西南财经大学经济信息工程学院

深度学习与文本挖掘

Deep Learning for Text Mining

研究生课程讲义

邱江涛 编著

Tenflow1.10 版

目录

第一章：前言

第一节：深度学习的兴起

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

图 1-1.

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 更是添了一把火。

Reinforcement Learning 不是本课程的授课内容。下面简单介绍一下其原理(选自维基百科)。

Reinforcement Learning 通常翻译做增强学习（或强化学习）是机器学习领域的一项技术。它的产生灵感来自“行为心理学”。它研究的问题是怎样让 Agent（有翻译作主体）在一个环境中采取的行为最大化“累积奖励”（ cumulative *reward*）。一个增强学习的 Agent 在离散的时间点（时刻）和它的环境交互。在每个时间 t，agent 接收一个观察 o_t，它包括一个奖励（reward）r_t。Agent 然后从一系列行为（action）中选择一个行为（action）a_t，其后这项选择被送回到环境。环境于是转移到一个新的状态 s_{t+1}，与这一转移相关联的 reward r_{t+1} 被确定。增强学习 agent 的目标是尽可能的收集奖励（reward）

Agent 为了采取最优的行动，必须 推理它的行为的长期结果。例如，为了最大化我的未来收入，最好的选择是进学校学习，虽然，这一行为为当前带来的是金钱是损失。增强学习特别适合这一类问题，它们需要在长期和短期利益之间做权衡。增强学习已经在许多领域取得了成功，如机器人控制、电梯调度和游戏领域。

深度学习取得的成就

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

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

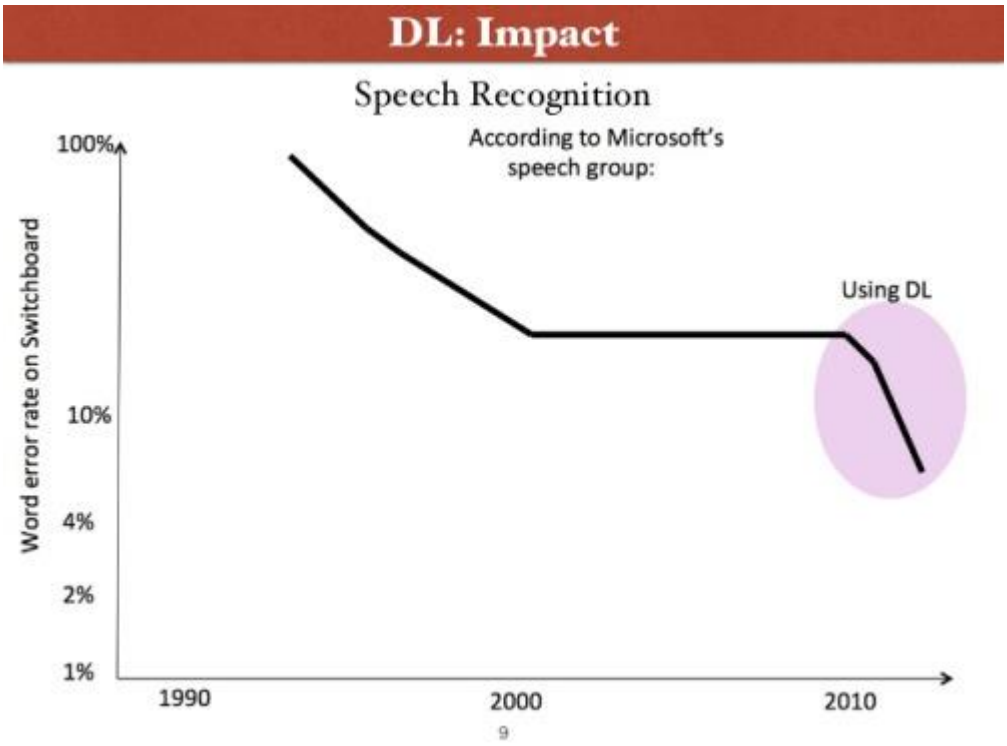


图 1-2.

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

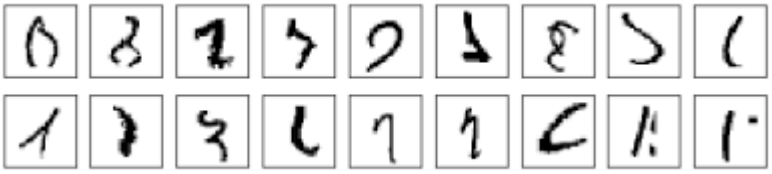


图 1-3.

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

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

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

神经网络与深度学习的关系

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

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

神经网络是由生物学启发的编程机器学习模型（有称是编程范式 programming paradigm），由此计算机可以从观察的数据进行学习。深度学习是一个非常有力的在

神经网络上进行学习的技术集合。神经网络和深度学习当前对图像处理、语言识别和自然语言处理中的许多问题提供了最好的解决方案。

深度学习比起传统神经网络的优势在于：

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

知识获取与深度学习[1]

在专家系统时期，领域专家（主要是语言学家）直接提供显性知识。显性知识人类可以直接构建和理解的知识。详见图 1-4。

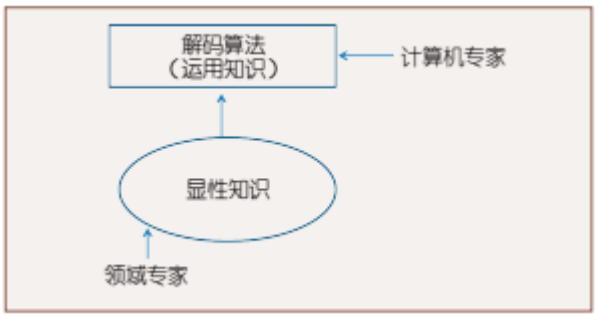
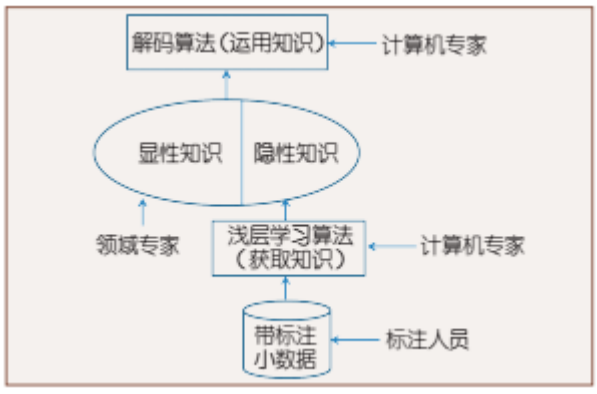


图 1-4. 专家系统

当进入语料库方法主导的时期，领域专家提供的显性知识仍然起到关键作用，例如决定识别对象的特征等。此时标注人员标注的小规模数据中所蕴含的知识更全面、 更精确、 更加量化。详见图 1-5。



[1]刘 挺 车万翔，自然语言处理中的知识获取问题， 中国计算机学会通讯，第 13 卷 第 5 期 2017 年 5 月

图 1-5. 语料库方法

深度学习"端到端" 的方法兴起后， 有些简单问题， 如果能够找到足够多的大数据，则会按照图 3 的模式获取知识， 即显性知识完全退场。 **深度学习系统几乎不需要领域专家提供的元知识， 由机器自己确定特征， 并分配各层的功能**， 尽管这些功能的划分是隐性的， 难以解释的。详见图 1-6。

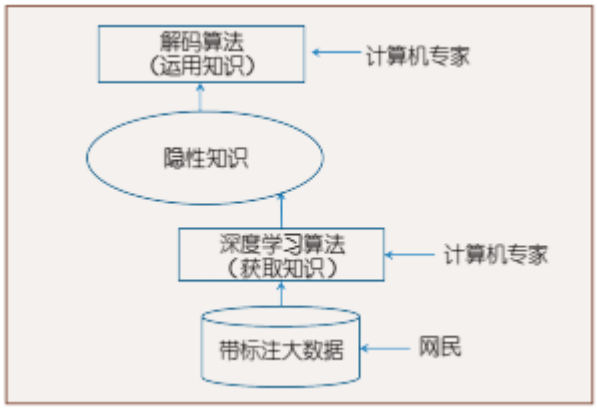


图 1-6. 基于大数据的深度学习。

需要强调的是， 这里所谓"深度学习" 的方法， 关键在于"端到端"，即把从输入到输出的全部工作交给机器去处理， 而不再人为地分层。 下面试以"信息抽取" 为例加以说明。信息抽取有两种做法 ： 一是先做句法分析，再做信息抽取 ； 二是直接做信息抽取，后者就是所谓的"端到端"。 在"端到端" 的模型中， 也是分层的， 但是由机器自己去分层处理， 各层的含义不是直观可以理解的。 当用于端到端的训练数据不足时， 就需要人为的帮助， 比如把信息抽取的过程分成两步去做 ： 第一步， 先做出句法树（这一步增加了显性知识， 但也引入了误差）； 第二步， 实施信息抽取。 当用于端到端的训练数据充足时， 就可以一步到位——直接做信息抽取， 而且性能更好。 与信息抽取类似的还有情感二元分类、 句间关系分类、问答对匹配等。

但是， 对很多问题而言， 带标注数据是有限的， 甚至是很有限的， 因此图 1-6 所示的理想情况并不多。 图 1-7 是比较现实的解决方案 ：

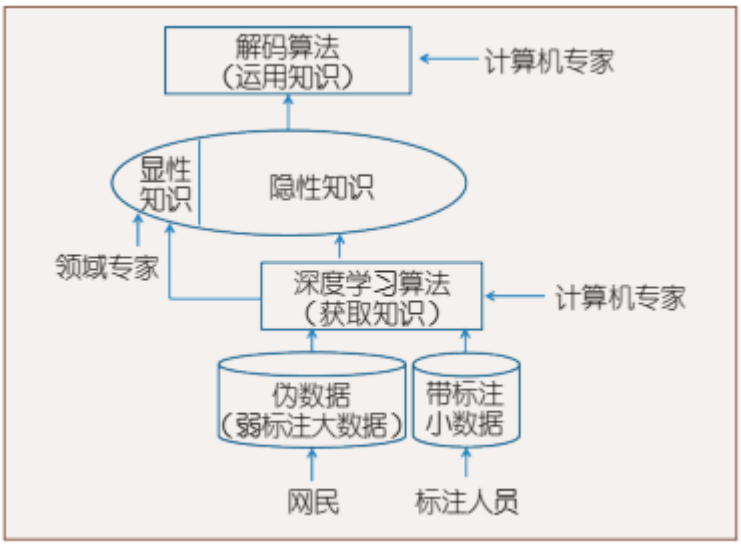


图 1-7. 基于伪数据的深度学习

将大规模“伪数据”与人工标注的小数据结合在一起，同时接受领域专家提供的部分元知识，如此，在数据不充分的情况下，借助人工的力量，追求最优的系统性能。

大数据增加深度学习的威力

深度学习不是万能的，当数据不足时，深度学习的效果大打折扣。图 1-8 显示了有用不能获得充足的数据，即使在简单的问题上，深度学习也没有明星的超越传统方法。在复杂问题上甚至有劣势。因为问题复杂， 而数据量不够时， 学习工具越强大就越容易形成过拟合， 所以效果自然不好。

	简单问题	复杂问题
小数据	无明显优势 （例如：词性标注）	有劣势 （例如：深层语义分析）
大数据	优势最明显 （例如：语言模型）	优势较明显 （例如：机器翻译）

图 1-8. 深度学习应用于自然语言处理的效果

从上述分析可以看出， 一旦拥有大规模的训练数据， 深度学习的威力是巨大的， 可以在短时间内以摧枯拉朽的气势替代原有技术。

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

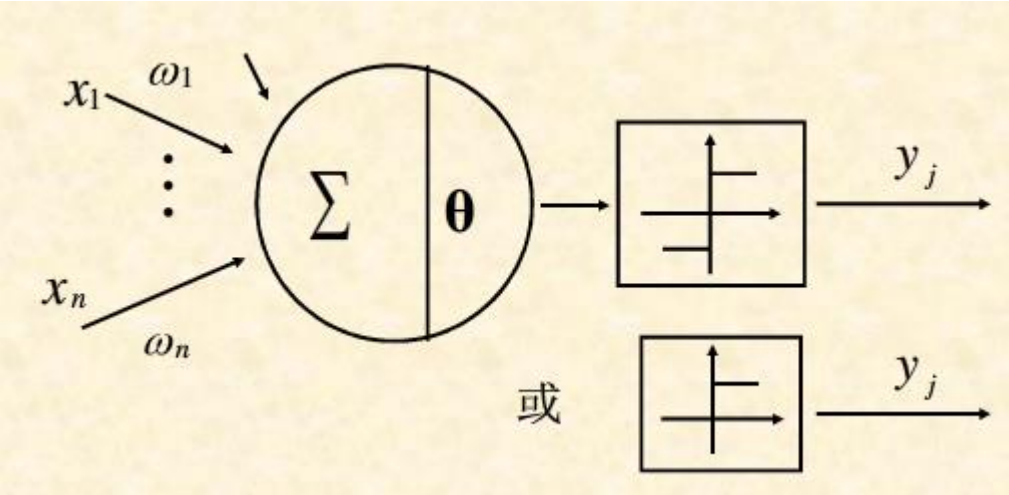
第二节：深度学习框架介绍

第三节：关于本课程

在本课程中，我们先介绍神经网络的基础知识，然后介绍 Deep Learning，进一步介绍实现 Deep Learning 的工具 TensorFlow，并用 TensorFlow 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

神经网络NN(Neural Networka)或人工神经网络ANN（Artificial Neural Network）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。



感知机的输入是向量 x=(x1,…,xn)，输出是一个 0,1 或-1,1 的值 y_j

$$s_j = \sum_{i=1}^{n} \omega_{ji} x_i - \Theta_j$$

$$s_j = \sum_{i=0}^{n} \omega_{ji} x_i, \quad \text{where} \quad (x_0 = \Theta_j, \omega_{j0} = -1)$$

$$y_j = f(s_j)$$

其中Θ_j为偏置，f 为激活函数（activation function）

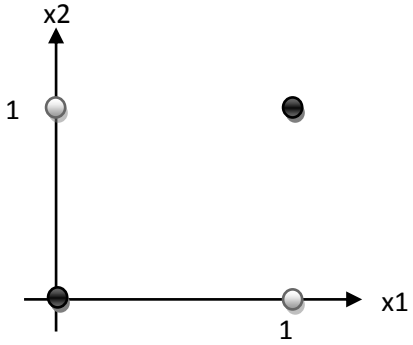
$$f(s) = \begin{cases} 1, & \text{if } s > 0 \\ 0, & \text{otherwise} \end{cases}$$

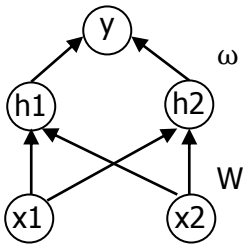
或

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

然而 Minsky and 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，它不能解决 XOR 问题。从此，神经网络的研究陷入了停滞。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。

看下列的图，有四个点对应输入 x={(0,0), (1,0), (0,1), (1,1)}。每个点对应的类别标签由他们的颜色确定。感知机不可能找一条线，将这四个点分成两类。





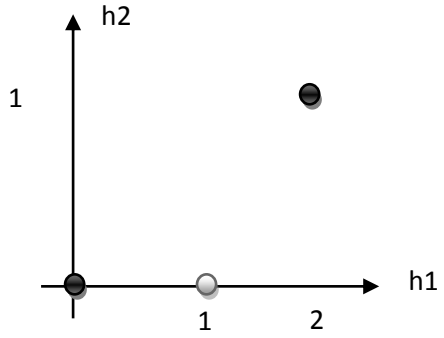
而当神经网络加入隐层后

设$W=\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}$，$c=\begin{bmatrix}0\\ -1\end{bmatrix}$，$\omega=\begin{bmatrix}1\\ -2\end{bmatrix}$，$b=0$

输出$y=f(x;W,c,\omega,b)=\omega^{T}\max\left\{0，W^{T}x+c\right\}+b$（即隐层的每个神经元使用了 relu 激活函数）

考虑对应输入 x={$(0,0)^{T}$, $(1,0)^{T}$, $(0,1)^{T}$, $(1,1)^{T}$}，输出 y 得到什么结果？

可以看到，对应的 y 是 0,1,1,0。该神经网络可以将输入数据正确分类。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。



然而，到 90 年代神经网络的研究有开始缓慢，因为神经网络的一些缺陷。直到 2006 随着 Deep Learning 的诞生，神经网络又一次新生。

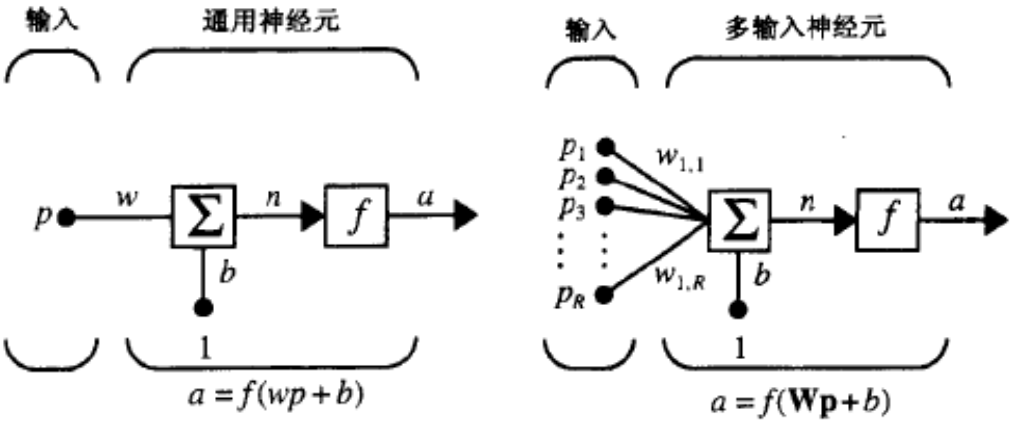
反向传播 BP 算法是神经网络最受欢迎的训练算法，但是当神经网络的层次增加时，BP 算法不能很好的训练模型。当层次增加时，训练神经网络的目标函数是一个非凸（non-convex）的函数。BP 算法通常陷入局部最优。而且当层数增加的越多，问题越严重。因此 2006 年以前的神经网络通常是浅层神经网络，它们最多包含两层非线性的特征转换（隐层）。在 2006 年，Hinton 提出了受限波兹曼机 RMB（restricted Boltzmann machines），它有效的解决了更多层神经网络的学习问题。深度学习由此而来。

第二节：神经网络的结构

神经网络的结构包括神经元（neuron），连接神经元的有向有权重的边，

2.1.1 神经元

神经网络最基本的处理单元是神经元。一个单输入神经元如图所示。标量输入 p（*这里是以标量输入作为讨论，但实际中输入很多情况下是向量*）乘上标量权重 w 得到 wp，再讲其送入累加器。另一个输入 1 乘上偏置 b，再将其送入累加器。累加器输出 *n* 通常称为净输入，它被送入到一个激活函数 f，在 f 中产生神经元的标量输出 a。



神经元按照下式计算

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

实际输出取决于激活函数。w 和 b 是神经元待学习的参数。

通常神经元有不止一个输入。具有 R 个输入的神经元如图所示。其输入 p1,…,pR 分别对应权重矩阵 W 的元素 W1,1，W1,2，…, W1,R，。该神经元只有一个偏置值 b，它与所有输入的加权和累加，从而形成净输入 n。

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

这个表达式写成矩阵为

$$n = Wp + b$$

其中单个神经元的权重矩阵只有一行元素

神经元的输出可以写成

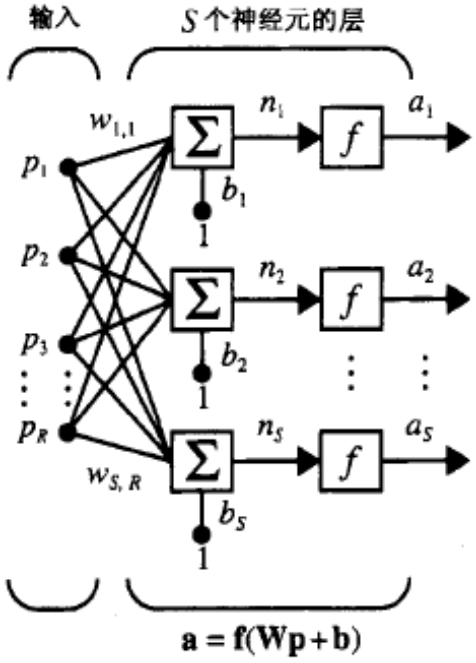
$$a = f(Wp + b)$$

神经网络通常可以用矩阵来描述

2.1.2 网络结构

一般来说，有多个输入的神经元并不能满足实际应用的要求，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**。

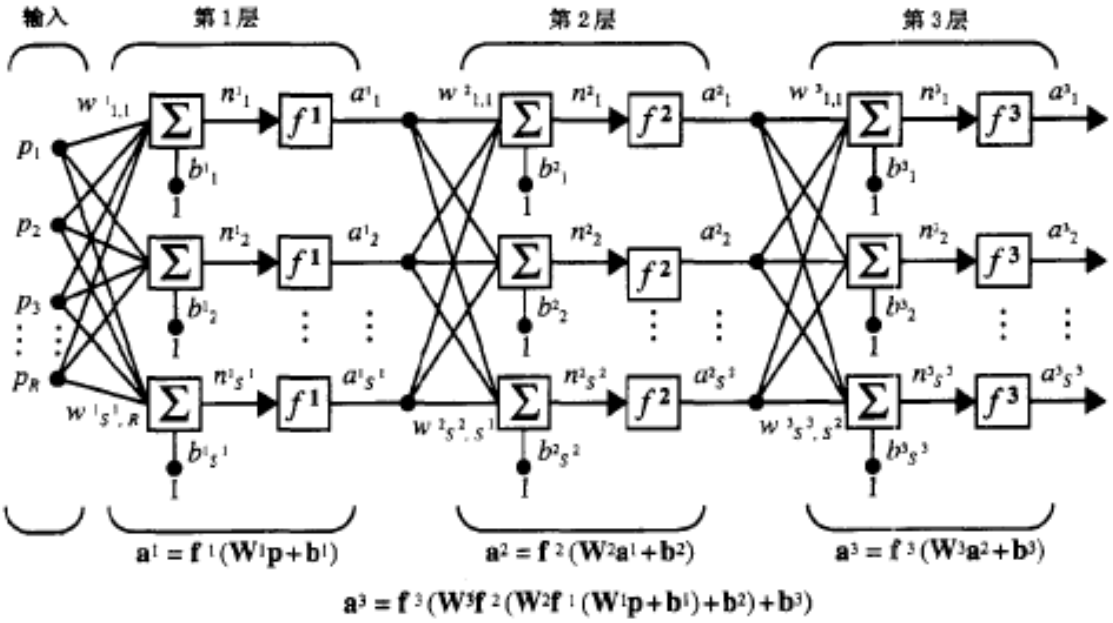
神经元的层是由 S 个神经元组成的单层网络。注意 R 个输入中的每一个均与每个神经元相连，权重矩阵现有 S 行。



通常每层的输入个数并不等于该层中神经元的数目（即$R \neq S$）。同一层中的神经元不必有相同的激活函数。输入向量（这里是指多个标量输入，组成一个输入向量）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。下图是一个三层网络（这里输入未算作是一个层）



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，那么层该层为**输出层**。和其他层称为**隐层**。

多层网络的功能要比单层网络的功能强大许多。例如，如果第一层有 S 形激活函数，第二层具有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有问题所定义的。所以，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。同样如果网络又 7 个输出，则网络的输出层就有 7 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。如果一个输出是-1 或 1，那么该输出神经元就可以用对称硬极限激活函数。对于确定多层网络中其他层的神经元个数并没有明确的确定方法。对于层数，普遍是小余 3 层。

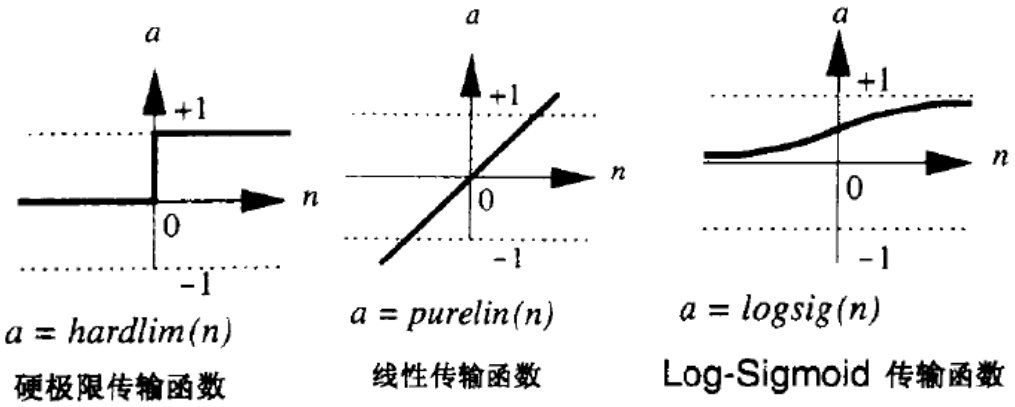
对于偏置。是否使用偏置是可以选择的。偏置给网络提供了额外的变量，从而使网络又了更强的能力。

2.1.3 激活函数（Activation Function）

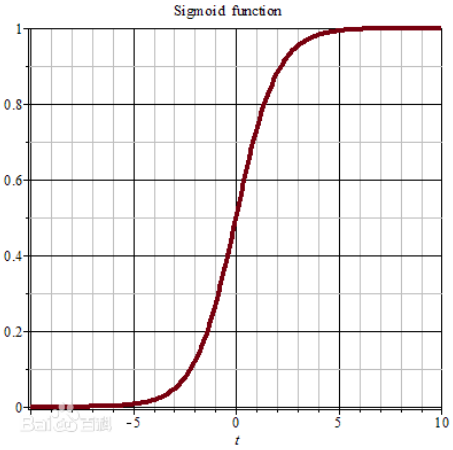
激活函数也有人称为，活化函数或传输函数。激活函数可以是 n 的线性或非线性函数。可以用特定的激活函数满足神经元要解决的特定问题。

最简单的激活函数是 binary step function（有翻译做硬极限函数）。如果输入超出了一个预设的阈值，该函数的输出从一个值转变到另一个值，否则保持原值不变。

线性传输函数。它的输出等于输入 a=n。



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



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。

Softmax 函数，将一个任意实数值的 k 维向量 z 归一化到一个实数值在[0,1]的 k 维向量$\sigma(z)$，满足向量元素的和为 1。

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

2.1.4 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。这里我们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。前面我们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

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，我们先讨论一下神经网络的训练过程。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（即线性回归）为例。

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

这里 y 是每天吃的早餐的价格。W 是三样食物的价格，x 是每样食物的份量。训练模型，即得到模型的参数是一个优化过程。

优化过程中通常需要建立一个目标函数。目标函数的建立有很多种，平方误差是其中一种。

第一步，建立损失函数（目标函数）

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

$t^{(n)}$是训练集中一顿早餐的价格，$y^{(n)}$是用模型估计的价格。用梯度下降方法来训练模型，需要对目标函数求导获得梯度

第二步，对参数（当前是权重 w_i）求导, 获得梯度

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

第三步，更新权重。用梯度乘上学习率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。

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

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

Steps:

1. 初始化权重 W
2. 对于训练集中的每个实例 n 完成下面的步骤
   a. 初始$\Delta W \leftarrow 0$
   b. 计算输出$y^{(n)} = f(Wx^{(n)})$
   c. 累加$\Delta w_i = \Delta w_i + \eta x^{(n)} \left(t^{(n)} - y^{(n)} \right)$
3. 更新权重$W = W + \Delta W$

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),165
0),((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）

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$\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$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

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，考察每个训练数据实例时，就计算一个参数的梯度，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参数的梯度，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 error=850-500=350。计算$\mathrm{delta} = \varepsilon(t^n - y^n)x^n$ =[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),165
0),((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、minibach 和标准梯度下降算法性能上有什么区别？

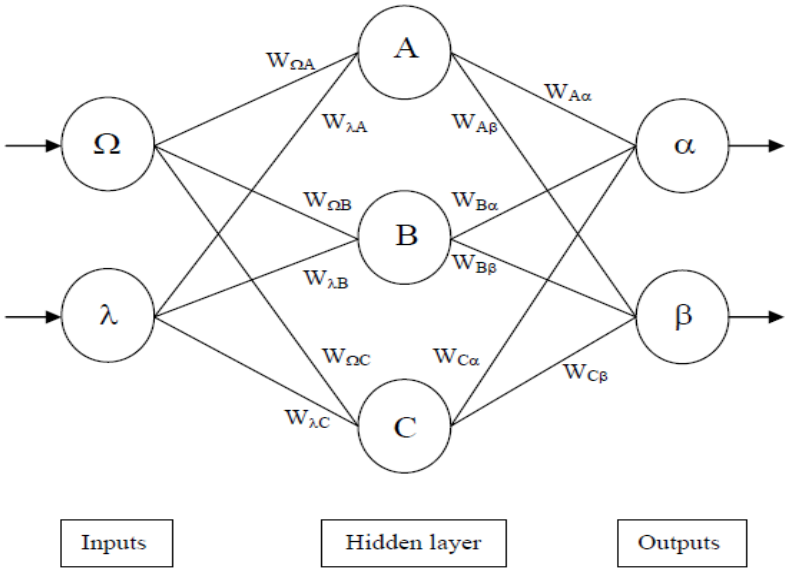
标准梯度下降每一轮迭代需要所有样本参与，对于大规模的机器学习应用，经常有 billion 级别的训练集，计算复杂度非常高。因此，有学者就提出，反正训练集只是数据分布的一个采样集合，我们能不能在每次迭代只利用部分训练集样本呢？这就是 minibatch 算法。

我们这里对 online GD 的解释是 minibathc 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。

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 Werboss 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



