



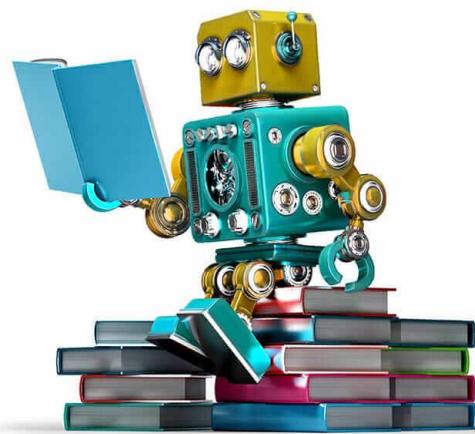
---

# 深度学习与文本挖掘

DEEP LEARNING FOR TEXT MINING

---

邱江涛



2020, TF.KERAS 2.1 版  
西南财经大学经济信息工程学院

# 目录

目录 .....	2
第一章：前言 .....	6
第一节：深度学习的兴起 .....	6
第二节：深度学习框架介绍 .....	9
第三节：关于本课程 .....	10
第二章：神经网络基础 .....	12
第一节：神经网络的发展 .....	12
第二节：神经网络的结构 .....	14
第三节：神经网络的训练 .....	14
第四节：反向传播算法 .....	24
第三章：Keras .....	27
第一节：Keras 中的基本概念 .....	28
第二节：初识 Keras .....	31
第三节：Keras 练习 .....	35
第四章：深度神经网络 .....	42
第一节：为什么需要深度神经网络？ .....	42
第二节：梯度消失、梯度爆炸与解决方法 .....	45
第三节：构建深度前馈神经网络 .....	48
第四节：训练深度神经网络 .....	51
第五节：实例：手写数字识别 .....	59

第五章：卷积神经网络 .....	63
第一节：卷积 .....	63
第二节：卷积神经网络的结构 .....	66
第三节：Keras 卷积层函数 .....	75
第四节：Dropout .....	79
第五节：实例 1：基于 CNN 的手写数字识别 .....	80
第六节：实例 2：基于 1D CNN 的活动识别 .....	82
第六章：神经语言模型和词的表示学习 .....	87
第一节：背景知识 .....	88
第二节：word2vec .....	91
第三节：Keras 实现词表示学习 .....	94
第七章：基于 CNN 的文本分类 .....	104
第一节：Keras 的文本处理 .....	104
第二节：一个简单的文本分类模型 .....	107
第三节：CNN 文本分类模型 .....	109
第八章：循环神经网络 .....	121
第一节：RNN 结构 .....	121
第二节：使用 Keras 构建 RNN .....	126
第三节：LSTM .....	132
第四节：LSTM 的变体 .....	138
第五节：用 Keras 构建词级 LSTM 语言模型 .....	140
第六节：训练 RNN 的一些技巧 .....	144
第九章：基于 LSTM 的文本情感分析 .....	145
第一节：介绍 .....	145

第二节：keras 实施 .....	146
第三节：定制 RNN 层的方法.....	148
第四节：开发 peephole LSTM 情感分类模型 .....	151
第十章：RNN Encoder-Decoder .....	154
第一节：Encoder-Decoder 模型介绍 .....	154
第二节：实例：一个翻译模型.....	157
第十一章：注意力机制.....	162
第一节：动机与原理.....	162
第二节：注意力机制的类型.....	165
第三节：Keras 定制正则项和损失函数.....	171
第四节：实例 1：基于自注意力机制的作者画像 .....	173
第五节：实例 2：文章标题自动生成 .....	176
Reference.....	181
第十二章：预训练、Transformer 和 Bert.....	183
第十三章：机器阅读理解与问答系统 .....	184
第十四章：深度学习在信息检索中的应用.....	185
第一节：简介.....	185
第二节：文本块的表示学习 .....	185
Appendix：常用 keras 的类和函数.....	192
第一节：数据初始化的类和函数 .....	192
第二节：keras 常用的激活函数.....	192
第三节：keras 常用的损失函数.....	194
第四节：keras 常用的优化函数.....	195
第五节：keras 中的层 .....	197

第六节：Merge 层 ..... 201

# 第一章：前言

## 第一节：深度学习的兴起

2016 年人工智能界最激动人心的一件事就是 Google 旗下的 DeepMind 公司开发的 AlphaGo 以 4 : 1 的比分击败了韩国围棋九段棋手李世石。2017 年则应该是排名第一的柯洁被 AlphaGo 以 3:0 血洗。



图 1-1. AlphaGo 的人机对弈

AlphaGo 的核心技术是 Deep Learning+Reinforcement Learning 技术。AlphaGo 的胜利证明了 Deep Learning 技术的强大，为本来已经在学术界已经很火的 Deep Learning 更是浇上一桶油。

Reinforcement Learning 不是本课程的授课内容。Reinforcement Learning 通常翻译做强化学习。被称作是机器学习领域有监督学习和无监督学习外的第三种学习。

为什么说强化学习是第三种机器学习，因为前两种都面对这一个数据集。而强化学习面对的是一个环境，而且这个环境是可交互的。

## 深度学习取得的成就

按照 Nature 文章 Human-level control through deep reinforcement learning，该文章中提出的增强深度学习方法在机器玩游戏的任务中取得了 23/43/49 的性能。即，49 项游戏中，有 43 项可以基本平均的人类选手，在 23 项游戏中击败人类顶尖高手。

在语音识别领域，按照微软语音识别研究组的报告，自从 2010 年采用深度学习技术以来，语音识别领域有了突破性的进展，的错误率有了大幅的下降。

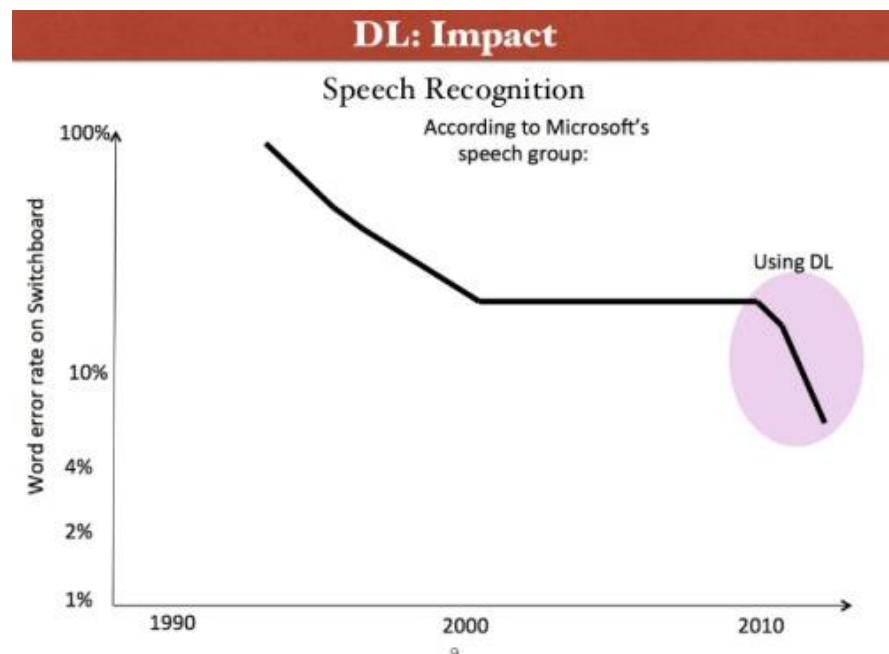


图 1-2：语音识别的错误率变化

按照在 MNIST ( 手写数字数据集 ) 测试集上的研究报告。采用纯神经网络的准确率是 96.59%，采用支持向量机 SVM，当采用 LibSVM 的默认参数，准确率是 94.53%；采用优化了参数的 SVM 准确率达到 98.56%；而采用卷积神经网络准确率达到了 99.79%。MNIST 有训练集有 6 万张图片，测试集有 1 万张图片。卷积神经网络仅有 21 张图片未能正确分类。如图 1-3 所示。



图 1-3. 未被识别的手写数字

深度学习技术被 MIT Technology Review 评为了 2013 年 10 大突破技术之一。其评价语是

With massive amounts of computational power, machines can now recognize objects and translate speech in real time. Artificial intelligence is finally getting smart.

可以访问一个深度学习的演示网站 : playground.tensorflow.org 了解深度学习的强大功能

### 神经网络与深度学习的关系

有人评价神经网络是最优美的编程范式 ( programming paradigms ) 之一。传统的编程方法中，我们告诉计算机做什么，将大问题分解为小的计算机能处理的任务。对比之下，在神经网络中我们不告诉计算机怎样解决问题。神经网络会自己从观察到的数据学习，并计算出解决方案。

从数据中自动学习听起来很诱人。然而直到 2006 年，研究人员才知道怎样超越传统的方法来训练神经网络。这一年在深度神经网络进行学习的技术被发明，这就是现在称之为的**深度学习**。随后这些技术被进一步开发。今天，深度神经网络和深度学习解决在计算机视觉、语言识别和自然语言处理等领域的问题达到了非常优秀的性能。一些商业公司如 Google, Microsoft, Facebook 已经开发和大规模部署了深度学习框架。

神经网络是由生物学启发的编程机器学习模型（有称是编程范式 programming paradigm），由此计算机可以从观察的数据进行学习。深度学习是一个非常有力的在神经网络上进行学习的技术集合。神经网络和深度学习当前对图像处理、语言识别和自然语言处理中的许多问题提供了最好的解决方案。

深度学习比起传统神经网络的优势在于：

1. 采用了新的激活函数 ReLu，可以使得网络的层次超过三层
2. 采用 Dropout, Maxout 和随机池化技术 ( Stochastic Pooling ) 解决过拟合问题
3. 可以采用 GPU，解决多层网络训练速度慢的问题。

### 深度学习的问题

但是深度学习有它的问题：深度学习目前还停留在实验科学的阶段，其严格的数学解释还未完全建立。Geometric Understanding of Deep Learning 一文从几何的角度理解深度学习，为深度学习提供严密的数学论证。NIPS2018 有论文从数学角度尝试解释 Dropout 的作用，深入探究 dropout 的本质。

## 第二节：深度学习框架介绍

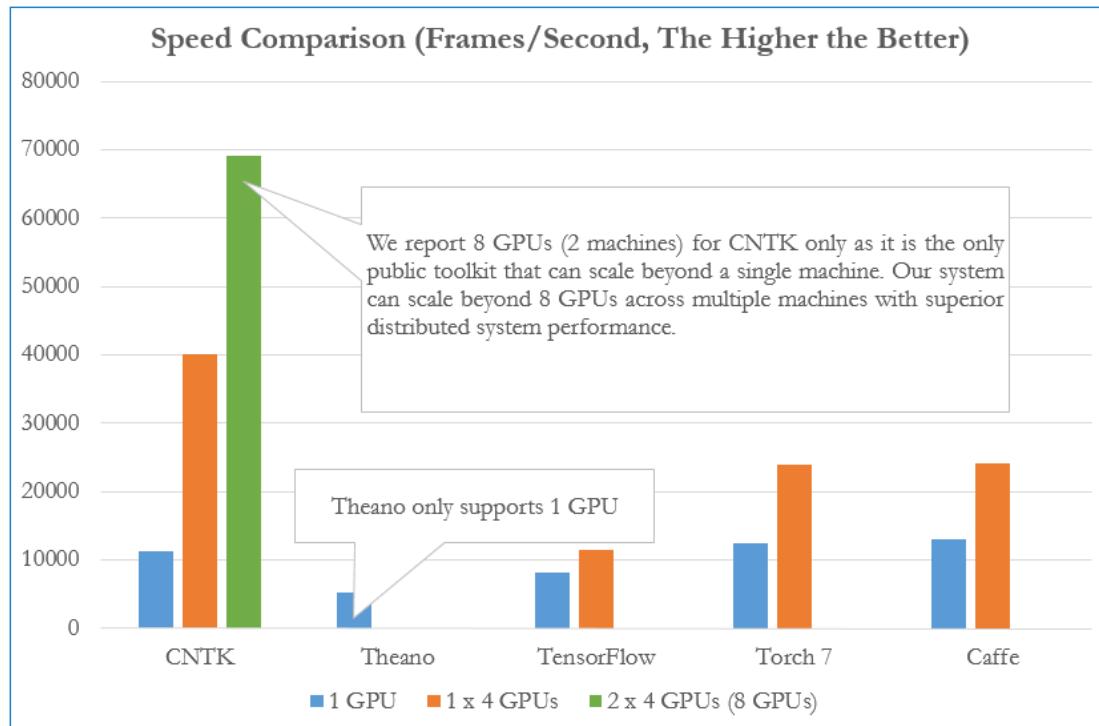


图 1-4：流行的深度学习框架

现在流行的深度学习框架有很多。包括微软开发的 CNTK , Google 的 TensorFlow , 蒙特利尔大学开发的 Theano, facebook 开发的 Caffe 和开放社区的 Torch ( 几个人联合开发 , 无公司背景 ) 和由中国人发起的 MXNet。图 1-9 是微软公司发布的各种深度学习框架性能对比图。

framework	base language	multi-GPU?	pros	cons
<b>TensorFlow</b>	Python and C++	yes	-	-
<b>Torch</b>	Lua	yes	1.) easy to set up 2.) helpful error messages 3.) large amount of sample code and tutorials	can be somewhat difficult to set up in CentOS
<b>Caffe</b>	C++	yes	-	general
<b>Theano</b>	Python	By default, no. Can use more than one but requires a workaround	1.) expressive Python syntax 2.) higher-level spin-off frameworks 3.) large amount of sample code and tutorials	error messages are cryptic

图 1-5：几种深度学习框架简介

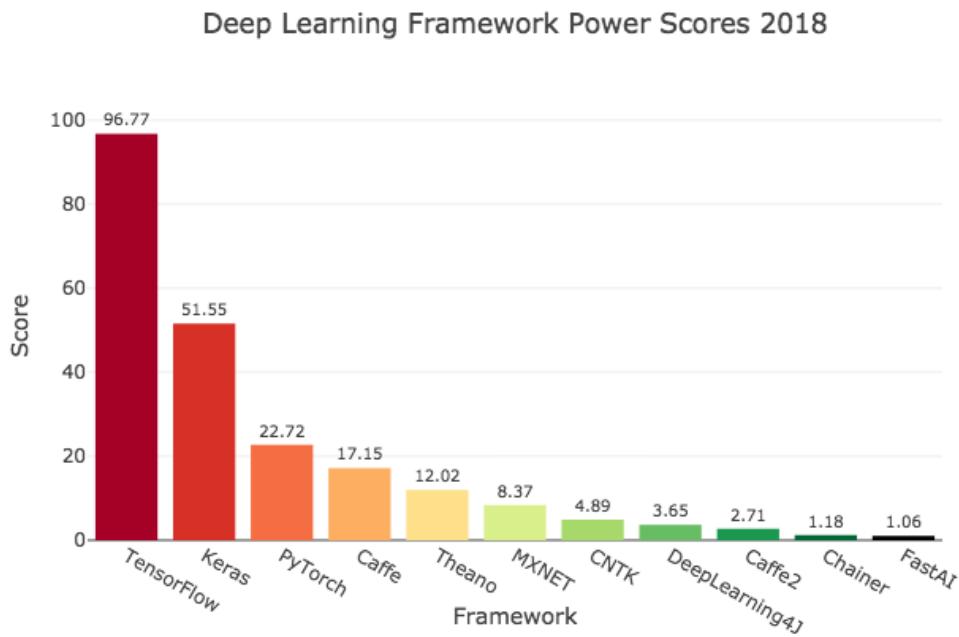


图 1-6 : Deep learning frameworks ranking computed by Jeff Hale, based on 11 data sources across 7 categories

微软的 CNTK 是采用 C++ 开发，提供了 C++ 和 Python 的接口。Torch 有个 Python 版本 Pytorch。相比 Tensorflow 它有个优势是调试方便。图 1-5 对几种深度学习框架做一个简单的描述。

图 1-6 这是网友总结的个深度学习框架的受欢迎评分。

### 第三节：关于本课程

在本课程中，我们先介绍神经网络的基础知识，然后介绍 Deep Learning，进一步介绍实现 Deep Learning 的工具 Keras，并用 Keras 实现几个 Deep Learning 的模型。Deep Learning 在 NLP 领域取得成功，我们将介绍相应的模型，并开发这些模型。最后我们用 Deep Learning 去解决文本挖掘的实际应用问题。

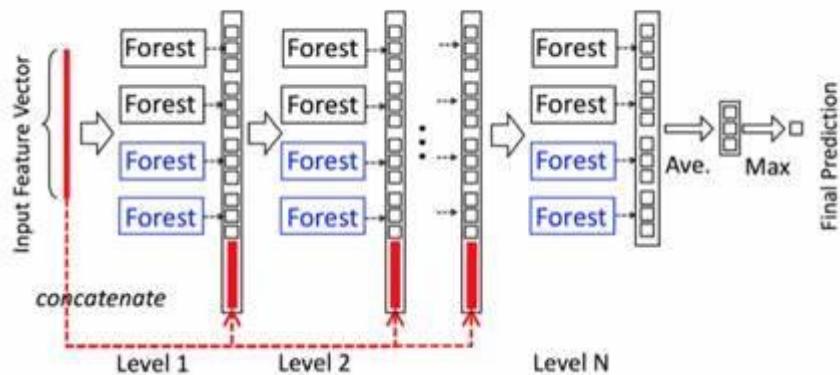
由于深度学习的发展非常迅猛，而本课程只有 9 周，另外由于深度学习对硬件要求的越来越高。因此本课程面向的是深度学习的基础知识和实践操作。更多的新知识理论会在在最后一章的最新发展中介绍。

**拓展与思考：**

深度学习等价于深度神经网络吗？

SIAM，即国际工业与应用数学学会，的报纸 SIAM news 在 2018 年的 6 月份头版上文章说**深度学习是机器学习中使用深度神经网络的子领域**。这也是我们通常对深度学习的认知。

南京大学周志华团队推出了 gcForest，即 multi-Grained Cascade Forest。它是一个基于决策树的集成模型，是一个级联树结构的深度模型。



该模型的每一层中包含多个随机森林。该模型据周志华团队介绍，“它也可以在除了大规模图像数据之外的任务中获得和深度神经网络相似的表现。在大规模图像数据任务中不能取胜，其中原因是“我们目前没有适当的硬件训练足够大的模型”。

## 第二章：神经网络基础

### 第一节：神经网络的发展

神经网络 (Neural Network , NN)或人工神经网络 ( Artificial Neural Network , ANN ) 这一术语的起源于试图发现人脑进行信息处理的数学描述。现在神经网络的概念已经扩展和迁移到由生物神经网络灵感激发的一种计算模型。1958年麻省理工学院的Frank Rosenblatt创建了感知机 ( perceptron ) , 感知机也叫单层神经网络。它是一个二分类模型，如图2.1所示。

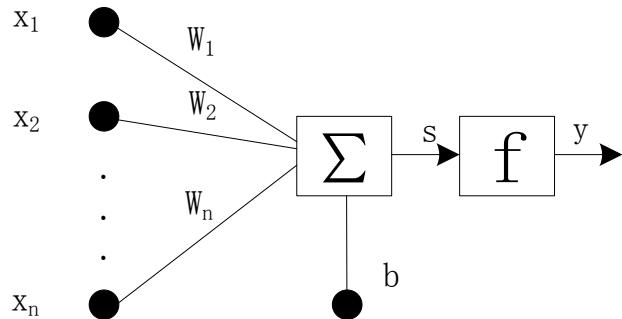


图 2.1 感知机

感知机的输入是向量， $x = (x_1, \dots, x_n)$ 输出是一个 0,1 或 -1,1 的值  $y$ 。 $y$  的计算参见公式

$$s = \sum_{i=1}^n w_i x_i + b$$

$$y = f(s)$$

其中  $b$  为偏置，激活函数 ( activation function )  $f$  为

$$f(s) = \begin{cases} 1, & \text{if } s > 0 \\ 0, & \text{否则} \end{cases}$$

或

$$f(s) = \begin{cases} 1, & \text{if } s > 0 \\ -1, & \text{否则} \end{cases}$$

然而 Minsky 和 Papert 在 1969 年的一篇论文中指出感知机的重大缺陷，它只能解决线性可分问题，不能解决 XOR（异或）问题。如图 2.2 (a) 所示，有四个点对应感知机的输入数据  $x=\{(0,0), (1,0), (0,1), (1,1)\}$ 。黑色和白色代表数据的类别，我们不可能找到一根直线将两类数据分开。因为不能解决异或问题，神经网络的研究陷入了停滞。

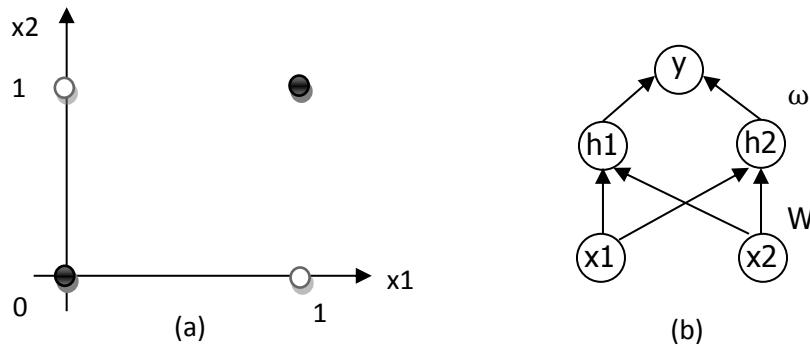


图 2.2 XOR 问题 (a) 和加入隐层的感知机 (b)

到 1975 年随着多层神经网络的诞生和反向传播算法的发明，神经网络的研究又迎来了一个高峰。图 2.2 (b) 是加入一个隐层后的神经网络。下面分析感知机加入一个隐层后异或问题是如何被解决的。

$W$  是输入层到隐层两个节点的边的权重； $\omega$  是隐层到输出层的边的权重； $c$  是隐层两个节点的偏置； $b$  是输出节点  $y$  的偏置。设  $W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ ,  $c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$ ,  $\omega = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$ ,  $b = 0$ 。

设隐层的计算是  $h = \max\{0, W^T x + c\}$ ，输出  $y = \omega^T \max\{0, W^T x + c\} + b$ 。考虑对应输入  $x=\{(0,0), (0,1), (1,0), (1,1)\}$ ，隐层的输出是  $h=\{(0,0), (1,0), (1,0), (2,1)\}$ 。图 2.3 是将输入  $x$  经过隐层计算后得到结果绘制的图。可以看到，此时可以找到一根直线将两类数据完美分开。这是因为该神经网络加入了隐层后，将原始空间中的二维数据进行了非线性变换，映射到了一个新的空间。进一步计算可以得到  $y=\{0,1,1,0\}$ ，加入隐层的神经网络可以将输入数据正确分类，完美的解决了异或问题。

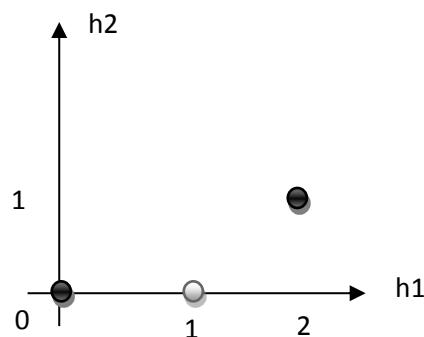


图 2.3 在隐层空间的转换后的输入数据

然而，因为随着隐层层数的增加带来神经网络学习困难的问题，进入到二十世纪 90 年代神经网络的研究又开始放缓。反向传播 ( back propagation, BP ) 算法是神经网络最受欢迎的训练算法。但是当神经网络的层次增加时，训练神经网络的目标函数是一个非凸 ( non-convex ) 的函数，BP 算法通常陷入局部最优。特别地，当层数增加的越多，问题越严重，BP 算法不能很好的训练模型。因此 2006 年以前的神经网络通常是浅层神经网络，它们最多包含两层非线性的特征转换 ( 隐层 )。在 2006 年，Hinton 提出了受限波兹曼机 RMB ( restricted Boltzmann machines )，它有效的解决了更多层神经网络的学习问题，深度学习 ( Deep Learning ) 由此而来。随着深度学习的诞生，神经网络又一次迎来新生。本书只讨论浅层的前馈神经网络，深度学习的内容超出了本书的讨论范围。

## 第二节：神经网络的结构

神经网络的结构包括神经元 ( neuron ) 和连接神经元的有向有权重的边。

### 1. 神经元

神经网络最基本的处理单元是神经元。神经元的输入通常是一个向量。单个神经元就是感知机，如图 2.4 所示，输入是一个长度为  $R$  的向量。向量的每个元素通过一条有权重的边和神经元相连，进行加权求和的运算。

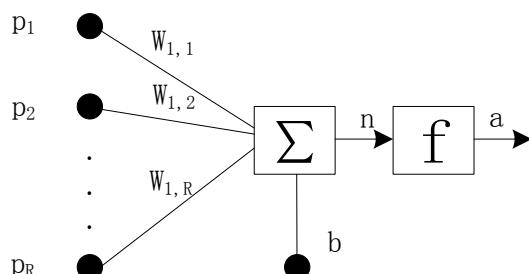


图 2.4 一个神经元的运算过程

该神经元有一个偏置值  $b$ 。它与所有输入的加权和累加，从而形成净输入  $n$ 。

$$n = W_{1,1}p_1 + \dots + W_{1,R}p_R + b$$

这个表达式写成矩阵形式为

$$n = Wp + b$$

其中单个神经元的权重矩阵只有一行元素。经过一个激活函数  $f$ ，神经元的输出可以写成

$$a = f(Wp + b)$$

## 2.网络结构

一般来说，单个神经元并不能满足实际应用的需求。在实际操作中需要有多个并行操作的神经元，这些并行神经元组成的集合称为层，如图 2.5 所示。神经元的层是由  $S$  个神经元组成的单层网络。 $R$  个输入中的每一个均与每个神经元相连，权重矩阵是  $S$  行  $R$  列的矩阵。

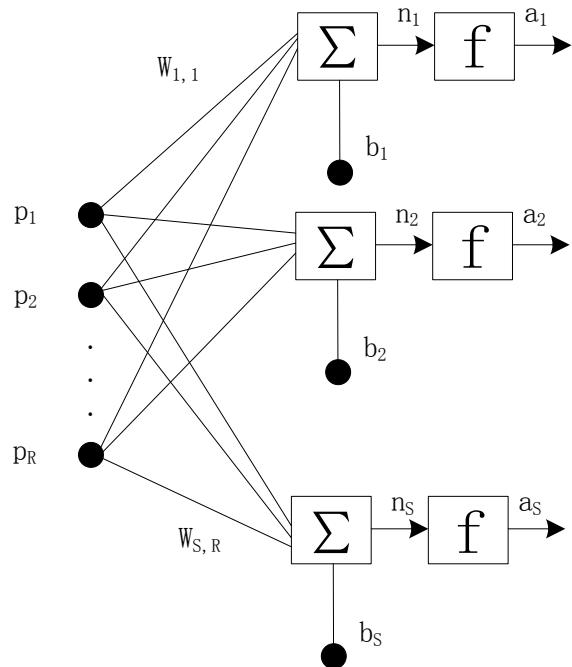


图 2.5 一层网络的运算

通常每层的输入个数并不等于该层中神经元的数目，即  $R \neq S$ 。同一层中的神经元有相同的激活函数。输入向量通过权重矩阵  $W$  进入网络。

## 多层神经元

现在考虑有几层神经元的网络，每层都有自己的权重矩阵  $W$ ，偏置向量  $b$ 、净输入向量  $n$  和一个输出向量  $a$ 。图 2.6 是一个三层网络（这里输入没有算作是一个层）

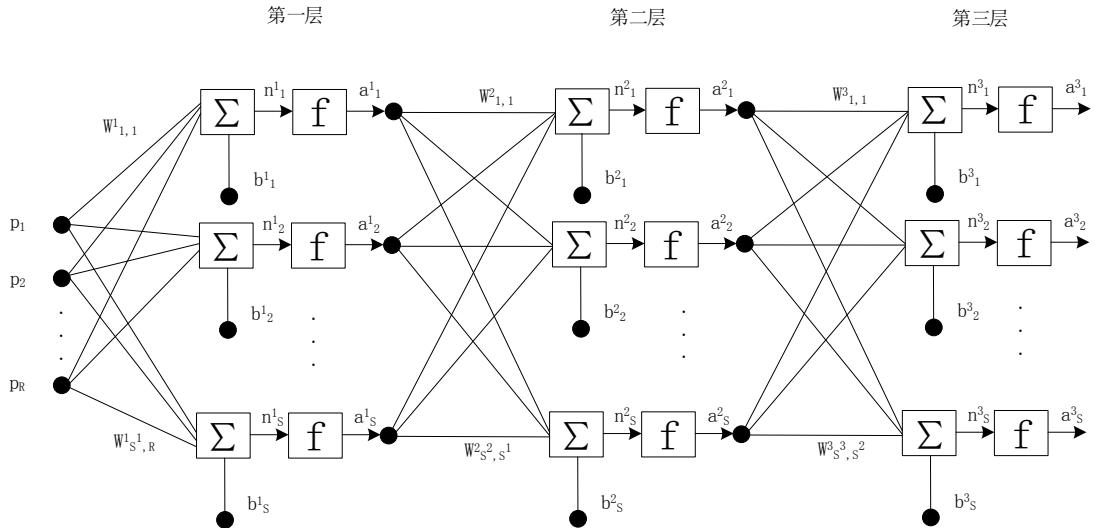


图 2.6 三层神经网络

如图所示第一层有  $R$  个输入， $S^1$  个神经元，第二层有  $S^2$  个神经元。不同层可以有不同的数目的神经元。第一层和第二层的输出分别是第二层和第三层的输入。如果某层是网络的输出，那么称该层为输出层，其他层称为隐层。

多层网络的功能要比单层网络的功能强大许多。例如，如果第一层采用 S 形激活函数，第二层采用线性传输函数的网络，经过训练可对大多数函数达到任意精度的逼近，而单层网络做不到这一点。

如此，决定一个网络的层数和神经元个数非常重要。首先，网络的输入和输出是由问题所定义的。如果有 4 个外部变量作为网络输入，那么网络就有 4 个输入。如果是一个多分类问题，有 3 个类别，则网络的输出层就有 3 个神经元。最后，输出信号所期望的特征有助于选择输出层的激活函数。如果是分类问题，要求输出是 0 或 1，那么该输出神经元就可以用 S 形激活函数。对于确定多层网络中其他层的神经元个数并没有明确的方法。对于层数，普遍是小余 3 层。

对于偏置。是否使用偏置是可以选择的。偏置给网络提供了额外的变量，从而使网络又了更强的能力。

### 3. 激活函数 ( Activation Function )

激活函数也有翻译为，活化函数或传输函数。激活函数  $f(x)$  可以是线性或非线性函数。下面介绍几个在现在的深度学习中常用的激活函数（同样适用于浅层的神经网络）。

#### ( 1 ) ReLU

ReLU ( rectified linear unit ) 即  $f(x) = \max(x, 0)$ 。其输出结果  $x$  大于 0 则输出  $x$ ，否则输出 0。如图 2.7 所示。

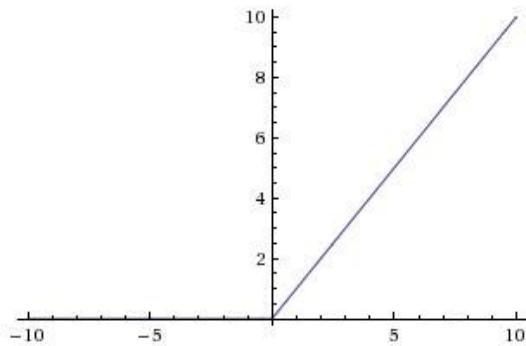


图 2.7 ReLU 激活函数

### ( 2 ) Sigmoid

对数 S 形激活函数，又称 log-sigmoid 或 sigmoid 函数。该函数输入在  $(-\infty, +\infty)$  之间，输出则在 0,1 之间。其数学表达式为  $f(x) = \frac{1}{1+e^{-x}}$ 。如图 2.8 所示。

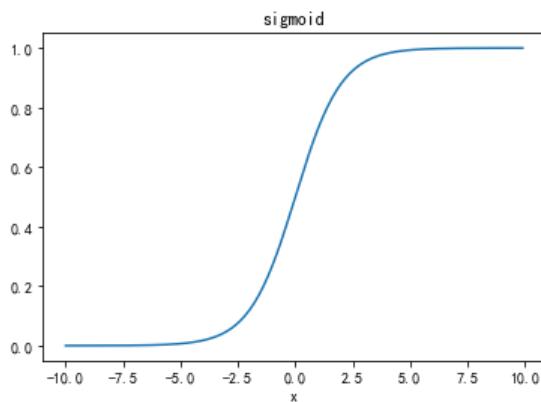


图 2.8 sigmoid 激活函数

从某种程度上说，正是由于 sigmoid 函数是可微的，所以用于反向传播算法训练的多层网络使用了该函数。

### ( 3 ) tanh

tanh 激活函数也是一种 S 形激活函数。 $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ 。如图 2.9

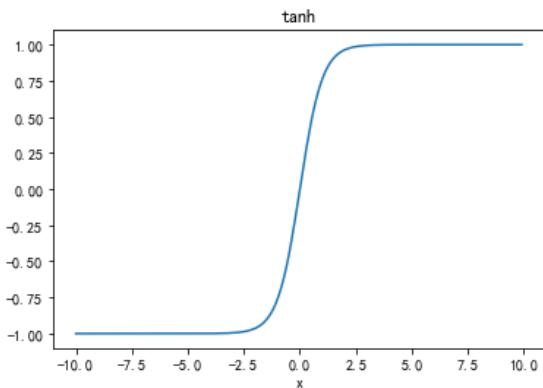


图 2.9 tanh 激活函数

#### ( 4 ) softmax

Softmax 函数，将一个任意实数值的  $k$  维向量  $z$  规范化到一个实数值在  $[0,1]$  的  $k$  维向量  $\sigma(z)$ ，满足向量元素的和为 1。计算过程参见公式

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

#### ( 5 ) 线性激活函数

线性激活函数即激活函数即  $f(x) = cx$ 。在神经网络完成回归任务时的输出层经常采用线性激活函数。

### 4.前馈神经网络

神经网络包含多种拓扑结构，如前馈神经网络，循环神经网络，自组织映射网络（Self Organizing Map, SOM）等。这里只讨论最流行的前馈神经网络。前馈神经网络（FeedForward Network）中各神经元从输入层开始，接收前一级输入，并输出到下一级，直至到输出层。整个网络中无反馈。它包含感知机（最简单的前馈网络），BP（反向传播）神经网络，RBF（Radial Basis Function，径向基函数）网络等。图 2.6 讨论的神经网络就是前馈神经网络。

## 第三节：神经网络的训练

### 2.3.1 基于梯度的优化

机器学习通常模型训练的方法是：

We have a way of evaluating the **loss**, and now we have to **minimize** it. We'll do so with gradient descent. That is, we start with random parameters, and **evaluate** the gradient of the loss function with respect to the parameters, so that we know how we should change the parameters to decrease the loss.

优化是通过调整  $x$  来最大化或最小化一个目标函数  $f(x)$  的一个任务。当最小化  $f(x)$  时， $f(x)$  称为代价函数 (cost function)、损失函数 (loss function) 或误差函数 (error function)。在一些机器学习任务中，通常对模型进行最大似然估计，建立的是模型的似然函数。模型学习的过程是最大化似然函数。但通常把似然函数转换成负 log 似然函数，这就是损失函数了。因此，最大化和最小化是相对的。

现在我们有个函数  $y=f(x)$ ，该函数的导数标记为  $f'(x)$ 。导数  $f'(x)$  给出了函数在  $x$  点的斜率。换句话说，该导数定量了输入的一个小改变怎样导致了输出的小改变。

$$f(x + \epsilon) = f(x) + \epsilon f'(x)$$

导数对最小化一个函数是有用的，它告诉了我们怎样改变  $x$  来制造  $y$  的一点点减小。如果  $\epsilon f'(x) < 0$ ，就可以使得  $f(x + \epsilon) < f(x)$ 。因为  $f'(x)$  的符号可正可负，如此只要  $\epsilon$  的正负符号随着  $f'(x)$  的符号做相应改变就可以做到  $\epsilon f'(x) < 0$ 。也即，对于我们的最小化任务来说，对于一个足够小的  $\epsilon$ ，可以使得  $f(x - \epsilon \text{ sign}(f'(x)))$  小于  $f(x)$ 。那么，我们可以通过在和导数相反的符号方向移动  $x$  一小步来减小  $f(x)$ 。这个技巧称为梯度下降 (gradient descent)

注：如果  $x > 0$ ,  $\text{sign}(x)=1$ ; 否则  $\text{sign}(x)=-1$ .

在机器学习任务中目标函数  $f(x)$  中的  $x$  就是模型的参数。机器学习的过程就是，不断调整  $x$ ，以获得目标函数最值（或损失函数最小值）的过程。例如，如图 2.10 所示。

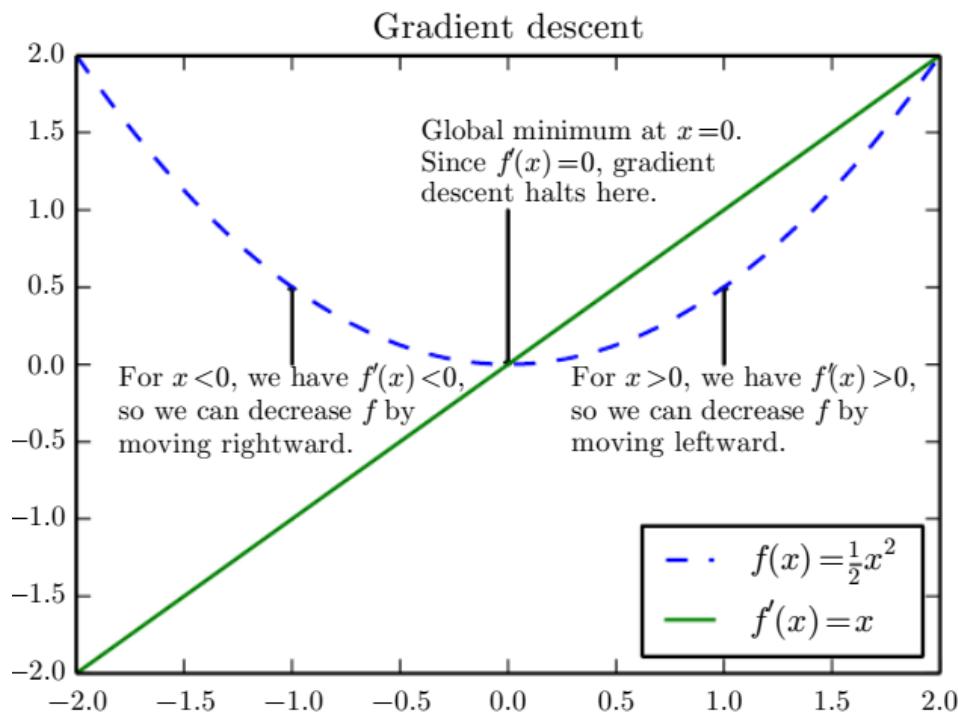


图 2.10 梯度下降

现在有个函数 $f(x) = \frac{1}{2}x^2$ ，它的导数是 $f'(x) = x$ 。我们现在想求得该函数的最小值时的对应 $x$ 值。假设现在 $x=-1$ ，那么 $f'(x) = -1$ 。我们调整 $x$ 足够小的一步，即与梯度相反的方向改变，即此时增加 $x$ ，则可以减小 $f(x)$ 。同样的，当 $x=1$ ， $f'(x) = 1$ ，我们沿着梯度相反的方向，即减小 $x$ 的方向，就可以减小 $f(x)$ 。

当然梯度下降的方法，并没有保证一定能到达全局最小值，即全局最优。很有可能陷入一个局部最优，如图 2.11 所示。

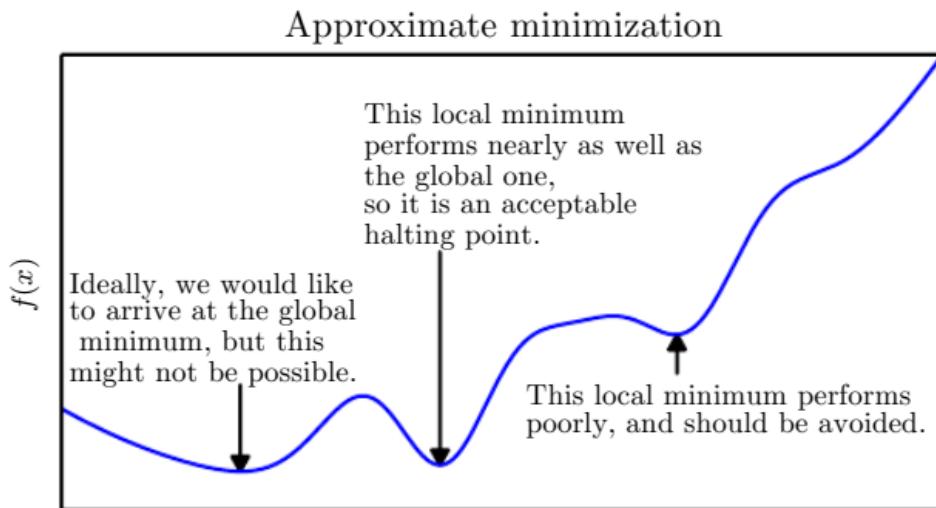


图 2.11 优化过程

当机器学习模型有多个参数时，就是求得每个参数的偏导。此时的梯度就是一个包含所有参数偏导的向量，用符号 $\nabla_x f(x)$ 表示。 $x$ 是参数向量， $x_i$ 就是第 $i$ 个参数。

### 2.3.2 训练一个感知机

BP 神经网络就是采用反向传播算法训练的前馈神经网络。讨论反向传播算法前，我们先看怎样训练一个感知机。

我们用一个例子来解释一个感知机的训练过程。问题描述如下：每天你去餐馆里吃早餐。每天的早餐点三样食物：小菜、包子和饮料。每天你点的这三样食物量不一样，收银员仅仅告诉你总价。几天后，可以用一个感知机来推算出各食物的价格。我们以一个线性激活函数的单神经元的感知机为例。

$$y = \sum_i w_i x_i = W^T X$$

这里 $y$ 是每天吃的早餐的价格。 $W$ 是三样食物的价格， $x$ 是每样食物的份量。训练模型，即得到模型的参数是一个优化过程。优化过程中通常需要建立一个目标函数。目标函数的建立有很多种，误差平方和是其中一种。

第一步，建立损失函数（目标函数）

$$E = \frac{1}{2} \sum_{n \in N} (t^{(n)} - y^{(n)})^2$$

$t^{(n)}$ 是训练集中一顿早餐的价格， $y^{(n)}$ 是用模型估计的价格。N是训练集中的样本数。

第二步，用梯度下降方法来训练模型，目标函数对各参数求偏导获得梯度。对参数（当前是权重  $w_i$ ）求偏导

$$g_i = \frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_n \frac{\partial y^{(n)}}{\partial w_i} \frac{dE}{dy^{(n)}} = - \sum_n x_i^{(n)} (t^{(n)} - y^{(n)})$$

第三步，更新权重。用梯度乘上学习率 $\eta$ 来更新参数。通过多次迭代，最终算法收敛，得到最终的估计参数。

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} = \eta \sum_n x_i^{(n)} (t^{(n)} - y^{(n)})$$

$$w_i \leftarrow w_i + \Delta w_i$$

### 感知机训练算法

Steps:

1. 初始化权重 W
2. Loop:
3. 初始 $\Delta W \leftarrow 0$
4. 对于训练集中的每个实例 n 完成下面的步骤
5. 计算输出 $y^{(n)} = f(Wx^{(n)})$
6. 累加 $\Delta w_i = \Delta w_i + x_i^{(n)} (t^{(n)} - y^{(n)})$
7. 更新权重 $W = W + \eta \Delta W$
8. END

这里 $\eta$ 是学习率。

Python 代码如下：

```
eta=0.005
ws=[50,50,50]
train=((2,5,3),850),((1,4,7),1050),((2,3,5),950),((3,6,9),1650),((7,4,1),1350))

for _ in range(500):
    y=[]
    d=[]
    delta_ws=[0,0,0]
```

```

for xs,t in train:
    yn=sum([w*x for w,x in zip(ws, xs)])
    dn=t-yn
    delta_ws=[xi*dn+dw for xi,dw in zip(xs,delta_ws)]
    ws=[w+eta*delta_w for w,delta_w in zip(ws,delta_ws)]
print(ws)

```

运行结果：

[149.99854969104385, 50.00249323213941, 99.9987175643975]

下面我们再介绍随机梯度下降算法 ( Stochastic Gradient Descent, SGD )。梯度是损失函数对每个参数求偏导得到的一个向量。梯度的每个元素是梯度的方向。SGD 在每次算法迭代中根据偏导值反向调整参数的变化。

SGD 的原理是，我们对一个函数求二阶导数。当  $f''(x) = 0$  时可以获得该函数的极值。然而计算机只有通过数值计算的方式求解二阶导数。即，先随机初始化各参数；计算函数对各参数的一阶偏导  $u = f'(x_i)$ ，求得梯度；根据梯度反方向调整参数值  $x_{i+1} \leftarrow x_i - \eta u$ ；当  $\eta$  足够小， $f(x_{i+1})$  比  $f(x_i)$  更小。重复这个过程将可以发现最优的  $x_i$ 。

### 随机梯度下降算法

输入：学习率  $\eta$

1. 初始化参数  $\theta$
2. While (未达到停止迭代的标准) do
  - a. 从训练集  $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$  中抽样  $m$  个实例
  - b. 计算梯度  $\hat{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
  - c. 更新参数  $\theta \leftarrow \theta - \eta \hat{g}$
3. End While

$f(x^{(i)}; \theta)$  表示参数  $\theta$  确定时将输入  $x^{(i)}$  带入模型计算的结果。 $L(f(x^{(i)}; \theta), y^{(i)})$  表示计算值和真实值的差值。当随机梯度下降算法中  $m$  取值  $m > 1$  算法称为 minibatch SGD；当  $m=1$  称为 Online GD。上面的步骤 2.b 求了样本的均值 ( $1/m$ )，求与不求均值效果是一样的，只需调整学习率  $\eta$  的大小。与前面感知机训练算法相比，随机梯度下降算法关键就是抽样 minibatch。

可以看到，对于 online GD，考察每条训练数据实例后，就计算损失函数的梯度，紧接着就更新参数。而前述的一般梯度下降算法是，考察完所有实例后计算损失函数的梯度，然后再更新参数。一个 Online GD 的例子如下：

当设定初始权重为 $[50, 50, 50]$ 。学习率 $\eta = 1/35$ 。当一天的早餐是：小菜 2 份，包子 5 个，饮料 3 杯，计算价格是 500，但实际价格是 850。差值是  $\text{error} = 850 - 500 = 350$ 。计算  $\text{delta} = \eta(t^n - y^n)x^n = [1/35 * 350 * 2, 1/35 * 350 * 5, 1/35 * 350 * 3] = [20, 50, 30]$

则更新权重为 $[70, 100, 80]$ 。

Online GD 的 python 代码如下：

```
eta=1/35.0
ws=[50,50,50]

train=((2,5,3),850),((1,4,7),1050),((2,3,5),950),((3,6,9),16
50),((7,4,1),1350))

for _ in range(100):
    for xs,t in train:
        y=sum([w*x for w, x in zip(ws,xs)])
        delta_ws=[eta*x*(t-y)for x in xs]
        ws=[w+delta_w for w,delta_w in zip(ws,delta_ws)]

print(ws)
```

运行结果如下：

```
[149.9999854696171, 50.00004609118093, 99.99997795027406]
```

关于随机梯度下降算法的一些讨论：

(1) 学习过程最终会得到完美答案吗？

很有可能得到的结果不是最优的。

(2) 权重收敛到正确值的速度有多快？

跟你的训练集有一定关系。如果输入向量的某些维度高度相关，会收敛的很慢。例如，前面的例子中，每天买的早餐包子、稀饭、小菜的比例都是相同的。几乎不会得到正确结果。

(3) online、minibatch 和标准梯度下降算法性能上有什么区别？

标准梯度下降每一轮迭代需要所有样本参与，对于大规模的机器学习应用，经常有 billion 级别的训练集，计算复杂度非常高。因此，有学者就提出，反正训练集只是数据分布的一个采样集合，我们能不能在每次迭代只利用部分训练集样本呢？这就是 minibatch 算法。

我们这里对 online GD 的解释是 minibatch 中的  $m=1$  的情况。也有人将 online GD 描述为一条训练数据仅使用一次。随着互联网行业的蓬勃发展，数据变得越来越“廉价”。很多应用有实时的，不间断的训练数据产生。在线学习（Online Learning）算法就是充分利用实时数据的一个训练算法。Online GD 于 mini-batch GD/SGD 的区别在于，所有训练数据只用一次，然后丢弃。这样做的好处是可以最终模型的变化趋势。

可以想象。标准梯度下降使用所有数据，还要迭代多次。如果有 10 万条数据，迭代 10 次。算法要计算 100 万次。Online ( $m=1$ ) 每次只使用一条数据。迭代 10 万次，也才计算 10 万次。可见 online GD 的效率很高。然而随机梯度下降易受到噪声干扰，可能陷入局部最优。Minibatch GD 是两者的一个折中。

在 4.3 节我们对优化算法还有更多的讨论。

## 第四节：反向传播算法

2.2 节讲述的是训练一个感知机的算法。该算法并不适用于多层网络。Paul Werbos 在他 1974 年的论文中提出一个多层神经网络算法，称为反向传播算法。这里我们不讨论数学推导，只讲述算法的执行过程。多层网络中的某一层的输出是下一层的输入。以一个二层网络为例。

注：按照 Deep Learning 一书。反向传播只是一种计算梯度的方法，而实际的多层感知机的训练算法还是采用诸如前面讲的 SGD 等，优化算法。

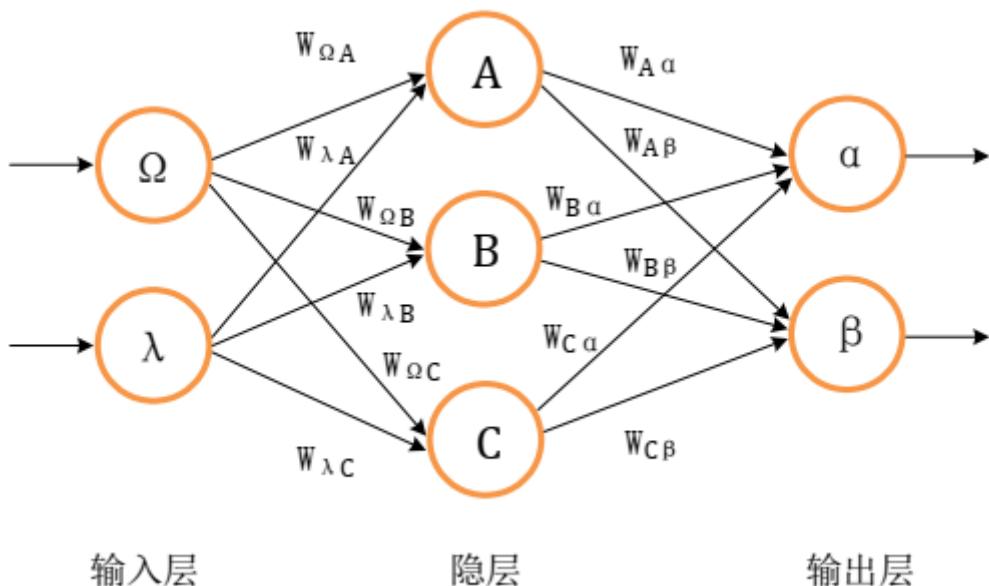


图 2.12：一个多层神经网络

反向传播算法工作步骤如下：

#### 1. 计算输出神经元的误差的梯度。

注意当激活函数是 sigmoid 函数时，采用下面的方法计算误差。

$$\delta_\alpha = \text{out}_\alpha(1 - \text{out}_\alpha)(\text{Target}_\alpha - \text{out}_\alpha)$$
$$\delta_\beta = \text{out}_\beta(1 - \text{out}_\beta)(\text{Target}_\beta - \text{out}_\beta)$$

如果激活函数是 binary step 函数则直接用  $\delta_\alpha = \text{Target}_\alpha - \text{out}_\alpha$

当采用梯度下降算法来优化参数，梯度等于误差函数对参数求偏导，例如

$$\frac{\partial E}{\partial W_{A\alpha}} = 2\delta_\alpha \text{out}_A$$

#### 2. 改变输出层权重

$\eta$  是学习率，权重的更新函数如下

$$W_{A\alpha}^+ = W_{A\alpha} + \eta \delta_\alpha \text{out}_A \quad W_{A\beta}^+ = W_{A\beta} + \eta \delta_\beta \text{out}_A$$
$$W_{B\alpha}^+ = W_{B\alpha} + \eta \delta_\alpha \text{out}_B \quad W_{B\beta}^+ = W_{B\beta} + \eta \delta_\beta \text{out}_B$$
$$W_{C\alpha}^+ = W_{C\alpha} + \eta \delta_\alpha \text{out}_C \quad W_{C\beta}^+ = W_{C\beta} + \eta \delta_\beta \text{out}_C$$

#### 3. 计算隐层的误差梯度（反向传播）

先计算隐层的误差

$$\delta_A = \text{out}_A(1 - \text{out}_A)(\delta_\alpha W_{A\alpha} + \delta_\beta W_{A\beta})$$
$$\delta_B = \text{out}_B(1 - \text{out}_B)(\delta_\alpha W_{B\alpha} + \delta_\beta W_{B\beta})$$
$$\delta_C = \text{out}_C(1 - \text{out}_C)(\delta_\alpha W_{C\alpha} + \delta_\beta W_{C\beta})$$

$\delta_\alpha W_{A\alpha} + \delta_\beta W_{A\beta}$  表示隐层的误差是由输出层的误差反向传播（乘上边的权重）得到的。

隐层误差的梯度

$$\delta_A \text{in}_\lambda$$

#### 4. 改变隐层的权重

$$W_{\lambda A}^+ = W_{\lambda A} + \eta \delta_A \text{in}_\lambda \quad W_{\Omega A}^+ = W_{\Omega A} + \eta \delta_A \text{in}_\Omega$$
$$W_{\lambda B}^+ = W_{\lambda B} + \eta \delta_B \text{in}_\lambda \quad W_{\Omega B}^+ = W_{\Omega B} + \eta \delta_B \text{in}_\Omega$$
$$W_{\lambda C}^+ = W_{\lambda C} + \eta \delta_C \text{in}_\lambda \quad W_{\Omega C}^+ = W_{\Omega C} + \eta \delta_C \text{in}_\Omega$$

$W^+$ 代表更新后的权重

神经网络里面的计算单元，最重要的激活函数是连续的、可微的。比如常用的 sigmoid 函数，它是连续可微的。这使得可以容易地进行梯度计算，如此就可以使用 BP 算法来训练。通过这样的算法神经网络已经取得了非常多的胜利。

神经网络的训练依赖梯度。**如果一个问题不可微分的、不可计算梯度的，就无法为它训练一个神经网络。这非常重要。**而且在 2006 年之前，也没有人知道如何训练深度超过 5 层的神经网络，不是因为计算设施不够强大，而是我们无法解决梯度消失的问题。Geoffery Hinton 等人做出了巨大的贡献，他们表明通过逐层训练（layer-by-layer precision）和预训练（pre-training）我们可以克服梯度消失的问题。介绍这些都是为了说明神经网络需要可微的函数、需要能够计算梯度，这是最根本最重要的。虽然如今有一些研究不可微的函数的，但还是需要转换成某种可微的。

# 第三章：Keras

TensorFlow 是实现 Deep Learning 的工具之一。按照官网介绍：TensorFlow 是一个使用数据流图（ data flow graph ）进行数值计算的开源的软件库。图中的节点表示操作（它可以是数学运算也可以是赋值等非数学运算操作），边表示沟通两个运算的多维数据阵列（ tensor, 又翻译做张量 ）。数据流图灵活的结构允许用户通过 API 将计算部署到一个或多个 CPU 或 GPU 。 TensorFlow 最初被 Google 机器智能研究组的 Google Brain Team 的科学家开发，用于机器学习和深度神经网络的研究。但该系统可以适用于非常宽广的领域。

当前 TensorFlow 的最新版本是 2.1 版。在 windows 上安装 TensorFlow , 参见  
[https://tensorflow.google.cn/install/install\\_windows](https://tensorflow.google.cn/install/install_windows)

Keras 是一个高层（ high-level ）的神经网络 API , 它可以运行在几个深度学习框架（ Tensorflow, CNTK, Theano ）之上。它被开发的动机是为了快速的开展深度学习的实验。因为 keras 相对于 tensorflow 要简单好用，本讲义将使用 keras 作为深度学习的工具。另外，因为我们使用的是构建在 tensorflow 上层的 keras ，我们还是从 tensorflow 的工作原理开始介绍。

注：Tensorflow 2.X 把 keras 纳入了自己的框架，对 keras 有一个自己的实施称为 tf.keras 。我们安装了 tensorflow 就安装了运行在 tensorflow 之上的 keras 。如果你单独安装 keras ( Francois Chollet 开发的 ) ，它的内容和本讲义介绍的 tf.keras 会有区别，称为 stand-alone keras 。 tf.keras



*At this time, we recommend that Keras users who use multi-backend Keras with the TensorFlow backend switch to tf.keras in TensorFlow 2.0. tf.keras is better maintained and has better integration with TensorFlow features (eager execution, distribution support and other).*

— [Keras Project Homepage](#), Accessed December 2019.

