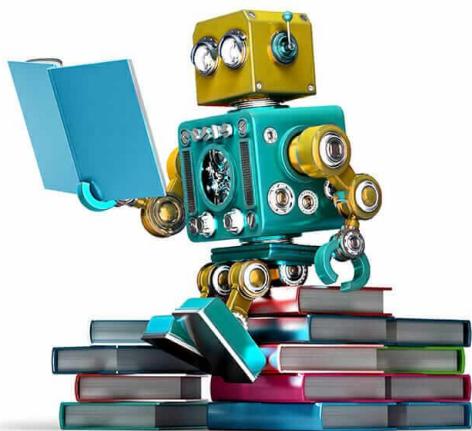




深度学习与文本挖掘

DEEP LEARNING FOR TEXT MINING

邱江涛



2025, TF.KERAS 2.X 版
西南财经大学计算机与人工智能学院

第一章：前言	7
第一节：深度学习的兴起	7
第二节：深度学习框架介绍	10
第三节：关于本讲义	10
第二章：神经网络基础	11
第一节：神经网络的发展	11
第二节：神经网络的结构	13
第三节：神经网络的训练	17
第四节：反向传播算法	23
第三章： Tensorflow 和 Keras	26
第一节： keras 中的基本概念	27
第二节： 模型的构建与训练	30
第三节： keras 练习	34
第四节： tf.data.Dataset	41
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	49
第四节：训练深度神经网络	50
第五节：实例：手写数字识别	62
第六节：解决类不平衡问题	66
第五章：卷积神经网络	75
第一节：卷积	75
第二节：卷积神经网络的结构	78

第三节：Keras 卷积层函数	87
第四节：Dropout	91
第五节：实例 1：基于 CNN 的手写数字识别	93
第六节：实例 2：基于 1D CNN 的活动识别	95
第六章：神经语言模型和词的表示学习	99
第一节：背景知识	99
第二节：word2vec	104
第三节：Keras 实现词表示学习	107
第七章：基于 CNN 的文本分类	117
第一节：文本处理基础	117
第二节：Keras 的文本处理	124
第三节：一个简单的文本分类模型	127
第四节：CNN 文本分类模型	131
第八章：循环神经网络	143
第一节：RNN 结构	143
第二节：Keras 中构建 RNN	148
第三节：LSTM	154
第四节：LSTM 的变体	160
第五节：基于 RNN 的时间序列预测模型	162
第六节：一些需要强调的问题	165
第九章：定制 RNN 与 RNN 文本情感分析	167
第一节：定制 RNN 层的方法	167
第二节：基于定制 RNN 的活动识别	171
第三节：基于 RNN 的文本情感分析	172

第四节：文本情感分析的 keras 实施	174
第五节：开发 peephole LSTM 情感分类模型	177
第十章：RNN Encoder-Decoder	181
第一节：Encoder-Decoder 模型介绍	181
第二节：实例：实现一个翻译模型	184
第十一章：注意力机制	190
第一节：动机与原理	190
第二节：注意力机制的类型	194
第三节：Keras 定制正则项和损失函数	202
第四节：实例 1：基于自注意力机制的作者画像	204
第五节：实例 2：文章标题自动生成	206
Reference	212
第十二章：Transformer 架构	213
第一节：Transformer 介绍	213
第二节：实例：使用 Transformer 进行文本分类	219
第三节：Transformer 的解码策略	223
第四节：MoE Transformer	226
第五节：DeepSeek 的架构	229
第十三章：预训练语言模型	236
第一节：预训练语言模型介绍	236
第二节：认识 Hugging Face 的 Transformers 库	247
第三节：使用 Transformers 库	250
第四节：实例 1：基于 Bert 预测 masked token	262
第五节：实例 2：基于 LLaMA 的小样本学习	263

第十四章：自编码器.....	270
第一节：自编码器.....	270
第二节：定制训练过程.....	272
第三节：变分自编码器.....	273
第四节：生成对抗网络.....	281
第五节：对抗自编码器.....	286
第六节：文本对抗自编码器.....	290
第十五章：相关基础知识	291
第一节：线性代数.....	291
第二节：概率论.....	295
第三节：信息论.....	300
第四节：数值计算.....	303
第五节：机器学习基础.....	307
附录 A：常用 keras 的类和函数	315
第一节：数据初始化的类和函数.....	315
第二节：keras 常用的激活函数	315
第三节：keras 常用的损失函数	317
第四节：keras 常用的优化函数	319
第五节：keras 中的层	321
第六节：Merge 层	325
第七节：Backend Function	327
第八节：创建一个自定义层	330
第九节：tf.Variables	332
附录 B：函数光滑	333

附录 C: 多 GPU 训练模型.....	337
附录 D: 一些技巧.....	339
附录 E: 出错信息处理.....	341
附录 F: Tensorflow Hub and Addons	343
附录 G: Tensorflow 的一些数据操作函数.....	344
附录 H: Encoder-Decoder+注意力机制的英中翻译模型.....	345

第一章：前言

第一节：深度学习的兴起

2025年初，科技领域最火热的一个话题就是 DeepSeek 的 AI 大模型以低成本、高性能超过了大多数 AI 大模型产品，以 DeepSeek 为代表的 AI 大模型已经在非常多的领域成功应用，包括知识问答、文本、图片、语言生成等。AI 大模型就是一种深度学习模型。



而上一次深度学习的火爆出圈可以追溯到 2016 年。当年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，AlphaGo 以 3:0 血洗当时排名第一的柯洁。



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 即强化学习，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。

建立深度学习的初步印象

可以访问一个深度学习的演示网站：playground.tensorflow.org 了解深度学习的基本结构和强大功能。

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。

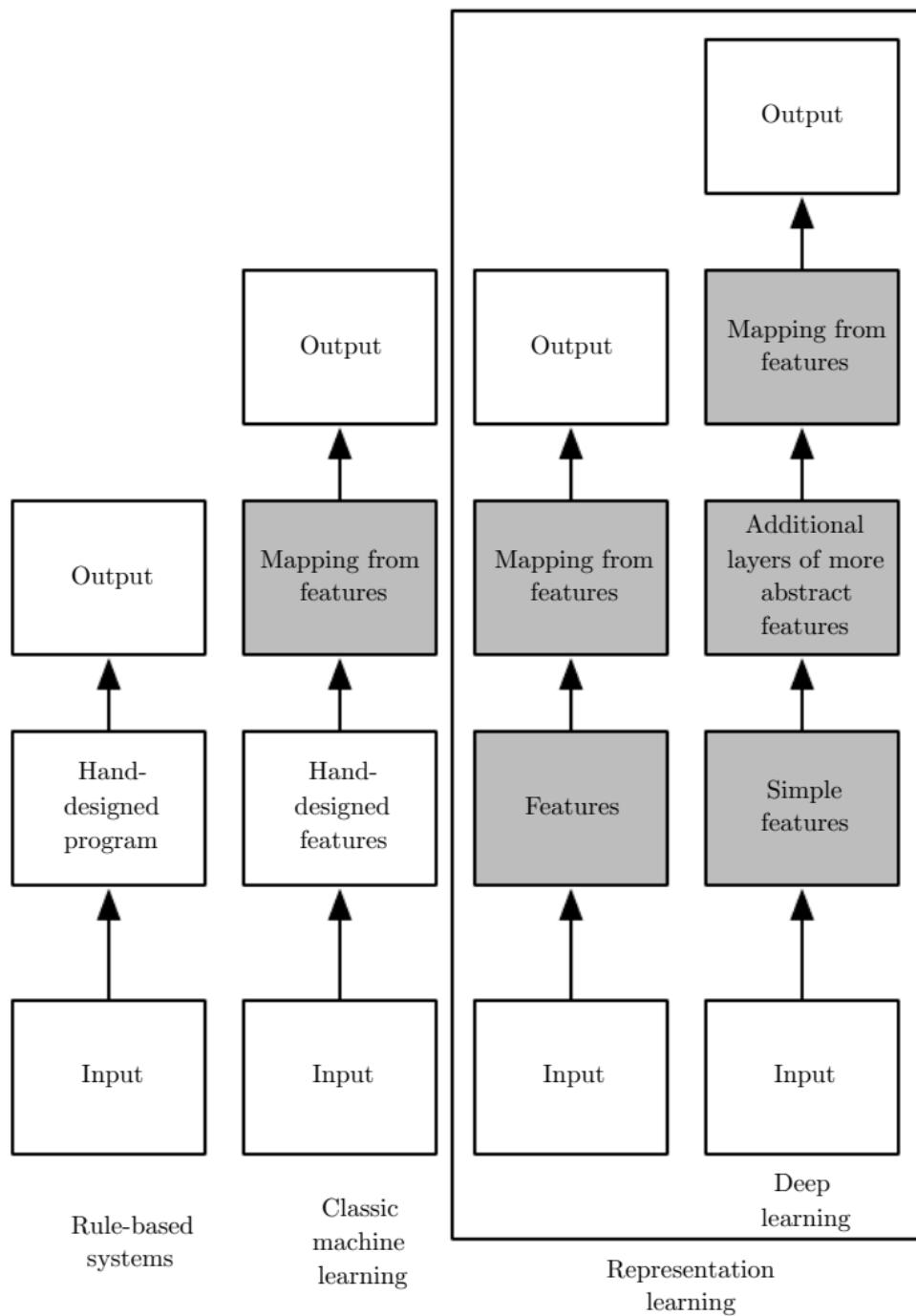
深度学习模型比起传统神经网络的优势在于：由于深度学习可以建立远远超过三层的深度神经网络，他具有远更强大的 capacity，如此可以解决更难、更复杂的问题。

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。

深度学习与人工智能中其他理论的关系



第二节：深度学习框架介绍

深度学习框架是用于构建、训练和部署深度学习模型的软件工具和库。它们提供了预定义的函数、类和模块，简化了神经网络的设计和实现过程。常见的深度学习框架包括：

TensorFlow

由 Google 开发，支持多种语言，主要用于研究和生产。提供高级 API 如 Keras，适合快速开发。

PyTorch

由 Facebook 开发，以动态计算图著称，适合研究和实验。提供灵活性和易用性，社区支持强大。

Keras

最初由 François Chollet 开发，现在主要由谷歌支持和维护的开源深度学习库，通常与 TensorFlow 结合使用，适合初学者。提供简单接口，便于快速构建和训练模型。

第三节：关于本讲义

在本讲义中，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。

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。

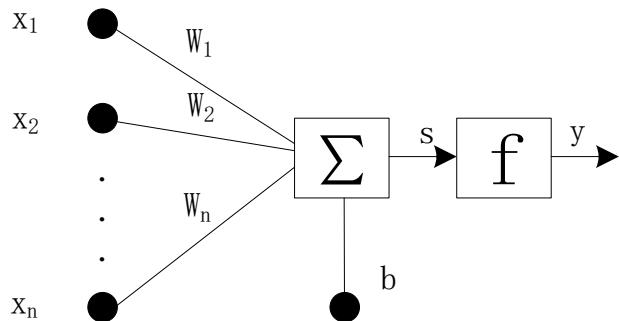


图 2.1 感知机

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。

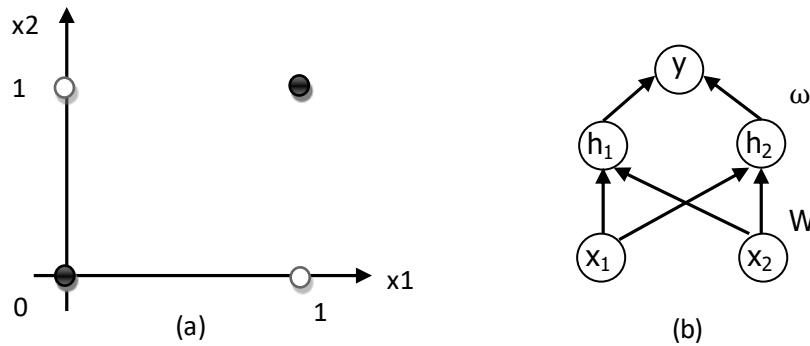


图 2.2 XOR 问题 (a) 和加入隐层的感知机 (b)

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； ω 是隐层到输出层的边的权重； c 是隐层两个节点的偏置； b 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。

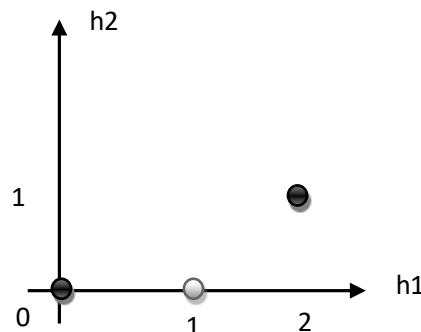


图 2.3 在隐层空间的转换后的输入数据

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。

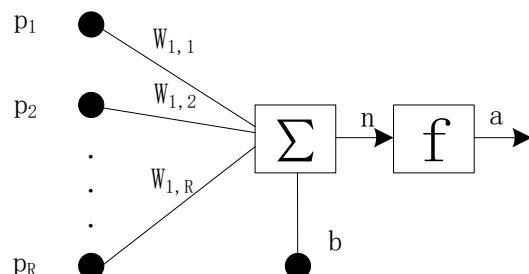


图 2.4 一个神经元的运算过程

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。

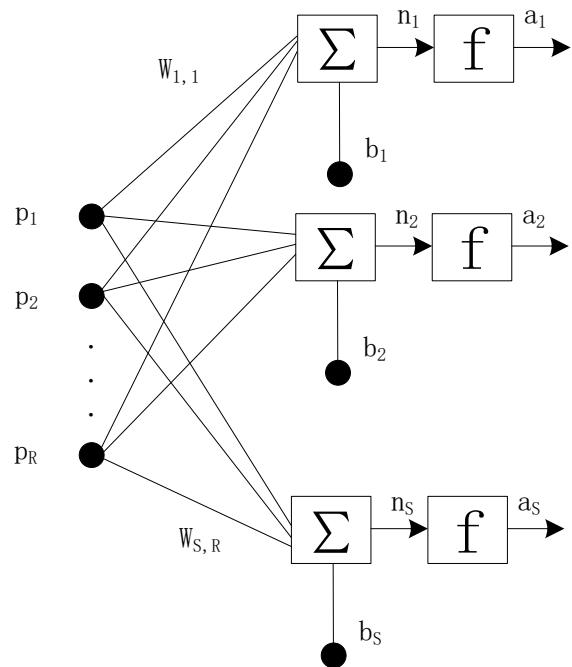


图 2.5 一层网络的运算

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）

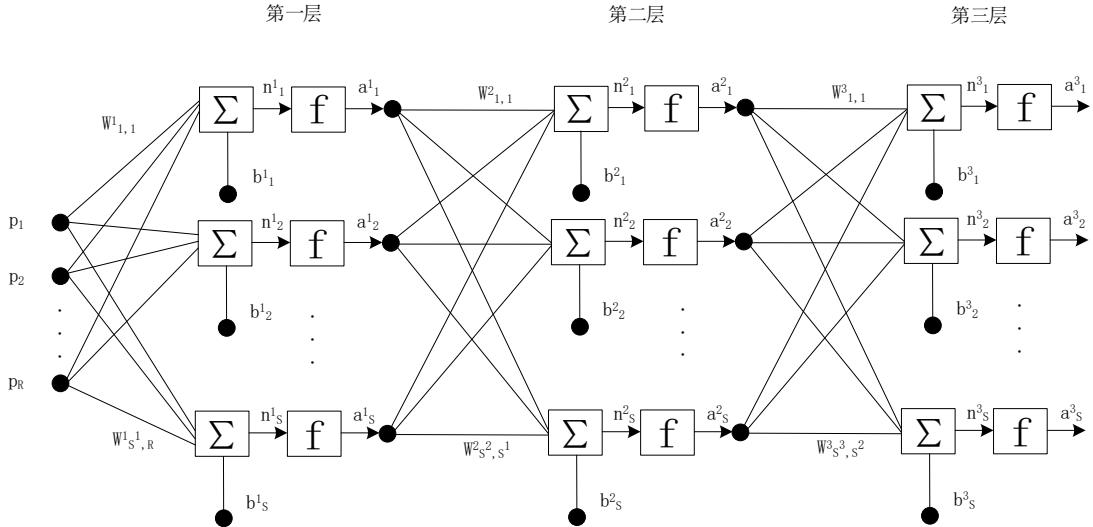


图 2.6 三层神经网络

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 x 。如图 2.7 所示。

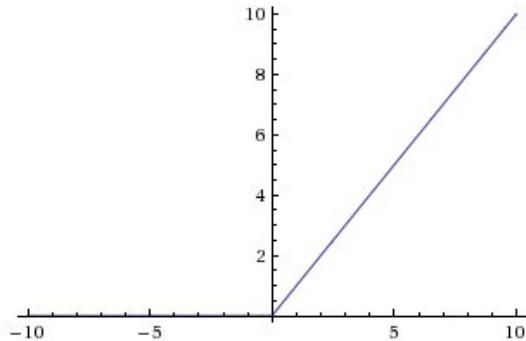


图 2.7 ReLU 激活函数

(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。

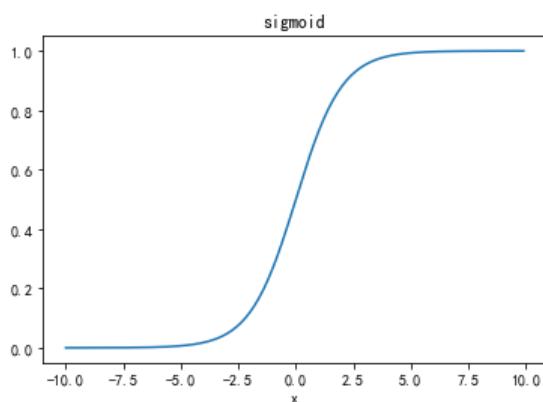


图 2.8 sigmoid 激活函数

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

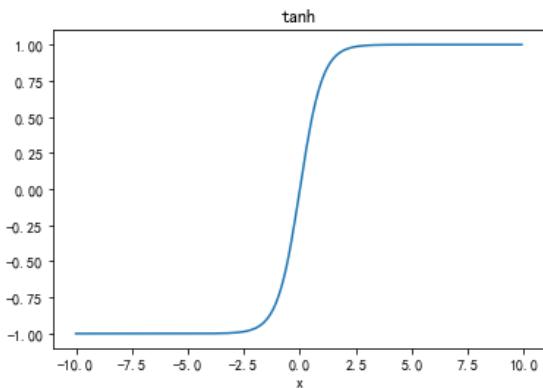


图 2.9 tanh 激活函数

(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。

设有个函数 $y=f(x)$ ，该函数的导数标记为 $f'(x)$ 。按照函数的一阶泰勒展开式，当 ϵ 很小时

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

该公式对求得函数 f 的最小值是有用的，它告诉了我们怎样改变 x 来制造函数 f 的值的一点点减小。如果 $f(x)$ 是正数， $\epsilon f'(x) < 0$ 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 ϵ 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 ϵ ，可以使得 $f(x - \epsilon \operatorname{sign}(f'(x)))$ 小余 $f(x)$ 。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 $x > 0$, $\operatorname{sign}(x) = 1$; 否则 $\operatorname{sign}(x) = -1$.

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。

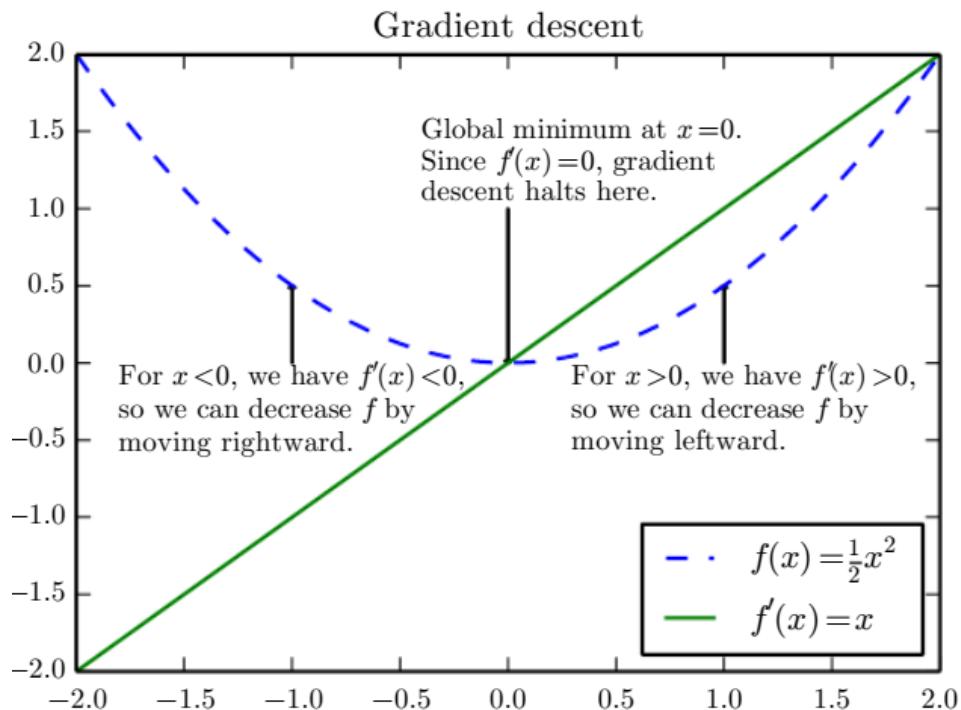


图 2.10 梯度下降

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。

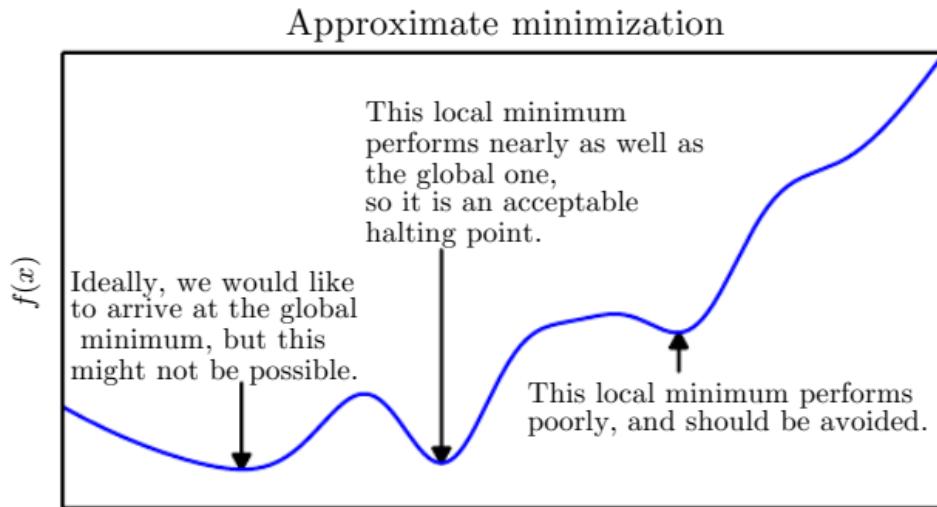


图 2.11 优化过程

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。

$$y = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x}$$

这里 y 是每天吃的早餐的价格。 \mathbf{w} 是三样食物的价格, \mathbf{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）

$$L = \frac{1}{2} \sum_{n \in \text{num}} (t^{(n)} - y^{(n)})^2$$

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

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

第三步，更新权重。用梯度乘上学习率 η 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。

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

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

随机梯度下降算法训练感知机

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 $y^{(n)} = f(w^T x^{(n)})$
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 η 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 η 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 η

1. 初始化参数 θ
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 θ 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 η 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。

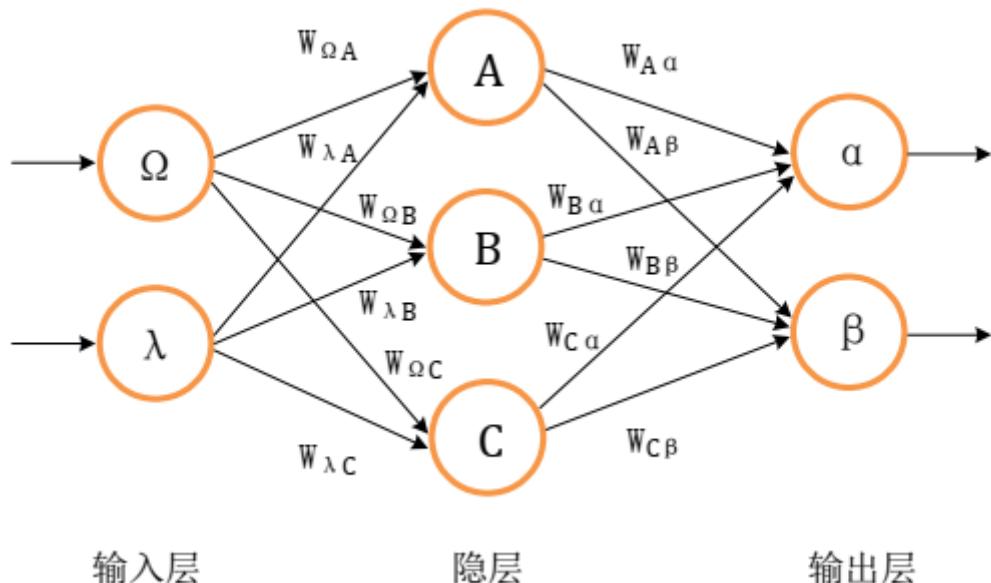


图 2.12：一个多层次神经网络

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

η 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：Tensorflow 和 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 2.2 版。在 windows 上安装 TensorFlow，参见
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`

如果是 GPU 版本则是
pip install --upgrade tensorflow-gpu

或者安装某个固定版本
pip install tensorflow-gpu==2.6

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

我们先介绍一下 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。.

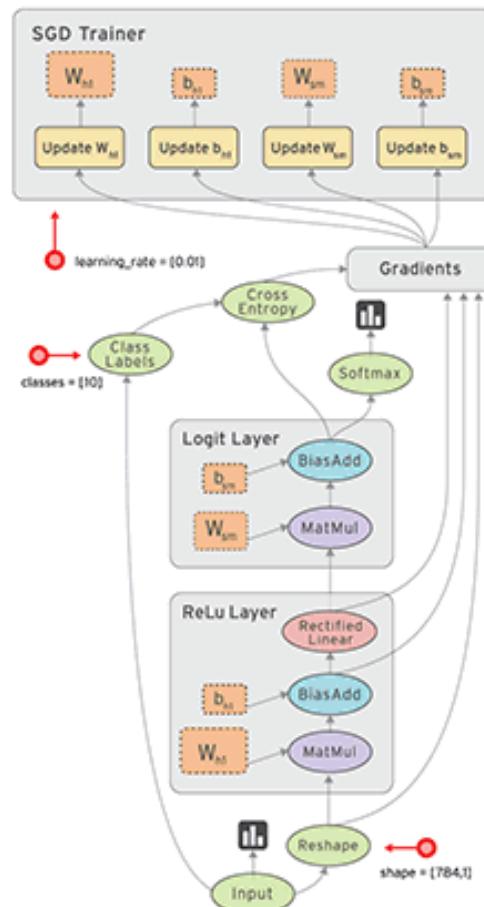


图 3.1 数据流图

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
H1 = Dense(64, activation='relu')
hidden_1 = H1(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

模型是完成特定功能的对象。它可以被训练，然后完成推断。Keras 中的模型是以层 (Layers) 为基础进行构建。主要有两种方法：创建 Sequential 类的对象或者创建 Model 类的对象。将再 3.2 节相信讨论。

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 允许在每个 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。

3.1.7 在 keras 中使用 backend 的基本 tensor 运算

Keras 是一个模型级的库，提供了高层的建立深度学习模型的模块。但它没有处理一下底层的运算，如果 tensor 相乘，求均值等等。Keras 可以使用依靠的 backend 提供的底层运算库来计算。具体的一些运算函数，参见附录 A 的第六节。

第二节：模型的构建与训练

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。

3.2.1 创建模型

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。Keras 中构建模型有三种方法。本文仅介绍最常用的前两种。第三种是创建 Model 类的子类，读者需要时自学。<https://keras.io/api/models/model/>

1. 使用 Sequential 类

Sequential 是一种创建模型的简单方法。它顺序堆叠 “层” 创建一个模型。首先创建一个 Sequential 模型对象，然后用 add 方法添加层。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，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。我们用 Sequential 类创建模型

```
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)。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) 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

2. 使用 Model 类

相比 Sequential, Model 可以创建运算更复杂的模型。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。而新的层将前一个层输出的 tensor 进行计算，输出自己层的 tensor。例如，因此我们就可以用如下的方式创建上面的单层感知机。

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

input = Input(shape=(3,))
output_layer = Dense(1)
output = output_layer(input)
model = Model(inputs=input, outputs=output)
```

这里的 Input 是输入层。Input 保存输入的数据 tensor。创建一个 Dense 的对象，即创建了一个全连接层 output_layer。output = output_layer(input) 是该层以 input 为参数进行运算，产生自己层的输出 tensor, output。最后，model = Model(inputs=input, outputs=output) 搭建好模型。

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

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

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，`Verbose=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)
```

注意，在 `verbose=1` 时，显示一个 epoch 训练的进度条，这时也有个 loss 值和性能度量值（下面以 `mse` 为例）在显示。此时的 `loss` 和 `mse` 是在训练集上的拟合的损失值和性能度量值。当设置了 `fit` 函数中的 `validation_split>0` 的一个值，即设定了校验集，无论 `verbose=1` 或 `2`，一个 epoch 训练完成后，显示 `val_loss`, `val_mse`，这是在校验集上的损失值和 `mse`。如果 `fit` 函数中的 `validation_split=0`，即无校验集。每个 epoch 的结果只显示在训练集上的拟合的 `loss` 和 `mse`。

上面的方法是把训练集直接送入 `fit` 函数。为了解决在训练模型时，数据集太大，一次都读入内存可以导致 `out of memory` 的问题，在 `fit` 函数中的输入数据也可以是 `tf.data.Dataset` 对象。详细见 9.4 节。

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来的数据进行预测

```
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。（源码及数据集见本书 github 网站 keras-bi.py 和 data/ionosphere.txt）

(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。（源码及数据集见本书 github 网站 keras-ml.py 和 data/iris.txt）

(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', '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 LR 三个模型的预测结果（源码及数据集见本书 github 网站 boston.py 和 data/housing.txt）

```
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)
```

```

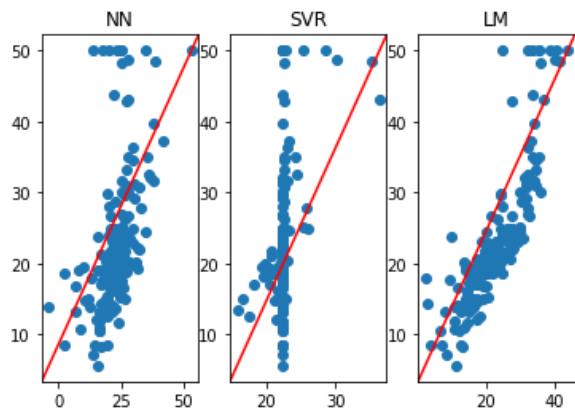
y_pred2=reg.predict(X_test)
err=mean_squared_error(y_test, y_pred2)
print('MSE: %.3f' % (err))

#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()

```

我们也会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，本列不能说明哪个模型更优。



3.3.4 曲线拟合

该例子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。

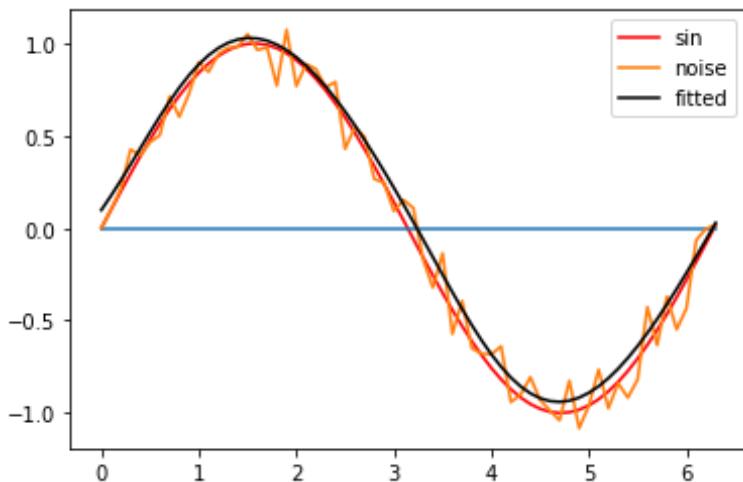


图 3.6 拟合的曲线

第四节：tf.data.Dataset

在训练模型时，数据集很大时，建议读本节。

tf.data.Dataset 是用于创建输入数据 pipeline 的 API。它的内容很丰富，我们不详细讨论。当我们训练模型的数据集很大时，可以使用该类，它不会一次把数据读入内存。而是采用流的方式。见 9.4.1 节

还有一种使用方法是在定制训练过程时，需要逐个 batch 的访问数据集。此时产生 tf.data.Dataset 对象可以用循环语句获得逐个 batch。详见 14.2 节。

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成为了一般规律。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。

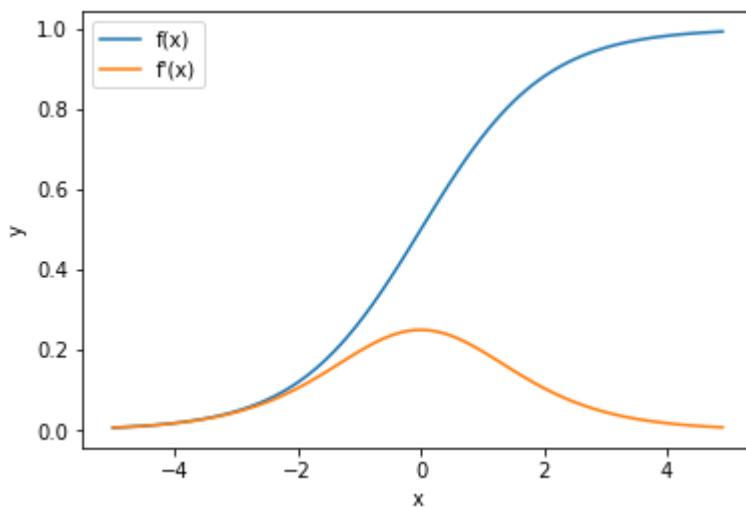


图 4.1 Sigmoid 函数和它的梯度

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，如图 4.2 所示，当输入大于 0 时，它的导数是 1。否则是 0。

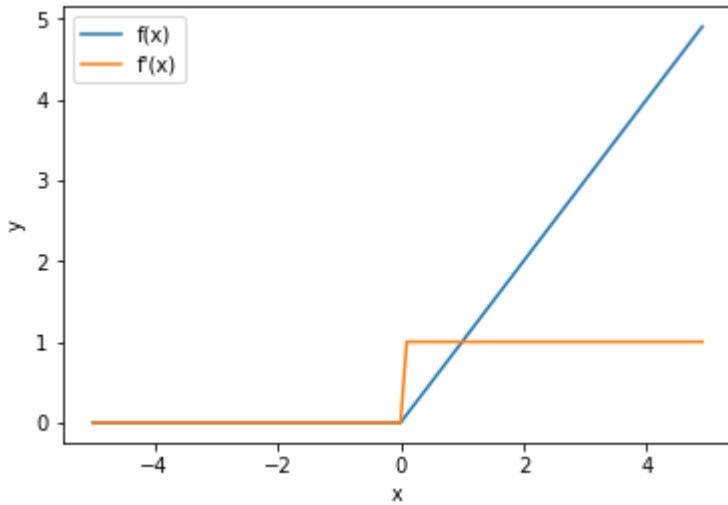


图 4.2 ReLu 激活函数

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。

有两种梯度剪裁的方法：梯度范数标定 (Gradient Norm Scaling) 和梯度梯度值剪裁 (Gradient Value Clipping)。梯度范数标定是说，梯度是损失函数对参数求偏导的一个向量，该向量的 L2 范数（所有值的平方和开方）超过一个阈值，该偏导的向量（梯度）会进行标定，以保证向量的 L2 范数 0 等于阈值。梯度值剪裁是说，如果梯度向量中的一个值超过了阈值，则设定该值为阈值。在 keras 中优化函数中提供了参数来进行梯度剪裁

```
opt = SGD(lr=0.01, momentum=0.9, clipnorm=1.0)
```

是通过参数 clipnorm 进行梯度范式标定，阈值是 1.0

```
opt = SGD(lr=0.01, momentum=0.9, clipvalue=0.5)
```

是通过参数 clipvalue 进行梯度值剪裁，其值 0.5 表示阈值是在 -0.5 ~ 0.5 之间。

- (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。

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

σ 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 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。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。

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。

补充：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。4.4.1 节将介绍损失函数，4.4.2 节将介绍优化算法。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。4.4.3 节介绍正则化。

4.4.1 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。对于回归任务（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) 交叉熵是信息论中的概念，用于度量两个分布的差异，也被用于机器学习中建立损失函数。对于二分类任务（sigmoid 输出层）可以使用 binary cross-entropy（二值交叉熵）。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）， y' 是类别标签（0 或 1）。

对于多分类任务（softmax 输出层），通常采用 categorical 交叉熵来计算损失函数。设有 C 个类别

$$H_{y'}(y) = \sum_k^C y'_k \log(y_k)$$

y'_k 是第 k 个类别上的类别标签， y_k 是在第 k 个类上的预测概率。

对于上面两个交叉熵，因为 $H_{y'}(y)$ 是一个负值，且预测的越不准确负值越大。

因此，在一个有 N 个样本的集合上计算二值交叉熵损失函数时，采用 $\text{loss} = -\frac{1}{N} \sum_i H_{y'_i}(y_i)$

categorical 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

$$\begin{array}{ccc} \text{Pr(Class A)} & \text{Pr(Class B)} & \text{Pr(Class C)} \\ 0.0 & 1.0 & 0.0 \end{array}$$

softmax 回归预测的每个类别的概率 y_i 是

$$\begin{array}{ccc} \text{Pr(Class A)} & \text{Pr(Class B)} & \text{Pr(Class C)} \\ 0.228 & 0.619 & 0.153 \end{array}$$

则计算的交叉熵是

$$H = - (0.0 * \ln(0.228) + 1.0 * \ln(0.619) + 0.0 * \ln(0.153)) = 0.479$$

- (3) 合页损失 (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)

我个人的经验认为 Hinge loss 特别适用匹配的场景。比如一个用户 u 和商品 i 的推荐匹配模型。这样的匹配模型，一般训练集只有用户购买过什么商品，即只有正例。该匹配模型给一个用户和一件商品计算一个 0-1 的匹配评分 $f(u,i)$ 。要训练该模型可以采用 Hinge Loss

$$\text{Loss} = \max(0, 1 - f(u,i) + f(u,i'))$$

该函数是上面 hinge loss 的变体。 i' 可以看作是用户在决策中，放弃购买的上面，即不喜欢的商品。一般来说， i' 的产生可以随机抽样产生，理论上就可以认为是客户不喜欢的。**这是一种常用的处理方法。**

Keras 提供的损失函数的详细介绍见附录第三节。

思考：在机器学习中，我们可以用精确率 (accuracy)、准确率 (precision) 等指标评价模型性能。为什么我们不能用这些指标作为神经网络的损失函数？答案见本章末。

4.4.2 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

动量 (momentum) 是物理学中的概念，是指具有一定质量的运动物体当遭受一个外部反向力时，仍然会沿着当前的运动方向运动一段距离。同样的概念运用于神经网络的训练，当动量被加入到参数更新时，即参数的更新会因为动量而保持更新方向一定的抵抗力，不会变化很剧烈。当目标函数接近了一个局部最优值时，它不会立即停止在这个局部最优，而是沿着原有的更新方向继续前进一段，因此就有可能越过这个比较浅的局部最优点，而最终停在更深的全局最优点。当然动量太大会引起系统震荡，训练的过程不收敛。如图 4.3 所示，我们可以形象的理解为，因为有动量的存在，在局部最优点刹不住车，而滑向了全局最优点。

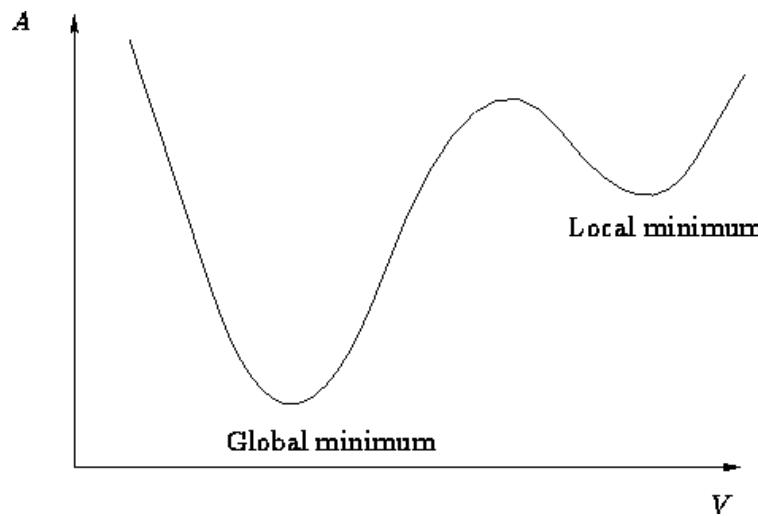


图 4.3 全局最优和局部最优点

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 ϵ , momentum parameter α .

Require: Initial parameter θ , 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 velocity 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 ρ ($0 < \rho < 1$) 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 ϵ , momentum parameter α .

Require: Initial parameter θ , 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: $\theta \leftarrow \theta + v$

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:  $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$ 
        Compute update:  $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$ . (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。 $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$ 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 ϵ 除以 $\delta + \sqrt{\mathbf{r}}$ 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:  $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g} \odot \mathbf{g}$ 
        Compute parameter update:  $\Delta\theta = -\frac{\epsilon}{\sqrt{\delta + \mathbf{r}}} \odot \mathbf{g}$ . ( $\frac{1}{\sqrt{\delta + \mathbf{r}}}$  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 $\Delta\theta = -\epsilon \frac{s}{\sqrt{r} + \delta}$

有评论，Adam 适用于数据量大和参数多的模型。

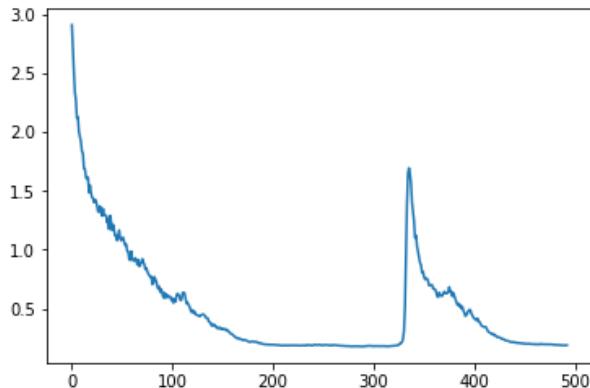
3. 怎样选择合适的优化算法

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。

4.4.3 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。也包括训练模型时的一些技巧。

1. 损失函数中加入正则项

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。

补充：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（隐层的输出）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，我们可以考虑将该层的权重和激活函数正则项加入损失函数

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

其中 l2(0.01) 和 l1(0.01) 中的值 0.01 是乘在正则化项前的系数。

2. 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）。

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 $W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$ 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。

3. 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 4.4 所示。

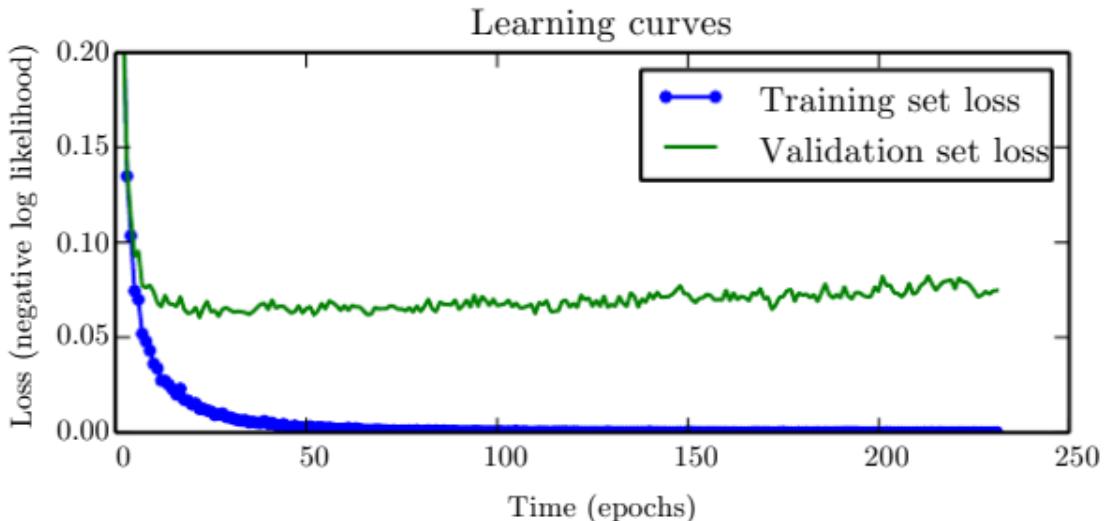


图 4.4 校验集误差的变化

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

1. 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）
2. 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)

3. 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，在 fit 函数的参数中加入该类。

```
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))
```

4. 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

5. 数据标定

神经网络中，输入的数据值大，模型学习的权重大，会导致模型不稳定，敏感，更高的泛化误差。目标值变化范围大可以导致大的梯度值，甚至会导致梯度爆炸。图 4.5 是一个解释。它展示了在一个凸函数上进行优化，随机梯度下降算法进行参数调整的过程。中心是全局最优点。

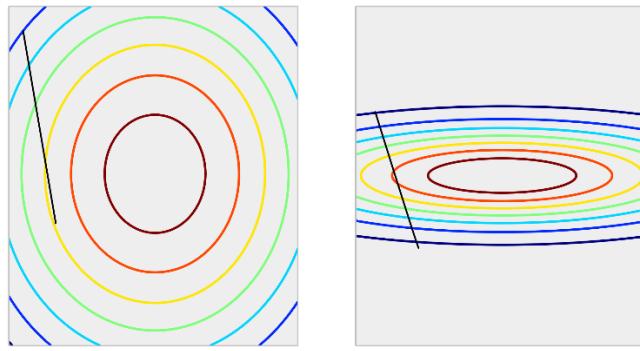


图 4.5 标定对梯度下降算法的影响

左图是一个数据（特征值）在二维空间的分布被调整以后的分布，右图是未调整的数据分布。直线表示在随机下降的调整模型参数时，参数的变化。在标定后的数据分布上，模型参数的一步调整，使得模型更接近全局最优点。而在未标定的数据分布上，一步调整使得模型跑到了远离全局最优的另外一边，如此优化过程出现震荡。（[学习率自适应的优化算法，如 adagrad、adam 是否可以避免该问题？](#)）

6. 批量规范化 (Batch Normalization)

神经网络的一个隐层的输出，是下一层的输入。这个输入对应一个分布，这个分布受到前一层的参数变化的影响，也受到了 minibatch SGD 时，每个批次样本差异的影响。这个影响称为 **internal covariate shift**。观察到的现象就是，训练模型时，网络内部隐层的参数改变，使得下一层的输入的分布的均值和标准差在变化。这导致神经网络的训练困难。（Deep Learning 一书从模型参数的 coordinate update cross layers 的角度分析了问题，指出优化函数的学习率不好选择）

Batch Normalization 是用于解决上述问题导致训练深度神经网络困难的一个技巧：神经网络中的每一隐层的输入（或上一层的输出）被规范化到固定的均值（接近 0）和标准差（接近 1）。

当采用 minibatch 梯度下降训练模型时，一个隐层的输入是一个 $m \times n$ 的矩阵 H 。 m 是 minibatch 的大小， n 是上个隐层神经元的个数。设 μ 是一个向量，对应上个隐层每个神经元在一个 minibatch 输出的均值； σ 是一个向量对应上个隐层每个神经元在一个 minibatch 输出的标准差。对 H 矩阵的每一行，逐元素做规范化运算

$$H'_{ij} = \frac{H_{ij} - \mu_j}{\sigma_j}$$

规范化后的 H' 作为当前隐层的输入。使用了 Batch Normalization 可以允许优化函数有更高的学习率来加快训练过程，而不会梯度爆炸。Batch Normalization 也有正则化的效果，提高了深度模型的泛化能力。

Keras 中提供了一个 `BatchNormalization` layer 来实施对隐层输入的批量规范化。参考附录 A。keras 中的实现如下：

对前一个层在每一个 batch 的输出进行规范化，使得一个激活函数的输出的均值接近 0，标准差接近 1。批量规范化在训练阶段和推断阶段的工作是不一样的。

训练阶段：即使用 `fit()` 函数，或者调用该层或模型时，设置了参数 `training=True`。规范化的公式是 `(batch - mean(batch)) / (var(batch) + epsilon) * gamma + beta`

推断阶段：即使用 `evaluate` 或 `predict` 函数，或者调用该层时，设置了参数 `training=False`（默认值）。该层将使用均值和标准差在训练阶段所有 batch 上计算的移动平均来规范化该层的输出。

```
(batch - self.moving_mean) / (self.moving_var + epsilon) * gamma + beta.
```

其中的 `self.moving_mean` 和 `self.moving_var` 不是学习参数，而是在训练阶段计算出来并保存的均值和标准差移动平均。

```
moving_mean = moving_mean * momentum + mean(batch) * (1 - momentum)
```

```
moving_var = moving_var * momentum + var(batch) * (1 - momentum)
```

通常的使用方法是在一个层后面加一个 `BatchNormalization` 层，例如

```
model.add(Dense(256, activation='relu'))
```

```
model.add(BatchNormalization())
```

第五节：实例：手写数字识别

MNIST 是一个手写数字的数据集。它包含的图片如图 4.5：

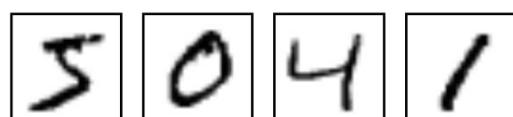


图 4.5 mnist 数据集中的图片示例

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 4.6。

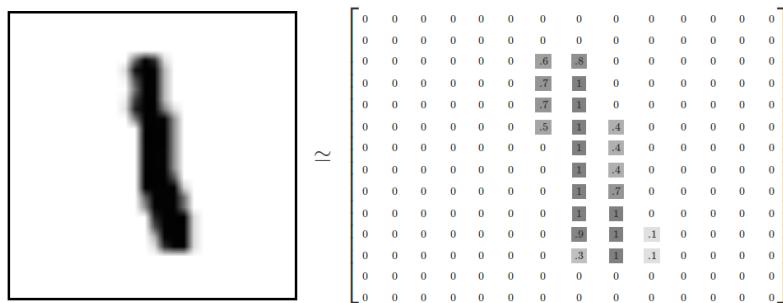


图 4.6 图片的矩阵描述

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 $28 \times 28 = 784$ 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 4.7 所示：（此处假设，输入向量的维度是 3，输出的类别个数也是 3）

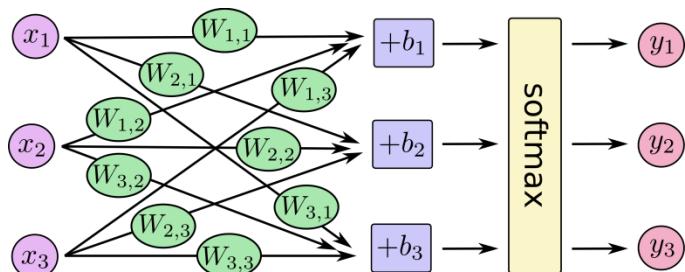


图 4.7 softmax regression 模型

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=False)
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概率值，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。

需要强调一下 `from_logits` 的含义。在 `tf.keras` 中很多函数中有个参数 `from_logits`，它的含义是指处理的数据是一个概率值吗（0-1 之间）？默认 `from_logits=False`，表示处理的数据是概率值，上面的例子中模型的最后一个层是 softmax 层，因此输出的是概率分布。此处，特别设置 `from_logits=False` 只是为了说明该参数的含义，不写也行。

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

看下面这一段代码可以理解 `from_logit` 的含义。下面的两个 loss 值是一样的。

```
import tensorflow as tf
from tensorflow.keras import backend as K

y_true = [0, 1, 0, 0]
y_pred = [-18.6, 0.51, 2.94, -12.8]
y_soft = K.sigmoid(y_pred)
bce = tf.keras.losses.BinaryCrossentropy(from_logits=True)
loss = bce(y_true, y_pred).numpy()
print(loss)

bce2 = tf.keras.losses.BinaryCrossentropy()
loss = bce2(y_true, y_soft).numpy()
print(loss)
```

因此，对于这个 softmax regression 模型，如果最后一个层不要，即只是一个 Dense，此时把损失函数的 `from_logits` 设为 True。这样的操作也是正确的。即此时的损失函数，会自己把数据转换成概率分布后再计算交叉熵损失值。见下面的代码。

我们采用 adam 优化算法训练模型，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, verbose=1)
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)  
_, acc=model.evaluate(x_test, y_test, verbose=2)  
print("acc:%s"%(acc))
```

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，以及进行正则化，进一步提高模型的性能

“思考”的答案

答：当采用 SGD 来训练模型时，需要为损失函数求偏导来获得梯度。而准确率、精确率不能计算梯度。因此，方法就是用可导函数对不可导函数逼近。精确率 accuracy 的光滑近似函数再经过优化就是二值交叉熵损失函数。详见附录 B。

第六节：解决类不平衡问题

4.6.1 关于类不平衡问题

前面讲述的分类方法都假设工作在一个平衡数据集上，例如二分类的训练集中正例和反例的数量一致，或不会相差太大。但在实践中往往遇到的数据集没有这么理想，很多时候都是严重类不平衡数据集，即训练集中每个类别的数据记录数有较大的差别。使用前面讲述的算法在严重类不平衡数据集上建立模型往往会有问题。例如，医学领域检测病人是否患某种疾病。此时的数据集，除了很小的部分，往往大部分都是未患病的数据记录。常规的分类方法和评价方法可以在数据多的类别上表现很好，在数据稀少的类别上表现很差。这样的模型仍会得到很高的精确度。但实际应用中，在数据稀少类别上的准确预测才是最重要的任务。例如，疾病检测中，更在乎的是可以准确检测出患病的病人；在欺诈检测的任务中。有欺诈行为的交易记录也往往比正常交易记录少很多。相关人员更关心可以把占很少数的欺诈交易行为识别出来。

4.6.2 keras 解决类不平衡问题的方法

https://www.tensorflow.org/tutorials/structured_data/imbalanced_data

在使用 keras 时，解决类不平衡问题有两种方法：Class weight 和 over sampling。下面讲使用 kaggle 的一个信用卡欺诈数据集作为实例。<https://www.kaggle.com/mlg-ulb/creditcardfraud>。该数据集包含 284,807 条交易记录，其中只有 492 条是欺诈交易（它们被看作是正例）。完整源码参看文件 imbalanced-fraud.py

1. 考察数据集

```
raw_df = pd.read_csv('crditcard.csv')
raw_df.head()
neg, pos = np.bincount(raw_df['Class'])
total = neg + pos
print('Examples:\n    Total: {}\n    Positive: {} ({:.2f}% of total)\n'.format(
    total, pos, 100 * pos / total))
```

读入数据，并考察数据集中的列，显示有 31 列。Class 是目标列

Time V1 V2 V3 ... V27 V28 Amount Class

再显示数据样本数量，及正例和反例的比例，是

Examples:

```
Total: 284807
Positive: 492 (0.17% of total)
```

下面的代码进行预处理

```
cleaned_df = raw_df.copy()
cleaned_df.pop('Time')
eps = 0.001
cleaned_df['Log_Amount'] = np.log(cleaned_df.pop('Amount')+eps)
```

数据集中的 Time 列描述的是时间信息，因此排除出数据集。Amount 值的变化范围有些大，因此用对数处理。然后按照划分数据集为训练集、测试集和校验集。

```
train_df, test_df = train_test_split(cleaned_df, test_size=0.2)
train_df, val_df = train_test_split(train_df, test_size=0.2)

# Form np arrays of labels and features.
train_labels = np.array(train_df.pop('Class'))
bool_train_labels = train_labels != 0
val_labels = np.array(val_df.pop('Class'))
test_labels = np.array(test_df.pop('Class'))

train_features = np.array(train_df)
val_features = np.array(val_df)
test_features = np.array(test_df)

scaler = StandardScaler()
train_features = scaler.fit_transform(train_features)
```

```

val_features = scaler.transform(val_features)
test_features = scaler.transform(test_features)

train_features = np.clip(train_features, -5, 5)
val_features = np.clip(val_features, -5, 5)
test_features = np.clip(test_features, -5, 5)

```

StandardScaler()将数据集按照数据分布均值为 0，标准差为 1 进行规范化。这里，训练集、测试集和校验集分开进行规范化处理，是为了保证训练集不会了解到测试集和校验集中的信息。接着，对数据集中偏大的数据（超过-5 和 5）进行裁剪，以便不让离群点影响模型。

紧接着把训练集中正例（交易欺诈样本）和反例的样本分离

```

pos_df = pd.DataFrame(train_features[ bool_train_labels],
columns=train_df.columns)
neg_df = pd.DataFrame(train_features[~bool_train_labels],
columns=train_df.columns)
sns.jointplot(pos_df['V5'], pos_df['V6'],
kind='hex', xlim=(-5,5), ylim=(-5,5))
plt.suptitle("Positive distribution")

sns.jointplot(neg_df['V5'], neg_df['V6'],
kind='hex', xlim=(-5,5), ylim=(-5,5))
_ = plt.suptitle("Negative distribution")

```

然后用可视化的方式，展示预处理后的数据分布，其中只展示 V5 和 V6 两列。如图

4.8 所示。可以看到数据大部分都是分布再-2 到+2 区间，其中正例中有些极端值。

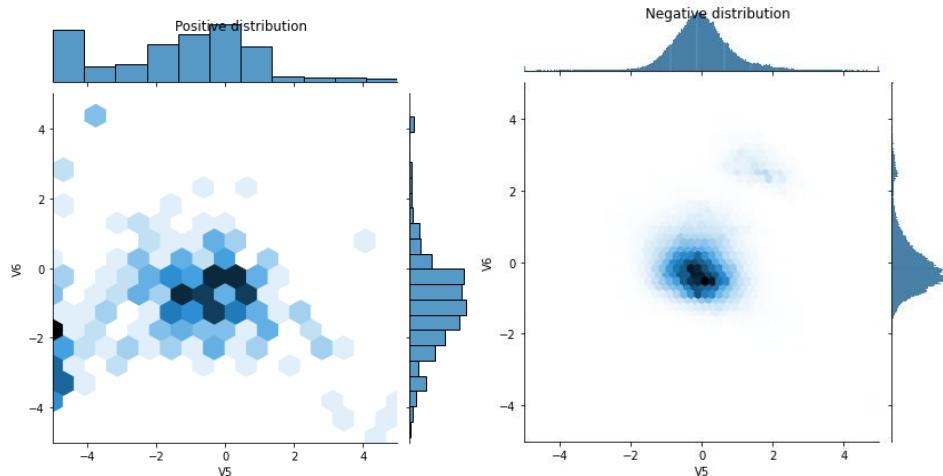


图 4.8 考察数据分布

下面是建立模型，设定度量指标。其中编译模型时设置的参数 batch_size 很大，这是为了保证，每个批次中至少有正例，因为正例太少了。下面的模型没有处理类不平衡问题。只是为了比对设定的基线模型。后面将改进该模型。下面的程序还有一点，设置了训练模型是的 early stopping。它是创建了一个 tf.keras.callbacks.EarlyStopping

的对象。monitor='val_loss'表示监控的指标是校验集的损失函数值；patience=10·表示连续 10 轮；mode='min'表示，如果指标没有下降。restore_best_weights=True 表示从最好的一趟的模型性能恢复模型参数。该函数更多的信息见 api 帮助文档。

```

METRICS = [
    keras.metrics.TruePositives(name='tp'),
    keras.metrics.FalsePositives(name='fp'),
    keras.metrics.TrueNegatives(name='tn'),
    keras.metrics.FalseNegatives(name='fn'),
    keras.metrics.BinaryAccuracy(name='accuracy')
]

def make_model(metrics=METRICS, output_bias=None):
    if output_bias is not None:
        output_bias = tf.keras.initializers.Constant(output_bias)
    model = keras.Sequential([
        keras.layers.Dense(
            16, activation='relu',
            input_shape=(train_features.shape[-1],)),
        keras.layers.Dropout(0.5),
        keras.layers.Dense(1, activation='sigmoid',
                          bias_initializer=output_bias),
    ])
    model.compile(
        optimizer=keras.optimizers.Adam(lr=1e-3),
        loss=keras.losses.BinaryCrossentropy(),
        metrics=metrics)

    return model

EPOCHS = 100
BATCH_SIZE = 2048

early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    verbose=1,
    patience=10,
    mode='min',
    restore_best_weights=True)

```

上面函数 make_model 是用于产生模型，它有个参数 output_bias。现在进行类不平衡处理的一个步骤，设置输出层的初始偏置，以反映类别上的不平衡。**其原理是，设置的输出层的偏置 b_0 经过 sigmoid 激活函数 $1/(1+e^{-b_0})$ 以后可以反映正例（类别标签为 1）的分布，即**

$$p_0 = pos / (pos + neg) = 1 / (1 + e^{-b_0})$$

$$b_0 = \log_e \left(\frac{1}{p_0} - 1 \right)$$

因此，初始偏差按照 $b_0 = \log_e(pos/neg)$ 来设置。这和朴素贝叶斯中类别的先验概率是一样的。

```
initial_bias = np.log([pos/neg])
model = make_model(output_bias=initial_bias)
```

多次训练模型时还有个技巧，因为多次训练比较模型时，每次模型的初始参数都不一致。则可比性要差。因此，应该把模型的初始参数保存到文件，然后每次都是从该文件读取初始参数。前面的代码中，我们都还没有训练模型（调用 fit 函数），但模型已经有了初始化的参数，因此把它们保存到文件

```
initial_weights = os.path.join(tempfile.mkdtemp(), 'initial_weights')
model.save_weights(initial_weights)
```

下面的代码，比较有偏置初始化和无偏置初始化时，两个模型的性能。当前模型初始化参数已经保存到了临时文件 initial_weights。它包含了前面初始化了的输出层的偏置参数。因此下面的代码将模型初始参数装进来时 model.load_weights(initial_weights)，包含了初始化了的的输出层偏置。因此，为了考察无初始化的情况，model.layers[-1].bias.assign([0.0])将，初始偏置都设为 0。

```
model = make_model()
model.load_weights(initial_weights)
model.layers[-1].bias.assign([0.0])
zero_bias_history = model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=20,
    validation_data=(val_features, val_labels),
    verbose=0)

model = make_model()
model.load_weights(initial_weights)
careful_bias_history = model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=20,
    validation_data=(val_features, val_labels),
    verbose=0)

def plot_loss(history, label, n):
    # Use a log scale on y-axis to show the wide range of values.
    plt.semilogy(history.epoch, history.history['loss'],
                 color=colors[n], label='Train ' + label)
    plt.semilogy(history.epoch, history.history['val_loss'],
                 color=colors[n], label='Val ' + label,
                 linestyle="--")
    plt.xlabel('Epoch')
```

```

plt.ylabel('Loss')

plot_loss(zero_bias_history, "Zero Bias", 0)
plot_loss(careful_bias_history, "Careful Bias", 1)

```

两个模型训练时，再训练集和校验集上的损失值显示再图 4.9。蓝色线是没有进行偏置校正，黄色线是进行了偏置校正。实线是测试集上的损失值，虚线是在校验集上的损失值。可以看到，经过偏置校正，模型在训练集和测试集上的损失值都小于没经过偏置校正的模型。（注：校验误差低于训练误差是因为模型中采用了 dropout）

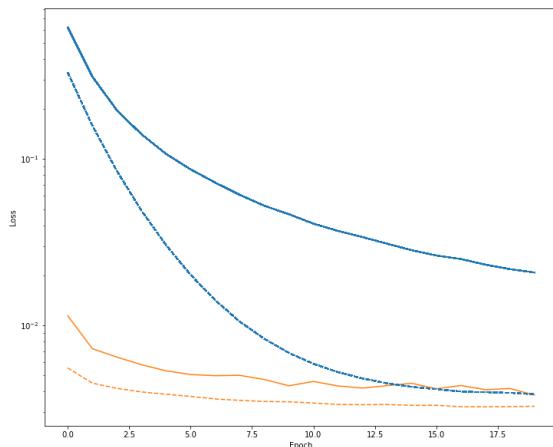


图 4.9 考察偏置校正后的性能

我们用 balanced accuracy 作为模型评价指标。下面的函数 bacc_acc 从评估结果计算 balanced accuracy 和 accuracy。然后使用没有任何处理不平衡数据集的情况下训练模型，评估模型。

