理嵌入，我们描述了一个实用工具类，即预训练。该类在内存中构建所有单词向量的索引，以方便使用近似的最近邻包进行快速查找和最近邻查询。这个类显示在示例 5-1 中的第一个输入中。

示例 5-1：使用预训练词嵌入

```
Input[0]
import numpy as np
```

```

from annoy import AnnoyIndex

class PreTrainedEmbeddings(object):
    def __init__(self, word_to_index, word_vectors):
        """
        Args:
            word_to_index (dict): mapping from word to integers
            word_vectors (list of numpy arrays)
        """
        self.word_to_index = word_to_index
        self.word_vectors = word_vectors
        self.index_to_word = \
            {v: k for k, v in self.word_to_index.items()}
        self.index = AnnoyIndex(len(word_vectors[0]),
                                metric='euclidean')
        for _, i in self.word_to_index.items():
            self.index.add_item(i, self.word_vectors[i])
        self.index.build(50)

    @classmethod
    def from_embeddings_file(cls, embedding_file):
        """Instantiate from pre-trained vector file.

        Vector file should be of the format:
            word0 x0_0 x0_1 x0_2 x0_3 ... x0_N
            word1 x1_0 x1_1 x1_2 x1_3 ... x1_N

        Args:
            embedding_file (str): location of the file
        Returns:
            instance of PretrainedEmbeddings
        """
        word_to_index = {}
        word_vectors = []
        with open(embedding_file) as fp:
            for line in fp.readlines():
                line = line.split(" ")
                word = line[0]
                vec = np.array([float(x) for x in line[1:]])

                word_to_index[word] = len(word_to_index)
                word_vectors.append(vec)
        return cls(word_to_index, word_vectors)

Input[1]
embeddings = \
    PreTrainedEmbeddings.from_embeddings_file('glove.6B.100d.txt')

```

在这些例子中，我们使用 Glove 词嵌入。下载它们之后，可以使用 `PretrainedEmbeddings` 类进行实例化，如示例 5-1 中的第二个输入所示。

词嵌入之间的关系

词嵌入的核心特征是对句法和语义关系进行编码，这些句法和语义关系表现为词的使用规律。例如，人们谈论猫和狗的方式非常相似（讨论宠物、喂食等）。因此，它们的嵌入彼此之间的距离比它们与其他动物（如鸭子和大象）的距离要近得多。我们可以从几个方面探讨嵌入词编码的语义关系。最流行的一种方法是类比任务（SAT 等考试中常见的推理任务）：Word1 :

Word2 :: Word3 : __

在这个任务中，你被提供了前三个单词，需要确定第四个单词与前两个单词之间的关系是一致的。使用嵌入词，我们可以对其进行空间编码。首先，我们从单词 1 中减去单词 2。这个差异向量编码了 Word1 和 Word2 之间的关系。然后将这个差异添加到 Word3 中，从而生成一个接近第四个单词的向量，即空白符号所在的位置。使用这个结果向量对索引执行最近邻查询可以解决类比问题。计算这个的函数，如例 5-2 所示，完成了刚刚描述的工作:使用向量算法和近似的最近邻索引，完成类比。

示例 5-2：使用词嵌入的类比任务

```
Input[0]
import numpy as np
from annoy import AnnoyIndex

class PreTrainedEmbeddings(object):
    """ implementation continued from previous code box"""
    def get_embedding(self, word):
        """
        Args:
            word (str)
        Returns:
            an embedding (numpy.ndarray)
        """
        return self.word_vectors[self.word_to_index[word]]

    def get_closest_to_vector(self, vector, n=1):
        """Given a vector, return its n nearest neighbors

        Args:
            vector (np.ndarray): should match the size of the vectors
                                in the Annoy index
            n (int): the number of neighbors to return
        Returns:
            [str, str, ...]: words nearest to the given vector.
                            The words are not ordered by distance
        """
        nn_indices = self.index.get_nns_by_vector(vector, n)
        return [self.index_to_word[neighbor]
                for neighbor in nn_indices]

    def compute_and_print_analogy(self, word1, word2, word3):
        """Prints the solutions to analogies using word embeddings

        Analogies are word1 is to word2 as word3 is to __
```

```

This method will print: word1 : word2 :: word3 : word4

Args:
    word1 (str)
    word2 (str)
    word3 (str)
    """

    vec1 = self.get_embedding(word1)
    vec2 = self.get_embedding(word2)
    vec3 = self.get_embedding(word3)

    # Simple hypothesis: Analogy is a spatial relationship
    spatial_relationship = vec2 - vec1
    vec4 = vec3 + spatial_relationship

    closest_words = self.get_closest_to_vector(vec4, n=4)
    existing_words = set([word1, word2, word3])
    closest_words = [word for word in closest_words
                     if word not in existing_words]

    if len(closest_words) == 0:
        print("Could not find nearest neighbors for the vector!")
        return

    for word4 in closest_words:
        print("{} : {} :: {} : {}".format(word1, word2, word3,
                                           word4))

```

有趣的是，简单的单词类比任务可以证明单词嵌入捕获了各种语义和语法关系，如示例 5-3 所示。

示例 5-3：一组语言关系以向量类比编码

```

Input[0]
# Relationship 1: the relationship between gendered nouns and pronouns
embeddings.compute_and_print_analogy('man', 'he', 'woman')
Output[0]
man : he :: woman : she
Input[1]
# Relationship 2: Verb-Noun relationships
embeddings.compute_and_print_analogy('fly', 'plane', 'sail')
Output[1]
fly : plane :: sail : ship
Input[2]
# Relationship 3: Noun-Noun relationships
embeddings.compute_and_print_analogy('cat', 'kitten', 'dog')
Output[2]
cat : kitten :: dog : puppy
Input[3]
# Relationship 4: Hypernymy (broader category)
embeddings.compute_and_print_analogy('blue', 'color', 'dog')
Output[3]
blue : color :: dog : animal
Input[4]
# Relationship 5: Meronymy (part-to-whole)

```

```

embeddings.compute_and_print_analogy('toe', 'foot', 'finger')
Output[4]
toe : foot :: finger : hand
Input[5]
# Relationship 6: Troponymy (difference in manner)
embeddings.compute_and_print_analogy('talk', 'communicate', 'read')
Output[5]
talk : communicate :: read : interpret
Input[6]
# Relationship 7: Metonymy (convention / figures of speech)
embeddings.compute_and_print_analogy('blue', 'democrat', 'red')
Output[6]
blue : democrat :: red : republican
Input[7]
# Relationship 8: Adjectival Scales
embeddings.compute_and_print_analogy('fast', 'fastest', 'young')
Output[7]
fast : fastest :: young : youngest

```

虽然这种关系似乎是系统的语言功能，事情可能变得棘手。如例 5-4 所示，由于单词向量只是基于共现，关系可能是错误的。

示例 5-4：类比可能在简单的规模上失败

```

Input[0]
embeddings.compute_and_print_analogy('fast', 'fastest', 'small')
Output[0]
fast : fastest :: small : largest

```

示例 5-5 说明了最常见的类比对之一是如何编码带有性别的角色。

示例 5-5：在向量类比中编码的性别

```

Input[0]
embeddings.compute_and_print_analogy('man', 'king', 'woman')
Output[0]
man : king :: woman : queen

```

事实证明，区分语言规则和文化偏见是困难的。例如，医生并不是事实上的男性，护士也不是事实上的女性，但是这些长期存在于文化中的偏见被观察到作为语言的规律性，并被编入 `vector` 这个词中，如例 5-6 所示。

示例 5-6：编码在向量类比中的文化性别偏见

```

Input[0]
embeddings.compute_and_print_analogy('man', 'doctor', 'woman')
Output[0]
man : doctor :: woman : nurse

```


考虑到嵌入在 NLP 应用程序中的流程度和使用正在上升，您需要注意嵌入中的偏差。去偏现有词嵌入在一个新的和令人兴奋的研究领域（Bolukbasi et al., 2016）。此外，我们建议您访问 ethicsinnlp.org，以获取伦理与 NLP 交叉的最新结果。

示例：学习单词嵌入的连续词袋

在本例中，我们将介绍构建和学习通用嵌入词的最著名模型之一，即 Word2Vec 连续词包（CBOW）模型。在本节中，当我们提到“CBOW 任务”或“CBOW 分类任务”时，隐含的意思是我们构建分类任务的目的是为了学习 CBOW 嵌入。CBOW 模型是一种多类分类任务，其表现为对单词文本进行扫描，创建单词的上下文窗口，从上下文窗口中删除中心单词，并将上下文窗口分类为丢失的单词。直觉上，你可以把它想象成一个填空任务。有一个句子缺了一个词，模特的工作就是找出那个词应该是什么。

这个例子的目的是介绍 `nn.Embedding` 层，封装嵌入矩阵（embedding matrix）的 PyTorch 模块。利用嵌入层，我们可以将标记的整数 ID 映射到用于神经网络计算的向量上。当优化器更新模型权重以最小化损失时，它还更新向量的值。通过这个过程，模型将学习如何以最有效的方式嵌入单词。

在本例的其余部分中，我们遵循标准的示例格式。在第一部分，我们介绍数据集，玛丽·雪莱（Mary Shelley）的小说《弗兰肯斯坦》。然后，我们讨论了从标记到向量化小批量的向量化流水线。然后，我们概述了 CBOW 分类模型以及如何使用嵌入层。接下来，我们将介绍训练程序；尽管如此，如果你已经连续阅读了这本书，在这一点上训练应该是相当常规的。最后，我们讨论了模型评估、模型推理以及如何检查模型。

《弗兰肯斯坦》数据集

在本例中，我们将从玛丽·雪莱（Mary Shelley）的小说《弗兰肯斯坦》（Frankenstein）的数字化版本构建一个文本数据集，可以通过古登堡计划（Project Gutenberg）获得。本节介绍预处理过程，为这个文本数据集构建一个 PyTorch 数据集类，最后将数据集分解为训练、验证和测试集。

从古腾堡项目分发的原始文本文件开始，预处理是最小的：我们使用 NLTK 的 Punkt 标记器将文本分割成不同的句子。然后，将每个句子转换为小写字母，并完全去掉标点符号。这种预处理允许我们稍后在空白中拆分字符串，以便检索标记列表。这一预处理功能是从“示例：餐厅评论情绪分类”中重用的。

Preprocessed Sentence	i pitied frankenstein my pity amounted to horror i abhorred myself
Window #1	i pitied frankenstein my pity amounted to horror i abhorred myself
Window #2	i pitied frankenstein my pity amounted to horror i abhorred myself
Window #3	i pitied frankenstein my pity amounted to horror i abhorred myself
Window #4	i pitied frankenstein my pity amounted to horror i abhorred myself

下一步是将数据集枚举为一系列窗口，以便对 CBOW 模型进行优化。为此，我们迭代每个句子中的标记列表，并将它们分组到指定窗口大小的窗口中，如图 5-2 所示。

构建数据集的最后一步是将数据分割为三个集：训练集、验证集和测试集。回想一下，训练和验证集是在模型训练期间使用的：训练集用于更新参数，验证集用于度量模型的性能。测试集最多使用一次，以提供偏差较小的测量。在本例中（以及本书中的大多数示例中），我们使用了 70% 的训练集、15% 的验证集和 15% 的测试集。

窗口和目标的结果数据集装载了一个 `panda DataFrame`，并在 `CBOWDataset` 类中建立了索引。示例 5-7 展示了 `__getitem__` 代码片段，该代码片段利用向量化器将上下文（左右窗口）转换为向量。目标——窗口中心的单词——使用词汇表转换为整数。

示例 5-7：构造数据集类用于 CBOW 任务

```
class CBOWDataset(Dataset):
    # ... existing implementation from Section 3.5
    @classmethod
    def load_dataset_and_make_vectorizer(cls, cbow_csv):
        """Load dataset and make a new vectorizer from scratch

        Args:
            cbow_csv (str): location of the dataset
        Returns:
            an instance of CBOWDataset
        """
        cbow_df = pd.read_csv(cbow_csv)
        train_cbow_df = cbow_df[cbow_df.split=='train']
        return cls(cbow_df, CBOWVectorizer.from_dataframe(train_cbow_df))

    def __getitem__(self, index):
        """the primary entry point method for PyTorch datasets

        Args:
            index (int): the index to the data point
        Returns:
            a dict with features (x_data) and label (y_target)
        """
        row = self._target_df.iloc[index]

        context_vector = \
            self._vectorizer.vectorize(row.context, self._max_seq_length)
        target_index = self._vectorizer.cbow_vocab.lookup_token(row.target)
```

```
return {'x_data': context_vector,
        'y_target': target_index}
```

Vocabulary, Vectorizer 和 DataLoader

在 CBOW 分类任务中，从文本到量化的迷你批量的管道大部分都是标准的:词汇表和 DataLoader 功能都与“示例:餐厅评论分类情感”中的示例完全一样。然而，与我们在第 3 章和第 4 章中看到的向量化器不同，向量化器不构造单热向量。相反，构造并返回一个表示上下文索引的整数向量。示例 5-8 给出了向量化函数的代码。

示例 5-8：用于 CBOW 数据的向量化器

```
class CBOWVectorizer(object):
    """ The Vectorizer which coordinates the Vocabularies and puts them to
    use """

    def vectorize(self, context, vector_length=-1):
        """
        Args:
            context (str): the string of words separated by a space
            vector_length (int): an argument for forcing the length of index
        """
        indices = \
            [self.cbowl_vocab.lookup_token(token) for token in context.split('
')]
        if vector_length < 0:
            vector_length = len(indices)

        out_vector = np.zeros(vector_length, dtype=np.int64)
        out_vector[:len(indices)] = indices
        out_vector[len(indices):] = self.cbowl_vocab.mask_index

        return out_vector
```

请注意，如果上下文中的标记数量小于最大长度，则其余条目将被填入零。但在实践中,这可以称为用零填充。

CBOW 分类器

示例 5-9 中显示的 CBOW 分类器有三个基本步骤。首先，将表示上下文单词的索引与嵌入层一起使用，为上下文中的每个单词创建向量。其次，目标是以某种方式组合这些向量，使其能够捕获整个上下文。在这个例子中，我们对向量求和。然而，其他选项包括取最大值、平均值，甚至在顶部使用多层感知器（MLP）。第三，上下文向量与线性层一起用来计算预测向量。这个预测向量是整个词汇表的概率分布。预测向量中最大（最可能）的值表示目标词的可能预测——上下文中缺少的中心词。

这里使用的嵌入层主要由两个数字参数化:嵌入的数量(词汇表的大小)和嵌入的大小(嵌入维度)。示例 5-9 中的代码片段中使用了第三个参数: `padding_idx`。对于像我们这样数据点长度可能不相同的情况,这个参数被用作嵌入层的标记值。该层强制与该索引对应的向量及其梯度都为 0。

示例 5-9: CBOW 分类器模型

```
class CBOWClassifier(nn.Module):
    def __init__(self, vocabulary_size, embedding_size, padding_idx=0):
        """
        Args:
            vocabulary_size (int): number of vocabulary items, controls the
                number of embeddings and prediction vector size
            embedding_size (int): size of the embeddings
            padding_idx (int): default 0; Embedding will not use this index
        """
        super(CBOWClassifier, self).__init__()

        self.embedding = nn.Embedding(num_embeddings=vocabulary_size,
                                       embedding_dim=embedding_size,
                                       padding_idx=padding_idx)
        self.fc1 = nn.Linear(in_features=embedding_size,
                              out_features=vocabulary_size)

    def forward(self, x_in, apply_softmax=False):
        """The forward pass of the classifier

        Args:
            x_in (torch.Tensor): an input data tensor.
                x_in.shape should be (batch, input_dim)
            apply_softmax (bool): a flag for the softmax activation
                should be false if used with the Cross Entropy losses
        Returns:
            the resulting tensor. tensor.shape should be (batch, output_dim)
        """
        x_embedded_sum = self.embedding(x_in).sum(dim=1)
        y_out = self.fc1(x_embedded_sum)

        if apply_softmax:
            y_out = F.softmax(y_out, dim=1)

        return y_out
```

训练例程

在这个例子中,训练程序遵循我们在整本书中使用的标准。首先,初始化数据集、向量化器、模型、损失函数和优化器。然后,对数据集的训练和验证部分进行一定次数的迭代,优化训练部分的损失最小化,测量验证部分的进度。关于训练程序的更多细节,我们建议您参考“示例:餐厅评论的情绪分类”,在这里我们将详细介绍。示例 5-10 展示了我们用于训练的参数。

示例 5-10: CBOW 训练脚本的参数

```

Input[0]
args = Namespace(
    # Data and Path information
    cbow_csv="data/books/frankenstein_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch5/cbow",
    # Model hyper parameters
    embedding_size=300,
    # Training hyper parameters
    seed=1337,
    num_epochs=100,
    learning_rate=0.001,
    batch_size=128,
    early_stopping_criteria=5,
    # Runtime options omitted for space
)

```

模型评估和预测

本例中的评估是为测试集中的每个目标和上下文对从提供的单词上下文预测目标单词。正确分类的单词意味着模型正在学习从上下文预测单词。在本例中，该模型在测试集上达到了 15% 的目标词分类准确率。首先，本例中 CBOW 的构建旨在说明如何构建通用嵌入。因此，原始实现的许多特性被忽略了，因为它们增加了不必要的复杂性，但对于优化性能却是必要的。第二，我们使用的数据集是微不足道的——一本大约 7 万字的书不足以在从零开始训练时识别出许多规律。相比之下，最先进嵌入技术通常是在文本为 tb 的数据集上进行训练。

在这个例子中，我们展示了如何使用 PyTorch `nn.Embedding` 层通过设置一个名为 `CBOWClassification` 的人工监督任务来从头开始训练嵌入。在下一个示例中，我们将研究如何在一个语料库中进行预训练嵌入，如何使用它并对其进行微调以进行其他任务。在机器学习中，使用在一个任务上训练的模型作为另一个任务的初始化器称为迁移学习（transfer learning）。

示例：使用预训练嵌入用于文档分类的迁移学习

前面的示例使用一个嵌入层（embedding layer）做简单分类，这个例子构建在三个方面：首先，通过加载预训练的词嵌入，然后微调这些预训练嵌入整个新闻文章分类，最后用卷积神经网络（CNN）来捕获单词之间的空间关系。

在这个例子中，我们使用 AG News 数据集。为了对 AG News 中的单词序列进行建模，我们引入了词汇表类的一个变体 `SequenceVocabulary`，以捆绑一些对建模序列至关重要的标记。向量化器演示了如何使用这个类。

在描述了数据集以及向量化的小批量是如何构建的之后，我们将逐步将预先训练好的单词向量加载到一个嵌入层中，并演示如何根据我们的设置对它们进行定制。然后，将预训练的嵌入层

与“用 CNN 对姓氏进行分类的例子”中使用的 CNN 结合起来。为了将模型的复杂性扩大到更实际的结构，我们还确定了使用丢弃作为正则化技术的地方。给出了模型描述，然后讨论了训练过程。您可能不会感到惊讶的是，与第 4 章和本章中的前两个示例相比，训练例程几乎没有什么变化。最后，通过在测试集上对模型进行评价并对结果进行讨论，得出了算例。

AG 新闻数据集

AG news 数据集是由学者们在 2005 年为实验数据挖掘和信息提取方法而收集的 100 多万篇新闻文章的集合。这个例子的目的是说明预先训练的词嵌入在文本分类中的有效性。在本例中，我们使用精简版的 120,000 篇新闻文章，这些文章平均分为四类:体育、科学/技术、世界和商业。除了精简数据集之外，我们还将文章标题作为我们的观察重点，并创建多类分类任务来预测给定标题的类别。

和以前一样，我们通过删除标点符号、在标点符号周围添加空格（如逗号、撇号和句点）来预处理文本，并将文本转换为小写。另外，我们将数据集分解为训练、验证和测试集，首先通过类标签聚合数据点，然后将每个数据点分配给三个分割中的一个。通过这种方式，保证了跨分支的类分布是相同的。

AG news 数据集的 `__getitem__` ,第 5-11 所示的例子,是一个相当基本的公式你现在应该熟悉:检索的字符串表示模型的输入数据集从一个特定的行, `Vectorizer` 向量化,并搭配整数代表新闻类别（类标签）。

示例 5-11: AG 新闻数据集的 `__getitem__` 方法

```
class NewsDataset(Dataset):
    @classmethod
    def load_dataset_and_make_vectorizer(cls, news_csv):
        """Load dataset and make a new vectorizer from scratch

        Args:
            news_csv (str): location of the dataset
        Returns:
            an instance of SurnameDataset
        """
        news_df = pd.read_csv(news_csv)
        train_news_df = news_df[news_df.split=='train']
        return cls(news_df, NewsVectorizer.from_dataframe(train_news_df))

    def __getitem__(self, index):
        """the primary entry point method for PyTorch datasets

        Args:
            index (int): the index to the data point
        Returns:
            a dict holding the data point's features (x_data) and label
            (y_target)
        """
        row = self._target_df.iloc[index]
```

```
title_vector = \
    self._vectorizer.vectorize(row.title, self._max_seq_length)

category_index = \
    self._vectorizer.category_vocab.lookup_token(row.category)

return {'x_data': title_vector,
        'y_target': category_index}
```

Vocabulary, Vectorizer和数据Loader

在这个例子中，我们引入了 `SequenceVocabulary`，它是标准 `Vocabulary` 类的子类，它捆绑了用于序列数据的四个特殊标记：`UNK` 标记，`MASK` 标记，`BEGIN-SEQUENCE` 标记和 `END-SEQUENCE` 标记。我们在第 6 章中更详细地描述了这些标记，但简而言之，它们有三个不同的用途。我们在第 4 章中看到的 `UNK` 标记（Unknown 的缩写）允许模型学习稀有单词的表示，以便它可以容纳在测试时从未见过的单词。当我们有可变长度的序列时，`MASK` 标记充当嵌入层和损失计算的标记。最后，`BEGIN-SEQUENCE` 和 `END-SEQUENCE` 标记给出了关于序列边界的神经网络提示。图 5-3 显示了在更广泛的向量化管道中使用这些特殊标记的结果。

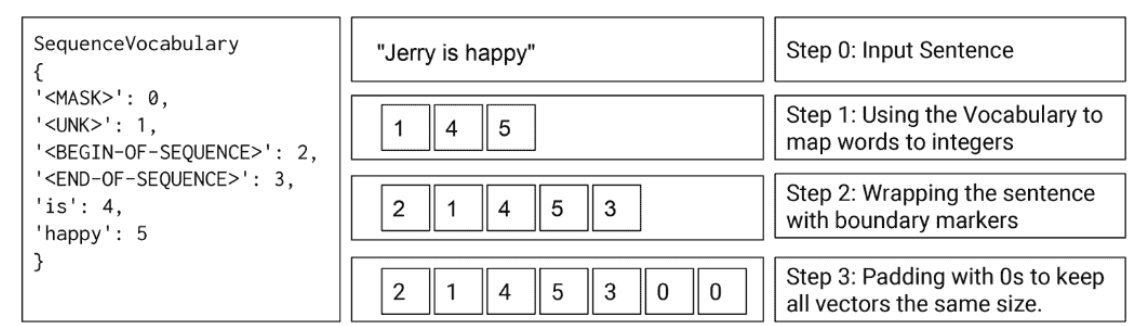


图 5-3. 向量化管道的一个简单示例从基本序列序列开始。 `Sequencevocabulary` 有四个特殊标记描述在文本。首先，它用于将单词映射到整数序列。因为单词 `Jerry` 不在序列表中，所以它被映射到 `unk` 整数。接下来，标记句子边界的特殊标记放在前面并附加到整数中。最后，整数用 0 填充到特定的长度，这允许数据集中的每个向量都是相同的长度。