反向传播算法工作步骤如下：

1. 计算输出神经元的误差。

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

$$\delta_\beta = \mathrm{out}_\beta\big(1-\mathrm{out}_\beta\big)\big(\mathrm{Target}_\beta - \mathrm{out}_\beta\big)$$

注意当激活函数是 sigmoid 函数时，采用这种方法计算误差。如果激活函数是 binary step 函数则直接用$\delta_\alpha = \mathrm{Target}_\alpha - \mathrm{out}_\alpha$

2. 改变输出层权重

$$W^+_{A\alpha} = W_{A\alpha} + \eta\delta_\alpha \mathrm{out}_A \qquad W^+_{A\beta} = W_{A\beta} + \eta\delta_\beta \mathrm{out}_A$$

$$W^+_{B\alpha} = W_{B\alpha} + \eta\delta_\alpha \mathrm{out}_B \qquad W^+_{B\beta} = W_{B\beta} + \eta\delta_\beta \mathrm{out}_B$$

$$W^+_{C\alpha} = W_{C\alpha} + \eta\delta_\alpha \mathrm{out}_C \qquad W^+_{C\beta} = W_{C\beta} + \eta\delta_\beta \mathrm{out}_C$$

3. 计算隐层的误差（反向传播）

$$\delta_A = \mathrm{out}_A(1-\mathrm{out}_A)\big(\delta_\alpha W_{A\alpha} + \delta_\beta W_{A\beta}\big)$$

$$\delta_B = \mathrm{out}_B(1-\mathrm{out}_B)\big(\delta_\alpha W_{B\alpha} + \delta_\beta W_{B\beta}\big)$$

$$\delta_C = \mathrm{out}_C(1-\mathrm{out}_C)\big(\delta_\alpha W_{C\alpha} + \delta_\beta W_{C\beta}\big)$$

$\delta_\alpha W_{A\alpha} + \delta_\beta W_{A\beta}$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。

4. 改变隐层的权重

$$W^+_{\lambda A} = W_{\lambda A} + \eta\delta_A \mathrm{in}_\lambda \qquad W^+_{\Omega A} = W_{\Omega A} + \eta\delta_A \mathrm{in}_\Omega$$

$$W^+_{\lambda B} = W_{\lambda B} + \eta\delta_B \mathrm{in}_\lambda \qquad W^+_{\Omega B} = W_{\Omega B} + \eta\delta_B \mathrm{in}_\Omega$$

$$W^+_{\lambda C} = W_{\lambda C} + \eta\delta_C \mathrm{in}_\lambda \qquad W^+_{\Omega C} = W_{\Omega C} + \eta\delta_C \mathrm{in}_\Omega$$

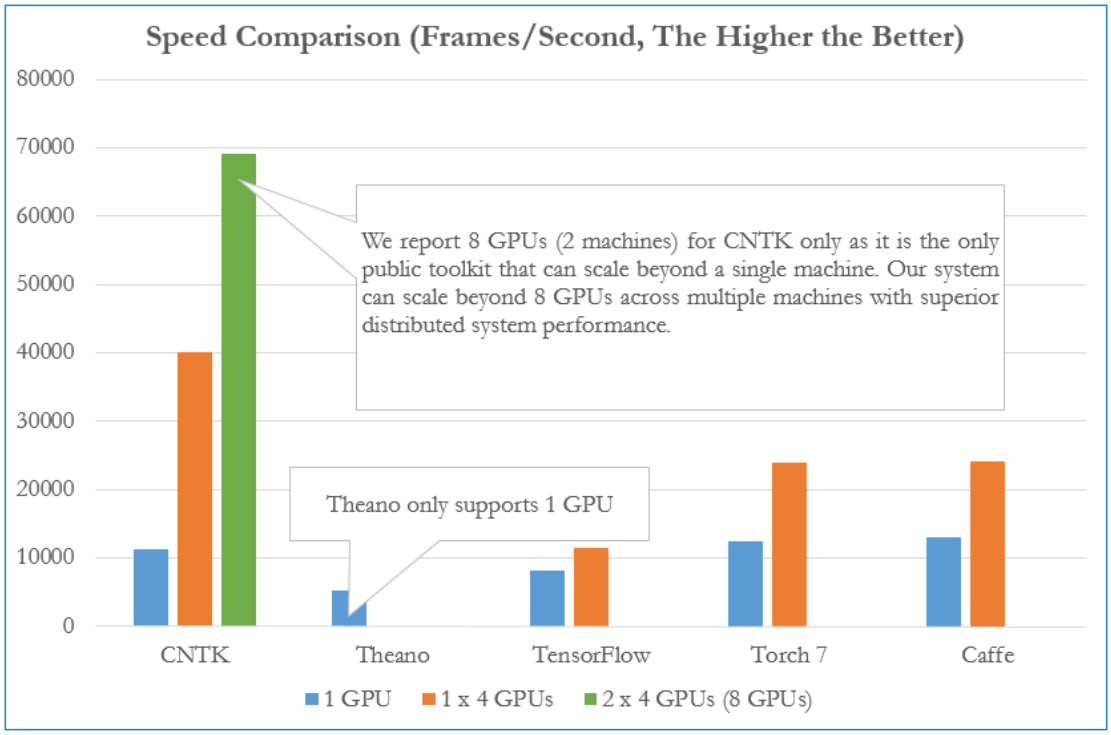
W^+代表更新后的权重

第三章：TensorFlow

在 windows 上安装 TensorFlow，参见

https://tensorflow.google.cn/install/install_windows

TensorFlow 是实现 Deep Learning 的工具之一。其他常用的 Deep Learning 工具还有 Theano, CNTK 等。下图是微软发布的各 Deep Learning 实现工具的性能比较。



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。图中的节点表示数学运算（operation），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。但该系统可以适用到非常宽广的领域。

第一节：基本概念

要使用 TensorFlow 需要理解它的基本工作原理:

（1）用图来描述计算。

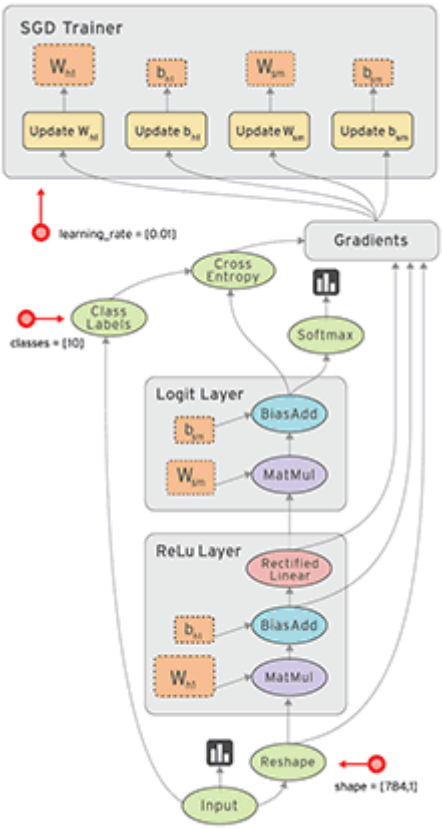
（2）在 Session 的环境下执行图

（3）用 tensor 来描述数据。

（4）用 Variable 来维护状态

（5）使用 feed 和 fetch 向任意 operation（运算）放入数据或从运算得到数据。

TensorFlow 是一套编程系统或编程框架，这里用户可以将他的计算描述成图。图中的节点称为 op (operation 的缩写)。一个 op 可以处理 0 个或多个 tensor，然后完成计算，产生 0 个或多个 tensor。一个 tensor 是一个多维阵列，例如，可以将一组图片的 mini-描述成一个 4-D 浮点数阵列。其维度为 [batch, height, width, channels].



一个 TensorFlow 的图是一个计算的描述。在计算前，该图需要在一个 session 中 launch。一个 session 将图的 operation（运算）放置到 device (装置)上，如 CPU，GPU 并提供执行运算的方法（编程概念中的方法）。这些方法将运算结果即 tensor 以 python 的 numpy 包的 ndarray 对象返回。

数据流图

数据流图用一个有边和节点的有向图描述数学运算。节点实施算术运算，但也能描述导入（feed in）数据，导出运算结果，或读写持久变量（persistent variable）的端点。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。

Session

用 TensorFlow 范式写程序，实际上是向计算图上添加操作，数据。最后执行 session 的 run（ ）以后才会进行计算操作。

Tensor

TensorFlow 使用一个 tensor 数据结构描述所有数据。在计算图上传递的只能是 tensor。一个 tensor 有一个静态类型，一个 rank，和一个 shape。

Rank

在 TensorFlow 的计算系统中，rank 是 tensor 的一个维度单位。该 rank 不同于矩阵的 rank（阶）。Tensor rank（有时称为 order, degree or n-dimention）是 tensor 维度的数量。例如，下面的 tensor（用 python 的 list 数据结构定义）的 rank 为 2。

t = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

一个 rank 为 0 的 tensor 即一个标量；一个 rank 为 1 的 tensor 即一个向量；一个 rank 为 2 的 tensor 即一个矩阵。对于 rank 为 2 的 tensort，可以通过 t[i, j]访问它的元素；对于一个 rank 为 3 的 tensor t，可以通过 t[i, j, k]来访问它的元素。

Shape

TensorFlow 使用三类符号来描述 tensor 的维度。Rank, shape 和维度数。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 [1, 4, 3].
n	[D0, D1, ... Dn-1]	n-D	A tensor with shape [D0, D1, ... Dn-1].

数据类型

除了维度，tensor 还有一个数据类型。用户可以分配下面任意一个数据类型给一个 tensor。

Data type	Python type	Description
DT_FLOAT	tf.float32	32 bits floating point.
DT_DOUBLE	tf.float64	64 bits floating point.
DT_INT8	tf.int8	8 bits signed integer.
DT_INT16	tf.int16	16 bits signed integer.

Data type	Python type	Description
DT_INT32	tf.int32	32 bits signed integer.
DT_INT64	tf.int64	64 bits signed integer.
DT_UINT8	tf.uint8	8 bits unsigned integer.
DT_STRING	tf.string	Variable length byte arrays. Each element of a Tensor is a byte array.
DT_BOOL	tf.bool	Boolean.
DT_COMPLEX64	tf.complex64	Complex number made of two 32 bits floating points: real and imaginary parts.
DT_COMPLEX128	tf.complex128	Complex number made of two 64 bits floating points: real and imaginary parts.
DT_QINT8	tf.qint8	8 bits signed integer used in quantized Ops.
DT_QINT32	tf.qint32	32 bits signed integer used in quantized Ops.
DT_QUINT8	tf.quint8	8 bits unsigned integer used in quantized Ops.

Variables

Variables maintain state across executions of the graph. 当训练一个模型时，用变量来保存和更新参数（Variable 也可以设置为是某个全局变量，而不是模型训练的参数，详见 3.2 节的 Tips）。变量是在内存中保存 tensor 的一片区域。它们必须被明确的初始化，然后在训练期间和训练完后保存在磁盘。稍后可以恢复这些变量来分析模型。来看一段代码

```
import tensorflow as tf

# Create a Variable, that will be initialized to the scalar value 0.
state = tf.Variable(0, name="counter")

# Create an Op to add one to `state`.
one = tf.constant(1)
new_value = tf.add(state, one)
update = tf.assign(state, new_value)

# Variables must be initialized by running an `init` Op after having
# launched the graph.  We first have to add the `init` Op to the
```

```
graph.
init_op = tf.initialize_all_variables()

# Launch the graph and run the ops.
with tf.Session() as sess:
  # Run the 'init' op
  sess.run(init_op)
  # Print the initial value of 'state'
  print(sess.run(state))
  # Run the op that updates 'state' and print 'state'.
  for _ in range(3):
    sess.run(update)
    print(sess.run(state))

# output:

# 0
# 1
# 2
# 3
```

思考怎么理解这段代码：

```
y=tf.Variable(21)
x=y*2
init_op=tf.initialize_all_variables()
with tf.Session() as sess:
   sess.run(init_op)
   r=sess.run(x)
print(r)
```

这段代码的含义是：

创建了 op 操作: x, y 和 init_op （它们不是 python 传统概念中的变量）

r=sess.run(x) 表示运行操作 x 的结果放到了 r 对象中

此时的 r 才是 python 传统概念中的变量。TensorFlow 中的 Variable 是指 tensor 对象。所以本文中，若涉及 ‘变量’ 是指 python 传统意义中的变量，若谈到 Variable，是指 TensorFlow 中的 tensor 对象。

代码中的赋值操作是表达式图的一部分（expression graph），就像 add()操作一样，它实际上直到 run()被执行后才完成分配。

用户实际上描述一个统计模型的参数为一个 Variables 的集合。例如，应该把神经网络的权重以 Variable 的形式存储成 tensor。在训练期间，可以通过重复的运行（run）训练图来更新这个 tensor。

Fetches

要获得运行结果，在一个 session 对象上用 run()来执行图，来获得一个或多个 tensor（保存了运行结果）。前面的例子是获得了一个 tensor 的运行结果，下面的例子是获得了两个 tensor 的运行结果（输出有两个结果）。

```
input1 = tf.constant([3.0])
input2 = tf.constant([2.0])
input3 = tf.constant([5.0])
intermed = tf.add(input2, input3)
mul = tf.multiply(input1, intermed)

with tf.Session() as sess:
  result = sess.run([mul, intermed])
  print(result)

# output:
# [array([ 21.], dtype=float32), array([ 7.], dtype=float32)]
```

sess.run()中其实只给出 mul 操作，intermed 操作也会被计算，但 intermed 的计算结果不会在 sess.run()的返回结果中。上面把 intermed 也列为了 sess.run()要运行的操作，因此，intermed 运算结果也输出了。输出的是一个 list 对象。而 list 对象中的每个元素是一个 array 对象。试一试 print(result[1][0])

All the ops needed to produce the values of the requested tensors are run once (not once per requested tensor).（这句话还没理解到）

Placeholder 和 Feeds

上面的例子通过存储 tensor 在 Constant 和 Variables 引入 tensor 到计算图（Computation Graph）. TensorFlow 也提供了一个 feed 机制将一个 tensor 直接放到图中的任何运算。

一个 feed 用一个 tensor 的值临时替换一个 operation 的输出。用户提供 feed 数据作为某个函数调用的参数。通常情况下，要设计一个特殊的 operation 来实施 feed，需要通过使用 tf.placeholder()来创建他们。

这句话这样理解：前面的代码中定义一个 variable，并赋值一个常数
input1 = tf.constant([3.0])
在 tensorflow 中这是一个 operation。该操作在创建计算图时该值就固定了。现在可以用一个临时的对象替换它(这里是 tf.placeholder)，在实际运算时再赋值，即 feed.

一个 placeholder 操作会产生错误，如果没有为它提供 feed。

```
input1 = tf.placeholder(tf.float32)
input2 = tf.placeholder(tf.float32)
output = tf.mul(input1, input2)

with tf.Session() as sess:
  print(sess.run([output], feed_dict={input1:[7.], input2:[2.]}))

# output:
# [array([ 14.], dtype=float32)]
```

Placeholder 是一个简单的 Variable，将在稍后的时候分配数据。它允许在没有数据的情况下先创建“操作”，建立“计算图”。然后可以通过这些 placeholder 向计算图 feed 数据。此时的

```
sess.run([output], feed_dict={input1:[7.], input2:[2.]})
```

需要两个参数，第一个是 Operation，第二个是 feed 数据。第二个参数的参数名是 feed_dict，它需要的值是 dictionary 对象。该 dictionary 对象中的“键”是 placeholder 名，对应的值就是要 feed 给计算图的值。

创建 placeholder 时，需要给出它的 shape。例如，

```
x = tf.placeholder("float", 3)
y = x * 2

with tf.Session() as session:
    result = session.run(y, feed_dict={x: [1, 2, 3]})
    print(result)


x=tf.placeholder(tf.int32, [None,3])
y=x*2

with tf.Session() as session:
```

```
    result=session.run(y,feed_dict={x:[[1,2,3]]})
    print(result)
```

第一段程序的 tensor 的 shape 是一维的。

在第二段程序，None 表示该 tensor 的 shape 是二维的，但第一个维度不定，根据实际输入的数据来定。

在 feed_dict 两段程序的 feed 数据是不一样的。

第二节：Variable 的创建、初始化、存储与装载

更详细的内容请参考 TensorFlow 的 API。

创建

当用户创建一个 Variable 时，需要初始化该 variable。可以传递一个 tensor 到构造方法 Variable()。例如：

W = tf.Variable(tf.zeros([784, 10]))

b = tf.Variable(tf.zeros([10]))

tf.zeros([784, 10])就是在产生一个 784*10 的 tensor，值均为 0。

TensorFlow 提供一个产生 tensor 的函数集合，可以产生并初始化 tensor.

Constant value tensor	**描述**
tf.zeros(shape, dtype=tf.float32, name=None)	产生一个所有元素为 0 的 tensor
tf.zeros_like(tensor, dtype=None, name=None)	给定一个 tensor 作为参数，返回一个和该 tensor 有相同结果，但值为 0 的 tensor
tf.ones(shape, dtype=tf.float32, name=None)	创建一个 tensor 所有的元素为 1
tf.ones_like(tensor, dtype=None, name=None)	给定一个 tensor 作为参数，返回一个和该 tensor 有相同 shape，但值为 1 的 tensor
tf.fill(dims, value, name=None)	创建一个 tensor 它的元素用一个标量值填充
tf.constant(value, dtype=None, shape=None, name='Const')	创建一个 tensor，它的值和类型和维度，由参数设定
Sequence	
tf.linspace(start, stop, num, name=None)	从 start 的值开始，到 stop 结束，产生 num 个值
tf.range(start, limit=None, delta=1, name='range')	从 start 开始，以 delta 为步长，到 limit 结束（不包括）创建一个整数序列。如果

	只是 tf.range(n)则创建从 0 开始，步长为 1，到 n(不包括)的序列
Random Tensors	创建不同分布的随机数
tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)	产生符合正态分布的随机数
tf.random_uniform(shape, minval=0, maxval=None, dtype=tf.float32, seed=None, name=None)	产生均匀分布的随机数

上表未列举所有的函数，详见
https://www.tensorflow.org/versions/r0.10/api_docs/python/constant_op.html

例 1：创建一个 tensor, shape 是[1, 3], 并显示第一个元素值

```
w= tf.zeros([1, 3], tf.int32)
sess=tf.Session()
rw=sess.run(w)
print(rw[0,0])
```

例 2：创建一个序列，该序列从 1 到 4 产生 5 个数，因为如此产生的是实数，因此给的产生是 1.0 和 4.0，而不是 1 和 4

```
sess = tf.Session()
w=tf.linspace(1.0,4.0,5)
rw=sess.run(w)
print(rw)
```

例 3：产生随机数

```
a = tf.random_uniform([1])
with tf.Session() as sess1:
  print(sess1.run(a))
```

创建 variables 时，作为参数的 tensor 的 shape 就是 Variable 的 shape。Variable 的 shape 通常是固定的，但 TensorFlow 也提供了函数进行修改，这里不讨论。

创建了 Variable 的操作，需要再创建初始化操作来初始化 Variable。看下面代码：

```
# Create two variables.
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
                      name="weights")
init_op=tf.initialize_all_variables()
with tf.Session() as sess:
    sess.run(init_op)

print(sess.run(weights))
```

创建了一个定义 Variable 的操作，同时还需要定义一个初始化 operation。然后用 session 的 run（ ）方法来运行这些操作。

由 tf.Variable()返回的值实际上是 tf.Variable 类的实例，即 tensor 对象

一个 Variable 可以放到一个指定的装置（CPU 或 GPU）上，此处我们不做讨论。

初始化

创建 Variable 时，也可以从其他 Variable 来产生初始值，即通过其他 Variable 的 initialized_value()方法

```
# Create a variable with a random value.
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
                      name="weights")
# Create another variable with the same value as 'weights'.
w2 = tf.Variable(weights.initialized_value(), name="w2")
# Create another variable with twice the value of 'weights'
w_twice = tf.Variable(weights.initialized_value() * 2.0,
name="w_twice")
init_op=tf.initialize_all_variables()
```

给变量赋值

在 tensorFlow 中，一个变量如果赋值是通过该变量的 assign 方法。例如

```
input1 = tf.Variable(tf.constant([3.0]))
input2 = tf.constant([2.0])
input3 = tf.constant([5.0])
intermed = tf.add(input2, input3)
mul = tf.mul(input1, intermed)
init_op=tf.initialize_all_variables()
op=input1.assign([1])
with tf.Session() as sess:
   sess.run(init_op)
   sess.run(op)
   result = sess.run([mul, intermed])
   print(result)
```

需要记住的是，op=input1.assign([1])也是一个操作。需要用 session 的 run 方法来执行，才能完成赋值操作。

Tips:
当建立机器学习模型时，通常把模型的参数建立成 Variable，它是模型要训练的部分。有时我们也需要全局的 variable。例如，对学习的步数进行计数的 Variable。它不能作为模型的参数。这时在创建 Variable 时，加上一个参数 trainable=<bool>。如果参数为 True，则新的 variable 被加到 graph collection GraphKeys.TRAINABLE_VARIABLES. 如果为 False，则不会把 Varialble 加入。

前面讲的都是作为模型参数的 Variable，下面是一个全局变量的 Variable
global_step = tf.Variable(0, name="global_step", trainable=False)

存储与装载

最简单的存储一个模型的方法是使用 tf.train.Saver 对象

关于 tf.Variable_scope(), tf.name_scope(), tf.Variable(), tf.get_variable()

命名空间是 tensorflow 按照体系结构组织变量名和操作名的方式。

tf.name_scope 在默认图上为“操作”创建命名空间

tf.variable_scope 在默认图上为"变量"和"操作"创建命名空间

tf.get_variable 创建一个新的变量，或者获得一个已经创建的变量

tf.Variable 创建一个新的变量。tf.name_scope 是在使用 tf.Variable()创建变量时，给变量分配命名空间，例如：

```
with tf.name_scope('conv1'):
    weights1 = tf.Variable([1.0, 2.0], name='weights')
print (weights1.name)
```

运行结果是：

```
conv1/weights:0
```

代码 `weights1 = tf.Variable([1.0, 2.0], name='weights')`中，weights1 是 tensorflow 数据流图上的一个操作 op，该操作定义了一个变量，名字是 weights，该变量的 shape=[1.0, 2.0]

上面的代码含义是在命名空间 conv1 下创建变量 weights。

tf.get_variable 和 tf.variable_scope 一起使用，可以创建新变量或获取已经创建的变量（变量共享）。

```
with tf.variable_scope('conv2') as scope:
    weights2 = tf.get_variable("weights", shape=[4.0, 2.0])
print (weights2.name)

with tf.variable_scope('conv2', reuse=True) as scope:
    w3=tf.get_variable("weights", shape=[4.0, 2.0])
print(w3.name)
```

运行结果是：

```
conv2/weights:0
conv2/weights:0
```

weight2 和 w3 是两个相同的变量

第三节：初识 TensorFlow：手写数字识别

MNIST 是一个手写数字的数据集。它包含的图片如下图：

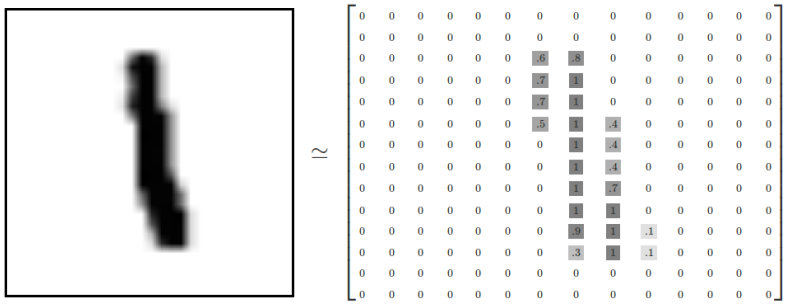
每个图片也被分配了一个标签，即图片对应的数字。在本节我们将建立一个预测模型，给定一张图片预测它对应的数字。本节不讨论如何训练一个性能最优的分类器（像是第一节所介绍的），只是探讨如何用 TensorFlow 完成该工作。这里将构建一个简单的 Softmax regression 模型。

MNIST 数据集可以在 Yann LeCun's website 的网站下载 http://yann.lecun.com/exdb/mnist/。但 TensorFlow 已经预装载了该数据集，使用下面的代码即可获得该数据集

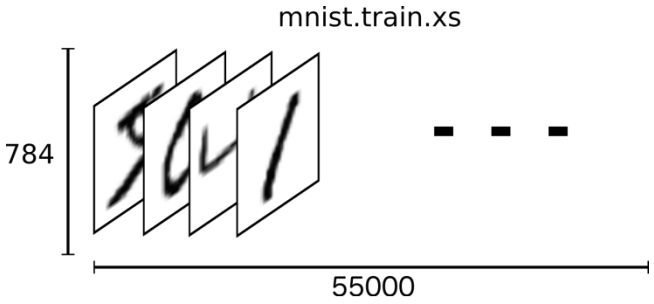
```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

获得的该数据集包含三个部分：55000 条训练数据（mnist.train），10000 条测试数据（mnist.test）和 5000 条校验数据集（mnist.validation）。一条 mnist 数据包含两个部分：手写数字图片和对应的标签。这里称图片 xs，标签 ys。

mnist.train.images 就是 xs；mnist.train.labels 就是 ys。每张图片是一个 28*28 像素的矩阵。它们被转换成了 28*28=784 长度的向量



因此，mnist.train.images 是一个 shape=(55000, 784)的 tensor。第一个维度是图片的编号，第二个维度是每个图片的像素。像素值在 0-1 之间取值。

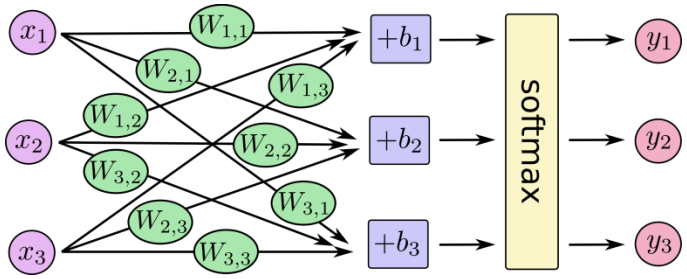


MNIST 的标签 label 数据是一个向量，描述 0-9 中的一个值，称为 one-hot 向量。one-hot 向量中只有一个元素的值是 1，其他都是 0。 ‘1’ 对应该图片对应的数字。例如，标签 3 的 one-hot 向量[0,0,0,1,0,0,0,0,0,0]。相应的 mnist.train.labels 是一个[55000, 10]的矩阵。

Softmax Regression

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 的网络描述如下图：（此处假设，输入向量的维度是 3，输出的类别个数也是 3）



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 $z=\{z_1,z_2,\ldots,z_n\}$

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

上图的数学描述就是

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

W 是权重矩阵，b 是偏置向量，x 是输入向量。

实施 softmax 模型

要使用 TensorFlow 首先 import 该包

```
import tensorflow as tf
```

因为在运行计算图时，需要将训练数据 feed 给模型，因此创建 placeholder

```
x = tf.placeholder(tf.float32, [None, 784])
```

因为，图片数据已经被转换成了向量，shape 中的列数是 784。此处 None 表示不限定输入数据的个数。

该模型中需要权重和偏置（bias）。因此，创建 Variable。Variable 可以理解为值可更改的 tensor。图计算的过程中使用，并会更改它。在使用 TensorFloss 进行机器学习时，模型的参数必须设置成 Variable。

```
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
```

tf.zeros([784, 10])是创建一个 tensor 它的 shape 是[784， 10]，它作为 Variable W 的初始值。这里的权重的 Shape 是[784,10]，因为有 784 个输入，10 个输出。偏置 Variable b 的 shape 是 10，因为给 10 个输出，每个加上一个偏置。

现在可以实施模型：

```
y = tf.nn.softmax(tf.matmul(x, W) + b)
```

tf.matmul 是矩阵乘。加上偏置后，进行 softmax 的操作。得到的 y 就是每个输入对应到每个输出的概率矩阵。

训练模型

要训练模型，首先需要定义什么样的模型是好的或者差的，即定义代价或损失函数。交叉熵是一种损失函数，其定义为

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

y 是预测的概率分布，y'是真实的概率分布。简单的说，交叉熵定义了模型是多么的无效率。实施交叉熵，需要定义一个 placeholder，label 数据将在运行阶段放在该 placeholder。

```
y_ = tf.placeholder(tf.float32, [None, 10])
```

然后实施交叉熵

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),
reduction_indices=[1]))
```

y_*tf.log(y)进行两个矩阵的点乘（注意，两个矩阵相乘用的是 mul 函数）。reduction_indices=[1]规定了 tf.reduce_sum 在矩阵点乘的结果的列的方向进行求和。计算的结果是一个 shape=[n, 1]的 tensor。即每个输入被计算了一个交叉熵。Reduce_mean 对这 n 个交叉熵求平均值，得到模型的损失值。

定义了损失函数，就可以继续定义优化操作。TensorFlow 可以根据定义的计算图，进行优化操作训练模型，从而得到估计的模型参数。因此，训练操作就是一个优化操作，我们将优化器的定义和优化操作写在一起来定义训练操作如下：

```
train_step =
tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

此处我们采用的是梯度下降的方法训练模型，是学习率是 0.5。TensorFlow 也提供了其他的优化器。

在上面定义操作的背后，实际上是把“操作”添加到一个默认的计算图，它实施反向传播算法（默认实施）和梯度下降优化算法来训练模型。（第二章介绍的 BP 算法没介绍数学推导过程。实际上神经网络将输入先正向传播；神经网络在每个层计算梯度，但此时需要一个敏感度，敏感度的计算是从输出反向传播过来的；每一层有计算的梯度更新权重）

在开始正式训练模型之前，需要一个初始化操作。