```

def bacc_acc(metrics_names, baseline_results):
    m = dict()
    for name, value in zip(metrics_names, baseline_results):
        m[name]=value
    pacc = m['tp']/(m['tp']+m['fn'])
    nacc = m['tn']/(m['tn']+m['fp'])
    acc = (m['tp']+m['tn'])/(m['tp']+m['fn']+m['tn']+m['fp'])
    bacc = (pacc+nacc)/2
    return bacc, acc

model = make_model()
model.load_weights(initial_weights)
# model.layers[-1].bias.assign([0.0]) # original model without bias
baseline_history = model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    callbacks=[early_stopping],

```

```

validation_data=(val_features, val_labels))

baseline_results = model.evaluate(test_features, test_labels,
                                 batch_size=BATCH_SIZE, verbose=0)
bacc, acc = bacc_acc(model.metrics_names, baseline_results)
print('imbalanced: bacc=%0.4f, acc=%0.4f'%(bacc, acc))

```

显示的结果是： imbalanced: bacc=0.8760, acc=0.9994

下面使用 class weight 来对付类不平衡问题。思想是给少数类分配更高的权重值。以便训练模型时，让模型更加关注这些权重更高的样本。给每个类别分配类别权重时，应该是每个类的样本在这个权重上的和是一样的。因此，类别权重按照 $1/\text{neg}$ 和 $1/\text{pos}$ 来分配（ neg 和 pos 是反例和正例样本数）。然后，分别乘上 $\text{total}/2$ 进行标定（2 表示类别的个数）。

```

weight_for_0 = (1 / neg) * (total / 2.0)
weight_for_1 = (1 / pos) * (total / 2.0)

class_weight = {0: weight_for_0, 1: weight_for_1}

print('Weight for class 0: {:.2f}'.format(weight_for_0))
print('Weight for class 1: {:.2f}'.format(weight_for_1))

```

下面是设置 class weight 时，进行的模型训练

```

weighted_model = make_model()
weighted_model.load_weights(initial_weights)

weighted_history = weighted_model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    callbacks=[early_stopping],
    validation_data=(val_features, val_labels),
    # The class weights go here
    class_weight=class_weight)

weighted_results = weighted_model.evaluate(test_features, test_labels,
                                           batch_size=BATCH_SIZE, verbose=0)

bacc, acc = bacc_acc(model.metrics_names, weighted_results)
print('class weight: bacc=%0.4f, acc=%0.4f'%(bacc, acc))

```

显示模型性能如下：（注：这是没有使用 early_stopping 得到的性能，本例子中使用了 early_stopping 性能会略差）

class weight: bacc=0.9376, acc=0.9861

可以看到，确实使用 class weight 帮助提高了模型在 balanced accuracy 上的性能。下面使用少数类上的过采用来解决类不平衡问题。首先是将正例和反例样本数据分开。然后使用 numpy 的 random.choice 进行过采样，获得采样后的下标集合。并过采用后获得的正例样本数据

```
res_pos_features = pos_features[choices]
res_pos_labels = pos_labels[choices]
```

让后再将正例样本和反例样本拼接，再打乱顺序后，得到少数类上过采样的数据集。

```
pos_features = train_features[bool_train_labels]
neg_features = train_features[~bool_train_labels]

pos_labels = train_labels[bool_train_labels]
neg_labels = train_labels[~bool_train_labels]
ids = np.arange(len(pos_features))
choices = np.random.choice(ids, len(neg_features))

res_pos_features = pos_features[choices]
res_pos_labels = pos_labels[choices]

resampled_features = np.concatenate([res_pos_features, neg_features], axis=0)
resampled_labels = np.concatenate([res_pos_labels, neg_labels], axis=0)

order = np.arange(len(resampled_labels))
np.random.shuffle(order)
resampled_features = resampled_features[order]
resampled_labels = resampled_labels[order]
```

下面的程序是使用过采样的数据集训练模型。此时，就没有设置 class weight。另外，因为此时类别是平衡的，也没有再设置输出层的偏置了。

```
resampled_model = make_model()
resampled_model.load_weights(initial_weights)

output_layer = resampled_model.layers[-1]
output_layer.bias.assign([0]) # 数据集是平衡的，因此设置最后一层的偏置无偏差

resampled_history = resampled_model.fit(
    resampled_features,
    resampled_labels,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    callbacks=[early_stopping],
    validation_data=(val_features, val_labels))

resampled_results = resampled_model.evaluate(test_features, test_labels,
                                             batch_size=BATCH_SIZE, verbose=0)
```

```
bacc, acc = bacc_acc(resampled_model.metrics_names, resampled_results)
print('over sampling: bacc=%0.4f, acc=%0.4f%(bacc, acc))
```

程序性能如下： over sampling: bacc=0.9330, acc=0.9940

和设置 class weight 的方法差不多。

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。

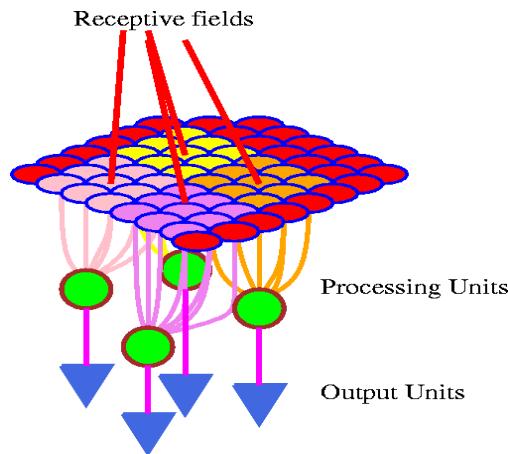


图 5.1：视觉系统上的感受野

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:

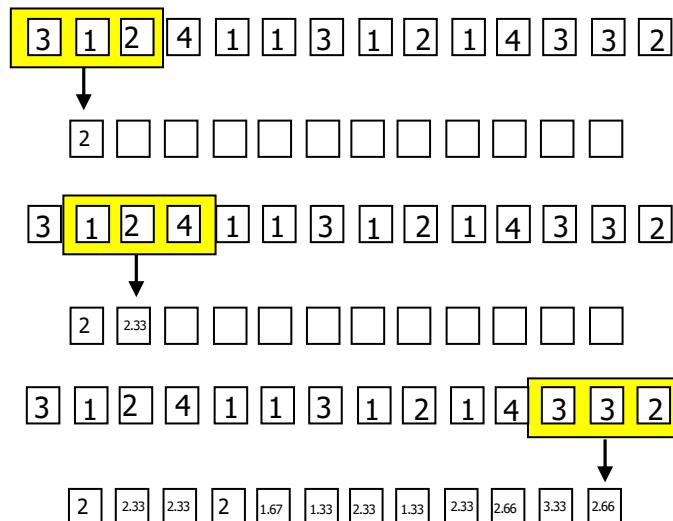
$$y_t = \sum_{k=1}^m f_k \cdot x_{t-k-1}$$

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:

0 0 3 1 2 4 1 1 3 1 2 1 4 3 3 2 0 0

1 1.33 2 2.33 2.33 2 1.67 1.33 2.33 1.33 2.33 2.66 3.33 2.66 1.67 0.6

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

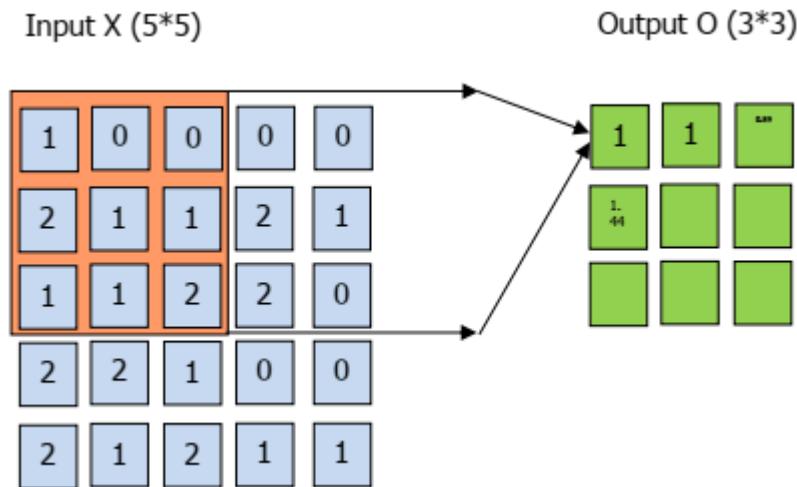


图 5.2 二维卷积

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）。下面一节可以看到，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。

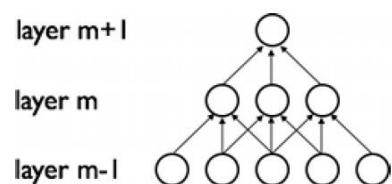


图 5.3：一个 CNN 示例

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。

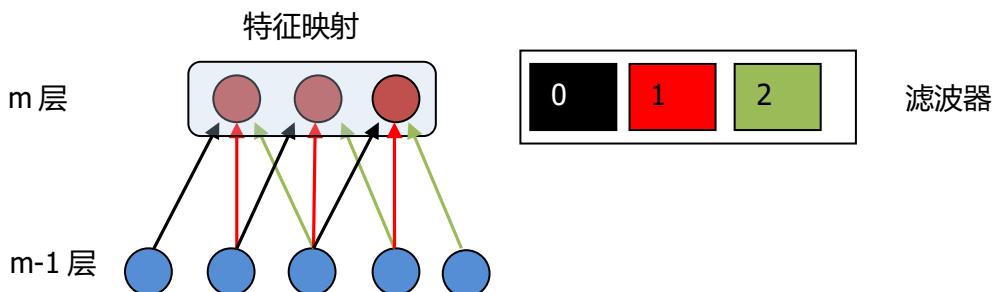


图 5.4：特征映射

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。

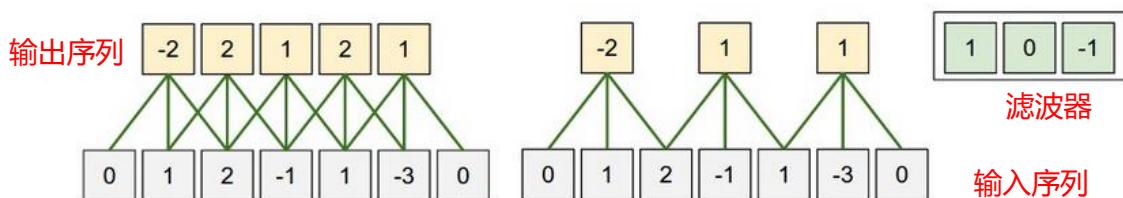


图 5.5 一维卷积

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)。

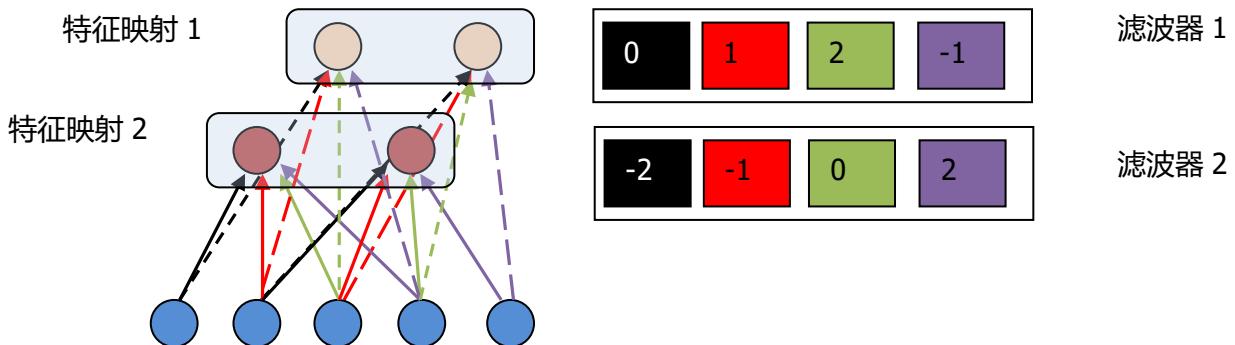


图 5.6 多个滤波器

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 $W^{(l)} \in R^m$ 为 m 维的滤波器, $a_{(i+m-1):i}^{(l)} = [a_{i+m-1}^{(l)}, \dots, a_i^{(l)}]^T$; 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, $stride=1$)

在实现卷积操作时, 滤波器 $W^{(l)}$, 也就是权重矩阵。一个 3×3 的滤波器的结构如下:

$$\begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} \end{pmatrix}$$

卷积操作也就是矩阵运算

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

反向传播时, 怎么保证改权重矩阵, 为 0 的部分不修改, 而一些值, 如 $w_{0,0}$ 不同位置多次出现, 而怎么保持它们一致?

一篇网文 “A guide to convolution arithmetic for deep learning” 有更理论的介绍。

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。

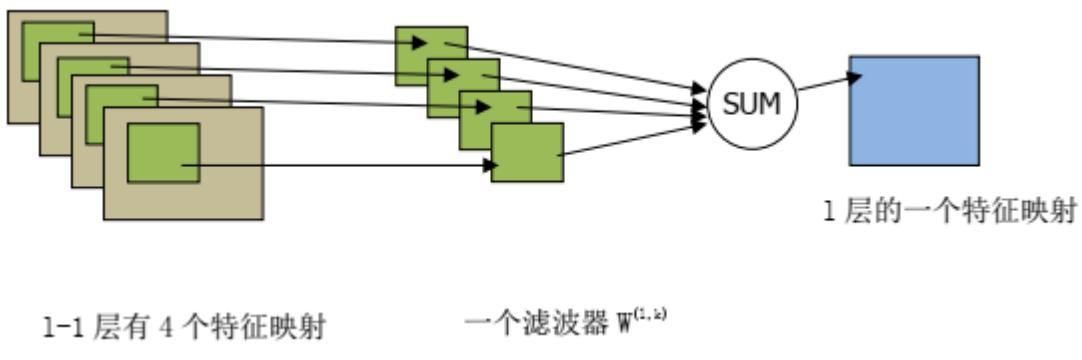


图 5.7 卷积操作示例

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 tensor（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。

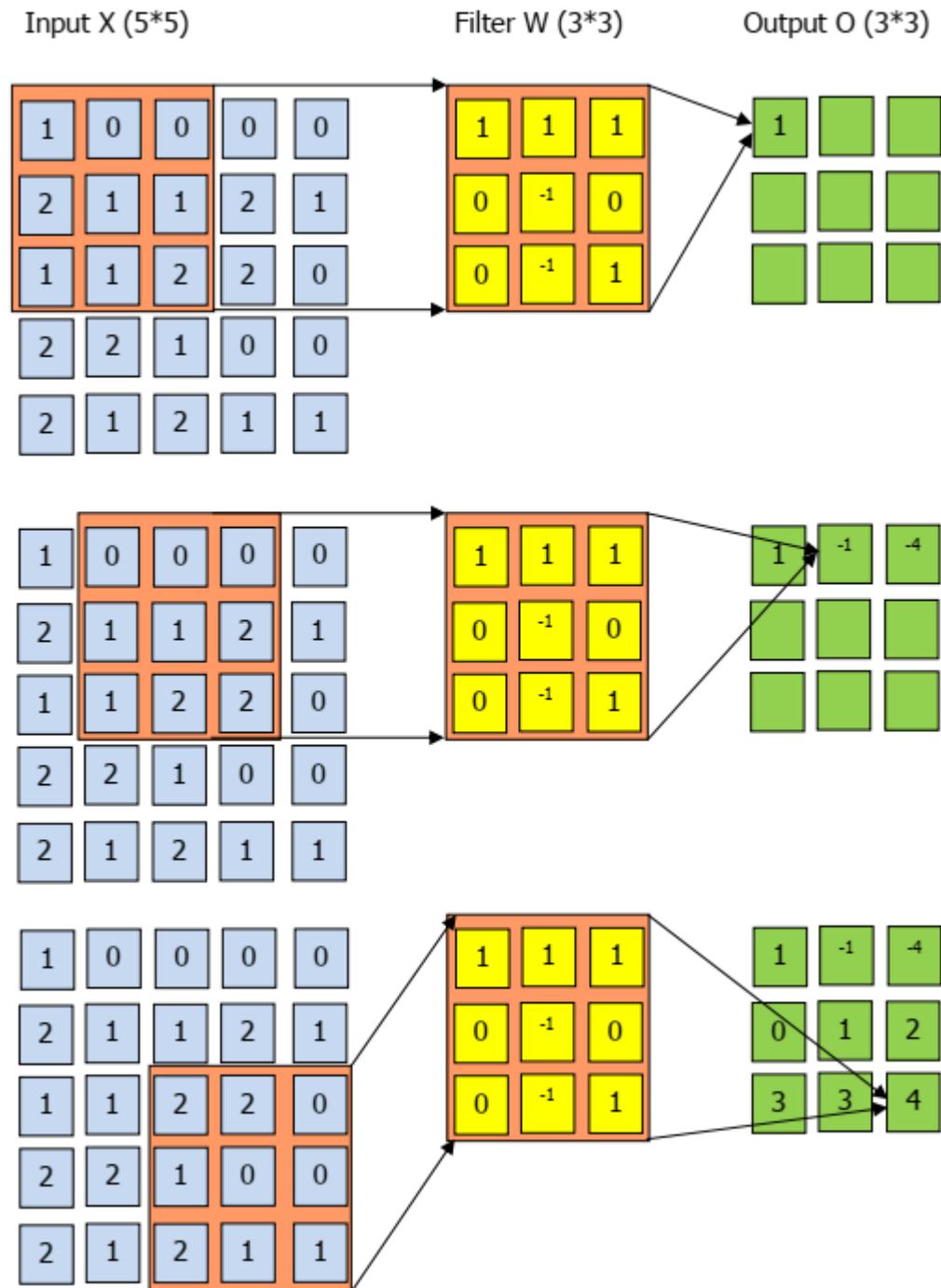


图 5.8 卷积操作示例 2

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

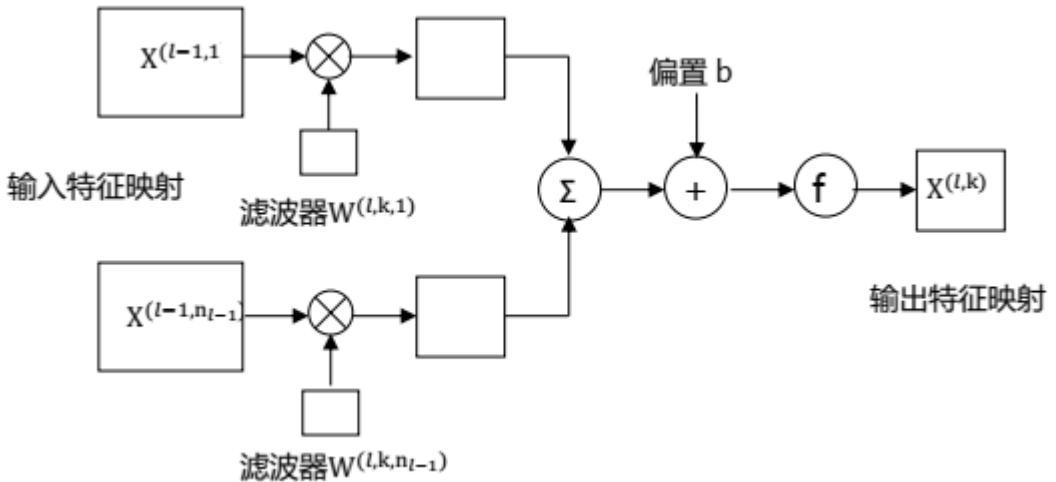


图 5.10 二维卷积层的从输入到输出的映射关系（第 k 个滤波器的例子）

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 subsampling(或下采用 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, 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。

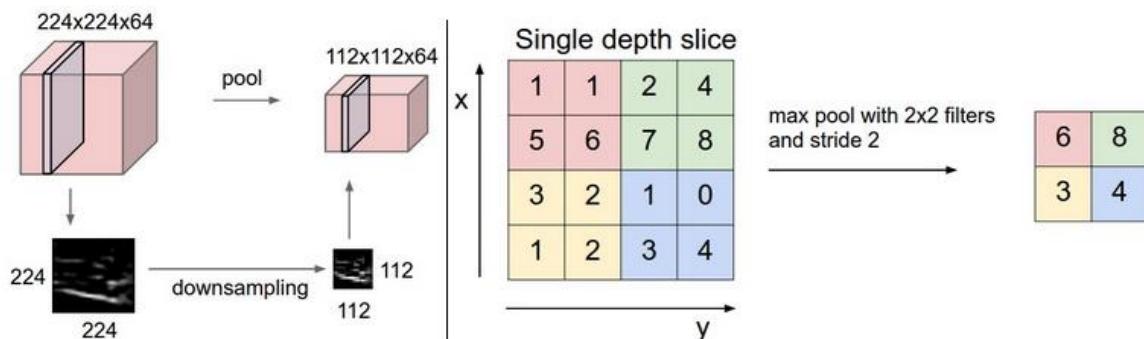


图 5.11 子采样示意图

先看左边，输入是一个 $224*224*64$ 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 $2*2$ （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 $4*4$ ；滤波器的维度是 $2*2$ ，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。

可以这样定性的来理解卷积神经网络。一个滤波器就是一个特征提取器。当以一定的步长在输入的 tensor 上移动滤波器窗口，就相当于在输入上寻找特征。当输入有局部信息和滤波器窗口匹配，即此时滤波器在该区域可以计算一个大的值（理解为视觉系统感受野接受到刺激有大的输出），该滤波器即提取到了一个特征。我们设置很多的滤波器就是在提取各种不同的特征。以一张椅子图片为例，如果滤波器提取出了“椅子腿”的特征，“靠背”的特征，则可以判断这张图片描述的是椅子。

第三节：Keras 卷积层函数

Keras 提供了函数来构建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。

5.3.1 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: 濾波器窗口大小。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。
- (3) strides: 濾波的步长
- (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)

怎么理解 Input shape 中的 channels? 在一维卷积中，输入的数据是序列数据，看作是一个 step*features 的矩阵。一个滤波器卷积结果是一个 new_step*1 的向量，n 个滤波器的结果拼接，得到 new_step*n 的矩阵。因此 n 即特征映射数目 (channel)，也是滤波器的数目。当前卷积操作结果作为下一个卷积操作的输入时，输入的 tensor 就是 (batch, steps, channels) .

一维卷积的例子如下。

1D CONVOLUTIONAL - EXAMPLE

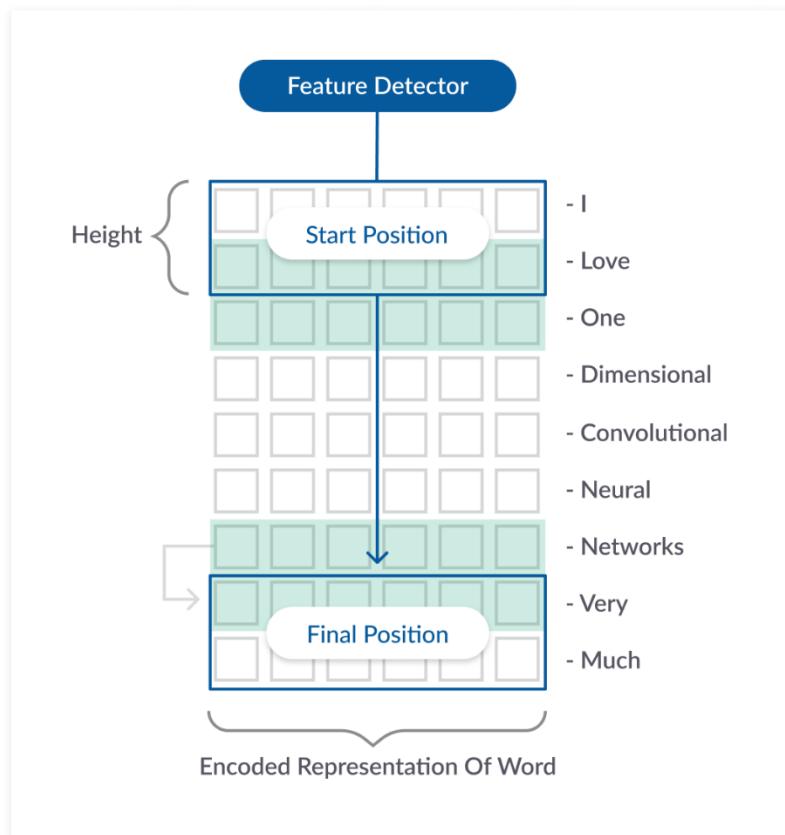


图 5.24 1D 卷积示例

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。深度对应输入特征映射的个数。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 and cols values might have changed due to padding.

5.3.2 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，在预测阶段不工作

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 $28*28*1$ 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：

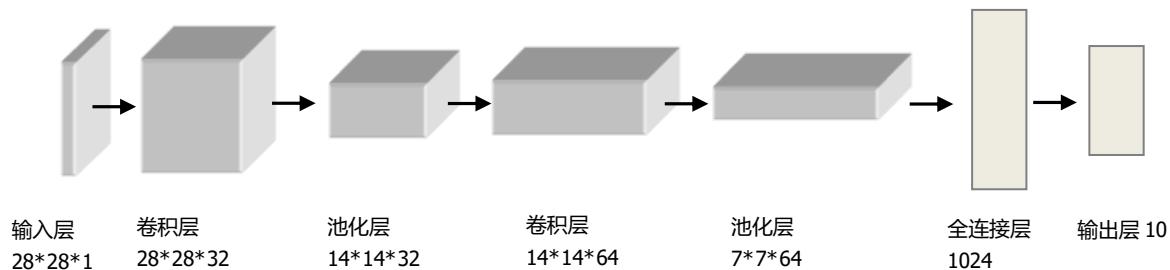


图 5.14 一个用于手写数字识别的卷积神经网络

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 5×5 。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 5×5 ；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=False（默认值）。

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
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¹。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）划分好了训练集和测试集；（2）每条数据描述一个人的活动，是一个时间序列数据，包括 128 个时间步；（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。（4）数据集包括 6 种人类活动，例如，行走，跑步，坐着等。用图 5.15 来描述该模型如下：

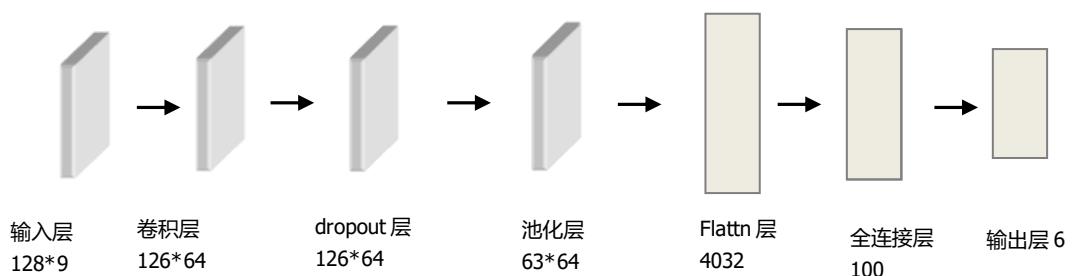


图 5.15 模型

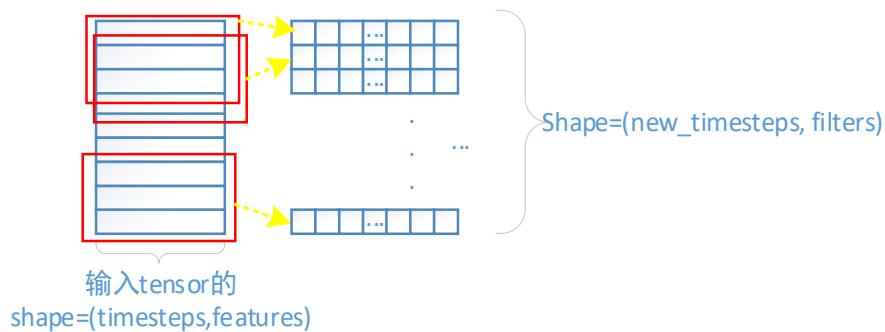


图 5.16 1D 卷积操作

¹ <https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones>

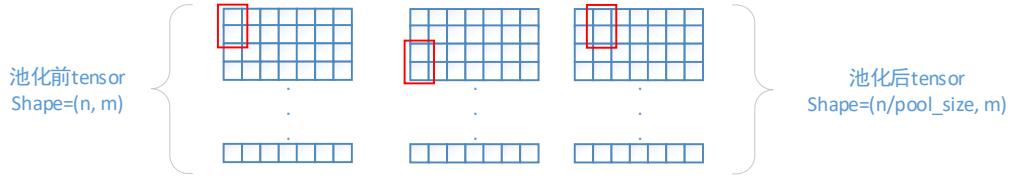


图 5.17 1DMaxPooling 操作

```

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

第一节：背景知识

6.1.1 关于表示学习

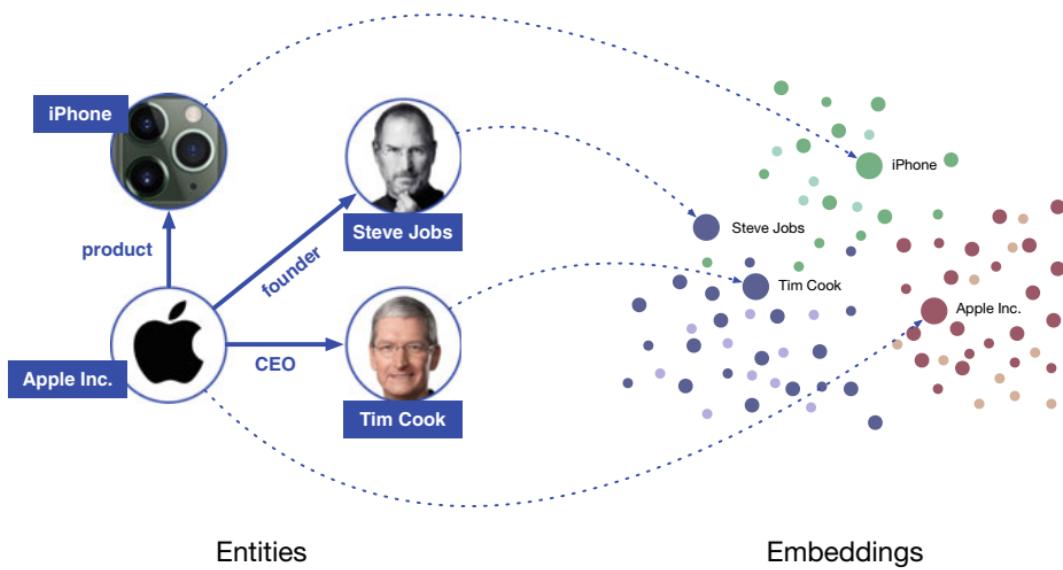
一个典型的机器学习系统，可以看做是下面的三个部分

$$\text{Machine Learning} = \text{Representation} + \text{Objective} + \text{Optimization}$$

首先，在一个原始的数据集上将有用的信息转换成内部的表示，例如特征向量。然后设计合适的目标函数；再采用优化算法发现模型最优的参数。

数据的表示的过程决定了怎样原始数据中的有用的信息被抽取出来，然后给下游任务，如分类、预测等。如果更多的有用的信息被抽取出来，下游任务的性能就会更好。数据表示学习是机器学习关键的部分。

传统的机器学习用特征工程从原始数据集抽取特征，建立特征向量。这是一种手工构建表示向量的过程。而表示学习，为目标对象自动的构建表示向量。这些目标可是词，也可以是网络的节点、边，可以是推荐系统的商品，客户。经过表示学习，它们都被用一个紧密的实数向量来表示。然后，深度学习模型可以基于这些表示向量，构建下游任务。



这是一个知识图谱表示学习的例子。知识图谱中的实体都被映射到了一个高维空间。即每个实体都用一个紧密的实数向量来表示了。

6.1.2 词的表示学习

One-hot Representation

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）。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（词）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。然后为每个词建立一个向量，向量的维度等于词表大小。一个词的向量中只有与编号对应位置上的数字为 1，其他都为 0。例如，假设“话筒”的词的编号是 3，“麦克”的词的编号是 8（假设从 0 开始记）

“话筒”表示为 [0 0 0 **1** 0 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 ...]

然而，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识别出语义相似性。

但在这种编码方法上，有语言模型可以用来建立词之间的关系。

Distributed Representation of Words

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”的方式表示可谓是将 Deep Learning 算法引入 NLP 领域的一个核心技术。

一个 word embeddings **W:words→Rⁿ**

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。语义相似的词被映射到空间相邻的位置，语义不相似的词在空间的位置比较远。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，

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

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

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。

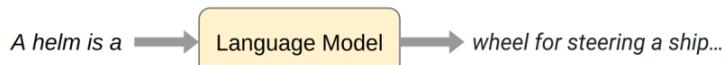
词向量的用途

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

6.1.3 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。在大语言模型之前，它是机器翻译、语音识别的关键技术。在大语言模型时代，它可以完成更多的自然语言处理的任务。在大模型时代，A language model is a box that takes text (a prompt) and generates text (a completion) probabilistically.



(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：

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

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。

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

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。

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

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，

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

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

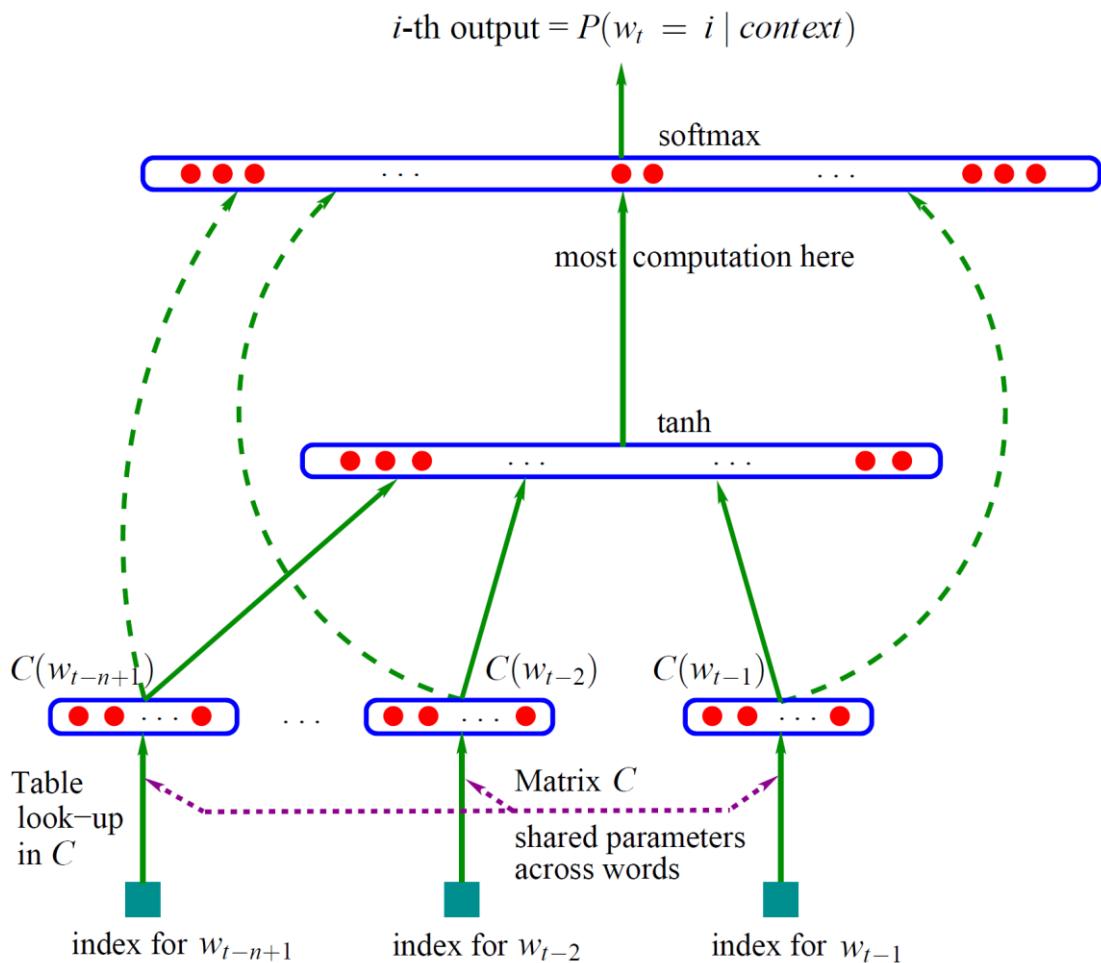


图 6.1. 神经网络语言模型

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。它将输出压缩到了-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：

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

$$\text{output} = \text{softmax}(y)$$

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 ，这个矩阵是对从输入层到输出层的直连边 x 的计算。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。

6.1.4 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**（预训练的通用的词向量）。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。学习的词向量用在进一步的文本挖掘或 NLP 任务中。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千亿词。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，因为都是要计算平均）

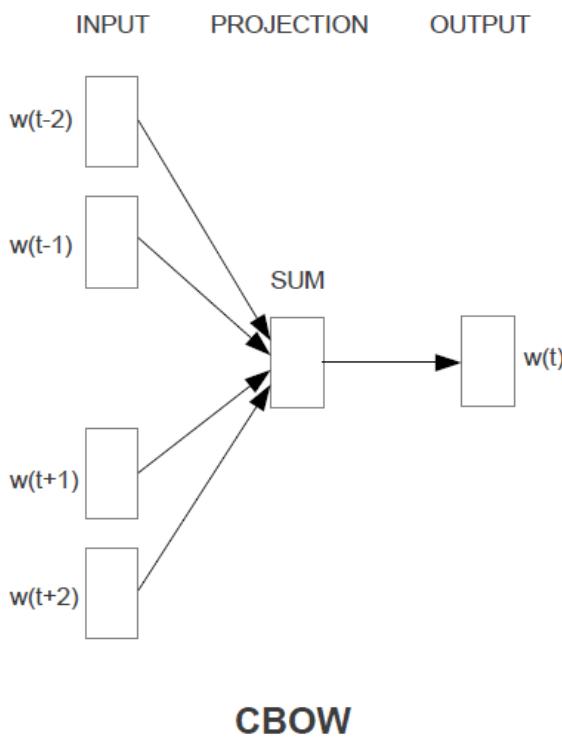


图 6.2. CBOW 模型

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 w_t 来预测

该词之前和之后各 c 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:

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

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。

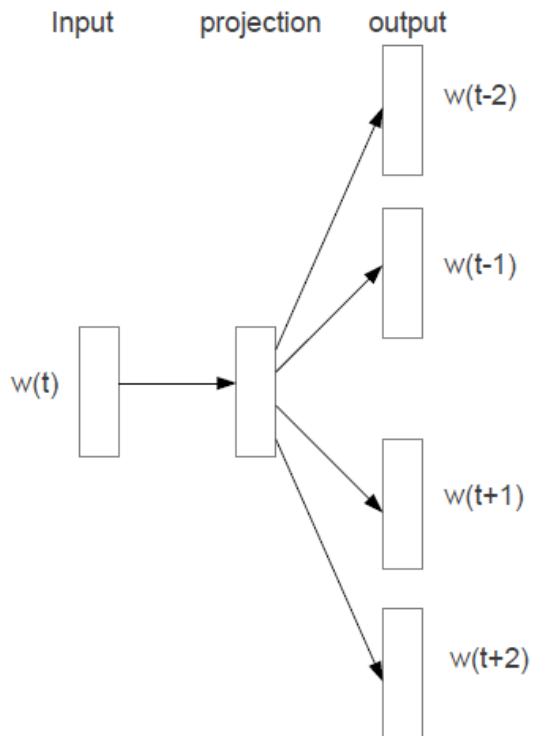


图 6.3. Skip-gram

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

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

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 。

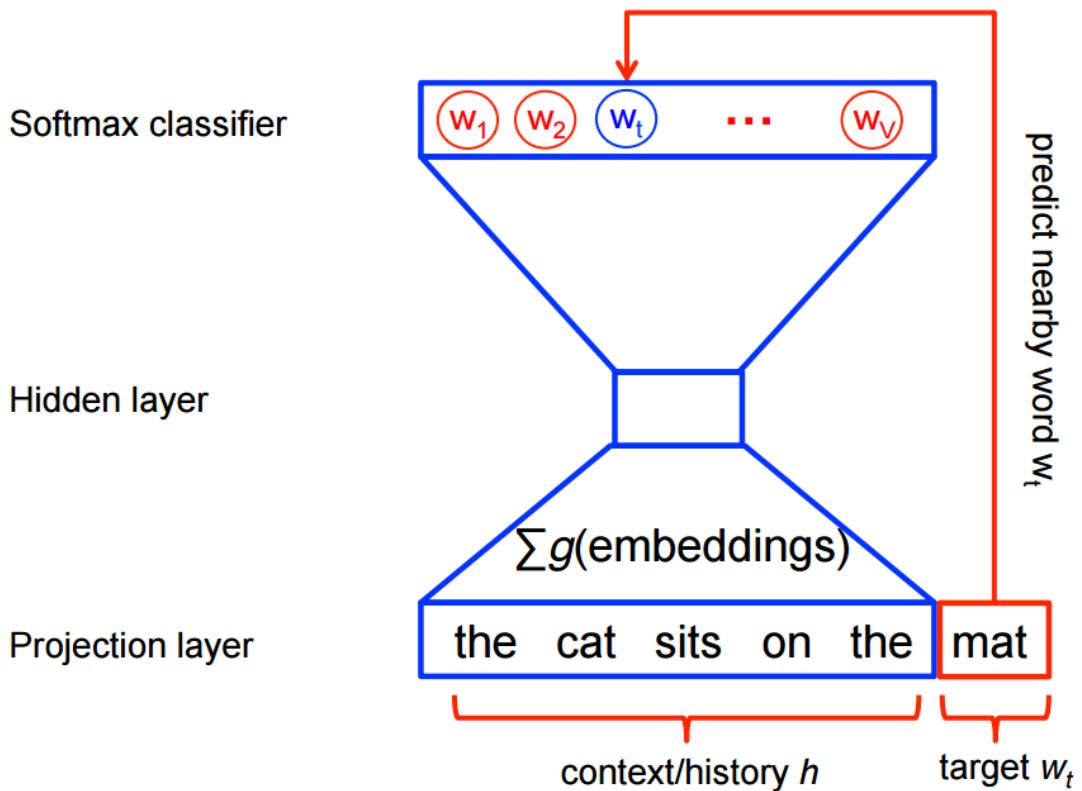


图 6.4. 神经网络语言模型

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。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

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

$Q_{\theta}(D = 1|w_t, h)$ 是 Logistics 回归概率，即给定上下文 h 在数据集 D 中看见词 w_t 的概率（所谓数据集 D 是指我们的目标词和负采样的噪声词的集合，图 6.5 的最上面一层）。该值按照学习到的词向量 θ 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} 。

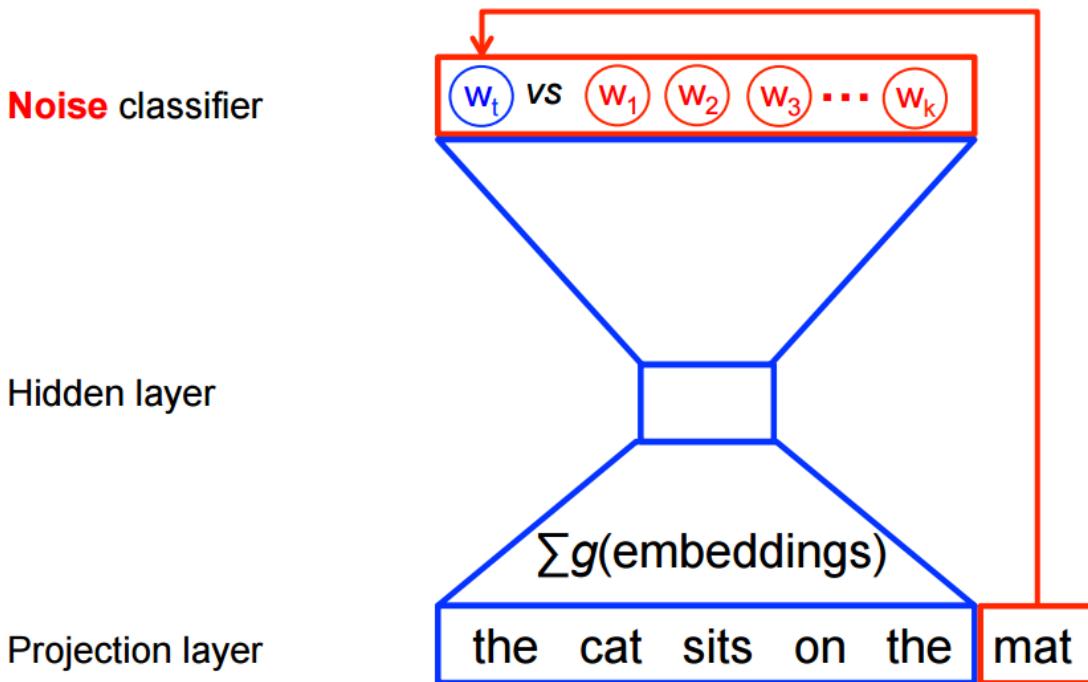


图 6.5. CBOW

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）

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 θ 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。

我们可以把上面第 t 步的公式拆开成两步， $J_{\text{NEG}}^{(t1)} = \log Q_{\theta}(D = 1|\text{the}, \text{quick})$ 和 $J_{\text{NEG}}^{(t2)} = \log Q_{\theta}(D = 0|\text{sheep}, \text{quick})$ 。这就是希望，模型喂入词 `the` 和 `quick` 时预测标签是 1，模型喂入的是 `sheep` 和 `quick` 时，模型的预测标签是 0。因此可以构建图 6.6 的模型

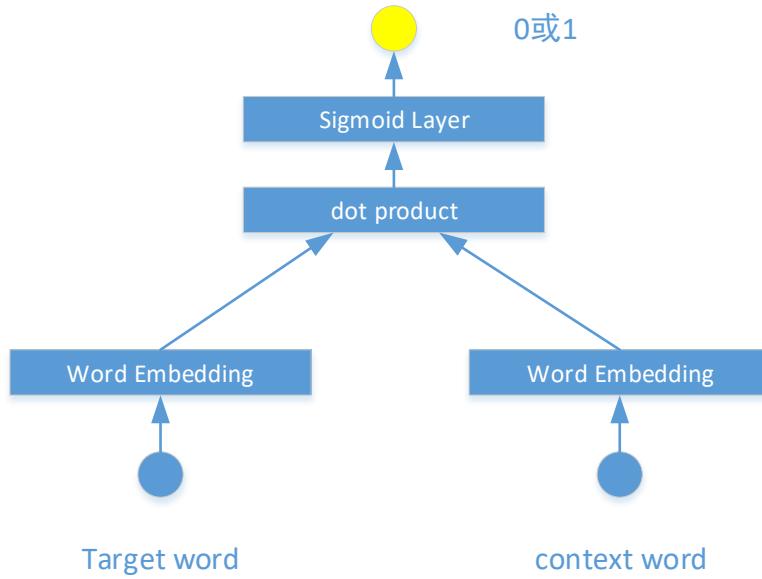


图 6.6 在 keras 中实现的一个 skip-grams 模型

图 6.6 是我们下面要在 keras 中实现的一个模型。

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)

shape 中的 sequence_length，即参数中的 input_length。该层使用的示例代码如下：

```
model = Sequential()
model.add(Embedding(1000, 64, input_length=10))
```

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) 。

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。为了平衡，频繁的词会被少采样。sampling_factor 是采样因子。设为 1, 即均匀的采样。越小的值表示，频繁的词越会被少采样。

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
    reverse_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
    return data, count, dictionary, reverse_dictionary, unk_count
```

```

        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 是词的编号的序列； 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。 Embedding 类中的参数 input_length 设为 1，是因为模型的输入就是一个词。

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

```

ndlabels=np.array(labels)
for cnt in range(epochs):
    idx = random.sample(range(vocabulary_size),batch_size)
    loss = model.train_on_batch([word_target[idx],word_context[idx]],
ndlabels[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

train_on_batch 和 fit 两个函数的差别是：

- A. 在 fit 函数中规定了 epoch，即在整个训练集上的训练趟数。在每一趟训练中，将训练集按照 batch_size 进行划分。；在每个 batch 上进行训练更新模型参数。
- B. train_on_batch 函数仅仅在给定的训练集上训练一次。

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。因此，train_on_batch 就需要自己写循环语句进行足够趟的训练，如上面的程序所示。

(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。

Keras 中也提供了 callback 类来完成在 fit 函数训练模型的过程中，考察模型或保存训练结果，或者完成 early_stopping。参见 <https://keras.io/api/callbacks/>

第七章：基于 CNN 的文本分类

第一节：文本处理基础

本节介绍在传统的文本挖掘中的一些基础知识。

7.1.1. 文本处理的一些相关概念

我们首先介绍文本处理的一些相关概念和技术术语，如下：

1. 文本、文档和语料库：文本（text）是指数据格式，文档（Document）是指文本文件。在信息检索中，文档检索系统检索对象，或可以理解为保存在计算机上的一个待检索文件（一个文本文件，一篇网页等）。语料库（corpus）：多个文档组成的文档集合。
2. 结构化数据和非结构化数据：信息可以划分为两大类。一类是没有清晰和明显语义结构的数据，称为非结构化数据；而与之相对应的另一类信息称之为结构化数据。如文本、图像、声音、网页等，我们称之为非结构化数据。结构化数据最典型的例子是数据库中的数据。如，

姓名	性别	年龄	婚姻
王五	M	20	未婚
李四	F	30	已婚

结构化的数据很方便进行分析，如上表中，如果需要查找未婚的男性，因为数据格式都是固定的，标准的很方便进行查找。甚至格式化的数据可以进行量化，然后进行数据分析。

文本是非结构化数据。例如，

“王五是个男的今年 20 岁了还没结婚。李四今年 30 岁是个女的已经结婚了”

我们不能从这段文本中直接获得被描述的人的姓名、年龄、婚姻状况等信息。我们是说通常文本数据是非结构化的数据。但是如果文本中的数据按照一定规律存放，很容易读取，我们也不说它是非结构化数据。比如一段文本内容

Name=王五; Age=20; sex=男; marriage=未婚

Name=李四; Age=30; sex=女; marriage=已婚

还有像 HTML、XML 等文件，我们说它们是半结构化数据。

传统的数据分析方法不能直接作用在文本数据上。因此文本数据必须有相应的处理方法，将其量化后才能进行分析。所以，非结构化数据分析的思想都是将它们进行量化后才能进行分析。

3. 词条(token)和词项 (term)：词条是指对文档进行分词操作得到的一个单元；词项是指词典中的一个词。这里词典的含义是进行文本分析时建立的一个词的集合。词条不一定会出现在词典中。

7.1.2. 一些文本处理的技术

1. 词干化或词干还原：是指将英文中可以表达为多种形态的一个词，如一个词有多种派生形式，如名词化，动词分词，形容词等，演变成了含义相近的新的词。在处理时，需要用一个词干表示这些词。例如，are, is 词干化为 be; prices 和 pricing 可以词干化为 price。有很多词干化的工具包，如最常见的 porter

(<http://www.tartarus.org/~martin/PorterStemmer/>)。斯坦福的自然语言工具箱中也有词干化的工具。 (<http://nlp.stanford.edu/software/tagger.html#Download>)。

当然，词干的精确形式并不重要，重要的是能够得到等价类。这句话的意思是，如果把 “quickly” 这个词词干化得到的是 “quic”，这个操作得到的结果是不精确，甚至是错误的。但是，如果所有的 quick 的衍生词，包括 quick 本身的词干化的结果都是 quic，而其他词的词干化结果不好得到 quic，就没有关系。因为所有的 quick 和 quick 衍生词的词干化结果是相同的，它们得到了等价类。

2. 词项归一化 (normalization)：是将看起来完全不一致的多个词条归纳成等价类，以便在它们之间进行匹配。比如，查询 USA 时，肯定希望检索系统能够返回包含 U.S.A. 的文档。大小写转换也是一种词项归一化操作。

3. 停用词表使用的探讨

一些词项在文档中出现的太频繁了，不能表达文档的主题信息，它们称为停用词。如 “the”, “and”, 啊, 吧等。在文本挖掘中，我们更喜欢频繁出现在一篇文档中，而没有出现在其他文档中的词。它们更能描述文档的主题信息。而语料库中每篇文档中都出现的词，不具有描述文档含义的作用。在文本处理时，会预先建立一个停用词表，然后根据停用词表把文本中的停用词删除。

在一些特定的文本挖掘任务中，比如文本分类中，会使用停用词表，以提高系统的性能，以及效率。因为文本分类中对实词（名词、动词、形容词）更感兴趣，因为他们通常描述了文本的内容。

7.1.3 中文分词

由于中文的词之间没有分隔符，如果要进行中文文本处理，必须要用特殊的方法将中文的句子分词一个个的词条。中文分词是中文信息处理的最基本任务，几十年的研究里有丰富的成果，技术很成熟，而且有很多的开源中文分词的软件。

中文分词的技术分为两大类，基于词典的分词和基于机器学习的分词。2002 年以前的研究都是基于词典的分词方法。（1）实践证明，基于手工规则的分词系统（注：基于词典的分词方法中再添加人工规则，如词法规则等，也可以是统计出的一些规则）在评测中不敌基于统计学习的分词系统（注：这里是指机器学习的方法）；（2）未登录词（out-of-vocabulary）造成的分词精度失落至少比分词歧义大 5 倍；（3）迄今为止的实验结果证明，能够大幅度提高未登录词识别性能的字标注统计学习方法优于以往的基于词典的方法，并使得分词精度达到新高（注：未登录词是造成分词精度低的主要原因，而基于字标注的机器学习方法提高了未登录词识别性能）。

7.1.4 倒排索引

传统方法在语料库上进行信息检索，或者出了另一个文档集合时，需要将它们建立成一个词项文档矩阵。但如果有一个语料库，有文档 100 万篇，而词项的个数是 50 万，可以试想一下，为了进行信息检索建立的词项-文档关联矩阵有多大？它的缺点和优点是什么？

观察词项-文档矩阵，我们可以发现该矩阵是高度稀疏的。即大部分的元素是 0，极少部分元素是 1。对上面的语料库粗略做个计算，由于每篇文档的平均长度是 1000 个单词，所以 100 万篇文档在词项-文档矩阵中最多对应 10 亿个 1，而在这个语料库上这个文档词项矩阵的大小是 $100 \text{ 万} * 50 \text{ 万} = 5000 \text{ 亿}$ 。也即 $1 - 1/500 = 99.8\%$ 的元素是 0。因此转换存储方式，只保存元素为 1 的信息将大大减小保存语料库信息所付出的空间开销。

上述思路引出信息检索中的一个核心概念：倒排索引（inverted index）。这里倒排的概念是建立从词项到文档的映射。其数据结构如图 4-3

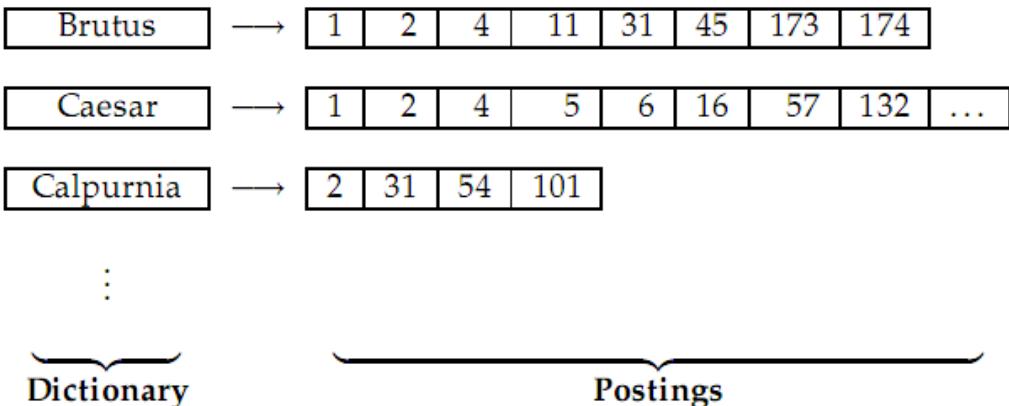


图 7.1 倒排索引

左边是词项词典 (lexicon, 有时也称作 dictionary, vocabulary) , 右边是记录每个词项在文档中出现了一次的文档编号的列表, 称作倒排记录表 (posting list) , 表中的元素称作倒排记录 (posting) 。词典按照字母顺序排序, 倒排记录表按照文档编号进行排序。

7.1.5 向量空间模型

1. tf-idf 词项权重的计算

在布尔检索中, 给定一个布尔查询, 一篇文档要么满足查询要么不满足。在文档集规模很大的情况下, 满足布尔查询结果的文档数量可能非常多, 往往会大大超过用户能够浏览的文档的数目。因此对于信息检索系统来说, 对文档进行评分排序很重要。用户可以根据评分 (文档和查询的相关度) 进行检索。例如, 搜索引擎就是这么做的。

布尔检索模型只考虑了词项在文档域中出现与否的情况。如果我们想更准确的给文档评分, 需要换一种方式来描述文档 (原始的文档描述就是词的集合)。粗略的说, 文档内容(以下均简称为文档)中出现频率越高的词项, 越能描述该文档 (不考虑停用词)。因此可以统计每个词项在每篇文档中出现的次数, 即词项频率, 记为, $tf_{t,d}$ t 为词项, d 为文档。获得文档中每个词的 tf 权重, 一篇文档则转换成了词-权重的集合, 通常称为词袋模型 (bag of words model)。我们用词袋模型来描述一篇文档。词袋的含义就是说, 像是把一篇文档拆分成一个一个的词条, 然后将它们扔进一个袋子里。在袋子里的词与词之间是没有关系的。因此词袋模型中, 词项在文档中出现的次序被忽略, 出现的次数被统计。例如, “a good book” 和 “book good a” 具有同样的意义。

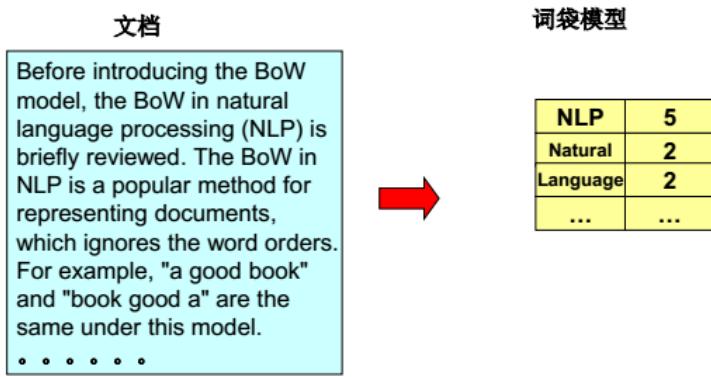


图 7.2 词袋模型

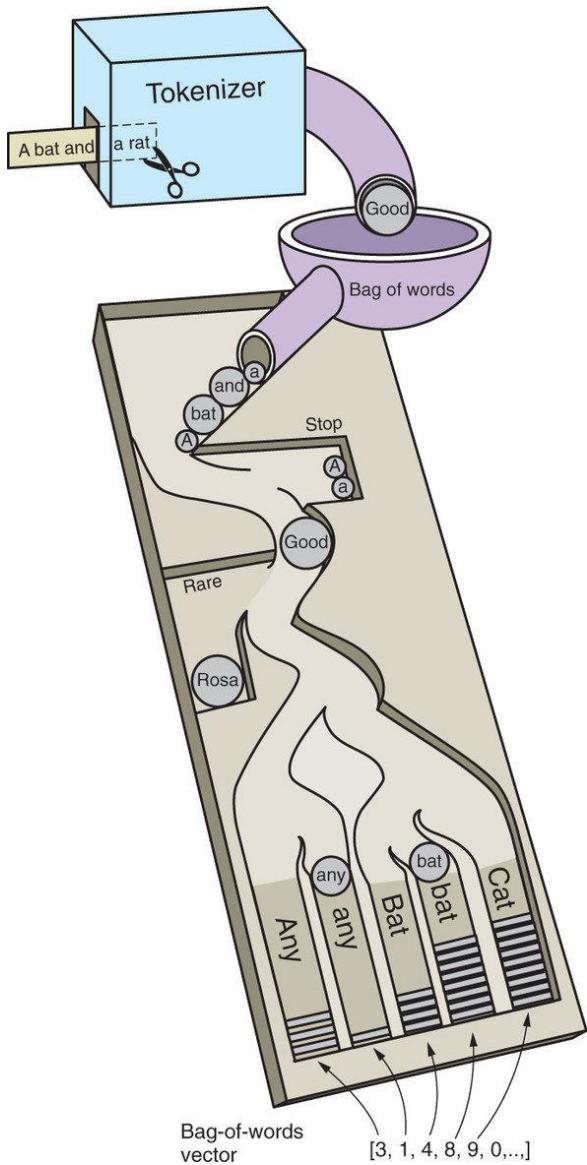


图 7.3 词袋模型处理语料库示意图

词袋模型中只为文档中的词计算了 tf 权重。tf 权重只考虑词在文档中出现的频率。如果一个词，只在某篇文档中出现，而没在文档集合中的其他文档中出现，则该词可以

很好的区分描述这篇文档，则应该给该词更高的权重。例如，在描述汽车的文档集合中，几乎每篇文档都会出现 car, auto 这样的词，这样的词不具有区分描述能力也即，我们想计算词项的权重时也考虑是否该词具有很好的描述性。

因此，又会统计一个词项的文档频率 df ，即在文档集合中，出现该词项的文档的数目。在实际应用中会采用逆文档频率 idf 。

$$idf_t = \log \frac{N}{df_t}$$

N 是文档集合中的文档数。可以发现，一个词如果在文档集合中出现的次数越少，它的 idf 得分越高。

现在文档中的每个词可以计算两个权重 tf 和 idf ，单凭哪一个权重来衡量一个词项的重要性很多时候并不合适（注：也有的应用只单独根据 tf 或 idf 来作为权重）。所以，用 $tf-idf$ 权重来表示词项的权重。

$$tfidf_{t,d} = tf_{t,d} \times idf_t$$

词项的 $tf-idf$ 权重的含义如下：

- (1) 一个词在少数几篇文档中多次出现，它的权重越大（此时对文档能够提供最强的区分能力）
- (2) 但词项在一篇文档中出现次数少，或者在很多文档中出现，权重取值次之。
- (3) 如果词项在所有文档中都出现，那么它的权重值最小（为什么？）

如果把一篇文档看做是向量，其中每个分量都对应词典中的一个词项，分量是 $tf-idf$ 计算出的权重值。某词项在文档中没有出现，其对应的分量为 0。我们就可以用一个向量来描述文档。一系列文档在同一向量空间中表示，就称为向量空间模型（vector space model）VSM。同一向量空间的含义是，所有文档中相同位置的分量（元素）所对应的词是相同的。

	Doc1	Doc2	Doc3
car	0.88	0.09	0.58
auto	0.10	0.71	0
insurance	0	0.71	0.70
best	0.46	0	0.41

2. 余弦相似度

前面提出了文档向量的概念。其中每个分量代表词项在文档中的相对重要性。一系列文档在同一向量空间的表示称为 VSM (Vector Space Model)。VSM 是词袋模型。向量空间模型是信息检索、文本分析中基本的模型。通过该模型，可以进行有序文档检索、文档聚类、文档分类等。当然，现在的研究有新发展。出现了很多模型代替 VSM。

每篇文档在 VSM 中用向量表示，那么计算两篇文档的相似度自然的想到用两个向量的差值。但是，可能存在的情况是。如果两篇相似的文档，由于文档长度不一样。他们的向量的差值会很大。

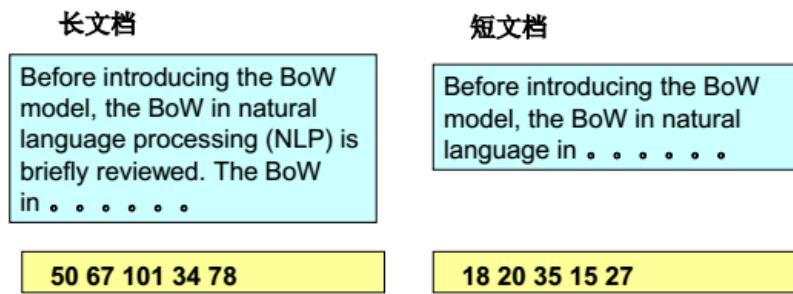


图 7.4 长文档和短文档的差异

用重合度评分指标就没有考虑文档长度的影响。余弦相似度是使用的非常广泛的计算两个向量相似度的公式，它可以去除文档长度的影响。

$$\text{sim}(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)| |\vec{V}(d_2)|}$$

公式中 $\vec{V}(d_1)$ 和 $\vec{V}(d_2)$ 是文档 d_1 和 d_2 用向量形式的描述。 $\vec{V}(d_1) \cdot \vec{V}(d_2)$ 是两个向量的点积（内积）。两个向量的内积定义为

$$\vec{x} \cdot \vec{y} = \sum_{i=1}^M x_i y_i$$

公式的分母是两个向量的欧几里得长度。文档 d 的向量表示为 $\vec{V}(d)$ ，它是一个 M 维向量 $\vec{V}_1(d) \dots \vec{V}_M(d)$ ， $\vec{V}_M(d)$ 是向量中的一个元素。文档 d 的欧几里得长度是 $\sqrt{\sum_{i=1}^M \vec{V}_i^2(d)}$

如果把每篇文档的向量除以该文档的欧式长度，得到的就是欧式归一化结果。也即前页的向量相似度公式是两个归一化向量的点积。

$$\text{sim}(d_1, d_2) = \vec{v}(d_1) \cdot \vec{v}(d_2)$$

该值也称作是两个向量的余弦夹角。因此，我们可以看到余弦相似度的计算是抵消了文档长度的影响。**向量余弦夹角相似度，不考虑向量的长度，只考虑两个向量的余弦夹角 $\cos(\theta)$ 大小。**

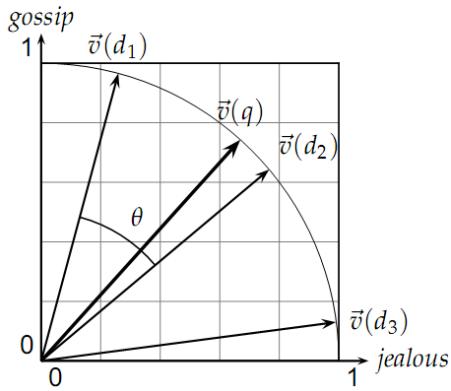


图 7.5 向量的余弦夹角

注：可以证明如果把查询向量和文档向量都规范化到单位向量（即前面讲余弦相似度提到的，将每个向量除以它的欧几里得长度（L2 范数），就是将向量规范化到单位向量），那么采用欧式距离计算查询和文档的距离，然后用该距离进行文档排序的检索结果和余弦相似度是一样的。（欧式距离 $|\vec{x} - \vec{y}| = \sqrt{\sum_{i=1}^M (x_i - y_i)^2}$ ）。因为单位向量之间的距离就是上面的一个圆上的两个节点间的距离。可以看到圆上两点的距离和向量的夹角是等价的。

第二节：Keras 的文本处理

keras 提供了几个文本处理的类。

7.2.1 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

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。

该类提供的主要方法

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

对语料库进行预处理, 基于文档的列表更新词汇表。

在使用下面的 texts_to_sequences 或者 texts_to_matrix 方法前, 需要先用 fit_on_texts 对文档集合进行预处理。

texts: can be a list of strings (列表中的每个元素是一篇文档, 见下面的例子), a generator of strings (for memory-efficiency), or a list of list of strings.

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

将文档转换成编号的序列。每个词分配一个从 1 开始的编号, 编号 0 是在后面进行补齐操作时使用。

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)
```