文本到向量化的小批量管道中的第二个组件是 `Vectorizer`，它实例化并封装了 `SequenceVocabulary` 的使用。在这个例子中，`Vectorizer` 遵循我们在第 3-5 节中演示的模式，通过对特定频率进行计数和阈值处理来限制词汇表中允许的总词集。此操作的核心目的是通过消除噪声，低频字并限制内存模型的内存使用来提高模型的信号质量。

实例化后，`Vectorizer` 的 `vectorize()` 方法将新闻标题作为输入，并返回与数据集中最长标题一样长的向量。重要的是，它有两个关键行为。第一个是它在本地存储最大序列长度。通常，数据集跟踪最大序列长度，并且在推断时，测试序列的长度被视为向量的长度。但是，因为我们有 CNN 模型，所以即使在推理时也要保持静态大小。示例 5-11 中的代码片段中显示的第二个键行为是它输出一个零填充的整数向量，它表示序列中的单词。此外，这个整数向量具有前缀为开头的 `BEGIN-SEQUENCE` 标记的整数，以及附加到向量末尾的 `END-SEQUENCE` 标记的整数。从分类器

的角度来看，这些特殊标记提供了序列边界的证据，使其能够对边界附近的单词作出反应，而不是对靠近中心的单词作出反应。

示例 5-12：为 AG 新闻数据集实现向量化器

```
class NewsVectorizer(object):
    def vectorize(self, title, vector_length=-1):
        """
        Args:
            title (str): the string of words separated by a space
            vector_length (int): forces the length of index vector
        Returns:
            the vectorized title (numpy.array)
        """
        indices = [self.title_vocab.begin_seq_index]
        indices.extend(self.title_vocab.lookup_token(token)
                       for token in title.split(" "))
        indices.append(self.title_vocab.end_seq_index)

        if vector_length < 0:
            vector_length = len(indices)

        out_vector = np.zeros(vector_length, dtype=np.int64)
        out_vector[:len(indices)] = indices
        out_vector[len(indices):] = self.title_vocab.mask_index

        return out_vector

    @classmethod
    def from_dataframe(cls, news_df, cutoff=25):
        """Instantiate the vectorizer from the dataset dataframe

        Args:
            news_df (pandas.DataFrame): the target dataset
            cutoff (int): frequency threshold for including in Vocabulary
        Returns:
            an instance of the NewsVectorizer
        """
        category_vocab = Vocabulary()
        for category in sorted(set(news_df.category)):
            category_vocab.add_token(category)

        word_counts = Counter()
        for title in news_df.title:
            for token in title.split(" "):
                if token not in string.punctuation:
                    word_counts[token] += 1

        title_vocab = SequenceVocabulary()
        for word, word_count in word_counts.items():
            if word_count >= cutoff:
                title_vocab.add_token(word)

        return cls(title_vocab, category_vocab)
```


新闻分类器

在本章的前面，我们看到了如何从磁盘加载预训练嵌入，并使用 Spotify 的 Annoy 库中的近似最近邻数据结构有效地使用它们。在该示例中，我们将向量相互比较以发现有趣的语言学见解。但是，预训练的单词向量具有更有效的用途：我们可以使用它们来初始化嵌入层的嵌入矩阵。

使用单词嵌入（word embedding）作为初始嵌入矩阵的过程首先从磁盘加载嵌入，然后为数据中实际存在的单词选择正确的嵌入子集，然后最后设置嵌入层的权重矩阵 作为加载的子集。在例 5-13 中演示了选择子集的第一步和第二步。通常出现的一个问题是数据集中存在的单词，但不包含在预训练的 GloVe 嵌入中。处理此问题的一种常用方法是使用 PyTorch 库中的初始化方法，例如 Xavier Uniform 方法，如例 5-13 所示（Glorot 和 Bengio，2010）。

示例 5-13：基于词汇表选择词嵌入的子集

```
def load_glove_from_file(glove_filepath):
    """Load the GloVe embeddings

    Args:
        glove_filepath (str): path to the glove embeddings file
    Returns:
        word_to_index (dict), embeddings (numpy.ndarray)
    """
    word_to_index = {}
    embeddings = []
    with open(glove_filepath, "r") as fp:
        for index, line in enumerate(fp):
            line = line.split(" ") # each line: word num1 num2 ...
            word_to_index[line[0]] = index # word = line[0]
            embedding_i = np.array([float(val) for val in line[1:]])
            embeddings.append(embedding_i)
    return word_to_index, np.stack(embeddings)

def make_embedding_matrix(glove_filepath, words):
    """Create embedding matrix for a specific set of words.

    Args:
        glove_filepath (str): file path to the glove embeddings
        words (list): list of words in the dataset
    Returns:
        final_embeddings (numpy.ndarray): embedding matrix
    """
    word_to_idx, glove_embeddings = load_glove_from_file(glove_filepath)
    embedding_size = glove_embeddings.shape[1]
    final_embeddings = np.zeros((len(words), embedding_size))

    for i, word in enumerate(words):
        if word in word_to_idx:
            final_embeddings[i, :] = glove_embeddings[word_to_idx[word]]
        else:
            embedding_i = torch.ones(1, embedding_size)
            torch.nn.init.xavier_uniform_(embedding_i)
            final_embeddings[i, :] = embedding_i
```



```

    )

    self._dropout_p = dropout_p
    self.fc1 = nn.Linear(num_channels, hidden_dim)
    self.fc2 = nn.Linear(hidden_dim, num_classes)

def forward(self, x_in, apply_softmax=False):
    """The forward pass of the classifier

    Args:
        x_in (torch.Tensor): an input data tensor.
            x_in.shape should be (batch, dataset._max_seq_length)
        apply_softmax (bool): a flag for the softmax activation
            should be false if used with the Cross Entropy losses
    Returns:
        the resulting tensor. tensor.shape should be (batch, num_classes)
    """
    # embed and permute so features are channels
    x_embedded = self.emb(x_in).permute(0, 2, 1)

    features = self.convnet(x_embedded)

    # average and remove the extra dimension
    remaining_size = features.size(dim=2)
    features = F.avg_pool1d(features, remaining_size).squeeze(dim=2)
    features = F.dropout(features, p=self._dropout_p)

    # final linear layer to produce classification outputs
    intermediate_vector = F.relu(F.dropout(self.fc1(features),
                                           p=self._dropout_p))
    prediction_vector = self.fc2(intermediate_vector)

    if apply_softmax:
        prediction_vector = F.softmax(prediction_vector, dim=1)

    return prediction_vector

```

训练例程

训练例程包括以下操作序列：实例化数据集；实例化模型；实例化损失函数；实例化优化器；迭代数据集的训练分区并更新模型参数，迭代数据集的验证分区并测量性能；然后重复数据集迭代一定次数。此时，您应该非常熟悉这个序列。示例 5-15 中显示了此示例的超参数和其他训练参数。

示例 5-15：使用与训练嵌入的 CNN 新闻分类器的参数

```

args = Namespace(
    # Data and path hyper parameters
    news_csv="data/ag_news/news_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch5/document_classification",
    # Model hyper parameters
    glove_filepath='data/glove/glove.6B.100d.txt',
    use_glove=False,

```

```

embedding_size=100,
hidden_dim=100,
num_channels=100,
# Training hyper parameter
seed=1337,
learning_rate=0.001,
dropout_p=0.1,
batch_size=128,
num_epochs=100,
early_stopping_criteria=5,
# ... runtime options not shown for space
)

```

模型评估和分类

在这个例子中，任务是将新闻标题分类到它们各自的类别。正如您在前面的示例中看到的，有两种方法可以理解模型执行任务的好坏:使用测试数据集的定量评估，以及亲自检查分类结果的定性评估。

在测试集上评估

虽然这是你第一次看到的任务分类新闻头条,定量评价例程应该出现一模一样的每一评估程序:设置模型在求值模式关掉丢弃和反向传播（使用 `classifier.eval()`）,然后遍历测试集以同样的方式作为训练集和验证集。在典型的环境中，您应该尝试不同的训练选项，当您满意时，您应该执行模型评估。我们会把这个留到练习结束。在这个测试集中你能得到的最终准确度是多少?请记住，在整个实验过程中，您只能使用测试集一次。

预测新的新闻头条的类别

训练分类器的目标是将其部署到生产环境中，以便能够对不可见的新闻标题执行推理或预测。要预测尚未处理和数据集中的新闻标题的类别，有几个步骤。第一种是对文本进行预处理，其方式类似于对训练中的数据进行预处理。对于推理，我们对输入使用与训练中相同的预处理函数。该预处理字符串使用训练期间使用的向量化器向量化，并转换为 PyTorch 张量。接下来，对它应用分类器。计算预测向量的最大值以查找类别的名称。示例 5-16 给出了代码。

示例 5-16：使用训练模型做出预测

```

def predict_category(title, classifier, vectorizer, max_length):
    """Predict a News category for a new title

    Args:
        title (str): a raw title string
        classifier (NewsClassifier): an instance of the trained classifier
        vectorizer (NewsVectorizer): the corresponding vectorizer
        max_length (int): the max sequence length
        Note: CNNs are sensitive to the input data tensor size.
    """

```

```

        This ensures to keep it the same size as the training data
    """
    title = preprocess_text(title)
    vectorized_title = \
        torch.tensor(vectorizer.vectorize(title, vector_length=max_length))
    result = classifier(vectorized_title.unsqueeze(0), apply_softmax=True)
    probability_values, indices = result.max(dim=1)
    predicted_category =
vectorizer.category_vocab.lookup_index(indices.item())

    return {'category': predicted_category,
            'probability': probability_values.item()}

```

总结

在本章中，我们研究了单词嵌入，这是一种在空间中将单词（如单词）表示为固定维度向量的方式，使得向量之间的距离编码各种语言属性。重要的是要记住，本章介绍的技术适用于任何离散单元，如句子，段落，文档，数据库记录等。这使得嵌入技术对于深度学习是必不可少的，特别是在 NLP 中。我们展示了如何以黑盒方式使用预训练嵌入。我们简要讨论了直接从数据中学习这些嵌入的几种方法，包括连续词袋（CBOW）方法。我们展示了如何在语言建模的背景下训练 CBOW 模型。最后，我们通过一个在文档分类等任务中使用预训练嵌入和微调嵌入的示例。

不幸的是，本章由于缺乏空间而遗漏了许多重要的主题，例如消除词嵌入，建模上下文和一词多义。语言数据是世界的反映。社会偏见可以通过有偏见的训练语料库编码成模型。在一项研究中，最接近代词“她”的词是家庭主妇，护士，接待员，图书管理员，理发师等，而最接近“他”的词则是外科医生，保护者，哲学家，建筑师，金融家等等。。对这种有偏见的嵌入进行过训练的模型可以继续做出可能产生不公平结果的决策。不再使用单词嵌入仍然是一个新生的领域，我们建议您阅读 Bolukbasi 等人.（2016 年）和最近的论文引用了这一点。我们使用的嵌入词不依赖于上下文。例如，根据上下文，单词 `play` 可能有两个不同的含义，但这里讨论的所有嵌入（embedding）方法都会破坏这两个含义。最近的作品如 Peters（2018）探索了以上下文为条件提供嵌入的方法。

六、自然语言处理的序列模型

本文标题: [Natural-Language-Processing-with-PyTorch \(六\)](#)

文章作者: [Yif Du](#)

发布时间: 2018 年 12 月 24 日 - 12:12

最后更新: 2018 年 12 月 28 日 - 11:12

原始链接: <http://yifdu.github.io/2018/12/24/Natural-Language-Processing-with-PyTorch> (六) /

许可协议: [署名-非商业性使用-禁止演绎 4.0 国际](#) 转载请保留原文链接及作者。

序列是项目的有序集合。传统的机器学习假设数据点是独立的、相同分布的 (IID)，但在许多情况下，如语言、语音和时间序列数据，一个数据项取决于它之前或之后的数据项。这种数据也称为序列数据。在人类语言中，顺序信息无处不在。例如，语音可以被看作是音素的基本单元序列。在像英语这样的语言中，句子中的单词不是随意的。他们可能会被它之前或之后的词所束缚。例如，在英语中，介词 `of` 后面可能跟着冠词 `the`；例如，`The lion is the king of the jungle.`。例如，在许多语言中，包括英语，动词的数量必须与句子主语的数量一致。这里有一个例子：`The book is on the table` `The books are on the table.`。有时这些依赖项或约束可以是任意长的。例如：

`The book that I got yesterday is on the table. The books read by the second grade children are shelved in the lower rack.`。简而言之，理解序列对于理解人类语言至关重要。在前几章中，我们介绍了前馈神经网络，如多层感知器 (MLPs) 和卷积神经网络 (CNNs)，以及向量表示的能力。尽管使用这些技术可以完成大量的自然语言处理 (NLP) 任务，但正如我们将在本章以及第 7 章和第 8 章中学习的那样，它们并不能充分建模序列。

传统的方法，模型序列在 NLP 使用隐马尔科夫模型，条件随机场，和其他类型的概率图形模型，虽然没有讨论在这本书仍然是相关的。我们邀请您 (Koller and Friedman, 2009)。

在深度学习中，建模序列涉及到维护隐藏的“状态信息”或隐藏状态。当序列中的每个条目被匹配时——例如，当一个句子中的每个单词被模型看到时——隐藏状态就会被更新。因此，隐藏状态（通常是一个向量）封装了到目前为止序列所看到的一切。这个隐藏的状态向量，也称为序列表示，可以根据我们要解决的任务以无数种方式在许多序列建模任务中使用，从对序列进行分类到预测序列。在本章中，我们将研究序列数据的分类，但是第 7 章将介绍如何使用序列模型来生成序列。

我们首先介绍最基本的神经网络序列模型:循环神经网络。在此基础上，给出了分类设置中循环神经网络的端到端实例。具体来说，您将看到一个基于字符的 RNN 来将姓氏分类到它们各自的国籍。姓氏示例表明序列模型可以捕获语言中的正字法（子词）模式。这个示例的开发方式使读者能够将模型应用于其他情况，包括建模文本序列，其中数据项是单词而不是字符。

循环神经网络简介

循环神经网络（RNNs）的目的是建立张量序列的模型。rnn 和前馈网络一样，是一类模型。RNN 家族中有几个不同的成员，但在本章中，我们只讨论最基本的形式，有时称为 Elman RNN。循环网络（基本的 Elman 形式和第 7 章中概述的更复杂的形式）的目标是学习序列的表示。这是通过维护一个隐藏的状态向量来实现的，它捕获了序列的当前状态。隐藏状态向量由当前输入向量和前一个隐藏状态向量计算得到。这些关系如图 6-1 所示，图 6-1 显示了计算依赖项的函数（左）视图和“展开”（右）视图。

（左）Elman RNN 的函数视图将递归关系显示为隐藏向量的反馈循环。（右）“展开”视图可以清楚地显示计算关系，因为每个时间步的隐藏向量依赖于该时间步的输入和前一个时间步的隐藏向量。

在每次步骤中使用相同的权重将输入转换为输出是参数共享的另一个例子。在第 4 章中，我们看到了 CNNs 如何跨空间共享参数。CNNs 使用称为内核的参数来计算来自输入数据子区域的输出。卷积核在输入端平移，从每一个可能的位置计算输出，以学习平移不变性。与此相反，rnn 通过依赖一个隐藏的状态向量来捕获序列的状态，从而使用相同的参数来计算每一步的输出。通过这种方式，rnn 的目标是通过计算给定的隐藏状态向量和输入向量的任何输出来学习序列不变性。你可以想象一个 RNN 跨时间共享参数，一个 CNN 跨空间共享参数。

由于单词和句子可以是可变长度的，因此 rnn 或任何序列模型都应该能够处理可变长度的序列。一种可能的技术是人为地将序列限制在一个固定的长度。在本书中，我们使用另一种技术，称为掩蔽，通过利用序列长度的知识来处理可变长度序列。简而言之，屏蔽允许数据在某些输入不应计入梯度或最终输出时发出信号。PyTorch 提供了处理称为打包的序列的可变长度序列的原语，这些序列从这些不太密集的序列中创建密集的张量。“例子:使用字符 RNN 对姓氏国籍进行分类”就是一个例子。

在这两个图中，输出与隐藏向量相同。这并不总是正确的，但是在 Elman RNN 的例子中，隐藏的向量是被预测的。

实现 Elman RNN

为了探究 RNN 的细节，让我们逐步了解 Elman RNN 的一个简单实现。PyTorch 提供了许多有用的类和帮助函数来构建 rnn。PyTorch RNN 类实现了 Elman RNN。在本章中，我们没有直接使用 PyTorch 的 RNN 类，而是使用 `RNNCell`，它是对 RNN 的单个时间步的抽象，并以此构建 RNN。我们这样做的目的是显式地向您展示 RNN 计算。示例 6-1 中显示的类 `ElmanRNN` 利用了 `RNNCell`。`RNNCell` 创建了“循环神经网络导论”中描述的输入隐藏和隐藏权重矩阵。对 `RNNCell`

的每次调用都接受一个输入向量矩阵和一个隐藏向量矩阵。它返回一个步骤产生的隐藏向量矩阵。

除了控制 RNN 中的输入和隐藏大小超参数外，还有一个布尔参数用于指定批量维度是否位于第 0 维度。这个标志也出现在所有 PyTorch RNNs 实现中。当设为真时，RNN 交换输入张量的第 0 维和第 1 维。

在 ElmanRNN 中，`forward()` 方法循环遍历输入张量，以计算每个时间步长的隐藏状态向量。注意，有一个用于指定初始隐藏状态的选项，但如果没有提供，则使用所有 0 的默认隐藏状态向量。当 ElmanRNN 循环遍历输入向量的长度时，它计算一个新的隐藏状态。这些隐藏状态被聚合并最终堆积起来。在返回之前，将再次检查 `batch_first` 标志。如果为真，则输出隐藏向量进行排列，以便批量再次位于第 0 维上。

ElmanRNN 的输出是一个三维张量——对于批量维度上的每个数据点和每个时间步长，都有一个隐藏状态向量。根据手头的任务，可以以几种不同的方式使用这些隐藏向量。您可以使用它们的一种方法是将每个时间步骤分类为一些离散的选项集。该方法是通过调整 RNN 权值来跟踪每一步预测的相关信息。另外，您可以使用最后一个向量来对整个序列进行分类。这意味着 RNN 权重将被调整以跟踪对最终分类重要的信息。在本章中，我们只看到分类设置，但在接下来的两章中，我们将更深入地讨论逐步预测。

示例 6-1：使用 PyTorch 的 `RNNCell` 的 Elman RNN 的实现示例

```
class ElmanRNN(nn.Module):
    """ an Elman RNN built using the RNNCell """
    def __init__(self, input_size, hidden_size, batch_first=False):
        """
        Args:
            input_size (int): size of the input vectors
            hidden_size (int): size of the hidden state vectors
            batch_first (bool): whether the 0th dimension is batch
        """
        super(ElmanRNN, self).__init__()

        self.rnn_cell = nn.RNNCell(input_size, hidden_size)

        self.batch_first = batch_first
        self.hidden_size = hidden_size

    def _initialize_hidden(self, batch_size):
        return torch.zeros((batch_size, self.hidden_size))

    def forward(self, x_in, initial_hidden=None):
        """The forward pass of the ElmanRNN

        Args:
            x_in (torch.Tensor): an input data tensor.
                If self.batch_first: x_in.shape = (batch_size, seq_size,
                feat_size)
                Else: x_in.shape = (seq_size, batch_size, feat_size)
            initial_hidden (torch.Tensor): the initial hidden state for the
```


RNN

```
Returns:
    hiddens (torch.Tensor): The outputs of the RNN at each time step.
    If self.batch_first:
        hiddens.shape = (batch_size, seq_size, hidden_size)
    Else: hiddens.shape = (seq_size, batch_size, hidden_size)
"""
if self.batch_first:
    batch_size, seq_size, feat_size = x_in.size()
    x_in = x_in.permute(1, 0, 2)
else:
    seq_size, batch_size, feat_size = x_in.size()

hiddens = []

if initial_hidden is None:
    initial_hidden = self._initialize_hidden(batch_size)
    initial_hidden = initial_hidden.to(x_in.device)

hidden_t = initial_hidden

for t in range(seq_size):
    hidden_t = self.rnn_cell(x_in[t], hidden_t)
    hiddens.append(hidden_t)

hiddens = torch.stack(hiddens)

if self.batch_first:
    hiddens = hiddens.permute(1, 0, 2)

return hiddens
```

示例：使用字符 RNN 分类姓氏国籍

现在我们已经概述了 RNNs 的基本属性，并逐步实现了 ElmanRNN，现在让我们将它应用到任务中。我们将考虑的任务是第 4 章中的姓氏分类任务，在该任务中，字符序列（姓氏）被分类到起源的国籍。

姓氏数据集

本例中的数据集是姓氏数据集，前面在第 4 章中介绍过。每个数据点由姓氏和相应的国籍表示。我们将避免重复数据集的细节，但是您应该参考“姓氏数据集”来刷新关于数据集的一些关键点。

在本例中，就像“使用 CNN 对姓氏进行分类”一样，我们将每个姓氏视为字符序列。与往常一样，我们实现一个数据集类，如示例 6-2 所示，它返回向量化的姓氏和表示其国籍的整数。此外，返回的是序列的长度，它用于下游计算，以知道序列中的最终向量的位置。这是我们熟悉的步骤序列的一部分——实现数据集、向量化器和词汇表——在实际的训练开始之前。

示例 6-2：为姓氏数据实现数据集

```

class SurnameDataset(Dataset):
    @classmethod
    def load_dataset_and_make_vectorizer(cls, surname_csv):
        """Load dataset and make a new vectorizer from scratch

        Args:
            surname_csv (str): location of the dataset
        Returns:
            an instance of SurnameDataset
        """
        surname_df = pd.read_csv(surname_csv)
        train_surname_df = surname_df[surname_df.split=='train']
        return cls(surname_df,
SurnameVectorizer.from_dataframe(train_surname_df))

    def __getitem__(self, index):
        """the primary entry point method for PyTorch datasets

        Args:
            index (int): the index to the data point
        Returns:
            a dictionary holding the data point's:
                features (x_data)
                label (y_target)
                feature length (x_length)
        """
        row = self._target_df.iloc[index]

        surname_vector, vec_length = \
            self._vectorizer.vectorize(row.surname, self._max_seq_length)

        nationality_index = \
            self._vectorizer.nationality_vocab.lookup_token(row.nationality)

        return {'x_data': surname_vector,
                'y_target': nationality_index,
                'x_length': vec_length}