```
init = tf.initialize_all_variables()
```

并在运行训练操作之前，运行初始化操作。

```
sess = tf.Session()
sess.run(init)
```

训练模型的过程如下

```
for i in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

在每一次训练的迭代中，仅从训练数据集抽去 100 条训练数据，称为一个 batch。将一个 batch 的训练数据中的数据和 label 分别 feed 到训练操作中。它们实际上是 feed 给了训练操作中用到的 placeholder。

使用部分训练集的数据训练模型称为，随机训练（stochastic training）,此例中即是第二章中我们提到的随机梯度下降训练。理想情况下，可以一次使用所有的训练数据在每一次迭代中训练模型，但这种方法代价太大。采用随机训练，方法简便，需要的代价小，但训练效果不变。

评估模型

tf.argmax 给出一个 tensor 中，沿着某个维度最高值的索引下标。例如，tf.argmax(y，1)是我们的模型认为输入最可能属于某个类的标签。tf.argmax(y_, 1)是正确的标签。使用 tf.equal 来检测两个值是否相等

```
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
```

这里又是再定义 operation。其中的 y 是在前面已经定义了的操作。correct_prediction 存储了每条输入数据是否被正确预测的判断，其值是[True，False, True]的数据。

进一步我们可以计算精确率

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

tf.cast 将 True 转换成 1，False 转换成 0。

最后，输出预测结果。这里又是运行定义的操作。

```
res = sess.run(accuracy, feed_dict={x: mnist.test.images, y_:
mnist.test.labels})
print(res)
```

此时运行 accuracy 操作时，accuracy 操作运行了 correct_prediction 操作；而 correct_prediction 操作又运行了 y 操作；y 操作

y = tf.nn.softmax(tf.matmul(x, W) + b)

y 操作又运行了矩阵乘，此时的 W 保存的是已经训练出的模型参数。

第四节：使用 TensorFlow 构建神经网络的步骤

1. 定义训练数据和标签数据

2. 为训练数据和标签数据定义 placeholder

3. 考虑好有几个隐层，然后定义相应的权重 Variable

4. 定义从输入，到隐层，最后到输出的数据矩阵乘操作。

5.定义损失计算操作，该操作就是将标签数据与输出层的结果进行损失计算

6. 定义优化器

7. 定义训练操作，就是用优化器优化损失操作

8. 创建 Session 的对象，用该对象的 run 方法运行训练操作。并将训练数据和标签 feed 给训练操作

第五节：TensorFlow 练习（1）：实现感知机

这一节我们通过用 TensorFlow 来实现 2.2 节的感知机例子，来认识 TensorFlow。

用 TensorFlow 开发机器学习算法，关键要定义几个关键 ops:损失函数，优化器和训练操作。损失函数，实际上是定义了从输入得到最终输出结果的操作，然后定义预测结果和实际结果的损失函数，如可以用误差平方和。设 iPh 是输入，weight 是边权重矩阵，y 是模型计算的输出，lPh 是训练集标签

```
y=tf.matmul(iPh, weight)
loss=tf.reduce_mean(tf.square(y-lPh))
```

优化器可以选择 TensorFlow 提供的优化器

```
optimizer = tf.train.GradientDescentOptimizer(eta)
```

训练操作就是将损失函数带入优化器

```
train_op = optimizer.minimize(loss)
```

训练的过程就是不断的迭代执行 train 操作。迭代过程中，参与运算的 Variable 被当做是模型的参数，如程序中的 weights，会在迭代过程中安装相应优化算法的更新公式更新。此例中，就是 weight<-weight + 梯度

```
import tensorflow as tf
import numpy as np

train=(((2.0,5.0,3.0),850.0),((1.0,4.0,7.0),1050.0),((2.0,3.0,5.0),950.0),
      ((3.0,6.0,9.0),1650.0),((7.0,4.0,1.0),1350.0))
labels=[]
dat=[]
eta=0.005  # learning rate
w=[50.0,50.0,50.0]

for x, l in train:
    labels.append(l)
    dat.append(x)

labels=np.reshape(labels,[-1,1])
size=len(dat)
dim=len(dat[0])

# training
with tf.Graph().as_default():
    lPh=tf.placeholder(tf.float32,[None,1])
    iPh=tf.placeholder(tf.float32,[None,dim])

    weight=tf.Variable(tf.constant(w,shape=[dim,1]),dtype=tf.float32)

    y=tf.nn.relu(tf.matmul(iPh, weight))
    loss=tf.reduce_mean(tf.square(y-lPh))

    optimizer = tf.train.GradientDescentOptimizer(eta)
    train_op = optimizer.minimize(loss)

    init_op=tf.initialize_all_variables()
    sess=tf.Session()
    sess.run(init_op)

    for _ in range(1,1000):
        feed_dict={iPh:dat,lPh:labels}
        _,val=sess.run([train_op,loss],feed_dict=feed_dict)
        print(val,sess.run(weight))

    d=sess.run(weight)
    print(d)
```

使用 TensorFlow 的 Tips：

(1) 用 TensorFlow 开发机器学习程序时，通常会有多次迭代，迭代过程中参与运算的 Variable 的值被当做模型的参数，它的值会改变。

(2) 在调用 session 的 run 方法进行训练模型前，定义一个 Variable 初始化操作，然后用 session 的 run 方法执行该操作

(3) 要想查看一个 Variable 的内容，调用 session 的 run 方法运行它，其返回结果就是内容

对“默认图”的理解：

我们创建 TensorFlow 程序时，默认的会有个 Graph，只要我们创建操作 op，都是在向这个图添加操作（TensorFlow 中把创建 Variable，Constant 也理解成是 operation）。上面的 with tf.Graph().as_default():语句是创建一个图，作为默认图（我们不要这条语句程序也可以允许）。如果需要创建多个图时，需要使用该方法。

```
# 1. 创建一个图，并作为默认图
g = tf.Graph()
with g.as_default():
  c = tf.constant(5.0)
  assert c.graph is g

# 2. 再构建一个图作为默认图
with tf.Graph().as_default() as g:
  c = tf.constant(5.0)
  assert c.graph is g
```

我们定义的操作，都会添加到当前的默认图上的。

第六节：TensorFlow 练习（2）：曲线拟合

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

程序如下：

```
import numpy
import tensorflow as tf
import math

# generate data
x=numpy.arange(0,6.3,0.1)
y=[math.sin(val) for val in x]
s = numpy.random.normal(0, 0.1, len(x))
z=[sum(x2) for x2 in zip(y,s)]
x=numpy.array([x])
z=numpy.array([z])
size=len(z)    # the number of instances
idim=1         # the dim of input
wsize=7        # the number of nodes in hidden layer
```

```
eta=0.06
with tf.Graph().as_default():
    iph=tf.placeholder(tf.float32, [idim, None])
    lph=tf.placeholder(tf.float32, [idim, None])
    w1=tf.Variable(tf.random_uniform((wsize,idim), 0, 1, dtype=tf.float32))
    w2=tf.Variable(tf.random_uniform((idim, wsize), 0, 1, dtype=tf.float32))
    b1 = tf.Variable(tf.zeros([wsize,1]))
    b2 = tf.Variable(tf.zeros([1]))

    hidden1=tf.nn.sigmoid(tf.matmul(w1, iph)+b1)
    modle  =tf.matmul(w2, hidden1)+b2

    loss=tf.reduce_mean(tf.square(modle-lph))
    step = tf.Variable(0, trainable=False)
    rate = tf.train.exponential_decay(0.15, step, 1, 0.9999)
    optimizer = tf.train.AdamOptimizer(rate)

    train_op = optimizer.minimize(loss)

    init_op=tf.initialize_all_variables()
    sess=tf.Session()
    sess.run(init_op)

    for step in range(1,80000):
        feed_dict={iph:x,lph:z}
        _,val=sess.run([train_op,loss],feed_dict=feed_dict)
        if step % 1000 == 0:
            print(val)
```

在上面的程序中，我们将训练数据和标签数据转换成了 numpy 的 array 数据结构，如此就可以将数据直接 feed 给 placeholder。用普通 Python 的 list 数据结构则不能。如 3.3 节的程序所示。

TensorFlow 中可以设置动态调整学习率。如，此例子中

step = tf.Variable(0, trainable=False)

rate = tf.train.exponential_decay(0.15, step, 1, 0.9999)

动态调整学习率可以使得收敛速度更快

不同的优化算法对学习性能有很大影响，例如，试一试下面两种优化算法

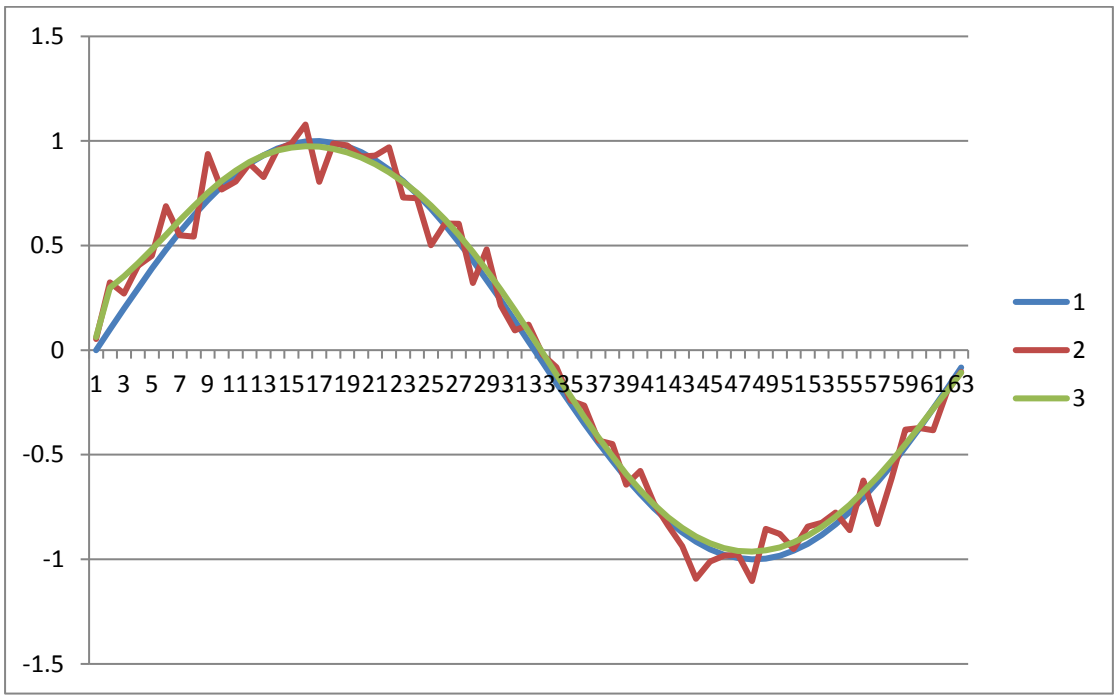
optimizer = tf.train.GradientDescentOptimizer(rate)

optimizer = tf.train.AdamOptimizer(rate)

使用 TensorFlow 的 Tips：

(4) 我们在此程序中没有显性的使用反向传播算法来训练神经网络，但实际上 TensorFlow 会自动使用反向传播算法。

数据如下图所示。其中曲线 1 是标准正弦曲线；曲线 2 是加上了噪声的曲线；曲线 3 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。



第七节：tf.contrib.learn

tf.contrib.learn 是一个高层的 TensorFlow API。它使得构建、训练、评估各种机器学习模型非常方便。

详见：https://www.tensorflow.org/versions/r0.10/tutorials/tflearn/index.html

第八节：一些操作

1. tf.name_scope()

2. tf.variable_scope()

第四章：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。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。

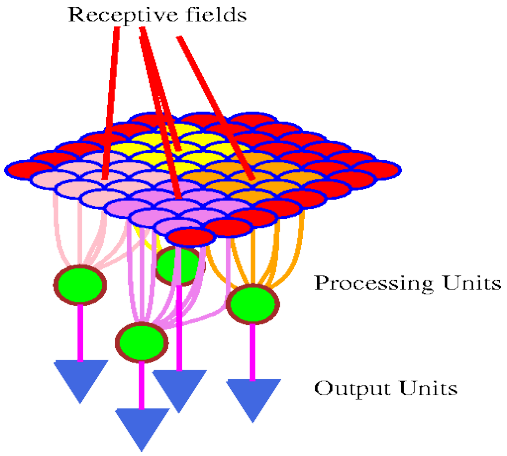


图 4.1：视觉系统上的感受野

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：

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

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

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

其余 y(n)=0

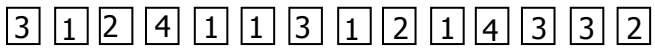
我们也可以按照滤波器的方式来理解离散卷积。给定一个输入信号序列 x_t, t=1,...,n, 和滤波器 f_t, t=1,..., m, 一般情况下滤波器的长度 m 远小于信号序列长度 n。

卷积的输出为：

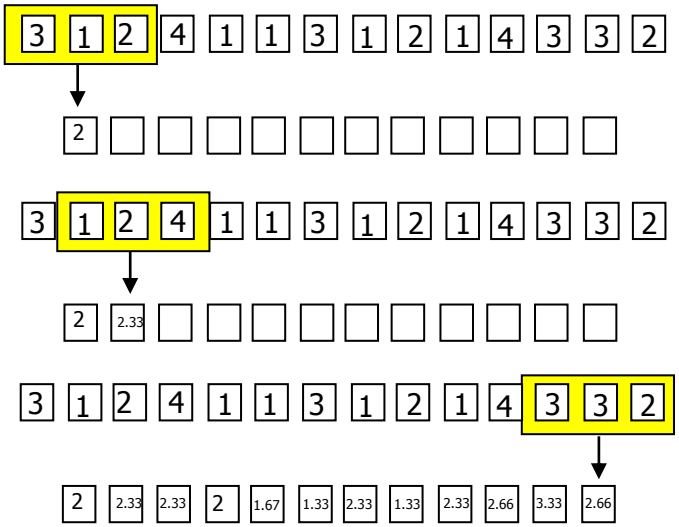
$$y_t = \sum_{k=1}^{m} f_k \cdot x_{t-k-1}$$

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，宽带为 m 的子序列求平均。

例 2：有一个 x 序列



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。

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：

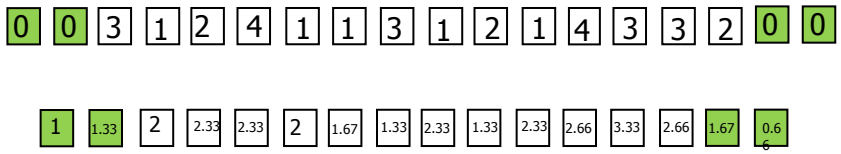


图 4.2 是一个一维窄卷积的例子。而在图像处理中经常用二维卷积。给定一个图像 $x_{ij}, 1 \le i \le M, 1 \le j \le N$，和滤波器$f_{ij}, 1 \le i \le m, 1 \le j \le n$，一般 m≪M, n≪N。卷积的输出为：

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

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$f_{uv} = \frac{1}{mn}$

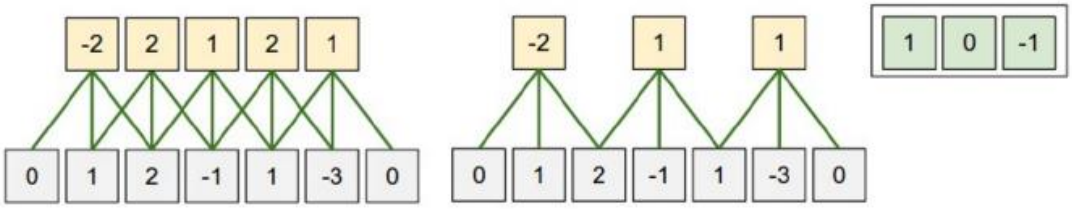


图 4.2：一维窄卷积

第二节：卷积神经网络的结构

4.2.1 CNN 的特征

卷积神经网络具有下面的特性:

1. 局部连接

CNNs 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 4.3 所示。

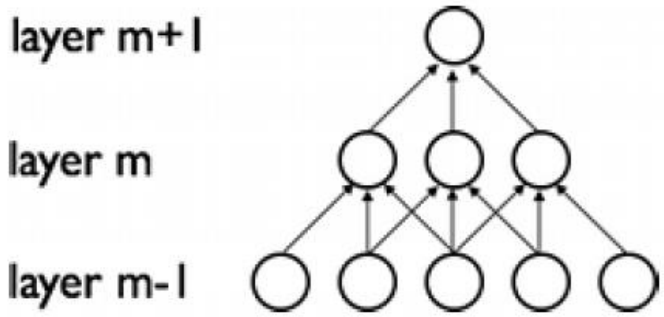


图 4.3：一个 CNN 示例

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

把滤波器的概念引入到 CNN 中会有些抽象。我把 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 4.3 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* 为连接到隐层 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 4.4 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 feature map 特征映射。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。

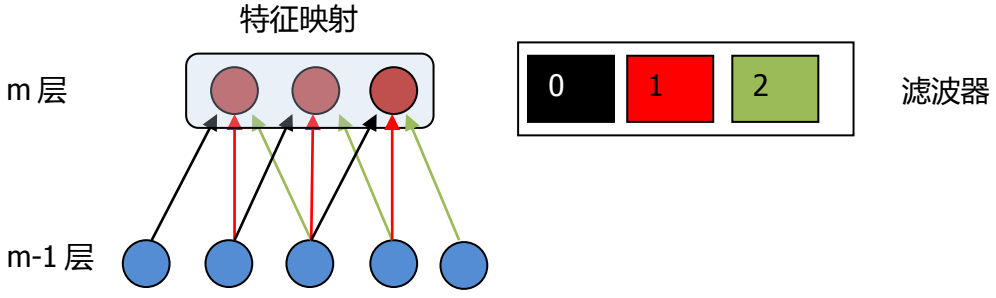


图 4.4：特征映射

图 4.4 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 的三个单元属于同一个 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。

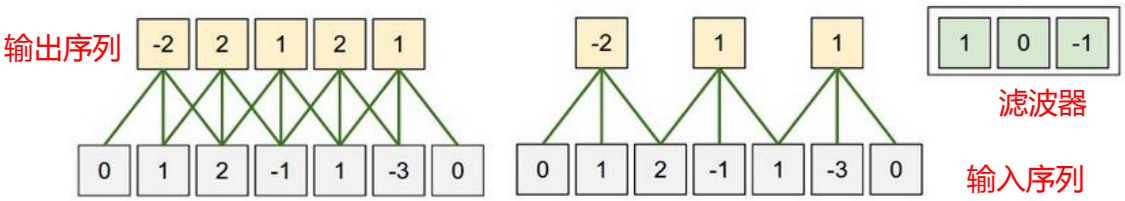


图 4.5 一维卷积

将输入序列理解为神经网络的$l-1$层；滤波器实际上是边的权重。输出序列是神经网络的l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 4.5 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。

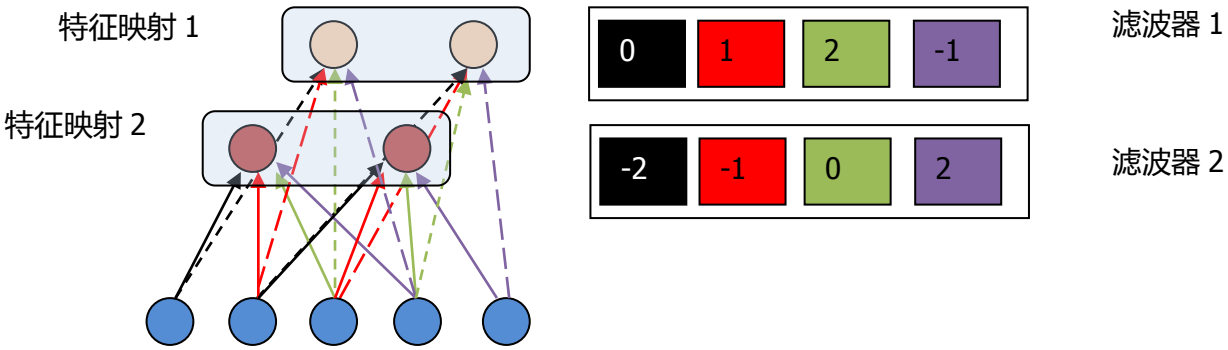


图 4.6 多个滤波器

4.2.2 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。

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，$a_{(i+m-1):i}^{(l)} = [a_{i+m-1}^{(l)}, \dots, a_i^{(l)}]^T$

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$n^{(l+1)} = n^{(l)} - m + 1$。

Tips: 一个滤波器有一个偏置（标量）

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，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。$X^{(l)}$的每一个元素为：

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

注意：这里 W 和 w 含义不同。W 是滤波器，w 是以矩阵描述一层的神经元的个数时的宽度。w_l是第 l层的神经元组的宽度。$W^{(l)}$是二维滤波器。第$l-1$层的神经元个数为$w_l \times h_l$，并且$w_l = w_{l-1} - u + 1$，$h_l = h_{l-1} - v + 1$。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。每一组输出都共享一个滤波器。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$n_l$。第$l$层的第 k 组特征映射$X^{(l,k)}$为

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

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。

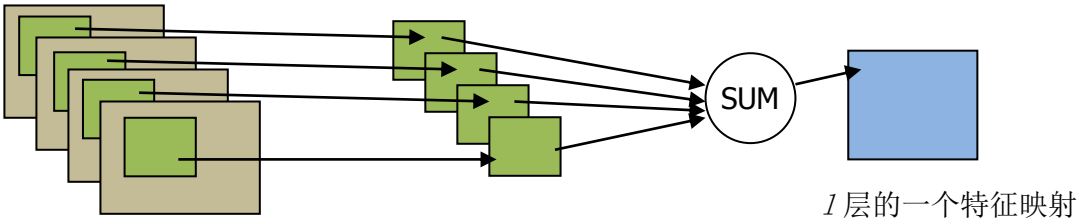


图 4.7 卷积操作示例

下面我们用图来演示一下二维卷积操作：

设输入为 5*5 的矩阵（下图假设输入只有一个特征映射），滤波器是 3*3，不填充，步长（stride）为 1。

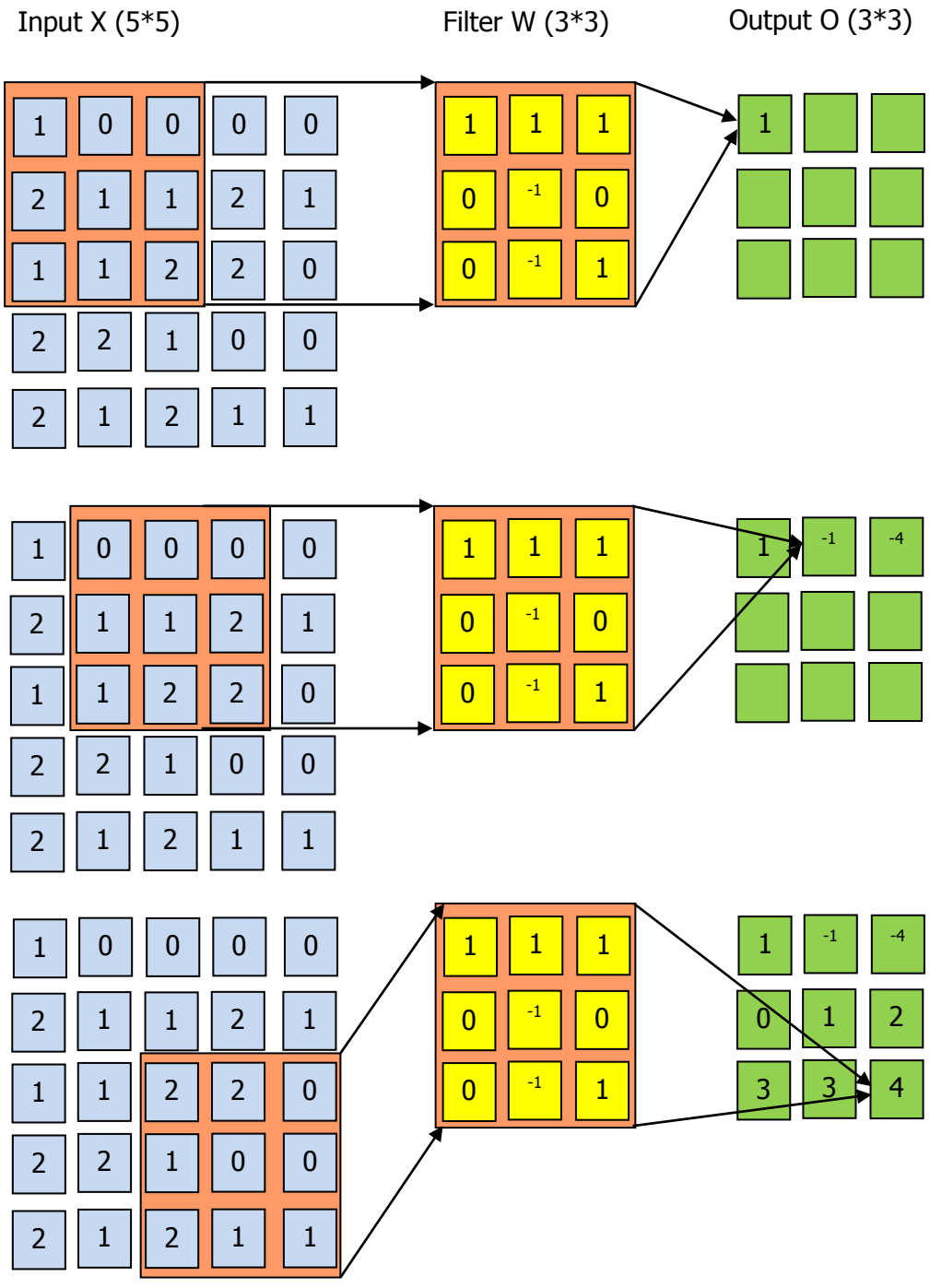


图 4.8 卷积操作示例 2

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

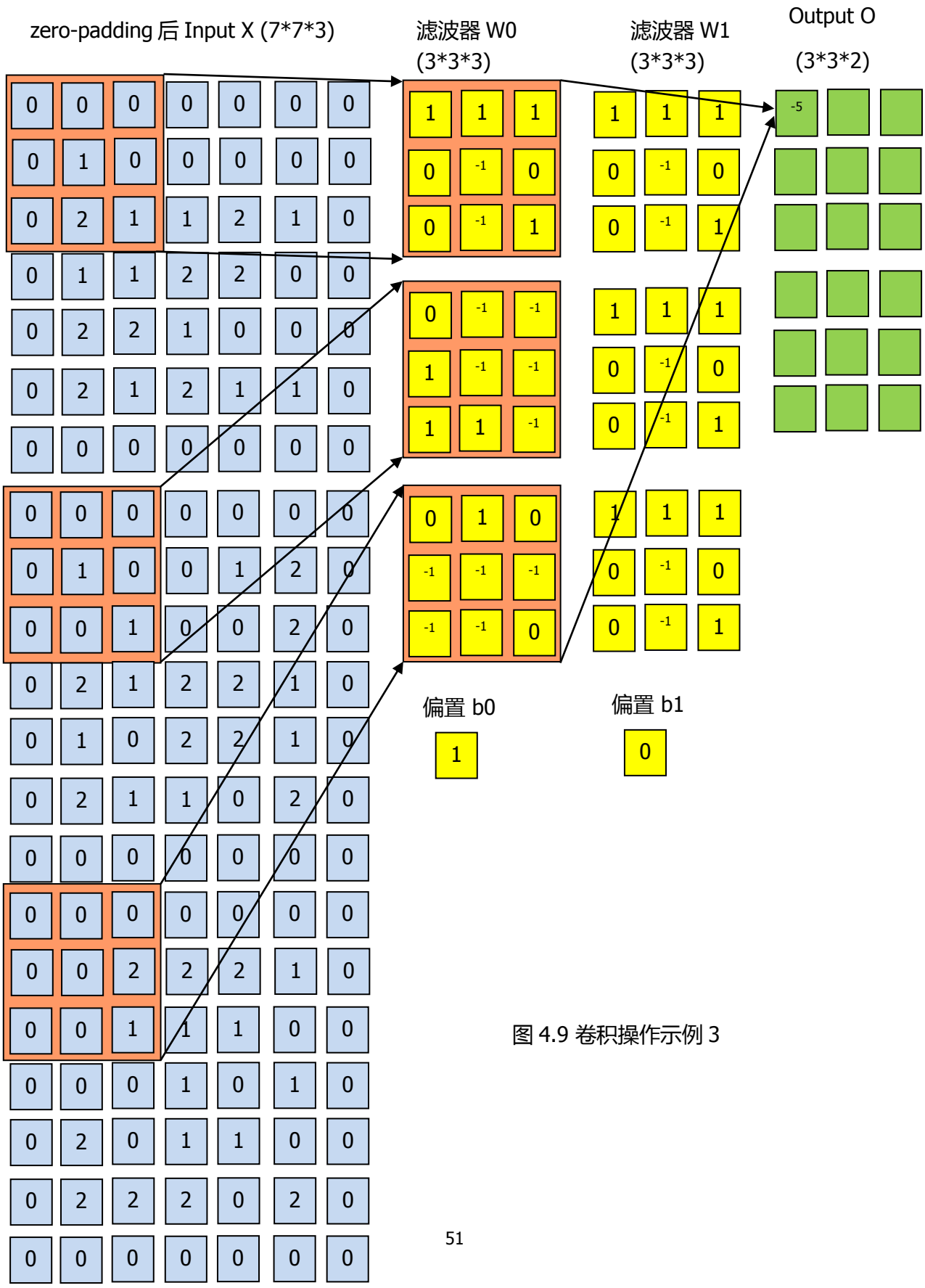


图 4.9 卷积操作示例 3

输出 output 为 3*3*2，其中的 2 是 output_channles，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）

