**深度学习**

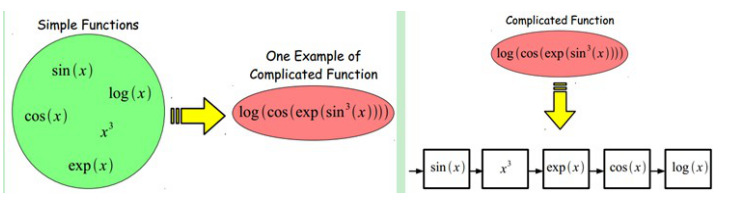
## 深度学习概述

1 深度学习（Deep Learning）是机器学习研究中的一个新的领域，深度学习是当下最热门的方向之一。其动机在于建立、模拟人脑进行分析学习的神经网络，它模仿人脑的机制来解释数据，例如图像，声音和文本。

2 深度学习热潮爆发以来，诸多研究者都在不懈地努力着，希望能够把它应用于解决计算机视觉的各种任务上，从高层次（high-level）的识别（recognition），分类（classification）到低层次（low-level）的去噪（denoise）。让人不禁联想起当年的稀疏表达（sparse representation）的热潮，而深度学习如今的风靡程度看上去是有过之而无不及。深度学习也有横扫high-level问题的趋势，high-level的很多方向都在被其不断刷新着数据。

3 作为强大的特征（feature）学习工具，在给定足够多的训练集的情况下，它可以帮助用户学习到这个任务下的具有很强分辨能力的特征。

4 深度学习可通过学习一种深层非线性网络结构，实现复杂函数逼近，表征输入数据分布式表示，并展现了强大的从少数样本集中学习数据集本质特征的能力。多层的好处在于可以用较少的参数表示复杂的函数。



5 深度学习的实质，是通过构建具有很多隐层的机器学习模型和海量的训练数据，来学习更有用的特征，从而最终提升分类或预测的准确性。因此，“深度模型”是手段，“特征学习”是目的。区别于传统的浅层学习，深度学习的不同在于：

强调了模型结构的深度，通常有5层、6层，甚至10多层的隐层节点；

明确突出了特征学习的重要性，也就是说，通过逐层特征变换，将样本在原空间的特征表示变换到一个新特征空间，从而使分类或预测更加容易。与人工规则构造特征的方法相比，利用大数据来学习特征，更能够刻画数据的丰富内在信息。

6 深度学习在近期赢得了很多关注， 特别是 百度也开始发力深度学习后（Institute of Deep Learning）， 更是在国内引起了很多关注。在计算能力变得日益廉价的今天，深度学习试图建立大得多也复杂得多的神经网络。很多深度学习的算法是无监督或半监督式学习算法，用来处理存在少量未标识数据的大数据集。常见的深度学习算法包括：

受限波尔兹曼机（Restricted Boltzmann Machine）

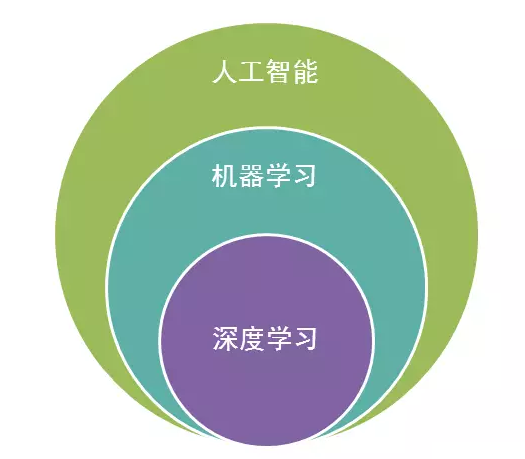
深度信念网络（ Deep Belief Networks）

卷积神经网络（Convolutional Neural Networks）

堆栈式自动编码器（Stacked Auto-encoders）

**7 人工智能、机器学习、深度学习联系和区别**

1. 先看什么是人工智能。人工智能（Artificial Intelligence），英文缩写为AI。是计算机科学的一个分支，二十世纪七十年代以来被称为世界三大尖端技术之一（空间技术、能源技术、人工智能）。也被认为是二十一世纪三大尖端技术（基因工程、纳米科学、人工智能）之一。1956年夏季，以麦卡赛、明斯基、罗切斯特和申农等为首的一批有远见卓识的年轻科学家在一起聚会，共同研究和探讨用机器模拟智能的一系列有关问题，并首次提出了“人工智能”这一术语，它标志着“人工智能”这门新兴学科的正式诞生。人工智能是对人的意识、思维的信息过程的模拟。人工智能不是人的智能，但能像人那样思考、也可能超过人的智能。数学常被认为是多种学科的基础科学，数学也进入语言、思维领域，人工智能学科也必须借用数学工具。



人工智能实际应用：机器视觉，指纹识别，人脸识别，视网膜识别，虹膜识别，掌纹识别，专家系统，自动规划，智能搜索，定理证明，博弈，自动程序设计，智能控制，机器人学，语言和图像理解，遗传编程等。涉及到哲学和认知科学，数学，神经生理学，心理学，计算机科学，信息论，控制论，不定性论等学科。研究范畴包括自然语言处理，知识表现，智能搜索，推理，规划，机器学习，知识获取，组合调度问题，感知问题，模式识别，逻辑程序设计软计算，不精确和不确定的管理，人工生命，神经网络，复杂系统，遗传算法等。人工智能目前也分为：强人工智能(BOTTOM-UP AI)和弱人工智能(TOP-DOWN AI)，有兴趣大家可以自行查看下区别。

（2）机器学习(Machine Learning, ML)，是人工智能的核心，属于人工智能的一个分支，是一个大的领域，是让计算机拥有像人一样的学习能力，模拟和实现人的学习行为和能力，可以像人一样具有识别和判断的能力，可以看作是仿生学。机器学习的核心就是数据，算法（模型），算力（计算机运算能力）。以前也有人工智能，机器学习。不过最近几年网络发展和大数据的积累，使得人工智能能够在数据和高运算能力下发挥它的作用。机器学习应用领域十分广泛，例如：数据挖掘、数据分类、计算机视觉、自然语言处理(NLP)、生物特征识别、搜索引擎、医学诊断、检测信用卡欺诈、证券市场分析、DNA序列测序、语音和手写识别、战略游戏和机器人运用等。

（3）总结：机器学习和深度学习的核心就是数据、算法（模型函数）、算力（计算机运算能力）。我们研究的核心就是算法，也就是针对某一应用需求场景，选择各种合适的函数公式进行构建和叠加形成一个处理数据的模型（也就是常说的训练数据模型）。

这个模型的作用是什么？就是能够将我们的普通数据输入进去，经过这个模型处理后，输出数据或者一个结果，这个输出的就是供我们参考或者使用的。那我们怎么评判这个模型的好坏？例如每年 ImageNet 都会进行比赛，那么评判的结果就是你这个模型算法进行数据分析和处理的准确度。例如人脸识别，有的算法识别率 80%，有的达到 98%。那么这个高识别率的模型算法就要好一些。我们需要改进的也就是这个模型算法（也就是调优），其中就会涉及到权重（Weight）和损失函数（Loss）。当然损失函数越低说明我们的算法处理数据越接近最佳结果。

整个流程就是：选择训练数据（用于训练模型）->模型设计和调优（通过训练数据进行反复调优）->形成比较准确的模型->输入真实数据（用于模型进行数据处理）->模型处理数据->输出处理结果。

举个例子：我们进行预测某天天气的状况，这是一个需求。某天天气的状况，会受到很多因素的影响，例如季节、地理位置、时间、人为因素等等，这些因素可能会有几十个。在深度学习模型里，这些影响结果的因素我们称之为特征。我们可以通过一些有标签的天气数据去输入到我们的模型里，不断的给各个特征设置权重、损失函数，不断的进行调优。最后形成能够很好的准确预测天气的一个模型。

我们在进行机器学习和深度学习的核心，就是在算法这块，也就是设计处理问题（数据）的模型。模型需要针对不同的问题和需求进行合理的选择或者搭配。会涉及到很多数学公式和函数。其实这些数学函数在一些框架里已经囊括了，如tensorflow，python 的 numpy 库。我们需要的就是通过训练数据进行模型测试训练，调优，再应用。先有需求和目的：这个模型是干嘛的，处理什么问题的。输入的数据什么样的，想要有什么输出结果才是最好的，能够达到我们想要的目的的。是数据和需求驱动我们选择什么样的模型和函数算法。确定了大概的方案模型和算法函数，我们就要给模型不断的喂数据进行调整优化，使他能够输出更加准确的数据处理结果，也就是常说的训练模型。当模型通过数据训练好了后，我们就可以进行应用了，喂给它真实的数据进行处理。

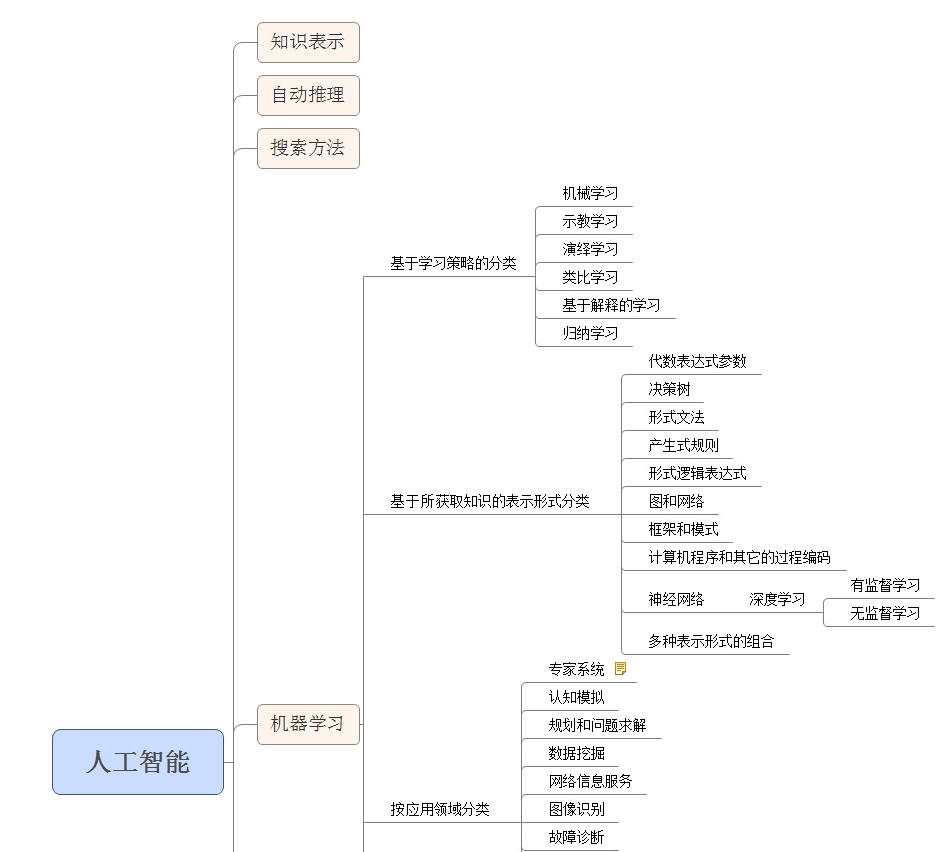
接下来看下深度学习。深度学习（Deep Learning）是机器学习的一种，是它的一个大的分支，深度学习的概念由Hinton等人于2006年提出，深度学习的概念源于人工神经网络的研究。既然叫做学习，那么自然与我们人类的学习过程有相似之处，其实就是仿生学，仿照我们人类大脑的思维方式以及神经网络的接收和反馈方式进行计算机模拟深度学习的。我们的大脑就是一个深度学习的超级计算机。深度学习实际上指的的深度神经网络学习，普通神经网络由于训练代价较高，一般只有3-4层，而深度神经网络由于采用了特殊的训练方法加上一些技术算法，可以达到8-10层。深度神经网络能够捕捉到数据中的深层联系，从而能够得到更精准的模型，而这些联系不容易被普通的机器学习方法所发觉。

用官方的含义就是：含多隐层的多层感知器就是一种深度学习结构。深度学习通过组合低层特征形成更加抽象的高层表示属性类别或特征，以发现数据的分布式特征表示。同机器学习方法一样，深度机器学习方法也有监督学习与无监督学习之分．不同的学习框架下建立的学习模型很是不同．例如，卷积神经网络（Convolutional neural networks，简称CNNs）就是一种深度的监督学习下的机器学习模型，而深度置信网（Deep Belief Nets，简称DBNs）就是一种无监督学习下的机器学习模型。

深度学习整个过程就是数据收集、数据清洗处理、传入数据进行训练模型和学习优化、经过不断优化调节后形成高准确率的识别分类模型，供相应的领域进行传入相关数据进行应用分类。举个例子，我们人类在刚出生时看到一个手机，那么他是不知道这个是什么的，并且有各种各样形状和样式的手机，此时我们的深度学习系统初始状态就是这样的。但是经过父母和周围的分类和指导，我们渐渐的知道了这种样子和形状、功能的物体是手机，那么我们通过大量的数据就具有了学习和分辨手机的能力模型，那么这就是我们的深度学习系统在经过数据和算法训练后所具备的功能和学习能力。就是这么简单，可以说就是仿生学。

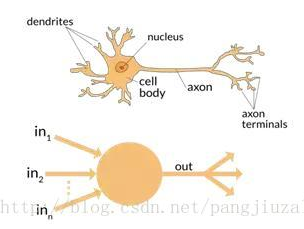
2017年CCF青年精英大会，香港中文大学教授汤晓鸥指出，深度学习的三大核心要素，就是算法设计、高性能的计算能力、大数据。我觉得应该按照这个顺序排序：大数据、算法设计、高性能的计算能力。

例如深度学习在语音识别和自然语言领域，微软研究人员通过与hinton合作，首先将RBM和DBN引入到语音识别声学模型训练中，并且在大词汇量语音识别系统中获得巨大成功，使得语音识别的错误率相对减低30%。但是，DNN还没有有效的并行快速算法，很多研究机构都是在利用大规模数据语料通过GPU平台提高DNN声学模型的训练效率。在国际上，IBM、google等公司都快速进行了DNN语音识别的研究，并且速度飞快。国内方面，阿里巴巴，科大讯飞、百度、中科院自动化所等公司或研究单位，也在进行深度学习在语音识别上的研究。

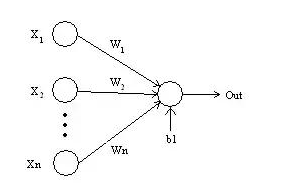


**8 深度学习基础概念**

1）神经元（Neuron）——就像形成我们大脑基本元素的神经元一样，神经元形成神经网络的基本结构。想象一下，当我们得到新信息时我们该怎么做。当我们获取信息时，我们一般会处理它，然后生成一个输出。类似地，在神经网络的情况下，神经元接收输入，处理它并产生输出，而这个输出被发送到其他神经元用于进一步处理，或者作为最终输出进行输出。



2）权重（Weights）——当输入进入神经元时，它会乘以一个权重。例如，如果一个神经元有两个输入，则每个输入将具有分配给它的一个关联权重。我们随机初始化权重，并在模型训练过程中更新这些权重。训练后的神经网络对其输入赋予较高的权重，这是它认为与不那么重要的输入相比更为重要的输入。为零的权重则表示特定的特征是微不足道的。让我们假设输入为a，并且与其相关联的权重为W1，那么在通过节点之后，输入变为a \* W1



3）偏差（Bias）——除了权重之外，另一个被应用于输入的线性分量被称为偏差。它被加到权重与输入相乘的结果中。基本上添加偏差的目的是来改变权重与输入相乘所得结果的范围的。添加偏差后，结果将看起来像a\* W1 +偏差。这是输入变换的最终线性分量。

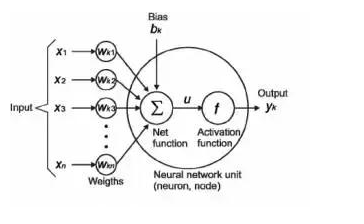
4）激活函数（Activation Function）——一旦将线性分量应用于输入，将会需要应用一个非线性函数。这通过将激活函数应用于线性组合来完成。激活函数将输入信号转换为输出信号。应用激活函数后的输出看起来像f（a \* W1 + b），其中f（）就是激活函数。

在下图中，我们将“n”个输入给定为X1到Xn而与其相应的权重为Wk1到Wkn。我们有一个给定值为bk的偏差。权重首先乘以与其对应的输入，然后与偏差加在一起。而这个值叫做u。

U =ΣW\* X+ b

激活函数被应用于u，即 f(u)，并且我们会从神经元接收最终输出，如yk = f（u）。

常用的激活函数



最常用的激活函数就是Sigmoid，ReLU和softmax

a）Sigmoid——最常用的激活函数之一是Sigmoid，它被定义为：

Sigmoid变换产生一个值为0到1之间更平滑的范围。我们可能需要观察在输入值略有变化时输出值中发生的变化。光滑的曲线使我们能够做到这一点，因此优于阶跃函数。

b）ReLU（整流线性单位）——与Sigmoid函数不同的是，最近的网络更喜欢使用ReLu激活函数来处理隐藏层。该函数定义为：

当X>0时，函数的输出值为X；当X<=0时，输出值为0。函数图如下图所示：

使用ReLU函数的最主要的好处是对于大于0的所有输入来说，它都有一个不变的导数值。常数导数值有助于网络训练进行得更快。

c） Softmax——Softmax激活函数通常用于输出层，用于分类问题。它与sigmoid函数是很类似的，唯一的区别就是输出被归一化为总和为1。Sigmoid函数将发挥作用以防我们有一个二进制输出，但是如果我们有一个多类分类问题，softmax函数使为每个类分配值这种操作变得相当简单，而这可以将其解释为概率。

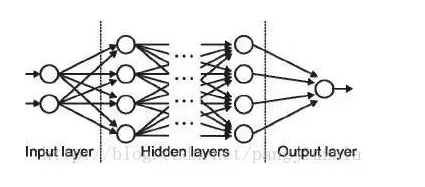
以这种方式来操作的话，我们很容易看到——假设你正在尝试识别一个可能看起来像8的6。该函数将为每个数字分配值如下。我们可以很容易地看出，最高概率被分配给6，而下一个最高概率分配给8，依此类推……

5）神经网络（Neural Network）——神经网络构成了深度学习的支柱。神经网络的目标是找到一个未知函数的近似值。它由相互联系的神经元形成。这些神经元具有权重和在网络训练期间根据错误来进行更新的偏差。激活函数将非线性变换置于线性组合，而这个线性组合稍后会生成输出。激活的神经元的组合会给出输出值。