```

向量化数据结构

向量化管道的第一阶段是将姓氏中的每个字符标记映射到唯一的整数。为了实现这一点，我们使用了 `SequenceVocabulary` 数据结构，这是我们在“示例:使用预先训练的嵌入进行文档分类的传输学习”中首次介绍和描述的。回想一下，这个数据结构不仅将推文中的单词映射到整数，而且还使用了四个特殊用途的标记: `UNK` 标记、`MASK` 标记、`BEGIN-SEQUENCE` 标记和 `END-SEQUENCE` 标记。前两个标记对语言数据至关重要: `UNK` 标记用于输入中看不到的词汇表外标记，而 `MASK` 标记允许处理可变长度的输入。第二个标记为模型提供了句子边界特征，并分别作为前缀和追加到序列中。我们请您参阅“示例:使用预先训练的嵌入来进行文档分类的迁移学习”，以获得关于序列表的更长的描述。

整个向量化过程由 `SurnameVectorizer` 管理，它使用序列 `evocabulary` 来管理姓氏字符和整数之间的映射。示例 6-3 展示了它的实现，看起来应该非常熟悉。在“示例:使用预训练嵌入进行文

档分类的迁移学习”中，我们研究了如何将新闻文章的标题分类到特定的类别中，而向量化管道几乎是相同的。

示例 6-3：姓氏的向量化器

```
class SurnameVectorizer(object):
    """ The Vectorizer which coordinates the Vocabularies and puts them to
    use """
    def vectorize(self, surname, vector_length=-1):
        """
        Args:
            title (str): the string of characters
            vector_length (int): an argument for forcing the length of index
        vector
        """
        indices = [self.char_vocab.begin_seq_index]
        indices.extend(self.char_vocab.lookup_token(token)
                       for token in surname)
        indices.append(self.char_vocab.end_seq_index)

        if vector_length < 0:
            vector_length = len(indices)

        out_vector = np.zeros(vector_length, dtype=np.int64)
        out_vector[:len(indices)] = indices
        out_vector[len(indices):] = self.char_vocab.mask_index

        return out_vector, len(indices)

    @classmethod
    def from_dataframe(cls, surname_df):
        """Instantiate the vectorizer from the dataset dataframe

        Args:
            surname_df (pandas.DataFrame): the surnames dataset
        Returns:
            an instance of the SurnameVectorizer
        """
        char_vocab = SequenceVocabulary()
        nationality_vocab = Vocabulary()

        for index, row in surname_df.iterrows():
            for char in row.surname:
                char_vocab.add_token(char)
            nationality_vocab.add_token(row.nationality)

        return cls(char_vocab, nationality_vocab)
```

SurnameClassifier模型

SurnameClassifier 模型由嵌入层、ElmanRNN 和线性层组成。我们假设模型的输入是在它们被 SequenceVocabulary 映射到整数之后作为一组整数表示的标记。模型首先使用嵌入层嵌入整数。然后，利用 RNN 计算序列表示向量。这些向量表示姓氏中每个字符的隐藏状态。由于目标

是对每个姓氏进行分类，因此提取每个姓氏中最终字符位置对应的向量。考虑这个向量的一种方法是，最后的向量是传递整个序列输入的结果，因此是姓氏的汇总向量。这些汇总向量通过线性层来计算预测向量。预测向量用于训练损失，或者我们可以应用 softmax 函数来创建姓氏的概率分布

模型的参数是:嵌入的大小，嵌入的数量（即类的数量，以及 RNN 的隐藏状态大小。其中两个参数——嵌入的数量和类的数量——由数据决定。其余的超参数是嵌入的大小和隐藏状态的大小。尽管这些模型可以具有任何价值，但通常最好从一些小的、可以快速训练以验证模型是否有效的东西开始。

示例 6-4：实现姓氏分类器模型

```
class SurnameClassifier(nn.Module):
    """ An RNN to extract features & a MLP to classify """
    def __init__(self, embedding_size, num_embeddings, num_classes,
                  rnn_hidden_size, batch_first=True, padding_idx=0):
        """
        Args:
            embedding_size (int): The size of the character embeddings
            num_embeddings (int): The number of characters to embed
            num_classes (int): The size of the prediction vector
                Note: the number of nationalities
            rnn_hidden_size (int): The size of the RNN's hidden state
            batch_first (bool): Informs whether the input tensors will
                have batch or the sequence on the 0th dimension
            padding_idx (int): The index for the tensor padding;
                see torch.nn.Embedding
        """
        super(SurnameClassifier, self).__init__()

        self.emb = nn.Embedding(num_embeddings=num_embeddings,
                                embedding_dim=embedding_size,
                                padding_idx=padding_idx)
        self.rnn = ElmanRNN(input_size=embedding_size,
                             hidden_size=rnn_hidden_size,
                             batch_first=batch_first)
        self.fc1 = nn.Linear(in_features=rnn_hidden_size,
                              out_features=rnn_hidden_size)
        self.fc2 = nn.Linear(in_features=rnn_hidden_size,
                              out_features=num_classes)

    def forward(self, x_in, x_lengths=None, apply_softmax=False):
        """The forward pass of the classifier

        Args:
            x_in (torch.Tensor): an input data tensor.
                x_in.shape should be (batch, input_dim)
            x_lengths (torch.Tensor): the lengths of each sequence in the
batch.
                They are used to find the final vector of each sequence
            apply_softmax (bool): a flag for the softmax activation
                should be false if used with the Cross Entropy losses
        Returns:
```

```

        out (torch.Tensor); `out.shape = (batch, num_classes)`
        """
        x_embedded = self.emb(x_in)
        y_out = self.rnn(x_embedded)

        if x_lengths is not None:
            y_out = column_gather(y_out, x_lengths)
        else:
            y_out = y_out[:, -1, :]

        y_out = F.dropout(y_out, 0.5)
        y_out = F.relu(self.fc1(y_out))
        y_out = F.dropout(y_out, 0.5)
        y_out = self.fc2(y_out)

        if apply_softmax:
            y_out = F.softmax(y_out, dim=-1)

        return y_out

```

您将注意到，正向函数需要序列的长度。长度用于检索从 RNN 返回的带有名为 `column_gather` 函数的张量中每个序列的最终向量，如示例 6-5 所示。该函数迭代批量行索引，并检索位于序列相应长度所指示位置的向量。

示例 6-5：使用 `column_gather` 在每个序列中获取最终向量

```

def column_gather(y_out, x_lengths):
    '''Get a specific vector from each batch datapoint in `y_out`.

    Args:
        y_out (torch.FloatTensor, torch.cuda.FloatTensor)
            shape: (batch, sequence, feature)
        x_lengths (torch.LongTensor, torch.cuda.LongTensor)
            shape: (batch,)

    Returns:
        y_out (torch.FloatTensor, torch.cuda.FloatTensor)
            shape: (batch, feature)
    '''
    x_lengths = x_lengths.long().detach().cpu().numpy() - 1

    out = []
    for batch_index, column_index in enumerate(x_lengths):
        out.append(y_out[batch_index, column_index])

    return torch.stack(out)

```

训练例程和结果

训练程序遵循标准公式。对于单个批数据，应用模型并计算预测向量。利用横熵损失和地面真值来计算损失值。使用损失值和优化器，计算梯度并使用这些梯度更新模型的权重。对训练数据中的每批重复此操作。对验证数据进行类似的处理，但是将模型设置为 `eval()` 模式，以防止在验

证数据上反向传播。相反，验证数据仅用于对模型的执行情况给出不那么偏颇的感觉。这个例程在特定的时期重复执行。代码见补充资料我们鼓励您使用超参数来了解影响性能的因素以及影响程度，并将结果制成表格。我们还将为该任务编写合适的基线模型作为练习，让您完成。

在“SurnameClassifier 模型”中实现的模型是通用的，并不局限于字符。模型中的嵌入层可以映射出离散项序列中的任意离散项;例如，一个句子是一系列的单词。我们鼓励您在其他序列分类任务（如句子分类）中使用示例 6-6 中的代码。

示例 6-6：基于 RNN 的姓氏分类器的参数

```
args = Namespace(  
    # Data and path information  
    surname_csv="data/surnames/surnames_with_splits.csv",  
    vectorizer_file="vectorizer.json",  
    model_state_file="model.pth",  
    save_dir="model_storage/ch6/surname_classification",  
    # Model hyper parameter  
    char_embedding_size=100,  
    rnn_hidden_size=64,  
    # Training hyper parameter  
    num_epochs=100,  
    learning_rate=1e-3,  
    batch_size=64,  
    seed=1337,  
    early_stopping_criteria=5,  
    # ... Runtime options not shown for space  
)
```

总结

在本章中，您学习了用于对序列数据建模的循环神经网络，以及最简单的一种循环网络，即 Elman RNNs。我们确定序列建模的目标是学习序列的表示（即向量）。根据任务的不同，可以以不同的方式使用这种学习过的表示。我们考虑了一个示例任务，涉及到将这种隐藏状态表示分类到许多类中的一个。姓氏分类任务展示了一个使用 RNNs 在子词级别捕获信息的示例。

七、自然语言处理的进阶序列模型

本文标题: [Natural-Language-Processing-with-PyTorch \(七\)](#)

文章作者: [Yif Du](#)

发布时间: 2018 年 12 月 26 日 - 09:12

最后更新: 2018 年 12 月 28 日 - 11:12

原始链接: <http://yifdu.github.io/2018/12/26/Natural-Language-Processing-with-PyTorch> (七) /

许可协议: [署名-非商业性使用-禁止演绎 4.0 国际](#) 转载请保留原文链接及作者。

本章的目标是序列预测。序列预测任务要求我们对序列中的每一项进行标记。这类任务在自然语言处理 (NLP) 中很常见。一些例子包括语言建模 (参见图 7-1), 在语言建模中, 我们在每个步骤中给定单词序列预测下一个单词; 词性标注的一部分, 对每个词的语法词性进行预测; 命名实体识别, 我们预测每个词是否属于一个命名实体, 如人、位置、产品、组织; 等等。有时, 在 NLP 文献中, 序列预测任务也被称为序列标记。

虽然理论上我们可以使用第六章中介绍的 Elman 循环神经网络 (RNNs) 进行序列预测任务, 但是在实际应用中, 它们并不能很好地捕捉到长期的依赖关系, 并且表现不佳。在本章中, 我们将花一些时间来理解为什么会这样, 并学习一种新的 RNN 架构, 称为门控网络。

我们还介绍了自然语言生成作为序列预测应用的任务, 并探讨了输出序列在某种程度上受到约束的条件生成。

原始 RNN (Elman RNNs) 的问题

尽管在第 6 章中讨论的普通 RNN/Elman-RNN 非常适合于建模序列, 但它有两个问题使其不适用于许多任务: 无法保留用于长期预测的信息, 以及梯度稳定性。为了理解这两个问题, 回想一下, 在它们的核心, rnn 在每个时间步上使用前一个时间步的隐藏状态向量和当前时间步上的输入向量计算一个隐藏状态向量。正是这种核心计算使得 RNN 如此强大, 但也产生了大量的数值问题。

Elman RNNs 的第一个问题是很难记住长期的信息。例如, 在第 6 章的 RNN 中, 在每次步骤中, 我们仅仅更新隐藏的状态向量, 而不管它是否有意义。因此, RNN 无法控制隐藏状态中保留的值和丢弃的值, 而这些值完全由输入决定。直觉上, 这是说不通的。我们希望 RNN 通过某种方式来决定更新是可选的, 还是发生了更新, 以及状态向量的多少和哪些部分, 等等。

Elman RNNs 的第二个问题是，它们会导致梯度螺旋地失去控制，趋近于零或无穷大。不稳定的梯度，可以螺旋失控被称为消失梯度或爆炸梯度取决于方向梯度的绝对值正在收缩/增长。梯度绝对值非常大或非常小（小于 1）都会使优化过程不稳定（Hochreiter et al., 2001; Pascanu et al., 2013）。

在一般的神经网络中，有解决这些梯度问题的方法，如使用校正的线性单元（relu）、梯度裁剪和小心初始化。但是没有一种解决方案能像门控技术那样可靠地工作。

原始 RNN 的挑战的门控解决方案

为了直观地理解门控，假设您添加了两个量， a 和 b ，但是您想控制 b 放入和的多少。数学上，你可以把 $a + b$ 的和改写成 $a + \lambda b$ ， λ 是一个值在 0 和 1 之间。如果 $\lambda = 0$, b 没有贡献，如果 $\lambda = 1$, b 完全贡献。这种方式看,你可以解释 λ 充当一个“开关”或“门”控制的 b 进入之和。这就是门控机制背后的直觉。现在，让我们重新访问 Elman RNN，看看如何将门控与普通的 RNN 合并以进行条件更新。如果前面的隐藏状态是 $h[t-1]$ 和当前输入 $x[t]$ ，Elman RNN 的周期性更新看起来像

其中 F 是 RNN 的递归计算。显然，这是一个无条件的和，并且有“原始 RNNs（或 Elman RNNs）的问题”中描述的缺点。现在想象一下,替代常数,如果前面的例子的 λ 是一个函数之前的隐藏状态向量 $h[t-1]$ 和当前输入 $x[t]$,而且还产生所需的控制行为;也就是说，0 到 1 之间的值。通过这个门控函数，我们的 RNN 更新方程如下：

现在就清楚函数 λ 控制当前输入的多少可以更新状态 $h[t-1]$ 。进一步的函数 λ 是上下文相关的。这是所有门控网络的基本直觉。 λ 的函数通常是一个 s 形的函数,我们知道从第三章到产生一个值在 0 和 1 之间。

在长短期记忆的情况下,这个基本的直觉是扩展仔细将不仅条件更新,而且还故意忘记之前的隐藏状态 $h[t-1]$ 的值。这种“忘记”乘以发生前隐藏状态与另一个函数 μ 值 $h[t-1]$,还产生值在 0 和 1 之间,取决于当前的输入：

您可能已经猜到, μ 是另一个控制功能。在实际的 LSTM 描述中，这变得很复杂，因为门函数是参数化的，导致对未初始化的操作的复杂序列。但是，在掌握了本节的直观知识之后，如果您想深入了解 LSTM 的更新机制，现在就可以了。我们推荐 Christopher Olah 的经典文章。在本书中，我们将不涉及这些内容，因为这些细节对于 LSTMs 在 NLP 应用程序中的应用和使用并不是必需的。

LSTM 只是 RNN 的许多门控变体之一。另一种越来越流行的门控变量是门控循环单元

（GRU; Chung et al., 2015）。幸运的是，在 PyTorch 中，您可以简单地替换 `nn.LSTM` 或神经网络。`nn.LSTMCell` 没有其他代码更改来切换到 LSTM（为 GRU 做必要的修改）！

门控机制是“普通 RNNs（或 Elman RNNs）问题”中列举的问题的有效解决方案。它不仅控制更新，还可以控制梯度问题，使训练相对容易。不再赘述，我们将使用两个示例来展示这些封闭体系结构的实际应用。

示例：用于生成姓氏的字符 RNN

在本例中，我们将完成一个简单的序列预测任务：使用 RNNs 生成姓氏。在实践中，这意味着对于每个时间步骤，RNN 都在计算姓氏中可能的字符集的概率分布。使用这些概率分布，我们可以优化网络来改进它的预测（假设我们知道应该预测哪些字符），也可以使用它们来生成全新的姓氏！

虽然这个任务的数据集已经在前面的例子中使用过，看起来很熟悉，但是在构建用于序列预测的每个数据样本的方式上有一些不同。在描述了数据集和任务之后，概述了支持通过系统簿记进行序列预测的数据结构。

在描述了数据集、任务和支持数据结构之后，我们引入了两个生成姓氏的模型：无条件姓氏生成模型和条件姓氏生成模型。该模型在不了解姓氏的情况下，对姓氏序列进行预测。与此相反，条件模型利用特定的国籍嵌入作为 RNN 的初始隐藏状态，从而使模型对序列的预测产生偏差。

SurnamesDataset

姓氏数据集是姓氏及其来源国的集合，最早出现在“带有多层感知器的姓氏分类”中。到目前为止，该数据集已经被用于一个分类任务——给出一个新的姓氏，正确地将姓氏来自哪个国家。然而，在本例中，我们将展示如何使用数据集来训练一个模型，该模型可以为字符序列分配概率并生成新的序列。

`SurnamesDataset` 类与前几章基本相同：我们使用 `panda DataFrame` 加载数据集，并构造了一个向量化器，它将标记封装为模型和手边任务所需的整数映射。为了适应任务的不同，修改了 `SurnamesDataset.__getitem__()` 方法，以输出预测目标的整数序列，如示例 7-1 所示。该方法引用向量器来计算作为输入的整数序列（`from_vector`）和作为输出（`to_vector`）的整数序列。下一小节将描述向量化的实现。

示例 7-1：用于序列预测任务的 `SurnamesDataset.__getitem__` 方法

```
class SurnameDataset(Dataset):
    @classmethod
    def load_dataset_and_make_vectorizer(cls, surname_csv):
        """Load dataset and make a new vectorizer from scratch

        Args:
            surname_csv (str): location of the dataset
        Returns:
            an instance of SurnameDataset
        """
```

```

surname_df = pd.read_csv(surname_csv)
return cls(surname_df, SurnameVectorizer.from_dataframe(surname_df))

def __getitem__(self, index):
    """the primary entry point method for PyTorch datasets

    Args:
        index (int): the index to the data point
    Returns:
        a dictionary holding the data point: (x_data, y_target,
class_index)
    """
    row = self._target_df.iloc[index]

    from_vector, to_vector = \
        self._vectorizer.vectorize(row.surname, self._max_seq_length)

    nationality_index = \
        self._vectorizer.nationality_vocab.lookup_token(row.nationality)

    return {'x_data': from_vector,
            'y_target': to_vector,
            'class_index': nationality_index}

```

向量化数据结构

与前面的示例一样，有三种主要的数据结构将每个姓氏的字符序列转换为其向量化形式: `SequenceVocabulary` 将单个标记映射到整数，`SurnameVectorizer` 协调整数映射，`DataLoader` 将 `SurnameVectorizer` 的结果分组为小批。由于 `DataLoader` 实现及其使用在本例中保持不变，我们将跳过其实现细节。

SURNAMEVECTORIZER和END-OF-SEQUENCE

对于序列预测任务，编写训练例程以期望在每个时间步骤中出现两个表示标记观察和标记目标的整数序列。通常，我们只想预测我们正在训练的序列，例如本例中的姓氏。这意味着我们只有一个标记序列可以使用，并通过调整这个标记序列来构造观察和目标。

为了将其转化为序列预测问题，使用 `SequenceVocabulary` 将每个标记映射到其适当的索引。然后，序列的起始标记索引 `begin_seq_index` 位于序列的开头，而序列的结束标记索引 `end_seq_index` 位于序列的末尾。此时，每个数据点都是一系列索引，具有相同的第一个和最后一个索引。要创建训练例程所需的输入和输出序列，我们只需使用索引序列的两个切片:第一个切片包含除最后一个之外的所有标记索引，第二个切片包含除第一个之外的所有标记索引。当排列和配对在一起时，序列就是正确的输入-输出索引。

为了更明确，我们展示了 `SurnameVectorizer` 的代码。在示例 7-2 中向量化。第一步是将姓氏（字符串）映射到索引（表示这些字符的整数列表）。然后，用序列索引的开始和结束来包装索引:具体来说，`begin_seq_index` 在索引之前，`end_seq_index` 在索引之后。接下来，我们测试 `vector_length`，它通常在运行时提供，但是代码的编写允许向量的任何长度。在训练期间，

提供 `vector_length` 是很重要的，因为小批是由堆叠的向量表示构造的。如果向量的长度不同，它们不能堆放在一个矩阵中。在测试 `vector_length` 之后，创建两个向量: `from_vector` 和 `to_vector`。不包含最后一个索引的索引片放在 `from_vector` 中，不包含第一个索引的索引片放在 `to_vector` 中。每个向量的剩余位置都填充了 `mask_index`。将序列填充（或填充）到右边是很重要的，因为空位置将改变输出向量，我们希望这些变化发生在序列被看到之后。

示例 7-2: 序列预测任务中的 `SurnameVectorizer.vectorize` 代码区

```
class SurnameVectorizer(object):
    """ The Vectorizer which coordinates the Vocabularies and puts them to
    use """
    def vectorize(self, surname, vector_length=-1):
        """Vectorize a surname into a vector of observations and targets

        Args:
            surname (str): the surname to be vectorized
            vector_length (int): an argument for forcing the length of index
        """
        Returns:
            a tuple: (from_vector, to_vector)
            from_vector (numpy.ndarray): the observation vector
            to_vector (numpy.ndarray): the target prediction vector
        """
        indices = [self.char_vocab.begin_seq_index]
        indices.extend(self.char_vocab.lookup_token(token) for token in
            surname)
        indices.append(self.char_vocab.end_seq_index)

        if vector_length < 0:
            vector_length = len(indices) - 1

        from_vector = np.zeros(vector_length, dtype=np.int64)
        from_indices = indices[:-1]
        from_vector[:len(from_indices)] = from_indices
        from_vector[len(from_indices):] = self.char_vocab.mask_index

        to_vector = np.empty(vector_length, dtype=np.int64)
        to_indices = indices[1:]
        to_vector[:len(to_indices)] = to_indices
        to_vector[len(to_indices):] = self.char_vocab.mask_index

        return from_vector, to_vector

    @classmethod
    def from_dataframe(cls, surname_df):
        """Instantiate the vectorizer from the dataset dataframe

        Args:
            surname_df (pandas.DataFrame): the surname dataset
        Returns:
            an instance of the SurnameVectorizer
        """
        char_vocab = SequenceVocabulary()
        nationality_vocab = Vocabulary()
```

```

for index, row in surname_df.iterrows():
    for char in row.surname:
        char_vocab.add_token(char)
        nationality_vocab.add_token(row.nationality)

return cls(char_vocab, nationality_vocab)

```

从ElmanRNN到 GRU

在实践中，从普通的 RNN 转换到门控变体是很容易的。在以下模型中，虽然我们使用 GRU 代替普通的 RNN，但是使用 LSTM 也同样容易。为了使用 GRU，我们实例化了 `torch.nn.GRU`。GRU 模块使用与第六章 ElmanRNN 相同的参数。

模型 1：非条件姓氏生成模型

第一个模型是无条件的：它在生成姓氏之前不观察国籍。在实践中，非条件意味着 GRU 的计算不偏向任何国籍。在下一个例子（例子 7-3）中，通过初始隐藏向量引入计算偏差。在这个例子中，我们使用一个全为 0 的向量，这样初始的隐藏状态向量就不会影响计算。通常，

`SurnameGenerationModel` 嵌入字符索引，使用 GRU 计算其顺序状态，并使用线性层计算标记预测的概率。更明确地说，非条件 `SurnameGenerationModel` 从初始化嵌入层、GRU 和线性层开始。与第 6 章的序列模型相似，该模型输入了一个整数矩阵。我们使用一个 PyTorch 嵌入实例 `char_embed` 将整数转换为一个三维张量（每个批量项的向量序列）。这个张量传递给 GRU，GRU 计算每个序列中每个位置的状态向量。