7.2.2 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.
```

```
from tensorflow.keras.preprocessing.text import one_hot  
vocab_size = 50  
encoded_docs = [one_hot(d, vocab_size) for d in docs]  
print(encoded_docs)
```

7.2.3 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. Hello world, good morning.'  
result = text_to_word_sequence(text)  
print(result)
```

第三节：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。模型如图 7.6 所示。（注：图中显示的是送入的一个 batch 的文档，相应的输出是一个 batch 的结果）

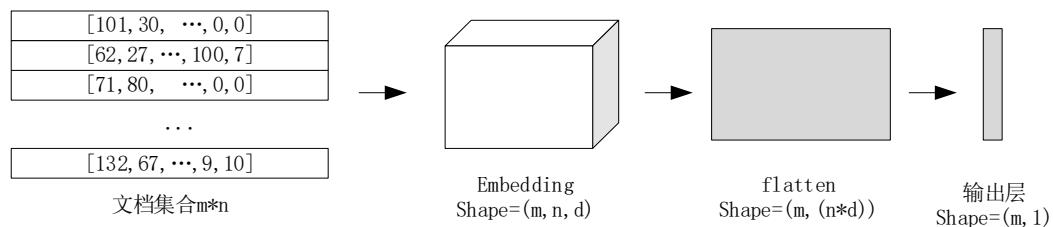


图 7.6：一个简单的文本分类模型

在 keras 中处理文本集合建立数据集的基本步骤总结如下：

1. 准备好数据集，它是一个列表，每个元素是一篇文档。
2. 创建 tokenizer 类的对象
3. tokenizer 对象在文本集合的列表上调用 fit_on_texts 方法进行拟合，学习一个词汇表。
4. 调用 tokenizer 对象的 texts_to_sequences 方法将文本集合的列表转换成编号序列。
5. 调用 pad_sequences 函数将文本编号序列进行对齐。即每篇文档的长度一致。

如此，就创建好了深度学习需要的数据集。

设 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。

```
vocab_size = 10
t=Tokenizer(num_words=vocab_size)
t.fit_on_texts(docs)
encoded_docs=t.texts_to_sequences(docs)
```

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，n 是词向量长度，d 是文档长度。但在使用 Embedding 类时，不需要给出 m，下面的代码中给出了词汇表大小，词向量长度（设的是 8）和文档长度（input_length 参数）。

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

vocab_size = 10
# 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([1,1,1,1,0,0,0,0,0])

t=Tokenizer(num_words=vocab_size)
t.fit_on_texts(docs)
encoded_docs=t.texts_to_sequences(docs)

# 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)

```

练习：在路透社预料数据集的训练集上训练一个分类器，在测试集上评估模型。

第四节：CNN 文本分类模型

7.4.1 模型结构

一个详细描述的文本分类 CNN 结构图见图 7.7 (详见论文 A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification)。图 7.7 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：

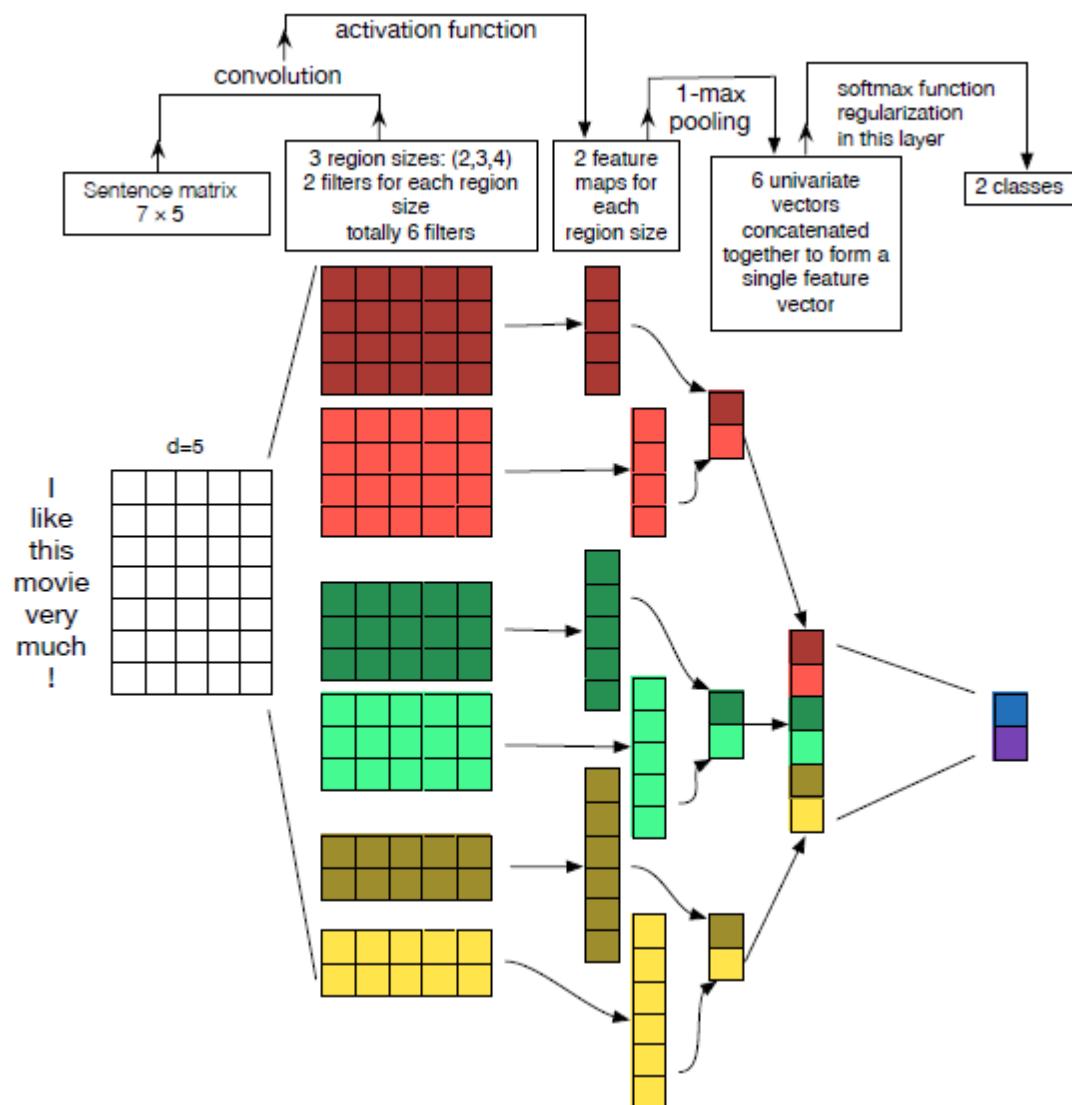


图 7.7 有多个 window 的滤波器的文本分类 CNN

(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 $x_i \in \mathbb{R}^k$ 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 $w \in \mathbb{R}^{h*k*d}$ 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 7.7 中有 $2*3=6$ 个滤波器（每个 region size 两个滤波器，一共 3 个 region size）。

(3) dropout：在卷积层的输出加 dropout 操作。

(4) 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 7.8 所示。

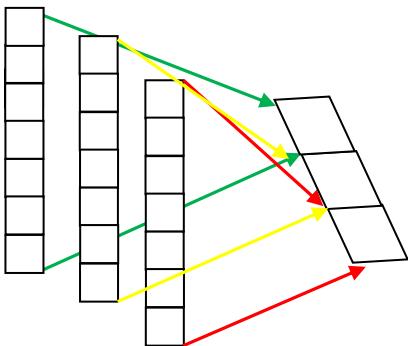


图 7.8 pooling 层

(5) 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，此时用 sigmoid 激活函数）

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。

7.4.2. 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:]
```

5. 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`。（在 keras 中建模时不需要考虑数据的 `batch_size` 维度，即前面的 `samples`）

```
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=(fszie,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。**注: 如果是 1D 卷积, 它输出的结果就不是这个 shape。请试验用 Conv1D 和 GlobalAveragePooling1D(), 替换上面的卷积和池化操作。**

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))
```

7.4.3. 加入预训练词向量的 CNN 文本分类模型

7.4.2 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))k

```

7.4.4 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/

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()

model.add(Embedding(vocab_size,
                    embedding_dims,
                    input_length=doc_length))
model.add(Dropout(0.2))

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 或 RvNN。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或者是链。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。

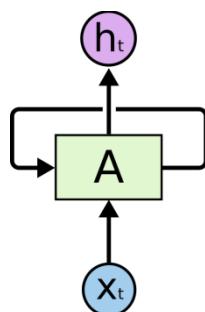


图 8.1 有循环的 RNN

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。红框指示的是一个时间步 (time step)。

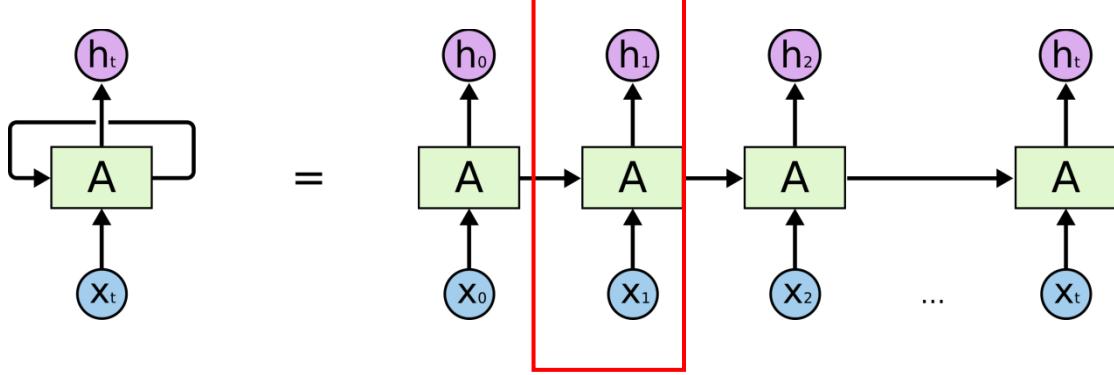


图 8.2 一个展开的 RNN

在实际应用中，是一个神经网络被使用多次。每次处理序列数据的一个时刻 i 的输入 x_i ，并把计算的输出 h_i ，传递给下一个计算。当用这个神经网络再计算时刻 $i+1$ 的输入 x_i 时，也使用了 i 时刻输出 h_i 。如此用一个神经网络计算了输入序列每个时刻的数据，并形成了信息传递。

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。

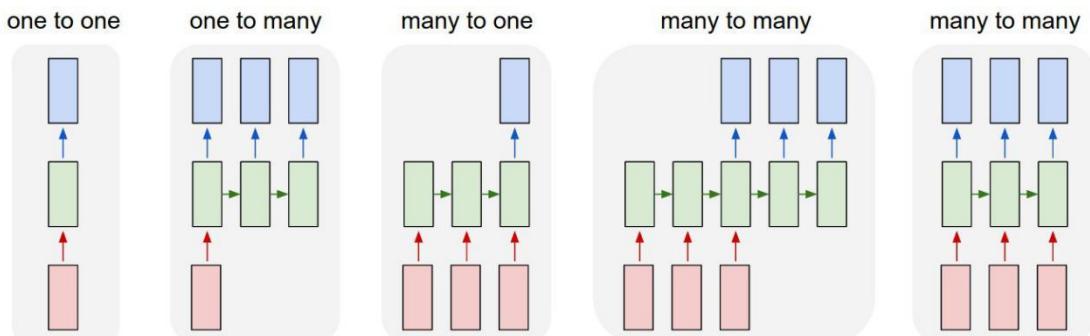


图 8.3 RNN 的各种结构

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 vanila 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。

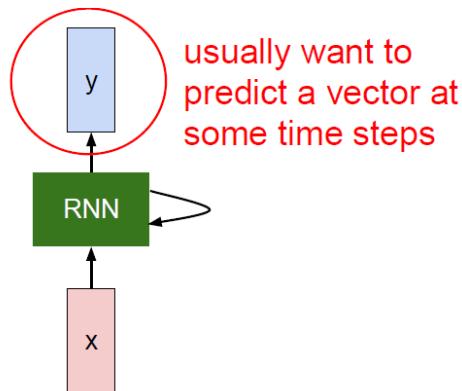


图 8.4 一个循环未展开的基本 RNN 结构

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”字符序列“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 可以建模紧跟着该序列的下一个字符的概率分布。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 (英文里用 context 或 history 表示之前出现的信息) “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，将该向量转换成字符集里对应的字符。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 $\langle 1.0, 2.2, -3.0, 4.1 \rangle$ 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”。

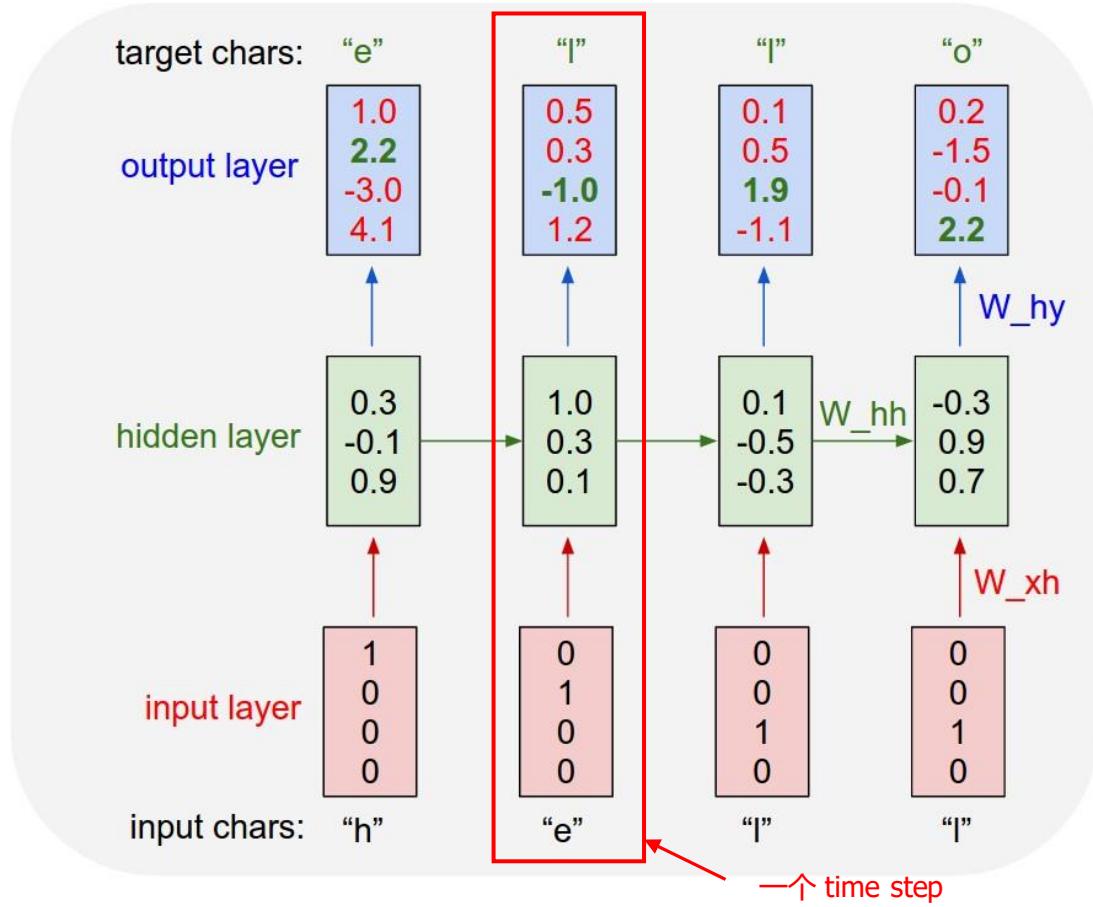


图 8.5 展开的 RNN 实例

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 表示隐层后面加上的输出层的输出，即 RNN 一个时间步的输出） 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。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：

$$y_t = f(W_{hy} h_t + b)$$

函数f 表示激活函数。

对照图 8.5 我们看看设计 RNN 时的几个重要参数：(1) 输入序列的长度，即 RNN 时间步的步数。不需要用户确定，喂入模型的序列长度是多少，则 RNN 的时间步就是多少。但需要用户把所有的输入序列 padding 到等长。(2) 每个时间步的输入是一个向量（输入层），向量的长度。对于处理的文本序列，就是词向量的长度。(3) 隐层神经元的个数。(4) 输入层到隐层的权重矩阵 W_{xh} 。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) 状态 h_{t-1} 传递到时间步 t 来计算 h_t 时的权重矩阵 W_{hh} 。它的 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矩阵的 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。图 8.6 中隐层只有一层。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，对应图 8.5 的权重 W_{hh} 。

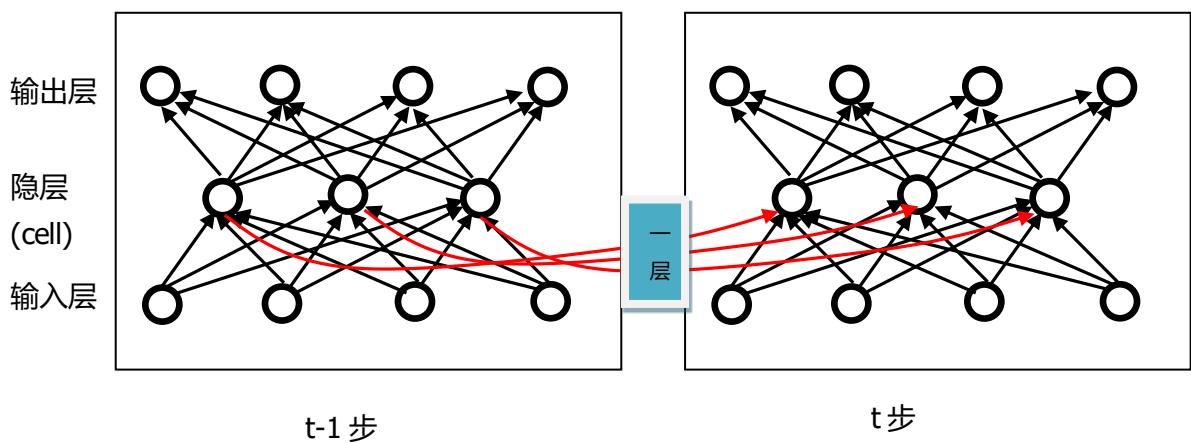


图 8.6 对应图 8.5 的 RNN 一个时间步

第二节：Keras 中构建 RNN

8.2.1 介绍

在 Tensorflow 和 keras 中实现的 RNN，它的核心是 cell。Cell 就是一个神经网络。例如，图 8.6 中的神经网络只有一层。而图 8.7 所示的 cell 是一个两层的神经网络

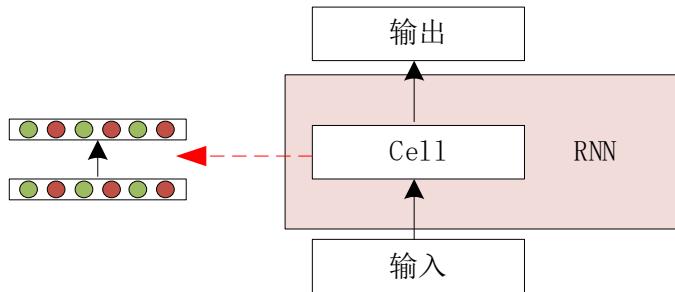
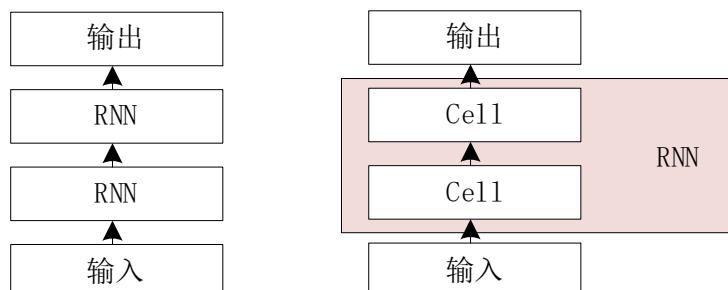


图 8.7 Cell 结构

每个 cell 的输出会传递到下一个时间步，在下一个 cell 的输入做计算。

Keras 提供了两种级别的 API 来构建 RNN。一种简单的方式就是提供了 SimpleRNN、LSTM 和 GRU 等类直接构建 RNN，将在 8.2.2 节的介绍。需要用户在 RNN 的输出上自己再加上输出层来完成特定的任务。Keras 提供的这些 RNN 只包含一个 cell。
SimpleRNN 只是相当于只有一层的神经网络。如果想构造更复杂功能的 RNN，有下图的两种方式。一是可以使用多个 RNN 堆叠。如下图左图所示。



还有一种方法是在 cell 的级别上。自己创建有多个 cell 的 RNN。如上图右图所示。Keras 提供了三种 cell。

`keras.layers.SimpleRNNCell` corresponds to the SimpleRNN layer.

`keras.layers.GRUCell` corresponds to the GRU layer.

`keras.layers.LSTMCell` corresponds to the LSTM layer.

创建 cell 对象后，作为创建 RNN 对象时的参数，就可以创建自己的 RNN。下面的代码创建了有两个 cell 的 RNN。

```
from tensorflow.keras.layers import SimpleRNNCell, StackedRNNCells, RNN
cells = [SimpleRNNCell(24), SimpleRNNCell(128)]
stacked_cell = StackedRNNCells(cells)
rnn_layer = RNN(stacked_cell)
```

9.1 节最后的一个例子，演示了 keras 中如何创建一个有多个 cell 的 LSTM。

8.2.2 构建 RNN 的类

<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 和 tf.keras.layers.GRU。这些类已经包含了默认类型的 cell。用户也可以使用 tf.keras.layers.RNN 和 Cell 类构建自己的 RNN。9.3 节介绍了定制方法。

本节先介绍 tf.keras.layers.SimpleRNN。它是只有一个全连接层的 RNN。前一个时间步喂入的数据经过全连接隐层的计算，输出的结果喂到下一个时间步。对于两种特殊的 RNN：LSTM 和 GRU 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 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, SimpleRNN, Input, Dense

seq_length=20
model = Sequential()
model.add(Input(shape=(seq_length,)))
model.add(Embedding(input_dim=1000, output_dim=64))
model.add(SimpleRNN(256, return_sequences=True))
model.add(SimpleRNN(128))
model.add(Dense(1))
model.summary()
```

这个例子中，第一个 SimpleRNN 的输出是一个 3D tensor，它的 shape= (batch_size, timesteps, 256)。第二个 SimpleRNN 的输出是一个 2D tensor，它的 shape= (batch_size, 128)。就因为第一个 SimpleRNN 输出了每个时间步，如此可以在其上再加一个 SimpleRNN.

3. 最后一个时间步的状态 (state)

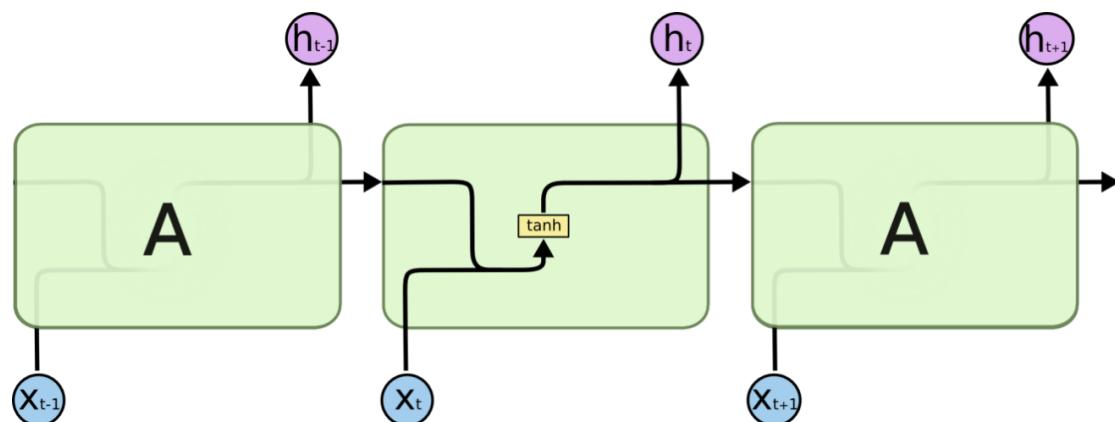
在每个时间步之间传递的 tensor 称作状态 (state)。根据 RNN 实现的不同，一个时间步的输出可以不等于一个时间步传递到下一个时间步的状态。例如，RNN 一个时间步的计算中有的模型在隐层的输出的计算

$$h_t = f(W_{hh} h_{t-1} + W_{xh} x_t)$$

后再加一层

$$o_t = f(W_{ho} h_t)$$

h_t 是时间步传递的状态(state)， o_t 是时间步的输出(output)。不过 keras 实现的 SimpleRNN 中，没有 o_t 。这个 h_t 既是状态 state，又是输出。如下图所示。



RNN 除了可以返回输出 tensor，也可以返回最后一个时间步的内部状态，即 state。

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，如果设置参数 `return_state=True`，则 RNN 层返回的结果是一个 list 结构，包括输出 (output) 和最后一个时间步的状态 (state)。[output, state]。

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 rnn-char-LM.py。对于字符采用 one-hot 编码。模型如图 8.8 所示。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。

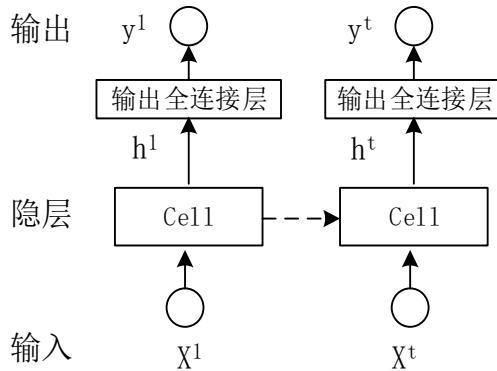


图 8.8：RNN 字符级语言模型

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 + seq_length + 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。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。这个可以通过 `model.summary()` 考察模型结构发现，在最后一层是 `time_distributed (TimeDistri (None, 25, 79))`

这是一个 TimeDistributed layer，即将一个层应用在了每个时间步上。

```

model= Sequential()
model.add(Input(shape=(seq_length,vocab_size), name='input'))
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，例如使用先前的文本帮助当前文本的理解。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。

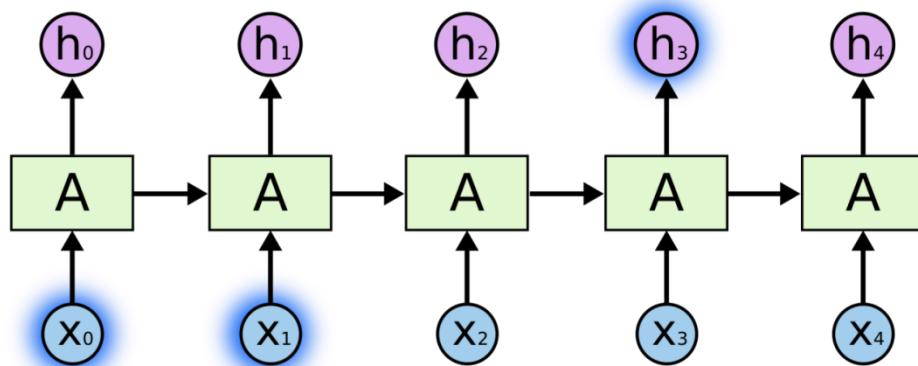


图 8.11. Short Term dependency

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。

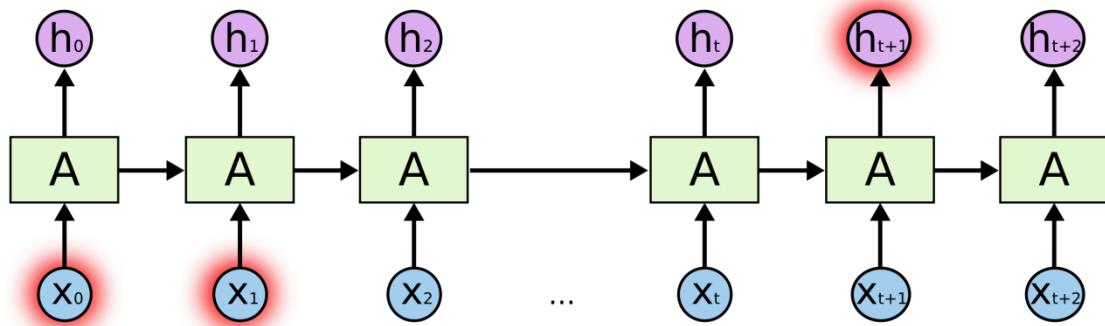


图 8.12. Long Term Dependency

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**，即 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。

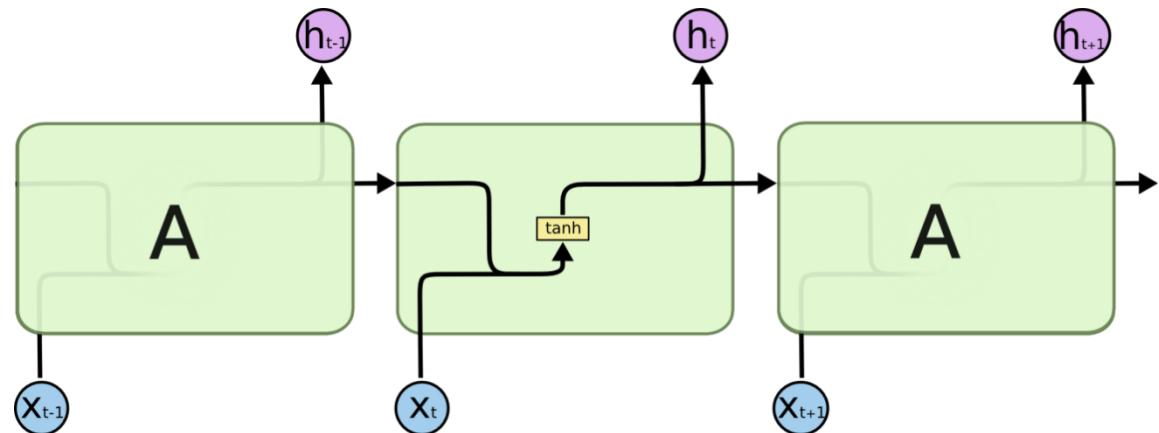


图 8.13. 在一个标准 RNN 中包含一个单独的层

LSTM 也有这种链式结构，但是它的 cell 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。

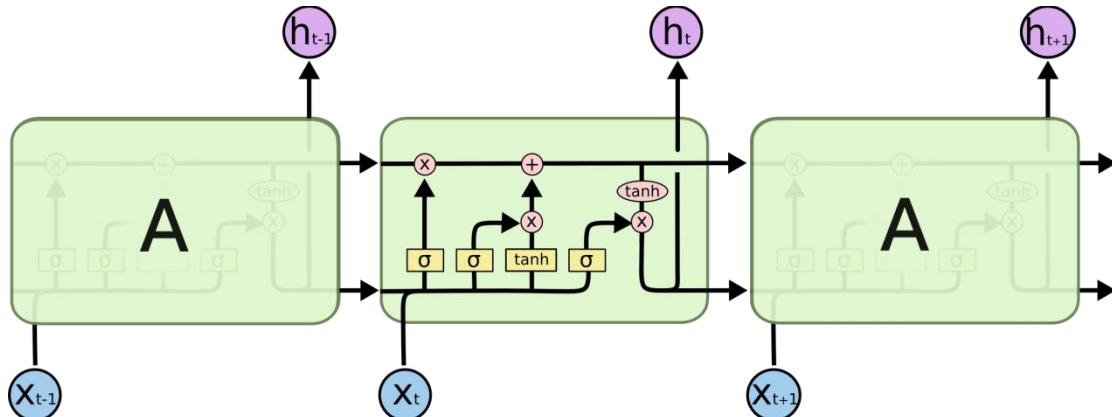


图 8.14 LSTM 中的 Cell

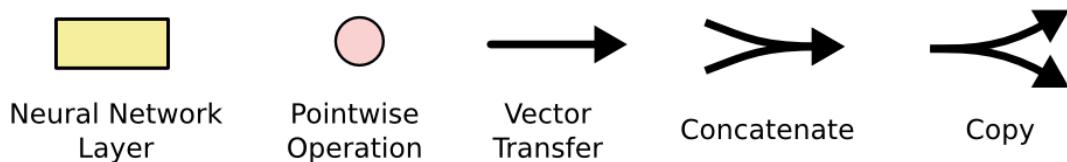


图 8.15 一些注释符号

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（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。

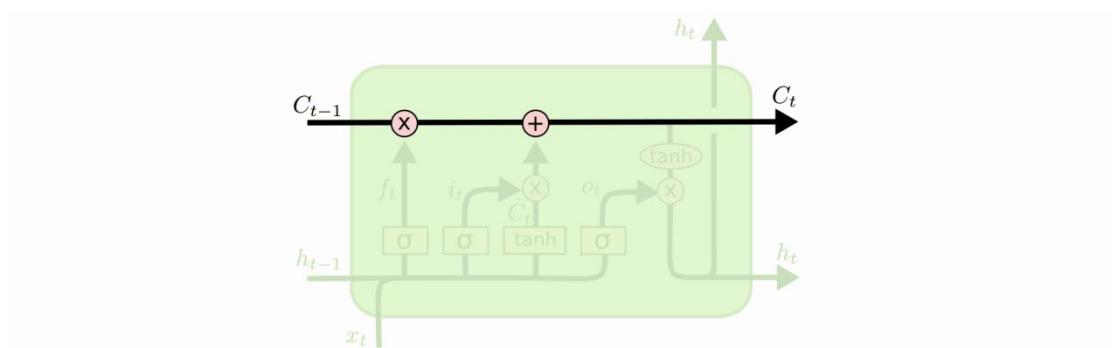


图 8.16 RNN 的信息传送

通过门 (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。

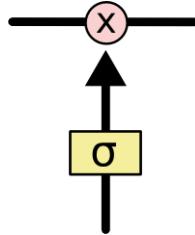


图 8.17 Gate 结构

Sigmoid 层输出 0~1 之间的**数值向量**（长度和 hidden_size 一致），描述了传递的数据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，位于 0-1 则表示部分通过。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 σ ，即三个门）

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。即，图 8.18 中的 f_t 是一个向量，对 C_{t-1} 向量中每个元素值的传递进行控制。

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）

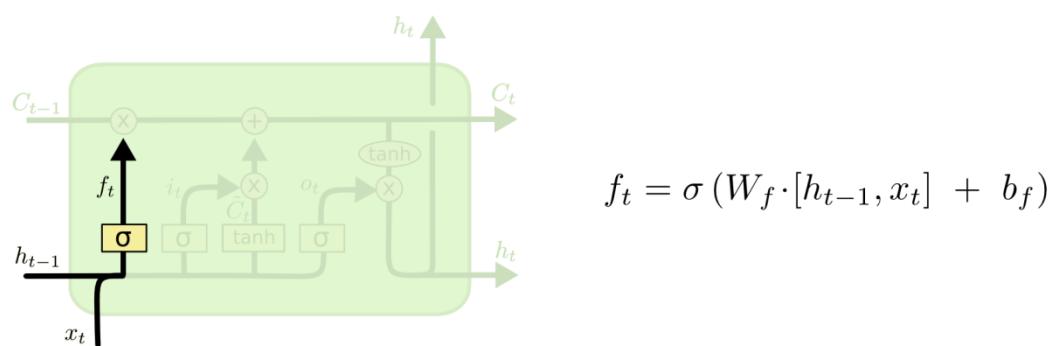


图 8.18 LSTM 块的第一层：一个 Sigmoid 层

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 $W_{fh} \cdot h_{t-1} + W_{fx} \cdot x_t$ 。也可以理解成 h_{t-1} 和 x_t 拼接后与权重矩阵 W_f 相乘。

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。

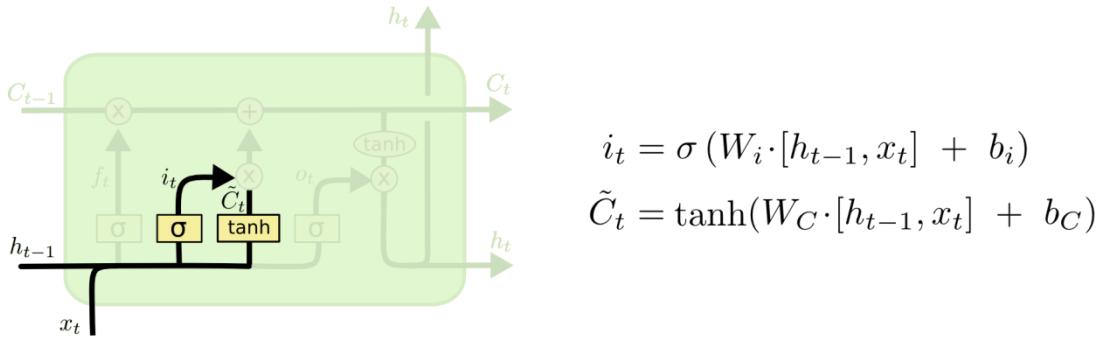


图 8.19 LSTM 块的第二，三层

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。

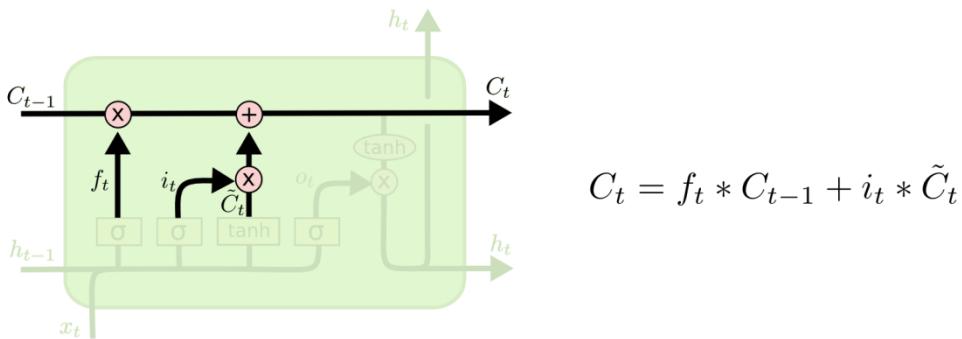


图 8.20 LSTM 块的第二，三层的输出

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。

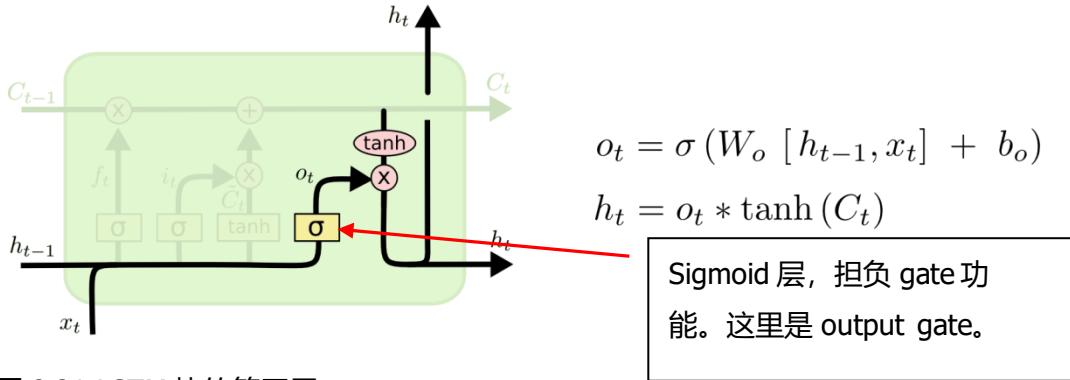


图 8.21 LSTM 块的第四层

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)

```

如果设置了 `return_sequence=True`, 上面的输出 `encoder_outputs` 是所有时间步的输出, `state_h` 是最后一个时间步的输出, 如果是 `return_sequence=False`, `encoder_outputs` 是最后一个时间步的输出, 和 `state_h` 一样。

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。

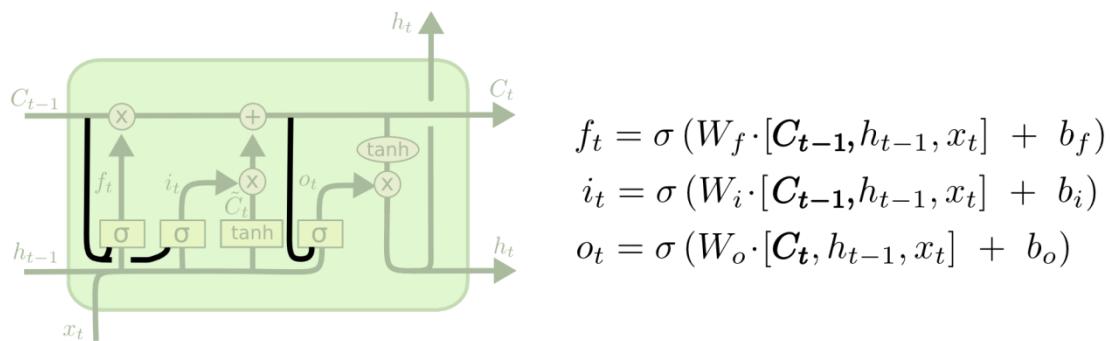


图 8.22 peephole connection LSTM

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。

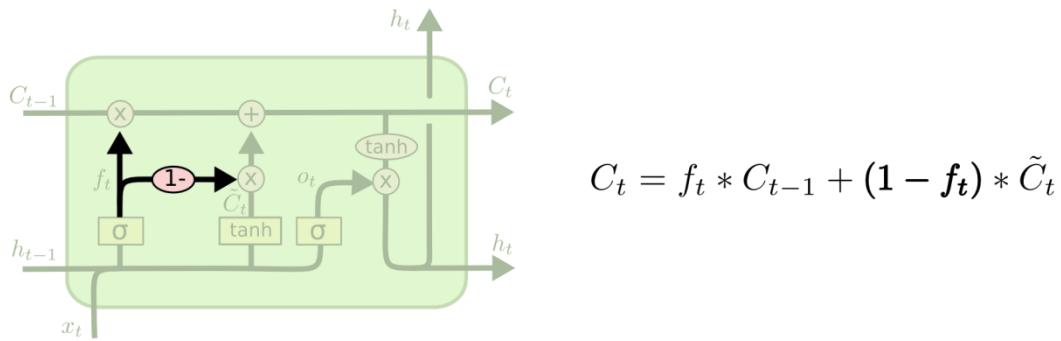


图 8.23 LSTM 的一个变体

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。

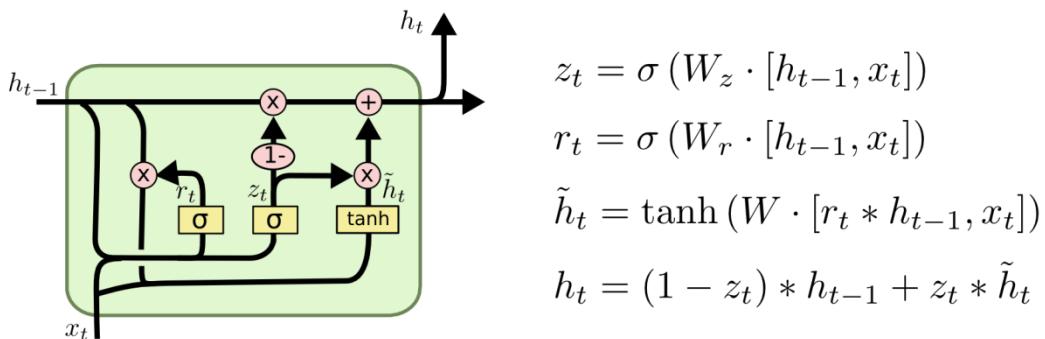


图 8.24 GRU

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 GRU 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

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. \(2015\)](#)。

第五节：基于 RNN 的时间序列预测模型

8.5.1 介绍

股票价格数据是一个时间序列数据。本节我们用 RNN 实施一个股票价格预测模型。见 `rnn-stock-price.py`。

8.5.2 数据集

有两个文件 `trainset.csv` 和 `testset.csv`。训练集包括从 2013 年到 2017 google 的股票价格数据，共有 1259 条。测试集包括 2018 年的 125 条股票价格数据。原始的每条股票价格数据包括：日期，开盘价、最高价、最低价、收盘价和成交量。数据如图 8.25 所示。

	Date	Open	High	Low	Close	Adj Close	Volume
0	2013-01-02	357.385559	361.151062	355.959839	359.288177	359.288177	5115500
1	2013-01-03	360.122742	363.600128	358.031342	359.496826	359.496826	4666500
2	2013-01-04	362.313507	368.339294	361.488861	366.600616	366.600616	5562800

图 8.25 Google 价格数据

本节仅是说明使用 RNN 如果做时间序列预测，不涉及股票价格的预测性能。因此，我们只选每天的开盘价。

8.5.3 模型结构

基于 RNN 的时间序列预测模型是根据前 n 个时刻的数据预测当前时刻的数据。

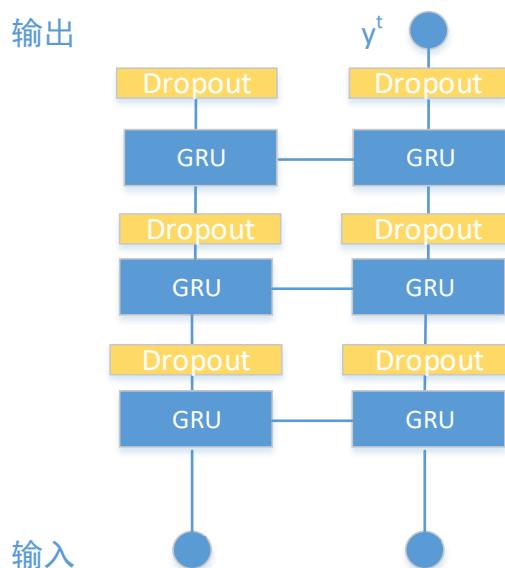


图 8.25 多层 GRU 模型

模型结构如图 8.25 所示。它是一个三层的 GRU 模型。最后一层只采用最后一个时间步的输出，经过一个全连接层输出。

数据预处理部分的程序如下：

```
dataset_train = pd.read_csv("google-stock/trainset.csv")
dataset_test = pd.read_csv("google-stock/testset.csv")
dataset_total = pd.concat((dataset_train['Open'],dataset_test['Open']),axis = 0)
real_stock_price = dataset_test['Open']

sc = MinMaxScaler(feature_range = (0,1))
dataset_total = dataset_total.values
dataset_total = dataset_total.reshape(-1,1)
dataset_scaled = sc.fit_transform(dataset_total)

trainset = dataset_scaled[:len(dataset_train)]
testset = dataset_scaled[-len(dataset_test)-60:]

x_train = []
y_train = []

for i in range(60,1259):
    x_train.append(trainset[i-60:i, 0])
    y_train.append(trainset[i,0])
```

```
x_train,y_train = np.array(x_train),np.array(y_train)
x_train = np.reshape(x_train, (x_train.shape[0],x_train.shape[1],1))
```

只读入开盘价的数据、训练集和测试集的数据一起标定到 0-1 之间。这里使用的是
from sklearn.preprocessing import MinMaxScaler

因为 scikit-learn 的规范化工具针对的数据是矩阵类型的，上面的 reshape 把向量转换成矩阵。在活动测试集 testset 时，因为我们是要用前 60 天的数据预测当天的。因此，给 testset 补充了前 60 天的数据。

在创建训练集时，使用每 60 个时间步的数据构建一个样本。

这个语句 x_train = np.reshape(x_train, (x_train.shape[0],x_train.shape[1],1)) 是在调整数据集的 shape 给它增加一个 channel 的维度，这是 RNN 模型规定的输入数据 shape。

模型构建代码如下：

```
regressor = Sequential()
regressor.add(GRU(units = 50,return_sequences = True,input_shape =
(x_train.shape[1],1)))
regressor.add(Dropout(0.2))
regressor.add(GRU(units = 50,return_sequences = True))
regressor.add(Dropout(0.2))
regressor.add(GRU(units = 50))
regressor.add(Dropout(0.2))
regressor.add(Dense(units = 1))
regressor.compile(optimizer = 'adam',loss = 'mean_squared_error')
regressor.fit(x_train,y_train,epochs = 100, batch_size = 32)
```

最后准备测试集的数据，并待入训练好的模型

```
x_test = []
for i in range(60,185):
    x_test.append(testset[i-60:i,0])

x_test = np.array(x_test)
x_test = np.reshape(x_test, (x_test.shape[0],x_test.shape[1],1))
predicted_price = regressor.predict(x_test)
predicted_price = sc.inverse_transform(predicted_price)
```

其中 predicted_price = sc.inverse_transform(predicted_price) 将预测的结果（也是在 0-1 之间的）再变换回原来的价格。绘制预测价格和真实价格如图 8.26 所示。

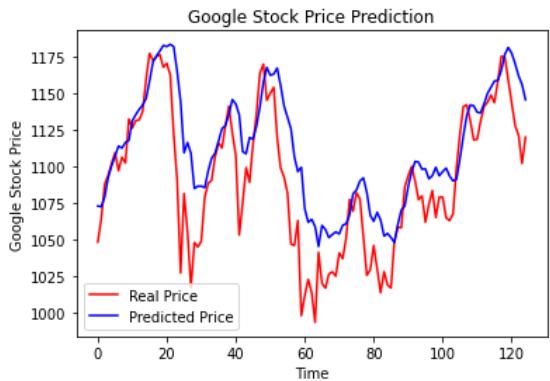


图 8.26 股票价格预测图

第六节：一些需要强调的问题

8.6.1 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。

8.6.2 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)), merge_mode="sum")
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`]。

上面这段代码, 用图来描述, 见图 8.27

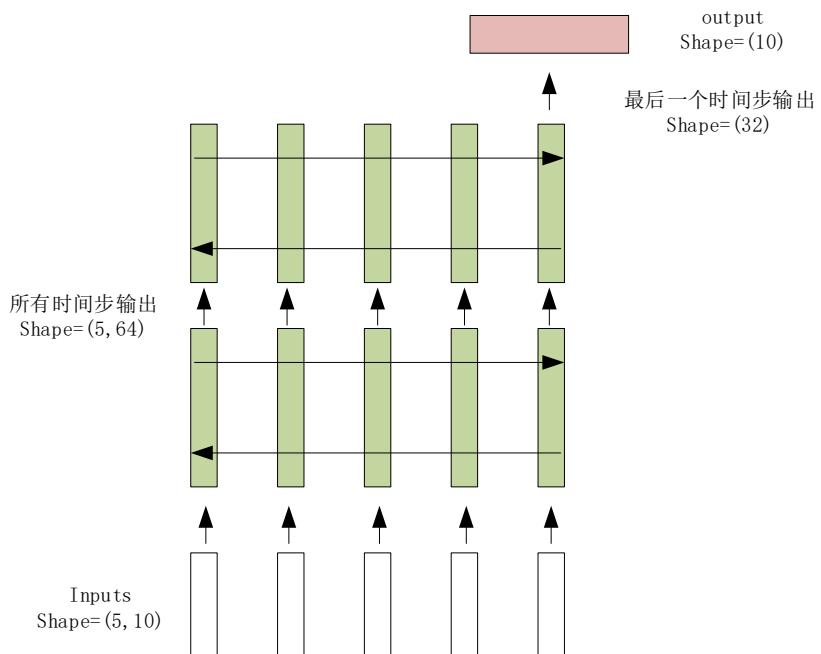


图 8.27 模型

8.6.3 RNN 训练技巧

参考下面的网址

<https://danijar.com/tips-for-training-recurrent-neural-networks/>

第九章：定制 RNN 与 RNN 文本情感分析

第一节：定制 RNN 层的方法

9.1.1 定制开发普通层

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

如果构建简单的层可以用 lambda 层，它可以用写 lambda 函数的方式，定制地对输入数据进行转换处理。9.3 节构建 mean pooling 层采用的就是 lambda 层来定制的。

9.1.2 定制开发 RNN 层

定制开发 RNN 层。核心是开发一个 cell，即完成一个时间步的计算的单元。然后将该 cell 作为 tf.keras.layers.RNN 的构造方法的参数，用于开发定制的 RNN 层。我们先了解 RNN 函数

1. RNN 函数

```
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.

2. Cell 类

开发定制的 RNN 层，其核心就是定制开发一个 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.

3. 示例

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

其中的代码 `self.kernel = self.add_weight (...)`

是在当前层添加 variable。variable 是 tensorflow 中的概念，即训练模型时，要学习的参数。`add_weight` 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
)

```

其中的 `kernel` 是 `cell` 内部层的权重；`recurrent_kernel` 是时间步之间传递的状态经过的全连接层权重。

在 `call` 方法里，返回两个 tensor，一个是当前时间步的输出，第二个是在时间步之间传递的 tensor。因为时间步中可以传递多个状态，因此用一个 list 结构来传递。本例中，在时间步之间传递的就是一个输出。因此 `call` 方法返回的第二个结果是 `[output]`。这样也可以理解为什么 `call` 方法的参数中的第二个参数是状态 `state`，在 `call` 方法里有一条语句 `prev_output = states[0]`。

另外，如果要创建一个包含多个堆叠的 cell 的 RNN。需要使用类 `tf.keras.layers.StackedRNNCells` 来创建一个堆叠的 cell。再将该 cell 作为 RNN 的参数。

```
rnn_cells = [tf.keras.layers.LSTMCell(128) for _ in range(2)]
stacked_lstm = tf.keras.layers.StackedRNNCells(rnn_cells)
lstm_layer = tf.keras.layers.RNN(stacked_lstm)
```

首先创建一个 cell 对象的列表 `rnn_cells`。然后创建堆叠的 cell 对象 `stacked_lstm`。再使用它作为创建 RNN 时的参数。如此进阿里的 RNN 模型是一个由 2 个 lstm cell 的 LSTM 模型。

第二节：基于定制 RNN 的活动识别

在 5.6 节我们介绍了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。然后开发了 1D CNN 模型来识别人类活动。每条数据描述一个人的活动，是一个时间序列数据，包括 128 个时间步；（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。因此我们也可以开发一个定制 RNN 来进行活动识别。

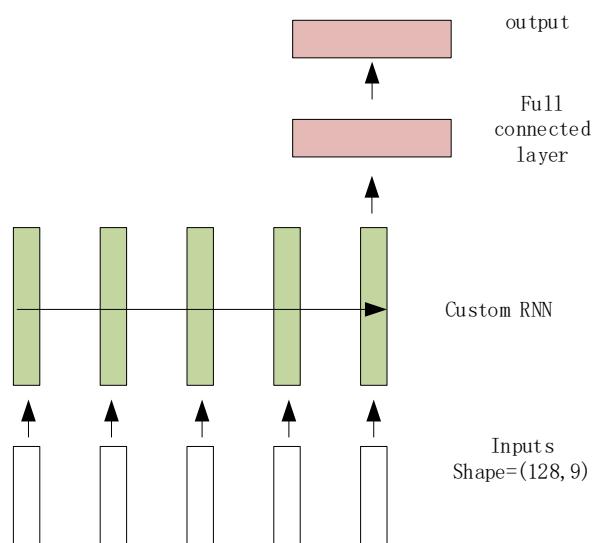


图 9.1 活动识别模型

详细程序见 `rnn-activity-customcell.py`。除了装载数据的程序部分，建模的部分见下面的代码。定制的 RNN 层，它的 cell 就是 9.1 节的 cell。

```
# hyper-parameters
verbose, epochs, batch_size = 2, 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]

# custome cell
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]

model = Sequential()
model.add(Input(shape=(n_timesteps,n_features)))
cell = MinimalRNNCell(128)
model.add(RNN(cell))
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))

```

第三节：基于 RNN 的文本情感分析

<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。

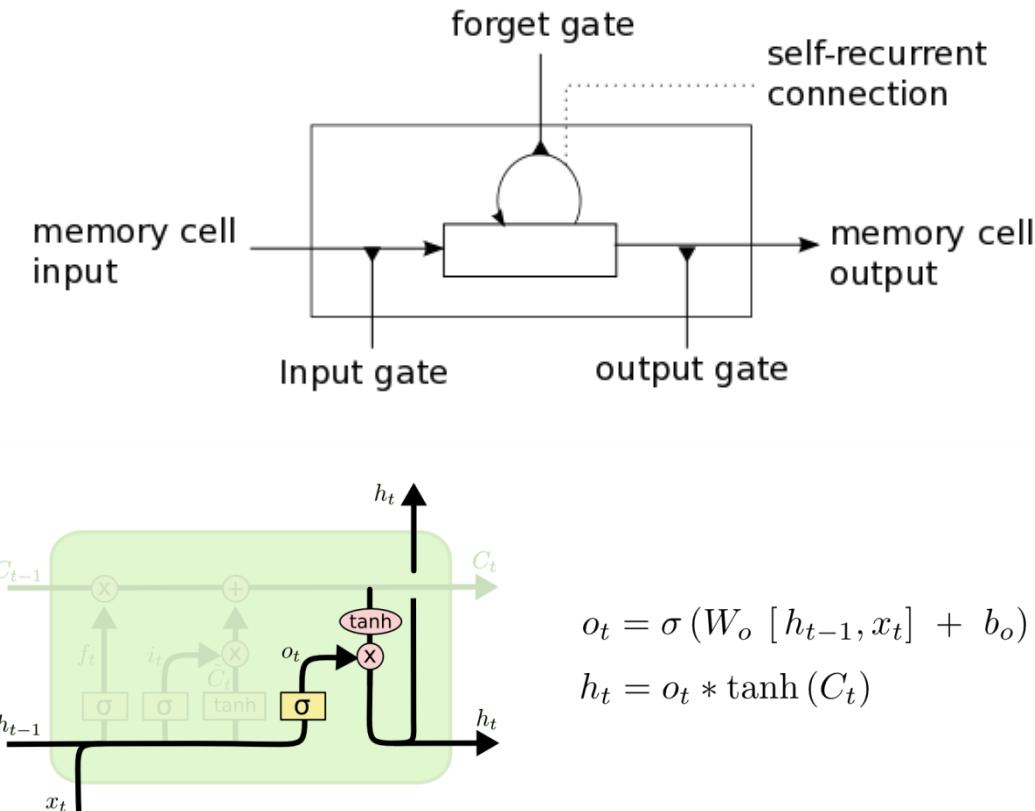


图 9.1 简化版的 LSTM

该文的模型如图 9.2 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。

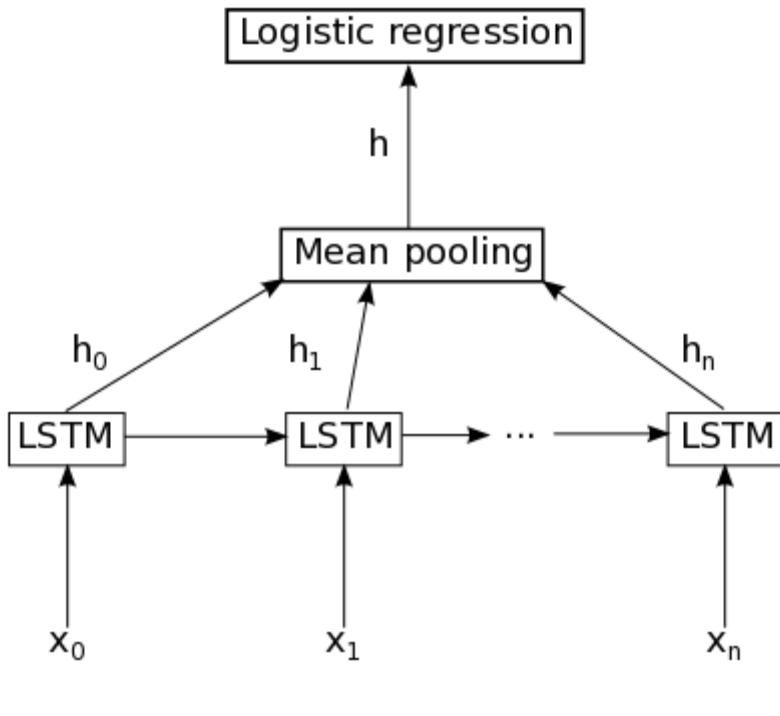


图 9.2 一个评论数据情感分类的 RNN 模型

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。

第四节：文本情感分析的 keras 实施

在介绍 keras 的文本情感分类模型前，我们先介绍使用 `tf.data.Dataset`。在训练模型时，数据集很大时，建议使用 `tf.data.Dataset`。`tf.data.Dataset` 是用于创建输入数据 pipeline 的 API。它的内容很丰富，我们不详细讨论。当我们需要训练模型的数据集很大时，可以使用该类，它不会一次把数据读入内存。而是采用流的方式。

9.4.1 `tf.data.Dataset`

如果只是创建 `model` 的 `fit` 和 `evaluate` 函数需要喂入的数据集，它的使用很简单，例如

```

train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train)).batch(50)
test_dataset = tf.data.Dataset.from_tensor_slices((x_dev, y_dev)).batch(50)

model.fit(train_dataset, epochs=epochs, verbose=verbose)
loss, acc, myacc = model.evaluate(test_dataset, verbose=verbose)

```

`x_train, y_train` 就是准备好的数据集的 numpy 的 ndarray 数据结构。然后用 `from_tensor_slices` 方法，把特征数据 `x_train` 和目标数据 `y_train` 组合成元组来创建。后面的 `batch(50)` 设定 batch 大小为 50（用户自己修改）。

此时在 `fit` 函数中，不要再设置 `batch_size` 和 `validation_split` 参数。如果需要校验数据，则自己准备好，在 `fit` 函数中，放入校验数据。下面是一个例子

```
import tensorflow as tf
from tensorflow import keras

def get_compiled_model():
    inputs = keras.Input(shape=(784,))
    x = keras.layers.Dense(256, activation="relu")(inputs)
    x = keras.layers.Dense(256, activation="relu")(x)
    outputs = keras.layers.Dense(10)(x)
    model = keras.Model(inputs, outputs)
    model.compile(
        optimizer=keras.optimizers.Adam(),
        loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=[keras.metrics.SparseCategoricalAccuracy()],
    )
    return model

def get_dataset():
    batch_size = 32
    num_val_samples = 10000

    (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

    x_train = x_train.reshape(-1, 784).astype("float32") / 255
    x_test = x_test.reshape(-1, 784).astype("float32") / 255
    y_train = y_train.astype("float32")
    y_test = y_test.astype("float32")

    x_val = x_train[-num_val_samples:]
    y_val = y_train[-num_val_samples:]
    x_train = x_train[:-num_val_samples]
    y_train = y_train[:-num_val_samples]
    return (
        tf.data.Dataset.from_tensor_slices((x_train, y_train)).batch(batch_size),
        tf.data.Dataset.from_tensor_slices((x_val, y_val)).batch(batch_size),
        tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(batch_size),
    )

strategy = tf.distribute.MirroredStrategy()
print("Number of devices: {}".format(strategy.num_replicas_in_sync))

with strategy.scope():
    model = get_compiled_model()

train_dataset, val_dataset, test_dataset = get_dataset()
model.fit(train_dataset, epochs=2, validation_data=val_dataset)

model.evaluate(test_dataset)
```