我们可以用下图来描述二维卷积层的从输入到输出的映射关系

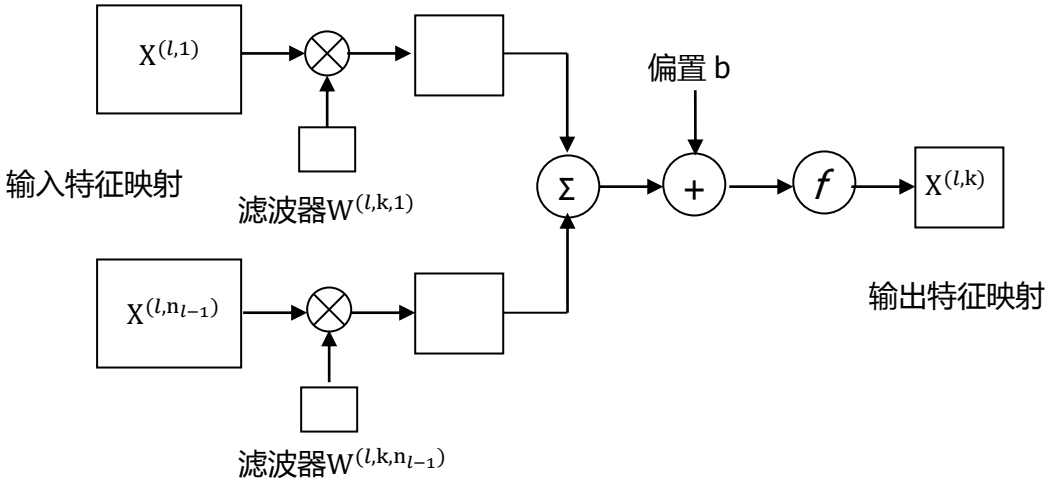


图 4.10 二维卷积层的从输入到输出的映射关系

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_channel 等于输入特征映射的个数。之所以此处使用 in_channel 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的中的每个 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。

4.2.3 子采样层（或池化层）

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。

对于卷积层得到的一个特征映射$X^{(l)}$，我们可以将$X^{(l)}$划分为很多区域 R_k, k=1,..,K，这些区域可以重叠，也可以不重叠。

$$X_k^{(l+1)} = f\left(Z_k^{(l+1)}\right) = f(w^{(l+1)}.\mathbf{down}(R_k) + b^{(l+1)})$$

$$X^{(l+1)} = f\left(Z^{(l+1)}\right) = f(w^{(l+1)}.\mathbf{down}\left(X^l\right) + b^{(l+1)})$$

其中，$w^{(l+1)}$和$b^{(l+1)}$分别是可训练的权重和偏置参数。$\mathbf{down}(X^l)$是指采样后的特征映射。子采样函数$\mathrm{down}(\cdot)$一般是取区域内所有神经元的最大值（Maximum Pooling）或平均值（Average Pooling）

$$\mathrm{pool}_{\max}(R_k) = \max_{i \in R_k} a_i$$

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

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。

> *注：各种文献对 CNN 的描述中使用的术语不一致。如，对卷积层的输入，有的描绘成是一组特征映射，一个特征映射是一个二维的图；有的描绘成是输入立方体 (Volume)，该立方体的 depth 维上的一个切片（slice）对应一个特征映射。在 TensorFlow 中又都称作 tensor，如输入 tensor。再比如，池化 pooling 和子采样 subsampling 两个术语含义相同。*

子采样层（或池化层）在输入的每个 depth 维度上的切片进行操作。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 切片进行步长 stride 为 2，采样 MAX 操作进行下采样。这样有 75%的神经元被放弃，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

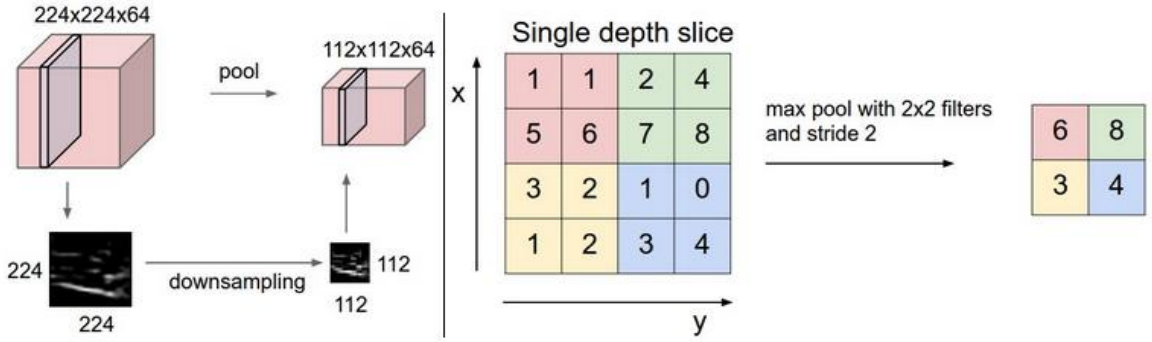
(3) Pooling 层输出立方体的 size: W_2*H_2*D_2

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

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



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。

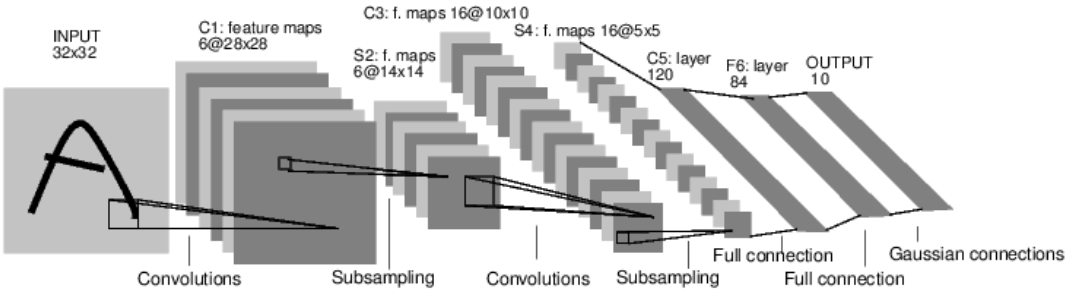
也有人多 pooling 层有不同意见，他们认为完全可以抛弃 pooling 层。他们建议在卷积层使用更大的步长，就可以有效的减小神经元的数目。

4.2.4 全连接层

卷积神经网络中通常会包含一个全连接层。全连接层的神经元与前一层的所有神经元都有连接，与传统的神经网络一样。

第三节：卷积神经网络示例

LeNet 是第一个成功应用的卷积神经网络，由 Yann LeCun 在上世纪九十年代开发。LeNet-5 在美国银行系统中已经非常才成功的应用在了支票上手写数字的识别。LeNet-5 的网络结构如下图所示。



不算输入层，LeNet-5 共有 7 层，每一层结构为：

（1） 输入层：输入图像大小为 32*32=1024.

（2） C1 层：卷积层。滤波器的大小是 5*5=25，共有 6 个滤波器。得到 6 组大小为 28*28=784 的特征映射。因此，C1 层的神经元个数为 6*768=4,704。可训练参数个数为 6*25+6=156。连接数为 156*784=122,304（包括偏置在内，下同）。

（3） S2 层：子采样层。由 C1 层每组特征映射中的 2*2 领域点按照平均值操作方法次采样（down sampling）为 1 个点。这一层的神经元个数为 14*14=196。可训练参数个数为 6*（1+1）=12。连接数为 6*196*（4+1）=122,304（包括偏置的连接）。

（4） C3 层：卷积层。由于 S2 层也有多组特征映射，需要一个连接表来定义不同层特征映射之间的依赖关系。LeNet-5 的连接表如下图所示。这样的连接机制的基本假设是：C3 层的最开始 6 个特征映射依赖于 S2 层的特征映射的每 3 个连续子集。接下来的 6 个特征映射依赖于 S2 层的特征映射的每 4 个连续子集。再接下来的 3 个特征映射依赖于 S2 层特征映射的每 4 个不连续子集。最后一个特征映射依赖于 S2 层的所有特征映射。这样共有 60 个滤波器，大小是 5*5=25。得到 16 组大小为 10*10=100 的特征映射。C3 层的神经元个数为 16*100=1600。可训练参数个数是 60*25+16=1516；连接数为 1516*100=151,600。

（5） S4 层：是一个子采样层，由 2*2 领域点 down sampling 为 1 个点，得到 16 组 5*5 大小的特征映射。可训练参数个数为 16*2=32。连接数为 16*(4+1)*5*5=2000。

（6） C5 层：卷积层。得到 120 组大小为 1*1 的特征映射。每个特征映射与 S4 层的全部特征映射连接。有 120 个滤波器，大小是 5*5=25。C5 层的神经元个数为

120，可训练参数个数为 1920*25+120=48,120。连接数为 120*(16*25+1)=48,120。

（7） F6 层：一个全连接层，有 84 个神经元，可训练参数个数为 84*(120+1)=10,160。连接数和可训练参数个数相同，为 10,164。

（8） 输出层：由 10 个径向基函数（Radial Basis Function, RBF）组成。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

每列指示 S2 层的哪个特征映射和 C3 层的特征映射的单元相连。

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)

按照这篇文章，在使用 Dropout 时训练阶段和测试阶段做了如下操作：

在模型训练阶段，在没有采用 pre-training 的网络时，hintion 并不是像通常那样对权值采用 L2 范数惩罚（正则化），而是对每个隐含节点的权值 L2 范数设置一个上限 bound，当训练过程中如果该节点不满足 bound 约束，则用该 bound 值对权值进行一个规范化操作（即同时除以该 L2 范数值），说是这样可以让权值更新初始的时候有个大的学习率供衰减，并且可以搜索更多的权值空间。

在模型的测试阶段，在网络前向传播到输出层前时隐含层节点按一个概率 prob 随机选择的节点输出；输出值都为原值*1/prob（详见下一节的 TensorFlow 的 dropout 实施）。

问：TensorFlow 中在模型的训练中，设置 prob=0.5。但是在测试阶段设置 prob=1。即 dropout 在测试阶段不起作用。我觉得这个是合理的

关于 Dropout，文章中没有给出任何数学解释，Hintion 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。

3. naive bayes 是 dropout 的一个特例。Naive bayes 有个错误的前提，即假设各个特征之间相互独立，这样在训练样本比较少的情况下，单独对每个特征进行学习，测试时将所有的特征都相乘，且在实际应 用时效果还不错。而 Droput 每次不是训练一个特征，而是一部分隐含层特征。

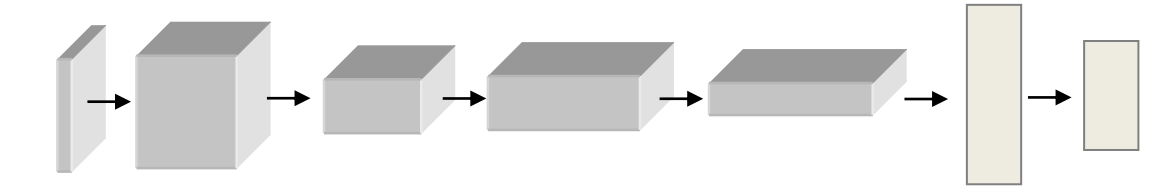
4. 还有一个比较有意思的解释是，Dropout 类似于性别在生物进化中的角色，物种为了使适应不断变化的环境，性别的出现有效的阻止了过拟合，即避免环境改变时物种可能面临的灭亡。

TensorFlow 实现了 Dropout 的操作。4.9 节的第 4 段，给出了 TensorFlow 实施 dropout 的描述

第五节：用 TensorFlow 实现 CNN

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该模型的精确度达到了 91%，实际上这是一个很差的性能。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所示：



1. 输入层

首先读入数据，然后创建 placeholder 占位符

```
mnist = input_data.read_data_sets("data/", one_hot=True)

x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])
```

此处占位符 x 保存的是长度为 784 的向量，即把输入数据 mnist 保存的数据是把 28*28 的图像转换成了向量的形式。占位符 y_中的 10 对应输出有 10 种数字，输出是 one-hot 向量。

2. 第一个卷积层和 pooling 层

创建卷积层的第一步是创建滤波器。按照前面讲述的内容，滤波器的参数就是卷积层的权重，是要学习的参数。创建两个 variable，一个是卷积层的权重，即滤波器，一个是偏置变量

```
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
```

创建的权重的 shape=[5,5,1,32]其实是卷积层的滤波器的 size：滤波器的宽和高是 5*5，因为输入数据的通道（特征映射）是 1，这里滤波器也是 1，创建 32 个滤波器。每个滤波器有一个偏置，因此偏置变量的设置是 32 个。这里自定义了两个函数

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

tf.truncated_normal 函数从一个 truncated normal distribution 产生随机数；tf.constant 产生常数。

将读入的以向量形式描述的数据转换成 28*28 的矩阵。

```
x_image = tf.reshape(x, [-1,28,28,1])
```

tf.reshape 重新安排 tensor。它的第二个参数就是重新安排的 tensor 的 shape。如果 shape 中的某个维度是-1，它表示该维度的值不定，由其他维度的值固定后计算来得到。此处表示转换成维度为 4 的 tensor 时，第一个维度即图像的数量不定，但每张图像的 shape 是固定的 28*28*1。

再建立第一个卷积层和池化层

```
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

conv2d 是自定义函数，如下

```
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
```

它调用 tf.nn.conv2d 函数来创建卷积层。该函数是给定一个 4-D 的输入 tensor 和滤波器 tensor，计算一个 2-D 卷积。

```
tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=None, data_format=None, name=None)
```

input 是输入 tensor，它的 shape 是 [batch, in_height, in_width, in_channels]。因为采用的是随机梯度下降 SGD 的优化方法（参见第二章），需要给出一个 batch 参数，即进行训练的一个批次的图像数量。in_height, in_width, in_channels 即一张图像的高、宽和通道数。

一个滤波器（或 kernel）tensor 的 shape [filter_height, filter_width, in_channels, out_channels]。此处 in_channels 即对应输入图像的通道个数；out_channels 是该卷积层输出的通道数量，它对应的是滤波器的个数，也即特征映射个数。

Strides 是卷积操作中的步长。Stride 的是一个长度为 4 的向量 strides = [1, 1, 1, 1]。其中的每个元素对应到输入 tensor 的 shape 的每个维度。如以当前的例子，strides[0]对应图片数量这个维度；strides[1]对应一张图片的高度这个维度；strides[2]对应一张图片的宽度这个维度；strides[3]对应一张图片的通道维度。Strides 的设置中 strides[0]和 strides[3]必须值为 1，strides[1]和 strides[2]必须值相等。

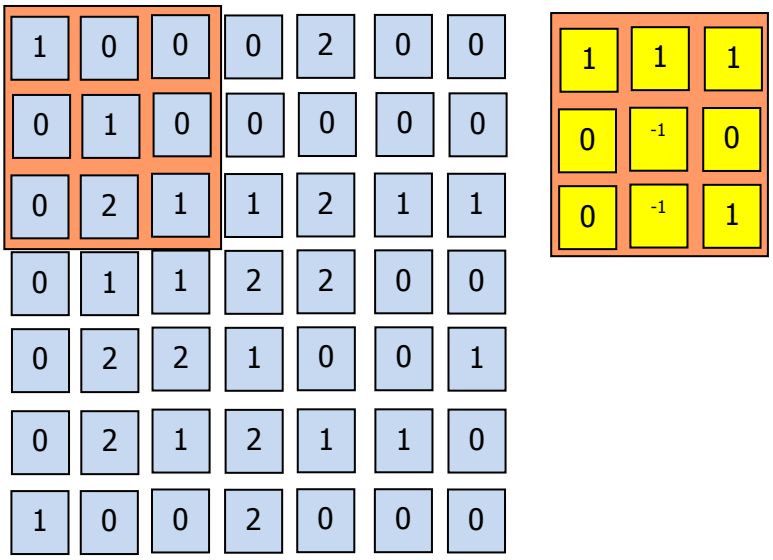
TensorFlow 的补齐方式有些复杂。Padding 参数的设置是两个值的选择： "SAME"和"VALID"。每种选择，通过下面的计算可以算出最后输出 tensor（特征映射组）的 shape。

对于“SAME”padding，我们计算

```
out_height = ceil(float(in_height) / float(strides[1]))
out_width  = ceil(float(in_width) / float(strides[2]))
pad_along_height = ((out_height - 1) * strides[1] +
                    filter_height - in_height)
pad_along_width = ((out_width - 1) * strides[2] +
                   filter_width - in_width)
pad_top = pad_along_height / 2
pad_left = pad_along_width / 2
```

分别算出的 pad_along_height 和 pad_along_width 是在高度和宽度的方向补齐几个值。pad_top=pad_along_height/2 是计算出沿高度方向在顶部和底部各补齐多少个值。如果 pad_along_height=5 则在顶部补齐 2 个值，在底部补齐 3 个值。pad_left 类同。

举例：输入 tensor 是 7*7，滤波器 shape 是 3*3，stride 为 2




```
out_height = 4
out_width  = 4
pad_along_height = (4 - 1) * 2 + 3  -  7 = 2
pad_along_width = (4 - 1) * 2 + 3 -7 = 2
pad_top = 1 （上下各填充一行）
pad_left = 1 （左右各填充一列）
```

补齐以后的输入变成了 9*9 的矩阵，输出是 4*4 的矩阵

上例，如果 Stride 为 1，则输出是 7*7 的矩阵。

```
out_height = 7
out_width  = 7
pad_along_height = (7 - 1) * 1 + 3 -7 = 2
pad_along_width = ((7 - 1) * 1 + 3 -7 = 2
pad_top = 1
pad_left =1
```

对于“VALID”padding，我们计算

```
out_height = ceil(float(in_height - filter_height + 1) /
float(strides[1]))
out_width  = ceil(float(in_width - filter_width + 1) /
```

注：所有上面涉及的补齐都是零补齐

则，上例：输入 tensor 是 7*7，滤波器 shape 是 3*3，stride 为 2，输出矩阵是 3*3；如果，输入 tensor 是 7*7，滤波器 shape 是 3*3，stride 为 1，输出矩阵是 5*5。可以看出，实际上 VALID 就是没有进行补齐运算。

卷积层应用了 ReLu 激活函数 tf.nn.relu(features, name=None)。它计算 rectified linear: max(features, 0).

Pooling 层 h_pool1 的定义中使用的自定义函数定义如下：

```
h_pool1 = max_pool_2x2(h_conv1)
def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

函数 tf.nn.max_pool 在输入上进行 max pooling 操作。其参数如下：

value: 是一个 4-DTensor 即输入， 其 shape=[batch, height, width, channels]；数据类型是 tf.float32。

ksize: 是一个 int 型 list，长度>= 4。 它是池化操作的滤波器的窗口大小，对应输入 tensor 的每个维度。

strides: 是一个 int 型 list，长度>= 4。滑动窗口的步长。

padding: VALID’ 或者 ’SAME’。

data_format: 'NHWC' 或者 'NCHW'。

name: 可选项，分配操作的名称.

3. 第二个卷积层和 pooling 层

再定义第二个卷积层和 Pooling 层的权重

```
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
```

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。

定义第二个卷积层和 pooling 层

```
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
```

```
h_pool2 = max_pool_2x2(h_conv2)
```

4. 全连接层

定义全连接层的权重

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
```

第二个 pooling 层的输出 tensor 的 shape 是 7*7*64，而全连接层有 1024 个神经元。因此，偏置的数量也是 1024。我们会把第二个 pooling 层输出的 tensor 转换成一个向量，即 flaten 操作（可以参考传统神经网络中的一层的神经元都是一个向量的排列）。然后进行全连接层的计算，再加上一个 ReLu 激活函数。

```
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

对全连接层引入 dropout 操作

```
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

dropout 函数 tf.nn.dropout(x, keep_prob, noise_shape=None, seed=None, name=None)

按照概率 keep_prob 对输入的元素输出，输出的元素值按照 1/keep_prob 进行标定；否则输出为 0。标定是让期望和不变 The scaling is so that the expected sum is unchanged.（没理解）

每个元素被保留或放弃是独立的。如果给定了 noise_shape，它是一个向量（或 1-D tensor），仅仅满足 noise_shape[i] == shape(x)[i]的维度进行独立的保留或放弃的决策。例如, 如果 `shape(x) = [k, l, m, n]`，并且 `noise_shape = [k, 1, 1, n]`, each batch and channel component will be kept independently and each row and column will be kept or not kept together.

运行下面的代码可以帮助理解 TensorFlow 中 dropout 是怎么工作的

```
import tensorflow as tf

d=[3.0,1.0,2.0,4.0]
input1 = tf.Variable(tf.constant(d))
drop = tf.nn.dropout(x=input1, keep_prob=0.5)
init_op=tf.initialize_all_variables()

mycount=len(d)*[0]

sess=tf.Session()
for _ in range(10000):
    sess.run(init_op)
    d=sess.run(drop)
    ind=[i for i, x in enumerate(d) if x==0]
```

```
    for x in ind:
        mycount[x] = mycount[x]+1

print(d)
print(mycount)
```

运行结果是：（因为存在随机选择节点，因此每次运行的结果会不同）

```
[ 6.  2.  0.  8.]
[4995, 5009, 5039, 4916]
```

可以看出 tf.nn.dropout 函数对输入（x=input1）按照 keep_prob（50%）的概率来决定每个输入的元素是否作为 tf.nn.dropout 的输出。如果决定输出，其输出值是原值乘上 1/keep_prob

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，因此定义的权重和偏置如下：

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
```