一个很好的神经网络定义——

“神经网络由许多相互关联的概念化的人造神经元组成，它们之间传递相互数据，并且具有根据网络”经验“调整的相关权重。神经元具有激活阈值，如果通过其相关权重的组合和传递给他们的数据满足这个阈值的话，其将被解雇;发射神经元的组合导致“学习”。

6）输入/输出/隐藏层（Input / Output / Hidden Layer）——正如它们名字所代表的那样，输入层是接收输入那一层，本质上是网络的第一层。而输出层是生成输出的那一层，也可以说是网络的最终层。处理层是网络中的隐藏层。这些隐藏层是对传入数据执行特定任务并将其生成的输出传递到下一层的那些层。输入和输出层是我们可见的，而中间层则是隐藏的。



7）MLP（多层感知器）——单个神经元将无法执行高度复杂的任务。因此，我们使用堆栈的神经元来生成我们所需要的输出。在最简单的网络中，我们将有一个输入层、一个隐藏层和一个输出层。每个层都有多个神经元，并且每个层中的所有神经元都连接到下一层的所有神经元。这些网络也可以被称为完全连接的网络。



8）正向传播（Forward Propagation）——正向传播是指输入通过隐藏层到输出层的运动。在正向传播中，信息沿着一个单一方向前进。输入层将输入提供给隐藏层，然后生成输出。这过程中是没有反向运动的。

9）成本函数（Cost Function）——当我们建立一个网络时，网络试图将输出预测得尽可能靠近实际值。我们使用成本/损失函数来衡量网络的准确性。而成本或损失函数会在发生错误时尝试惩罚网络。

我们在运行网络时的目标是提高我们的预测精度并减少误差，从而最大限度地降低成本。最优化的输出是那些成本或损失函数值最小的输出。

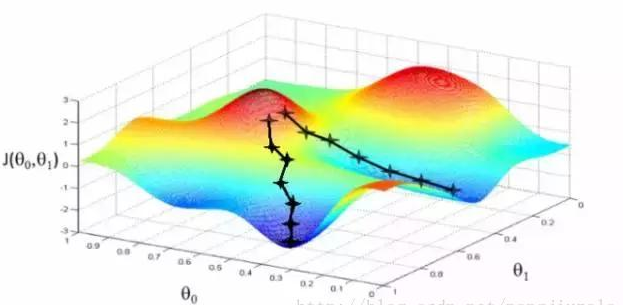
如果我将成本函数定义为均方误差，则可以写为：

C= 1/m ∑(y–a)^2，

其中m是训练输入的数量，a是预测值，y是该特定示例的实际值。

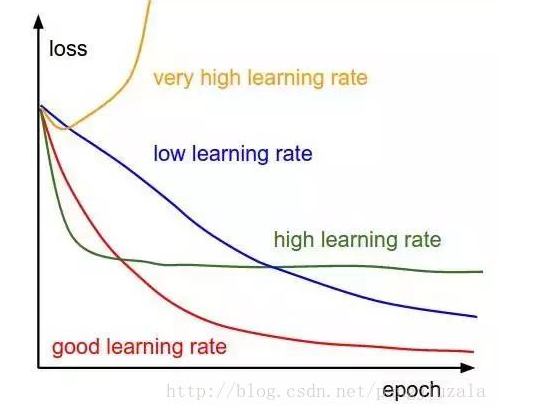
学习过程围绕最小化成本来进行。

10）梯度下降（Gradient Descent）——梯度下降是一种最小化成本的优化算法。要直观地想一想，在爬山的时候，你应该会采取小步骤，一步一步走下来，而不是一下子跳下来。因此，我们所做的就是，如果我们从一个点x开始，我们向下移动一点，即Δh，并将我们的位置更新为x-Δh，并且我们继续保持一致，直到达到底部。考虑最低成本点。



在数学上，为了找到函数的局部最小值，我们通常采取与函数梯度的负数成比例的步长。

11）学习率（Learning Rate）——学习率被定义为每次迭代中成本函数中最小化的量。简单来说，我们下降到成本函数的最小值的速率是学习率。我们应该非常仔细地选择学习率，因为它不应该是非常大的，以至于最佳解决方案被错过，也不应该非常低，以至于网络需要融合。



12）反向传播（Backpropagation）——当我们定义神经网络时，我们为我们的节点分配随机权重和偏差值。一旦我们收到单次迭代的输出，我们就可以计算出网络的错误。然后将该错误与成本函数的梯度一起反馈给网络以更新网络的权重。 最后更新这些权重，以便减少后续迭代中的错误。使用成本函数的梯度的权重的更新被称为反向传播。

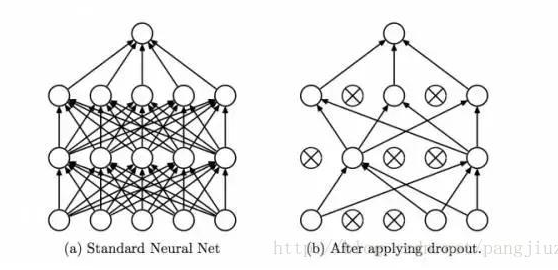
在反向传播中，网络的运动是向后的，错误随着梯度从外层通过隐藏层流回，权重被更新。

13）批次（Batches）——在训练神经网络的同时，不用一次发送整个输入，我们将输入分成几个随机大小相等的块。与整个数据集一次性馈送到网络时建立的模型相比，批量训练数据使得模型更加广义化。

14）周期（Epochs）——周期被定义为向前和向后传播中所有批次的单次训练迭代。这意味着1个周期是整个输入数据的单次向前和向后传递。

你可以选择你用来训练网络的周期数量，更多的周期将显示出更高的网络准确性，然而，网络融合也需要更长的时间。另外，你必须注意，如果周期数太高，网络可能会过度拟合。

15）丢弃（Dropout）——Dropout是一种正则化技术，可防止网络过度拟合套。顾名思义，在训练期间，隐藏层中的一定数量的神经元被随机地丢弃。这意味着训练发生在神经网络的不同组合的神经网络的几个架构上。你可以将Dropout视为一种综合技术，然后将多个网络的输出用于产生最终输出。



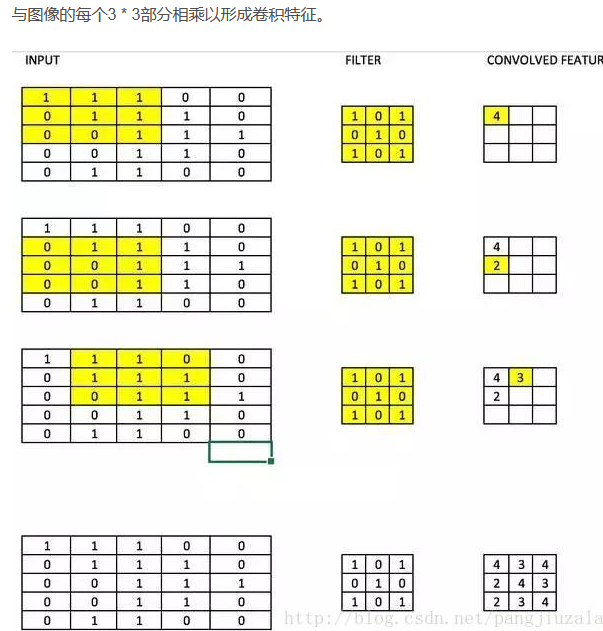
16）批量归一化（Batch Normalization）——作为一个概念，批量归一化可以被认为是我们在河流中设定为特定检查点的水坝。这样做是为了确保数据的分发与希望获得的下一层相同。当我们训练神经网络时，权重在梯度下降的每个步骤之后都会改变，这会改变数据的形状如何发送到下一层。

但是下一层预期分布类似于之前所看到的分布。 所以我们在将数据发送到下一层之前明确规范化数据。

卷积神经网络

17）滤波器（Filters）——CNN中的滤波器与加权矩阵一样，它与输入图像的一部分相乘以产生一个回旋输出。我们假设有一个大小为28 \* 28的图像，我们随机分配一个大小为3 \* 3的滤波器，然后与图像不同的3 \* 3部分相乘，形成所谓的卷积输出。滤波器尺寸通常小于原始图像尺寸。在成本最小化的反向传播期间，滤波器值被更新为重量值。

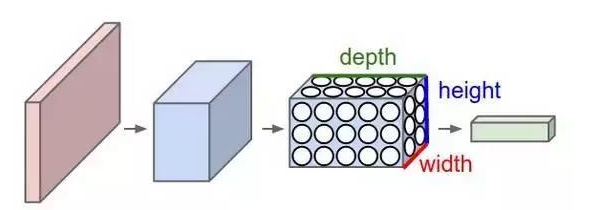
参考一下下图，这里filter是一个3 \* 3矩阵：



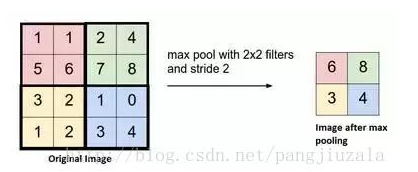
与图像的每个3 \* 3部分相乘以形成卷积特征。

18）卷积神经网络（CNN）——卷积神经网络基本上应用于图像数据。假设我们有一个输入的大小（28 \* 28 \* 3），如果我们使用正常的神经网络，将有2352（28 \* 28 \* 3）参数。并且随着图像的大小增加参数的数量变得非常大。我们“卷积”图像以减少参数数量（如上面滤波器定义所示）。当我们将滤波器滑动到输入体积的宽度和高度时，将产生一个二维激活图，给出该滤波器在每个位置的输出。我们将沿深度尺寸堆叠这些激活图，并产生输出量。

你可以看到下面的图，以获得更清晰的印象。

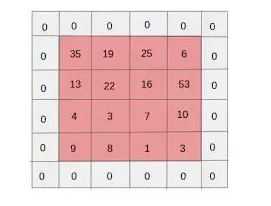


19）池化（Pooling）——通常在卷积层之间定期引入池层。这基本上是为了减少一些参数，并防止过度拟合。最常见的池化类型是使用MAX操作的滤波器尺寸（2,2）的池层。它会做的是，它将占用原始图像的每个4 \* 4矩阵的最大值。



你还可以使用其他操作（如平均池）进行池化，但是最大池数量在实践中表现更好。

20）填充（Padding）——填充是指在图像之间添加额外的零层，以使输出图像的大小与输入相同。这被称为相同的填充。



在应用滤波器之后，在相同填充的情况下，卷积层具有等于实际图像的大小。

有效填充是指将图像保持为具有实际或“有效”的图像的所有像素。在这种情况下，在应用滤波器之后，输出的长度和宽度的大小在每个卷积层处不断减小。

21）数据增强（Data Augmentation）——数据增强是指从给定数据导出的新数据的添加，这可能被证明对预测有益。例如，如果你使光线变亮，可能更容易在较暗的图像中看到猫，或者例如，数字识别中的9可能会稍微倾斜或旋转。在这种情况下，旋转将解决问题并提高我们的模型的准确性。通过旋转或增亮，我们正在提高数据的质量。这被称为数据增强。

循环神经网络



22）循环神经元（Recurrent Neuron）——循环神经元是在T时间内将神经元的输出发送回给它。如果你看图，输出将返回输入t次。展开的神经元看起来像连接在一起的t个不同的神经元。这个神经元的基本优点是它给出了更广义的输出。

23）循环神经网络（RNN）——循环神经网络特别用于顺序数据，其中先前的输出用于预测下一个输出。在这种情况下，网络中有循环。隐藏神经元内的循环使他们能够存储有关前一个单词的信息一段时间，以便能够预测输出。隐藏层的输出在t时间戳内再次发送到隐藏层。展开的神经元看起来像上图。只有在完成所有的时间戳后，循环神经元的输出才能进入下一层。发送的输出更广泛，以前的信息保留的时间也较长。

然后根据展开的网络将错误反向传播以更新权重。这被称为通过时间的反向传播（BPTT）。

24）消失梯度问题（Vanishing Gradient Problem）——激活函数的梯度非常小的情况下会出现消失梯度问题。在权重乘以这些低梯度时的反向传播过程中，它们往往变得非常小，并且随着网络进一步深入而“消失”。这使得神经网络忘记了长距离依赖。这对循环神经网络来说是一个问题，长期依赖对于网络来说是非常重要的。

这可以通过使用不具有小梯度的激活函数ReLu来解决。

25）激增梯度问题（Exploding Gradient Problem）——这与消失的梯度问题完全相反，激活函数的梯度过大。在反向传播期间，它使特定节点的权重相对于其他节点的权重非常高，这使得它们不重要。这可以通过剪切梯度来轻松解决，使其不超过一定值。

## TensorFlow

**1 TensorFlow介绍**

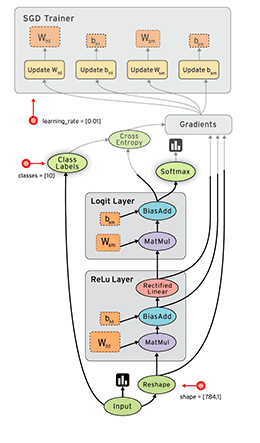
**TensorFlow是一个采用数据流图（data flow graphs），用于数值计算的开源软件库。**TensorFlow 最初由Google大脑小组（隶属于Google机器智能研究机构）的研究员和工程师们开发出来，用于机器学习和深度神经网络方面的研究，但这个系统的通用性使其也可广泛用于其他计算领域。它是谷歌基于DistBelief进行研发的第二代人工智能学习系统。2015年11月9日，Google发布人工智能系统TensorFlow并宣布开源。

**2 TensorFlow名字的来源**

其命名来源于本身的原理，Tensor（张量）意味着N维数组，Flow（流）意味着基于数据流图的计算。Tensorflow运行过程就是张量从图的一端流动到另一端的计算过程。张量从图中流过的直观图像是这个工具取名为“TensorFlow”的原因。

**3 什么是数据流图（Data Flow Graph）**

数据流图用“节点”(nodes)和“线”（edges）的有向图来描述数学计算。“节点”一般用来表示施加的数学操作，但也可以表示数据输入（feed in）的起点/输出（push out）的终点，或者是读取/写入持久变量（persistent variable）的终点。“线”表示“节点”之间的输入/输出关系。这些数据“线”可以运输“size可动态调整”的多维数组，即“张量”（tensor）。一旦输入端所有张量准备好，节点将被分配到各种计算设备完成异步并行地执行计算。



**4 Tensorflow的特性**

（1）高度的灵活性： TensorFlow不是一个严格的“神经网络”库。只要你可以将你的计算表示为一个数据流图，你就可以使用TensorFlow。

（2）可移植性（Portability）：Tensorflow可以运行在台式机、服务器、手机移动等等设备上。而且它可以充分使用计算资源，在多CPU和多GPU上运行。

（3）多语言支持：Tensorflow提供了一套易用的Python使用接口来构建和执行graphs，也同样提供了一套易于C++使用的接口（目前训练神经网络只支持python，C++接口只能使用已经训练好的模型）。未来还会支持Go、Java、Lua、Javascript、R等等。

（4）性能最优化：TensorFlow给予了线程、队列、异步操作等最佳的支持，TensorFlow可以把你手边硬件的计算潜能全部发挥出来，它可以充分利用多CPU和多GPU。

**5 下载及安装**

既可以直接使用二进制程序包也可以从github源码库克隆源码编译安装。

**（1）要求**

TensorFlow 提供的Python API支持Python2.7和Python3.3+

GPU版本的二进制程序包只能使用Cuda Toolkit8.0 和 cuDNN v5。如果你使用的是其他版本（Cuda toolkit >= 7.0 and cuDNN >= v3），那你就必须使用源码重新编译安装。

**（2）推荐几种Linux平台的安装方式：**

**Pip install：** 可能会升级你之前安装过的Python包，对你机器上的Python程序造成影响。

**Anaconda install：**把TensorFlow安装在Anaconda提供的环境中，不会影响其他Python程序。

**Installing from sources**：把TensorFlow源码构建成一个pip wheel 文件，使用pip工具安装它。

**（3）Pip installation**

Pip是一个用来安装和管理Python软件包的包管理系统。

安装pip（如果已经安装，可以跳过）

# Ubuntu/Linux 64-bit

$ sudo apt-get install python-pip python-dev

直接使用pip安装TensorFlow

$ pip install tensorflow

如果提示找不到对应的包，使用

pip install --ignore-installed --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.12.0rc1-cp27-none-linux\_x86\_64.whl

安装GPU支持的版本：

# Requires CUDA toolkit 8.0 and CuDNN v5. 其他版本，参考下面的 “Installing from sources”

$ pip install tensorflow-gpu

如果提示找不到对应的包，使用

pip install --ignore-installed --upgrade TF\_BINARY\_URL=https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow\_gpu-0.12.0rc1-cp27-none-linux\_x86\_64.whl

注意：如果是从一个较老的版本（TensorFlow<0.7.1）进行升级，需要首先卸载之前的TensorFlow和protobuf使用：pip uninstall

Anaconda installation

Anaconda是一个Python发行版，包括大量的数字和科学计算包。使用“conda”来管理软件包，并且拥有自己的环境系统。安装步骤

安装Anaconda

创建conda环境

激活conda环境，在其中安装TensorFlow

每次使用TensorFlow时，激活conda环境

Anaconda具体的安装和使用可以参考：https://www.continuum.io/downloads

Installing from sources

从源码构建TensorFlow，具体的步骤参考：http://blog.csdn.net/toormi/article/details/52904551#t8

**6 基本使用**

**（1）基本概念**

**图（Graph）：**图描述了计算的过程，TensorFlow使用图来表示计算任务。

**张量（Tensor）：**TensorFlow使用tensor表示数据。每个Tensor是一个类型化的多维数组。

**操作（op）：**图中的节点被称为op（operation的缩写），一个op获得0个或多个Tensor，执行计算，产生0个或多个Tensor。

**会话（Session）：**图必须在称之为“会话”的上下文中执行。会话将图的op分发到诸如CPU或GPU之类的设备上执行。

**变量（Variable）：**运行过程中可以被改变，用于维护状态。

**（2）计算图（The computation graph）**

Tensorflow程序通常被组织成一个构建阶段和一个执行阶段。在构建阶段，op的执行步骤被描述成一个图。在执行阶段，使用会话执行图中的op。

**（3）构建图**

构建图的第一步是创建源op（sources op）。源op不需要任何输入，例如常量（Constant）。源op的输出被传递给其他op做运算。在TensorFlow的Python库中，op构造器的返回值代表这个op的输出。这些返回值可以作为输入传递给其他op构造器。TensorFlow的Python库中包含了一个默认的graph，可以在上面使用添加节点。如果你的程序需要多个graph那就需要使用Graph类管理多个graph。

import tensorflow as tf

# 创建一个常量 op, 产生一个 1x2 矩阵. 这个 op 被作为一个节点

# 加到默认图中.

#

# 构造器的返回值代表该常量 op 的返回值.

matrix1 = tf.constant([[3., 3.]])