其中的代码 `strategy = tf.distribute.MirroredStrategy()` 是用于多 GPU 训练时采用的，详细参见附录 C。

9.4.2 情感分类模型

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。使用 9.4.1 节的 `tf.data.Dataset` 数据集模式。下面，讲述核心的代码。完整的源码参见 `lstm-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())

train_dataset=tf.data.Dataset.from_tensor_slices((x_train,
y_train)).batch(batch_size)
test_dataset=tf.data.Dataset.from_tensor_slices((x_dev,
y_dev)).batch(batch_size)
model.fit(train_dataset, epochs=epoch, verbose=1)

# evaluate the model
loss, accuracy = model.evaluate(test_dataset, verbose=0)
print('Accuracy: %f' % (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)。函数 mean

```
from tensorflow.keras.backend import mean
```

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)
```

第五节：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。

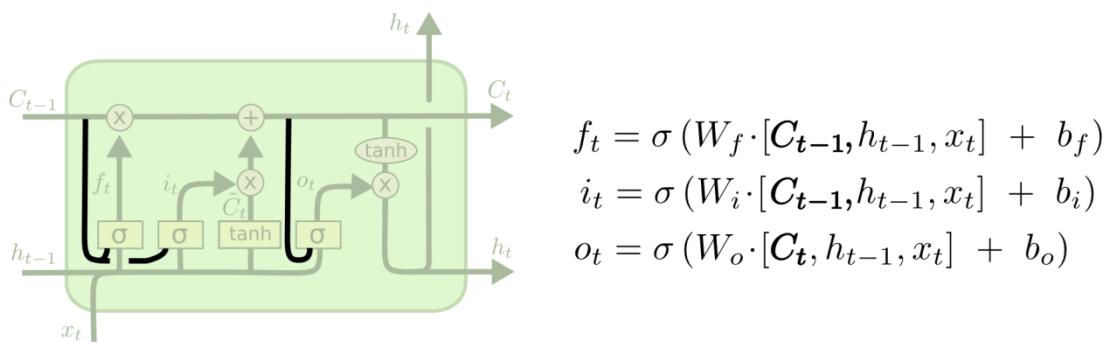


图 9.3 peephole connection LSTM

本节将开发这个 peephole LSTM。除了图 9.3 显示的三个门的计算和 LSTM 不一样， 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。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 peepholecell.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))

```

我们用 peephole 的 LSTM cell 实施了两种情感分类模型。

(1) 第一种就是把第二节实施的基于 LSTM 的文本分类模型，用 peepholecell 构建的 LSTM，替代经典 LSTM，并使用了双向 LSTM，具体参见文件 peephole-sentiment.py。

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，7 留给读者自己完成。

(2) 我们再开发了第二种模型。模型结构见图 9.4，代码参见文件 lstm-sentiment-twoblock.py。首先构建了两种 embedding 层，一种是随机初始化，一种是用预训练的 embeddings。但和前面使用预训练的 embedding 时，embedding 层的值在训练模型时不调整，见下面的代码。

```

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

```

这个模型将 `trainable` 设为 `True`，即可调整的。

两种 LSTM，只使用最后一个时间步的输出，然后进行拼接成一个向量，再接全连接层和输出层。

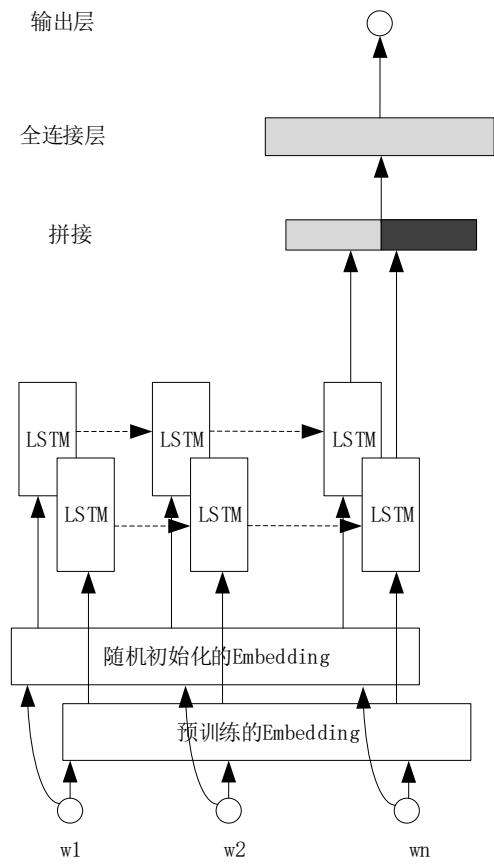


图 9.4 一个两种 LSTM 组合的分类模型

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 版和预印版。预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。

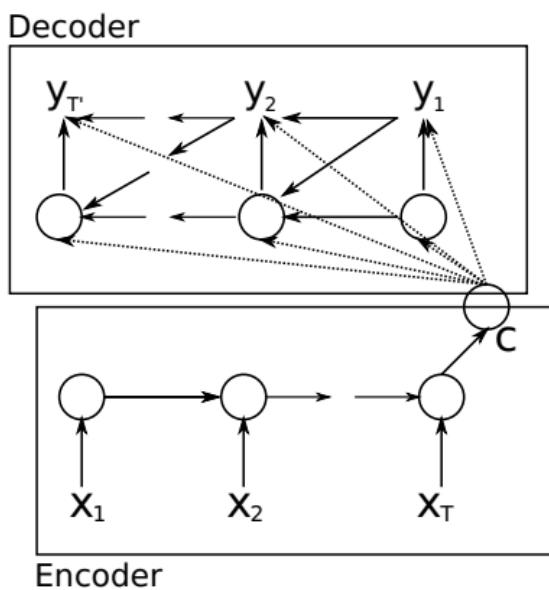


图 10.1 一个 RNN Encoder-Decoder 模型

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。

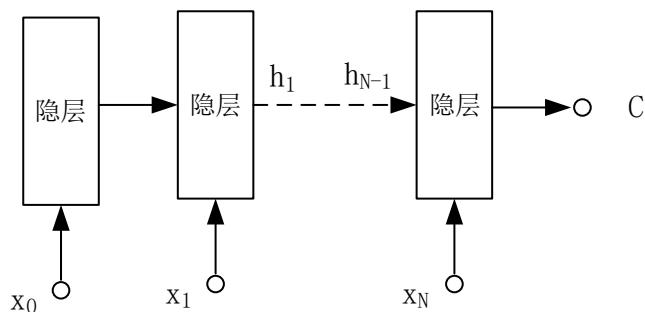


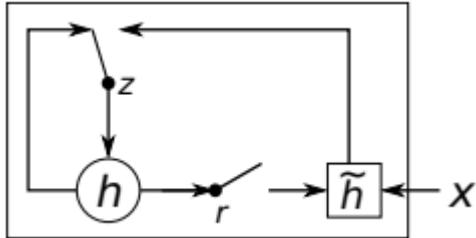
图 10.2 Encoder 结构

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 一个非线性函数, 即可以是一个简单的全连接层, 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 cell, 如图所示。



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 的计算如下

$$h_t = zh_{t-1} + (1 - z)\tilde{h}_t$$

其中

$$\begin{aligned}\tilde{h}_t &= \tanh(\mathbf{W}e(x_t) + \mathbf{U}(r \odot h_{t-1})) \\ z &= \sigma(\mathbf{W}_z e(x_t) + \mathbf{U}_z h_{t-1}) \\ r &= \sigma(\mathbf{W}_r e(x_t) + \mathbf{U}_r h_{t-1})\end{aligned}$$

$e(x_t)$ 是词 x_t 对应的词向量。 σ 是一个 sigmoid 激活函数。 \odot 是一个逐个元素相乘运算符。计算 h_0 时初始隐状态设为 0。编码器的输出是

$$c = \tanh(\mathbf{V}h_N)$$

矩阵 $\mathbf{W}, \mathbf{U}, \mathbf{V}$ 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

$$h'_t = f(h'_{t-1}, y_{t-1}, c)$$

符号' 用于表示和 Encoder RNN 中的参数有区分。Decoder 输出序列的第 t 个元素（例如一个词）的产生是一个条件分布

$$P(y_t | y_{t-1}, y_{t-2}, \dots, y_1, c) = g(h'_t, y_{t-1}, c)$$

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

$$h_0 = \tanh(\mathbf{V}'c)$$

在时间步 t 的输出的计算如下：

$$\mathbf{h}'_t = \mathbf{z}'\mathbf{h}'_{t-1} + (1 - \mathbf{z}')\tilde{\mathbf{h}}'_t$$

其中

$$\tilde{\mathbf{h}}'_t = \tanh(\mathbf{W}'e(\mathbf{y}_{t-1}) + \mathbf{r}'\odot(\mathbf{U}'\mathbf{h}'_{t-1} + \mathbf{C}\mathbf{c}))$$

$$\mathbf{z}' = \sigma(\mathbf{W}'_z e(\mathbf{y}_{t-1}) + \mathbf{U}'_z \mathbf{h}'_{t-1} + \mathbf{C}_z \mathbf{c})$$

$$\mathbf{r}' = \sigma(\mathbf{W}'_r e(\mathbf{y}_{t-1}) + \mathbf{U}'_r \mathbf{h}'_{t-1} + \mathbf{C}_r \mathbf{c})$$

$e(\mathbf{y}_{t-1})$ 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

$$P(y_{t,j} = 1 | y_{t-1}, y_{t-2}, \dots, y_1, X) = \frac{\exp(G_j s_t)}{\sum_{i=1}^K \exp(G_i s_t)}$$

X 表示词汇表， $G \in R^{K \times d}$ 是权重矩阵， K 是词汇表的大小， d 是向量 s 的维度。 G_j 是权重矩阵的一行。 s_t 的计算如下

$$s_t = \max\{s'_{t,2i-1}, s'_{t,2i}\}$$

这个操作叫 max-out，就像是 pooling 操作，输出的 s_t 的向量长度是 s'_t 的一半。

其中

$$s'_t = \mathbf{O}_h h'_t + \mathbf{O}_y e(\mathbf{y}_{t-1}) + \mathbf{O}_c \mathbf{c}$$

可以看到，Decoder 在计算输出序列的一个词时，采用了解码器当前时间步的隐状态 h'_t ，序列上一个词 y_{t-1} 的表示向量和 Encoder 输出的 context \mathbf{c} 。

第二节：实例：实现一个翻译模型

本节我们建立一个字符级的英-中翻译模型。详细代码见文件 encoder-decoder-translation-char.py。

要得到一个翻译模型，它包含两个阶段：一个是训练阶段；一个是推断阶段。第一阶段建立的模型结构与第二阶段的不完全一致。而是推断阶段使用第一阶段训练好的模型组件，搭建一个可以进行翻译的模型。下图是一个训练阶段的模型结构。

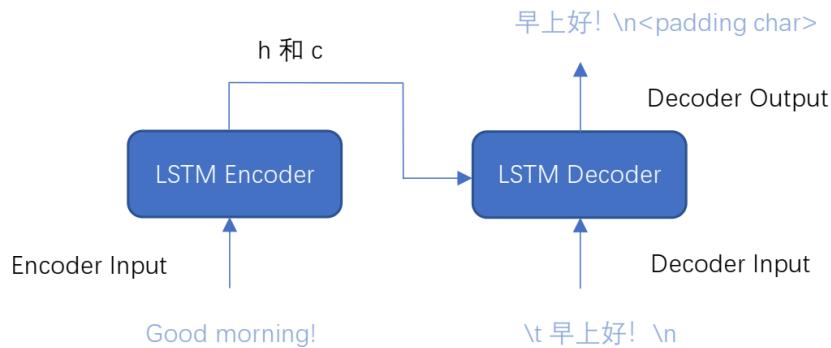


图 10.3 训练阶段的 Encoder-Decoder 翻译模型

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（英文字符加符号）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中文字符或符号。本节的翻译模型很简单，只为说明怎样使用 Encoder-Decoder 模型。

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。本例子中，使用一个英-中数据集。

数据集被处理成三个部分：Encoder 的输入 encoder_inputs，Decoder 的输入 decoder_inputs 和 Decoder 的输出 decoder_target。

Encoder 的输入数据将英文文本按照字符序列来处理。每个字符用 one-hot 编码。因此 encoder_inputs 是一个 3d 的 tensor，它的 shape 是 [样本数, 最长序列长度, 英文字符表大小]。每个英文样本按照最长样本字符长度用空格符进行补齐。

Decoder 的输入是将中文文本按照字符序列来处理。同样每个字符用 one-hot 编码。在输入的中文文本补充一个开始字符'\t'和一个结束符'\n'，同样每个中文文样本按照最长样本字符长度用空格符进行补齐。

Decoder 是一个语言模型，因此 Decoder 的输出的目标数据是 Decoder 输入数据的左移一位。见上图。

2. 建立模型和训练模型

模型包括 Encoder 和 Decoder。其中 Encoder 就是一个 LSTM 模型，我们只需获取最后一个时间步的输出。**Decoder 在训练阶段是按照一个语言模型来训练**。即输入是一

个字符的序列，目标数据是错开一位的输入数据。例如，输入是 ‘hello’，目标数据是 ‘ello<结束符>’。

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]

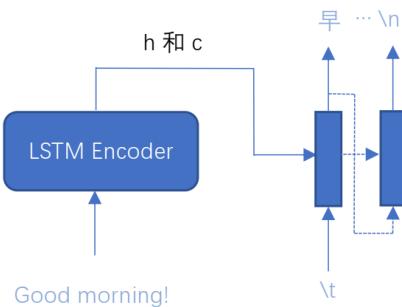
decoder_inputs = Input(shape=(None, num_decoder_tokens))
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
                                     initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
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) LSTM 每个时间步产生一个状态 c 和一个输出 h。我们说，基于 LSTM 的 encoder 将输入序列编码成两个向量 h 和 c。

(2) 在 Decoder，接收 Encoder 传递的向量 h 和 c 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，本文我们用的是字符'\t'。Decoder 将预测一个输出字符。

(4) 将预测的字符再喂入 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，本文我们用的是'\n'作为结束符，或者达到了设定的序列长度。

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. # one-hot 编码起始字符'\t'
    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. #one-hot 编码上一个预测的字
    符, 作为下一个时间步的输入
    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: Try it.

Decoded sentence: 试试吧。

Input sentence: We try.

Decoded sentence: 我们来试试。

Input sentence: Why me?

Decoded sentence: 为什么是我?

Input sentence: Ask Tom.

Decoded sentence: 去问汤姆。

Input sentence: Awesome!

Decoded sentence: 好棒！

可以看到有些翻译的可以，有些还是不准。模型还是有很大的改善的空间。

练习：

将上面的模型英文编码器部分改成 word 级的，解码器部分按照本章第一节介绍的论文中的 Decoder 来实施。

示例代码见 encoder-decoder-translation-word.py

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 。

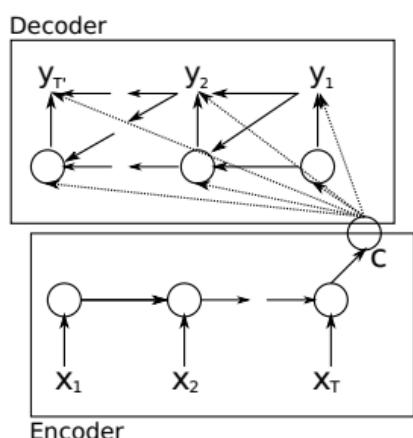


图 11.1 一个基本的 Encoder-Decoder 模型

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 向量 c 和已经输出的词的序列。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”。

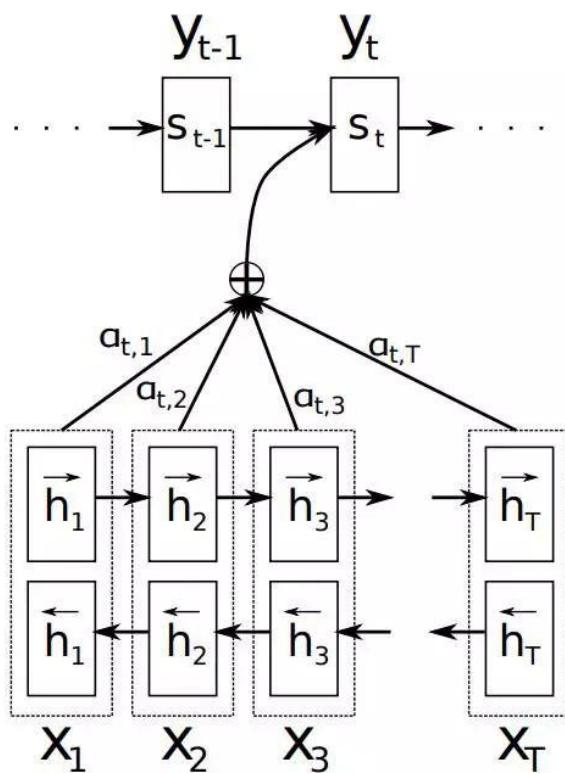


图 11.2 加入了注意力机制的 seq2seq 模型

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 α 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(alignment)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。对齐模型有多种形式的计算，参考 11.1.2 节。在原论文中，alignment model a 是一个前馈神经网络。

解码器 RNN 中一个时间步 i 的输出 s_i 的计算

$$s_i = f(s_{i-1}, y_{i-1}, c_i)$$

这里把解码器时间步 i 输出的词 y_i 的计算过程补充完整。其计算和 10.1 节最后部分的计算是相似的，除了 context 向量在 10.1 节是固定的，而注意力机制中解码器每个时间步计算的 c_i 是不同的。

$$p(y_i | s_i, y_{i-1}, c_i) \propto \exp(y_i^T W_o t_i)$$

其含义就是通过 softmax 输出层计算每个词 y_i 的概率。其中

$$t_i = [\max\{\tilde{t}_{i,2j-1}, \tilde{t}_{i,2j}\}]_{j=1, \dots, l}^T$$

这一步的 max 操作就是 max pooling 操作。 $\tilde{t}_{i,k}$ 是向量 \tilde{t}_i 第 k 个元素，计算公式如下

$$\tilde{t}_i = U_0 s_i + V_0 e(y_{i-1}) + C_0 c_i$$

即解码器的 RNN 中上一个时间步的输出 s_{i-1} ，输出的词 y_{i-1} 和当前时间步计算的注意力 context 向量 c_i 都参与到当前输出词 y_i 的计算。 $e(y_{i-1})$ 表示从词向量查找表 E 中计算当前词 y_{i-1} 对应的词向量。

对于解码器中一个时间步 s_i 的计算

$$s_i = (1 - z_i) \odot s_{i-1} + z_i \odot \tilde{s}_i$$

其中

$$\begin{aligned}\tilde{s}_i &= \tanh(\mathbf{W}e(y_{i-1}) + \mathbf{U}[r_i \odot s_{i-1}] + \mathbf{C}c_i) \\ z_i &= \sigma(\mathbf{W}_z e(y_{i-1}) + \mathbf{U}_z s_{i-1} + \mathbf{C}_z c_i) \\ r_i &= \sigma(\mathbf{W}_r e(y_{i-1}) + \mathbf{U}_r s_{i-1} + \mathbf{C}_r c_i)\end{aligned}$$

○是哈达玛积，一个逐个元素相乘运算符。

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 一个时间步的隐状态和 Decoder 一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	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \mathbf{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.
General	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s}_t^\top \mathbf{W}_a \mathbf{h}_i$ 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(^)	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \frac{\mathbf{s}_t^\top \mathbf{h}_i}{\sqrt{n}}$ 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。

本文实施了一个加入注意力机制的英-中翻译模型，参见附录H。

第二节：注意力机制的类型

除了第一节讲的在 Encoder-Decoder 之间的注意力机制，还有很多注意力机制类型。

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，摘要任务中很多采用自注意力机制[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 sentence 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。

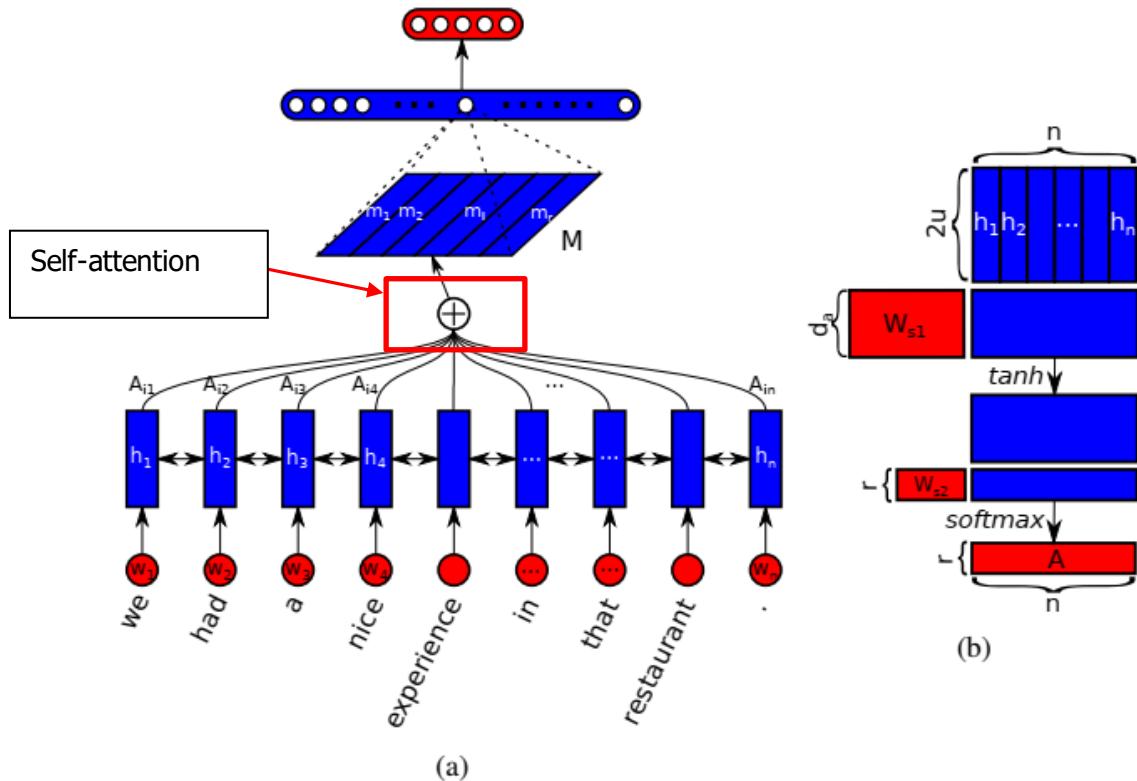


图 11.3 一个自注意力机制的 sentence embedding 模型. (a) 整个模型; (b) 注意力机制得到权重矩阵 A

现在有个句子 S, 它有 n 个词项, 以词向量的序列来描述。

$$S = (\mathbf{w}_1, \dots, \mathbf{w}_n)$$

\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 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。

$$\begin{aligned}\vec{h}_t &= \overrightarrow{\text{LSTM}}(\mathbf{w}_t, \vec{h}_{t-1}) \\ \hat{h}_t &= \overleftarrow{\text{LSTM}}(\mathbf{w}_t, \hat{h}_{t+1})\end{aligned}$$

然后将 \vec{h}_t 和 \hat{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 shape=[n, 2u]。

该文的目标是把一个长度可变的输入词序列编码成一个固定长度的向量。因此在 H 上采用 self-attention 机制来完成计算。

$$\mathbf{a} = \text{softmax}(\mathbf{w}_{s2} \tanh(W_{s1} H^T))$$

W_{s1} 是权重矩阵, 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 \mathbf{a} 的长度是 n。然后 H 按照向量 \mathbf{a} 提供的权重求和, 如此得到一个长度为 2u 的向量 \mathbf{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 \mathbf{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 \mathbf{m} 。因此, 将 W_{s2} 扩展成一个 shape=[r, d_a] 的矩阵。因此注意力机制得到 shape=[r, n] 的权重矩阵 A

$$A = \text{softmax}(W_{s2} \tanh(W_{s1} H^T))$$

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 shape=[r, 2u] 的矩阵 M

$$M = AH$$

(注: 图 11.3 的矩阵 M 按照列向量来显示, shape=[2u, r]。与上面的计算不冲突)

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 时，编码器的所有时间步的隐层输出都要参与计算。

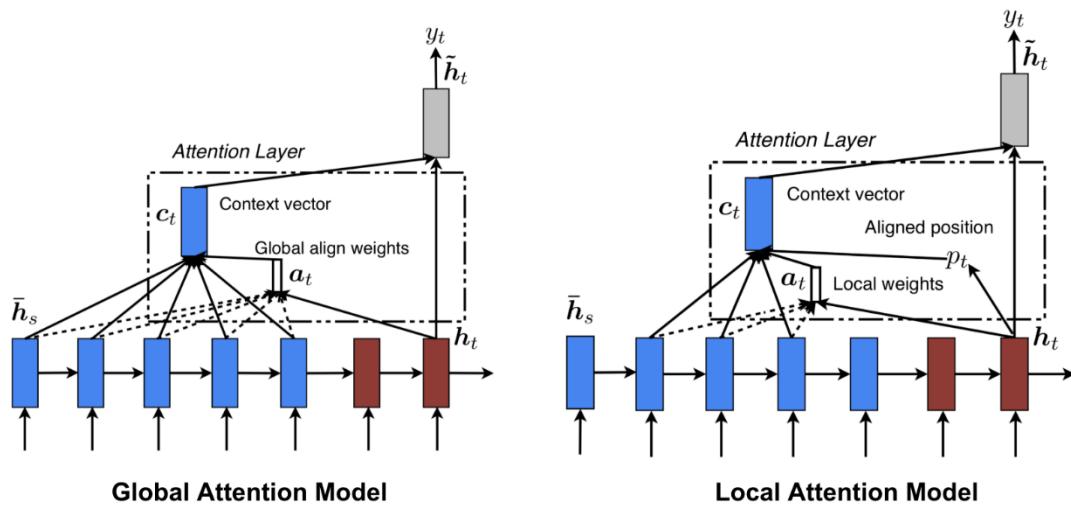


图 11.4 Global 和 Local Attention 模型

一个长度可变的对齐向量（alignment vector） a_t （注意力权重向量），它的长度等于 Encoder 输入的时间步的数量（输入序列的长度）。 a_t 中一个注意力权重的计算公式如下：

$$\begin{aligned}\mathbf{a}_t(s) &= \text{align}(\mathbf{h}_t, \bar{\mathbf{h}}_s) \\ &= \frac{\exp(\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_s))}{\sum_{s'} \exp(\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_{s'}))}\end{aligned}$$

11.1.1 节中讲的注意力机制和 Global Attention 机制，在注意力计算的原理上是相同的。不同的是此处计算路径是 $\mathbf{h}_t \rightarrow \mathbf{a}_t \rightarrow \mathbf{c}_t \rightarrow \tilde{\mathbf{h}}_t$ 。而 11.1.1 节的计算是 $\mathbf{h}_{t-1} \rightarrow \mathbf{a}_t \rightarrow \mathbf{c}_t \rightarrow \mathbf{h}_t$ (11.1.1 节的 s_t 就是此处的 \mathbf{h}_t)。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。Local Attention 的内容具体参考该论文。

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。我们将层次注意力模型应用在了对电商评论数据进行编码（将一条评论数据转换成一个向量，以应用于下游任务）。网络结构如图 11.5 所示。

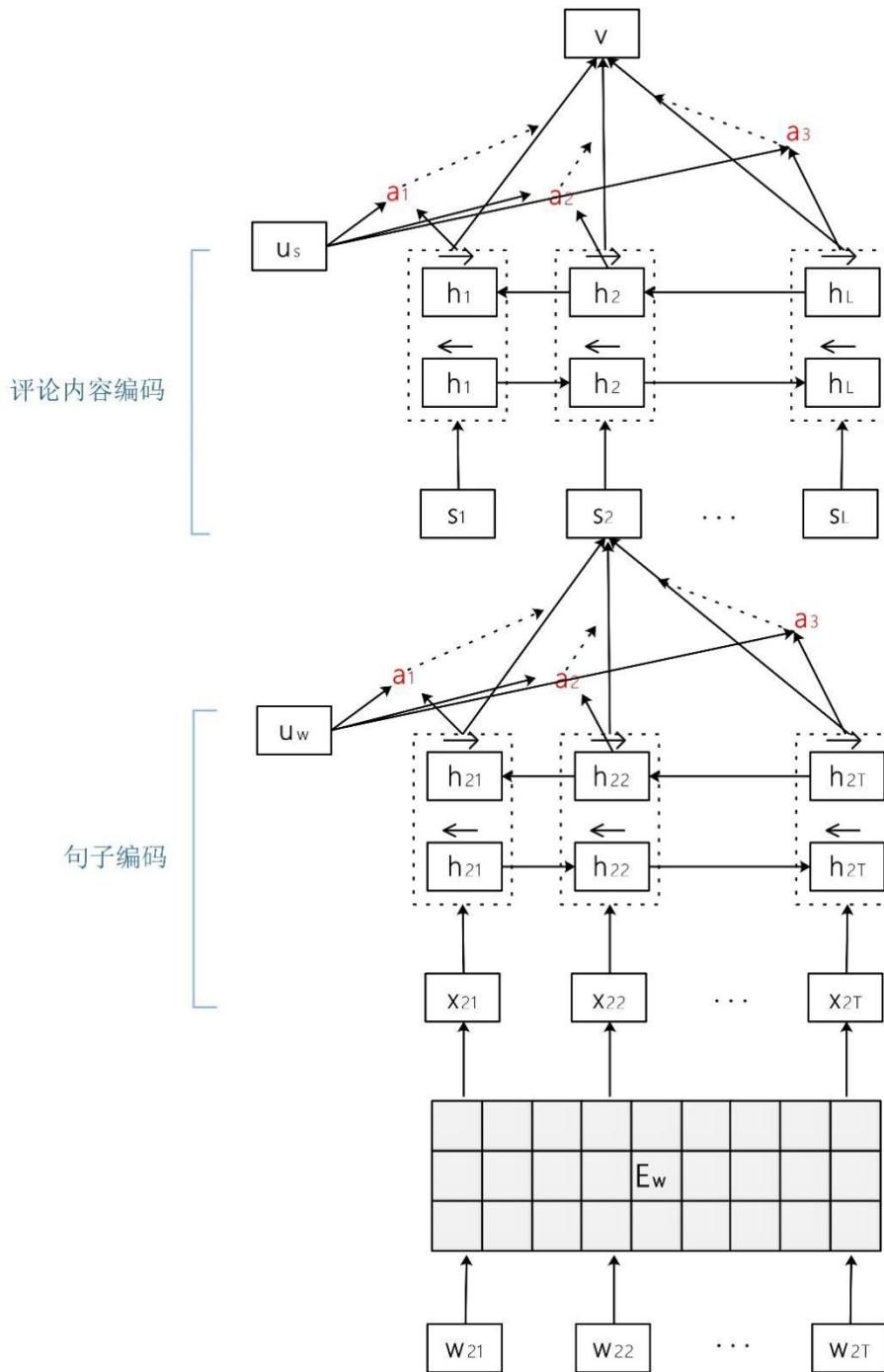


图 11.5 层次注意力模型

为每条评论进行编码，得到一个表示向量，这个过程称为评论内容特征抽取。因为在线评论的内容较为丰富，我们采用层次注意力模型来获得评论内容的表示向量。模型包括句子级的注意力机制和词级的注意力机制。送入模型的是评论内容， s_l 是对一个句子编码后得到的表示向量， v 是最终对评论进行编码得到的向量。

1. 句子编码：设一条评论 n_i 包含 L 条句子，其中的句子 l 包括 T 个词，是一个词的序列 $w_l = (w_{l1}, \dots, w_{lT})$ 。首先通过词嵌入矩阵 $E_w \in R^{m \times d_1}$ 得到一个词 w_{lt} 的表示向量 x_{lt} 。然

后将句子 l 的词向量序列 $x_l = (x_{l1}, \dots, x_{lT})$ 送入双向 GRU 网络[1]。其中的一个时间步 t 的计算如式 (1,2) 所示

$$\vec{h}_t = \overrightarrow{\text{GRU}}(x_{lt}), t \in [1, T] \quad (1)$$

$$\overleftarrow{h}_t = \overleftarrow{\text{GRU}}(x_{lt}), t \in [1, T] \quad (2)$$

得到两个隐状态 $\vec{h}_t, \overleftarrow{h}_t$ 。再拼接两个隐状态得到双向 GRU 时间步 t 的输出 $h_t = [\vec{h}_t, \overleftarrow{h}_t]$ 。

由 GRU 的输出产生句子 l 的表示向量 s_l 时，我们采用了词级的自注意力机制。在评论的一条句子中，每个词对当前句子的含义有不同权重的影响。我们使用自注意力机制去获取每个词的注意力权重，然后再通过加权求和获得整个句子的表示向量 s_l 。对于一个时间步 t 具体做法如下：首先将 h_t 送入一个多层感知机得到 u_t ，如式 (3) 所示。然后计算与向量 u_w 的内积，这里 u_w 是一个随机初始化的向量也是待学习的参数。再通过一个 softmax 函数得到每个时间步输出的权重 a_t 如式 (4) 所示 (u'_t 是向量 u_t 的转置)。最后加权求和每个时间步输出得到句子 l 的表示向量 s_l 。

$$u_t = \tanh(W_w h_t + b_w) \quad (3)$$

$$a_t = \frac{\exp(u'_t u_w)}{\sum_{i \in T} \exp(u'_i u_w)} \quad (4)$$

$$s_l = \sum_{t \in T} a_t h_t \quad (5)$$

2. 评论内容编码：评论 n_i 包含 L 条句子，在通过上面的句子编码步骤为每个句子建立表示向量后，得到评论 n_i 的句子表示向量集合 $s = (s_1, \dots, s_L)$ 。我们使用另外一个双向 GRU 网络对句子向量集合进行编码。在时间步 l 的计算见公式 (6) 和 (7)

$$\vec{h}_l = \overrightarrow{\text{GRU}}(s_l), l \in [1, L] \quad (6)$$

$$\overleftarrow{h}_l = \overleftarrow{\text{GRU}}(s_l), l \in [1, L] \quad (7)$$

然后，拼接双向 GRU 的两个隐状态，得到 $h_l = [\vec{h}_l, \overleftarrow{h}_l]$ 。设向量 h_l 的长度是 u 。

在由评论内容编码的 GRU 输出产生评论的表示向量 v 时。评论中不同的句子对整个评论的含义具有不同的权重。因此，我们采用了句子级注意力机制，获得每个句子的注意力权重。注意力计算方法见公式 (9) 和 (10)。这里 u_s 是一个随机初始化的向量也是待学习的参数。然后再加权求和每个时间步的输出，获得评论的表示向量 v 。

$$u_l = \tanh(W_s h_l + b_s) \quad (8)$$

$$a_l = \frac{\exp(u'_l u_s)}{\sum_{i \in L} \exp(u'_i u_s)} \quad (9)$$

$$v = \sum_{l \in L} a_l h_l \quad (10)$$

最终，通过层次注意力机制我们获得评论的表示向量 $v \in R^u$ 。

11.2.5 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。

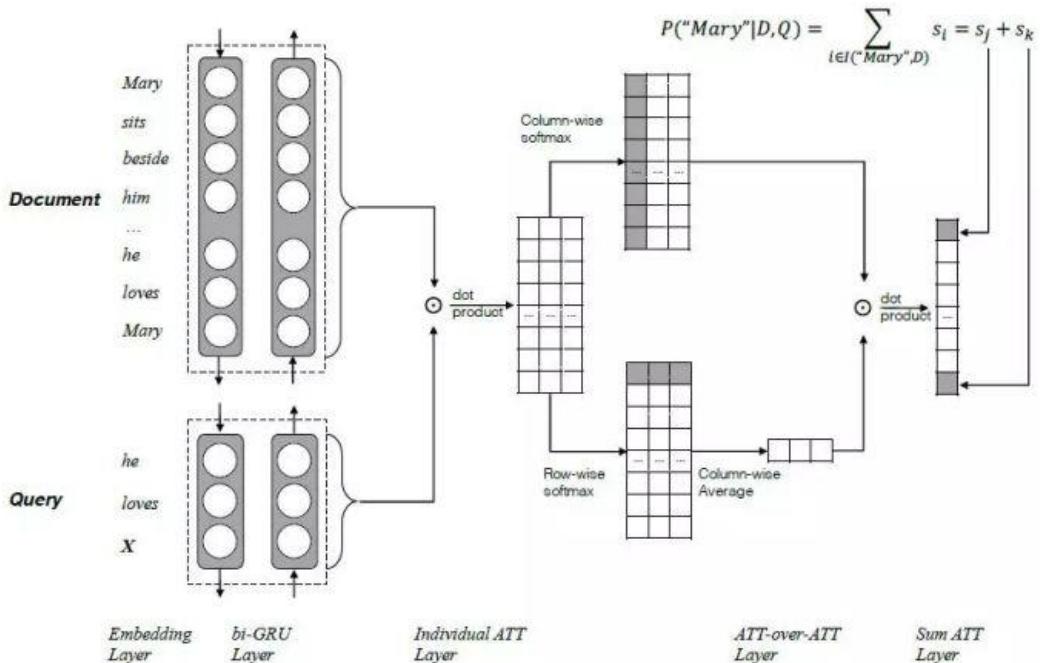


图 11.6

11.2.6 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，有实验证明，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。

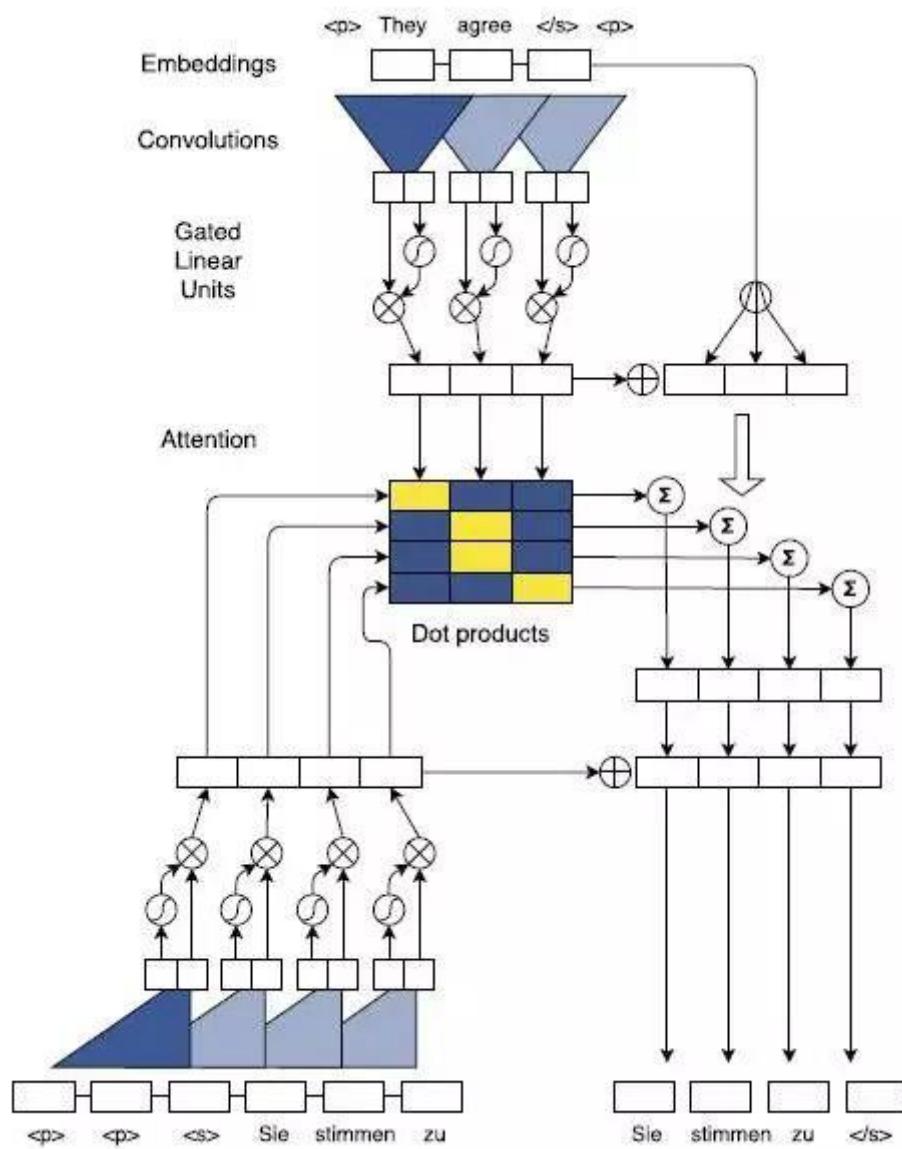


图 11.7

11.2.7 Multi-dimensional Attention

《Coupled Multi-Layer Attentions for Co-Extraction of Aspect and Opinion Terms》

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4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矩阵。函数中针对这个矩阵进行正则项计算，例如

```
def l1_reg(matrix):
    return 0.01 * K.sum(K.abs(matrix))
```

在一个层设置正则项时，给出该函数即可。例如下面的代码是为一个层的输出设置正则项。

```
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，该方法的参数对应一个矩阵。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.sqrt(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)
```

该例子为层 layer2 的输出设置可调节参数的正则项。

11.3.2 定制损失函数

定义损失函数必须有两个参数 y_{true} 和 y_{pred} 。定制的损失函数里面应该有一个 loss 方法，loss 方法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 的例子中，替代使用 model.add_loss 函数来给 layer2 的输出添加正则项。

```
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)
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

源码见 author-profile.py

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 (<https://zenodo.org/record/3715864>) 里随机抽取出 1 万条构造的。

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年龄段。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。因此，这是一个多分类任务。

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 times_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怎么实施模型，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 层计算两个 tensor A 和 H 的相乘得到 $M = AH$ 。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 $\text{len}(\text{set}(\text{age}))$ 。

```

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(2*hidden_size,
activation='softmax',
use_bias=False, name='A')(ws)
M = Dot(axes=2, name='M')([H, A])

# mid_layer = Lambda(lambda x: K.mean(x, axis=1))(M)
mid_layer = Flatten()(M)
fc = Dense(128, activation='relu')(mid_layer)
outputs = Dense(len(set(age)), activation='softmax')(fc)
model = Model(inputs=inputs, outputs=outputs)

```

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

```
opt = Adam(learning_rate=lr)
loss = SparseCategoricalCrossentropy()
model.compile(loss=loss,
              optimizer=opt,
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=epoch, validation_split=0.1,
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

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 $\mathbf{x} = (x_1, x_2, \dots, x_T)$ 转换到一个隐状态输出 $\mathbf{h} = (h_1, h_2, \dots, h_T)$ 。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 。就是注意力机制。为 Decoder 的第 i 步计算 context 向量 c_i 的过程如下

$$c_i = \sum_{j=1}^T \alpha_{ij} h_j$$

$$\alpha_{ij} = \frac{\exp\{a(s_{i-1}, h_j)\}}{\sum_{k=1}^T \exp\{a(s_{i-1}, h_k)\}}$$

权重 α_{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

1

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

$$s_t = f(s_{t-1}, c_t, x_t)$$

(对应 10.1 节的 $s^{(t)} = O_h h^{(t)} + O_y y_{t-1} + O_c c$, 但没包括 y_{t-1})。这里 s_{t-1} 是从时间步 $t-1$ 传递给时间步 t 的状态。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 。

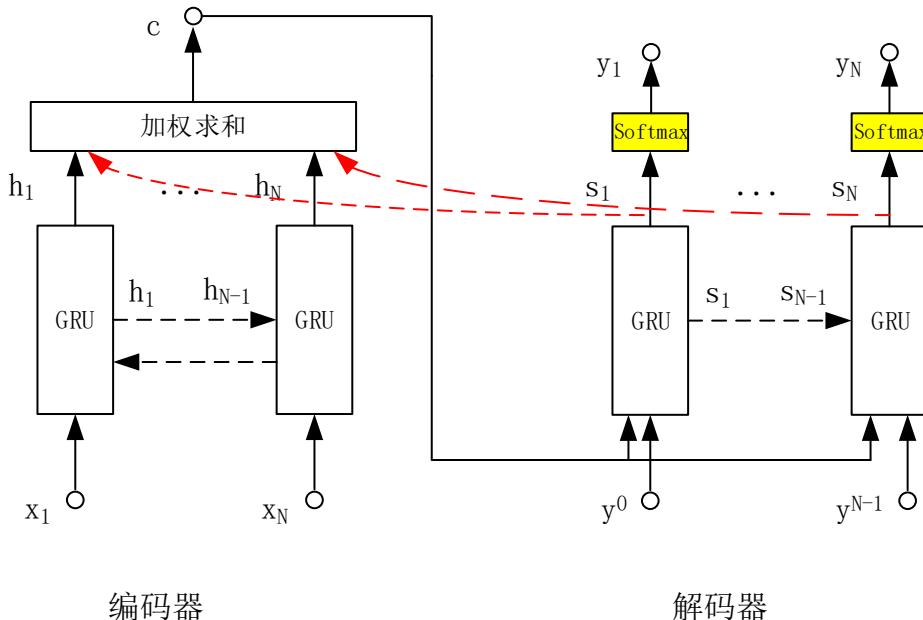


图 11.8 具有注意力机制的 Encoder-Decoder 模型

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)
    expanded_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) 推断

实际推断是在 title_predict.py 中完成。首先读入两个推断模型

```

from tensorflow.keras.models import load_model

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：Transformer 架构

第一节：Transformer 介绍

RNN 是处理序列数据的模型，它使得并行处理训练样本不可能。且内存的约束，限制了处理序列长度很长时数据。而注意力机制，按照前面章节的讨论，是与序列模型相伴的一个非常有力的技术。Transfomer 模型是在 Google Brain 科学家的论文

《Attention is all you need》[1] 中提出的。它是自然语言处理领域一个非常强有力 的模型。它既不包含“循环”，也不包含“卷积”，却非常依赖注意力机制。模型结构如图 12.1 所示。

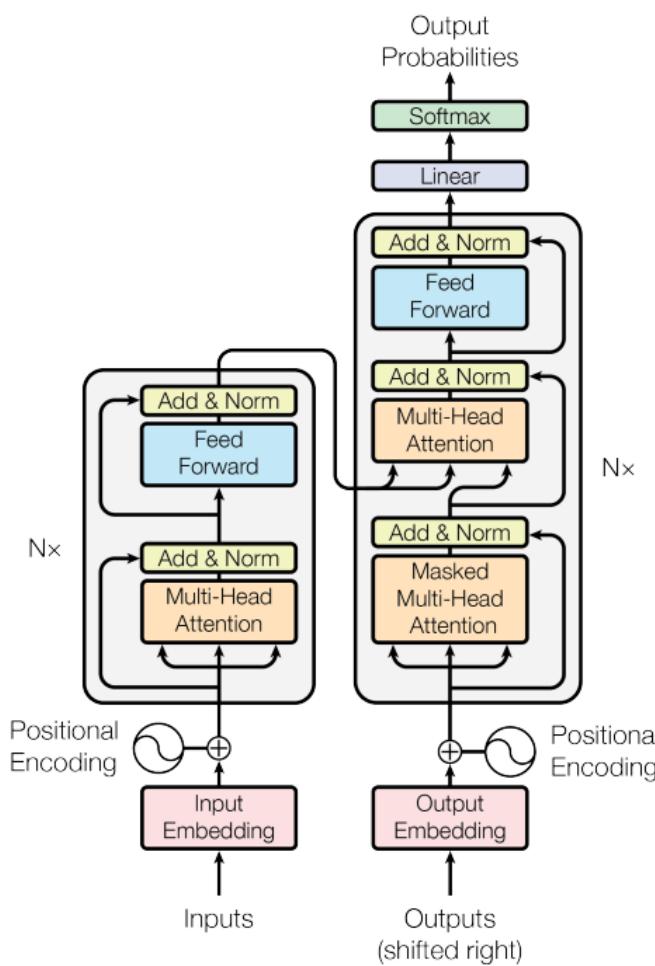


图 12.1 Transformer 的结构

Transformer 也是一个 Encoder-Decoder 结构。Transformer 中的 position 的概念，对应 RNN 中的时间步。Encoder 部分都包含一个多层（或叫 block）堆叠结构，每个层（block）的结构一致。图中的 Nx，表示一个 block 的多次堆叠。该文中 Encoder 是 6 个层（block）的堆叠。每个层（block）有两个子层。第一个子层是一个**多头自注意力机制**；第二子层则是逐点全连接层。每个子层使用了 residual connection。即，将第一层的输入和第一层的输出再进行一个直连相加。residual connection 的思想来自于 ResNet[2]。然后再做 layer normalization [3]。每个子层的计算形式化描述是 $\text{LayerNorm}(x + \text{Sublayer}(x))$ 。Sublayer(x) 是注意力机制部分。为了方便做 residual connection，所有子层的输出，包括输入经过 embedding 层，向量维度都是 d_{model} ，该文设为 512。

设 Encoder 的输入是一个 n 个词（token）的序列 (x_1, \dots, x_n) ，但 Transformer 中没有进行序列处理。因为是对每个词（token）进行了 positional encoding，所以每个词可以并行处理。Encoder 的输出就是 n 个向量。为了好理解，可以把图 12.1 中的 Encoder 描述的是 n 个词的并行处理，即每个词经过同一个 Encoder 的计算。但实际计算时，是输入是 n 个词经过嵌入层和位置编码（positional encoding）后的词向量组成的矩阵。即 Encoder 的输入是一个矩阵。

Decoder 也是六个 block 的堆叠。每个 block 除了包含 Encoder 一样的子层，它还在两个子层总结插入了第三个子层。该子层用 Encoder 的输出（n 个向量）完成多头注意力。而第一个子层的多头注意力机制和 Encoder 中的一样，不过多了一个 mask 操作。具体计算见 12.1.1 节的描述。

Transformer 的工作原理：首先根据输入序列 (x_1, \dots, x_n) ，在 Encoder 一端为每个词 x_i 计算了一个向量，一共 n 个向量作为 Encoder 的输出；在 Decoder 一端是逐个词计算。首先喂入一个开始符，例如，“<start>”，然后计算出一个 Decoder 的输出。例如，翻译任务中翻译了一个词。再将该输出和之前的输出（此处是“<start>”），作为下一步 Decoder 计算的输入，重复直至产生了停止符，例如，“<end>”。Decoder 的输入长度是固定的，每次**逐步解码**时，输入的左边逐步增加的字符，右边的位置被 masked.

下面多模型细节的各个部分分别做介绍。

12.1.1 多头注意力 (Multi-Head Attention)

该文中把注意力机制抽象为，将一个查询 Q 和一个键-值对 (K, V) 映射到一个输出 O 的过程。**查询、键、值和输出都是** $\text{shape} = n \times d$ 的 tensor，n 是序列长度，d 是一个时间步向量的长度（d 这里就是指代，具体的值见下面的计算）。输出就是值 V 的加权求和。而权重的计算是根据查询 Q 和键 K 的兼容程度计算。

注：transformer 中使用了多种多头注意力。对应的 Q、K 和 V 不同。12.1.2 节进行了解释。

1. Scaled Dot-Product Attention

多头注意力机制包含一个 Scaled Dot-Product Attention，如图 12.2 所示。

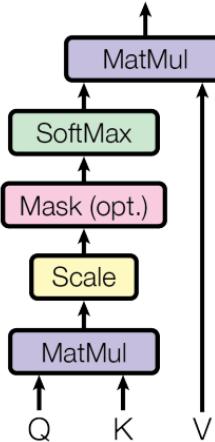


图 12.2. Scaled Dot-Product Attention

输入是查询 Q 和键 K，它们的 shape 都是 $n \times d_k$ 。计算查询 Q 和所有键的点乘；然后通过除以 $\sqrt{d_k}$ 进行规范化；然后通过一个 softmax 函数获得权重。V 的 shape= $n \times d_v$ ，注意力机制的计算如下所示

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (12.1)$$

其输出是一个 shape= $n \times d_v$ 的 tensor。该文指出，对齐函数 softmax $\left(\frac{QK^T}{\sqrt{d_k}}\right)$ 采用点乘计算效率更高，需要的空间更小。而此处除以 $\sqrt{d_k}$ 的规范化操作是因为，当 d_k 很大时点乘的结果在数量级上将增大，从而推动 softmax 函数到一个梯度非常小的区域。（该文举例说明为什么点乘的数量级会变大：假设 q 和 k 是均值为 0，方差为 1 的两个随机变量。它们的点乘结果均值为 0，方差为 d_k ）。

2. 多头注意力

多头注意力的计算如图 12.3。多头注意力中的输入 Q、K、V 都是 shape= $n \times d_{model}$ 的 tensor。它们经过投影（公式 12.3 中的 QW_i^Q, KW_i^K, VW_i^V ）变成图 12.2 中 Q、K、V，其一个时间步向量长度为 d_k, d_k, d_v 。它们分别经过 h 次的投影（h 个不同线性变换，公式 12.3，这就是多头的含义）。每次投影（产生每个头 head）将查询 Q、值 V 和键 K 分别线性投影到 d_k, d_k 和 d_v 维度的向量（公式 12.3 中的 QW_i^Q, KW_i^K, VW_i^V ）。然

后通过 scaled dot-product attention 计算，再将计算结果进行拼接；再通过一个线性层计算。计算公式如下：

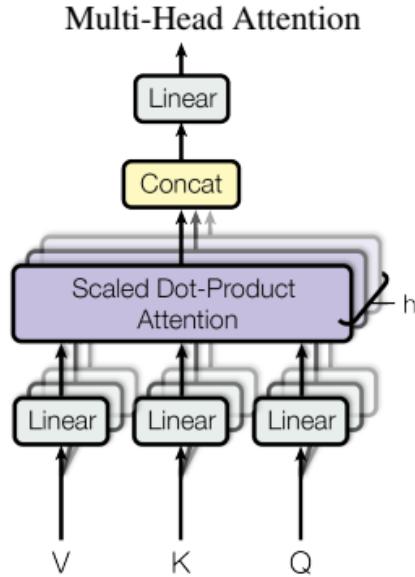


图 12.3. 多头注意力的结构

$$\text{MultiHead}(Q, K, V) = \text{concat}(\text{head}_1, \dots, \text{head}_h)W^o \quad (12.2)$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \quad (12.3)$$

其中 $W_i^Q \in R^{d_{model} \times d_k}$, $W_i^K \in R^{d_{model} \times d_k}$, $W_i^V \in R^{d_{model} \times d_v}$, $W^o \in R^{(h \times d_v) \times d_{model}}$ 。

head_i 是一个 $n \times d_v$ 的 tensor。拼接 h 个 head 后是 $n \times (h \times d_v)$ 的 tensor。MultiHead 的输出是 $n \times d_{model}$ 的 tensor。

需要说明的是，transformer 是多个 block 的堆叠（block 里面的 layer 称作子层）。因此每个 block 的输入和输出的 tensor 的 shape 应该一致。从图 12.1 可以看到，Transformer 中 multihead 的输出 tensor 是下一个 block 的输入，即下一 block 的 $Q=K=V=$ 上一个 block 的 Multihead 的输出。

在该文中给出的参数 $h = 8$, $d_k = d_v = \frac{d_{model}}{h} = 64$ 。由于每个 head（即每个映射，共 h 个）的维度并不高（这里是 $h=8$ ），因此计算成本和单头的全连接差不多。

当 Q 、 K 、 V 是一个向量时，多头注意力的输出 $\text{MultiHead}(Q, K, V)$ 是一个长度为 d_k 的向量（注：在 keras 的 multiheadattention 层，可以通过设置 `output_shape` 来设置向量长度，如果未设置就是 d_{model} ）。在 Transformer 具体计算时， Q 、 K 、 V 都是矩阵，因此 MultiHead 的输出是一个 $n \times d_{model}$ 矩阵。

12.1.2 Transformer 中三个地方应用了多头注意力机制

1. Encoder

Encoder 每个 block 应用的是多头自注意力机制。Encoder 中的第一个 block 的输入 Q 就是输入的 n 个词。其他 block 的输入，就是上一个 block 的输出。经过变换（12.1.5 节介绍）后的向量集合， $Q \in \mathbb{R}^{n \times d_k}$ 。键值对 $K \in \mathbb{R}^{n \times d_k} = Q$ ， $V \in \mathbb{R}^{n \times d_k} = Q$ 。按照公式 12.2，Encoder 的一个 block 的输出就是一个 $n \times d_k$ 的矩阵。

2. Decoder 的每一个 block 的第一个子层

Decoder 的每一个 block 的第一个子层是 masked 多头自注意力。在 Decoder，第一个子层的多头自注意力机制使用了 mask 操作。即在图 12.2 中，的“mask (opt.)”层。这里 (opt.) 可选是指在别的多头注意力中，mask 操作可选。但在 decoder 的第一个子层中的 mask 操作是必须的。它的含义是，decoder 的自注意力机制中，仅仅允许注意到前面的输出的词，对未来输出的词的位置（预留了位置，即限定了输出最多的词的个数？）需要设置为 -inf，即进行 mask 操作。然后再进行 softmax 的运算。

3. Decoder 每一个 block 的第二个子层

Decoder 每一个 block 的第二个子层采用 encoder-decoder 注意力。按照 Encoder-Decoder 中的注意力机制（第十一章）。Decoder 时间步 $i-1$ 的 RNN 的输出 s_{i-1} 就是查询 Q；Encoder 的输出就是键 K，也是值 V；输出就是 MultiHead 的 $n \times d_{model}$ 的 tensor。

12.1.3 逐位置前馈网络

Encoder、Decoder 的每个层(block)中有个“Feed Forward”子层。它是一个逐位置前馈网络（position-wise feed forward network），是对每个位置（也即时间步）应用这个网络。

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

12.1.4 Layer normalization

在 4.4 节我们讨论 Batch Normalization 时指出：

神经网络的一个隐层的输出，是下一层的输入。这个输入对应一个分布，这个分布受到前一层的参数变化的影响，也受到了 minibatch SGD 时，每个批次样本差异的影响。这个影响称为 **internal covariate shift**。观察到的现象就是，训练模型时，网络内部隐层的参数改变，使得下一层的输入的分布的均值和标准差在变化。这导致神经网络的训练困难。在 batch normalization 中，第 l 层的第 i 个神经元的输入 a_i^l 被重新标定到固定的均值（接近 0）和标准差（接近 1）。

$$\bar{a}_i^l = \frac{g_i^l}{\sigma_i^l} (a_i^l - \mu_i^l) \quad \mu_i^l = \mathbb{E}_{\mathbf{x} \sim P(\mathbf{x})} [a_i^l] \quad \sigma_i^l = \sqrt{\mathbb{E}_{\mathbf{x} \sim P(\mathbf{x})} [(a_i^l - \mu_i^l)^2]}$$

\bar{a}_i^l 表示规范化后的输入。 g_i 是一个增益参数。 μ 和 σ 可以从当前 batch 的样本估计。

Layer normalization 则是在同一层的所有神经元上进行均值和方差的计算，

$$\bar{a}_i = \frac{a_i - \mu}{\sigma} g_i$$

$$\mu = \frac{1}{n} \sum_{i=1}^n a_i, \quad \sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (a_i - \mu)^2}$$

12.1.5 Positional Encoding

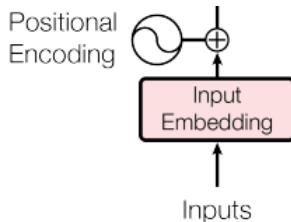


图 12.4 positional encoding

位置编码是 Transformer 实现可以并行处理序列数据的关键。Transformer 为序列中的每个 token 进行位置编码，结果是一个长度和 embeddings 一样的向量，都是 d_{model} ，以便它们两个可以求和。如图 12.4 所示。为序列中第 pos 个 token 进行位置编码的公式如下

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

pos 代表 token 的位置， i 是向量上的维度。它表示在一个 token 的位置编码的向量中，偶数维度上的值是用正弦函数计算，奇数上的值是用余弦函数计算。所有 token 的位置编码在同一个维度的值，绘制出来是一个正弦或余弦曲线。如图 12.5 所示

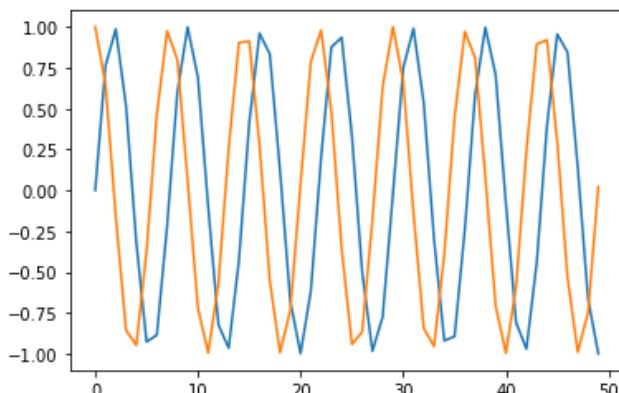


图 12.5 所有位置编码在第 1 和第 2 个维度上的曲线

第二节：实例：使用 Transformer 进行文本分类

官方的 tensorflow 和 keras 并没有提供 Transformer block。有一些第三方开发了 tensorflow 的 (<https://pypi.org/project/tf-transformers/>) 和 keras 的 (<https://pypi.org/project/keras-transformer/>) transformer 库。但 keras 提供了实施 transformer 的基本功能包括多头注意力，Layer normalization。本节我们按照 keras 一篇文档自己开发 Transmormer layer

https://keras.io/examples/nlp/text_classification_with_transformer/

本节使用第一节介绍的 Transformer 的 Encoder 建立一个基于 transformer 的文本分类模型。

12.2.1 构建 Transformer 的基本类

首先介绍构建 Transformer 中的几个关键类

1. 多头注意力 (tensorflow >=2.5 版本)

`tf.keras.layers.MultiHeadAttention`

```
tf.keras.layers.MultiHeadAttention(  
    num_heads, key_dim, value_dim=None, dropout=0.0, use_bias=True,  
    output_shape=None, attention_axes=None,  
    kernel_initializer='glorot_uniform',  
    bias_initializer='zeros', kernel_regularizer=None,  
    bias_regularizer=None, activity_regularizer=None, kernel_constraint=None,  
    bias_constraint=None, **kwargs  
)
```

它的参数包括：

<code>num_heads</code>	多头注意力中的头数
<code>key_dim</code>	查询向量 \mathbf{Q} 和键向量 \mathbf{K} 投影后的长度 d_k
<code>value_dim</code>	值向量 \mathbf{V} 投影后的长度 d_v 。（不给的话和 <code>key_dim</code> 一样）
<code>dropout</code>	Dropout probability.
<code>use_bias</code>	Boolean, whether the dense layers use bias vectors/matrices.
<code>output_shape</code>	多头注意力的输出 tensor 的 shape=[batch-size, target 序列的长度 T, 输出向量的长度]，此处设置的 <code>output_shape</code> 就是输出向量的长度，即图 12.3 中的 d_{model}
<code>attention_axes</code>	axes over which the attention is applied. None means attention over all axes, but batch, heads, and features.

创建多头注意力层的对象后，调用该对象时给的参数包括

- **query:** Query Tensor of shape [B, T, dim]. B 是 batch size, T 是 query 序列长度, dim 是序列一个时刻的向量长度 d_{model}
- **value:** Value Tensor of shape [B, S, dim]. S 是 value 序列长度, 其他同上
- **key:** **Optional key** Tensor of shape [B, S, dim]. If not given, will use value for both key and value, which is the most common case.
- **attention_mask:** a boolean mask of shape [B, T, S], that prevents attention to certain positions. The boolean mask specifies which query elements can attend to which key elements, 1 indicates attention and 0 indicates no attention. Broadcasting can happen for the missing batch dimensions and the head dimension.
- **return_attention_scores:** A boolean to indicate whether the output should be attention output if True, or (attention_output, attention_scores) if False. Defaults to False.
- **training:** Python boolean indicating whether the layer should behave in training mode (adding dropout) or in inference mode (no dropout). Defaults to either using the training mode of the parent layer/model, or False (inference) if there is no parent layer.

调用后返回的结果是

attention_output	The result of the computation, of shape [B, T, E], B 是 batch size; T 是 target 序列长度; 如果参数 out_shape 规定了, E=out_shape, 否则是输入 Q 的 shape 的最后一个维度的长度 d_{model}
attention_scores	[Optional] multi-head attention coefficients over attention axes.

举例：下面是在两个序列上计算多头注意力