输出层对 dropout 的输出和权重进行矩阵相乘，然后使用 softmax 激活函数。

```
y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

6. 训练模型

下面是用 CNN 在 MNIST 数据集上进行手写数字识别的实验.

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_conv), reduction_indices=[1]))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
sess.run(tf.initialize_all_variables())
for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1], keep_prob: 1.0})
        print("step %d, training accuracy %g"%(i, train_accuracy))
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

注：构造 CNN 的一个难点就是对每层的输入 tensor，filter 的 shape，以及最后输出 tensor 的 shape 要计算清楚。

练习：运行上述的程序，看一看 CNN 在 MNIST 数据集上的性能如何。

第五章：词的表示学习

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

在 NLP 领域采用词的分布式表示（distributed representation of words）和神经网络构建的神经网络语言模型很多研究已经证明其性能超过了 N-gram 语言模型。

“word embeddings 词嵌入”或者“word representation 词表示”或者“distributed representations of words 分布式词表示”三个术语都是一个意思，是用一个实数向量表示词，本文后面将其称为词向量。词的表示学习在深度学习出现之前就已经有了。2001 年 Bengio 就在两篇论文中提出了 word embeddings。而和 word embedding 思想相似的关于符号的分布式描述则在更是早在 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。大多数宣称用了 Deep Learning 的论文，其中往往也用了词向量。

一个 word embeddings $\mathbf{W:words \rightarrow R^n}$

是一个函数将词映射到高纬向量 (可以达到 200 到 500 个维度).例如

W("cat")=(0.2, -0.4, 0.7,...)

W("mat")=(0.0, 0.6, -0.1,...)

通常建立 word embedding 即词的表示学习是一个学习任务。

有很多学习词的表示的方法，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，

vec (King") - vec ("Man") + vec ("Woman") ≈ vec ("Queen")

vec("Madrid") - vec("Spain") + vec("France") ≈ vec("Paris")

第一节：背景知识

1. 词向量

One-hot Representation

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

"话筒"表示为 [0 0 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 …]
"麦克"表示为 [0 0 0 0 0 0 0 0 **1** 0 0 0 0 0 0 0 …]

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

Distributed Representation of Words

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

2. 统计语言模型

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

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

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

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

当 n=2，这是二元语言模型，一个词的条件概率仅考虑它前面的一个词。

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

因为 n 的值越大，语言模型越复杂。在信息检索和文本挖掘中，一元模型往往已经足够。深度学习中很多研究拓展到二元语言模型。如果 n>2 更高阶的模型往往过于复杂，得不偿失。

统计语言模型的应用非常广泛，如语音识别，机器翻译等。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，

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

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

3. NNLM

NNLM 即神经网络语言模型，也称作 Feedforward Neural Net Language Model（前馈 NNLM），是用神经网络训练语言模型。Bengio 发表的论文“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 5.1 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},..., w_{t-2}, w_{t-1} 就是前 n−1 个词。模型根据已知的 w_{t-n+1}，...，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。

网络的隐层是将 C(w_{t-n+1}),...,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。

网络的输出层一共有|V|个节点（V 是词表），节点 y_i 表示下一个词为 w_i 的未归一化 log 概率。最后使用 softmax 激活函数将输出值 y 归一化成概率。最终，y 的计算公式为：

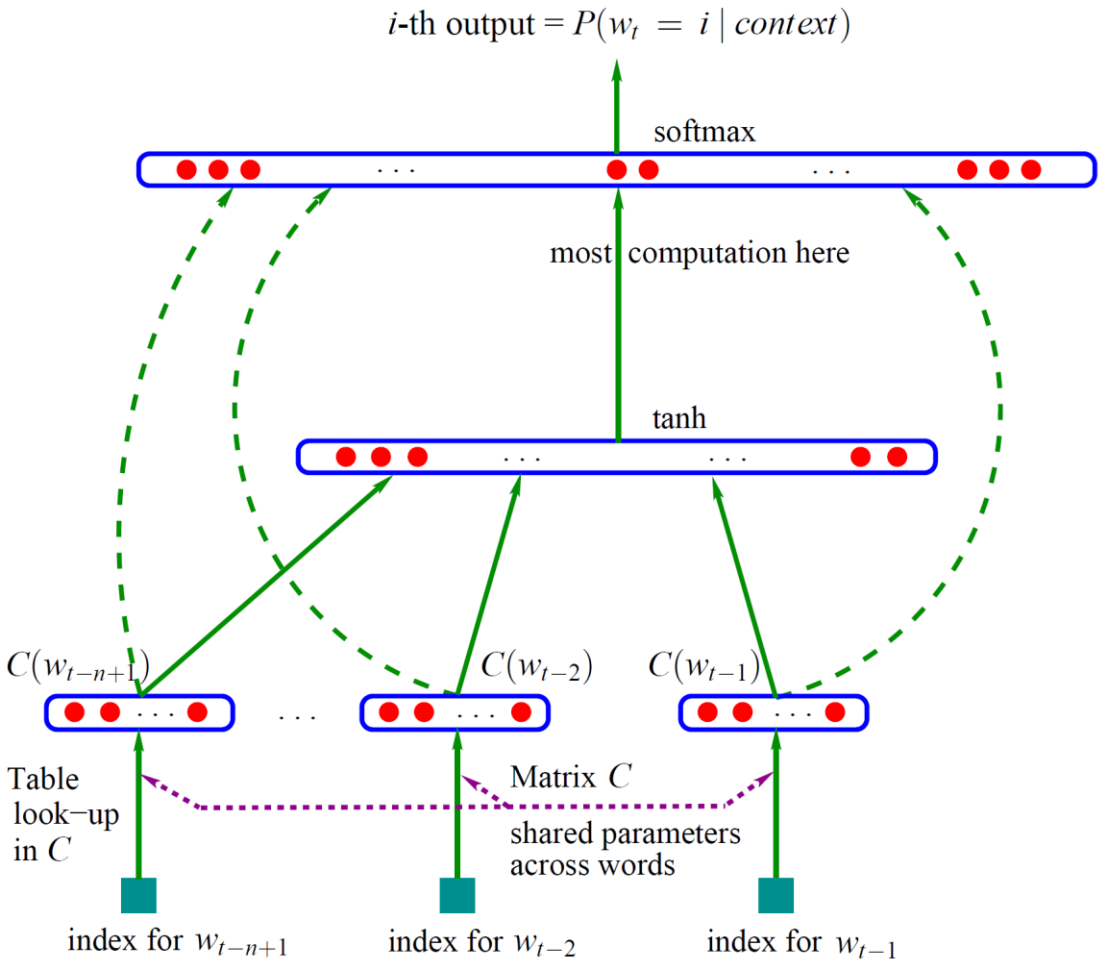


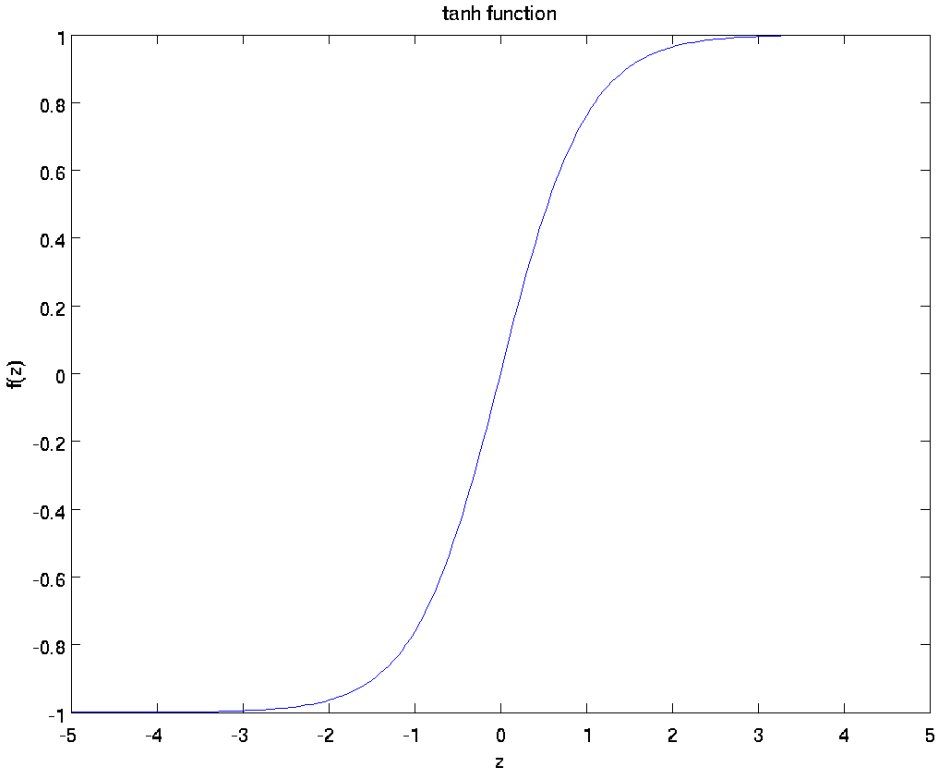
图 5.1. 神经网络语言模型

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

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

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

Tanh 激活函数，将输出压缩到了-1 到 1 之间。



第二节：word2vec 和 GloVe

word2vec 其实是一个词表示学习的工具箱（https://code.google.com/archive/p/word2vec/），该工具箱的实施基于 Google 公司的 Tomas Mikolov 的两篇论文：“*Efficient Estimation of Word Representations in Vector Space*”和 ”Distributed Representations of Words and Phrases and their Compositionality”。这个工具箱实施了词表示学习的两个模型 continuous bag-of-words（CBOW）和 skip-gram architectures（见上述论文）。学习的词向量或 word embedding 用在进一步的文本挖掘，NLP 中。例如，再用在深度学习中作为深度神经网络的输入。

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

一种简单的方法来发现词之间的相似性是计算词向量之间的距离。例如，如果输入 france，与 france 距离最近的词如下：

spain	0.678515
belgium	0.665923
netherlands	0.652428
italy	0.633130

switzerland	0.622323
luxembourg	0.610033
portugal	0.577154
russia	0.571507
germany	0.563291
catalonia	0.534176

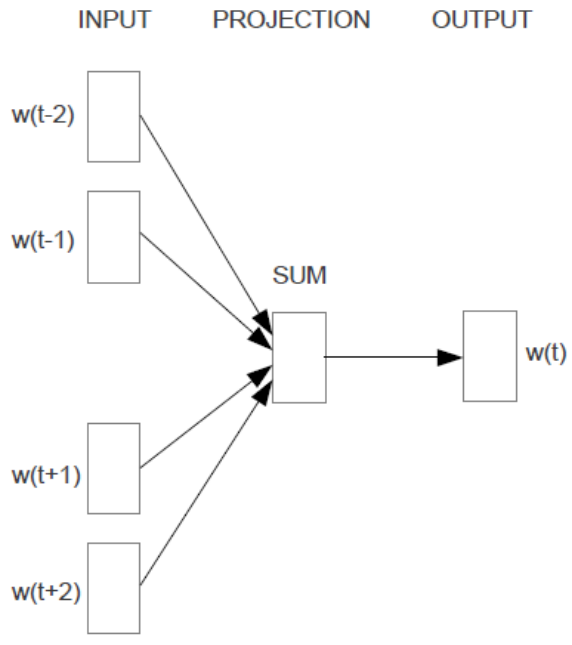
word2vec 非常受欢迎的另一个原因是其高效性，Mikolov 在论文[2]中指出一个优化的单机版本一天可训练上千亿词。

1. CBOW

CBOW 是一种与前馈 NNLM 类似的模型，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。

如下图所示。可以看出，CBOW 模型是预测 $P(w_t|w_{t-k},...,w_{t-1}, w_{t+1},...,w_{t+k})$。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。（是说，在训练文本上使用一个 k*2 的窗口来获得连续的词的序列，但实际上在这个词的序列内部实际上是无关次序的，因为都是要求和）



CBOW

图 2. CBOW

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。

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，但不是基于当前词 w_t 的上下文预测 $p(w_t|context)$，skip-gram 使用当前的词向量来预测该词之前和之后各 k 个词的概率。即预测概率 $p(w_i|w_t)$，其中$t-c\leq i\leq t+c$且$i\neq t$，参数 c 决定窗口大小。假设存在一个 $w_1,w_2,w_3,\ldots,w_T$ 的词组序列，Skip-gram 的目标是最大化：

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

基本的 skip-ngram 使用 softmax 函数定义如下：

$$p(w_O|w_I)=\frac{\exp({v'_{w_O}}^{T}v_{w_I})}{\sum_{t=1}^{W}\exp({v'_t}^{T}v_{w_I})}$$

v_w 和 v'_w 是词 w 的输入和输出的词向量表示；W 是词汇表中的大小。在具体的实施中，该模型很不实用，因为计算梯度$\nabla\log(p(w_{t+j}|w_t))$的代价太大。

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

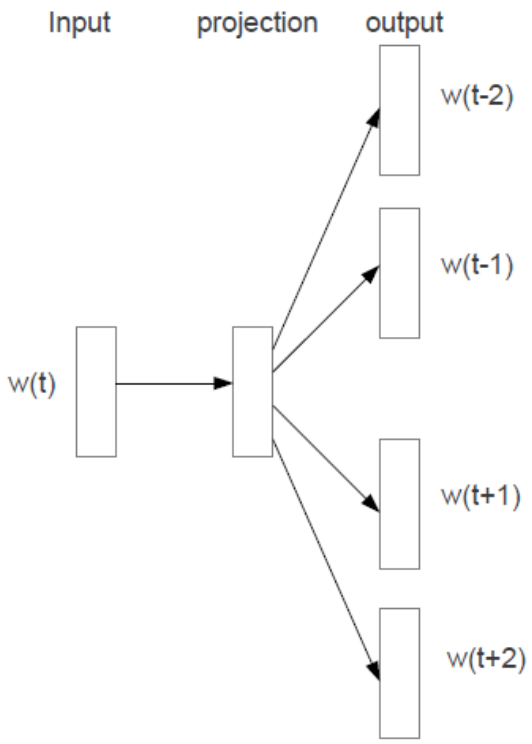


图 3. Skip-gram

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

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

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。

3. GloVe

两个主要的词表示学习的方法是：全局的矩阵分解方法，如 LSA，和局部的上下文窗口方法，如 skip-gram 方法。两个系列都有其缺点(GloVe 的论文中指出的)，LSA 重复的利用了统计信息，但它们在 word analogy 任务中表现比较差；而 skip-gram 虽然在 word analogy 任务上表现的很好，但它们没有充分的利用语料库的统计信息，因为它是在局部的上下文窗口中训练，而没有利用全局的共现性。统计语料库中的 word occurences 是所有无监督学习 word representation 的方法的主要信息源。

Global Vector for Word Representation(GloVe)自称是 word2Vec 的发展。它在 2014 年提出后，吸引了相当多的关注。

GloVe 模型是

$$w_i^T \tilde{w}_k + b_i + \tilde{b}_k = \log(X_{ik})$$

X 是 word-word 共现计数矩阵。X_{ij} 指示词 i 出现在词 j 上下文 context 的次数。b_i 是词向量 w_i 的偏置是一个标量。

$w \in R^d$ 是词向量，$\tilde{w} \in R^d$ 是词 w 作为别的词的 context 时的词向量。即，意味着每个词 w_i 有两个词向量，一个是作为一个中心词时的词向量。每个词有 context，即这个词前后窗口内的词集合。w_i 作为别的词 w_j 的 context 时也有个词向量。

建立上述模型的目标函数，在目标函数中引入一个加权函数 $f(X_{ij})$。得到

$$J = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

V 是词汇表，$f(X_{ij})$ 是权重函数，定义为

$$f(x) = \begin{cases} (x/x_{\max})^{\alpha} & \text{if } x < x_{\max} \\ 1 & \text{otherwise} \end{cases}$$

我理解的该模型的工作方法是

（1） 首先从语料库建立 word-word 共现矩阵 X。GloVe 的论文中，设置共现的 context 窗口是前 15 和后 15 个 words。

（2） 词向量 w 是模型最后要学习的结果，学习时要给 w 初始化。初始化的值是在 (-0.5~0.5)/wordVectorSize 的随机值，wordVectorSize 是词向量维度，GloVe 论文中设为 50。

（3） 最后用 $w + \tilde{w}$ 做为词 w 的词向量。

（4） 该方法就是一个矩阵分解方法。

（5） GloVe 批评 skip-gram 是在局部的上下文窗口中训练，而没有利用全局的共现性。而 GloVe 也是在统计上下文窗口中词的共现啊？

GloVe 的模型训练是用 AdaGrad 算法。原论文对训练过程的讲解不好理解，可以参考 CMU 的一篇对 GloVed 注释论文。这里不详细讨论了。

第三节：TensorFlow 的词表示学习

TensorFlow 提供 word2vec 的实现，包括 CBOW 和 skip-gram。算法上这两个模型是相似的，除了 CBOW 从 context(上下文)（如，the cat sits on the）预测目标词（如，mat），而 skip-gram 做相反的预测过程，从目标词预测上下文。从统计学上看，CBOW 更适合小数据集，skip-gram 在大数据集下表现的更好。

为了方便了解 TensorFlow 的 word2vec 的实施，我们再用 tensorflow 资料中的描述，介绍一下用神经网络实现的语言模型（和前面的 NNLM 不同），下面简称神经语言模型。

神经语言模型使用最大似然法按照 Softmax 函数训练模型，获得给定前一个或几个词（history, H）的目标词（target）的最大概率。Softmax 函数就是将一个任意的实数向量 z 规范化产生一个向量$\sigma(z)$，满足每个元素值在（0,1），并且所有元素值的指数和是 1。

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for } j = 1, \dots, K$$

神经网络语言模型如下：

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

V 是词汇表；score(w_t,h)计算词 w_t和 context h 的相匹配性。最大似然法训练该模型，最大化目标函数

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

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。

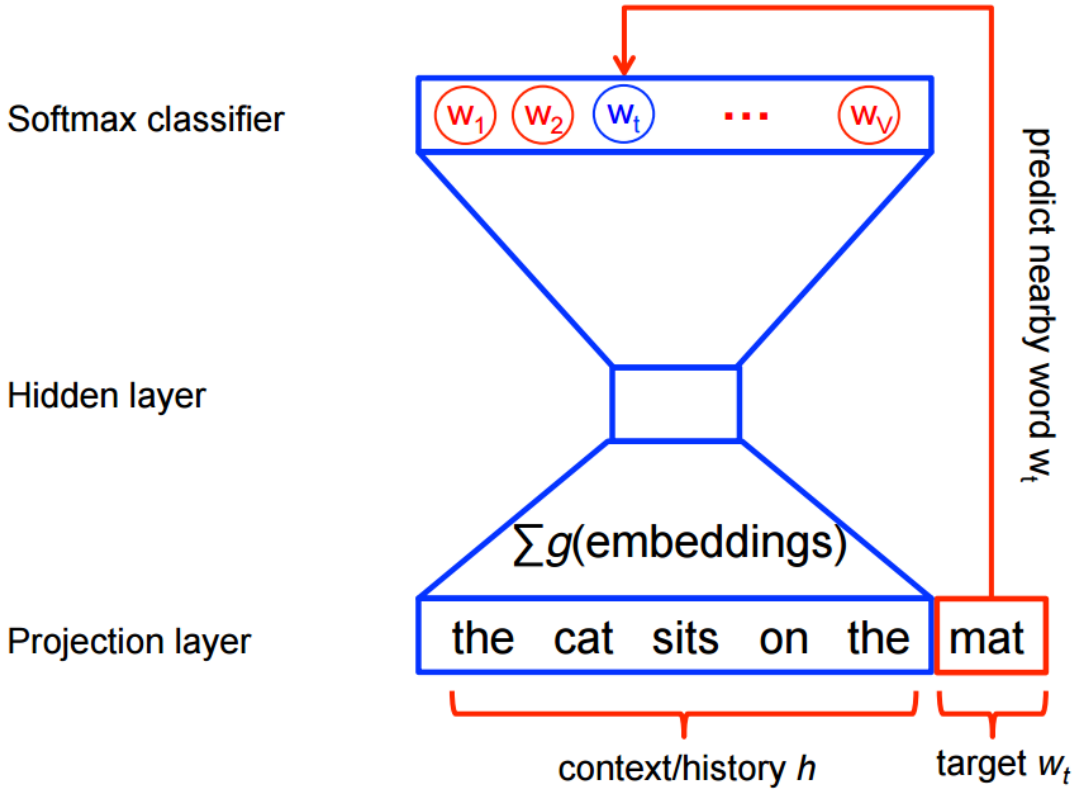


图 4. 神经网络语言模型

然而，在 word2vec 的模型学习中，不需要上图的全概率模型。CBOW 和 skip-gram 使用了一个二分类器（logistics 回归）在相同 context 下，区分真实的目标向量 w_t 和 k 个噪声向量 $\tilde{w}$。下图 5 是一个 CBOW 模型。Skip-gram 模型可以理解就是把该图倒置。

该模型的目标函数是

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

这里$\log Q_\theta(D = 1|w_t, h)$是已知 context 上下文 h，能在训练集 D 看见词 w_t 的二分类 logistics 回归的概率。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 context 上下文 h 能在 D 看见目标词 w_t，而不会看见 k 个噪声词$\tilde{w}$。

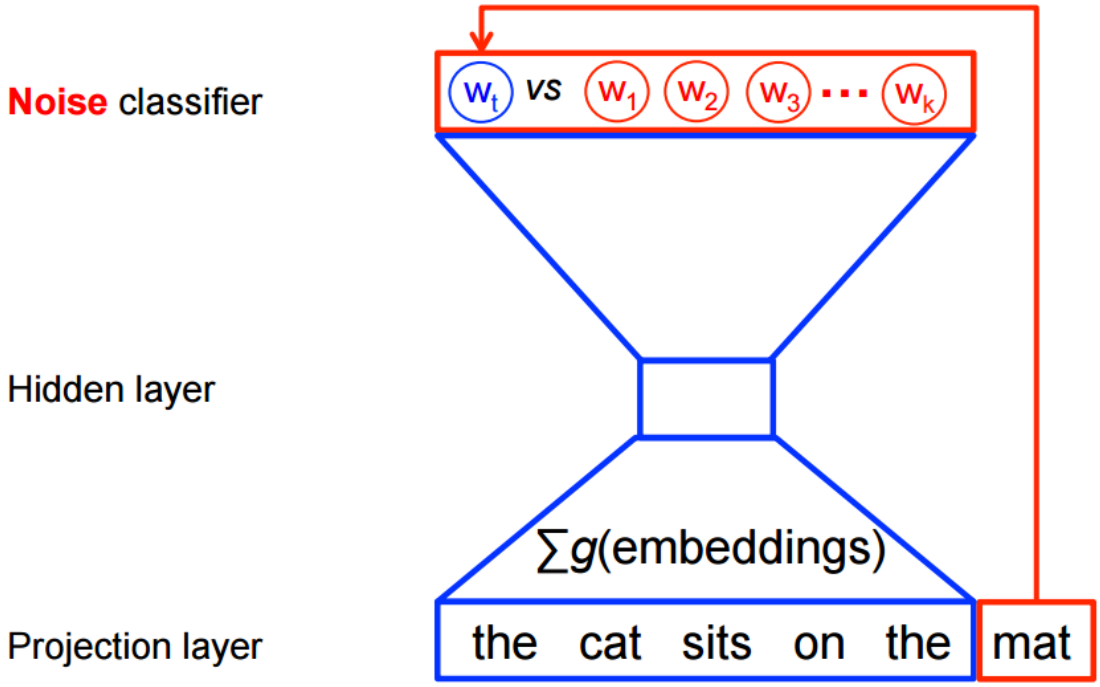


图 5. CBOW

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-contrative estimation (NCE)理论相似。TensorFlow 提供了 NCE 计算损失函数的方法 tf.nn.nce_loss()。

下面我们将介绍 skip-gram 的 tensorflow 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）对’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 ‘（输入，输出）对’ 集合。

(quick, the), (quick, brown), (brown, quick), (brown, fox)…

目标函数是在整个数据集上定义的，但 word2vec 采用随机梯度下降 SGD 来训练模型（一次只使用一条训练数据）或者 minibath。batch 的取值是$16 \leq \text{batch_size} \leq 512$。

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

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

优化的目标是对词向量θ做更新，来最大化该目标函数。首先需要获得梯度$\frac{\partial J_{NEG}}{\partial \theta}$（我们不需要自己计算梯度，TensorFlow 提供了很方便的方式来计算梯度）。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。

用 TensorFlow 实施 skip-gram 时，首先定义词向量的矩阵（或称作查找表 lookup），这是一个很大的随机矩阵。

```
embeddings = tf.Variable(tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
```

因为词向量就是模型要学习的参数，因此定义成 TensorFlow 的 Variable。初始化每个元素的值在[-1, 1]。

noise-contrastive estimation 损失函数按照 logistics 回归模型定义。为此，需要为词汇表中的每个词定义权重和偏置。也称作与输入词向量对应的输出权重。

```
nce_weights = tf.Variable(tf.truncated_normal([vocabulary_size, embedding_size], stddev=1.0 / math.sqrt(embedding_size)))

nce_biases = tf.Variable(tf.zeros([vocabulary_size]))
```

对于词汇表中的每个词，分配了一个整数编码。Skip-gram 模型的输入是一个 batch 的整数集合，即 context 词的集合；另一个是目标词。输入被定义成 placeholder

```
train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
```

紧接着需要从查找表，将输入的词的整数编号映射成词向量（embeddings）

```
embed = tf.nn.embedding_lookup(embeddings, train_inputs)
```

模型定义的损失函数是 NCE 损失函数，使用 tf.nn.nce_loss 来定义

```
tf.nn.nce_loss(weights, biases, inputs, labels, num_sampled, num_classes, num_true=1, sampled_values=None, remove_accidental_hits=False, partition_strategy='mod', name='nce_loss')
```

重要的参数如下：

- weights: A Tensor of shape [num_classes, dim], or a list of Tensor objects whose concatenation along dimension 0 has shape [num_classes, dim]. The (possibly-partitioned) class embeddings.
- biases: A Tensor of shape [num_classes]. The class biases.
- inputs: A Tensor of shape [batch_size, dim]. The forward activations of the input network.
- labels: A Tensor of type int64 and shape [batch_size, num_true]. The target classes.
- num_sampled: An int. The number of classes to randomly sample per batch.
- num_classes: An int. The number of possible classes.

我们建立损失函数

```
loss = tf.reduce_mean(tf.nn.nce_loss(nce_weights, nce_biases, embed, train_labels, num_sampled, vocabulary_size))
```

上面在计算图上定义了损失函数节点，就需要定义一个优化器来计算梯度和更新参数。这里使用随机梯度下降的优化方法。

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=1.0).minimize(loss)
```

训练模型的过程，就是用 feed_dict 将数据放入 placeholder，在循环中调用 session.run()用新数据训练模型。

```
for inputs, labels in generate_batch(...):
  feed_dict = {training_inputs: inputs, training_labels: labels}
  _, cur_loss = session.run([optimizer, loss], feed_dict=feed_dict)
```

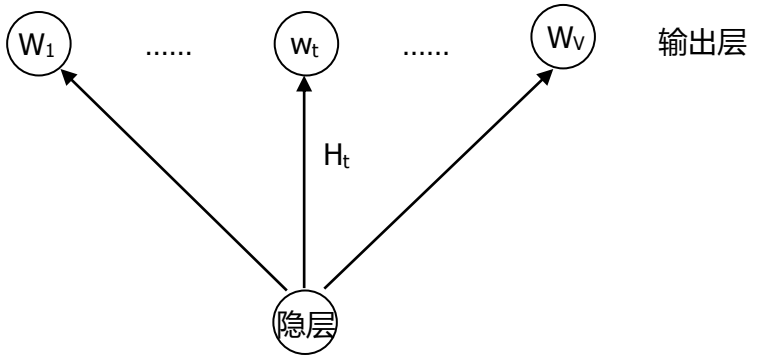
对于如何实施 word2vec 我还是有很多疑惑，原论文很多细节没讲。通过剖析 tensorFlow 的 word2vec 示例程序 word2vec_basic.py。我理解的 skip-gram 模型如下图 6。

设词汇表 V，它的大小是|V|，词向量的维度是 k。

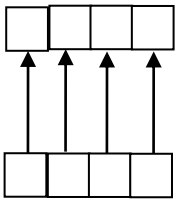
训练集是(quick, the), (quick, brown), (brown, quick), (brown, fox)…这样的（输入，输出）对的集合。

我们以 SGD 为例，即此时模型的输入就是一个（输入，输出）对，然后更新参数。模型的参数有两个一个是从隐层到输出层的边的权重 H，维度是[|V|,k]。H 对应上面

TensorFlow 代码中的 nce_weights。隐层是一个节点，它的输出的维度是 k 维和输出层节点的全连接，但连接是向量对应。



单独的一条边权重是 H_t 是一个 k 维向量。



输出层的 w_t 节点

边权重 H_t，维度为 k

隐层的节点

模型的输出层有|V|个节点，激活函数是 logistics 回归，每个节点的输出是一个概率值。

TensorFlow 中 Skip-gram 实施过程如下：为了好讲解，我们假设采用的是 SGD，即每次用一个实例训练，而实际上采用的是 minibatch。

（1） 建立 embeddings，shape 为[|V|, k]，初始化元素值（-1,1）。embeddings 被定义为模型的 Variable。因此在训练中它的值会被更新。

（2） 建立权重矩阵 H，对应前面代码的 nce_weights。它的 shape 是[|V|,k]。它也被定义为模型的 Variable。再建立一个偏置 Variable b，它的 shape 是[|V|,1]

（3） 从训练集取一条训练数据（input, output）。Input, output 是一个词在词汇表中的编号。从查找表 embeddings，将 input, output 映射成词向量。

（4） 采用 negative sampling，抽样 n 个 Output 的对应负样例。

（5） Input 的词向量*H_t+b，经过 logistics 回归激活函数，算出一个 true logit。

（6） Input 的词向量*H_{neg}+b，经过 logistics 回归激活函数，算出一个 sampled logit。

（7） 使用 true logit 和 sampled logit 计算交叉熵损失函数。

（8） 按照误差反向传播理解，由上一步计算的误差修正权重 Ht

（9） 反向传播修正输入 input 的词向量。在一趟训练中，只有目标词和负例抽样的词（均是输出层的节点）参与计算，所以只有这些词的边的权重被修正，也只有这些词的词向量被修正。

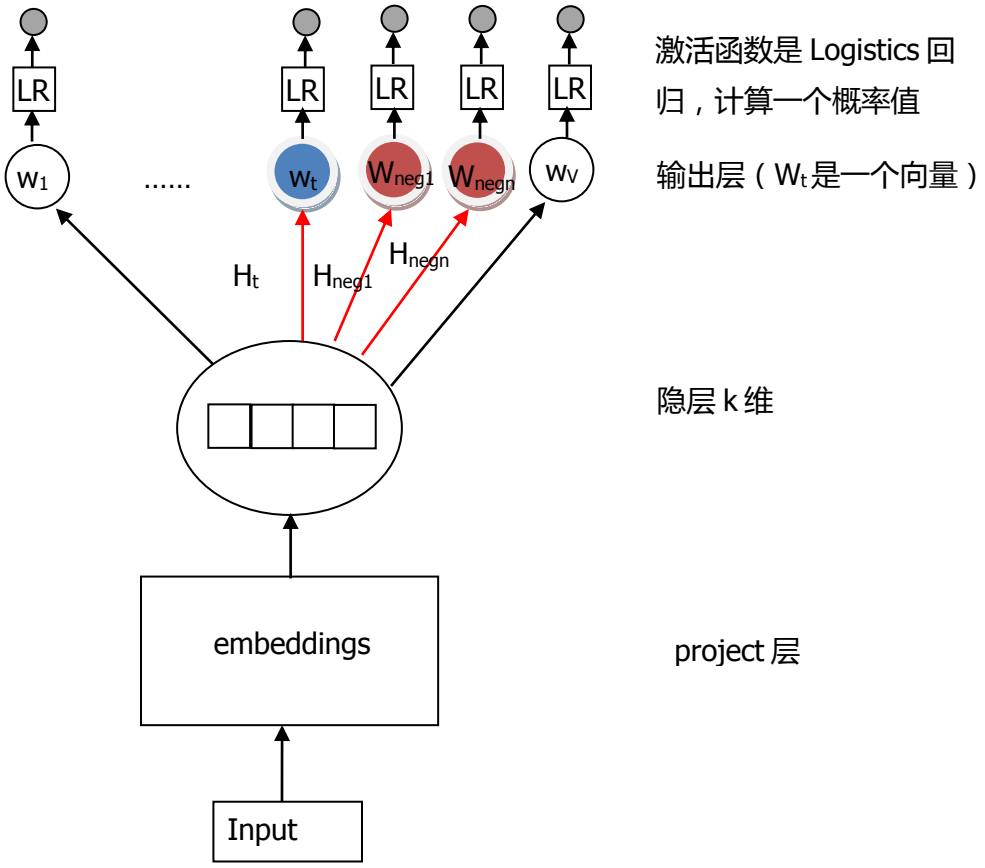


图 6. TensorFlow 中实施的 skip-gram 模型

第六章：基于 CNN 的文本分类

第一节：CNN 文本分类模型

Yoon Kim 在 EMNLP2014 上发表的论文 Convolutional Neural Networks for Sentence Classification，这篇文章可以说是 cnn 模型用于文本分类的开山之作（其实第一个用的不是他，但是 Kim 提出了几个变体 variants，并有详细的调参）。

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：

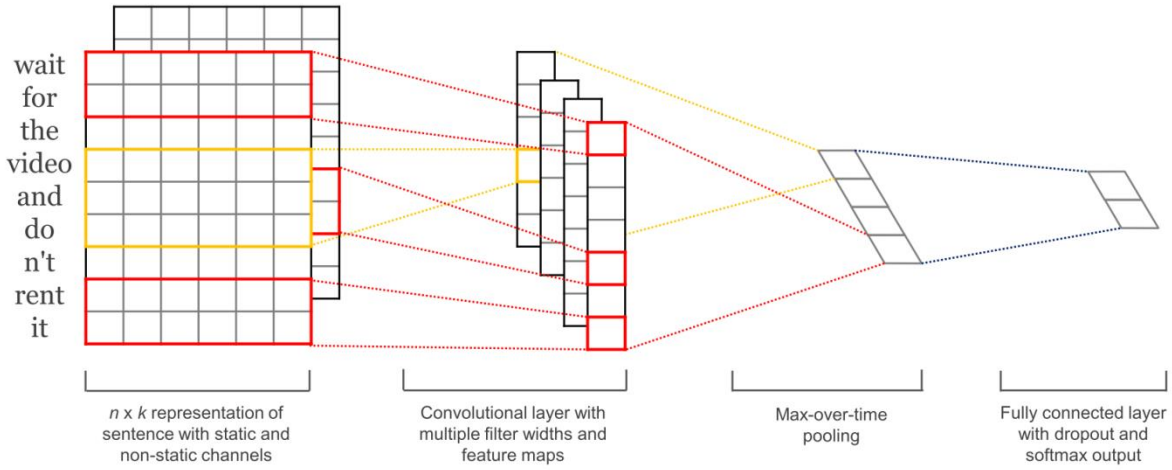


图 6.1 Model architecture with two channels for an example sentence.