第六章的序列分类与本章序列预测的主要区别在于如何处理由 RNN 计算出的状态向量。在第 6 章中，我们为每个批量索引检索一个向量，并使用这些向量执行预测。在这个例子中，我们将我们的三维张量重塑为一个二维张量（一个矩阵），以便行维表示每个样本（批量和序列索引）。利用这个矩阵和线性层，我们计算每个样本的预测向量。我们通过将矩阵重新构造成一个三维张量来完成计算。由于排序信息是通过整形操作保存的，所以每个批和序列索引仍然处于相同的位置。我们需要整形的原因是因为线性层需要一个矩阵作为输入。

示例 7-3：非条件化的姓氏生成模型

```

class SurnameGenerationModel(nn.Module):
    def __init__(self, char_embedding_size, char_vocab_size, rnn_hidden_size,
                  batch_first=True, padding_idx=0, dropout_p=0.5):
        """
        Args:
            char_embedding_size (int): The size of the character embeddings
            char_vocab_size (int): The number of characters to embed
            rnn_hidden_size (int): The size of the RNN's hidden state
            batch_first (bool): Informs whether the input tensors will
                                have batch or the sequence on the 0th dimension
            padding_idx (int): The index for the tensor padding;
                               see torch.nn.Embedding
            dropout_p (float): the probability of zeroing activations using

```

```

        the dropout method.
    """
    super(SurnameGenerationModel, self).__init__()

    self.char_emb = nn.Embedding(num_embeddings=char_vocab_size,
                                  embedding_dim=char_embedding_size,
                                  padding_idx=padding_idx)

    self.rnn = nn.GRU(input_size=char_embedding_size,
                      hidden_size=rnn_hidden_size,
                      batch_first=batch_first)

    self.fc = nn.Linear(in_features=rnn_hidden_size,
                        out_features=char_vocab_size)

    self._dropout_p = dropout_p

def forward(self, x_in, apply_softmax=False):
    """The forward pass of the model

    Args:
        x_in (torch.Tensor): an input data tensor.
            x_in.shape should be (batch, input_dim)
        apply_softmax (bool): a flag for the softmax activation
            should be False during training

    Returns:
        the resulting tensor. tensor.shape should be (batch, output_dim)
    """
    x_embedded = self.char_emb(x_in)

    y_out, _ = self.rnn(x_embedded)

    batch_size, seq_size, feat_size = y_out.shape
    y_out = y_out.contiguous().view(batch_size * seq_size, feat_size)

    y_out = self.fc(F.dropout(y_out, p=self._dropout_p))

    if apply_softmax:
        y_out = F.softmax(y_out, dim=-1)

    new_feat_size = y_out.shape[-1]
    y_out = y_out.view(batch_size, seq_size, new_feat_size)

    return y_out

```

模型 2：条件姓氏生成模型

第二个模型考虑了要生成的姓氏的国籍。在实践中，这意味着有某种机制允许模型对特定姓氏的行为进行偏差。在本例中，我们通过将每个国籍嵌入为隐藏状态大小的向量来参数化 RNNs 的初始隐藏状态。这意味着模型在调整模型参数的同时，也调整了嵌入矩阵中的值，从而使预测偏于对特定的国籍和姓氏的规律性更加敏感。例如，爱尔兰国籍向量偏向于起始序列 m_c 和 o 。

例 7-3 显示了条件模型之间的差异。具体地说，引入额外的嵌入来将国籍索引映射到与 RNN 的隐藏层相同大小的向量。然后，在正向函数中嵌入民族指标，作为 RNN 的初始隐含层简单传入。虽然这是对第一个模型的一个非常简单的修改，但是它对于让 RNN 根据生成的国籍改变其行为有着深远的影响。

示例 7-4：条件化的姓氏生成模型

```
class SurnameGenerationModel(nn.Module):
    def __init__(self, char_embedding_size, char_vocab_size,
                 num_nationalities,
                 rnn_hidden_size, batch_first=True, padding_idx=0,
                 dropout_p=0.5):
        # ...
        self.nation_embedding = nn.Embedding(embedding_dim=rnn_hidden_size,
                                              num_embeddings=num_nationalities)

    def forward(self, x_in, nationality_index, apply_softmax=False):
        # ...
        x_embedded = self.char_embedding(x_in)
        # hidden_size: (num_layers * num_directions, batch_size,
        #                rnn_hidden_size)
        nationality_embedded = self.nation_emb(nationality_index).unsqueeze(0)
        y_out, _ = self.rnn(x_embedded, nationality_embedded)
        # ...
```

训练例程和结果

在本例中，我们介绍了用于生成姓氏的字符序列预测任务。虽然许多实现细节和训练例程与第 6 章的序列分类示例相似，但有几个主要区别。在这一节中，我们将重点讨论差异、使用的超参数和结果。与前面的例子相比，计算这个例子中的损失需要两个更改，因为我们在序列中的每一步都要进行预测。首先，我们将三维张量重塑为二维张量（矩阵）以满足计算约束。其次，我们协调掩蔽索引，它允许可变长度序列与损失函数，使损失不使用掩蔽位置在其计算。

我们通过使用示例 7-5 中所示的代码片段来处理这两个问题——三维张量和可变长度序列。首先，预测和目标被标准化为损失函数期望的大小（预测是二维的，目标是一维的）。现在，每一行代表一个示例：按顺序执行一个时间步骤。然后，交叉熵损失用于 `ignore_index` 设置为 `mask_index`。这将导致损失函数忽略与 `ignore_index` 匹配的目标中的任何位置。

示例 7-5：处理三维张量和序列级损失计算

```
def normalize_sizes(y_pred, y_true):
    """Normalize tensor sizes

    Args:
        y_pred (torch.Tensor): the output of the model
            If a 3-dimensional tensor, reshapes to a matrix
        y_true (torch.Tensor): the target predictions
            If a matrix, reshapes to be a vector
    """
```



```

if len(y_pred.size()) == 3:
    y_pred = y_pred.contiguous().view(-1, y_pred.size(2))
if len(y_true.size()) == 2:
    y_true = y_true.contiguous().view(-1)
return y_pred, y_true

def sequence_loss(y_pred, y_true, mask_index):
    y_pred, y_true = normalize_sizes(y_pred, y_true)
    return F.cross_entropy(y_pred, y_true, ignore_index=mask_index)

```

使用这种修正的损失计算，我们构造了一个训练例程，看起来与本书中的每个例子相似。它首先迭代训练数据集，每次只处理一小批数据。对于每个小批量，模型的输出是由输入计算出来的。因为我们在每一个时间步上执行预测，所以模型的输出是一个三维张量。使用前面描述的 `sequence_loss` 和优化器，可以计算模型预测的错误信号，并用于更新模型参数。

大多数模型超参数是由字符词汇表的大小决定的。这个大小是可以观察到的作为模型输入的离散标记的数量，以及每次步骤输出分类中的类的数量。剩下的模型超参数是字符嵌入的大小和内部 RNN 隐藏状态的大小。示例 7-6 给出了这些超参数和训练选项。

示例 7-6：姓氏生成的超参数

```

args = Namespace(
    # Data and Path information
    surname_csv="data/surnames/surnames_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch7/model1_unconditioned_surname_generation",
    # or: save_dir="model_storage/ch7/model2_conditioned_surname_generation",
    # Model hyper parameters
    char_embedding_size=32,
    rnn_hidden_size=32,
    # Training hyper parameters
    seed=1337,
    learning_rate=0.001,
    batch_size=128,
    num_epochs=100,
    early_stopping_criteria=5,
    # Runtime options omitted for space
)

```

尽管预测的每个字符的准确性是模型性能的度量，但是在本例中，通过检查模型将生成的姓氏类型来进行定性评估会更好。为此，我们在 `forward()` 方法中步骤的修改版本上编写一个新的循环，以计算每个时间步骤的预测，并将这些预测用作下一个时间步骤的输入。我们将展示示例 7-7 中的代码。模型在每个时间步上的输出是一个预测向量，利用 `softmax` 函数将预测向量转换为概率分布。利用概率分布，我们利用火炬。多项式抽样函数，它以与索引的概率成比例的速率选择索引。抽样是一个每次产生不同输出的随机过程。

示例 7-7：从非条件化生成模型采样


```

def sample_from_model(model, vectorizer, num_samples=1, sample_size=20,
                      temperature=1.0):
    """Sample a sequence of indices from the model

    Args:
        model (SurnameGenerationModel): the trained model
        vectorizer (SurnameVectorizer): the corresponding vectorizer
        num_samples (int): the number of samples
        sample_size (int): the max length of the samples
        temperature (float): accentuates or flattens
            the distribution.
            0.0 < temperature < 1.0 will make it peakier.
            temperature > 1.0 will make it more uniform

    Returns:
        indices (torch.Tensor): the matrix of indices;
        shape = (num_samples, sample_size)
    """
    begin_seq_index = [vectorizer.char_vocab.begin_seq_index
                        for _ in range(num_samples)]
    begin_seq_index = torch.tensor(begin_seq_index,
                                   dtype=torch.int64).unsqueeze(dim=1)

    indices = [begin_seq_index]
    h_t = None

    for time_step in range(sample_size):
        x_t = indices[time_step]
        x_emb_t = model.char_emb(x_t)
        rnn_out_t, h_t = model.rnn(x_emb_t, h_t)
        prediction_vector = model.fc(rnn_out_t.squeeze(dim=1))
        probability_vector = F.softmax(prediction_vector / temperature, dim=1)
        indices.append(torch.multinomial(probability_vector, num_samples=1))
    indices = torch.stack(indices).squeeze().permute(1, 0)
    return indices

```

您需要将采样的索引从 `sample_from_model()` 函数转换为用于人类可读输出的字符串。如示例 7-8 所示，要做到这一点，需要使用用于向量化姓氏的 `SequenceVocabulary`。在创建字符串时，只使用序列结束索引之前的索引。这是假设模型能够了解姓氏应该在何时结束。

示例 7-8：将采样的索引映射为姓氏字符串

```

def decode_samples(sampled_indices, vectorizer):
    """Transform indices into the string form of a surname

    Args:
        sampled_indices (torch.Tensor): the indices from `sample_from_model`
        vectorizer (SurnameVectorizer): the corresponding vectorizer
    """
    decoded_surnames = []
    vocab = vectorizer.char_vocab

    for sample_index in range(sampled_indices.shape[0]):
        surname = ""
        for time_step in range(sampled_indices.shape[1]):
            sample_item = sampled_indices[sample_index, time_step].item()

```

```

        if sample_item == vocab.begin_seq_index:
            continue
        elif sample_item == vocab.end_seq_index:
            break
        else:
            surname += vocab.lookup_index(sample_item)
        decoded_surnames.append(surname)
    return decoded_surnames

```

使用这些函数，您可以检查模型的输出，如示例 7-9 所示，以了解模型是否正在学习生成合理的姓氏。从检查输出中我们可以学到什么？我们可以看到，尽管这些姓氏似乎遵循着几种形态模式，但这些姓氏显然并不是来自一个国家或另一个国家。一种可能是，学习姓氏的一般模型会混淆不同民族之间的性格分布。有条件的姓氏生成模型就是用来处理这种情况的。

示例 7-9：从非条件化模型采样

```

Input[0]
samples = sample_from_model(unconditioned_model, vectorizer,
                             num_samples=10)
decode_samples(samples, vectorizer)
Output[0]
['Aqtaliby',
 'Yomaghev',
 'Mauasheev',
 'Unander',
 'Virrovo',
 'NInev',
 'Bukhumohe',
 'Burken',
 'Rati',
 'Jzirmar']

```

对于有条件的 SurnameGenerationModel，我们修改 sample_from_model() 函数来接受国籍索引列表，而不是指定数量的样本。在例 7-10 中，修改后的函数使用带有国籍嵌入的国籍索引来构造 GRU 的初始隐藏状态。在此之后，采样过程与非条件模型完全相同。

示例 7-10：从序列模型采样

```

def sample_from_model(model, vectorizer, nationalities, sample_size=20,
                      temperature=1.0):
    """Sample a sequence of indices from the model

    Args:
        model (SurnameGenerationModel): the trained model
        vectorizer (SurnameVectorizer): the corresponding vectorizer
        nationalities (list): a list of integers representing nationalities
        sample_size (int): the max length of the samples
        temperature (float): accentuates or flattens
            the distribution.
            0.0 < temperature < 1.0 will make it peakier.
            temperature > 1.0 will make it more uniform

    Returns:

```

```

        indices (torch.Tensor): the matrix of indices;
        shape = (num_samples, sample_size)
    """
    num_samples = len(nationalities)
    begin_seq_index = [vectorizer.char_vocab.begin_seq_index
                       for _ in range(num_samples)]
    begin_seq_index = torch.tensor(begin_seq_index,
                                   dtype=torch.int64).unsqueeze(dim=1)

    indices = [begin_seq_index]
    nationality_indices = torch.tensor(nationalities,
                                      dtype=torch.int64).unsqueeze(dim=0)

    h_t = model.nation_emb(nationality_indices)

    for time_step in range(sample_size):
        x_t = indices[time_step]
        x_emb_t = model.char_emb(x_t)
        rnn_out_t, h_t = model.rnn(x_emb_t, h_t)
        prediction_vector = model.fc(rnn_out_t.squeeze(dim=1))
        probability_vector = F.softmax(prediction_vector / temperature, dim=1)
        indices.append(torch.multinomial(probability_vector, num_samples=1))
    indices = torch.stack(indices).squeeze().permute(1, 0)
    return indices

```

用条件向量采样的有效性意味着我们对生成输出有影响。在示例 7-11 中，我们迭代国籍索引并从每个索引中取样。为了节省空间，我们只显示一些输出。从这些输出中，我们可以看到，该模型确实采用了姓氏拼写的一些模式。

示例 7-11：从条件 SurnameGenerationModel 采样（没有展示所有输出）

```

Input[0]
for index in range(len(vectorizer.nationality_vocab)):
    nationality = vectorizer.nationality_vocab.lookup_index(index)

    print("Sampled for {}: ".format(nationality))

    sampled_indices = sample_from_model(model=conditioned_model,
                                       vectorizer=vectorizer,
                                       nationalities=[index] * 3,
                                       temperature=0.7)

    for sampled_surname in decode_samples(sampled_indices,
                                       vectorizer):
        print("- " + sampled_surname)
Output[0]
Sampled for Arabic:
- Khatso
- Salbwa
- Gadi
Sampled for Chinese:
- Lie
- Puh
- Pian
Sampled for German:
- Lenger
- Schanger

```

```
- Schumper
Sampled for Irish:
- Mcochin
- Corran
- O'Baintin
Sampled for Russian:
- Mahghatsunkov
- Juhin
- Karkovin
Sampled for Vietnamese:
- Lo
- Tham
- Tou
```

训练序列模型的提示和技巧

序列模型很难训练，而且在这个过程中会出现许多问题。在这里，我们总结了一些技巧和技巧，我们发现不仅在我们的工作中有用，而且也被其他人在文献报道。1) 如果可能，使用门控变量门控体系结构通过解决非通配型的许多数值稳定性问题简化了训练。2) 如果可能，请选择 GRUs 而不是 LSTMs GRUs 提供了与 LSTMs 几乎相同的性能，并且使用更少的参数和计算。幸运的是，从 PyTorch 的角度来看，除了简单地使用不同的模块类之外，在 LSTM 上使用 GRU 没有什么可做的。3) 使用 Adam 作为您的优化器 在第 6 章、第 7 章和第 8 章中，我们只使用 Adam 作为优化器，这是有充分理由的:它是可靠的，收敛速度更快。对于序列模型尤其如此。如果由于某些原因，您的模型没有与 Adam 收敛，那么在这种情况下，切换到随机梯度下降可能会有所帮助。4) 梯度剪裁 如果您注意到在应用这些章节中学习到的概念时出现了数字错误，请在训练过程中使用您的代码绘制梯度值。知道愤怒之后，剪掉任何异常值。这将确保更顺利的训练。在 PyTorch 中，有一个有用的实用程序 `clip_grad_norm` 可以为您完成此工作，如示例 7-12 所示。一般来说，你应该养成剪切渐变的习惯。

示例 7-12: 在 PyTorch 中应用梯度剪裁

```
# define your sequence model
model = ..
# define loss function
loss_function = ..

# training loop
for _ in ...:
    ...
    model.zero_grad()
    output, hidden = model(data, hidden)
    loss = loss_function(output, targets)
    loss.backward()
    torch.nn.utils.clip_grad_norm(model.parameters(), 0.25)
    ...
```

5.早期停止 对于序列模型，很容易过度拟合。我们建议您在评估错误（在开发集上测量的）开始出现时尽早停止训练过程。

在第 8 章中，我们继续讨论序列模型，使用序列到序列模型来预测和生成与输入长度不同的序列，并讨论序列模型的更多变体。

八、自然语言处理的高级序列模型

本文标题: [Natural-Language-Processing-with-PyTorch \(八\)](#)

文章作者: [Yif Du](#)

发布时间: 2018 年 12 月 28 日 - 09:12

最后更新: 2018 年 12 月 28 日 - 11:12

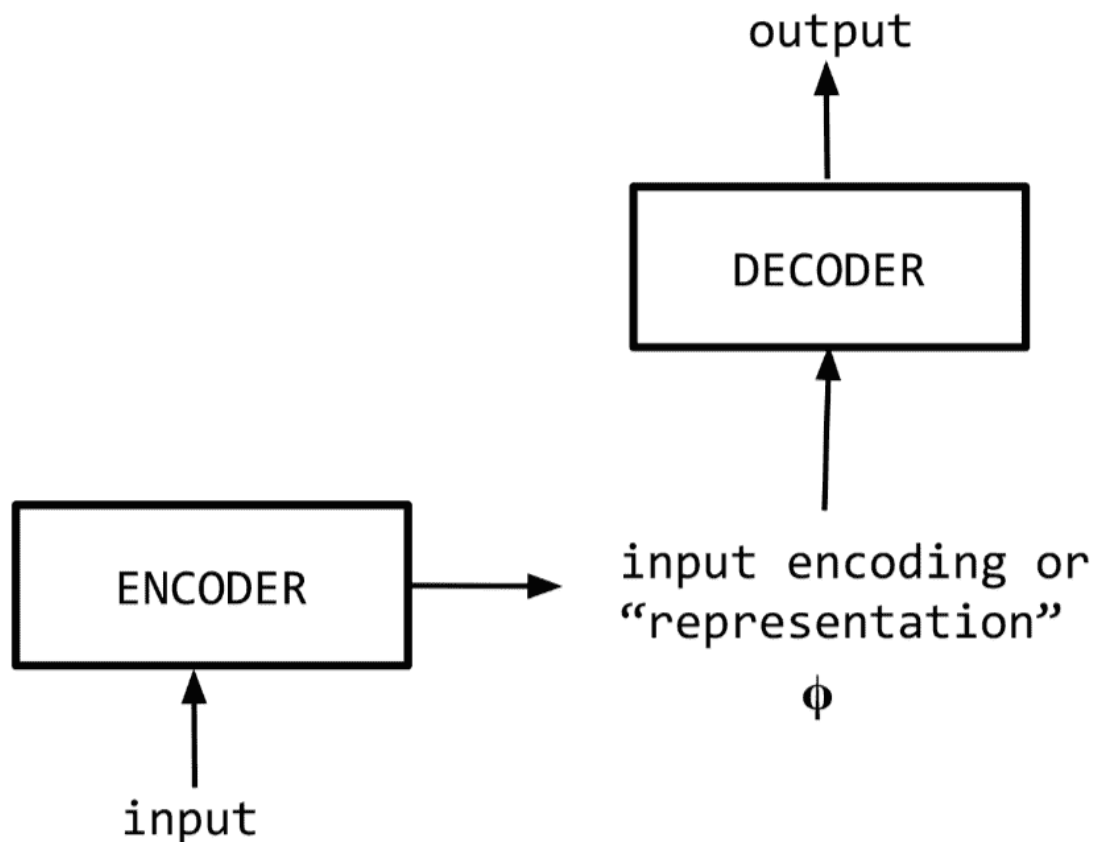
原始链接: [http://yifdu.github.io/2018/12/28/Natural-Language-Processing-with-PyTorch \(八\) /](http://yifdu.github.io/2018/12/28/Natural-Language-Processing-with-PyTorch (八) /)

许可协议: [署名-非商业性使用-禁止演绎 4.0 国际](#) 转载请保留原文链接及作者。

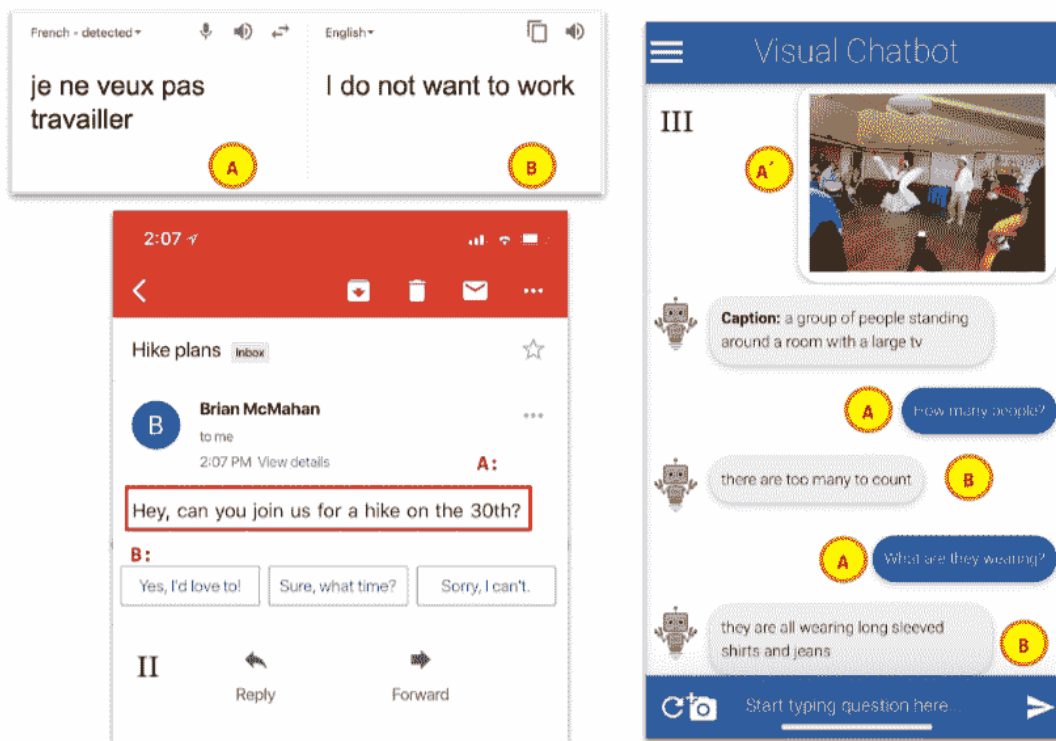
在本章中，我们以第六章和第七章讨论的序列建模概念为基础，将它们扩展到序列到序列建模的领域，其中模型以一个序列作为输入，并产生另一个可能不同长度的序列作为输出。序列对序列问题的例子随处可见。例如，给定一封电子邮件，我们可能希望预测响应。给出一个法语句子，预测它的英语翻译。或者，给定一篇文章，写一篇摘要。我们还讨论了序列模型的结构变体，特别是双向模型。为了最大限度地利用序列表示，我们介绍了注意机制并对其进行了深入讨论。最后，本章以实现本章描述的概念的神经机器翻译的详细演练结束。