## Tips:

在 Anaconda 安装 TensorFlow 2.X。首先在一个 environment 的 terminal 下升级 pip 到最新版，安装 tensorflow2.x 需要最新版的 pip

pip install --upgrade pip

在线安装，则输入命令

pip install --upgrade tensorflow

也可以下载 tensorflow 的最新版本到本地，然后本地安装

<https://www.tensorflow.org/install/pip>

运行下面的代码来检查 tensorflow 是否安装成功

```
import tensorflow  
print(tensorflow.__version__)
```

## 第一节： keras 中的基本概念

### 3.1.1 数据流图

我们先将一下 Tensorflow 的工作流程。Keras 也是这样一个工作流程。TensorFlow 是一套编程系统或编程框架，用户可以将他的计算描述成图，称为数据流图或计算图。Keras 的模型构建和计算也可以用数据流图来理解。数据流图用一个有边和节点的有向图描述数学运算。节点实施运算。边描述了节点间的输入输出关系。边携带了 tensor 数据。TensorFlow 名称来自于 tensor 通过边的流动。节点被分配到可计算装置（CPU 或 GPU），可以异步、并行执行。.

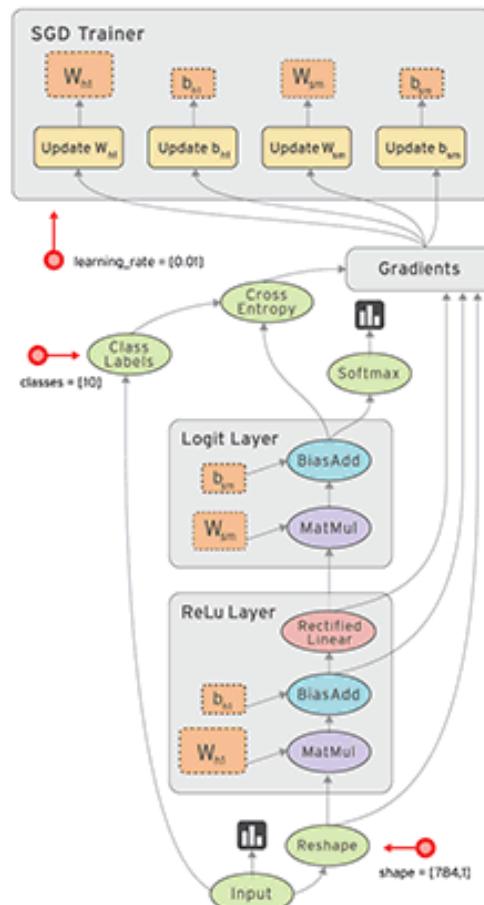


图 3.1 数据流图

### 3.1.1 tensor (张量)

keras 使用一个 tensor 数据结构描述所有数据。在构建的模型上传递的只能是 tensor。一个 tensor 有一个静态类型 shape。

TensorFlow 使用三类符号来描述 tensor 的维度。Rank, shape 和维度数。Shape 有的文章中文翻译为“形状”。我们还是使用英文。下表显示了三者的关系。

Rank	Shape	Dimension number	Example
0	[]	0-D	A 0-D tensor. A scalar.
1	[D0]	1-D	A 1-D tensor with shape [5].
2	[D0, D1]	2-D	A 2-D tensor with shape [3, 4].
3	[D0, D1, D2]	3-D	A 3-D tensor with shape [3, 4, 3].
n	[D0, D1, ..., Dn-1]	n-D	A tensor with shape [D0, D1, ..., Dn-1].

### 3.1.2 层 layer

在 keras 中建立神经网络的基础构件是层。这样的一个层是可以向函数一样调用。它的输入是 tensor，返回的结果也是 tensor。定义模型时，其参数是一个输入 tensor 和一个输出 tensor

```
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

# This returns a tensor
inputs1 = Input(shape=(784,))

# a Layer instance is callable on a tensor, and returns a tensor
hidden_1 = Dense(64, activation='relu')(inputs1)
hidden_2 = Dense(64, activation='relu')(hidden_1)
predictions = Dense(10, activation='softmax')(hidden_2)

# This creates a model that includes
# the Input Layer and three Dense Layers
model = Model(inputs=inputs1, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels) # starts training
```

层 layer 是构建深度学习模型的基本构件。Keras 中的层有很多种，可以构造复杂的深度模型。在附录的第 5 节有介绍。而具体的这些层的使用会在后面涉及到时介绍。一个层有属性

- `layer.input`
- `layer.output`

- `layer.input_shape`
- `layer.output_shape`

可以获得层的输入 tensor , 输出 tensor , 输入的 shape 和输出的 shape。

注 :

( 1 ) 创建第一个层时 , 可以首先创建一个输入层 , 然后创建一个隐层以输入层为输入

```
inputs = Input(shape=(784,))
hidden_1 = Dense(64, activation='relu')(inputs)
```

也可以创建第一个隐层时 , 规定 input\_shape 参数 , 而就不用创建输入层

```
hidden_1 = Dense(64, activation='relu', input_shape=(784,))
```

所有的类型的层都是 `tf.keras.layers.Layer` 的子类都继承了 `input_shape` 属性。

( 2 ) 模型的输入 tensor 的 shape 都是 `(batch_size, ...)` , 即第一个维度是 `batch_size`。但在每个层的 `input_shape` 参数中给出的元组中不需要写 `batch_size` 这个维度 ( axis )。因此 , `inputs = Input(shape=(28,))`

设定了输入 tensor 的一条数据记录的是一个一维向量 , 维度是 784。参数

`input_shape` 的值必须是一个元组。因此 , 对于一维向量它的 `shape` 必须写成一个元组 `( n, )` 这样的形式 , 此处的 `n` 是一个整数。而不能写成 `(n)`。因为 , 在 python 中 , `( n )` 被认为是一个 `int` 类型。只有 `( n, )` 才是元组。

如果模型的输入 tensor 是一个 `28*28` 的图片 , 那构建输入层可以是

```
inputs = Input(shape=(28, 28))
```

### 3.1.3 模型

Keras 有两种构建模型的方法 , 常用的方法是构建 Sequential model。该方法是创建一个 Sequential 模型对象 , 然后用 `add` 方法添加层。3.2.2 节将详细介绍。另一种是使用 Model 类。我们下面介绍一下第二种方法。

Model 类创建一个模型实例时 , 需要两个参数 , 一个是 `input` 的 tensor , 一个 `output` 的 tensor。我们可以创建一个 `Input` 层。然后一步步创建新的层 , 而创建新的层时的函数参数是以前一个层为输入参数。而每个层的对象的调用返回的是一个 tensor。因此我们就可以用如下的方式创建一个神经网络。

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense

a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(inputs=a, outputs=b)
```

这里的 Input 和 Dense 都是一个类型的层。

Tips:

```
考察一个模型 model 中的层的 shape  
for layer in model.layers:  
    print(layer.output_shape)
```

### 3.1.4 sample

一个样本是数据集中的一条数据记录。例如，喂给模型的一张图片，一段文本。

### 3.1.5 batch

Batch 是 N 个样本的集合。一个 batch 的样本被送入模型，被并行的处理，独立的计算。得到 N 个样本的输出。

### 3.1.6 epoch

训练模型的一个参数。在训练模型时的一个 epoch 是指在整个训练集上的一趟训练。epoch=N，即使用整个训练集 N 次来训练模型。训练模型时的“校验”是在每趟训练完成后，即每个 epoch，校验一次模型。

Keras 运行在每个 epoch 训练之后，加一个 callback。Callback 是一个函数集合，这些函数可以用于改变模型，例如，训练模型时的学习率，也可以考察模型内部。

## 第二节：初识 keras

tf.keras 是 tensorflow 实施的 keras。它将使得 tensorflow 更容易使用。首先，在你的第一个程序中引入 keras 包

```
import tensorflow as tf  
from tensorflow import keras
```

### 3.2.1 keras 中机器学习的流程

使用 keras 实施一个完整的机器学习的过程如下：

- ( 1 ) 构建模型。
- ( 2 ) 编译模型 ( compile model )。这里的编译的含义和高级编程语言中的“编译”的含义不一样，而是对模型的训练过程进行设置
- ( 3 ) 训练模型

( 4 ) 评估模型

( 5 ) 应用模型进行预测

我们下面以第二章第三节的预测食物价格的例子来分别介绍这五个过程：

### 3.2.2 构建模型

在 keras 中建立神经网络模型的基础是建立 layer。神经网络模型或深度模型可以理解为是多个 layer 的组合。最通常的模型是使用 tf.keras.Sequential 实施 layer 的堆叠 ( stack )。Sequential 模型之所以称为是“序列”的，是因为该模型顺序的将神经网络的层加入到模型中，且该模型线性的从输入到输出的工作。

预测食物价格的例子中，这是一个单层感知机，有三个输入值是每种食物的购买数量；输出层有一个值，是一次的花费。没有隐层。

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
model = Sequential()
model.add(Dense(1, input_shape=(3,)))
```

Dense 类构建 densely-connected NN layer ( 全连接的神经网络层 )。该函数实施了这样的操作

output = activation(dot(input, kernel) + bias)

kernel 是一个权重矩阵，bias 是偏置向量，activation 是激活函数。

```
tf.keras.layers.Dense(
    units, activation=None, use_bias=True,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros', kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None, kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

Dense 函数 ([https://www.tensorflow.org/api\\_docs/python/tf/keras/layers/Dense](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense)) 有下面的可选参数：

( 1 ) units, 该层的神经元数。

- ( 2 ) activation, 选用的激活函数。如果不设置激活函数，即表示是一个线性激活函数  $a(x)=x$
- ( 3 ) use\_bias, 一个布尔值，设定是否使用偏置。
- ( 4 ) kernel\_initializer: Initializer for the kernel weights matrix.
- ( 5 ) bias\_initializer: Initializer for the bias vector.
- ( 6 ) kernel\_regularizer: Regularizer function applied to the kernel weights matrix.
- ( 7 ) bias\_regularizer: Regularizer function applied to the bias vector.
- ( 8 ) activity\_regularizer: Regularizer function applied to the output of the layer (its "activation")..
- ( 9 ) kernel\_constraint: Constraint function applied to the kernel weights matrix.
- ( 10 ) bias\_constraint: Constraint function applied to t

### 3.2.2 编译模型

训练模型前，需要为模型选择一个损失函数。损失函数在机器学习的过程中，用于评价模型对训练数据的拟合。常用的损失函数有

[https://www.tensorflow.org/api\\_docs/python/tf/keras/losses](https://www.tensorflow.org/api_docs/python/tf/keras/losses)

通常是根据不同的任务选择不同的损失函数，例如，三个常用的损失函数

binary\_crossentropy : 用于二分类

sparse\_categorical\_crossentropy : 用于多类分类

mse (mean squared error) : 用于回归

然后选择一个优化算法对模型进行训练。常用的优化函数有

[https://www.tensorflow.org/api\\_docs/python/tf/keras/optimizers](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers)

从 API 的角度来看，这个阶段的操作是调用函数对模型进行设置，

```
from tensorflow.keras.optimizers import SGD
opt = SGD(learning_rate=0.001, momentum=0.9)
model.compile(optimizer=opt, loss='binary_crossentropy')
```

进一步，我们需要设置模型评价标准，例如，我们使用 accuracy 来评价模型，则上一条语句改成

```
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
```

这里的 metrics 参数给出的是一个 list 结构，即可以给出多种评价指标，例如，

```
['accuracy', 'mse']
```

### 3.2.3 训练模型

训练模型，首先需要对训练过程进行设置。例如：

The number of epoch: 在整个训练集上循环训练的次数

Bath size：一个批次训练数据大小

训练过程将在训练集上，使用优化算法，最小化损失函数。

```
model.fit(X, y, epochs=100, batch_size=32, verbose=0)
```

X 是训练集，y 是训练数据的目标值。默认的训练中间过程的结果会被显示在控制台，如果不显示设置 verbose=0。Verbose=1 会显示训练过程的进度条，=2 会在每趟训练后显示模型的评估。

```
fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None,
validation_split=0.0, validation_data=None, shuffle=True, class_weight=None,
sample_weight=None, initial_epoch=0, steps_per_epoch=None, validation_steps=None,
validation_freq=1, max_queue_size=10, workers=1, use_multiprocessing=False)
```

更多的参数设置参见 API<https://keras.io/models/model/>

### 3.2.4 评估模型

评估即用一部分数据集（未出现在训练集中）来评估模型的训练。

```
loss = model.evaluate(X2, y2, verbose=0)
```

evaluate 函数返回损失值和 metric 值

### 3.2.5 预测

训练好模型后就可以用模型对新的数据进行预测

```
res = model.predict(X)
```

该模型返回对 X 的预测结果，是 numpy 的 ndarray 数据结构。

下面的代码是用 tf.keras 实现的第二章第三节那个价格估计的程序

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

X=[[2,5,3],[1,4,7],[2,3,5],[3,6,9],[7,4,1]]
y=[850, 1050, 950, 1650, 1350]
```

```

X2=[[1,1,1]]
y2=[300]

model = Sequential()
model.add(Dense(1, input_shape=(3,)))

model.compile(optimizer='sgd', loss='mse', metrics=['mse'])
model.fit(X, y, epochs=500, batch_size=1, verbose=0)
loss = model.evaluate(X2, y2, verbose=0)
print(loss)
print(model.get_weights())

```

模型类下面的方法 `get_weights()` 可以显示神经网络的权重。

## 第三节： keras 练习

### 3.3.1 二分类多层感知机

本节我们使用 `tf.keras` 构建可以实现二分类的多层感知机模型。我们使用 Ionosphere 数据集。该数据集是一个电磁信号评价的数据集。有 351 条数据记录，前 34 个属性均为数值型。而第 35 个属性是目标值。

( 1 ) 装载库，使用 `pandas` 的 `read_csv` 来读取数据。

```

from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
path = 'D:/qjt/beike/deeplearning/2020/ionosphere.txt'
df = read_csv(path, header=None)

```

( 2 ) 然后对数据集进行预处理，划分数据集

```

X, y = df.values[:, :-1], df.values[:, -1]
X = X.astype('float32')

```

将数据转换成浮点值

```
y = LabelEncoder().fit_transform(y)
```

将目标列的数据进行编码，转换成 0,1 值

**注：`sklearn.preprocessing.LabelEncoder`**

将数据集中的目标列编码成 0-n\_class-1 的数值。

特别强调是对 y 进行编码，不是 x

sklearn.preprocessing.OrdinalEncoder

对类别属性编码，转化成(0 to n\_categories – 1)的整数

sklearn.preprocessing.OneHotEncoder

对类别属性进行 one-hot 编码

上面的三个类，在使用时应该先创建对象，然后调用 fit\_transform 方法进行转换

```
import sklearn as sk
```

```
le = sk.preprocessing.LabelEncoder()
```

```
le.fit_transform(["tokyo", "tokyo", "paris"])
```

进行数据集划分，33%作为测试集，剩下的是训练集

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
```

**注：sklearn.model\_selection** 类提供了很多种的数据集划分方法，例如

sklearn.model\_selection.KFold

k 折交叉确认的类

sklearn.model\_selection.train\_test\_split(\*arrays, \*\*options)

一个数据集划分方法 arrays 可以是多个长度相同的数据结构，如 pandas 的数据帧，array 等。

Options 可以有很多参数：

test\_size：是一个 0-1 的值，表示从当前数据集中划分百分之多少为测试集。

Shuffle 是一个布尔值，指示数据集是否先打乱，然后再划分数据集。

该函数返回的结果是划分好的数据集。例如，

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
```

### ( 3 ) 创建模型，训练模型

```
n_features = X_train.shape[1]
```

```
model = Sequential()
```

```
model.add(Dense(10, activation='relu', kernel_initializer='he_normal',
```

```
input_shape=(n_features,)))
```

创建输入层和第一个隐层。激活函数是 relu。参数 kernel\_initializer 设置当前一层的权重如何初始化，he\_normal 是按照 truncated 正太分布初始化。查看附录 A 中的描述。

创建第二个隐层

```
model.add(Dense(8, activation='relu', kernel_initializer='he_normal'))
```

创建输出层，只有一个神经元

```
model.add(Dense(1, activation='sigmoid'))
```

编译模型，设置优化函数是 adam, 损失函数是 binary\_crossentropy , 采用 accuracy 度量模型的性能

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

开始训练模型

```
model.fit(X_train, y_train, epochs=150, batch_size=32, verbose=0)
```

在测试集上评估模型

```
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print('Test Accuracy: %.3f % acc)
```

### 3.3.2 多类分类多层感知机

本节我们使用 iris 数据集做多类分类。该数据集描述了三种花。前四个属性是花的特征，第 5 列是目标类别。这是一个多类分类的问题。

#### ( 1 ) 装载类和数据集

```
from numpy import argmax
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
path = 'D:/qjt/beike/deeplearning/2020/iris.csv'
df = read_csv(path, header=None)
X, y = df.values[:, :-1], df.values[:, -1]
X = X.astype('float32')
y = LabelEncoder().fit_transform(y)
```

labelEncoder 函数的解释见上一节

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.33, shuffle=True)
n_features = X_train.shape[1]
```

train\_test\_split 函数的解释见上一节。n-features 是当前模型输入向量的长度。

## ( 2 ) 建立模型

```
model = Sequential()  
model.add(Dense(10, activation='relu', kernel_initializer='he_normal',  
input_shape=(n_features,)))
```

创建输入层和第一个隐层 ( 有 10 个神经元 )

```
model.add(Dense(8, activation='relu', kernel_initializer='he_normal'))
```

创建第二个隐层和输出层

```
model.add(Dense(3, activation='softmax'))  
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',  
metrics=['accuracy'])
```

使用的损失函数是 sparse\_categorical\_crossentropy。它是一个多类分类模型的损失函数。附录 A 中有详细解释。

训练模型，然后在测试集上评估模型

```
model.fit(X_train, y_train, epochs=150, batch_size=32, verbose=0)  
loss, acc = model.evaluate(X_test, y_test, verbose=0)  
print('Test Accuracy: %.3f % acc)
```

### 3.3.3 回归

本节采用 boston housing 数据集建立回归模型。该数据集是关于 Housing Values in Suburbs of Boston，每条记录是一栋房屋的数据，包含 14 列。第 14 列是 **Medv** : median value of owner-occupied homes。自住房中位数值，单位为\$1000.

下面的代码比较了 NN , SVR 和 LM 三个模型的预测结果

```
from pandas import read_csv  
from sklearn.model_selection import train_test_split  
from tensorflow.keras import Sequential  
from tensorflow.keras.layers import Dense  
# load the dataset  
path = 'D:/qjt/beike/deeplearning/2020/housing.txt'  
df = read_csv(path, header=None)  
# split into input and output columns  
X, y = df.values[:, :-1], df.values[:, -1]  
# split into train and test datasets  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,  
shuffle=True)  
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)  
# determine the number of input features  
n_features = X_train.shape[1]  
# define model  
model = Sequential()
```

```

model.add(Dense(10, activation='relu', kernel_initializer='he_normal',
input_shape=(n_features,)))
model.add(Dense(8, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(1))
# compile the model
model.compile(optimizer='adam', loss='mse')
# fit the model
model.fit(X_train, y_train, epochs=150, batch_size=32, verbose=0)
# evaluate the model
error = model.evaluate(X_test, y_test, verbose=0)
print('MSE: %.3f' % (error))

yhat=model.predict(X_test)

# svr
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error
svr_rbf = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=.1)
svr_rbf.fit(X_train, y_train)
y_pred=svr_rbf.predict(X_test)
err=mean_squared_error(y_test, y_pred)
print('MSE: %.3f' % (err))

# LR
from sklearn.linear_model import LinearRegression
reg = LinearRegression().fit(X_train, y_train)
y_pred2=reg.predict(X_test)
err=mean_squared_error(y_test, y_pred2)
print('MSE: %.3f' % (err))

#plot
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
fig = plt.figure()
ax1 = fig.add_subplot(1,3,1)
ax2 = fig.add_subplot(1,3,2)
ax3 = fig.add_subplot(1,3,3)

ax1.scatter(yhat,y_test)
transform = ax1.transAxes
line1 = mlines.Line2D([0, 1], [0, 1], color='red')
line1.set_transform(transform)
ax1.add_line(line1)
ax1.set_title('NN')
ax2.scatter(y_pred,y_test)
transform = ax2.transAxes
line2 = mlines.Line2D([0, 1], [0, 1], color='red')
line2.set_transform(transform)
ax2.add_line(line2)
ax2.set_title('SVR')
ax3.scatter(y_pred2,y_test)
transform = ax3.transAxes

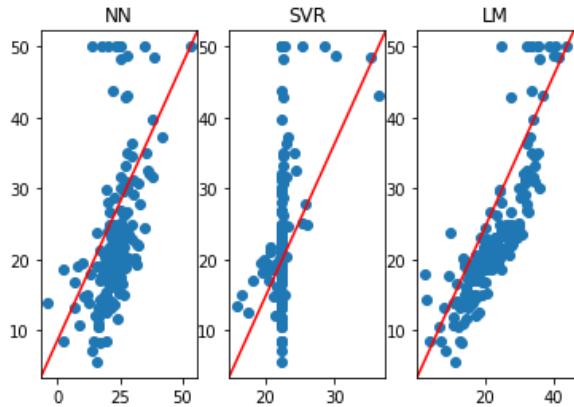
```

```

line3 = mlines.Line2D([0, 1], [0, 1], color='red')
line3.set_transform(transform)
ax3.add_line(line3)
ax3.set_title('LM')
plt.show()

```

我们也绘制了预测结果偏差的散点图。散点的分布如果越接近中线，表示预测的性能越好。需要说明的是，我们只是说明怎么使用模型。因为没有仔细的参数调优，本例不能说明哪个模型更优。



### 3.3.4 曲线拟合

该例子中，我们将构建具有一个隐层的神经网络，进行曲线拟合（或函数逼近）。在神经网络中，函数逼近通常隐层采用 sigmoid 激活函数，而输出层采用线性输出函数。曲线是一段正弦波曲线，并加上了随机噪声。

程序如下：

```

import matplotlib.pyplot as plt
import numpy as np
import math
import tensorflow.keras as keras
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Nadam

x=np.arange(0,6.4,0.1)
y=[math.sin(val) for val in x]
s = np.random.normal(0, 0.1, len(x))
y2=np.array([sum(x2) for x2 in zip(y,s)])

model=keras.Sequential()
model.add(Dense(10, activation='sigmoid', input_shape=(1,)))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(1))

opt = Nadam(learning_rate=0.02, beta_1=0.9, beta_2=0.999)
model.compile(optimizer=opt, loss='mse')
model.fit(x, y2, epochs=1000, batch_size=10, verbose=0)
loss=model.evaluate(x,y2,verbose=0)

```

```

print(loss)

# plot
plt.plot(x,y, 'k-', color = 'r', label="sin")
plt.plot(x,[0]*len(x))
plt.plot(x,y2, label="noise")
y3=model.predict(x)
plt.plot(x,y3,color = 'black', label="fitted")
plt.legend()

```

我们构建了一个三层的神经网络，每层均采用 sigmoid 激活函数。采用 Nadam 优化器（该优化算法细节见 4.7 节）。数据如下图所示。其中曲线 “sin” 是标准正弦曲线；曲线“noise”是加上了噪声的曲线；曲线“fitted”是神经网络拟合的曲线。我们可以看到神经网络可以克服噪声的干扰，较好的拟合了正弦曲线。对于数据中的噪声，几乎没有过拟合。

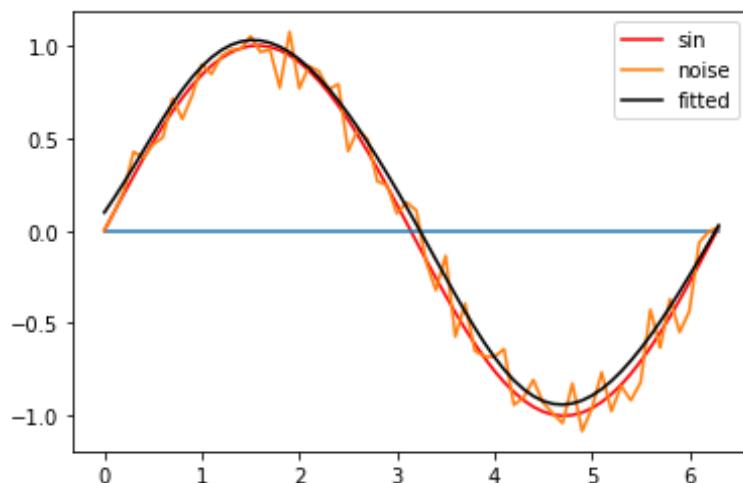


图 3.6 拟合的曲线

# 第四章：深度神经网络

深度前馈神经网络、深度卷积神经网络和深度循环神经网络是当前深度学习中最常用的深度神经网络模型。更有相当多的研究采用这些模型的组合。本章讨论这些深度神经网络的一些共性问题。

## 第一节：为什么需要深度神经网络？

下面内容来自周志华 IJCAI2019 年的大会报告。

深度神经网络和传统的神经网络的区别是什么？简单来说，就是深度神经网络的层数比传统神经网络会多很多。在 2012 年深度学习、卷积神经网络刚刚开始受到大家重视的时候，那时候 ImageNet 竞赛的冠军是用了 8 层的神经网络。那么到了 2015 年是用了 152 层，到了 2016 年是 1207 层。如今，数千层深的网络非常常见。这是个非常庞大非常巨大的系统，把这么一个系统训练出来难度非常大。

### 1. 为什么深度模型是有效的？

为什么深的模型要比浅的模型表现好那么多？到今天为止，学术界都还没有统一的看法。周志华从模型的复杂度的角度来讨论。

一个机器学习模型，它的复杂度实际上和它的容量有关，而容量又跟它的学习能力有关。所以就是说学习能力和复杂度是有关的。机器学习界早就知道，如果我们能够增强一个学习模型的复杂度，那么它的学习能力能够提升。

注：

一个模型的容量（capacity）是指模型拟合一个宽范围的函数的能力。模型的容量低，它很可能会欠拟合（underfit），如果模型的容量大它更容易过拟合（overfit）

对神经网络这样的模型来说，提高模型复杂的有两种方法：一是把模型变深，另一种是把模型变宽。如果从提升复杂度的角度，变深是会更有效的。当变宽的时候，只不过是增加了一些计算单元，在变深的时候不仅增加了个数，其实还增加了模型提取数据特征的能力。所以从这个角度来说，应该尝试去把它变深。

如此，大家可能会问，既然早就知道要建立更深的模型？为什么现在才开始做？这就涉及到另外一个问题，把机器学习的学习能力变强了，这其实未必是一件好事。因为机器学习一直在斗争的一个问题就是过拟合。也即，给定一个数据集，机器学习的目的要能够学习数据集里面的一般规律，能够用来预测未来的事情。但是有时候可能把这个数据本身的一些独特特性（或者说是噪声）学出来了，而不是一般规律。错误地把它当成一般规律来用的时候，会犯巨大的错误。这种现象就是所谓的过拟合，就是因为模型的学习能力太强了。所以以往通常不太愿意用太复杂的模型。

那现在我们为什么可以用很复杂的模型？是因为现在设计了许多方法来对付过拟合，比如神经网络有 dropout、early-stop 等。但有一个因素非常简单、非常有效，那就是用很大的数据。比如说手上如果只有 3000 个数据，那学出来的特性一般不太可能是一般规律，但是如果有 3000 万、30 亿的数据，那这个数据里面的特性可能本身就已经是一般规律。所以使用大的数据是缓解过拟合的一个关键的途径。第二，今天有了很多很强大的计算设备，这使得能够使用大规模数据训练模型。第三，通过这个领域很多学者的努力，有了大量的训练这样复杂模型的技巧、算法，这使得我们使用复杂模型成为可能。总结一下就是：第一我们有了更大的数据；第二我们有强力的计算设备；第三我们有很多有效的训练技巧。这导致我们可以用高复杂度的模型，而深度神经网络恰恰就是一种很便于实现的高复杂度模型。

## 2. 深度神经网络的有效性

如果从复杂度这个角度去解释的话，我们没法说清楚为什么扁平的（flat），或者宽的网络做不到深度神经网络的性能？实际上我们把网络变宽，虽然它的效率不是那么高，但是它同样也能起到增加复杂度的能力。

实际上只要有一个隐层，加无限多的神经元进去，它的复杂度也会变得很大。但是这样的模型在应用里面怎么试都发现它不如深度神经网络好。所以从复杂度的角度可能很难回答这个问题，我们需要一点更深入的思考。所以我们要问这么一个问题：深度神经网络里面最本质的东西到底是什么？

今天我们的回答是，本质是表征学习的能力(Representation Learning)。这已经成为了学术界的新的共识，甚至有了专门的会议 ICLR。以往我们用机器学习解决一个问题的时候，首先我们拿到一个数据，比如说这个数据对象是个图像，然后我们就用很多特征把它描述出来，比如说颜色、纹理等等。这些特征都是我们人类专家通过手工来设计的，表达出来之后我们再去进行学习。而今天我们有了深度学习之后，现在不再需要手工去设计特征了。你把数据从一端扔进去，结果从另外一端就出来了，中间所有的特征完全可以通过学习自己来解决。所以这就是我们所谓的特征学习，或者说表征学习。我们都认可这和以往的机器学习技术相比可以说是一个很大的进步，这一点非常重要。我们不再需要依赖人类专家去设计特征了。

这个过程中的关键点是什么呢？是逐层计算（layer-by-layer processing）。

这就回答了为什么增加深度比增加宽度有效。

当拿到一个图像的时候，如果把神经网络看作很多层，首先它在最底层，好像我们看到的是一些像素这样的东西。当一层一层处理的时候，慢慢的可能有边缘，再到高的层上可能有轮廓，甚至对象的部件等等。当然这实际上只是个示意，在真正的神经网络模型里面不见得会有这么清楚的分层。但是总体上当逐渐往高的层走，它确实是不断在对对象进行抽象。我们现在认为这好像是深度学习为什么成功的关键因素之一。因为扁平神经网络能做很多深层神经网络能做的事，但是有一点它是做不到的。当它是扁平的时候，它就没有进行这样的一个深度的加工。所以深度的逐层抽象这件事情，可能是很浅层神经网络和深层神经网络之间的关键区别。当然了，这也是一种猜测，我们目前还无法从数学上证明。

**逐层计算**在机器学习里面也不是新东西。比如说决策树就是一种逐层处理，这是非常典型的。决策树模型已经有五六十年的历史了，但是它为什么做不到深度神经网络这么好呢？答案很简单。首先它的复杂度不够，决策树的深度，如果我们只考虑离散特征的话，它最深的深度不会超过特征的个数，所以它的模型复杂度是有限的；而在神经网络中，当我们想要增加模型复杂度的时候，我们增加任意数目的层，没有任何的限制。第二，也是更重要的，在整个决策树的学习过程中，它内部没有进行特征的变换，从第一层到最后一层始终是在同一个原始特征空间里面进行的。特征变换这一点非常重要。相信这两点对于深度神经网络是非常重要的。

而当我们考虑到这两件事情的时候，我们就会发现，其实深度模型是一个非常自然的选择。有了这样的模型，我们很容易就可以做上面两件事。但是当我们选择用这么一个深度模型的时候，我们就会有很多问题，它容易过拟合，所以我们要用大数据；它很难训练，我们要有很多训练的技巧；这个系统的计算开销非常大，所以我们要有非常强有力的计算的设备，比如 GPU 等等。

实际上所有这些东西是因为我们选用了深度模型之后产生的一个结果，它们不是我们用深度学习的原因。所以这和以往的思考不太一样，以往我们认为有了这些东西，导致我们用深度模型。其实现在我们觉得这个因果关系恰恰是反过来，因为我们要用它，所以我们才会考虑上面这些东西。这曾经是使用浅层网络的原因，如今也可以是使用很深的网络的原因。

另外还有一点我们要注意的，当我们有很大的训练数据的时候，这就要求我们必须要有很复杂的模型。否则假设我们用一个线性模型的话，给你 2000 万样本还是 2 亿的样本，其实对它没有太大区别。它已经学不进去了。而我们有了充分的复杂度，恰恰它又给我们使用深度模型加了一分。所以正是因为这几个原因，我们才觉得这是深度

模型里面最关键的事情。（这个和另一个流行的观点不一致：当数据量足够大时，什么模型的性能都是差不多的）

这是我们现在的一个认识：**第一，我们要有逐层的处理；第二，我们要有特征的内部变换；第三，我们要有足够的模型复杂度（我认为，现在有够大的数据，和相应的学习算法技巧）。**这三件事情是我们认为深度神经网络为什么能够成功的比较关键的原因。或者说，这是我们给出的一个猜测。（选自周志华在IJCAI2019的报告）

### 3. 深度神经网络是万能的吗？

虽然在今天深度神经网络已经这么的流行，这么的成功，但是其实我们可以看到在很多的任务上，性能最好的不见得完全是深度神经网络。比如说 Kaggle 上面的很多竞赛有各种各样的真实问题，有买机票的，有订旅馆的，有做各种的商品推荐等等，还有一些来自企业的真实的工业问题，他们只考虑模型的表现，我们就可以看到在很多任务上的胜利者并不是神经网络，它往往是像随机森林，像 xgboost 等等这样的模型。深度神经网络获胜的任务，往往就是在图像、视频、声音这几类典型任务上，都是连续的数值建模问题。**而在别的凡是涉及到混合建模、离散建模、符号建模这样的任务上，其实深度神经网络的性能可能比其他模型还要差一些。**这也就是我们说的「没有免费的午餐定理」，已经有数学证明，一个模型不可能在所有任务中都得到最好的表现。

## 第二节：梯度消失、梯度爆炸与解决方法

### 4.1.1 梯度消失

传统的机器学习需要模型开发者非常清晰的了解应该从数据集抽取什么样的特征。例如，在做文本情感分析时，需要计算词频，词的极性或词的情感倾向等工作，**即特征工程在传统的机器学习中很重要。**而对于神经网络来说，把数据喂给网络，网络可以自动的抽取特征。神经网络在层次化非线性特征抽取上的表现出了优秀的能力。为了抽取更多的语义特征，以得到更好的模型性能，很多研究尝试加深模型的深度。然而超过三层的神经网络，我们称之为深度神经网络，在训练时会有一个**梯度消失**的问题（Vanishing gradient problem）。

在机器学习中，梯度消失问题是一种在使用梯度下降法和反向传播训练人工神经网络时出现的难题。在这类训练方法的每个迭代中，神经网络权重的更新值与误差函数梯度成比例，然而在某些情况下，梯度值会几乎消失，使得权重无法得到有效更新，甚至神经网络可能完全无法继续训练。对第  $L$  层权重更新时计算的梯度

$$\frac{\partial C}{\partial W_l} \propto f'(z^{(l)})f'(z^{(l+1)})f'(z^{(l+2)}) \dots$$

$f'(z^{(l)})$  是第  $L$  层激活函数的导数。当所有的  $f'$  小于 1 时，随着网络层数的增加，梯度  $\frac{\partial C}{\partial W_l}$  也远远小于 1，越深的层，权重几乎就不会被更新，这就是梯度消失。梯度消失问题在深度前馈神经网络和循环神经网络中表现明显。

举例来说，传统的激活函数，如双曲正切函数  $\tanh$  的梯度值在  $(0, 1)$  范围内，而反向传播通过链式法则来计算梯度。这种做法计算前一层的梯度时，相当于将  $n$  个这样小的数字相乘，这就使梯度（误差信号）随  $n$  呈指数下降，导致前面的层训练非常缓慢。图 4.1 中，蓝线是 sigmoid 函数，黄线是 sigmoid 函数的导数。从图 4.1 可以观察到，sigmoid 函数的输入值太大或太小，导数值都很小 ( $<<1$ )。当网络的权重值初始化很差，即在太大的正、负值之间，sigmoid 激活函数会很明显表现出梯度消失问题。

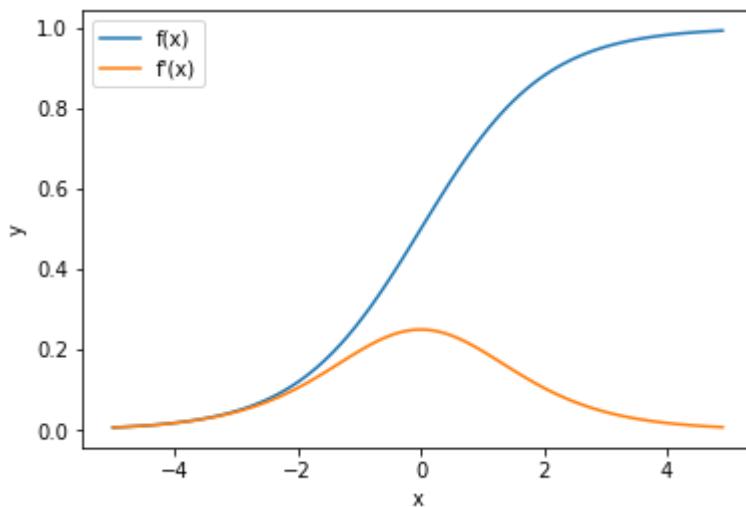
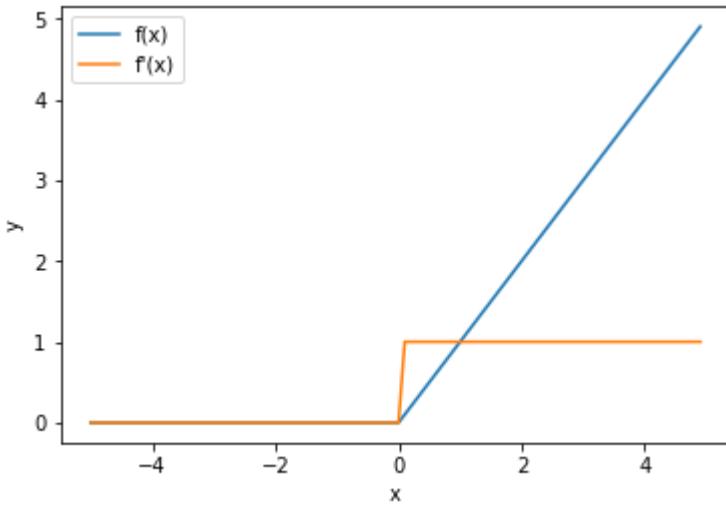


图 4.1 Sigmoid 函数和它的梯度

然而，即使权重的初始值选择的比较好，但随着层数的加深，这一问题仍旧在 sigmoid 激活函数中表现明显。而对于 ReLU 激活函数。当输入大于 0 时，它的导数是 1。否则是 0.



因此，梯度消失问题的解决方案包括：

- (1) 隐层使用 ReLu 激活函数，替代 sigmoid 和 tanh。
- (2) 在 RNN 中，使用 LSTM 或 GRU cell
- (3) 注意权重的初始化。权重采用随机初始化，值不要太大。偏置可以初始化为 0。

#### 4.1.2 梯度爆炸

采用梯度下降算法训练网络时，网络权重更新应该沿着合适的方向和量进行更新。误差梯度就是这个更新的方向和量。**梯度爆炸** ( Exploding Gradients ) 是指大的误差梯度的累积导致训练网络时权重更新值过大，网络训练时不稳定，网络不能从训练数据学习。极端情况下，导致 NaN 的权重值。

怎样判断是否你的模型出现了梯度爆炸。下面三个特征指示可能存在梯度爆炸：

- (1) 训练时，模型不能收敛，例如，损失函数值很高
- (2) 训练时，模型不稳定，损失函数变化很大
- (3) 得到了 NaN 的损失函数值

下面这些信息显示梯度爆炸的存在：

- (1) 模型的权重值很快的成为很大
- (2) 模型的权重值很变成 NaN

(3) 训练时，每个层的每个节点的误差梯度值都是大于 1.0

解决梯度爆炸，包括下面的策略

(1) 重新设计网络结构。减少网络的深度。也可以在训练网络时 batch\_size 设置比较小。在 RNN 中可以使用 truncated Backpropagation through time。

(2) 使用修正的线性激活函数 (Rectified Linear Activation) Relu。在深度前馈神经网络中，sigmoid 和 tanh 激活函数会导致梯度爆炸的产生。使用 ReLu 激活函数可以减小产生梯度爆炸的风险。在隐层采用 ReLu 激活函数是最佳策略。

(3) 在 RNN 中采用 LSTM cell 或者有 Gate 结构的 Cell 可以减少梯度爆炸。

(4) 使用梯度剪裁 (Gradient Clipping)。在深度前馈神经网络中如果训练模型的 batch\_size 过大，在 LSTM 中如果输入序列很长，都会发生梯度爆炸。如果采用了上面的策略后仍旧发生梯度爆炸，可以在网络训练时限制梯度的大小到一个阈值，这称为 Gradient Clipping。在 TensorFlow 中，在进行优化算法时，不采用 minimize 函数，而是拆开成 compute\_gradients 和 apply\_gradients 两个方法，在两个方法之间应用 Gradient Clipping 即可（使用 tf.clip\_by\_value 函数）。例如

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
gvs = optimizer.compute_gradients(cost)
capped_gvs = [(tf.clip_by_value(grad, -1., 1.), var) for grad, var in gvs]
train_op = optimizer.apply_gradients(capped_gvs)
```

(5) 使用权重正则化 (Weight Regularization)。即将网络的权重值作为正则化项加入到损失函数，可以采用 L1 (权重绝对值) 或 L2 (权重平方值) 正则化。一个例子，在 keras 中构建层时，都有权重、偏置和激活函数正则项的选项。

```
kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
```

(6) 减小优化函数的学习率

可以参考一篇论文详细了解梯度爆炸 <https://arxiv.org/abs/1712.05577>

### 第三节：构建深度前馈神经网络

深度前馈神经网络的结构包括输出层、隐层和输入层。网络的设计包括输出层选用什么样的激活函数，隐层的深度，每层的神经元的数目等。

#### 1. 结构

按照 Deep Learning 一书 6.4 节，很多任务选择一个隐层就可以很好的完成。但面对一些问题，这个隐层的神经元数需要非常大，有时会导致学习参数失败。越深的网络可以每层使用较少的神经元（单元 unit）。因此可以通过增加隐层，然后减小隐层的神经元数达到即可以很好的训练参数，又可以达到需要的模型性能。理想的网络结构必须通过试验不断调整参数，结构，监视校验集的误差去发现。

## 2. 输出层

输出层的设计是面向任务的，即将隐层的输出结果转换成任务需要的输出形式。

( 1 ) 最简单的输出层即线性输出层

$$\hat{y} = W^T h + b$$

( 2 ) sigmoid 函数的输出层用于完成产生二元变量  $y$  的任务，例如二分类任务。

$$\hat{y} = \sigma(W^T h + b)$$

$\sigma$  是 Logistics sigmoid 函数。我们也可以理解成输出层是对  $z = W^T h + b$  的计算结果应用了一个 sigmoid 激活函数，将计算值  $z$  转换到了一个概率值。

( 3 ) 当有一个具有  $n$  个可能值的离散变量，我们希望描述它的概率分布时，输出层采用 softmax 函数（或者说给输出层加上一个 softmax 激活函数）。Softmax 经常用于描述一个多类分类器的输出的概率分布。注：softmax 函数基本不用作隐层的激活函数。当一个线性输出层

$$z = W^T h + b$$

$z$  是一个向量。Softmax 函数将  $z$  规范化到一个概率分布

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

## 3. 隐层

这里讨论的隐层设计其实是在讨论隐层的激活函数选择。在学术界隐层的设计非常活跃，但没有一个明确的指导关于怎么设计深度网络的隐层。

( 1 ) ReLU。修正的线性单元 Rectified Linear Units

$$g(z) = \max\{0, z\}$$

如果没有明确的想法，Rectified linear units 是推荐的默认的隐层单元选择。第一节也介绍了在隐层使用 ReLu 激活函数是解决梯度消失和梯度爆炸有效的方法。ReLu 有很多变体，我们不详细讨论。

- ( 2 ) Sigmoid 和 tanh。在 ReLu 出现之前最常用的激活函数是 sigmoid 和 tanh。这两个激活函数是很相关的，因为 $\tanh(z) = 2\sigma(2z) - 1$ 。在一些必须使用 sigmoid 函数的场合，tanh 表现的更好。Sigmoid 在一些前馈神经网络之外，使用的更频繁。
- ( 3 ) 其他类型。RBF 函数，softplus, hard tanh。Softplus 可以看做是 ReLu 的平滑版。但 Deep Learning 一书推荐还是使用 ReLu。

dropout 是一个简单的防止深度网络过拟合的方法。它在神经网络的训练期间，随机选择一些神经元的输出。**Hinton 的论文中，建议 dropout 层可以放在除了输出层的任何全连接层后，选择概率是 0.5**。5.4 节给出了详细的介绍。

附录的第二节给出了 keras 提供的各种激活函数。激活函数的使用可以是在创建层时的一个参数选择，也可以是创建一个新层的方式加在一个层的后面。

## 4. 正则化

在 Deep learning 一书中将正则化 ( regularization ) 定义为所有的以减小测试误差为目的的策略。这些策略有可能会增加训练误差。这些策略包括：对参数的范数惩罚 ( parameter norm penalty ) 等等。我们下面就讨论三种在损失函数中加入正则项的方法。这也是在 keras 的 layer 类中提供的选项。见附录的第五节对 keras 提供的层的介绍

### ( 1 ) 权重正则项

在损失函数中加入权重正则项，可以限制模型的 capacity。L1 正则项和 L2 正则项两种。L1 正则项是所有参数的绝对值的和，L2 正则项是所有参数的平方和。和 L2 比较，L1 更趋于导致稀疏性，即更多的参数取值为零。

### ( 2 ) 偏置正则项

在 Deep learning 一书 P23 中提到神经网络不需要对偏置进行正则化。不过 keras 提供了偏置正则化的选项。

### ( 3 ) 激活函数正则项 Activation Regularization

与神经网络中“权重应该保持一定小的值”一样，大的激活函数输出可以导致过拟合。激活函数正则项是把神经网络的一个层的输出作为正则项加入到损失函数。L1 正则项就是把一层所有输出的绝对值求和，L2 正则项是把一层所有输出的平方和求和。

从 sparse coding 理解正则化：

在对数据特征抽取或编码时有两种方式：（1）把数据从高维空间映射到低维空间。它可以用在数据压缩、可视化的任务；（2）把数据映射到更高维的空间，如支持向量机。在神经网络中，希望隐层比输入层的维度更大，以更有能力捕获输入数据的特征。但大了以后会面临过拟合的问题。其中如果学习到的向量的值比较大，就很有可能指示过拟合了。因此，希望隐层的输出小且稀疏（有很多 0 值或接近 0 的值）。此时采取的方法是在损失函数中增加对权重的正则项和增加激活函数正则项。

Keras 中有个 regularizer 类。它提供了 L1，L2，L1+L2 三种正则化选项。在创建一个层时，我们可以考虑给该层加入正则项

```
from tensorflow.keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01),
                activity_regularizer=regularizers.l1(0.01)))
```

## 第四节：训练深度神经网络

在 2.3 节我们已经初步讨论了使用 SGD 训练一个感知机。这一节我们给出更详细的分析。

tips :

凸函数 convex ( 凹函数 concave ) 和非凸函数 ( 非凹函数 )

一个凸函数是指，它的二阶导数总是非负的。因此函数有一个全局最小值。凹函数就是说一个函数的二阶导数总是非正的，它有个全局最大值。凸或凹函数都可以容易的使用 SGD 得到它们的全局最小或最大值。而对于非凸和非凹的函数，SGD 可能收敛到一个局部极值，而非全局最值

训练一个深度神经网络，数据集划分成三个部分：训练集（training set），校验集（validation set），和测试集（test set）。三个部分互不重合。当多人进行比赛或我们在考察多个模型时，测试集作为最终评判的数据集合。模型用训练集训练，用校验集检验模型，然后调整模型的超参数。超参数是指那些不能在模型训练中自动学

习，需要人工设定的参数，如神经网络的层数，每层的节点数。测试集是在训练过程中看不到的数据。

构建好了深度神经网络，从任务出发我们可以构建损失函数。然后根据损失函数选择合适的优化算法来训练模型。在训练模型时有一些策略可以帮助减少 test error，这有可能以增加 training error 为代价（机器学习模型的泛化能力是我们追求的目标，即在测试集上有小的 test error），这些策略称为 regularization，翻译做正则化。

## 1. 损失函数 ( loss ) 或代价函数 ( cost )

常见的损失函数有 Zero-one Loss ( 0-1 损失 )，Perceptron Loss ( 感知损失 )，Hinge Loss ( 合页损失 )，Log Loss ( Log 损失 )，Cross Entropy ( 交叉熵 )，Square Loss ( 平方误差 )，Absolute Loss ( 绝对误差 )，Exponential Loss ( 指数误差 ) 等。深度学习中，损失函数的选择是和输出层紧密相关的。我们介绍几种深度学习中最常用的损失函数。

- ( 1 ) Mean Squared Error ( MSE ) **均方误差**是最基本的误差函数，即计算训练数据上每条数据的目标值和预测值的误差的平方，然后对所有数据的计算结果再求和，再开方。对于预测任务（线性输出层），通常选用 MSE 计算损失函数。但 MSE 对离群点敏感。
- ( 2 ) 交叉熵。对于多分类任务（softmax 输出层），通常采用 softmax 交叉熵来计算损失函数。对于二分类任务（sigmoid 输出层）可以使用 binary cross-entropy ( 二值交叉熵 )。交叉熵的计算公式

$$H_{y'}(y) = - \sum_i y'_i \log(y_i)$$

$y$  是目标分布， $y'$  是预测分布。 $y_i$  是目标分布中的一个元素， $y'_i$  是预测分布的一个元素。二值交叉熵计算公式如下

$$H_{y'}(y) = y' \log(y) + (1 - y') \log(1 - y)$$

$y$  是一个预测结果（二分类中是标量）。

- ( 3 ) Hinge loss。可用于最大间隔分类，SVM 就是采用 Hinge loss 作为目标函数。
- ```
loss = maximum(1 - y_true * y_pred, 0)
```
- 标签的  $y_{true}$  的取值应该是  $-1$  或  $1$ 。如果给的标签是  $(0, 1)$ ，会被自动转换成  $(-1, 1)$

Keras 提供的损失函数的详细介绍见附录第三节。

交叉熵举例：

一个单标签多类分类问题，假设有三个类 A、B、C。训练数据集给出的一条数据记录的标签是 B。即， $y$  是  
 $\Pr(\text{Class A}) \quad \Pr(\text{Class B}) \quad \Pr(\text{Class C})$   
 0.0        1.0        0.0  
 softmax 回归预测的每个类别的概率  $y_i$  是  
 $\Pr(\text{Class A}) \quad \Pr(\text{Class B}) \quad \Pr(\text{Class C})$   
 0.228      0.619      0.153  
 则计算的交叉熵是  
 $H = - (0.0 * \ln(0.228) + 1.0 * \ln(0.619) + 0.0 * \ln(0.153)) = 0.479$

## 2. 优化算法

本节介绍各种优化算法的原理。Keras 中对各种优化算法的实施见 4.7 节。

随机梯度下降 ( SGD ) 算法是最常用的深度神经网络训练算法。2.3 节已经介绍，这里不再重复。传统的 SGD 算法会有些问题。基于 SGD 的改进算法主要包括 momentum ( 翻译作动量 ) 和自适应学习率两大类。

### ( 1 ) 使用动量的 SGD 算法

传统的 SGD 算法学习会很慢。Momentum 算法被设计用于加速学习过程。Momentum 算法对之前计算的梯度累积一个指数衰减的移动平均，并继续沿着梯度的方向移动。

#### Stochastic Gradient Decent (SGD) with momentum

Require: Learning rate  $\epsilon$ , momentum parameter  $\alpha$ .

Require: Initial parameter  $\theta$ , initial velocity  $v$ .

while stopping criterion not met do

    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with corresponding targets  $\mathbf{y}^{(i)}$ .

    Compute gradient estimate:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

    Compute velocity update:  $v \leftarrow \alpha v - \epsilon g$

    Apply update:  $\theta \leftarrow \theta + v$

end while

速度  $v$  是负梯度的指数衰减平均。超参数  $\alpha \in [0,1)$  决定前面的梯度对指数衰减贡献有多大。 $L$  表示损失函数。

指数加权移动平均 ( Exponentially Weighted Moving Average , EWMA ) 是一种常用的序列数据处理方式。在时间  $t$ , 根据实际的观测值 ( 或量测值 ) 我们可以求取 EWMA  
 $( t ) : \text{EWMA}(t) = \rho Y(t) + (1-\rho) \text{EWMA}(t-1) \quad \text{for } t = 1, 2, \dots, n$

$\text{EWMA}(t)$  :  $t$  时刻的估计值

$Y(t)$  :  $t$  时间的量测值 .

从信号处理角度看，EWMA 可以看成是一个低通滤波器，通过控制  $\rho$  ( $0 < \rho < 1$ ) 值，剔除短期波动、保留长期发展趋势提供了信号的平滑形式。那么引入了动量的 SGD，剔除了梯度的波动，保持了梯度的变化趋势。

Nesterov momentum 是 momentum 算法的变体。它是结合 Nesterov 的加速梯度方法和 momentum 方法的随机梯度下降优化算法。

#### Stochastic Gradient Descent (SGD) with Nesterov momentum

**Require:** Learning rate  $\epsilon$ , momentum parameter  $\alpha$ .

**Require:** Initial parameter  $\theta$ , initial velocity  $v$ .

**while** stopping criterion not met **do**

    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with corresponding labels  $\mathbf{y}^{(i)}$ .