# 创建另外一个常量 op, 产生一个 2x1 矩阵.

matrix2 = tf.constant([[2.],[2.]])

# 创建一个矩阵乘法 matmul op , 把 'matrix1' 和 'matrix2' 作为输入.

# 返回值 'product' 代表矩阵乘法的结果.

product = tf.matmul(matrix1, matrix2)

默认图中包含了3个节点：两个constant() op和一个matmul() op。为了真正的执行矩阵相乘运算，并得到矩阵乘法的结果，你必须在会话中启动这个图。

**（4）启动图**

构造阶段完成后，才能在会话中启动图。启动图的第一步是创建一个Session对象。如果没有任何参数，会话构造器将启动默认图。

# 启动默认图.

**sess = tf.Session()**

# 调用 sess 的 'run()' 方法来执行矩阵乘法 op, 传入 'product' 作为该方法的参数.

# 上面提到, 'product' 代表了矩阵乘法 op 的输出, 传入它是向方法表明, 我们希望取回

# 矩阵乘法 op 的输出.

# 整个执行过程是自动化的, 会话负责传递 op 所需的全部输入. op 通常是并发执行的.

# 函数调用 'run(product)' 触发了图中三个 op (两个常量 op 和一个矩阵乘法 op) 的执行.

# 返回值 'result' 是一个 numpy `ndarray` 对象.

result = sess.run(product)

print result

# ==> [[ 12.]]

# 任务完成, 关闭会话.

sess.close()

Session对象在使用完成或需要关闭以释放资源。除了显示调用close外，也可以使用“with”代码块来自动完成关闭动作。

with tf.Session() as sess:

result = sess.run([product])

print result

Tensorflow的实现上，会把图转换成可分布式执行的操作，以充分利用计算资源（例如CPU或GPU）。通常情况下，你不需要显示指使用CPU或者GPU。TensorFlow能自动检测，如果检测到GPU，TensorFlow会使用第一个GPU来执行操作。如果机器上有多个GPU，除第一个GPU外的其他GPU是不参与计算的，为了使用这些GPU，你必须将op明确指派给他们执行。with…Device语句用来指派特定的CPU或GPU执行操作：

with tf.Session() as sess:

with tf.device("/gpu:1"):

matrix1 = tf.constant([[3., 3.]])

matrix2 = tf.constant([[2.],[2.]])

product = tf.matmul(matrix1, matrix2)

...

设备用字符串进行标识. 目前支持的设备包括:

“/cpu:0”: 机器的 CPU.

“/gpu:0”: 机器的第一个 GPU, 如果有的话.

“/gpu:1”: 机器的第二个 GPU, 以此类推.

**（5）Tensor**

Tensorflow使用tensor数据结构来代表所有的数据。计算图的操作之间仅能传递tensor。你可以把tensor当作多维数组或列表。每一个tensor包含有一个静态类型，一个rank和一个shape。想了解更多TensorFlow是如何操作这些概念的，参考Rank, Shape, and Type

**（6）变量**

变量维持图计算过程中的状态信息。下面的例子演示了如何使用变量作为一个简单的计数器。

# 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.global\_variables\_initializer()

# 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

通常可以将一个统计模型中的参数表示为一组变量。例如，你可以将一个神经网络的权重当作一个tensor存储在变量中。在训练图的重复运行过程中去更新这个tensor。

**（7）Fetch**

为了取回操作的输出内容，在使用Session对象的run()方法执行图时，传入一些tensor，这些tensor会帮你取回结果。之前的例子中，我们只取回了state节点，但是你也可以取回多个tensor：

input1 = tf.constant(3.0)

input2 = tf.constant(2.0)

input3 = tf.constant(5.0)

intermed = tf.add(input2, input3)

mul = tf.mul(input1, intermed)

with tf.Session() as sess:

result = sess.run([mul, intermed])

print result

# 输出:

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

需要获取的多个 tensor 值，在 op 的一次运行中一起获得（而不是逐个去获取 tensor）。

**（8）Feed**

上面的例子中展示了在计算图中引入tensor，以常量和变量的形式存储。TensorFlow还提供了feed机制，该机制可以临时替换图中的tensor。

feed使用一个tensor值临时替换一个操作的输出。可以把feed数据作为参数提供给run()方法。标记的方法是使用tf.placeholder()为这些操作创建占位符。

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.]})

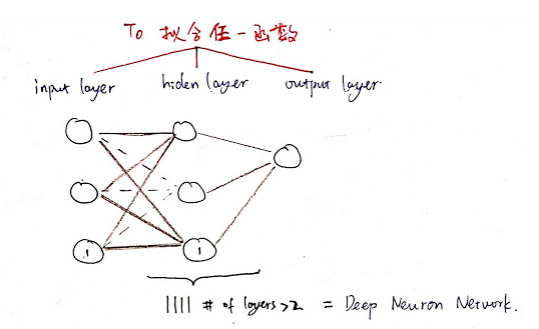
# 输出:

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

## 感知器神经网络

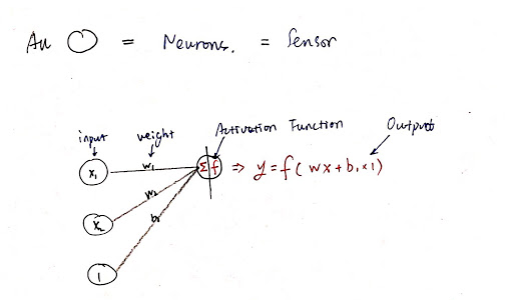
**1 什么是感知器**

**如下图，这个神经网络中，每个圆圈都是一个神经元，神经元也叫做感知器。**



只有一个隐藏层的神经网络就能拟合任何一个函数，但是它需要很多很多的神经元。

而深层网络用相对少的神经元就能拟合同样的函数，但是层数增加了，不太容易训练，需要大量的数据。为了拟合一个函数，可以使用一个浅而宽的网络，也可以使用一个深而窄的网络，后者更节约资源。下图单挑出一个感知器来看：向它输入 inputs，经过 加权 求和，再作用上激活函数后，得到一个输出值。



感知器的激活函数可以有很多选择，关于激活函数可以看 常用激活函数比较。

**（1）神经网络**

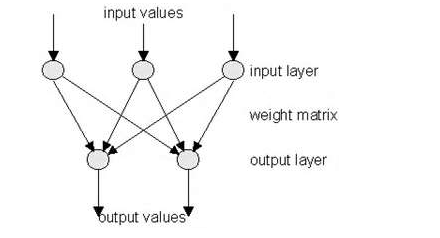
就像进化计算，神经网络又是一个类似的概念。神经网络由一个或者多个神经元组成。而一个神经元包括输入、输出和“内部处理器”。神经元从输入端接受信息，通过“内部处理器”将这些信息进行一定的处理，最后通过输出端输出。

**（2）感知器**

感知器（Perceptron），是神经网络中的一个概念，在1950s由Frank Rosenblatt第一次引入。

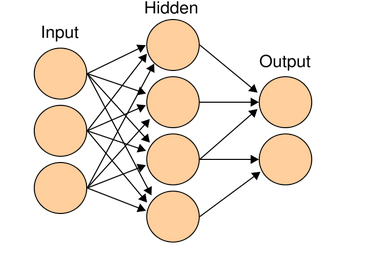
**（3）单层感知器**

单层感知器（Single Layer Perceptron）是最简单的神经网络。它包含输入层和输出层，而输入层和输出层是直接相连的。



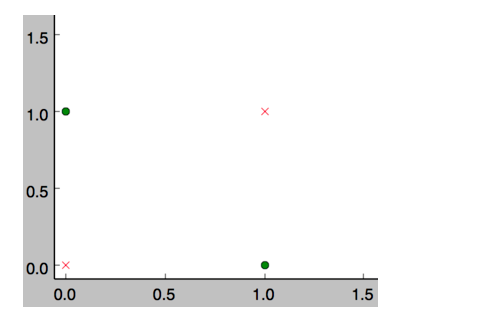
**（4）多层感知器**

多层感知器（Multi-Layer Perceptrons），包含多层计算。相对于单层感知器，输出端从一个变到了多个；输入端和输出端之间也不光只有一层，现在又两层:输出层和隐藏层。

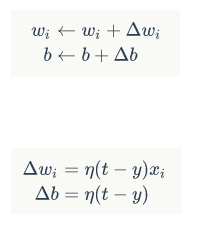


**2. 感知器用处**

用感知器可以实现 and 函数，or 函数，还可以拟合任何的线性函数，任何线性分类或线性回归问题都可以用感知器来解决。但是，感知器却不能实现异或运算，如下图所示，异或运算不是线性的，无法用一条直线把 0 和 1 分开。



训练权重和偏置的算法如下：



其中，t 是训练样本的实际值，y 是感知器的输出值，即由 f 计算出来的。eta 称为学习速率，是个常数，作用是控制每一步调整权的幅度。

**3. 感知器代码实现**

**（1）［main］**

先训练and感知器

and\_perception = train\_and\_perceptron()

得到训练后获得的权重和偏置

print and\_perception

weights :[0.1, 0.2]

bias :-0.200000

再去测试，看结果是否正确

print '1 and 1 = %d' % and\_perception.predict([1, 1]

**（2）［train\_and\_perceptron］**

先创建感知器，输入参数个数为2（因为and是二元函数），激活函数为f

p = Perceptron(2, f)

f 为

def f(x):

return 1 if x > 0 else 0

输入训练data，迭代10次, 学习速率为0.1

input\_vecs, labels = get\_training\_dataset()

p.train(input\_vecs, labels, 10, 0.1)

训练data为

input\_vecs = [[1,1], [0,0], [1,0], [0,1]]

labels = [1, 0, 0, 0]

**（3）［train］**

一共迭代 10 次，每次迭代时，

先计算感知器在当前权重下的输出，然后更新weights

output = self.predict(input\_vec)

self.\_update\_weights(input\_vec, output, label, rate)

**（4）［\_update\_weights］**

就是用训练算法里面的两个公式

delta = label - output

self.weights = map(

lambda (x, w): w + rate \* delta \* x,

zip(input\_vec, self.weights) )

self.bias += rate \* delta

**（5）［predict］**

就用感知器的函数 f：

return self.activator(

reduce(lambda a, b: a + b,

map(lambda (x, w): x \* w,

zip(input\_vec, self.weights))

, 0.0) + self.bias)

（6）完整代码：

**#!/usr/bin/python**

**#-\*-coding:utf-8 -\*-**

**class Perceptron(object):**

**def \_\_init\_\_(self, input\_num, activator):**

**'''**

**初始化感知器，设置输入参数的个数，以及激活函数。**

**激活函数的类型为double -> double**

**'''**

**self.activator = activator**

**# 权重向量初始化为0**

**self.weights = [0.0 for \_ in range(input\_num)]**

**# 偏置项初始化为0**

**self.bias = 0.0**

**def \_\_str\_\_(self):**

**'''**

**打印学习到的权重、偏置项**

**'''**

**return 'weights\t:%s\nbias\t:%f\n' % (self.weights, self.bias)**

**def predict(self, input\_vec):**

**'''**

**输入向量，输出感知器的计算结果**

**'''**

**# 把input\_vec[x1,x2,x3...]和weights[w1,w2,w3,...]打包在一起**

**# 变成[(x1,w1),(x2,w2),(x3,w3),...]**

**# 然后利用map函数计算[x1\*w1, x2\*w2, x3\*w3]**

**# 最后利用reduce求和**

**return self.activator(**

**reduce(lambda a, b: a + b,**

**map(lambda (x, w): x \* w,**

**zip(input\_vec, self.weights))**

**, 0.0) + self.bias)**

**def train(self, input\_vecs, labels, iteration, rate):**

**'''**

**输入训练数据：一组向量、与每个向量对应的label；以及训练轮数、学习率**

**'''**

**for i in range(iteration):**

**self.\_one\_iteration(input\_vecs, labels, rate)**

**def \_one\_iteration(self, input\_vecs, labels, rate):**

**'''**

**一次迭代，把所有的训练数据过一遍**

**'''**

**# 把输入和输出打包在一起，成为样本的列表[(input\_vec, label), ...]**

**# 而每个训练样本是(input\_vec, label)**

**samples = zip(input\_vecs, labels)**

**# 对每个样本，按照感知器规则更新权重**

**for (input\_vec, label) in samples:**

**# 计算感知器在当前权重下的输出**

**output = self.predict(input\_vec)**

**# 更新权重**

**self.\_update\_weights(input\_vec, output, label, rate)**

**def \_update\_weights(self, input\_vec, output, label, rate):**

**'''**

**按照感知器规则更新权重**

**'''**

**# 把input\_vec[x1,x2,x3,...]和weights[w1,w2,w3,...]打包在一起**

**# 变成[(x1,w1),(x2,w2),(x3,w3),...]**

**# 然后利用感知器规则更新权重**

**delta = label - output**

**self.weights = map(**

**lambda (x, w): w + rate \* delta \* x,**

**zip(input\_vec, self.weights) )**

**# 更新bias**

**self.bias += rate \* delta**

**def f(x):**

**'''**

**定义激活函数f**

**'''**

**return 1 if x > 0 else 0**

**def get\_training\_dataset():**

**'''**

**基于and真值表构建训练数据**

**'''**

**# 构建训练数据**

**# 输入向量列表**

**input\_vecs = [[1,1], [0,0], [1,0], [0,1]]**

**# 期望的输出列表，注意要与输入一一对应**

**# [1,1] -> 1, [0,0] -> 0, [1,0] -> 0, [0,1] -> 0**

**labels = [1, 0, 0, 0]**

**return input\_vecs, labels**

**def train\_and\_perceptron():**

**'''**

**使用and真值表训练感知器**

**'''**

**# 创建感知器，输入参数个数为2（因为and是二元函数），激活函数为f**

**p = Perceptron(2, f)**

**# 训练，迭代10轮, 学习速率为0.1**

**input\_vecs, labels = get\_training\_dataset()**

**p.train(input\_vecs, labels, 10, 0.1)**

**#返回训练好的感知器**

**return p**

**if \_\_name\_\_ == '\_\_main\_\_':**

**# 训练and感知器**

**and\_perception = train\_and\_perceptron()**

**# 打印训练获得的权重**

**print and\_perception**

**# 测试**

**print '1 and 1 = %d' % and\_perception.predict([1, 1])**

**print '0 and 0 = %d' % and\_perception.predict([0, 0])**

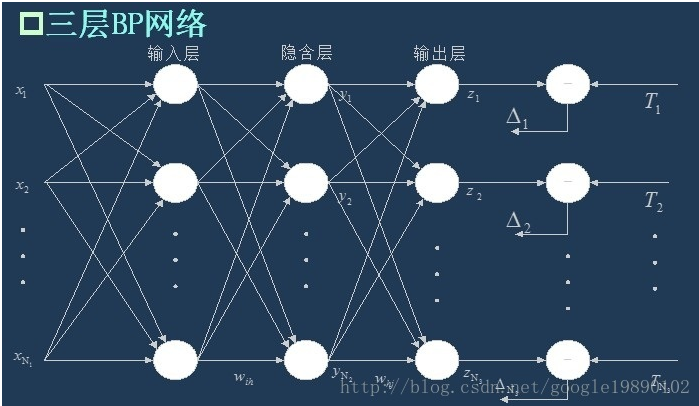
**print '1 and 0 = %d' % and\_perception.predict([1, 0])**

**print '0 and 1 = %d' % and\_perception.predict([0, 1])**

## BP神经网络

**1 BP概念**

**BP(Back Propagation)神经网络是一种具有三层或者三层以上的多层神经网络，每一层都由若干个神经元组成，它的左、右各层之间各个神经元实现全连接，即左层的每一个神经元与右层的每个神经元都由连接，而上下各神经元之间无连接。**BP神经网络按有导师学习方式进行训练，当一对学习模式提供给神经网络后，其神经元的激活值将从输入层经各隐含层向输出层传播，在输出层的各神经元输出对应于输入模式的网络响应。然后，按减少希望输出与实际输出误差的原则，从输出层经各隐含层，最后回到输入层（从右到左）逐层修正各连接权。由于这种修正过程是从输出到输入逐层进行的，所以称它为“误差逆传播算法”。随着这种误差逆传播训练的不断修正，网络对输入模式响应的正确率也将不断提高。BP神经网络是一种多层的前馈神经网络，其主要的特点是：信号是前向传播的，而误差是反向传播的。具体来说，对于如下的只含一个隐层的神经网络模型：



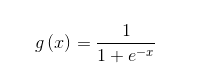
BP神经网络的过程主要分为两个阶段，第一阶段是信号的前向传播，从输入层经过隐含层，最后到达输出层；第二阶段是误差的反向传播，从输出层到隐含层，最后到输入层，依次调节隐含层到输出层的权重和偏置，输入层到隐含层的权重和偏置。

**2 BP神经网络的流程**

在知道了BP神经网络的特点后，我们需要依据信号的前向传播和误差的反向传播来构建整个网络。

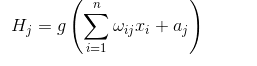
**（1）网络的初始化**

假设输入层的节点个数为，隐含层的节点个数为，输出层的节点个数为。输入层到隐含层的权重，隐含层到输出层的权重为，输入层到隐含层的偏置为，隐含层到输出层的偏置为。学习速率为，激励函数为。其中激励函数为取Sigmoid函数。形式为：