序列到序列模型，编码器-解码器模型，和条件生成

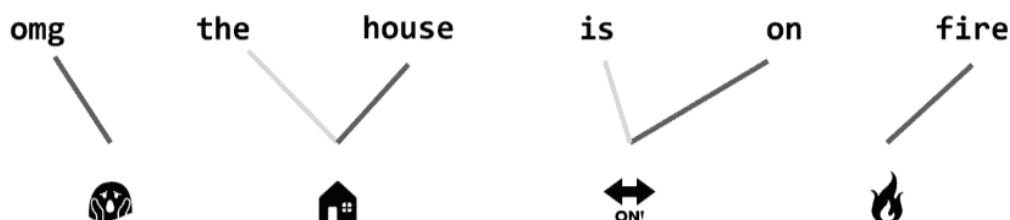
序列到序列 (S2S) 模型是一种称为编码器-解码器模型的一般模型家族的特殊情况。编码器-解码器模型是两个模型 (图 8-1) 的组合，一个是“编码器”模型，另一个是“解码器”模型，这两个模型通常是联合训练的。编码器模型需要输入并产生一个编码或表示 ϕ 的输入,通常一个向量。编码器的目标是捕获与当前任务相关的输入的重要属性。解码器的目标是获取编码输入并产生所需的输出。通过对编码器和解码器的理解，我们将 S2S 模型定义为编码器-解码器模型，其中编码器和解码器是序列模型，输入和输出都是序列，可能长度不同。



一种查看编码器-解码器模型的方法是将其作为称为条件生成模型的模型的特殊情况。在条件生成中,替代输入表示 ϕ ,一般条件上下文 c 影响译码器产生一个输出。当条件上下文 c 来自编码器模型时,条件生成与编码器-解码器模型相同。并非所有的条件生成模型都是编码器-解码器模型,因为条件上下文可能来自结构化源。以天气报告生成器为例。温度、湿度、风速和风向的值可以“调节”解码器,生成文本天气报告。在“模型 2:条件性姓氏生成模型”中,我们看到了一个基于国籍条件性姓氏生成的例子。图 8-2 展示了一些条件生成模型的实际示例。



在这一章中，我们深入研究了 S2S 模型，并在机器翻译任务的背景下进行了说明。考虑一个“智能”的 iOS/Android 键盘，它可以在你打字时自动将文本转换成表情符号。如果你输入 omg! 房子着火了!，你希望键盘输出类似内联输出的内容。注意，输出的长度（4 个标记）与输入的长度（6 个标记）不同。输出和输入之间的映射称为对齐，如图 8-3 所示。

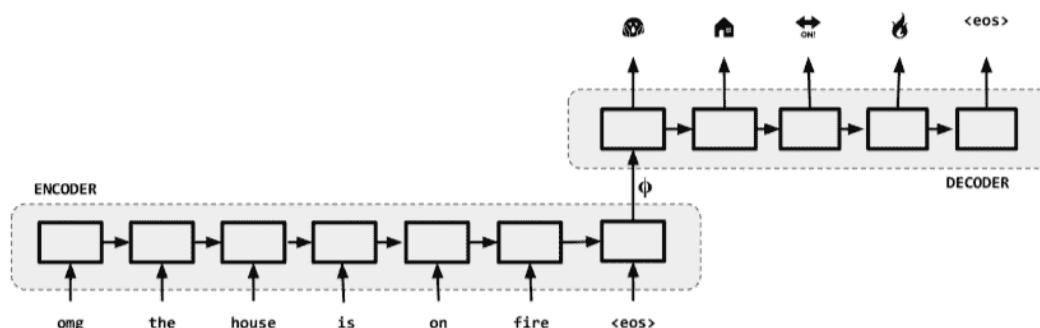


同样，在本例中，输入中的单个标记可以在输出中生成零个或多个标记。传统上，许多解决 S2S 问题的方法都是尝试使用工程和启发式重统计方法。虽然回顾这些方法超出了本章和本书的范围，但是我们建议您阅读 Koehn（2009）并参考 statmt.org 中的参考资料。

在第 6 章中，我们学习了序列模型如何将任意长度的序列编码成向量。在第 7 章中，我们看到单个向量如何使循环神经网络（RNN）有条件地产生不同的姓氏。S2S 模型是这些概念的自然延伸。

图 8-4 显示了编码器整个输入一个表示“编码”， ϕ ，条件解码器生成正确的输出。您可以使用任何 RNN 作为编码器，无论是 Elman RNN, 长短期记忆（LSTM），还是门控单元（GRU），。在接下来的两个部分中，我们将介绍现代 S2S 模型的两个重要组件。首先，我们引入了双向递归模

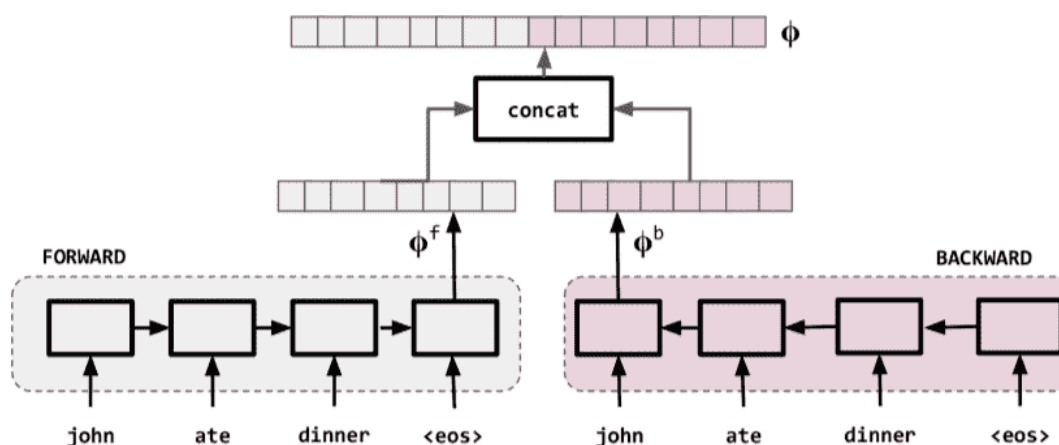
型，该模型将向前和向后传递组合在一个序列上，以创建更丰富的表示。然后，在“从序列中获取更多信息:注意力”中，我们介绍并考察了注意力机制，它在关注与任务相关的输入的不同部分时非常有用。这两个部分对于构建基于 S2S 模型的解决方案都非常重要。

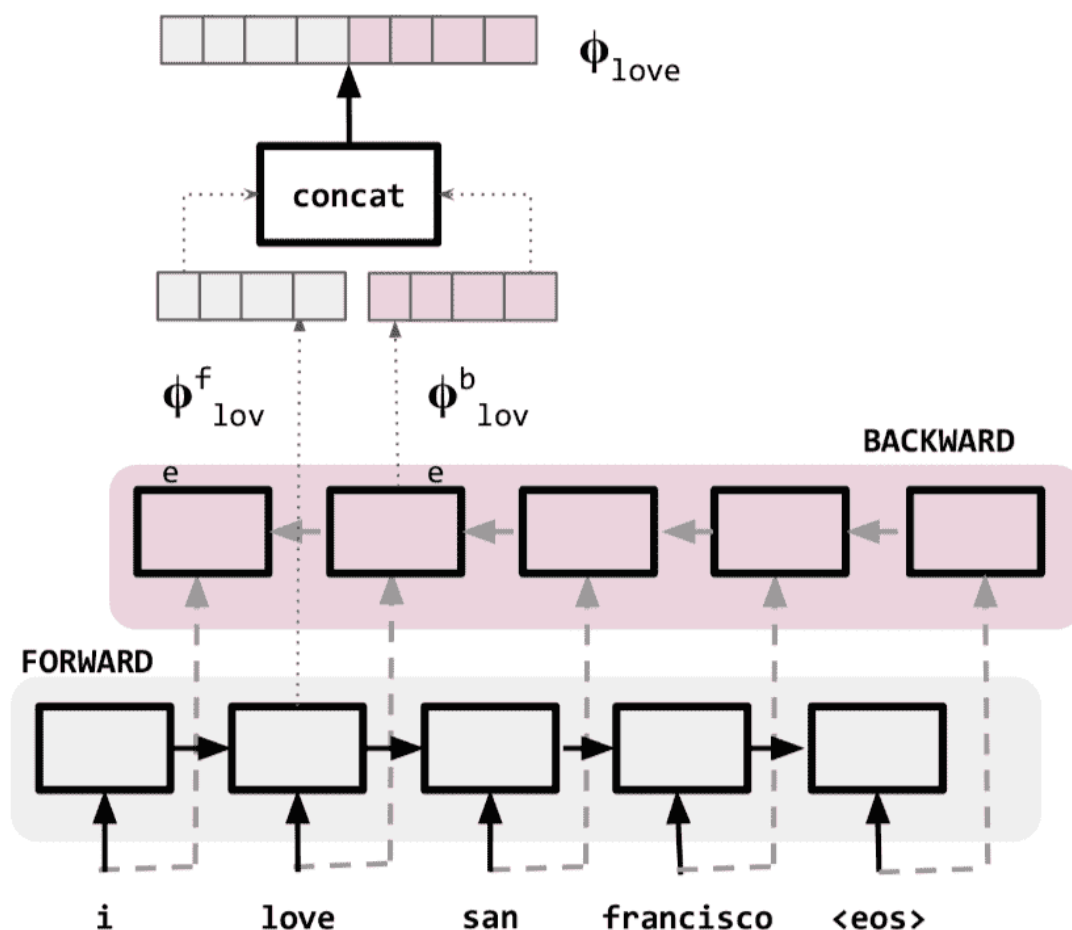


从序列中捕获更多：双向循环模型

理解递归模型的一种方法是把它看作一个将序列编码为向量的黑盒子。在建模序列时，不仅要观察过去的单词，而且还要观察将来出现的单词。考虑以下句子: The man who hunts ducks out on the weekends.。如果模型只从左到右观察，那么 duck 的表示将不同于从右到左观察单词的模型。人类一直在做这种回溯性的更新。

因此，如果把过去和未来的信息结合在一起，就能够有力地按顺序表示一个单词的意思。这就是双向递归模型的目标。递归家族的任何模型，如 Elmann RNNs 或 LSTMs 或 GRUs，都可以用于这种双向表达。与第 6 章和第 7 章中的单向模型一样，双向模型可以用于分类和序列标记设置，我们需要预测输入中每个单词的一个标签。图 8-5 和图 8-6 详细说明了这一点。在图 8-6 中， $\phi[\text{love}]$ 是表示、编码或该时刻网络的“隐藏的状态”，当输入的词是 love。当我们讨论注意力时，这种状态信息在“从一个序列中获取更多信息:注意力”中变得很重要。

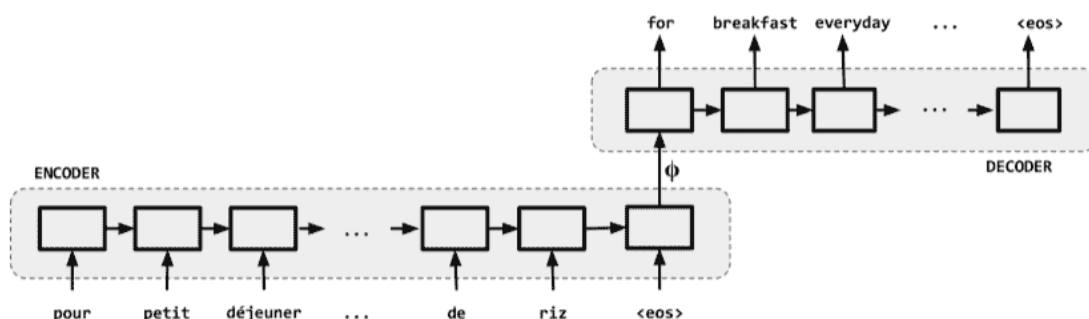




从序列中捕获更多：注意力

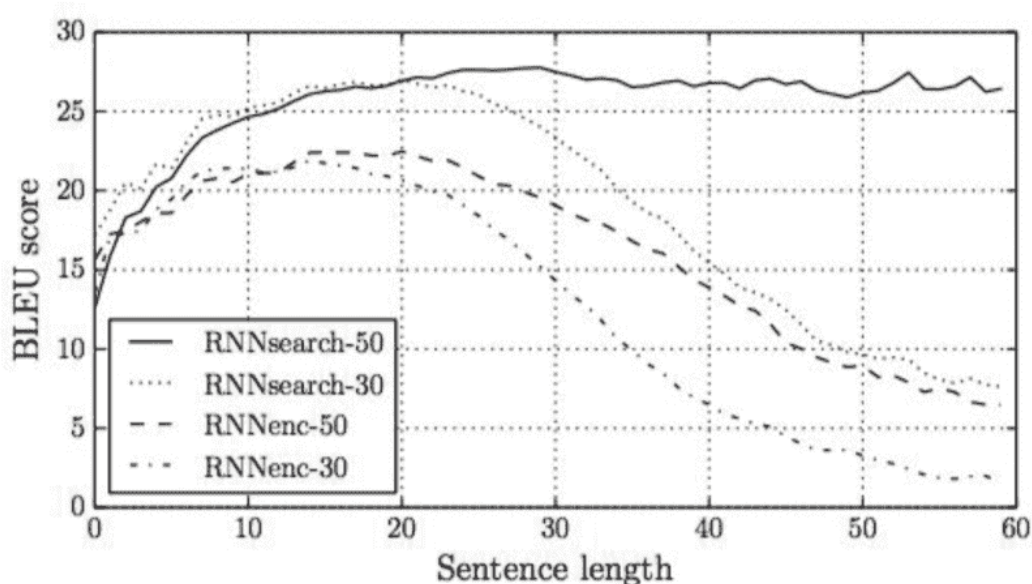
“序列到序列模型，编码器 - 解码器模型和条件生成”中引入的 S2S 模型公式的一个问题是它将整个输入句子变成单个向量（“编码”） ϕ 并使用该编码生成输出，如图 8-7 所示。虽然这可能适用于非常短的句子，但对于长句，这样的模型无法捕获整个输入中的信息；例如，见 Bengio 等。

（1994）和 Le 和 Zuidema（2016）。这是仅使用最终隐藏状态作为编码的限制。长输入的一个问题是，当长时间输入反向传播时，梯度消失，使训练变得困难。



对于曾尝试翻译的双语/多语言读者来说，这种首先编码然后解码的过程可能会有点奇怪。作为人类，我们通常不会提炼句子的含义并从意义中产生翻译。对于图 8-7 中的示例，当我们看到 `pour`，我们知道会有一个 `for`；类似地，当我们看到 `petit-déjeuner` 时，我们会想到 `breakfast`，等等。换句话说，我们的思维在产生输出时专注于输入的相关部分。这种现象称为注意力。注意已经在神经科学和其他相关领域得到了广泛的研究，这使我们尽管记忆有限，却取得了相当的成功。注意无处不在。实际上，亲爱的读者，现在正在发生这种情况。**现在每个你阅读的单词都受到注意。**即使您记忆犹新，您可能也不会读整本书。在阅读单词时，您会注意相邻的单词，可能是本节和本章的主题，等等。

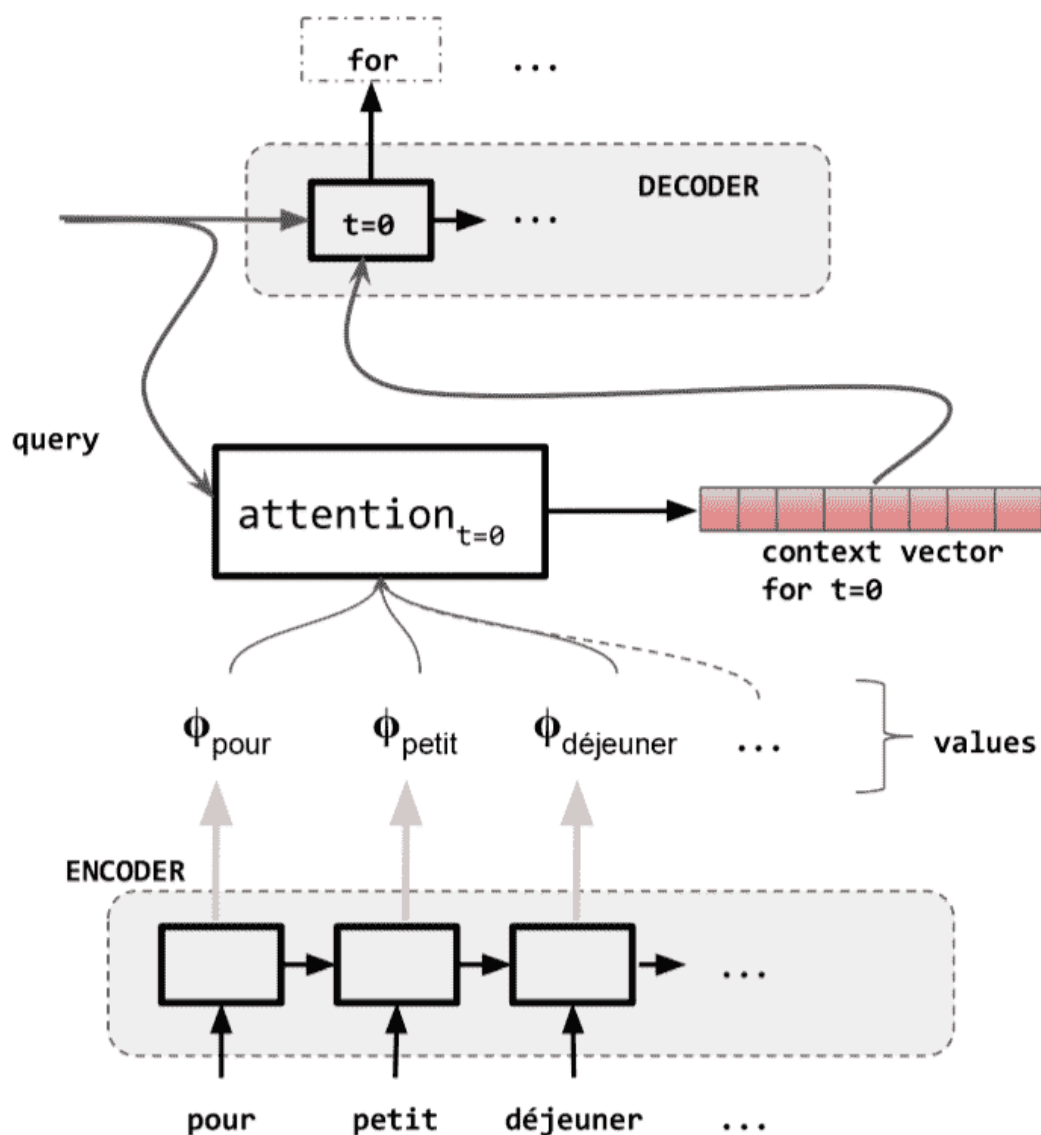
以类似的方式，我们希望序列生成模型将注意力集中到输入的不同部分，而不仅仅是整个输入的最终总结。这就是注意力机制。第一个包含自然语言处理（NLP）注意概念模型是 Bahdanau 等人（2015）的机器翻译模型。从那时起，人们提出了几种注意机制和提高注意的方法。在本节中，我们将回顾一些基本的注意机制，并介绍一些与注意相关的术语。事实证明，注意力对于改进输入和输出复杂的深度学习系统非常有用。事实上，Bahdanau 等人通过“BLEU 得分”（我们在“评估序列生成模型”中看到的）来衡量机器翻译系统的性能，当输入变长时，机器翻译系统在没有注意机制的情况下会下降，如图 8-8 所示。增加注意力可以解决问题。



深度神经网络中的注意力

注意力是一种通用的机制，可以用于本书前面讨论过的任何一种模型。但我们在这里用编码器-解码器模型来描述它，因为这些模型是注意力机制真正发挥作用的地方。考虑一个 S2S 模型。回想一下，在一个典型的 S2S 模型中，每个时间步生成一个隐藏的状态表示，表示 $\phi[w]$ ，特定于该时间步的编码器。（如图 8-6 所示。）为了引起注意，我们不仅要考虑编码器的最终隐藏状态，还要考虑每个中间步骤的隐藏状态。这些编码器隐藏状态，在某种程度上是非信息性的，称为值。在某些情况下，编码器的隐藏状态也称为键。注意力还取决于调用查询的解码器的前一个隐藏状态。图 8-9 说明了时间步骤 0 的所有这些。时间步长 $t=0$ 的查询向量是一个固定的超参数。注意

由一个向量来表示，这个向量的维数与它所关注的值的维数相同。这被称为注意力向量，或注意力权重，有时也称为对齐。注意力权重与编码器状态（“值”）相结合，生成一个有时也称为瞥见的上下文向量。这个上下文向量成为解码器的输入，而不是完整的句子编码。使用兼容性函数更新下一个时间步骤的注意力向量。相容函数的确切性质取决于所使用的注意机制。



有几种方法可以实现关注。最简单和最常用的是内容感知机制。您可以在“示例：神经机器翻译”中看到内容感知注意力。另一种流行的注意机制是位置感知注意力，它仅依赖于查询向量和密钥。注意权重通常是 0 到 1 之间的浮点值。这称为软注意。相反，可以学习二进制 0/1 向量以引起注意。这被称为硬关注。

图 8-9 中所示的注意机制取决于输入中所有时间步长的编码器状态。这也被称为全球关注。相反，对于本地注意力，您可以设计一种注意机制，该机制仅依赖于当前时间步长周围的输入窗口。

有时，特别是在机器翻译中，可以明确地提供对齐信息作为训练数据的一部分。在这种情况下，可以设计受监督的注意力来使用共同训练的单独神经网络来学习注意力功能。对于诸如文档之类的大型输入，可以设计粗粒度到细粒度的注意机制，也称为分级注意，不仅关注立即输入，而且还考虑文档的结构 - 段落，部分，章节等。Vaswani 等人对变压器网络的研究。（2017），引入多头注意，其中多个注意向量用于跟踪输入的不同区域。他们还普及了自我关注的概念，这是一种机制，通过该机制，模型可以了解输入的哪些区域相互影响。

当输入是多模式时 - 例如，图像和语音 - 可以设计多模式注意力。关于注意力的文献虽然很新，但已经非常广泛，这表明了这一主题的重要性。详细介绍它们的每一个都超出了本书的范围，我们将引导您到 Luong, Pham 和 Manning（2011）以及 Vaswani 等人。（2017）作为起点。

评估序列生成模型

当生成任务中可以看到多个有效答案时，精度，召回，准确度和 F1 等分类指标无法帮助模型 - 单个法语句子可以有多个英语翻译。序列模型根据称为参考输出的预期输出进行评估。在比较不同的模型时，我们使用分数来表明模型输出的“良好”与参考的接近程度。例如，在像机器翻译这样的任务中，如果一个模型只有一个单词关闭，我们可能不希望像另一个产生完全无法理解的答案的模型那样惩罚该模型。单个输入示例可以有多个参考输出。例如，对于特定的法语句子，可能存在多个有效的英语翻译，使用略微不同的单词。序列生成模型有两种评估：人工评估和自动评估。

人体评估涉及一个或多个人类受试者，要么对模型输出给出“竖起拇指”或“拇指向下”评级，要么进行编辑以纠正翻译。这导致了一个简单的“错误率”，它非常接近系统输出与人工任务相关的最终目标。人类评价很重要，但是很少使用，因为人类注释者往往是缓慢，昂贵和难以获得的。最后，人类也可能彼此不一致，并且，与任何其他金标准一样，人类评估与注释器间协议率配对。测量注释器间协议率也是另一个昂贵的主张。一种常见的人类评估指标是人为目标翻译错误率（HTER）。HTER 是一个加权编辑距离，由人类为了合理充分的意义和流畅性而“修复”翻译输出而进行的插入，删除和转置次数计算得出（参见图 8-10）。

Judge Sentence

You have already judged 14 of 3064 sentences, taking 86.4 seconds per sentence.