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

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

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。通常$x_{i:i+j}$是指词向量$x_i, x_{i+1}, \ldots, x_{i+j}$。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）。

在前一章中讲到才是图片有三个通道（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由两种方式。一是 Mikolov 使用它的 word2vec 在 Google News dataset 上训练词向量。产生的词向量的维度是 300，包含三百万个词和词组。这个是公开的词向量集合（https://code.google.com/archive/p/word2vec/）。第二个是如前一章所示，将词向量也作为 CNN 模型中的参数，在 CNN 的分类模型中经过训练后得到一组词向量。Yoon Kim 在输入层，或 embeddings 层，建立两种词向量的输入，于是得到输入层有两个特征映射，第一个称为 static channel;第二个称为 non-static channel。

（2）卷积层：在输入层上加一个卷积层。一个滤波器$w \in R^{hk}$应用到一个有 h 个词的窗口。窗口大小是 h*k，k 是词向量的长度，见图 6.1。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 $c=[c_1,c_2,...,c_{n-h+1}]$。

卷积层有个参数 region size（h*k）。k 值是 embeddings 维度，h 是词的个数。可以有多个不一样的 h 值。在每个 region size 上可以设定多个滤波器。如果 region size 是 m，滤波器的个数是 n，则卷积层上有 m*n 个滤波器（参考图 6.3）。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。（注：图 6.1 的卷积操作输入中第一个特征映射的 shape 是[7,1]，正确应该是[8, 1]；该图没有描述出一个滤波器在两个输入特征映射上的操作）。

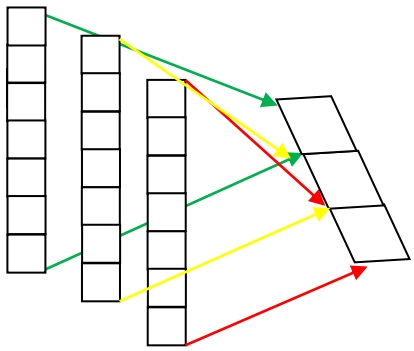


图 6.2 pooling 层

（3）pooling 层：子采样操作应用的是 maxpooling 操作。简单地说就是子采样时的滤波器采样最大值的方法$\hat{c} = \max(C)$进行子采样；滤波器的 shape 是和一个特征映射的 shape 一致。其思想是捕获特征映射中最重要的特征。因此，其结果是一个 feature map 子采样操作后得到的是一个值。每个子采样操作的结果拼接成一个向量，构成 pooing 层的输出。如图 6.2 所示。

（4）softmax 输出层：输出层有两个节点，即二分类的结果。Pooling 层和输出层采用全连接。

（5）dropout：在 pooling 层的输出加 dropout 操作。

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。

一个详细描述的文本分类 CNN 结构图见图 6.3（详见论文 A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification）。图 3 的 CNN 的结构是文本分类的结构示例图。它的一些参数不太一样，如滤波器的 region size，滤波器的个数等。

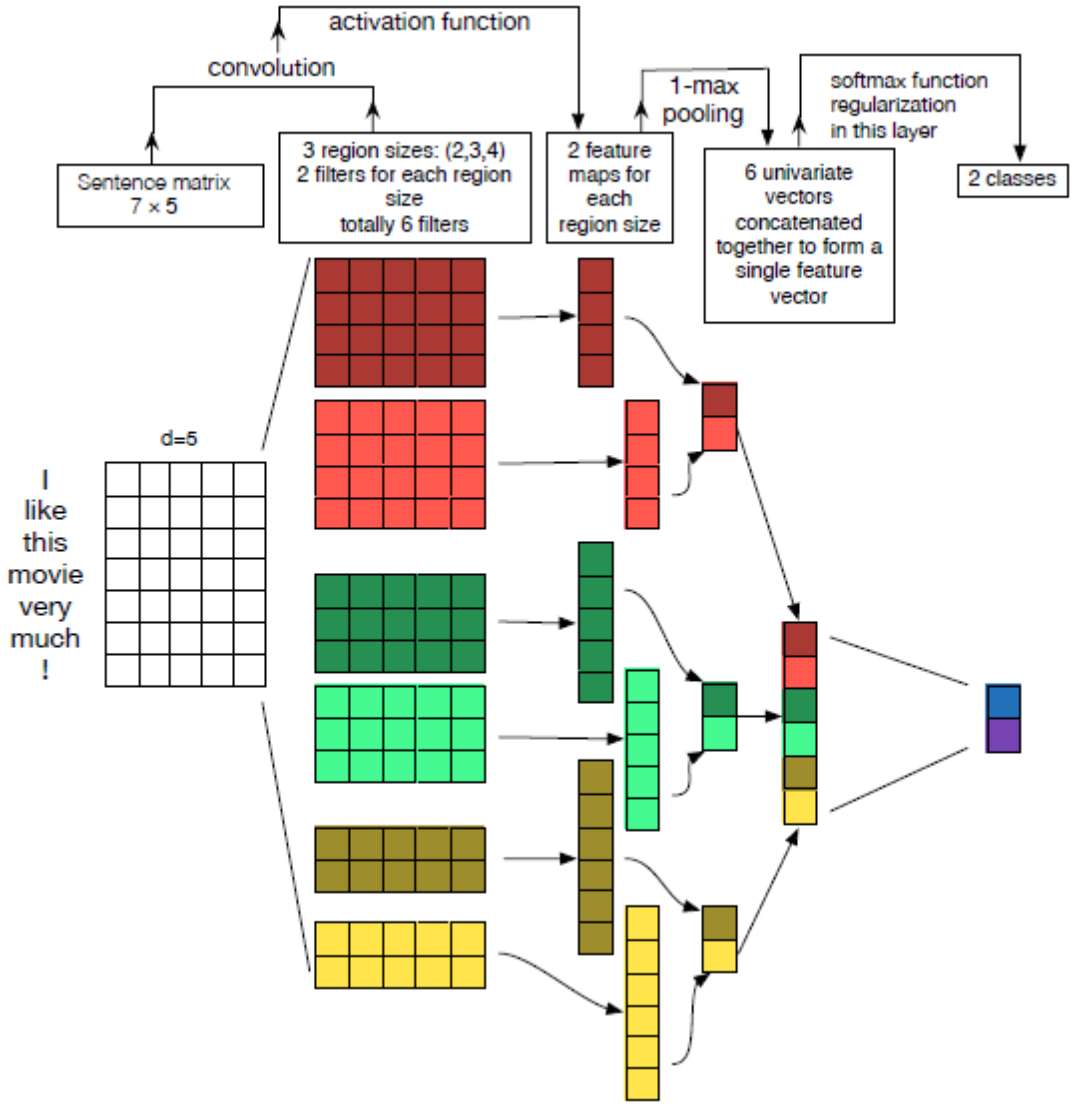


图 6.3 有多个 window 的滤波器的文本分类 CNN

第二节：TensorFlow 实现 CNN 文本分类模型

wildml 对这篇 paper 有一个 tensorflow 的实现（http://www.wildml.com/2015/12/implementing-a-cnn-for-text-classification-in-tensorflow/）。源程序有 4 个文件。为了好理解，便于讲授和理解工作过程。我将源文件中的 train.py 和 text_cnn.py 合并改写了一个 mycnn4text.py 文件，它放弃了一些辅助功能，代码更简单直观。下面以 mycnn4text.py 讲授实现 CNN 文本分类的过程。

1 参数设置

在该版本的模型中，未考虑使用公开词向量建立输入，即只有一个随机初始化的词向量。也即输入的特征映射（通道）是 1。词向量的长度是 128；卷积层滤波器有三种

window 或 region size：[3, 4, 5]。每种 window size 的滤波器是 128 个。Dropout 的 keep_prob 是 0.5；minibatch 的随机梯度下降训练中，batch size 是 64。总结其重要参数设置是：

参数名	值	解释
embedding_dim	128	Dimensionality of character embedding
filter_sizes	[3, 4, 5]	Comma-separated filter sizes
num_filters	128	Number of filters per filter size
keep_prob	0.5	Dropout keep probability
batch_size	64	Batch Size
l2_reg_lambda	0	L2 regularizaion lambda
num_epochs	200	训练模型时迭代次数
num_classes	2	输出层节点数

2 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）

```
def load_data_and_labels():
    """
    Loads MR polarity data from files, splits the data into words and generates labels.
    Returns split sentences and labels.
    """
    # Load data from files
    positive_examples = list(open("./data/rt-polaritydata/rt-polarity.pos",
"r").readlines())
    positive_examples = [s.strip() for s in positive_examples]
    negative_examples = list(open("./data/rt-polaritydata/rt-polarity.neg",
"r").readlines())
    negative_examples = [s.strip() for s in negative_examples]
    # Split by words
    x_text = positive_examples + negative_examples
    x_text = [clean_str(sent) for sent in x_text]
    # Generate labels
    positive_labels = [[0, 1] for _ in positive_examples]
    negative_labels = [[1, 0] for _ in negative_examples]
    y = np.concatenate([positive_labels, negative_labels], 0)
    return [x_text, y]
```

对应到输出层有两个节点，标签是一个 list 结构[0,1]是正例标签；[1, 0]是负例标签。

输入层"喂"给模型的数据的 shape 是[batch_size, document_max_size]。document_max_size 是根据输入数据中最长的句子来定。数据的一行是一条评论的词的序列。每个词已经被编号。（ mycnn4text.py ）

```
x_text, y = data_helpers.load_data_and_labels()

# Build vocabulary
max_document_length = max([len(x.split(" ")) for x in x_text])
vocab_processor = learn.preprocessing.VocabularyProcessor(max_document_length)
x = np.array(list(vocab_processor.fit_transform(x_text)))
```

learn.preprocessing.VocabularyProcessor 初始化一个 VocabularyProcessor 实例. 它的参数包括

max_document_length: Maximum length of documents.if documents are longer, they will be trimmed, if shorter - padded.

min_frequency: Minimum frequency of words in the vocabulary.

vocabulary: CategoricalVocabulary object.

经 vocab_processor 转换后得到的 numpyy array 数据对象 x 是一个[文档集合文本个数，max_document_length]的二维数据结构。x 的一行是被编码后的一篇文档。

3.输入层

详见 mycnn4text.py

输入层在源码中称为 embedding layer。创建 placeholder 作为 "喂" 给模型的输入数据。

```
sequence_length=x_train.shape[1]

input_x = tf.placeholder(tf.int32, [None, sequence_length], name="input_x")
input_y = tf.placeholder(tf.float32, [None, num_classes], name="input_y")
dropout_keep_prob = tf.placeholder(tf.float32, name="dropout_keep_prob")
```

这里 sequence_length 等于 max_document_length。num_classes 等于输出层的节点数，这里是二分类，所以值为 2。

在输入层，建立所有词向量的 Variable，又称为查找表，命名为 C。

```
C = tf.Variable(tf.random_uniform([vocab_size, embedding_dim], -1.0, 1.0), name="lookuptable")
```

因为，在当前这个版本的文本分类 CNN 中词向量是作为了待学习的参数，所以建立 Variable 作为词向量。该词向量 Variable 的 shape=[词汇表的大小，词向量的长度]。

前面建立的 VocabularyProcessor 实例 vocab_processor 可以获得词汇表的大小

```
vocab_size=len(vocab_processor.vocabulary_)
```

tensorflow 提供一个查找表函数，它可以根据 word 编号查找对应的 word embeddings（词向量）。

```
tf.nn.embedding_lookup(params, ids, partition_strategy='mod', name=None, validate_indices=True)
```

重要的两个参数，parames 是所有的词向量，例如上面的 W；ids 是词的编号集合。该函数从词向量集合上查找 ids 这个集合里的词的词向量。返回的结果是一个 tensor，它的 shape=[？，max_document_length，embeddings_size]。

输入层创建转换成词项量的输入 embedded_chars

```
embedded_chars = tf.nn.embedding_lookup(C, input_x)
```

因为 tensorflow 的卷积操作（详见第 4 章）

```
tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=None, data_format=None, name=None)
```

input 是输入 tensor，它的 shape 是 [batch, in_height, in_width, in_channels]。因此需要给 embedded_chars 增加一个维度

```
embedded_chars_expanded = tf.expand_dims(embedded_chars, -1)
```

tf.expand_dims(input, dim, name=None)函数插入一个维度到一个 tensor 的 shape。新增加的维度的 size 是 1。维度索引 dim 从 0 开始。如果给一个负值则是从后面开始插入维度。

举例：

```
# 't'一个tensor，它的shape=[2]
shape(expand_dims(t, 0)) ==> [1, 2]
shape(expand_dims(t, 1)) ==> [2, 1]
shape(expand_dims(t, -1)) ==> [2, 1]

# 't2'是一个tensor它的shape=[2, 3, 5]
shape(expand_dims(t2, 0)) ==> [1, 2, 3, 5]
shape(expand_dims(t2, 2)) ==> [2, 3, 1, 5]
shape(expand_dims(t2, 3)) ==> [2, 3, 5, 1]
```

4.卷积层+pooling 层

TensorFlow 创建卷积层使用 conv2d 函数

tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=None, data_format=None, name=None)

其参数 filter 是一个 variable，它的 shape 是 [filter_height, filter_width, in_channels, out_channels]。此处 in_channels 即对应输入图像的通道个数；out_channels 是该卷积层输出的通道数量，它对应的是滤波器的个数，也即特征映射个数。

所以，定义滤波器的 shape

```
filter_shape = [filter_size, embedding_dim, 1, num_filters]
```

创建卷积层的权重矩阵，即创建滤波器和偏置

```
W = tf.Variable(tf.truncated_normal(filter_shape, stddev=0.1), name="W")
b = tf.Variable(tf.constant(0.1, shape=[num_filters]), name="b")
```

创建卷积层，采用 relu 激活函数

```
conv = tf.nn.conv2d(
                embedded_chars_expanded,
                W,
                strides=[1, 1, 1, 1],
                padding="VALID",
                name="conv")
h = tf.nn.relu(tf.nn.bias_add(conv, b), name="relu")
```

conv 和 h 是 tensor，它们的 shape=[batch_size, sequence_length-filter_size+1, 1, 滤波器个数)。

再在卷积层上加上一个 pooling 层。pooling 层采用最大值采样，其函数如下：

tf.nn.max_pool(value, ksize, strides, padding, data_format='NHWC', name=None)

Performs the max pooling on the input.

参数：

value: A 4-D Tensor with shape [batch, height, width, channels] and type tf.float32.

ksize: A list of ints that has length >= 4. The size of the window for each dimension of the input tensor.

strides: A list of ints that has length >= 4. The stride of the sliding window for each dimension of the input tensor.

padding: A string, either 'VALID' or 'SAME'. The padding algorithm. See the comment here

data_format: A string. 'NHWC' and 'NCHW' are supported.

name: Optional name for the operation.

Returns: A Tensor with type tf.float32. The max pooled output tensor.

子采样层如下：

```
pooled = tf.nn.max_pool(
                h,
                ksize=[1, sequence_length - filter_size + 1, 1, 1],
                strides=[1, 1, 1, 1],
                padding='VALID',
                name="pool")
```

sequence_length 是一个句子建立二维矩阵的高度（即 max_document_length）；sequence_length - filter_size + 1 就是卷积操作后的特征映射的高度。因此 pooling 操作后得到的 pooled 是一个 tensor，它的 shape=[batch_size,1,1, 滤波器个数]。

因为一个输入的在卷积+pooling 操作后的所有结果要拼接到一个向量，因此在程序的设计上用一个循环语句对所有 size 的滤波器迭代的分别进行卷积+pooling 操作，并将操作的结果 pooled 添加到一个 list 结构 pooled_outputs 中。

```
pooled_outputs = []
for i, filter_size in enumerate(filter_sizes):
    # Convolution Layer
    filter_shape = [filter_size, embedding_dim, 1, num_filters]
    W = tf.Variable(tf.truncated_normal(filter_shape, stddev=0.1), name="W")
    b = tf.Variable(tf.constant(0.1, shape=[num_filters]), name="b")
    conv = tf.nn.conv2d(
        embedded_chars_expanded,
        W,
        strides=[1, 1, 1, 1],
        padding="VALID",
        name="conv")

    # Apply nonlinearity
    h = tf.nn.relu(tf.nn.bias_add(conv, b), name="relu")

    # Maxpooling over the outputs
    pooled = tf.nn.max_pool(
        h,
        ksize=[1, sequence_length - filter_size + 1, 1, 1],
        strides=[1, 1, 1, 1],
        padding='VALID',
        name="pool")
    pooled_outputs.append(pooled)
```

进一步然后将 pooled_outputs 转换成一个扁平的结构

```
num_filters_total = num_filters * len(filter_sizes)
h_pool = tf.concat(3, pooled_outputs)
h_pool_flat = tf.reshape(h_pool, [-1, num_filters_total])
```

5．Dropout

```
h_drop = tf.nn.dropout(h_pool_flat, dropout_keep_prob)
```

6. 输出层+sofmax

```
W_output_shape=[num_filters_total, num_classes]
```

```
W_output = tf.Variable(tf.truncated_normal(W_output_shape, stddev=0.1))
b_output = tf.Variable(tf.constant(0.1, shape=[num_classes]))
```

pooling层和输出层是全连接，因此权重参数的shape是[num_filters_total, num_classes] 计算输出，因为是二分类所以最终的输出层需要先进行 softmax 操作，然后进行 logit 变换，将输出转换成 0 或 1 的值。建立的交叉熵损失函数如下：

```
scores = tf.nn.xw_plus_b(h_drop, W_output, b_output, name="scores")
losses = tf.nn.softmax_cross_entropy_with_logits(scores, input_y)
```

上面的函数 xw_plus_b 完成矩阵相乘加上偏置的操作；
softmax_cross_entropy_with_logits 将 softmax, logit 变换，计算交叉熵融合到一个函数里进行操作了。

```
l2_loss = tf.nn.l2_loss(W_output)
l2_loss += tf.nn.l2_loss(b_output)
loss = tf.reduce_mean(losses) + l2_reg_lambda * l2_loss
```

在机器学习中，一个防止过拟合的技巧是加上 regularization，翻译做“正则化”。例如，上面对参数 W_output 求 L2 范数，并将结果加入到损失函数中。

7.计算模型的准确率

这里定义一个 predictions 操作，是为了检验模型的准确率。通过比较模型计算的输出和训练集的真实标签计算模型准确率。

```
predictions = tf.argmax(scores, 1, name="predictions")
correct_predictions = tf.equal(predictions, tf.argmax(input_y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_predictions, "float"), name="accuracy")
```

8.训练操作

定义优化器，即定义训练操作

```
optimizer = tf.train.AdamOptimizer(0.01)
grads_and_vars = optimizer.compute_gradients(loss)
global_step = tf.Variable(0, name="global_step", trainable=False)
train_op = optimizer.apply_gradients(grads_and_vars, global_step=global_step)
```

这里的优化模型的学习率设为了 0.01。用户可以自己根据需要调节。

Global_step 是一个全局变量统计模型被训练的次数。因为该变量的不需要作为模型的参数去学习，而只是完成一个计数功能作为全局变量，因此创建它的 varible 时，设置 trainable=False。当把该全局变量作为参数传递给优化器进行模型训练时，该全局变量的值会被自动更新。

9.模型训练

在 data_helper.py 产生一个迭代器对象，用户可以用该迭代器对象，一次获得一个 batch 的训练数据。

```
batches = data_helpers.batch_iter(list(zip(x_train, y_train)), batch_size, num_epochs)
```

在方法 batch_iter 中，num_epochs 参数可以理解为训练数据要使用多少次来训练模型（即）；batch_size 参数即 minbatch GSD 方法中，一个批次数据的大小。因此，模型实际的迭代次数是 num_epochs*batch_size 次。

最终模型训练的代码如下：

```
for batch in batches:
    x_batch, y_batch = zip(*batch)
    feed_dict = {
        input_x: x_batch,
        input_y: y_batch,
        dropout_keep_prob: keep_prob
    }
    _, step, summaries, loss_val,acc_val = sess.run(
            [train_op, global_step, loss_summary, loss, accuracy],
            feed_dict)
    print("step {}, loss {:g}, acc {:g}".format(step, loss_val, acc_val))
    current_step = tf.train.global_step(sess, global_step)
```

第七章：循环神经网络

参见：（1）http://colah.github.io/posts/2015-08-Understanding-LSTMs/

（2）http://karpathy.github.io/2015/05/21/rnn-effectiveness/

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

RNN 是一类神经网络，它的 cell 之间的连接形成另一个有向环。RNN 创建了一个网络内部状态，允许展示动态时态行为。不像前馈神经网络，RNN 可以使用内部记忆来处理任意输入序列。如此，RNN 可以应用在建立序列模型的任务。

第一节：RNN 结构

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

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

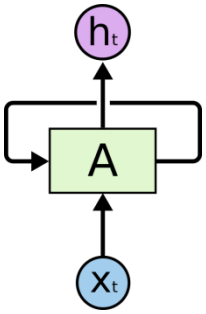


图 7.1 有循环的 RNN

图 7.1 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，如它可以是一个前馈神经网络。

这些循环（loop）使得 recurrent neural networks 看起来有些神秘。然而，RNN 结构上并不是和传统神经网络有完全的差异。RNN 可以看做是同一个网络的多次复制，每一次传递一个信息到循环中的下一个网络。我们展开这个 RNN。红框指示的是一个 time step。

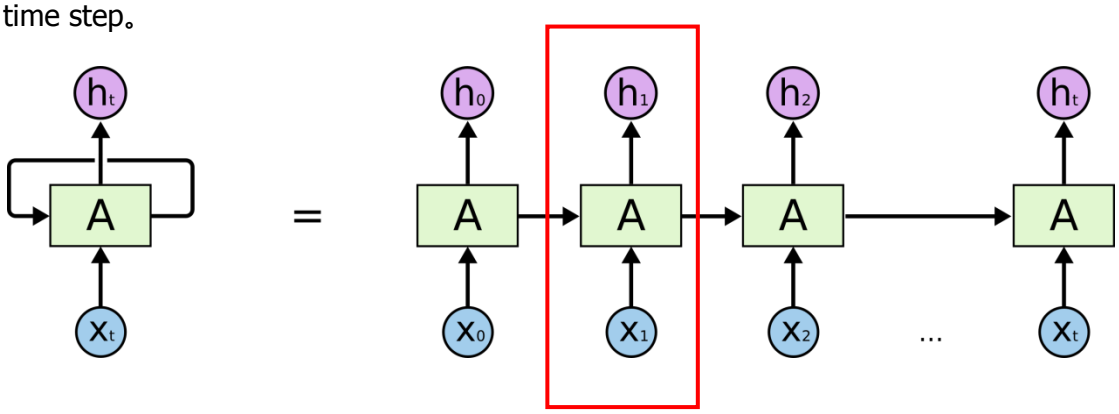


图 7.2 一个展开的 RNN

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

RNN 有很大的灵活性。有很多形式的 RNN。如图 7.3 所示。

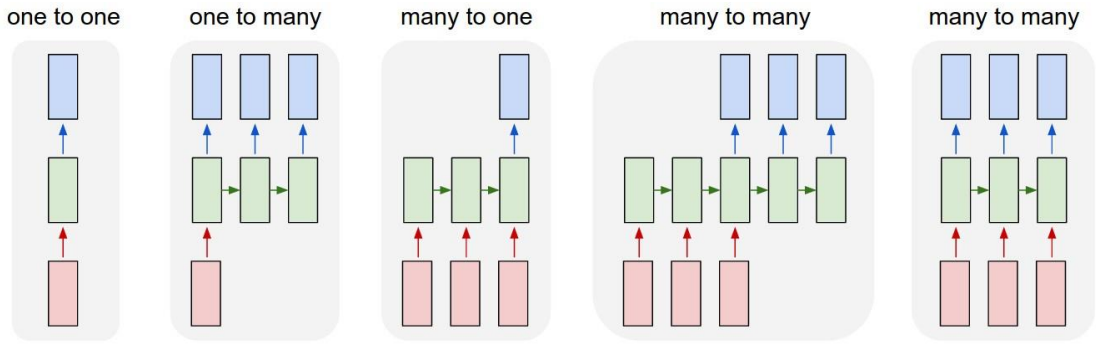


图 7.3 RNN 的各种结构

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 transformantion（绿色部分）是固定的，能按照我们的要求应用多次。

下面我们看一个基本 RNN 的例子，如图 7.4 所示。我们将用它建立一个字符级的语言模型。模型实现的代码参看 https://gist.github.com/karpathy/d4dee566867f8291f086。或我的学习资料中的 min-char-rnn.py

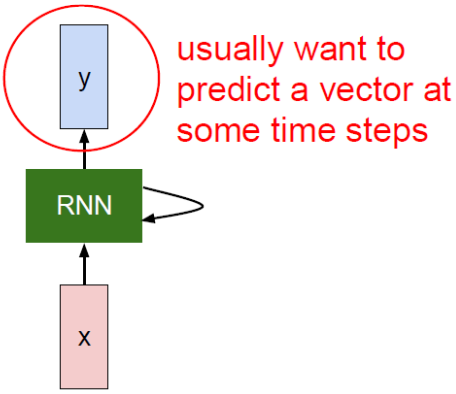


图 7.4 一个循环未展开的基本 RNN 结构

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

假设当前字符表只有四个字符"h,e,l,o"。图 7.5 是一个图 7.4 展开后的 RNN 示例。输入和输出层的维度为 4（因为字符表只有四个字符）；隐层有三个神经元。该图显示当 RNN 被“喂”字符“hell”作为输入，前向传递被激活。输出层包含 RNN 分配下一个字符（从字符表中选）的确信程度。绿色数值是高值，红色数值是低值。每个 time step 的隐层之间有个状态的传递。

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

具体实施中，每个字母采用 one-hot 编码。一次“喂给”RNN 一个字母。观察一个输出序列（维度为 4 的向量，每个维度一个字符）。可以将输出解释为 RNN 当前分配下一个到来的字符的确信程度。如输出<1.0， 2.2， -3.0， 4.1>对应输出字符是 helo 的确信程度。因为 2.2 最高，因此输出的是字符是“e”。

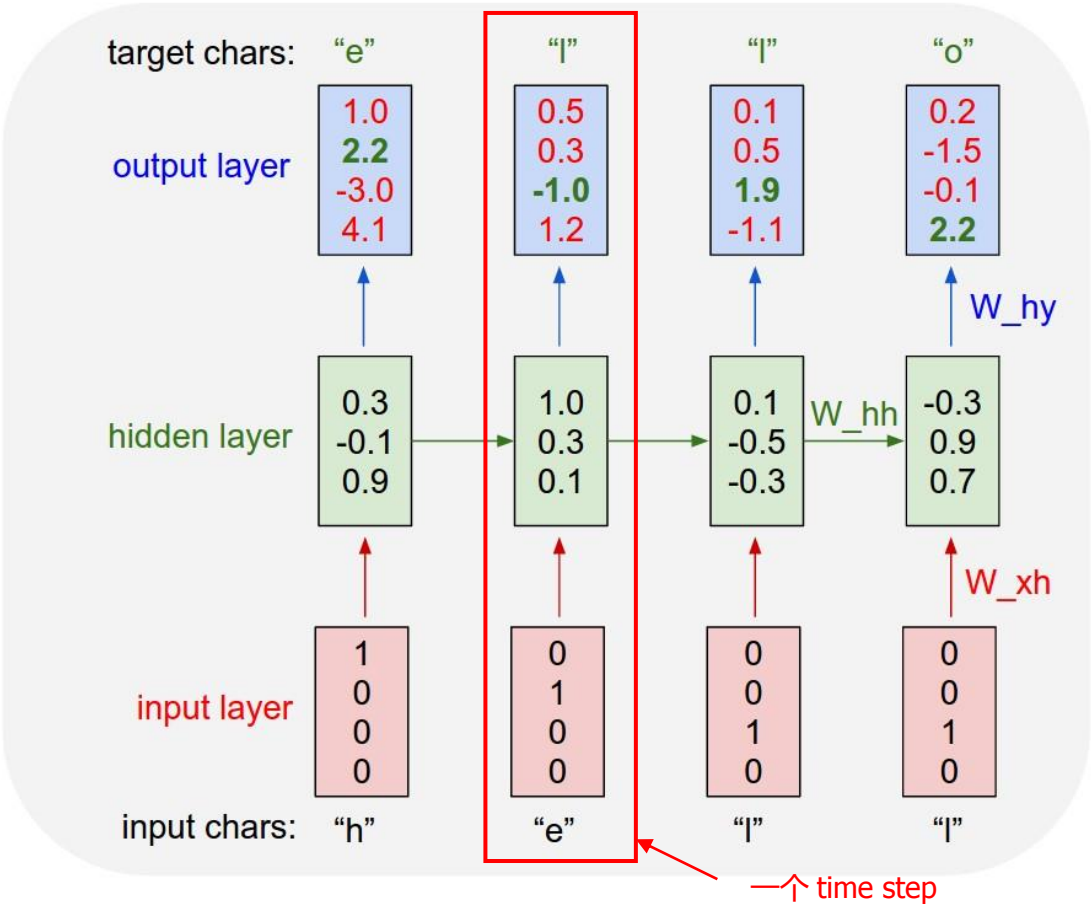
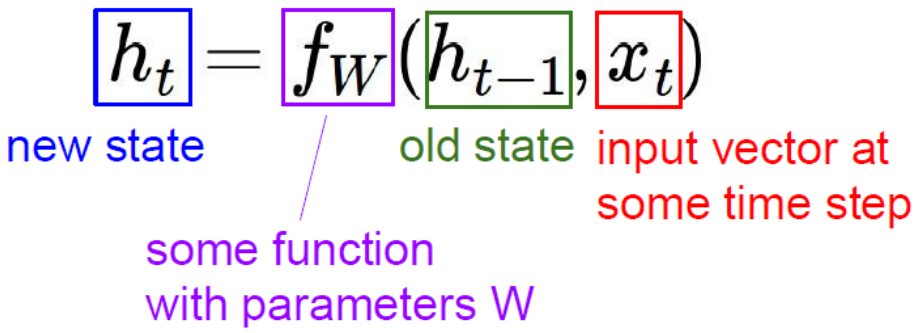


图 7.5 展开的 RNN 实例

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。

RNN 的计算过程如下：

（1）隐层的输出：（注意，和图 7.1,7.2 不同这里用符号 h 表示隐层的输出，y 表示 RNN 的输出）h_t是时间步为 t 时的隐层输出。它是根据上一个状态（时间步 t-1 的隐层输出）和当前的输入 x_t来计算的。$h_t = f_W(h_{t-1}, x_t)$



注意在每个“时间步”使用的是同样的函数和参数。当隐层的激活函数是 tanh

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

因此计算时有个循环过程，循环 time steps 次数，每次用上一个时间步的状态进行计算。

（2）输出层的输出：

$$y_t = W_{hy}h_t$$

Tips:
（1） 基本 RNN 中输入的序列长度是可变的。之所以可变，关键就是每个时间步使用同样的函数和参数
（2） 理论上 RNN 可以处理任意长度的序列，实践中还是设定了序列的长度。即规定了时间步的步数。

设计 RNN 时几个重要的参数需要确定。输入序列的长度；序列是一组向量，设定一个向量的维度；隐层神经元的个数；输入到隐层的权重 W_xh 是一个 tensor，它的 shape;从状态 h_{t-1} 到传统 h_t 传递时的权重 tensorW_hh 的 shape；输出层的神经元数目；从隐层到输出层的权重 tensor 的 shape。

图 7.5 中的一个时间步，我们可以按照一个前馈神经网络来理解。

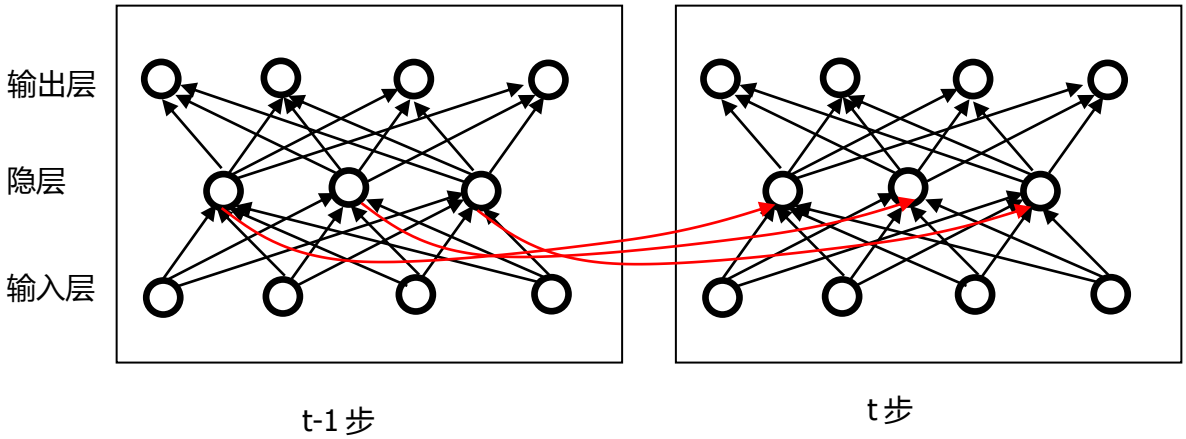


图 7.6 RNN 一个时间步

这里的隐层（即 cell）可以是多层。这就是我为什么说一个基本的 RNN 的 cell 是一个前馈神经网络。图 7.6 中隐层只是一层。如果是多层，t-1 步隐层的每一层都会参与到 t 步隐层的对应每一层的计算。

min-char-rnn.py 中的 rnn 实施过程概要介绍：

（1）预处理：

一个文本文件作为训练集，将该文本文件转换成一个长的字符串。然后统计 unique char 作为字符表，统计该字符串的长度。

```
data = open('t1.txt', 'r').read()
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
```

然后对字符编码

```
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }
```

（2）建立权重和偏置

```
Wxh = np.random.randn(hidden_size, vocab_size)*0.01
Whh = np.random.randn(hidden_size, hidden_size)*0.01
Why = np.random.randn(vocab_size, hidden_size)*0.01
bh = np.zeros((hidden_size, 1))
by = np.zeros((vocab_size, 1))
```

（3）训练集的产生

将长串按照 seq_length，即 time step 的步数，切分。假设 seq_length=3

Helloworld

可以产生训练集：

输入：输出

Hel：ell

low：owo

orl:rld

（4）模型的训练。定义了一个 lossFun 函数。lossFun 的输入，就是上面的一条训练数据，数据的标签和状态值

```
def lossFun(inputs, targets, hprev):
  for t in xrange(len(inputs)):
    xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
    loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
  # backward pass: compute gradients going backwards
  dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
  dbh, dby = np.zeros_like(bh), np.zeros_like(by)
  dhnext = np.zeros_like(hs[0])
```

```
  for t in reversed(xrange(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1 # backprop into y. see http://cs231n.github.io/neural-networks-case-study/#grad if confused here
    dWhy += np.dot(dy, hs[t].T)
    dby += dy
    dh = np.dot(Why.T, dy) + dhnext # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)
  for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
    np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
  return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
```

因为训练集中一条数据的长度已经设定为了 seq_length。我们可以看到循环了 seq_length 次。在第 t 趟循环将输入的字符要 one-hot 编码。然后按照上面讲的方式根据上一趟循环（上一个 time step）的装 hs[t-1]和输入 xs[t]，计算状态值 hs[t]，输出值 ys[t]。再在每个预测的结果上计算损失函数。该函数将最后计算的状态值，也作为了结果返回。

然后根据损失函数反向传播调整参数值。

（5）训练

```
while True:
  if p+seq_length+1 >= len(data) or n == 0:
    hprev = np.zeros((hidden_size,1)) # reset RNN memory
    p = 0 # go from start of data
  inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
  targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]

  loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
```

可以看到当前一条数据训练产生的状态值也作为了下一条数据训练的初始参数。直到当前训练集的数据训练完了，重新初始化状态值。这样的使用训练集进行的一趟训练在 TensorFlow 语境中称为一个 epoch。

我们可以理解到，其实该方法是将整个字符串带入到了 RNN 中进行计算。Time step 等于字符串的长度。

第二节：使用 TensorFlow 构建 RNN

使用 TensorFlow 构建 RNN，实际上是使用一个 cell 函数和 rnn 函数把隐层给实现。图 7.5 的输出和输出部分还是要自己实现。

Tensorflow 中构建 RNN 包括两个基本的元素：cell 和 rnn(参考 https://tensorflow.google.cn/versions/r1.9/api_guides/python/contrib.rnn)

Cell 的种类包括:

(1) tf.contrib.rnn.BasicRNNCell（或者 tf.nn.rnn.BasicRNNCell）

基本的 RNN Cell, 构建一个前馈神经网络的一层。参数如下：

. **num_units:** int, cell（隐层）前馈神经网络的神经元数.