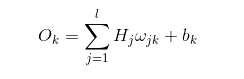


**（2）隐含层的输出**

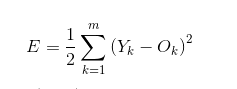
如上面的三层BP网络所示，隐含层的输出为



**（3）输出层的输出**



**（4）误差的计算**

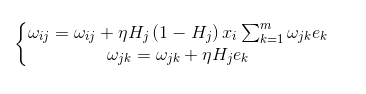
我们取误差公式为：

其中为期望输出。我们记，则可以表示为

以上公式中，，，。

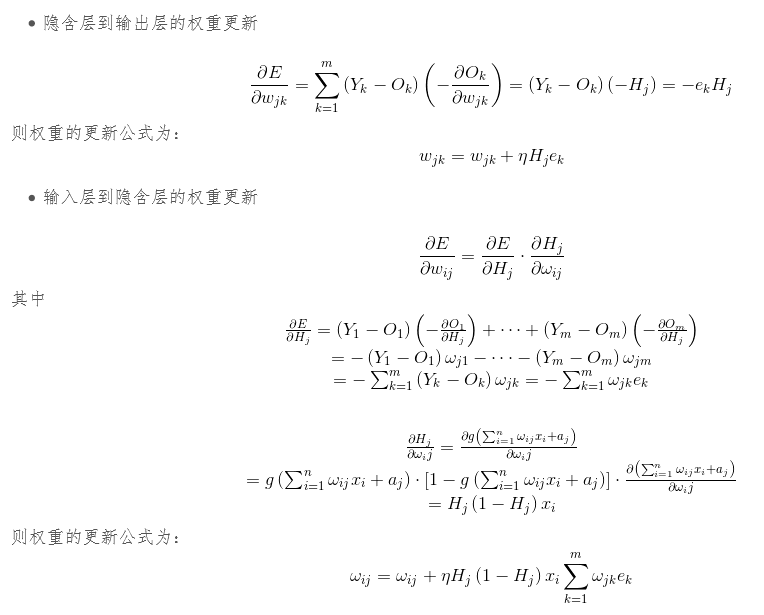
**（5）权值的更新**

权值的更新公式为：

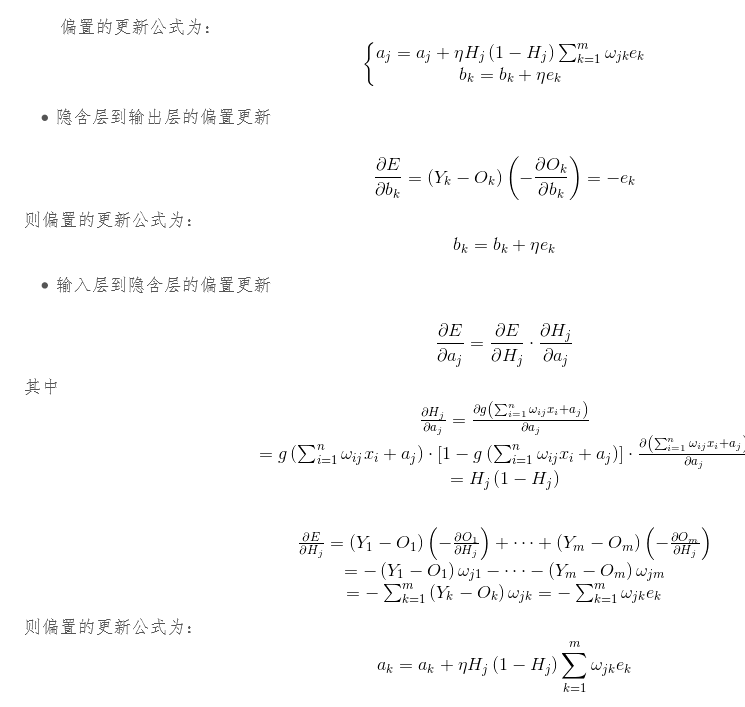


这里需要解释一下公式的由来：

这是误差反向传播的过程，我们的目标是使得误差函数达到最小值，即，我们使用梯度下降法：



**（6）偏置的更新**



**（7）判断算法迭代是否结束**

有很多的方法可以判断算法是否已经收敛，常见的有指定迭代的代数，判断相邻的两次误差之间的差别是否小于指定的值等等。

**3 实验的仿真**

在本试验中，我们利用BP神经网络处理一个四分类问题，最终的分类结果为：

主程序

<span style="font-size:14px;">%% BP的主函数

% 清空

clear all;

clc;

% 导入数据

load data;

%从1到2000间随机排序

k=rand(1,2000);

[m,n]=sort(k);

%输入输出数据

input=data(:,2:25);

output1 =data(:,1);

%把输出从1维变成4维

for i=1:2000

switch output1(i)

case 1

output(i,:)=[1 0 0 0];

case 2

output(i,:)=[0 1 0 0];

case 3

output(i,:)=[0 0 1 0];

case 4

output(i,:)=[0 0 0 1];

end

end

%随机提取1500个样本为训练样本，500个样本为预测样本

trainCharacter=input(n(1:1600),:);

trainOutput=output(n(1:1600),:);

testCharacter=input(n(1601:2000),:);

testOutput=output(n(1601:2000),:);

% 对训练的特征进行归一化

[trainInput,inputps]=mapminmax(trainCharacter');

%% 参数的初始化

% 参数的初始化

inputNum = 24;%输入层的节点数

hiddenNum = 50;%隐含层的节点数

outputNum = 4;%输出层的节点数

% 权重和偏置的初始化

w1 = rands(inputNum,hiddenNum);

b1 = rands(hiddenNum,1);

w2 = rands(hiddenNum,outputNum);

b2 = rands(outputNum,1);

% 学习率

yita = 0.1;

%% 网络的训练

for r = 1:30

E(r) = 0;% 统计误差

for m = 1:1600

% 信息的正向流动

x = trainInput(:,m);

% 隐含层的输出

for j = 1:hiddenNum

hidden(j,:) = w1(:,j)'\*x+b1(j,:);

hiddenOutput(j,:) = g(hidden(j,:));

end

% 输出层的输出

outputOutput = w2'\*hiddenOutput+b2;

% 计算误差

e = trainOutput(m,:)'-outputOutput;

E(r) = E(r) + sum(abs(e));

% 修改权重和偏置

% 隐含层到输出层的权重和偏置调整

dw2 = hiddenOutput\*e';

db2 = e;

% 输入层到隐含层的权重和偏置调整

for j = 1:hiddenNum

partOne(j) = hiddenOutput(j)\*(1-hiddenOutput(j));

partTwo(j) = w2(j,:)\*e;

end

for i = 1:inputNum

for j = 1:hiddenNum

dw1(i,j) = partOne(j)\*x(i,:)\*partTwo(j);

db1(j,:) = partOne(j)\*partTwo(j);

end

end

w1 = w1 + yita\*dw1;

w2 = w2 + yita\*dw2;

b1 = b1 + yita\*db1;

b2 = b2 + yita\*db2;

end

end

%% 语音特征信号分类

testInput=mapminmax('apply',testCharacter',inputps);

for m = 1:400

for j = 1:hiddenNum

hiddenTest(j,:) = w1(:,j)'\*testInput(:,m)+b1(j,:);

hiddenTestOutput(j,:) = g(hiddenTest(j,:));

end

outputOfTest(:,m) = w2'\*hiddenTestOutput+b2;

end

%% 结果分析

%根据网络输出找出数据属于哪类

for m=1:400

output\_fore(m)=find(outputOfTest(:,m)==max(outputOfTest(:,m)));

end

%BP网络预测误差

error=output\_fore-output1(n(1601:2000))';

k=zeros(1,4);

%找出判断错误的分类属于哪一类

for i=1:400

if error(i)~=0

[b,c]=max(testOutput(i,:));

switch c

case 1

k(1)=k(1)+1;

case 2

k(2)=k(2)+1;

case 3

k(3)=k(3)+1;

case 4

k(4)=k(4)+1;

end

end

end

%找出每类的个体和

kk=zeros(1,4);

for i=1:400

[b,c]=max(testOutput(i,:));

switch c

case 1

kk(1)=kk(1)+1;

case 2

kk(2)=kk(2)+1;

case 3

kk(3)=kk(3)+1;

case 4

kk(4)=kk(4)+1;

end

end

%正确率

rightridio=(kk-k)./kk </span>

激活函数

<span style="font-size:14px;">%% 激活函数

function [ y ] = g( x )

y = 1./(1+exp(-x));

end </span>

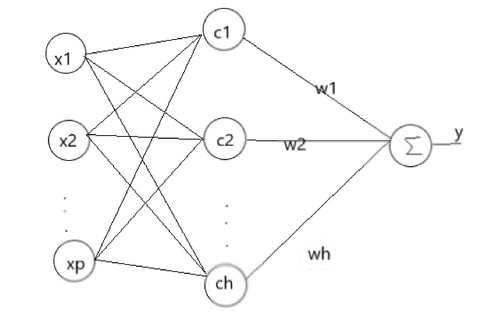
## RBF径向基神经网络

**1 什么是径向基函数**

1985年，Powell提出了多变量插值的径向基函数（RBF）方法。**径向基函数是一个取值仅仅依赖于离原点距离的实值函数，也就是Φ（x）=Φ(‖x‖),或者还可以是到任意一点c的距离，c点称为中心点，也就是Φ（x，c）=Φ(‖x-c‖)。**任意一个满足Φ（x）=Φ(‖x‖)特性的函数Φ都叫做径向基函数，标准的一般使用欧氏距离（也叫做欧式径向基函数），尽管其他距离函数也是可以的。最常用的径向基函数是高斯核函数 ,形式为 k(||x-xc||)=exp{- ||x-xc||^2/(2\*σ)^2) } 其中x\_c为核函数中心,σ为函数的宽度参数 , 控制了函数的径向作用范围。

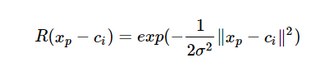
**2 RBF神经网络**

RBF神将网络是一种三层神经网络，其包括输入层、隐层、输出层。从输入空间到隐层空间的变换是非线性的，而从隐层空间到输出层空间变换是线性的。流图如下：

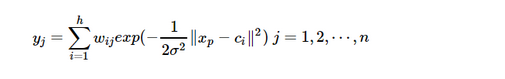


RBF网络的基本思想是：用RBF作为隐单元的“基”构成隐含层空间，这样就可以将输入矢量直接映射到隐空间，而不需要通过权连接。当RBF的中心点确定以后，这种映射关系也就确定了。而隐含层空间到输出空间的映射是线性的，即网络的输出是隐单元输出的线性加权和，此处的权即为网络可调参数。其中，隐含层的作用是把向量从低维度的p映射到高维度的h，这样低维度线性不可分的情况到高维度就可以变得线性可分了，主要就是核函数的思想。这样，网络由输入到输出的映射是非线性的，而网络输出对可调参数而言却又是线性的。网络的权就可由线性方程组直接解出，从而大大加快学习速度并避免局部极小问题。

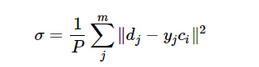
径向基神经网络的激活函数可表示为：



其中xp径向基神经网络的结构可得到网络的输出为：



当然，采用最小二乘的损失函数表示：



**3 RBF神经网络的学习问题**

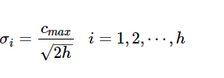
求解的参数有3个：基函数的中心、方差以及隐含层到输出层的权值。

**（1）自组织选取中心学习方法**

　　第一步：无监督学习过程，求解隐含层基函数的中心与方差

　　第二步：有监督学习过程，求解隐含层到输出层之间的权值

首先，选取h个中心做k-means聚类，对于高斯核函数的径向基，方差由公式求解：



　　cmax为所选取中心点之间的最大距离。

隐含层至输出层之间的神经元的连接权值可以用最小二乘法直接计算得到，即对损失函数求解关于w的偏导数，使其等于0，可以化简得到计算公式为：



**（2）直接计算法**

隐含层神经元的中心是随机地在输入样本中选取，且中心固定。一旦中心固定下来，隐含层神经元的输出便是已知的，这样的神经网络的连接权就可以通过求解线性方程组来确定。适用于样本数据的分布具有明显代表性。

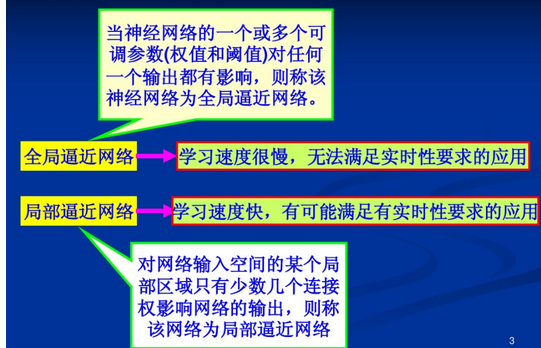
**（3）有监督学习算法**

通过训练样本集来获得满足监督要求的网络中心和其他权重参数，经历一个误差修正学习的过程，与BP网络的学习原理一样，同样采用梯度下降法。因此RBF同样可以被当作BP神经网络的一种。

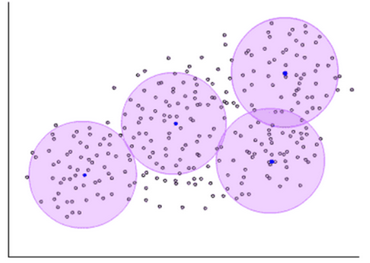
**4 RBF神经网络与BP神经网络之间的区别**

**（1）局部逼近与全局逼近：**

BP神经网络的隐节点采用输入模式与权向量的内积作为激活函数的自变量，而激活函数采用Sigmoid函数。各调参数对BP网络的输出具有同等地位的影响，因此BP神经网络是对非线性映射的全局逼近。RBF神经网络的隐节点采用输入模式与中心向量的距离（如欧式距离）作为函数的自变量，并使用径向基函数（如Gaussian函数）作为激活函数。神经元的输入离径向基函数中心越远，神经元的激活程度就越低（高斯函数）。RBF网络的输出与部分调参数有关，譬如，一个wij值只影响一个yi的输出（参考上面第二章网络输出），RBF神经网络因此具有“局部映射”特性。



所谓局部逼近是指目标函数的逼近仅仅根据查询点附近的数据。而事实上，对于径向基网络，通常使用的是高斯径向基函数，函数图象是两边衰减且径向对称的，当选取的中心与查询点（即输入数据）很接近的时候才对输入有真正的映射作用，若中心与查询点很远的时候，欧式距离太大的情况下，输出的结果趋于0，所以真正起作用的点还是与查询点很近的点，所以是局部逼近；而BP网络对目标函数的逼近跟所有数据都相关，而不仅仅来自查询点附近的数据。

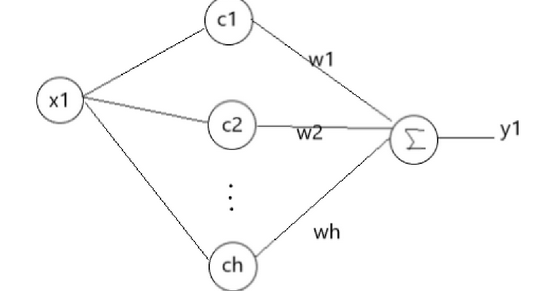


**（2）中间层数的区别**

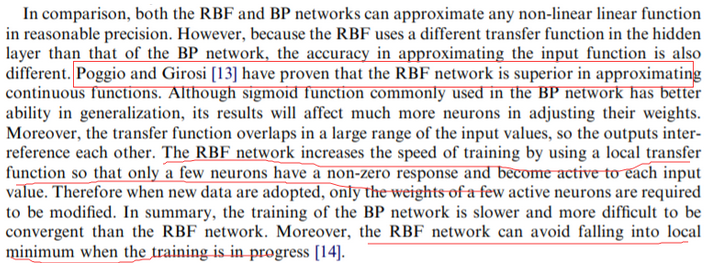
BP神经网络可以有多个隐含层，但是RBF只有一个隐含层。

**（3）训练速度的区别**

使用RBF的训练速度快，一方面是因为隐含层较少，另一方面，局部逼近可以简化计算量。对于一个输入x，只有部分神经元会有响应，其他的都近似为0，对应的w就不用调参了。

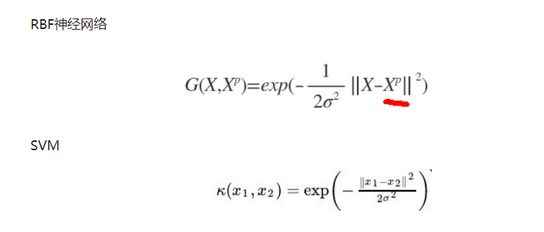


**（4）Poggio和Girosi已经证明，RBF网络是连续函数的最佳逼近，而BP网络不是。**

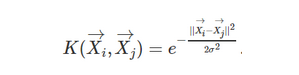


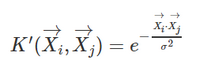
**5 RBF神经网络与SVM的区别**

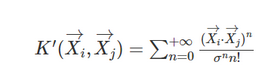
SVM等如果使用核函数的技巧的话，不太适应于大样本和大的特征数的情况，因此提出了RBF。另外，SVM中的高斯核函数可以看作与每一个输入点的距离，而RBF神经网络对输入点做了一个聚类。RBF神经网络用高斯核函数时,其数据中心C可以是训练样本中的抽样，此时与svm的高斯核函数是完全等价的，也可以是训练样本集的多个聚类中心，所以他们都是需要选择数据中心的，只不过SVM使用高斯核函数时，这里的数据中心都是训练样本本身而已。



**6 为什么高斯核函数就是映射到高维空间**

首先给出高斯核函数的定义公式：

实际上，可以化简为：

当然通过幂级数展开：

可以看到，其中X向量会生成类似多项式核展开的形式，譬如原来的参数有x1,x2。映射后，参数包含了x1\*x1 ,x1\*x2,x2\*x2将原来2维映射到3维上了。

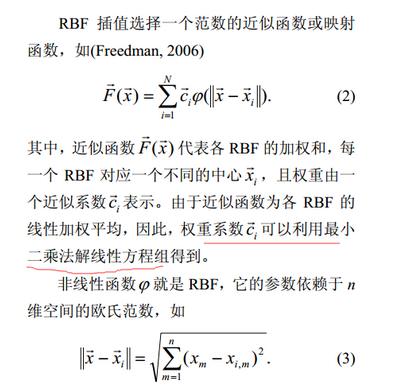
**7 前馈网络、递归网络和反馈网络**

前馈网络一般指前馈神经网络或前馈型神经网络。它是一种最简单的神经网络，各神经元分层排列。每个神经元只与前一层的神经元相连。接收前一层的输出，并输出给下一层，各层间没有反馈。包括：BP神经网络、RBF神经网络等。**递归神经网络（RNN）是两种人工神经网络的总称。**一种是时间递归神经网络（recurrent neural network），又名循环神经网络，包括RNN、LSTM、GRU等；另一种是结构递归神经网络（recursive neural network）。

反馈网络(Recurrent Network)，又称自联想记忆网络，其目的是为了设计一个网络，储存一组平衡点，使得当给网络一组初始值时，网络通过自行运行而最终收敛到这个设计的平衡点上。包括CHNN、DHNN等。

**8 完全内插法**

之所以RBF能够拟合任意函数，可以从内插法的角度去理解。要拟合一个曲线，我们可以通过内插法获得这个曲线的表达函数，譬如：多项式插值、拉格朗日插值等。RBF 插值是一系列精确插值方法的组合；即表面必须通过每一个测得的采样值。



对于RBF插值，其特点即为，在输入数据集中，与中心点距离近的点对映射函数的贡献最大。　　完全内插法即要求所有插值点都经过曲面，由于RBF内插对于每个x都有用到，所以是一种完全内插的形式，存在的问题就是当样本中包含噪声时，神经网络将拟合出一个错误的曲面，从而使泛化能力下降。另外，若样本x的数据远大于非线性函数φ，该求解变得不稳定，即为解超定方程。因此需要引入正则化方法，正则化的方法即通常加上正则化项。