Source: les deux pays constituent plutôt un laboratoire nécessaire au fonctionnement interne de l'ue .

Reference: rather , the two countries form a laboratory needed for the internal working of the eu .

Translation	Adequacy	Fluency
both countries are rather a necessary laboratory the internal operation of the eu .	<div><div></div><div></div><div></div><div></div><div></div></div> <div>1 2 3 4 5</div>	<div><div></div><div></div><div></div><div></div><div></div></div> <div>1 2 3 4 5</div>
both countries are a necessary laboratory at internal functioning of the eu .	<div><div></div><div></div><div></div><div></div><div></div></div> <div>1 2 3 4 5</div>	<div><div></div><div></div><div></div><div></div><div></div></div> <div>1 2 3 4 5</div>
the two countries are rather a laboratory necessary for the internal workings of the eu .	<div><div></div><div></div><div></div><div></div><div></div></div> <div>1 2 3 4 5</div>	<div><div></div><div></div><div></div><div></div><div></div></div> <div>1 2 3 4 5</div>
the two countries are rather a laboratory for the internal workings of the eu .	<div><div></div><div></div><div></div><div></div><div></div></div> <div>1 2 3 4 5</div>	<div><div></div><div></div><div></div><div></div><div></div></div> <div>1 2 3 4 5</div>
the two countries are rather a necessary laboratory internal workings of the eu .	<div><div></div><div></div><div></div><div></div><div></div></div> <div>1 2 3 4 5</div>	<div><div></div><div></div><div></div><div></div><div></div></div> <div>1 2 3 4 5</div>
Annotator: Philipp Koehn Task: WMT06 French-English	<div>Annotate</div>	
Instructions	5= All Meaning 4= Most Meaning 3= Much Meaning 2= Little Meaning 1= None	5= Flawless English 4= Good English 3= Non-native English 2= Disfluent English 1= Incomprehensible

另一方面，自动评估操作简单快捷。有两种度量标准可用于自动评估生成的序列。我们再次使用机器翻译作为示例，但这些指标也适用于涉及生成序列的任何任务。这些指标包括基于 N 元组重叠的指标和困惑。基于 N 元组重叠的度量倾向于通过使用 N 元组重叠统计来计算得分来测量输出相对于参考的接近程度。BLEU，ROUGE 和 METEOR 是基于 N 元组重叠的度量的示例。其中，BLEU 经受了时间的考验，成为机器翻译文献中的衡量标准。BLEU 代表“双语评估学习”。我们跳过 BLEU 的确切表述，并建议您阅读 Papineni 等人。（2002 年）。出于实际目的，我们使用像 NLTK7 或 SacreBLEU8 这样的包来计算分数。当参考数据可用时，BLEU 本身的计算非常快速和容易。

困惑是基于信息理论的另一种自动评估指标，您可以将其应用于可以测量输出序列概率的任何情况。对于序列 x ，如果 $P(x)$ 是序列的概率，则困惑定义如下：

这为我们提供了一种比较不同序列生成模型的简单方法 - 测量保持数据集的模型的困惑度。虽然这很容易计算，但是当用于序列生成评估时，困惑会有许多问题。首先，它是一个膨胀的指标。请注意，困惑的表达式涉及取幂。因此，模型性能（可能性）的微小差异可能导致困惑的巨大差异，从而产生重大进展的错觉。其次，对困惑的改变可能不会转化为通过其他指标观察到的模型错误率的相应变化。最后，就像 BLEU 和其他基于 N 元组的指标一样，困惑的改善可能不会转化为人类判断的可察觉的改进。

在下一节中，我们将跟进机器翻译示例，并通过 PyTorch 实现将这些概念巧妙地结合在一起。

示例：神经机器翻译

在本例中，我们将介绍 S2S 模型最常用的实现：机器翻译。随着深度学习在 2010 年代早期的流行，很明显，使用嵌入词汇和 RNNs 是一种非常强大的两种语言之间的翻译方法——只要有足够的数据。引入“序列生成模型评价”中的注意机制，进一步完善了机器翻译模型。在这一部分，我们描述了一个基于 Luong, Pham, 和 Manning（2015）的实现，它简化了 S2S 模型中的注意方法。

我们首先概述数据集和神经机器翻译所需的特殊记账类型。数据集是一个平行的语料库；它由成对的英语句子和相应的法语翻译组成。因为我们正在处理两个可能不同长度的序列，所以我们需要跟踪输入序列和输出序列的最大长度和词汇。在大多数情况下，这个例子是对完整读者在前几章中所看到的内容的直接扩展。

在覆盖数据集和簿记数据结构之后，我们通过参与源序列中的不同位置来遍历模型以及它如何生成目标序列。我们模型中的编码器使用双向 GRU（bi-GRU）来计算源序列中每个位置的向量，这些向量由序列的所有部分通知。为此，我们使用 PyTorch 的 `PackedSequences` 数据结构。我们在“NMT 模型中的编码和解码”中更深入地介绍了这一点。在“从序列捕获更多：注意力”中讨论的注意机制应用于 bi-GRU 的输出并用于调节目标序列生成。我们讨论模型的结果以及在“训练常规和结果”中可以改进的方法。

机器翻译数据集

对于此示例，我们使用来自 Tatoeba Project 的英语 - 法语句子对的数据集。数据预处理首先将所有句子设为小写，并将 NLTK 的英语和法语标记符应用于每个句子对。接下来，我们应用 NLTK 的特定于语言的单词标记生成器来创建标记列表。即使我们进行了进一步的计算，我们将在下一段中描述，但这个标记列表是一个预处理的数据集。

除了刚刚描述的标准预处理之外，我们还使用指定的语法模式列表来选择数据的子集，以简化学习问题。从本质上讲，这意味着我们将数据范围缩小到只有有限范围的句法模式。反过来，这意味着在训练期间，模型将看到更少的变化，并在更短的训练时间内具有更高的性能。

Note

在构建新模型和尝试新体系结构时，您应该在建模选择和评估这些选择之间实现更快的迭代周期。

我们用来选择数据子集的句法模式是以 I am, he is, she is, they are, you are 或 we are 开头的英语句子。数据集从 135,842 个句子对减少到 13,062 个句子对，系数为 10。为了最终确定学习设置，我们将剩余的 13,062 个句子对分为 70% 训练，15% 验证和 15% 测试分裂。从刚刚列出的语法开始的每个句子的比例通过首先按句子开始分组，从这些组创建分割，然后合并每个组的分割来保持不变。

NMT 向量化流水线

对源英语和目标法语句子进行向量化需要比前面章节中看到的更复杂的管道。复杂性增加有两个原因。首先，源序列和目标序列在模型中具有不同的角色，属于不同的语言，并且以两种不同的方式进行向量化。其次，作为使用 PyTorch 的 PackedSequences 的先决条件，我们按源句的长度对每个小批量进行排序。为了准备这两个复杂性，NMTVectorizer 实例化了两个独立的 SequenceVocabulary 对象和两个最大序列长度的测量，如例 8-1 所示。

示例 8-1：构造 NMTVectorizer

```
class NMTVectorizer(object):
    """ The Vectorizer which coordinates the Vocabularies and puts them to
    use """
    def __init__(self, source_vocab, target_vocab, max_source_length,
                  max_target_length):
        """
        Args:
            source_vocab (SequenceVocabulary): maps source words to integers
            target_vocab (SequenceVocabulary): maps target words to integers
            max_source_length (int): the longest sequence in the source
            max_target_length (int): the longest sequence in the target
        """
        self.source_vocab = source_vocab
        self.target_vocab = target_vocab

        self.max_source_length = max_source_length
        self.max_target_length = max_target_length

    @classmethod
    def from_dataframe(cls, bitext_df):
        """Instantiate the vectorizer from the dataset dataframe

        Args:
            bitext_df (pandas.DataFrame): the parallel text dataset
        Returns:
            an instance of the NMTVectorizer
        """
        source_vocab = SequenceVocabulary()
        target_vocab = SequenceVocabulary()
```

```

max_source_length, max_target_length = 0, 0

for _, row in bitext_df.iterrows():
    source_tokens = row["source_language"].split(" ")
    if len(source_tokens) > max_source_length:
        max_source_length = len(source_tokens)
    for token in source_tokens:
        source_vocab.add_token(token)

    target_tokens = row["target_language"].split(" ")
    if len(target_tokens) > max_target_length:
        max_target_length = len(target_tokens)
    for token in target_tokens:
        target_vocab.add_token(token)

return cls(source_vocab, target_vocab, max_source_length,
           max_target_length)

```

复杂性的第一个增加是处理源序列和目标序列的不同方式。源序列在开始时插入 `BEGIN-OF-SEQUENCE` 进行向量化，并将 `END-OF-SEQUENCE` 标记添加到结尾。该模型使用 bi-GRU 为源句子中的每个标记创建摘要向量，并且这些摘要向量极大地受益于具有句子边界的指示。相反，目标序列被向量化为两个副本，偏移一个标记：第一个副本需要 `BEGIN-OF-SEQUENCE` 标记，第二个副本需要 `END-OF-SEQUENCE` 标记。如果您从第 7 章回忆起来，序列预测任务需要在每个时间步骤观察输入标记和输出标记。S2S 模型中的解码器正在执行此任务，但增加了编码器上下文的可用性。为了解决这种复杂性，我们制定了核心向量化方法 `_vectorize`，无论它是源索引还是目标索引都无关紧要。然后，编写两个方法来分别处理源索引和目标索引。最后，使用 `NMTVectorizer.vectorize` 方法协调这些索引集，该方法是数据集调用的方法。例 8-2 显示了代码。

```

class NMTVectorizer(object):
    """ The Vectorizer which coordinates the Vocabularies and puts them to
    use """
    def _vectorize(self, indices, vector_length=-1, mask_index=0):
        """Vectorize the provided indices

        Args:
            indices (list): a list of integers that represent a sequence
            vector_length (int): an argument for forcing the length of index
vector
            mask_index (int): the mask_index to use; almost always 0
        """
        if vector_length < 0:
            vector_length = len(indices)
        vector = np.zeros(vector_length, dtype=np.int64)
        vector[:len(indices)] = indices
        vector[len(indices):] = mask_index
        return vector

    def _get_source_indices(self, text):
        """Return the vectorized source text

        Args:

```

[illegible]

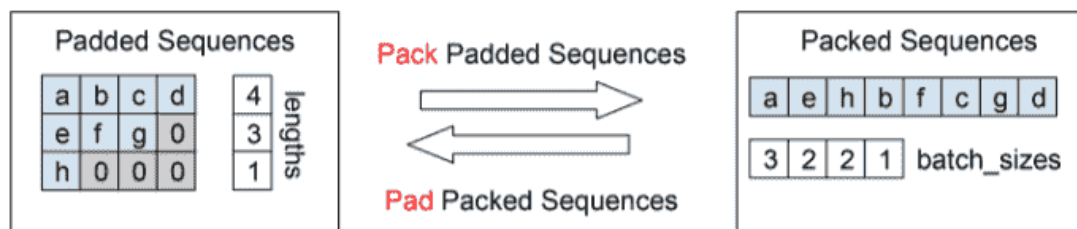
```

mask_index=self.target_vocab.mask_index)
    target_y_vector = self._vectorize(target_y_indices,
                                      vector_length=target_vector_length,

mask_index=self.target_vocab.mask_index)
    return {"source_vector": source_vector,
            "target_x_vector": target_x_vector,
            "target_y_vector": target_y_vector,
            "source_length": len(source_indices)}

```

复杂性的第二次增加再次来自源序列。为了使用 bi-GRU 对源序列进行编码，我们使用 PyTorch 的 `PackedSequences` 数据结构。通常，可变长度序列的小批量数字表示为整数矩阵中的行，其中每个序列左对齐并且零填充以适应可变长度。`PackedSequences` 数据结构通过在每个时间步，一个接一个地连接序列的数据并知道每个时间步的序列数，将可变长度序列表示为数组，如图 8-11 所示。



创建 `PackedSequence` 有两个先决条件：了解每个序列的长度，并按源序列的长度按降序对序列进行排序。为了反映这个新排序的矩阵，小批量中的剩余张量按相同的顺序排序，以便它们与源序列编码保持一致。在例 8-3 中，`generate_batches` 函数被修改为 `generate_nmt_batches` 函数。

示例 8-3：为 NMT 示例生成小批量

```

def generate_nmt_batches(dataset, batch_size, shuffle=True,
                        drop_last=True, device="cpu"):
    """A generator function which wraps the PyTorch DataLoader; NMT Version
    """
    dataloader = DataLoader(dataset=dataset, batch_size=batch_size,
                           shuffle=shuffle, drop_last=drop_last)

    for data_dict in dataloader:
        lengths = data_dict['x_source_length'].numpy()
        sorted_length_indices = lengths.argsort()[::-1].tolist()

        out_data_dict = {}
        for name, tensor in data_dict.items():
            out_data_dict[name] = data_dict[name]
        [sorted_length_indices].to(device)
        yield out_data_dict

```

NMT 模型的编码和解码

在这个例子中，我们从源序列开始 - 一个英语句子 - 我们生成一个目标序列 - 相应的法语翻译。标准方法是使用“序列到序列模型，编码器 - 解码器模型和条件生成”中描述的编码器 - 解码器模型。在示例 8-4 和示例 8-5 中呈现的模型中，编码器首先将每个源序列映射到具有 bi-GRU 的向量状态序列（参见“从序列中捕获更多：双向递归模型”）。然后，解码器以解码器的隐藏状态作为其初始隐藏状态开始，并使用注意机制（参见“从序列中捕获更多：注意”）来选择源序列中的不同信息以生成输出序列。在本节的其余部分，我们将更详细地解释此过程。

示例 8-4: NMTModel 在单个前向方法中封装和协调了编码器和解码器

```
class NMTModel(nn.Module):
    """ A Neural Machine Translation Model """
    def __init__(self, source_vocab_size, source_embedding_size,
                  target_vocab_size, target_embedding_size, encoding_size,
                  target_bos_index):
        """
        Args:
            source_vocab_size (int): number of unique words in source language
            source_embedding_size (int): size of the source embedding vectors
            target_vocab_size (int): number of unique words in target language
            target_embedding_size (int): size of the target embedding vectors
            encoding_size (int): the size of the encoder RNN.
            target_bos_index (int): index for BEGIN-OF-SEQUENCE token
        """
        super(NMTModel, self).__init__()
        self.encoder = NMTEncoder(num_embeddings=source_vocab_size,
                                   embedding_size=source_embedding_size,
                                   rnn_hidden_size=encoding_size)
        decoding_size = encoding_size * 2
        self.decoder = NMTDecoder(num_embeddings=target_vocab_size,
                                   embedding_size=target_embedding_size,
                                   rnn_hidden_size=decoding_size,
                                   bos_index=target_bos_index)

    def forward(self, x_source, x_source_lengths, target_sequence):
        """The forward pass of the model

        Args:
            x_source (torch.Tensor): the source text data tensor.
                x_source.shape should be (batch, vectorizer.max_source_length)
            x_source_lengths (torch.Tensor): the length of the sequences in
            x_source
            target_sequence (torch.Tensor): the target text data tensor
        Returns:
            decoded_states (torch.Tensor): prediction vectors at each output
            step
        """
        encoder_state, final_hidden_states = self.encoder(x_source,
                                                            x_source_lengths)
        decoded_states = self.decoder(encoder_state=encoder_state,
                                       initial_hidden_state=final_hidden_states,
```

```

target_sequence=target_sequence)

return decoded_states

```

编码器

示例 8-5: 编码器嵌入了源单词和使用 bi-GPU 提取的特征

```

class NMTEncoder(nn.Module):
    def __init__(self, num_embeddings, embedding_size, rnn_hidden_size):
        """
        Args:
            num_embeddings (int): size of source vocabulary
            embedding_size (int): size of the embedding vectors
            rnn_hidden_size (int): size of the RNN hidden state vectors
        """
        super(NMTEncoder, self).__init__()

        self.source_embedding = nn.Embedding(num_embeddings, embedding_size,
                                              padding_idx=0)

        self.birnn = nn.GRU(embedding_size, rnn_hidden_size,
                             bidirectional=True,
                             batch_first=True)

    def forward(self, x_source, x_lengths):
        """The forward pass of the model

        Args:
            x_source (torch.Tensor): the input data tensor.
            x_source.shape is (batch, seq_size)
            x_lengths (torch.Tensor): vector of lengths for each item in the
batch
        Returns:
            a tuple: x_unpacked (torch.Tensor), x_birnn_h (torch.Tensor)
            x_unpacked.shape = (batch, seq_size, rnn_hidden_size * 2)
            x_birnn_h.shape = (batch, rnn_hidden_size * 2)
        """
        x_embedded = self.source_embedding(x_source)
        # create PackedSequence; x_packed.data.shape=(number_items,
embedding_size)
        x_lengths = x_lengths.detach().cpu().numpy()
        x_packed = pack_padded_sequence(x_embedded, x_lengths,
batch_first=True)

        # x_birnn_h.shape = (num_rnn, batch_size, feature_size)
        x_birnn_out, x_birnn_h = self.birnn(x_packed)
        # permute to (batch_size, num_rnn, feature_size)
        x_birnn_h = x_birnn_h.permute(1, 0, 2)

        # flatten features; reshape to (batch_size, num_rnn * feature_size)
        # (recall: -1 takes the remaining positions,
        # flattening the two RNN hidden vectors into 1)
        x_birnn_h = x_birnn_h.contiguous().view(x_birnn_h.size(0), -1)

```

```
x_unpacked, _ = pad_packed_sequence(x_birnn_out, batch_first=True)
return x_unpacked, x_birnn_h
```

通常，编码器将整数序列作为输入，并为每个位置创建特征向量。在该示例中，编码器的输出是这些向量以及用于制作特征向量的 bi-GRU 的最终隐藏状态。该隐藏状态用于在下一节中初始化解码器的隐藏状态。

深入了解编码器，我们首先使用嵌入层嵌入输入序列。通常，只需在嵌入层上设置 `padding_idx` 标志，我们就可以使模型处理可变长度序列，因为任何等于 `padding_idx` 的位置都会被赋予零值向量，该向量在优化期间不会更新。回想一下，这被称为面具。然而，在这种编码器 - 解码器模型中，掩蔽位置需要以不同方式处理，因为我们使用 bi-GRU 来编码源序列。主要原因是后向分量可能受到屏蔽位置的影响，其因子与在序列上开始之前遇到的屏蔽位置的数量成比例。

为了处理 bi-GRU 中可变长度序列的掩码位置，我们使用 PyTorch 的 `PackedSequence` 数据结构。`PackedSequences` 源自 CUDA 如何允许以批量格式处理可变长度序列。如果满足两个条件，则可以将任何零填充序列（例如示例 8-6 中所示的编码器中的嵌入源序列）转换为 `PackedSequence`：提供每个序列的长度，并根据以下顺序对小批量进行排序。这些序列的长度。这在图 8-11 中以可视方式显示，因为它是一个复杂的主题，我们在例 8-6 及其输出中再次演示它。

示例 8-6： `packed_padded_sequences` 和 `pad_packed_sequences` 的简单演示

```
Input[0]
abcd_padded = torch.tensor([1, 2, 3, 4], dtype=torch.float32)
efg_padded = torch.tensor([5, 6, 7, 0], dtype=torch.float32)
h_padded = torch.tensor([8, 0, 0, 0], dtype=torch.float32)

padded_tensor = torch.stack([abcd_padded, efg_padded, h_padded])

describe(padded_tensor)
Output[0]
Type: torch.FloatTensor
Shape/size: torch.Size([3, 4])
Values:
tensor([[ 1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  0.],
        [ 8.,  0.,  0.,  0.]])
Input[1]
lengths = [4, 3, 1]
packed_tensor = pack_padded_sequence(padded_tensor, lengths,
                                     batch_first=True)

packed_tensor
Output[1]
PackedSequence(data=tensor([ 1.,  5.,  8.,  2.,  6.,  3.,  7.,  4.]),
               batch_sizes=tensor([ 3,  2,  2,  1]))
Input[2]
unpacked_tensor, unpacked_lengths = \
    pad_packed_sequence(packed_tensor, batch_first=True)

describe(unpacked_tensor)
describe(unpacked_lengths)
```

```

Output[2]
Type: torch.FloatTensor
Shape/size: torch.Size([3, 4])
Values:
tensor([[ 1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  0.],
        [ 8.,  0.,  0.,  0.]])
Type: torch.LongTensor
Shape/size: torch.Size([3])
Values:
tensor([ 4,  3,  1])

```

我们在生成每个小批量时处理排序，如上一节所述。然后，如例 8-7 所示，通过传递嵌入的序列，序列的长度和表示第一个维度是批量维度的布尔标志来激发 PyTorch 的 `pack_padded_sequence` 函数。此函数的输出是 `PackedSequence`。将得到的 `PackedSequence` 输入到 bi-GRU 中以为下游解码器创建状态向量。使用另一个布尔标志将 bi-GRU 的输出解压缩为完整张量，指示批量在第一维上。解包操作，如图 8-11 所示，将每个屏蔽位置 15 设置为零值向量，保留下游计算的完整性。

示例 8-7: NMT 解码器从编码的源句子中构造目标句子

```

class NMTDecoder(nn.Module):
    def __init__(self, num_embeddings, embedding_size, rnn_hidden_size,
bos_index):
        """
        Args:
            num_embeddings (int): number of embeddings is also the number of
                unique words in target vocabulary
            embedding_size (int): the embedding vector size
            rnn_hidden_size (int): size of the hidden rnn state
            bos_index(int): begin-of-sequence index
        """
        super(NMTDecoder, self).__init__()
        self._rnn_hidden_size = rnn_hidden_size
        self.target_embedding = nn.Embedding(num_embeddings=num_embeddings,
                                             embedding_dim=embedding_size,
                                             padding_idx=0)
        self.gru_cell = nn.GRUCell(embedding_size + rnn_hidden_size,
                                    rnn_hidden_size)
        self.hidden_map = nn.Linear(rnn_hidden_size, rnn_hidden_size)
        self.classifier = nn.Linear(rnn_hidden_size * 2, num_embeddings)
        self.bos_index = bos_index

    def _init_indices(self, batch_size):
        """ return the BEGIN-OF-SEQUENCE index vector """
        return torch.ones(batch_size, dtype=torch.int64) * self.bos_index

    def _init_context_vectors(self, batch_size):
        """ return a zeros vector for initializing the context """
        return torch.zeros(batch_size, self._rnn_hidden_size)

    def forward(self, encoder_state, initial_hidden_state, target_sequence):
        """The forward pass of the model