    Apply interim update:  $\tilde{\theta} \leftarrow \theta + \alpha v$

    Compute gradient (at interim point):  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$

    Compute velocity update:  $v \leftarrow \alpha v - \epsilon g$

    Apply update:  $\theta \leftarrow \tilde{\theta} + v$

**end while**

与传统 momentum 算法相比，nesterov momentum 是在应用了当前的 velocity 后计算梯度。这被看做是加了一个校正因子。

#### ( 2 ) 自适应学习率 ( adaptive learning rate )

学习率是深度学习最难设置的超参数，它可以显著地影响模型的性能。损失函数经常是对参数的某些方向非常敏感，而对其他方向又不敏感。Momentum 可以一定程度的缓解这一问题，但它引入了新的超参数 velocity  $v$ 。自适应学习率的一系列算法为解决这一问题，**为每个参数单独设置学习率**，且在学习的过程中可以自动修改学习率。

**AdaGrad** 算法为所有的模型参数单独的调整学习率。它在每次迭代中累积参数梯度的平方值。并用该值调整学习率

#### AdaGrad 算法

```

Require: Global learning rate  $\epsilon$ 
Require: Initial parameter  $\theta$ 
Require: Small constant  $\delta$ , perhaps  $10^{-7}$ , for numerical stability
    Initialize gradient accumulation variable  $r = \mathbf{0}$ 
    while stopping criterion not met do
        Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with
        corresponding targets  $\mathbf{y}^{(i)}$ .
        Compute gradient:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$ 
        Accumulate squared gradient:  $r \leftarrow r + \mathbf{g} \odot \mathbf{g}$ 
        Compute update:  $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot \mathbf{g}$ . (Division and square root applied
        element-wise)
        Apply update:  $\theta \leftarrow \theta + \Delta\theta$ 
    end while

```

运算符 $\odot$ 表示逐个元素相乘。 $\mathbf{g} \odot \mathbf{g}$ 表示每个参数的梯度平方运算。 $r \leftarrow r + \mathbf{g} \odot \mathbf{g}$ 表示在迭代的过程中累积每个参数的梯度的平方值。学习率 $\epsilon$ 除以 $\delta + \sqrt{r}$ 表示在每次迭代中动态调整学习率。可以看到，AdaGrad 累积梯度，因此学习率在迭代多次后会很小，以至于导致收敛到一个并不理想的 local minimum。因此，AdaGrad 在很多情况下不理想。RMSProp 的提出就改进了该问题。

**RMSProp 算法**对 AdaGrad 算法进行改进，它改变 AdaGrad 的梯度累积为指数加权移动平均。该算法在非凸环境下表现的更好。

```

RMSProp 算法
Require: Global learning rate  $\epsilon$ , decay rate  $\rho$ .
Require: Initial parameter  $\theta$ 
Require: Small constant  $\delta$ , usually  $10^{-6}$ , used to stabilize division by small
numbers.
    Initialize accumulation variables  $r = 0$ 
    while stopping criterion not met do
        Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with
        corresponding targets  $\mathbf{y}^{(i)}$ .
        Compute gradient:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$ 
        Accumulate squared gradient:  $r \leftarrow \rho r + (1 - \rho)\mathbf{g} \odot \mathbf{g}$ 
        Compute parameter update:  $\Delta\theta = -\frac{\epsilon}{\sqrt{\delta + r}} \odot \mathbf{g}$ . ( $\frac{1}{\sqrt{\delta + r}}$  applied element-wise)
        Apply update:  $\theta \leftarrow \theta + \Delta\theta$ 
    end while

```

下面是结合 RMSProp 和 momentum 产生了一个新的算法

|                                   |
|-----------------------------------|
| 结合 Nesterov momentum 的 RMSProp 算法 |
|-----------------------------------|

```

Require: Global learning rate  $\epsilon$ , decay rate  $\rho$ , momentum coefficient  $\alpha$ .
Require: Initial parameter  $\theta$ , initial velocity  $v$ .
    Initialize accumulation variable  $r = \mathbf{0}$ 
    while stopping criterion not met do
        Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with
        corresponding targets  $\mathbf{y}^{(i)}$ .
        Compute interim update:  $\tilde{\theta} \leftarrow \theta + \alpha v$ 
        Compute gradient:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$ 
        Accumulate gradient:  $r \leftarrow \rho r + (1 - \rho) \mathbf{g} \odot \mathbf{g}$ 
        Compute velocity update:  $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot \mathbf{g}$ . ( $\frac{1}{\sqrt{r}}$  applied element-wise)
        Apply update:  $\theta \leftarrow \theta + v$ 
    end while

```

结合 Nesterov momentum 的 RMSProp 算法已经被证明是深度学习中非常有效的优化算法。

Adam 是另外一种自适应学习率优化算法。Adam 是 Adaptive moments 的简写。可以将 Adam 看做是 RMSProp+momentum+新特性的组合。

### Adam 算法

```

Require: Step size  $\epsilon$  (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates,  $\rho_1$  and  $\rho_2$  in  $[0, 1)$ 
    (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant  $\delta$  used for numerical stabilization. (Suggested default:
 $10^{-8}$ )
Require: Initial parameters  $\theta$ 
    Initialize 1st and 2nd moment variables  $s = \mathbf{0}$ ,  $r = \mathbf{0}$ 
    Initialize time step  $t = 0$ 
    while stopping criterion not met do
        Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with
        corresponding targets  $\mathbf{y}^{(i)}$ .
        Compute gradient:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$ 
         $t \leftarrow t + 1$ 
        Update biased first moment estimate:  $s \leftarrow \rho_1 s + (1 - \rho_1) \mathbf{g}$ 
        Update biased second moment estimate:  $r \leftarrow \rho_2 r + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$ 
        Correct bias in first moment:  $\hat{s} \leftarrow \frac{s}{1 - \rho_1^t}$ 
        Correct bias in second moment:  $\hat{r} \leftarrow \frac{r}{1 - \rho_2^t}$ 
        Compute update:  $\Delta\theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$  (operations applied element-wise)
        Apply update:  $\theta \leftarrow \theta + \Delta\theta$ 
    end while

```

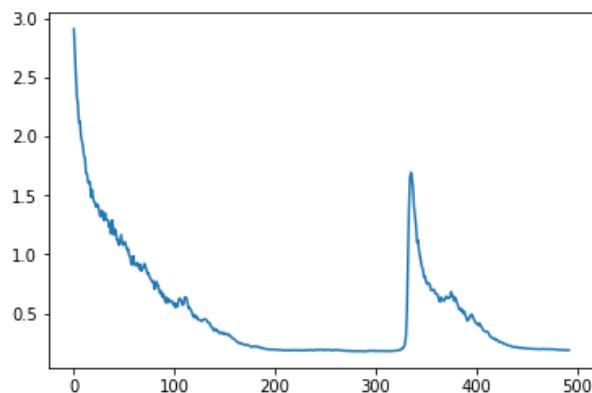
首先，Adam 中要为梯度计算一个一阶动量  $s \leftarrow \rho_1 s + (1 - \rho_1)g$  和一个二阶动量  $r \leftarrow \rho_2 r + (1 - \rho_2)g \odot g$ 。然后用修正的一、二阶动量去调整学习率  $\Delta\theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r} + \delta}}$

有评论，Adam 使用数据量大和参数多的模型。

**怎样选择合适的优化算法？**没有一个统一的看法。有论文对不同任务时不同算法的性能进行了比较。论文总结，自适应学习率的算法表现更好。本文前面介绍的算法是在深度学习任务中广泛采用的算法，选择哪个算法取决于用户对算法的了解，例如如何设置超参数。《Neural Network Method for NLP》一书的作者指出在他的研究中发现 Adam 算法在大的网络上很有效。

#### 注：使用 Adam 算法的一个问题

一阶动量  $s$  是梯度均值的指数移动平均，二阶动量  $r$  是梯度平方的指数移动平均。当训练一个长时间后，参数接近最优值后， $r$  比  $s$  减小的更快成为很小，此时梯度会反而增加甚至梯度爆炸。如下图所示损失函数在训练到第 300 多次后突然增加。



对付这个问题，要么换一个优化函数，要么在损失函数突变前结束训练。对于该图中的问题，我尝试使用 GradientDescentOptimizer，确实不会出现损失函数的突变。但，损失函数收敛到 1.4，不然 adam 优化算法可以收敛到 0.1。

### 3. 参数初始化

深度学习算法被参数的初始值影响。初始值甚至能影响算法是否收敛。对于偏置，通用的做法是设置初始偏置为 0。但对于权重则比较复杂。例如，如果全部初始化为 0，tanh 激活函数会产生为 0 的梯度；如果权重都一样，隐层单元将产生同样的梯度，它们的行为一致，将浪费模型的 capacity（模型的 capacity 是指模型拟合各种函数的能力，模型越复杂 capacity 越大）。参数初始化的启发规则有很多。下面列举出部分权重初始化的建议：

- (1) 所有的权重初始化从一个均匀分布 $[-b, b]$ 。就 $b$ 的选择有很多研究。普遍认为，一层的输入越多，权重值应越小。例如，有研究建议有 $m$ 个输入， $n$ 个输出的全连接层初始权重按照均匀分布 $W_{l,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$ 初始化。
- (2) 有研究使用一个增益因子(gain factor) $g$ ，仔细选择该增益因子成功训练了有1000层的网络。

#### 4. Early Stopping

当训练一个大 capacity 的模型，它对面对的任务很有可能过拟合。我们可以观察到一个现象，训练误差稳定的在减小，但校验集上的误差开始增加。如图 4.2 所示。

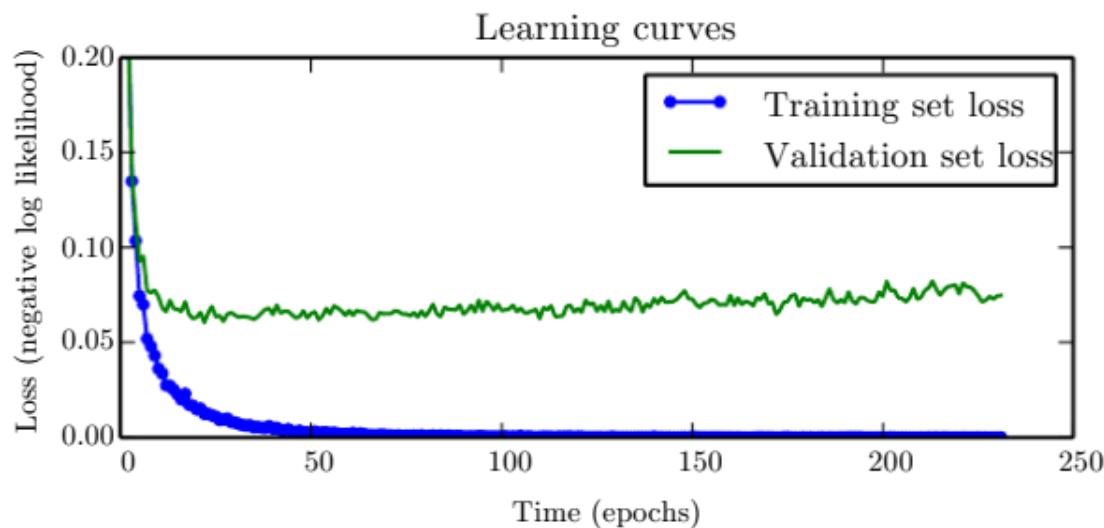


图 4.2 校验集误差的变化

我们希望能够回到具有最好 validation error 时间点的模型（很可能对应最好的 test error），采取的做法是，如果 validation error 在一个时间段内没有变得更大，则停止模型训练。这称为 early stopping。下面的 early stopping 算法选自 Lutz Prechelt 的“Early Stopping – But When?”

##### Early stopping 算法

- 将训练数据划分成训练集和校验集，例如 2:1 的比例。（有很多任务中只需划分训练集和测试集，如果没有校验集则从训练集中划分）
- 在训练集上训练，在每几趟训练后在校验集上评估误差，例如每 5 趟训练（每几趟后评估是考虑到了训练的效率问题，很多实践是每趟训练后都在校验集上评估）。（在 keras 中，训练模型时的 fit 函数提供了从训练集划分校验集的比例。而且，参数 verbose=1 或 2 时，每趟的训练都会输出在校验集上的评估结果）
- 一旦校验误差在持续了一段时间的高于最低校验误差，则停止训练。

#### 4. 保存的最低校验误差时的权重作为最终训练得到的网络权重。

tf.keras 有个 early\_stopping 的类: tf.keras.callbacks.EarlyStopping

可以在训练模型时，加入该类

```
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3)
# This callback will stop the training when there is no improvement in
# the validation loss for three consecutive epochs.
model.fit(data, labels, epochs=100, callbacks=[callback],
           validation_data=(val_data, val_labels))
```

#### 5. 扩大数据集 ( data augmentation )

要想模型有更好的泛化能力，就是在更多的数据上训练模型。在现实中，我们能获取的数据总是有限的，因此创建一些假数据把它们添加到数据集中也是常有的一种方法。对于分类任务这种方法可行。在图像处理的任务中，经常将图像进行旋转、变形等处理来 data augmentation。许多模型对数据集中类不平衡 ( class imbalance ) 问题很敏感。例如，用误差平方和做损失函数，在类不平衡问题很严重的数据集上训练分类型模型，多数类将主宰训练过程。对付不平衡数据集的方法有很多研究，最简单的方法是在少数类的数据上重复抽样，扩大到两类数据基本相等。

Tips:

每种优化器的 learning\_rate 的量级都不太一样。如梯度下降优化器 GradientDescentOptimizer 的学习率在 0.5 左右调整。AdamOptimizer 优化器的学习率的默认值是 0.001；RMSPropOptimizer 没给默认值，可以设为 0.001

#### 6. 批量规范化 ( Batch Normalization )

### 第五节：实例：手写数字识别

MNIST 是一个手写数字的数据集。它包含的图片如下图：

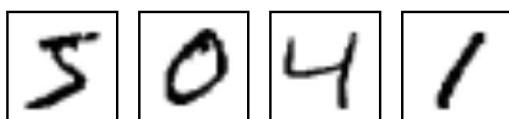


图 4.2 mnist 数据集中的图片示例

每个图片也被分配了一个标签，即图片对应的数字。在本节我们将建立一个预测模型，给定一张图片预测它对应的数字。本节不讨论如何训练一个性能最优的分类器，只是探讨如何用 keras 完成该工作。

Keras 内置了该数据集。获得的该数据集包含：60000 条训练数据，10000 条测试数据。一条 mnist 数据包含两个部分：手写数字图片和对应的标签。每张图片是一个 28\*28 像素的矩阵，见图 3.3。

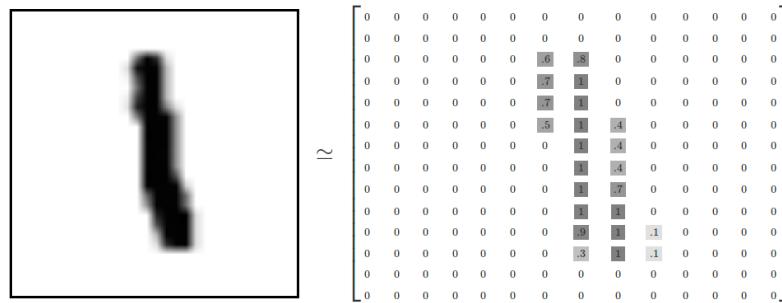


图 3.3 图片的矩阵描述

读入数据集。数据集中的数据是从 0-255 的整数，按照最大值规范化。

```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255
```

在使用神经网络处理图像时，因为神经网络的输入层是一个向量。因此，我们的模型需要将图像转化成  $28*28=784$  长度的向量。

我们建立的模型是一个 **Softmax Regression** 模型。每张 mnist 的图片对应 0-9 中的一个数字。预测模型应该对输入的图片给出对应每个数字的概率，例如，给出一张图片是 ‘9’ 的概率是 80%，是 ‘8’ 的概率是 5%。在神经网络的多分类任务中，输出层经常选用 softmax 激活函数，因为它可以给出输入对应每个类的概率。

在当前例子中，Softmax Regression 的网络描述如图 3.5：（此处假设，输入向量的维度是 3，输出的类别个数也是 3）

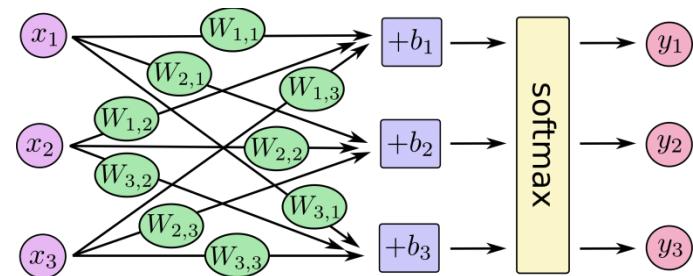


图 3.5 softmax regression 模型

Softmax 其实就是将一组数据求指数后规范化的一个操作，其数学描述是：

给定一组数据  $z=\{z_1, z_2, \dots, z_n\}$

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j \in n} \exp(z_j)}$$

上图的数学描述就是

$$y = \text{softmax}(Wx + b)$$

$W$  是权重矩阵， $b$  是偏置向量， $x$  是输入向量。

### 建立 softmax regression 模型

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(10),
    tf.keras.layers.Softmax()
])
```

这里构建模型的方法是，建立多个层的 list，然后作为类 Sequential 的构造方法的参数。

`tf.keras.layers.Flatten(input_shape=(28, 28))`

表示建立一个层，它把输入的一个  $\text{shape}=(28, 28)$  的 tensor 拉伸成一个  $28*28$  的向量。

`tf.keras.layers.Dense(10)`

建立一个神经元个数为 10 的输出层。紧接着的

`tf.keras.layers.Softmax()`

表示把前面的一个层进行 softmax 规范化。也即建立了一个 softmax 输出层

### 编译模型

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])
```

先建立损失函数。因为模型的输出是一个长度为类别数的向量，向量的每个元素反映了在每个类别上的一个评分，而测试集的标签数据是一个类别值（当前例子是 0-9）。因此适合用 SparseCategoricalCrossentropy 损失函数。另外，模型的输出是在每个类别上的分布，因此参数 `from_logits` 应该为 True

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

我们采用 Nadam 优化算法训练模型，评价指标为 accuracy

### 训练模型和评估模型

```
model.fit(x_train, y_train, epochs=10, batch_size=10)
model.evaluate(x_test, y_test, verbose=0)
```

下面的代码是建立一个两层神经网络的模型

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10)
])

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='Nadam',
              loss=loss_fn,
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=10, batch_size=10)
model.evaluate(x_test, y_test, verbose=0)
```

试一试删除 `tf.keras.layers.Dropout(0.2)` 这条语句对模型性能的影响。

练习：加入更多的层，进一步提高模型的性能

# 第五章：卷积神经网络

卷积神经网络（convolutional neural network，CNN or ConvNet）是一类前馈神经网络，是受生物学启发的多层感知机的变体。生物学家研究猫的视觉皮层发现视觉皮层上的细胞的排列很复杂。这些细胞对视域的一个小的子区域敏感，称作 receptive field 感受野。感受野指听觉系统，视觉系统和本体感觉系统的一些特质。比如在视觉神经系统中，一个神经元的感受野是指视网膜上的特定区域，只有这个区域内的刺激才能激活该神经元。多个子区域排列，覆盖整个视域。这些细胞相当于在输入空间上加入局部滤波器。非常适合展现自然图像的空间上的局部相关性。

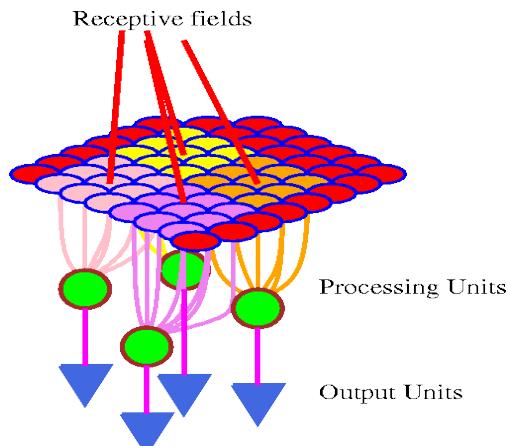


图 5.1：视觉系统上的感受野

卷积神经网络非常适合处理网格拓扑结构的数据，例如，时间序列数据，看做是 1-D 网格；图像数据，看做是 2-D 网格。卷积神经网络有非常的成功例子。

## 第一节：卷积

卷积是分析数学中一种重要的运算。在泛函分析中，它是通过两个函数  $f$  和  $g$  生成第三个函数的一种数学算子。我们这里只考虑离散序列的情况。一维卷积经常用在信号处理中。

例 1：

有两个离散序列：

$$x(n) = \begin{cases} 1, & 0 \leq n \leq 5 \\ 0, & \text{otherwise} \end{cases}$$

$$h(n) = \begin{cases} 1, & 0 \leq n \leq 2 \\ 0, & \text{otherwise} \end{cases}$$

进行卷积计算得到一个新的序列  $y(n)$

$$y(n) = \sum_{i=-\infty}^{\infty} x(i) \cdot h(n-i)$$

我们可以得到  $y$  的序列

$$y(0)=1, y(1)=2, y(2)=3, y(3)=3, y(4)=3, y(5)=3, y(6)=2, y(7)=1$$

$$\text{其余 } y(n)=0$$

我们也可以按照滤波器的方式来理解离散卷积。给定一个输入信号序列  $x_t, t=1, \dots, n$ , 和滤波器  $f_t, t=1, \dots, m$ , 一般情况下滤波器的长度  $m$  远小于信号序列长度  $n$ 。

卷积的输出为：

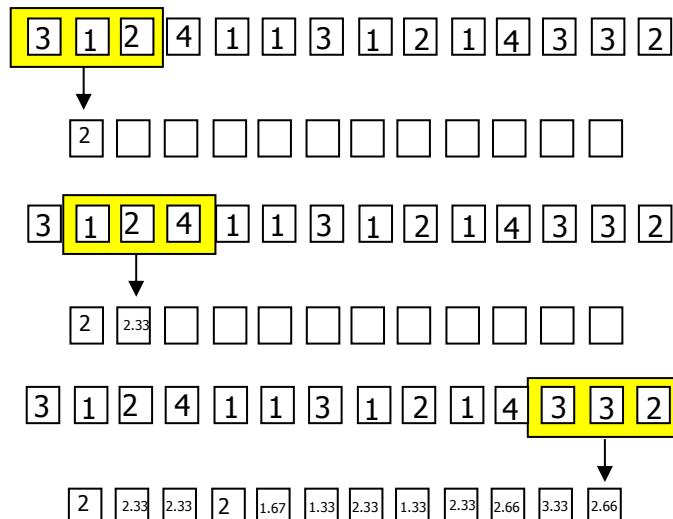
$$y_t = \sum_{k=1}^m f_k \cdot x_{t-k-1}$$

当滤波器  $f_t=1/m$  时, 卷积相对于信号序列的移动平均。即可以理解为对  $x$  序列上, 宽为  $m$  的子序列求平均。

例 2：有一个  $x$  序列

$$[3] [1] [2] [4] [1] [1] [3] [1] [2] [1] [4] [3] [3] [2]$$

滤波器为  $f_t=1/m, m=3$ 。则产生的  $y$  序列为



如果对于不在 $[1,n]$ 范围内的 $x_t$ 用零补齐 ( zero-padding ) ,  $y$ 序列输出的长度是 $n+m-1$ , 称为宽卷积。如果不补零, 输出序列长度是 $n-m+1$ , 称为窄卷积。除非特殊声明, 下面所说的卷积默认为窄卷积。

补充 :

设  $m$  是原始序列长度,  $n$  是滤波器的宽度,  $s$  是 stride。一个序列卷积操作后的长度的计算公式

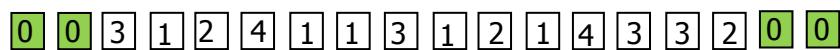
$$l = \left\lfloor \frac{m - n}{s} \right\rfloor + 1$$

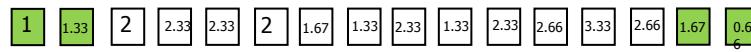
如果有补齐操作, 原始序列长度为  $m$ , 补齐后的长度是 $m + (n - 1) \times 2$

则有补齐的卷积操作的输出长度是

$$l = \left\lfloor \frac{m + n - 2}{s} \right\rfloor + 1$$

上述例子中, 例 1 是宽卷积, 例 2 是窄卷积。例 2 对应的宽卷积如下:





上面例子 2 是一个一维窄卷积。而在图像处理中经常用二维卷积。给定一个图像  $x_{ij}$ ,  $1 \leq i \leq M$ ,  $1 \leq j \leq N$ , 和滤波器  $f_{ij}$ ,  $1 \leq i \leq m$ ,  $1 \leq j \leq n$ , 一般  $m \ll M$ ,  $n \ll N$ 。卷积的输出为:

$$y_{ij} = \sum_{u=1}^m \sum_{v=1}^n f_{uv} \cdot x_{i-u+1, j-v+1}$$

在图像处理中, 常用均值滤波器, 就是当前位置的像素值设为滤波器窗口中所有像素的平均值, 也就是  $f_{uv} = \frac{1}{mn}$

二维卷积就在一个矩阵上面应用一个滤波器的过程。如图 5.2 所示

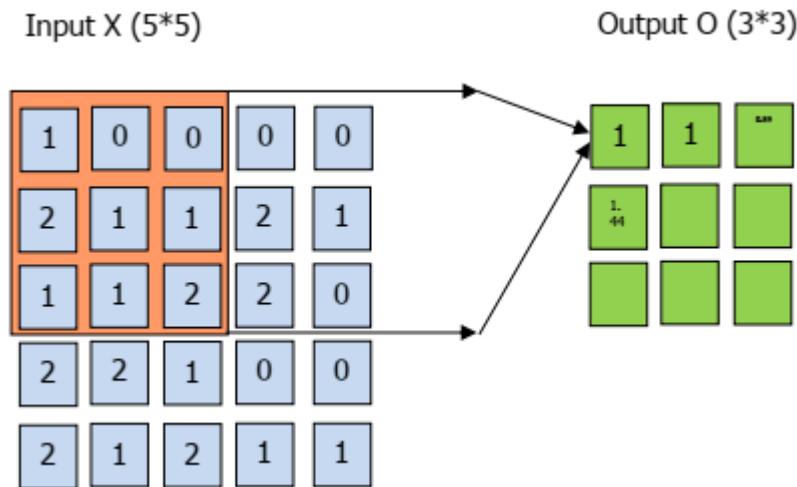


图 5.2 二维卷积

需要说明的是，上面例子中卷积的滤波器都是计算的窗口内元素的均值（移动平均）。下一节可以看到，滤波器也可以有别的运算。

## 第二节：卷积神经网络的结构

### 5.2.1 CNN 的特征

根据卷积滤波的思想我们构建卷积神经网络，它具有下面的特性：

#### 1. 局部连接

CNN 利用局部空间上的相关性，它强制在邻近层的神经元之间建立一个局部联通模式。即，在隐层  $m$  层的输入来自于  $m-1$  层神经元（单元）的一个子集，一个空间上邻近的单元子集。如图 5.3 所示。

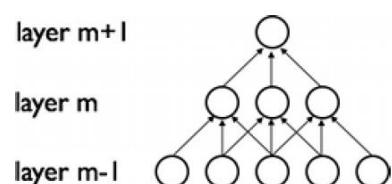


图 5.3：一个 CNN 示例

注：我们前面讲的前馈神经网络是全连接的。

将  $m-1$  层想象为视网膜，在  $m$  层的单元在视网膜上有宽度为 3 的感受野，因此仅仅连接到邻近的 3 个神经元（单元）。 $m+1$  层和其低层（ $m$  层）也有相似的连接性。我们说， $m+1$  层的单元相对于  $m$  层有宽度为 3 的感受野，但相对于输入层（ $m-1$ ）有宽度

为 5 的感受野。每个单元对感受野外的变化不响应。这样的体系结构确保‘滤波器’对空间上的局部输入模式做最强的响应。

然而，我们也可以看到，该结构上如果加的隐层越多导致‘滤波器’成为更加全局化，即对更大的像素空间（输入）进行响应。例如， $m+1$  隐层的单元可以对宽带为 5 的非线性特征进行编码（encode）。

## 2. 共享权重

换个角度，可以把 CNN 的滤波器理解为一组边的权重值。滤波器的大小（或组中，值的个数）与局部连接中连接到  $m$  层中一个神经元的  $m-1$  层的神经元个数相等。例如，图 5.3 中  $m-1$  层有邻近的三个神经元连接到  $m$  层的一个神经元，因此滤波器的大小为 3。CNN 中一个滤波器  $h$  给连接到隐层  $m$  每个神经元的边分配权重，因此边共享相同的权重。例如图 5.4 中， $m$  层有三个神经元， $m-1$  层的邻近三个神经元连接到  $m$  层的一个神经元  $v_i$  时，使用滤波器分配边权重；连接到  $m$  层神经元  $v_{i+1}$  的三个边也使用该滤波器分配边权重。进而形成特征映射（feature map）。CNN 中可以设置多个滤波器，进而一个隐层的输出是多个特征映射。

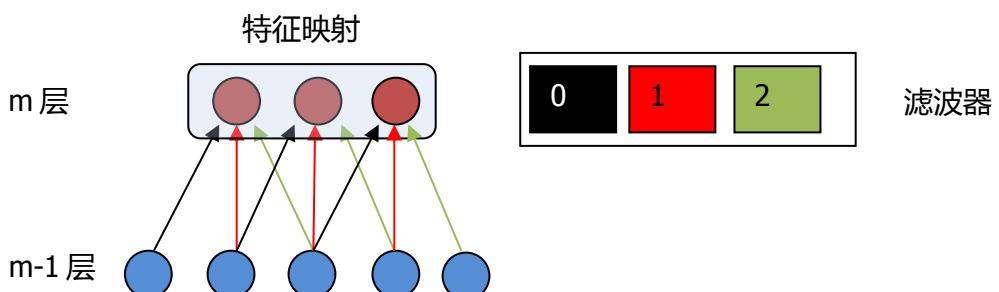


图 5.4：特征映射

图 5.4 中，有一个滤波器，滤波器大小为 3。因此  $m-1$  隐层每三个相邻的神经元连接到一个  $m$  层的神经元，边权重由滤波器分配（一组滤波器权重值用红、蓝、绿三色表示）。可以看到隐层  $m$  的三个单元构成一个 feature map。相同颜色的边共享权重。虽然边共享了权重，但只需做小的算法上的改动，仍可以用梯度下降的方法学习模型参数。一个共享权重的梯度是共享的参数的梯度和。权重共享可以提高模型训练的效率，因为相对的参数数量减少了。卷积操作就来自于共享权重，相当于对一个区域的单元做了算术运算。

我们再用刚才的一维卷积的例子来理解 CNN 的操作。

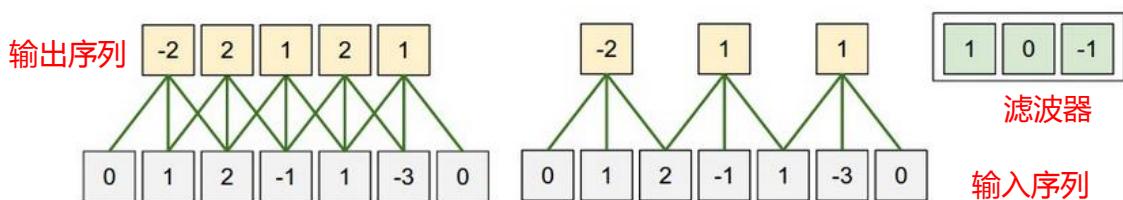


图 5.5 一维卷积

将输入序列理解为神经网络的输入层；滤波器实际上是边的权重。输出序列是神经网络的一层。滤波器实际上表现为了边的权重，即上述的共享边权重。得到的一个输出序列是一个特征映射 feature map。从图中可以看出，将上图看做是神经网络中的两层，卷积操作中需要学习的边权重实际上只有 3 个，即滤波器中的三个值。图 5.5 给了不同步长的两个例子，左边是步长为 1 时的卷积操作；右边是步长为 2 的卷积操作。

再举例：有两个滤波器，滤波器大小为 4。因此， $m-1$  层每 4 个相邻神经元连接到  $m$  层的一个神经元。如图 5.6 所示。可以看到应用多个滤波器后可以将原始的数据升维，图中从一维，升维到二维。这样就提供了更丰富的数据处理。我们把输入数据应用滤波器得到的输出称为特征映射（feature map）。

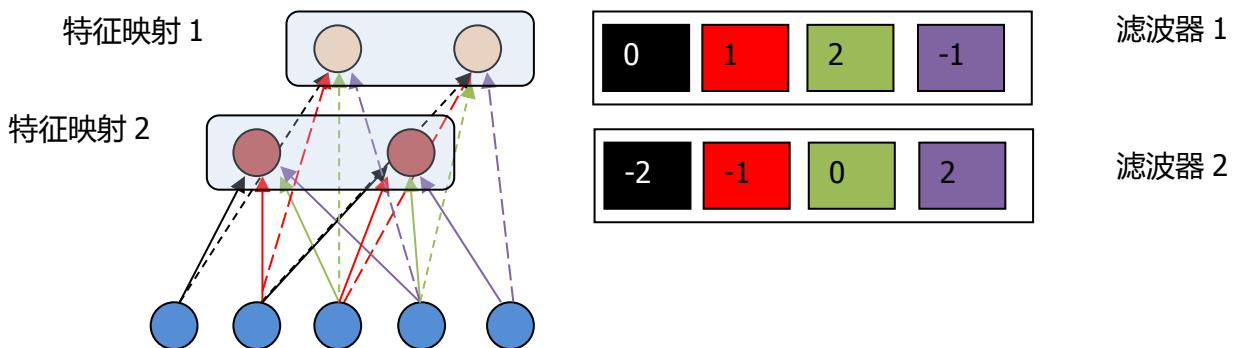


图 5.6 多个滤波器

## 5.2.2 卷积层

这里的卷积层是指构建卷积神经网络中的一个进行卷积操作的层。

Tips:

为了好理解下面的内容，我们说卷积神经网络的卷积操作运算结果即为神经网络的一层。运算结果（向量或矩阵）的每个元素，就是一个神经元。

### 1. 一维卷积层

在全连接前馈神经网络中，如果第  $l$  层有  $n^l$  个神经元，第  $l-1$  层有  $n^{l-1}$  个神经元，连接边有  $n^l \times n^{l-1}$  个。也就是权重矩阵有  $n^l \times n^{l-1}$  个参数。当隐层增加，或一层中的神经元增加，权重矩阵的训练参数非常多，训练效率会降低。

如果采用卷积来代替全连接，第  $l$  层的每一个神经元都只和第  $l-1$  层的一个局部窗口内的神经元相连，构成一个局部连接网络。第  $l$  层的第  $i$  个神经元的输出定义为：

$$a_i^{(l)} = f \left( \sum_{j=1}^m w_j^{(l)} \cdot a_{i-j+m}^{(l-1)} + b^{(l)} \right) = f(W^{(l)} \cdot a_{(i+m-1):i}^{(l-1)} + b^{(l)})$$

其中  $W^{(l)} \in R^{m \times m}$  为  $m$  维的滤波器，  $a_{(i+m-1):i}^{(l-1)} = [a_{i+m-1}^{(l-1)}, \dots, a_i^{(l-1)}]^T$ ；  $f$  是激活函数。这里  $a^{(l)}$  的下标从 1 开始。上述的公式也可以写成

$$a^{(l)} = f(W^{(l)} \otimes a^{(l-1)} + b^{(l)})$$

$\otimes$  表示卷积运算。从该公式可以看出， $W^{(l)}$  对于所有的神经元都是相同的。这就是卷积层的权重共享特性。这样，在卷积层，只需要  $m+1$  个参数。**另外，第  $l+1$  层的神经元个数不是任意选择的，而是满足 P82“补充”中的公式。（此时假设的卷积操作中没有补齐， $stride=1$ ）**

Tips:

- (1) 一个滤波器有一个偏置（标量）
- (2) 滤波器的参数和偏置都是待学习的参数

## 2. 二维卷积层

上述公式描述的是一维卷积层。在图像处理中，图像是以二维矩阵的形式输入到神经网络中，在文本处理时，很多情况也是将文本转换成了二维矩阵。因此需要二维卷积。假设  $X^{(l)} \in R^{(w_l \times h_l)}$  和  $x^{(l-1)} \in R^{(w_{l-1} \times h_{l-1})}$  分别是第  $l$  层和第  $l-1$  层的神经元。  $X^{(l)}$  的每一个元素（一个神经元）为：

$$X_{s,t}^{(l)} = f \left( \sum_{i=1}^u \sum_{j=1}^v W_{i,j}^{(l)} \cdot X_{s-i+u, t-j+v}^{(l-1)} + b^{(l)} \right)$$

注意：这里  $W$  和  $w$  含义不同。 $W^{(l)} \in R^{(u \times v)}$  是第  $l$  层的二维滤波器。 $w$  是以矩阵描述一层的神经元的个数时的宽度（因为计算的数据是矩阵形式，**想象一下神经元是按照矩阵排列的**） $w_l$  是第  $l$  层的神经元组的宽度， $h_l$  是排列的列数，第  $l$  层的神经元个数为  $w_l \times h_l$ ，并且  $w_l = w_{l-1} - u + 1$ ， $h_l = h_{l-1} - v + 1$ 。则上面的公式也可以写为

$$X^{(l)} = f(W^{(l)} \otimes X^{(l-1)} + b^{(l)})$$

为了增强卷积层的表示能力，可以在输入上使用  $K$  个不同的滤波器来得到  $K$  组输出。如果把滤波器看做是一个特征提取器，每一组输出都可以看成是输入图像经过一个特征提取后得到的特征。因此，在卷积神经网络中每一组输出也叫作一组**特征映射**（feature map）。不失一般性，我们假设第  $l-1$  层输出的特征映射组数为  $n_{l-1}$ ，每组特征映射的大小为  $m_{l-1} = w_{l-1} \times h_{l-1}$ 。第  $l-1$  层的总神经元数  $n_{l-1} \times m_{l-1}$ 。第  $l$  层的特征映射组数为  $n_l$ （有  $n_l$  个滤波器）。第  $l$  层的第  $k$  组特征映射  $X^{(l,k)}$  为

$$X^{(l,k)} = f\left(\sum_{p=1}^{n_{l-1}} (W^{(l,k,p)} \otimes X^{(l-1,p)}) + b^{(l,k)}\right)$$

其中， $W^{(l,k,p)}$ 表示第 $l$ 层的第 $k$ 个滤波器。一个滤波器的维度是由输入决定的。输入有 $p$ 个特征映射，每个特征映射是二维的。滤波器则是三维的，前两个维度是对应一个输入特征映射的滤波器的大小，第三个维度大小是 $p$ ，即输入特征映射的数目。

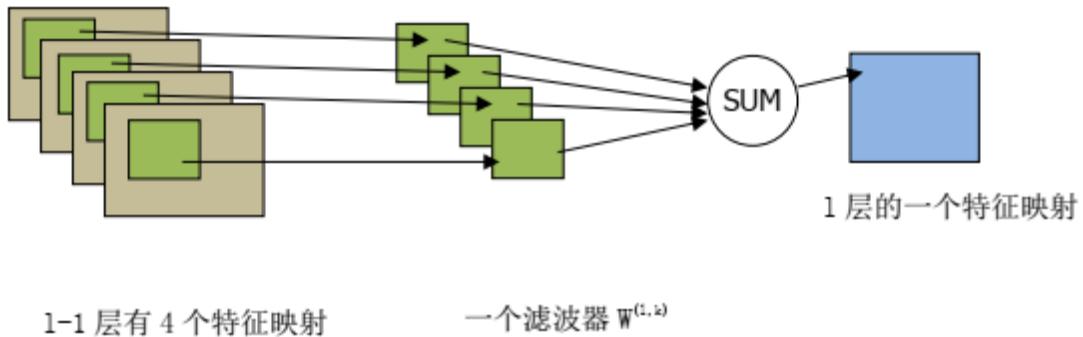


图 5.7 卷积操作示例

图 5.7 的滤波器是的 $l$ 层的第 $k$ 个滤波器 $W^{(l,k)}$ 。该滤波器有多个 kernel，即卷积窗口。每个卷积窗口对一个输入的特征映射进行卷积操作，多个卷积窗口操作的结果求和，然后得到一个输出的特征映射。

我们再强调，一个滤波器在一个输入上的卷积操作结果是一个输出特征映射；多个滤波器得到多个输出特征映射。

下面我们用图来演示一下二维卷积操作：设输入为  $5 \times 5 \times 1$  的矩阵（下图假设输入只有一个特征映射），滤波器是  $3 \times 3 \times 1$ ，不填充，步长（stride）为 1。输入特征映射经过一个滤波器的运算后，得到一个输出的特征映射。

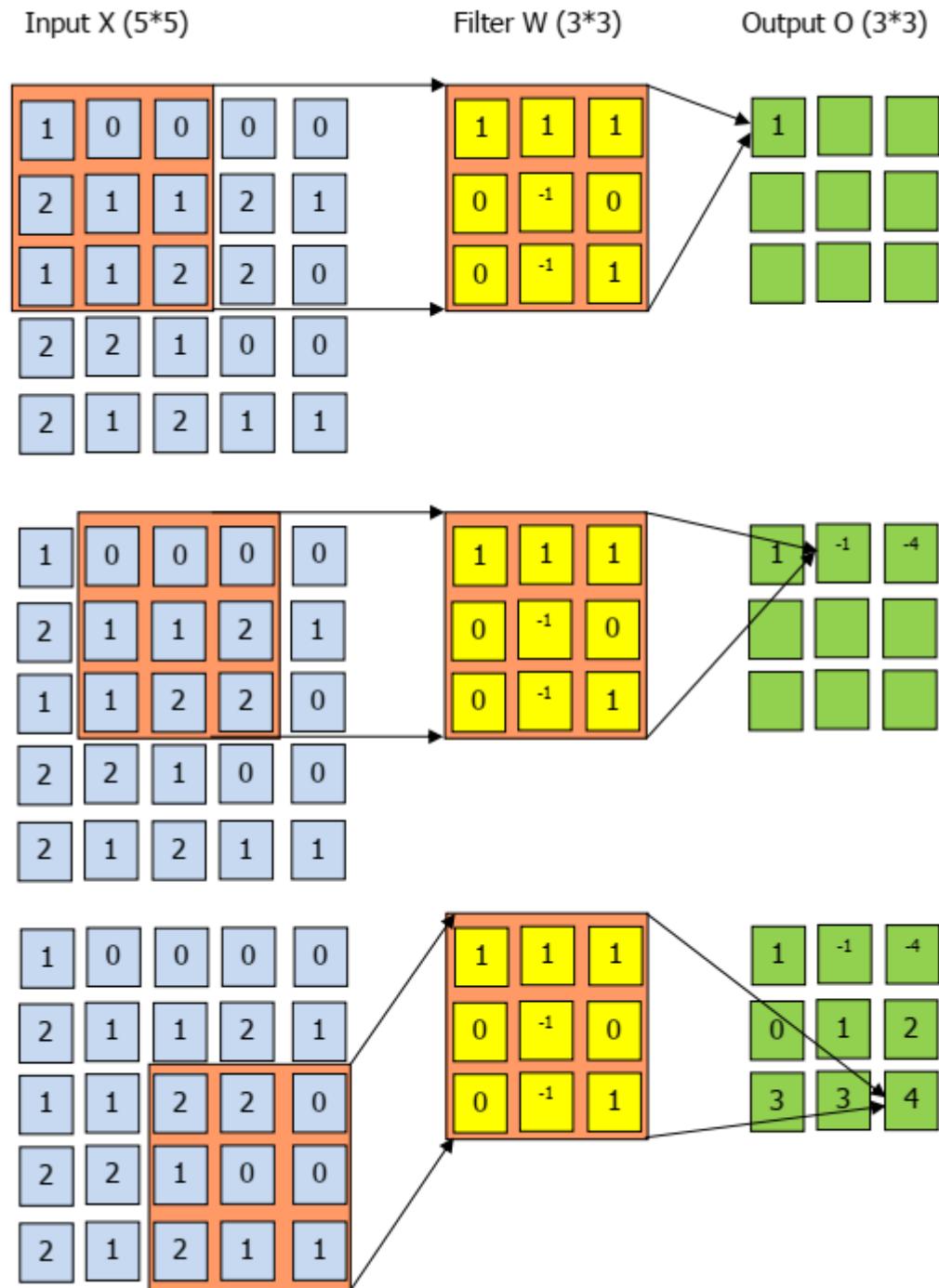


图 5.8 卷积操作示例 2

再举例：输入（ $5 \times 5 \times 3$ ）（假设输入特征映射有 3 个），步长 stride 为 2，滤波器（ $3 \times 3 \times 3$ ）个数为 2，zero-padding 填充量为 1。



图 5.9 卷积操作示例 3

图 5.8 的例子中个滤波器只是一个二维的矩阵，而图 5.9 的例子中滤波器是一个 cube。即对每个输入的特征映射应用不同的卷积窗口，且按照图 5.7 对一个滤波器的输出特征映射进行了求和。输出 output 为  $3 \times 3 \times 2$ ，其中的 2 是 output\_channels，它由 filter 的个数决定。在 <http://cs231n.github.io/convolutional-networks/> 有个二维卷积操作过程的动画演示。（注，我们的图和该网页的图数据不一样）。我们可以用图 5.10 来描述二维卷积层的从输入到输出的映射关系

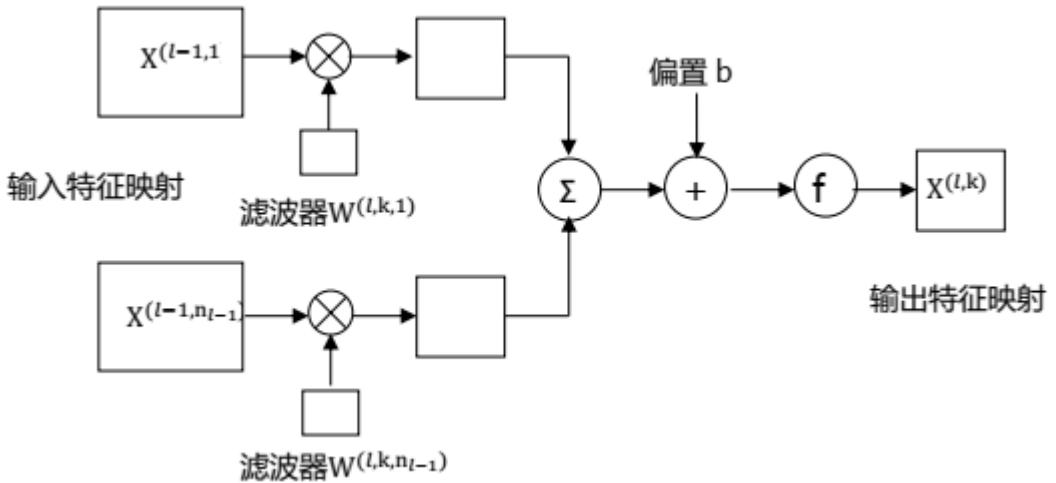


图 5.10 二维卷积层的从输入到输出的映射关系（第 k 个滤波器的例子）

此处的输入的多个特征映射，有  $n_{l-1}$  个，可以看成是整个 CNN 的输入（如果第一层是卷积层），此时以输入是图像为例，图像的 shape 是 [width, height, channel]，channel 是指图像的通道；如果是 RGB 三色，那么此时输入的是三个特征映射。输入的多个特征映射也可以将当前卷积层看做是 CNN 的第 l 层，则第 l 层的输入特征映射是  $l-1$  层的输出。此时滤波器的 shape 是 [filter\_height \* filter\_width \* in\_channels]。图上显示的是第 k 个滤波器。滤波器  $W^{(l,k,n_{l-1})}$  的含义是第 l 层第 k 个滤波器的第  $n_{l-1}$  个 channel，它负责对输入的第  $n_{l-1}$  个特征映射进行卷积操作。这里的 in\_channels 等于输入特征映射的个数。之所以此处使用 in\_channels 这个表示法，是为了和 TensorFlow 中的参数表示法对应。

我们可以看到一个滤波器中的每个 channel 对一个输入特征映射进行卷积操作，再将各结果求和，再加上偏置，最后得到一个输出映射。有几个滤波器，就有几个输出映射。

#### 5.2.4 子采样层 downsampling (或池化层 pooling )

卷积层虽然可以显著的减少连接的个数，但是每一个特征映射的神经元个数并没有显著减少。这样，如果后面接一个分类器，分类器的输入维数依然很高，很容易出现过拟合。为了解决这个问题，在卷积神经网络一般会在卷积层后再加上一个池化操作

( Pooling ) , 也就是子采样 ( subsampling ) , 构成一个子采样层。子采样层可以大大降低特征的维度 , 避免过拟合。子采样函数  $\text{down}(\cdot)$  一般是取区域内所有神经元的最大值 ( Maximum Pooling ) 或平均值 ( Average Pooling )

$$\text{down}(\cdot) = \text{pool}_{\max}(R_k) = \max_{i \in R_k} a_i$$

$$\text{down}(\cdot) = \text{pool}_{\text{avg}}(R_k) = \frac{1}{|R_k|} \sum_{i \in R_k} a_i$$

子采样的作用还在于可以使得下一层的神经元对一些小的形态改变保持不变性 , 并拥有更大的感受野。

子采样层 ( 或池化层 ) 在输入的 depth 维度上的每个切片进行操作。最常用的形式是一个 pooling 层使用 size 为  $2*2$  的滤波器 , 在输入立方体的 depth 维度 , 对每个 depth 切片 ( 即一个特征映射 ) 进行步长 stride 为 2 ,  $\text{Pool}_{\max}$  操作进行子采样 , 数据被压缩 75% , depth 维保持不变。

更一般性的描述 pooling 层 :

- (1) Pooling 层的输入是一个  $W_1*H_1*D_1$  的输入立方体 ( Volume )
  - (2) 需要两个超参数: 进行采样的滤波器的 size( 滤波器是一个正方形 )  $F$  ; 步长  $S$
  - (3) Pooling 层输出立方体的 size:  $W_2*H_2*D_2$
- $$W_2 = (W_1 - F)/S + 1; H_2 = (H_1 - F)/S + 1; D_2 = D_1$$
- (4) Pooling 层不会使用零填充 zero-padding
  - (5) Pooling 层没有引入参数

图 5.11 展示了一个 pooling 操作的过程。pooling 层空间独立地在输入立方体的每个 depth 切片上下采样立方体。

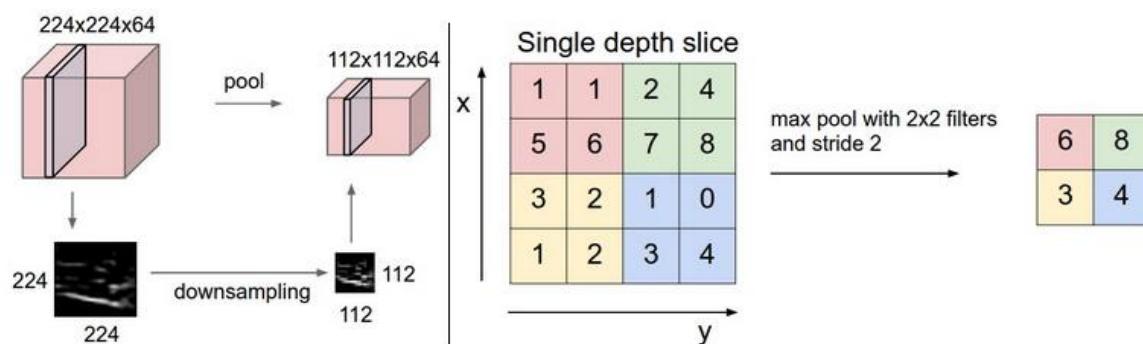


图 5.11 子采样示意图

先看左边，输入是一个  $224 \times 224 \times 64$  的立方体；它的 depth 维是 64，即有 64 个切片 slice，或者可以理解为输入有 64 个 feature map。Pooling 操作的滤波器 size 是  $2 \times 2$ （对应前述的 pooling 层超参数  $F=2$ ），操作步长为  $S=2$ 。因此 pooling 层的输出立方体是： $W_2 = (224-2)/2+1 = 112$ ;  $H_2 = 112$ ;  $D_2 = D_1 = 64$ 。

再看右边的 pooling 操作过程。此例中的一个 depth slice(或 feature map)的维度是  $4 \times 4$ ；滤波器的维度是  $2 \times 2$ ，步长为 2；pooling 操作采样 MAX 操作，即去滤波器对应区域中的最大值。因此，pooling 操作的输出得到 4 个值。

也有人对 pooling 层有不同意见，他们认为完全可以抛弃 pooling 层。他们建议在卷积层使用更大的步长，就可以有效的减小神经元的数目。

### 第三节：Keras 卷积层函数

Keras 提供了函数来卷积神经网络的各种层。本节介绍这些函数和使用这些函数的例子。

#### 6.4.1. 卷积层

Keras 提供了很多种的卷积层 <https://keras.io/layers/convolutional/>。本讲义只介绍其中常用的两种。

##### 1. Conv1D

```
keras.layers.Conv1D(filters, kernel_size, strides=1, padding='valid',
                     data_format='channels_last', dilation_rate=1, activation=None, use_bias=True,
                     kernel_initializer='glorot_uniform', bias_initializer='zeros',
                     kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
                     kernel_constraint=None, bias_constraint=None)
```

该函数针对在一个维度上的数据，例如时态数据，创建一个卷积层。常用的参数含义如下：

- (1) filters : 滤波器的个数。
- (2) kernel\_size : 滤波器窗口大小。Kernel\_size 是一个整数，是卷积窗口的长度，即图 5.24 中的 height。其宽度是固定的，等于一个时间步向量的长度 (channel 的大小)。
- (3) strides : 滤波的步长

(4) padding : 补齐方式。有三种选择“valid”，“causal”或者“same”（注意，都是小写）。Valid 即不补齐，“same”即补齐后的输出和输入的长度是一样的。“causal”称为 dilated 卷积。

(5) data\_format : 定义输入 tensor 数据的维度安排。默认是“channel\_last”对应的输入的 shape 是 $(batch, steps, channels)$ ；“channel\_first”对应 $(batch, channels, steps)$ 。

Input shape : 3D tensor with shape:  $(batch, steps, channels)$

Output shape : 3D tensor with shape:  $(batch, new\_steps, filters)$

（我理解此处 input shape 中的 channels 是序列的一个时间步（即一个向量）的长度）

CNN 最初开发上是针对图像数据。模型接受的输入数据是 2 维的，图像的“像素”和“通道”。这样的操作也可以应用在 1 维的序列数据上。当然，后面我们要介绍的 RNN 也可以处理序列数据。一维卷积的例子如下。

### 1D CONVOLUTIONAL - EXAMPLE

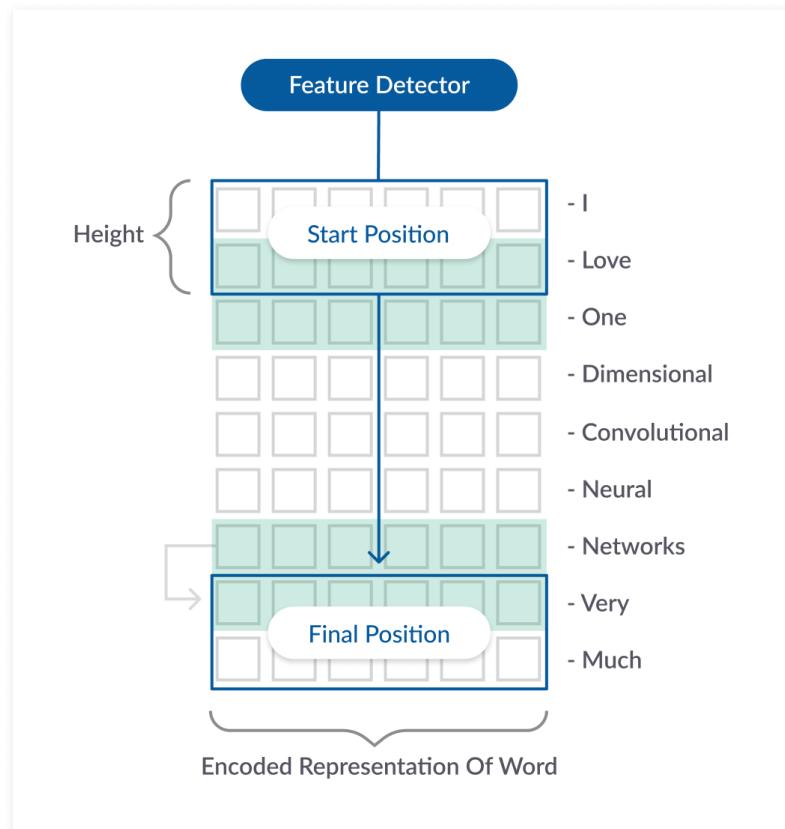


图 5.24 1D 卷积示例

图中的例子是一个句子，句子中的每个词用一个向量描述，称为 word embeddings。如果是序列数据，每个词可以换成序列中的一个 time step。而每个 time step 有多个特征描述。具体实例参考第七节，使用 1D CNN 进行活动识别。

## 2. Conv2D