**9 RBF总结**

（1）输入层到隐藏层之间不是通过权值和阈值进行连接的，而是通过输入样本与隐藏层点之间的距离（与中心点的距离）连接的。

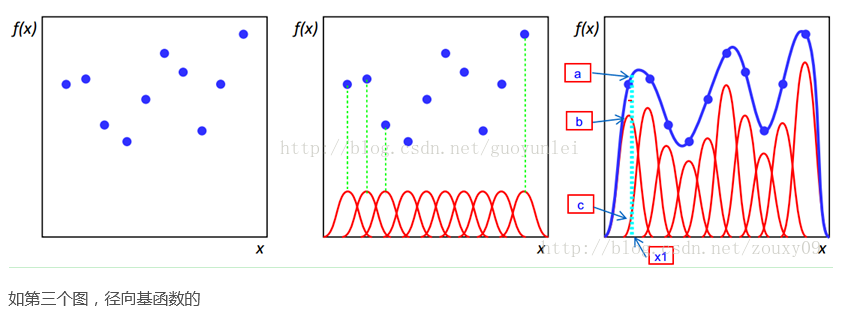
（2）得到距离之后，将距离代入径向基函数，得到一个数值。数值再与后边的权值相乘再求总和，就得到了相应输入的输出。

（3）在训练网络之前，需要确定中心点的个数，和中心点的位置。以及求出隐藏层各径向基函数的方差（宽窄程度）。和隐藏层和输出层之间的权值。

（4）中心点个数、中心点位置、方差、权值都可以通过下文所述的方法求出来。

（5）径向基函数也是一种基，可以通过对其线性组合，来对非线性函数进行拟合。

（6）RBF神经网络只需要了解其中的原理。然后给了训练数据，求出上述几个参数，再输入测试数据，就可以预测输出了。



**10 代码实现**

#include<iostream>

#include<algorithm>

#include<limits>

#include<cassert>

#include<cmath>

#include<ctime>

#include<cstdlib>

#include<vector>

#include<iomanip>

#include"matrix.h"

using namespace std;

const int P=100;

//输入样本的数量

vector<double>

X(P); //输入样本

Matrix<double>

Y(P,1); //输入样本对应的期望输出

const int M=10;

//隐藏层节点数目

vector<double>

center(M); //M个Green函数的数据中心

vector<double>

delta(M); //M个Green函数的扩展常数

Matrix<double>

Green(P,M); //Green矩阵

Matrix<double>

Weight(M,1); //权值矩阵

/\*Hermit多项式函数\*/

inline double Hermit(double x){

return 1.1\*(1-x+2\*x\*x)\*exp(-1\*x\*x/2);

}

/\*产生指定区间上均匀分布的随机数\*/

inline double uniform(double floor,double ceil){

return floor+1.0\*rand()/RAND\_MAX\*(ceil-floor);

}

/\*产生区间[floor,ceil]上服从正态分布N[mu,sigma]的随机数\*/

inline double RandomNorm(double mu,double sigma,double floor,double ceil){

double x,prob,y;

do{

x=uniform(floor,ceil);

prob=1/sqrt(2\*M\_PI\*sigma)\*exp(-1\*(x-mu)\*(x-mu)/(2\*sigma\*sigma));

y=1.0\*rand()/RAND\_MAX;

}while(y>prob);

return x;

}

/\*产生输入样本\*/

void generateSample(){

for(int i=0;i<P;++i){

double in=uniform(-4,4);

X[i]=in;

Y.put(i,0,Hermit(in)+RandomNorm(0,0.1,-0.3,0.3));

}

}

/\*寻找样本离哪个中心最近\*/

int nearest(const vector<double>&

center,double sample){

int rect=-1;

double dist=numeric\_limits<double>::max();

for(int i=0;i<center.size();++i){

if(fabs(sample-center[i])<dist){

dist=fabs(sample-center[i]);

rect=i;

}

}

return rect;

}

/\*计算簇的质心\*/

double calCenter(const vector<double>

&g){

int len=g.size();

double sum=0.0;

for(int i=0;i<len;++i)

sum+=g[i];

return sum/len;

}

/\*KMeans聚类法产生数据中心\*/

void KMeans(){

assert(P%M==0);

vector<vector<double>

> group(M); //记录各个聚类中包含哪些样本

double gap=0.001;

//聚类中心的改变量小于为个值时，迭代终止

for(int i=0;i<M;++i){

//从P个输入样本中随机选P个作为初始聚类中心

center[i]=X[10\*i+3];

//输入是均匀分布的，所以我们均匀地选取

}

while(1){

for(int i=0;i<M;++i)

group[i].clear();

//先清空聚类信息

for(int i=0;i<P;++i){

//把所有输入样本归到对应的簇

int c=nearest(center,X[i]);

group[c].push\_back(X[i]);

}

vector<double>

new\_center(M); //存储新的簇心

for(int i=0;i<M;++i){

vector<double>

g=group[i];

new\_center[i]=calCenter(g);

}

bool flag=false;

for(int i=0;i<M;++i){

//检查前后两次质心的改变量是否都小于gap

if(fabs(new\_center[i]-center[i])>gap){

flag=true;

break;

}

}

center=new\_center;

if(!flag)

break;

}

}

/\*生成Green矩阵\*/

void calGreen(){

for(int i=0;i<P;++i){

for(int j=0;j<M;++j){

Green.put(i,j,exp(-1.0\*(X[i]-center[j])\*(X[i]-center[j])/(2\*delta[j]\*delta[j])));

}

}

}

/\*求一个矩阵的伪逆\*/

Matrix<double>

getGereralizedInverse(const Matrix<double>

&matrix){

return (matrix.getTranspose()\*matrix).getInverse()\*(matrix.getTranspose());

}

/\*利用已训练好的神经网络，由输入得到输出\*/

double getOutput(double x){

double y=0.0;

for(int i=0;i<M;++i)

y+=Weight.get(i,0)\*exp(-1.0\*(x-center[i])\*(x-center[i])/(2\*delta[i]\*delta[i]));

return y;

}

int main(int argc,char \*argv[]){<br>

srand(time(0));

generateSample();

//产生输入和对应的期望输出样本

KMeans();

//对输入进行聚类，产生聚类中心

sort(center.begin(),center.end());

//对聚类中心（一维数据）进行排序

//根据聚类中心间的距离，计算各扩展常数

delta[0]=center[1]-center[0];

delta[M-1]=center[M-1]-center[M-2];

for(int i=1;i<M-1;++i){

double d1=center[i]-center[i-1];

double d2=center[i+1]-center[i];

delta[i]=d1<d2?d1:d2;

}

calGreen();

//计算Green矩阵

Weight=getGereralizedInverse(Green)\*Y;

//计算权值矩阵

//根据已训练好的神经网络作几组测试

for(int x=-4;x<5;++x){

cout<<x<<"\t";

cout<<setprecision(8)<<setiosflags(ios::left)<<setw(15);

cout<<getOutput(x)<<Hermit(x)<<endl;

//先输出我们预测的值，再输出真实值

}

return 0;

}

## 卷积神经网络(CNN)

**1 卷积神经网络的基本概念**

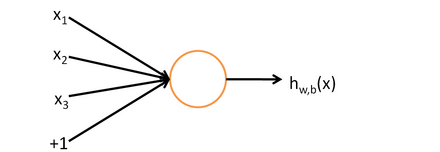
受Hubel和Wiesel对猫视觉皮层电生理研究启发，有人提出卷积神经网络（CNN），Yann Lecun 最早将CNN用于手写数字识别并一直保持了其在该问题的霸主地位。近年来卷积神经网络在多个方向持续发力，在语音识别、人脸识别、通用物体识别、运动分析、自然语言处理甚至脑电波分析方面均有突破。

**卷积神经网络与普通神经网络的区别在于，卷积神经网络包含了一个由卷积层和子采样层构成的特征抽取器。**在卷积神经网络的卷积层中，一个神经元只与部分邻层神经元连接。在CNN的一个卷积层中，通常包含若干个特征平面(featureMap)，每个特征平面由一些矩形排列的的神经元组成，同一特征平面的神经元共享权值，这里共享的权值就是卷积核。卷积核一般以随机小数矩阵的形式初始化，在网络的训练过程中卷积核将学习得到合理的权值。共享权值（卷积核）带来的直接好处是减少网络各层之间的连接，同时又降低了过拟合的风险。子采样也叫做池化（pooling），通常有均值子采样（mean pooling）和最大值子采样（max pooling）两种形式。子采样可以看作一种特殊的卷积过程。卷积和子采样大大简化了模型复杂度，减少了模型的参数。

**2 卷积神经网络的原理**

**（1）神经网络**

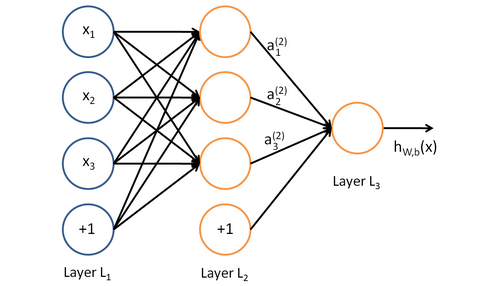
首先介绍神经网络，这一步的详细可以参考资源1。简要介绍下。神经网络的每个单元如下：



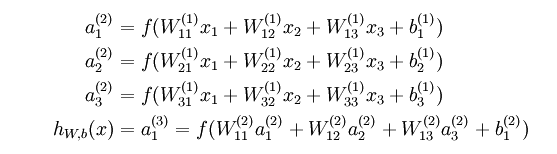
其对应的公式如下：



其中，该单元也可以被称作是Logistic回归模型。当将多个单元组合起来并具有分层结构时，就形成了神经网络模型。下图展示了一个具有一个隐含层的神经网络。



其对应的公式如下：

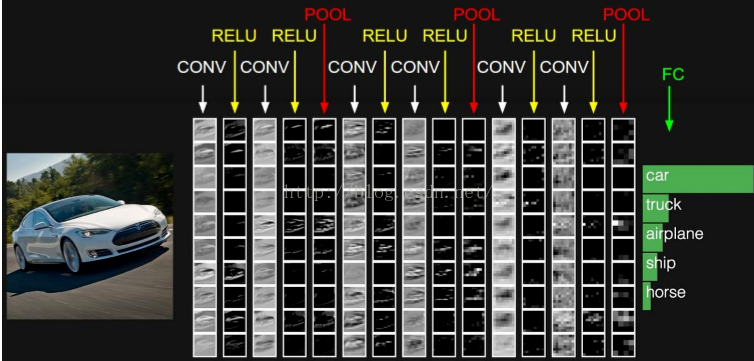


比较类似的，可以拓展到有2,3,4,5，…个隐含层。神经网络的训练方法也同Logistic类似，不过由于其多层性，还需要利用链式求导法则对隐含层的节点进行求导，即梯度下降+链式求导法则，专业名称为反向传播。关于训练算法，本文暂不涉及。

**（2）卷积神经网络**

受Hubel和Wiesel对猫视觉皮层电生理研究启发，有人提出卷积神经网络（CNN），Yann Lecun 最早将CNN用于手写数字识别并一直保持了其在该问题的霸主地位。近年来卷积神经网络在多个方向持续发力，在语音识别、人脸识别、通用物体识别、运动分析、自然语言处理甚至脑电波分析方面均有突破。

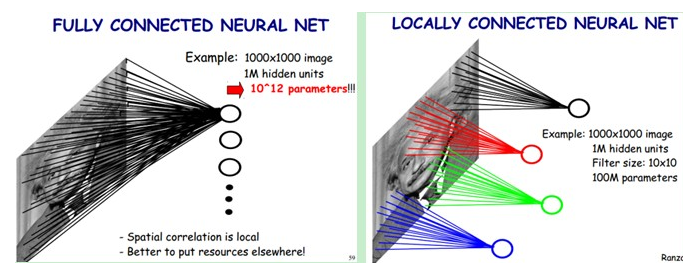
卷积神经网络与普通神经网络的区别在于，卷积神经网络包含了一个由卷积层和子采样层构成的特征抽取器。在卷积神经网络的卷积层中，一个神经元只与部分邻层神经元连接。在CNN的一个卷积层中，通常包含若干个特征平面(featureMap)，每个特征平面由一些矩形排列的的神经元组成，同一特征平面的神经元共享权值，这里共享的权值就是卷积核。卷积核一般以随机小数矩阵的形式初始化，在网络的训练过程中卷积核将学习得到合理的权值。共享权值（卷积核）带来的直接好处是减少网络各层之间的连接，同时又降低了过拟合的风险。子采样也叫做池化（pooling），通常有均值子采样（mean pooling）和最大值子采样（max pooling）两种形式。子采样可以看作一种特殊的卷积过程。卷积和子采样大大简化了模型复杂度，减少了模型的参数。卷积神经网络的基本结构如图所示：



**卷积神经网络由三部分构成。第一部分是输入层。第二部分由n个卷积层和池化层的组合组成。第三部分由一个全连结的多层感知机分类器构成**

**（3）局部感受野**

卷积神经网络有两种神器可以降低参数数目，第一种神器叫做局部感知野。一般认为人对外界的认知是从局部到全局的，而图像的空间联系也是局部的像素联系较为紧密，而距离较远的像素相关性则较弱。因而，每个神经元其实没有必要对全局图像进行感知，只需要对局部进行感知，然后在更高层将局部的信息综合起来就得到了全局的信息。网络部分连通的思想，也是受启发于生物学里面的视觉系统结构。视觉皮层的神经元就是局部接受信息的（即这些神经元只响应某些特定区域的刺激）。如下图所示：左图为全连接，右图为局部连接。

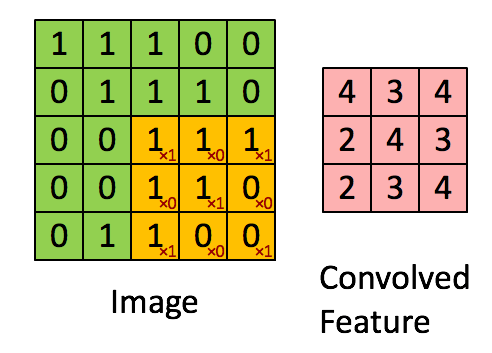
在上右图中，假如每个神经元只和10×10个像素值相连，那么权值数据为1000000×100个参数，减少为原来的万分之一。而那10×10个像素值对应的10×10个参数，其实就相当于卷积操作。

**（4）权值共享**

但其实这样的话参数仍然过多，那么就启动第二级神器，即权值共享。在上面的局部连接中，每个神经元都对应100个参数，一共1000000个神经元，如果这1000000个神经元的100个参数都是相等的，那么参数数目就变为100了。

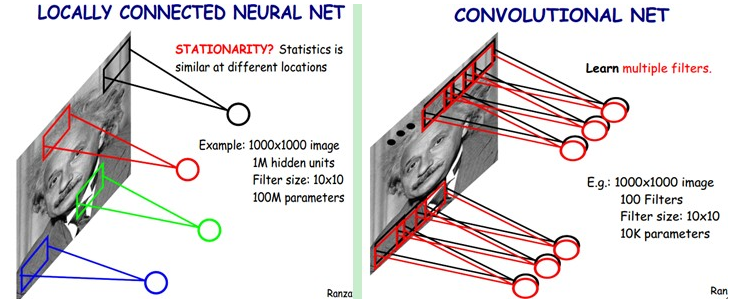
怎么理解权值共享呢？我们可以这100个参数（也就是卷积操作）看成是提取特征的方式，该方式与位置无关。这其中隐含的原理则是：图像的一部分的统计特性与其他部分是一样的。这也意味着我们在这一部分学习的特征也能用在另一部分上，所以对于这个图像上的所有位置，我们都能使用同样的学习特征。更直观一些，当从一个大尺寸图像中随机选取一小块，比如说 8x8 作为样本，并且从这个小块样本中学习到了一些特征，这时我们可以把从这个 8x8 样本中学习到的特征作为探测器，应用到这个图像的任意地方中去。特别是，我们可以用从 8x8 样本中所学习到的特征跟原本的大尺寸图像作卷积，从而对这个大尺寸图像上的任一位置获得一个不同特征的激活值。

如下图所示，展示了一个3×3的卷积核在5×5的图像上做卷积的过程。每个卷积都是一种特征提取方式，就像一个筛子，将图像中符合条件（激活值越大越符合条件）的部分筛选出来。

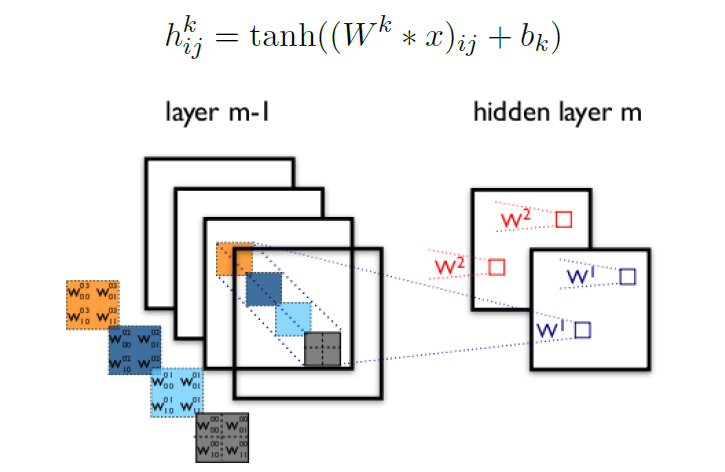


**（5）多卷积核**

上面所述只有100个参数时，表明只有1个10\*10的卷积核，显然，特征提取是不充分的，我们可以添加多个卷积核，比如32个卷积核，可以学习32种特征。在有多个卷积核时，如下图所示：

上图右，不同颜色表明不同的卷积核。每个卷积核都会将图像生成为另一幅图像。比如两个卷积核就可以将生成两幅图像，这两幅图像可以看做是一张图像的不同的通道。如下图所示，下图有个小错误，即将w1改为w0，w2改为w1即可。下文中仍以w1和w2称呼它们。

下图展示了在四个通道上的卷积操作，有两个卷积核，生成两个通道。其中需要注意的是，四个通道上每个通道对应一个卷积核，先将w2忽略，只看w1，那么在w1的某位置（i,j）处的值，是由四个通道上（i,j）处的卷积结果相加然后再取激活函数值得到的。