. **activation:** 输出的激活函数，需要非线性激活函数。默认是 tanh.

. **reuse:** (optional) Python boolean describing whether to reuse variables in an existing scope. If not True, and the existing scope already has the given variables, an error is raised.

. **name:** String, the name of the layer. Layers with the same name will share weights, but to avoid mistakes we require reuse=True in such cases.

. **dtype:** Default dtype of the layer (default of None means use the type of the first input). Required when build is called before call.

TensorFlow 可以构建多种 RNN。包括基本 RNN: tf.nn. rnn，动态 RNN: tf.nn.dynamic_rnn，双向 RNN：tf.nn.bidirectional_ rnn，双向动态 RNN：tf.nn.bidirectional_dynamic_rnn。这里只介绍基本 RNN 的构建函数。

(2) tf.contrib.rnn.BasicLSTMCell 详见 7.5 节

(3) tf.contrib.rnn.GRUCell（或者 tf.nn.rnn_cell.GRUCell）

这个函数实现 Gated Recurrent Unit cell（参考论文 http://arxiv.org/abs/1406.1078）。参数如下：

num_units: int, The number of units in the GRU cell.

activation: Nonlinearity to use. Default: tanh.

reuse: (optional) Python boolean describing whether to reuse variables in an existing scope. If not True, and the existing scope already has the given variables, an error is raised.

kernel_initializer: (optional) The initializer to use for the weight and projection matrices.

bias_initializer: (optional) The initializer to use for the bias.

name: String, the name of the layer. Layers with the same name will share weights, but to avoid mistakes we require reuse=True in such cases.

dtype: Default dtype of the layer (default of None means use the type of the first input). Required when build is called before call.

1. rnn 函数

tf.nn.rnn(cell, inputs, initial_state=None, dtype=None, sequence_length=None, scope=None)

该函数创建一个由参数 cell 设定的 RNN。需要用户提供初始状态 initial_state。如果提供了序列长度向量 sequence_length，会进行动态计算。即，它不计算所有的 steps，而是计算完设定的长度后，传递计算结果到最终输出。将节约计算时间。可参看 7.1 节的例子，它是将整个文本作为一个长串，带入 RNN，RNN 的 time step 是字符串的长度。

如果提供了 equence_length vector，则进行动态计算. This method of calculation does not compute the RNN steps past the maximum sequence length of the minibatch (thus saving computational time), and properly propagates the state at an example's sequence length to the final state output.The dynamic calculation performed is, at time t for batch row b, python (output, state)(b, t) = (t >= sequence_length(b)) ? (zeros(cell.output_size), states(b, sequence_length(b) - 1)) : cell(input(b, t), state(b, t - 1))

参数：

cell: 一个 RNNCell 实例

inputs: Inputs 是一个 list，list 的长度和 time steps 一样。list 中的每个元素是一个 tensor。Tensor 的 shape = [batch_size, input_size]。（注： input_size 应该是隐层的神经元个数。这里的输入是隐层的输入而不是模型的输入。）

initial_state: (可选) RNN 的初始状态。如果 cell.state_size 是一个整数, initial_state 必须是一个 Tensor 它的 shape=[batch_size, cell.state_size]. 如果 cell.state_size 是 tuple, initial_state 应该是一个由 tensor 构成的 tuple 它的 shapes 是[batch_size, s]，s =cell.state_size。简单的说，initial state 的输入是一个 tensor，它的 shape=[batch_size, num]。num 值由使用的 RNN 类型（cell 的类型）确定，如，BasicRNNCell 或 GNRCell，num 值就是隐层的神经元数，如果是 BasicLSTMCell，num=2*隐层的神经元数。

sequence_length: 设定输入中每个序列的长度。是一个 int32 or int64 vector (tensor) size [batch_size], values in [0, T).

scope: VariableScope for the created subgraph; defaults to "RNN".

返回值

返回一对 (outputs, state) 其中 Outputs 是一个 list，长度为 time steps。每个元素是一个 tensor，它的 shape=[batch_size, 隐层的神经元数]。State 是最终的状态。

注：rnn 函数到底有多少个 time step 是由输入数据决定的。rnn 函数本身没有设置这一参数。

2. cell 函数

RNN 的 Cell 即图 7.1 中指示的块 A。TensorFlow 提供 Cell 函数来设定了 RNN 的类型。tf.nn.rnn_cell 提供一个接口访问所有类型的 RNN 的 cell。

tf.nn.rnn_cell.BasicRNNCell

是最基本的 RNN cell。它的必须的一个参数是 num_units：隐层神经元的个数。该函数创建一个 cell 对象，将作为 rnn 函数的参数。BasicRNNCell 就是 7.1 节介绍的 RNN。

tf.nn.rnn_cell.BasicLSTMCell

LSTM 是一种 RNN，详细内容下一节将介绍。此函数是一个基本实施。它的重要参数包括 num_units：隐层神经元的个数；forget_bias 加到 forget gate 的偏置，默认值是 1.0；activation: 内部状态激活函数，默认是 tanh。

tf.nn.rnn_cell.GRUCell

Gated Reccurent Unit 神经网络。它的重要参数包括 num_units：隐层神经元的个数；activation: 内部状态激活函数，默认是 tanh。

tf.nn.rnn_cell.LSTMCell

LSTM 的另一个实施版本，称作 The peephole implementation。
https://research.google.com/pubs/archive/43905.pdf

该类使用了 optional peep-hole connections， optional cell clipping, 和 an optional projection layer。

3. 建立可以识别手写数字的 rnn 模型

本节我们将用 RNN 构建一个数字识别模型（rnn4mnist_basic.py）。不是说，RNN 构建的模型在识别手写数字时性能会更好。我们是想通过该模型理解怎样用 TensorFlow 构建 RNN。（大家可以比较使用不同类型的 RNN 获得的性能如何）

（1）参数设置

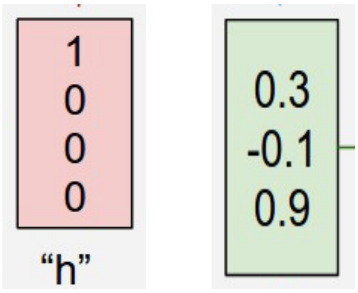
```
learning_rate = 0.001
training_iters = 100000
batch_size = 128
display_step = 10
```

learning_rate 是优化函数的学习率；training_iters 是训练的迭代次数；batch_size 是 minibatch 学习中选取的批量数据的大小；display_step 是每隔这些学习步骤显示学习结果。

（2）RNN 网络的参数

```
n_input = 28
n_steps = 28
n_hidden = 128
n_classes = 10
```

mnist 数据集是灰度图片数据集。一张图片的大小是 28*28。当需要把一张图片带入 RNN 模型时，把一行作为输入的向量；而每行是一个 time step。**注：我们讨论的输入、隐层、输出层都是指在一个 time step 的范围**。因此 n_input 是一个输入向量的长度 =28； n_hidden 是隐层的神经元个数；n_classes 是输出层的神经元个数；n_steps 是 rnn 的 time steps=28。图 7.5 中的 rnn 输入的一个输入向量如下图。



(a) (b)

图 7.7（a）rnn 的输入；（b）rnn 的隐层

（3）定义 placeholder。

```
x = tf.placeholder("float", [None, n_steps, n_input])
istate = tf.placeholder("float", [None, 2*n_hidden])
y = tf.placeholder("float", [None, n_classes])
```

创建 x 和 y 占位符，在每趟训练时，将批量训练数据“喂给”模型训练数据和标签。rnn 隐层需要的“state”需要初始化。因为希望在每趟训练时使用不同的随机初始的值。因此也采用占位符的方式“喂给”模型。

x = tf.placeholder("float", [None, n_steps, n_input])表示创建的 x 占位符接受的训练数据的 shape=[None, n_steps, n_input]。None 表示训练数据的批量大小，不预设；训练数据是图片，图片的大小= n_steps* n_input。

istate = tf.placeholder("float", [None, n_hidden])。None 是 batch_size。第二个参数和 RNN 的类型（Cell 的类型）有关。如果是 BasicRNNCell 或 GNRCell，它等于隐层的神经元数目；如果是 LSTMCell，他是 2*隐层的神经元数目。

（4）定义 Variable。前面已经讲过。TensorFlow 使用 tf.nn.rnn 和 tf.nn.rnn_cell 实施的 RNN，相当于把图 7.5 中隐层部分实施了。用户仍需要实施其他部分。即仍需要创建两个权重 W_xh 和 W_hy。

```
weights = {
    'hidden': tf.Variable(tf.random_normal([n_input, n_hidden])), # Hidden layer weights
    'out': tf.Variable(tf.random_normal([n_hidden, n_classes]))
}
biases = {
    'hidden': tf.Variable(tf.random_normal([n_hidden])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

图 7.5 的 W_xh 权重这里定义为一个名为 weights 的 dictionary 结构中的一个“键：值”对，键的名称为“hidden”。W_hy 也是一个“键：值”对，键的名称为“out”。

这些权重是待学习的参数，因此定义为 Variable。W_xh 的 shape=[n_input, n_hidden]。这是因为，该矩阵参与运算$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$。$x_t$是输入的行向量，$x_t W_{xh}$的运算结果是一个列向量，长度等于隐层的神经元数。

W_hy 的权重是[n_hidden, n_classes]。

注：所有的 time steps 共享 W_xh 和 W_hy 权重。

（5）构建 RNN：创建一个函数完成创建 RNN。

```
def RNN(_X, _istate, _weights, _biases):
    _X = tf.transpose(_X, [1, 0, 2])
    _X = tf.reshape(_X, [-1, n_input])
    _X = tf.matmul(_X, _weights['hidden']) +  _biases['hidden']

    cell = tf.nn.rnn_cell.BasicRNNCell(n_hidden)
    _X = tf.split(0, n_steps, _X)
    outputs, states = tf.nn.rnn(cell, _X, initial_state=_istate)
    return tf.matmul(outputs[-1], _weights['out']) + _biases['out']
```

该函数中将输入数据进行了变化。图 7.8 描述把 batch_size=3 的一个批量的图片的转换过程。_X = tf.transpose(_X, [1, 0, 2])，此时的参数_X 是整个模型的输入。输出_X 是经过转置后的 tensor。它的 shape=[n_steps, batch_size, n_inputs]。_X = tf.reshape(_X, [-1, n_input])继续将_x 转换成一个 tensor，它的 shape=[n_steps*batch_size, n_inputs]。

_X = tf.matmul(_X, _weights['hidden']) + _biases['hidden']将转换后的_X 和权重相乘。得到的 tensor 的 shape=[n_steps*batch_size, n_classes]。

因为 tf.nn.rnn 需要的 input 是一个 list，list 长度为 time steps，list 每个元素是一个 tensor，它的 shape=[batch_size, n_hidden]。_X = tf.split(0, n_steps, _X))继续将_X，进行分割。在维度 0 上按照 n_steps 进行分割。结果如图 7.8 所示。

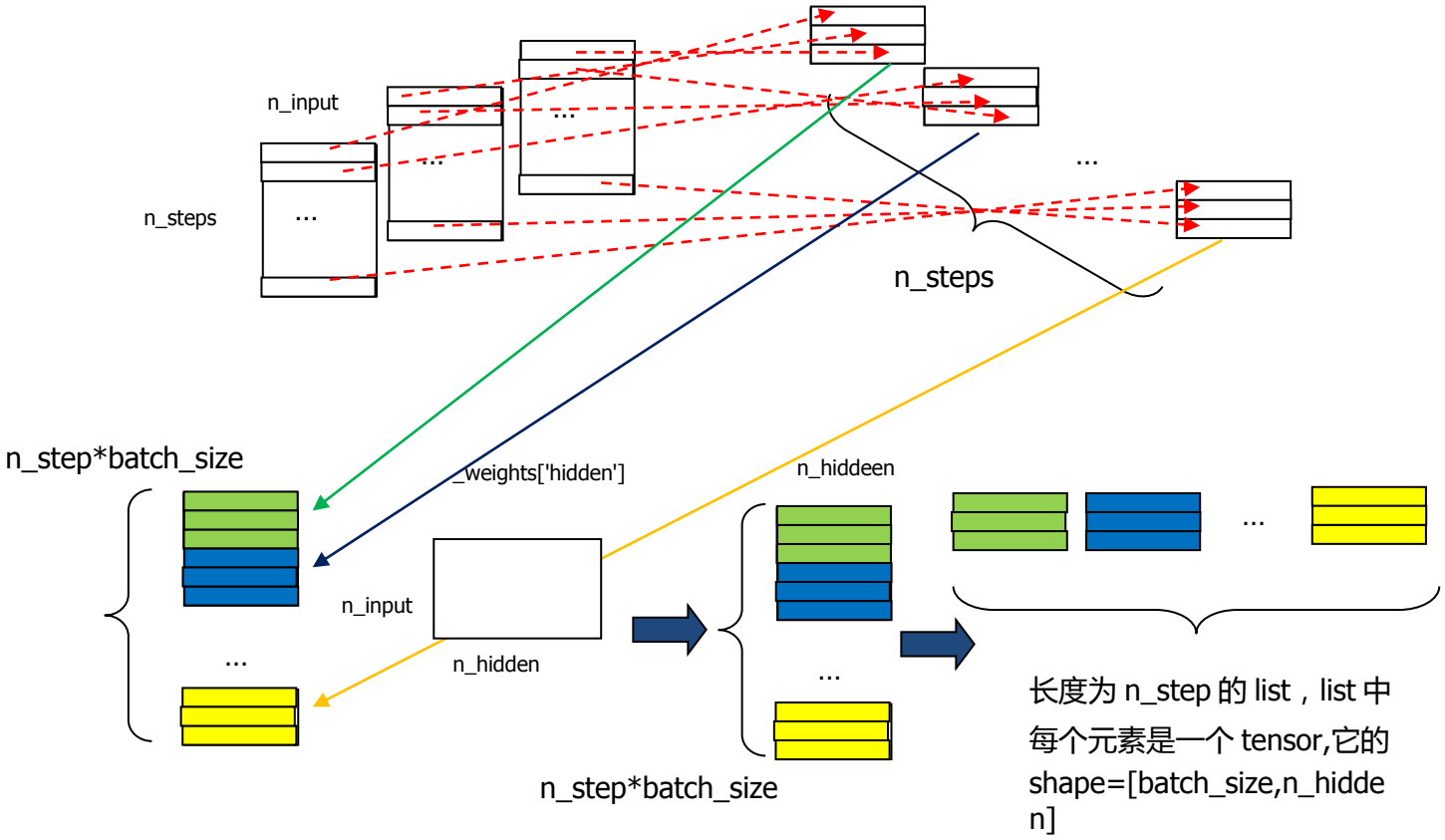


图 7.8 输入图片转换到 rnn 函数需要的格式的过程

输入图片 tensor 的 shape 是[batch_size, n_steps, n_input]。权重 W_xh 的 shape=[n_input，n_hidden]。因此需要把输入的图片 tensor 进行转换_X = tf.reshape(_X, [-1, n_input])。如此得到的_X 的 shape=[n_steps*batch_size, n_input]。再将输入和权重 W_xh 进行矩阵乘加上偏置进行计算

```
tf.matmul(outputs[-1], _weights['out']) + _biases['out']
```

得到的 tensor 的 shape=[n_steps*batch_size, n_classes]。

创建 RNN Cell

```
cell = tf.nn.rnn_cell.BasicRNNCell(n_hidden)
```

因为 rnn 函数的输入需要的是一个 list，list 中的每个元素是一个 tensor。Tensor 的 shape = [batch_size, n_hidden]。因此需要进行划分

```
_X = tf.split(0, n_steps, _X)
```

然后进行隐层的计算

```
outputs, states = tf.nn.rnn(cell, _X, initial_state=_istate)
```

outputs 是隐层的计算结果。Outputs 是一个 list，长度为 time steps。每个元素是一个 tensor。一个 tensor 的 shape=[batch_size, 隐层的神经元数]。outputs[-1]是获取最后一个 time step。将它作为模型的输出。在输出层进行计算时将 outputs[-1]乘上权重 W_hy 得到输出，它是一个 tensor，它的 shape=[batch_size, n_classes]

```
tf.matmul(outputs[-1], _weights['out']) + _biases['out']
```

调用定义的 RNN 函数得到预测结果 pred。

```
pred = RNN(x, istate, weights, biases)
```

（6）迭代训练

```
while step * batch_size < training_iters:
    batch_xs, batch_ys = mnist.train.next_batch(batch_size)
    # Reshape data to get 28 seq of 28 elements
    batch_xs = batch_xs.reshape((batch_size, n_steps, n_input))
    # Fit training using batch data
    sess.run(optimizer, feed_dict={x: batch_xs, y: batch_ys,
                                   istate: np.zeros((batch_size, n_hidden))})

        ......
```

可以看到在每次迭代中，产生一个批次的数据，带入模型。而每个批次中的状态是被初始化为 0。

比较一下 7.1 节对文本的处理。可以看到处理文本和处理图片时的差别。7.1 节是将一个文本文件作为训练集。将这一个文本转换成长串，然后切分成 time step 长度的子串但每次迭代时，使用上一次迭代的 RNN 最后的状态，作为下一次迭代时 RNN 的输入状态。因此实际上是整个长串作为了一个输入，即 RNN 的 time step 是整个长串的字符数。

而本节处理图片时，每个图片和下一个图片没有关联，因此将图片转换成 time step 个向量后，RNN 的 time step 之间存在状态传递。

（7）定义损失函数和优化器