```
layer = MultiHeadAttention(num_heads=2, key_dim=2)
target = tf.keras.Input(shape=[8, 16])
source = tf.keras.Input(shape=[4, 16])
output_tensor, weights = layer(target, source,
                               return_attention_scores=True)
print(output_tensor.shape)
```

设置了两个头，设置了 $d_k = 2$ ，即输入序列的每个时刻的向量，经过映射后 QW_i^Q 的向量长度。其实就是在设置 W_i^Q 权重矩阵的大小。查询 Q 是输入 target，K 和 V 是输入 source，因为在多头注意力对象 layer 的调用中只给出了 Q (target) 和 K

(source) , 没给出 V, 就默认 V=source。多头注意力的输出的 shape=[batch_size, Q 的序列长度, S 的 shape 的最后一个维度]，因此输出结果是

(None, 8, 16)

2. LayerNormalization

```
tf.keras.layers.LayerNormalization(  
    axis=-1, epsilon=0.001, center=True, scale=True,  
    beta_initializer='zeros', gamma_initializer='ones',  
    beta_regularizer=None, gamma_regularizer=None, beta_constraint=None,  
    gamma_constraint=None, **kwargs  
)
```

3. 逐位前馈网络 (position-wise feed forward network)

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

看公式，就是两个全连接层，第一个是 Relu 激活函数，第二个是线性激活函数。

4. 位置编码 Positional Encoding

本节的实施将位置编码简化了，就使用了一个位置的查找表（嵌入层）。

12.2.2 使用 Transformer 中的 Encoder 实施文本分类

1. 首先构建 Transformer 的两个类

```
# 1. transformer layer  
class TransformerBlock(layers.Layer):  
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):  
        super(TransformerBlock, self).__init__()  
        self.att = layers.MultiHeadAttention(num_heads=num_heads,  
key_dim=embed_dim)  
        self.ffn = keras.Sequential(  
            [layers.Dense(ff_dim, activation="relu"), layers.Dense(embed_dim),]  
        ) # 用 Sequential 组合多个层的技巧可以学习一下  
        self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)  
        self.layernorm2 = layers.LayerNormalization(epsilon=1e-6)  
        self.dropout1 = layers.Dropout(rate)  
        self.dropout2 = layers.Dropout(rate)  
  
    def call(self, inputs, training):  
        attn_output = self.att(inputs, inputs)  
        attn_output = self.dropout1(attn_output, training=training)  
        out1 = self.layernorm1(inputs + attn_output)  
        ffn_output = self.ffn(out1)  
        ffn_output = self.dropout2(ffn_output, training=training)
```

```

        return self.layernorm2(out1 + ffn_output)

class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, maxlen, vocab_size, embed_dim):
        super(TokenAndPositionEmbedding, self).__init__()
        self.token_emb = layers.Embedding(input_dim=vocab_size,
output_dim=embed_dim)
        self.pos_emb = layers.Embedding(input_dim=maxlen,
output_dim=embed_dim)

    def call(self, x):
        maxlen = tf.shape(x)[-1]
        positions = tf.range(start=0, limit=maxlen, delta=1)
        positions = self.pos_emb(positions)
        x = self.token_emb(x)
        return x + positions

```

TransformerBlock 是构建的 Transformer 的一个层，它继承了 keras 的 Layer 类。在其中的初始化方法中，创建了需要的对象，包括多头注意力、逐位前馈网络、layernormalize 等。

在 call 方法中完成计算。

在 TokenAndPositionEmbedding 类中完成输入 token 到词向量和位置向量的转换和两者的相加计算。这里把位置编码给简化了，就是用了一个嵌入层。

注：用 Sequential 组合多个层的技巧可以学习一下

2. 数据集准备

数据就是利用了 keras 自带的 imdb 数据集

```

vocab_size = 20000 # Only consider the top 20k words
maxlen = 200 # Only consider the first 200 words of each movie review
(x_train, y_train), (x_val, y_val) =
keras.datasets.imdb.load_data(num_words=vocab_size)
print(len(x_train), "Training sequences")
print(len(x_val), "Validation sequences")
x_train = keras.preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_val = keras.preprocessing.sequence.pad_sequences(x_val, maxlen=maxlen)

```

3. 构建和训练模型

在 Transformer 的那篇论文中，它的使用了 6 个 transformer 层（每个层包含多头注意力和逐位前馈子层）。下面的例子中两次使用创建的 transformer 层。然后把输出结果向量求均值。最后带入一个全连接的输入层。

```
embed_dim = 32 # Embedding size for each token
```

```

num_heads = 2 # Number of attention heads
ff_dim = 32 # Hidden layer size in feed forward network inside transformer

inputs = layers.Input(shape=(maxlen,))
embedding_layer = TokenAndPositionEmbedding(maxlen, vocab_size,
embed_dim)
x = embedding_layer(inputs)
transformer_block = TransformerBlock(embed_dim, num_heads, ff_dim)
x = transformer_block(x)
x = transformer_block(x)
x = layers.GlobalAveragePooling1D()(x)
x = layers.Dropout(0.1)(x)
x = layers.Dense(20, activation="relu")(x)
x = layers.Dropout(0.1)(x)
outputs = layers.Dense(2, activation="softmax")(x)

model = keras.Model(inputs=inputs, outputs=outputs)

model.compile("adam", "7", metrics=["accuracy"])
history = model.fit(
    x_train, y_train, batch_size=32, epochs=2, validation_data=(x_val, y_val)
)

```

第三节：Transformer 的解码策略

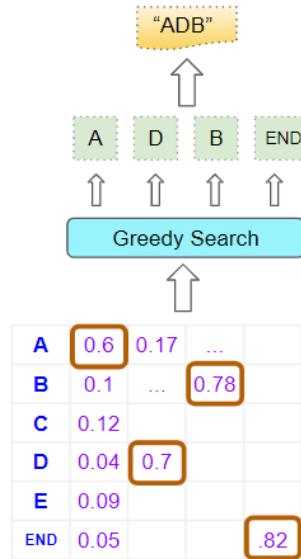
之所以 Encodr-Decoer 模型、Transformer 模型是生成模型，是因为它们有个解码器可以生成 token 序列。12.1 节讲到，Transformer 的解码器在解码时仍然是一步一步的生成 token。设 s_t 是 Decoder 在时间步 t 的 logit 输出。而 decoder 计算在时间步 t 产生词 j 的概率如下：（注：为了方便描述，下面的公式符号和 10.1 节和 12.1 节对解码器的描述并不一致）

$$P(y_{t,j} = 1 | y_{t-1}, y_{t-2}, \dots, y_1, X) = \frac{\exp(s_{t,j})}{\sum_{j'=1}^K \exp(s_{t,j'})}$$

设 s_t 是解码器在时间步 t 的输出， X 是词汇表有 K 个词项。 s_t 是一个长度为 K 的向量，每个元素对应词汇表中的一个词项。上面的公式就是将 s_t 经过 softmax 变换，转换成在词汇表上的概率分布。而在时间步 t 输出哪个 token 有多种策略：

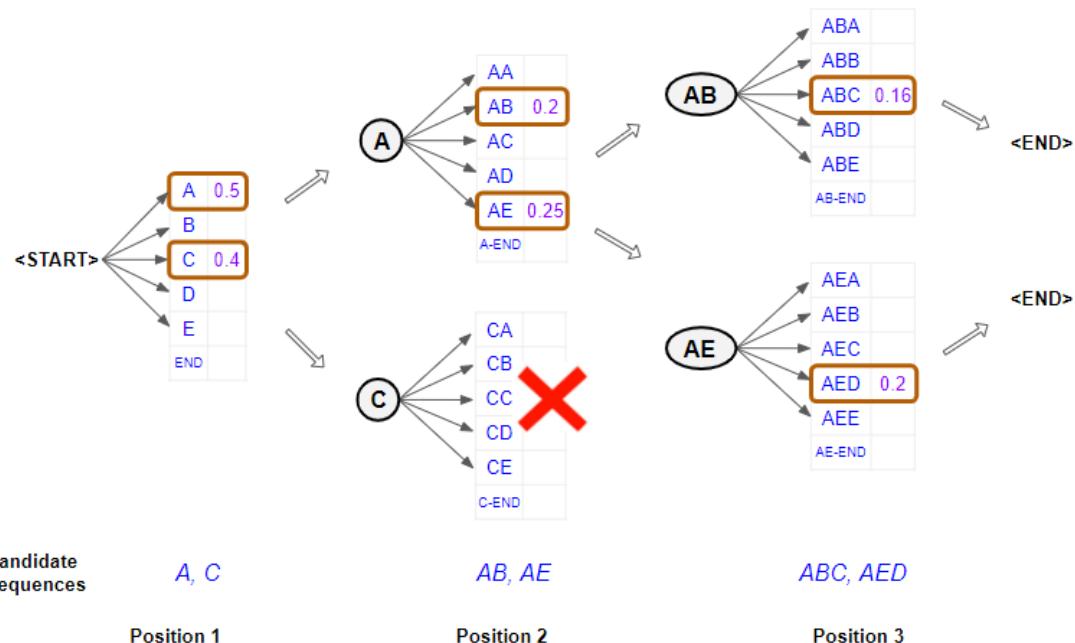
1. 贪婪搜索

最简单的方法就是在 $\text{softmax}(s_t)$ 上，选择最大的概率值对应的 token。每个时间步上都是最大概率对应的 token，如此生成一个 token 序列。



2. 集束搜索

集束搜索 (beam search, 或称束搜索) 在贪婪搜索的基础上做了改进。贪婪搜索中在每个时间步选择概率最高的 token，而集束搜索中每个时间步搜索概率最高的 N 个 token。即在每一步生成时，维护 N 个最佳候选序列，避免贪心搜索的局部最优问题。N 称之为**束宽** (Beam Width)。



集束搜索的工作原理如图所示。设束宽 N=2，在第一个时间步，选择概率最高的两个 token，得到当前的两个序列{A}, {C}。

在第二个时间步，根据当前维护的 2 个序列分别计算第二个时间步的每个 token 被生成的概率： $P(y_{2,j} = 1|A, X)$ 和 $P(y_{2,j} = 1|C, X)$ 。然后以每个序列的联合概率最大为原则挑选第二个时间步应该输出的两个 token。上图中的例子是 $P(B|A, X) = 0.4$, $P(B|A, X) = 0.5$ 。{AB} 和 {AE} 的联合概率分别是 0.2 和 0.25。它们大于其他的任何 {A*} 和 {C*} 序列。因此，第二个时间步维护的量序列是挑 {AB} 和 {AE}。

在第三个时间步，根据当前维护的 2 个序列分别计算第三个时间步的每个 token 被生成的概率： $P(y_{3,j} = 1|AB, X)$ 和 $P(y_{3,j} = 1|AE, X)$ 。然后以每个序列的联合概率最大为原则挑选第三个时间步应该输出的两个 token。上图中的例子是 $P(C|AB, X) = 0.64$, $P(D|AE, X) = 0.8$ 。因此，{ABC} 和 {AED} 的联合概率分别是 0.16 和 0.2。它们大于其他的任何 {AB*} 和 {AE*} 序列。因此，第三个时间步维护的量序列是挑 {ABC} 和 {AED}。

重复上述的步骤，知道产生了结束符 <END>。

比较贪婪搜索只生成一个 token 序列，集束搜索可以生成 N 个 token 的序列。

3. 随机抽样

前面的解码策略都不具有随机性。例如，如果是贪婪搜索，每次生成的序列都是一样的。我们在使用大模型时，能感觉到，每次得到的结果虽然意思可以是一样的，但在形式上是不一样的；或者有时意思都不一样。这是因为解码策略引入了随机性。这样做的好处是，生成的结果可以更多样、更丰富。

随机抽样的策略是在解码每个 token 的时候，不是选择概率最高的 token，二是从该 token 的分布中抽样得到。同时在将每个时间步的 logit 输出进行 softmax 转换成概率分布时，引入了 temperature 参数。

$$\text{softmax}(s_i) = \frac{\exp(s_i/t)}{\sum_j \exp(s_j/t)}$$

当 t 的值小于 1 时，越小，向量 s_t 的每个元素相当于被放大了更大的倍数。如此，再做 softmax 规范化时，本身大的元素会被增加的更多。如此的 softmax 转换，将原本高的概率值更突出，因此在随机抽样时，会更加注意选择原本概率就高的 token。如此的解码结果得到的更具有确定性。想反当 t 的值越大，原本高的概率值会被压缩，每个 token 之间的概率之间的差异会被缩小，如此随机抽样时，每个 token 都会被抽到的可能性在增加。如此的抽样策略生成的结果会更丰富。

这也就是我们在使用大模型时，设置 temperature 参数，越大的值生成的结果越丰富，越小的值生成的结果越确定。

4. Top-k 抽样

贪婪搜索选择概率最高的 token，随机抽样从整个分布中抽样一个 token。Top-k 抽样进行了两者的结合，它选择概率最高的 top k 个 token，再从中抽样一个 token。在各种大模型的 API 都有这个参数。例如通义千问，叫做 topk（参见我的讲义《大语言模型与应用》）。

5. Nucleus 抽样或 top-p 抽样

Nucleus 抽样在各 LLM 的 API 提供的参数中通常叫 top-p 抽样。

Top-k 抽样有一个问题，在每个解码的时间步 k 值是固定的。 k 值选大了会有长尾漏洞。即有可能选择很小概率的 token，造成句子不通顺，而 k 选小了又和 beam search 是一样的。

在 ICLR2020 的一篇论文“*The curious case of neural text degeneration*”中提出了 nucleus sampling 的 ce 率，翻译作核抽样或核采样。它就是为了解决 top-k 抽样中的 k 值的问题。Nucleus 抽样的策略如下：

给定一个概率阈值 p ，从 token 候选集合 V 中选择一个最小子集 $V_p \subset V$ ，使得 V_p 中 token 的出现概率和大于等于 p 。也即选择前 n 个最高概率的 token，使得它们的概率和正好大于等于 p 。然后重新对 V_p 中的 token 的概率进行标定，使得它们的概率和等于 1。再按照标定后的概率分布从 V_p 中抽样一个 token。参数 p 的变化范围是 0~1。如果 $p=0$ ，则就是贪婪搜索；如果 $p=1$ 则表示没有使用 Nucleus，就是随机抽样。参数 p 的值越大，则表示越大的生成结果越多样；越小表示生成结果越确定。

和 top-k 抽样策略比较，nucleus 抽样在每个解码的时间步候选的 token 集合大小是动态变化的。

第四节：MoE Transformer

我们知道训练集的增加和模型规模的增加可以提高深度神经网络的性能。在典型的深度神经网络中，每送入一个样例，将激活模型的每个神经元。这样可以导致模型训练成本的增加。如此条件计算（Conditional Computation）被提出，即当增加模型的 capacity，计算成本并不线性地增加。条件计算中，每个送入模型的样例的计算部分网络是不激活的。而决定哪些部分的网络激活称为 gating decision。

在论文“*OUTRAGEOUSLY LARGE NEURAL NETWORKS: THE PARSELY-GATED MIXTURE-OF-EXPERTS LAYER*”中，提出了一个可以进行条件计算的 MoE 层，如果图 12.6 所示。

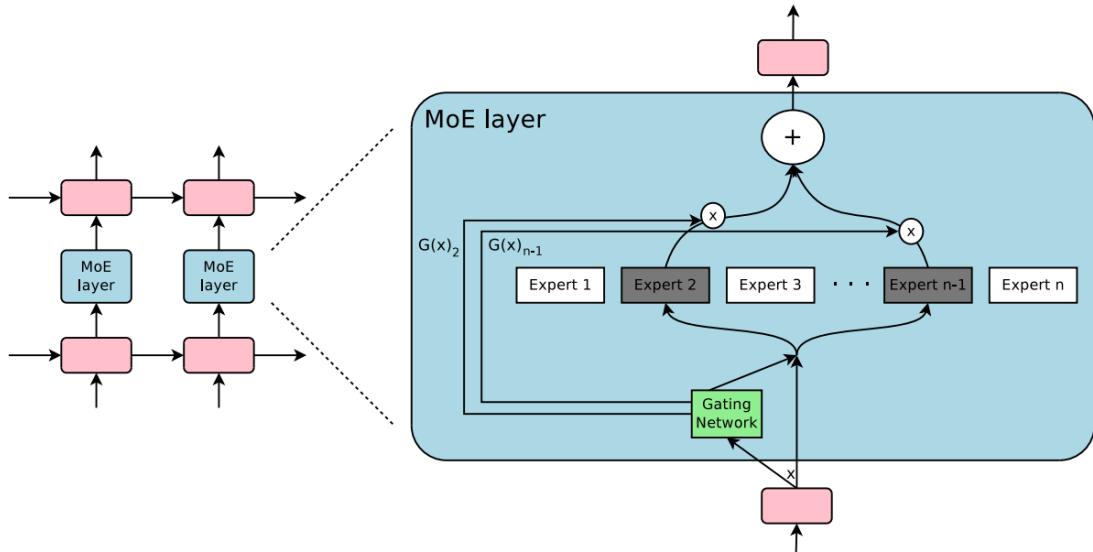


图 12.6. MoE 层

图 12.6 是嵌入在一个 RNN 网络中的 MoE 层。此例中，Gating Network (有的也叫 Gating function) 选择了两个 Experts 完成计算。层的输出由 Gating network 对选中的一个 Expert 的输出进行调整计算后输出。

下面我们先了解一些 MoE (Mixture of Experts) 的概念。MoE 是在 1990 年代提出的一种机器学习技巧。这里多个 Experts 划分问题空间到多个区域，是一种集成学习方法。Expert 可以看作是解决问题的一个模型。例如，SVM，ANN。在神经网络中，它可以是一个子网络。

MoE 层包含 n 个 Expert Networks E_1, \dots, E_n ，一个 Gating Network G 。每个 Expert Network 是相同架构的神经网络，但有它们自己的参数。Gating 网络的输出是一个稀疏的 n 维向量。设 $G(x)$ 是 Gating 网络的输出， $E_i(x)$ 是第 i 个 Expert 的输出。MoE 层的输出 y

$$y = \sum_{i=1}^n G(x)_i E_i(x)$$

因为 Gating 网络的输出 $G(x)$ 的稀疏性可以节约计算成本。因为如果 $G(x)_i = 0$ 就不需要计算 $E_i(x)$ 。

对于 Gating Network，最简单的形式是

$$G_\sigma(x) = \text{Softmax}(x \cdot W_g)$$

即将输入和可训练的权重矩阵 W_g 相乘，再经过一个 Softmax 激活函数，输出一个长度为 n 的向量。但它不是稀疏的。

一个 Noisy Top-K Gating 机制如下，它在上面的简单形式的 Gating 网络中加入了噪声和引入稀疏性。

$$G(x) = \text{Softmax}(\text{KeepTopK}(H(x), k))$$

$$H(x)_i = (x \cdot W_g)_i + \text{StandardNormal}() \cdot \text{Softplus}((x \cdot W_{noise})_i)$$

$$\text{KeepTopK}(v, k)_i = \begin{cases} v_i & \text{if } v_i \text{ is in the top } k \text{ elements of } v. \\ -\infty & \text{otherwise.} \end{cases}$$

对于输入 x ，仍然是和权重矩阵 W_g 相乘，输出一个长度为 n 的向量。再添加一个 Gaussian 噪声。即从标准正太分布中抽样的结果乘上一个权重。权重的生成是由输入 x 和一个权重矩阵 W_{noise} 相乘得到的长度为 n 的向量，再经过一个 Softplus 激活函数得到的。softplus 是一个平滑版 ReLU， $f(x) = \log(1 + \exp(x))$ 。如此，可以得到一个 n 维向量 $H(x)$ 。

KeepTopK 函数从 $H(x)$ 中挑选前 k 个值保持原值，其他值设为负无穷大。如此，再经过 Softmax 激活函数得到长度为 n 的门控向量 $G(x)$ 中，只有 k 个位置有值，其余值为 0。

该论文指出，MoE 技术可以 1000 倍的提升模型 Capacity，而模型的损失很小。正是由于对于输入，模型的部分（有 Gating 网络控制）才参与计算，减少了模型的计算成本（包括训练成本）。现在 MoE 是一种非常有效的增加模型规模而不线性模型计算量的技术。因此，很多大模型都采用了 MoE 技术。它们把 MoE 层添加到 Transformer 中，称为 MoE Transformer。例如，DeepSeek MoE-16B 的计算量和 LLaM2-7B 相当。但它的模型规模是后者的两倍多。

MoE 层可以添加到 Transformer 的 Encoder 和 Decoder 中，如图 12.7 所示。左图中是标准的 Transformer Encoder 中的一个 block。Block 的输入经过自注意力计算后，再喂入一个前馈神经网络，其间都有残差连接和 layer normalization 的操作。右图展示了插入 MoE 层的一个 block。它是间隔的替换一个 block 中的前馈神经网络为 MoE 层。

现在对 MoE 的研究，包括 Gating network 的设计，Expert Network 的设计等。详细的内容可参考论文 “A Survey on Mixture of Experts”。

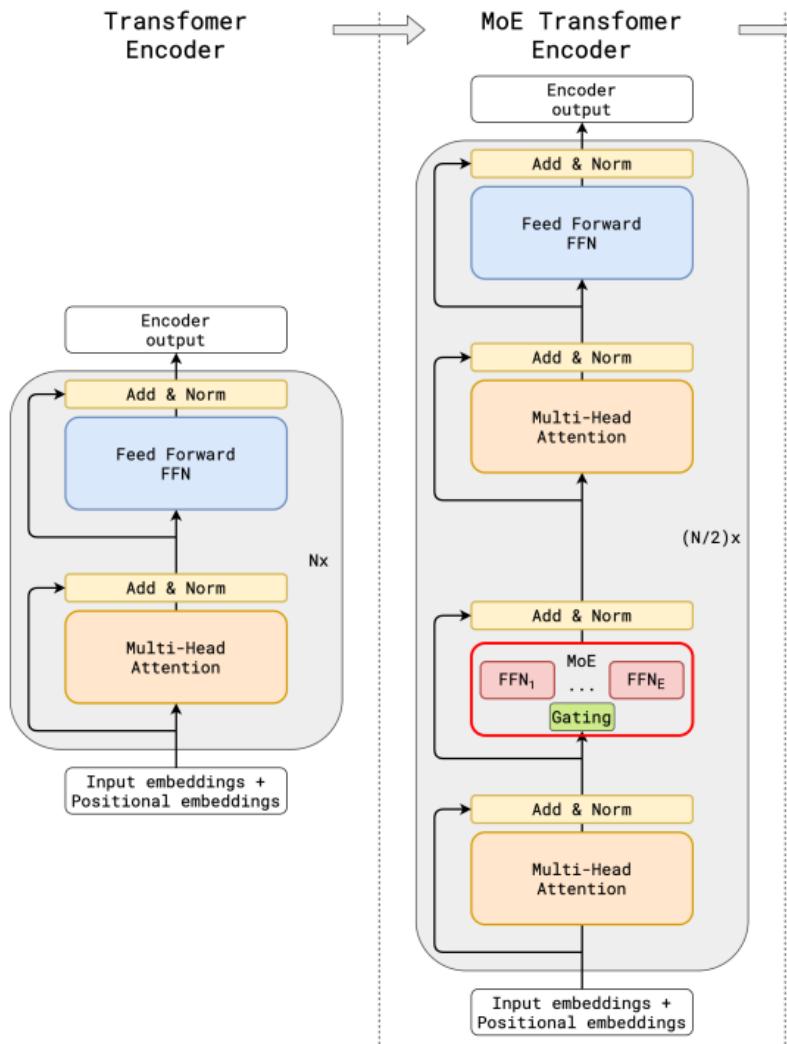


图 12.7 MoE Transformer 的结构

第五节：DeepSeek 的架构

DeepSeek 是优秀的国产大模型，它不但应用广泛，还开源。DeepSeek V3 在很多项目上号称是当前时间点最好的大模型。大模型都是基于的 Transformer 架构。有的是保存了 Transformer 的 Encoder-Decoder 结构，如 T5。更多的是使用了 Decoder-only 架构，即只使用了 Transformer 中的 Decoder。DeepSeek、GPT 采用的是 Transformer 中的 only-decoder 架构。

本节我们通过了解 DeepSeek 的架构来熟悉 Transformer 的应用。

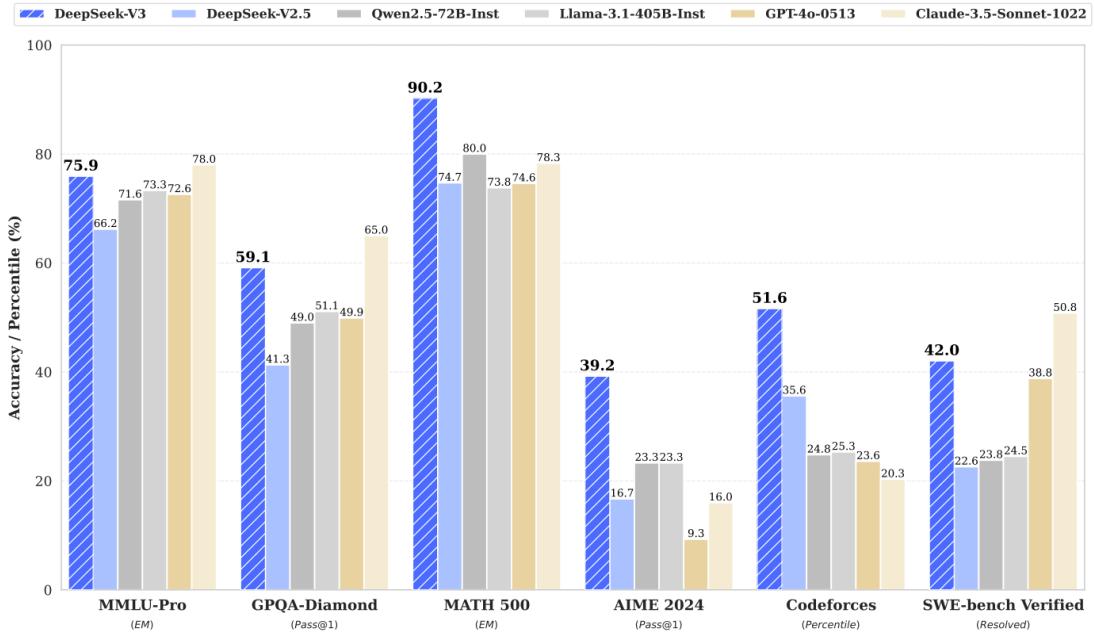


图 12.7 DeepSeek 模型的性能

图中的数据集介绍：

- MMLU-Pro 是大规模多任务语言理解数据集。
- GPQA-Diamond 是涉及生物、物理、化学知识的多项选择数据集。
- MATH 500 是 500 道数学题的数据集。
- AIME2024 是 Artificial Intelligence Math Evaluation 数据集
- Codeforce 是 codeforce.com 平台提供的编程数据集。
- SWE-bench Verified 是人工校正的 SWE-bench 数据集，它用于 evaluates AI models' ability to solve real-world software issues.

下面介绍 DeepSeek V3 的架构。它采用的 only decoder Transformer 的基本架构，再加上 MLA (Multi-head Latent Attention) 用于有效的推断和 MoE (Mixture of Expert) 用于降低训练成本。DeepSeek V3 采用 61 个 blocks。一个 block 的结构如图 12.8 所示。

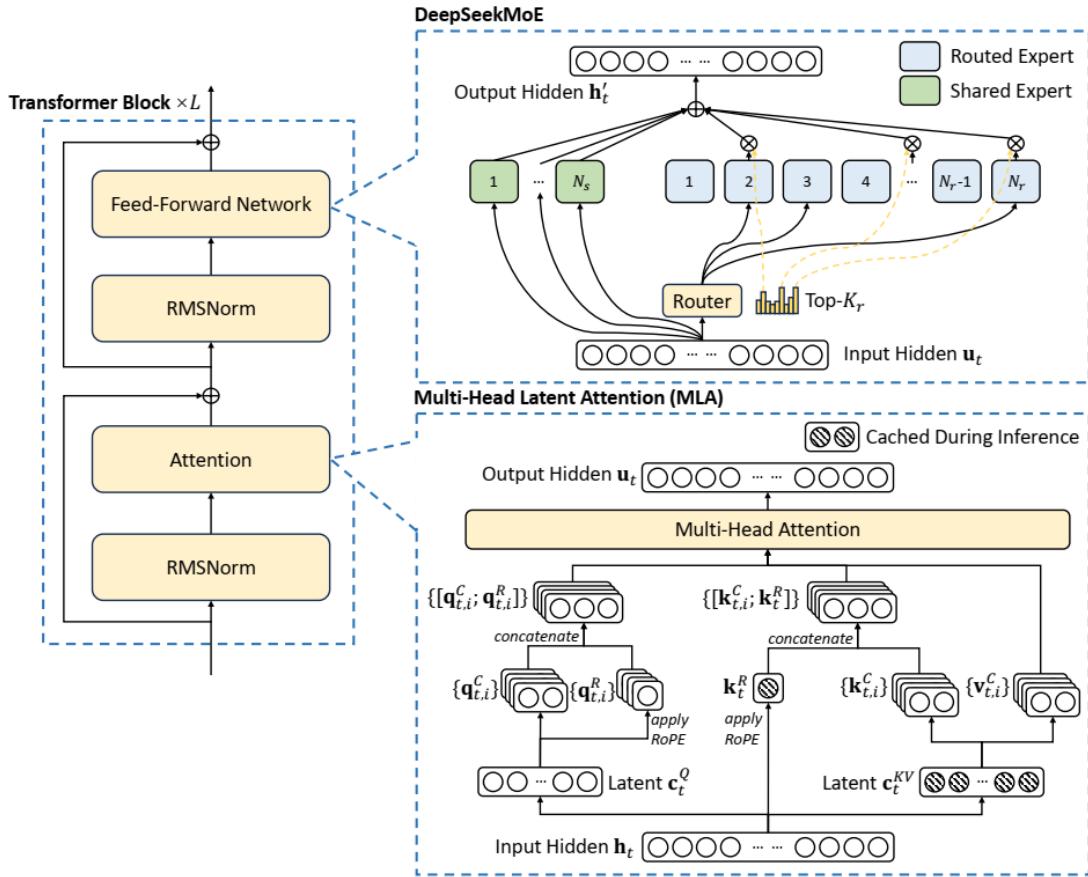


图 12.8 DeepSeek 的基本架构

多头注意力中有 128 个 head。除了前三个 block，后面所有 block 的 FNN 层用 MoE 层替代。每个 MoE 层包含 1 个 shared expert 和 256 个 routed expert。

1. RMSNorm

在经典 Transformer 的论文中 Layer normalization 是在同一层的所有神经元上进行均值和方差的计算。设 a_i 是第 i 个神经元的输入， \bar{a}_i 是规范化后的输入， g_i 是一个增益系数。规范化时的均值 μ 是当前层的所有神经元输入的均值，标准差 σ 则是前层的所有神经元输入上计算的标准差。

$$\bar{a}_i = \frac{a_i - \mu}{\sigma} g_i$$

$$\mu = \frac{1}{n} \sum_{i=1}^n a_i, \quad \sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (a_i - \mu)^2}$$

而 DeepSeek 用 Root Mean Square Layer Normalization。

$$\bar{a}_i = \frac{a_i}{\text{RMS}(\mathbf{a})} g_i, \quad \text{where } \text{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n} \sum_{i=1}^n a_i^2}$$

即直接把当前神经元的输入 a_i 除以当前层所有神经元输入的均方根。

相比较 Layer normalization, RMSNorm 在计算上更有效率。

2. Multi-head Latent Attention

设 d 是 embeddings 的维度, n_h 是注意力 head 的数量, d_h 是每个 head 的维度, $\mathbf{h}_t \in \mathbb{R}^d$ 在一个给定的注意力层第 t 个 token 的输入。MLA 的核心是对注意力计算中的 key 和 value 进行联合的低秩压缩来减少推断时 key-value (KV) 缓存。MLA 的第 t 个时间步的输出 (或者说对应第 t 个输入 token 的输出) \mathbf{u}_t 是每个头的输出的拼接后的转换。

$$\mathbf{u}_t = W^O [\mathbf{o}_{t,1}; \mathbf{o}_{t,2}; \dots; \mathbf{o}_{t,n_h}]$$

而第 i 个 head 的计算

$$\mathbf{o}_{t,i} = \sum_{j=1}^t \text{Softmax}_j \left(\frac{\mathbf{q}_{t,i}^T \mathbf{k}_{j,i}}{\sqrt{d_h + d_h^R}} \right) \mathbf{v}_{j,i}^C$$

这两步的计算与标准的多头注意力 (MHA) 计算是一致的。MLA 和 MHA 的区别体现在 Q、K、V 的计算上。对于在第 t 步, 第 i 个头上的查询 Q 的计算如下, 它是两个计算部分的拼接

$$\begin{aligned} \mathbf{q}_{t,i} &= [\mathbf{q}_{t,i}^C; \mathbf{q}_{t,i}^R] \\ [\mathbf{q}_{t,1}^C; \mathbf{q}_{t,2}^C; \dots; \mathbf{q}_{t,n_h}^C] &= \mathbf{q}_t^C = W^{UQ} \mathbf{c}_t^Q, \\ [\mathbf{q}_{t,1}^R; \mathbf{q}_{t,2}^R; \dots; \mathbf{q}_{t,n_h}^R] &= \mathbf{q}_t^R = \text{RoPE}(W^{QR} \mathbf{c}_t^Q) \\ \mathbf{c}_t^Q &= W^{DQ} \mathbf{h}_t \end{aligned}$$

RoPE (Rotary Position Embedding) 即旋转位置编码 RoPE 用一个旋转矩阵编码 token 的绝对位置, 并且集成了在自注意力机制中 token 之间的相对位置关系。相比较前面介绍的 Transformer 中的位置编码, RoPE 能够有效处理不同序列长度, 处理随着相对距离增加而衰减的 token 间依赖性问题, 以及能够为线性自注意力机制配备相对位置编码的能力等。

键 K 的计算如下

$$\mathbf{k}_{t,i} = [\mathbf{k}_{t,i}^C; \mathbf{k}_t^R]$$

$$\boxed{\mathbf{k}_t^R} = \text{RoPE}(W^{KR} \mathbf{h}_t)$$

$$[\mathbf{k}_{t,1}^C; \mathbf{k}_{t,2}^C; \dots; \mathbf{k}_{t,n_h}^C] = \mathbf{k}_t^C = W^{UK} \mathbf{c}_t^{KV}$$

$$\boxed{\mathbf{c}_t^{KV}} = W^{DKV} \mathbf{h}_t$$

它也是由两部分组成，一部分是时间步 t 的输入 \mathbf{h}_t 经过旋转位置编码 RoPE 后的结果 \mathbf{K}_t^R ，另一部分是键值压缩隐向量 \mathbf{c}_t^{KV} 经过转换后的结果 \mathbf{K}_t^C 。其中 \mathbf{K}_t^R 和 \mathbf{c}_t^{KV} 被保存在缓存。如此减少了计算量。

值 V 的计算如下，

$$[\mathbf{v}_{t,1}^C; \mathbf{v}_{t,2}^C; \dots; \mathbf{v}_{t,n_h}^C] = \mathbf{v}_t^C = W^{UV} \mathbf{c}_t^{KV}$$

它也是键值压缩隐向量 \mathbf{c}_t^{KV} 经过转换后的结果 \mathbf{v}_t^C

3. DeepSeekMoE

与经典的 MoE 相比较，DeepSeekMoE 使用更精细粒度的 expert。

$$\mathbf{h}'_t = \mathbf{u}_t + \sum_{i=1}^{N_s} \text{FFN}_i^{(s)}(\mathbf{u}_t) + \sum_{i=1}^{N_r} g_{i,t} \text{FFN}_i^{(r)}(\mathbf{u}_t),$$

\mathbf{u}_t 是 MoE 层的第 t 个时间步的输入， \mathbf{h}'_t 是第 MoE 层第 t 个时间步的输出。有 N_s 个 shared expert，每个 shared expert 都参与最终的计算；有 N_r 个 routed expert，有 Gating 函数控制选择 routed expert 参与计算。 $\text{FFN}_i^{(s)}$ 是第 i 个 shared expert， $\text{FFN}_i^{(r)}$ 是第 i 个 routed expert。（每个 expert 是一个前馈网络 FFN）。

Gating 函数的计算如下：

$$\begin{aligned} g_{i,t} &= \frac{g'_{i,t}}{\sum_{j=1}^{N_r} g'_{j,t}}, \\ g'_{i,t} &= \begin{cases} s_{i,t}, & s_{i,t} \in \text{Topk}(\{s_{j,t} | 1 \leq j \leq N_r\}, K_r), \\ 0, & \text{otherwise}, \end{cases} \\ s_{i,t} &= \text{Sigmoid}(\mathbf{u}_t^T \mathbf{e}_i), \end{aligned}$$

\mathbf{e}_i is the centroid vector of the i -th routed expert（这句话我没有理解到）。第 i 个 expert 的 Gating 值 $g_{i,t}$ ，是所有 expert 的 Gating 评分 $g'_{i,t}$ 的归一化的结果。时间步 t 的输入和每个 \mathbf{e}_i 计算一个评分 $s_{i,t}$ 。前 k 个评分最高的评分保留，其余的置为 0。

4. 解码器的逐步解码

DeepSeek、GPT 这些模型采用的是 Transformer Decoder-only 的架构，它们仍然是逐步生成 token 的。几个关键步骤如下：

(1) 掩码自注意力：解码器使用掩码自注意力，这使得它在生成过程中无法“看到”序列中的未来 token。这确保了模型按顺序生成输出，一次一个 token。

(2) 自回归生成：这个过程本质上是自回归的。每个生成的 token 都被用作预测下一个 token 的输入，从而形成一个链式生成过程。

简而言之，虽然解码器仅有的架构简化了模型的结构，但它并没有改变 GPT 模型中 token 生成本质上的顺序性。

参考文献

- [1] Ashish Vaswani, et. al., Attention Is All You Need. NIPS2017
- [2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 770–778, 2016
- [3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. arXiv preprint arXiv:1607.06450, 2016.
- [4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet classification with deep convolutional neural networks. In Proceedings of NeurIPS, pages 1097–1105.
- [5] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In Proceedings of ICLR.
- [6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and JianSun. 2016. Deep residual learning for image recognition. In Proceedings of CVPR, pages 770–778.
- [7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In Proceedings of CVPR, pages 248–255.
- [8] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. Learned in translation: Contextualized word vectors. In NeurIPS, 2017.
- [9] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In NAACL-HLT, 2018
- [10] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018. URL <https://s3-us-west-2.amazonaws.com/openai-assets/researchcovers/languageunsupervised/languageunderstandingpaper.pdf>.
- [11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In NAACL-HLT, 2019.

[12] Zhengyan Zhang et. al., CPM-2: Large-scale Cost-effective Pre-trained Language Models. arxiv.org/abs/2106.10715

第十三章：预训练语言模型

现在风靡的大模型或大语言模式都是预训练语言模型，通常说参数数量在 1b (billion) 以上的预训练语言模型称为大语言模型。而预训练语言模型都是采用的 Tranformer 架构。

第一节：预训练语言模型介绍

大规模预训练模型简称大模型。从 2012 年起，一系列研究在著名的人工标注的视觉识别数据集 ImageNet[7]上预训练大规模 CNN 模型[4, 5, 6]。其后的一些研究使用预训练的 CNN 模型，使用特定任务的数据集（相对来说比较小）来精调（fine-tuneing）预训练的模型，然后在下游任务中取得了很好的性能。如此，开启了第一波对预训练模型（Pre-Trained Model, PTM）的研究。在这一波研究中，几乎所有的计算机视觉 CV 领域的研究都使用了预训练模型。

自然语言处理领域也开始意识到了预训练模型的重要性。最近的研究指出，在大规模语料库上预训练的语言模型能够学习通用的语言表示（universal language representation）。如此一些下游 NLP 任务能够避免从零训练一个新的模型。第一代的预训练语言模型的目标是学习到比较好的词向量（word embeddings）。例如，都有使用 word2vec 和 GloVe 在大规模语料库上预训练的词向量。但学到的词向量不能解决不同语境下的一词多义现象。第二代的预训练语言模型目标是学习上下文感知的词向量 contextual word embeddings），他们包括 CoVe[8], ELMo [9], OpenAI GPT [10]和 Bert [11]。这些模型通常是一个预训练的语言模型，在下游任务中通过精调（fine-tuning）这些预训练模型，在上下文中学习词的表示向量。而面向任务的数据集通常可以很小，但任务完成的却异乎寻常的好。如此，预训练语言模型掀起了第二波预训练模型的热潮。

预训练模型的发展见图 12.6

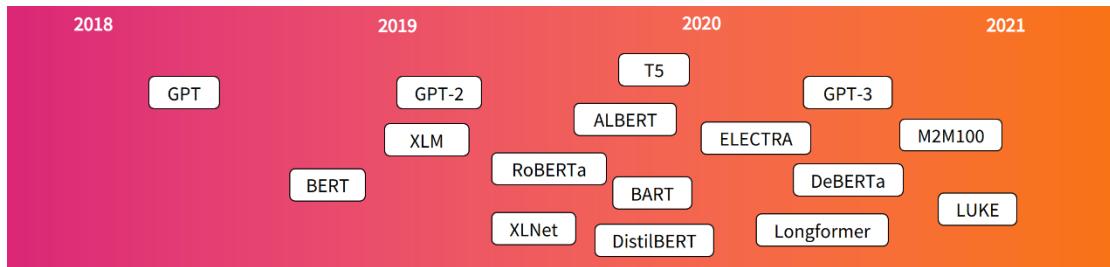


图 12.6 预训练模型的发展

Transformer 模型被发明时是针对的翻译任务，后来的以 Transformer 为基础的预训练模型几乎可以完成自然处理种的各种任务。有影响的模型包括

2018 年 6 月 GPT。它是第一个预训练 Transformer 模型。

2018 年 10 月 BERT。

2019 年 2 月 GPT-2。一个更大的 GPT 版本。

2019 年 10 月 DistilBERT。Bert 的精华版。它比 bert 快 60%，内存占用少 40%，仍旧保存了 97% 的性能。

2019 年 10 月 BART 和 T5。两个很大的预训练模型，使用了与原始的 Transformer 同样的结构。

2020 年 5 月 GPT-3。比 GPT-2 更大的模型，不需要精调就可以在各种任务上完成的很好（这称为 zero-shot learning, ZSL）。

2021 年 6 月，北京智源研究院发布了中文的预训练语言模型 ‘悟道’ CPM-2[12]。它的 github 地址，<https://github.com/TsinghuaAI/CPM>。

2022 年 GPT-3 以及其后非常多的大模型涌现。

基于 Transformer 的 GPT 和 Bert 模型的研究发现，随着模型复杂性的增加，具有上亿个参数的模型的大规模预训练模型可以捕获歧义、语法、词法的结构，也可以捕获文本中的实事性知识 (factual knowledge，定义为任意一个领域内的术语、基本信息等)。

最新的 GPT-3 模型，如图 12.7，其模型参数甚至达到 1750 亿个。它是在 560G 的数据上使用 1 万块 GPU 训练而成。它显示出了学习世界知识 (world knowledge)、常识和逻辑推理的能力。

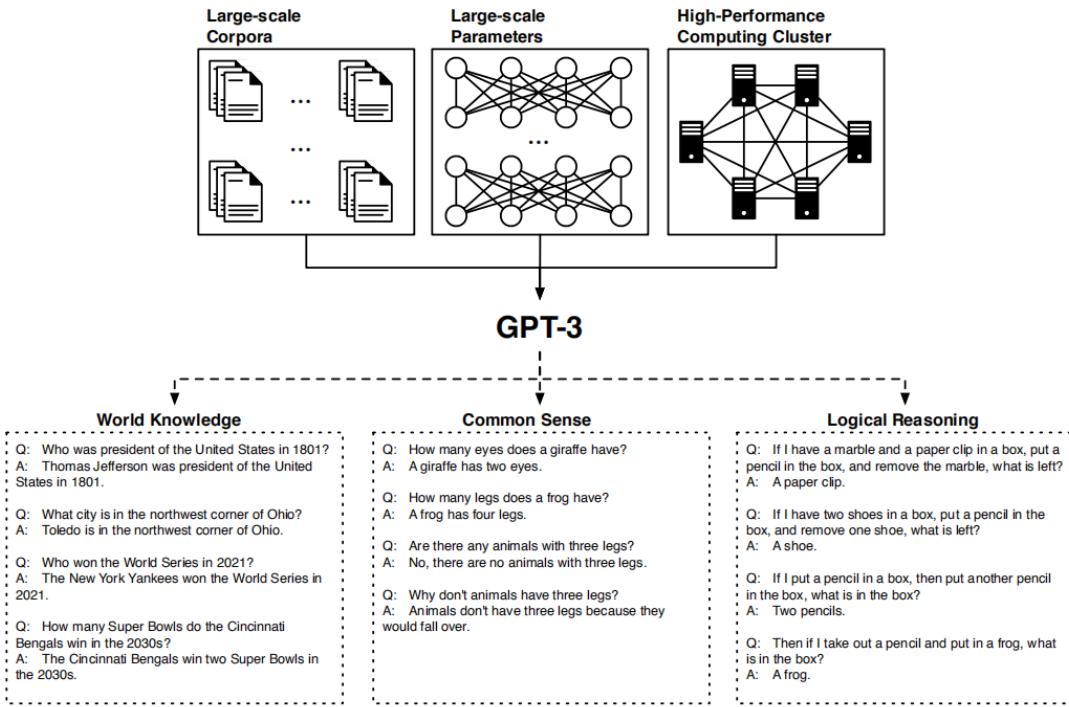


图 12.7 GPT 模型

2022 年发布的‘悟道 2.0’达到了 1.75 万亿的参数。

宽泛的来说，预训练语言模型分为三类

- GPT-like (也称为 auto-regressive Transformer models)
- BERT-like (也称为 auto-encoding Transformer models)
- BART/T5-like (也称为 sequence-to-sequence Transformer models)

下面我们分别介绍 Bert 和 GPT。

12.3.1 BERT

1. 模型结构

单就语言模型来说，它分为两种：causal LM 和 Masked LM。

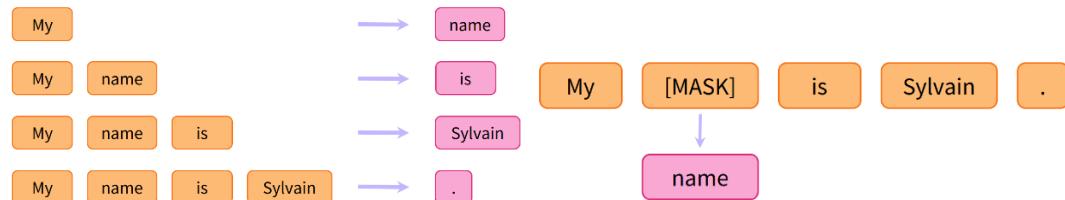


图 12-8. 两种语言模型 causal LM 和 masked LM

CausalLM 根据之前的信息来预测下一个词，也称为自回归语言模型（auto-regressive LM）。如图 12-8 左图所示。之前在第六章介绍的语言模型就是 Causal LM，它是单向语言模型（unidirectional LM）。

Masked LM，即遮掩语言模型。Masked LM 是根据前后的信息，预测一段文本中间的词。在训练时，随机的遮住部分输入的 token，目标是基于上下文预测被遮住的 token。

BERT (Bidirectional Encoder Representations from Transformers) 是由 Google 科学家在 2018 年开发的一个非常成功的一个语言模型。最初开发时，它是一个英语的语言模型，包含两个大小不同的版本。（1）BERT_{BASE}: 12 个 Blocks，每个 Block 有 12 bidirectional self-attention heads，总共 110M 个参数；（2）BERT_{LARGE}: 24Blocks，每个 Block 有 16 bidirectional self-attention heads，总共 340M 个参数。两个模型的隐层大小都是 768。它们能处理的输入序列长度是 512 个 token。

Bert 的结构是一个多层双向 Transformer encoder (所以 Bert 只有 Encoding 功能，而没有生成功能)，是遮掩（masked）语言模型，如图 12.7 所示。它的训练分成预训练和精调两个阶段。预训练任务 Google 在发布时就已经完成了，即我们拿到的模型就是预训练 BERT。我们在应用时，根据下游任务，基于 BERT 搭建自己的模型，再精调。

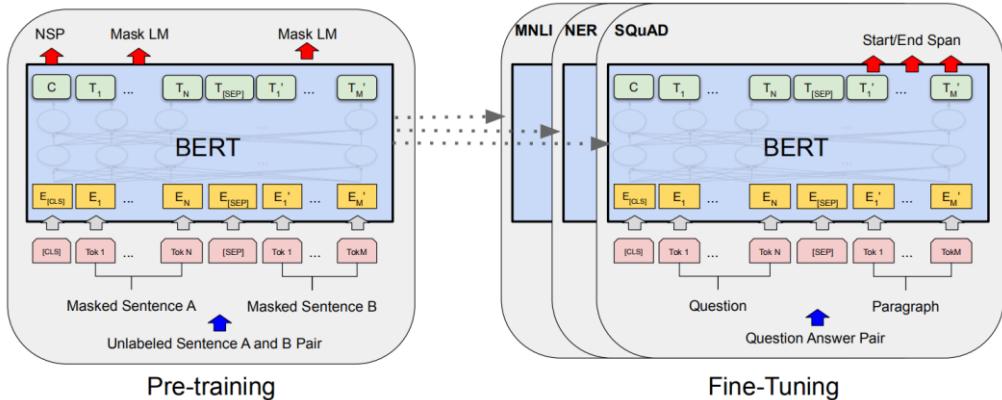


图 12.7 BERT 模型

BERT 的预训练是在 unlabeled 数据集上完成多个预训练任务，包括 NSP(Next Sentence Prediction) 和 Masked LM。在精调阶段，面向每个下游任务使用 labeled 数据集，以预训练的模型作为初始模型，所有参数面向下游任务精调。如图 12.7 所示，可以是问答系统，可以是命名实体识别 NER。

下面我们了解一下 BERT 对输入的处理（这些步骤是 BERT 自己完成，不是由用户来完成的）。为了使 Bert 能处理各种下游任务，输入即可以是一个单独的句子，也可以是

一对句子（例如，`<Question, Answer>`）。这里的‘句子’是任意一个连续的文本块，不一定是语言学上实际的句子。一个‘序列’既可以是一个‘句子’，也可以是用符号`[SEP]`拼接到一起的两个句子。每个序列的第一个 token 是一个特殊的符号`[CLS]`。输入的 embeddings 的计算见 12.8 节。它是 token embedding+positional embeddings+segment embeddings 的和。Token embedding 就是词向量，positional embeddings 是 Transformer 的位置编码，segment embeddings 是输入一个句子对时，区分两个句子的 embeddings。

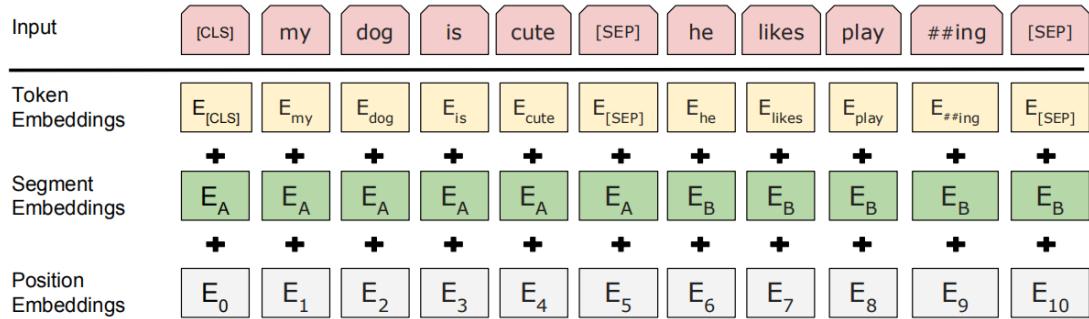


图 12.8 输入 embeddings 的计算

2. 预训练

Bert 的预训练包括两个无监督任务：Masked LM 和 Next Sentence Prediction (NSP)。

Masked LM。 传统的语言模型是单向的。为了训练一个深度双向语言模型，输入的 token 序列会被随机的部分遮掩（用 token `[mask]`替换），然后预测被遮掩的这些 token，这就是 Masked LM(MLM)。在图 12.7 的模型中，模型的输出（隐层）会被带到 softmax 输出层来预测输出的 token，与标准的语言模型一样。在每个序列中 15% 的 token 被随机遮掩。Bert 仅仅预测被遮掩的 token，而不是重构整个输入序列。如此训练的模型，该文称之为是 bidirectional pre-trained model。

讨论：

前面多次提到了 BERT 是双向语言模型。怎么理解？

传统语言模型的训练是有方向的，例如，left-to-right 的单向语言模型，在训练时，喂入模型的序列，向左提前错开一位，作为模型的 target。那 Transformer 有方向吗？我们知道 Transformer 采用位置编码处理序列数据，因此可以并行的处理输入的文本中的每个 token。在 Encoder 部分，所有时刻的数据进行多头注意力的计算，此时，一个 token 上的注意力的计算参考了前面的 token 和后面的 token，那它就无所谓方向的问题，或者称为双向的。在 Decoder 的部分，在解码的过程中，喂

入了一个序列 $x_0 \sim x_i$, 解码第 i 个 token 的输出。第 i 个 token 在计算注意力时, 只能看到前面的 token, 因此是有向的。

BERT 仅仅使用 Transformer 的 encoder 作为语言模型。GPT 仅仅使用了 Decoder。
这就可以理解在 BERT 的原始论文中

the BERT Transformer uses **bidirectional self-attention**, while the GPT Transformer uses constrained self-attention where every token can only attend to context to its left

这句话的意思。

如此训练的模型在精调阶段会有些问题, 因为预训练阶段使用了 token '[mask]', 在精调阶段并不会出现。所以在预训练模型时, 并不是总是对被遮掩的部分使用 token '[mask]'。实际的训练集的生成过程如下:

- (1) 随机选择 15% 的 token 位置做预测;
- (2) 如果 i -th 位置的 token 被选择, 替换该 token (a) 以 0.80 的概率用 token '[mask]'; (b) 以 0.1 的概率用随机的其他的词; (c) 以 0.1 的概率不改变, 还是原始的词。
- (3) 仅使用原始输入的 token T_i 用于做目标数据, 和预测的 token 计算二元交叉熵作为损失函数。

原文给了一个例子

- 80% of the time: Replace the word with the [MASK] token, e.g., my dog is hairy → my dog is [MASK]
- 10% of the time: Replace the word with a random word, e.g., my dog is hairy → my dog is apple
- 10% of the time: Keep the word unchanged, e.g., my dog is hairy → my dog is hairy. The purpose of this is to bias the representation towards the actual observed word.

NSP 任务。许多重要的下游任务, 如问答、自然语言推断, 都是基于两个句子之间关系的理解。语言模型并不能直接捕获这种关系。因此 bert 使用一个句子预测任务训练模型。构建训练集的样本时, 使用句子对 A 和 B, 其中 50% 的 B 是 A 的真实 next

sentence (分配标签 IsNext) , 50%不是真实的 (分配标签 NotNext) 。图 12.7 中的 C, 即 token [CLS]的输出用于二分类预测 A 和 B 是不是匹配的。

3. 精调

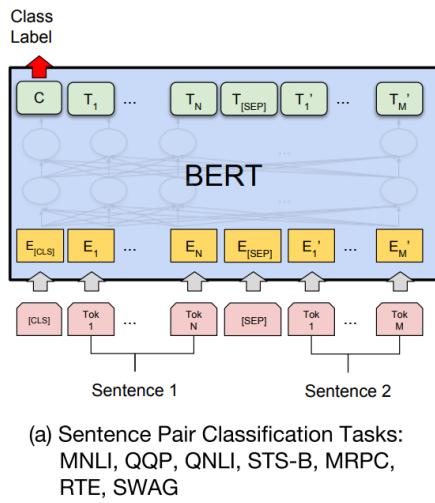
精调就是我们拿到了预训练模型, 在它上面完成下游任务, 而进行模型训练的过程。

对于下游任务, 无论它们是涉及单个句子的任务 (如, 分类) 或句子对的任务 (如, 问答) 都可以使用预训练的 BERT。对于特定的任务, 将任务需要的输入和输出插入到 Bert, 然后端到端的精调所有参数。

精调为什么可以使用很普通的梯度下降优化算法 (没有特殊的正则化机器) 和较小的训练集去面向下游任务调节一个上亿乃至更大的模型? 论文 “INTRINSIC DIMENSIONALITY EXPLAINS THE EFFECTIVENESS OF LANGUAGE MODEL FINE-TUNING” 对此进行了研究, 本文不做进一步的讨论。

基于 BERT 可以进一步完成的下游任务有四大类: 句子对分类, 单句分类, 问答和序列标注任务。

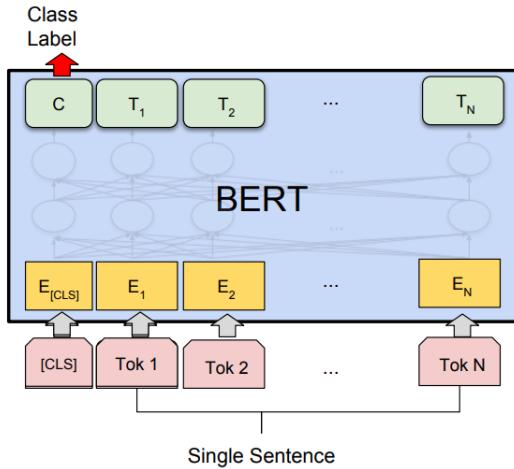
(1) 句子对分类。



该类任务的输入是一个句子对, 预测两个句子之间的关系。例如 QQP 任务是针对一个 Quora 上的问题对, 判断它们是否语义相关。QNLI 任务是给定一个 (question, sentence) 文本对, 预测 sentence 是否包含了 question 的答案。

它们都是用[CLS]标记对应的输出, 在其上建立分类模型。

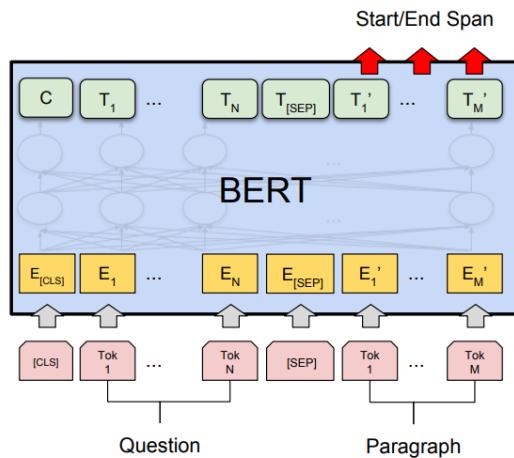
(2) 单句分类。



(b) Single Sentence Classification Tasks:
SST-2, CoLA

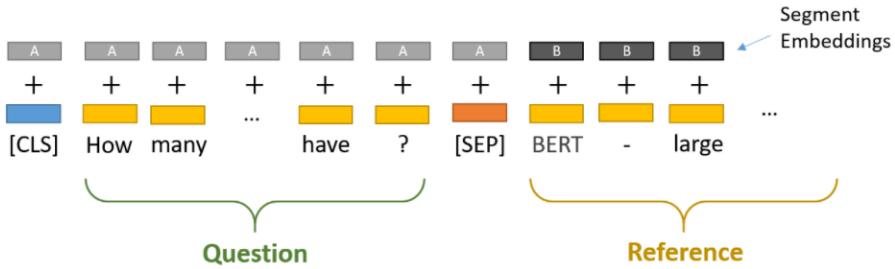
在分类任务中，输入端输入单独一段文本。输出端是在 Transformer 的 token [CLS] 的输出，加上一个输出层。token [CLS] 的输出，被看作是累积了输入的语义信息。

(3) 问答任务（信息抽取）。



(c) Question Answering Tasks:
SQuAD v1.1

以 The Stanford Question Answering Dataset (SQuAD v1.1)为例。该数据集给出了问题 (question) 和一个段落 (reference)，要求是从该段落中找出一小段文本 (span) 作为该问题的答案，参见图 12.9。Question 和 reference 中的每个词都会按照图 12.8 进行编码。 (<https://mccormickml.com/2020/03/10/question-answering-with-a-fine-tuned-BERT/>)



Question: How many parameters does BERT-large have?

Reference Text: BERT-large is really big... it has 24 layers and an embedding size of 1,024, for a total of 340M parameters! Altogether it is 1.34GB, so expect it to take a couple minutes to download to your Colab instance.

图 12.9 在 SQuAD 问答任务中 bert 的输入

在精调任务中引入一个 start 向量 S 和一个 end 向量 E 。输出端则是对应的段落中的每个 word 的输出，参见图 12-10。每个输出向量和 start 向量点乘，经过 softmax 函数。计算每个 word 是答案 start 的概率。同样的，段落中的每个 word 也和 end 向量 E 计算点乘，经过 softmax 函数后，计算出每个 word 是答案 end 的概率。满足段落中 $S \cdot T_i + E \cdot T_j$ 且 $i < j$ 的最大的一个文本序列 T_i, \dots, T_j 就是答案。

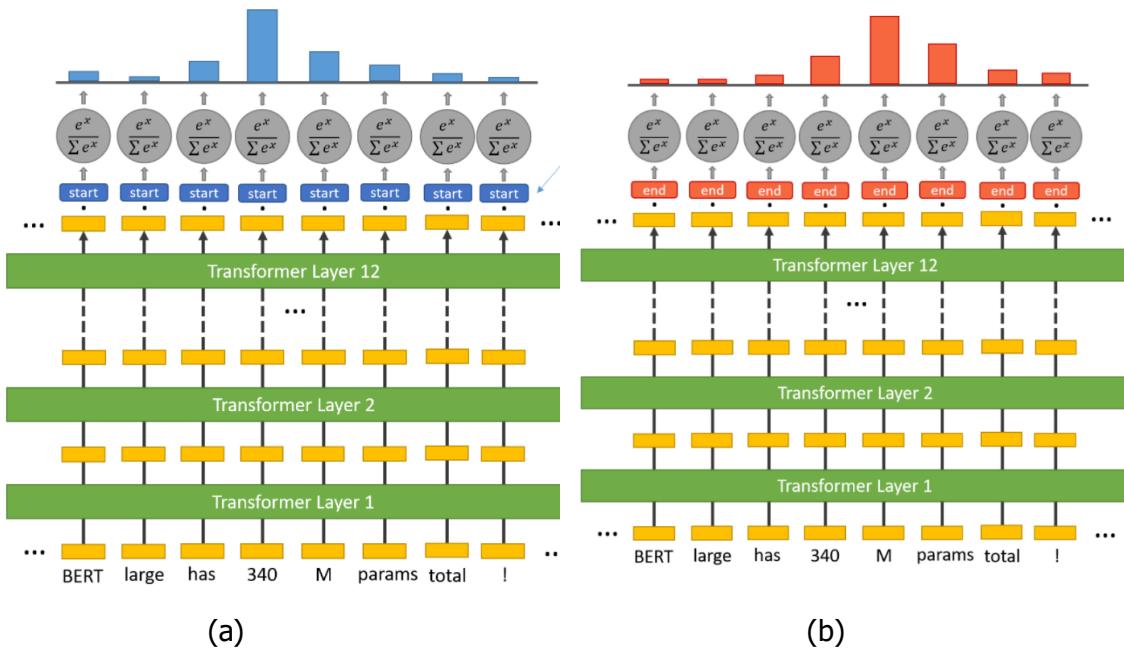
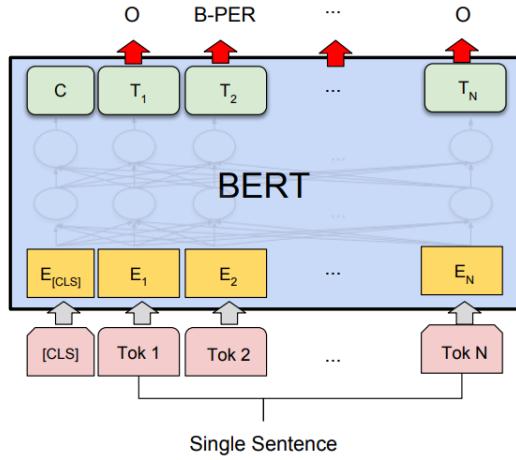


图 12.10 基于 SQuAD 问答数据集的问答任务精调阶段的输出。(a) 用于产生 start 标记；(b) 用于产生 end 标记。

(4) 序列标注任务。



(d) Single Sentence Tagging Tasks:
CoNLL-2003 NER

在序列标注任务中，如命名实体识别，输入端也是同分类任务一样喂入“文本序列 $- \emptyset$ ”对，即喂入的是一个单句。在输出端则每个对应的输出，加一个多类分类层，对应着标注结果。

4. 推断 (Inference)

模型推断（有的称为推理，因为 reasoning 被翻译成了推理，按照统计学的术语 Inference 翻译成推断比较好）就是使用模型的过程。即把数据喂入模型得到结果。

12.3.2 GPT

GPT(Generative pre-trained transformer)是由著名的人工智能公司 Open AI 开发的大模型。它的结构是基于 Transformer，不像 BERT 只使用了 Transformer 的 Encoder 部分，GPT 是 Decoder-only 的 Transformer。图 12.11 中，左图是 GPT 的结构，右图是一个 block 的结构。

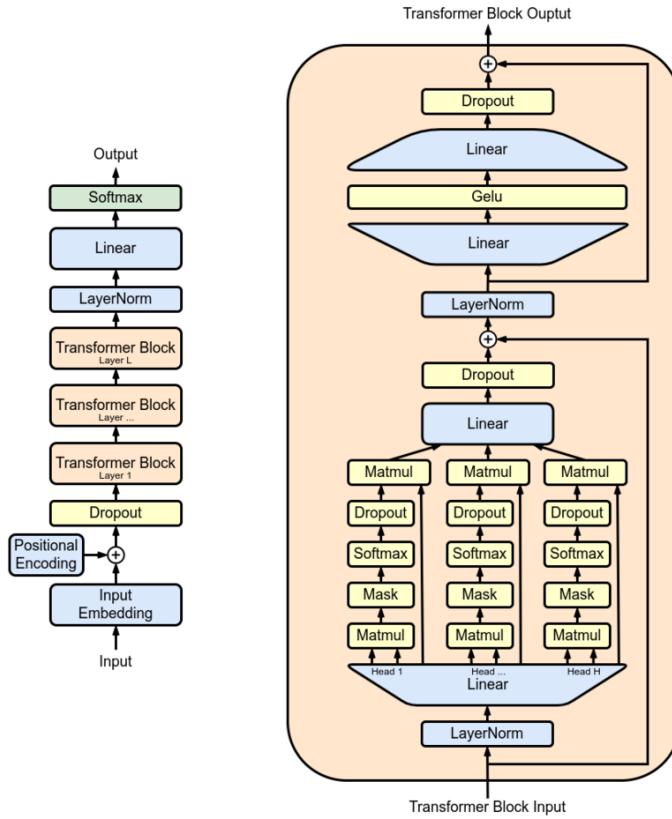


图 12.11 GPT 的结构

访问 Open AI 的 playground，提供了一个近乎 world knowledge 问答的系统。该系统就是基于 GPT-3 开发的。

Open AI 也基于 GPT 开发了一个自动生成代码的系统 CodeX。



GitHub Copilot 基于 CodeX 开发。

Open AI 基于大模型的图片生成系统 [DALL-E 2 \(openai.com\)](#)



基于 Web 的问答系统 WebGPT

[WebGPT | GPT-3 Demo \(gpt3demo.com\)](#)

OpenAI trained a research version of GPT-3 that can search the web, synthesize information, and cite its sources to provide more accurate answers to questions.

这个讲义将不过多的涉及大规模预训练语言模型 LLMs，相关的内容可以参看《大语言模型及其应用》讲义。下面主要介绍的预训练语言模型只涉及了 BERT 和 LLaMA。

第二节：认识 Hugging Face 的 Transformers 库

预训练模型库（或者 Transformers 库）有很多，最著名的是 Hugging Face。Hugging Face 是一家人工智能公司。Hugging Face ecosystem 包含 Transformers, Datasets , Tokenizers 和 Accelerate，也包含 Hugging Face Hub。

huggingface 的 Transformer 库 (<https://huggingface.co/transformers>) 截止 2023 年 10 月实现了包括 BERT, GPT, RoBERTa, XLM, DistilBert, XLNet 在内的 229 个预训练模型。该库可以在 pytorch 和 tensorflow 种使用。本章对该库的功能进行介绍。

huggingface 的 Dataset 库提供了各种数据集。

huggingface 的 Tokenizers 库提供了高性能的文本处理工具。

Accelerate 库是为 PyTorch 用户创建的，本文不讨论。

Hugging Face Hub 是一个网站 (<https://huggingface.co/models>) 提供了很多各种用途预训练好的模型（不限于 Transformer 模型）。

在 anaconda 下安装 Transformer，在 environment 中运行命令