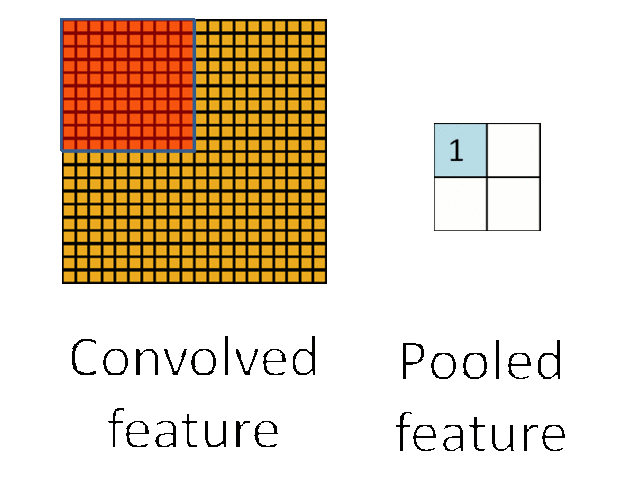


所以，在上图由4个通道卷积得到2个通道的过程中，参数的数目为4×2×2×2个，其中4表示4个通道，第一个2表示生成2个通道，最后的2×2表示卷积核大小。

**（6）Down-pooling**

在通过卷积获得了特征 (features) 之后，下一步我们希望利用这些特征去做分类。理论上讲，人们可以用所有提取得到的特征去训练分类器，例如 softmax 分类器，但这样做面临计算量的挑战。例如：对于一个 96X96 像素的图像，假设我们已经学习得到了400个定义在8X8输入上的特征，每一个特征和图像卷积都会得到一个 (96 − 8 + 1) × (96 − 8 + 1) = 7921 维的卷积特征，由于有 400 个特征，所以每个样例 (example) 都会得到一个 7921 × 400 = 3,168,400 维的卷积特征向量。学习一个拥有超过 3 百万特征输入的分类器十分不便，并且容易出现过拟合 (over-fitting)。

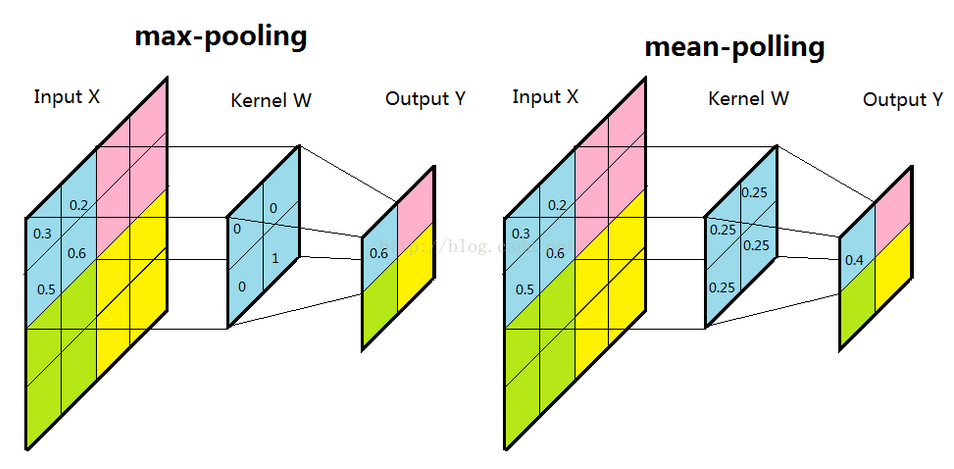
为了解决这个问题，首先回忆一下，我们之所以决定使用卷积后的特征是因为图像具有一种“静态性”的属性，这也就意味着在一个图像区域有用的特征极有可能在另一个区域同样适用。因此，为了描述大的图像，一个很自然的想法就是对不同位置的特征进行聚合统计，例如，人们可以计算图像一个区域上的某个特定特征的平均值 (或最大值)。这些概要统计特征不仅具有低得多的维度 (相比使用所有提取得到的特征)，同时还会改善结果(不容易过拟合)。这种聚合的操作就叫做池化 (pooling)，有时也称为平均池化或者最大池化 (取决于计算池化的方法)。



子采样有两种形式，一种是均值子采样（mean-pooling），一种是最大值子采样（max-pooling）。两种子采样看成特殊的卷积过程，如图下图所示：

1)均值子采样的卷积核中每个权重都是0.25，卷积核在原图inputX上的滑动的步长为2。均值子采样的效果相当于把原图模糊缩减至原来的1/4。

2)最大值子采样的卷积核中各权重值中只有一个为1，其余均为0，卷积核中为1的位置对应inputX被卷积核覆盖部分值最大的位置。卷积核在原图inputX上的滑动步长为2。最大值子采样的效果是把原图缩减至原来的1/4，并保留每个2\*2区域的最强输入。



至此，卷积神经网络的基本结构和原理已经阐述完毕。

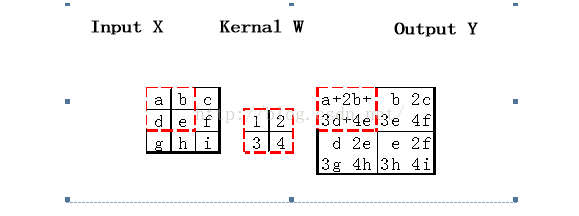
**（7）多卷积层**

在实际应用中，往往使用多层卷积，然后再使用全连接层进行训练，多层卷积的目的是一层卷积学到的特征往往是局部的，层数越高，学到的特征就越全局化。

**3 卷积神经网络的训练**

**（1）Forward前向传播**

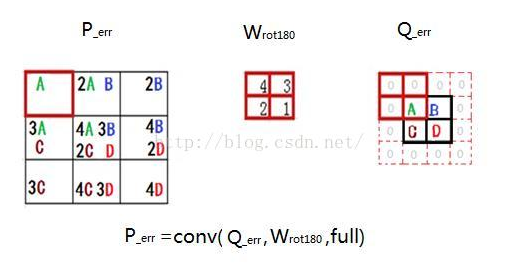
前向过程的卷积为典型valid的卷积过程，即卷积核kernalW覆盖在输入图inputX上，对应位置求积再求和得到一个值并赋给输出图OutputY对应的位置。每次卷积核在inputX上移动一个位置，从上到下从左到右交叠覆盖一遍之后得到输出矩阵outputY(如图4.1与图4.3所示)。如果卷积核的输入图inputX为Mx\*Nx大小，卷积核为Mw\*Nw大小，那么输出图Y为（Mx-Mw+1）\*（Nx-Nw+1）大小。



**（2）BackForward反向传播**

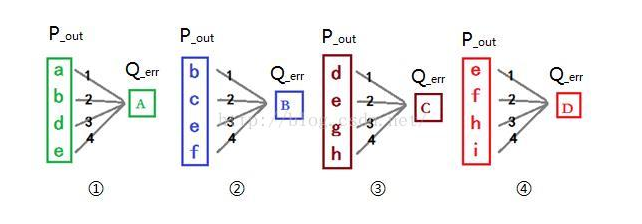
在错误信号反向传播过程中，先按照神经网络的错误反传方式得到尾部分类器中各神经元的错误信号，然后错误信号由分类器向前面的特征抽取器传播。错误信号从子采样层的特征图（subFeatureMap）往前面卷积层的特征图（featureMap）传播要通过一次full卷积过程来完成。这里的卷积和上一节卷积的略有区别。如果卷积核kernalW的长度为Mw\*Mw的方阵，那么subFeatureMap的错误信号矩阵Q\_err需要上下左右各拓展Mw-1行或列，与此同时卷积核自身旋转180度。subFeatureMap的错误信号矩阵P\_err等于featureMap的误差矩阵Q\_err卷积旋转180度的卷积核W\_rot180。

下图错误信号矩阵Q\_err中的A，它的产生是P中左上2\*2小方块导致的，该2\*2的小方块的对A的责任正好可以用卷积核W表示，错误信号A通过卷积核将错误信号加权传递到与错误信号量为A的神经元所相连的神经元a、b、d、e中，所以在下图中的P\_err左上角的2\*2位置错误值包含A、2A、3A、4A。同理，我们可以论证错误信号B、C、D的反向传播过程。综上所述，错误信号反向传播过程可以用下图中的卷积过程表示。

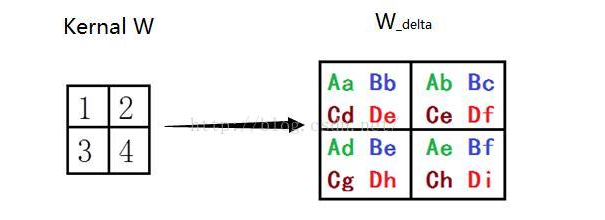


**（3）权值更新过程中的卷积**

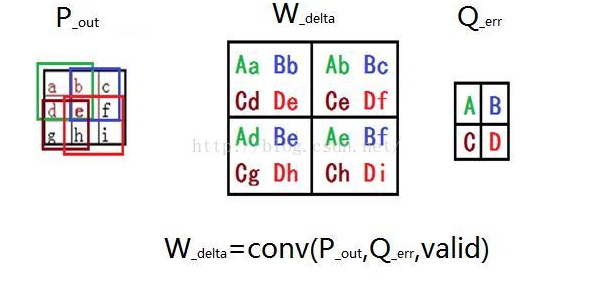
卷积神经网络中卷积层的权重更新过程本质是卷积核的更新过程。由神经网络的权重修改策略我们知道一条连接权重的更新量为该条连接的前层神经元的兴奋输出乘以后层神经元的输入错误信号，卷积核的更新也是按照这个规律来进行。



在前向卷积过程中，卷积核的每个元素（链接权重）被使用过四次，所以卷积核每个元素的产生四个更新量。把前向卷积过程当做切割小图进行多个神经网络训练过程，我们得到四个4\*1的神经网络的前层兴奋输入和后层输入错误信号，如图所示。



根据神经网络的权重修改策略，我们可以算出如图所示卷积核的更新量W\_delta。权重更新量W\_delta可由P\_out和Q\_err卷积得到，如图下图所示。

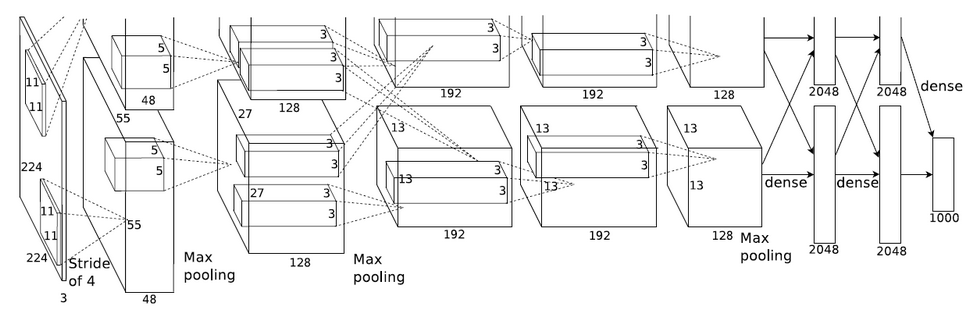


**4 常见网络结构**

**（1）ImageNet-2010网络结构**

ImageNet LSVRC是一个图片分类的比赛，其训练集包括127W+张图片，验证集有5W张图片，测试集有15W张图片。本文截取2010年Alex Krizhevsky的CNN结构进行说明，该结构在2010年取得冠军，top-5错误率为15.3%。值得一提的是，在今年的ImageNet LSVRC比赛中，取得冠军的GoogNet已经达到了top-5错误率6.67%。可见，深度学习的提升空间还很巨大。

下图即为Alex的CNN结构图。需要注意的是，该模型采用了2-GPU并行结构，即第1、2、4、5卷积层都是将模型参数分为2部分进行训练的。在这里，更进一步，并行结构分为数据并行与模型并行。数据并行是指在不同的GPU上，模型结构相同，但将训练数据进行切分，分别训练得到不同的模型，然后再将模型进行融合。而模型并行则是，将若干层的模型参数进行切分，不同的GPU上使用相同的数据进行训练，得到的结果直接连接作为下一层的输入。



上图模型的基本参数为：

输入：224×224大小的图片，3通道

第一层卷积：11×11大小的卷积核96个，每个GPU上48个。

第一层max-pooling：2×2的核。

第二层卷积：5×5卷积核256个，每个GPU上128个。

第二层max-pooling：2×2的核。

第三层卷积：与上一层是全连接，3\*3的卷积核384个。分到两个GPU上个192个。

第四层卷积：3×3的卷积核384个，两个GPU各192个。该层与上一层连接没有经过pooling层。

第五层卷积：3×3的卷积核256个，两个GPU上个128个。

第五层max-pooling：2×2的核。

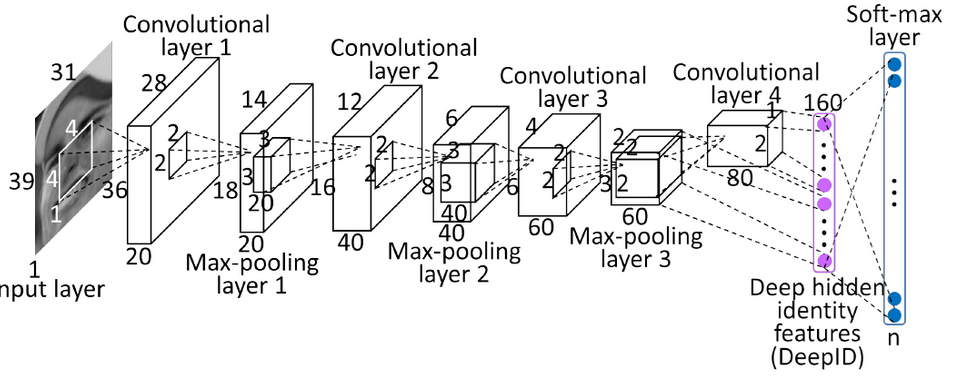
第一层全连接：4096维，将第五层max-pooling的输出连接成为一个一维向量，作为该层的输入。

第二层全连接：4096维

Softmax层：输出为1000，输出的每一维都是图片属于该类别的概率。

**（2）DeepID网络结构**

DeepID网络结构是香港中文大学的Sun Yi开发出来用来学习人脸特征的卷积神经网络。每张输入的人脸被表示为160维的向量，学习到的向量经过其他模型进行分类，在人脸验证试验上得到了97.45%的正确率，更进一步的，原作者改进了CNN，又得到了99.15%的正确率。如下图所示，该结构与ImageNet的具体参数类似，所以只解释一下不同的部分吧。

上图中的结构，在最后只有一层全连接层，然后就是softmax层了。论文中就是以该全连接层作为图像的表示。在全连接层，以第四层卷积和第三层max-pooling的输出作为全连接层的输入，这样可以学习到局部的和全局的特征。

**5 CNN代码实现**

**（1）导入必要的模块**

[python] view plain copy

import cPickle

import gzip

import os

import sys

import time

import numpy

import theano

import theano.tensor as T

from theano.tensor.signal import downsample

from theano.tensor.nnet import conv

**（2）定义CNN的基本"构件"**

CNN的基本构件包括卷积采样层、隐含层、分类器，如下

**定义LeNetConvPoolLayer（卷积+采样层）**

"""

卷积+下采样合成一个层LeNetConvPoolLayer

rng:随机数生成器，用于初始化W

input:4维的向量，theano.tensor.dtensor4

filter\_shape:(number of filters, num input feature maps,filter height, filter width)

image\_shape:(batch size, num input feature maps,image height, image width)

poolsize: (#rows, #cols)

"""

class LeNetConvPoolLayer(object):

def \_\_init\_\_(self, rng, input, filter\_shape, image\_shape, poolsize=(2, 2)):

#assert condition，condition为True，则继续往下执行，condition为False，中断程序

#image\_shape[1]和filter\_shape[1]都是num input feature maps，它们必须是一样的。

assert image\_shape[1] == filter\_shape[1]

self.input = input

#每个隐层神经元（即像素）与上一层的连接数为num input feature maps \* filter height \* filter width。

#可以用numpy.prod(filter\_shape[1:])来求得

fan\_in = numpy.prod(filter\_shape[1:])

#lower layer上每个神经元获得的梯度来自于："num output feature maps \* filter height \* filter width" /pooling size

fan\_out = (filter\_shape[0] \* numpy.prod(filter\_shape[2:]) /

numpy.prod(poolsize))

#以上求得fan\_in、fan\_out ，将它们代入公式，以此来随机初始化W,W就是线性卷积核

W\_bound = numpy.sqrt(6. / (fan\_in + fan\_out))

self.W = theano.shared(

numpy.asarray(

rng.uniform(low=-W\_bound, high=W\_bound, size=filter\_shape),

dtype=theano.config.floatX

),

borrow=True

)

# the bias is a 1D tensor -- one bias per output feature map

#偏置b是一维向量，每个输出图的特征图都对应一个偏置，

#而输出的特征图的个数由filter个数决定，因此用filter\_shape[0]即number of filters来初始化

b\_values = numpy.zeros((filter\_shape[0],), dtype=theano.config.floatX)

self.b = theano.shared(value=b\_values, borrow=True)

#将输入图像与filter卷积，conv.conv2d函数

#卷积完没有加b再通过sigmoid，这里是一处简化。

conv\_out = conv.conv2d(

input=input,

filters=self.W,

filter\_shape=filter\_shape,

image\_shape=image\_shape

)

#maxpooling，最大子采样过程

pooled\_out = downsample.max\_pool\_2d(

input=conv\_out,

ds=poolsize,

ignore\_border=True

)

#加偏置，再通过tanh映射，得到卷积+子采样层的最终输出

#因为b是一维向量，这里用维度转换函数dimshuffle将其reshape。比如b是(10,)，

#则b.dimshuffle('x', 0, 'x', 'x'))将其reshape为(1,10,1,1)

self.output = T.tanh(pooled\_out + self.b.dimshuffle('x', 0, 'x', 'x'))

#卷积+采样层的参数

self.params = [self.W, self.b]

**定义隐含层HiddenLayer**

这个跟上一篇文章《 DeepLearning tutorial（3）MLP多层感知机原理简介+代码详解》中的HiddenLayer是一致的，直接拿过来：

[python] view plain copy

"""

注释：

这是定义隐藏层的类，首先明确：隐藏层的输入即input，输出即隐藏层的神经元个数。输入层与隐藏层是全连接的。

假设输入是n\_in维的向量（也可以说时n\_in个神经元），隐藏层有n\_out个神经元，则因为是全连接，

一共有n\_in\*n\_out个权重，故W大小时(n\_in,n\_out),n\_in行n\_out列，每一列对应隐藏层的每一个神经元的连接权重。

b是偏置，隐藏层有n\_out个神经元，故b时n\_out维向量。

rng即随机数生成器，numpy.random.RandomState，用于初始化W。

input训练模型所用到的所有输入，并不是MLP的输入层，MLP的输入层的神经元个数时n\_in，而这里的参数input大小是（n\_example,n\_in）,每一行一个样本，即每一行作为MLP的输入层。

activation:激活函数,这里定义为函数tanh

"""

class HiddenLayer(object):

def \_\_init\_\_(self, rng, input, n\_in, n\_out, W=None, b=None,

activation=T.tanh):

self.input = input #类HiddenLayer的input即所传递进来的input

"""