```
keras.layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid',
data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True,
kernel_initializer='glorot_uniform', bias_initializer='zeros',
kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
kernel_constraint=None, bias_constraint=None)
```

参数的含义同上面的 Conv1D。在第 6 节给出了一个使用 Conv2D 的手写数字识别的例子。

Kernel\_size 是一个整数或一个元组 ( n, m )。它规定了卷积窗口的尺寸。给一个值 n 时即卷积窗口是(n , n)。注：图 5-10 中的滤波器是一个立方体，即还有一个深度，对于输入特征映射的 channel ( 或输入特征映射的数目 )。因为，这个深度是固定的，是和输入的特征映射数目一致的，所以 kernel\_size 这个参数里面不需要规定。

Data\_format：默认值是 channels\_last。

Input shape：

```
if data_format is "channels_first" 4D tensor with shape: (batch, channels, rows,
cols)

if data_format is "channels_last" 4D tensor with shape: (batch, rows, cols,
channels)
```

Output shape：

```
if data_format is "channels_first" 4D tensor with shape: (batch, filters,
new_rows, new_cols)

if data_format is "channels_last" 4D tensor with shape: (batch, new_rows,
new_cols, filters)
```

注：rows 和 cols 值可能由于 padding 而改变。

### 5.4.2 池化层

本讲义仅介绍常用的最大池化和平均池化的函数，其他的见  
<https://keras.io/layers/pooling/>

#### 1. MaxPooling1D

```
keras.layers.MaxPooling1D(pool_size=2, strides=None, padding='valid',
data_format='channels_last')
```

pool\_size：最大池化窗口的大小

strides：步长，默认值是 pool\_size。

padding：补齐方式“same”或“valid”。Valid 即不补齐，“same”即补齐后的输出和输入的长度是一样的

data\_format：定义输入数据的维度安排。默认是“channel\_last”对应的输入的 shape 是 (batch, steps, channels)；“channel\_first”对应 (batch, channels, steps)。

注：池化或下采样的方式是沿着 row, 以步长 strides 移动。沿着 features 步长为 1。第六节有例子。

input shape：

如果 `data_format='channels_last'`：3D tensor with shape: (batch\_size, steps, features)

如果 `data_format='channels_first'` 3D tensor with shape: (batch\_size, features, steps)

Output shape:

If `data_format='channels_last'`：3D tensor with shape: (batch\_size, downsampled\_steps, features)

If `data_format='channels_first'`：3D tensor with shape: (batch\_size, features, downsampled\_steps)

## 2. MaxPooling2D

```
keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid',  
data_format=None)
```

参数的含义同 MaxPooling1D

Input shape

If `data_format='channels_last'`：4D tensor with shape: (batch\_size, rows, cols, channels)

If `data_format='channels_first'`：4D tensor with shape: (batch\_size, channels, rows, cols)

Output shape

```
If data_format='channels_last': 4D tensor with shape: (batch_size, pooled_rows, pooled_cols, channels)
```

```
If data_format='channels_first': 4D tensor with shape: (batch_size, channels, pooled_rows, pooled_cols)
```

## 第四节：Dropout

有大量参数的深度神经网络是非常有力的机器学习系统，然而这样的网络过拟合是个很严重的问题。Dropout 是深度学习中防止过拟合的一个简单却非常有效的技巧（参看 Hinton 的文章 Dropout: A Simple Way to Prevent Neural Networks from Overfitting）。术语"dropout"是指在神经网络中 dropping out units (隐层节点)

**注：dropout 只是在模型训练时期作用，在预测阶段不起作用**

**dropout 函数** `keras.layers.Dropout(rate, noise_shape=None, seed=None)`

按照概率 rate 对输入的元素进行选择输出。选择了的输出，其元素值乘上  $1/rate$  进行标定；否则输出为 0。

关于 Dropout，文章中没有给出任何数学解释，Hinton 的直观解释和理由如下：

1. 由于每次用输入网络的样本进行权值更新时，隐含节点都是以一定概率随机出现，因此不能保证每 2 个隐含节点每次都同时出现，这样权值的更新不再依赖于有固定关系隐含节点的共同作用，阻止了某些特征仅仅在其它特定特征下才有效果的情况。
2. 可以将 dropout 看作是模型平均的一种。对于每次输入到网络中的样本（可能是一个样本，也可能是一个 batch 的样本），其对应的网络结构都是不同的，但所有的这些不同的网络结构又同时共享隐含节点的权值。这样不同的样本就对应不同的模型，是 bagging 的一种极端情况。Deep learning 一书的 7.12 节从 bagging 角度解释了 dropout 的工作原理。

Keras 中，实现 dropout 功能是创建一个层。附录中第五节对该 dropout 类有介绍，下面是一个实现 dropout 的代码。

```
from tensorflow.keras.layers import Input, Dense  
from tensorflow.keras.models import Model  
from tensorflow.keras import regularizers
```

```

from tensorflow.keras.layers import Dropout

# This returns a tensor
inputs = Input(shape=(784,))
# a layer instance is callable on a tensor, and returns a tensor
output_1 = Dense(64, activation='relu')(inputs)

output_2 = Dense(64, activation='relu')(output_1)
output_3 = Dense(64, input_dim=64,
                 kernel_regularizer=regularizers.l2(0.01),
                 activity_regularizer=regularizers.l1(0.01))(output_2)
output_4 = Dropout(rate=0.5)(output_3)
predictions = Dense(10, activation='softmax')(output_4)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)

```

**Tips:**

实践中使用 dropout 的技巧是。在 CNN 中，最后的卷积层（包括 pooling 层）后加一个全连接层，再加一个 dropout 操作。实验证实可以大幅度的提供模型的性能。至于为什么，没有一个科学的解释。

## 第五节：实例 1：基于 CNN 的手写数字识别

在第 3 章我们已经用 TensorFlow 实现了一个基本的 SoftMax 回归模型来实现手写数字的识别。这一节我们将实现一个 CNN 再次进行手写数字的识别，看看精确度的提升。在 3.3 节我们已知 MNIST 数据集的一张图片是  $28 \times 28 \times 1$  的数据（灰度图片是单通道）。我们构建的的模型框架如图 5.14 所示：

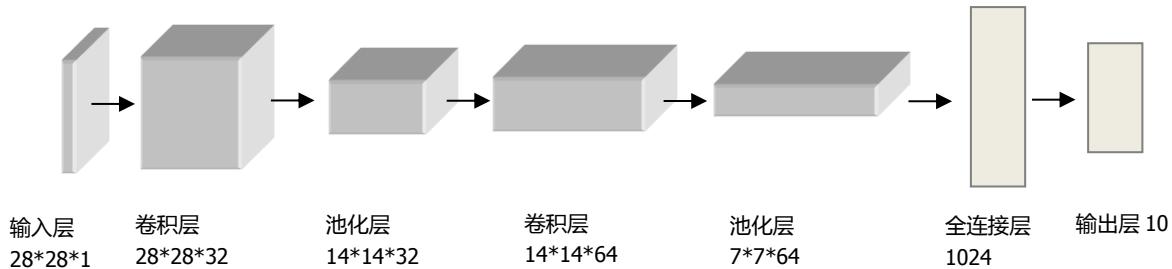


图 5.14 一个用于手写数字识别的卷积神经网络

### 1. 输入层

首先读入数据

```
mnist = tf.keras.datasets.mnist  
(x_train, y_train), (x_test, y_test) = mnist.load_data()  
x_train, x_test = x_train / 255.0, x_test / 255.0  
n,m = x_train.shape[1], x_train.shape[2]
```

此时读入的数据集 `x_train` 的 `shape=(samples, row, col)`。而实际上对于图片数据，它的 `tensor` 的 `shape` 应该是 `shape=(samples, row, col, channel)`。只是当前数据集中的 `channel=1`。且当我们在该数据集上用 `Conv2D` 函数实施卷积操作，它要求的输入的 `tensor` 的 `shape` 也是 `= (samples, row, col, channel)`

因此，模型的第一个层是创建的一个 `reshape` 层，它负责将输入的 `tensor` “变形”。

```
model = Sequential()  
model.add(Reshape((n,m,1), input_shape=(n,m)))
```

## 2. 第一个卷积层和 pooling 层

```
model.add(Conv2D(filters=32, kernel_size=5, activation='relu', padding="same"))  
model.add(MaxPool2D())
```

卷积层的滤波器有 32 个，窗口是  $5 \times 5$ 。卷积层采用 `relu` 激活函数。补全方式是 “same”，即输入的 `tensor` 和输出的 `tensor` 具有相同的维度。

卷积层输入 `tensor` 的 `shape=(28,28,1)`。输出 `tensor` 的 `shape=(28,28,32)`

经过一个最大池化层 `MaxPool2D`，采用默认参数，输出 `tensor` 的 `shape=(14,14,32)`

## 3. 第二个卷积层和 pooling 层

```
model.add(Conv2D(filters=64, kernel_size=(5,5), activation='relu', padding="same"))  
model.add(MaxPool2D())
```

第二个卷积层的权重（即滤波器）的长和宽是  $5 \times 5$ ；因为第一个卷积层的滤波器有 32 个，因此输出的特征映射（通道）是 32 个；第二个卷积层使用 64 个滤波器。因此输出的 `tensor` 的 `shape=(14,14,64)`

再经过一个默认参数的最大池化层，输出 `tensor` 的 `shape=(7,7,64)`

## 4. 全连接层

把池化操作的结果转换成向量，这是一个长度为  $7 \times 7 \times 64$  的向量。再定义一个全连接层，它有 1024 个神经元。激活函数是 `relu`。再经过一个 `dropout` 层

```
model.add(Flatten())  
model.add(Dense(1024,activation='relu', kernel_initializer='he_normal'))  
model.add(Dropout(0.5))
```

## 5. 输出层

输出层有 10 个神经元，因为输出是对 10 个数字的判断。全连接层有 1024 个神经元，使用 softmax 激活函数

```
model.add(Dense(10, activation='softmax'))
```

## 6. 训练模型

定义损失函数，因为是多类分类，因此采用 SparseCategoricalCrossentropy. 因为模型 model 的输出（即预测结果）经过 softmax 激活函数，是在每个类别上的概率分布，因此应该设 from\_logits=True。

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='Nadam',
              loss=loss_fn,
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=10, batch_size=100)
model.evaluate(x_test, y_test, verbose=2)

for layer in model.layers:
    print('%.s - %.s' %(layer.input_shape, layer.output_shape))
```

## 第六节：实例 2：基于 1D CNN 的活动识别

一个使用三星手机记录的人类活动数据的加速度序列数据集。现在需要在这个数据集上进行分类，识别这个人是在做什么运动。它是一个 UCI 数据集

<https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones>

在论文《Human Activity Recognition on Smartphones using a Multiclass Hardware-Friendly Support Vector Machine》有对该数据集的描述和在该数据集上的研究。现在看到的数据是预处理后的数据，包括：（1）划分好了训练集合测试集；（2）每条数据是一个时间序列数据，包括 128 个时间点或时间步；（3）每个时间步有 9 个数据是各种和各方向上的加速度数据。用图 5.15 来描述该模型如下：

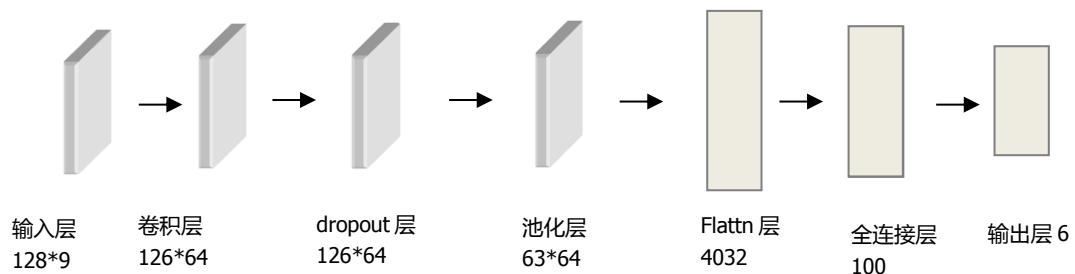


图 5.15 模型

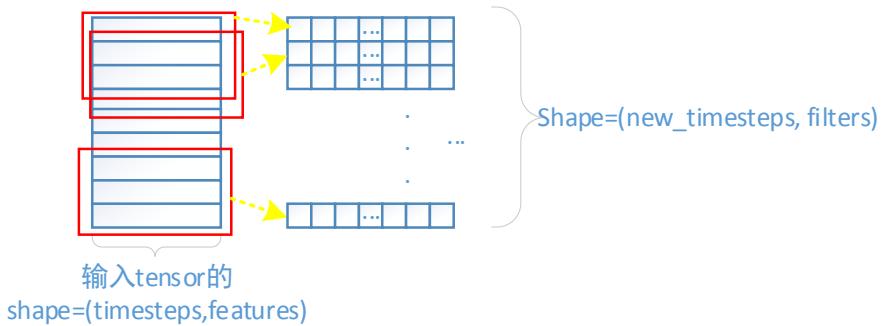


图 5.16 1D 卷积操作

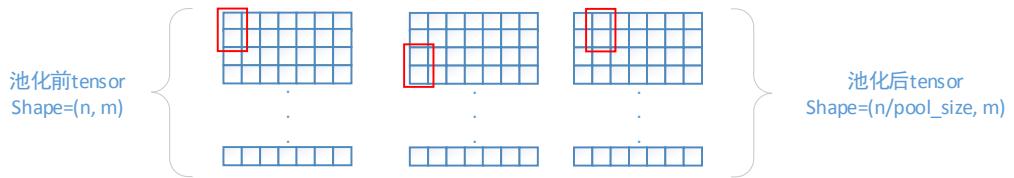


图 5.17 1DMaxPooling 操作

```
from numpy import mean
from numpy import std
from numpy import dstack
from pandas import read_csv
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Conv1D
from tensorflow.keras.layers import MaxPooling1D
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Model

# load a single file as a numpy array
def load_file(filepath):
    dataframe = read_csv(filepath, header=None, delim_whitespace=True)
    return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix ""):
    loaded = list()
    for name in filenames:
        data = load_file(prefix + name)
        loaded.append(data)
    # stack group so that features are the 3rd dimension
```

```

loaded = dstack(loaded)
return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=""):
    filepath = prefix + group + '/Inertial Signals/'
    # load all 9 files as a single array
    filenames = list()
    # total acceleration
    filenames += ['total_acc_x_' + group + '.txt', 'total_acc_y_' + group + '.txt',
    'total_acc_z_' + group + '.txt']
    # body acceleration
    filenames += ['body_acc_x_' + group + '.txt', 'body_acc_y_' + group + '.txt',
    'body_acc_z_' + group + '.txt']
    # body gyroscope
    filenames += ['body_gyro_x_' + group + '.txt', 'body_gyro_y_' + group + '.txt',
    'body_gyro_z_' + group + '.txt']
    # load input data
    X = load_group(filenames, filepath)
    # load class output
    y = load_file(prefix + group + '/y_' + group + '.txt')
    return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=""):
    # load all train
    trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
    print(trainX.shape, trainy.shape)
    # load all test
    testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
    print(testX.shape, testy.shape)
    # zero-offset class values
    trainy = trainy - 1
    testy = testy - 1
    # one hot encode y
    trainy = to_categorical(trainy)
    testy = to_categorical(testy)
    print(trainX.shape, trainy.shape, testX.shape, testy.shape)
    return trainX, trainy, testX, testy

# hyper-parameters
verbose, epochs, batch_size = 0, 10, 32
trainX, trainy, testX, testy = load_dataset('D:/qjt/beike/deeplearning/2020/')

n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2],
trainy.shape[1]

model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
input_shape=(n_timesteps,n_features)))
model.add(Dropout(0.5))
model.add(MaxPooling1D(pool_size=2))

```

```

model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dense(n_outputs, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
_, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)

score = accuracy * 100.0
print('accuracy: %.3f' % (score))

for layer in model.layers:
    print(layer.output_shape)

```

该模型的构建首先创建一个 1D 卷积层

```
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
input_shape=(n_timesteps,n_features)))
```

设定了 input\_shape 参数，即输入层。模型输入的是一个固定时间步的子序列。每个时间步是一个向量。隐层，tensor 的 shape=(n\_timesteps,n\_features)。

卷积层使用了 64 个滤波器，每个滤波器的窗口大小是 3。即 5.4.1 节的图 5.24 中的 height 是 3。

卷积层后紧跟一个池化层

```
MaxPooling1D(pool_size=2)
```

池化层的输出被拉伸成一个向量

Flatten()

再加一个全连接层

```
Dense(100, activation='relu')
```

经过 Drop 处理（添加一个 Dropout 层）

Dropout(0.5)

最后是输出层。因为是多类分类问题，因此输出层采用 softmax 激活函数

```
Dense(n_outputs, activation='softmax')
```

这段代码考察每个层的 output shape

```
for layer in model.layers:
    print(layer.output_shape)
```



# 第六章：神经语言模型和词的表示学习

在传统的文本挖掘、NLP 领域中，每个词被当做一个原子单元（atomic unit）。例如，一个词就是一个字符串，词之间的相似性只有通过词典的定义来评估；在文本处理时一个词被编码分配一个编号。基于这样思想的 N 元（N-gram）语言模型在 NLP，文本挖掘中仍是非常受欢迎。机器学习领域的研究有这样的共识：**简单的模型+非常大量的训练数据>复杂模型+少量数据**。N-gram 这些简单的技巧在许多任务中会遇到问题。例如，在语音识别中需要非常大量的领域内相关数据，然而在这样的任务中获取高质量的这样数据的量被限制了。因此在这些领域简单技巧+海量数据的模式被限制了。必须寻找更高级的技巧。

在 NLP 领域采用词的分布式表示（distributed representation of words）和神经网络构建的神经网络语言模型很多研究已经证明其性能超过了 N-gram 语言模型。“word embeddings 词嵌入”或者“word representation 词表示”或者“distributed representations of words 分布式词表示”三个术语都是一个意思，是用一个实数向量表示词，本文后面将其称为**词向量**。词的表示学习在深度学习出现之前就已经有了。2001 年 Bengio 就在两篇论文中提出了 word embeddings。而和 word embedding 思想相似的关于符号的分布式描述则在更早 1986 年就由 Hinton 提出。但是随着深度学习在许多领域取得成功，将深度学习应用在自然语言处理时需要词的表示学习，因此这几年词的表示学习的研究也火热起来。将词用“词向量”的方式表示可谓将 Deep Learning 算法引入 NLP 领域的一个核心技术。大多数宣称用了 Deep Learning 的论文，其中往往也用了词向量。

一个 word embeddings **W:words→R<sup>n</sup>**

是一个函数将词映射到高维向量（可以达到 200 到 500 个维度）。例如

$$W("cat")=(0.2, -0.4, 0.7, \dots)$$

$$W("mat")=(0.0, 0.6, -0.1, \dots)$$

通常建立 word embedding 即词的表示学习是一个学习任务。

注：

这么理解，词的表示学习把词映射到了高维的实数空间。经常在一起出现的词被映射到空间相邻的位置，不经常出现的就映射到比较远的位置。因此，在这个向量空间就可以计算词的语义相似性。

有很多词的表示学习的方法，Word2vec 和 Glove 是其中著名的两个。Word2vec 是基于神经网络的方法，Glove 是基于张量分解的方法。它们基于训练集训练出 word 向量，其维度可以在[50-100]。而且令人惊奇的，在这些新方法获得的词向量上的一些算术操作和它们的语义关系可以对应。例如，

$$\text{vec}(\text{"King"}) - \text{vec}(\text{"Man"}) + \text{vec}(\text{"Woman"}) \approx \text{vec}(\text{"Queen"})$$

$$\text{vec}(\text{"Madrid"}) - \text{vec}(\text{"Spain"}) + \text{vec}(\text{"France"}) \approx \text{vec}(\text{"Paris"})$$

## 第一节：背景知识

### 1.1.1 词向量

#### One-hot Representation

自然语言理解的问题要转化为机器学习的问题，第一步是要找一种方法把这些符号数学化。NLP 中最直观的方法是 One-hot Representation。这种方法建立一个词表，给每个词顺序编号，每个词就是一个很长的向量，向量的维度等于词表大小，只有对应位置上的数字为 1，其他都为 0。当然在实际应用中，一般采用稀疏编码存储，主要采用词的编号。例如

“话筒”表示为 [0 0 0 **1** 0 0 0 0 0 0 0 0 0 ...]

“麦克”表示为 [0 0 0 0 0 0 0 **1** 0 0 0 0 0 0 ...]

采用稀疏编码后，话筒记为 3，麦克记为 8（假设从 0 开始记）。这种表示方法也存在一个重要的问题就是“词汇鸿沟”现象：任意两个词之间都是孤立的。光从这两个向量中看不出两个词是否有关系，哪怕是话筒和麦克这样的同义词也不能被识别出语义相似性。

#### Distributed Representation of Words

前面已经谈论了，“word embeddings 词嵌入”或者“word representation 词表示”或者“distributed representations of words 分布式词表示”三个术语都是一个意思。它们都是关于用实数向量描述一个词，称之为词向量。

#### 词向量的用途

词向量除了在深度学习中使用，还有其他的用途：

(1) 词之间的相似性。当在一个语料库上训练完了词向量。就可以用余弦相似度等来计算两个词之间的相似性。进而，可以做词聚类，发现相似的词的集合。

(2) 短文本的相似性计算。两个文本中的每个词向量的逐对相似度之和的平均值

### 1.1.2 语言模型、统计语言模型和神经语言模型

语言模型 (language model) 可以完成这样一个任务，它为一种语言中的句子计算概率分布 (产生一个句子的可能性)。它也可以完成这样的任务，计算在一个词序列之后跟随一个给定的词 (会一个词的序列) 的概率。例如，barked这个词跟在the lazy dog之后的概率。目前的语言模型的性能 (智能) 还达不到和人一样的程度，但是机器翻译、语音识别的关键技术。

#### (1) 统计语言模型

传统的统计语言模型是表示语言基本单位 (一般为句子) 的概率分布函数，这个概率分布也就是该语言的生成模型。一般语言模型可以使用各个词语条件概率的形式表示，我们也称之为n-gram语言模型：

$$p(s) = p(w_1, w_2, \dots, w_N) = \prod_{n=1}^N p(w_n | \text{context})$$

这里 context 是一个词的上下文，即一个词的概率是一个条件概率，和它的上下文有关。如果不考虑上下文，则这个语言模型是一元语言模型，即 n-gram 中的 n=1。

$$p(w_1, w_2, \dots, w_N) = \prod_{n=1}^N p(w_n)$$

当 n=2，这是二元语言模型，一个词的条件概率仅考虑它前面的一个词。

$$p(w_1, w_2, \dots, w_N) = \prod_{n=1}^N p(w_n | w_{n-1})$$

因为 n 的值越大，语言模型越复杂。在信息检索和文本挖掘中，一元模型往往已经足够。如果 n>2 更高阶的模型往往过于复杂，得不偿失。但在深度学习中很多研究拓展到更高阶的语言模型。

传统的语言模型的参数估计利用语料库进行最大似然估计。如，

$$p(w_i | w_{i-1}) = \text{count}(w_{i-1}, w_i) / \text{count}(w_i)$$

count( $w_i, w_{i-1}$ )是词  $w_i$  和  $w_{i-1}$  以这样的次序在语料库中出现的次数。count( $w_{i-1}$ )是词  $w_{i-1}$  在语料库中出现的次数。

## (2) 神经语言模型

Bengio 在 2003 年提出神经网络语言模型 NNLM，是用前馈神经网络训练语言模型。他的论文 “A Neural Probabilistic Language Model” 做了详细描述。Bengio 用三层神经网络训练语言模型，如图 6.1 所示。NNLM 的目标是学习一个语言模型  $p(w_1, w_2, \dots, w_T) = \prod_{t=1}^T p(w_t | w_{t-n+1}, \dots, w_{t-1})$

图中最下方的  $w_{t-n+1}, \dots, w_{t-2}, w_{t-1}$  就是前  $n-1$  个词。模型根据已知的  $w_{t-n+1}, \dots, w_{t-2}, w_{t-1}$  这已知的前  $n-1$  个词预测  $w_t$ 。 $c(w)$  表示词对应的词向量。通常会有个词查找表，可以根据一个词获取该词的词向量。

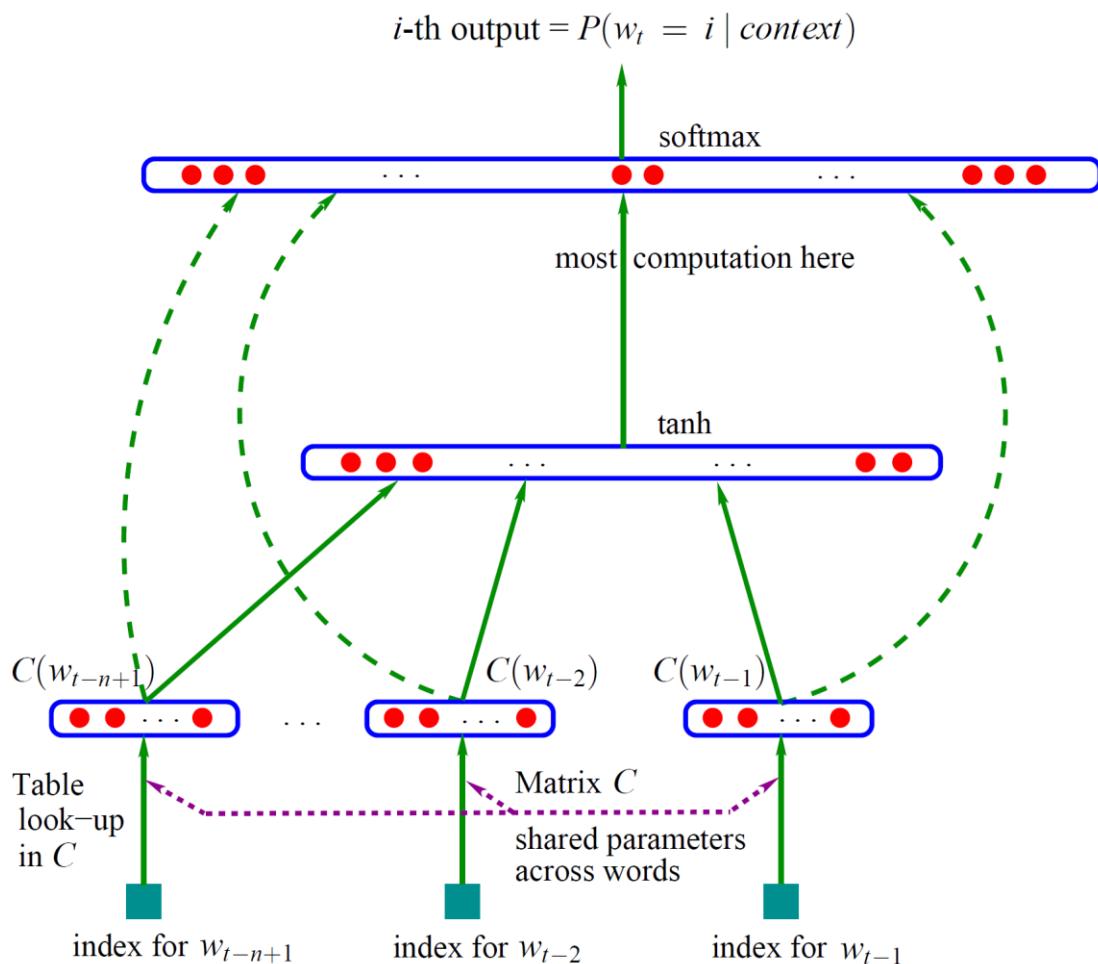


图 6.1. 神经网络语言模型

网络的隐层是将  $C(w_{t-n+1}), \dots, C(w_{t-2}), C(w_{t-1})$  这  $n-1$  个向量首尾相接拼起来，形成一个  $(n-1)*m$  的向量，下面记为  $x$ 。

网络的隐层进行计算  $D + Hx$  计算得到。 $D$  是偏置向量， $H$  是权重向量， $x$  是输入向量。隐层使用  $\tanh$  作为激活函数。 $\tanh$  激活函数，将输出压缩到了 -1 到 1 之间。

网络的输出层一共有 $|V|$ 个节点（ $V$ 是词汇表），节点 $y_i$ 表示下一个词为 $w_i$ 的未规范化log概率。最后使用 softmax 激活函数将输出值 $y$ 规范化成概率。最终， $y$ 的计算公式为：

$$y = b + Wx + U * \tanh(d + Hx)$$

公式中 $U$ （一个 $|V| \times h$ 的矩阵）是隐层到输出层的参数，式子中还有一个矩阵 $W$ ，这个矩阵包含了从输入层到输出层的直连边。直连边就是从输入层直接到输出层的一个线性变换，也是神经网络中的一种常用技巧。如果不需要直连边的话，将 $W$ 置为0就可以了。

需要注意的是，一般神经网络的输入层只是一个输入值，而 NNLM 有个矩阵 $C$ 也是模型学习的参数。 $C$ 是一个词向量的集合构成的矩阵。优化结束之后，词向量有了，语言模型也有了。

### 1.1.3 词的表示学习

词向量的表示学习有两种形式。一种是专门用工具，如神经语言模型，词向量是模型的参数，训练完成一个语言模型时就获得了词向量。通常我们会在很大的语料库上专门学习词向量，这称为 **pre-trained embeddings**（通用的词向量）。例如 Google 就提供了通用的词向量，用户可以去下载。语言模型可以用于词的表示学习。其实 Bengio 的 NNLM 也可以专门用作训练词向量（语言模型都可以训练词向量），下面介绍的 word2vec 中的 CBOW 就是对 NNLM 的改进。

另一种，词向量是其他任务的副产品。例如，词向量是一个 CNN 文本分类模型待学习的参数，当训练完了语言模型，也就得到了每个词的向量。很多深度学习的文本挖掘任务中就是如此操作，即不需要事先准备词向量。词向量可以从任务的训练过程中，从训练集中学习。

在特殊任务中建议还是使用自己训练的词向量。也有研究两种词向量结合使用。例如，在用 CNN 处理文本时，每种词向量构成输入数据的一个 Channel。第二节讨论训练词向量的两种算法。

## 第二节：word2vec

word2vec 其实是一个词表示学习的工具箱

（<https://code.google.com/archive/p/word2vec/>），该工具箱的实施基于 Google 公司的 Tomas Mikolov 的两篇论文：“Efficient Estimation of Word Representations in Vector Space” 和 “Distributed Representations of Words and Phrases and their Compositionality”。这个工具箱实施了词表示学习的两个模型 continuous bag-of-

words ( CBOW ) 和 skip-gram architectures ( 见上述论文 ) 。学习的词向量用在进一步的文本挖掘 , NLP 中。例如 , 再用在深度学习中作为深度神经网络的输入。

Word2vec 工具使用语料库作为输入产生 word 向量作为输出。它首先从训练文本数据构造词典 , 然后学习词的向量表示。其结果可以用在许多机器学习的应用中作为特征。

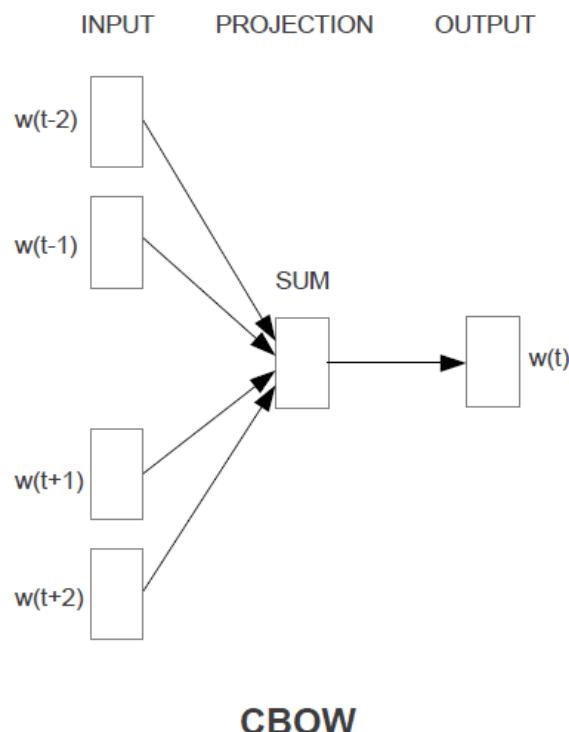
word2vec 非常受欢迎的另一个原因是其高效性 , Mikolov 在论文[2]中指出一个优化的单机版本一天可训练上千万词。我们下面介绍 Word2Vec 工具箱中实现的两个模型。

### 6.2.1 CBOW

CBOW 是一种与前馈 NNLM 类似的模型 ( 简称 FNNLM ) , 不同点在于 CBOW 去掉了最耗时的非线性隐层 tanh , 且所有词共享隐层。“词共享隐层” 的含义如下 :

FNNLM 中是拼接 (  $n-1$  ) 个  $m$  维向量 , 因此隐层的神经元数是  $(n-1)*m$  ; 而在 CBOW 模型中是把  $(n-1)$  个  $m$  维向量取平均值 , 隐层的神经元数是  $m$  。与 FNNLM 不同的是 , CBOW 中不仅会使用要预测的词前面的  $k$  个词 , 也会使用之后的  $k$  个词。

如图 6.2 所示。可以看出 , CBOW 模型是预测  $P(w_t|w_{t-k}, \dots, w_{t-1}, w_{t+1}, \dots, w_{t+k})$  。在 Mikolov 的论文中指出 ,  $k=4$  可以获得更好的性能。这里  $w_t$  的前  $k$  个和后  $k$  个词可以无关顺序 , 但有个连续的窗口  $k*2$  , 因此该模型称作 continuous bag of words 模型 CBOW。( 就是说 , 在训练文本上使用一个  $k*2$  的窗口来获得连续的词的序列 , 但实际上在这个词的序列内部实际上是无关次序的 , 因为都是要求平均 )



## 图 6.2. CBOW 模型

从输入层到隐层所进行的操作实际就是上下文向量的求和。图中没画出 NNLM 中的矩阵  $C$ ，即词向量集合。 $C$  也是 CBOW 的参数。比 FNNLM，CBOW 还少了直连操作。另外，图 6.2 是 TensorFlow 介绍的 CBOW 模型，各输入向量做了求和操作，不是求平均，这两种方法没有本质的不同。

### 6.2.2 Skip-gram

Skip-gram 也是 Mikolov 在和 CBOW 同一篇论文中提出的。它与 CBOW 相似，但不是基于上下文预测当前词  $w_t$  的概率  $p(w_t | \text{context})$ ，skip-gram 使用当前的词向量来预测该词之前和之后各  $k$  个词的概率。即预测概率  $p(w_i | w_t)$ ，其中  $t-c \leq i \leq t+c$  且  $i \neq t$ ，参数  $c$  决定窗口大小。假设存在一个  $w_1, w_2, w_3, \dots, w_T$  的词组序列，Skip-gram 的目标是最大化：

$$\frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c, j \neq t} \log(p(w_{t+j} | w_t))$$

概率的计算使用 softmax 函数，定义如下：

$$p(w_o | w_l) = \frac{\exp(v_{w_o}^T v_{w_l})}{\sum_{t=1}^W \exp(v_t^T v_{w_l})}$$

$v_w$  和  $v'_w$  是词  $w$  的输入和输出的词向量表示； $W$  是词汇表中的大小。在具体的实施中，该模型很不实用，因为计算梯度  $\nabla \log(p(w_{t+j} | w_t))$  的代价太大（要和词汇表中的每个词进行计算，word2vec 采用了 Negative Sampling 技巧解决此问题）。

在 Milolov 的论文“Distributed Representations of Words and Phrases and their Compositionality”给出了改进的模型。

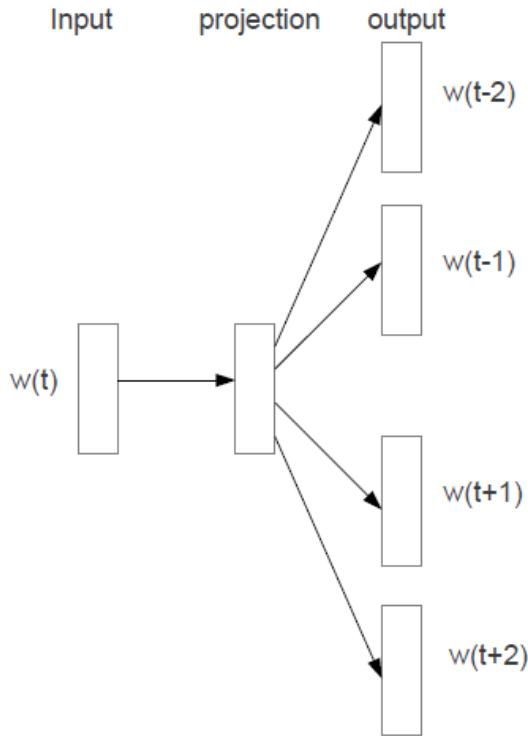


图 6.3. Skip-gram

该模型之所以叫 skip-gram，是在一个窗口内， $w_t$  不止和它的前一个词  $w_{t-1}$  和后一个词  $w_{t+1}$  计算概率  $p(w_{t-1}|w_t)$ 、 $p(w_{t+1}|w_t)$ ，而是和窗口内所有的词，这就是 skip。这样“白色汽车”和“白色的汽车”都会被识别为相同的短语。而且 skip-gram 是一个对称模型，即

$$\frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j}|w_t) = \frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c, j \neq 0} \log p(w_t|w_{t+j})$$

Tips:

CBOW 和 skip-gram 本身都是语言模型，但他们的目的不是建立语言模型，而是使用语言模型来产生词向量。CBOW 是计算给的一个词的序列，中心词为  $w$  的概率。Skip-gram 是给定一个词  $w_i$ ，在一个窗口内可以看到另一个词  $w_j$  的概率。

### 第三节：Keras 实现词表示学习

#### 1. 模型设计

我们再回顾一下神经语言模型，如下

$$p(w_t|h) = \text{softmax(score}(w_t, h)) = \frac{\exp\{\text{score}(w_t, h)\}}{\sum_{\text{word } w' \text{ in } V} \exp\{\text{score}(w', h)\}}$$

它描述了给定一个语境  $h$ ，可以观察到目标词  $w_t$  的概率。 $V$  是词汇表； $\text{score}(w_t, h)$  计算词  $w_t$  和 context（或称作 history, 简写  $h$ ）的相匹配性，通常采用两个向量里的点乘计算。最大似然法训练该模型，最大化目标函数（log 似然）（注：负 log 似然等价于交叉熵损失函数）

$$J_{ML} = \log P(w_t|h) = \text{score}(w_t, h) - \log \left( \sum_{\text{word } w' \text{ in } V} \exp\{\text{score}(w', h)\} \right)$$

然而，该模型的计算代价很高。在训练的每一步需要为  $h$  计算词汇表  $V$  中所有其他词  $w'$  的 score。

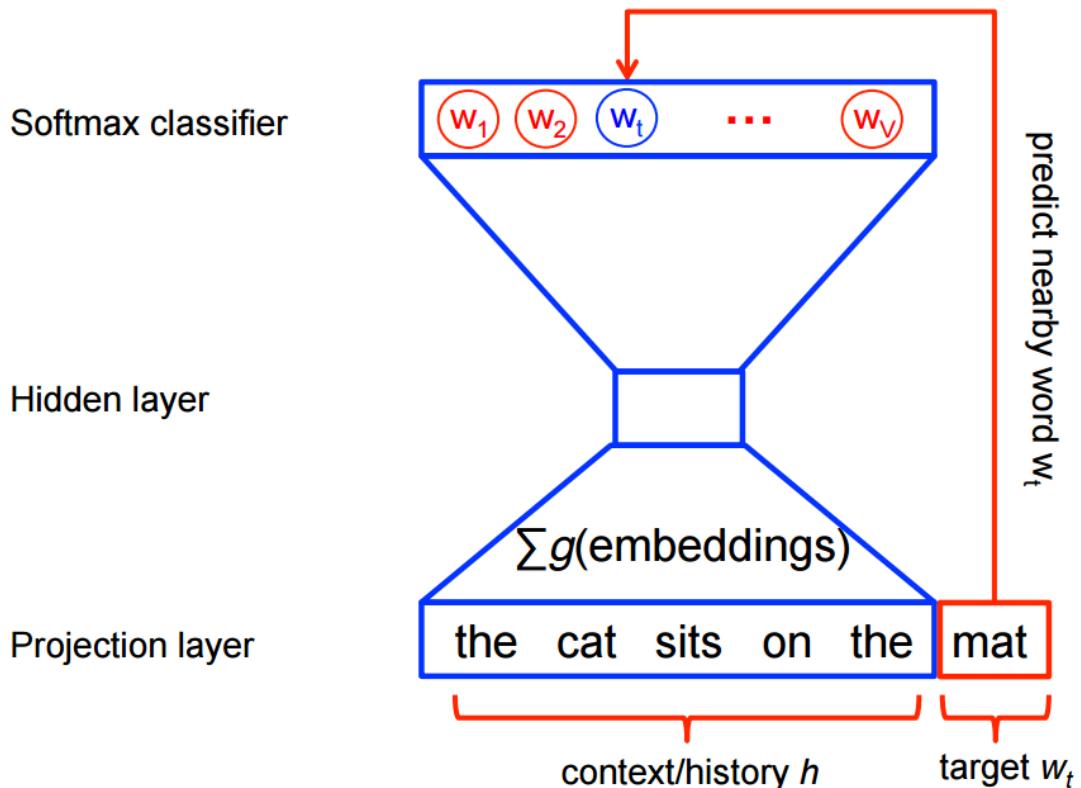


图 6.4. 神经网络语言模型

在 word2vec 的模型学习中，不需要上图的全概率模型。CBOW 和 skip-gram 使用了一个二分类器（logistics 回归）在相同 context 下，区分真实的目标词  $w_t$  的向量和  $k$  个噪声向量  $\tilde{w} \in P$ 。图 6.5 是一个 CBOW 模型。Skip-gram 模型可以理解就是把该图倒置。CBOW 模型的目标函数是

$$J_{NEG} = \log Q_\theta(D=1|w_t, h) + \sum_{\tilde{w} \in P} \log Q_\theta(D=0|\tilde{w}, h)$$

$Q_\theta(D = 1|w_t, h)$  是 Logistics 回归概率，即在数据集  $D$  中给定上下文  $h$  看见词  $w_t$  的概率。该值按照学习到的词向量  $\theta$  来计算。 $Q_\theta(D = 0|\tilde{w}, h) = 1 - Q_\theta(D = 1|\tilde{w}, h)$ 。优化目标就是最大化  $Q_\theta(D = 1|w_t, h)$ ，而最小化  $Q_\theta(D = 1|\tilde{w}, h)$ 。即，给定上下文  $h$  能在  $D$  看见目标词  $w_t$ ，而不会看见  $k$  个噪声词  $\tilde{w}$ 。

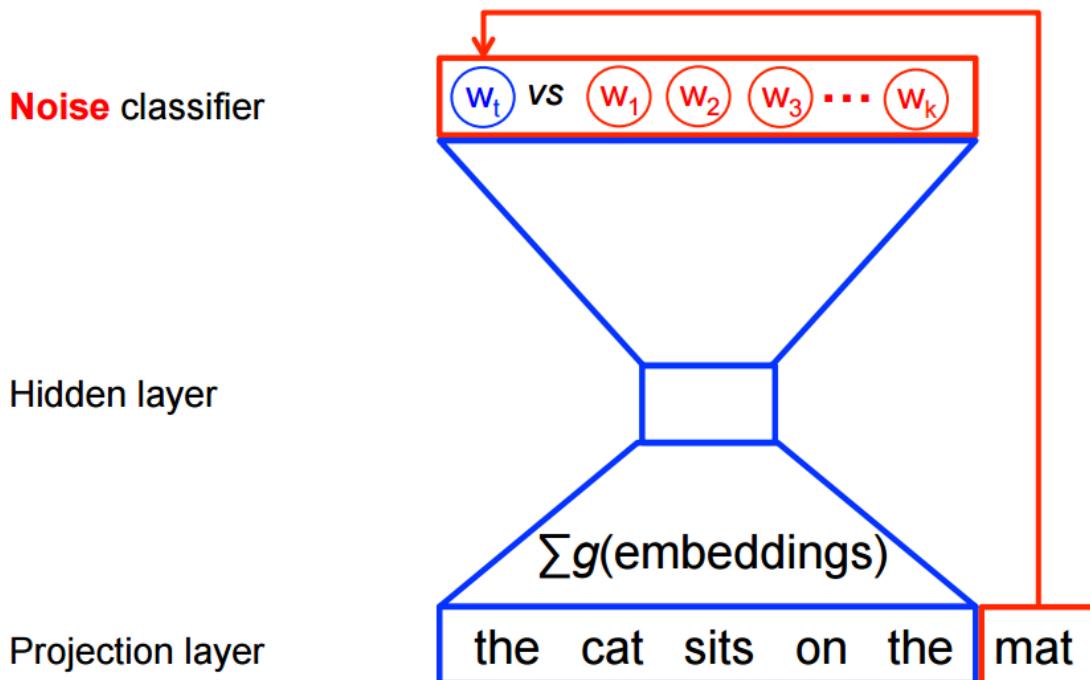


图 6.5. CBOW

在模型的训练过程中，从词汇表随机选择  $k$  个噪声词，即抽样  $k$  个负例。这种方法称作 **negative sampling**。因为不是计算所有的词汇表的词，因此 word2vec 训练速度提高很多。上述计算损失函数的方法和 noise-contrastive estimation (NCE) 理论相似。TensorFlow 提供了 NCE 计算损失函数的方法 `tf.nn.nce_loss()`。（ negative sampling, NCE 都属于 Candidates Sampling，  
[https://tensorflow.google.cn/extras/candidate\\_sampling.pdf](https://tensorflow.google.cn/extras/candidate_sampling.pdf) ）

下面我们将介绍 skip-gram 的 keras 实施。看一个例子

The quick brown fox jumped over the lazy dog.

从这个句子我们建立一个目标词和它的 context 的数据集，即 ‘( context, target )’ 的集合。这里的上下文 context 就是一个目标词左右两边的词。即，一个窗口。设窗口大小为 1。

([the, brown], quick), ([quick, fox], brown), ([brown, jumped], fox), ...

因为 skip-gram 倒置了 context 和 target，试图预测从目标词预测每个 context 词。以上面句子为例，就是从 quick 预测 the 和 brown。因此，skip-gram 的数据集就是元组

((输入，输出), label)

的集合。label 的值为 1，表示这一个词对在一个窗口中共现；为 0 表示没有共现。由词的序列，我们可以产生正例。同样，我们通过负采样产生负例，如此，训练集如下：

((quick, the), 1), ((quick, brown), 1), ((brown, quick), 1), ((quick, sheep), 0), ((quick, world), 0) ...

我们设想一下训练的第 t 步，训练数据是 (quick, the)，目标是从 quick 预测 the。选择 num\_noise 数量个负例。为描述的简单，假设 num\_noise=1，选择了 sheep 作为噪声负例。下面为观察的词对 (quick, the) 和负例词对 (quick, sheep) 计算损失。则在 t 步的目标函数是

$$J_{\text{NEG}}^{(t)} = \log Q_{\theta}(D = 1 | \text{the}, \text{quick}) + \log Q_{\theta}(D = 0 | \text{sheep}, \text{quick})$$

优化的目标是对词向量  $\theta$  做更新，来最大化该目标函数。首先需要获得目标函数的梯度  $\frac{\partial J_{\text{NEG}}}{\partial \theta}$ ，然后沿着梯度的方向更新词向量。当在整个数据集上进行重复进行训练模型，其效果将是每个词移动词向量，直到模型可以成功的区分真实的词和噪声词。

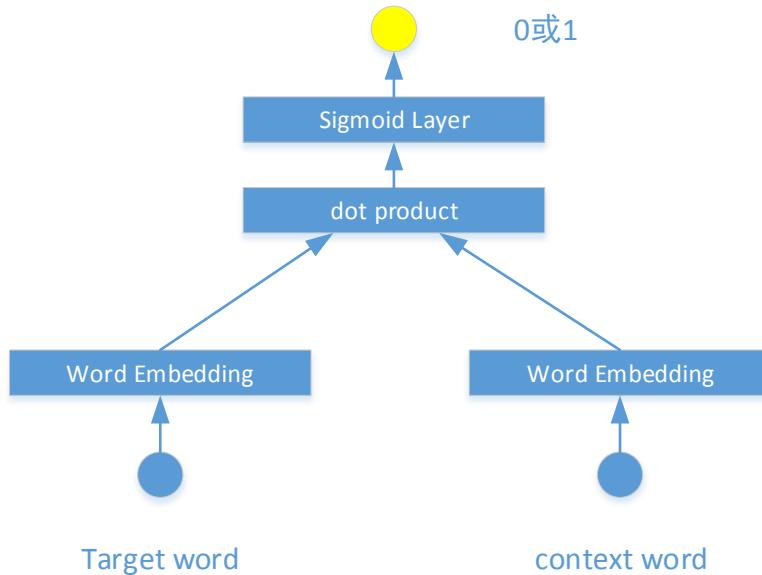


图 6.6 在 keras 中实现的一个 skip-grams 模型

图 6 是我们下面要在 keras 中实现的一个模型。它使用上面产生的数据集，

## 2. 实现 word2vec 的函数

(1) Embeddings 层

Keras 提供了 Embedding 层。它就是前面模型中的查找表。是学习词向量的一个重要环节。

```
keras.layers.Embedding(input_dim, output_dim, embeddings_initializer='uniform',
embeddings_regularizer=None, activity_regularizer=None, embeddings_constraint=None,
mask_zero=False, input_length=None)
```

该层将编码的输入数据，转换为分布式向量（词嵌入）。这个层只能作为模型的第一层。它的主要参数如下：

**input\_dim**：词汇表的大小

**output\_dim**: 词向量的长度

**embeddings\_initializer**: 初始化 embeddings

**embeddings\_regularizer**: 给学习的 embeddings 加上正则化

**activity\_regularizer** : 激活函数正则化

**embeddings\_constraint**: 给 embeddings 加上约束

**input\_length** : 输入序列的长度

Input\_shape: 2D tensor, shape= (batch\_size, sequence\_length)

Output shape: 3D tensor, shape= (batch\_size, sequence\_length, output\_dim)

该层使用的示例代码如下：

```
model = Sequential()
model.add(Embedding(1000, 64, input_length=10))
# the model will take as input an integer matrix of size (batch, input_length).
# the largest integer (i.e. word index) in the input should be
# no larger than 999 (vocabulary size).
# now model.output_shape == (None, 10, 64), where None is the batch dimension.

input_array = np.random.randint(1000, size=(32, 10))

model.compile('rmsprop', 'mse')
output_array = model.predict(input_array)
assert output_array.shape == (32, 10, 64)
```

Embedding 层要求输入数据整数编码，每个词应该对应一个整数编号。Keras 有个 Tokenizer 函数可以完成这个工作。

## ( 2 ) skipgram 函数

```
keras.preprocessing.sequence.skipgrams(sequence, vocabulary_size, window_size=4,
negative_samples=1.0, shuffle=True, categorical=False, sampling_table=None,
seed=None)
```

该函数产生 skipgram 词对。它将一个编号了的词的序列转换成形式如下的元组

- (word, word in the same window), with label 1 (positive samples).
- (word, random word from the vocabulary), with label 0 (negative samples).

它的参数如下：

Sequence: 编号了的词的序列。如果使用了 sampling\_table , word 的下标是和它在数据集中的词频排序匹配的。例如，编号 10 对应第 10 个最频繁的词。

Vocabulary\_size: 应该是 word 最大下标+1

Window\_size: 采样窗口的大小 ( 半窗 ) 。窗口内词的数目是 : [ i - window\_size, i + window\_size+1 ]

Negative\_sampling: 大于 0 的浮点数。0 表示不负采样 , 1 表示抽样和正例一样多的负例样本。

Shuffle: 是否重新打乱产生的元组的序列。

Categorical: 如果为 False , 标签的形式是整数 , [0, 1, 1 .. ] 。如果为 True , 标签的形式是 [[1,0],[0,1],[0,1] .. ]

Sampling\_table: 大小为 vocabulary\_size 的一维矩阵。其元素值是 word 被采样的概率  
返回结果 :

Couples, label。 Couples 是词对 , label 是类别标签

( 3 ) 产生采样表的函数

```
keras.preprocessing.sequence.make_sampling_table(size, sampling_factor=1e-05)
```

它是作为其他函数的参数产生一个词的概率采样表。

计算公式如下 :

```
p(word) = (min(1, sqrt(word_frequency / sampling_factor) /  
(word_frequency / sampling_factor)))
```

### 3. 模型的 keras 实现

( 1 ) 产生数据集

```
def build_dataset(filename, n_words):  
    """Process raw inputs into a dataset."""  
    with zipfile.ZipFile(filename) as f:  
        words = tf.compat.as_str(f.read(f.namelist()[0])).split()
```

```

count = [['UNK', -1]]
count.extend(collections.Counter(words).most_common(n_words - 1))
dictionary = dict()
for word, _ in count:
    dictionary[word] = len(dictionary)
data = list()
unk_count = 0
for word in words:
    index = dictionary.get(word, 0)
    if index == 0: # dictionary['UNK']
        unk_count += 1
    data.append(index)
count[0][1] = unk_count
reversed_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
return data, count, dictionary, reversed_dictionary

filename = 'text8.zip'
vocabulary_size = 50000
data, count, dictionary, reverse_dictionary = build_dataset(
    filename, vocabulary_size)

```

具体产生数据集的代码不讲了，自己看源程序。产生的 data 是编号了的词的序列； dictionary 是词到编号的映射； reverse\_dictionary 是编号到词的映射。

### ( 2 ) 设置模型的参数

```

window_size = 1
vector_dim = 300
epochs = 10000
batch_size=1000
vocab_size=len(dictionary)

valid_size = 16 # Random set of words to evaluate similarity on.
valid_window = 100 # Only pick dev samples in the head of the distribution.
valid_examples = np.random.choice(valid_window, valid_size, replace=False)

```

窗口大小为 1，词向量长度为 300。在训练集上的训练了次数 epochs。Valid\_size 是训练完模型后，演示词向量之间相似度抽取的校验集的大小。

### ( 3 ) 产生训练集

```

couples, labels = skipgrams(data, vocab_size, window_size=window_size ,
negative_samples=0.1)
word_target, word_context = zip(*couples)
word_target = np.array(word_target, dtype="int32")
word_context = np.array(word_context, dtype="int32")

```

产生 ( target, context ) 元组集合和对应的标签 ( 0 或 1 ) 的集合。负采样率是 0.1.