```
conda install -c huggingface transformers
```

或者可以把使用 pip 命令安装

```
pip install transformers
```

更多信息见 <https://huggingface.co/docs/transformers/installation>。可以通过下面的程序查看是否安装成功

```
import transformers  
print(transformers.__version__)
```

13.1.1 pipeline

Hugging Face 的 Transformers 库（以下简称 Transformers 库）中最基本的对象是 pipeline，它是一种最容易使用 Transformers 库的方法。它给那些不需要了解模型细节，只想快速使用模型来完成某个任务的用户来使用。它连接一个模型和预处理步骤，以及后处理（postprocessing）步骤。我们可以输入任意的文本，然后得到想要的模型输出。例如，

```
from transformers import pipeline
classifier = pipeline("sentiment-analysis")
classifier([
    "I've been waiting for a HuggingFace course my whole life.",
    "I hate this so much!"
])
```

就可以得到情感分类结果和评分

```
[{'label': 'POSITIVE', 'score': 0.9598047137260437},
 {'label': 'NEGATIVE', 'score': 0.9994558095932007}]
```

上面的代码执行过程如下：根据参数("sentiment-analysis")，pipeline 选择了一个预训练好，**且精调了**的英文分类模型（还包含它的 tokenizer），下载到本地缓存起来。以后每次运行 pipeline 命令时，调用的是缓存的模型。

注：对于 pipeline("sentiment-analysis")默认下载的是模型“distilbert-base-uncased-finetuned-sst-2-english”。可以在 huggingface 的 models 里面进行查看

1. 面向多种任务的 pipeline

Transformers 库提供了很多种的 pipeline，它们可以完成不同的功能。创建 pipeline 对象的参数包括：

- "feature-extraction": 创建特征抽取 Pipeline.
- "text-classification": 文本分类器 Pipeline.
- "sentiment-analysis": 文本情感分类（是 text-classification 的别名）
- "token-classification": 词条分类 Pipeline.
- "ner": 命名实体识别 Pipeline（是 token-classification 的别名）。
- "question-answering": 问答 Pipeline.
- "fill-mask": 遮掩语言模型预测 Pipeline。就是在输入的文本中填空。
- "summarization": 文本摘要 Pipeline.
- "translation_xx_to_yy": 翻译模型 Pipeline.
- "text2text-generation": 文本到文本生成 Pipeline.
- "text-generation": 文本生成 Pipeline.

"zero-shot-classification": Zero Shot 分类 Pipeline.

"conversational": 多轮对话 Pipeline.

我们下面再看几个例子。

(1) zero-shot-classification 的例子。Zero shot learning (ZSL) 是说，一个训练好的模型，它们在测试阶段对样本分类的类别，在训练阶段并没有看到。Zero-shot-classificatin pipeline 允许用户在分类时自己规定标签。例如，

```
from transformers import pipeline
classifier = pipeline("zero-shot-classification")
classifier(
    "This is a course about the Transformers library",
    candidate_labels=["education", "politics", "business"],
)
```

创建一个 zero-shot-classification pipeline 后，在对一个句子进行分类时，自己规定了类别标签 candidate_labels= ["education", "politics", "business"]。分类结果是，

```
{'sequence': 'This is a course about the Transformers library',
'labels': ['education', 'business', 'politics'],
'scores': [0.8445963859558105, 0.111976258456707, 0.043427448719739914]}
```

它指示该矩阵属于每个类别的概率。

(2) 一个文本生成的例子，

```
from transformers import pipeline
generator = pipeline("text-generation")
generator("In this course, we will teach you how to")
```

用户在使用该文本生成 pipeline 时，需要先给一段文本，pipeline 会自动补充后面的文本。

```
[{'generated_text': 'In this course, we will teach you how to understand and use '
'data flow and data interchange when handling user data. We '
'will be working with one or more of the most commonly used '
'data flows — data flows of various types, as seen by the '
'HTTP'}]
```

(3) 填充 masked token 的例子

```
from transformers import pipeline
nlp_fill = pipeline('fill-mask', top_k=5)
r=nlp_fill("I am going to guess <mask> in this sentence")
for item in r:
    print(item)
```

运行程序会把，在输入句子在<mask>位置填充的词给显示出来。当前只显示最可能的 top 5 个句子。

2. 使用第三方的 pipeline 模型

上面的例子创建的 pipeline 使用的是默认模型，用户可以从 Hugging Face Hub (<https://huggingface.co/models>) 选择第三方模型。例如，我们使用 DistilGPT2 模型 (<https://huggingface.co/distilgpt2>)，它是一个预训练的 GPT2 模型，来生成文本。

```
from transformers import pipeline
generator = pipeline("text-generation", model="distilgpt2")
generator(
    "In this course, we will teach you how to",
    max_length=30,
    num_return_sequences=2,
)
```

更多使用 pipeline 的例子参看网站 <https://huggingface.co/course/chapter1/3?fw=pt>

第三节：使用 Transformers 库

13.2.1 Behind pipeline

虽然 13.1 节介绍的 pipeline 方法使用 transformer 模型很方便，但为了更灵活的应对面临的不同任务，本节学习 pipeline 的工作原理。Transformers 库提供一个单独的 API，可以帮助搭建自己的 pipeline。

第一节例子中的 pipeline

```
from transformers import pipeline
classifier = pipeline("sentiment-analysis")
classifier([
    "I've been waiting for a HuggingFace course my whole life.",
    "I hate this so much!",
])
```

实际上包含了三个步骤：（1）使用 tokenizer 进行预处理，（2）使用模型处理数据，（3）再处理得到的结果。如图 13.1。

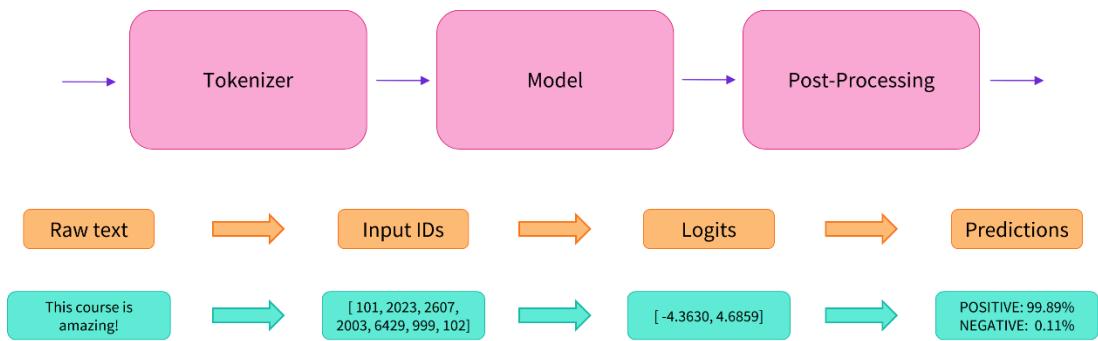


图 13.1 pipeline 内部的工作步骤

下面我们分别介绍，怎样拆解 pipeline 的这三个步骤。

1. 使用 tokenizer 进行预处理

Transformer 模型不能自己处理输入的原始文本。因此第一步是使用 tokenizer 将文本转换成编码序列。每个 Transformers 库的预训练模型有自己的 tokenizer 处理器（对象）。因此需要针对不同的模型，获得它们的 tokenizer 对象。可以使用 AutoTokenizer 类的 from_pretrained 方法完成。

```

from transformers import AutoTokenizer
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

"sentiment-analysis" pipeline 的默认 checkpoint 名称是" distilbert-base-uncased-
finetuned-sst-2-english "。

```

注：对于 checkpoint 的含义，我理解就是 huggingface 提供的不同版本的预训练模型，它们被分配了不同的名字。AutoTokenizer.from_pretrained(checkpoint) 表示获得预训练模型中的 tokenizer 对象。

一旦获得了 tokenizer 对象，则可以处理自己输入的文本。返回的结果是准备喂入模型的数据，是 dictionary 数据结构。

```

raw_inputs = [
    "I've been waiting for a HuggingFace course my whole life.",
    "you hate this so much!",
]
inputs = tokenizer(raw_inputs, padding=True, truncation=True,
return_tensors="tf")
print(inputs)

```

先介绍一下 tokenizer 对象。调用该对象当作方法使用时，包含的参数有

text (str, List[str], List[List[str]]) – The sequence or batch of sequences to be encoded. Each sequence can be a string or a list of strings (pretokenized string). If
--

the sequences are provided as list of strings (pretokenized), you must set `is_split_into_words=True` (to lift the ambiguity with a batch of sequences).

text_pair (str, List[str], List[List[str]]) – The sequence or batch of sequences to be encoded. Each sequence can be a string or a list of strings (pretokenized string). If the sequences are provided as list of strings (pretokenized), you must set `is_split_into_words=True` (to lift the ambiguity with a batch of sequences).

add_special_tokens (bool, *optional*, defaults to True) – Whether or not to encode the sequences with the special tokens relative to their model.

padding (bool, str or PaddingStrategy, *optional*, defaults to False) –

Activates and controls padding. Accepts the following values:

True or 'longest': Pad to the longest sequence in the batch (or no padding if only a single sequence if provided).

'max_length': Pad to a maximum length specified with the argument `max_length` or to the maximum acceptable input length for the model if that argument is not provided.

False or 'do_not_pad' (default): No padding (i.e., can output a batch with sequences of different lengths).

truncation (bool, str or TruncationStrategy, *optional*, defaults to False) –

Activates and controls truncation. Accepts the following values:

True or 'longest_first': Truncate to a maximum length specified with the argument `max_length` or to the maximum acceptable input length for the model if that argument is not provided. This will truncate token by token, removing a token from the longest sequence in the pair if a pair of sequences (or a batch of pairs) is provided.

'only_first': Truncate to a maximum length specified with the argument `max_length` or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the first sequence of a pair if a pair of sequences (or a batch of pairs) is provided.

'only_second': Truncate to a maximum length specified with the argument `max_length` or to the maximum acceptable input length for the model if that argument is not provided. This will only truncate the second sequence of a pair if a pair of sequences (or a batch of pairs) is provided.

False or 'do_not_truncate' (default): No truncation (i.e., can output batch with sequence lengths greater than the model maximum admissible input size).

max_length (int, *optional*) –

Controls the maximum length to use by one of the truncation/padding parameters.

If left unset or set to None, this will use the predefined model maximum length if a maximum length is required by one of the truncation/padding parameters. If the

model has no specific maximum input length (like XLNet) truncation/padding to a maximum length will be deactivated.

stride (int, *optional*, defaults to 0) – If set to a number along with max_length, the overflowing tokens returned when return_overflowing_tokens=True will contain some tokens from the end of the truncated sequence returned to provide some overlap between truncated and overflowing sequences. The value of this argument defines the number of overlapping tokens.

is_split_into_words (bool, *optional*, defaults to False) – Whether or not the input is already pre-tokenized (e.g., split into words). If set to True, the tokenizer assumes the input is already split into words (for instance, by splitting it on whitespace) which it will tokenize. This is useful for NER or token classification.

pad_to_multiple_of (int, *optional*) – If set will pad the sequence to a multiple of the provided value. This is especially useful to enable the use of Tensor Cores on NVIDIA hardware with compute capability >= 7.5 (Volta).

return_tensors (str or TensorType, *optional*) –

If set, will return tensors instead of list of python integers. Acceptable values are:

'tf': Return TensorFlow tf.constant objects.

'pt': Return PyTorch torch.Tensor objects.

'np': Return Numpy np.ndarray objects.

return_token_type_ids (bool, *optional*) –

Whether to return token type IDs. If left to the default, will return the token type IDs according to the specific tokenizer's default, defined by the return_outputs attribute.

return_attention_mask (bool, *optional*) –

Whether to return the attention mask. If left to the default, will return the attention mask according to the specific tokenizer's default, defined by the return_outputs attribute.

return_overflowing_tokens (bool, *optional*, defaults to False) – Whether or not to return overflowing token sequences.

return_special_tokens_mask (bool, *optional*, defaults to False) – Whether or not to return special tokens mask information.

return_offsets_mapping (bool, *optional*, defaults to False) –

Whether or not to return (char_start, char_end) for each token.

This is only available on fast tokenizers inheriting from PreTrainedTokenizerFast, if using Python's tokenizer, this method will raise NotImplementedError.

return_length (bool, *optional*, defaults to False) – Whether or not to return the lengths of the encoded inputs.

```
verbose (bool, optional, defaults to True) – Whether or not to print more information and warnings.
```

该对象被调用返回的数据格式是 BatchEncoding 是一个词典。

```
{'input_ids': <tf.Tensor: shape=(2, 16), dtype=int32, numpy= array([[ 101, 1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 12172, 2607, 2026, 2878, 2166, 1012, 102], [ 101, 2017, 5223, 2023, 2061, 2172, 999, 102, 0, 0, 0, 0, 0, 0, 0]])>, 'token_type_ids': <tf.Tensor: shape=(2, 16), dtype=int32, numpy= array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])>, 'attention_mask': <tf.Tensor: shape=(2, 16), dtype=int32, numpy= array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]])>}
```

词典结构中包含三个 key: 'input_ids'、'token_type_ids'和'attention_mask'。这里的 token id '101' 是 bert 在处理文本时自动插入到序列开始端一个标记 '[CLS]'，它的编号是 101。

2. 带入模型

Transformer 库提供了很多模型类，例如 TFAutoModel、TFBertModel。这些类可以理解为在通用的预训练模型上，加上了一个 head，以支持完成不同的任务。见下面的 model head。注意这里的 head 就是指神经网络的层，不是 Transformer 中的 head.

可以使用这些类下的 from_pretrained method 方法来下载预训练模型（也可以从本地装载模型）。

```
#紧接上面的代码
from transformers import TFAutoModel
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
model = TFAutoModel.from_pretrained(checkpoint)
```

模型的输出就是 hidden states 或称为 features。模型相当于完成对输入数据进行向量表示的工作，即将输入数据表示成了向量。模型的输出是 tensor，它包含三个维度：

- Batch size: The number of sequences processed at a time (2 in our example).
- Sequence length: The length of the numerical representation of the sequence (16 in our example).
- Hidden size: The vector dimension of each model input.

对于小模型，Hidden size 通常是 768，大模型可以达到 3072 甚至更大。

可以使用下面的语句来考察模型输出的 tensor 的 shape

```
#紧接上面代码
```

```
outputs = model(inputs)
print(outputs.last_hidden_state.shape)
```

或者 `print(outputs[0].shape)`

结果是

(2, 16, 768)

Model head 可以理解为在预训练的模型的隐层输出 hidden states 再加上的面向特定任务的层。

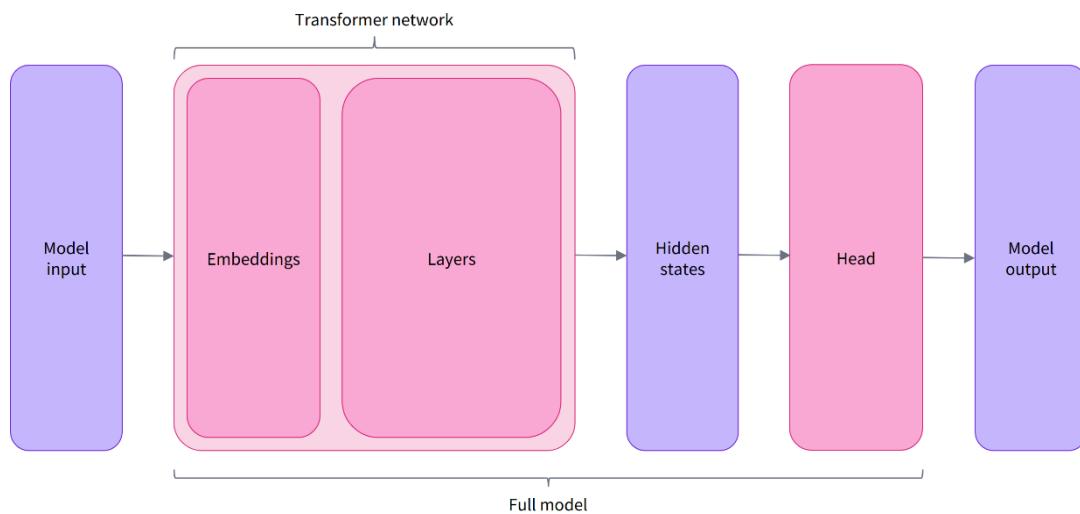


图 13.2 预训练模型精调的模型结构

Tramsnformer 库中完成不同任务的 Model head 有

- *Model (retrieve the hidden states)
- *ForCausalLM
- *ForMaskedLM
- *ForMultipleChoice
- *ForQuestionAnswering
- *ForSequenceClassification
- *ForTokenClassification

上面只列举了部分 model head。

使用 TFAutoModel 类得到的模型就是没有加 head 的部分。如果需要完成一个 sequence classification 的任务，我们需要获得以给 sequence classification head。此时使用 TFAutoModelForSequenceClassification 类获得预训练 transformers 模型再加上 head。

```
from transformers import AutoTokenizer
from transformers import TFAutoModelForSequenceClassification

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
```

```

tokenizer = AutoTokenizer.from_pretrained(checkpoint)

raw_inputs = [
    "I've been waiting for a HuggingFace course my whole life.",
    "I hate this so much!",
]
inputs = tokenizer(raw_inputs, padding=True, truncation=True,
return_tensors="tf")

model = TFAutoModelForSequenceClassification.from_pretrained(checkpoint)
outputs = model(inputs)
print(outputs)

```

注：用于问答的模型的头（Model head）ForQuestionAnswering 的使用，不在本讲义相信讨论。参见讲义《大语言模型与应用》

3. 模型输出处理

我们需要对模型的输出进行处理，以理解其含义。例如，上面程序在做 sequence classification 的输出是

```
array([[-1.560697 ,  1.6122818],
       [ 4.1692314, -3.3464484]], dtype=float32)>
```

它们不是概率，是 logit。为转换成概率，我们需要加一个 softmax 层。

注：所有的 transformer 的输出都是 logit，不是概率。

带入一个 softmax 层

```

#紧接上面的代码
import tensorflow as tf
predictions = tf.math.softmax(outputs.logits, axis=-1)
print(predictions)

```

其输出为，

```
[[4.0195331e-02 9.5980465e-01]
 [9.9945587e-01 5.4418348e-04]]
```

可以看到，在第每个句子上，给出的在不同类别标签的评分。为了知道，

[4.0195331e-02 9.5980465e-01]每个位置对应的类别标签，可以使用
model.config.id2label 来查看。

```
{0: 'NEGATIVE', 1: 'POSITIVE'}
```

因此，我们可以得出结论，第一个句子应该分类为 positive，第二个句子分类为 negative。

13.2.2 创建和使用模型

使用 TFAutoModel 类可以方便的从任意一个 checkpoint 初始化模型。但注意，像 13.2.1 节描述的，是一个无 head 的模型。TFBertModel 类则是装载 bert 模型。

1. 创建一个 Transformer

下面我们将初始化一个 bert 模型。

```
from transformers import BertConfig, TFBertModel
config = BertConfig()
print(config)
model = TFBertModel(config)
```

第一步是装载一个 configuration 对象，它包含了很多属性，即对模型的配置信息。

print(config)语句将显示

```
BertConfig {
    [...]
    "hidden_size": 768,
    "intermediate_size": 3072,
    "max_position_embeddings": 512,
    "num_attention_heads": 12,
    "num_hidden_layers": 12,
    [...]
}
```

上面的程序是直接从默认配置装载模型，模型参数被随机初始化。通常我们不会从零训练一个模型，因此，一般都是装载一个预训练模型。此时使用 from_pretrained 方法。（注：huggingFace 的预训练模型包含 pytorch 的和 tensorflow 的，其中以 TF 开头的模型是 Tensorflow 的，否则是 pytorch 的）

```
from transformers import TFBertModel
model = TFBertModel.from_pretrained("bert-base-cased")
```

该语句装载一个 tensorflow 的 Bert 模型，其中预训练部分来自于 “bert-base-cased”

注：我们前面讲的方法都是从网上装载预训练模型。在有些地方如果网络不行，

**generator = pipeline("text-generation", model="distilgpt2")
这样的命令就会显示出错。那是因为模型装载不了。**

此时，可以考虑把模型下载到本地，然后装载。步骤如下：

- (1) 在 huggingface.co/models 里面找到你要的模型，然后点进去，在模型的网页选择“files and version”。再进去后，可以看到该模型的文件。
- (2) 通常我们需要保存的是三个文件及一些额外的文件，第一个是配置文件；`config.json`。第二个是词典文件，`vocab.json`。第三个是预训练模型文件，如果你使用 pytorch 则保存 `pytorch_model.bin` 文件，如果你使用 tensorflow 2，则保存 `tf_model.h5`。额外的文件，指的是 `merges.txt`、`special_tokens_map.json`、`added_tokens.json`、`tokenizer_config.json`、`sentencepiece.bpe.model` 等，这几类是 `tokenizer` 需要使用的文件，如果出现的话，也需要保存下来。没有的话，就不必。
- (3) 下载这些文件时，应该右键点击下载按钮来下载。如果是右键点击文件，选另存为保存的则不是文件，而是网页。



- (4) 下载的模型文件，它的名字是随机生成的，需要用户自己改回。例如，下载的是 tensorflow 的模型，就改回 ‘tf_model.h5’

装载模型时，给出下载的模型所在的文件夹即可，例如

```
checkpoint = r"C:\Users\qjt16\Desktop\bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
classifier = TFAutoModelForSequenceClassification.from_pretrained(checkpoint)
```

13.2.3 精调一个预训练模型

本节讲述怎样在自己的数据集上精调一个预训练模型。

```
import tensorflow as tf
import numpy as np
from transformers import AutoTokenizer, TFAutoModelForSequenceClassification

checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = TFAutoModelForSequenceClassification.from_pretrained(checkpoint)
sequences = [
    "I've been waiting for a HuggingFace course my whole life.",
    "This course is amazing!",
]
```

```
batch = dict(tokenizer(sequences, padding=True, truncation=True,
return_tensors="tf"))

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
labels = tf.convert_to_tensor([1, 1])
model.train_on_batch(batch, labels)
```

上面的代码是一个简单的例子。步骤如下：（1）获得一个预训练模型的 tokenizer，和模型；（2）对自己的数据集应用 tokenizer 进行处理；（3）将处理结果转换成词典数据结构，作为训练模型（精调）时的数据集；（4）按照 keras 中的方法，编译模型；（5）将类别标签转换成 tensor；（6）训练模型

13.2.4 使用预训练模型作为定制模型的一层

使用预训练模型作为定制模型的一个层时，应该都是第一层。因此，都是要首先建立模型的输入层。输入层是用预训练模型的 tokenizer 处理文本后的数据作为输入。它包含三个部分：“input_ids”，“token_type_ids”，“attention_mask”。因此，自己的模型的输入层应该包括三个。

下面的例子是 Yelp 商品评论数据进行星级评分预测。详细代码参见 bert-rating.py。当前例子中，将星级评分预测的问题简单的看作是一个回归问题。

下面的程序是准备数据集

```
fin = open('../data/yelp-reviews.json')
lines = fin.readlines()
reviews = list()
stars = list()

for line in lines:
    rdict = json.loads(line)
    reviews.append(rdict['text'])
    stars.append(rdict['stars'])

np.random.seed(10)
shuffle_indices = np.random.permutation(np.arange(len(stars)))

x_shuffled = np.array(reviews)[shuffle_indices]
y_shuffled = np.array(stars)[shuffle_indices]

# Split train/test set
x_train, x_test = x_shuffled[:-1000], x_shuffled[-1000:]
y_train, y_test = y_shuffled[:-1000], y_shuffled[-1000:]
```

关键的一步是准备 bert 模型需要的输入数据。需要使用 bert 模型的 tokenizer 类对文本进行预处理

```
local = "bert-base-uncased"
encodings = tokenizer(x_train.tolist(), truncation=True, padding='max_length',
max_length=seq_len)
X_train = [np.array(encodings["input_ids"]),
           np.array(encodings["token_type_ids"]),
           np.array(encodings["attention_mask"])]  
  
encodings = tokenizer(x_test.tolist(), truncation=True, padding='max_length',
max_length=seq_len)
X_test = [np.array(encodings["input_ids"]),
           np.array(encodings["token_type_ids"]),
           np.array(encodings["attention_mask"])]
```

BertTokenizer.from_pretrained(local)获得一个 bert 预训练模型的 tokenizer 对象（从网上下载）。该 tokenizer 对象对保存在列表结构中的文本数据进行预处理。处理后的结果 encoding 是一个词典，包含三个部分：input_ids, token_type_ids 和 attention_mask。把这三个部分转换成 numpy 的 ndarray 数据结构，并放入列表作为输入模型的数据。

建立模型时，首先建立输入层，有三个输入层，对应上面产生的输入数据的三个部分。

```
input_ids = Input(shape=(seq_len,), dtype=tf.int32,
name='input_ids')
input_type = Input(shape=(seq_len,), dtype=tf.int32,
name='token_type_ids')
input_mask = Input(shape=(seq_len,), dtype=tf.int32,
name='attention_mask')
inputs = [input_ids, input_type, input_mask]
```

然后获得预训练的模型

```
bert = TFBertModel.from_pretrained(local)
```

因为这是一个回归任务，所以在 bert 模型的最后一层的多个时间步的输出向量上做平均，得到一个对输入文本的编码（一个表示向量），然后带入一个全连接层

```
bert_outputs = bert(inputs)
last_hidden_states = bert_outputs.last_hidden_state
avg = GlobalAveragePooling1D()(last_hidden_states)
output = Dense(1)(avg)
```

在编译模型时，我们使用 MAE 和 RMSE 评价模型的性能

```
opt = tf.keras.optimizers.Adam(learning_rate=3e-5)
model = Model(inputs=inputs, outputs=output)
rmse=tf.keras.metrics.RootMeanSquaredError()
model.compile(loss='mse', optimizer=opt, metrics=['mae', rmse])
```

特别强调的是：预训练模型精调时，学习率一定要设的很小，比如当前例子中用的是 $3e-5$ 。大家可以试一下，如果使用默认的 adam 学习率（是 0.001），精调的模型对所有的输入数据产生相同的输出。

在精调阶段，就是使用准备好的数据集训练模型。

```
model.fit(X_train, y_train, epochs=epoch, batch_size=batch_size,
verbose=1)
model.evaluate(X_test,y_test,verbose=1)
res=model.predict(X_test)
for i in range(len(y_test)):
    print("%s, %s"%(res[i], y_test[i]))
```

注意：如果想设置 bert 模型在精调阶段是不可训练的 untrainable，可以

```
bert = TFBertModel.from_pretrained(checkpoint)
for layer in bert.layers:
    layer.trainable = False
```

13.2.5 一些出错的解决

1. 在学院的服务器上使用预训练模型时，出现不能连接或下载预训练模型的情况。我则把模型下载到本地。
2. 在我的一个模型上，使用"bert-base-uncased"预训练模型可以，但使用"distilbert-base-uncased"则出现错误信息

NotImplementedError: in user code:

这是因为在使用

```
bert=TFAutoModel.from_pretrained(checkpoint)
```

来得到预训练模型，改为

```
bert = TFBertModel.from_pretrained(checkpoint)
```

就好了

3. Process finished with exit code 137

内存耗用太大，被 OS 把进程终止了。

5. 构建一个模型时，包含了 `huggingface bert`。训练完的模型想保存，适用 keras 的 `model.save`，但是出错。

可以采用只保存模型的权重 `model.save_weights('model_weight.h5')`。在另外的程序中使用该模型时，重新创建模型，但不训练，而是用 `model.load_weights('model_weight.h5')`装载权重后，就恢复了该模型。

第四节：实例 1：基于 Bert 预测 masked token

Bert 是一个 masked LM。下面我们用 bert 来预测输入的句子中被遮掩的词是什么。见本书 github 网站上的文件 `bert-masked-word.py`

```
from transformers import BertTokenizer
from transformers import TFBertForMaskedLM, TFBertModel
import tensorflow as tf

checkpoint = "bert-base-uncased"
tokenizer = BertTokenizer.from_pretrained(checkpoint)
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    model = TFBertForMaskedLM.from_pretrained(checkpoint)

text = "his mother is a [MASK]."
tokenized_text = tokenizer.tokenize(text)
inputs = tokenizer(text, padding=True, truncation=True, return_tensors="tf")
masked_index = tokenized_text.index("[MASK]")
indexed_tokens = tokenizer.convert_tokens_to_ids(tokenized_text)
outputs = model(inputs)
predictions = outputs[0]

m = predictions[0,masked_index+1,:]
out = tf.keras.backend.softmax(m)
r = tf.math.top_k(out, k=5)
idx = r.indices.numpy()
for i in idx:
    word = tokenizer.convert_ids_to_tokens([i])[0]
    print(word)
```

程序运行结果是：

```
nurse
lawyer
teacher
singer
doctor
```

注意：上面的例子中 masked_index 的值是 4，但是每个喂入 bert 的序列会在头和尾分别加入两个标记[cls]和[sep]。因此，考察 predictions 的 shape 会发现模型输出的长度是 8，虽然输入序列的长度是 6（包括了句号）。

另外，因为 bert 模型是 masked LM。即每个时间步输出是对该时间步输入的 word 的预测。因此，在获取 bert 对输入的[MASK]的预测时，是

```
m = predictions[0,masked_index+1,:]
```

因为，我们当前采用的是 TFBertForMaskedLM，即在 bert 的输出上加了一个用于 masked LM 的 head。因此，每个输出的向量的维度是词汇表的大小。当前是 30522。大家可以试一试，如果是使用 TFBertModel，它的一个时间步输出的维度是 768，即隐层神经元的个数。

大家可以再试一试下面的文本

"he loves this movie? [MASK], he hates it."

注意，标点符号要写全。

第五节：实例 2：基于 LLaMA 的小样本学习

13.4.1 元学习、小样本学习、单样本学习和零样本学习

1. 元学习

元学习 (Meta Learning) 是机器学习的一个子领域。简单的说，是从前面的任务中学习知识，再基于学习到的知识在新的数据集上泛化到新的任务。（**注意，这里新的任务可以是一个小的概念，如在将一个分类器泛化到新的类别上，也叫新的任务，下面的讨论都是这样的任务**）

与单独用新数据集学习一个新的模型不同，元学习是泛化到新的任务，即仍保留了原来任务上的能力。元学习包括两个阶段：

- (1) meta-training：在有 N 个类别的训练集上训练一个模型 θ ；
- (2) meta-testing：在新类 (Novel classes) 即训练阶段没见过的类别的数据集上精调模型 θ 。并在新类上评估模型性能。

元学习的形式化定义如下。针对两个学习阶段，有两个数据集 $D_{meta-train}$ 和 $D_{meta-test}$ 。每个数据集又分别划分成训练集 D^{train} 和测试集 D^{test} 。两个数据集面对的任务不同。在 meta-training 阶段用数据集 $D_{meta-train}$ 训练模型 θ

$$\theta = \arg \min_{\theta} \sum_{\mathcal{D}_i \in \mathcal{D}_{meta-train}} \sum_{(x,y) \in \mathcal{D}_i^{test}} \mathcal{L}(f(\mathcal{D}_i^{train}, x; \theta), y).$$

元学习的任务是

$$y \approx f(\mathcal{D}_i^{train}, x; \theta) \text{ where } (x, y) \in \mathcal{D}_i^{test}$$

其中，

$$\mathcal{D}_i = \{\mathcal{D}_i^{train}, \mathcal{D}_i^{test}\} \text{ where } \mathcal{D}_i \in \mathcal{D}_{meta-test}$$

在第二阶段使用数据集 $\mathcal{D}_{meta-test}$ 对第一阶段训练的模型 θ 进行评估。第二阶段是否进行精调，其实各种研究有的需要，有的不需要。

对于元学习也有不同看法，例如 “Pattern Classification Using Ensemble Methods” 一书，提到 “Stacking is probably the most-popular meta-learning technique”。它将集成技术中的 stacking 也看作是一种元学习。

2. 小样本学习

小样本学习 (Few-Shot Learning, FSL) 简单的说就是，给定数据丰富的基类，FSL 的任务是泛化到去识别训练样本有限的新类。FSL 有两个阶段，第一阶段从丰富的基类中学习，第二阶段泛化到新类。

可以看到 FSL 和元学习很像，不过 FSL 在新类上的数据集比较小，所以才称为是小样本学习。另外，FSL 也可以是指在预训练的模型上基于小样本泛化到新的任务。

FSL 中的几个概念：

(1) 支持集 (support set)。支持集里包含少量的标注的样本，每个新的类别对应几个样本。预训练模型将使用支持集泛化到新的类别上。

(2) 查询集 (query set)。查询集包含的样本既有新的在支持集中出现类别，又可以有旧的类别，即在模型预训练阶段见过的类别。

(3) N-way K-shot 学习模式 (schema)。是指支持集中，有 N 个新的类别，每个类别有 K 个样本。

FSL 包含三大类方法：

(1) Initialization based methods。在第一阶段首先学习一个好的初始模型 θ 。第二阶段在新类上精调 θ 。

(2) Distance metric learning based method。第一阶段学习一个“比较模型”，第二阶段用查询与支持集比较，然后对查询分类。此种方法中，第二阶段没有精调模型。

(3) Hallucination based method。第一阶段基于丰富的基类学习一个数据生成器。第二阶段用数据生成器在小样本上进行 data augmentation。然后用新生成的数据加基类数据去训练分类模型。

当把 FSL 应用在 NLP，FSL 的第一阶段也可以是一个预训练的语言模型。提示工程中的示例可以看作是 FSL 第二阶段的小样本。

3. 单样本学习

单样本学习 (one-shot learning, OSL) 可以看作是 FSL 的特例。即在第二阶段 N-way K-shot 学习模式中， $N=1$, $K=1$ 。FSL 中的 Initialization based methods 和 Hallucination based method 方法就不适合。如此，OSL 不是关于分类和目标识别的，它就是关于对象匹配的。即一个 query 和已有的数据中的哪一个匹配。例如，人脸识别是将拍摄到的人脸和数据库中保存的人脸进行匹配，找到最相似的，并评估是否一致的概率。因此大部分的 OSL 和 Distance metric learning based method 是一致的，但也有自己的特殊之处。下面是一个 Siamese Neural Network 做 OSL 的例子。参见论文 “Siamese Neural Networks for One-shot Image Recognition”

Siamese Neural Network 的结构如图。

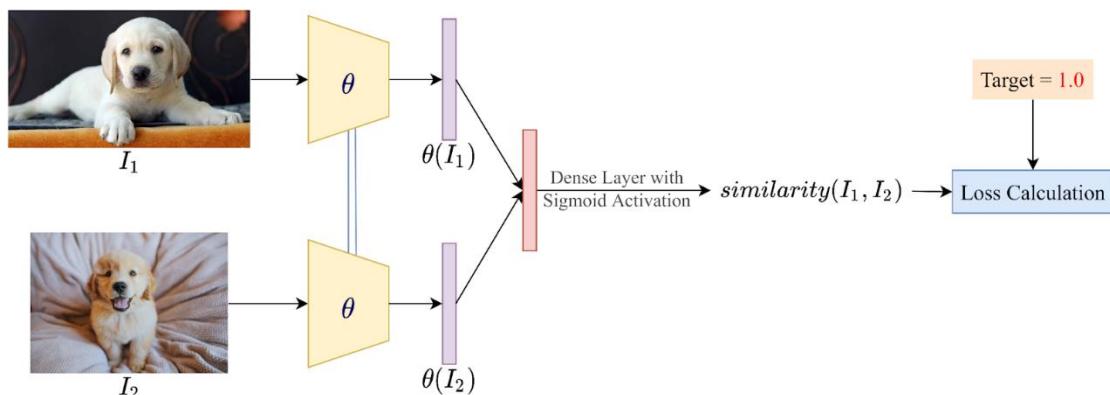


图 13.3 Siamese 网络 OSL 的结构

它主要结构是两个卷积神经网络，这两个网络共享权重。需要比对的两个图片，送入网络后被编码，然后再计算它们编码向量的欧式距离，再经过一个 sigmoid 层，产生两个图片相似的概率。新来一个 query，会和支持集中的图片（每个类别一张）进行比对。然后产生一个输出概率。选择概率最高的，或者概率低于 0.5 就没有匹配成功。

4. 零样本学习

零样本学习 (Zero-shot learning, ZSL) 是将预训练的模型泛化到新的类别，而新的类别没有任何样本，只有描述信息。

ZSL 中有三类数据：

- (1) seen classes：训练集
- (2) unseen classes：需要分类的新类别数据。
- (3) 辅助信息：关于 unseen classes 的描述或语义信息。

ZSL 包括两个阶段：(1) 使用 seen classes 训练模型；(2) 使用 unseen 的辅助信息对 unseen classes 中的数据进行分类。

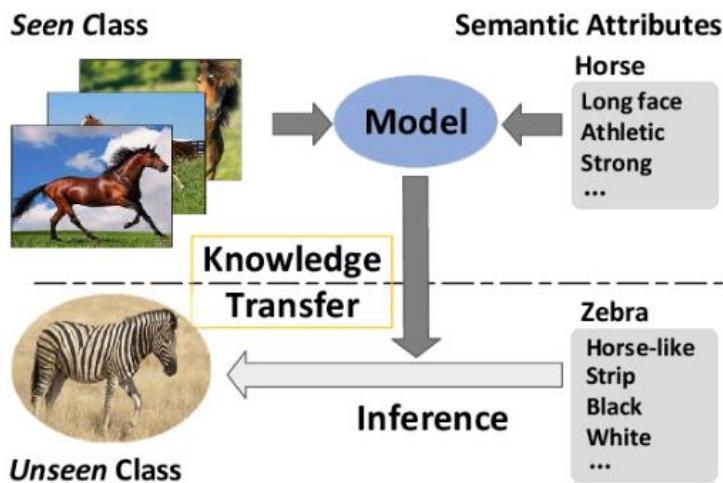


图 13.4 ZSL 示意图

它的工作过程可以用上面的图结合下面的故事来描述。

小明和爸爸到了动物园，看到了马，然后爸爸告诉他，这就是马；之后，又看到了老虎，告诉他：“这种身上有条纹的动物就是老虎。”；最后，又带他去看了熊猫，对他说：“你看这熊猫是黑白色的。”然后，爸爸给小明安排了一个任务，让他在动物园里找一种他从没见过的动物，叫斑马，并告诉了小明有关于斑马的信息：“斑马有着马的轮廓，身上有像老虎一样的条纹，而且它像熊猫一样是黑白色的。”最后，小明根据爸爸的提示，在动物园里找到了斑马。

前面我们讲过，用 Siamese 网络做 one-shot 学习。与 ZSL 的区别是，在推断阶段 OSL 还是有关于新类的图片，虽然只有一张。而 ZSL 一张都没有，只有描述信息。

13.4.2 LLaMA

LLaMA 是 META 开发的一个 foundation language models 集合，包含 7B 到 65B 参数的模型。作者声称

" LLaMA-13B outperforms GPT-3 (175B) on most benchmarks, and LLaMA-65B is competitive with the best models," 。

LLaMA 模型的权重可以被精调。这允许开发者创建更高级的可以和用户进行自然语言交互的模型。例如，聊天机器人、虚拟助理。

使用 LLaMA 的硬件需求：NVIDIA GPU(s) with a minimum of 16GB of VRAM.

Huggingface 提供了很多版本的 LLaMA 模型。比如 openllama 是对 llama 的复现

https://github.com/openlm-research/open_llama

LongLLaMA 是对 openllama 进一步的精调。在 huggingface 中使用这些模型，给出模型的 checkpoints 即可，例如

```
model_path = 'openlm-research/open_llama_7b_v2'
tokenizer = LlamaTokenizer.from_pretrained(model_path)
model = LlamaForCausalLM.from_pretrained(
    model_path, torch_dtype=torch.float16, device_map='auto',
)
```

但是因为国内的服务器直接访问不了，我下载了一个 open_llama-7b 的版本。

model_path 给出模型所在的具体的路径，例如，

```
from transformers import LlamaForCausalLM, LlamaTokenizer
checkpoint = '/root/autodl-tmp/llama-7b'
tokenizer = LlamaTokenizer.from_pretrained(checkpoint)
model = LlamaForCausalLM.from_pretrained(checkpoint)
```

13.4.3 基于 LLaMA 的小样本文本多分类

下面使用 open_llama-7b 完成一个小样本的文本多分类任务。因为 open_llama 只提供了 pytorch 的模型。因此，下面的代码是基于 pytorch 的。

我们采用的是大模型中的 In-Context-Learning (ICL) 技术，它是一种大模型上的小样本学习技术。即大模型在看到几个例子后，可以完成之前未被训练过的任务，而模型的参数没有被更新。下面是用 ICL 在 LLaMa 模型上完成情感分类任务，包括三个标签：Positive, Neutral 和 Negative。

```
import torch
import os
import numpy as np
from transformers import AutoTokenizer, AutoModelForCausalLM
from transformers import LlamaForCausalLM, LlamaTokenizer

checkpoint = '/root/autodl-tmp/llama-7b'
tokenizer = LlamaTokenizer.from_pretrained(checkpoint)
model = LlamaForCausalLM.from_pretrained(checkpoint,
    torch_dtype=torch.float16, device_map='auto')
```

```

model = model.to(device)

prompts = """
###
1. Tweet: "I hate it when my phone battery dies."
Sentiment: Negative
###
2. Tweet: "My day has been 👍"
Sentiment: Positive
###
3. Tweet: "This is the link to the article"
Sentiment: Neutral
###
4. Tweet: "This new music video was incredibile"
Sentiment:""""

input_ids = tokenizer(prompts, return_tensors="pt").input_ids
generation_output = model.generate(
    input_ids=input_ids.to(device), max_new_tokens=32
)
print(tokenizer.decode(generation_output[0]))

```

上面的代码，首先装载 llama 模型，

```
model = LlamaForCausalLM.from_pretrained(checkpoint,
torch_dtype=torch.float16, device_map='auto')
```

可以看到，这是 Causal 语言模型，即给定文本，它会产生紧接该文本的内容。

紧接着创建 ICL 中的例子，我们给出了几个正向、中性和负向情感的文本。最后给了
一段文本（第 4 句），我们需要知道它的情感倾向。因此，“Sentiment:” 后面就没有
内容了。需要让模型自动产生后面的内容。

然后调用 llama 模型提供的文本处理器 tokenizer。再调用模型的 generate 函数去生成
后面的内容。生产的的内容如下：

```

<S>
###
1. Tweet: "I hate it when my phone battery dies."
Sentiment: Negative
###
2. Tweet: "My day has been 👍"
Sentiment: Positive
###
3. Tweet: "This is the link to the article"
Sentiment: Neutral
###
4. Tweet: "This new music video was incredibile"
Sentiment: Positive
###
5. Tweet: "I'm so excited for this new movie"
Sentiment: Positive

```

```
###  
6.
```

其中的第 4 条是我们需要模型预测的，它产生了预测结果 Positive。另外，他也多生成了内容，我们忽略。

注：model.generate(input_ids=input_ids.to(device), max_new_tokens=32)
max_new_tokens 是生成的 token 数。上面的例子，生成的 token 太多，你可以调节该参数，以简化得到的 answer。

练习：有一个收集的 twitter 的数据集。需要预测 tweets 的内容是否是和中国相关的。

13.4.4 基于 LLaMA 的问答

可以使用 LLaMa 完成简单的问答

```
import torch  
prompt = 'Q: What is the largest animal?\nA:'  
input_ids = tokenizer(prompt, return_tensors="pt").input_ids  
  
generation_output = model.generate(  
    input_ids=input_ids.to(device), max_new_tokens=128  
)  
print(generation_output)  
print(tokenizer.decode(generation_output[0]))
```

第十四章：自编码器

本章将介绍如何将文本通过无监督的方式进行编码，即生成一个表示向量，以供下游任务使用。这个技术称为自编码器（Autoencoder）。自编码器的一些变体用到了生成模型（Generative Models）的技术。在生成模型的大家族里面，有两个家族特别著名，分别是变分自编码器（Variational Auto Encoder, VAE）和生成对抗网络（Generative Adversarial Networks, GAN）。本章也将介绍 VAE、GAN 和它们在文本自编码器上的应用。

第一节：自编码器

Autoencoder 翻译作自编码器，是将图片、文本等数据编码，产生表示向量的一种方法。AutoEncoder 是一个 Encoder-Decoder 模型。Encoder 把输入编码成一个向量，Decoder 再解码该向量，还原成与输入相似的数据。这个过程中不需要任何目标数据，因此它是一种无监督学习方法。AutoEncoder 的目的是获得数据的表示向量。这样的操作有很多用途，例如，数据压缩、可视化、特征学习等。

形式化的描述，设 x 是输入，Encoder 用一个函数 q 表示， $h = q(x)$ 就是获得的数据表示向量。Decoder 用函数 g 表示， $r = g(h)$ 是解码器重构后的数据。Autoencoder 学习使得 $g(f(x)) \approx x$ 。

下面在 MNIST 数据集上构建了一个简单的 AutoEncoder，用于将图片投影到二维空间，可视化它们的分布。图 14.1 是构建的 AutoEncoder 模型，它包含一个 Encoder，将一个图片（ $28*28$ 的图片转换成了长度为 784 的向量），压缩到二维空间。然后再解码回长度为 784 的向量。该 AutoEncoder 的目的就是将一张图片投影到二维空间。

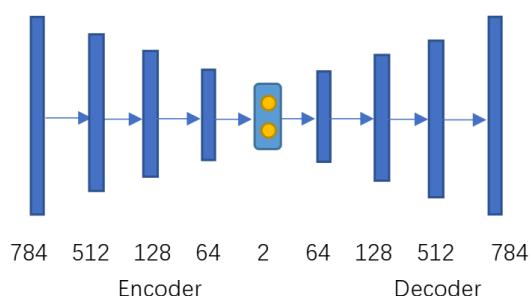


图 14.1 一个简单的图像 AutoEncoder

图 14.2 是 MNIST 中图片投影到二维空间后，它们的分布。每种颜色对应一种数字。左图对应模型未训练时，随机的投影；右图是在迭代 20 趟后训练的 AutoEncoder 模型对图片的投影。可以看见，AutoEncoder 获得的图片数据的表示在二维空间的分布是有规律的，每个类差不多分布在不同的空间。即 Autoencoder 学习到了或抽取出了手写数字图片的特征。

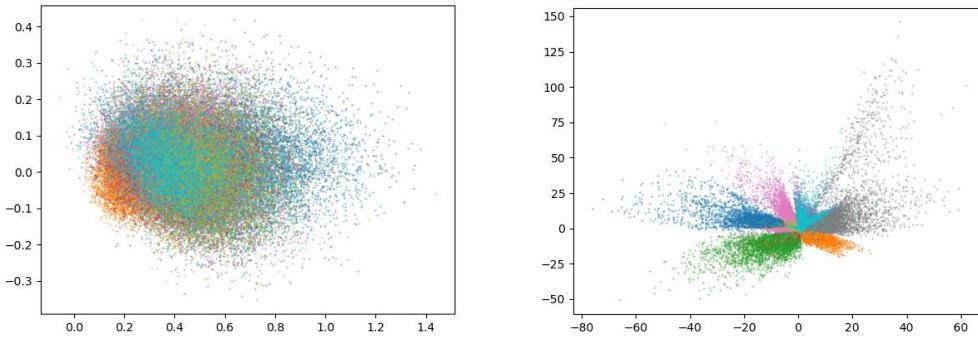


图 14.2. MNIST 数据集在二维空间的分布

建立模型和训练模型的代码如下。完整的代码见文件 `mnist-autoencoder.py`。该程序中 `encoding_dim` 为 2，即用一个长度为 2 的向量表示图片；使用 MSE 作为损失函数，模型的输入 `autoencoder.fit(x_train, x_train, ...)` 即用解码器的输出和输入计算误差平方均值。

```
input_img = keras.Input(shape=(784,))
n_layer1 = BatchNormalization()(input_img)
encoded1 = Dense(512, activation='relu')(n_layer1)
encoded2 = Dense(128, activation='relu')(encoded1)
encoded3 = Dense(64, activation='relu')(encoded2)
encoded = Dense(encoding_dim)(encoded3)

decoded1 = Dense(64, activation='relu')(encoded)
decoded2 = Dense(128, activation='relu')(decoded1)
decoded3 = Dense(512, activation='relu')(decoded2)
n_layer2 = BatchNormalization()(decoded3)
decoded = Dense(784, name="decoder",
activation='sigmoid')(n_layer2)

autoencoder = Model(input_img, decoded)
encoder     = Model(input_img, encoded)

mse = tf.keras.losses.MeanSquaredError()
op  = tf.keras.optimizers.Adamax(learning_rate=lr)
autoencoder.compile(optimizer=op, loss=mse)

(x_train, y_train), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
```

```

x_train = x_train.reshape((len(x_train),
np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test),
np.prod(x_test.shape[1:]))

autoencoder.fit(x_train, x_train, epochs=epochs,
batch_size=batch_size, shuffle=True, verbose=1)

```

第二节：定制训练过程

在开发更复杂的模型时，很有可能需要定制训练过程。例如，需要在每个 batch 上，实现每个 batch 上更新模型参数。这就需要在 tensorflow 上操作时不能使用 model.fit()。而是要定制的训练过程。

定制实现每个 batch 上的更新模型参数，需要两个关键步骤。一是怎样产生每个 batch，二是在每个 batch 里把训练数据带入模型得到模型输出怎样计算了 loss 后更新模型参数。

可以想象，需要一个循环，遍历每个 batch，此时可以用 tf.data.Dataset 来实现。首先创建好 numpy array 结构的数据，设定好每个 batch 的大小，下面创建 dataset = tf.data.Dataset.from_tensor_slices(x_train).batch(batch_size) 下一步用 for 循环就可以遍历每个 batch。下面的例子是把第一节的 autoencoder，用逐个 batch 进行训练的代码

```

dataset = tf.data.Dataset.from_tensor_slices(x_train).batch(batch_size)
for epoch in np.arange(epochs):
    for x in dataset:
        z = np.random.normal(size=(len(x),encoding_dim))
        with tf.GradientTape() as gen_tape:
            x_output = autoencoder(x, training=True)
            gen_loss = mse(x, x_output)
            gradients_of_generator = gen_tape.gradient(gen_loss,
autoencoder.trainable_variables)
            op.apply_gradients(zip(gradients_of_generator,
autoencoder.trainable_variables))

        v_loss = autoencoder.evaluate(x_test, x_test, verbose=0)
        print(f'epoch: {epoch}, training loss: {gen_loss}, validation: {v_loss}')

```

定制训练过程的第二步是，每个 batch 计算 loss，然后更新模型参数。因此，需要根据模型计算的损失值计算梯度，称为自动微分。先看一下 tf.GradientTape()
TensorFlow provides the tf.GradientTape API for automatic differentiation; that is, computing the gradient of a computation with respect to some inputs, usually tf.Variables. TensorFlow "records" relevant operations executed inside the context of a tf.GradientTape onto a "tape". TensorFlow then uses that tape to compute the gradients of a "recorded" computation using reverse mode differentiation.

因此，把模型计算和损失值的计算放到 `tf.GradientTape` 的 context，`GradientTape` “记录” 计算过程。

```
with tf.GradientTape() as gen_tape:  
    x_output = autoencoder(x, training=True)  
    gen_loss = mse(x, x_output)
```

再使用该 `GradientTape` 计算一个被“记录”的计算过程的梯度。

最后使用优化函数的 `apply_gradients` 方法更新模型参数