注释：

代码要兼容GPU，则必须使用 dtype=theano.config.floatX,并且定义为theano.shared

另外，W的初始化有个规则：如果使用tanh函数，则在-sqrt(6./(n\_in+n\_hidden))到sqrt(6./(n\_in+n\_hidden))之间均匀

抽取数值来初始化W，若时sigmoid函数，则以上再乘4倍。

"""

#如果W未初始化，则根据上述方法初始化。

#加入这个判断的原因是：有时候我们可以用训练好的参数来初始化W，见我的上一篇文章。

if W is None:

W\_values = numpy.asarray(

rng.uniform(

low=-numpy.sqrt(6. / (n\_in + n\_out)),

high=numpy.sqrt(6. / (n\_in + n\_out)),

size=(n\_in, n\_out)

),

dtype=theano.config.floatX

)

if activation == theano.tensor.nnet.sigmoid:

W\_values \*= 4

W = theano.shared(value=W\_values, name='W', borrow=True)

if b is None:

b\_values = numpy.zeros((n\_out,), dtype=theano.config.floatX)

b = theano.shared(value=b\_values, name='b', borrow=True)

#用上面定义的W、b来初始化类HiddenLayer的W、b

self.W = W

self.b = b

#隐含层的输出

lin\_output = T.dot(input, self.W) + self.b

self.output = (

lin\_output if activation is None

else activation(lin\_output)

)

#隐含层的参数

self.params = [self.W, self.b]

**定义分类器 （Softmax回归）**

采用Softmax，这跟《DeepLearning tutorial（1）Softmax回归原理简介+代码详解》中的LogisticRegression是一样的，直接拿过来：

[python] view plain copy

"""

定义分类层LogisticRegression，也即Softmax回归

在deeplearning tutorial中，直接将LogisticRegression视为Softmax，

而我们所认识的二类别的逻辑回归就是当n\_out=2时的LogisticRegression

"""

#参数说明：

#input，大小就是(n\_example,n\_in)，其中n\_example是一个batch的大小，

#因为我们训练时用的是Minibatch SGD，因此input这样定义

#n\_in,即上一层(隐含层)的输出

#n\_out,输出的类别数

class LogisticRegression(object):

def \_\_init\_\_(self, input, n\_in, n\_out):

#W大小是n\_in行n\_out列，b为n\_out维向量。即：每个输出对应W的一列以及b的一个元素。

self.W = theano.shared(

value=numpy.zeros(

(n\_in, n\_out),

dtype=theano.config.floatX

),

name='W',

borrow=True

)

self.b = theano.shared(

value=numpy.zeros(

(n\_out,),

dtype=theano.config.floatX

),

name='b',

borrow=True

)

#input是(n\_example,n\_in)，W是（n\_in,n\_out）,点乘得到(n\_example,n\_out)，加上偏置b，

#再作为T.nnet.softmax的输入，得到p\_y\_given\_x

#故p\_y\_given\_x每一行代表每一个样本被估计为各类别的概率

#PS：b是n\_out维向量，与(n\_example,n\_out)矩阵相加，内部其实是先复制n\_example个b，

#然后(n\_example,n\_out)矩阵的每一行都加b

self.p\_y\_given\_x = T.nnet.softmax(T.dot(input, self.W) + self.b)

#argmax返回最大值下标，因为本例数据集是MNIST，下标刚好就是类别。axis=1表示按行操作。

self.y\_pred = T.argmax(self.p\_y\_given\_x, axis=1)

#params，LogisticRegression的参数

self.params = [self.W, self.b]

到这里，CNN的基本”构件“都有了，下面要用这些”构件“组装成LeNet5（当然，是简化的，上面已经说了），具体来说，就是组装成：LeNet5=input+LeNetConvPoolLayer\_1+LeNetConvPoolLayer\_2+HiddenLayer+LogisticRegression+output。然后将其应用于MNIST数据集，用BP算法去解这个模型，得到最优的参数。

**（3）加载MNIST数据集（mnist.pkl.gz）**

[python] view plain copy

"""

加载MNIST数据集load\_data()

"""

def load\_data(dataset):

# dataset是数据集的路径，程序首先检测该路径下有没有MNIST数据集，没有的话就下载MNIST数据集

#这一部分就不解释了，与softmax回归算法无关。

data\_dir, data\_file = os.path.split(dataset)

if data\_dir == "" and not os.path.isfile(dataset):

# Check if dataset is in the data directory.

new\_path = os.path.join(

os.path.split(\_\_file\_\_)[0],

"..",

"data",

dataset

)

if os.path.isfile(new\_path) or data\_file == 'mnist.pkl.gz':

dataset = new\_path

if (not os.path.isfile(dataset)) and data\_file == 'mnist.pkl.gz':

import urllib

origin = (

'http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz'

)

print 'Downloading data from %s' % origin

urllib.urlretrieve(origin, dataset)

print '... loading data'

#以上是检测并下载数据集mnist.pkl.gz，不是本文重点。下面才是load\_data的开始

#从"mnist.pkl.gz"里加载train\_set, valid\_set, test\_set，它们都是包括label的

#主要用到python里的gzip.open()函数,以及 cPickle.load()。

#‘rb’表示以二进制可读的方式打开文件

f = gzip.open(dataset, 'rb')

train\_set, valid\_set, test\_set = cPickle.load(f)

f.close()

#将数据设置成shared variables，主要时为了GPU加速，只有shared variables才能存到GPU memory中

#GPU里数据类型只能是float。而data\_y是类别，所以最后又转换为int返回

def shared\_dataset(data\_xy, borrow=True):

data\_x, data\_y = data\_xy

shared\_x = theano.shared(numpy.asarray(data\_x,

dtype=theano.config.floatX),

borrow=borrow)

shared\_y = theano.shared(numpy.asarray(data\_y,

dtype=theano.config.floatX),

borrow=borrow)

return shared\_x, T.cast(shared\_y, 'int32')

test\_set\_x, test\_set\_y = shared\_dataset(test\_set)

valid\_set\_x, valid\_set\_y = shared\_dataset(valid\_set)

train\_set\_x, train\_set\_y = shared\_dataset(train\_set)

rval = [(train\_set\_x, train\_set\_y), (valid\_set\_x, valid\_set\_y),

(test\_set\_x, test\_set\_y)]

return rval

**（4）实现LeNet5并测试**

[python] view plain copy

"""

实现LeNet5

LeNet5有两个卷积层，第一个卷积层有20个卷积核，第二个卷积层有50个卷积核

"""

def evaluate\_lenet5(learning\_rate=0.1, n\_epochs=200,

dataset='mnist.pkl.gz',

nkerns=[20, 50], batch\_size=500):

"""

learning\_rate:学习速率，随机梯度前的系数。

n\_epochs训练步数，每一步都会遍历所有batch，即所有样本

batch\_size,这里设置为500，即每遍历完500个样本，才计算梯度并更新参数

nkerns=[20, 50],每一个LeNetConvPoolLayer卷积核的个数，第一个LeNetConvPoolLayer有

20个卷积核，第二个有50个

"""

rng = numpy.random.RandomState(23455)

#加载数据

datasets = load\_data(dataset)

train\_set\_x, train\_set\_y = datasets[0]

valid\_set\_x, valid\_set\_y = datasets[1]

test\_set\_x, test\_set\_y = datasets[2]

# 计算batch的个数

n\_train\_batches = train\_set\_x.get\_value(borrow=True).shape[0]

n\_valid\_batches = valid\_set\_x.get\_value(borrow=True).shape[0]

n\_test\_batches = test\_set\_x.get\_value(borrow=True).shape[0]

n\_train\_batches /= batch\_size

n\_valid\_batches /= batch\_size

n\_test\_batches /= batch\_size

#定义几个变量，index表示batch下标，x表示输入的训练数据，y对应其标签

index = T.lscalar()

x = T.matrix('x')

y = T.ivector('y')

######################

# BUILD ACTUAL MODEL #

######################

print '... building the model'

#我们加载进来的batch大小的数据是(batch\_size, 28 \* 28)，但是LeNetConvPoolLayer的输入是四维的，所以要reshape

layer0\_input = x.reshape((batch\_size, 1, 28, 28))

# layer0即第一个LeNetConvPoolLayer层

#输入的单张图片(28,28)，经过conv得到(28-5+1 , 28-5+1) = (24, 24)，

#经过maxpooling得到(24/2, 24/2) = (12, 12)

#因为每个batch有batch\_size张图，第一个LeNetConvPoolLayer层有nkerns[0]个卷积核，

#故layer0输出为(batch\_size, nkerns[0], 12, 12)

layer0 = LeNetConvPoolLayer(

rng,

input=layer0\_input,

image\_shape=(batch\_size, 1, 28, 28),

filter\_shape=(nkerns[0], 1, 5, 5),

poolsize=(2, 2)

)

#layer1即第二个LeNetConvPoolLayer层

#输入是layer0的输出，每张特征图为(12,12),经过conv得到(12-5+1, 12-5+1) = (8, 8),

#经过maxpooling得到(8/2, 8/2) = (4, 4)

#因为每个batch有batch\_size张图（特征图），第二个LeNetConvPoolLayer层有nkerns[1]个卷积核

#，故layer1输出为(batch\_size, nkerns[1], 4, 4)

layer1 = LeNetConvPoolLayer(

rng,

input=layer0.output,

image\_shape=(batch\_size, nkerns[0], 12, 12),#输入nkerns[0]张特征图，即layer0输出nkerns[0]张特征图

filter\_shape=(nkerns[1], nkerns[0], 5, 5),

poolsize=(2, 2)

)

#前面定义好了两个LeNetConvPoolLayer（layer0和layer1），layer1后面接layer2，这是一个全连接层，相当于MLP里面的隐含层

#故可以用MLP中定义的HiddenLayer来初始化layer2，layer2的输入是二维的(batch\_size, num\_pixels) ，

#故要将上层中同一张图经不同卷积核卷积出来的特征图合并为一维向量，

#也就是将layer1的输出(batch\_size, nkerns[1], 4, 4)flatten为(batch\_size, nkerns[1]\*4\*4)=(500，800),作为layer2的输入。

#(500，800)表示有500个样本，每一行代表一个样本。layer2的输出大小是(batch\_size,n\_out)=(500,500)

layer2\_input = layer1.output.flatten(2)

layer2 = HiddenLayer(

rng,

input=layer2\_input,

n\_in=nkerns[1] \* 4 \* 4,

n\_out=500,

activation=T.tanh

)

#最后一层layer3是分类层，用的是逻辑回归中定义的LogisticRegression，

#layer3的输入是layer2的输出(500,500)，layer3的输出就是(batch\_size,n\_out)=(500,10)

layer3 = LogisticRegression(input=layer2.output, n\_in=500, n\_out=10)

#代价函数NLL

cost = layer3.negative\_log\_likelihood(y)

# test\_model计算测试误差，x、y根据给定的index具体化，然后调用layer3，

#layer3又会逐层地调用layer2、layer1、layer0，故test\_model其实就是整个CNN结构，

#test\_model的输入是x、y，输出是layer3.errors(y)的输出，即误差。

test\_model = theano.function(

[index],

layer3.errors(y),

givens={

x: test\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],

y: test\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]

}

)

#validate\_model，验证模型，分析同上。

validate\_model = theano.function(

[index],

layer3.errors(y),

givens={

x: valid\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],

y: valid\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]

}

)

#下面是train\_model，涉及到优化算法即SGD，需要计算梯度、更新参数

#参数集

params = layer3.params + layer2.params + layer1.params + layer0.params

#对各个参数的梯度

grads = T.grad(cost, params)

#因为参数太多，在updates规则里面一个一个具体地写出来是很麻烦的，所以下面用了一个for..in..,自动生成规则对(param\_i, param\_i - learning\_rate \* grad\_i)

updates = [

(param\_i, param\_i - learning\_rate \* grad\_i)

for param\_i, grad\_i in zip(params, grads)

]

#train\_model，代码分析同test\_model。train\_model里比test\_model、validation\_model多出updates规则

train\_model = theano.function(

[index],

cost,

updates=updates,

givens={

x: train\_set\_x[index \* batch\_size: (index + 1) \* batch\_size],

y: train\_set\_y[index \* batch\_size: (index + 1) \* batch\_size]

}

)

###############

# 开始训练 #

###############

print '... training'

patience = 10000

patience\_increase = 2

improvement\_threshold = 0.995

validation\_frequency = min(n\_train\_batches, patience / 2)

#这样设置validation\_frequency可以保证每一次epoch都会在验证集上测试。

best\_validation\_loss = numpy.inf #最好的验证集上的loss，最好即最小

best\_iter = 0 #最好的迭代次数，以batch为单位。比如best\_iter=10000，说明在训练完第10000个batch时，达到best\_validation\_loss

test\_score = 0.

start\_time = time.clock()

epoch = 0

done\_looping = False

#下面就是训练过程了，while循环控制的时步数epoch，一个epoch会遍历所有的batch，即所有的图片。

#for循环是遍历一个个batch，一次一个batch地训练。for循环体里会用train\_model(minibatch\_index)去训练模型，

#train\_model里面的updatas会更新各个参数。

#for循环里面会累加训练过的batch数iter，当iter是validation\_frequency倍数时则会在验证集上测试，

#如果验证集的损失this\_validation\_loss小于之前最佳的损失best\_validation\_loss，

#则更新best\_validation\_loss和best\_iter，同时在testset上测试。

#如果验证集的损失this\_validation\_loss小于best\_validation\_loss\*improvement\_threshold时则更新patience。

#当达到最大步数n\_epoch时，或者patience<iter时，结束训练

while (epoch < n\_epochs) and (not done\_looping):

epoch = epoch + 1

for minibatch\_index in xrange(n\_train\_batches):

iter = (epoch - 1) \* n\_train\_batches + minibatch\_index

if iter % 100 == 0:

print 'training @ iter = ', iter

cost\_ij = train\_model(minibatch\_index)

#cost\_ij 没什么用，后面都没有用到,只是为了调用train\_model，而train\_model有返回值

if (iter + 1) % validation\_frequency == 0:

# compute zero-one loss on validation set

validation\_losses = [validate\_model(i) for i

in xrange(n\_valid\_batches)]

this\_validation\_loss = numpy.mean(validation\_losses)

print('epoch %i, minibatch %i/%i, validation error %f %%' %

(epoch, minibatch\_index + 1, n\_train\_batches,

this\_validation\_loss \* 100.))

if this\_validation\_loss < best\_validation\_loss:

if this\_validation\_loss < best\_validation\_loss \* \

improvement\_threshold:

patience = max(patience, iter \* patience\_increase)

best\_validation\_loss = this\_validation\_loss

best\_iter = iter

test\_losses = [

test\_model(i)

for i in xrange(n\_test\_batches)

]

test\_score = numpy.mean(test\_losses)

print((' epoch %i, minibatch %i/%i, test error of '

'best model %f %%') %

(epoch, minibatch\_index + 1, n\_train\_batches,

test\_score \* 100.))

if patience <= iter:

done\_looping = True

break

end\_time = time.clock()

print('Optimization complete.')

print('Best validation score of %f %% obtained at iteration %i, '

'with test performance %f %%' %

(best\_validation\_loss \* 100., best\_iter + 1, test\_score \* 100.))

print >> sys.stderr, ('The code for file ' +

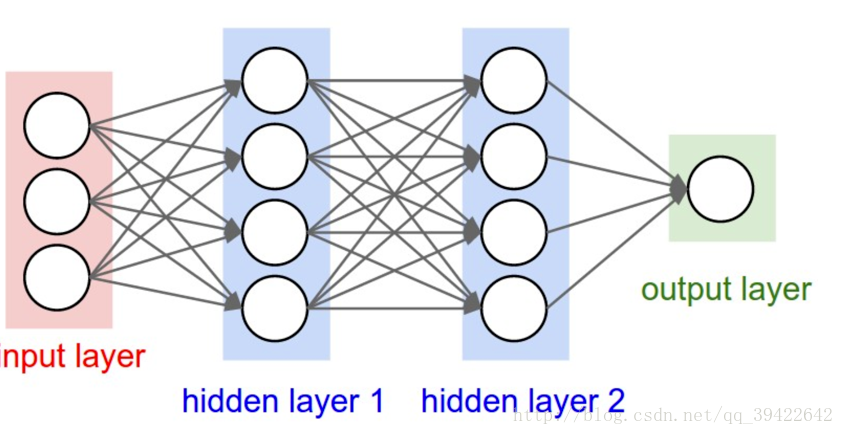
os.path.split(\_\_file\_\_)[1] +

' ran for %.2fm' % ((end\_time - start\_time) / 60.))