### ( 3 ) 建立模型

```

input_target = Input((1,))
input_context = Input((1,))

embedding = Embedding(vocab_size, vector_dim, input_length=1,
name='embedding')

target = embedding(input_target)
target = Reshape((vector_dim, 1))(target)
context = embedding(input_context)
context = Reshape((vector_dim, 1))(context)
dot_product = dot([target, context], axes=1, normalize=False)
dot_product = Reshape((1,))(dot_product)

similarity = dot([target, context], axes=1, normalize=False)

# add the sigmoid output layer
output = Dense(1, activation='sigmoid')(dot_product)

# create the primary training model
model = Model(inputs=[input_target, input_context], outputs=output)
model.compile(loss='binary_crossentropy', optimizer='adam')

# create a secondary validation model to run our similarity checks during training
validation_model = Model(inputs=[input_target, input_context],
outputs=similarity)

```

分别为，目标词和 context word 创建一个输入层。然后创建一个查找表 embedding。

target = embedding(input\_target)从查找表获得目标词的词向量。

target = Reshape((vector\_dim, 1))(target)是将词向量转换成列向量。同样的也对 context word 进行相同的操作。

Target word 和 context word 的词向量进行点乘。此时使用的 dot 函数是 keras 提供的 Merge 层中的一个操作。具体见附录的第六节，或见帮助文档

<https://keras.io/layers/merge/>

点乘结果得到一个数值，经过一个 sigmoid 层后，把该数值转换成 0-1 之间的一个值。

因为是二分类，采用了 binary\_crossentropy 损失函数；采用 adam 优化器。

validation\_model 模型是在校验阶段，用该模型来寻找和一个词最相似的其他词。后面可以看到，该模型没有训练。

#### ( 4 ) 考察词向量

```

class SimilarityCallback:
    def run_sim(self):

```

```

for i in range(valid_size):
    valid_word = reverse_dictionary[valid_examples[i]]
    top_k = 8 # number of nearest neighbors
    sim = self._get_sim(valid_examples[i])
    nearest = (-sim[:,0,0]).argsort()[1:top_k + 1]
    log_str = 'Nearest to %s:' % valid_word
    for k in range(top_k):
        close_word = reverse_dictionary[nearest[k]]
        log_str = '%s %s,' % (log_str, close_word)
    print(log_str)

def _get_sim(self,valid_word_idx):
    in_arr1 = np.ones((vocab_size,))*valid_word_idx
    in_arr2 = np.arange(vocab_size)
    sim = validation_model.predict_on_batch([in_arr1, in_arr2])
    return sim
sim_cb = SimilarityCallback()

```

该类用于考察和一些词，最相似的的词的集合

### ( 5 ) 训练模型

```

ndlables=np.array(labels)
for cnt in range(epoches):
    idx = random.sample(range(vocabulary_size),batch_size)
    loss = model.train_on_batch([word_target[idx],word_context[idx]],
                               ndlables[idx])
    if cnt % 1000 == 0:
        print("Iteration {}, loss={}".format(cnt, loss))

```

我们前面讲的在 keras 中训练模型用的是模型对象的 fit 函数。模型对象还有 train\_on\_batch 函数

`train_on_batch(x, y, sample_weight=None, class_weight=None, reset_metrics=True)`

它实现每个 batch 的训练数据训练完后，更新参数。该函数的参数如下：

x, 是 numpy array 类型的训练数据集

y, numpy array 类型的目标值

使用 train\_on\_batch 可以方便的考察训练过程。比如，此处自定义地想考察最相似的 words，用 fit 函数来训练就不能实现。

### ( 5 ) 考察词向量

`sim_cb.run_sim()`

即调用前面创建的 SimilarityCallback 类的对象 sim\_cb，来寻找和校验集中的词最相似的词。



# 第七章：基于 CNN 的文本分类

## 第一节：Keras 的文本处理

keras 提供了几个文本处理的类。

### 7.1.1 Tokenizer 类

```
tensorflow.keras.preprocessing.text.Tokenizer(num_words=None,  
filters='!"#$%&()*+,-./:;=>?@[\\]^_`{|}~\\t\\n', lower=True, split=' ',  
char_level=False, oov_token=None, document_count=0)
```

[https://www.tensorflow.org/api\\_docs/python/tf/keras/preprocessing/text/Tokenizer](https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer)

该类可以把一篇文档或者转换成整数的序列，或者转换成一个向量。向量的元素可以是，0-1 值，或者词的计数或者是 tfidf。该函数的主要参数有

num\_words : 要保留的最大的词的数量。基于词频，最频繁的前 num\_words-1 个词被保存。

filters: 需要被过滤掉的字符

lower: 是否将文本都转化成小写

split : 分割符

char\_level: 如果为 True，按照字符进行处理，否则是词

oov\_token: 词汇表之外的词，用 该符号替代。例如，给“UNK”。也可以不给。不给的话所有词汇表外的词会被移走。

“0” 是默认保留的索引编号，不会分配给其他词。

该类提供的主要方法

```
1. fit_on_texts(  
    texts  
)
```

Updates internal vocabulary based on a list of texts. In the case where texts contains lists, we assume each entry of the lists to be a token. Required before

using `texts_to_sequences` or `texts_to_matrix`. `texts`: can be a list of strings, a generator of strings (for memory-efficiency), or a list of list of strings.

```
2. texts_to_sequences(  
    texts  
)
```

`texts`: A list of texts (strings). 使用该方法前，需要先使用 `fit_on_texts` 方法在数据集上进行了预处理。如此处理后的 `tokenizer` 对象再调用 `texts_to_sequences` 方法，才可以将文档转换成编号的序列。未出现在 `Tokenizer` 类中的。如果应用 `texts_to_sequences` 方法在一个新的文档集合上时（做预测时，面对的是新的文档集合），为出现在 `tokenizer` 对象中的词会被移除或用 `oov_token` 替换。第二节给出了具体使用。

该类还有一些属性，它们都是在执行了 `fit_on_texts` 后，在当前文档集合上的一些统计信息。注意，这些统计信息不受创建 `Tokenizer` 对象时的参数 `num_words` 的影响。

#### 1. word\_counts

在执行了方法 `fit_on_texts` 后，该属性展示文档集合的词汇表。包括词项和它在文档集合里的词频。

#### 2. document\_count

文档数

#### 3. word\_index

一个词典，每个词和它的编号

#### 4. word\_docs

每个词出现的文档数

例如：

```
docs = ['Well done!',  
        'Good work',  
        'Great effort',  
        'nice work',  
        'Excellent!',  
        'Weak',  
        'Poor effort!',  
        'not good',  
        'poor work',  
        'Could have done better.'][  
t=Tokenizer()  
t.fit_on_texts(docs)  
print(t.texts_to_sequences(docs))
```

```
print(t.word_counts)
print(t.document_count)
print(t.word_index)
print(t.word_docs)
```

### 7.1.2 one-hot

```
keras.preprocessing.text.one_hot(text, n,
filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\\t\\n', lower=True, split=' ')
```

One-hot encodes a text into a list of word indexes of size n。它是对 keras.preprocessing.text.hashint\_track 类的封装，用 hash 函数对每个词项产生一个编码。

参数：

```
text: Input text (string).
n: int. Size of vocabulary.
filters: list (or concatenation) of characters to filter out,
such as
    punctuation. Default:
``!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\\t\\n``,
    includes basic punctuation, tabs, and newlines.
lower: boolean. Whether to set the text to lowercase.
split: str. Separator for word splitting.
```

```
vocab_size = 50
encoded_docs = [one_hot(d, vocab_size) for d in docs]
print(encoded_docs)
```

### 7.1.3 text\_to\_word\_sequence

```
keras.preprocessing.text.text_to_word_sequence(text,
filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\\t\\n', lower=True, split=' ')
```

将文本转换成一个词项序列。它就是一个英文分词工具。参数如下：

text: Input text (string).

filters: list (or concatenation) of characters to filter out, such as

punctuation. Default: ``!"#\$%&()\*+,-./:;<=>?@[\\]^\_`{|}~\\t\\n``,  
includes basic punctuation, tabs, and newlines.

lower: boolean. Whether to convert the input to lowercase.

split: str. Separator for word splitting.

例如：

```

from tensorflow.keras.preprocessing.text import text_to_word_sequence
text = 'The quick brown fox jumped over the lazy dog.'
result = text_to_word_sequence(text)
print(result)

```

## 第二节：一个简单的文本分类模型

下面我们介绍使用 keras 构建的一个简单的基于神经网络的分类模型。

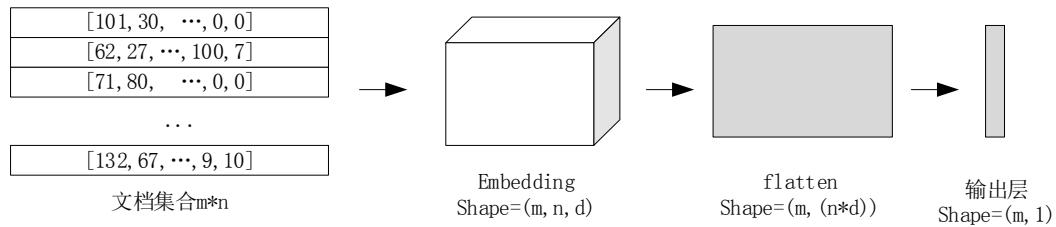


图 7-1：一个简单的文本分类模型

设  $\text{batch\_size}=m$ ，当前文档集合最长的文档的长度是  $n$ ，词向量的长度是  $d$ 。喂入模型的文档集合应该是一个 list 数据结构。而 list 中的每个元素是一篇文档，它又是一个 list。

```

t=Tokenizer()
t.fit_on_texts(docs)
encoded_docs=t.texts_to_sequences(docs)
vocab_size = len(t.word_index)+1

```

`docs` 是还未处理的文档集合。使用 `Tokenizer` 类的 `fit_on_texts` 方法处理该文档集合，然后用 `texts_to_sequences` 方法，将每篇文档转换成编号序列。每篇文档需要处理成等长的 list。每个元素是一个词的编号，短的文档用“0”补齐。例如：`[132,67,...,9,0]`

```

max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length,
padding='post')
print(padded_docs)

```

`pad_sequences`(`encoded_docs`, `maxlen=max_length`, `padding='post'`)方法对编码后的文档集合，进行补齐。`Maxlen` 规定了最长的文档长度，`padding='post'` 设定在向量的后面进行补“0”。

模型的第一个层就是 `Embedding` 层，每个词用一个词向量来表示。然后得到一个 `tensor`，它的 `shape=(m,n,d)`。此时一篇文档是一个  $n*d$  的矩阵，为了使用全连接层，把一篇文档转换成一个向量。即构建了一个 `flatten` 层。最后的输出层就一个神经元，它需要判断一篇文档是不是属于某个类别，是个二分类问题，因此输出层采用 `sigmoid` 激活函数。

```
model = Sequential()
model.add(Embedding(vocab_size, 8, input_length=max_length))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
```

训练模型，再在训练集上考察模型拟合

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
print(model.summary())
model.fit(padded_docs, labels, epochs=50, verbose=0)
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

新来了文本，对它们进行分类。

```
# prediction
docs2 = ['weak poor situation',
          'Good morning,done']
encoded_docs2=t.texts_to_sequences(docs2)
padded_docs2 = pad_sequences(encoded_docs2, maxlen=max_length,
                             padding='post')
res = model.predict(padded_docs2)
print(res)
```

此处是用前面在训练集上拟合的 Tokenizer 对象对新的文档 docs2 进行处理。前面我们创建 tokenizer 对象时没设定 oov\_token 属性，则 out-of-vocabulary 的词 texts\_to\_sequences(docs2)会把这些词移除。然后把新的文档转换成了编号序列。再使用 pad\_sequences 进行补“0”。再在处理后的数据集上进行预测。预测结果  
[[0.47857347]  
 [0.5546594 ]]

返回的是每篇文档对于类别“1”的概率。

注：如果创建 Tokenizer 对象时设定了 t=Tokenizer(oov\_token='UNK')则，词汇表外的词会被分配编号 1，即‘UNK’

完整的代码如下：

```
from numpy import array
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Embedding
from tensorflow.keras.preprocessing.text import Tokenizer

# define documents
docs = ['Well done!',
        'Good work',
```

```

'Great effort',
'nice work',
'Excellent!',
'Weak',
'Poor effort!',
'not good',
'poor work',
'Could have done better.']

# define class labels
labels = array([1,1,1,1,0,0,0,0])

t=Tokenizer()
t.fit_on_texts(docs)
encoded_docs=t.texts_to_sequences(docs)
vocab_size = len(t.word_index)+1

# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length,
padding='post')
print(padded_docs)
# define the model
model = Sequential()
model.add(Embedding(vocab_size, 8, input_length=max_length))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# summarize the model
print(model.summary())
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))

# prediction
docs2 = ['weak poor situation',
         'Good morning,done']
encoded_docs2=t.texts_to_sequences(docs2)
padded_docs2 = pad_sequences(encoded_docs2, maxlen=max_length,
padding='post')
res = model.predict(padded_docs2)
print(res)

```

### 第三节： CNN 文本分类模型

#### 7.3.1 模型结构

一个详细描述的文本分类 CNN 结构图见图 7.1 ( 详见论文 A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification ) 。图 7.3 的 CNN 的结构是文本分类的结构示例图。具体实施时 , 使用的一些参数不太一样 , 如滤波器的 region size , 滤波器的个数等。在这篇论文中构建的 CNN 是针对句子进行分类。这里其实是用句子指代短文本 , 如评论数据。该论文构建的是一个评论的情感分类器。该深度模型如下 :

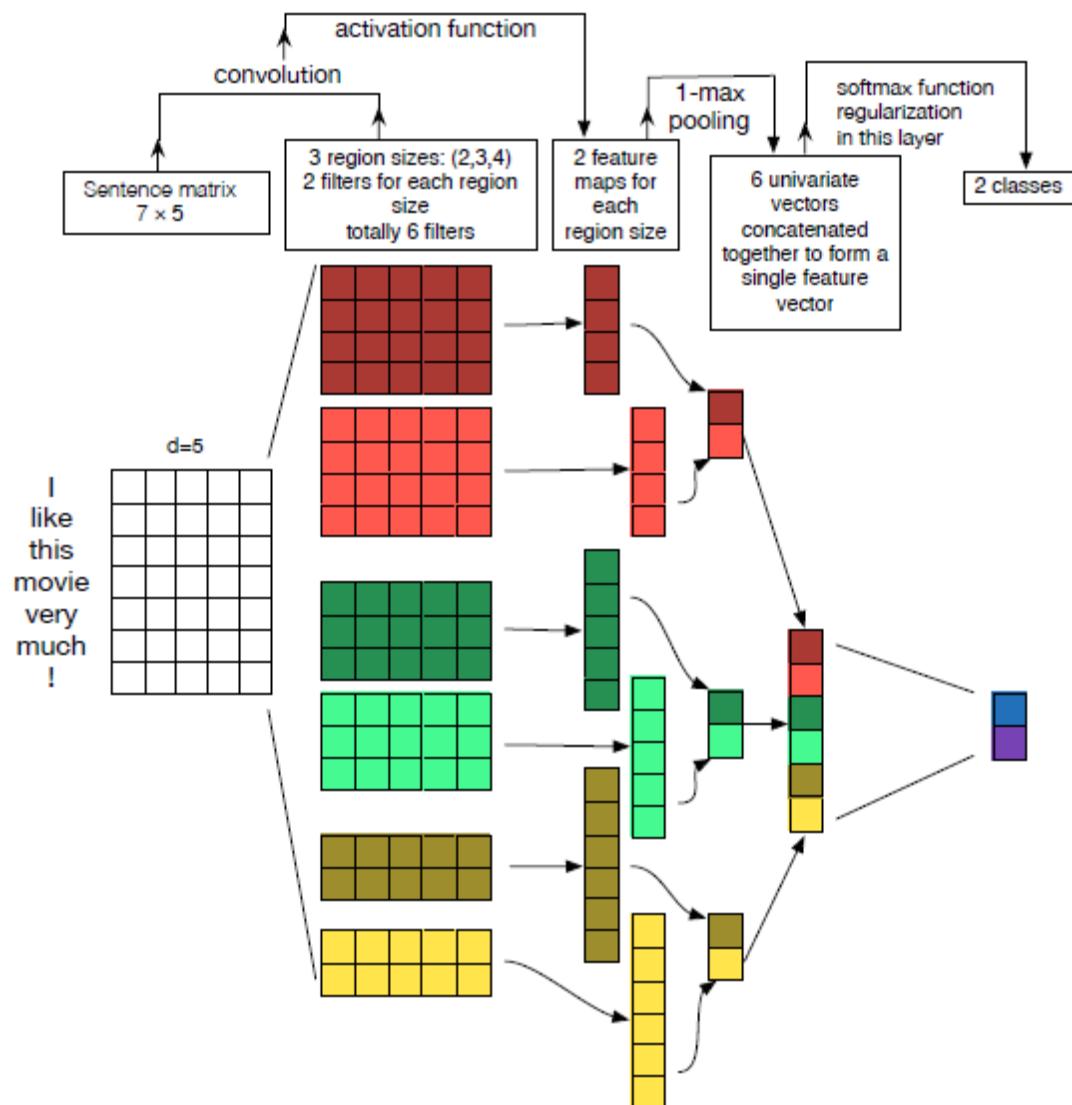


图 7.1 有多个 window 的滤波器的文本分类 CNN

( 1 ) 输入层 : 该模型的输入是一个句子 ; 句子中的每个词用词向量表示。输入的词向量维度为  $k$ 。设  $x_i \in R^k$  是句子中第  $i$  个词的  $k$  维词向量。一个长度为  $n$  的句子被描述成

$$x_{1:n} = x_1 \oplus x_2 \oplus \dots \oplus x_n$$

这里  $\oplus$  是拼接操作符 , 不是连接成长向量 , 而是拼接成一个矩阵。此文中  $x_{i:i+j}$  是指词向量  $x_i, x_{i+1}, \dots, x_{i+j}$  的拼接操作。该文提到 padded where necessary , 是说如果句子长

度不足  $n$  用零填充， $n$  为数据集中最长的文本长度。在前一章讲 CNN 就行图像处理时，一张图片的高宽是固定的，这里也将一个句子转换成了一个高宽固定的矩阵（高是  $n$ ，宽是词向量的长度）。

在前一章中讲到图片有三个通道（RGB），因此我们可以描述成输入层有三个特征映射。这里为了处理文本，建立了两个通道或特征映射。输入的句子的每个词的词向量获得有两种方式。一是 Mikolov 使用它的 word2vec 在 Google News dataset 上训练词向量。产生的词向量的维度是 300，包含三百万个词和词组。这个是公开的词向量集合 (<https://code.google.com/archive/p/word2vec/>)。第二个是如前一章所示，将词向量也作为 CNN 模型中的参数，在 CNN 的分类模型中经过训练后得到一组词向量。Yoon Kim 在输入层，或 embeddings 层，建立两种词向量的输入，于是得到输入层有两个特征映射，第一个称为 static channel; 第二个称为 non-static channel。

(2) 卷积层：在输入层上加一个卷积层。一个滤波器  $w \in R^{h*k*d}$  的一个窗口大小是  $h*k$ ， $h$  是词的个数， $k$  是词向量的长度， $d$  是滤波器的 depth，见图 7.1。从一个滤波器窗口产生一个特征  $c_i$ 。

$$c_i = f(W \cdot x_{i:i+h-1} + b)$$

$b$  是偏置。如此产生的一个特征映射是一个向量  $c=[c_1, c_2, \dots, c_{n-h+1}]$ 。卷积层还有个参数 region size。在每个 region size 上可以设定多个滤波器。如果 region size 是  $m$ ，每个 region size 上滤波器的个数是  $n$ ，则实际卷积层上有  $m*n$  个滤波器。这里的滤波器又是一个 volume 结构，volume 的 depth 和输入的特征映射的个数相同，在多个输入特征映射上的卷积操作后求和，与第五章描述的卷积层操作相同。图 7.1 中有  $2*3=6$  个滤波器（每个 region size 两个滤波器，一共 3 个 region size）。

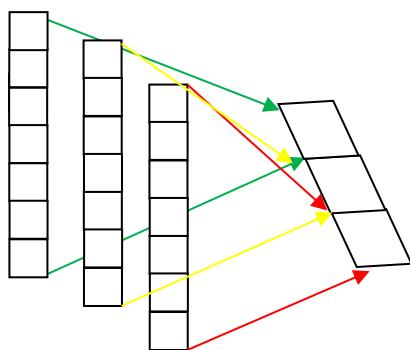


图 7.2 pooling 层

(3) dropout：在卷积层的输出加 dropout 操作。

( 3 ) pooling 层：子采样操作应用的是 maxpooling 操作。简单地说就是子采样时的滤波器采样最大值的方法 $\hat{c} = \max(C)$ 进行子采样；滤波器的 shape 是和一个特征映射的 shape 一致。其思想是捕获特征映射中最重要的特征。因此，其结果是一个 feature map 子采样操作后得到的是一个值。每个子采样操作的结果拼接成一个向量，构成 pooling 层的输出。如图 7.2 所示。

( 4 ) softmax 输出层：输出层有两个节点，即二分类的结果。Pooling 层和输出层采用全连接。（注：我们的实际实施时，就是一个节点，输出的是 0-1 的概率）

在实践中，该 CNN 模型在卷积层给出三种 region size，：[3, 4, 5]。在每种 region size 上建立 100 个滤波器。

### 7.3.2. keras 实施

#### 1. 数据准备

使用电影评论数据 sentence polarity dataset v1.0

(( <http://www.cs.cornell.edu/people/pabo/movie-review-data/> )

该数据集包括两个文件一个正向评论数据和一个负向评论数据。各包含 5331 条评论，每条评论占一行。每条评论视为一个句子。该数据由 Pang/Lee 创建，在 ACL 2005 的论文中使用。因此文本分类（句子分类）的任务即判断一条评论的情感倾向正向或负向。将两个文件合并产生数据和标签（在 data\_helper.py 中）。

在 cnn-text.py 中调用下面的 data\_helpers.load\_data\_and\_labels() 函数。可以获得数据集。

```
x_text, y = data_helpers.load_data_and_labels()  
y=np.argmax(y, axis=1, out=None)
```

此时的 y 即类别标签。它现在是[[0,1],[0,1],...,[1,0]]这种 one-hot 形式。我们的模型需要的是类别标签列表的形式，[0,0,...,1]。y=np.argmax(y, axis=1, out=None)则将 one-hot 形式，转化成类别标签列表的形式。

当前的一篇文档，即 x\_text 中的一行，是一个字符串。我们需要将它转换成词的编号的序列。且每篇文档的长度一致。

```
t=Tokenizer(num_words=vocab_size, oov_token=None)  
t.fit_on_texts(x_text)  
encoded_docs=t.texts_to_sequences(x_text)  
doc_length = max([len(x) for x in encoded_docs])  
x = pad_sequences(encoded_docs, maxlen=doc_length, padding='post')
```

下面的代码划分训练集和测试集

```

np.random.seed(10)
shuffle_indices = np.random.permutation(np.arange(len(y)))
x_shuffled = x[shuffle_indices]
y_shuffled = y[shuffle_indices]

# Split train/test set
x_train, x_dev = x_shuffled[:-1000], x_shuffled[-1000:]
y_train, y_dev = y_shuffled[:-1000], y_shuffled[-1000:]

```

## 2. 构建模型

创建输入层。Keras 的层不需要考虑 batch\_size。输入层的 shape 因此是一篇文档的长度。然后创建 word embeddings 的查找表。Embedding 函数创建查找表时，需要参数：词汇表的大小 ( input\_dim )、词向量的维度 ( output\_dim )。然后使用查找表把输入转换成 shape=(文档长度,词向量长度)的 tensor。

```

n,m = x_train.shape[0], x_train.shape[1]
inputs=Input((m,))
embed=Embedding(input_dim=vocab_size, output_dim=embed_size,
input_length=doc_length)
hidden=embed(inputs)

```

我们采用 CNN 来进行文本分类。用 Conv2D 函数实施卷积操作，它要求的输入的 tensor 的 shape=(samples, row, col, channel)。当前例子中，row 即文档长度，col 即词向量长度，channel=1。我们因此使用 Reshape 层，转换 tensor 的 shape。

```
reshape=Reshape((m,embed_size,1), input_shape=(m,embed_size))(hidden)
```

下面这段代码实现有多个 region size 的滤波器的卷积操作

```

pooled_output=[]

for fsize in filter_sizes:
    conv=Conv2D(filters=filter_nums,
               kernel_size=(fsize,embed_size),
               activation='relu',
               padding="valid")(reshape)
    pool=MaxPool2D(pool_size=(m-fsize+1,1))(conv)
    drop=Dropout(0.5)(pool)
    reshaped=Reshape((filter_nums,), input_shape=(1,1,filter_nums))(drop)
    pooled_output.append(reshaped)

```

前面设定了三种 filter\_size，filter\_sizes=[3,4,5]。因此用了一个循环，为每一种 size 创建一个卷积层，然后把卷积层的输出，经过 reshape 后。放到一个 list 结构 pooled\_output 中。Reshape 的操作把卷积层的多个特征映射（一个特征映射的

shape=(1,1) ) 拼接成了一个向量。Concatenate 是 keras 一个 Merge Layer 的一个操作。

下面的代码把多个卷积层的输出又拼接成了一个层 hidden2 , 它的 shape=(batch\_size, 3\*filter\_nums)。然后 , 再加上一个输出层。

```
hidden2=Concatenate()(pooled_output)
output=Dense(1, activation='sigmoid')(hidden2)
model = Model(inputs=inputs, outputs=output)
```

最后是模型训练参数的设定 , 模型训练 , 模型评估

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
print(model.summary())
model.fit(x_train, y_train, epochs=10, verbose=1)

# evaluate the model
loss, accuracy = model.evaluate(x_dev, y_dev, verbose=0)
print('Accuracy: %f' % (accuracy))
```

完整的代码如下 :

```
import data_helpers
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Embedding
from tensorflow.keras.layers import Reshape
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPool2D
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Concatenate
from tensorflow.keras.layers import Dropout

# parameters
filter_sizes=[3,4,5]
filter_nums=10
embed_size = 50
vocab_size = 10000

x_text, y = data_helpers.load_data_and_labels()
y=np.argmax(y, axis=1, out=None)

# encoding
t=Tokenizer(num_words=vocab_size,oov_token=None)
t.fit_on_texts(x_text)
```

```

encoded_docs=t.texts_to_sequences(x_text)
doc_length = max([len(x) for x in encoded_docs])
x = pad_sequences(encoded_docs, maxlen=doc_length, padding='post')

# Randomly shuffle data
np.random.seed(10)
shuffle_indices = np.random.permutation(np.arange(len(y)))
x_shuffled = x[shuffle_indices]
y_shuffled = y[shuffle_indices]

# Split train/test set
x_train, x_dev = x_shuffled[:-1000], x_shuffled[-1000:]
y_train, y_dev = y_shuffled[:-1000], y_shuffled[-1000:]

# build model
n,m = x_train.shape[0], x_train.shape[1]
inputs=Input((m,))
embed=Embedding(input_dim=vocab_size, output_dim=embed_size,
input_length=doc_length)
hidden=embed(inputs)
reshape=Reshape((m,embed_size,1), input_shape=(m,embed_size))(hidden)
pooled_output=[]

for fsize in filter_sizes:
    conv=Conv2D(filters=filter_nums,
                kernel_size=(fsize,embed_size),
                activation='relu',
                padding="valid")(reshape)
    pool=MaxPool2D(pool_size=(m-fsize+1,1))(conv)
    drop=Dropout(0.5)(pool)
    reshaped=Reshape((filter_nums,), input_shape=(1,1,filter_nums))(drop)
    pooled_output.append(reshaped)

hidden2=Concatenate()(pooled_output)
output=Dense(1, activation='sigmoid')(hidden2)
model = Model(inputs=inputs, outputs=output)
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
print(model.summary())
model.fit(x_train, y_train, epochs=10, verbose=1)

# evaluate the model
loss, accuracy = model.evaluate(x_dev, y_dev, verbose=0)
print('Accuracy: %f % (accuracy)')

```

### 7.3.3. 加入预训练词向量的 CNN 文本分类模型

7.3.2 节的分类模型中使用的词向量也是模型的参数。本节我们将 GloVe 预训练的词向量作为一个新的 channel。详见 <http://nlp.stanford.edu/projects/glove/>

思路是：读入预训练的 embedding。根据当前词汇表，即每个词和他的编号。创建一个矩阵，其第 i 行，即编号为 i 的词项的一个 embedding。创建一个 Embedding 对象，即查找表，使用该矩阵作为初始化的值。

```
embeddings_index = {}
with open(pname, encoding="utf8") as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, 'f', sep=' ')
        embeddings_index[word] = coefs
```

上面的代码将预训练的 word embedding 读进，其一行是即词和词向量的值，用空格做分隔，形式如下：

```
good -0.54403 0.60274 -...
```

读进的词向量放入了词典 embeddings\_index 中保存。

```
embedding_matrix = np.zeros((vocab_size, embed_size))
for word, i in t.word_index.items():
    if i >= vocab_size:
        continue
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

这一段代码参考，已经从文本集合建立的词典，创建一个当前读进的预训练的词向量，用矩阵来表示。矩阵的一行是一个词向量，行号对应词典中对应的词的编号。

t.word\_index 是在文档集合上拟合的 Tokenizer 对象，它的词典。

embeddings\_index.get(word) 从读入的预训练的词向量中查找词 word 对应的词向量。然后放入 embedding\_matrix。

在建立模型时

```
n,m = x_train.shape[0], x_train.shape[1]
inputs=Input((m,))
embed=Embedding(input_dim=vocab_size, output_dim=embed_size,
input_length=doc_length)
preembed = Embedding(input_dim=vocab_size, output_dim=embed_size,
                     embeddings_initializer=Constant(embedding_matrix),
                     input_length=doc_length,
                     trainable=False)
embed_layer1=embed(inputs)
reshape1=Reshape((m,embed_size,1), input_shape=(m,embed_size))
(embed_layer1)

embed_layer2=preembed(inputs)
reshape2=Reshape((m,embed_size,1), input_shape=(m,embed_size))
(embed_layer2)
hidden=Concatenate()([reshape1,reshape2])
```

创建预训练的词向量的查找表，即 Embedding 对象。

```
preembed = Embedding(input_dim=vocab_size, output_dim=embed_size,
                     embeddings_initializer=Constant(embedding_matrix),
                     input_length=doc_length,
                     trainable=False)
```

它用刚才创建的 embedding\_matrix 来初始化该查找表，另外设定 trainable=False，表示该查找表不需要训练。默认 trainable=True。

根据两种查找表（预训练的，和本程序中需要训练的），将输入分别转换成两个 tensor。再将两个 tensor 进行 reshape 操作后，拼接成一 Tensor

```
hidden=Concatenate()([reshape1,reshape2])
```

完整的代码如下：

```
import data_helpers
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Embedding
from tensorflow.keras.layers import Reshape
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPool2D
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Concatenate
from tensorflow.keras.layers import Dropout
from tensorflow.keras.initializers import Constant

# parameters
filter_sizes=[3,4,5]
filter_nums=10
embed_size = 100 # 25 50 100 200
vocab_size = 10000
pname='D:/qjt/data/glove.twitter.27B/glove.twitter.27B.%sd.txt'%(embed_size)

# load text
x_text, y = data_helpers.load_data_and_labels()
y=np.argmax(y, axis=1, out=None)

# encoding
t=Tokenizer(num_words=vocab_size,oov_token=None)
t.fit_on_texts(x_text)
encoded_docs=t.texts_to_sequences(x_text)
doc_length = max([len(x) for x in encoded_docs])
x = pad_sequences(encoded_docs, maxlen=doc_length, padding='post')

# Randomly shuffle data
```

```

np.random.seed(10)
shuffle_indices = np.random.permutation(np.arange(len(y)))
x_shuffled = x[shuffle_indices]
y_shuffled = y[shuffle_indices]

# Split train/test set
x_train, x_dev = x_shuffled[:-1000], x_shuffled[-1000:]
y_train, y_dev = y_shuffled[:-1000], y_shuffled[-1000:]

# load pretrained embeddings
embeddings_index = {}
with open(pname, encoding="utf8") as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, 'f', sep=' ')
        embeddings_index[word] = coefs

embedding_matrix = np.zeros((vocab_size, embed_size))
for word, i in t.word_index.items():
    if i >= vocab_size:
        continue
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector

# build model
n,m = x_train.shape[0], x_train.shape[1]
inputs=Input((m,))
embed=Embedding(input_dim=vocab_size, output_dim=embed_size,
input_length=doc_length)
preembed = Embedding(input_dim=vocab_size, output_dim=embed_size,
                     embeddings_initializer=Constant(embedding_matrix),
                     input_length=doc_length,
                     trainable=False)
embed_layer1=embed(inputs)
reshape1=Reshape((m,embed_size,1),
input_shape=(m,embed_size))(embed_layer1)

embed_layer2=preembed(inputs)
reshape2=Reshape((m,embed_size,1),
input_shape=(m,embed_size))(embed_layer2)
hidden=Concatenate()([reshape1,reshape2])

pooled_output=[]

for fsize in filter_sizes:
    conv=Conv2D(filters=filter_nums,
               kernel_size=(fsize,embed_size),
               activation='relu',
               padding="valid")(hidden)
    pool=MaxPool2D(pool_size=(m-fsize+1,1))(conv)
    drop=Dropout(0.5)(pool)

```

```

reshaped=Reshape((filter_nums,), input_shape=(1,1,filter_nums))(drop)
pooled_output.append(reshaped)

hidden2=Concatenate()(pooled_output)
output=Dense(1, activation='sigmoid')(hidden2)
model = Model(inputs=inputs, outputs=output)
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
print(model.summary())
model.fit(x_train, y_train, epochs=10, verbose=1)

# evaluate the model
loss, accuracy = model.evaluate(x_dev, y_dev, verbose=0)
print('Accuracy: %f' % (accuracy))

```

### 7.3.4 使用 1DConv 构建的文本分类器

下面的程序是使用 keras 提供的一个 1DConv 文本分类的例子

[https://keras.io/examples/imdb\\_cnn/](https://keras.io/examples/imdb_cnn/)

应用在我们的数据集上的程序。大家自己运行比较模型性能。

```

import data_helpers
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation
from tensorflow.keras.layers import Embedding
from tensorflow.keras.layers import Conv1D, GlobalMaxPooling1D

# set parameters:
batch_size = 32
embedding_dims = 50
filters = 250
kernel_size = 3
hidden_dims = 250
epochs = 2
vocab_size=5000

print('Loading data...')
x_text, y = data_helpers.load_data_and_labels()
y=np.argmax(y, axis=1, out=None)

# encoding
t=Tokenizer(num_words=vocab_size,oov_token=None)
t.fit_on_texts(x_text)
encoded_docs=t.texts_to_sequences(x_text)
doc_length = max([len(x) for x in encoded_docs])

```

```

x = pad_sequences(encoded_docs, maxlen=doc_length, padding='post')

# Randomly shuffle data
np.random.seed(10)
shuffle_indices = np.random.permutation(np.arange(len(y)))
x_shuffled = x[shuffle_indices]
y_shuffled = y[shuffle_indices]

# Split train/test set
x_train, x_dev = x_shuffled[:-1000], x_shuffled[-1000:]
y_train, y_dev = y_shuffled[:-1000], y_shuffled[-1000:]

print('Build model...')
model = Sequential()

# we start off with an efficient embedding layer which maps
# our vocab indices into embedding_dims dimensions
model.add(Embedding(vocab_size,
                    embedding_dims,
                    input_length=doc_length))
model.add(Dropout(0.2))

# we add a Convolution1D, which will learn filters
# word group filters of size filter_length:
model.add(Conv1D(filters,
                 kernel_size,
                 padding='valid',
                 activation='relu',
                 strides=1))
# we use max pooling:
model.add(GlobalMaxPooling1D())

# We add a vanilla hidden layer:
model.add(Dense(hidden_dims))
model.add(Dropout(0.2))
model.add(Activation('relu'))

# We project onto a single unit output layer, and squash it with a sigmoid:
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          validation_data=(x_dev, y_dev))

```

# 第八章：循环神经网络

Recurrent Neural Networks，翻译为循环神经网络，简称 RNN。Recursive Neural Networks，翻译为递归神经网络，也简称 RNN。注意两者的区别。

RNN 是一类神经网络，它的 cell 之间的连接形成一个有向环。RNN 创建了一个网络内部状态，允许展示动态时态行为。不像前馈神经网络，RNN 可以使用内部记忆来处理任意输入序列。如此，RNN 可以应用在建立序列模型的任务。

## 第一节：RNN 结构

人类思考时并不仅仅是从某个时刻的信息开始思考。当你读文章时，你可以基于前面的词来理解当前的词。当前的思考不是在思考后就放弃，而是作为下个阶段思考的基础。人类的思考具有持久性和连续性。

传统的神经网络不能模拟上述的人类思考过程，这是它的一个主要缺点。RNN 强调这一问题。RNN 是用循环串起来的一组网络，可以持久保存信息。

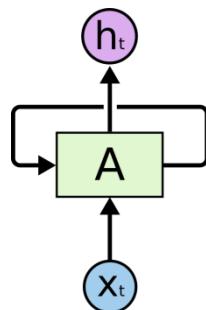


图 8.1 有循环的 RNN

图 8.1 是一个简单结构的 RNN，它的一个 chunk (有翻译做“块”，在 tensorflow 的术语里称为 cell) A 有个输入  $x_t$  (向量)，输出一个值  $h_t$  (向量)。一个循环 (loop) 使得信息被传递从网络的一步到下一步。这里的块 A，内部可以有简单或复杂的结构。简单，如它可以是一个前馈神经网络。复杂的结构可以参看第三节的 LSTM。

这些循环 (loop) 使得 recurrent neural networks 看起来有些神秘。然而，RNN 结构上并不是和传统神经网络有完全的差异。RNN 可以看做是同一个网络的多次复制，每

一次传递一个信息到循环中的下一个网络。我们展开这个 RNN。红框指示的是一个 time step。

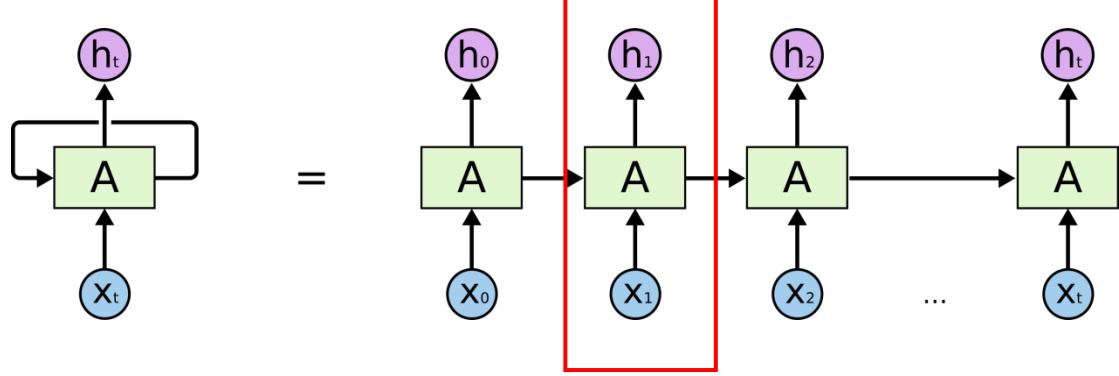


图 8.2 一个展开的 RNN

这种看起来像是链式的状态反映出 RNN 是和“序列”高度相关的。RNN 是神经网络处理序列数据理想的结构。在过去几年里 RNN 在很多问题上取得成功：语音识别、语音模型、机器翻译和图像处理等。可参看一篇文章 “The Unreasonable Effectiveness of Recurrent Neural Networks” <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

RNN 有很大的灵活性。有很多形式的 RNN。如图 8.3 所示。

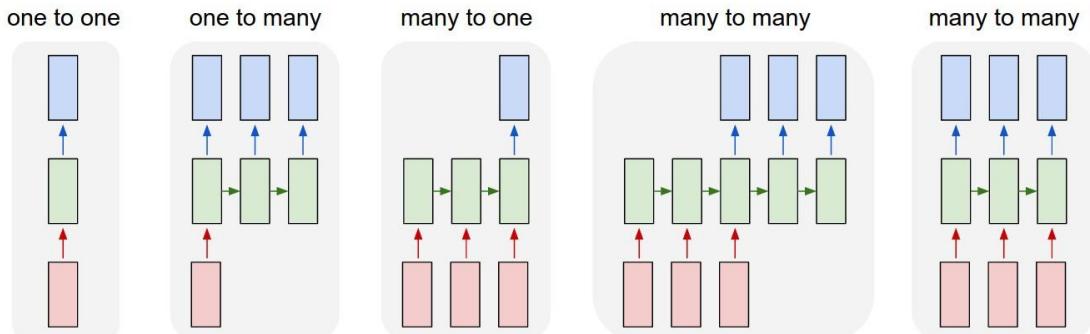


图 8.3 RNN 的各种结构

图中方块表示一个向量，箭头表示函数（例如，矩阵相乘）。输入向量用红色表示，绿色保存 RNN 的状态。One to one 模式称为 vanilla neural network（不算作是 RNN）。它接受固定大小的输入（例如，图像文件），给出固定大小的输出（例如，类别）。而 RNN 可以给定一个向量的序列，而输出可以是一个序列向量或就一个向量。One to many 模式，固定大小的输入，计算一个序列的输出，例如在 image caption 任务中给出一个图片，输出对图像的内容注释的句子。Many to one 是序列输入，计算一个固定大小的输出。例如，输入是一个句子，输出是句子的情感极性。Many to many 可以是序列输入，序列输出。例如在机器翻译中，输入序列是英文句子，输出是中文句子。第二种 many to many 是同步的输入序列到输出序列。例如，对 video 做分类，希望在视频的每一帧上贴标签。注意在上面的每种 RNN 中，都没有

预先规定序列的长度，因为 recurrent transformation (一个绿色方块) 是固定的，能按照我们的要求应用多次。

### 8.1.1 字符级语言模型：一个 RNN 模型实例

下面我们看一个基本 RNN 的例子，如图 8.4 所示。它是一个字符级的语言模型。在 8.2.3 我们采用 keras 实施了该语言模型。

<https://gist.github.com/karpathy/d4dee566867f8291f086>，给出了采用纯 Python 实现的模型。

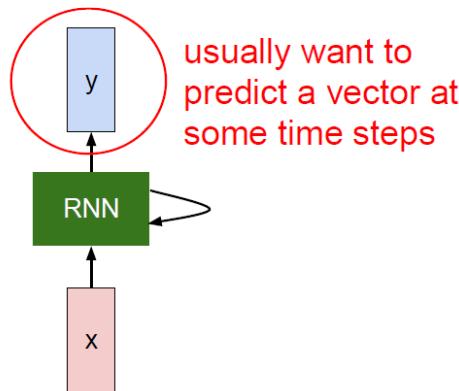


图 8.4 一个循环未展开的基本 RNN 结构

展开后 RNN 的每个时刻的结构，术语称为 time step，翻译做“时间步”。为了描述 RNN 的结构，我们举个最简化的例子：

假设当前字符表只有四个字符 "h,e,l,o"。图 8.5 是一个图 8.4 展开后的 RNN 示例。输入和输出层的维度为 4；隐层有三个神经元。该图显示当 RNN 被“喂”字符 “hell” 作为输入，前向传递被激活。输出层包含 RNN 分配下一个字符（从字符表中选）的确信程度。绿色数值是高值，红色数值是低值。每个 time step 的隐层之间有个状态的传递。

使用该 RNN 构建一个字符集的语言模型。训练集是文本，要求给定一个字符序列 RNN 可以建模下一个字符的概率分布。这个模型可以产生文本，它一次产生一个字符。我们假设有个字母表，仅有 4 个字母 “h”、 “e”、 “l”、 “o”。训练集是一个单词 “hello”。这个序列可以产生四条训练数据：(1) 给定 context “h”，看见 “e”的概率；(2) 在 context “he”， “l” 被看见的概率；(3) 在 context “hel” 看见 “l”的概率；(4) 在 context “hell” 看见 “o”的概率。

具体实施中，每个字母采用 one-hot 编码。给定一个输入字符序列，RNN 的每个时间步“喂”一个字母 (one-hot 编码的一个向量)。每个时间步输出一个向量，将该向量转换成对应的字符。由此得到一个输出序列 (维度为 4 的向量，每个维度一个字

符）。可以将输出解释为 RNN 当前分配下一个到来的字符的确信程度。如输出 $<1.0, 2.2, -3.0, 4.1>$ 对应输出字符是 helo 的确信程度。因为 2.2 最高，因此输出的是字符是 “e”。

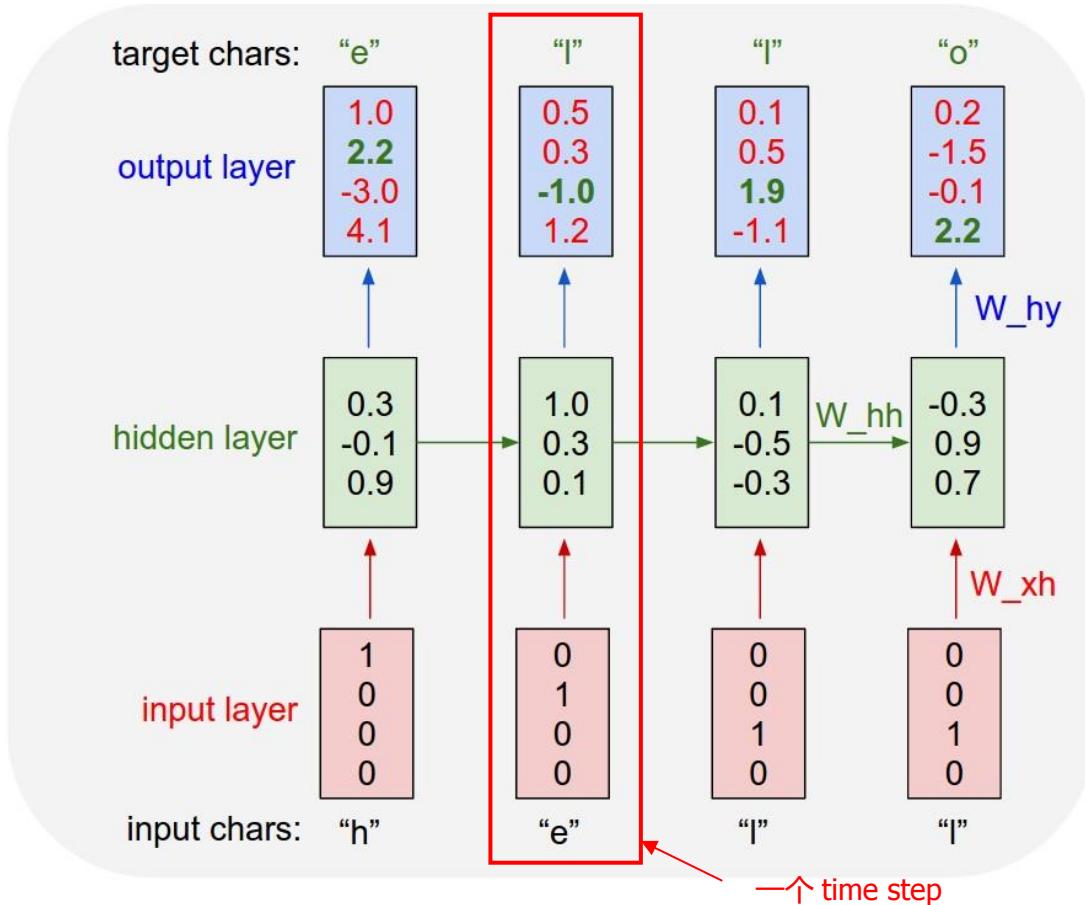


图 8.5 展开的 RNN 实例

以模型训练阶段为例。在第一步，当 RNN 看见了字符 “h”，在确定下一个字符时，它分配 1.0 的确信度到 “h”，2.2 的确信度到字符 “e”，-3.0 的确信度到字符 “l”，4.1 的确信度到字符 “o”。因为训练数据中，正确的下一个字符是 “e”，因此，训练算法将增加字符 “e”（绿色）的确信度，降低其他字符（红色）的确信度。

这个 RNN 的计算过程如下：

(1) 隐层的输出：(注意，与图 8.1, 8.2 不同，这里用符号  $h$  表示隐层的输出， $y$  表示 RNN 的输出)  $h_t$  是时间步为  $t$  时的隐层输出。它是根据上一个状态 (时间步  $t-1$  的隐层输出) 和当前的输入  $x_t$  来计算的。 $h_t = f_W(h_{t-1}, x_t)$

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

new state      |      old state      input vector at  
 some function      some time step  
 with parameters W

注意在每个“时间步”使用的是同样的函数和参数。当隐层的激活函数是 tanh

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

因此计算时有个循环过程，循环 time steps 次数，每次用上一个时间步的状态进行计算。

(2) 输出层的输出：

$$y_t = W_{hy}h_t$$

Tips:

- (1) 基本 RNN 中输入的序列长度是可变的。之所以可变，关键就是每个时间步使用同样的函数和参数
- (2) 理论上 RNN 可以处理任意长度的序列，实践中还是设定了序列的长度。即规定了时间步的步数。实际上在训练的过程中，将模型上的一次训练的最后一个时间步的状态的输出，作为下一次训练时模型的初始状态，则等同于扩展了输入的序列。

设计 RNN 时几个重要的参数需要确定。(1) 输入序列的长度，即 RNN 时间步的步数。(2) 每个时间步的输入是一个向量(输入层)，向量的长度。(3) 隐层神经元的个数。(4) 输入层到隐层的权重  $W_{xh}$  是一个 tensor，它的 shape=(输入向量的长度，隐层神经元的个数)(这个不需要用户自己确定)。(5) 从状态  $h_{t-1}$  到传统  $h_t$  传递时的权重 tensor  $W_{hh}$  的 shape=(隐层神经元的个数, 隐层神经元的个数)(这个不需要用户自己确定)。(6) 输出层的神经元数目。(7) 从隐层到输出层的权重 tensor 的 shape=(隐层神经元的个数, 输出层的大小)(这个不需要用户自己确定)。

图 8.5 中的一个时间步，我们可以按照一个前馈神经网络来理解。这里的隐层(即 cell)可以是多层。这就是我为什么说一个基本的 RNN 的 cell 是一个前馈神经网络。图 8.6 中隐层只是一层。如果是多层， $t-1$  步隐层的每一层都会参与到  $t$  步隐层的对应每一层的计算。两个时间步之间的方块表示一个全连接层。它的神经元数和隐层的神经元数一致。对于图 8.5 的权重  $W_{hh}$ 。

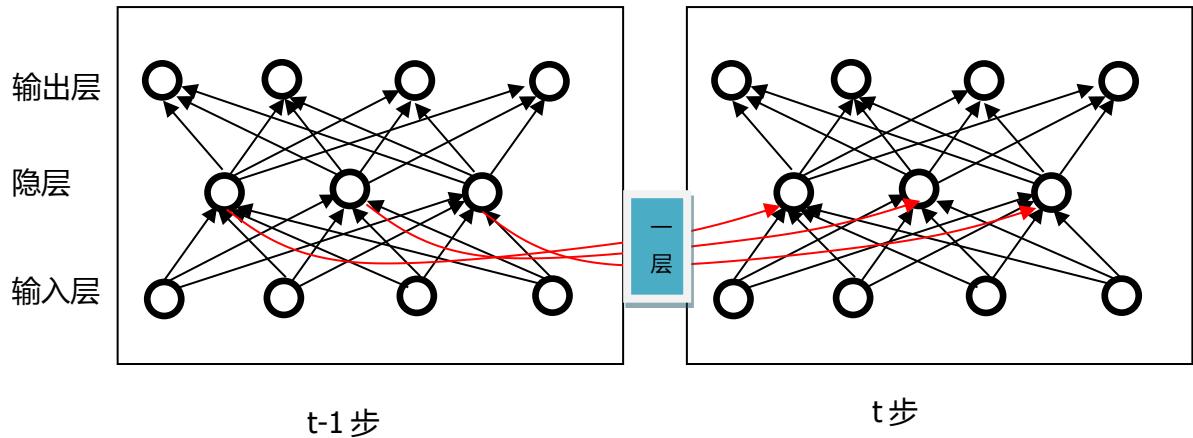


图 8.6 RNN 一个时间步

按照图 8.5 两个时间步之间的状态传递也经过了神经网络的一个层。不过在 tensorflow 是实施中，没有要求用户自己建立这一层，不知道是不是 RNN 函数内部建立了。另外每个时间步共享输入到隐层和隐层到输出层两个层的权重（共享权重）。

## 第二节：使用 Keras 构建 RNN

### 8.2.1 介绍

构建 RNN 模型时，先理解几个术语：

(1) - *num\_layers* - the number of RNN Layers

层数是指隐层中包含多少个子层，一个 cell 对应一层。例如，图 8.6 有一层，图 8.7 有两层。隐层有多个子层称为多层 RNN。

(2) - *num\_steps* - the number of unrolled steps of RNN

时间步 time step 的步数，图 8.5 时间步为 4

(3) - *hidden\_size* - the number of units

术语 hidden size 是指一个时间步的一个隐层的子层（一个 cell）包含多少个 unit。在基本的 RNN 结构，一个 unit 就是一个神经元。

RNN 实际上是对输入的序列中的一个时间步，如  $x_t$ ，这个向量的每一位进行计算。每一位的计算都是由一个 unit 完成。每一个时间步就是一个神经网络，参考图 8.7 的描述。

注：我理解 cell 是指对一个向量进行计算的隐层单元。Cell 是由多个 unit 组成。对向量的每一位进行计算的。每个 unit 的结构在基本 RNN 中是一个神经元，在 LSTM 等复杂结构中，是如图 8.19 所示的结构。只不过该图应该是对标量进行计算，计算结果也是标量。

我们用图 8.7 来描述一个时间步的计算。输入  $x^t$  是一个向量。有一个全连接层和输入层相连。全连接层的神经元个数就是  $hidden\_size$ 。全连接层的每个神经元的输出都和一个 unit 连接。这里的每个隐层是一个 cell，多个 cell 构成多层的隐层。时间步  $t$  每个 unit 的计算要使用时间步  $t-1$  对应的 unit 的状态  $s_{L,n}^{t-1}$ ，图中用的是一个向上的箭头表示。

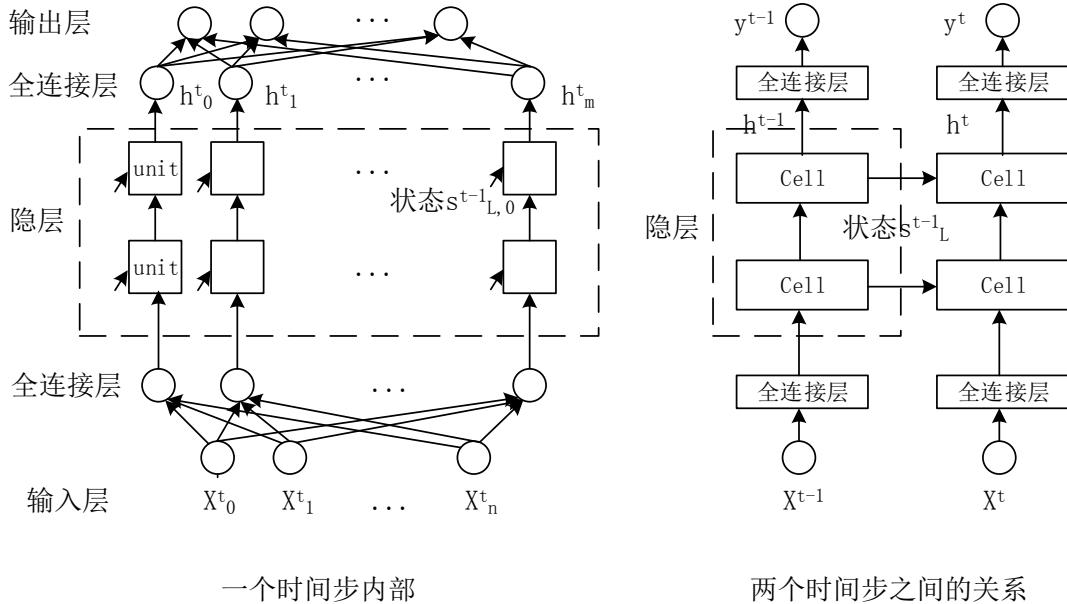


图 8.7 时间步

Tensorflow 提供的 cell 和 RNN 函数，只是实现了多个时间步的隐层部分。两个全连接层需要用户自己建立。每个时间步共享全连接层。

注：两个全连接层不是必须的。例如，在 `ptb_word_lm.py` 建立的模型中，它让词向量的长度等于  $hidden\_size$ 。也就是说输入层全连接层的作用是变换向量的长度到  $hidden\_size$ 。如果输入向量的长度等于  $hidden\_size$ ，就没必要建立这个全连接层。

## 8.2.2 构建 RNN 的基本函数

<https://www.tensorflow.org/guide/keras/rnn>

Keras 提供了三个基本的 RNN 层的类: tf.keras.layers.SimpleRNN, tf.keras.layers.LSTM, tf.keras.layers.GRU。用户也可以使用 tf.keras.layers.RNN 和 Cell 类构建自己的 RNN。我们将在 9.3 节介绍定制方法。

本节我先介绍 tf.keras.layers.SimpleRNN。它是一个全连接的 RNN，前一个连接的时间步喂到下一个时间步。对于两种特殊的 RNN：LSTM 和 GRU 我们在第三和第四节介绍。

```
tf.keras.layers.SimpleRNN(  
    units, activation='tanh', use_bias=True, kernel_initializer='glorot_uniform',  
    recurrent_initializer='orthogonal', bias_initializer='zeros',  
    kernel_regularizer=None, recurrent_regularizer=None, bias_regularizer=None,  
    activity_regularizer=None, kernel_constraint=None, recurrent_constraint=None,  
    bias_constraint=None, dropout=0.0, recurrent_dropout=0.0,  
    return_sequences=False, return_state=False, go_backwards=False,  
    stateful=False, unroll=False, **kwargs  
)
```

主要参数：

units : RNN 一个时间步输出的向量的维度，也即 RNN 一个 cell 的隐层的神经元的数目。

RNN 类不需要设定时间步参数，喂入模型的时间步是多少，模型的时间步就是多少。

对于模型细节的理解：

## 1. RNN 层的输入

输入 tensor 的 shape= (batch\_size, timesteps, input\_dim)。当然，在 keras 中，batch\_size 不需要给出。

## 2. RNN 层的输出

默认的 RNN 的输出是每个 sample 一个向量。这里的 sample 是喂入模型的一个序列，例如一个句子。输出的向量是 RNN 最后一个时间步的输出。这个最后的时间步包含了整个输入序列的信息。输出 tensor 的 shape=(batch\_size, units)。units 是 RNN 一个时间步的隐层的神经元数。

RNN 也可以返回每个时间步的输出，此时设参数 return\_sequences=True，输出 tensor 的 shape=(batch\_size, timesteps, units)

```
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Embedding, GRU, SimpleRNN
```

```

model = Sequential()
model.add(Embedding(input_dim=1000, output_dim=64))
model.add(GRU(256, return_sequences=True))
model.add(SimpleRNN(128))
model.add(Dense(10))
model.summary()

```

这个例子中，The output of GRU will be a 3D tensor of shape (batch\_size, timesteps, 256)。The output of SimpleRNN will be a 2D tensor of shape (batch\_size, 128)

### 3. 最后一个时间步的状态

另外，一个 RNN 层返回的输出 tensor 也可以返回最后一个时间步的内部状态。这样可以用于稍后恢复 RNN 的执行，或者初始化另外的一个 RNN。这样的操作通常用在我们将第十章讨论的 Encoder-Decoder 结构中，编码器的最后一个状态用在解码器的初始状态。9.3 节介绍 RNN 类时，提到

If `return_state`: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each with shape `[batch_size, state_size]`, where `state_size` could be a high dimension tensor shape.