```
op.apply_gradients(zip(gradients_of_generator, autoencoder.trainable_variables))
```

第二种，在一个 batch 上训练模型的方法，是用 `model` 的 `train_on_batch` 方法。它应用在训练 GAN 的生成器和判别器时，分开构建的生成器和判别器模型，详见下一节。

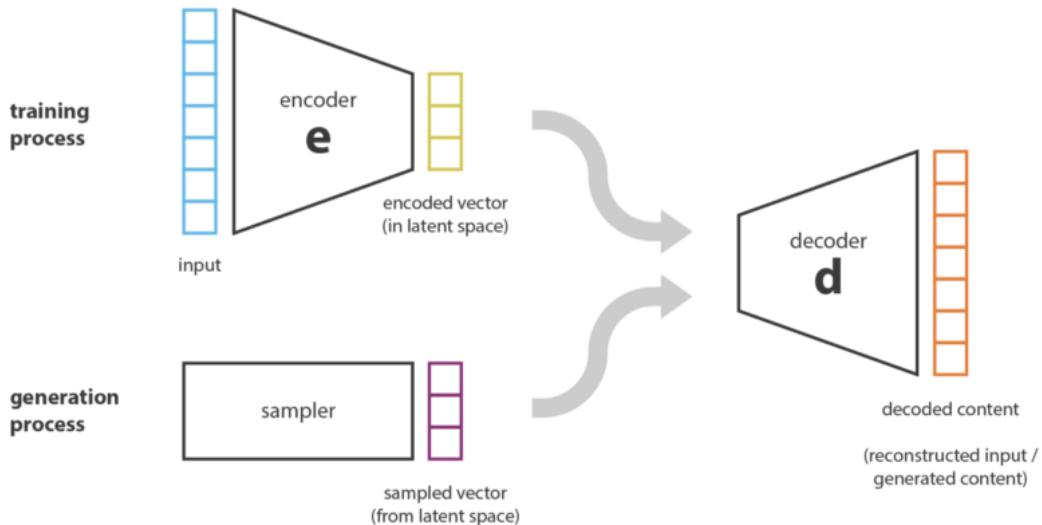
第三节：变分自编码器

第一节介绍了自编码器。可以了解到，自编码器用 Encoder-Decoder 的方式训练。编码器部分可以将输入数据映射到隐空间，得到它的表示向量。但自编码器被认为只能简单地“记住”数据，不能生成数据。因此，变分自编码器（Variational AutoEncoder, VAE）出现了，这是 AE 家族里面非常秀的一员了，它是一种生成模型。本节将从基本原理开始介绍 VAE，并给出 VAE 的实现。

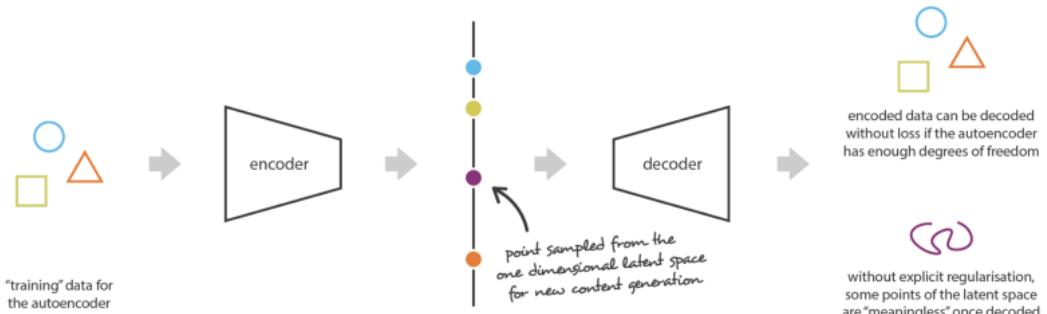
以下内容部分选自网文：<https://towardsdatascience.com/understanding-variational-autoencoders-vae-f70510919f73>

14.3.1 自编码器的局限

当训练了一个自编码器，就拥有了一个编码器和一个解码器。编码器将输入数据映射到了一个隐空间，即编码。如果训练过程保证了隐空间是规则的（regular，输入数据被合理的映射到隐空间，隐空间的每个位置都是有意义的），则可以从隐空间随机采样得到中间数据，再喂入到解码器，则可以生成内容。



但在自编码器中，怎样让隐空间规则性是一个问题。它取决于输入数据在原始空间的分布、隐空间的维度、编码器的结构。可以看一个例子，一个编码器是有能力安排任意多个输入数据到隐空间的一个位置，解码器也可以从这个隐空间的位置无损失的重构输入数据。但自编码器的自由度越高，越可以使得编码器-解码器的重构损失值越小，但有更高的过拟合的风险。即，隐空间中的一些点是无意义的，在这些点是不能生成有意义的内容的。



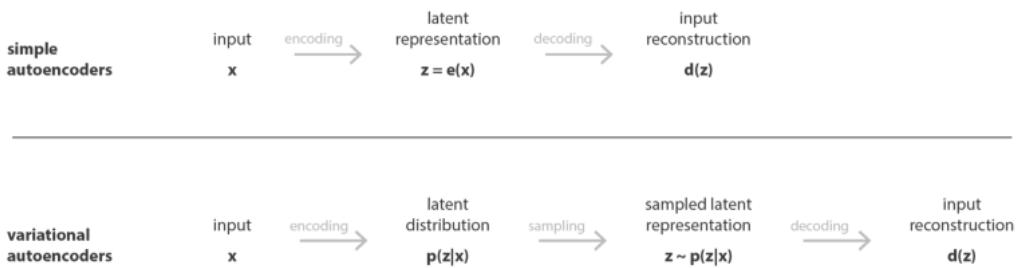
这个问题是由于 autorencoder 在训练过程中只考虑重构损失，而没有考虑怎样规则的组织隐空间。

14.3.2 变分自编码器的定义

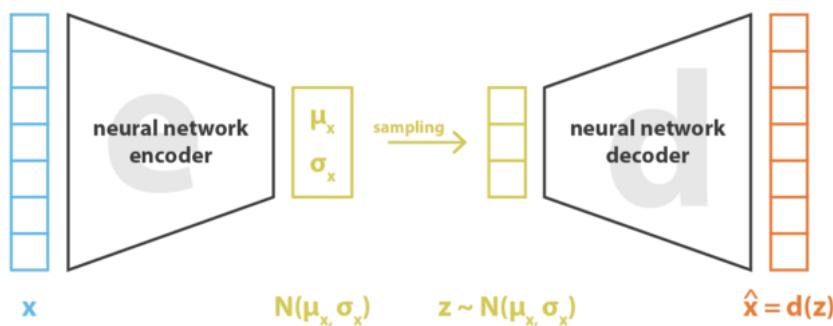
为了将 autorencoder 中的解码器用于生成内容，必须确信隐空间是足够规则化的，即每个位置都是有语义的。要达到这个目的就需要在训练过程引入“正则化 (regularization)”。简单的说，变分自编码器就是在自编码器的训练过程中引入了一个引导过程以防止过拟合，并保证隐空间有足够的特性可以进行内容生成。不过，不像 AE 的编码器将输入数据编码成隐空间的一个位置，VAE 的编码器是产生在隐空间的一个概率分布。模型的训练过程如下：

- 输入被编码为隐空间的一个分布
- 按照该分布从隐空间采样一个点，

- 将该点解码，计算重构损失
- 重构损失反向传播



在实践中，这个分布通常选择是正态分布。如此，编码器的输出是描述正太分布的均值和协方差矩阵。如此，可以强迫编码器的输出是符合正太分布的，这就可以规范的组织隐空间（regularizing）。方法是计算编码器输出的分布和标准正太分布的 KL 散度，然后作为正则化项添加在损失函数里。



$$\text{loss} = \|x - \hat{x}\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)] = \|x - d(z)\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)]$$

关于这个正则化过程的定性理解如下：

隐空间的规范性包括两个方面：

- 连续性：隐空间中相邻的两个点应该语义上是相近的
- 完整性：隐空间中按照给定的分布抽样得到的点，经过解码后得到的内容应该是有意义的。



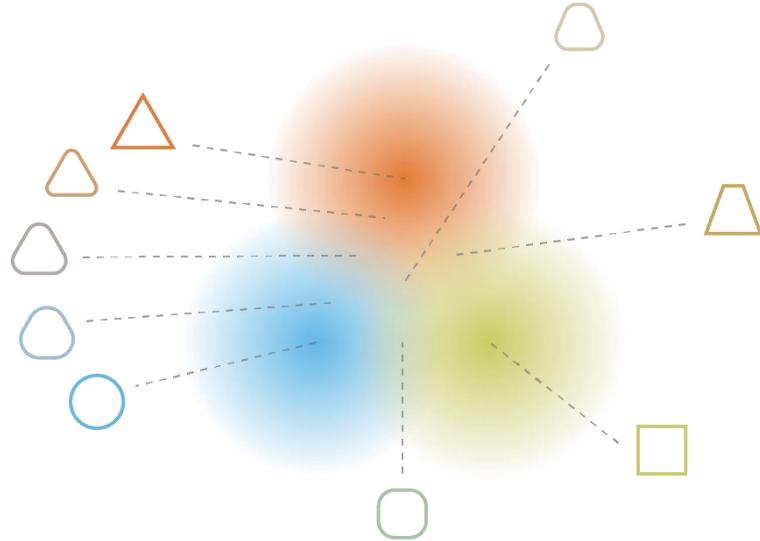
如果没有很好的定义正则项，VAE 的编码器将输入数据编码得到分布的均值和方差还并不能保证上面两点。此时，VAE 编码器返回的方差或者很小，或者返回的均值差异很大（数据在隐空间的分布相距很远）。这不能保证连续性和完整性。

为了避免这一问题，必须对编码器返回的协方差矩阵和均值进行正则化。实践中，正则化的操作是通过强迫分布接近标准正态分布来完成。此时要求协方差矩阵接近单位矩阵（防止成为 punctual distributions），均值接近 0（防止编码后在隐空间数据之间分隔太远）。



加入正则项后，可以防止编码后在隐空间的数据相隔太开，鼓励尽可能的返回分布相互重叠。从而满足上面的连续性和完整性。自然的，这会带来在训练集上重构损失增大。然而，在重构损失和 KL 散度之间的平衡可以调正。

我们可以观察到，通过正则化达到的连续性和完整性趋近于在隐空间上编码的信息创建了梯度。例如，隐空间上一个点处于被编码的两个类别数据之间，经过解码器得到的数据包含有两个类的特征。

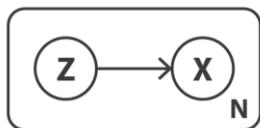


14.3.3 VAE 的数学描述

首先我们定义概率图模型来描述数据。

变量 x 描述了训练数据。 x 由隐变量产生，生成数据 x 的过程有下面两个假设：

z 是从一个先验分布 $p(z)$ 中抽样；数据 x 是从条件似然分布 $p(x|z)$



按照这个概率模型，我们重新定义编码器和解码器。概率上的解码器就是 $p(x|z)$ ，它描述了在给定被编码的条件下解码的变量的分布。概率上的编码器则是 $p(z|x)$ ，它描述了给定变量 x 的情况下，被编码的变量的分布。

如此，正则化就可以让编码器输出的 z ，在隐空间跟随先验分布 $p(z)$ 。贝叶斯定理描述了先验 $p(z)$ 、似然 $p(x|z)$ 和后验 $p(z|x)$ 的关系。

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} = \frac{p(x|z)p(z)}{\int p(x|u)p(u)du}$$

假设先验 $p(z)$ 是一个标准的正态分布， $p(x|z)$ 也是正态分布它的均值是由一个关于 z 的确定函数 f 定义，它的协方差矩阵是正值常数 c 乘上单位矩阵 I 。函数 f 假设属于一个函数族 F ，则可以得到

$$p(z) \equiv \mathcal{N}(0, I)$$

$$p(x|z) \equiv \mathcal{N}(f(z), cI) \quad f \in F \quad c > 0$$

现在假设函数 f 是固定的，当我们知道 $p(z)$ 和 $p(x|z)$ ，由贝叶斯定理则可以计算 $p(z|x)$ 。这是经典的贝叶斯推理问题。然而，由于计算的复杂性，这个推断过程的计算经常是不现实的，需要使用近似技术，例如变分推断（variational inference）。

统计学上，变分推断是一种对复杂分布近似的技巧。其思想是设置一个参数化的分布族（例如，正态分布族，它的参数是均值和协方差），在分布族中寻找对目标分布最好的近似。其最好的分布应该能最小化近似误差。大部分时候是用 KL 散度计算选择的分布和目标分布的误差。它可以在参数上用梯度下降的方法来寻找。

在 VAE 中，我们用一个正态分布 $q_x(z)$ 来近似 $p(z|x)$ 。它的均值和方差由两个函数 g 和 h 确定。这两个函数假设分别属于函数族 G 和 H 。因此这个正太分布可以表示成

$$q_x(z) \equiv \mathcal{N}(g(x), h(x)) \quad g \in G \quad h \in H$$

现在我们需要通过优化函数 g 和 h （实际上是挑选这两个函数的参数）来最小化 KL 散度，从分布族里挑选最优的分布 $q_x(z)$ 。换句话说，我们寻找最优的 g^* 和 h^* 以便，

$$\begin{aligned} (g^*, h^*) &= \arg \min_{(g, h) \in G \times H} KL(q_x(z), p(z|x)) \\ &= \arg \min_{(g, h) \in G \times H} \left(\mathbb{E}_{z \sim q_x} (\log q_x(z)) - \mathbb{E}_{z \sim q_x} \left(\log \frac{p(x|z)p(z)}{p(x)} \right) \right) \\ &= \arg \min_{(g, h) \in G \times H} (\mathbb{E}_{z \sim q_x} (\log q_x(z)) - \mathbb{E}_{z \sim q_x} (\log p(z)) - \mathbb{E}_{z \sim q_x} (\log p(x|z)) + \mathbb{E}_{z \sim q_x} (\log p(x))) \\ &= \arg \max_{(g, h) \in G \times H} (\mathbb{E}_{z \sim q_x} (\log p(x|z)) - KL(q_x(z), p(z))) \\ &= \arg \max_{(g, h) \in G \times H} \left(\mathbb{E}_{z \sim q_x} \left(-\frac{\|x - f(z)\|^2}{2c} \right) - KL(q_x(z), p(z)) \right) \end{aligned}$$

在倒数第二个公式，可以看到在近似后验时的一种权衡：最大化 log 似然的期望和保持接近先验分布（最小化 KL 散度）。上面已经假设 f 是已知和固定的，如此可以用变分推断的技巧近似后验分布 $p(z|x)$ 。

然而在实践中，函数 f 定义了解码器，是不知道它的参数的，也需要去学习（从函数族 F 中选择）。我们再回顾一下初始目标：发现一个 encoder-decoder 模型，对于“生成”的目的，其隐空间是足够规则的。如果正则化的过程是由先验分布在隐空间上进行主导，则 encoder-decoder 模型的性能高度依赖于函数 f 的选择。当 $p(z|x)$ 可以由 $p(z)$ 和 $p(x|z)$ 近似（通过变分推断的技巧），且 $p(z)$ 是一个简单的标准正态分布，则我们只需做两步，优化参数 c （似然的协方差）和函数 f （似然的均值）。

我们再考虑，可以得到任意的 $f \in F$ （每一个 f 是概率解码器 $p(x|z)$ ），挑选最好的近似 $p(z|x)$ 的 f ，标记为 $q_x(z)$ 。我们想训练的 encoder-decoder 模型尽可能有效率的，则选择函数 f ，它最大化给定 z 时 x 的期望似然，此时 z 从 $q_x(z)$ 抽样。换句话说，对于一个给定的 x ，当从 $q_x(z)$ 抽样 z 和从 $p(x|z)$ 抽样 \hat{x} ，我们想最大化 $x = \hat{x}$ 的概率。如此，是寻找最优函数 f^*

$$\begin{aligned} f^* &= \arg \max_{f \in F} \mathbb{E}_{z \sim q_x^*} (\log p(x|z)) \\ &= \arg \max_{f \in F} \mathbb{E}_{z \sim q_x^*} \left(-\frac{\|x - f(z)\|^2}{2c} \right) \end{aligned}$$

此处， $q_x^*(z)$ 依赖于函数 f ，可以由下面的方法获得。

把所有上面内容总结一下，我们的目的就是优化下面的目标函数。

$$(f^*, g^*, h^*) = \arg \max_{(f, g, h) \in F \times G \times H} \left(\mathbb{E}_{z \sim q_x} \left(-\frac{\|x - f(z)\|^2}{2c} \right) - KL(q_x(z), p(z)) \right)$$

该目标函数包括： x 和 $f(z)$ 之间的重构损失，在 $q_x(z)$ 和 $p(z)$ 之间 KL 散度的正则项。

$p(z)$ 是标准正太函数。参数 c 控制两项之间的平衡， c 越高，越支持正则项。

14.3.4 VAE 的神经网络实施

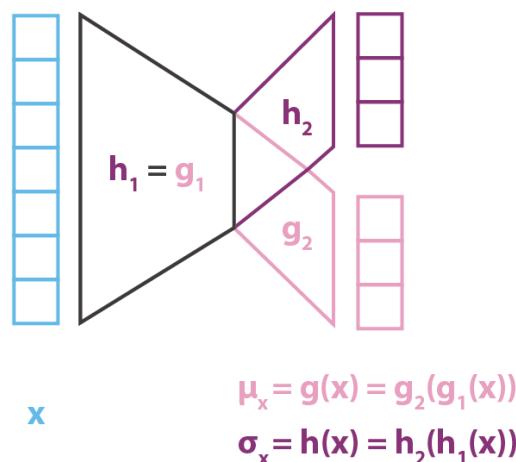
上一小节，建立了一个概率模型来描述 encoder-decoder 模型。模型依赖于三个函数 f, g, h 。模型的训练是一个优化任务获得 f^*, g^*, h^* 。当用神经网络来实施时，函数族 F, G, H 对应到不同的神经网络结构。获得 f^*, g^*, h^* 的优化问题就是，网络参数的学习问题。

实践中， g 和 h 并不是两个完全独立的网络。它们共享一部分结构和参数。

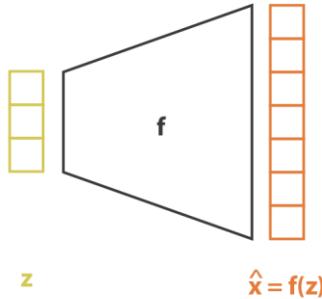
$$g(x) = g_2(g_1(x)) \quad h(x) = h_2(h_1(x)) \quad g_1(x) = h_1(x)$$

当定义了 $q_x^*(z)$ 的协方差矩阵， $h(x)$ 就是一个方阵。然而，为了简化计算，减少参数量，我们做另外一个假设，对于 $q_x(z)$ 是一个多维度正态分布，有一个对角的协方差矩阵（变量独立假设）。依据这个假设， $h(x)$ 就是一个协方差矩阵对角向量，其长度和 $g(x)$ 一样。

下图是 VAE 中编码器部分的结构



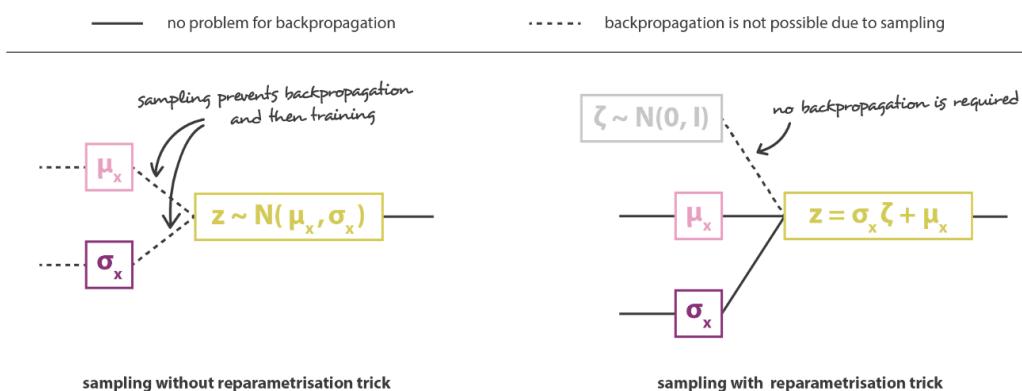
在解码器部分，模型假设 $p(x|z)$ 是一个具有固定方差的高斯分布。变量 z 的函数 f 定义了高斯分布的均值，用一个神经网络建模，结构描述如下：



上面就是模型的全部结构。然而，我们需要很小心的处理在模型训练过程中从编码器的分布中抽样的方法。抽样的过程的表达形式，必须允许误差的反向传播。一个简单的技巧称作重参数化 (**reparameterization trick**)，它可以保证，即使随机抽样出现在模型的中间部分，该技巧保证梯度下降的实施。如果 z 是一个服从正态分布的变量，均值 $\mu_x = g(x)$ ，协方差 $\sigma_x = h(x)$ 。 z 可以表示为

$$z = h(x)\zeta + g(x) \quad \zeta \sim \mathcal{N}(0, I)$$

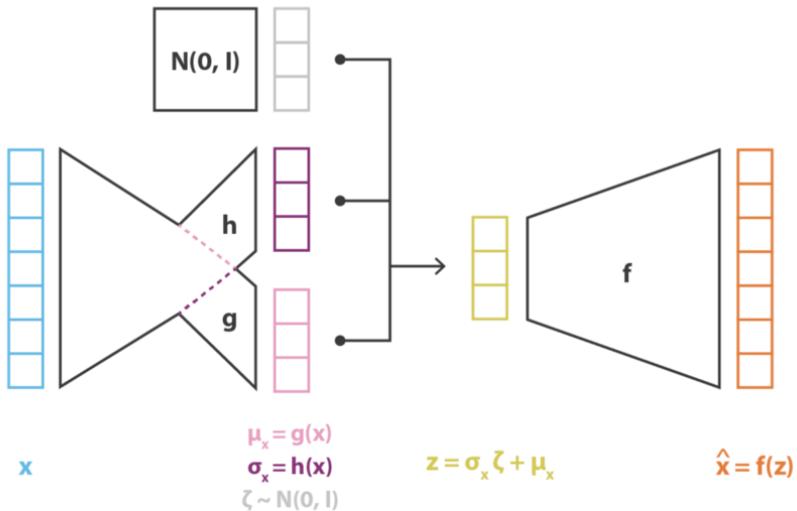
这就是重参数化技巧。下面是图示的重参数化技巧的计算过程



最终，变分自编码器的目标函数

$$(f^*, g^*, h^*) = \arg \max_{(f, g, h) \in F \times G \times H} \left(\mathbb{E}_{z \sim q_x} \left(-\frac{\|x - f(z)\|^2}{2c} \right) - KL(q_x(z), p(z)) \right)$$

其中的理论期望被一个蒙特卡洛近似 (Monte-Carlo approximation) 替换。用 C 表示 $1/(2c)$ ，则损失函数是



$$\text{loss} = C \|x - \hat{x}\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)] = C \|x - f(z)\|^2 + \text{KL}[N(g(x), h(x)), N(0, I)]$$

在 keras 里实现这一过程的代码如下

```

z_mean = Dense(latent_dim)(h)
z_log_var = Dense(latent_dim)(h)

epsilon = K.random_normal(shape=(batch_size, latent_dim), mean=0.,
                           stddev=epsilon_std)
z = z_mean + K.exp(z_log_var / 2) * epsilon

```

设 h 是前面的一个隐层, z_mean 和 z_log_var 是对应上图 h 和 g , 参数均值向量和协方差向量的隐层。从这两个隐层抽样的重参数化计算 z , 即从分布 $N(\mu_x, \sigma_x)$ 抽样得到的结果。

14.3.5 实例

第四节：生成对抗网络

14.4.1 生成对抗网络的基本原理

面对从一个未知分布 $P_{data}(x)$ 抽取样本 x , 生成建模 (generative modeling) 的目标是学习一个模型 model, 它描述了一个分布 $P_{model}(x)$ 。它尽可能的近似 $P_{data}(x)$ 。用深度学习做生成建模, 即建立一个生成模型可以从分布 P_{model} 生成样本 x [2]。这个难度比较大。在 GAN 之前, 学术领域对深度生成模型关注的较少, 多是在 discriminative model, 如分类模型上的研究。这是因为没有找到有效的训练深度生成模型的方法。

生成对抗网络 (Generative Adversarial Networks) 即 GANs[1, 2], 是一种深度生成模型 (deep generative model) 的训练方法。Yann LeCun 评价说

"Generative Adversarial network is the most interesting idea in the last ten years in Machine Learning."

GAN 的思想是给一个生成模型 (Generator)，建立一个判别模型 (Discriminator) 作为对手。Discriminator 尽力区分数据是由 Generator 产生的还是来自于真实数据。而 Generator 尽量让产生的数据能够欺骗 Discriminator，让它误以为是真实数据。

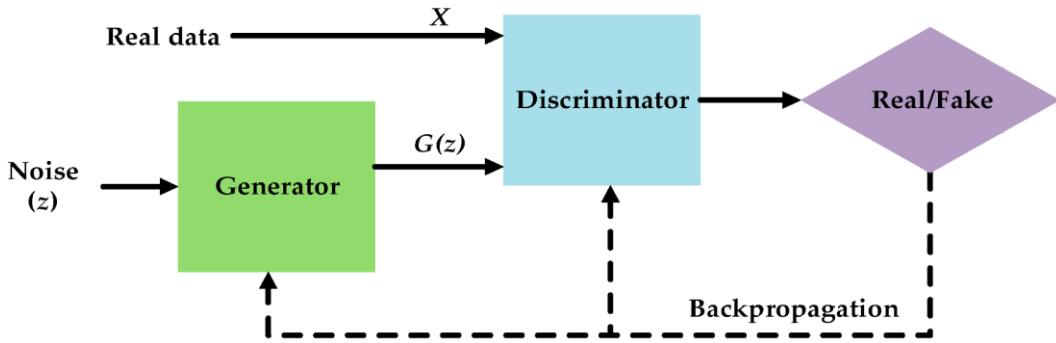


图 14.3 GAN 的结构

GAN 的结构如图 14.3 所示。已有的训练集 X 看作是真实数据；生成器输入一个随机数，它由此生成一个样本 $G(z)$ ；判别器不知道 X 和 $G(z)$ 哪个是生成数据，哪个是真实数据。判别器会对两者进行判断，是真实数据还是生成数据。由此产生的损失函数会反向传播更新生成器和判别器。如此，在判别器和生成器的对抗博弈过程，生成器能力越来越强，能够学习到一个分布 $P_{model}(x)$ 尽可能的近似 $P_{data}(x)$ 。即能够生成和真实数据相似的数据。

在实际操作中，图 14.3 中的 z 是从某个分布抽样产生随机数据。具体是什么分布，由用户自己定义。例如，可以定义为均值为零，方差为 1 的正态分布。从该分布抽样的随机向量作为生成器的输入，生成一个样本 $G(z)$ 。然后让判别器对比判别从真实数据集抽取的样本 x 和 $G(z)$ 谁是真实样本（真例）和假样本（假例）。判别器 D 就是一个分类模型，输出 0 或 1，判断输入的样本是生成的还是真实的。

GAN 的训练中，在一个 batch 的样本同时训练生成器和判别器，训练过程如下：

- (1) 得到一个 batch 的训练样本 x
- (2) 从某个用户设定的分布抽样（生成）一个 batch 数量的样本 z 。
- (3) 将 z 带入生成器，得到生成的样本 $G(z)$ 。
- (4) 将真实样本 x 和生成的样本 $G(z)$ 带入判别器 D 得到分类结果 $D(x)$ 和 $D(G(z))$
- (5) 计算生成器的损失值 g-loss。生成器损失函数用 binary cross entropy。因为希望生成器生成的模型被判别器为真实样本，则损失函数为 $\text{cross-entropy}(1, D(G(z)))$ 。
- (6) 计算判别器的损失值 d-loss。因为希望判别器能正常区分生成的样本还是真实样本，因此损失函数为 $\text{cross-entropy}(1, D(x)) + \text{cross-entropy}(0, D(G(z)))$

(7) 用损失值 g-loss 和 d-loss 分别更新生成器和判别器的参数。

论文 UNSUPERVISED REPRESENTATION LEARNING WITH DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS

用对抗生成网络的方法构建了一个从一个向量到图片的生成模型。详细代码见文件 mnist-dcgan.py。生成模型部分如下图。输入的向量还是从一个正太分布随机抽样产生。

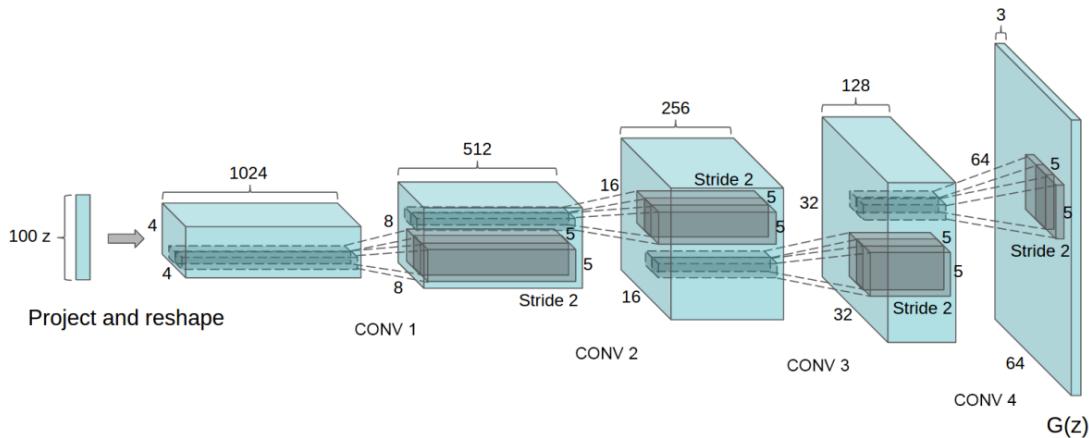


图 14.4 转置卷积的图片生成模型

其关键是转置卷积的操作。简单地讲，卷积操作中，数据的维度是在减少的。例如图 5.8 输入是 5×5 的矩阵卷积操作后是 3×3 的矩阵。在上面的由向量到图片生成的过程中，这个 tensor 的 shape 是在增加的（除了 depth），如第一个卷积过后 $4 \times 4 \times 1024$ 的 tensor 变成了 $8 \times 8 \times 512$ 的 tensor。这个过程像是卷积的相反过程，因此称作转置卷积，其原理本文不讨论，详见上面那篇论文。Tensorflow 提供了转置卷积层 Conv2DTranspose。在 MNIST 数据集上采用 GAN 训练的图片生成器代码如下。需要说明下面的代码中的模型参数并不与图 14.4 的模型完全一致。详见文件 mnist-dcgan.py。

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
                                   padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
```

```

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
use_bias=False, activation='tanh'))

return model

```

输入的向量，先经过一个由 7*7*256 个神经元的全连接层，再经过 reshape 层，转换成一个 7*7*256 的 tensor；再经过三个转置卷积操作输出的是 28x28x1 的图片。

判别器模型的如下：

```

def define_discriminator(in_shape=(28,28,1)):
    init = RandomNormal(stddev=0.02)
    model = Sequential()
    model.add(Conv2D(64, (4,4), strides=(2,2), padding='same',
kernel_initializer=init, input_shape=in_shape))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Conv2D(64, (4,4), strides=(2,2), padding='same',
kernel_initializer=init))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt,
metrics=[ 'accuracy'])
    return model

```

它就是一个 CNN 的图片分类器。模型的训练如下：

```

def train(generator, discriminator, dataset, epochs, noise_dim):
    for epoch in range(epochs):
        start = time.time()

        for image_batch in dataset:
            train_step(generator, discriminator, image_batch, noise_dim)

        # 产生图片
        display.clear_output(wait=True)
        generate_and_save_images(generator,
                                epoch + 1,
                                seed)

        print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

```

在每一趟的训练中，抽取一个批次训练 (train_step)。每一趟训练完成显示生成的图片，以便观察生成器的效果。在一个批次的训练中，代码如下：

```

def train_step(generator, discriminator, images, noise_dim):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

```

```

with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
    generated_images = generator(noise, training=True)

    real_output = discriminator(images, training=True)
    fake_output = discriminator(generated_images, training=True)

    gen_loss = generator_loss(fake_output)
    disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss,
                                                generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
                                                      discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
                                             generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
                                                 discriminator.trainable_variables))

```

首先从正太分布抽样向量 noise, 然后由生成器产生图片 generated_images。训练集里的数据 images 作为真实的数据。判别器对两个数据的判别, 产生结果 real_output (由 images 产生) 和 fake_output (由 generated_images 产生)。进一步产生生成损失 gen_loss 和判别损失 disc_loss。最后由这两个损失函数分布去优化生成器和判别器。

生成器和判别器的损失函数计算如下:

```

def discriminator_loss(real_output, fake_output):
    cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
    return cross_entropy(tf.ones_like(fake_output), fake_output)

```

14.4.2 GAN 的训练问题

<https://towardsdatascience.com/what-is-going-on-with-my-gan-13a00b88519e>

当训练的 GAN 达到下面的目标, 可以说 GAN 训练成功了:

- (1) 生成器可靠的生成数据, 且骗过了判别器;
- (2) 生成器生成的样本是与真实世界的数据分布一样的。

然而, GAN 的训练是一个挑战, 包括下面四个部分:

- 模式坍塌 Mode collapse
- 不收敛和不稳定 Non-convergence and instability

- 对超参数非常敏感
- GAN 的评估

下面评述各种挑战和解决方法：

1. 模式坍塌：

GAN 有时生成的样本对总体表示的能力不足。即，不能达到上面的目标（2），所有生成的样本很相似。例如，在 MNIST 数据集上，生成器只能生成一种数字（例如，0）。生成器可以创建一个样本总是可以欺骗判别器，即通过牺牲目标（2）达到目标（1）。

第二种情况是部分模式坍塌。生成器生成的样本多样性不够。例如，当生成人脸时，生成器可以很好的生成男性人脸，但不能很好的生成女性人脸。

解决方法包括：

- (1) 用 wasserstein 损失函数 [4]。它规定 GAN 的损失函数最小化两个概率分布的距离。原始 GAN 的损失函数的设计是“零和游戏”(minmax 损失函数)。该损失函数会有个问题，生成器赢了一轮，但减少生成的和实际的概率分布的距离不相关。
- (2) unrolling [5]。生成器的梯度反向传播在 k 步判别器的梯度反向传播后。这导致生成器可以看 k 步骤后的未来。如此可以生成的样本更具多样性。
- (3) packing [6]。让判别器基于多个同类样本，无论是真实样本还是生成的样本，再做决策。判别器一次看到一批同类样本时，它可以更好的确定一个 packed 不具备多样性的样本是人工生成的。

第五节：对抗自编码器

第三节曾提到自编码器的隐空间规则问题。编码器对数据编码就是把数据映射到一个隐空间，这个隐空间应该是规则的。变分自编码器通过由编码器产生一个分布，并和一个标准分布计算 KL 距离作为损失函数，来强制编码器的隐空间规则性。而对抗自编码器是另外一种强制编码器隐空间规则性的方法。

对抗自编码器 (Adversarial AutoEncoder, AAE) [3] 是把对抗网络应用在自编码器。其结构如图 14.5 所示。它包含两个部分：上面是一个标准的自编码器；下面是一个判别器。

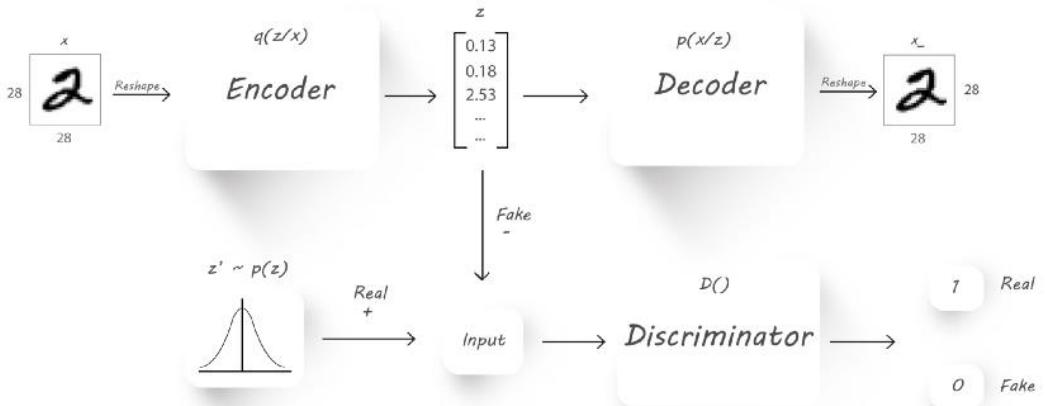


图 14.5 对抗自编码器的结构

AAE 训练时就有了两个目标函数，一个是输入和输出重构的误差，另外还要加上对抗训练的目标。在对抗训练的过程，自编码器试图让判别器不能区分编码后的表示向量 z 和从某个分布抽样的样本 z' 。于是，自编码器产生的中间编码向量 z ，最后会趋近于用户设定的分布 p 。

用 AAE 可视化 MNIST 数据集见图 14.5。其中给定的分布 P 满足均值为 0，标准差为 1 的正太分布。直观的感受效果要比图 14.2 用 AutoEncoder 做的效果好。向量 z 的分布差不多拟合分布 P 。

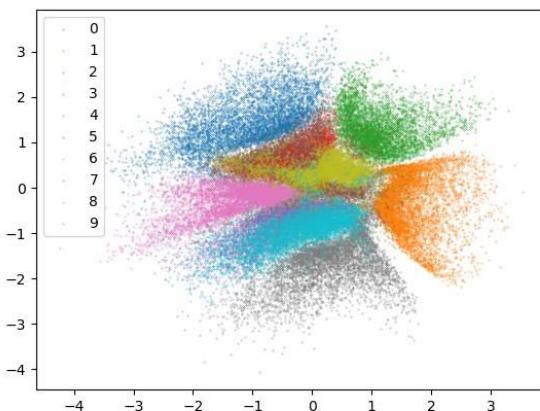


图 14.5 用 AAE 可视化 MNIST 数据集的分布

模型部分见下面的代码。完整的代码见文件 `mnist-aae.py`

```
import numpy as np
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras import layers
```

```

from tensorflow.keras.layers import Dense
from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt

encoding_dim = 2
epochs = 20
lr = 0.001
batch_size = 256
lam = 0.2

# %% building model
lrelu = tf.keras.layers.LeakyReLU(0.2)
input_img = keras.Input(shape=(784,))
n_layer1 = tf.keras.layers.BatchNormalization()(input_img)
encoded1 = layers.Dense(512, activation=lrelu)(n_layer1)
encoded2 = layers.Dense(128, activation=lrelu)(encoded1)
encoded3 = layers.Dense(64, activation=lrelu)(encoded2)
encoded = layers.Dense(encoding_dim)(encoded3)

decoded1 = layers.Dense(64, activation=lrelu)(encoded)
decoded2 = layers.Dense(128, activation=lrelu)(decoded1)
decoded3 = layers.Dense(512, activation=lrelu)(decoded2)
n_layer2 = tf.keras.layers.BatchNormalization()(decoded3)
decoded = layers.Dense(784, name="decoder",
activation='sigmoid')(n_layer2)

autoencoder = keras.Model(input_img, decoded)
encoder = keras.Model(input_img, encoded)

encoded_input = keras.Input(shape=(encoding_dim,))

decoder = tf.keras.Sequential([encoded_input,
                               autoencoder.layers[-5],
                               autoencoder.layers[-4],
                               autoencoder.layers[-3],
                               autoencoder.layers[-2],
                               autoencoder.layers[-1]])

mse = tf.keras.losses.MeanSquaredError()
op = tf.keras.optimizers.Adamax(learning_rate=lr)
autoencoder.compile(optimizer=op, loss=mse)

# discriminator
discriminator = tf.keras.Sequential([
    Dense(128, activation='relu', input_shape=(encoding_dim,)),
    Dense(64, activation='relu', input_shape=(encoding_dim,)),
    Dense(1, activation="sigmoid")
]);
cross_entropy = tf.keras.losses.BinaryCrossentropy()

# loss function of discriminator
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output),
real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output),
fake_output)
    total_loss = lam*real_loss + fake_loss

```

```

    return total_loss

def generator_loss(fake_output):
    fake_loss = cross_entropy(tf.ones_like(fake_output),
fake_output)
    return fake_loss

# %% preparing dataset
(x_train, y_train), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train),
np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test),
np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)

# %% fitting model with train_on_batch
dataset =
tf.data.Dataset.from_tensor_slices(x_train).batch(batch_size)

for epoch in np.arange(epochs):
    dataset = dataset.shuffle(buffer_size=len(x_train), seed=0)
    for x in dataset:
        z = np.random.normal(size=(len(x), encoding_dim))
        with tf.GradientTape() as gen_tape, tf.GradientTape() as dis_tape:
            x_decode = autoencoder(x, training=True)
            x_encode = encoder(x, training=True)
            f_output = discriminator(x_encode)
            r_output = discriminator(z)

            gen_loss = mse(x, x_decode) +
            lam*generator_loss(f_output)
            gen_loss = mse(x, x_decode)
            dis_loss = discriminator_loss(r_output, f_output)
            gradients_of_generator = gen_tape.gradient(gen_loss,
autoencoder.trainable_variables)
            op.apply_gradients(zip(gradients_of_generator,
autoencoder.trainable_variables))
            gradients_of_disc = dis_tape.gradient(dis_loss,
discriminator.trainable_variables)
            op.apply_gradients(zip(gradients_of_disc,
discriminator.trainable_variables))

            v_loss = autoencoder.evaluate(x_test, x_test, verbose=0)
            print(f'epoch: {epoch}, gen_loss: {gen_loss}, gen_val:
{v_loss}; dis_loss:{dis_loss}')

# %% draw images
def draw_imgs(n, imgs):
    plt.figure(figsize=(20, 4))
    for i in range(n):
        ax = plt.subplot(1, n, i + 1)
        plt.imshow(imgs[i].reshape(28, 28))
        plt.gray()

```

```

    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    plt.show()

# %% visualization
encoded = encoder(x_train)
x_encoded = encoder(x_train).numpy()
labels = list(set(y_train))
plt.figure()

for i in np.arange(len(labels)):
    idx = y_train == labels[i]
    plt.scatter(x_encoded[idx, 0], x_encoded[idx, 1], s=0.1,
label=labels[i])
plt.legend()
plt.show()

```

第六节：文本对抗自编码器

本节将把 AAE 应用在文本数据上，开发一个可以把文本数据进行编码的模型文本对抗自编码器称为 TextAAE (Text Adversarial AutoEncoder) 。

参考文献

- [1] Generative Adversarial Networks. Ian Goodfellow, etc. NIPS 2014.
- [2] Generative Adversarial Networks. Ian Goodfellow, etc. Communication of ACM, 2020.
- [3] Adversarial Autoencoders, Alireza Makhzani, etc. ICLR 2016.
- [4] Martin Arjovsky, Soumith Chintala, Léon Bottou, Wasserstein GAN, <https://doi.org/10.48550/arXiv.1701.07875>
- [5] Luke Metz, Ben Poole, David Pfau, Jascha Sohl-Dickstein. Unrolled Generative Adversarial Networks. <https://doi.org/10.48550/arXiv.1611.02163>
- [6] Zinan Lin, Ashish Khetan, Giulia Fanti, Sewoong Oh. PacGAN: The power of two samples in generative adversarial networks.
<https://doi.org/10.48550/arXiv.1712.04086>

第十五章：相关基础知识

这一章介绍学习深度学习需要的基本数学概念和机器学习的基本知识

第一节：线性代数

线性代数是数学的一个分支，在工程中广泛应用。许多机器学习算法，特别是深度学习需要线性代数的知识。

14.1.1 标量、向量、矩阵和张量

标量 (Scalars) 就是一个数。一个实数标量 s 用 $s \in \mathbb{R}$ 表示。定义一个自然数标量用 $s \in \mathbb{N}$ 表示。在计科的文章中**标量用小写字母斜体表示**。

向量 (Vectors) 就是有序安排的一组数据。可以通过数组索引或下标 (index) 来定位一个数。例如，一个实数数组 $x \in \mathbb{R}^n$

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

x_1 表示数值的第一个元素。**向量用小写字母斜体加粗表示，表示数组中的一个元素时，数组不加粗。**例如， \boldsymbol{x} 表示向量， x_1 ，表示向量中的元素。

矩阵 (Matrices) 即一个二维数组。对于一个 m 行 n 列的实数数组 $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ ，需要通过两个下标来确定其中的元素，例如 $A_{i,j}$ 。如果指代在某个维度上所有的元素可以用符号 $:$ ，例如 $A_{:,j}$ 表示矩阵 \boldsymbol{A} 上第 i 行上的所有元素。**矩阵用大写字母斜体加粗表示。**表示矩阵中的元素矩阵名不加粗。

张量 (Tensors)，按照《deep learning》一书的定义

an array of numbers arranged on a regular grid with a variable number of axes is known as a tensor.

也有人解释，超过二维的数组都称为张量。**张量用大写字母加粗表示。**例如 \mathbf{A}

一个矩阵 \boldsymbol{A} 的转置即该矩阵沿着它的对角线的镜像，称为 \boldsymbol{A}^T 。

$$(A^T)_{ij} = A_{ji}$$

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} A_{1,1} & A_{2,1} & A_{3,1} \\ A_{1,2} & A_{2,2} & A_{3,2} \end{bmatrix}$$

因为，向量都是按照列向量的方式定义的。因此，一个向量的转置就是仅有一行的矩阵。 $x = [x_1, x_2, x_3]^T$ 。

只要两个矩阵有相同的 shape 就可以进行加法运算。例如， $C = A + B$ ，即 $C_{i,j} = A_{i,j} + B_{i,j}$

矩阵可以和标量相加或相乘。例如， $D = a \cdot B + b$ ，即 $D_{i,j} = a \cdot B_{i,j} + b$

在深度学习中，也会使用一些不太传统的表示法，例如，运行矩阵和一个向量想加产生一个新的矩阵。例如， $C = A + b$ ，即 $C_{i,j} = A_{i,j} + b_j$ 。换句话说，向量 b 被加到矩阵的每一行

14.1.2 矩阵相乘

两个矩阵 A 和 B 的矩阵乘 (matrix product) 操作 $C = AB$ ，定义为

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

它要求矩阵 A 的列数等于矩阵 B 的行数，shape 为 $m \times n$ 的矩阵 A 矩阵乘 'shape 为 $n \times p$ 的矩阵 B 得到 shape 为 $m \times p$ 的矩阵 C

两个矩阵的逐元素乘法又称为 Hadamard product，用符号 $C = A \odot B$ 表示。矩阵 A 、 B 、 C 都是 shape 为 $m \times n$ 的矩阵。 $C_{i,j} = A_{i,j} B_{i,j}$

两个长度相同的向量 x 和 y 的点乘 dot product，就是矩阵乘 $x^T y$ 。

矩阵乘 (matrix product) 满足一些定律：

分配律： $A(B + C) = AB + AC$

结合律： $A(BC) = (AB)C$

不满足交换律，即 $AB = BC$ 并不总是成立。

但两个向量之间的点乘满足交换律

14.1.3 单位矩阵和逆矩阵

单位矩阵 (identity matrix) 是指当我们用一个向量乘上一个矩阵，得到的向量等于原来的向量。单位矩阵形式化定义为： $I_n \in \mathbb{R}^{n \times n}$ ，并且 $\forall x \in \mathbb{R}^n, I_n x = x$

单位矩阵的结构非常简单，即它的对角线上的元素是 1，其他位置的值都为 0。

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

一个矩阵 A 的逆矩阵用符号 A^{-1} 表示。满足 $A^{-1}A = I_n$ 。但一个矩阵并不总是有对应的逆矩阵的。

14.1.4 范数

当我们需要度量向量的大小 (size) 时，机器学习中使用范数 (Norms) 来度量。其中 L^p 范数为

$$\|x\|_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}}$$

范数是一个函数，它把向量映射到一个非负值。直觉上，一个向量 x 的范数度量了从原点到点 x 的距离。严格的说，范数是满足下面属性的一个函数

- $f(x) = 0 \Rightarrow x = 0$
- $f(x + y) \leq f(x) + f(y)$ (the triangle inequality)
- $\forall \alpha \in \mathbb{R}, f(\alpha x) = |\alpha| f(x)$

对于 L^2 范数，即 $p=2$ 时，就是欧几里得范数。它就是度量了从原点到点 x 的欧式距离。欧几里得范数在机器学习中使用非常频繁，因此经常用 $\|x\|$ 就表示了欧几里得范数（省略了下标 2）。

还有一种很常见的做法，当我们度量一个向量的大小 (size) 使用平方的 L^2 范数。此时，对于向量 x 它的 L^2 范数就是 $x^T x$ 。在机器学习，使用平方的 L^2 范数更方便后继的一些计算。例如，向量 x 的平方 L^2 范数就每个元素求偏导时，结果仅仅取决于对应的元素，而 L^2 范数就取决于整个向量。

但平方 L^2 范数在机器学习中会有一个问题，在接近原点时，平方 L^2 范数会变化很慢。在一些机器学习任务中，对于区分一个数是零还是“不是零”这个很重要。 L^1 范数则是向量 x 在空间的任何位置变化率都是一样的。其定义为

$$\| \mathbf{x} \|_1 = \sum_i |x_i|$$

下面就“在一些机器学习任务中，对于区分一个数是零还是“不是零”这个很重要”我们拓展一下。在回归模型中

$$\mathbf{Y} = \boldsymbol{\beta} \mathbf{X} + \epsilon$$

它从一个线性组合的一组变量 \mathbf{X} 和一个正太分布的误差项 ϵ ，预测变量 \mathbf{Y} 。训练模型的目的就是学习到参数 $\boldsymbol{\beta}$ 。用最小二乘估计模型系数

$$L_{ols}(\hat{\boldsymbol{\beta}}) = \| \mathbf{Y} - \mathbf{X}\hat{\boldsymbol{\beta}} \|_2^2$$

岭回归是在上面的损失函数中加入系数的 L^2 惩罚项

$$L_{ridge} = \| \mathbf{Y} - \mathbf{X}\hat{\boldsymbol{\beta}} \|_2^2 + \lambda \| \hat{\boldsymbol{\beta}} \|_2^2$$

其作用是可以使得回归模型预测可以一定程度克服过拟合，性能更好。

Lasso 回归是在上面的损失函数中加入系数的 L^1 惩罚项

$$L_{lasso} = \| \mathbf{Y} - \mathbf{X}\hat{\boldsymbol{\beta}} \|_2^2 + \lambda \| \hat{\boldsymbol{\beta}} \|_1$$

它可以使得有些和模型行动不太相关的变量的系数等于 0，因此可以用来做特征选择。另外，当多个变量之间存多重共线性，它可以随机选择其中一个变量，使得其他的变量的系数等于 0。这样就消除了多重共线性。

有时我们在度量向量的大小时，是通过计数非零元素个数来度量的。有些地方称它为 L^0 范数。但《Deep Learning》指出这不正确，它就不是一个范数。因为，如果给向量乘上一个系数，向量的非零元素个数不变。

L^∞ 范数也称为 max 范数，定义为

$$\| \mathbf{x} \|_\infty = \max_i |x_i|$$

当我们希望度量矩阵的大小时，在机器学习的语境下，可以使用 Frobenius 范数

$$\| A \|_F = \sqrt{\sum_{i,j} A_{i,j}^2}$$

可以看到，它就是把 L^2 范数拓展到了矩阵。

14.1.5 一些特别的矩阵和向量

对角矩阵 (Diagonal matrix) 是指一个矩阵除了主对角线上的元素可以不为零，其他的元素都为零。形式化描述就是，对于一个矩阵 D ，当且仅当对于所有的 $i \neq j, D_{i,j} = 0$ 。单位矩阵就是一个对角阵，它的对角线上的元素都是 1，其他元素都是零。我们通常使用 $\text{diag}(\nu)$ 表示一个对角方阵，它对角线上的元素是向量 ν 。

对角矩阵在机器学习中有时是很有意义的，因为矩阵的计算代价会很高，但乘上一个对角阵计算上效率会很高。例如 $\text{diag}(\nu)x$ 的计算等于 $\nu \odot x$ 。计算一个对角矩阵的逆也很有效率， $\text{diag}(\nu)^{-1} = \text{diag}([1/\nu_1, \dots, 1/\nu_n]^T)$ 。在机器学习任务中，有时为了减少计算代价，会限制矩阵是对角矩阵。

对称矩阵是指一个矩阵等于自己的转置 $A = A^T$ 。例如，对于一个训练集，所有的元素之间计算欧式距离，然后用一个矩阵描述，它就是一个对称矩阵。

单位向量，是一个满足单位范数 $\|x\|_2 = 1$ 的向量。

如果两个向量 x 和 y 满足 $x^T y = 0$ ，称两个向量是正交的。如果两个向量中都有非零范数，这意味着向量夹角是 90 度。如果两个向量不但正交还有单位范数，它们被称为标准正交 (orthonormal)。一个正交矩阵是一个方阵，它的行是互相标准正交，它的列也是互相标准正交，即

$$A^\top A = AA^\top = I$$

也即

$$A^{-1} = A^\top$$

正交矩阵在有些情况下是有意义的，因为它的逆矩阵计算代价很小。

讲授机器学习的基本理论

第二节：概率论

概率论是描述不确定性的数学框架。它提供了定量不确定性的方法。人工智能领域在两个方面使用概率理论。首先概率法则告诉我们 AI 系统应该如何推理，因此我们设计算法来计算和近似各种概率表达。第二，使用概率论或统计理论来理论分析 AI 系统的行为。

概率论使得我们可以描述不确定性并进行推理，信息论使得我们在已知信息的情况下对一个概率分布中的不确定性进行定量

在计算机科学领域，它所处理的对象大部分都是确定的。比如，我们可以假设 CPU 可以毫无错误的处理每条指令。虽然，硬件系统会发生错误，但程序设计时可以完全忽略这种错误。但机器学习对现实事件建模时，就包含了很多的不确定性。不确定性主要来自于几个方面：（1）系统内在的随机性。比如在游戏中，游戏的进行都存在随机性。（2）不完整的观察。即使一个系统本身是确定性的，但由于我们不能观察到系统所有的变量，就带来了不确定性。例如，在 Monty Hall problem 游戏中，每个门后面有什么是确定的，但从参与者的角度，游戏结果是不确定的。（3）不完整建模。如果我们建模时，必须废弃掉一些观察到的信息，这些废弃掉的信息导致模型做预测时的不确定性。例如，假设一个机器人可以观察到它周围的物体的位置，但它在预测物体未来的位置时，把信息离散化（如，把位置网格化），则离散化的行为使得机器对位置的预测不够精确，就充满不确定性。

概率理论的最初发明是为了分析事件发生频率。这种事件通常是可重复的。当我们说一个事件发生的概率是 p 时，它意味着如果我们重复这个实验 n 次，可以观察到 $p \cdot n$ 次事件的发生。也可以重复到多个个体上。例如，把 40 个硬币同时抛向空中，落地后有多少个硬币正面朝上。但对于不可重复的实验，例如，医生对一个病人说，你有 40% 的可能得了流感。但我们不能重复这个病人多次，也不能相信有同样症状的多人适用于 40% 这个概率。这种情况下，医生给出的 40% 是一个相信程度，1 表示病人确定的得了流感。0 表示病人确定的没得流感。前一种概率是和事件发生率相关，被称为频率学派概率 (frequentist probability)；第二种是和“定性的确定性程度”相关，称为贝叶斯学派概率 (Bayesian probability)。

14.2.1 随机变量

随机变量是一个变量，它可以随机的取不同的值。我们用小写字母表示随机变量，用变量加下标表示它的可能取值。例如， x_1, x_2 是随机变量 x 的可能取值。对于向量形式的随机变量，例如，描述物体在二维平面的坐标的变量，用加粗的小写字母表示 \mathbf{x} ，而它的可能取值就是一个向量 x 。

随机变量描述了可能的状态，它必须有一个概率分布来规定每一种状态的可能性。

随机变量可以是连续值的也可以是离散的。

14.2.2 概率分布

概率分布是对一个随机变量它的各种状态（可能取值）的发生的概率描述。

在离散变量 (discrete variable) 上的概率分布使用概率质量函数 (probability mass function) 来描述。概率质量函数用大写斜体 P 来表示。注意， $P(x)$ 和 $P(y)$ 表示随机变量 x 和 y 的概率质量函数，但不意味着它们有相同的概率质量函数。

概率质量函数将一个随机变量的一个状态映射到随机变量发生这个状态的概率值。 $x = x$ 的概率用 $P(x)$ 表示。其概率值为 1 表示 $x = x$ 是确定发生，概率值为 0 表示 $x = x$ 是不确定发生。有时为了和概率密度函数区分，会把在某个状态的概率写为 $P(x = x)$ 。会用 $x \sim P(x)$ 表示随机变量 x 服从什么样的分布。

概率质量函数也可以用于同时描述多个变量，这称为这些变量的联合概率分布。 $P(x = x, y = y)$ 表示同时随机变量 x 在状态 x 和变量 y 在状态 y 的概率。也可以写为 $P(x, y)$ 。

一个随机变量 x 的概率质量函数 $P(x)$ 必须满足下面的属性

P 的域必须是随机变量 x 的所有可能状态（取值），满足

$$\forall x \in \text{X}, 0 \leq P(x) \leq 1$$

并且

$$\sum_{x \in \text{X}} P(x) = 1$$

例如，一个离散随机变量 x 有 k 个状态， x 上的均匀分布，即每个状态发生的概率都是一样的。它的概率质量函数是 $\forall i, P(x = x_i) = 1/k$

对于连续变量，使用概率密度函数来描述它的概率分布。对于一个概率密度函数 p ，它满足下面的条件：

(1) p 的域是随机变量 x 的所有可能取值

(2) $\forall x \in \text{X}, p(x) \geq 0$

(3) $\int p(x) dx = 1$

注意，第二个属性中并没有要求 $p(x) \leq 1$ 。这和概率质量函数不同。概率密度函数中我们不是考察不是取某个值的概率，而是考察 x 落在某个区域中的概率。通常用积分来获得。 $\int_{[a,b]} p(x) dx$ 。例如，举个例子。在连续随机变量 x 一个区间 $[a, b]$ 服从均匀分布，可以用函数 $u(x; a, b)$ 表示。“;”表示以...为参数。它满足 $\forall x \notin [a, b], u(x; a, b) = 0$ 。在 $[a, b]$ ， $u(x; a, b) = \frac{1}{b-a}$ 。且积分的值为 1。可以用公式 $x \sim U(a, b)$ 来表示随机变量 x 在区间 $[a, b]$ 服从正太分布。

14.2.3 边缘概率

有时我们知道一组变量的联合概率分布。其中一个变量子集的概率分布称为，边缘概率（Marginal Probability）分布。例如，我们有两个离散的随机变量 x 和 y ，我们已知它们的联合概率分布 $P(x, y)$ 。通过求和规则（sum rule）

$$\forall x \in X, P(x = x) = \sum_y P(x = x, y = y)$$

对于连续变量可以使用积分求得边缘概率分布

$$p(x) = \int p(x, y) dy$$

14.2.4 条件概率

我们有时会对给定某个事件其他事件发生的概率感兴趣。这称为条件概率。给定 $x = x$ 时， $y = y$ 发生的概率记为 $P(y = y | x = x)$ 。该条件概率可以用下面的公式计算

$$P(y = y | x = x) = \frac{P(y = y, x = x)}{P(x = x)}$$

多个变量的联合概率分布可以分解为一个变量上的条件概率分布。

$$P(x^{(1)}, \dots, x^{(n)}) = P(x^{(1)}) \prod_{i=2}^n P(x^{(i)} | x^{(1)}, \dots, x^{(i-1)})$$

这称为链式规则或乘法规则。例如，

$$\begin{aligned} P(a, b, c) &= P(a | b, c)P(b, c) \\ P(b, c) &= P(b | c)P(c) \\ P(a, b, c) &= P(a | b, c)P(b | c)P(c) \end{aligned}$$

14.2.5 独立与条件独立

如果两个随机变量 x 和 y 的联合概率分布能够表达为 x 和 y 单独的两个概率分布的乘积，则称两个随机变量是独立的。

$$\forall x \in X, y \in Y, p(x = x, y = y) = p(x = x)p(y = y)$$

给定一个随机变量 z ，如果 x 和 y 上的条件概率分布可以分解为下面的形式

$$\forall x \in X, y \in Y, z \in Z, p(x = x, y = y | z = z) = p(x = x | z = z)p(y = y | z = z)$$

则称 x 和 y 在给定 z 的情况下条件独立。可以用符号 $x \perp y | z$ 表示。

机器学习中有时为了简化的复杂性，经常引入条件独立性假设。例如，贝叶斯文本分类器需要计算给定一篇文档，它属于类别 c 的概率 $P(c|d)$ 。因为 $P(c|d) \propto P(d|c)P(c)$

即计算 $P(d|c)$ 和 $P(c)$ 。文本处理中通常把文档看做是词的序列因此 $P(d|c) = P(w_1, \dots, w_n | c)$

这个很难计算，因此引入条件独立假设，每个词项相对于类别是条件独立的。于是

$$P(w_1, \dots, w_n | c) = \prod_{i=1}^n P(w_i | c)$$

这样为每个词 w_i 计算 $P(w_i | c)$ 就把问题简化了。

14.2.6 期望，方差和协方差

一个函数 $f(x)$ 就一个随机变量 x 的概率分布 $P(x)$ 的期望就是从该分布抽取变量 x 的取值 x ，函数 f 在取值 x 上计算的结果的均值。如果是离散变量可以按照下式计算

$$\mathbb{E}_{x \sim P}[f(x)] = \sum_x P(x)f(x)$$

对于连续变量按照积分来计算

$$\mathbb{E}_{x \sim p}[f(x)] = \int p(x)f(x)dx$$

当在上下文中，这个分布是什么样是清楚的。可以简写为

$$\mathbb{E}_x[f(x)]$$

如果对随机变量也是清楚的可以简写为

$$\mathbb{E}[f(x)]$$

约定成俗的， $E[\cdot]$ 表示是方括号内的所有值的均值。

期望的计算是线性的，例如，当 α, β 不依赖于 x

$$\mathbb{E}_x[\alpha f(x) + \beta g(x)] = \alpha \mathbb{E}_x[f(x)] + \beta \mathbb{E}_x[g(x)]$$

当一个随机变量 x 服从一个分布。方差度量该随机变量带入一个函数的值的变化程度。

$$\text{Var}(f(x)) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])^2]$$

方差比较小，表示 $f(x)$ 的值都聚集在期望值附近。值越大，表示 $f(x)$ 的值变化很大。

把方差开方就得到标准差。

协方差用来度量两个变量的线性相关的程度。

$$\text{Cov}(f(x), g(y)) = \mathbb{E} [(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[g(y)])]$$

协方差绝对值越高表示两个变量的变化都很大。如果协方差是正数，表示两个变量的变化方向是一致的，而负数表示变化方向的不一致。

但是当 $f(x), g(y)$ 不在一个数量级时，就会计算不准。皮尔逊相关系数，就是在协方差的基础上再除以 $f(x)$ 和 $g(y)$ 的标准差，相当于给它们做了规范化。这样计算的相关性就准确了。

协方差与依赖性 (dependence，相对于独立性 independence 来说的) 两个概念是相关的，但实际上 是不同的两个概念。说它们是相关的，是因为两个相互独立的两个变量它们的协方差是 0。两个变量的协方差如果是非零，则它们是依赖的。但如果两个变量的协方差是零，只说明它们之间没有线性依赖。

《Deep Learning》给了一个例子。随机变量 x 服从区间 $[-1, 1]$ 的均匀分布。接着按照随机变量是服从下面的分布。以 0.5 的概率，选择 s 的值是 -1，否则是 1。如此可以产生一个随机变量 $y=sx$ 。很显然 y 和 x 之间不是相互独立的。但它们的协方差是 $\text{Cov}(x, y)=0$ 。

一个随机向量 x 的协方差矩阵是一个 $n \times n$ 矩阵，其中

$$\text{Cov}(x)_{i,j} = \text{Cov}(x_i, x_j)$$

为向量 x 计算的协方差矩阵 $\text{Cov}(x)$ 中的下标为 i, j 的元素，就是随机向量 x 中的第 i 个随机变量和第 j 个随机变量的协方差。

第三节：信息论

14.3.1 介绍

信息论是应用数学的一个分支，是由香农(Claude Shannon)在二十世纪四十年代创建的理论体系。它是关于定量信号里展示的信息量。它起初被发明时，是为了研究在噪声信道传输信息，例如无线电通信。在信息传输的语境下，信息论就是告诉怎样设计最优的编码。在机器学习的语境下，可以应用信息论来描述概率分布的特征，或计算概率分布之间的相似性。

信息论背后的基本直觉是，了解到“一个不可能的事件发生了”比了解到“一个可能的事件发生了”会获得更多的信息。一条信息“太阳今天早上升起了”，没有什么信息量。但“今天早上有日食”就很有信息量。

信息论通过把上面的直觉形式化来定量信息。定量化时应该满足三个属性：

- (1) 可能的事件应该有比较低的信息量。极端情况下，那些确切要发生的事情根本没有信息量
- (2) 越低可能发生的事情有越少的信息量。
- (3) 独立事件它们的信息量可以叠加。例如，“掷硬币有两次头朝上”比“掷硬币有一次头朝上”传递了两倍的信息量。

14.3.2 基本理论

1. 自信息

定义一个满足三个属性的事件 $x = x$ 的自信息 (self-information) 为

$$I(x) = -\log P(x)$$

当对数运算的底为 e 时， $I(x)$ 信息量的单位是 nat。一个 nat 是指信息增益的量当观察到一个发生概率为 $1/e$ 的事件。其他有的地方上面只信息的对数运算以 2 为底时，单位为 bits 或 shannons。

2. 熵 (entropy)

自信息仅仅处理一个事件，但我们可以在整个概率分布上定量随机变量 x 的不确定性。称为熵或香农熵。

$$H(x) = \mathbb{E}_{x \sim P}[I(x)] = -\mathbb{E}_{x \sim P}[\log P(x)]$$

随机变量的熵越大，包含的随机信息量越大，它的不确定性也越大。也就是说能正确估计其值的概率越小。如果 x 是离散随机变量熵的计算公式如下：

$$H(x) = -\sum_{x \in X} p(x) \log p(x)$$

按照熵的定义，它具备以下属性：

- $H(x) \geq 0$
- $H(x) = 0$ ，当且仅当随机变量 x 的值是确定的，没有任何信息量可言。

3. 联合熵

如果 (x, y) 是一对离散随机变量，其联合概率分布函数密度为 $p(x, y)$ 。 (x, y) 的联合熵 $H(x, y)$ 定义为

$$H(x, y) = -\sum_{x \in X} \sum_{y \in Y} p(x, y) \log p(x, y)$$

4. 条件熵：

如果离散随机变量 (x,y) 的联合概率分布函数密度为 $p(x,y)$, 已知随机变量 x 的情况下随机变量 y 的条件熵 $H(y|x)$, 实际上表示的是在已知 x 的情况下, 传输 y 额外需要的平均信息量

$$H(y|x) = - \sum_{x \in X} \sum_{y \in Y} p(x,y) \log p(y|x)$$

熵的链规则为: $H(x,y) = H(x) + H(y|x)$

5. 互信息

根据熵的链规则, 可以有如下的计算公式

$$I(x;y) = H(x) - H(x|y) = H(y) - H(y|x)$$

这个差值称为随机变量 x 和 y 之间的互信息。用 $I(x;y)$ 表示, 它实际上表示在已知 y 的值后 x 的不确定性的减少量, 即随机变量 y 揭示了多少关于 x 的信息量。互信息又称为信息增益。

我们可以用一张图 (图 5-1) 来描述上述概念之间的关系。

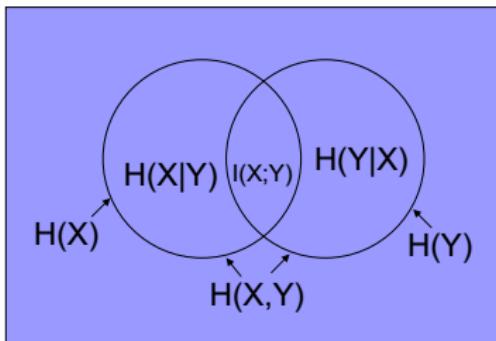


图 14.1 熵, 条件熵, 联合熵和互信息之间的关系

当 x 和 y 是离散随机变量时, 互信息的公式可以描述为

$$I(x;y) = \sum_{x \in X, y \in Y} \log \frac{p(x,y)}{p(x)p(y)}$$

它是一种计算两个随机变量之间共有信息的度量。满足非负性和对称性两个特点。它实际上作为验证两个变量是否互不相关的手段。它的取值 (1) 当两个随机变量无关时, 互信息为 0; (2) 当两个变量之间存在依赖关系时, 它们的互信息不仅和依赖程度相关, 而且和变量的熵也相关。

6. 相对熵和 Kullback-Leibler 距离

在同一个随机变量上如果有两个概率分布或概率密度函数 $P(x)$ 和 $Q(x)$ ，可以使用相对熵又称为 KL 散度 (Kullback-Leibler Divergence) 来度量两个分布有多大的差异。有下面的公式

$$D_{\text{KL}}(P\|Q) = \mathbb{E}_{x \sim P} \left[\log \frac{P(x)}{Q(x)} \right] = \mathbb{E}_{x \sim P} [\log P(x) - \log Q(x)]$$

相对熵满足下面的属性：

- (1) KL 散度是非负的。
- (2) 当且仅当 $p(x)=q(x)$ 时 $D(p\|q)=0$ 。
- (3) $D(p\|q) > D(q\|p)$ 即非对称性。

第四节：数值计算

计算机不能求解确定的数学公式。数值计算通常是指通过迭代的方法更新解，来解决数学问题的算法。机器学习中通常需要大量的数值计算。通常数值计算包括优化（发现可以使得目标函数最小化或最大化的参数）和求解线性方程。

14.4.1 上溢 Overflow 和下溢 Underflow

在计算机上进行数学计算的基本问题是，对很多实数计算机只能近似表示，这会导致近似误差。很多情况下，就是舍入误差 (rounding error)。如果，没有设法消除累积的舍入误差，有时会数学计算导致在理论上成立，但在实际中失败。

一种舍入误差是下界溢出 (underflow)。当一个数值接近零但不是零，当被四舍五入操作当作零时发生。对于很多函数，它们对待零和一个很小的正数是不同的。例如，我们通常避免一个数除以零，也避免一个值为零的对数运算，很多编程语言此时会产生异常。

还有一个叫上界溢出 (overflow)。当一个数太大，计算机不能描述时，把它当做了无穷大。计算机会把无穷大当做非数值。

计算机上的一个函数计算必须考虑对方上界和下界溢出的情况。我们来看一个例子。Softmax 函数在深度学习中使用的非常多，它把一个向量的值转换成概率分布。其定义如下：

$$\text{softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^n \exp(x_j)}$$

如果其中一个值 x_i 是非常大的，发生上界溢出，则最终计算的结果就错误了。此时可以采用一个小技巧，另 $z=x-\max(x)$ ，然后计算 $\text{Softmax}(z)$ 。该技巧不会导致结果改变，还可以防止上界和下界溢出。

大部分时候，当我们作为 high-level 的开发者，即调用别人的库来进行开发，不需要考虑这样的数值问题。当我们作为 low-level 底层的开发者，必须要注意上述问题。

14.4.2 病态条件数 (Poor Conditioning)

条件数 (conditioning) 是衡量对于一个函数，输入的微小变化引起输出变化迅速程度的量。当输入产生轻微的变化（输入的扰动或者实际值偏离真实值），函数变化迅速，这对于科学计算来说是有问题的。因为输入的舍入误差可以导致输出大的改变。

考虑一个函数 $f(\mathbf{x}) = \mathbf{A}^{-1}\mathbf{x}$ 。当矩阵 $\mathbf{A} \in \mathbb{R}^{n \times n}$ 有一个特征值分解。该矩阵的条件数 (condition number) 是

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|$$

即最大和最小特征值的比例。如果条件数越大，矩阵逆运算对输入误差则特别敏感。该敏感性是矩阵本身的内在属性，而不是矩阵逆运算舍入误差导致的结果。病态条件矩阵可以放大存在的误差。

14.4.2 基于梯度的优化

大部分的深度学习算法都涉及到优化。优化是指，对函数 $f(\mathbf{x})$ ，通过改变 \mathbf{x} 使得函数值最大化或最小化的过程。我们通常习惯描述成最小化问题。因此对于如果是最大化函数 $f(\mathbf{x})$ 的问题，可以转换成最小化 $-f(\mathbf{x})$ 。

我们想最大化或最小化的函数可以成为目标函数 (objective function) 或准则 (criterion)。如果是最小化该函数，它也可以称为代价函数 (cost function) 或损失函数 (loss function) 或误差函数 (error function)。

我们经常将最大化或最小化了一个函数的值用*上标表示。例如，

$$\mathbf{x}^* = \arg \min f(\mathbf{x})$$

假设有一个函数 $y = f(x)$ 。 x 和 y 都是实数值。函数的导数用符号 $f'(x)$ 和 $\frac{dy}{dx}$ 表示。导数 $f'(x)$ 表示的是函数 $f(x)$ 在点 x 的斜率。换句话说，它近似的定量了怎样输入的一个小改变导致输出的改变。

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

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 $f(x)$ 是正数， $\epsilon f'(x) < 0$ 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 ϵ 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 ϵ ，可以使得 $f(x - \epsilon \text{ sign}(f'(x)))$ 小余 $f(x)$ 。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 $\text{sign}(x) = -1$.