```



```

    Args:
        encoder_state (torch.Tensor): the output of the NMTEncoder
        initial_hidden_state (torch.Tensor): The last hidden state in
the NMTEncoder
        target_sequence (torch.Tensor): the target text data tensor
        sample_probability (float): the schedule sampling parameter
        probability of using model's predictions at each decoder step
    Returns:
        output_vectors (torch.Tensor): prediction vectors at each output
step
    """
    # We are making an assumption there: The batch is on first
    # The input is (Batch, Seq)
    # We want to iterate over sequence so we permute it to (S, B)
    target_sequence = target_sequence.permute(1, 0)

    # use the provided encoder hidden state as the initial hidden state
    h_t = self.hidden_map(initial_hidden_state)

    batch_size = encoder_state.size(0)
    # initialize context vectors to zeros
    context_vectors = self._init_context_vectors(batch_size)
    # initialize first y_t word as BOS
    y_t_index = self._init_indices(batch_size)

    h_t = h_t.to(encoder_state.device)
    y_t_index = y_t_index.to(encoder_state.device)
    context_vectors = context_vectors.to(encoder_state.device)

    output_vectors = []
    # All cached tensors are moved from the GPU and stored for analysis
    self._cached_p_attn = []
    self._cached_ht = []
    self._cached_decoder_state = encoder_state.cpu().detach().numpy()

    output_sequence_size = target_sequence.size(0)
    for i in range(output_sequence_size):

        # Step 1: Embed word and concat with previous context
        y_input_vector = self.target_embedding(target_sequence[i])
        rnn_input = torch.cat([y_input_vector, context_vectors], dim=1)

        # Step 2: Make a GRU step, getting a new hidden vector
        h_t = self.gru_cell(rnn_input, h_t)
        self._cached_ht.append(h_t.cpu().data.numpy())

        # Step 3: Use the current hidden to attend to the encoder state
        context_vectors, p_attn, _ = \
            verbose_attention(encoder_state_vectors=encoder_state,
                            query_vector=h_t)

        # auxiliary: cache the attention probabilities for visualization
        self._cached_p_attn.append(p_attn.cpu().detach().numpy())

        # Step 4: Use the current hidden and context vectors
        # to make a prediction to the next word
        prediction_vector = torch.cat((context_vectors, h_t), dim=1)

```

```

score_for_y_t_index = self.classifier(prediction_vector)

# auxiliary: collect the prediction scores
output_vectors.append(score_for_y_t_index)

```

在编码器利用其 bi-GRU 和打包 - 解包协调创建状态向量之后，解码器在时间步骤上迭代以生成输出序列。在功能上，这个循环应该看起来与第 7 章中的生成循环非常相似，但是有一些差异明显是 Luong 等人的注意方式的方法选择。首先，在每个时间步骤提供目标序列作为观察。通过使用 GRUCell 计算隐藏状态。通过将线性层应用于编码器 bi-GRU 的级联最终隐藏状态来计算初始隐藏状态。在每个时间步骤处对解码器 GRU 的输入是嵌入输入标记和最后时间步骤的上下文的级联向量。向量。上下文向量旨在捕获对该时间步骤有用的信息，并用于调节模型的输出。对于第一个步骤，上下文向量全部为 0（零）以表示无上下文并且在数学上仅允许输入对 GRU 计算做出贡献。

使用新的隐藏状态作为查询向量，使用当前时间步骤的关注机制创建一组新的上下文向量。这些上下文向量与隐藏状态连接以创建表示该时间步长处的解码信息的向量。该解码信息状态向量用在分类器（在这种情况下，简单的线性层）中以创建预测向量 `score_for_y_t_index`。这些预测向量可以使用 softmax 函数转换为输出词汇表上的概率分布，或者它们可以与交叉熵损失一起使用以优化地面实况目标。在我们转向如何在训练例程中使用预测向量之前，我们首先检查注意力计算本身。

注意力的详细解释

了解注意机制在此示例中的工作方式非常重要。回想一下前面的部分，可以使用查询，键和值来描述注意机制。分数函数将查询向量和键向量作为输入，以计算在值向量中选择的一组权重。在这个例子中，我们使用点积评分函数，但它不是唯一的。在这个例子中，解码器的隐藏状态被用作查询向量，编码器状态向量集是键和值向量。

解码器隐藏状态与编码器状态中的向量的点积为编码序列中的每个项创建标量。在使用 softmax 函数时，这些标量变为编码器状态中的向量的概率分布。这些概率用于在将编码器状态向量加在一起之前对其进行加权，以产生每个批次项目的单个向量。总而言之，允许解码器隐藏状态在每个时间步骤优先加权编码器状态。这就像一个聚光灯，使模型能够学习如何突出显示生成输出序列所需的信息。我们在例 8-8 中演示了这种版本的注意机制。第一个尝试详细说明操作。此外，它使用视图操作插入大小为 1 的维度，以便可以针对另一个张量广播张量。在

`terse_attention` 版本中，视图操作被更常用的练习替换，取消压缩。此外，不是将元素和求和相乘，而是使用更有效的 `matmul` 运算。

示例 8-8：注意力更加显式地执行逐元素乘法和求和

```

def verbose_attention(encoder_state_vectors, query_vector):
    """
    encoder_state_vectors: 3dim tensor from bi-GRU in encoder
    query_vector: hidden state in decoder GRU
    """

```

```

batch_size, num_vectors, vector_size = encoder_state_vectors.size()
vector_scores = \
    torch.sum(encoder_state_vectors * query_vector.view(batch_size, 1,
                                                         vector_size),
              dim=2)
vector_probabilities = F.softmax(vector_scores, dim=1)
weighted_vectors = \
    encoder_state_vectors * vector_probabilities.view(batch_size,
                                                         num_vectors, 1)

context_vectors = torch.sum(weighted_vectors, dim=1)
return context_vectors, vector_probabilities

def terse_attention(encoder_state_vectors, query_vector):
    """
    encoder_state_vectors: 3dim tensor from bi-GRU in encoder
    query_vector: hidden state
    """
    vector_scores = torch.matmul(encoder_state_vectors,
                                  query_vector.unsqueeze(dim=2)).squeeze()
    vector_probabilities = F.softmax(vector_scores, dim=-1)
    context_vectors = torch.matmul(encoder_state_vectors.transpose(-2, -1),
                                    vector_probabilities.unsqueeze(dim=2)).squeeze()
    return context_vectors, vector_probabilities

```

LEARNING TO SEARCH AND SCHEDULED SAMPLING

当前编写的方式，模型假定提供了目标序列，并将在解码器的每个时间步骤用作输入。在测试时，违反了这个假设，因为模型不能作弊并且知道它试图生成的序列。为了适应这一事实，一种技术是允许模型在训练期间使用自己的预测。这是一种在文献中探索为“学习搜索”和“预定抽样”的技术。理解这种技术的一种直观方法是将预测问题视为搜索问题。在每个时间步，模型有许多路径可供选择（选择的数量是目标词汇的大小），数据是正确路径的观察。在测试时，模型最终被允许“离开路径”，因为没有提供正确的路径，它应该从中计算概率分布。因此，让模型采样自己的路径的技术提供了一种方法，在该方法中，当模型偏离数据集中的目标序列时，可以优化模型以获得更好的概率分布。

代码有三个主要修改，以使模型在训练期间采样自己的预测。首先，初始索引更明确地作为 `BEGIN-OF-SEQUENCE` 标记索引。其次，为生成循环中的每个步骤绘制随机样本，如果随机样本小于样本概率，则在该迭代期间使用模型的预测。最后，第三个实际采样本身在条件 `if` 下使用 `use_sample`。在示例 8-9 中，注释行显示了如何使用最大预测，而未注释行显示了如何以与其概率成比例的速率实际采样索引。

示例 8-9：在前向过程中构建的带有采样过程的解码器（粗体）

```

class NMTDecoder(nn.Module):
    def __init__(self, num_embeddings, embedding_size, rnn_size, bos_index):
        super(NMTDecoder, self).__init__()
        # ... other init code here ...

        # arbitrarily set; any small constant will be fine
        self._sampling_temperature = 3

```

```

def forward(self, encoder_state, initial_hidden_state, target_sequence,
            sample_probability=0.0):
    if target_sequence is None:
        sample_probability = 1.0
    else:
        # We are making an assumption there: The batch is on first
        # The input is (Batch, Seq)
        # We want to iterate over sequence so we permute it to (S, B)
        target_sequence = target_sequence.permute(1, 0)
        output_sequence_size = target_sequence.size(0)

    # ... nothing changes from the other implementation

    output_sequence_size = target_sequence.size(0)
    for i in range(output_sequence_size):
        # new: a helper boolean and the teacher y_t_index
        use_sample = np.random.random() < sample_probability
        if not use_sample:
            y_t_index = target_sequence[i]

        # Step 1: Embed word and concat with previous context
        # ... code omitted for space
        # Step 2: Make a GRU step, getting a new hidden vector
        # ... code omitted for space
        # Step 3: Use the current hidden to attend to the encoder state
        # ... code omitted for space
        # Step 4: Use the current hidden and context vectors
        #             to make a prediction to the next word
        prediction_vector = torch.cat((context_vectors, h_t), dim=1)
        score_for_y_t_index = self.classifier(prediction_vector)
        # new: sampling if boolean is true.
        if use_sample:
            # sampling temperature forces a peakier distribution
            p_y_t_index = F.softmax(score_for_y_t_index *
                                    self._sampling_temperature, dim=1)
            # method 1: choose most likely word
            # _, y_t_index = torch.max(p_y_t_index, 1)
            # method 2: sample from the distribution
            y_t_index = torch.multinomial(p_y_t_index, 1).squeeze()

        # auxiliary: collect the prediction scores
        output_vectors.append(score_for_y_t_index)

    output_vectors = torch.stack(output_vectors).permute(1, 0, 2)

    return output_vectors

```

训练例程和结果

此示例的训练例程几乎与前面章节中介绍的训练例程相同。对于固定数量的历元，我们在称为小批量的块中迭代数据集。然而，这里的每个小批量由四个张量组成：源序列的整数矩阵，目标序列的两个整数矩阵，以及源序列长度的整数向量。两个靶序列矩阵是靶序列偏移 1 并用 BEGIN-OF-SEQUENCE 标记填充以充当靶序列观察，或 END-OF-SEQUENCE 标记充当靶序列预测标记。该

模型将源序列和目标序列观察作为输入，以产生目标序列预测。在损失函数中使用目标序列预测标签来计算交叉熵损失，然后将其反向传播到每个模型参数以使其知道其梯度。然后调用优化器并将每个模型参数更新一些与梯度成比例的量。

除了数据集的训练部分上的循环之外，验证部分上还有一个循环。验证分数用作模型改进的偏差较小的度量。该过程与训练例程相同，只是模型处于求值模式并且模型未相对于验证数据更新。

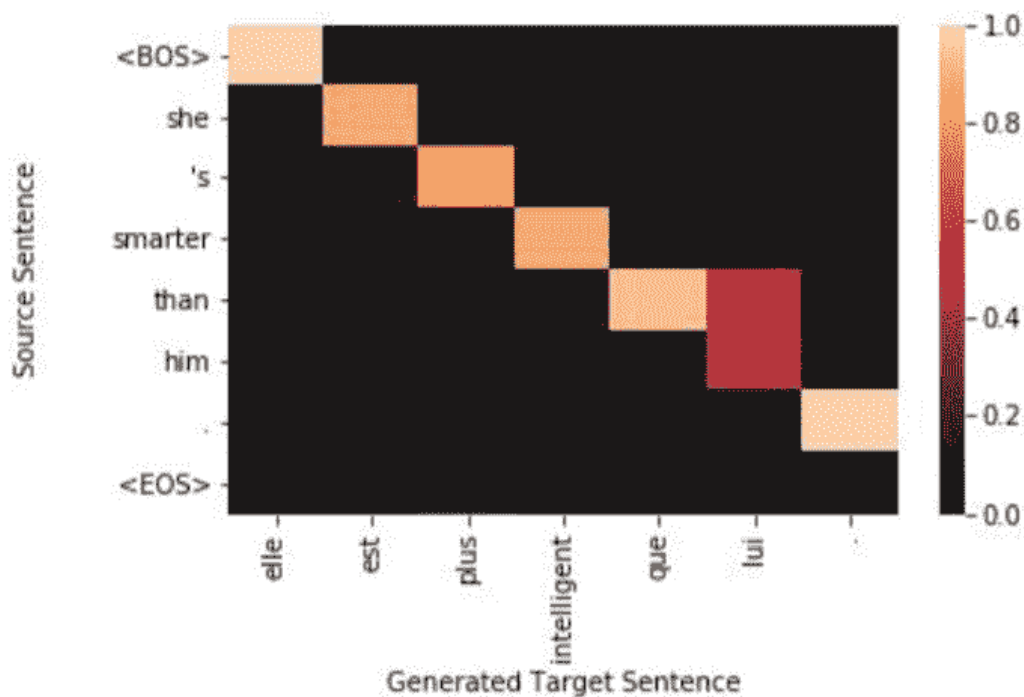
在训练模型之后，测量性能成为一个重要而重要的问题。在“评估序列生成模型”中描述了几个序列生成评估度量，但是诸如测量预测句子和参考语句之间的 N 元组的重叠的 BLEU 的度量已经成为机器翻译领域的标准。聚合结果的评估代码已被省略，但您可以在本书的随附代码库中找到它。在代码中，模型的输出与源句子，参考目标语句和注意概率矩阵聚合在一起。例。最后，为每对源和生成的句子计算 BLEU-4。

为了定性评估模型的工作情况，我们将注意概率矩阵可视化为源和生成文本之间的对齐。然而，值得注意的是，最近的研究表明，基于注意力的对齐与经典机器翻译中的对齐并不完全相同。基于注意力的对齐分数可以指示解码器的有用信息，例如在生成输出动词时参与句子的主语 (Koehn 和 Knowles, 2017)，而不是单词和短语之间的对齐指示翻译同义词。

我们比较了我们模型的两个版本，它们与目标句子的交互方式不同。第一个版本使用提供的目标序列作为解码器中每个时间步的输入。第二个版本使用预定采样，以允许模型将其自己的预测视为解码器中的输入。这有利于强制模型优化其自身的错误。表 8-1 显示了 BLEU 分数。重要的是要记住，为了便于训练，我们选择了标准 NMT 任务的简化版本，这就是为什么分数似乎高于您在研究文献中通常会发现的分数。虽然第二个模型，即具有预定采样的模型，具有更高的 BLEU 分数，但是得分相当接近。但这些得分究竟意味着什么呢？为了研究这个问题，我们需要定性地检查模型。

Model Name	Bleu Score
Model without Scheduled Sampling	46.8
Model with Scheduled Sampling	48.1

对于我们更深入的检查，我们绘制注意力分数，以查看它们是否在源句和目标句之间提供任何类型的对齐信息。我们发现在这次检查中两个模型之间形成了鲜明的对比。图 8-12 显示了具有预定采样的模型的每个解码器时间步长的注意概率分布。在该模型中，注意权重对于从数据集的验证部分采样的句子排列得相当好。



总结

本章重点介绍了在所谓的条件生成模型的条件上下文中生成序列输出。当条件上下文本身来自另一个序列时，我们将其称为序列到序列或 S2S 模型。我们还讨论了 S2S 模型如何成为编码器 - 解码器模型的特例。为了充分利用序列，我们讨论了第 6 章和第 7 章中讨论的序列模型的结构变体，特别是双向模型。我们还学习了如何结合注意机制来有效捕获更长距离的背景。最后，我们讨论了如何评估序列到序列模型，并使用端到端机器翻译示例进行演示。到目前为止，我们已将本书的每一章专门用于特定的网络架构。在下一章中，我们将前面的所有章节结合在一起，并查看如何使用各种模型体系结构的综合构建许多真实系统的示例。

参考

1. Yoshua Bengio, Patrice Simard, and Paolo Frasconi. (1994). "Learning long-term dependencies with gradient descent is difficult." IEEE transactions on neural networks.
2. Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. (2002) "BLEU: a method for automatic evaluation of machine translation." In Proceedings ACL..
3. Hal Daumé III, John Langford, Daniel Marcu. (2009). "Search-based Structured Prediction." In Machine Learning Journal.
3. Samy Bengio, Oriol Vinyals, Navdeep Jaitly, Noam Shazeer. "Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks." In Proceedings of NIPS 2015.

4. Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. (2015). "Effective Approaches to Attention-based Neural Machine Translation." In Proceedings of EMNLP.
5. Phong Le and Willem Zuidema. (2016). "Quantifying the Vanishing Gradient and Long Distance Dependency Problem in Recursive Neural Networks and Recursive LSTMs." Proceedings of the 1st Workshop on Representation Learning for NLP.
6. Philipp Koehn, Rebecca Knowles. (2017). "Six Challenges for Neural Machine Translation." In Proceedings of the First Workshop on Neural Machine Translation.
7. Graham Neubig. (2017). "Neural Machine Translation and Sequence-to-Sequence Models: A Tutorial." arXiv:1703.01619.
8. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. (2017). "Attention is all you need." In Proceedings of NIPS.

九、经典, 前沿和后续步骤

本文标题: [Natural-Language-Processing-with-PyTorch \(九\)](#)

文章作者: [Yif Du](#)

发布时间: 2018 年 12 月 28 日 - 11:12

最后更新: 2018 年 12 月 28 日 - 11:12

原始链接: <http://yifdu.github.io/2018/12/28/Natural-Language-Processing-with-PyTorch> (九) /

许可协议: [署名-非商业性使用-禁止演绎 4.0 国际](#) 转载请保留原文链接及作者。

在本章中, 我们将从整本书的角度回顾前面的章节, 并了解本书中讨论的看似独立的主题是如何相互依赖的, 以及研究人员如何将这些想法混合和匹配以解决手头的问题。我们还总结了自然语言处理 (NLP) 中的一些经典主题, 我们无法在这些封面之间进行深入讨论。最后, 我们指出了该领域的前沿, 截至 2018 年。在经验丰富的 NLP 和深度学习等快速发展的领域, 我们必须学习新的想法并使自己保持最新状态。我们致力于学习如何在 NLP 中学习新主题。

我们到目前为止学习了什么?

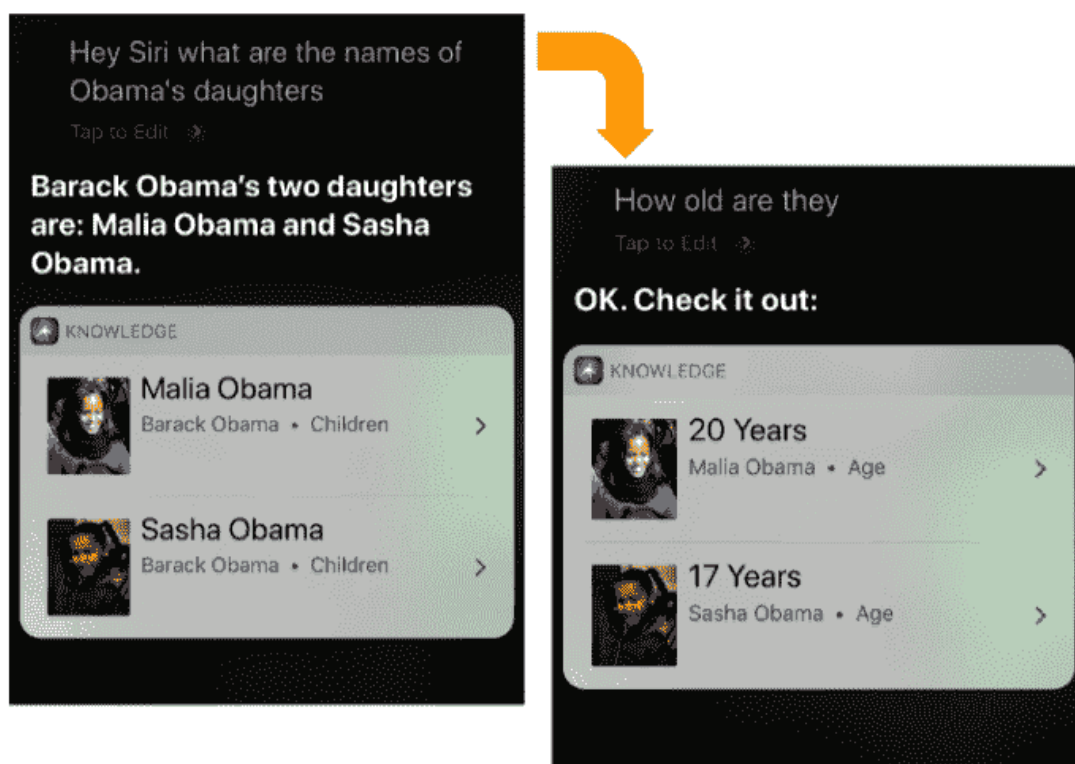
我们从监督学习范式开始, 以及我们如何使用计算图抽象将复杂的想法编码为可以通过反向传播进行训练的模型。PyTorch 是我们选择的计算框架。编写 NLP 书籍存在风险, 该书使用深度学习将文本输入视为要输入黑盒子的“数据”。在第 2 章中, 我们介绍了 NLP 和语言学的一些基本概念, 为本书的其余部分奠定了基础。激活函数, 损失函数和基于梯度的监督学习优化以及“训练 - 评估循环”等基本概念在本章的其他章节中变得非常方便。我们研究了前馈网络的两个例子 - 多层感知器 (MLP) 和卷积网络。我们看到了 L1, L2 规范和丢包等正规化机制, 使网络更加健壮。MLP 能够在其隐藏层中捕获类似 N 元组的关系, 但是它们效率低下。另一方面, 卷积网络使用称为“参数绑定”或“参数共享”的思想以计算有效的方式学习这个子结构。在第 6 章中, 我们看到了循环网络如何能够捕获远程依赖性跨越时间也只有很少的参数。你可以说卷积网络正在跨越空间绑定参数, 而循环网络正在跨时间绑定参数。我们看到了三种复发网络的变体, 从 Elman 循环神经网络 (RNN) 和门控变体如长短期记忆 (LSTM) 和门控复发单位 (GRU) 开始。我们还了解了如何在预测或序列标记设置中使用循环网络, 其中在输入的每个时间步骤预测输出。最后, 我们介绍了一类称为编码器 - 解码器模型的模型, 并研究了序列到序列模型作为解决条件生成问题 (如机器翻译) 的示例。我们在 PyTorch 中完成了许多这些主题的端到端示例。

NLP 中的永恒主题

NLP 比单本书范围内的内容还多，本书也不例外。在第 2 章中，我们确定了 NLP 中的一些核心术语和任务。我们在其余章节中介绍了许多 NLP 任务，但是在这里我们简要提到一些我们无法部分或全部涵盖的重要主题，因为我们的范围仅限于初始说明书。

对话和交互系统

计算机和人类之间的无缝对话被认为是计算的圣杯，并激发了图灵测验和勒伯纳奖。自人工智能（人工智能）早期以来，NLP 一直与对话系统相关联，并通过虚构系统在流行文化中普及，如星际迷航中的 USS Enterprise 上的主计算机和电影 2001：A Space Odyssey 中的 HAL 9000。1 对话和更广泛的交互系统设计领域是一个肥沃的研究领域，亚马逊的 Alexa，Apple 的 Siri 和 Google 的助手等近期产品取得了成功。对话系统可以是开放域（问我什么）或封闭域（例如，航班预订，汽车导航）。该领域的一些重要研究课题包括我们如何对对话行为进行建模，对话背景（见图 9-1）和对话状态？我们如何建立多模式对话系统（例如，语音和视觉或文本和视觉输入）？系统如何识别用户意图？我们如何为用户的偏好建模并生成针对用户量身定制的响应？如何回应更人性化的声音？例如，最近的生产对话系统已经开始将诸如“嗯”和“呃”之类的不流畅结合到响应中，以使系统看起来不那么机器人。



话语

话语涉及理解文本文档的部分整体性质。例如，话语解析的任务涉及理解两个句子在上下文中如何彼此相关。表 9-1 给出了 Penn Discourse Treebank (PDTB) 的一些例子来说明这项任务。

Table 9-1. Examples from the CoNLL 2015 Shallow Discourse Processing task

Example	Discourse relation
GM officials want to get their strategy to reduce capacity and the workforce in place before those talks begin .	Temporal.Asynchronous.Precedence
But that ghost wouldn't settle for words, he wanted money and people—lots. So Mr. Carter formed three new Army divisions and gave them to a new bureaucracy in Tampa called the Rapid Deployment Force .	Contingency.Cause.Result
<i>The Arabs had merely oil.</i> Implicit=while These farmers may have a grip on the world's very heart	Comparison.Contrast

理解话语还涉及解决其他问题，如回指解析和转喻检测。在 Anaphora Resolution 中，我们希望将代词的出现解析为它们所引用的实体。如图 9-2 所示，这可能成为一个复杂的问题。

- (a) The dog chewed the bone. It was delicious.
- (b) The dog chewed the bone. It was a hot day.
- (c) Nia drank a tall glass of beer. It was chipped.
- (d) Nia drank a tall glass of beer. It was bubbly.