即，设置参数 `return_state=True`。RNN 层返回的结果包括输出 tensor 和状态信息 tensor。

要设置一个层的初始状态需要设置参数 `initial_state`，它没显示在上面的 SimpleRNN 的参数中。注意：状态的 shape 需要与 RNN 层的 `units` 参数一致。

下面的模型是 10.2 节的一个基于 encoder-decoder 的翻译模型。其中应用的是 8.3 节的 LSTM 模型。我们只是想演示，Encoder 输出的状态，被传递到了 Decoder，作为为了初始状态。模型结构见图 10.3。

```

encoder_vocab = 1000
decoder_vocab = 2000

encoder_input = layers.Input(shape=(None, ))
encoder_embedded = layers.Embedding(input_dim=encoder_vocab,
output_dim=64)(encoder_input)

# Return states in addition to output
output, state_h, state_c = layers.LSTM(64, return_state=True, name='encoder')
(encoder_embedded)
encoder_state = [state_h, state_c]

decoder_input = layers.Input(shape=(None, ))
decoder_embedded = layers.Embedding(input_dim=decoder_vocab,
output_dim=64)(decoder_input)

# Pass the 2 states to a new LSTM layer, as initial state

```

```

decoder_output = layers.LSTM(
    64, name='decoder')(decoder_embedded, initial_state=encoder_state)
output = layers.Dense(10)(decoder_output)

model = tf.keras.Model([encoder_input, decoder_input], output)
model.summary()

```

#### 4. 跨 batch 的状态传递 ( cross-batch statefulness )

如果我们处理很长的序列。而我们设定了当前的 RNN 处理的序列长度，例如是 25。那么长序列会被切成 25 个时间步长的多个子序列。默认的 stateful 的状态是 False。此时，喂入模型的每个子序列的初始 state 都是随机初始化的，而如果设置 stateful=True，则当前喂入的子序列的最后一个时间步的状态将作为下一个子序列喂入模型时的初始状态。

### 8.2.3 实例 1：用 Keras 实现字符级语言模型

这一节用 Keras 实现图 8.5 的字符级的语言模型。实现程序参见 min-char-rnn-keras.py。对于字符采用 one-hot 编码。我们实施时，在输入层和 RNN 层之间加入了一个隐层，模型如图 8.8 所示。输入数据经过全连接层转换成适合隐层处理的形式，隐层的输出经过全连接层转换成输出。每个时间步的全连接层是共享的。

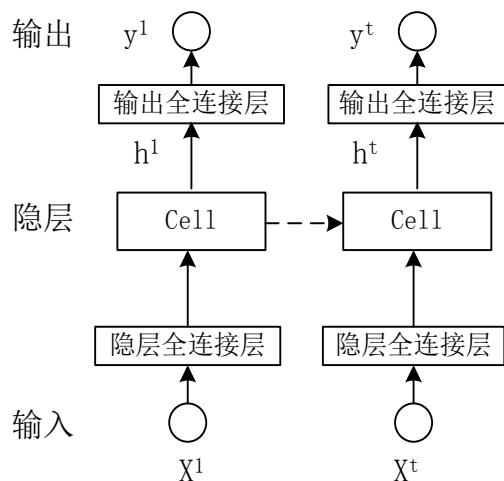


图 8.8 : min-char-RNN 模型

设计这个 RNN 模型时，我们需要考虑几个参数

```

hidden_size = 500 # size of hidden layer of neurons
seq_length = 25   # number of steps to unroll the RNN for
learning_rate = 1e-1
epoch_size = 20
batch_size = 100

```

hidden\_size 是隐层的 unit 数目；Seq\_length 是时间步数；Learning\_rate 是训练模型时的学习率；epoch\_size 是训练模型时的迭代次数；batch\_size 是我们采用 minibatch GSD 训练算法时，batch 的大小。这里我们对每个字符采用的 one-hot 编码，因此 onehot 字符向量的大小是字符集的大小。

下面的代码从一个文本文件建立数据集。文本文件被转换成一个长的字符串。建立字符集 chars；建立两个词典 char\_to\_ix 是字符到编号的映射，ix\_to\_char 是编号到字符的映射。将文本的长字符串转换成字符编码。然后用 to\_categorical 进行 one\_hot 编码。one\_input 和 one\_target 是错开一个字符的 one\_hot 编码的字符序列。x\_train 和 y\_train 是训练集数据和目标值。

```
data = open('t1.txt', 'r').read()
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print ('data has %d characters, %d unique.' % (data_size, vocab_size))
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }
idx_data=[char_to_ix[item]for item in data]
encoded_data=to_categorical(idx_data)
one_input = encoded_data[0 : data_size-2]
one_target = idx_data[1 : data_size-1]

x_train = []
y_train = []
p = 0
while p + seq_length < len(one_input)-seq_length:
    x_train.append(one_input [p:p+seq_length])
    y_train.append(one_target[p:p+seq_length])
    p = p + 1

x_train = np.array(x_train)
y_train = np.array(y_train)
```

下面的代码建立模型，并训练模型。需要强调的是，RNN 模型中的参数 return\_sequences=True，这表示 RNN 所有的时间步的结果会被输出。我们又添加的一个全连接层，即输出层 output，是作用在每个时间步的输出上的，即他们每个时间步的输出共享一个全连接层。

```
model= Sequential()
model.add(Input(shape=(seq_length,vocab_size), name='input'))
model.add(Dense(hidden_size, name='hidden'))
model.add(SimpleRNN(hidden_size, activation='relu', return_sequences=True,
name='rnn'))
model.add(Dense(vocab_size, activation='softmax', name='output'))
model.summary()
```

```

opt=Adam(learning_rate=learning_rate)
model.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, batch_size=batch_size, epochs=epoch_size)

```

训练模型后，我们考察一下模型的输入和输出。把训练集带入模型进行预测，我们考察输入的字符串和输出的字符串。理想的结果是两个字符串错开一个字符位。

```

res=model.predict(x_train)
i=2
x_=".join([ix_to_char[item] for item in np.argmax(x_train[i],axis=1)])
print(x_)
y_=".join([ix_to_char[item] for item in np.argmax(res[i],axis=1)])
print(y_)

```

输出的结果如下：

```

dr. goldberg offers every
i Goldberg iffers everyt

```

### 第三节： LSTM

RNN 的一个吸引人的地方在于它可以连接先前的信息到当前的任务，例如使用先前的视频帧帮助当前帧的理解。有时我们仅仅需要最近的信息，而不是太早以前的信息完成当前的任务。例如，一个语言模型试图根据前面的词预测下一个词。如果我们预测一个句子 “the clouds are in the sky” 中的最后一个词。根据句子中前面的词集合（前面的词是当前的词 context）“the clouds are in the”很明显这个词应该是 sky。这个例子中，我们需要的 context 相关信息可以不是很多。这个问题称为 Short Term Dependencies。

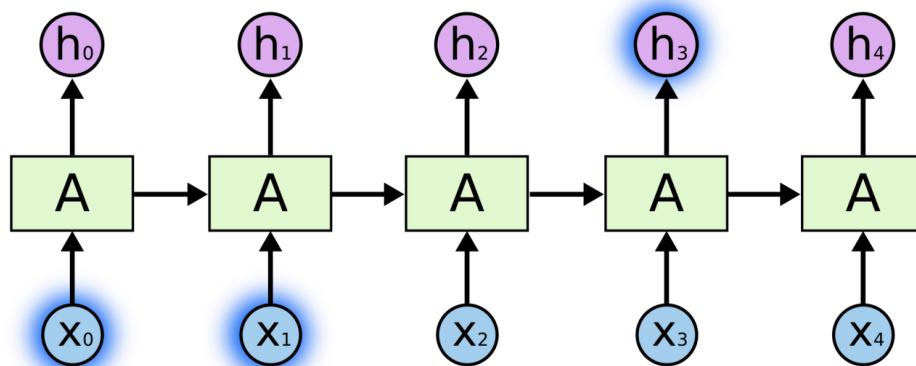


图 8.11. Short Term dependency

但有时我们需要更多的 context。再看一个例子，“I grew up in France... I speak fluent French.”（省略号表示还有很多句子）当预测最后一个词时，前面的信息 I speak fluent 给出暗示这个最后的词应该是一个语言名称。但如果想知道具体是哪一种语言，我们需要更多的 context。如此再往前寻找 context。“I grew up in France” 暗示是 French。可以看出从相关信息到待预测的词之间的 gap 很大。

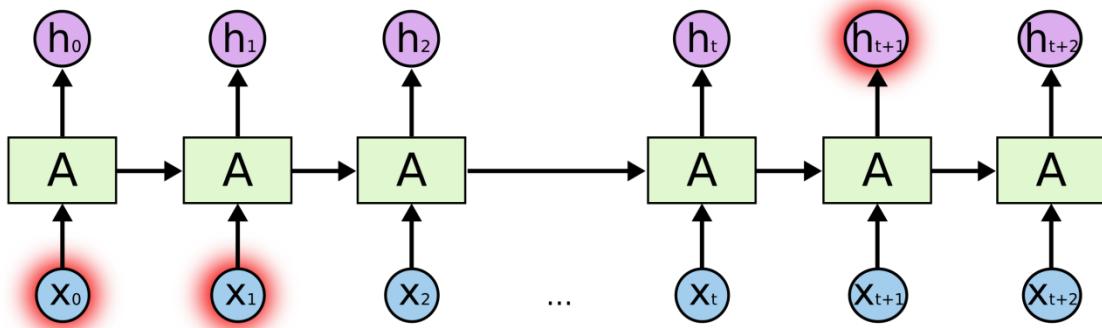


图 8.12. Long Term Dependency

当这个 gap 很大时前面讲述的基本 RNN 没有能力连接到 gap 前面的 context 去进行学习。这个问题称为 Long Term Dependencies.

### 8.3.1 LSTM 的结构

理论上 RNN 能够处理 long-term dependencies。但实践中有很多问题。[Hochreiter \(1991\) \[German\]](#) and [Bengio, et al. \(1994\)](#),指出了 RNN 在这个问题上的根本缺陷。但 LSTM 可以很好的解决这个问题。LSTM ( Long Short Term Memory networks ) 是一种特殊结构的 RNN，它可以学习 Long Term Dependencies。它由 Hochreiter & Schmidhuber 提出。在其后的研究中许多人的研究将 LSTM 改进使得它在很多任务上都非常成功。现在 LSTM 的应用非常广泛。所有的 RNN 都有一个链式结构，重复了神经网络的一个块 ( chunk 或 cell )。标准的 RNN 中重复的“块”有一个很简单的结构，例如一个单独的 tanh 层 ( 图中黄框表示一个层 )。 ( 8.1 节  $h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$  )。这里  $X_t$  是一个输入向量；tanh 层是神经网络的一层，即包含了多个 unit； $h_t$  是一个向量。

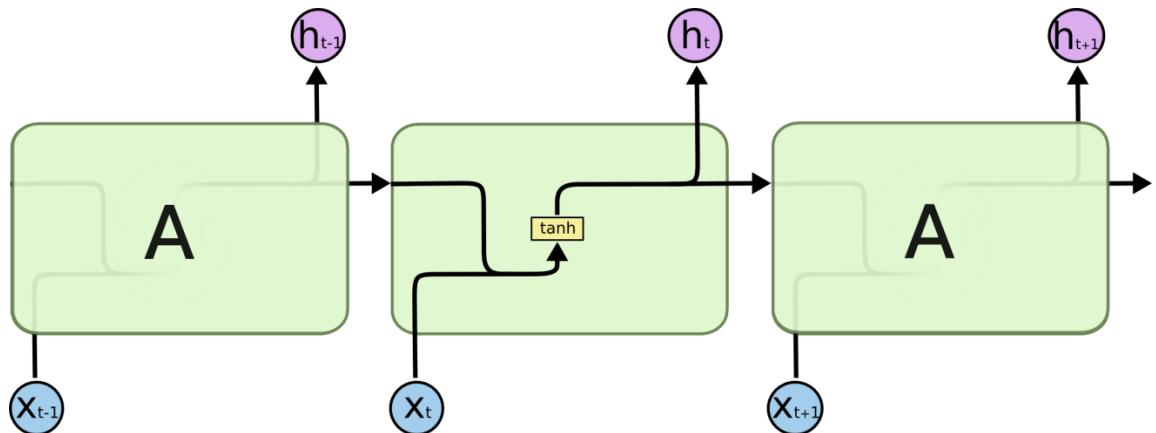


图 8.13. 在一个标准 RNN 中重复了一个单独的层

LSTM 也有这种链式结构，但是重复的块（chunk）有复杂的结构，例如有四个层，以一种特殊的形式交互。如图 8.14 所示。

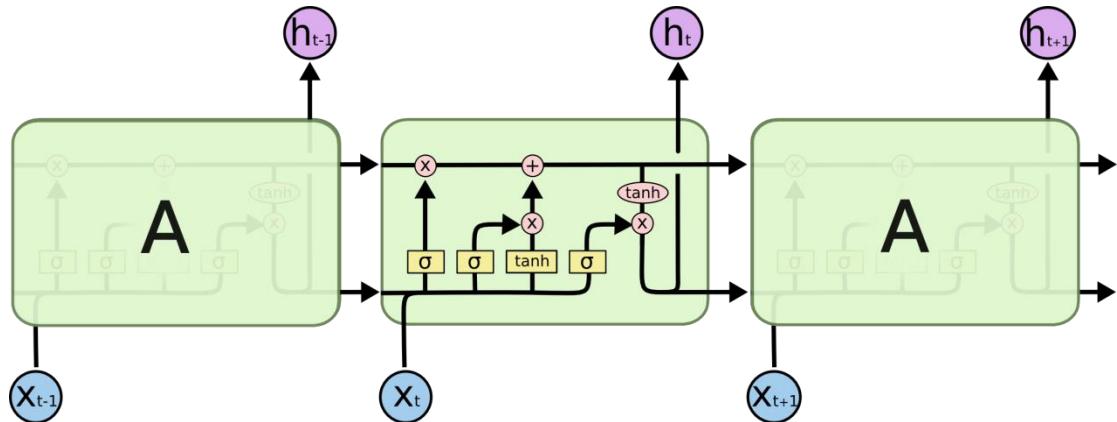


图 8.14 LSTM 中的块包含四个交互层

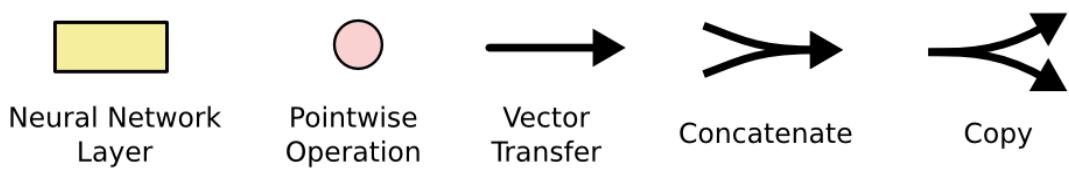


图 8.15 一些注释符号

图 8.15 给出描述 LSTM 需要的一些符号。图中每条线表示从一个节点的输出传递一个 Tensor 到一个节点的输入；粉色的圈表示逐点运算，例如向量相加；黄色的方框是待学习的神经网络的层；线的合并表示拼接操作；线段的分叉表示内容被复制，然后送到不同的节点。

### 8.3.2 LSTM 的核心思想

与基本 RNN 相比，LSTM 的关键是增加了块（ chunk 或 cell ）状态，即贯穿图的那条水平线。块的状态（ cell state ）可以理解为是一种传送带。信息沿着它传递。

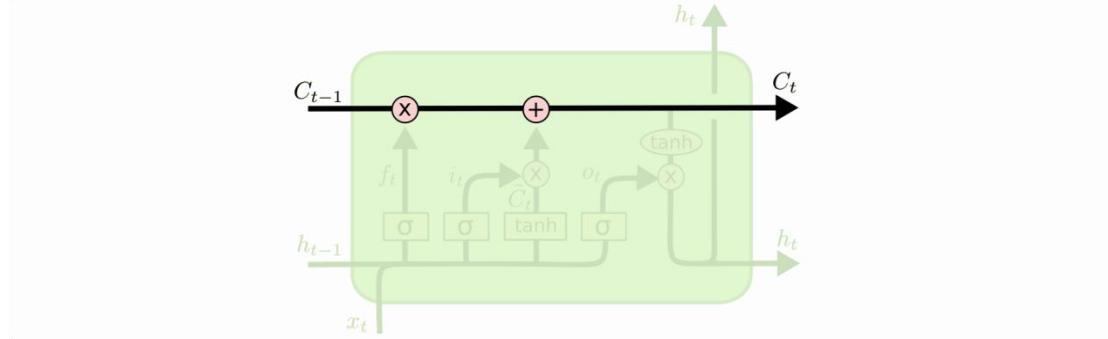


图 8.16 RNN 的信息传递

通过调整称为 gate 的结构，LSTM 有能力移除或添加信息给单元状态（ cell state ）。 Gate 是一种方式或通道，选择性的让信息通过。它由一个 sigmoid 层和一个逐点相乘的运算操作组成。如图 7.17 所示。

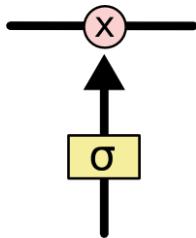


图 8.17 Gate 结构

Sigmoid 层输出 0~1 之间的数值，描述了每个构件（ component ）有多少部分允许通过，例如，0 表示不让通过，1 表示全部通过。一个 LSTM 有三种 gate ( forget gate, input gate, output gate ) 来包含和控制 cell state. ( 图 8.14 中一个块里面的三个  $\sigma$  )

### 8.3.3 LSTM 的工作过程

LSTM 的第一步是决定什么样的信息应该通过 cell state 传递。这个决策由第一个 sigmoid 层决定（图 8.18），称为 **forget gate**。Sigmoid 层的输入是  $h_{t-1}$  和  $x_t$ 。对于 cell state  $C_{t-1}$  的每个值，forget gate 输出一个 0~1 之间的一个值。1 表示完全通过，0 表示阻止。

我们回到语言模型的例子。该语言模型试图基于前面的词预测下一个词。该任务中，cell state 可以包括当前主语的性别这样的信息，如此可以使用正确的介词。当看见一个新的主语，我们应该忘记上一个主语对应的性别。（我理解上面这个例子的意思是一个句子包含多个主语）

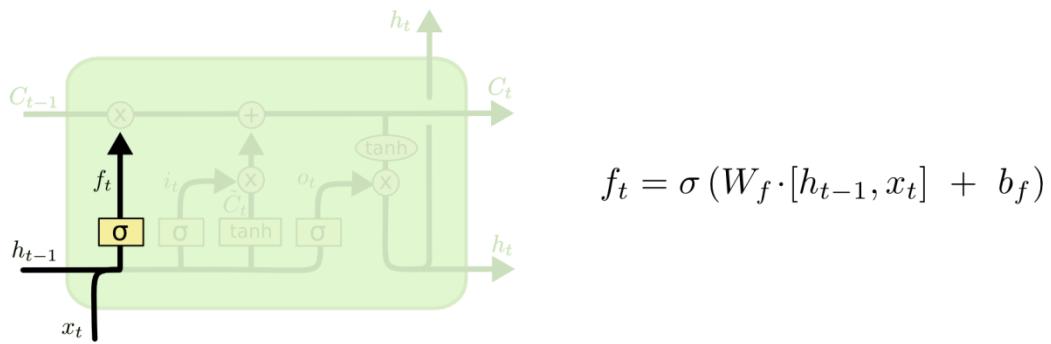


图 8.18 LSTM 块的第一层：一个 Sigmoid 层

$[h_{t-1}, x_t]$  表示的是将  $h_{t-1}$  和  $x_t$  拼接操作到一个矩阵。权重矩阵  $W_f$  实际上包含  $h_{t-1}$  的权重  $W_{fh}$  和  $x_t$  的权重  $W_{fx}$ 。上面的操作  $W_f \cdot [h_{t-1}, x_t]$  可以分解成  $W_{fh} \cdot h_{t-1} + W_{fx} \cdot x_t$ 。

下一步是要确定我们将要存储什么新信息在 cell state 中，见图 8.19。它包含两部分。首先一个 sigmoid 层称作 “**input gate**” 决定我们应该更新哪个值。下一步，一个 tanh 层创建一个新的候选值的向量  $\tilde{C}_t$ 。它能被加到这个状态中。紧接着，联合这两个状态来创建一个新的状态。在语言模型的例子中，我们想加新主语的性别到 cell state，来替换旧的状态。

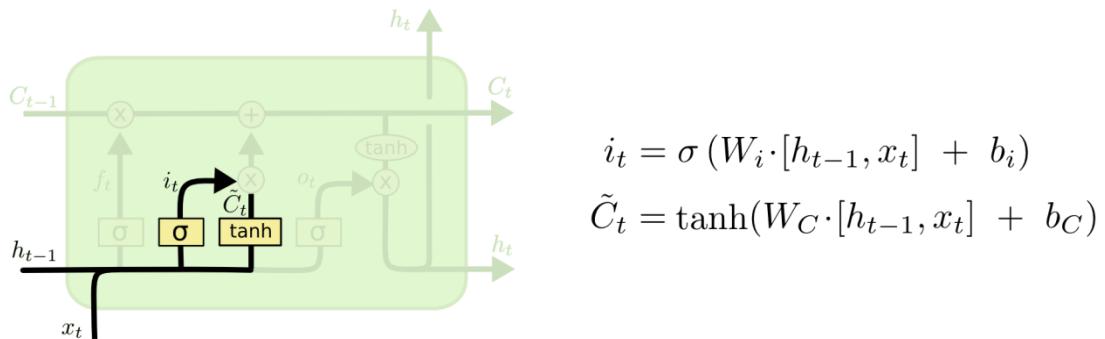


图 8.19 LSTM 块的第二，三层

下面的计算将旧的状态  $C_{t-1}$  更新到新的 cell state  $C_t$ 。旧状态乘上  $f_t$  于是忘记早先决定忘记的。然后加上  $i_t * \tilde{C}_t$ 。这是新的候选值。在语言模型的例子中，相当于我们实际上放弃了关于旧的主语的性别信息，加上了在上一步骤（图 7.19）中确定的新信息。

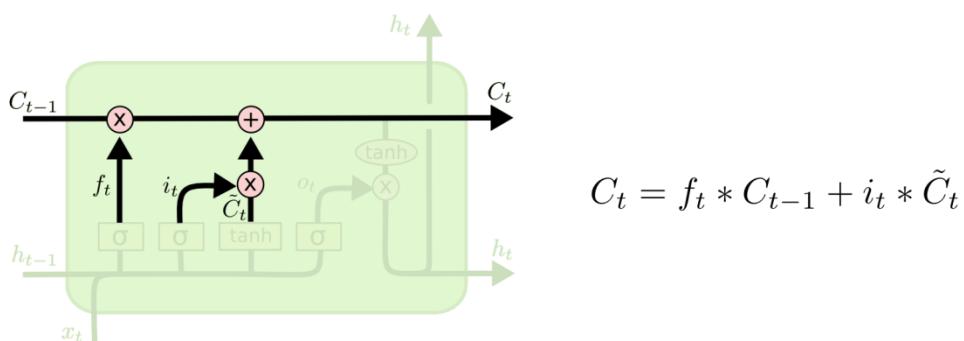


图 8.20 LSTM 块的第二，三层的输出

最后，需要确定输出值。输出应该基于 cell state。我们运行一个 sigmoid 层。它担负 gate 功能，即 output gate，它决定 cell state 的什么部分应该输出。让 cell state 通过 tanh（把值规范化到-1 和 1 之间），再乘上 sigmoid gate 的输出。如此我们仅仅输出我们确定想输出的部分。

再以语言模型为例。语言模型看见了一个主语，它可以想输出与动词相关的信息，因为动词是紧接着要到来的词。例如，它可以输出是否主语是单数还是复数。如此我们知道下一步形成一个动词时应该配合这个信息。

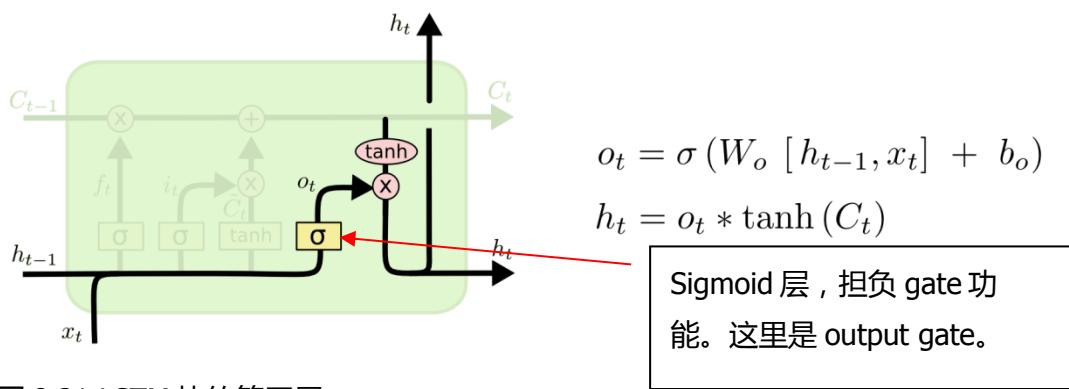


图 8.21 LSTM 块的第四层

### 8.3.4 Keras 中的 LSTM 层

```
tf.keras.layers.LSTM(
    units, activation='tanh', recurrent_activation='sigmoid',
    use_bias=True, kernel_initializer='glorot_uniform',
    recurrent_initializer='orthogonal',
    bias_initializer='zeros', unit_forget_bias=True,
    kernel_regularizer=None,
    recurrent_regularizer=None, bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None, recurrent_constraint=None,
    bias_constraint=None,
    dropout=0.0, recurrent_dropout=0.0, implementation=2,
    return_sequences=False,
    return_state=False, go_backwards=False, stateful=False,
    time_major=False,
    unroll=False, **kwargs
)
```

同 RNN 层的参数一样。

**units**：一个隐层神经元的个数

输入的 tensor 的 shape= ( batch, timesteps, feature )

如果 return\_sequences=True, 输出的 tensor 的 shape= ( batch, timesteps, hidden\_size )。否则，shape= ( batch, hidden\_size ) 即最后一个时间步的输出。

如果 return\_state=True。调用 lstm 对象返回的结果，包括输出，h 状态和 c 状态。  
例如，

```
encoder = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
```

## 第四节：LSTM 的变体

### 8.4.1 LSTM 变体

迄今我们描述的 LSTM 是一个标准的 LSTM。但不是所有 LSTM 都与上面的结构相同。  
事实上，每篇涉及 LSTM 的论文都有自己的结构，和标准结构会有些差异。

一个受欢迎的 LSTM 变体 Gers & Schmidhuber 加入了 peephole connection。这意味着  
让 gate layer 看见 cell state。见图 7.22 Cell state  $C_{t-1}$  参与了 forget gate  $f_t$  的计算。

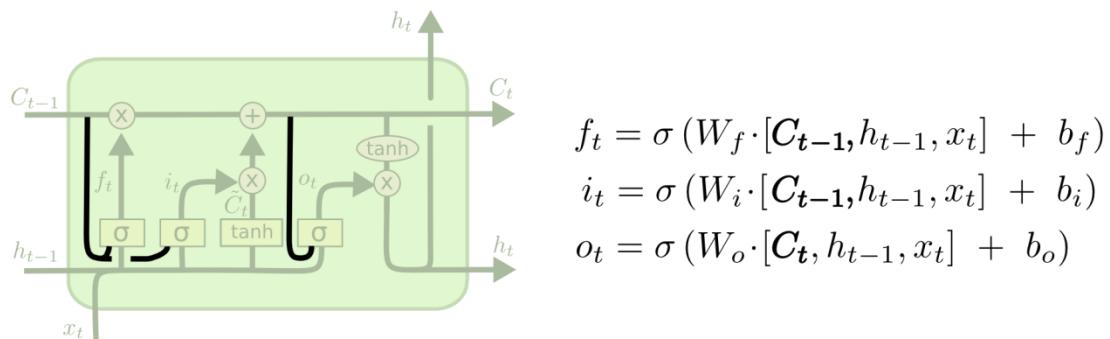


图 8.22 peephole connection LSTM

上图中所有的 gate 都添加了 peephole。有些论文并不是所有的 gate 都应用  
peephole。

另一个 LSTM 变体将 forget gate 和 input gate 结合使用。它不像基本 LSTM 中单独决定什么应该被忘记 (forget gate)，应该加什么新信息 (input gate)。该变体将两个决策一起决定。仅仅当要输入一些信息，那么相应位置的旧信息把它忘记，其他位置的信息不变。仅仅在旧的 cell state 中被忘记的部分输入新值。

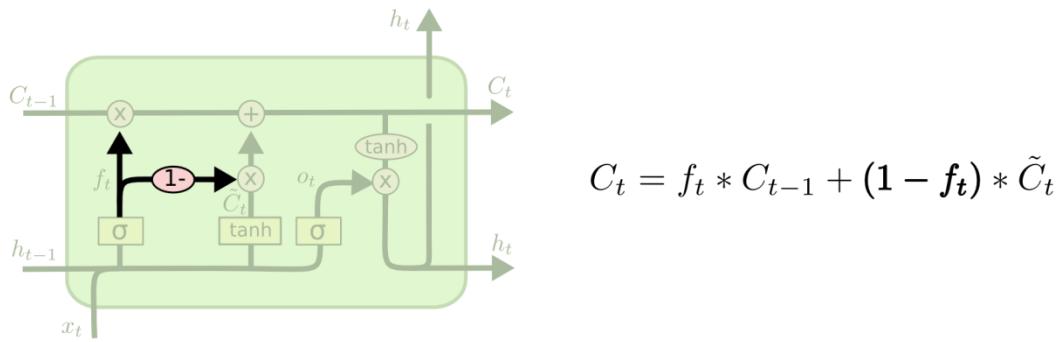


图 8.23 LSTM 的一个变体

一个更动态的变体是 GRU ( Gated Recurrent Unit ) [Cho, et al. \(2014\)](#)。它结合 input gate 和 forget gate 为一个新的 gate，称作 update gate。它也合并了 cell state 和隐层，并做了一些其他的改变。其模型比 LSTM 更简单，少了一个传递状态  $C_t$ ，也非常受欢迎。

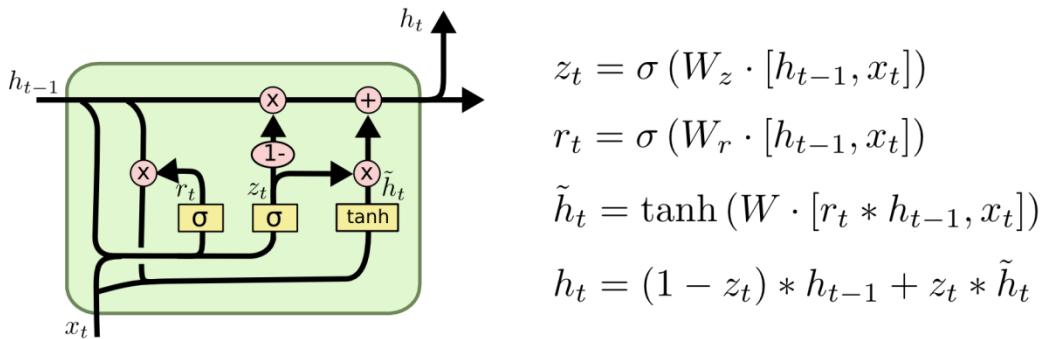


图 8.24 GRU

上面仅介绍了部分 LSTM 变体。有人对这些变体做了比较 [Greff, et al. \(2015\)](#)，发现它们其实都差不多。也有人测试了超过 1 万个 RNN 结构 [Jozefowicz, et al. \(2015\)](#)，发现有些变体在确定任务中比 LSTM 更好。

#### 8.4.2 keras 的 GRU 层

```
tf.keras.layers.GRU(
    units, activation='tanh', recurrent_activation='sigmoid',
    use_bias=True,
    kernel_initializer='glorot_uniform',
    recurrent_initializer='orthogonal',
    bias_initializer='zeros', kernel_regularizer=None,
    recurrent_regularizer=None,
    bias_regularizer=None, activity_regularizer=None,
    kernel_constraint=None,
    recurrent_constraint=None, bias_constraint=None, dropout=0.0,
    recurrent_dropout=0.0, implementation=2,
```

```
return_sequences=False,
    return_state=False, go_backwards=False, stateful=False,
unroll=False,
    time_major=False, reset_after=True, **kwargs
)
```

主要参数和 LSTM 的差不多。但，如果 `return_state=True`。调用 `lstm` 对象返回的结果，包括输出和 `h` 状态。

```
encoder = GRU(latent_dim, return_state=True)
encoder_outputs, state_h = encoder(encoder_inputs)
```

### 8.4.3 RNN 的发展方向

LSTM 在大部分任务上都工作的很好。LSTM 使得 RNN 向前发展了一大步。而下一个将使得 RNN 发展一大步的是 attention ( 翻译做注意力机制，将在第十一章介绍 ) 。

其思想是让 RNN 的每一个 time step 挑选信息，以看得到其他的信息集合。例如，在使用 RNN 创建一个描述图片内容的短文 ( caption ) 任务中，让输出的每个 word 可以看见挑选的一部分图片。关于 attention 可以参考论文 [Xu, et al. \(2015\)](#)。

RNN 发展的另外的方向还包括 Grid LSTMs by [Kalchbrenner, et al. \(2015\)](#)。

还有一些工作在 generative models 中使用 RNN，例如 [Gregor, et al. \(2015\)](#), [Chung, et al. \(2015\)](#), or [Bayer & Osendorfer \(2015\)](#) 也是一个 RNN 未来的方向。

## 第五节：用 Keras 构建词级 LSTM 语言模型

### 8.5.1 介绍

Wojciech Zaremba 的论文 RECURRENT NEURAL NETWORK REGULARIZATION 实施了一个 word 级的 LSTM 语言模型 ( 见 `ptb_word_lm.py` )。它在 Penn Tree Bank 数据集上进行 words 级别的预测，即输入是一个 words 的序列，输出是每个输入 word 的下一个邻近词的预测。Tensorflow 的 tutorial 讲解了该模型

<https://tensorflow.google.cn/tutorials/sequences/recurrent>。

我们现在用 keras 实施该模型。

### 8.5.2 数据集

该模型需要的 PTB 数据集 <http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz>。该数据集已经被预处理，包含 10000 个 word，包括句子结束标记和一个特殊的符号<unk>指代很少出现的词。

上面的代码中的大部分是关于数据集的准备，其核心是一个函数 load\_data 和一个类 KerasBatchGenerator。该函数装载数据集，而类是一个 python 的 generator。用于在训练模型时，逐 batch 的产生训练集。具体不详细讲解了。

注：深度学习处理文本时首先确定一个词汇表，且为一个特殊字符 UNK 分配编号。然后把原始文本中的词用编号替换。未出现在词汇表中的词用 UNK 的编号替换。

### 8.5.3 几个函数

#### 1. TimeDistributed(layer)

```
keras.layers.TimeDistributed(layer)
```

这是一个 wrapper，它将层 layer 应用在每个时间步上。（[8.2 节的字符语言模型中，没应用该函数可以一样工作啊？](#)）

它的输入的 tensor 的维度至少是 3D。例如下面的代码将 Dense 层分别应用在每个时间步上。

```
model = Sequential()
model.add(TimeDistributed(Dense(8), input_shape=(10, 16)))
```

该模型输入的 tensor 的 shape=(None, 10, 16)。输出的 tensor 的 shape=(None, 10, 8)

#### 2. Model 类的 fit\_generator 方法

```
fit_generator(generator, steps_per_epoch=None, epochs=1, verbose=1, callbacks=None,
validation_data=None, validation_steps=None, validation_freq=1, class_weight=None,
max_queue_size=10, workers=1, use_multiprocessing=False, shuffle=True,
initial_epoch=0)
```

它是 model 对象下的另外的一种训练模型的方法。

它训练模型时，使用一个 python generator 来一个 batch 一个 batch 的产生训练数据。我们在 8.2 节构建的模型，是把整个训练集划分好，然后 fit 函数自己读一个 batch 的训练数据。但是当训练集很大时，都读入内存会占用太大的内存空间。此时可以采用 python 的 generator。

参数：

Generator: 产生的数据的 generator

**steps\_per\_epoch:** 数据集可以被划分成多少个 batch , 可以按照公式  
 $\text{ceil}(\text{num\_samples} / \text{batch\_size})$  来计算

### 3. Model 类的 save 函数

该方法可以把模型保存到磁盘 , 其参数就是路径和文件名 , 例如

```
model.save(data_path + "final_model.hdf5")
```

### 4. load\_model 函数

```
from tensorflow.keras.models import load_model
```

装载用 load 保存的模型 , 例如

```
model = load_model(data_path + "\final_model.hdf5")
```

## 8.5.4 模型结构

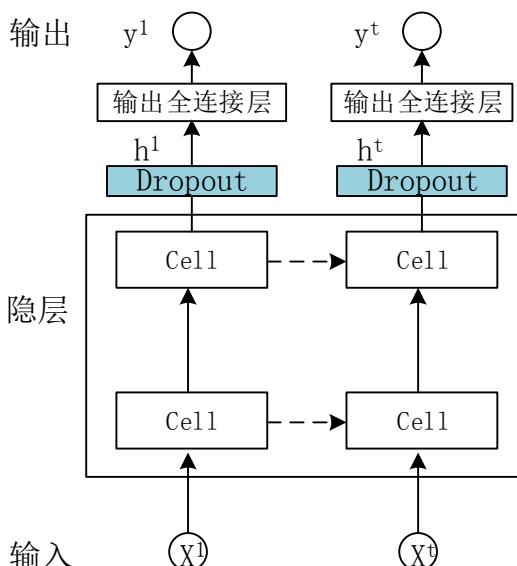


图 8.25 多层 LSTM

图中的  $X^t$  是一个词经过查找表映射到的一个词向量。 $y^t$  是一个长度为词汇表大小的向量。词向量的大小和 rnn 的隐层的 units 数一致。

```
model = Sequential()
model.add(Embedding(vocabulary, hidden_size, input_length=num_steps))
model.add(LSTM(hidden_size, return_sequences=True))
model.add(LSTM(hidden_size, return_sequences=True))
if use_dropout:
    model.add(Dropout(0.5))
model.add(TimeDistributed(Dense(vocabulary)))
model.add(Activation('softmax'))
```

```
optimizer = Adam()
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['categorical_accuracy'])

print(model.summary())
```

如果用户选择 run\_opt=1，则程序是训练模型，然后存储模型到磁盘

```
if run_opt == 1:
    model.fit_generator(train_data_generator.generate(),
len(train_data)//(batch_size*num_steps), num_epochs,
            validation_data=valid_data_generator.generate(),
            validation_steps=len(valid_data)//(batch_size*num_steps))
    model.save(data_path + "final_model.hdf5")
```

如果用户选择 run\_opt=2，则是在测试集上考察模型

#### 8.5.4 高维输出的问题

在上述的语言模型中，输出结果是词。最简单的方法是如上用一个向量来描述词的分布。如果词汇表很大，例如有 10 万个词，因此输出结果是一个长度为 10 万的向量。这需要非常大的计算代价。

传统的做法是限制词汇表的大小，例如限制到 1 万到 2 万。D-softmax(参见论文“Strategies for Training Large Vocabulary Neural Language Models”)是一种可行的方法。但目前没有 Tensorflow 和 Keras 的官方实施。

#### 8.5.5 双向 RNN

图 8.25 的语言模型是一个多层 LSTM 的模型。在处理文本时，双向 RNN 模型可以表现的更好。Keras 提供了一个 wrapper tf.keras.layers.Bidirectional。它可以将一个 rnn 层封装成一个双向 rnn。例如，

```
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential

model = Sequential()

model.add(layers.Bidirectional(layers.LSTM(64, return_sequences=True),
                               input_shape=(5, 10)))
model.add(layers.Bidirectional(layers.LSTM(32)))
model.add(layers.Dense(10))

model.summary()
```

Bidirectional 有个参数 `merge_mode`，是两个方向的 `LSTM` 的输出合并的模式，有几个选择：`'sum'`, `'mul'`, `'concat'`, `'ave'`, `None`。该参数默认是 `concat`，按照帮助手册，`None` 返回一个未处理的 list 结构，例如，[正向 `LSTM`, 反向 `LSTM`]。

上面的例子，用图来描述，见图 8.26

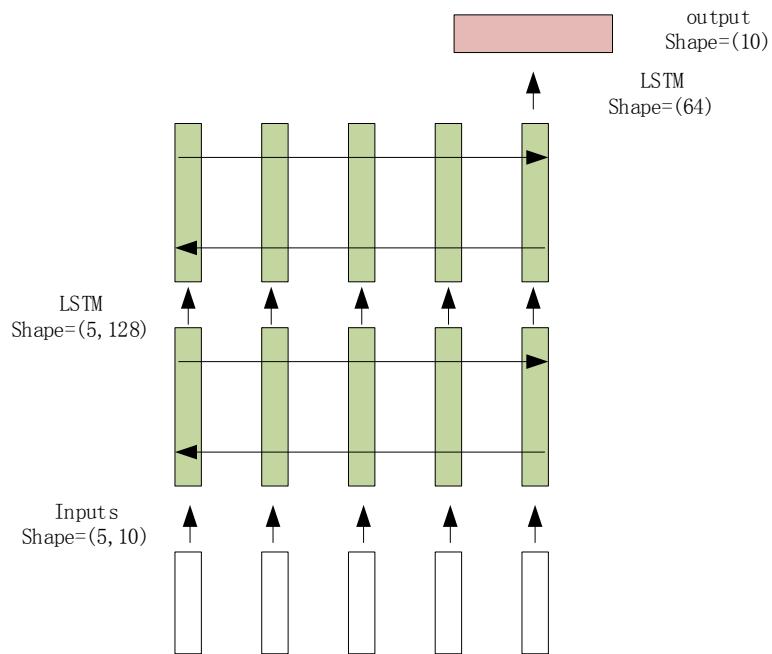


图 8.26 模型

## 第六节：训练 RNN 的一些技巧

<https://danijar.com/tips-for-training-recurrent-neural-networks/>

# 第九章：基于 LSTM 的文本情感分析

## 第一节：介绍

<http://deeplearning.net/tutorial/lstm.html>

这篇文章实施了一个电影评论数据集

( <http://ai.stanford.edu/~amaas/data/sentiment/> ) 的情感分类模型。它完成一个正向、负向情感分类的任务。该文建立的 LSTM 模型是传统 LSTM 模型的一个简化版。

Output gate 不依赖 cell 的状态  $C_t$ 。

注：按照该文的说法，第 8 章我们介绍的 LSTM 就是该简化版，如图 8.1 所示。而传统版计算  $o_t = \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o)$ 。 $V_o$  是一个权重矩阵。

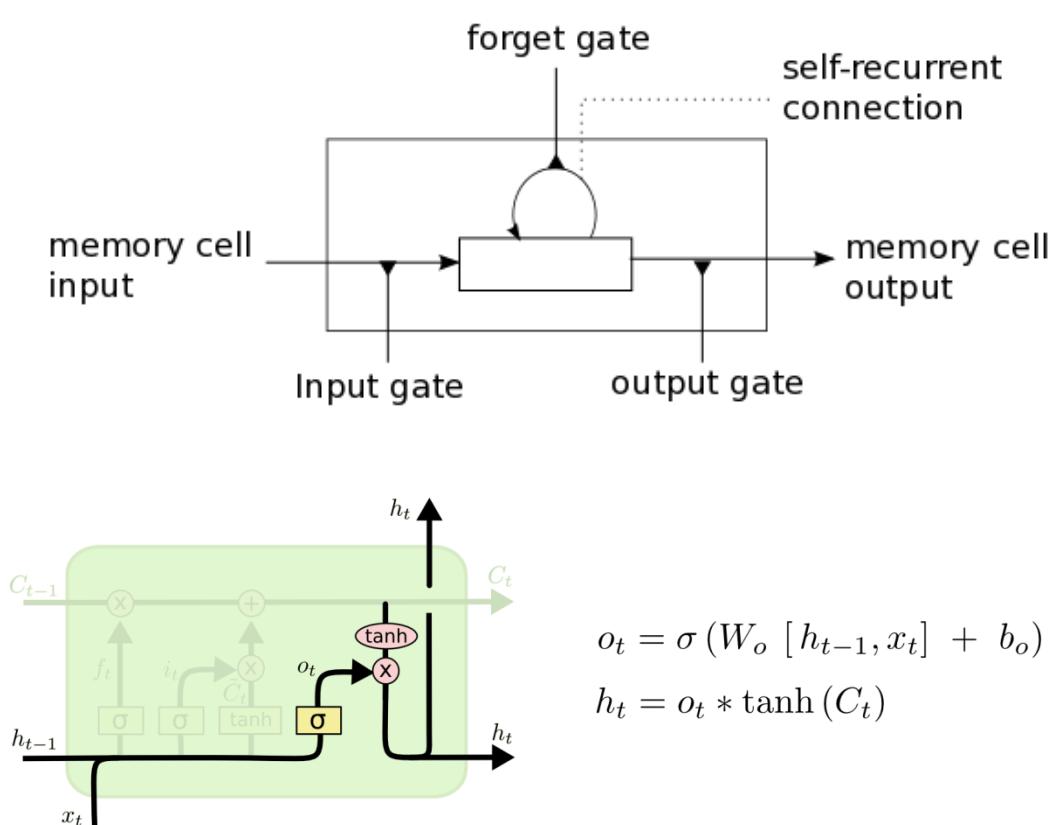


图 9.1 简化版的 LSTM

该文的模型如图 8.2 所示。和传统的 RNN 不同的是在 LSTM 层上加一个 average pooling 层，再加上一个 logistics 回归层。因此从输入序列  $x_0, x_1, x_2, \dots, x_n$ ，LSTM 层的 cell 将产生一个输出序列  $h_0, h_1, \dots, h_n$ 。这个序列被求平均，得到一个输出  $h$ 。最后 logistics 回归层产生一个类标签。

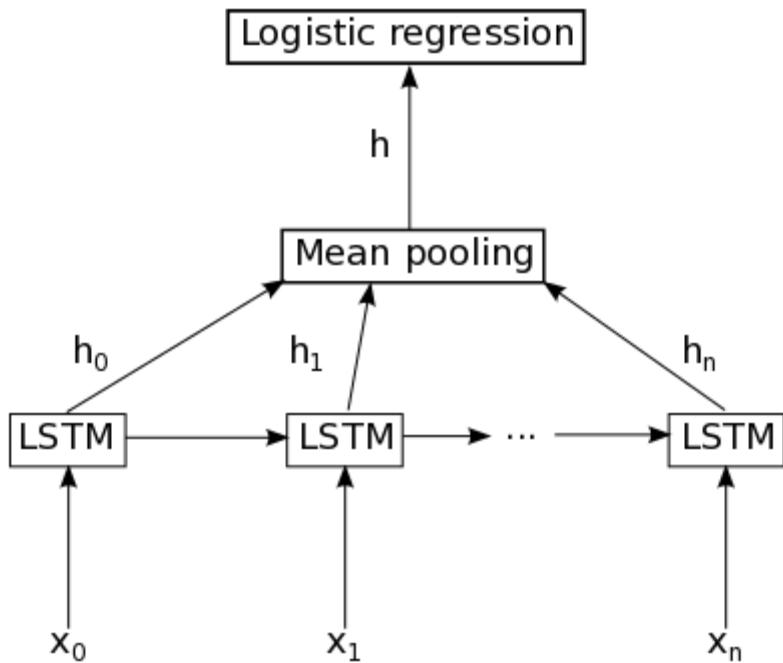


图 9.2 一个评论数据情感分类的 RNN 模型

输入的文本长度做成了固定长度。或者用户自己规定最大长度，或者以训练集中最长文本作为最大长度。不足的补零（词的编号 0，代表没有词）。查找表采用的是随机初始词向量。

## 第二节：keras 实施

为了简化问题，我们没有使用这篇论文提及的电影评论数据集，而是使用 7.3 节的电影评论数据集和它的数据集处理程序。另外，我们也没有去开发该文设计的 cell，而是使用了 LSTM。装载数据集的部分与 7.3 节一样。下面，讲述核心的代码。完整的源码参见 `Istm-sentiment-keras.py`。

```

# build model
inputs=Input(shape=(doc_length,))
embed=Embedding(vocab_size, hidden_size, input_length=doc_length)
embed_input=embed(inputs)
lstm=LSTM(hidden_size, return_sequences=True)(embed_input)

```

```

dropout=Dropout(0.5)(lstm)
meanpool = Lambda(lambda x: mean(x, axis=1))(dropout)

hidden=Dense(100)(meanpool)
output=Dense(1, activation='sigmoid')(hidden)
model = Model(inputs=inputs, outputs=output)

opt = Adam()
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
print(model.summary())

model.fit(x_train, y_train, epochs=epoch, verbose=1)

# evaluate the model
loss, accuracy = model.evaluate(x_dev, y_dev, verbose=0)
print('Accuracy: {} % (accuracy)'.format(accuracy))

```

输入层的 shape=(doc\_length) , doc\_length 是训练集中最长的文档长度，也是时间步的长度。因此，embedding 查找表要规定两个参数 input\_dim 和 output\_dim。input\_dim 是词汇表的大小，output\_dim 词向量的长度。此处，我们设词向量的长度和 lstm 隐层神经元的个数一致，都是 hidden\_size。

在 lstm 层，设置 return\_sequences=True，输出的 tensor 包含每个时间步的结果。

图 8.2 中显示，每个时间步的输出结果会求平均，得到一个向量。Keras 提供了一个 lambda 层，可以利用函数创建自己定制的层。例如，

```

from tensorflow.keras.layers import Lambda
model.add(Lambda(lambda x: x ** 2))

```

在本例子中，lstm 的输出经过了一个 dropout 层，完成 mean pooling 功能的 lambda 层。Lambda 层细节参见附录第五节。

Shape=(时间步，神经元个数)

meanpool = Lambda(lambda x: mean(x, axis=1))(dropout)

此时，dropout 的 tensor 的 shape=(时间步, 一个时间步隐层的大小)。

meanpool 的 tensor 的 shape=(一个时间步隐层的大小)

本文，在 mean pooling 操作后又加了一个全连接层，再接输出层

```

hidden=Dense(100)(meanpool)
output=Dense(1, activation='sigmoid')(hidden)

```

## 第三节：定制 RNN 层的方法

### 1. 定制开发层

用户如果要创建自己的层，可以先学习一下 Layer 这个类。该类封装了一个 state（层的权重），和一些计算。计算将输入转换成输出。如果开发计算复杂的层，需要用到 tensorflow 提供的一些基本功能

[https://www.tensorflow.org/guide/keras/custom\\_layers\\_and\\_models](https://www.tensorflow.org/guide/keras/custom_layers_and_models)

如果构建简单的层可以用 lambda 层，它可以用写 lambda 函数的方式，定制的对输入数据进行转换处理。前一节构建 mean pooling 层采用的就是 lambda 层来定制的。

### 2. 定制开发 RNN 层

我们下面要讲的是，针对 RNN 层可以有自己的定制方法。本节我们建立第一节的 RNN，建立一个 RNN 层。Keras 的类 tf.keras.layers.RNN 可以用于开发定制的 RNN 层。

```
tf.keras.layers.RNN(  
    cell, return_sequences=False, return_state=False,  
    go_backwards=False, stateful=False, unroll=False,  
    time_major=False, **kwargs  
)
```

参数：

cell : 是 keras 的 cell 类的实例

- `return_sequences`: Boolean (default `False`). Whether to return the last output in the output sequence, or the full sequence.
- `return_state`: Boolean (default `False`). Whether to return the last state in addition to the output.
- `go_backwards`: Boolean (default `False`). If True, process the input sequence backwards and return the reversed sequence.
- `stateful`: Boolean (default `False`). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
- `unroll`: Boolean (default `False`). If True, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.

- `time_major`: The shape format of the `inputs` and `outputs` tensors. If True, the inputs and outputs will be in shape `(timesteps, batch, ...)`, whereas in the False case, it will be `(batch, timesteps, ...)`. Using `time_major = True` is a bit more efficient because it avoids transposes at the beginning and end of the RNN calculation. However, most TensorFlow data is batch-major, so by default this function accepts input and emits output in batch-major form.

### Input shape

N-D tensor with shape `[batch_size, timesteps, ...]` or `[timesteps, batch_size, ...]` when `time_major` is True.

### Output shape

- If `return_state`: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each with shape `[batch_size, state_size]`, where `state_size` could be a high dimension tensor shape.
- If `return_sequences`: N-D tensor with shape `[batch_size, timesteps, output_size]`, where `output_size` could be a high dimension tensor shape, or `[timesteps, batch_size, output_size]` when `time_major` is True.
- Else, N-D tensor with shape `[batch_size, output_size]`, where `output_size` could be a high dimension tensor shape.