当 $f'(x) = 0$ ，导数对于朝哪个方向移动没有提供信息。 $f'(x) = 0$ 的点就是临界点 (critical points)。一个点 x 对应的函数值比它的邻居点的函数值都要小，称该点为局部最小点 (local minimum)。在函数 $f(x)$ 域的范围内的最小点就是全局最小点 (global minimum)。

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 14.2 所示。

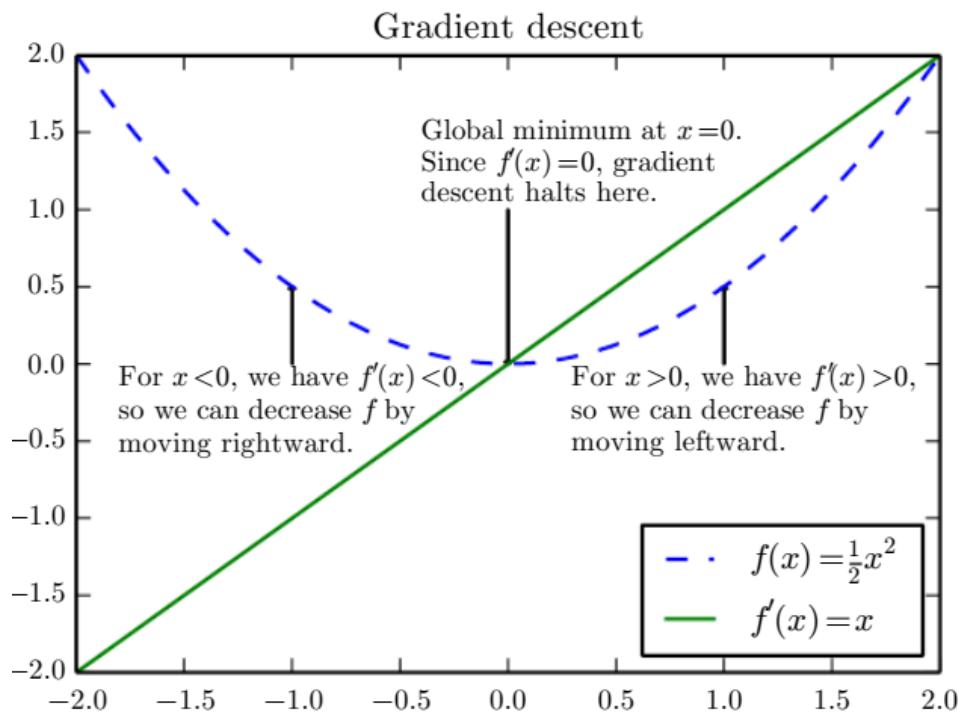


图 14.2 梯度下降

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 $f'(-1) = -1$ 。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 $x=1$, $f'(1) = 1$, 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。

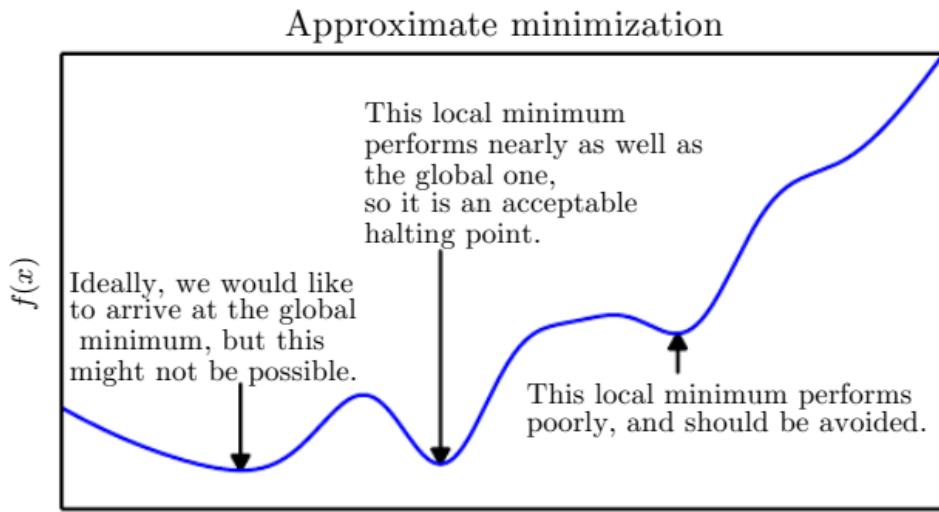


图 14.3 优化过程

当一个函数 $f(x)$ 有多个输入 (在机器学习的语境下就是机器学习模型有多个参数) 时, 偏导数 $\frac{\partial}{\partial x_i} f(x)$ 度量当在点 x 仅仅变量 x_i 增加, f 怎样改变。

就是求得 x 每个变量的偏导。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。在多维的情况下, 临界点就是梯度中的每个元素的值都是零。这就称为梯度下降。在该方法中一个新的点 x 通过下面的公式确定

$$x' = x - \epsilon \nabla_x f(x)$$

ϵ 称为学习率

对于有多个输入和多个输出的情况, 我们不讲了, 请参考《Deep Learning》一书的 4.3 节。

14.4.3 有约束的优化

有时，我们希望不仅仅是在 x 所有可能取值范围最大化或最小化一个函数 $f(x)$ ，而是希望在一个集合 S 内发现，这称为有约束的优化。位于集合 S 内的点 x 称为可行点 (feasible points)。

我们有时希望发现的解的值比较小。通常的方法是加上一个范数约束。例如， $\|x\| \leq 1$ 一种简单的有约束优化方法是在实施梯度下降时，考虑约束条件。例如，对于上面的问题。在梯度下降迭代过程的每一步

$$x' = x - \epsilon \nabla_x f(x)$$

都考察是不是新产生的点 x' 满足约束条件 $\|x\| \leq 1$ 。或者，当不满足约束条件时，改变学习率 ϵ 。直到找到满足约束条件的新的点。

更复杂的有约束优化是将有约束的优化转换成无约束的优化问题。详细内容见《Deep Learning》一书 4.4 节。

第五节：机器学习基础

关于机器学习的本质：

Machine learning is essentially a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decreased emphasis on proving confidence intervals around these functions

关于机器学习算法：

一个机器学习算法是能够从数据里进行学习的算法。但是，什么叫“学习”呢？

Mitchell 对学习的定义是：

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

14.5.1 capacity, overfitting and underfitting

机器学习任务的核心调整是要在之前未见到的数据集上表现很好，而不是在训练集上。这样的能力称为泛化 (generalization)。

在一个训练集上训练了模型，我们再在这个训练集上对模型评估，此时的误差称为训练误差。优化的过程或模型训练的过程，都是在不断的减小这个训练误差。而测试误差，即在测试集上对模型的评估，希望越低越好。当我们只有训练集来训练模型，怎么才可以让在测试上的误差也小呢。统计学习理论提供了一些回答。

统计学习理论认为，数据总体符合某个分布，训练集和测试集的数据都是按照这个分布从数据总体上产生的，这称作数据产生过程 (data generating process)。而产生过程满足独立同分布假设 (independent identically distributed, i.i.d.)，即产生每条数据时，和其他数据是相互独立的，而训练集和测试集都是按照这个分布从数据总体上抽样产生的。这个分布称为数据产生分布 (data generating distribution) 用符号 p_{data} 表示。因此根据该分布和独立同分布假设，就可以用数学的方法研究训练误差和测试误差的关系。

假设数据集服从一个分布 $p(x, y)$ ，从该分布重复的抽样以产生训练集和测试集。我们再假设有一个模型，假设模型的参数固定了。那在训练集上产生的模型期望误差是等于在测试集上产生的期望误差。因为，两个数据集的产生是从同一个数据分布上的抽样过程。唯一的区别是两个数据集的名字不同。

当然，如果该模型的参数不是固定的，我们在训练集上训练模型，然后再使用该模型在训练集上产生的期望训练误差是小余等于期望测试误差的。一个好的机器学习算法或模型，有两个因素：

- (1) 训练误差小； (2) 训练误差和测试误差之间的 gap 小。

这两个因素对应这机器学习里的两个中心挑战：欠拟合 (underfitting) 和过拟合 (overfitting)。当模型不能在训练集上获得足够低的误差率，称为欠拟合。当训练误差和测试误差之间的 gap 太大，产生过拟合。通过改变模型的容量 (capacity) 可以控制模型是可能过拟合还是可能欠拟合。一个模型的容量 (capacity) 是它拟合很宽范围函数的能力。具有高容量的模型很容易记住训练集上的那些可以导致在测试集上表现不好的特性而产生过拟合。

一个方法是控制模型的容量。《Deep Learning》这一段说的比较形式化，比较抽象。它说控制模型的容量，即控制允许机器学习算法选择作为解 的函数集合，也称为假设空间 (hypothesis space)。对于一个线性回归模型 $y = w_1x + b$ ，如果我们允许多项式的线性回归，如 $y = w_1x + w_2x^2 + b$ 。那么假设空间就扩大了。

注：之所以称这个多项式回归模型是线性的，是因为 y 和参数 w 是线性关系。

按照我的话说，模型的容量 capacity 就是模型参数的个数。模型参数越多，它的 capacity 越大。

当模型的容量和真实任务的复杂度匹配，以及有足够的训练数据，则模型可以完成的最好。如果模型的容量不够，则不适合解决复杂任务，会发生欠拟合。但是当模型容量比真实任务复杂度大，则会发生过拟合。

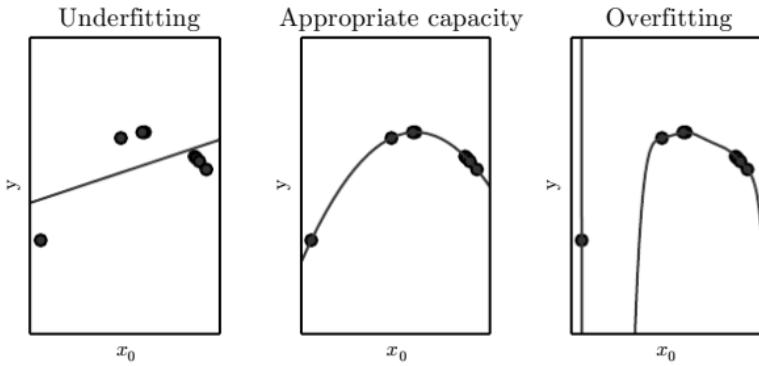


图 14.4 模型拟合

增加数据集的大小是缩小测试误差和训练误差之间 gap 的一个办法，如图 14.5 显示

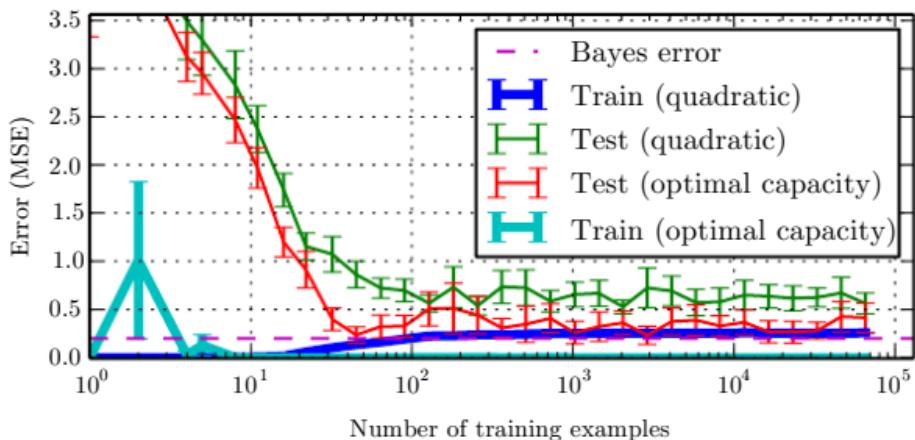


图 14.5 训练样本大小对误差的影响

可以看到随着数据集的增加，蓝色的线，即一个二项式的线性回归模型 $y = w_1 x + w_2 x^2 + b$ 的训练误差在增加，这是因为训练集越大拟合越困难。同时测试误差绿线在减小。但二项式的线性回归模型本身容量小，所以后面测试误差还是停留在一个比较高的值。当选择了一个最优容量的一个模型（红线）训练误差和测试误差最后都很小。

图 14.6 显示了一个多项式线性回归模型最优容量和训练集大小的关系。一个多项式线性回归模型

$$y = \sum_{i=1}^n w_i x^i$$

当 n 取值越大时，模型的复杂度越高，容量越大。但也不是越大越好，需要针对具体的问题。如针对图 14.6 所示的某个问题，当训练样本足够大时， $n=15$ 就是最好的容量。但当训练数据比较小的时候，合适的模型容量也比较小。

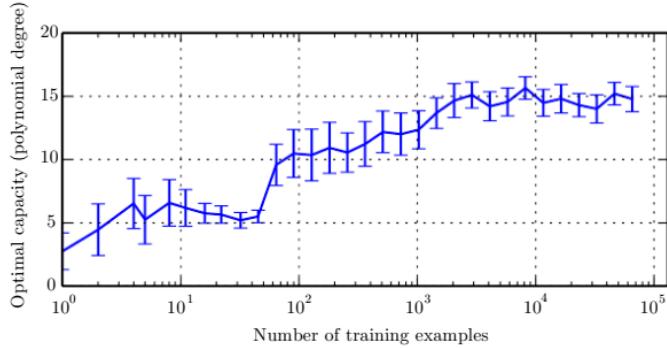


图 14.6 训练样本大小与模型容量的关系

没有免费的午餐定理 (no free lunch theorem)

机器学习里的“没有免费的午餐定理”是说，在所有可能的数据产生分布 (data generating distribution)，每个分类算法在未观察到的数据点上的分类，平均下来有同样的错误率。换句话说，在某种意义上，没有一个机器学习算法可以在任何一个任务上胜过所有其他的算法。但也可以说，我们针对特定的任务，设定特定的算法是可能胜过其他算法的。这意味着，机器学习研究的目标，不是寻找一个通用的最优机器学习算法，而是针对特定的任务设计特定的最优算法。

14.5.2 正则化 (Regularization)

我们可以给我们的学习算法加上偏好。例如，可以更改线性回归模型训练时的目标函数，以实现权重衰减 (weight decay)

$$J(\mathbf{w}) = \text{MSE}_{\text{train}} + \lambda \mathbf{w}^\top \mathbf{w}$$

该目标函数表示，希望不但要最小化误差平方均值，还希望权重有更小的平方 L2 范数。系数 λ 控制我们对小的权重的偏好。当 $\lambda = 0$ ，表示没有偏好。越大的 λ 使得，权重越小。在最小化目标函数 $J(\mathbf{w})$ 时，算法会在最小化误差和更小的权重之间做权衡。

下面的例子是怎样通过控制权重衰减来控制过拟合或欠拟合。下图是一个多项式回归模型，当 λ 过大时，权重太小，函数拟合的不好。但当 $\lambda = 0$ 时，因为对权重没有控制，则发生了过拟合。

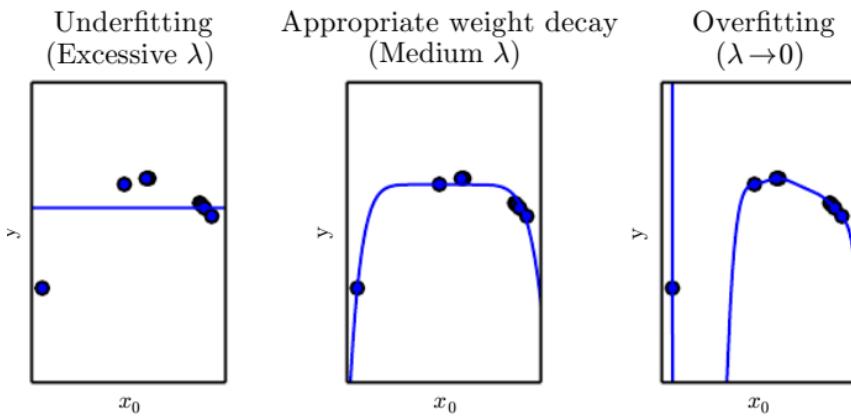


图 14.7 权重衰减对拟合的影响

上面的公式也是我们通常的做法，即通过在目标函数中加入惩罚项 $\lambda w^T w$ 来控制模型的过拟合。这个操作也称作正则化，惩罚项也称作正则化项。Deep Learning一书中，将所有防止过拟合的策略都称作是正则化。

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.

“没有免费的午餐定理”说明，没有一种最好的机器学习算法。同样的，也没有最好的形式的正则化。在深度学习通常的哲学是，一个很宽范围的任务类型都可以使用通用目的的正则化

14.5.3 超参数与校验集

大部分的机器学习算法会包含一些设置，用来控制机器学习算法的行为。这些设置称为超参数 (hyperparameters)。超参数的值并不会在机器学习算法的学习过程中被改变或被学习（但我们也可以用另外一个学习算法来学习当前算法的超参数）。例如

$$J(\mathbf{w}) = \text{MSE}_{\text{train}} + \lambda \mathbf{w}^\top \mathbf{w}$$

中的 λ 就是一个超参数。

经常，我们之所以不在训练集上学习超参数，是因为所有的超参数都在控制模型的容量。如果，在训练集上学习超参数，它会选择使得模型容量最大的超参数。如此，模型就会过拟合了。例如，上式中的 lamda，会倾向于选择 $\lambda = 0$ 。

因此，我们需要一个校验集 (validation set)。它是训练集上未观察到的样本。前面我们提到了将数据集划分成训练集和测试集。测试集是测试最终模型的，而我们为了调试超参数，需要从训练集中划分一部分出来构成校验集。典型的，可以使用训练集中的 80% 的数据作为训练集，剩下的 20% 数据作为校验集。

再描述一下上面的数据集划分过程，我们会把数据集划分成训练集和测试集。测试集用于评估模型的泛化能力。而当模型有超参数时，训练集又被划分成两部分：训练集和校验集。在训练集上训练模型，在校验集上考察模型的性能以寻找合适的超参数。

但如果测试集太小了，测试误差就有很大的统计不确定性。如此，在测试集上评估的模型性能就没有说服力。当数据集的大小达到数十万以上，这不是问题。但如果数据集比较小，可以采用 k -折交叉确认 (k -fold cross validation) 的方式。简单的说，就是把数据集划分成 k 份，然后训练 k 次模型。每一次，用不同的一份做测试集，剩下的 $k-1$ 份作为训练集。把 k 次测试的结果的均值作为最终模型的性能。

14.5.4 维度灾难 (curse of dimensionality)

机器学习中，当输入数据的维度增加，机器学习任务变得更加复杂，学习难度更高。这一现象称为维度灾难。下面通过一个例子来解释这个问题。



图 14.8 维度灾难的例子

假设当前的输入数据只有一个特征（一个维度）（左图），例如，教育程度。它是一个分类属性，有 10 个可选项（小学、初中、...）。我们的训练数据有 10 条，恰好每条数据的‘教育程度’的取值都不一样。则这 10 条数据恰好满足我们的训练模型的要求，因为每个可能取值（区域）都有样本覆盖。举个例子，如果一个区域没有覆盖样本，我们估计概率分布时，会错误的以为，在该区域（取值）的概率就是零。

现在我们增加了一个特征，收入水平。则当前数据有两个维度。‘收入水平’也是一个离散的分类属性，包括 10 个档次的收入水平（1000 以下、1000-2000、...）。我们看中间的图。对于两个特征，它们的划分区域则有 100 个。（<小学，1000 以下>,<小学，1000-2000>,...）。我们训练模型时，这 100 个区域需要都被覆盖。则我们最少需要 100 条数据。以此类推，当增加到维度为 3 时（右图），假设新增加的特征也有 10 个可能取值。则至少需要 1000 条训练数据。

由此可以看到，我们增加一个维度，对训练样本量的需求呈指数增长。而我们的训练样本数量不能满足这一需求时，这带来了维度灾难。

图 14.9 展示了一个分类器性能和训练数据中特征数的关系。当训练样本的特征数增加时，一开始分类器的性能也增加，但是达到最高点后，如果此时继续增加特征数，而没有相应的增加训练样本，则分类器的性能开始下跌。

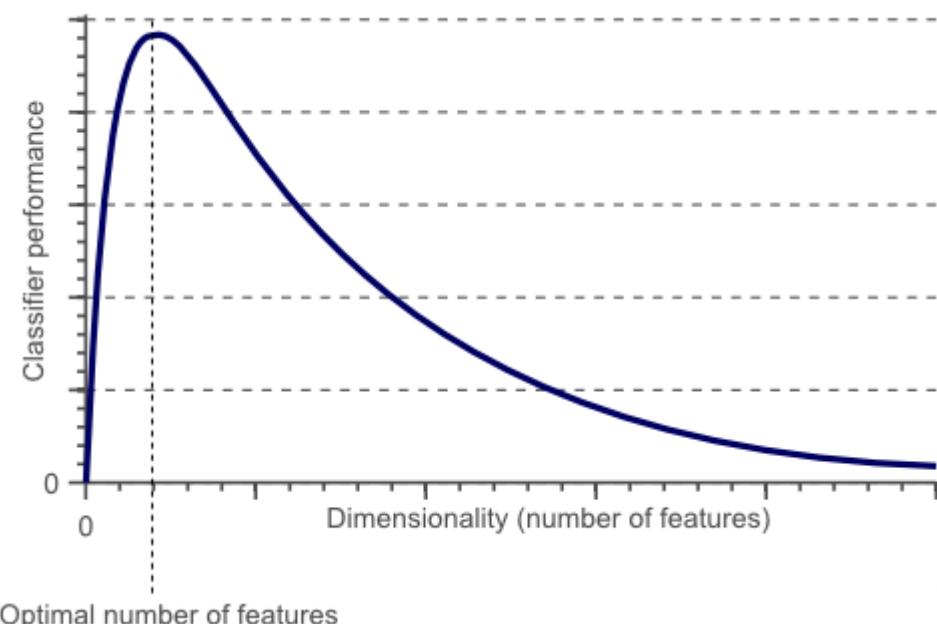


图 14.9

那么对于一个机器学习任务，我们应该有多大的训练样本呢？这取决于你面对的任务的复杂性和你采用的算法。例如，我们在 iris 数据集上对鸢尾花的类型进行分类。一条数据，只有三个特征。这是一个简单分类任务，我们用简单模型 Logistics 回归就可以完成任务。我因此需要的训练数据可以很小（iris 数据集只有 150 条数据）。但对于人脸识别任务，我们需要采用容量很大的模型，如深度学习模型，则对训练数据的需求量就非常大了。

下面有一些启发式的方法帮助你估算你的任务需要多大样本量。

(<https://machinelearningmastery.com/much-training-data-required-machine-learning/>)

对于一个分类问题，样本量是类别数目、输入特征、模型参数的函数。

- **Factor of the number of classes:** There must be x independent examples for each class, where x could be tens, hundreds, or thousands (e.g. 5, 50, 500, 5000).
- **Factor of the number of input features:** There must be $x\%$ more examples than there are input features, where x could be tens (e.g. 10).

- **Factor of the number of model parameters:** There must be x independent examples for each parameter in the model, where x could be tens (e.g. 10).

这篇网文的作者指出，尽量的获得你能获得的数据。

附录 A：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)

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 $x \text{ if } x > 0 \text{ and } \alpha * (\exp(x) - 1) \text{ if } x < 0$

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 \(ELUs\)](#)”

6. 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

7. 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>

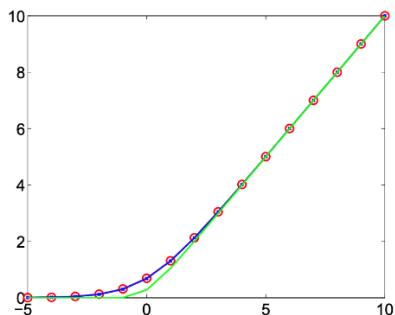
定义如下：

```
def selu(x):
    alpha = 1.6732632423543772848170429916717
    scale = 1.0507009873554804934193349852946
    return scale * K.elu(x, alpha)
```

8. softplus

```
keras.activations.softplus(x)
```

softplus 被看做是 relu 的平滑版。 $f(x) = \log(1 + \exp(x))$



在 Goodfellow 的 deep learning 一书中指出，relu 的性能比 softplus 更好，不鼓励使用 softplus.

9. Softsign

```
x / (abs(x) + 1)
```

10. 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 $\text{threshold} \leq x < \max_value$, $f(x) = \alpha * (x - \text{threshold})$

11. tanh

```
tf.keras.activations.tanh(x)  
  
tanh(x) = (exp(x) - exp(-x)) / (exp(x) + exp(-x))
```

12. sigmoid

```
keras.activations.sigmoid(x)  
  
1 / (1 + exp(-x))
```

13. hard_sigmoid

```
0 if x < -2.5  
  
1 if x > 2.5  
  
0.2 * x + 0.5 if -2.5 <= x <= 2.5.
```

14. exponential

$\exp(x)$

第三节： keras 常用的损失函数

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（**这个整数必须从 0 开始，且连续**），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], [.05, .89, .06], [.05, .01, .94]]))
print('Loss: ', loss.numpy())
```

或者在模型的 `compile` 函数中定义损失函数类型时，给字符串描述

`"sparse_categorical_crossentropy"`

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

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 ≥ 0 . Learning rate.

momentum: float hyperparameter ≥ 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 Nestervor 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 α

`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 ρ_1

`beta_2=0.999`, 二阶动量的指数衰减率, 4.4 节 Adam 算法中的 ρ_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 更有优势。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 Neterov 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 $\text{output} = \text{activation}(\text{dot}(\text{input}, \text{kernel}) + \text{bias})$

Unit 是该层包含的神经元个数。

Activation 是选择的激活函数，不选择则是线性激活函数

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 的 shape 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))
```

11. Batch Normalization

```
tf.keras.layers.BatchNormalization(  
    axis=-1, momentum=0.99, epsilon=0.001, center=True,  
    scale=True, beta_initializer='zeros', gamma_initializer='ones',  
    moving_mean_initializer='zeros',  
    moving_variance_initializer='ones', beta_regularizer=None,  
    gamma_regularizer=None, beta_constraint=None,  
    gamma_constraint=None, renorm=False, renorm_clipping=None,  
    renorm_momentum=0.99, fused=None, trainable=True,  
    virtual_batch_size=None, adjustment=None, name=None, **kwargs  
)
```

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

完成两个 tensor 加法操作的一个类，它是在创建一个层。对应的 add 方法也可以。例如，

```
added = tf.keras.layers.add([x1, x2])  
added = tf.keras.layers.Add()([x1, x2])
```

add 要求两个 tensor 在第一个维度，即 batch 上一致，后面的可以不一致，例如

```
a1=np.array([[1,2,3],[3,4,5]])  
a2=np.array([[1],[2]])  
u2 = tf.keras.layers.add([a1, a2])  
u2 的值是  
array([[2, 3, 4],  
       [5, 6, 7]])>
```

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)
```

一个层，计算两个 tensor 的相乘。 axes 设置沿着哪个维度进行点乘。通过设置 axes，它可以完成矩阵相乘的计算。举个例子：

```
x = np.arange(10).reshape(1, 5, 2)

# 此时 x 的 shape 是 1, 5, 2

y = np.arange(10, 20).reshape(1, 2, 5)

# 此时 y 的 shapes= (1,2,5)

# y 的第二个维度上和 x 的第一个维度点乘，因此得到 shape=(1,2,2)的 tensor

tf.keras.layers.Dot(axes=(2, 1))([y, x])

# x 的第一个维度上和 y 的第二个维度上点乘，得到的结果和上面一样

tf.keras.layers.Dot(axes=(1, 2))([x, y])

# x 的第二个维度上和 y 的第一个维度点乘，得到的是 shape=(1,5,5)的 tensor

tf.keras.layers.Dot(axes=(2, 1))([x, y])
```

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)
```

第七节：Backend Function

<https://faroit.com/keras-docs/2.1.3/backend/>

Keras is a model-level library, providing high-level building blocks for developing deep learning models. It does not handle itself low-level operations such as tensor products, convolutions and so on. Instead, it relies on a specialized, well-optimized tensor manipulation library to do so, serving as the "backend engine" of Keras. Rather than picking one single tensor library and making the implementation of Keras tied to that library, Keras handles the problem in a modular way, and several different backend engines can be plugged seamlessly into Keras.

1.keras.backend.dot(x, y)

2. keras.backend.batch_dot(x, y, axes=None)

Batchwise dot product.

3. keras.backend.transpose(x)

4. keras.backend.gather(reference, indices)

Retrieves the elements of indices indices in the tensor reference.

5. keras.backend.max(x, axis=None, keepdims=False)

6. keras.backend.min(x, axis=None, keepdims=False)

7. keras.backend.sum(x, axis=None, keepdims=False)

8. keras.backend.prod(x, axis=None, keepdims=False)

Multiply the values in a tensor, alongside the specified axis.

9. keras.backend.cumsum(x, axis=0)

Cumulative sum of the values in a tensor, alongside the specified axis.

10. keras.backend.cumprod(x, axis=0)

Cumulative product of the values in a tensor, alongside the specified axis.

11. keras.backend.mean(x, axis=None, keepdims=False)

12. keras.backend.std(x, axis=None, keepdims=False)

Standard error

13. keras.backend.var(x, axis=None, keepdims=False)

方差

14. keras.backend.any(x, axis=None, keepdims=False)

Bitwise reduction (logical OR).

15. keras.backend.all(x, axis=None, keepdims=False)

Bitwise reduction (logical AND).

16. keras.backend.argmax(x, axis=-1)

17. keras.backend.argmin(x, axis=-1)

18. keras.backend.square(x)

19. keras.backend.abs(x)

20. keras.backend.sqrt(x)

21. keras.backend.exp(x)

22. keras.backend.log(x)

keras.backend.logsumexp(x, axis=None, keepdims=False)

Computes $\log(\sum(\exp(\text{elements across dimensions of a tensor})))$.

This function is more numerically stable than $\log(\sum(\exp(x)))$. It avoids overflows caused by taking the exp of large inputs and underflows caused by taking the log of small inputs.

23. keras.backend.round(x)

24. keras.backend.sign(x)

25. keras.backend.pow(x, a)

26. keras.backend.clip(x, min_value, max_value)

27. keras.backend.equal(x, y)

28. keras.backend.not_equal(x, y)

29. keras.backend.greater(x, y)

30. keras.backend.greater_equal(x, y)

31. keras.backend.less(x, y)

32. keras.backend.less_equal(x, y)

33. keras.backend.maximum(x, y)

34. keras.backend.minimum(x, y)

35. keras.backend.sin(x)

36. keras.backend.cos(x)

keras.backend.normalize_batch_in_training(x, gamma, beta, reduction_axes, epsilon=0.001)

keras.backend.batch_normalization(x, mean, var, beta, gamma, epsilon=0.001)

Apply batch normalization on x given mean, var, beta and gamma.

keras.backend.concatenate(tensors, axis=-1)

37. keras.backend.reshape(x, shape)

keras.backend.permute_dimensions(x, pattern)

Transpose dimensions. pattern should be a tuple or list of dimension indices, e.g. [0, 2, 1].

38. keras.backend.repeat_elements(x, rep, axis)

Repeat the elements of a tensor along an axis, like np.repeat.

If x has shape (s_1, s_2, s_3) and $\text{axis}=1$, the output will have shape $(s_1, s_2 * \text{rep}, s_3)$.

第八节：创建一个自定义层

https://www.tensorflow.org/api_docs/python/tf/keras/layers/Layer

Keras 中一个层的类是

```
tf.keras.layers.Layer(  
    trainable=True, name=None, dtype=None, dynamic=False, **kwargs  
)
```

所有要创建的层都继承自该类。参数的含义如下：

trainable	Boolean, whether the layer's variables should be trainable.
name	String name of the layer.

当开发者创建自定义层时，可以有选择的把下面几个方法重写，完成自己的功能。

- `__init__()`: Defines custom layer attributes, and creates layer state variables that do not depend on input shapes, using `add_weight()`.
- `build(self, input_shape)`: This method can be used to create weights that depend on the shape(s) of the input(s), using `add_weight()`. `__call__()` will automatically build the layer (if it has not been built yet) by calling `build()`.
- `call(self, *args, **kwargs)`: Called in `__call__` after making sure `build()` has been called. `call()` performs the logic of applying the layer to the input tensors (which should be passed in as argument). Two reserved keyword arguments you can optionally use in `call()` are:
 - `training` (boolean, whether the call is in inference mode or training mode)
 - `mask` (boolean tensor encoding masked timesteps in the input, used in RNN layers)
- `get_config(self)`: Returns a dictionary containing the configuration used to initialize this layer. If the keys differ from the arguments in `__init__`, then override `from_config(self)` as well. This method is used when saving the layer or a model that contains this layer.

当进行定制时，主要步骤：

- (1) 建立一个新的类继承 Layer 这个类。例如，

```
class Linear(keras.layers.Layer):
```

(2) 在新类中，定义`__init__()`这个方法，它是新类的构造方法。例如，

```
def __init__(self, units=32, input_dim=32):
```

上面的例子中，`units`和`input_dim`是当前的新类自己需要的参数。本身`Layer`父类就有很多参数，新类也继承了。为了在新类中使用这些参数可以在`__init__`函数加上`**kwargs`。例如，

```
def __init__(self, n=None, **kwargs):
```

(3) 定义`call`方法，它完成该层的计算。

创建定制层，并使用它的例子如下：

```
class Linear(keras.layers.Layer):
    def __init__(self, units=32, input_dim=32):
        super(Linear, self).__init__()
        self.w = self.add_weight(
            shape=(input_dim, units),
            initializer="random_normal", trainable=True
        )
        self.b = self.add_weight(shape=(units,), 
                               initializer="zeros", trainable=True)

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b

x = tf.ones((2, 2))
linear_layer = Linear(4, 2)
y = linear_layer(x)
print(y)
```

如果有更复杂的定制层的需求，可参考

https://www.tensorflow.org/guide/keras/custom_layers_and_models

第九节：tf.Variables

当我们有更复杂的定制需求时，需要用到 tf.Variables 这个类。Tensorflow 中的 variable 用于描述编程操作中共享的、持久的状态。下面的内容涉及到怎样在 tensorflow 中创建、更新和管理 tf.Variables 的实例。

<https://www.tensorflow.org/guide/variable>

1. 创建 Variable

可以通过初始值来创建 Variable，此时 Variable 的类型与初始值一样。

```
my_tensor = tf.constant([[1.0, 2.0], [3.0, 4.0]])
my_variable = tf.Variable(my_tensor)
```

当然也可以创建布尔型的 variable，此处不讨论。

Variable 的 shape、dtype 和 numpy 属性，描述了他的 shape、类型和包含的值。完全可以把 variable 当作一个 tensor 来看待。设置 variable 的 trainable=False，可以不让训练模型时梯度更新该 variable。

Variable 可以直接使用，来参与运算。但如果要更新 variable 中的内容，不能直接修改，需要用 Variable.assign 来改变。

附录 B：函数光滑

一般来说，神经网络处理的东西都是连续的浮点数，标准的输出也是连续型的数字。但实际问题中，我们很多时候都需要一个离散的结果，比如分类问题中我们希望输出正确的类别，“类别”是离散的，“类别的概率”才是连续的；又比如我们很多任务的评测指标实际上都是离散的，比如分类问题的精确度（accuracy）和 F1、机器翻译中的 BLEU，等等。

还是以分类问题为例，常见的评测指标是精确度（accuracy），而常见的损失函数是交叉熵。交叉熵的降低与 accuracy 的提升确实会有一定的关联，但它们不是绝对的单调相关关系。换句话说，交叉熵下降了，accuracy 不一定上升。显然，如果能用 accuracy 的相反数（error rate）做损失函数，那是最理想的，但 accuracy 不可导的（涉及到 argmax 等操作），所以没法直接用。

这时候一般有两种解决方案；一是动用强化学习，将 accuracy 设为奖励函数，这是“用牛刀杀鸡”的方案；另外一种是试图给 accuracy 找一个光滑可导的近似公式。下面就来探讨一下常见的不可导函数的光滑近似，有时候称之为“光滑化”或“软化”。

1. max 函数的光滑近似

$$\max(x_1, x_2, \dots, x_n) = \lim_{K \rightarrow +\infty} \frac{1}{K} \log \left(\sum_{i=1}^n e^{Kx_i} \right)$$

选定常数 K 就有

$$\max(x_1, x_2, \dots, x_n) \approx \frac{1}{K} \log \left(\sum_{i=1}^n e^{Kx_i} \right)$$

在模型中很多时候 K 可以设为 1，这等价于把模型融合到自身中，所以简单的有：

$$\begin{aligned} \max(x_1, x_2, \dots, x_n) &\approx \log \left(\sum_{i=1}^n e^{x_i} \right) \\ &\triangleq \text{logsumexp}(x_1, x_2, \dots, x_n) \end{aligned}$$

logsumexp 算子是 max 函数的光滑近似。

2. Softmax 函数

softmax 函数是 onehot(argmax(x)) 的光滑近似。即先求出最大值所在的位置，然后生成一个等长的向量，最大值的位置为 1，其他位置为 0，比如

$$[2,1,3,5,4] \rightarrow [0,0,0,1,0]$$

从 logsumexp 到 softmax 的推导过程如下：

考虑向量 $x = [x_1, x_2, \dots, x_n]$ ，然后考虑 $x' = [x_1, x_2, \dots, x_n] - \max(x_1, x_2, \dots, x_n)$

即每一位都减去整体的最大值，这样的新向量与原向量最大值所在位置是一样的。即

$$\text{onehot}(\arg \max(x)) = \text{onehot}(\arg \max(x'))$$

不失一般性，考虑 x_1, x_2, \dots, x_n 两两不想等的情况。那么新向量的最大值为 0，并且除去最大值其他值都是负数。这样，可以考虑

$$e^{x'} = [e^{x_1 - \max(x_1, x_2, \dots, x_n)}, e^{x_2 - \max(x_1, x_2, \dots, x_n)}, \dots, e^{x_n - \max(x_1, x_2, \dots, x_n)}]$$

将 max 函数的近似打入上面公式就可得到

$$\begin{aligned} \text{onehot}(\arg \max(x)) &= \text{onehot}(\arg \max(x')) \\ &\approx \left(\frac{e^{x_1}}{\sum_{i=1}^n e^{x_i}}, \frac{e^{x_2}}{\sum_{i=1}^n e^{x_i}}, \dots, \frac{e^{x_n}}{\sum_{i=1}^n e^{x_i}} \right) \\ &\triangleq \text{softmax}(x_1, x_2, \dots, x_n) \end{aligned}$$

在神经网络实施多类分类时，输出层用 softmax 函数就是对找出分类的结果，即一个类别给出 1 的标签其他是 0，的近似。

3. argmax 函数

argmax 函数是直接给出向量最大值所在的下标，比如，假设下标从 1 开始

$$[2,1,3,5,4] \rightarrow 4$$

对 argmax 进行光滑近似，希望它输出一个接近 4 的浮点数，为了构造这样的近似，先观察 argmax 实际上等于

$$\text{sum}\left(\underbrace{[1, 2, 3, 4, 5]}_{\text{序向量 } [1, 2, \dots, n]} \otimes \underbrace{[0, 0, 0, 1, 0]}_{\text{onehot}(\arg \max(x))}\right)$$

即序向量[1,2,3,4,5]和 onehot(arg max([1,2,3,4,5]))的内积。如此将 onehot(argmax(x))换成 softmax(x)就可以了。

$$\arg \max(\mathbf{x}) \approx \sum_{i=1}^n i \times \text{softmax}(\mathbf{x})_i$$

4. 精确度 (accuracy)

上述讨论的若干个近似都是在 onehot 向量的基础上推导出正确形式，然后用 softmax 近似 onehot，从而得到光滑近似。用这个思想还可以得出许多算作的光滑近似，比如 精确度。

用符号 1_k 表示第 k 位为 1 的 onehot 向量。假设在分类问题中，目标类别是 i，预测类别是 j，那么可以考虑 onehot 向量 1_i 和 1_j ，然后考虑内积

$$\langle \mathbf{1}_i, \mathbf{1}_j \rangle = \begin{cases} 1, & (i = j) \\ 0, & (i \neq j) \end{cases}$$

也就是说，两个类别一样时，内积刚好是 1。两个类别不一样时，内积是 0。所以目标类别和预测类别对应的 onehot 向量的内积高刚好定义了一个“预测正确”的计数函数，有了计数函数就可以算精确度 accuracy。

$$\text{accuracy} = \frac{1}{|B|} \sum_{x \in B} \langle 1_i(x), 1_j(x) \rangle$$

B 是 batch。该公式及计算一个 batch 精确度的函数。而在神经网络中，为了保证可导性，最后的输出只能是一概率分布（softmax 函数的结果），所以精确度的光滑近似就是将预测类别的 onehot 向量，换成概率分布

$$\text{accuracy} = \frac{1}{|B|} \sum_{x \in B} \langle 1_i(x), p(x) \rangle$$

这样导出的 accuracy 是可导的，直接可以使用 1-accuracy 作为损失函数。以二分类为例， x 为输入， $y \in \{0,1\}$ 为目标（类别标签）， $p \in \{0,1\}$ 。光滑的 1-accuracy 是

$$\text{smooth_accuracy} = \mathbb{E}_{(x,y) \sim \mathcal{D}} [y p_\theta(x) + (1 - y)(1 - p_\theta(x))]$$

\mathcal{D} 是训练集集合。Smooth_accuracy 直接用于优化效果并不好，更好的是去优化交叉熵

$$\text{cross_entropy} = \mathbb{E}_{(x,y) \sim \mathcal{D}} [-y \log p_\theta(x) - (1 - y) \log(1 - p_\theta(x))]$$

类似的也可以导出 recall, F_β 的光滑近似。以二分类为例, 假设 $p(x)$ 是正类的概率, $t(x)$ 是样本 x 的标签 $(0,1)$, 则正类的 F_1 的光滑近似度是

$$\text{正类F1} \approx \frac{2 \sum_{\mathbf{x} \in \mathcal{B}} t(\mathbf{x})p(\mathbf{x})}{\sum_{\mathbf{x} \in \mathcal{B}} [t(\mathbf{x}) + p(\mathbf{x})]}$$

附录 C: 多 GPU 训练模型

https://keras.io/guides/distributed_training/

在 keras 中可以在具有多 GPU 的单机上训练模型，也可以在多机器上训练模型。

1. 单机多 GPU 训练模型

此时需要使用 API

```
# Create a MirroredStrategy.  
strategy = tf.distribute.MirroredStrategy()  
print('Number of devices: {}'.format(strategy.num_replicas_in_sync))  
  
# Open a strategy scope.  
with strategy.scope():  
    # Everything that creates variables should be under the strategy scope.  
    # In general this is only model construction & `compile()`.  
    model = Model(...)  
    model.compile(...)  
  
    # Train the model on all available devices.  
    model.fit(train_dataset, validation_data=val_dataset, ...)
```

即创建 strategy 后，所有建立模型的过程都放在 with stratege.scope 下面。即，层的创建，还包括创建模型 model，和模型的编译。不包括 fit

举例如下：

```
strategy = tf.distribute.MirroredStrategy()  
print("Number of devices: {}".format(strategy.num_replicas_in_sync))  
with strategy.scope():
```

```

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])
dot_product = Reshape((1,))(dot_product)

# 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')

```

附录 D：一些技巧

1. 在 tf.keras 中，将 tensor 转换成 numpy 的数据对象

可以直接调用 tensor 对象的 numpy()方法，就可得到。例如，

```
import tensorflow as tf  
t = tf.constant([1, 2, 3])  
n = t.numpy()  
print(type(t), type(n))  
print(t)  
print(n)
```

如果此时产生了错误信息：

```
AttributeError: 'Tensor' object has no attribute '_numpy'
```

在程序的最开始，加上语句

```
tf.config.run_functions_eagerly(True)
```

注：该函数只适用于 tensorflow 2.3 以上版本。另外，这是在 tensorflow 的 Eager Execution 环境下。该环境下，没有创建 tensorflow 的 graph。它适合调试。如果是在 GPU 上运行程序。在该环境下会导致 GPU 内存占满，但 GUP-util=0 的情况。即没有执行程序。那建议数据流图上的运算，还是在 tensor 的对象上进行运算，不要转换成 numpy。

2. 定制评价函数

一种简单的方法是，定义一个函数。它的输入是目标值和预测值

```
def my_metric(y_true, y_pred):  
    x = y_pred.numpy()  
    x1 = x.transpose()  
    x2 = x1/x1.max(axis=0)  
    x3 = x2.transpose()  
    return np.sum(np.array(y_true)*np.floor(x3))/len(y_true)
```

在 compile 模型时，将该函数作为期中一个参数值。

```
model.compile(loss='categorical_crossentropy', optimizer=op,
metrics=[m, my_metric])
```

3. 将训练中最优的模型保存

在 4.4 节介绍正则化策略时，提到过早停 earlystopping。当校验集上的性能就停止训练。另外，还有一种策略，将每趟训练中在校验集上性能最好的一次训练结果保存到文件。训练完成后，再读出该文件来构建最终的模型。

首先创建 ModelCheckpoint

```
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath,
    save_weights_only=True,
    monitor='val_accuracy',
    mode='max',
    save_best_only=True)
```

它的参数设置了，模型保存的文件，是否只是保存模型的权重；在什么指标上评估模型性能；按照指标最大值来保存还是最小值；是不是仅仅保存最优的模型。

在训练模型时，在 fit 函数的 callbacks 参数设置为上述的 model check point 对象

```
model.fit(epochs=EPOCHS, callbacks=[model_checkpoint_callback])
```

训练完成后，读出文件中的模型。因为上面是保存的模型参数，因此调用模型的 load_weights 方法将最优性能时的参数装回。

```
model.load_weights(checkpoint_filepath)
```

5. Tensorflow 的 `tf.norm(tensor, ord)` 函数可以计算一个 vector, matrix 或 tensor 的范数。参数 `ord` 可以是 '`fro`', '`euclidean`', 1, 2, 指示 frobenius 范数，向量的欧式长度，L1、L2 范数。也可以给一个整数，计算 P 范数。

例如，`tf.norm(T, ord=1)` 计算一个 vector, matrix 或 tensor 的 L1 范数

附录 E：出错信息处理

```
1. ValueError: Failed to find data adapter that can handle input:  
(<class 'list'> containing values of types {"<class  
'numpy.ndarray'>", '<class \'list\'> containing values of types  
{"<class \'int\'>"}}), (<class 'list'> containing values of types  
{"<class 'int'>"})
```

该问题是因为喂入模型的数据格式有问题，例如
`model.fit([train_docs,train_labels], train_match, epochs=10,
verbose=1)`

`train_docs` 是 numpy 的 array 数据结构

```
array([[ 7664,  436, 9508, ...,    0,    0,    0],  
       [ 1068, 6952,  724, ...,    0,    0,    0],  
       [  56,   40,   28, ...,    0,    0,    0],  
       ...,  
       [ 2496,  328,  680, ...,    0,    0,    0],  
       [ 1116, 9400, 4444, ...,    0,    0,    0],  
       [ 2632, 7596, 10660, ...,    0,    0,    0]])
```

而 `train_labels` 是 list 数据结构 [1,1,0,2,3,1...]

此时应该吧 list 的数据结构转换成列向量的 array，例如
`train_labels=np.array(train_labels).reshape(-1,1)`结果是

```
array([[0],  
       [7],  
       [0],  
       ...,  
       [3],  
       [1],  
       [3]])
```

2. ValueError: Failed to convert a NumPy array to a Tensor (Unsupported object type list).

跟第一个问题一样

3. InternalError: Failed copying input tensor from...

这是在多人共享一台服务器时。服务器的 GPU 资源被别人占用了。

4. NotImplementedError: Cannot convert a symbolic Tensor (2nd_target:0) to a numpy array

把 numpy 从 1.20 降级到 1.19 就好了。

6. ValueError: tf.function-decorated function tried to create variables on non-first call.

在共自定义的正则项中使用 K.eye 来建立共单位矩阵时，出现上面的错误。改用 tf.eye 即可。

勤

7. OP_REQUIRES failed at cudnn_rnn_ops.cc:1510 : Unknown: Fail to find the dnn implementation

我在学院服务器上运行程序时显示这个错误信息。

原因是和别的程序共用一个节点。资源不够了。

可以在程序里设置

```
physical_devices = tf.config.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(physical_devices[0],
enable=True)
tf.config.experimental.set_memory_growth(physical_devices[1],
enable=True)
```

附录 F: Tensorflow Hub and Addons

Tensorflow Hub 是一个预训练的机器学习模型库。用户可以使用该库中的模型，通过精调来完成下游任务。

如果安装了 tensorflow 2.X，在 anaconda 的一个环境下，在 terminal 中运行命令

```
pip install --upgrade tensorflow-hub
```

就可以安装 Tensorflow Hub。

Tensorflow addons 是没有包含在 tensorflow 的核心库，但按照 API 标准实施的一些新功能的库。因为深度学习发展的很快，一些新功能来不及集成到核心库里。安装命令如下：

```
pip install tensorflow-addons
```

附录 G: Tensorflow 的一些数据操作函数

在 tf.keras 中编程，当涉及到更核心的部分，比如需要定义自己的层和 module。很多时候需要 tensorflow 提供的函数。下面列举一些常用的函数。

1. tf.reshape

```
tf.reshape(  
    tensor, shape, name=None  
)
```

按照给出的 shape，给 tensor 重新调整 shape

2. tf.expand_dims

```
tf.expand_dims(  
    input, axis, name=None  
)
```

在 axis 的维度插入一个新的维度。例如

```
image = tf.zeros([10,10,3])
```

创建一个维度为[10, 10, 3]的 tensor

```
Image = tf.expand_dims(image, axis=0)
```

image 的维度是[1, 10, 10, 3]

3. tf.reduce_sum 或 tf.math.reduce_sum

```
tf.math.reduce_sum(  
    input_tensor, axis=None, keepdims=False, name=None  
)
```

计算一个 tensor 跨某个维度的和

4. 如果想进行 numpy 一样的切片操作参考

https://www.tensorflow.org/guide/tensor_slicing

附录 H: Encoder-Decoder+注意力机制的英中翻译模型

译模型

改编自这篇文章的代码 https://www.tensorflow.org/text/tutorials/nmt_with_attention

先介绍两个用到的类

1. TextVectorization

建立的模型是一个可以投入使用的模型，即喂入英文字符串返回中文字符串。因此文本的处理应该在模型内部完成。因此使用 `layers.TextVectorization`。它是一个预处理层，将文本映射到整数序列或者其他向量如 tf-idf 值。

```
tf.keras.layers.TextVectorization(  
    max_tokens=None,  
    standardize='lower_and_strip_punctuation',  
    split='whitespace',  
    ngrams=None,  
    output_mode='int',  
    output_sequence_length=None,  
    pad_to_max_tokens=False,  
    vocabulary=None,  
    idf_weights=None,  
    sparse=False,  
    ragged=False,  
    encoding='utf-8',  
    **kwargs  
)
```

这一层要使用的词汇表或者外部提供，或者通过 `adapt()` 来学习。

每个送入该层的样例的处理包括下面的步骤：

- (1) 对每个样例标准化处理，包括转换成小写和去除标点符号
- (2) 划分样例到 token
- (3) 分配 token 编号
- (4) 转换样例到编号序列。

Args	
<code>max_tokens</code>	Maximum size of the vocabulary for this layer. This should only be specified when adapting a vocabulary or when setting <code>pad_to_max_tokens=True</code> . Note that this vocabulary contains 1 OOV token, so the effective number of tokens is (<code>max_tokens - 1 - (1 if output_mode == "int" else 0)</code>).
<code>standardize</code>	Optional specification for standardization to apply to the input text. Values can be: <ul style="list-style-type: none">• None: No standardization.• "lower_and_strip_punctuation": Text will be lowercased and all punctuation removed.• "lower": Text will be lowercased.• "strip_punctuation": All punctuation will be removed.• Callable: Inputs will pass to the callable function, which should be standardized and returned.
<code>split</code>	Optional specification for splitting the input text. Values can be: <ul style="list-style-type: none">• None: No splitting.• "whitespace": Split on whitespace.• "character": Split on each unicode character.• Callable: Standardized inputs will pass to the callable function, which should be split and returned.
<code>ngrams</code>	Optional specification for ngrams to create from the possibly-split input text. Values can be None, an integer or tuple of integers; passing an integer will create ngrams up to that integer, and passing a tuple of integers will create ngrams for the specified values in the tuple. Passing None means that no ngrams will be created.
<code>output_mode</code>	Optional specification for the output of the layer. Values can be " <code>int</code> ", " <code>multi_hot</code> ", " <code>count</code> " or " <code>tf_idf</code> ", configuring the layer as follows: <ul style="list-style-type: none">• "<code>int</code>": Outputs integer indices, one integer index per split string token. When <code>output_mode == "int"</code>, 0 is reserved for masked locations; this reduces the vocab size to <code>max_tokens - 2</code> instead of <code>max_tokens - 1</code>.• "<code>multi_hot</code>": Outputs a single int array per batch, of either <code>vocab_size</code> or <code>max_tokens</code> size, containing 1s in all elements where the token mapped to that index exists at least once in the batch item.• "<code>count</code>": Like "<code>multi_hot</code>", but the int array contains a count of the number of times the token at that index appeared in the batch item.• "<code>tf_idf</code>": Like "<code>multi_hot</code>", but the TF-IDF algorithm is applied to find the value in each token slot. For "<code>int</code>" output, any shape of input and output is supported. For all other output modes, currently only rank 1 inputs (and rank 2 outputs after splitting) are supported.
<code>output_sequence_length</code>	Only valid in INT mode. If set, the output will have its time dimension padded or truncated to exactly <code>output_sequence_length</code> values, resulting in a tensor of shape (<code>batch_size, output_sequence_length</code>) regardless of how many tokens resulted from the splitting step. Defaults to None.
<code>pad_to_max_tokens</code>	Only valid in " <code>multi_hot</code> ", " <code>count</code> ", and " <code>tf_idf</code> " modes. If True, the output will have its feature axis padded to <code>max_tokens</code> even if the number of unique tokens in the vocabulary is less than <code>max_tokens</code> , resulting in a tensor of shape (<code>batch_size, max_tokens</code>) regardless of vocabulary size. Defaults to False.

vocabulary	Optional. Either an array of strings or a string path to a text file. If passing an array, can pass a tuple, list, 1D numpy array, or 1D tensor containing the string vocabulary terms. If passing a file path, the file should contain one line per term in the vocabulary. If this argument is set, there is no need to <code>adapt()</code> the layer.
idf_weights	Only valid when <code>output_mode</code> is " <code>tf_idf</code> ". A tuple, list, 1D numpy array, or 1D tensor of the same length as the vocabulary, containing the floating point inverse document frequency weights, which will be multiplied by per sample term counts for the final <code>tf_idf</code> weight. If the <code>vocabulary</code> argument is set, and <code>output_mode</code> is " <code>tf_idf</code> ", this argument must be supplied.
ragged	Boolean. Only applicable to " <code>int</code> " output mode. If True, returns a <code>RaggedTensor</code> instead of a dense <code>Tensor</code> , where each sequence may have a different length after string splitting. Defaults to False.
sparse	Boolean. Only applicable to " <code>multi_hot</code> ", " <code>count</code> ", and " <code>tf_idf</code> " output modes. If True, returns a <code>SparseTensor</code> instead of a dense <code>Tensor</code> . Defaults to False.
encoding	Optional. The text encoding to use to interpret the input strings. Defaults to " <code>utf-8</code> ".

该类的 adapt 方法

```
adapt(
    data, batch_size=None, steps=None
)
```

参数如下：

Arguments	
<code>data</code>	The data to train on. It can be passed either as a <code>tf.data.Dataset</code> , or as a numpy array.
<code>batch_size</code>	Integer or None. Number of samples per state update. If unspecified, <code>batch_size</code> will default to 32. Do not specify the <code>batch_size</code> if your data is in the form of datasets, generators, or <code>keras.utils.Sequence</code> instances (since they generate batches).
<code>steps</code>	Integer or None. Total number of steps (batches of samples) When training with input tensors such as TensorFlow data tensors, the default None is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined. If <code>x</code> is a <code>tf.data</code> dataset, and ' <code>steps</code> ' is None, the epoch will run until the input dataset is exhausted. When passing an infinitely repeating dataset, you must specify the <code>steps</code> argument. This argument is not supported with array inputs.

看一个例子：

```
text_dataset = tf.data.Dataset.from_tensor_slices(["foo", "bar", "baz"])
max_features = 5000 # Maximum vocab size.
max_len = 4 # Sequence length to pad the outputs to.

vectorize_layer = tf.keras.layers.TextVectorization(
    max_tokens=max_features,
    output_mode='int',
    output_sequence_length=max_len)

vectorize_layer.adapt(text_dataset.batch(64))

model = tf.keras.models.Sequential()
model.add(tf.keras.Input(shape=(1,), dtype=tf.string))
model.add(vectorize_layer)
```

```
input_data = [["foo qux bar"], ["qux baz"]]
model.predict(input_data)
```

输出结果是

```
array([[2,1,4,0],
[1,3,0,0]])
```

需要注意的一点是，TextVectorization 会自动的给词汇表在开始位置添加两个词，”和 ‘[UNK]’。有时会引起一些不必要的问题。

另外一个问题。保存 TextVectorization 对象到磁盘，下次处理文本时又将它读入。但该层不能直接保存，下面用一点小技巧，将该层加入到一个模型，可以保存模型。然后再读入模型，取出该层。例如，下面代码中 source_text_processor 是一个 TextVectorization 层

保存层

```
s_model = tf.keras.models.Sequential()
s_model.add(tf.keras.Input(shape=(1,), dtype=tf.string))
s_model.add(source_text_processor)
s_model.save("source_text_processor.model", save_format="tf")
```

取出层

```
s_model = tf.keras.models.load_model("source_text_processor.model")
source_text_processor = s_model.layers[0]
```

2. MultiHeadAttention

该层实现多头注意力。多头注意力是在 Transformer 中使用的，我们这里也可以使用该层来实现 encoder-decoder 中的注意力机制。

```
tf.keras.layers.MultiHeadAttention(
    num_heads,
    key_dim,
    value_dim=None,
    dropout=0.0,
    use_bias=True,
    output_shape=None,
```

```

        attention_axes=None,
        kernel_initializer='glorot_uniform',
        bias_initializer='zeros',
        kernel_regularizer=None,
        bias_regularizer=None,
        activity_regularizer=None,
        kernel_constraint=None,
        bias_constraint=None,
        **kwargs
)

```

Args

<code>num_heads</code>	Number of attention heads.
<code>key_dim</code>	Size of each attention head for query and key.
<code>value_dim</code>	Size of each attention head for value.
<code>dropout</code>	Dropout probability.
<code>use_bias</code>	Boolean, whether the dense layers use bias vectors/matrices.
<code>output_shape</code>	The expected shape of an output tensor, besides the batch and sequence dims. If not specified, projects back to the query feature dim (the query input's last dimension).
<code>attention_axes</code>	axes over which the attention is applied. <code>None</code> means attention over all axes, but batch, heads, and features.

创建该层后，进行注意力计算时的一些参数如下：

Call arguments

<code>query</code>	Query Tensor of shape (B, T, dim).
<code>value</code>	Value Tensor of shape (B, S, dim).
<code>key</code>	Optional key Tensor of shape (B, S, dim). If not given, will use value for both key and value, which is the most common case.
<code>attention_mask</code>	a boolean mask of shape (B, T, S), that prevents attention to certain positions. The boolean mask specifies which query elements can attend to which key elements, 1 indicates attention and 0 indicates no attention. Broadcasting can happen for the missing batch dimensions and the head dimension.
<code>return_attention_scores</code>	A boolean to indicate whether the output should be (<code>attention_output</code> , <code>attention_scores</code>) if True, or <code>attention_output</code> if False. Defaults to False.
<code>training</code>	Python boolean indicating whether the layer should behave in training mode (adding dropout) or in inference mode (no dropout). Will go with either using the training mode of the parent layer/model, or False (inference) if there is no parent layer.
<code>use_causal_mask</code>	A boolean to indicate whether to apply a causal mask to prevent tokens from attending to future tokens (e.g., used in a decoder Transformer).

计算返回的结果如下：

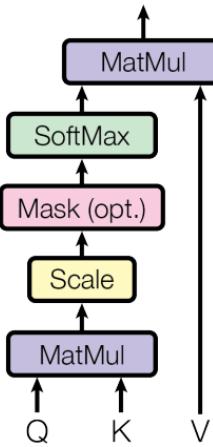
Returns ↴

<code>attention_output</code>	The result of the computation, of shape (B, T, E), where T is for target sequence shapes and E is the query input last dimension if <code>output_shape</code> is None. Otherwise, the multi-head outputs are projected to the shape specified by <code>output_shape</code> .
<code>attention_scores</code>	[Optional] multi-head attention coefficients over attention axes.

多头注意力层的输出包括 `attention_output` 和 `attention_scores`

`Attention_output` 的 shape 是 (B, T, E) B 是 `batch_size`, T 是目标序列长度, E 是输入的 valued 向量维度。如果从 encoder-decoder 的角度来理解, `attention_output[:, t, :]` 就是 decoder 中 RNN 的时间步 t 的向量与 encoder 中 RNN 的每个时间步向量计算了注意力, 并进行了加权求和得到的结果。

多头注意力机制的一个头的计算如图所示。本节中我们只使用一个头。



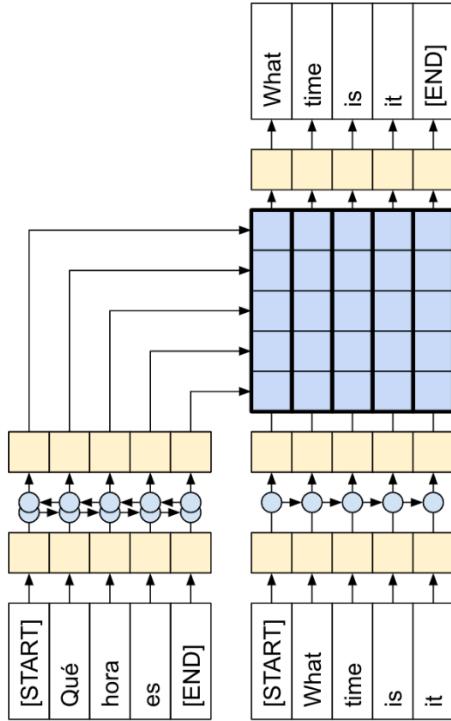
输入是查询 Q 和键 K , 它们的 shape 都是 $n \times d_k$ 。计算查询 Q 和所有键的点乘; 然后通过除以 $\sqrt{d_k}$ 进行规范化; 然后通过一个 softmax 函数获得权重。 V 的 shape= $n \times d_v$ 注意力机制的计算如下所示

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (12.1)$$

其输出是一个 shape= $n \times d_v$ 的 tensor。该文指出, 对齐函数 $\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$ 采用点乘计算效率更高, 需要的空间更小。而此处除以 $\sqrt{d_k}$ 的规范化操作是因为, 当 d_k 很大时点乘的结果在数量级上将增大, 从而推动 softmax 函数到一个梯度非常小的区域。(该文举例说明为什么点乘的数量级会变大: 假设 q 和 k 是均值为 0, 方差为 1 的两个随机变量。它们的点乘结果均值为 0, 方差为 d_k)。

3. 模型结构

该模型的结构如图所示。



在训练阶段数据集包含三个部分：encoder 的输入即 source, decoder 的输入即 target 和 decoder 的输出即错开一位的 target。

Encoder 端的 RNN 的输出和 Decoder 端的 RNN 的输出，应用注意力机制产生每个 Decoder RNN 时间步对应的 context 向量 c 。注： c_i 就是 decoder 第 i 个时间步计算的 context 向量。设 Encoder RNN 的输出为 $H \in \mathbb{R}^{h \times d}$, Decoder RNN 的输出为 $S \in \mathbb{R}^{s \times d}$ 。 h 和 s 是 RNN 的时间步，注意力计算的结果是 $C \in \mathbb{R}^{s \times d}$ 即产生的 context 向量 c_i 的集合。计算步骤如下：

$$A = \text{softmax}(H \times S^T), A \in \mathbb{R}^{h \times s}$$

解码输出的词在原始文章中的实施中是一个简化版，它把 $s'_i = s_i + c_i$ 作为了一个时间步的输出。我没有改动，但可以在这个地方进行改进。

$$Y = \text{softmax}(A^T H + S')$$

$y_i \in Y$ 是解码器在时间步 j 解码得到的一个词。

在推断阶段。首先 Encoder 把源文本喂入模型得到 RNN 的每个时间步的输出。在 Decoder 完成逐个词的解码（这里中文是在字符上的解码）。即采用循环方法，每趟在给 RNN 喂入上一趟循环解码的词和 RNN 的状态，然后用 Encoder RNN 的输出和 Decoder RNN 当前输出（只是一个时间步）计算注意力，并按照上面介绍的方法解码出最终的词。解码第一个词时，喂入的是起始符 ‘\t’，接收 Encoder 最后一个时间步的状态，作为 Decoder RNN 的初始状态。然而，原始的那篇文章中没有这么做，没有

给初始状态。我发现给 Encoder 最后一个时间步的状态作为 Decoder RNN 第一个时间步的初始状态，性能要好很多。

模型的构建过程中，创建了四个类 Encoder, CrossAttention, Decoder 和 Translator 来搭建模型。

更详细的实施步骤，参见文件

Encode_decoder_translation_attention_models.py 保存了模型的构件，即四个类。

Encode_decoder_translation_attention_train.py 训练模型并保存到磁盘

Encode_decoder_translation_attention_inference.py 装载训练好的模型，进行推断。