对象可以是转喻，如下例所示： Beijing imposed trade tariffs in response to tariffs on Chinese goods.

北京在这里指的不是中国政府的所在地。有时，成功解决所指对象可能需要使用知识库。

信息提取和文本挖掘

该行业中遇到的常见问题类别之一涉及信息提取。我们如何从文本中提取实体（人名，产品名称等），事件和关系？我们如何将文本中的实体提及映射到知识库中的条目（又称实体发现，实体链接，插槽填充）？我们如何首先构建和维护该知识库（知识库群体）？这些是在不同背景下的信息提取研究中经常回答的一些问题。

文档分析和检索

另一种常见的行业 NLP 问题包括理解大量文档。我们如何从文档中提取主题（主题建模）？我们如何更智能地索引和搜索文档？我们如何理解搜索查询（查询解析）？我们如何为大型集合生成摘要？

NLP 技术的范围和适用性很广，事实上，NLP 技术可以应用于存在非结构化或半结构化数据的任何地方。作为一个例子，我们将您介绍给 Dill 等人。（2007），他们应用自然语言解析技术来解释蛋白质折叠。

NLP 的前沿

当该领域正在进行快速创新时，写一篇题为“NLP 前沿”的部分似乎是一件愚蠢的事。但是，我们想让您一瞥 2018 年秋季的最新趋势：

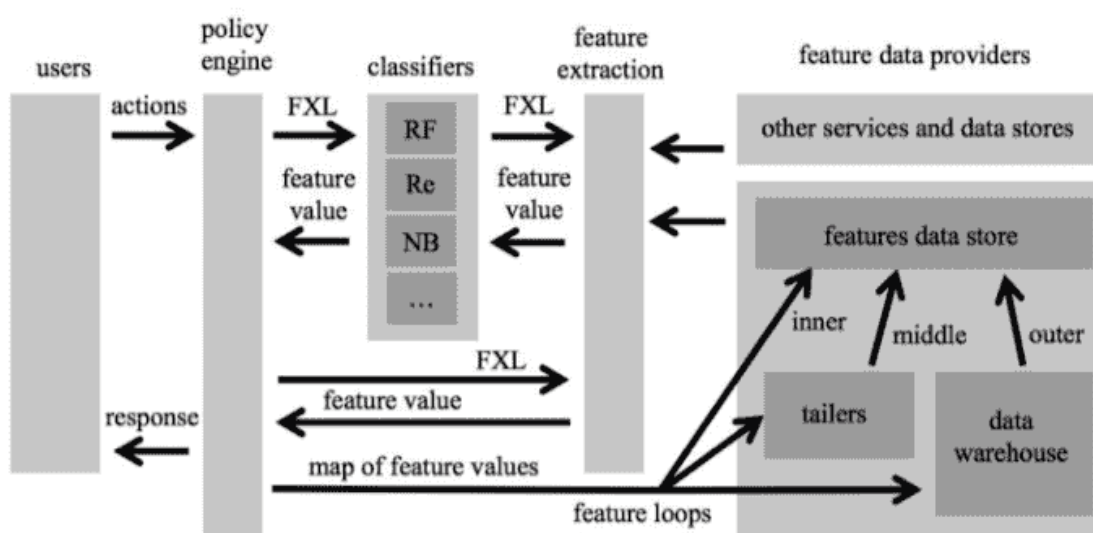
1. 将经典 NLP 文献引入可微学习范式 NLP 领域已有几十年的历史，尽管深度学习领域只有几年的历史。许多创新似乎都在研究新的深度学习（可微学习）范式下的传统方法和任务。阅读经典 NLP 论文（我们建议阅读它们！）时，一个很好的问题是作者试图学习什么？什么是输入/输出表示？我们如何通过前面章节中学到的技术简化这一过程？
2. 模型的组合性原则。在本书中，我们讨论了 NLP,MLP, CNN, 序列模型，序列到序列模型和基于注意力的模型的不同类型的深度学习架构。值得注意的是，尽管我们单独讨论了这些模型，但纯粹是出于教学原因。在文献中看到的一个趋势是组合不同的架构来完成工作。例如，您可以在单词的字符上编写卷积网络，然后在该表示上写入 LSTM，并且通过 MLP 完成 LSTM 编码的最终分类。能够根据任务需求组合地组合不同的架构是使深度学习成功的最有力的想法之一。
3. 序列的卷积。我们在序列建模中看到的最近趋势是使用卷积运算完全模拟序列。作为完全卷积机器翻译模型的一个例子，参见 Gehring 等人。（2018）。解码步骤使用去卷积操作。这是有利的，因为可以使用全卷积模型显着加速训练。
4. 《Attention is all you need》。最近的另一个趋势是用注意机制取代卷积（Vaswani 等，2017）。使用注意机制，特别是称为自我关注和多头注意的变体，您基本上可以捕获通常使用 RNN 和 CNN 建模的远程依赖关系。
5. 迁移学习。迁移学习是学习一项任务的表示和使用表示来改进另一项任务的学习的任务。在最近神经网络的复兴和 NLP 中的深度学习中，使用预训练的单词向量的迁移学习技术已经变得无处不在。最近的工作（Radford 等，2018; Peters 等，2018）证明了语言建模任务的无监督表示如何有助于各种 NLP 任务，如问答，分类，句子相似性和自然 - 语言推断。

此外，强化学习领域最近在对话相关任务方面取得了一些成功，而复杂的自然语言推理任务的记忆和知识基础的建模似乎引起了工业界和学术界的研究人员的高度关注。在下一节中，我们将从经典和前沿转向更直接的事物 - 开发一个系统思考设计生产 NLP 系统。

生产 NLP 系统的设计模式

生产 NLP 系统可能很复杂。在构建 NLP 系统时，重要的是要记住，您正在构建的系统正在解决任务，并且只是实现这一目标的手段。在系统构建期间，工程师，研究人员，设计人员和产品经理可以做出多种选择。虽然我们的书主要关注技术或基础构建块，但将这些构建块放在一起以提出满足您需求的复杂结构将需要一些模式思考。模式思维和描述模式的语言是“在专业领域内描述良好设计实践或有用组织模式的方法。”这在许多学科（Alexander，1979）中很流行，包括软件工程。在本节中，我们将介绍生产 NLP 系统的一些常见设计和部署模式。这些是团队经常需要做出的选择或权衡，以使产品开发与技术，业务，战略和运营目标保持一致。我们在六个轴下检查这些设计选择：

1. 在线 VS 离线系统。在线系统是需要实时或接近实时地进行模型预测的系统。诸如打击垃圾邮件和内容审核等一些任务本质上需要在线系统。另一方面，离线系统不需要实时运行。我们可以将它们构建为在批量输入上有效运行，并且可以利用转换学习等方法。一些在线系统可以是被动的，甚至可以以在线方式进行学习（也称为在线学习），但是许多在线系统是通过定期离线模型构建来构建和部署的，该构建被推向生产。使用在线学习构建的系统应该对抗抗性环境特别敏感。最近的一个例子是著名的 Twitter 聊天机器人 Tay，它误入歧途并开始从在线巨魔学习。正如后见之明所预见的那样，Tay 很快就开始回应令人反感的推文，其母公司微软不得不在推出后不到一天就关闭了这项服务。系统构建中的典型轨迹是首先构建一个离线系统，将其作为一个“在线”系统进行大量工程工作，然后通过添加反馈循环并可能改变学习方法使其成为“在线学习”系统。虽然这种路径在代码库中增加的复杂性方面是有机的，但它可能会引入诸如处理攻击者之类的盲点等等。图 9-3 显示了“Facebook 免疫系统”作为检测垃圾邮件的在线系统的一个例子（警告：大约 2012 年。不是当前 Facebook 基础设施的反映）。请注意在线系统如何比类似的离线系统需要更多的工程设计。



2. 交互 VS 非交互系统。大多数自然语言系统在预测仅来自模型的意义上是非交互式的。实际上，许多生产 NLP 模型深深嵌入到数据处理的提取，转换和加载（ETL）管道的转换步骤中。在某些情况下，人类参与预测循环可能会有所帮助。图 9-4 显示了 Lilt Inc 的交互式机器翻译界面

的一个例子，其中模型和人类共同参与所谓的“混合主动模型”（Green-Initiative Models）中的预测（Green 2014）。交互式系统难以设计，但通过将人类带入循环可以实现非常高的精度。

Nuevos hallazgos realizados por la sonda espacial Mars Reconnaissance Orbiter (MRO) de la NASA proporcionan la evidencia más fuerte hasta ahora de que el agua líquida fluye intermitentemente en el actual Marte.

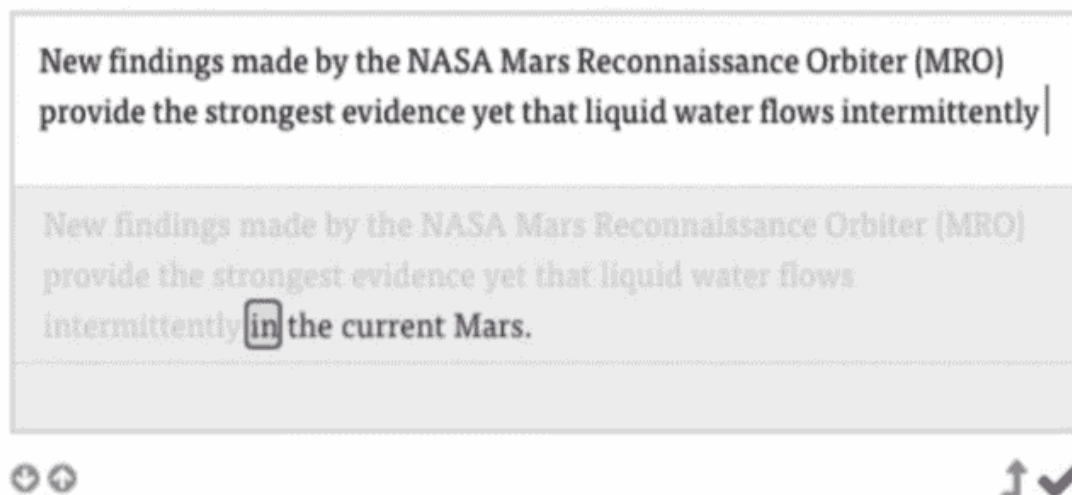
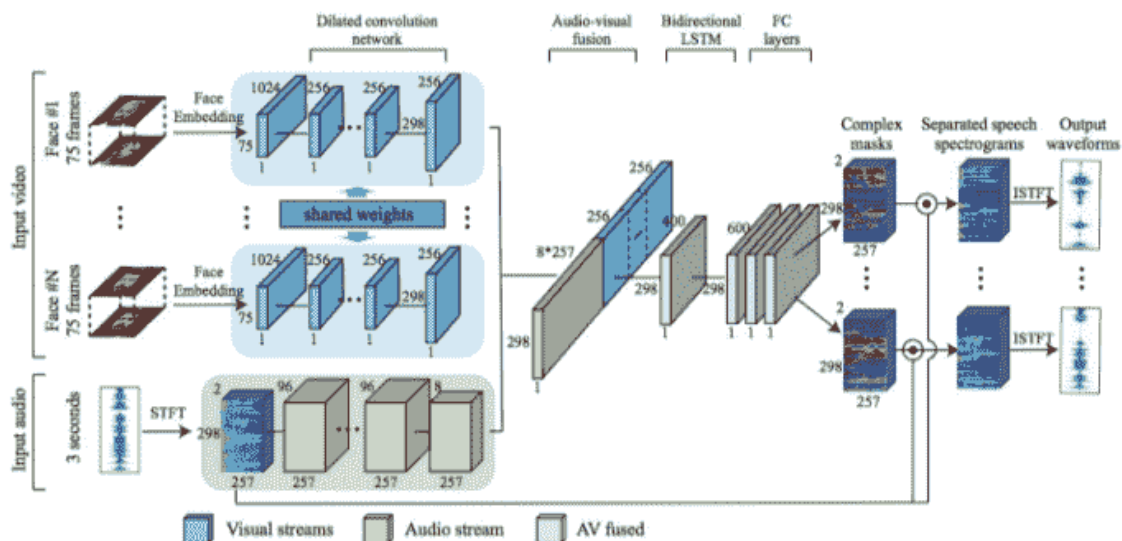


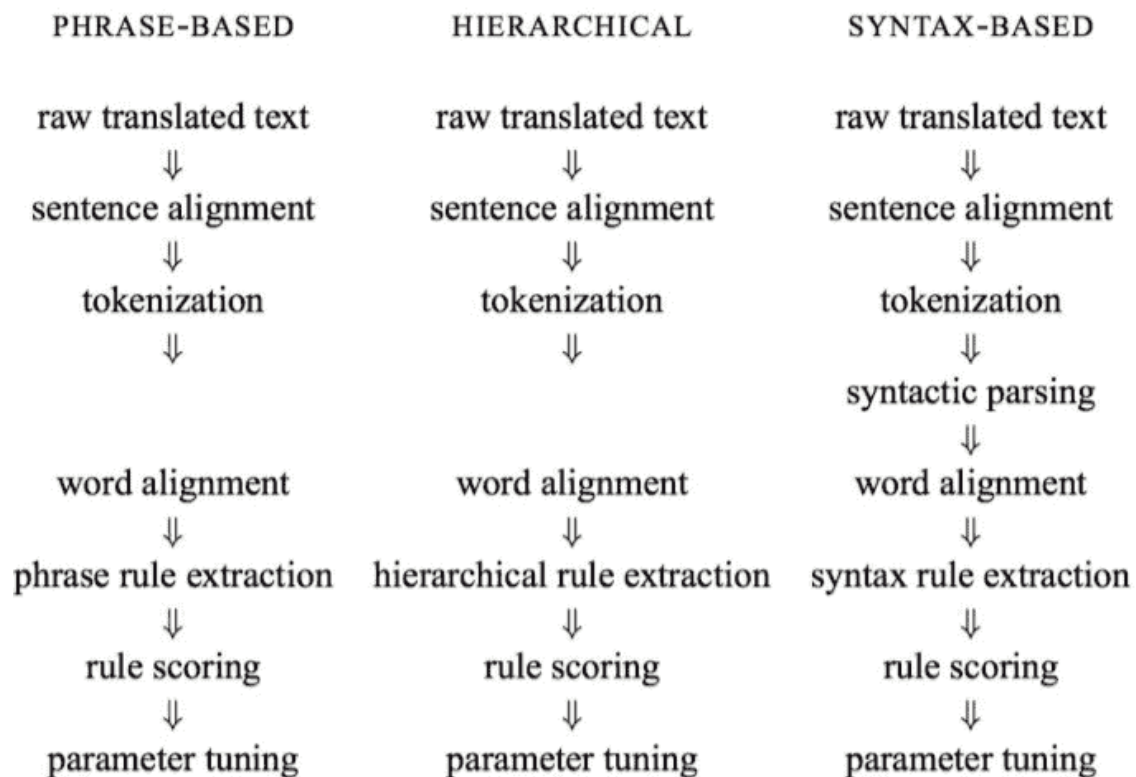
图 9-4。一种人-环机器翻译模型，允许人们修正或改写来自 MT 系统的建议，以产生非常高质量的翻译。（图片由 Lilt Inc 提供）

1. Unimodal versus multimodal systems

在许多情况下，在学习和预测过程中纳入多种模态可能会有所帮助。例如，新闻转录系统不仅使用音频流而且还使用视频帧作为输入是有帮助的。例如，Google 最近的一项名为《Look to Listen》（Ephrat et al.，2018）的作品使用多模式输入来解决扬声器源分离的困难问题（又名鸡尾酒会问题）。多模式系统的构建和部署成本很高，但是对于困难的问题，组合来自多个模态的输入提供了单独使用任何单一模态无法实现的信号。我们也在 NLP 中看到了这个例子。例如，在多模态翻译中，我们可以通过在可用时合并来自多种源语言的输入来提高翻译质量。生成网页主题（主题建模）时，除了网页上的文本外，还可以合并从其中包含的图像中提取的特征，如图 9-5 所示。



1. 端到端 VS 分片系统。自深度学习问世以来，研究人员和工程师可以选择的另一个选择点是构建一个复杂的 NLP 系统，既可以作为不同单元的管道，也可以作为单片端到端系统。端到端设计在机器翻译，摘要和语音识别等许多领域都具有吸引力，精心设计的端到端系统可以显著降低实施和部署的复杂性，并且肯定会减少代码行数。分段系统（图 9-6）将复杂的 NLP 任务分解为子任务，每个子任务单独优化，独立于最终任务目标。分段系统中的子任务使其非常模块化，易于“修补”生产中的特定问题，但通常会带来一些技术债务。



1. 封闭领域 VS 开放领域系统。封闭域系统明确地针对单一目的进行优化：在该域中表现良好。例如，机器翻译系统可以明确优化以与生物医学期刊一起使用 - 这不仅仅涉及生物医学

平行语料库的训练。另一方面，开放域系统旨在用于通用目的（例如，Google Translate）。再举一个例子，考虑一个文件标签系统。如果系统只预测了许多预定义类中的一个（典型情况），则会产生一个封闭域系统。但是，如果系统被设计为在运行时发现新类，那么它就是一个开放域系统。在翻译和语音识别系统的背景下，封闭域系统也被称为“有限词汇”系统。

2. Monolingual versus multilingual systems

为单一语言工作而构建的 NLP 系统称为单语系统。很容易构建和优化单语系统。相比之下，多语言系统可以处理多种语言。在对不同语言的数据集进行训练时，预计它们可以开箱即用。虽然构建多语言系统很有吸引力，但专注于单语版本有其优点。研究人员和工程师可以利用该语言中广泛可用的资源和领域专业知识来生成高质量的系统，否则一般的多语言系统是不可能的。出于这个原因，我们经常发现许多多语言产品被实现为单独优化的单语系统的集合，其中语言识别组件将输入分派给它们。

下一步是什么？

使用像 PyTorch 这样的即将到来的框架和像深度学习这样快速变化的领域，感觉就像在移动地面上建造一座豪宅。在本节中，我们指出了一些与深度学习，PyTorch 和 NLP 相关的资源，以帮助我们的读者继续加强我们在本书中构建的基础。

我们没有涵盖 PyTorch 的每一个功能。我们建议您遵循优秀的 PyTorch 文档并参与 PyTorch 论坛以继续您的 PyTorch 实践：

- [PyTorch 文档](#)
- [PyTorch 论坛](#)。

深度学习领域本身就是来自工业界和学术界的大量活动。大多数深度学习作品出现在 arXiv 的不同类别下：

- [机器学习](#)
- [语言和计算](#)
- [人工智能](#)

了解 NLP 新作品的最佳方法是遵循以下学术会议：

- 计算语言学协会（ACL）
- 自然语言处理中的经验方法（EMNLP）
- 北美计算语言学协会（NAACL）
- ACL 的欧洲分部（EACL）
- 计算自然语言学习会议（CoNLL）

我们建议 aclweb.org 跟踪这些会议和其他会议，研讨会和其他重要的 NLP 新闻的会议记录。

当您准备超越基础时，您可能会发现自己必须阅读研究论文。阅读论文是一门获得的艺术。您可以在[这里](#)找到一些有用的阅读 NLP 论文的提示。

最后，我们将继续提供更多教育材料，[以补充本书的内容](#)。

参考

1. Christopher Alexander. The Timeless Way of Building. Oxford University Press, 1979.
2. Ken A. Dill, Adam Lucas, Julia Hockenmaier, Liang Huang, David Chiang, Aravind K. Joshi. "Computational linguistics: A new tool for exploring biopolymer structures and statistical mechanics." In Polymer, 2007.
3. Hieu Hoang, Philipp Koehn, and Adam Lopez. "A Unified Framework for Phrase-Based, Hierarchical, and Syntax-Based Statistical Machine Translation." In Proceedings of IWSLT, 2009.
4. Tao Stein, Erdong Chen, Karan Mangla. "Facebook Immune System." SNS, 2011
5. Spence Green, "Mixed-Initiative Language Translation." PhD Thesis, Stanford University, 2014.
6. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, "Attention Is All You Need." on arXiv, 2017.
7. Ariel Ephrat, Inbar Mosseri, Oran Lang, Tali Dekel, Kevin Wilson, Avinatan Hassidim, William T. Freeman, Michael Rubinstein. "Looking to Listen: A Speaker-Independent Audio-Visual Model for Speech Separation." SIGGRAPH, 2018.
8. Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer. "Deep contextualized word representations." ACL 2018.