### 定制开发一个 cell 类，它应该有

- A `call(input_at_t, states_at_t)` method, returning `(output_at_t, states_at_t plus 1)`. The `call` method of the cell can also take the optional argument `constants`, see section "Note on passing external constants" below.
- A `state_size` attribute. This can be a single integer (single state) in which case it is the size of the recurrent state. This can also be a list/tuple of integers (one size per state). The `state_size` can also be `TensorShape` or tuple/list of `TensorShape`, to represent high dimension state.
- A `output_size` attribute. This can be a single integer or a `TensorShape`, which represent the shape of the output. For backward compatible reason, if this attribute is not available for the cell, the value will be inferred by the first element of the `state_size`.
- A `get_initial_state(inputs=None, batch_size=None, dtype=None)` method that creates a tensor meant to be fed to `call()` as the initial state, if the user didn't specify any initial state via other means. The returned initial state should have a shape of `[batch_size, cell.state_size]`. The cell might choose to create a tensor full of zeros, or full of other values based on the cell's implementation. `inputs` is the input tensor to the RNN layer,

which should contain the batch size as its shape[0], and also dtype. Note that the shape[0] might be `None` during the graph construction. Either the `inputs` or the pair of `batch_size` and `dtype` are provided. `batch_size` is a scalar tensor that represents the batch size of the inputs. `dtype` is `tf.DType` that represents the dtype of the inputs. For backward compatible reason, if this method is not implemented by the cell, the RNN layer will create a zero filled tensor with the size of [batch\_size, cell.state\_size]. In the case that `cell` is a list of RNN cell instances, the cells will be stacked on top of each other in the RNN, resulting in an efficient stacked RNN.

创建一个 `cell` 类，然后创建一个 RNN 层的代码如下。这里首先创建一个 `cell` 类，它应该是 `keras.layers.Layer` 类的子类。上面给出关于 `cell` 应该包含的内容中，没有给出一个 `build` 方法，但观察许多 `cell` 的实现，例如 `LSTMCell`，它们都有个 `build` 方法，用来定义 `cell` 中的权重（参数）。注意 `build` 方法的最后一个语句 `self.build=True`，是设置该类的成员变量 `build` 的值为 `True`。表示参数已经建立。在方法 `call` 中则具体完成计算。

```
from tensorflow.keras.layers import Layer, Input, RNN
from tensorflow.keras import backend as K
from tensorflow.keras.models import Model

class MinimalRNNCell(Layer):
    def __init__(self, units, **kwargs):
        self.units = units
        self.state_size = units
        super(MinimalRNNCell, self).__init__(**kwargs)

    def build(self, input_shape):
        self.kernel = self.add_weight(shape=(input_shape[-1], self.units),
                                      initializer='uniform',
                                      name='kernel')
        self.recurrent_kernel = self.add_weight(
            shape=(self.units, self.units),
            initializer='uniform',
            name='recurrent_kernel')
        self.bias = self.add_weight(shape=(self.units * 4,), 
                                   name='bias')

        self.built = True

    def call(self, inputs, states):
        prev_output = states[0]
        h = K.dot(inputs, self.kernel)
        output = h + K.dot(prev_output, self.recurrent_kernel)
        return output, [output]

cell = MinimalRNNCell(32)
x = Input((None, 5))
```

```

layer = RNN(cell)
y = layer(x)
model = Model(inputs=x, outputs=y)
model.summary()

```

class MinimalRNNCell(Layer)

创建一个类 MinimalRNNCell , 它的父类是 tensorflow.keras.layers.Layer。

[https://www.tensorflow.org/api\\_docs/python/tf/keras/layers/Layer](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Layer)

其中的代码 `self.kernel = self.add_weight( ... )`

是在当前层添加 variable。这个是 tensorflow 中的概念 , 即训练模型时 , 要学习的参数。它是 keras.layers.Layer 类的一个方法

```

add_weight(
    name=None, shape=None, dtype=None, initializer=None,
    regularizer=None,
    trainable=None, constraint=None, partitioner=None,
    use_resource=None,
    synchronization=tf.VariableSynchronization.AUTO,
    aggregation=tf.compat.v1.VariableAggregation.NONE, **kwargs
)

```

## 第四节：开发 peephole LSTM 情感分类模型

第 8.4 节介绍了一个 LSTM 的变体 peephole connection LSTM , 如图 9.3 所示。

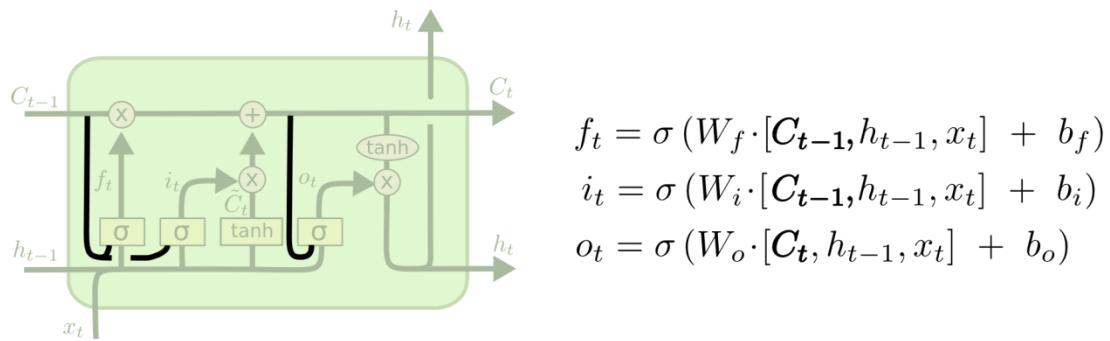


图 9.3 peephole connection LSTM

本节将开发这个 peephole LSTM。除了图 9.3 显示的三个门 ,  $h_t$  和  $c_t$  的计算公式和 LSTM 中的一样 , 如下 :

$$\widetilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

$$C_t = i_t * \widetilde{C}_t + f_t * C_{t-1}$$

$$h_t = o_t * \tanh(C_t)$$

我们获得了 LSTMCell 的原程序创建了一个新的类 PeepholeLSTMCell，只更改了三个门  $f_t$ ,  $i_t$  和  $o_t$  的计算。然后用 RNN 类创建一个 LSTM 层。进一步。用了一个 Bidirectional wraper 在我们定制的 RNN 建立了一个双向 RNN。除了这一部分，模型其他结构和第二节的代码一样。

```
inputs=Input(shape=(doc_length,))
embed=Embedding(vocab_size, hidden_size, input_length=doc_length)
embed_input=embed(inputs)
#lstm=LSTM(hidden_size, return_sequences=True)(embed_input)
cell=PeepholeLSTMCell(hidden_size, implementation=1)
lstm=Bidirectional(RNN(cell, return_sequences=True))(embed_input)
dropout=Dropout(0.5)(lstm)
meanpool = Lambda(lambda x: mean(x, axis=1))(dropout)

hidden=Dense(100)(meanpool)
output=Dense(1, activation='sigmoid')(hidden)
model = Model(inputs=inputs, outputs=output)
```

下面我们把 PeepholeLSTMCell 中更改的关键部分给出。其他部分参考 peephole-sentiment.py 文件。在我们的实施中，注意创建 PeepholeLSTMCell 对象时，需要设定参数 implementation=1.

( 1 ) 在 build 方法中增加 peephole kernel ( 权重矩阵 )

```
def build(self, input_shape):
...
# peephole kernel -- 2020.04
    self.peephole_kernel=self.add_weight(shape=(self.units, self.units * 3),
   name='peephole_kernel',
   initializer=self.recurrent_initializer,
   regularizer=self.recurrent_regularizer,
   constraint=self.recurrent_constraint)

...
# peephole --2020.04
    self.peephole_kernel_i = self.peephole_kernel[:, :self.units]
    self.peephole_kernel_f = self.peephole_kernel[:, self.units: self.units * 2]
    self.peephole_kernel_o = self.peephole_kernel[:, self.units * 2: self.units *
3]
```

( 2 ) 在 call 方法中修改三个门  $f, i, o$  的计算，增加 peephole 部分

```
def call(self, inputs, states, training=None):
...
# add peephole --2020.04
    i = self.recurrent_activation(x_i + K.dot(h_tm1_i,
  self.recurrent_kernel_i)
                                  + K.dot(c_tm1,self.peephole_kernel_i))
    f = self.recurrent_activation(x_f + K.dot(h_tm1_f,
  self.recurrent_kernel_f)
                                  + K.dot(c_tm1,self.peephole_kernel_f))
    c = f * c_tm1 + i * self.activation(x_c + K.dot(h_tm1_c,
  self.recurrent_kernel_c))
    o = self.recurrent_activation(x_o + K.dot(h_tm1_o,
  self.recurrent_kernel_o)
                                  + K.dot(c,self.peephole_kernel_o))
```

当情感分类模型改用 peephole LSTM 后，本文的实验发现，性能有所下降。当然，一方面是提供给用户的代码我们没有进行超参数的精调；另一方面，本文的目的是讲述如何创建定制 RNN 层。性能的提高，留给用户自己完成。

# 第十章：RNN Encoder-Decoder

## 第一节：Encoder-Decoder 模型介绍

RNN Encoder-Decoder 模型是一个 sequence to sequence 模型。该模型将输入序列映射到一个输出序列，而输入序列的长度和输出序列的长度可以是变化的。这种模型最常见的任务是机器翻译。从一种语言的句子（sequence）翻译成另一种语言的句子（sequence）。RNN Encoder-Decoder 模型的想法很简单：（1）一个编码器 Encoder 处理输入序列  $X = (x^{(1)}, \dots, x^{(n_x)})$ ，将它们表示成一个固定长度的向量  $C$ ，通常是 Encoder（是一个 RNN）最后一个时间步的输出；（2）一个解码器 Decoder 以  $C$  作为输入，产生一个输出序列  $Y = (y^{(1)}, \dots, y^{(n_y)})$ 。 $n_x$  和  $n_y$  是变化的。这是一个 sequence-to-sequence 结构，两个 RNN 联合训练，以最大化  $\log P(y^{(1)}, \dots, y^{(n_y)} | x^{(1)}, \dots, x^{(n_x)})$ 。 $(x, y)$  是所有的训练集中的序列对。当然 Encoder-Decoder 结构也有其他形式，例如 10.2 节介绍的翻译模型，就是 Encoder 的状态输出，传递给 Decoder。

该模型最初是在 EMNLP2014 上的论文《Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation》里提出。我们下面就以该论文中的 RNN Encoder-Decoder 模型（该论文还涉及到的其他部分就不介绍了）。注：该文有 EMNLP 版和预打印版。预打印版的附录部分对模型具体的实施做了详细的介绍。本文参见的是预打印版。

一个典型的 RNN Encoder-Decoder 例子如图 10.1 所示。

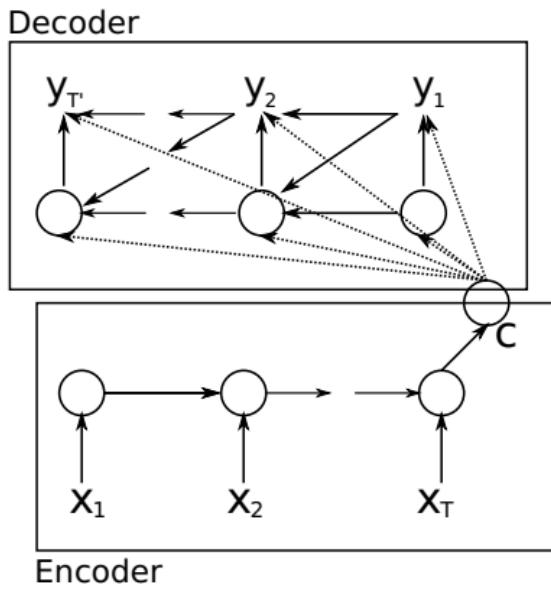


图 10.1 一个 RNN Encoder-Decoder 模型

设输入是一个词的序列  $X=(x_1, x_2, \dots, x_N)$ ，输出序列是  $Y=(y_1, y_2, \dots, y_M)$ 。输入序列和输出序列的长度可以是变化的。

### 1. Encoder

编码器 Encoder 是一个 RNN。

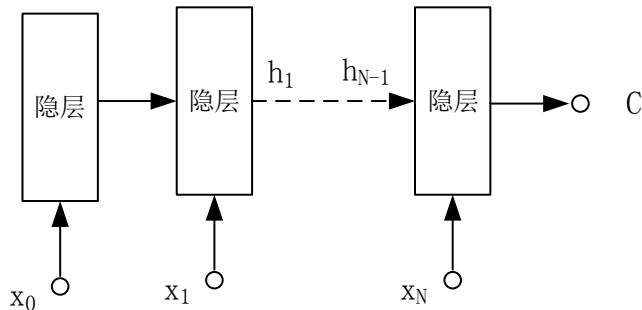


图 10.2 Encoder 结构

RNN 隐层的状态按照下面的公式计算

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

是  $f$ 一个非线性激活函数。 $f$ 即可以是一个简单的 element-wise logistic sigmoid 函数，也可以很复杂，如是一个 LSTM unit。最后一个时间步的输出是一个向量  $c$ 。它是对整个输入序列做的一个总结 (summary) 得到的结果，即编码结果。

原文自己设计了 Unit。给出具体的计算如下。喂入 Encoder 的是一个词的序列，每个词是一个词向量 embedding ( 该文的实验部分使用了维度为 500 的词向量 )，用  $e(x_i) \in \mathbb{R}^{500}$  表示。一个时间步的隐层包含 1000 个 hidden unit ( 即隐层的输出是一个长度为 1000 维的向量 )。时间步  $t$  的第  $j$  个 unit 的计算如下

$$h_j^{\langle t \rangle} = z_j h_j^{\langle t-1 \rangle} + (1 - z_j) \tilde{h}_j^{\langle t \rangle}$$

其中

$$\begin{aligned}\tilde{h}_j^{\langle t \rangle} &= \tanh \left( [\mathbf{W}e(\mathbf{x}_t)]_j + [\mathbf{U}(\mathbf{r} \odot \mathbf{h}_{\langle t-1 \rangle})]_j \right), \\ z_j &= \sigma \left( [\mathbf{W}_z e(\mathbf{x}_t)]_j + [\mathbf{U}_z \mathbf{h}_{\langle t-1 \rangle}]_j \right), \\ r_j &= \sigma \left( [\mathbf{W}_r e(\mathbf{x}_t)]_j + [\mathbf{U}_r \mathbf{h}_{\langle t-1 \rangle}]_j \right).\end{aligned}$$

$e(x_t)$  是词  $x_t$  对应的词向量。 $\sigma$  是一个 logistic sigmoid 激活函数。 $\odot$  是一个逐个元素相乘运算符。初始隐状态  $h_j^{<0>}$  设为 0。编码器的输出是

$$\mathbf{c} = \tanh \left( \mathbf{V} \mathbf{h}^{\langle N \rangle} \right)$$

矩阵  $W, U, V$  都是需要学习的参数。

## 2. Decoder

解码器 Decoder 也是一个 RNN，它用于产生输出序列。Decoder RNN 的隐状态是

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, y_{t-1}, \mathbf{c})$$

而 Decoder 输出序列的第  $t$  个元素的产生是一个条件分布

$$P(y_t | y_{t-1}, y_{t-2}, \dots, y_1, \mathbf{c}) = g(\mathbf{h}_{\langle t \rangle}, y_{t-1}, \mathbf{c})$$

激活函数  $g$  必须能产生概率值。例如，可以是 softmax。原文采用的 Decoder 计算过程如下：

Decoder RNN 的初始状态是

$$\mathbf{h}'^{\langle 0 \rangle} = \tanh (\mathbf{V}' \mathbf{c})$$

符号' 用于区分 Encoder RNN 中的参数。在时间步  $t$  的隐层状态是

$$h'_j^{\langle t \rangle} = z'_j h'_j^{\langle t-1 \rangle} + (1 - z'_j) \tilde{h}'_j^{\langle t \rangle}$$

其中

$$\begin{aligned}\hat{h}'_j^{(t)} &= \tanh \left( [\mathbf{W}' e(\mathbf{y}_{t-1})]_j + r'_j [\mathbf{U}' \mathbf{h}'_{(t-1)} + \mathbf{C} \mathbf{c}] \right), \\ z'_j &= \sigma \left( [\mathbf{W}'_z e(\mathbf{y}_{t-1})]_j + [\mathbf{U}'_z \mathbf{h}'_{(t-1)}]_j + [\mathbf{C}_z \mathbf{c}]_j \right), \\ r'_j &= \sigma \left( [\mathbf{W}'_r e(\mathbf{y}_{t-1})]_j + [\mathbf{U}'_r \mathbf{h}'_{(t-1)}]_j + [\mathbf{C}_r \mathbf{c}]_j \right),\end{aligned}$$

$e(\mathbf{y}_{t-1})$  表示是  $t-1$  步预测的输出的词的词向量。  $e(\mathbf{y}_0)$  是一个全部值为 0 的向量。

在每个时间步，decoder 计算产生词  $j$  的概率

$$p(y_{t,j} = 1 | \mathbf{y}_{t-1}, \dots, \mathbf{y}_1, X) = \frac{\exp(g_j s_{(t)})}{\sum_{j'=1}^K \exp(g_{j'} s_{(t)})},$$

$\mathbf{g}$  是权重矩阵， $g_j$  是权重矩阵的一行。 $s_{(t)}$  的第  $i$  个元素

$$s_i^{(t)} = \max \left\{ s'_{2i-1}^{(t)}, s'_{2i}^{(t)} \right\}$$

其中

$$\mathbf{s}'^{(t)} = \mathbf{O}_h \mathbf{h}'^{(t)} + \mathbf{O}_y \mathbf{y}_{t-1} + \mathbf{O}_c \mathbf{c}$$

可以看到，Decoder 在计算输出序列的一个词时，采用了当前的隐状态  $\mathbf{h}'^{(t)}$ ，序列上一个词  $\mathbf{y}_{t-1}$  和 Encoder 输出的 context  $\mathbf{c}$

## 第二节：实例：一个翻译模型

<https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html>

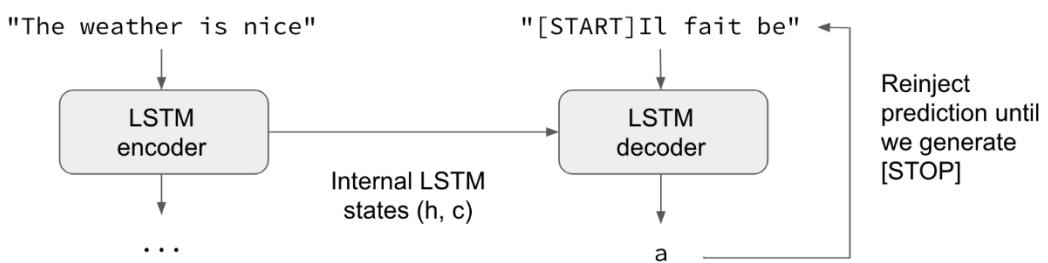


图 10.3 Encoder-Decoder 翻译模型

这是一个字符级的 LSTM Encoder-Decoder 模型。Encoder 是一个 LSTM 模型，对输入数据进行编码。Decoder 是一个基于 LSTM 的语言模型，根据输入预测下一个字符。

实现程序包括：数据集准备、模型的训练和推断（ inference ）三个部分。

## 1. 数据集准备

网站 <http://www.manythings.org/anki/> 提供了许多语言翻译成英语的预料库。比如，法语-英语、中文-英语。下面的模型都可以适用。

当前的程序，把所有的数据都按照 RNN 的时间步划分好并进行了 one-hot 编码。因此有个问题，中文字符集比较大建立的 shape 为 3D 的 np.array 太大导致内存溢出。因此我们减小了训练集样本数为 num\_samples = 10000。

一种解决方法是使用 python generator 来建立数据集，则不需要把所有数据一次都读入到内存。

## 2. 建立模型和训练模型

建立模型和训练模型的程序如下，

```
encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
encoder_states = [state_h, state_c]

decoder_inputs = Input(shape=(None, num_decoder_tokens))
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
                                     initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit([encoder_input_data, decoder_input_data], decoder_target_data,
          batch_size=batch_size,
          epochs=epochs,
          validation_split=0.2)
model.save('s2s.h5')
```

Encoder 模型的输出 encoder\_outputs 后面没有使用。而它的两个状态 encoder\_states = [state\_h, state\_c]，作为为了 decoder 的初始状态。训练好的模型会被保存到磁盘

## 3. 推断（翻译过程）

进行推断的工作原理如下：

（1）encoder 将输入序列编码成状态向量 state h, c。LSTM 输出两个状态 h 和 c。

- (2) 在 Decoder，接收 Encoder 的状态向量作为自己的状态向量的初始值。
- (3) 喂入 Decoder 的输入层一个开始符<START>。Decoder 将预测一个输出字符。
- (4) 将预测字符和前面的字符组合成词的序列，再喂入 Decoder 的输入层，再产生一个输出字符。重复这个过程，直到产生了结束符<END>

推断的核心是先搭建新的模型

```
encoder_model = Model(encoder_inputs, encoder_states)

decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
decoder_outputs, state_h, state_c = decoder_lstm(
    decoder_inputs, initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)
decoder_model = Model(
    [decoder_inputs] + decoder_states_inputs,
    [decoder_outputs] + decoder_states)
```

这段程序非常有意思的是，它利用前面的模型的构件，搭建新的模型。这些构件已经训练好了。新的模型

```
encoder_model = Model(encoder_inputs, encoder_states)
就是前面的
```

```
encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
encoder_states = [state_h, state_c]
```

而新的模型 decoder\_model 用到了前面定义的层 decoder\_dense 和 decoder\_lstm。模型的参数不变，但喂入模型的状态数据不同。

进行推断是一个函数 decode\_sequence，如下。**此处把模型简化了**，一次喂入 decoder 的只有一个 one-hot 编码了的字符，它的 tensor 的 shape=(1,1,字符集大小)。得到的输出包括，一个预测的字符和状态值。再下一趟的预测时，把预测的字符和状态值作为了模型的输入。即一个字符一个字符进行预测，然后拼接成翻译的句子。

```
def decode_sequence(input_seq):
    states_value = encoder_model.predict(input_seq)
    target_seq = np.zeros((1, 1, num_decoder_tokens))
    target_seq[0, 0, target_token_index['\t']] = 1.
    stop_condition = False
```

```

decoded_sentence = ""

while not stop_condition:
    output_tokens, h, c = decoder_model.predict(
        [target_seq] + states_value)

    sampled_token_index = np.argmax(output_tokens[0, -1, :])
    sampled_char = reverse_target_char_index[sampled_token_index]
    decoded_sentence += sampled_char

    if (sampled_char == '\n' or
        len(decoded_sentence) > max_decoder_seq_length):
        stop_condition = True

    target_seq = np.zeros((1, 1, num_decoder_tokens))
    target_seq[0, 0, sampled_token_index] = 1.
    states_value = [h, c]

return decoded_sentence

```

使用 decode\_sequence 函数进行测试的程序如下

```

for seq_index in range(100):
    input_seq = encoder_input_data[seq_index: seq_index + 1]
    decoded_sentence = decode_sequence(input_seq)
    print('-')
    print('Input sentence:', input_texts[seq_index])
    print('Decoded sentence:', decoded_sentence)

```

下面是列出的一些翻译结果

Input sentence: See you.

Decoded sentence: 再见！

Input sentence: Shut up!

Decoded sentence: 一点！

Input sentence: Skip it.

Decoded sentence: 不管它。

Input sentence: I'm fine.

Decoded sentence: 我生病了。

Input sentence: I'm full.

Decoded sentence: 我生病了。

可以看到有些翻译的可以，有些还是不准。上面的模型还是有很大的改善的空间。



# 第十一章：注意力机制

注意力机制 (Attention Mechanism) 在 RNN Encoder-Decoder 模型中应用的广泛。它的思想是给模型注入一种能力，学习一个序列中的哪些“位”对一个给定的任务是最重要的。注意力机制在神经翻译中被 Dzmitry Bahdanau [3] 提出。Attention 的意思就是聚焦于某个事物，给予更大的注意。深度学习中的注意力机制广义上来说是网络结构中的一个构件 (component)。用于处理“相互依赖”的关系，它主要包括：

- (1) 输入和输出的元素之间 (General Attention)
- (2) 在输入元素之间 (self-attention)

## 第一节：动机与原理

### 11.1.1 注意力机制主要解决的问题

2015 年，Dzmitry Bahdanau [3] 等人在《Neural machine translation by jointly learning to align and translate》一文中提出了 Attention Mechanism。该文的动机如下：神经机器翻译 (neural machine translation)，通常使用的是 Encoder-Decoder 模型。Encoder 读进句子，然后把它们转换成一个固定的向量 context  $c$ ，Decoder 基于该向量  $c$  产生一个输出序列 (句子)，如图 11.1 所示。这里是使用 Encoder RNN 最后一个时间步的输出作为 context 向量  $c$ ，因此在计算 Decoder 的每个时间步的输出时，都是使用这个固定的  $c$ 。

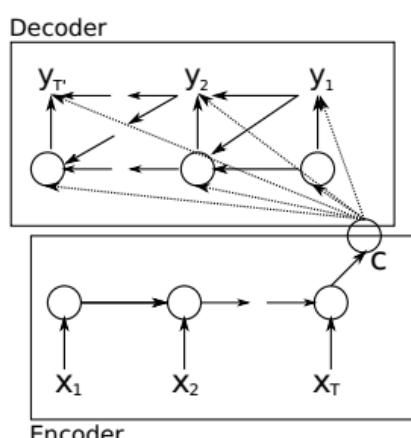


图 11.1 一个基本的 Encoder-Decoder 模型

但这样会存在一个问题 Encoder 需要压缩所有的输入句子的所有的必须信息到一个固定的向量。当遇到长句子的情况，这样的操作会有问题。特别是在实际中的句子甚至都比训练集中的语料库中的句子更长时。有论文指出，当输入序列的长度增加时，Encoder-Decoder 模型的性能会恶化。

该文提出的具有注意力机制的 Encoder-Decoder 模型结构如图 11.1 所示。每次在 Decoder 要产生一个词的输出时，它从输入句子软搜索（soft-search）一系列与当前要输出的这个词相关的“位置”或序列的具体的某几个单元。（注：此处 soft-search 的含义是没有一个明显的搜索行为，而是隐含完成的）。Decoder 在预测一个词的输出时，使用 context c，和前面输出的词的序列。而此时的 context 向量 c 反映出了这个“位置”。和基本 Encoder-Decoder 的区别是在产生 context c 时：基本的 Encoder-Decoder 产生的一个固定的向量 c，而加入注意力机制的模型在产生 c 时，是根据要翻译的词的不同动态的产生 c。而这个动态的 c 从要预测的结果出发，去在输入序列中寻找它应该集中注意的那些“位置”。

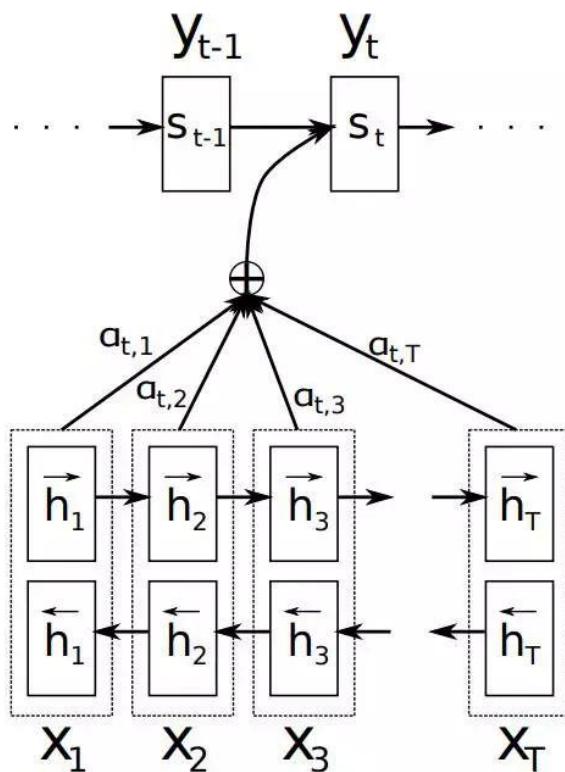


图 11.2 加入了注意力机制的 seq2seq 模型

图 11.2 中，下方为 Encoder，上方为 Decoder，中间是 context 向量 c。加入了注意力机制的 seq2seq 模型的核心是在计算 context 向量 c 时，是用 Encoder 的每个时间步的输出的加权求和。

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

这里  $c_i$  表示为解码器第  $i$  步计算一个 context 向量  $c_i$ 。这个权重  $\alpha$  的计算

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$$

$e_{ij}$  描述了目标词  $y_i$  对齐源词  $x_j$  ( 或是由  $x_j$  翻译过来 ) 的概率。其中，

$$e_{ij} = a(s_{i-1}, h_j)$$

$e_{ij}$  称作 associated energy。它反映了，在决定 Decoder 的时间步  $i$  的隐状态输出  $s_i$ ，并进而产生输出的词  $y_i$  时，Encoder 时间步  $j$  的隐状态输出  $h_j$  的重要性（重要性是依据隐状态  $s_{i-1}$  来计算的）。 $a$  是一个对齐模型 ( alignment model ) 或称作对齐评分函数。在该文中，alignment model  $a$  是一个前馈神经网络。

可以看到，在预测一个词  $y_t$  的输出时，Decoder 使用了前  $t$  个词和专门为该预测产生的向量 context  $c$ 。 $c$  和当前预测词  $y_t$  关注的输入中的其他词的 RNN Encoder 隐层的输出有关。

### 11.1.2 对齐模型

对齐模型或对齐评分函数 ( alignment score function ) 即计算 Encoder 每个时间步的隐状态和 Decoder 每个时间步的隐状态之间相关性评分的函数。评分函数有多种，代表不同的注意力机制。

| Name                   | Alignment score function                                                                                                                                                                                                          |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Content-base attention | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \text{cosine}[\mathbf{s}_t, \mathbf{h}_i]$                                                                                                                                            |
| Additive(*)            | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\mathbf{s}_t; \mathbf{h}_i])$                                                                                                                    |
| Location-Base          | $\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \mathbf{s}_t)$<br>Note: This simplifies the softmax alignment to only depend on the target position.                                                                                  |
| General                | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s}_t^\top \mathbf{W}_a \mathbf{h}_i$<br>where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.                                                             |
| Dot-Product            | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s}_t^\top \mathbf{h}_i$                                                                                                                                                       |
| Scaled Dot-Product(^)  | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \frac{\mathbf{s}_t^\top \mathbf{h}_i}{\sqrt{n}}$<br>Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state. |

最简单的就是两个向量的点乘 ( Dot-Product ) , 但如果向量进行了规范化 ( Scaled Dot-Product ) 则等价于用两个向量的余弦相似度。

拼接方式很常用 , 文献[4]中称为 concat 拼接 , 文献[5]称为 additive attention。 [3] 这篇论文中没有直接指出是拼接方式 , 但它说对齐模型 a 用了神经网络的一层来计算。 [5] 指出这是 additive attention ( 或拼接 ) 。

Locatin-Base[4] 将对齐模型简化了仅仅依赖 Decoder 中的输出。

General 方式[4] 模型比拼接方式要简单。

## 第二节：注意力机制的类型

### 11.2.1 self-attention

Self-attention 又称作 intra-attention。它的将一个单独输入的序列的不同位置进行关联 , 以计算一个向量对这个序列进行表示。在机器阅读 , 摘要任务中很有用[5, 7]。

我们下面以文献[7]为例 , 该文完成一个 sentence embedding 的任务 , 如图 11.3 所示。传统的完成 setence embedding 模型。该模型将 RNN 所有的时间步的输出进行求和或求平均或进行 max pooling。该文认为 , 没有必要这样做 , 而是抽取句子的不同 aspects ( 或特征 ) , 形成多个句子的表示向量。从而在 LSTM 层的顶上再建一个矩阵。

该模型包括两个部分：第一部分是一个双向 LSTM；第二部分是 self-attention mechanism，它提供了一系列的 LSTM 隐状态输出加权求和的结果向量。

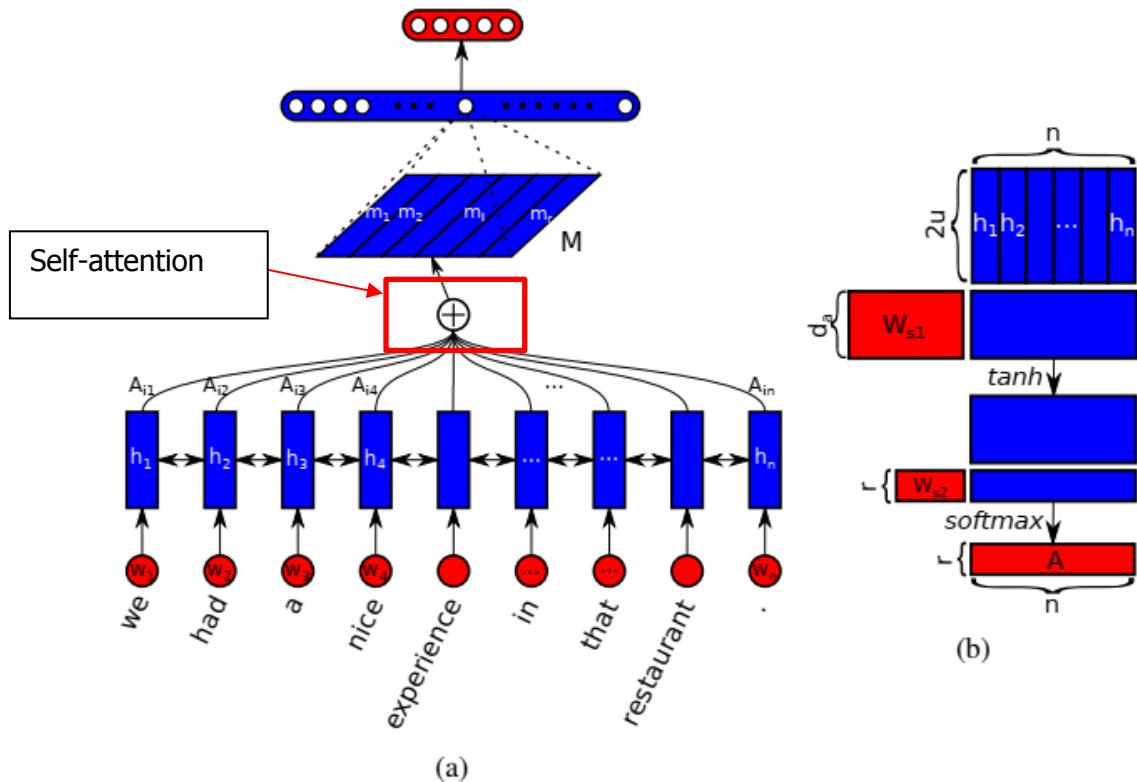


图 11.3 一个自注意力机制的 sentence embedding 模型。 ( a ) 整个模型； ( b ) 注意力机制得到权重矩阵 A

现在有个句子  $S$ ，它有  $n$  个词项，以词向量的序列来描述。

$$S = (\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_n)$$

$\mathbf{w}_i$  是句子中第  $i$  个词的  $d$  维词向量。 $S$  因此是一个拼接了词向量的矩阵。 $S$  的  $\text{shape}=[n, d]$ 。当前序列  $S$  中的每个元素是相互独立的。为了获得向量元素（词）之间的依赖关系，使用了双向 LSTM 来处理这个句子。

$$\overrightarrow{h_t} = \overrightarrow{\text{LSTM}}(w_t, \overrightarrow{h_{t-1}})$$

$$\overleftarrow{h_t} = \overleftarrow{\text{LSTM}}(w_t, \overleftarrow{h_{t+1}})$$

然后将  $\overrightarrow{h_t}$  和  $\overleftarrow{h_t}$  拼接获得一个隐状态  $h_t$ 。一个 LSTM 时间步输出的维度 (hidden unit number) 是  $u$ 。为了，讨论方便用  $H$  表示所有时间步的输出  $H=(h_1, \dots, h_n)$ ， $H$  的  $\text{shape}=[n, 2u]$ 。

该文的目标是把一个长度可变的输入编码成一个固定长度的向量。因此在  $H$  上采用 self-attention 机制来完成。

$$\mathbf{a} = \text{softmax}(\mathbf{w}_{s2} \tanh(W_{s1} H^T))$$

$W_{s1}$  是权重矩阵， $\text{shape}=[d_a, 2u]$ 。 $W_{s2}$  是一个大小为  $d_a$  的向量， $d_a$  是超参数。则向量  $a$  的大小是  $n$ 。然后  $H$  按照向量  $a$  提供的权重求和，如此得到一个向量  $m$ ，它是对输入的句子的一个表示学习得到的向量。

上述步骤得到的一个向量  $m$  只是聚焦于句子的某个方面（专注于某个特征）。然而句子可以有多个特征，特别是对于长句子。希望得到多个  $m$ 。因此，将  $W_{s2}$  扩展成一个  $\text{shape}=[r, d_a]$  的向量。因此注意力机制得到权重矩阵

$$A = \text{softmax}(W_{s2} \tanh(W_{s1} H^T))$$

此处的 softmax 在第二个维度上进行操作。该公式可以看做是一个 2 层的没有偏置的 MLP。隐层的神经元数是  $d_a$ 。参数是  $W_{s1}$  和  $W_{s2}$ 。将权重矩阵  $A$  和  $H$  相乘得到  $M$

$$M = AH$$

再在  $M$  上面构建神经网络，然后可以用于各种任务。比如，该文根据 twitter 用户发表的 tweets 来预测用户的年龄段。

从上面可以看出所谓的 self-attention mechanism 其实就是从输入向量产生权重系数  $a$ 。

在 11.3 节我们实现这个模型。

### 11.2.2 soft attention 和 hard attention

Kelvin Xu 等人与 2015 年发表论文[6]，在 Image Caption 中引入了 Attention，当生成第  $i$  个关于图片内容描述的词时，用 Attention 来关联与  $i$  个词相关的图片的区域。

Kelvin Xu 等人在论文中使用了两种 Attention Mechanism，即 Soft Attention 和 Hard Attention。我们之前所描述的传统的 Attention Mechanism 就是 Soft Attention。Soft Attention 是参数化的（Parameterization），因此可导，可以被嵌入到模型中去，直接训练。梯度可以经过 Attention Mechanism 模块，反向传播到模型其他部分。

相反，Hard Attention 是一个随机的过程。Hard Attention 不会选择整个 encoder 的输出做为其输入，Hard Attention 会依概率  $S_i$  来采样输入端的隐状态一部分来进行计算，而不是整个 encoder 的隐状态。为了实现梯度的反向传播，需要采用蒙特卡洛采样的方法来估计模块的梯度。

两种 Attention Mechanism 都有各自的优势，但目前更多的研究和应用还是更倾向于使用 Soft Attention，因为其可以直接求导，进行梯度反向传播。

### 11.2.3 Global Attention 和 Local Attention

文章[4]中指出，Global Attention 是在计算 Context 向量  $c_t$  时，编码器的所有时间步的隐层输出都要参与计算。

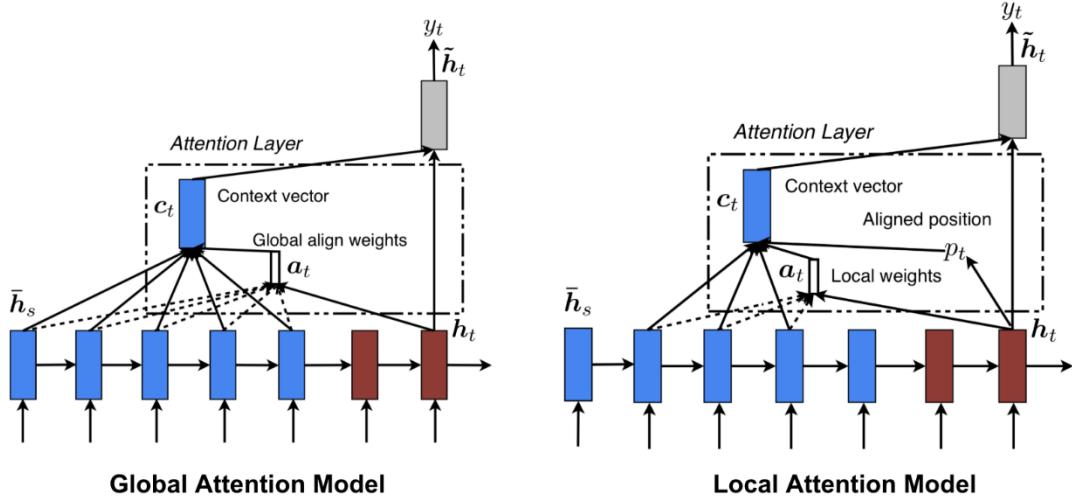


图 11.4 Global 和 Local Attention 模型

一个长度可变的对齐向量 ( alignment vector )  $a_t$ ，它的长度等于 Encoder 输入的时间步的数量 ( 输入序列的长度 )。它是由当前

$$\begin{aligned} \mathbf{a}_t(s) &= \text{align}(\mathbf{h}_t, \bar{\mathbf{h}}_s) \\ &= \frac{\exp(\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_s))}{\sum_{s'} \exp(\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_{s'}))} \end{aligned}$$

Global Attention 模型有一个缺点，每一次，encoder 端的所有 hidden state 都要参与计算，这样做计算开销会比较大，特别是当 encoder 的句子偏长，比如，一段话或者一篇文章，效率偏低。因此，为了提高效率提出了 Local Attention。

### 11.2.4 组合类型的注意力机制

很多研究已经将注意力机制发展的很复杂了。列举如下：

#### Hierarchical Attention

Zichao Yang 等人在论文《Hierarchical Attention Networks for Document Classification》提出了 Hierarchical Attention 用于文档分类。Hierarchical Attention 构

建立了两个层次的 Attention Mechanism，第一个层次是对句子中每个词的 attention，即 word attention；第二个层次是针对文档中每个句子的 attention，即 sentence attention。网络结构如图 11.5 所示。

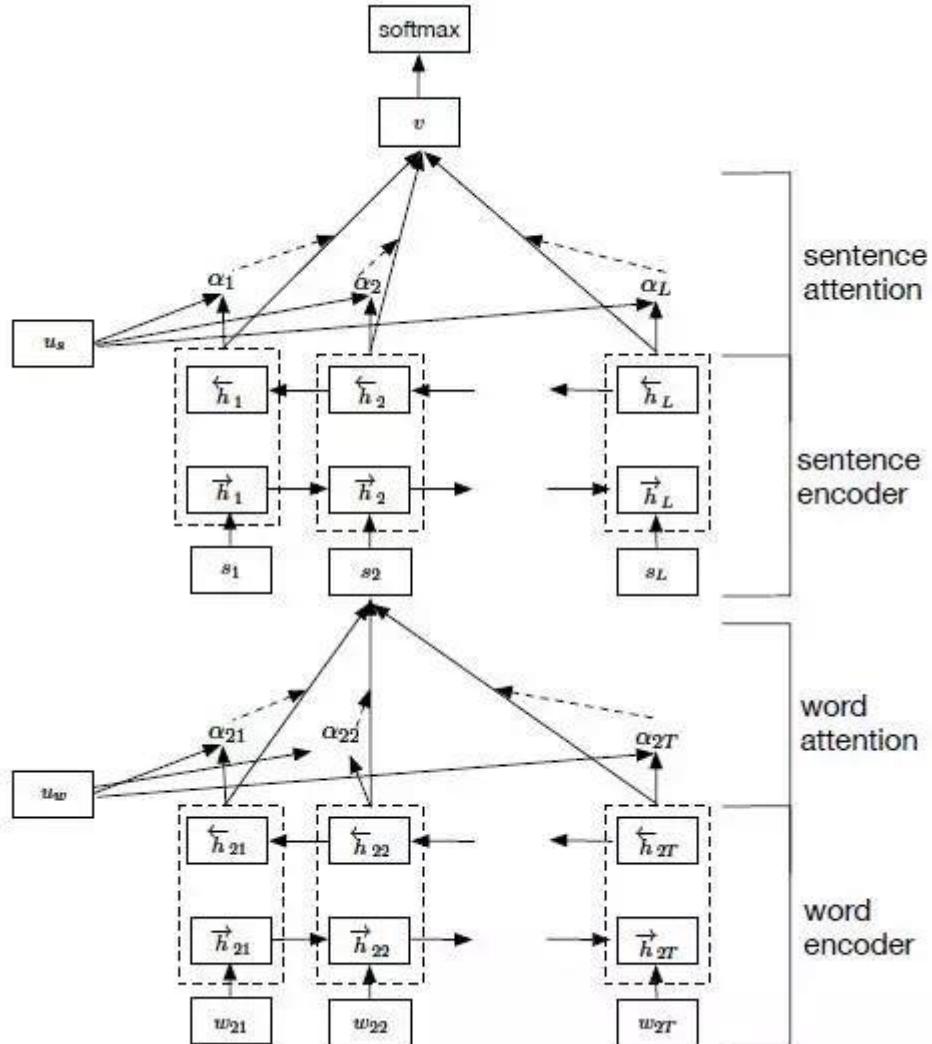


图 11.5

### Attention over Attention

Yiming Cui 与 2017 年在论文《Attention-over-Attention Neural Networks for Reading Comprehension》中提出了 Attention Over Attention 的 Attention 机制，结构如图 11.6 所示。

两个输入，一个 Document 和一个 Query，分别用一个双向的 RNN 进行特征抽取，得到各自的隐状态  $h$  ( doc ) 和  $h$  ( query )，然后基于 query 和 doc 的隐状态进行 dot product，得到 query 和 doc 的 attention 关联矩阵。然后按列 ( column ) 方向进行 softmax 操作，得到 query-to-document 的 attention 值  $a(t)$ ；按照行 ( row ) 方向

进行 softmax 操作，得到 document-to-query 的 attention 值  $b(t)$ ，再按照列方向进行累加求平均得到平均后的 attention 值  $b(t)$ 。最后再基于上一步 attention 操作得到  $a(t)$  和  $b(t)$ ，再进行 attention 操作，即 attention over attention 得到最终 query 与 document 的关联矩阵。

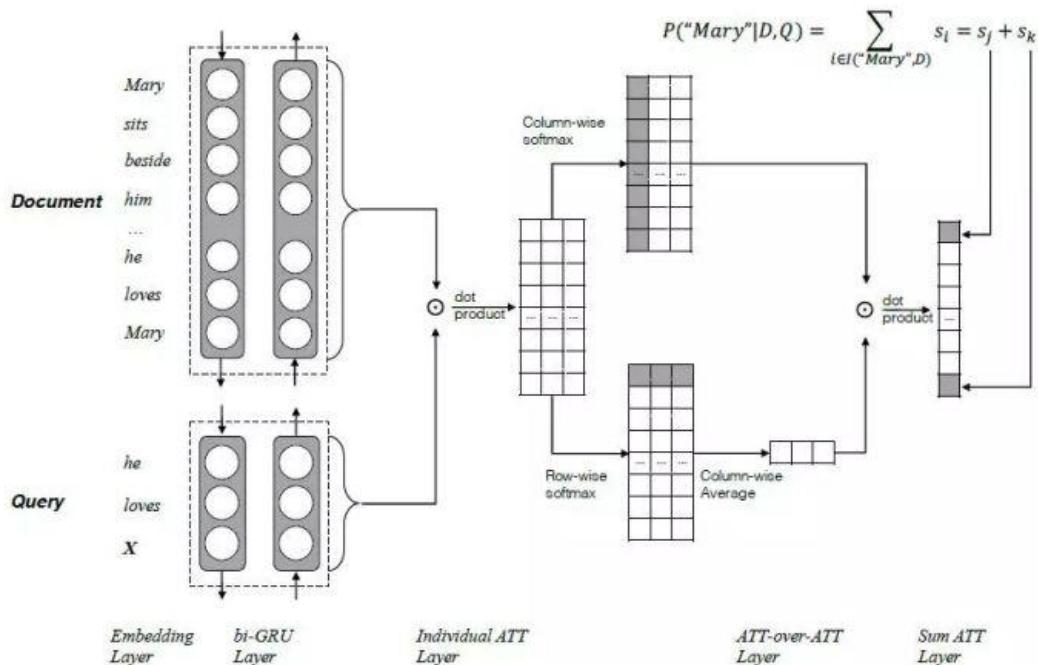


图 11.6

### Multi-step Attention

2017 年，FaceBook 人工智能实验室的 Jonas Gehring 等人在论文《Convolutional Sequence to Sequence Learning》提出了完全基于 CNN 来构建 Seq2Seq 模型，除了这一最大的特色之外，论文中还采用了多层 Attention Mechanism，来获取 encoder 和 decoder 中输入句子之间的关系，结构如图 11.7 所示。

完全基于 CNN 的 Seq2Seq 模型需要通过层叠多层来获取输入句子中词与词之间的依赖关系，特别是当句子非常长的时候，我曾经实验证明，层叠的层数往往达到 10 层以上才能取得比较理想的结果。针对每一个卷记得 step ( 输入一个词 ) 都对 encoder 的 hidden state 和 decoder 的 hidden state 进行 dot product 计算得到最终的 Attention 矩阵，并且基于最终的 attention 矩阵去指导 decoder 的解码操作。

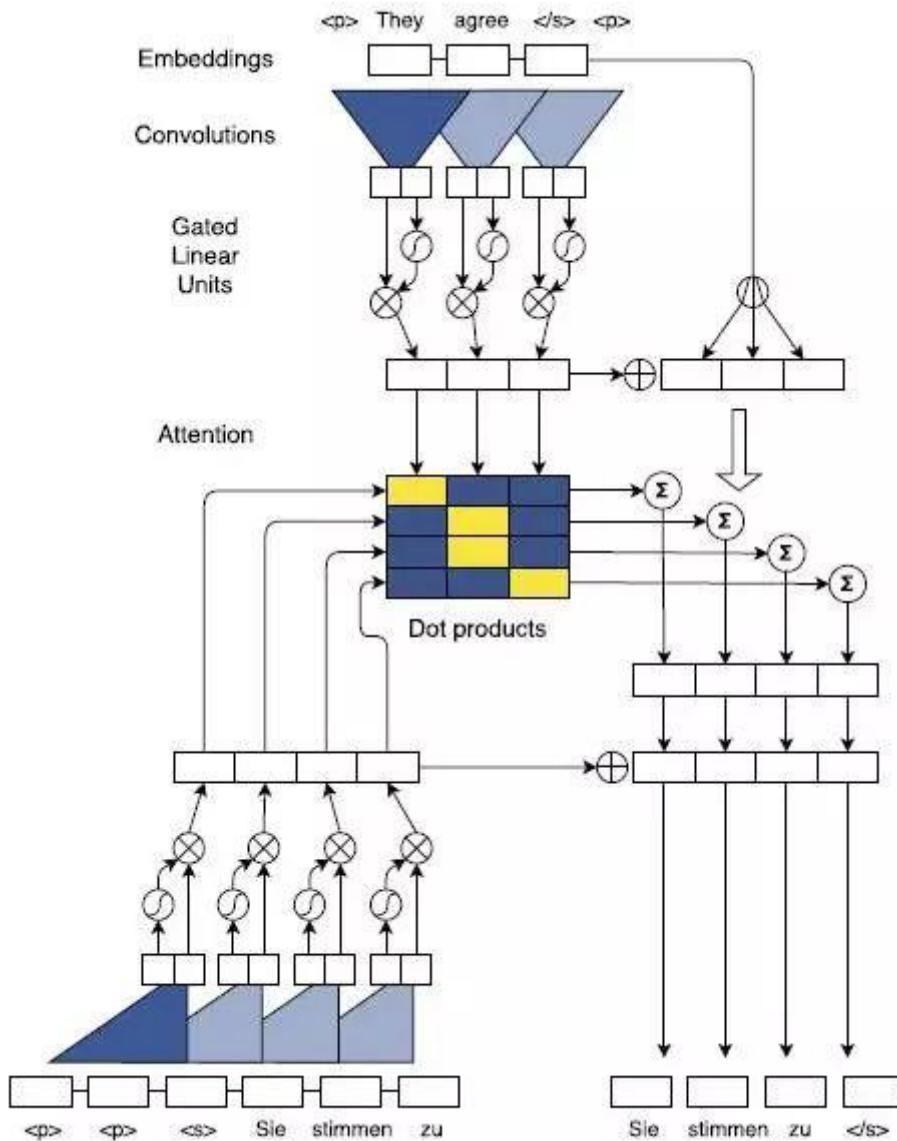


图 11.7

### Multi-dimensional Attention

《Coupled Multi-Layer Attentions for Co-Extraction of Aspect and Opinion Terms》

### Memory-based Attention

## 第三节：Keras 定制正则项和损失函数

做研究时会需要开发自己的损失函数，我们可以考虑一下三种情况：（1）如果只是针对预测结果（ $y_{pred}$ ）和目标数据（ $y_{true}$ ）进行操作，可以定制损失函数；（2）如果损失函数涉及到了模型中间一个层，需要将该层的参数或结果做成正则项添加到损失函数，可以通过给该层添加定制的正则项的方式；（3）一些更复杂的操作，

例如，如果损失函数涉及到了模型多个中间层的结果，采用 Model 对象的 add\_loss() 方法。下面分别介绍这三种操作。

### 11.3.1. 开发定制的正则项

有两种方法开发定制的正则项。一是定义一个函数，其参数默认的是一个层的权重矩阵。函数中针对这个权重矩阵进行正则项计算，例如

```
def l1_reg(weight_matrix):
    return 0.01 * K.sum(K.abs(weight_matrix))
```

在一个层设置正则项时，给出该函数即可。

```
layer1=Dense(10, activation='relu',
             kernel_initializer='he_normal',
             activity_regularizer=l1_reg)(input)
```

但该方法有个缺点，正则项的系数是一个固定值，用户不好调整。

第二种方法是创建一个正则项的类，该类的父类是 tf.keras.regularizers.Regularizer

该类的构造方法中给出一个参数，该参数可以用于调整系数。该类还有个\_call\_方法，该方法的参数对应层的权重矩阵。完整的例子代码如下

```
def l1_reg(x):
    return 0.01 * K.sum(K.abs(x))

class L2Regularizer(tf.keras.regularizers.Regularizer):
    def __init__(self, l2=0.):
        self.l2 = l2

    def __call__(self, x):
        return self.l2 * K.sum(K.dot(x, K.transpose(x)))

    def get_config(self):
        return {'l2': float(self.l2)}
# build model
input=Input(shape=(n_features,))
layer1=Dense(10, activation='relu',
             kernel_initializer='he_normal',
             activity_regularizer=l1_reg)(input)
layer2=Dense(8, activation='relu',
             kernel_initializer='he_normal',
             activity_regularizer=L2Regularizer(l2=0.01))(layer1)
output=Dense(1, activation='sigmoid')(layer2)
```

### 11.3.2. 开发定制的损失函数

定义损失函数必须有两个参数 `y_true` 和 `y_pred`。因此有时添加一些正则项时，定制损失函数的方法就有限。

定制的损失函数里面应该有一个 `loss` 函数，`loss` 函数完成具体的损失函数的计算。

```
def custom_loss():
    def loss(y_true, y_pred):
        bce = K.mean(K.binary_crossentropy(y_true, y_pred), axis=-1)
        return bce

    # Return a function
    return loss
```

然后在模型的 `compile` 函数中，设置损失函数 `loss` 时，用该函数

```
model.compile(optimizer='adam', loss=custom_loss(), metrics=['accuracy'])
```

### 11.3.3 使用 `model.add_loss` 函数

设计更复杂的损失函数时，可以在建立模型后，调用 `model.add_loss()` 函数，其参数是用户自己定义的惩罚项，它将被添加到损失函数里。例如，11.3.1 的例子

```
input=Input(shape=(n_features,))
layer1=Dense(10, activation='relu',
             kernel_initializer='he_normal',
             activity_regularizer=l1_reg)(input)
layer2=Dense(8, activation='relu',
             kernel_initializer='he_normal',
             activity_regularizer=None)(layer1)
#           activity_regularizer=L2Regularizer(l2=0.01))(layer1)
output=Dense(1, activation='sigmoid')(layer2)
model=Model(inputs=input, outputs=output)
a=K.dot(K.transpose(layer2),layer2)
model.add_loss(0.01*K.sum(a))

model.compile(optimizer='adam',
              loss=custom_loss(),
              metrics=['accuracy'])
```