```
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(pred, y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
```

练习：参考上面的 mnist 数据集的例子，将前面的字符级语言模型用 TensorFlow 实现了。

第三节：LSTM

RRN 的一个吸引人的地方在于它可以连接先前的信息到当前的任务，例如使用先前的视频帧帮助当前帧的理解。有时我们仅仅需要最近的信息，而不是太早以前的信息完成当前的任务。例如，一个语言模型试图根据前面的词预测下一个词。如果我们预测一个句子“the clouds are in the *sky*”中的最后一个词。根据句子中前面的词集合（前面的词是当前的词 context）”the clouds are in the”很明显这个词应该是 sky。这个例子中，我们需要的 context 相关信息可以不是很多。这个问题称为 Short Term Dependencies。

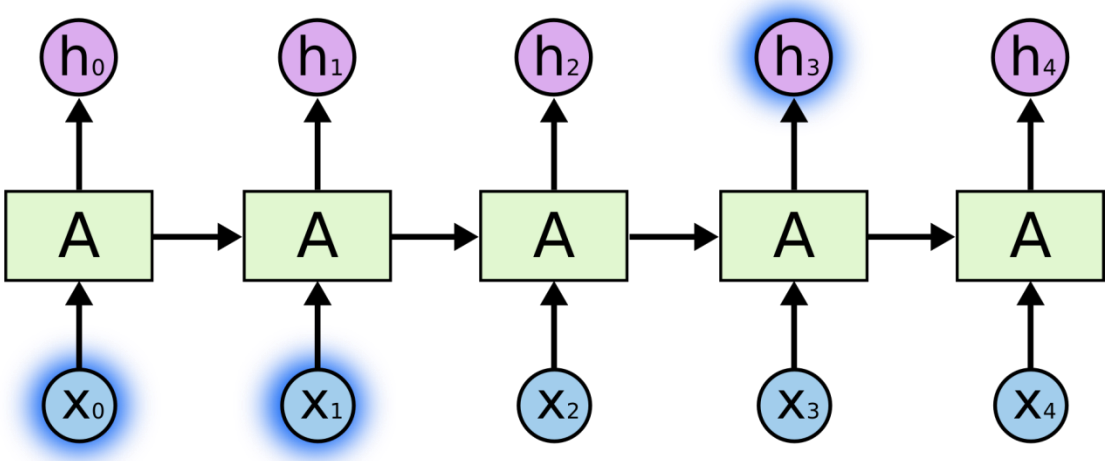


图 7.9. Short Term depencency

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。但如果我们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。

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 contex 去进行学习。这个问题称为 Long Term Dependencies.

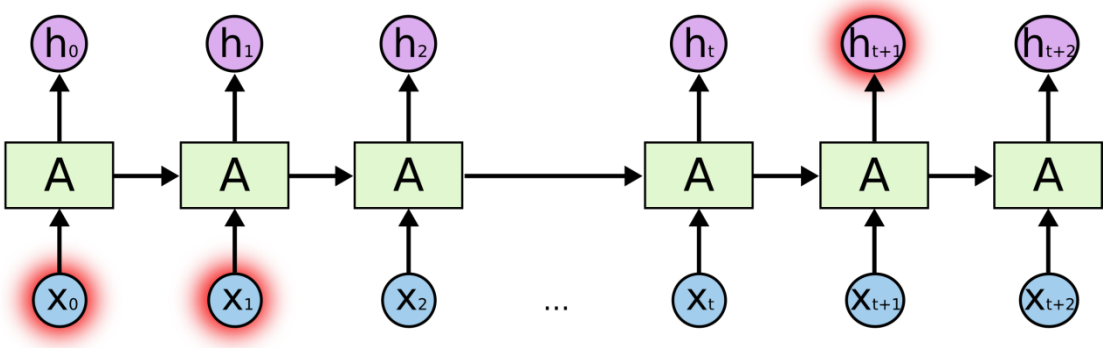


图 7.10. Long Term Dependecy

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**）。（7.1 节$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$）

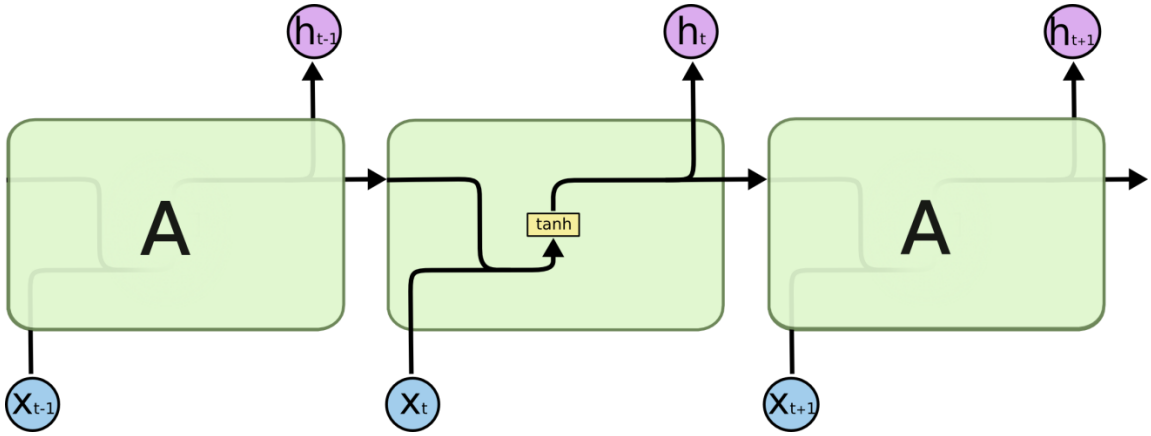


图 7.11. 在一个标准 RNN 中重复了一个单独的层

LSTM 也有这种链式结构，但是重复的块（chunk）有复杂的结构，例如有四个层，以一种特殊的形式交互。如图 7.5 所示。

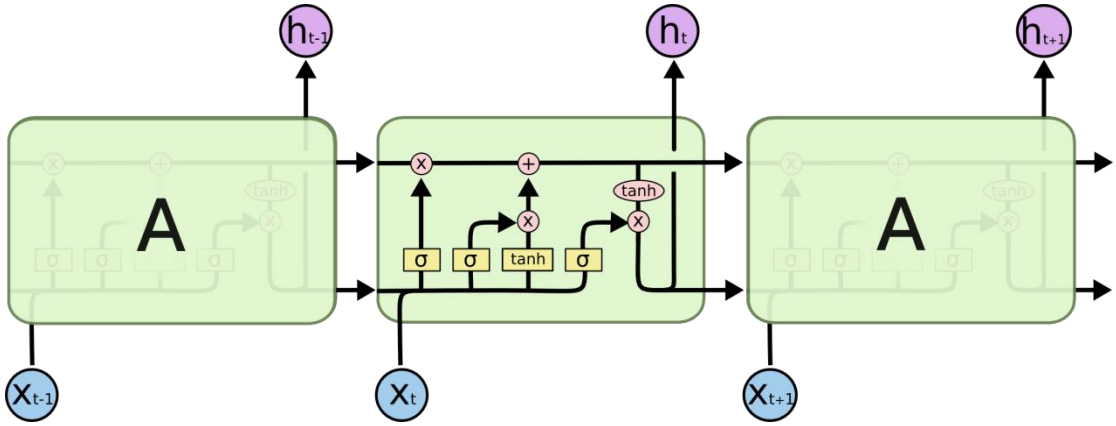


图 7.12 LSTM 中的块包含四个交互层

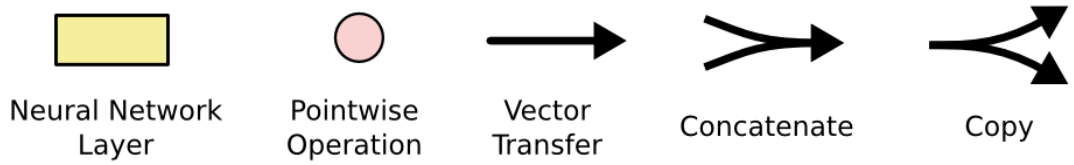


图 7.6 一些注释符号

图 7.6 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。

2. LSTM 的核心思想

与基本 RNN 相比，LSTM 的关键是增加了块（chunk 或 cell）状态，即贯穿图的那条水平线。块的状态（cell state）可以理解为是一种传送带。信息沿着它传递。

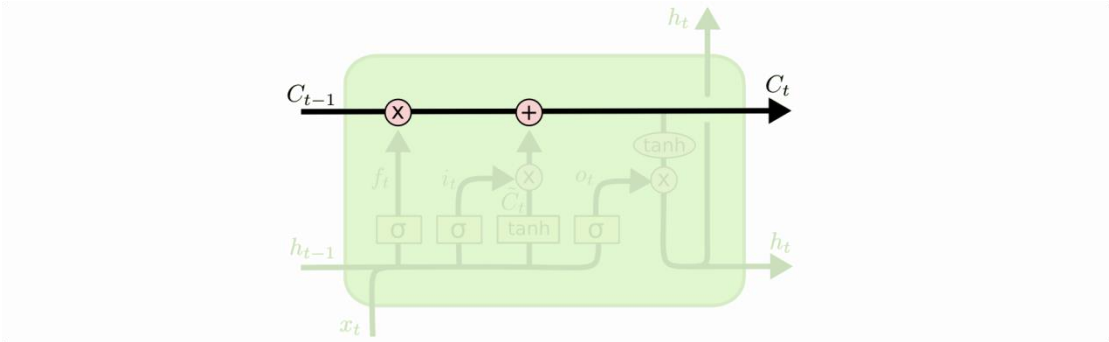


图 7.13 RNN 的信息传送

通过调整称为 gate 的结构，LSTM 有能力移除或添加信息给单元状态（cell state）。Gate 是一种方式或通道，选择性的让信息通过。它由一个 sigmoid 层和一个逐点相乘的运算操作组成。如图 7.14 所示。

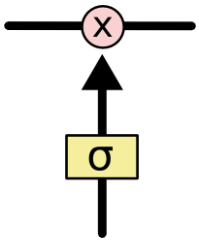


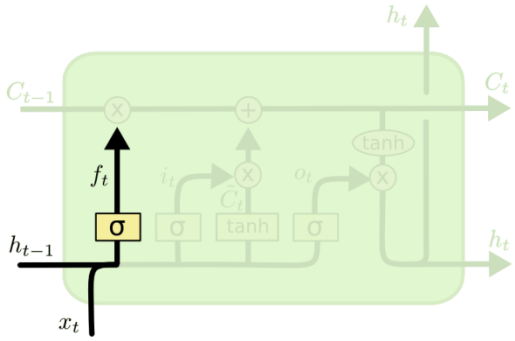
图 7.14 Gate 结构

Sigmoid 层输出 0~1 之间的数值，描述了每个构件（component）有多少部分允许通过，例如，0 表示不让通过，1 表示全部通过。一个 LSTM 有三种 gate（forget gate, input gate, output gate）来包含和控制 cell state。（图 7.12 中一个块里面的三个⊗）

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 sigmoid 层决定（图 7.15），称为 **forget gate**。Sigmoid 层的输入是 h_{t-1} 和 x_t。对于 cell state C_{t-1} 的每个值，forget gate 输出一个 0~1 之间的一个值。1 表示完全通过，0 表示阻止。

我们回到语言模型的例子。该语言模型试图基于前面的词预测下一个词。该任务中，cell state 可以包括当前主语的性别这样的信息，如此可以使用正确的介词。当看见一个新的主语，我们应该忘记上一个主语对应的性别。（我理解上面这个例子的意思是一个句子包含多个主语）

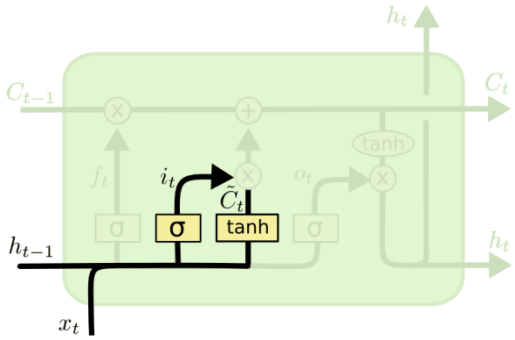


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

图 7.15 LSTM 块的第一层：一个 Sigmoid 层

[h_{t-1}, x_t]表示的是将 h_{t-1} 和 x_t 拼接操作到一个矩阵。权重矩阵 W_f 实际上包含 h_{t-1} 的权重 W_{fh} 和 x_t 的权重 W_{fx}。上面的操作 $W_f.[h_{t-1}, x_t]$ 可以分解成 $W_{fh}.h_{t-1}+W_{fx}.x_t$。

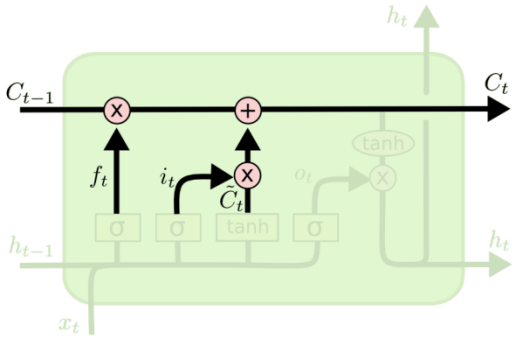
下一步是要确定我们将要存储什么新信息在 cell state 中。它包含两部分。首先一个 sigmoid 层称作“**input gate**”决定我们应该更新哪个值。下一步，一个 tanh 层创建一个新的候选值的向量$\tilde{C}_t$。它能被加到这个状态中。紧接着，联合这两个状态来创建一个新的状态。在语言模型的例子中，我们想加新主语的性别到 cell state，来替换旧的状态。



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

图 7.16 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 7.16）中确定的新的信息。

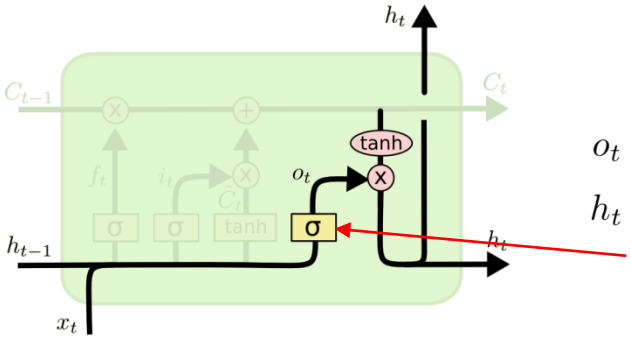


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

图 7.17 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 simgoid 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。



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

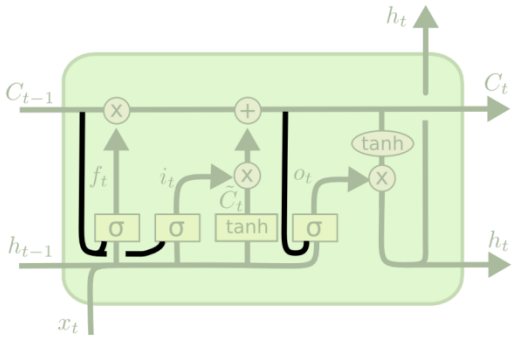
Sigmoid 层，担负 gate 功能。即 output gate。

图 7.18 LSTM 块的第四层

第四节：LSTM 的变体

迄今我们描述的 LSTM 是一个标准的 LSTM。但是不是所有 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 7.18Cell state C_{t-1} 参与了 forget gate f_t 的计算。



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_f\right)$$

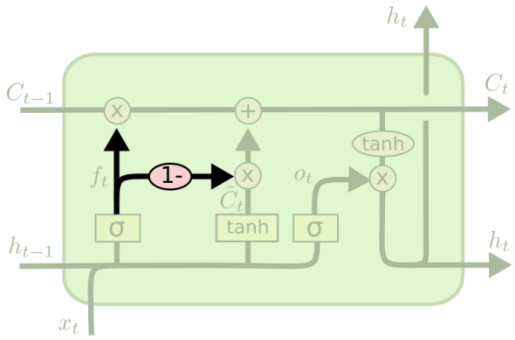
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_i\right)$$

$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] + b_o\right)$$

图 7.18 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。

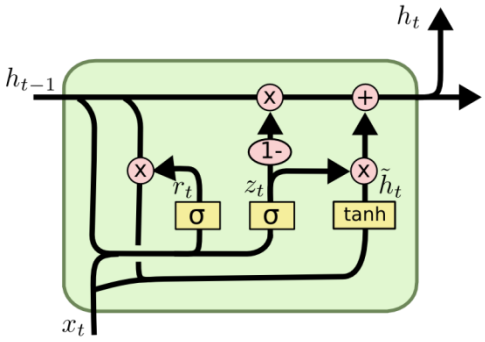
另一个 LSTM 变体将 forget gate 和 input gate 结合使用。它不像基本 LSTM 中单独决定什么应该被忘记（forget gate），应该加什么新信息（input gate）。该变体将两个决策一起决定。仅仅当要输入一些信息，那么相应位置的旧信息把它忘记，其他位置的信息不变。仅仅在旧的 cell state 中被忘记的部分输入新值。



$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

图 7.18 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，也非常受欢迎。



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

图 7.19 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。

RNN 研究发展方向

LSTM 在大部分任务上都工作的很好。LSTM 使得 RNN 向前发展了一大步。而下一个将使得 RNN 发展一大步的是 attention（翻译做注意力机制）。

其思想是让 RNN 的每一个 time step 挑选信息，以看得到其他的信息集合。例如，在使用 RNN 创建一个描述图片内容的短文（caption）任务中，让输出的每个 word 可以看见挑选的一部分图片。关于 attention 可以参考论文 Xu, *et al.* (2015)。

RNN 发展的另外的方向还包括 Grid LSTMs by Kalchbrenner, *et al.* (2015)。

还有一些工作在 generative models 中使用 RNN，例如 Gregor, *et al.* (2015), Chung, *et al.* (2015), or Bayer & Osendorfer (2015) 也是一个 RNN 未来的方向。

第五节：Tensorflow 构建 LSTM 语言模型

构建 LSTM 模型的步骤包括：构建 Cell，

构建 LSTMcell 的函数有很多

（1） tf.contrib.rnn.BasicLSTMCell（或者 tf.nn.rnn_cell.BasicLSTMCell）构建最基本的 LSTM Cell

（2） tf.contrib.rnn.LSTMBlockCell 实施了论文 http://arxiv.org/abs/1409.2329 的 LSTM

（3） tf.contrib.rnn.LSTMCell（或 tf.nn.rnn_cell.LSTMCell）可以构建更高级、复杂的 LSTM Cell。

我们这一节只使用基本 LSTM Cell。用 Tensorflow 构建 LSTM 模型时，先理解几个术语：

- num_layers - the number of LSTM layers
层数，图 7.6 是一层， 图 7.20 是二层

- num_steps - the number of unrolled steps of LSTM
时间步 time step 的步数，图 7.6 时间步为 4，图 7.20 为 5

- hidden_size - the number of LSTM units

LSTM 实际上是对输入的序列中的一个时间步，如 x_t，这个向量的每一位进行计算。每一位的计算都是由一个 unit 完成。图 7.11，7.19 描述的是 cell 对一个状态向量$h_t^l \in R^n$进行计算。实际上是对 n 维向量的每一位都是单独的一个 unit 进行计算。

注：我理解 celll 称为是对一个向量进行计算的隐层单元。Cell 是由多个 unit 组成。对向量的每一位进行计算的。每个 unit 的结构和图 7.11，,7.19 等图示的结构一样，只不过是对标量进行计算，计算结果也是标量

我们用图 7.20 来描述一个时间步的计算。输入 x^t 是一个向量。有一个全连接层和输入层相连。全连接层的神经元个数就是 hidden_size。全连接层的每个神经元的输出都和一个 unit 连接。这里的每个隐层是一个 cell，多个 cell 构成多层的隐层。时间步 t 每个 unit 的计算要使用时间步 t-1 对应的 unit 的状态$s_{t-1,n}^{L}$

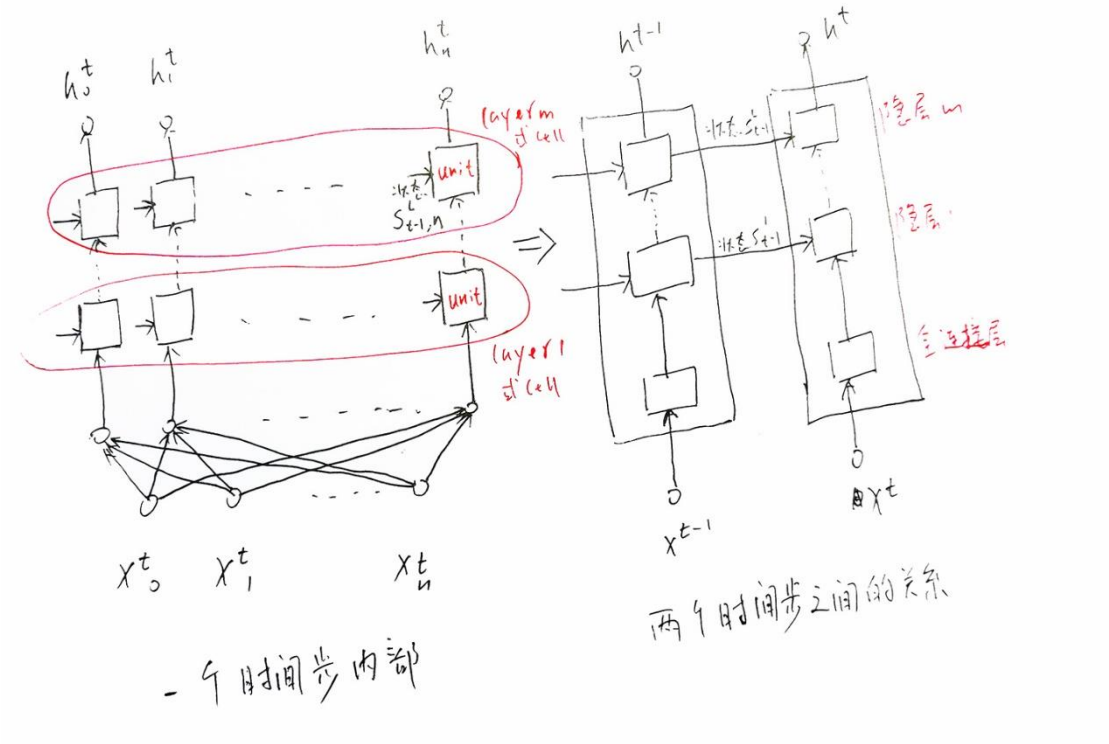


图 7.20 一个时间步的内部

基本的 LSTM RNN Cell（实施了论文 https://arxiv.org/pdf/1409.2329.pdf ）

tf.contrib.rnn.BasicLSTMCell 函数的参数如下：

num_units: int，nj 见上面的解释。

forget_bias: float, The bias added to forget gates (see above). Must set to 0.0 manually when restoring from CudnnLSTM-trained checkpoints.

state_is_tuple: If True, accepted and returned states are 2-tuples of the c_state and m_state. If False, they are concatenated along the column axis. The latter behavior will soon be deprecated.

activation: Activation function of the inner states. Default: tanh.

reuse: (optional) Python boolean describing whether to reuse variables in an existing scope. If not True, and the existing scope already has the given variables, an error is raised.

name: String, the name of the layer. Layers with the same name will share weights, but to avoid mistakes we require reuse=True in such cases.

dtype: Default dtype of the layer (default of None means use the type of the first input). Required when build is called before call.

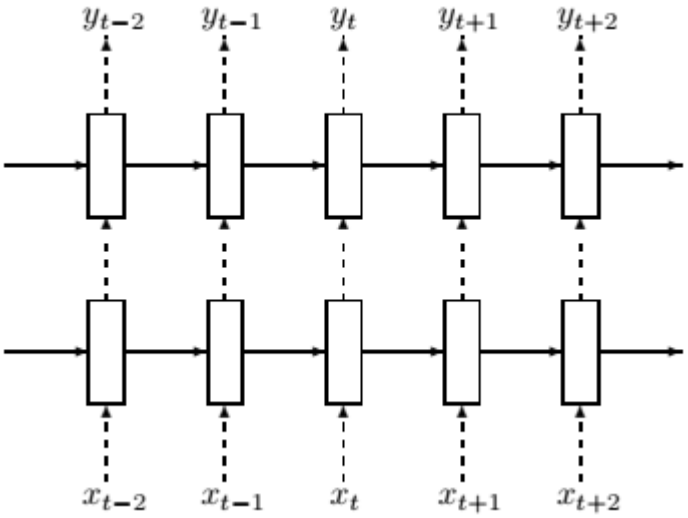


图 7.21 多层 LSTM ，其中的方块是一个 cell

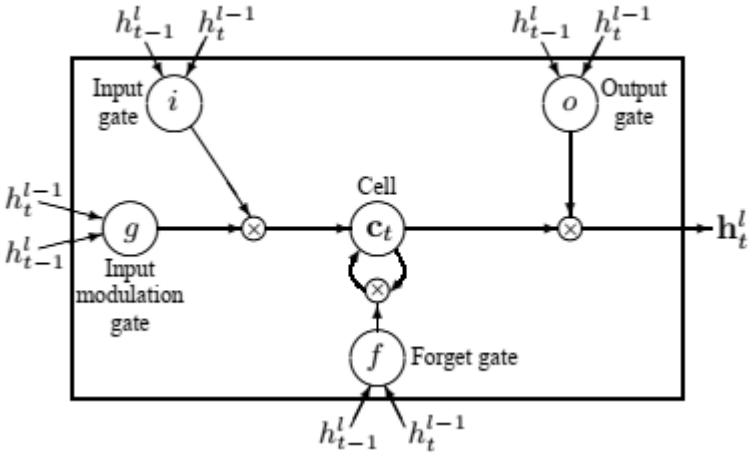


图 7.22 Basic LSTM Cell 的内部结构

TensorFlow 的 tutorial “Recurrent Neural Networks”参考 Wojciech Zaremba 的论文 RECURRENT NEURAL NETWORK REGULARIZATION 实施了一个 LSTM 语言模型。本节我是想按照该 tutorial 来讲解，但它讲的不好。另外，它提供的程序在我的环境下有

很多 bugs。我提供的 ptb_word_lm.py, reader.py 是修正了 bugs 的程序。我部分按照该 tutorial 和修正后的程序来讲解。

1.数据集

该模型需要的 PTB 数据集 http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz

该数据集已经被预处理，包含 10000 个 word，包括句子结束标记和一个特殊的符号<unk>指代很少出现的词。read.py 转换数据集，分配词编号。

2.输入

在将输入数据“喂给”LSTM 模型前，输入的词是以编号提供的，应该被转换成词向量。在该例子中，查找表被随机初始化，也作为了模型的参数来学习。

```
embedding = tf.get_variable(
          "embedding", [vocab_size, size], dtype=data_type())
inputs = tf.nn.embedding_lookup(embedding, input_.input_data)
```

这里 tf.get_variable 函数寻找当前环境下名为“embedding”的 variable。如果没找到会初始化一个该 variable。然后查找表函数 `tf.nn.embedding_lookup` 会将输入转换成词向量。

3.LSTM

模型的核心是 LSTM cell，它一次处理一个 word，并计算句子可能连续的概率。网络的 memory state (cell state) 用“零”向量初始化，每读入一个 word 就更新。模型的训练过程采用 minibatch。

```
inputs = [tf.squeeze(input_step, [1])
              for input_step in tf.split(1, num_steps, inputs)]
outputs, state = tf.nn.rnn(cell, inputs, initial_state=self._initial_state)
```

这里的 cell 参数是在下面构建的。

3.多层 RNN

要想使得模型有更强的表达能力，可以加多个 LSTM 层来处理数据。第一层的输出将成为第二层的输入，以此类推。MultiRNNCell 类可以实施多层 RNN。

代码如下：

```
lstm_cell = tf.nn.rnn_cell.BasicLSTMCell(size, forget_bias=0.0,
state_is_tuple=True)
if is_training and config.keep_prob < 1:
    lstm_cell = tf.nn.rnn_cell.DropoutWrapper(
       lstm_cell, output_keep_prob=config.keep_prob)
```

```
cell = tf.nn.rnn_cell.MultiRNNCell([lstm_cell] * config.num_layers,
state_is_tuple=True)
```

参数 size, 即第五节一开始解释的 Hidden-size.

代码中还使用了几个技巧来得到更好性能的模型：

有计划的减小学习率

在 LSTM 层之间运用 dropout

4.损失函数

这里的损失函数是 target word 的平均负 log 概率

$$\text{loss} = -\frac{1}{N}\sum_{i=1}^{N} \ln p_{\text{target}_i}$$

TensorFlow 的函数 sequence_loss_by_example 可以建立该损失函数

在 Wojciech Zaremba 这篇论文里使用的是 average per-word perplexity，经常称作 perplexity。它等于

$$e^{-\frac{1}{N}\sum_{i=1}^{N} \ln p_{\text{target}_i}} = e^{\text{loss}}$$

在训练过程中，该值作为训练评价指标。

第八章：基于 LSTM 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 7 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。

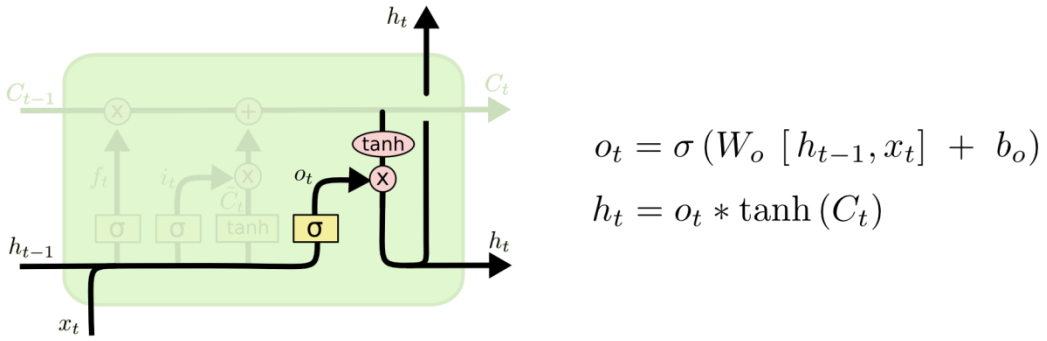


图 8.1 简化版的 LSTM

该文的模型如图 8.2 所示。和传统的 RNN 不同的是在 LSTM 层上加一个 average pooling 层，再加上一个 logistics 回归层。因此从输入序列 x0, x1, x2,…,xn，LSTM 层的 cell 将产生一个输出序列 h0, h1,…, hn。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。

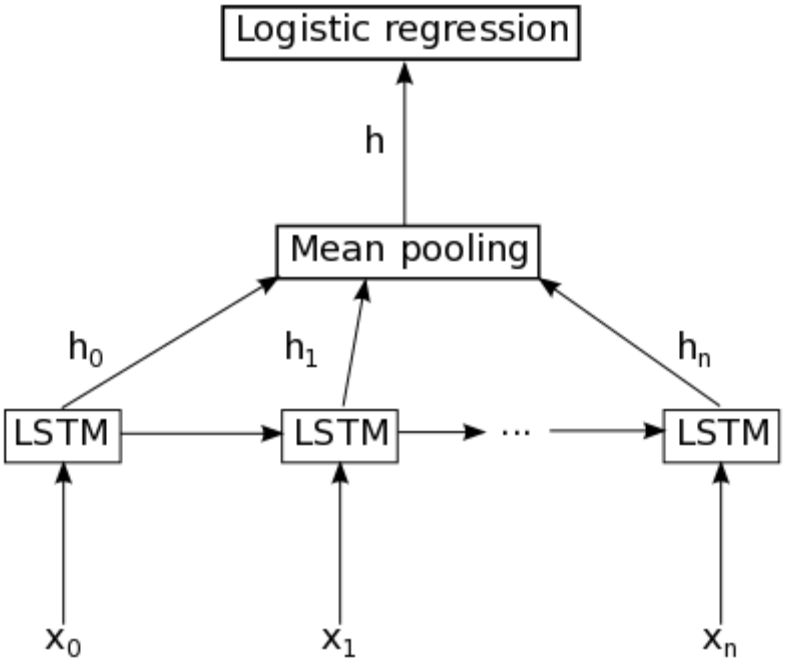


图 8.2 一个评论数据情感分类的 RNN 模型

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。

当前在用 TensorFlow 实施该模型时，一开始用训练集最长文本的长度作为 time step 的值，即每个文档被补齐到相同的长度。但这时模型在处理文本时效率非常之低。长时间的未能处理完文本。我决定用动态 time step 方式来建立模型，即每篇文本的长度不固定，time step 不固定。

第九章：Attention Mechanism

Attention Mechanism（有翻译做注意力机制）是 Dzmitry Bahdanau 在论文"Neural machine translation by jointly learning to align and translate" 中提出的。

两个很常用的两个 attention function: MLP Attention [1] and Bilinear Attention [2]

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

第十章：RNN Encoder-Decoder 和生成

新闻标题

RNN Encoder-Decoder 模型是一个 sequence to sequence 模型。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）。本章学习 RNN Encoder-Decoder 模型，找的应用场景是关键词抽取。参考了 ACL2017 论文《**Deep Keyphrase Generation**》，不过我对该论文存疑，因为产生的关键词不是序列（序列中前后词具有相关性），需要采用 RNN Encoder-Decoder 模型吗？（我觉得一个 RNN 模型就可以了）下面的内容结合了该论文和其他的 RNN Encoder-Decoder 相关论文。

Keywords 和 keyphrase 是指词语或词组的集合，用于对文档的内容做主要语义描述。传统的关键短语抽取（非深度学习的方法）通常包含两个步骤：（1）从文本内容里抽取潜在的候选短语；（2）对候选短语排序。传统的方法有个缺点，它们都是使用文本内容里出现的短语（present keyphrase），而不能基于文本的语义，即总结了文本的内容，但没有出现在文本里的词（absent keyphrase）不会被选择。人类在从一段文本内容中抽取关键词时，往往产生的词不一定是出现在文本中的，但是是和文本内容语义相关的。该文建立的 RNN 模型试图捕捉文本的语义和语法特征。

1. 工作原理

基于 RNN 的 Encoder-decoder 模型。其思想是，使用 Encoder 压缩文本内容到一个隐描述空间，然后用一个 Decoder 产生对应的 keyphrase。Encoder 和 Decoder 都是用 RNN 来实施。

Encoder 通过迭代执行下面的公式

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

转换长度可变的输入序列（文本）x=(x1,x2,…,xT)到一个隐描述序列 h=(h1,h2,…,hT)。这里，f 是一个非线性函数。我们可以获得一个 context 向量 c

c=q(h1,h2,…,hT)

Decoder 是另外一个 RNN。它使用一个条件语言模型

$s_t=f(y_{t-1},s_{t-1},c)$

$p(yt|y_{1,\ldots,t-1},x)=g(y_{t-1},s_t,c)$

将 context 向量分解成一个长度可变的序列 y=(y1,y2,…,y_T')。st 是 decoder RNN 在时刻 t 的隐状态。非线性函数 g 是一个 softmax 分类器，它的输出是词汇表中所有词的概率。yt 是在 t 时刻被预测的词项，即选取 g(.)中对应最大概率的词项。

Encoder 和 Decoder 网络联合训练，以达到给定源序列，最大化目标序列的条件概率。训练完成后使用 beam search 来产生 phrase。一个 max heap 被维护用于从预测的概率值中，选择预测的 word 序列。

2. 实施细节

Encoder 应用一个双向 gated recurrent unit (GRU)。有研究已经证实，GRU 比简单的 RNN 和 LSTM 中的简单结构在语言模型上具有更好的性能。因此，上面的非线性函数 f 用 GRU 替换。

Decoder 中使用一个前向 GRU。另外一个 attention mechanism 被采纳，用于提高性能。

第十一章：基于 BiLSTM 的问答系统

Appendix：深度学习工作站的配置

GPU: 4 x Titan X Pascal GPU。之所以不选择 1080 是因为只有 8GB 的内存，可能深度学习在处理大数据集的时候有可能会遇到点瓶颈。之前 Titan X Pascal 一直断货，几周前官网开始偶尔有货，我请系里的秘书帮忙刷了一周，总算买了几块（一天只能买两块）。http://www.geforce.com/hardware/10series/titan-x-pascal

主板：Asus X99-e WS。这个是 workstation 版本的 Asus X99，支持四块显卡，USB 3.1，很多的硬盘接口。支持 Intel 59xx 和 6 字头的 i7 处理器。不喜欢这块板子的也可以试试 rampage v edition 10，或者 rampage v extreme。可能也有便宜的板子支持 4 GPU，不过你要特别关注 PCIe 3.0 插槽的数量和布局：一块显卡通常会占据 2 个口的位置。

CPU: 通常来说 CPU 在多核 GPU 的深度学习系统里还是比较重要的，因为要并行处理参数。我这次选用了网友推荐的 i7 5930K，一共 6 核 12 线程，性价比还算凑合，跑起来也没什么太大问题。

内存：这个基本要看 CPU 能支持多少了，5930K 貌似只可以支持 64G，我就卖了两条 Kington valueRAM DDR4 32G。当然省钱的做法是买 8 条 8G 的用。

存储：SSD 还是比 HDD 快了不少，所以在这种情况下，我选择了 2 块 Samsung 850 EVO 1TB 的 SSD 内存。如果数据集太大，也可以考虑搞个 4TB 的 HDD 来存一下（10TB 和 8TB 的还是有点贵）。

CPU 冷却：我选了 Corsair H60 水冷。注意装的时候有两套 4 个螺丝钉，要选短的螺钉，短的装在板子上，另一端长的接在风冷上。H60 自带涂层，不过要注意水冷必须安装特别紧，一点点空气缝隙也不能留，不然估计深度学习压力测试你的 CPU 会到 80 度。不放心的可以上 H100i。

电源：电源还是很重要的基本 2 个选择 Corsair 1500W 或者 EVGA 1600W，因为一个 GPU 可能到 250W。当然实际运行的时候一般到不了那么高。我之前选了一个 Corsair 1200W，居然 self-test 风扇不转，只要连主板就会 reboot loop，明显是次品，赶紧趁机 RMA 换了 1600W。

机箱：不少人推荐 Corsair Carbide Air 540，这是一个中塔机箱。我最后选择了一个全塔机箱 Corsair 900D，通风好，但是特别重（配上所有东西超过 50 多斤重。。。保险箱的节奏）。大机箱可以放很多硬盘，如果你需要的话。

最后用 USB 3.1 启动机器，几分钟就装好了 Ubuntu 16.04。注意最好 UEFI BIOS 配置取消 Secure Boot 功能，不然你装 Titan X 驱动和 CUDA 8.0 RC 会有问题。装 TensorFlow 也没什么问题，就是要是找不到 CUDA 库的错误，可以用 sudo ldconfig /usr/local/cuda/lib64 和 LD_LIBRARY_PATH / LIBRARY_PATH 来解决。

我测试了 4 块 Titan X Pascal 跑 TF 的 CIFAR 多 GPU 训练，训练几天时间一切都很正常，GPU 的温度最高 70 度（设计 80 C 温度范围内，其他几块会低），GPU 风扇也不会到 50%速度。目前我也在测 Supermicro 的 superserver 多显卡配置，可能成本会更低。