我们下面一节也将采用这种方法在损失函数中添加一个自定义的正则项

## 第四节：实例 1：基于自注意力机制的作者画像

Keras 提供了一个 `attention` 层，它实施的是 Dot-product attention，即 11.1.2 节对齐方式中的 dot-product 方式， $\text{score}(s_t, h_i) = s_t^T h_i$ 。

```
tf.keras.layers.Attention(  
    use_scale=False, **kwargs  
)
```

但可能这个 attention layer 的功能可能过于简单，我们没有发现使用该注意力层的例子。都是自定义开发了 attention layer。

本节我们也是自己开发一个自注意力机制，来实现文献[7]中的 sentence embedding，进而完成用户画像的任务。Sentence embedding 的模型见图 11.3。

我们用的数据集 author-profile.json 是从 PAN2013 author profiling 竞赛的一个数据集里随机抽取出 1 万条构造的 <https://zenodo.org/record/3715864>

数据集包括用户发表的文章和用户的年龄段和性别。本节我们的任务构造模型从文章预测用户的性别。我们实现的就是图 11.3 的模型。最后的输出层是做年龄段的预测。

首先准备数据集的程序如下。从 json 文件里解析数据，分别把文本数据、性别和年龄数据放到三个 list 结构 text、gender 和 age 中。然后用 Tokenizer 类对文本进行编码；用 pad\_sequences 函数将文本补齐到规定的长度 time\_steps。这里 time\_steps 也是 lstm 模型的时间步数。因为，当前数据集的最长文本有 1 万多个词，时间步也为 1 万个，模型训练时间非常的长。因此，我们设超参数 time\_step=1000。我们的目的是想解释模型，不是创建一个最优的模型。

```
#build dataset  
text=[]  
gender=[]  
age=[]  
  
with open('author-profile.json') as f:  
    for line in f.readlines():  
        dic=json.loads(line)  
        text.append(dic['conversation'])  
        gender.append(dic['gender'])  
        age.append(dic['age_group'])  
  
t=Tokenizer(num_words=vocab_size,oov_token=None)  
t.fit_on_texts(text)  
encoded_docs=t.texts_to_sequences(text)  
x = pad_sequences(encoded_docs, maxlen=time_steps, padding='post')  
dic_age={item:id for id,item in enumerate(set(age))}  
y = [dic_age[item] for item in age]  
y = np.array(y)
```

建立模型的代码如下。双向 LSTM 中，Bidirectional 有个参数 merge\_mode，没有设置时，它是将两个 LSTM 的输出，在每个时间步上进行拼接。

## 自注意力机制的实现

$$A = \text{softmax} (W_{s2} \tanh (W_{s1} H^T))$$

$W_{s1} H^T$ 就可以理解为是一个全连接层的计算。因此，我们构建的是一 Dense 层 ws，它的激活函数是 tanh。同理，A 就是 ws 层的输出再经过应 softmax 全连接层的计算。用注意力权重 A 和 LSTM 输出再经过计算时，采用一个 Dot 层得到 M。M 转换成一个向量后，再经过一个 softmax 输出层，进行 age 的预测。该层的 size 是  $\text{len}(\text{set}(\text{age}))$ 。

```
# building model
inputs = Input(shape=(time_steps,))
embed = Embedding(vocab_size, embed_size)
embed_input = embed(inputs)
H = Bidirectional(LSTM(hidden_size, return_sequences=True),
name='H')(embed_input)
ws=Dense(da,activation='tanh',use_bias=False, name='ws')(H)
A=Dense(r,
      activation='softmax',
      use_bias=False, name='A')(ws)
M=Dot(axes=1, name='M')([H,A])

flat=Flatten()(M)
outputs=Dense(len(set(age)),activation='softmax')(flat)
model=Model(inputs=inputs,outputs=outputs)

penal=tf.matmul(tf.transpose(A,[0,2,1]),A)
penal=penal-K.eye(penal.shape[-1])
model.add_loss(tf.norm(penal))
model.summary()
```

在论文[7]构造这个模型时，特别强调在损失函数添加了一个对矩阵 A 冗余度的惩罚项。 $\| AA^T - I \|_F^2$

我们在创建了模型对象 model 后

```
penal=tf.matmul(tf.transpose(A,[0,2,1]),A)
penal=penal-K.eye(penal.shape[-1])
model.add_loss(tf.norm(penal))
```

tf.norm 是 Tensorflow 提供的计算 frobenius norm 的函数。模型训练和校验的代码如下

```
loss=SparseCategoricalCrossentropy()
model.compile(loss=loss,
              optimizer='adam',
              metrics=['categorical_accuracy'])
```

```

model.fit(x_train, y_train, epochs=epoch, verbose=1)

# evaluate the model
loss, accuracy = model.evaluate(x_dev, y_dev, verbose=0)
print('Accuracy: %f' % (accuracy))

```

## 第五节：实例 2：文章标题自动生成

本章结合两篇论文《Deep Keyphrase Generation》[8]和《NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE》[3]，建立一个实施了注意力机制的 RNN Encoder-Decoder 模型。该模型用于从摘要生成文章标题。源码和数据集放在了 [https://github.com/Allen-Qiu/title\\_generate](https://github.com/Allen-Qiu/title_generate)

### 1. 数据集

从 1600+ 论文收集它们的标题和摘要，构建成一个数据集 title.json。

### 2. 工作原理

基于翻译模型的思想，在 encoder 送入文本内容。Encoder 所有时间步的输出根据注意力机制产生一个向量 c。Decoder 是一个 RNN 模型，也是一个语言模型。C 作为 decoder 的初始状态，然后产生一个输出序列，即标题。模型如图 11.8 所示。模型工作原理的形式化描述如下：

Encoder 将长度变化的输入序列  $x = (x_1, x_2, \dots, x_T)$  转换到一个隐状态输出  $h = (h_1, h_2, \dots, h_T)$ 。每个 RNN 时间步的计算如下：

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

$f$  是非线性函数，即 RNN 时间步的一个 cell。本文采用的是双向 GRU。由此可以获得一个 context 向量  $c$

$$c = q(h_1, h_2, \dots, h_T)$$

本文中的函数  $q$  是对所有时间步的输出进行加权求和操作，结果作为  $c$ 。这就是注意力机制。计算过程如下

$$c_i = \sum_{j=1}^T \alpha_{ij} h_j$$

$$\alpha_{ij} = \frac{\exp\{a(s_{i-1}, h_j)\}}{\sum_{k=1}^T \exp\{a(s_{i-1}, h_k)\}}$$

权重 $\alpha_{ij}$ 的计算来自于 Encoder 中时间步 j 的输出 $h_j$ 和 Decoder 时间步 i-1 的输出 $s_{i-1}$ 。函数 $a(s_{i-1}, h_j)$ 计算两个向量 $s_{i-1}, h_j$ 的相似度。这是就是注意力机制中的对齐模型。本文的对齐模型采用的是

$$a(s_t, h_i) = s_t^T W_a h_i$$

Decoder 是另外一个 RNN。本文的 cell 是一个前向 GRU。该模型一个时间步 t 的输出是

$$s_t = f(y_{t-1}, s_{t-1}, c_t)$$

这里 $s_{t-1}$ 是从时间步 t-1 传递给时间步 t 的状态， $y_{t-1}$ 即上一个时间步的预测输出和用注意力机制计算的当前时间步 t 的 context 向量 $c_t$ 被拼接后送入 GRU 一个时间步计算，得到 decoder 时间步 t 的输出 $s_t$ 。进一步经过一个 softmax 全连接层，产生输出的词 $y_t$ 。

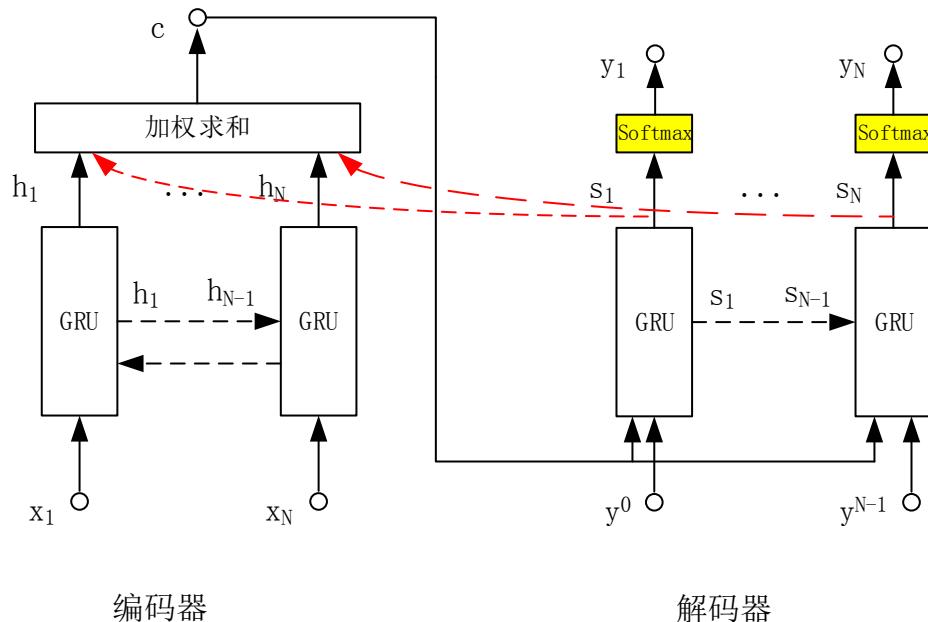


图 11.8 具有注意力机制的 Encoder-Decoder 模型

和 10.2 节的 encoder-decoder 翻译模型一样，模型包括两个阶段。训练阶段和推断阶段（即预测输出）。图 11.8 描述了训练阶段。在推断阶段，使用一个时间步来推断每个输出的词，此时时间步的输入就是上一个预测的词。而 context 向量 c 是在下面的输出层才参与的计算。具体的代码见下一节

### 3. 自动生成文章标题

本项目包括四个文件。title\_parameters.py 保存了超参数设置；title\_dataset.py 处理数据集并保存到一个文件；title\_train.py 训练模型；title\_predict.py 产生文章标题的示例。

## (1) 训练模型

首先，看训练模型。首先建立 encoder

```
from title_parameters import Hyperparameters as hp
encoder_inputs=Input(shape=(hp.encoder_time_steps,), name="einput")
embed = Embedding(hp.vocab_size, hp.embed_size, name='embed')
encoder_input_embed = embed(encoder_inputs)
encoder=Bidirectional(GRU(hp.hidden_size,
                           return_sequences=True,
                           return_state=True
                           ),name='encoder')
encoder_outputs, encoder_states, _ = encoder(encoder_input_embed)
```

hp.encoder\_time\_steps 是从保存有超参数的类 Hyperparameters，获得 encoder 的时间步参数。建立 encoder 的输入后，建立一个词向量的查找表 embed。通过该查找表，把输入 encoder 的词的序列，即摘要部分的文本，喂入一个双向 GRU 构成的 RNN。其中设立 return\_sequences=True 是要获得所有时间步的输出。return\_state=True 规定也获得最后一个时间步的状态。

Decoder 模型部分，包括注意力机制，要复杂一些

```
decoder_inputs = Input(shape=(hp.decoder_time_steps,), name='dinput')
decoder_input_embed = embed(decoder_inputs)
decoder = GRU(hp.hidden_size, return_state=True, name='decoder')
```

喂入 decoder 的部分，是训练模型中的标题，产生编码器模型 decoder。

```
c = K.mean(encoder_outputs, axis=1)
expanded_c=K.expand_dims(c,axis=1)
decoder_state = encoder_states
decoder_outputs = []
times=decoder_input_embed.shape[1]

attention_layer=Dense(hp.hidden_size,
                      name='attention_layer',
                      input_shape=(hp.encoder_time_steps,
                                  hp.encoder_output_hidden_size))
temp = attention_layer(encoder_outputs)
t = tf.transpose(temp,[0,2,1])
```

在实施注意力机制时，对齐模型  $a(s_t, h_i) = s_t^T W_a h_i$  中的一部分  $W_a h_i$  用一个全连接层实现，即上面代码中的 attention\_layer。当编码器 encoder 的模型输出得到后，它就是固定的。

下面的代码完成针对每个解码器的时间步的输出，计算 $a(s_t, h_i)$ ，进而计算权重alpha。这一步的实现，采用了一个小技巧，即把decoder当作只有一个时间步的模型。然后循环多次。循环次数由解码的输入的时间步数规定。

```

for i in range(times):
    one_decoder_input =
        K.concatenate([decoder_input_embed[:,i:i+1,:],expanded_c], axis=-1)
    one_output,decoder_state = decoder(one_decoder_input,
initial_state=decoder_state)
    expanded_s_tm1=K.expand_dims(decoder_state, axis=1)
    E = tf.matmul(expanded_s_tm1,t)
    alpha = K.softmax(E)
    c = tf.matmul(alpha, encoder_outputs)

    decoder_outputs.append(K.expand_dims(one_output, axis=1))

concat_decoder_outputs=Concatenate(axis=1)(decoder_outputs)
decoder_dense = Dense(hp.vocab_size, activation='softmax', name='outputs')
outputs = decoder_dense(concat_decoder_outputs)

```

每趟循环中，将计算的context向量和输入进行拼接，喂入解码器。解码器的初始状态就是上一个时间步的状态。其中，第一个时间步计算时的context向量是，编码器所有时间步输出的均值 $c = K.mean(encoder_outputs, axis=1)$

在每趟循环中， $E = tf.matmul(expanded_s_tm1, t)$ 计算对齐模型 $a(s_t, h_i) = s_t^T W_a h_i$

$\alpha = K.softmax(E)$ 计算注意力权重 $\alpha_{ij} = \frac{\exp\{a(s_{i-1}, h_j)\}}{\sum_{k=1}^T \exp\{a(s_{i-1}, h_k)\}}$

$c = tf.matmul(\alpha, encoder_outputs)$ 计算context向量c。

每个时间步的输出保存到一个list数据结构。然后，所有的时间步输出被拼接

$concat\_decoder\_outputs=Concatenate(axis=1)(decoder\_outputs)$

每个时间步的输出都转换成了一个词，作为最后的输出。

训练模型的代码如下：

```

model = Model([encoder_inputs, decoder_inputs], outputs)
loss=tf.keras.losses.SparseCategoricalCrossentropy()
model.compile(optimizer='adam',
              loss=loss,
              metrics=['sparse_categorical_accuracy'])
model.fit(inputs=[dataset.encoder_input_train, dataset.decoder_input_train],
          outputs=dataset.decoder_target_train,
          batch_size=hp.batch_size,
          epochs=hp.epochs,
          verbose=1)

```

## (2) 建立推断模型

我们再建立两个推断模型，并将它们保存到文件。推断模型就是在实际应用中从文本摘要产生标题的模型，上面的训练模型还不能完成这个任务。推断模型是使用训练好的模型的部分搭建的。编码器的推断模型如下。它的思想是，当用户喂入了文本摘要后，先产生编码器的所有时间步输出和最后一个时间步的状态。

```
encoder_model=Model(inputs=encoder_inputs,  
outputs=[encoder_outputs,encoder_states])  
encoder_model.save('encoder.h5')
```

解码器的推断模型如下。它的思想是，将解码器看做只有一个时间步的模型，它只计算一个时间步的输出。当用户使用解码器推断模型时，自行通过多次循环调用该模型来产生输出的词的序列。

该推断模型根据解码器产生的所有时间步的输出来计算注意力机制的 context 向量。编码器的最后一个时间步的状态作为解码器的初始状态。因此，解码器推断模型包括三个输入层

```
decoder_inputs_infer = Input(shape=(1, ), name='dinput')  
decoder_state_infer = Input(shape=(hp.hidden_size,),  
                           name='sinput')  
encoder_outputs_infer = Input(shape=(hp.encoder_time_steps,  
                                    hp.encoder_output_hidden_size),  
                           name='einput')  
  
decoder_input_embed_infer = embed(decoder_inputs_infer)
```

解码器的第一个输入词是<start>符号。输入解码器的字符通过查找表 embed（它就是在训练模型中的那个查找表 embed）转换成词向量。

下面的代码与训练模型中进行注意力机制计算的原理是一样的。由输入的 encoder 输出计算注意力机制的权重，再计算 context 向量 c。拼接后喂入 decoder 模型（在训练阶段训练的），再经过 softmax 全连接层 decoder\_dense 产生一个词的编号输出。

```
temp_infer = attention_layer(encoder_outputs_infer)  
t_infer = tf.transpose(temp_infer,[0,2,1])  
expanded_s_tm1_infer=K.expand_dims(decoder_state_infer, axis=1)  
E_infer = tf.matmul(expanded_s_tm1_infer,t_infer)  
alpha_infer = K.softmax(E_infer)  
expanded_c_infer = tf.matmul(alpha_infer, encoder_outputs_infer)  
  
one_decoder_input_infer = K.concatenate([decoder_input_embed_infer,  
   expanded_c_infer], axis=-1)  
one_output_infer,state_infer = decoder(one_decoder_input_infer,  
                                       initial_state=decoder_state_infer)  
  
decoder_outputs_infer=K.expand_dims(one_output_infer, axis=1)  
  
output_layer_infer = decoder_dense(decoder_outputs_infer)
```

```

outputs_infer=K.argmax(output_layer_infer, axis=-1)

decoder_model = Model(
    inputs=[decoder_inputs_infer, decoder_state_infer, encoder_outputs_infer],
    outputs=[outputs_infer, state_infer])

decoder_model.save('decoder.h5')

```

### ( 3 ) 预测

实际预测是在 title\_predict.py 中完成。首先读入两个推断模型

```

encoder_model=load_model('encoder.h5')
decoder_model=load_model('decoder.h5')

```

其关键是一个函数 decode\_sequence

```

def decode_sequence(input_seq):
    encoder_outputs, encoder_state = encoder_model.predict(input_seq)
    target_seq = dataset.dic_token_index['<start>']
    stop_condition = False
    decoded_sentence = []
    decoder_state = encoder_state
    target_seq=np.array([[target_seq]])

    while not stop_condition:
        outputs, decoder_state = decoder_model.predict(
            [target_seq, decoder_state, encoder_outputs])

        target_seq=outputs
        token = dataset.dic_index_token[outputs[0,0]]
        decoded_sentence.append(token)

        if (token == '<end>' or
            len(decoded_sentence) > hp.max_decoder_seq_length):
            stop_condition = True

    return decoded_sentence

```

它根据输入的词的序列 input\_seq，首先编码器产生输出。解码器一个词一个词的产生预测的 title，知道产生了结束符<end>或者已经输出了最大长度。

## Reference

- [1] Ming Tan, Cicero dos Santos, Bing Xiang, and Bowen Zhou. 2016. Improved Representation Learning for Question Answer Matching. In Proceedings of the 54th

Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 464–473

- [2] Danqi Chen, Jason Bolton, and Christopher D. Manning. 2016. A Thorough Examination of the CNN/Daily Mail Reading Comprehension Task. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2358–2367
- [3] Dzmitry Bahdanau, KyungHyun Cho, Yoshua Bengio. NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE. ICLR 2015
- [4] Thang Luong, Hieu Pham, Christopher D. Manning. “Effective Approaches to Attention-based Neural Machine Translation.” EMNLP 2015.
- [5] Ashish Vaswani, et al. “Attention is all you need.” NIPS 2017.
- [6] Kelvin Xu et.al.,. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. ICML 2015
- [7] Zhouhan Lin et.al., A STRUCTURED SELF-ATTENTIVE SENTENCE EMBEDDING. ICLR 2017
- [8] Rui Meng, Sanqiang Zhao, Shuguang Han, Daqing He, Peter Brusilovsky, Yu Chi. Deep Keyphrase Generation. ACL2017.

# **第十二章：预训练、Transformer 和 Bert**

# 第十三章：机器阅读理解与问答系统

机器阅读 ( Machine reading ) 这一术语涉及的任务很广泛。包括，阅读文本后回答问题、事实 ( fact ) 和关系抽取、textual entailment 等。

# 第十四章：深度学习在信息检索中的应用

本章内容部分选自 Information Retrieval Journal 2017 年的论文 Neural information retrieval: at the end of the early years。

## 第一节：简介

将神经网络应用在信息检索的任务中，称为 Neural IR。深度学习已经在图像领域取得了成功，在自然语言处理领域也有很长足的进展。信息检索和传统的自然语言处理两个领域是有明确的界限的，例如，NLP 关注的是词法、语法等，而 IR 关注在文本上的检索。现在的发展这两个领域有很多重叠，例如在“问答系统”上两个领域都关注这个问题。

Auto-EncoderNN 是一个无监督模型，用于数据的表示学习。Auto-encoder 用于重构输入数据。输出和输入具有相同的维度。

## 第二节：文本块的表示学习

Compositional distributional semantics 或者 semantic compositionality (SC)是这样一个问题：怎样由词的意思为更大块的文本，例如，词组、句子甚至段落，建立它们的意思。这些工作的原理来自“Principle of Compositionality”。该原理表示，一个复杂表达的意思，由它的组成部分的意思和这些部分的组合规则决定。这些工作在词的表示学习的基础上，进行句子级、段落级的表示学习。

一个基本的处理方法是把句子中各词对应的词向量进行累加而得到句子的分布式向量。这称为 Bottom-Up Method。这种方法很简单，性能有限。更复杂的方法可以分为有监督的方法和无监督的方法。有监督的方法适用于特定的任务，需要有标注的训练集。11.2.1 节介绍的引入 self-attention 机制的 sentence embedding 模型就是一种有监督的句子表示学习模型。无监督的方法适合产生通用的句子表示向量。下面介绍两种无监督方法。它们是基于神经网络模型的表示学习方法：paragraphVector 和 Skip-thought vector。

## 1. ParagraphVector

选自论文 Distributed Representation of Sentences and Documents

ParagraphVector 借鉴了使用神经语言模型学习词向量的思想。

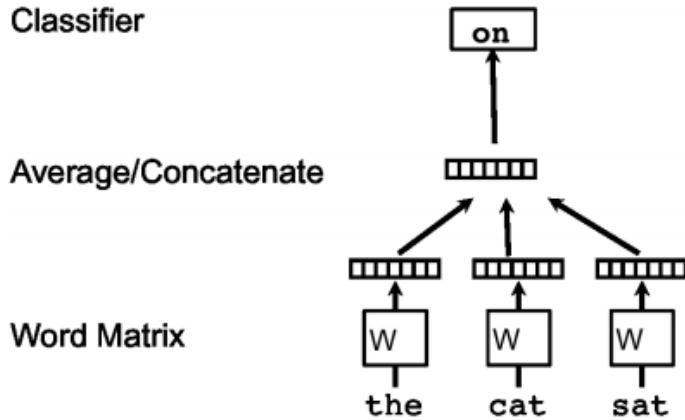


图 1. 神经语言模型

这是一个神经语言模型，输入一个句子，预测该句子后面紧接着的词。

ParagraphVector 为语料库中的段落学习一个向量。段落的向量表示学习的思想是，给定一个段落和段落中的上下文，可以预测一个词出现的概率。

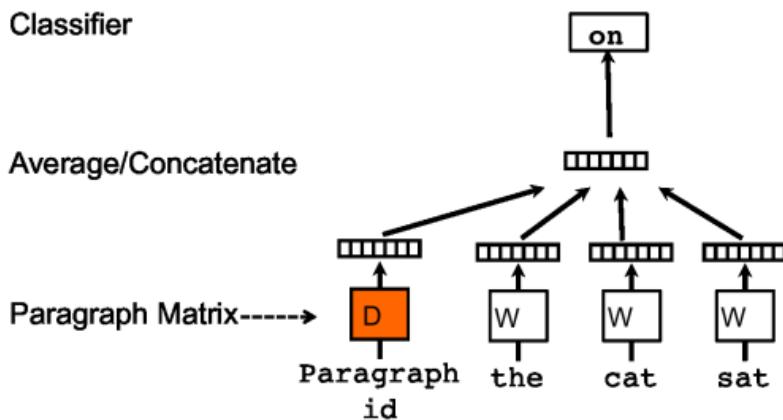


图 2. PV-DM

段落的表示学习模型如图所示。矩阵 D 中的一列就是一个段落的向量，矩阵 W 的一列是一个词向量。上图描述的是每个段落和每个词都映射到一个向量。然后进行拼接（或求均值）。段落 ID 可以被看做是另一个 word。这个 ID 就像是“记忆”，它记住了当前 context 丢失的信息。（paragraph ID, the, cat and sat 是词 on 的 context）。我理解这句话，词项 on 在当前这个段落（paragraph ID），这个段落本身就是 on 的

context。当前明确提供的 context 就是三个词，the, cat, sat。所以这个模型被称为 Distributed Memory Model of Paragraph Vector (PV-DM)。

Context 的长度是固定的（例如，上图 context 是四个词（把段落 ID 也看做是一个词）），是从段落的一个固定大小的窗口滑动中产生的。

该文说，该模型的训练分为了两个阶段，第一个阶段是从当前数据集训练词向量，第二个阶段是用训练好的词向量，产生段落向量。我怀疑，为什么不一次性训练两种向量。

该文还借鉴了 word2vec 的 skip-gram 模型，称为 PV-DBOW ( Distributed Bag of Word version of Paragraph Vector )

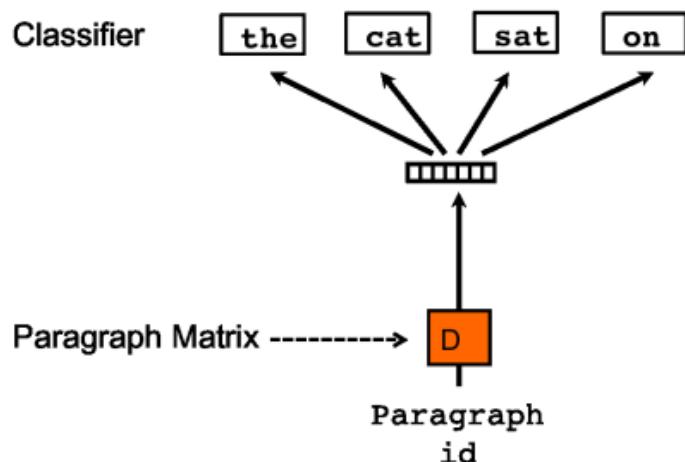


图 3. PV-DBOW

该文没有给出详细的该模型的实施。我猜测可以按照 skip-gram 的实施方法。该文推荐，在实践中使用 PV-DM 和 PV-DBOW 两个模型产生的段落向量的组合，可以取得很好的效果。

## 2. Skip-thought

选自论文 Skip-thought Vectors。该模型是一个 Encoder-Decoder 模型。

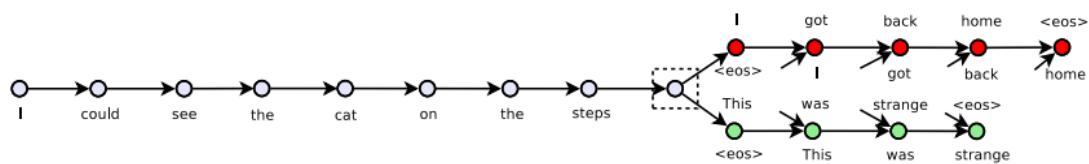


图 11.4. Skip-thought 模型

给定一个三元组 $(s_{i-1}, s_i, s_{i+1})$ ， $s_i$ 是语料库中第*i*条句子。 $s_{i-1}$ 和 $s_{i+1}$ 就是 $s_i$ 的前一条和后一条句子，即 $s_i$ 的 context。该模型的输入是一个句子 $s_i$ ，然后进行编码，再经过解码重构 $s_{i-1}$ 和 $s_{i+1}$ 。虚线框指示的是 Encoder 的输出，后面的节点中，一个未指出‘源’的箭头，表示来自 Encoder 的输出。

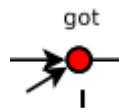


图 11.5 Decoder 一个时间步示例

图 4 中， $s_i$ 是句子“*I could see the cat on the steps*”， $s_{i-1}$ 是“*I got back home*”， $s_{i+1}$ 是 “*This was strange*”。

给定一个句子三元组 $(s_{i-1}, s_i, s_{i+1})$ ， $w_i^t$ 表示句子 $s_i$ 的第*t*个词， $x_i^t$ 表示它的词向量。该模型的三个部分：编码器 Encoder、解码器 Decoder 和目标函数。

需要强调的是图 4 所示的模型是一个训练模型，因此在解码器的模型中，有输入的句子 $s_{i-1}$ 和 $s_{i+1}$ 。在实际应用中，我们需要的是 Encoder 的输出，即虚框。它把一个输入句子编码成了一个向量，这就是我们需要的句子向量。如图 11.5 中，表示解码器的一个时间步的输入是词“*I*”，输入是词“*got*”。

**编码器 Encoder** 是一个 RNN 模型。设 $w_i^1, \dots, w_i^N$ 是句子 $s_i$ 中的词项序列。在每个时间步，encoder 产生一个隐状态 $h_i^t$ ，它可以解释为是词项序列 $w_i^1, \dots, w_i^t$ 的表示学习的结果，即 embeddings。隐状态 $h_i^N$ 则是对整个句子的表示

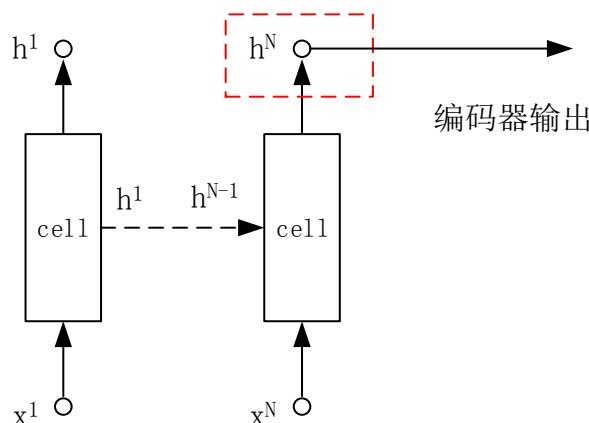


图 11.6 Encoder

编码器中一个 cell 内部的工作过程如下：

$$\begin{aligned}
\mathbf{r}^t &= \sigma(\mathbf{W}_r \mathbf{x}^t + \mathbf{U}_r \mathbf{h}^{t-1}) \\
\mathbf{z}^t &= \sigma(\mathbf{W}_z \mathbf{x}^t + \mathbf{U}_z \mathbf{h}^{t-1}) \\
\bar{\mathbf{h}}^t &= \tanh(\mathbf{W} \mathbf{x}^t + \mathbf{U}(\mathbf{r}^t \odot \mathbf{h}^{t-1})) \\
\mathbf{h}^t &= (1 - \mathbf{z}^t) \odot \mathbf{h}^{t-1} + \mathbf{z}^t \odot \bar{\mathbf{h}}^t
\end{aligned}$$

绘制成图如图 11.7

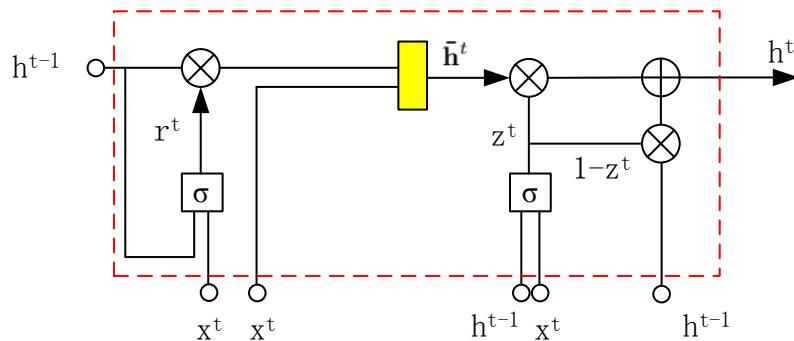


图 11.7 cell 内部结构

**解码器 Decoder** 是一个神经语言模型。该文包含两个解码器，一个用于解码产生  $s_{i-1}$ ，另一个解码产生  $s_{i+1}$ 。两个解码器使用单独的参数，但共同使用一个词汇表矩阵  $V$ 。 $V$  是一个权重矩阵，可以理解为每个词对应一个权重向量。 $V$  用于从解码器的隐层的输出计算词的分布。下面描述的是产生句子  $s_{i+1}$  的解码器，产生  $s_{i-1}$  的解码器工作原理一样。

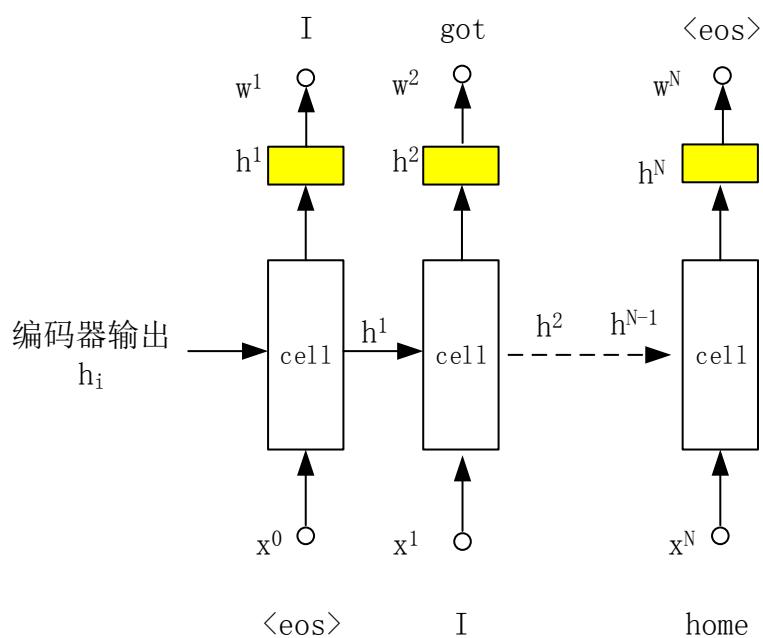


图 11.8 Decoder

Decoder一个时间步的 cell 内部工作原理如下：

设 $\mathbf{h}_{i+1}^t$ 是解码器在时间步 t 的状态输出， $i + 1$ 指示句子 $s_{i+1}$ 。图 11.8 中的黄色块是 softmax 层，它的输出神经元个数是词汇表的大小，即预测一个输出 word。每个时间步的这个 softmax 层共享权重矩阵。

$$\begin{aligned}\mathbf{r}^t &= \sigma(\mathbf{W}_r^d \mathbf{x}^{t-1} + \mathbf{U}_r^d \mathbf{h}^{t-1} + \mathbf{C}_r \mathbf{h}_i) \\ \mathbf{z}^t &= \sigma(\mathbf{W}_z^d \mathbf{x}^{t-1} + \mathbf{U}_z^d \mathbf{h}^{t-1} + \mathbf{C}_z \mathbf{h}_i) \\ \bar{\mathbf{h}}^t &= \tanh(\mathbf{W}^d \mathbf{x}^{t-1} + \mathbf{U}^d (\mathbf{r}^t \odot \mathbf{h}^{t-1}) + \mathbf{C} \mathbf{h}_i) \\ \mathbf{h}_{i+1}^t &= (1 - \mathbf{z}^t) \odot \mathbf{h}^{t-1} + \mathbf{z}^t \odot \bar{\mathbf{h}}^t\end{aligned}$$

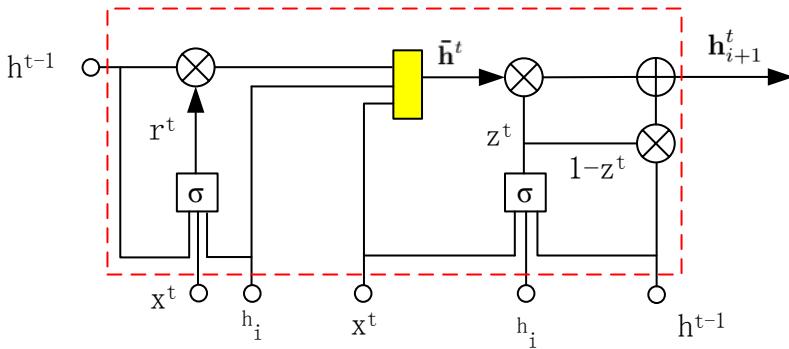


图 11.9 decoder 的 cell 的结构

给定一个时间步 t 的隐状态 $\mathbf{h}_{i+1}^t$ 和前面 t-1 个词 $w_{i+1}^{<t}$ ，预测词 $w_{i+1}^t$ 的概率

$$P(w_{i+1}^t | w_{i+1}^{<t}, \mathbf{h}_i) \propto \exp(\mathbf{v}_{w_{i+1}^t} \mathbf{h}_{i+1}^t)$$

$V_{w_{i+1}^t}$ 是词汇表矩阵 V 中对应词 $w_{i+1}^t$ 的权重向量。 $\exp(V_{w_{i+1}^t} \mathbf{h}_{i+1}^t)$ 表示图 11.8 的 Softmax 层，它输出一个概率最大的词。

## 目标函数

$$\sum_t \log P(w_{i+1}^t | w_{i+1}^{<t}, \mathbf{h}_i) + \sum_t \log P(w_{i-1}^t | w_{i-1}^{<t}, \mathbf{h}_i)$$

给定训练集中的一句三元组 $(s_{i-1}, s_i, s_{i+1})$ ，给定编码器 Encoder 的输出 $\mathbf{h}_i$ 。该目标函数包含两个解码器的预测结果，即在前一个句子 $s_{i-1}$ 和后一个句子 $s_{i+1}$ 上分别计算预测概率的 log 值的和。 $P(w_{i+1}^t | w_{i+1}^{<t}, \mathbf{h}_i)$ 表示对句子 $s_{i+1}$ 根据时间步 t 之前的词项 $w_{i+1}^{<t}$ 和 Encoder 输出 $\mathbf{h}_i$ 预测词项 $w_{i+1}^t$ 的概率。



# Appendix : 常用 keras 的类和函数

## 第一节：数据初始化的类和函数

tf.keras.initializers 类下面包含一些用于初始的子类和方法

( [https://www.tensorflow.org/api\\_docs/python/tf/keras/initializers](https://www.tensorflow.org/api_docs/python/tf/keras/initializers) )

tf.keras.initializers.he\_normal(seed=None)

从一个均值为 0 , 标准差为  $\text{stddev} = \sqrt{2 / \text{fan\_in}}$  的 truncated 正太分布中抽样 ,  
进行初始化。fan\_in 是神经元个数

tf.keras.initializers.he\_uniform(seed=None)

从一个 [-limit, limit] 的均匀分布截获 , limit=  $\sqrt{6 / \text{fan\_in}}$ 。 fan\_in 是神经元个数

tf.keras.initializers.lecun\_normal(seed=None)

从一个均值为 0 , 标准差为  $\text{stddev} = \sqrt{1 / \text{fan\_in}}$  的 truncated 正太分布中抽样 ,  
进行初始化。fan\_in 是神经元个数

从一个 [-limit, limit] 的均匀分布抽样 , limit=  $\sqrt{3 / \text{fan\_in}}$ 。 fan\_in 是神经元个数

## 第二节： keras 常用的激活函数

Keras 激活函数的使用既可以作为层的参数 , 也可以创建一个新的层的方式来使用。新  
创建一个层的好处是可以设置激活函数的参数。Keras 提供的常用激活函数有

### 1. elu

keras.activations.elu(x, alpha=1.0)

指数线性激活函数  $x$  if  $x > 0$  and  $\alpha * (\exp(x) - 1)$  if  $x < 0$

X 是输入的 tensor。它的特点详细见论文“[Fast and Accurate Deep Network Learning by Exponential Linear Units \(ELUs\)](#)”

### 5. softmax

keras.activations.softmax(x, axis=-1)

axis 是一个整数，指出沿着哪一个维度做 softmax 规范化

## 6. selu

```
keras.activations.selu(x)
```

selu 等价于  $\text{scale} * \text{elu}(x, \alpha)$

Computes scaled exponential linear:  $\text{scale} * \alpha * (\exp(\text{features}) - 1)$  if  $< 0$ ,  $\text{scale} * \text{features}$  otherwise。用于训练深层的前馈神经网络。详见  
<https://arxiv.org/abs/1706.02515>

定义如下：

```
def selu(x):
    alpha = 1.6732632423543772848170429916717
    scale = 1.0507009873554804934193349852946
    return scale * K.elu(x, alpha)
```

## 7. softplus

```
keras.activations.softplus(x)
```

softplus 被看做是 relu 的平滑版。Computes softplus:  $\log(\exp(\text{features}) + 1)$ .

在 Goodfellow 的 deep learning 一书中指出，relu 的性能比 softplus 更好，不鼓励使用 softplus.

## 8. Softsign

```
x / (abs(x) + 1)
```

## 9. relu

```
keras.activations.relu(x, alpha=0.0, max_value=None, threshold=0.0)
```

默认值是  $\max(x, 0)$

否则它是  $f(x) = \max\_value$  for  $x \geq \max\_value$ ,  $f(x) = x$  for  $\text{threshold} \leq x < \max\_value$ ,  $f(x) = \alpha * (x - \text{threshold})$

## 10.tanh

```
keras.activations.tanh(x)
```

```
tanh(x) = (\exp(x) - \exp(-x)) / (\exp(x) + \exp(-x))
```

## 11.sigmoid

```
keras.activations.sigmoid(x)
```

```
1 / (1 + \exp(-x))
```

## 12.hard\_sigmoid

```
0 if x < -2.5  
1 if x > 2.5  
0.2 * x + 0.5 if -2.5 <= x <= 2.5.
```

## 13.exponential

$\exp(x)$

### 第三节： keras 常用的损失函数

[https://www.tensorflow.org/api\\_docs/python/tf/keras/losses](https://www.tensorflow.org/api_docs/python/tf/keras/losses)

开发定制的损失函数见 11.3 节

( 1 ) `tf.keras.losses.BinaryCrossentropy`

二分类的交叉熵损失类，对应的损失函数是 `tf.keras.losses.binary_crossentropy`

Use this cross-entropy loss when there are only two label classes (assumed to be 0 and 1). For each example, there should be a single floating-point value per prediction.

```
bce = tf.keras.losses.BinaryCrossentropy()  
loss = bce([0., 0., 1., 1.], [1., 0.1, 1., 0.])  
print('Loss: ', loss.numpy())
```

( 2 ) `tf.keras.losses.CategoricalCrossentropy`

多类别的交叉熵损失类。对应的损失函数是 `tf.keras.losses.categorical_crossentropy`。

参数一个是 one-hot 编码的类别标签，一个是每个类别上的预测的概率值

Use this crossentropy loss function when there are two or more label classes. We expect labels to be provided in a `one_hot` representation. If you want to provide labels as integers, please use `SparseCategoricalCrossentropy` loss. There should be `# classes` floating point values per feature.

The shape of both `y_pred` and `y_true` are `[batch_size, num_classes]`.

```
cce = tf.keras.losses.CategoricalCrossentropy()  
loss = cce(  
    [[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]],  
    [[.9, .05, .05], [.05, .89, .06], [.05, .01, .94]])
```

```
print('Loss: ', loss.numpy()) # Loss: 0.0945
```

( 3 ) tf.keras.losses.MeanSquaredError

均方误差损失函数

```
mse = tf.keras.losses.MeanSquaredError()  
loss = mse([0., 0., 1., 1.], [1., 1., 1., 0.])  
print('Loss: ', loss.numpy()) # Loss: 0.75
```

或者在 compile 中使用

```
model = tf.keras.Model(inputs, outputs)  
model.compile('sgd', loss=tf.keras.losses.MeanSquaredError())
```

( 4 ) tf.keras.losses.SparseCategoricalCrossentropy

计算类别标签和预测值的交叉熵损失函数

Use this crossentropy loss function when there are two or more label classes. We expect labels to be provided as integers. If you want to provide labels using one-hot representation, please use CategoricalCrossentropy loss.

每个真实值给出一个类别对应的整数，而每条记录的预测值是在每一个类上给出一个概率值。

```
cce = tf.keras.losses.SparseCategoricalCrossentropy()  
loss = cce(  
    tf.convert_to_tensor([0, 1, 2]),  
    tf.convert_to_tensor([[.9, .05, .05], [.5, .89, .6], [.05, .01, .94]]))  
print('Loss: ', loss.numpy()) # Loss: 0.3239
```

## 第四节： keras 常用的优化函数

keras 提供了一系列的优化器函数帮助训练模型。这些优化器根据损失函数计算梯度，然后更新参数。参见

[https://tensorflow.google.cn/versions/r1.10/api\\_guides/python/train#Optimizers](https://tensorflow.google.cn/versions/r1.10/api_guides/python/train#Optimizers)

|                              |
|------------------------------|
| tf.keras.optimizers.SGD      |
| tf.keras.optimizers.Adadelta |
| tf.keras.optimizers.Adagrad  |
| tf.keras.optimizers.RMSprop  |
| tf.keras.optimizers.Adam     |
| tf.keras.optimizers.Adamax   |
| tf.keras.optimizers.Ftrl     |
| tf.keras.optimizers.Nadam    |

## 1. **tf.keras.optimizers.SGD**

使用动量的随机梯度下降优化算法。name='SGD'。它的构造函数主要有三个参数：

learning\_rate: float hyperparameter  $\geq 0$ . Learning rate.

momentum: float hyperparameter  $\geq 0$  that accelerates SGD in the relevant direction and dampens oscillations.

nesterov: boolean. Whether to apply Nesterov momentum.

## 2. **tf.keras.optimizers.Adadelta**

Adadelta 优化算法。name='Adadelta'。一个学习率自适应调整的梯度下降算法。它比传统的梯度下降算法有很小的计算开销。它的构造函数的参数包括

learning\_rate=0.001, 学习率

rho=0.95, 衰减率 decay

epsilon=1e-08, 给梯度更新设定的条件

## 3. **tf.keras.optimizers.Adagrad**

Adagrad 优化算法。name='Adagrad'。自适应调整 subgradient 的优化算法。主要是三个参数。

learning\_rate=0.001, 学习率

initial\_accumulator\_value=0.1, 累加器的初始值，必须是非负

epsilon=1e-07, 避免分母为 0 的一个很小的值

## 4. **tf.keras.optimizers.RMSprop**

`tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9)`

结合 plain momentum 的 RMSprop 算法。name='RMSprop'。（注：API 文件上说是实施的 plain momentum 的 RMSprop，但我看其公式就是 4.4 节我们讲的结合 Nesterov momentum 的 RMSprop）

learning\_rate=0.001, 初始学习率

rho=0.9, decay 率

momentum=0.0, 动量系数。4.4 节的 $\alpha$

centered=False, 如果为 True , 梯度会按照梯度估计的方差进行规范化。如果为 True , 会帮助改进训练过程 , 但需要更多的训练时间和更大的内存。

## 5. tf.keras.optimizers.Adam

```
tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999,  
amsgrad=False)
```

Adam 优化算法。name='Adam'。主要参数有

learning\_rate=0.001,

beta\_1=0.9, 一阶动量的指数衰减率 , 4.4 节 Adam 算法中的 $\rho_1$

beta\_2=0.999, 二阶动量的指数衰减率 , 4.4 节 Adam 算法中的 $\rho_2$

amsgrad=False, 是否应用 AMSGrad。论文"On the Convergence of Adam and beyond" 中提到的改进 Adam 的算法。

## 6. tf.keras.optimizers.Adamax

```
tf.keras.optimizers.Adamax(learning_rate=0.002, beta_1=0.9, beta_2=0.999)
```

实施了 adamax 算法。name='Adamax'。它是 adam 算法的变体。有时 adamax 比 adam 更有。特别是在 embeddings 模型中。主要参数同 adam。

## 7. tf.keras.optimizers.Nadam

```
tf.keras.optimizers.Nadam(learning_rate=0.002, beta_1=0.9, beta_2=0.999)
```

实施了 Nadam 算法。name='Nadam'。结合了 Neterrov 动量的 Adam 算法。参数同 adam。《Incorporating Nesterov Momentum into Adam》一文说 , 该算法优于 Adam。

# 第五节： keras 中的层

<https://keras.io/layers/core/>。这里介绍的是一些通用层 , 专用层 , 如 CNN , 会在相应的章节介绍。

## 1. Dense

全连接层

```
keras.layers.Dense(units, activation=None, use_bias=True,  
kernel_initializer='glorot_uniform', bias_initializer='zeros',
```

```
kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,  
kernel_constraint=None, bias_constraint=None)
```

Dense 完成的操作是 `output = activation(dot(input, kernel) + bias)`

Unit 是该层包含的神经元个数。

Activation 是选择的激活函数，不选择这是线性激活函数

Use\_bias 是否使用偏置

Kernel\_initializer 权重初始化方法

Bias\_initializer 偏置初始化方法

Kernel\_regularizer 在训练模型的过程中对权重应用何种正则化项。

Bias\_regularizer 在训练模型的过程中对偏置应用何种正则化项

Activity\_regularizer 在训练模型的过程中对激活函数的输出加何种正则项。

kernel\_constraint 给权重加上约束函数

bias\_constraint 给偏置加上约束函数

在 3.2.2 节中演示的代码，可以加 `input_shape=(3,)` 参数。该参数的使用是当创建一个隐层，同时包括输入层时。

## 2. Dropout

```
keras.layers.Dropout(rate, noise_shape=None, seed=None)
```

dropout 函数是在训练模型时，对前面一层选择一个比例（rate，取值 0-1）的输入 unit，设置他们的值为 0。可以帮助防止过拟合

rate 是对输入神经元放弃的比例

## 3. Flatten

```
keras.layers.Flatten(data_format=None)
```

如果前一个层的输出 tensor 的 shape 是大于一维的，将前一个层的输出转换成一个向量。

Data\_format 是一个字符串，规定了两种前一层 tensor 维度的顺序：

`channels_last` (default) 的顺序是 `(batch, ..., channels)`

```
channels_first 的顺序是 batch, channels, ...)
```

示例代码：

```
model = Sequential()
model.add(Conv2D(64, (3, 3),
                input_shape=(3, 32, 32), padding='same',))
# now: model.output_shape == (None, 64, 32, 32)

model.add(Flatten())
# now: model.output_shape == (None, 65536)
```

## 4. Input

```
keras.layers.Input()
```

input 层用于初始化一个 keras 的 tensor。它是模型的输入层。

它的参数 shape 规定了输入 tensor 的 shape，例如，`inputs = Input(shape=(784,))`

## 5. reshape

```
keras.layers.Reshape(target_shape)
```

把一层的输出 tensor 重新规定它的 shape。例如，

```
model = Sequential()
model.add(Reshape((32), input_shape=(12,)))
```

## 6. permute

```
keras.layers.Permute(dims)
```

把输入 tensor 是维度按照规定排列。相当于是 reshape。例如，

```
model = Sequential()
model.add(Permute((2, 1), input_shape=(10, 64)))
# now: model.output_shape == (None, 64, 10)
# note: `None` is the batch dimension
```

把原来的行和列进行了交换

## 7. RepeatVector

```
keras.layers.RepeatVector(n)
```

重复输入 n 次。例如，

```
model = Sequential()
model.add(Dense(32, input_dim=32))
# now: model.output_shape == (None, 32)
# note: `None` is the batch dimension

model.add(RepeatVector(3))
# now: model.output_shape == (None, 3, 32)
```

当输入是一个 2D 的 tensor 时，输出是一个 3D 的 tensor。相当于增加了一个维度

## 8. Lambda

```
keras.layers.Lambda(function, output_shape=None, mask=None, arguments=None)
```

用写 lambda 函数的方式，定制的对输入数据进行转换处理。创建用户自定义层。例如，

```
# add a x -> x^2 layer
model.add(Lambda(lambda x: x ** 2))
```

再比如，

```
# add a layer that returns the concatenation
# of the positive part of the input and
# the opposite of the negative part

def antirectifier(x):
    x -= K.mean(x, axis=1, keepdims=True)
    x = K.l2_normalize(x, axis=1)
    pos = K.relu(x)
    neg = K.relu(-x)
    return K.concatenate([pos, neg], axis=1)

def antirectifier_output_shape(input_shape):
    shape = list(input_shape)
    assert len(shape) == 2 # only valid for 2D tensors
    shape[-1] *= 2
    return tuple(shape)

model.add(Lambda(antirectifier,
                 output_shape=antirectifier_output_shape))
```

## 9. Masking

```
keras.layers.Masking(mask_value=0.0)
```

对一个序列的部分数据进行遮盖。假设有一个要喂入 LSTM 模型的数据 tensor，它的 shape 是(samples, timesteps, features)。现在想遮盖 sample #0 的 timestep #3 和 sample #2 的 timestep #5。我们可以设置

```
x[0, 3, :] = 0
x[2, 5, :] = 0
```

在一个 LSTM 层前面插入一个 mask 层

```
model = Sequential()
model.add(Masking(mask_value=0., input_shape=(timesteps, features)))
model.add(LSTM(32))
```

## 10. Lambda

```
tf.keras.layers.Lambda(  
    function, output_shape=None, mask=None, arguments=None,  
    **kwargs  
)
```

用 tensorflow 的函数来构建任意的层。例如，

```
def antirectifier(x):  
    x -= K.mean(x, axis=1, keepdims=True)  
    x = K.l2_normalize(x, axis=1)  
    pos = K.relu(x)  
    neg = K.relu(-x)  
    return K.concatenate([pos, neg], axis=1)  
  
model.add(Lambda(antirectifier))
```

## 第六节：Merge 层

<https://keras.io/layers/merge/>

keras 提供了各种层的合并操作，包括加、减、乘、除、拼接等。有两种使用方法，或者创建一个合并操作的层，或者使用方法

### 1. Add

一个类，创建一个层。对应的 add 方法也可以

### 2. Concatenate

```
keras.layers.Concatenate(axis=-1)
```

一个类，创建一个拼接层。默认是在最后一个维度上拼接。示例代码

```
from tensorflow.keras.layers import Input  
from tensorflow.keras.layers import Dense  
from tensorflow.keras.layers import Concatenate  
  
hidden_nums=10  
input_img = Input(shape=(37,))  
hn = Dense(hidden_nums, activation='relu')(input_img)  
out_u = Dense(37, activation='sigmoid')(hn)  
out_sig = Dense(37, activation='linear')(hn)  
out_both = Concatenate()([out_u, out_sig])
```

Out\_both 的 shape=(None, 74)

如果用的是 concatenate 方法，示例代码如下

```
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import concatenate

hidden_nums=10
input_img = Input(shape=(37,))
hn = Dense(hidden_nums, activation='relu')(input_img)
out_u = Dense(37, activation='sigmoid')(hn)
out_sig = Dense(37, activation='linear')(hn)
out_both = concatenate([out_u, out_sig])
```

### 3. Dot

```
keras.layers.Dot(axes, normalize=False)
```

一个类，将两个向量点乘。对应的方法是 **dot**。在 6.3 节有使用 dot 方法的例子，转换成 Dot 的代码如下

```
from tensorflow.keras.layers import Dot

input_target = Input((1,))
input_context = Input((1,))

embedding = Embedding(vocab_size, vector_dim, input_length=1,
name='embedding')

target = embedding(input_target)
target = Reshape((vector_dim, 1))(target)
context = embedding(input_context)
context = Reshape((vector_dim, 1))(context)
dot_product = Dot(axes=1, normalize=False)( [target, context])
```

### 4. Average

```
keras.layers.Average()
```

将输入看做是一个 tensor 的 list。返回一个 tensor

### 5. Multiply

```
keras.layers.Multiply()
```

逐元素相乘。

下面是 dot, multiply 和 backend 的 K 中的 dot 的例子

dot 层和 K.dot 可以实现矩阵乘。但 K.dot 对于维度大于 2 的 tensor，相乘的结果是按 theano 的方法，详细情况查看该函数的帮助

```
from tensorflow.keras.layers import Input
from tensorflow.keras import backend as K
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dot, Multiply
import numpy as np

x = Input(shape=(5))
y = Input(shape=(5))
y2=K.transpose(y)
output1=Dot(axes=1)([x,y])
output2=Multiply()([x,y])
output3=K.dot(x,y2)
model = Model(inputs=[x,y], outputs=[output1,output2, output3])

model.summary()

in1=np.array([[1,2,3,4,5]])
in2=np.array([[6,7,8,9,10]])

out1,out2,out3=model.predict([in1,in2])
print(out1)
print(out2)
print(out3)
```