## 循环神经网络(RNN)

**1.RNN怎么来的？**

**循环神经网络的应用场景比较多，比如暂时能写论文，写程序，写诗，但是，（总是会有但是的），但是他们现在还不能正常使用，学习出来的东西没有逻辑，所以要想真正让它更有用，路还很远。**这是一般的神经网络应该有的结构：



**既然我们已经有了人工神经网络和卷积神经网络，为什么还要循环神经网络？**

原因很简单，无论是卷积神经网络，还是人工神经网络，他们的前提假设都是：元素之间是相互独立的，输入与输出也是独立的，比如猫和狗。但现实世界中，很多元素都是相互连接的，比如股票随时间的变化，一个人说了：我喜欢旅游，其中最喜欢的地方是云南，以后有机会一定要去\_\_\_\_\_这里填空，人应该都知道是填“云南“。因为我们是根据上下文的内容推断出来的，但机会要做到这一步就相当得难了。因此，就有了现在的循环神经网络，他的本质是：像人一样拥有记忆的能力。因此，他的输出就依赖于当前的输入和记忆。

递归神经网络是时间递归神经网络（recurrent neural network）和结构递归神经网（recursive neural network）的总称。RNN一般指代时间递归神经网络。RNN早先被提到的可以追溯到1989年Axel Cleeremans的论文。详情查看：http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.160.2979&rep=rep1&type=pdf

RNN被提出的初衷是用来处理序列数据的。RNN相对于传统神经网络最大的不同是神经元的输入的改变。RNN隐藏层神经元的输入不止是上一层神经元的输出，还包括了本层的输出。但是鉴于RNN误差反向传播时候梯度消失的问题。1997年Sepp Hochreiter等人提出了典型的LSTM网络。详情查看：http://www.bioinf.jku.at/publications/older/2604.pdf

LMST是RNN的升级版，它的隐藏层神经元的输入和输出会在RNN的基础上做进一步的处理，在本文末有提到

**优点：**时间递归神经网络可以描述动态时间行为，因为和前馈神经网络接受较特定结构的输入不同，RNN将状态在自身网络中循环传递，因此可以接受更广泛的时间序列结构输入。

**缺点：**简单递归神经网络无法处理随着递归，梯度爆炸或者梯度消失的问题，并且难以捕捉长期时间关联；有效的处理方法是忘掉错误的信息，记住正确的信息。LSTM能够比较好的解决这个问题。

**应用**

RNN已经被在实践中证明对NLP是非常成功的。如词向量表达，语句合法性检查，词性标注等。在RNN中，目前使用最广泛最成功的模型是LSTM模型。基于LSTM的系统可以学习翻译语言、控制机器人、图像分析、文档摘要、语音识别图像识别、手写识别、控制聊天机器人、预测疾病、点击率和股票、合成音乐等等任务。举个例子，在2015年，谷歌通过基于CTC训练的LSTM程序大幅提升了安卓手机和其他设备中语音识别的能力，使用了我的实验室在2006年发表的方法。百度也使用了CTC；苹果的iPhone在QucikType和Siri中使用了LSTM；微软不仅将LSTM用于语音识别，还将这一技术用于虚拟对话形象生成和编写程序代码等等。亚马逊Alexa通过双向LSTM在家中与你交流，而谷歌使用LSTM的范围更加广泛，它可以生成图像字幕，自动回复电子邮件，它包含在新的智能助手Allo中，也显著地提高了谷歌翻译的质量（从2016年开始）。事实上，谷歌数据中心的很大一部分计算资源现在都在执行LSTM任务。

**2.RNN的网络结构及原理**

**（1）它的网络结构如下**

其中每个圆圈可以看作是一个单元，而且每个单元做的事情也是一样的，因此可以折叠呈左半图的样子。用一句话解释RNN，就是一个单元结构重复使用。

RNN是一个序列到序列的模型，假设xt−1,xt,xt+1

是一个输入：“我是中国“，那么ot−1,ot就应该对应”是”，”中国”这两个，预测下一个词最有可能是什么？就是ot+1

应该是”人”的概率比较大。

**（2）因此，我们可以做这样的定义：**

**Xt:表示t时刻的输入，ot:表示t时刻的输出，St:表示t时刻的记忆**

因为我们当前时刻的输出是由记忆和当前时刻的输入决定的，就像你现在大四，你的知识是由大四学到的知识（当前输入）和大三以及大三以前学到的东西的（记忆）的结合，RNN在这点上也类似，神经网络最擅长做的就是通过一系列参数把很多内容整合到一起，然后学习这个参数，因此就定义了RNN的基础：

**St=f(U∗Xt+W∗St−1)**

大家可能会很好奇，为什么还要加一个f()函数，其实这个函数是神经网络中的激活函数，但为什么要加上它呢？举个例子，假如你在大学学了非常好的解题方法，那你初中那时候的解题方法还要用吗？显然是不用了的。RNN的想法也一样，既然我能记忆了，那我当然是只记重要的信息啦，其他不重要的，就肯定会忘记，是吧。但是在神经网络中什么最适合过滤信息呀？肯定是激活函数嘛，因此在这里就套用一个激活函数，来做一个非线性映射，来过滤信息，这个激活函数可能为tanh，也可为其他。假设你大四快毕业了，要参加考研，请问你参加考研是不是先记住你学过的内容然后去考研，还是直接带几本书去参加考研呢？很显然嘛，那RNN的想法就是预测的时候带着当前时刻的记忆St去预测。假如你要预测“我是中国“的下一个词出现的概率，这里已经很显然了，运用softmax来预测每个词出现的概率再合适不过了，但预测不能直接带用一个矩阵来预测呀，所有预测的时候还要带一个权重矩阵V,用公式表示为:

**ot=softmax(VSt)**

其中ot就表示时刻t的输出。

**（3）RNN中的结构细节**

1）.可以把St

当作隐状态，捕捉了之前时间点上的信息。就像你去考研一样，考的时候记住了你能记住的所有信息。

2）.ot是由当前时间以及之前所有的记忆得到的。就是你考研之后做的考试卷子，是用你的记忆得到的。

3）.很可惜的是，St并不能捕捉之前所有时间点的信息。就像你考研不能记住所有的英语单词一样。

4）.和卷积神经网络一样，这里的网络中每个cell都共享了一组参数（U，V，W）,这样就能极大的降低计算量了。

5）.ot

在很多情况下都是不存在的，因为很多任务，比如文本情感分析，都是只关注最后的结果的。就像考研之后选择学校，学校不会管你到底怎么努力，怎么心酸的准备考研，而只关注你最后考了多少分。

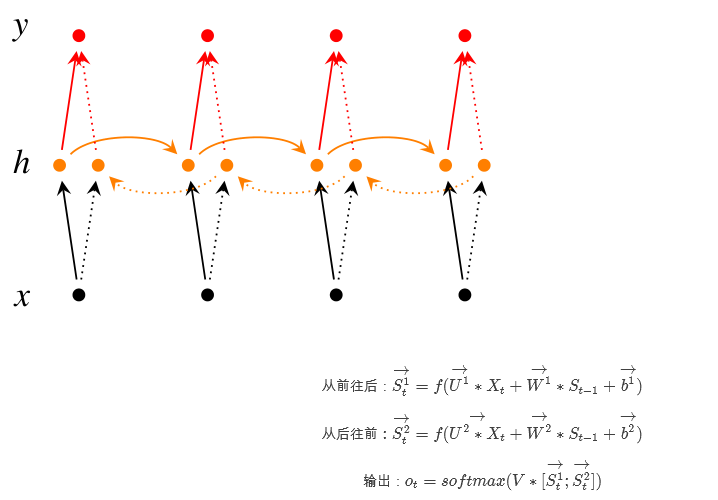
1. **RNN的改进1：双向RNN**

在有些情况，比如有一部电视剧，在第三集的时候才出现的人物，现在让预测一下在第三集中出现的人物名字，你用前面两集的内容是预测不出来的，所以你需要用到第四，第五集的内容来预测第三集的内容，这就是双向RNN的想法。如图是双向RNN的图解：

从前往后：S1t→=f(U1→∗Xt+W1→∗St−1+b1→)

从后往前:S2t→=f(U2∗Xt→+W2→∗St−1+b2→)

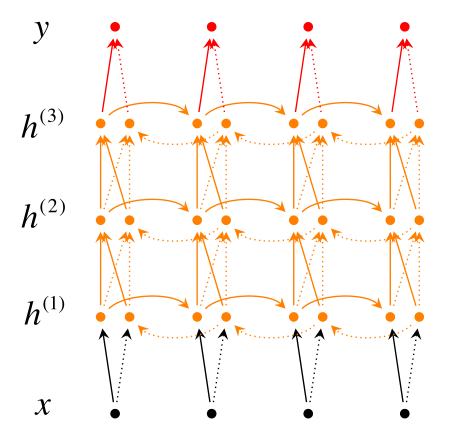
输出：ot=softmax(V∗[S1t→;S2t→])



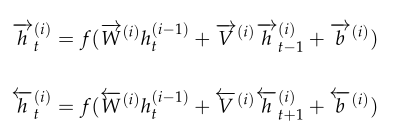
这里的[S1t→;S2t→]做的是一个拼接，如果他们都是1000X1维的，拼接在一起就是1000X2维的了。双向RNN需要的内存是单向RNN的两倍，因为在同一时间点，双向RNN需要保存两个方向上的权重参数，在分类的时候，需要同时输入两个隐藏层输出的信息。

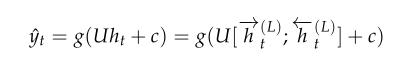
1. **RNN的改进2：深层双向RNN**

（1）深层双向RNN 与双向RNN相比，多了几个隐藏层，因为他的想法是很多信息记一次记不下来，比如你去考研，复习考研英语的时候，背英语单词一定不会就看一次就记住了所有要考的考研单词吧，你应该也是带着先前几次背过的单词，然后选择那些背过，但不熟的内容，或者没背过的单词来背吧。深层双向RNN就是基于这么一个想法，他的输入有两方面，第一就是前一时刻的隐藏层传过来的信息h→(i)t−1，和当前时刻上一隐藏层传过来的信息h(i−1)t=[h→(i−1)t;h←(i−1)t]，包括前向和后向的。



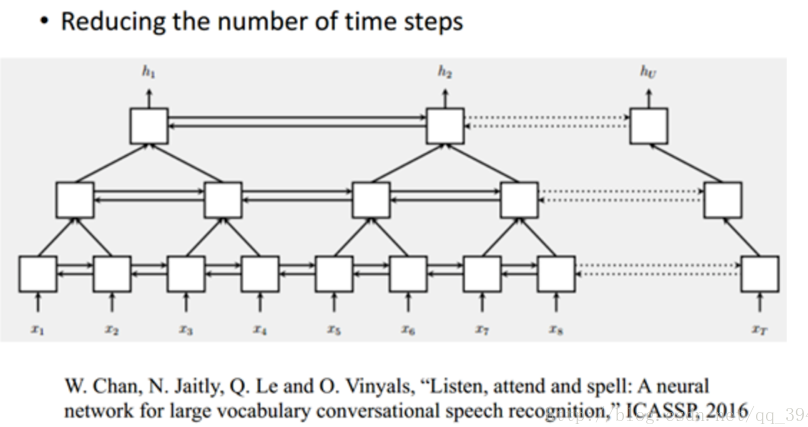
（2）我们用公式来表示是这样的:

然后再利用最后一层来进行分类，分类公式如下：



**（3）Pyramidal RNN**

其他类似的网络还有Pyramidal RNN：

我们现在有一个很长的输入序列，可以看到这是一个双向的RNN，上图是谷歌的W.Chan做的一个测试，它原先要做的是语音识别，他要用序列到序列的模型做语音识别，序列到序列就是说，输入一个序列然后就输出一个序列。由图我们发现，上一层的两个输出，作为当前层的输入，如果是非常长的序列的话，这样做的话，每一层的序列都比上一层要短，但当前层的输入f(x)也会随之增多，貌似看一起相互抵消，运算量并没有什么改进。但我们知道，对于一层来说，它是从前往后转的，比如要预测一个股市的变化，以天为单位，假如要预测明天的股市变化，你就要用今天，以及今天之前的所有数据，我们暂时无法只用昨天的数据，不用今天的数据，预测明天的数据，也即是说，预测必须具有连续性。但每一层的f运算是可以并行的，从这个角度来看，运算量还是可以接受的，特别是在原始输入序列较短的时候还是有优势的。

**5 RNN的训练-BPTT**

如前面我们讲的，如果要预测t时刻的输出，我们必须先利用上一时刻（t-1）的记忆和当前时刻的输入，得到t时刻的记忆：

**st=tanh(Uxt+Wst−1)**

然后利用当前时刻的记忆，通过softmax分类器输出每个词出现的概率：

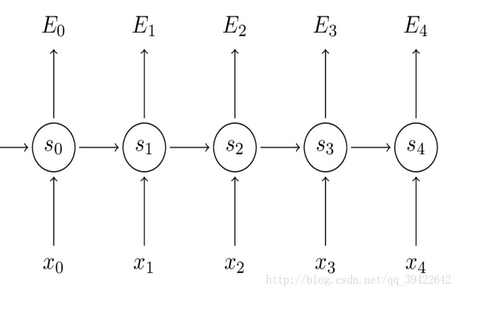
**y^t=softmax(Vst)**

为了找出模型最好的参数，U，W，V，我们就要知道当前参数得到的结果怎么样，因此就要定义我们的损失函数，用交叉熵损失函数：

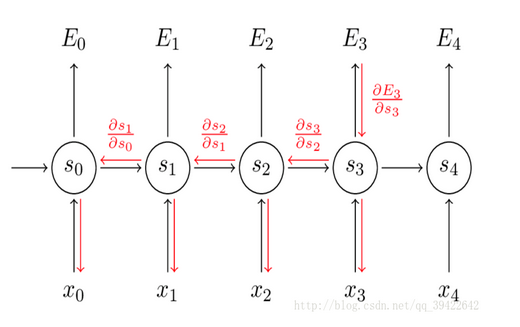
t时刻的损失：**Et(yt,y^t)=−ytlogy^t**

其中ytt时刻的标准答案，是一个只有一个是1，其他都是0的向量；y^t是我们预测出来的结果，与yt的维度一样，但它是一个概率向量，里面是每个词出现的概率。因为对结果的影响，肯定不止一个时刻，因此需要把所有时刻的造成的损失都加起来：

**E(yt,y^t)=∑tEt(yt,y^t)=−∑tytlogy^t**

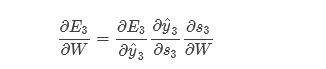


如图所示，你会发现每个cell都会有一个损失，我们已经定义好了损失函数，接下来就是熟悉的一步了，那就是根据损失函数利用SGD来求解最优参数，在CNN中使用反向传播BP算法来求解最优参数，但在RNN就要用到BPTT，它和BP算法的本质区别，也是CNN和RNN的本质区别：CNN没有记忆功能，它的输出仅依赖与输入，但RNN有记忆功能，它的输出不仅依赖与当前输入，还依赖与当前的记忆。这个记忆是序列到序列的，也就是当前时刻收到上一时刻的影响，比如股市的变化。因此，在对参数求偏导的时候，对当前时刻求偏导，一定会涉及前一时刻，我们用例子看一下：

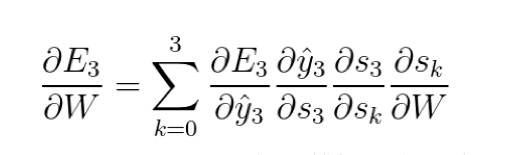


假设我们对E3的W求偏导：它的损失首先来源于预测的输出y^3，预测的输出又是来源于当前时刻的记忆s3,当前的记忆又是来源于当前的输出和截止到上一时刻的记忆：**s3=tanh(Ux3+Ws2)**

因此，根据链式法则可以有:

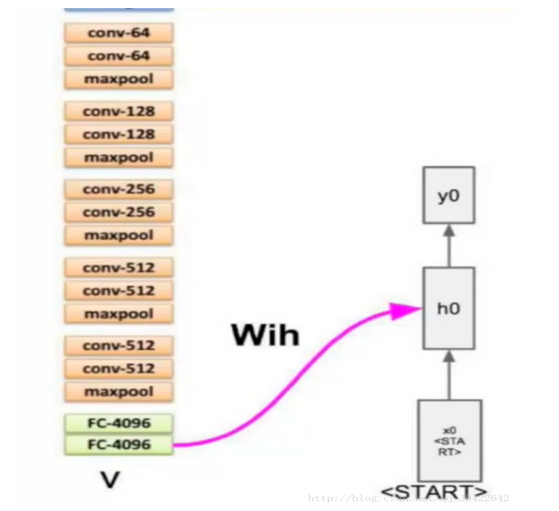


但是，你会发现，s2=tanh(Ux2+Ws1)，也就是s2里面的函数还包含了W，因此，这个链式法则还没到底，就像图上画的那样，所以真正的链式法则是这样的：

我们要把当前时刻造成的损失，和以往每个时刻造成的损失加起来，因为我们每一个时刻都用到了权重参数W。和以往的网络不同，一般的网络，比如人工神经网络，参数是不同享的，但在循环神经网络，和CNN一样，设立了参数共享机制，来降低模型的计算量。

**6.RNN与CNN的结合应用：看图说话**

在图像处理中，目前做的最好的是CNN，而自然语言处理中，表现比较好的是RNN，因此，我们能否把他们结合起来，一起用呢？那就是看图说话了，这个原理也比较简单，举个小栗子：假设我们有CNN的模型训练了一个网络结构，比如是这个



最后我们不是要分类嘛，那在分类前，是不是已经拿到了图像的特征呀，那我们能不能把图像的特征拿出来，放到RNN的输入里，让他学习呢？

之前的RNN是这样的：

**St=tanh(U∗Xt+W∗St−1)**

我们把图像的特征加在里面，可以得到：

**St=tanh(U∗Xt+W∗St−1+V∗X)**

其中的X就是图像的特征。如果用的是上面的CNN网络，X应该是一个4096X1的向量。

**7 用RNN来实现一个八位的二进制数加法运算。**

代码实现地址：http://iamtrask.github.io/2015/11/15/anyone-can-code-lstm/

**1) 网络的输入：**

#输入是（a[i],b[i]） 其中i>=0 且 i<8

a\_int = np.random.randint(largest\_number/2) # int version

a = int2binary[a\_int] # binary encoding

b\_int = np.random.randint(largest\_number/2) # int version

b = int2binary[b\_int] # binary encoding

**2) 计算y值：**

#真实值y （c[i]） i>=0 且 i<8

c\_int = a\_int + b\_int

c = int2binary[c\_int]

**3）隐藏层：**

#隐藏层1层 16个神经元

hidden\_dim = 1

**4）输入层和隐藏层的连接方式**，隐藏层和隐藏层的连接方式，隐藏层和输出层的连接方式：

输入层和隐藏层采用全连接；隐藏层中，位于同一层的神经元相互连接；隐藏层和输出层也采用全连接。

**5）隐藏层和输出层神经元的输入值、输出值：**

#隐藏层神经元的输入为输入层加上一时刻（t-1）神经元的输出：

layer\_1=sigmoid(np.dot(X,synapse\_0)+np.dot(layer\_1\_values[-1],synapse\_h))

#输出层的输入、输出：

layer\_2 = sigmoid(np.dot(layer\_1,synapse\_1))

**6）计算权重的调整值并迭代**

X = np.array([[a[position],b[position]]])

layer\_1 = layer\_1\_values[-position-1]

prev\_layer\_1 = layer\_1\_values[-position-2]

# error at output layer

layer\_2\_delta = layer\_2\_deltas[-position-1]

# error at hidden layer

layer\_1\_delta = (future\_layer\_1\_delta.dot(synapse\_h.T) + layer\_2\_delta.dot(synapse\_1.T)) \* sigmoid\_output\_to\_derivative(layer\_1)

# 更新一个序列的w值并累加

synapse\_1\_update+=np.atleast\_2d(layer\_1).T.dot(layer\_2\_delta)

synapse\_h\_update+=np.atleast\_2d(prev\_layer\_1).T.dot(layer\_1\_delta)

**7）查看最终训练结果：**

#这是训练10000次的运行效果

Error: [ 3.45638663]

Pred: [0 0 0 0 0 0 0 1]

True: [0 1 0 0 0 1 0 1]

9 + 60 = 1

------------

Error: [ 3.63389116]

Pred: [1 1 1 1 1 1 1 1]

True: [0 0 1 1 1 1 1 1]

28 + 35 = 255

------------

Error: [ 3.91366595]

Pred: [0 1 0 0 1 0 0 0]

True: [1 0 1 0 0 0 0 0]

116 + 44 = 72

------------

Error: [ 3.72191702]

Pred: [1 1 0 1 1 1 1 1]

True: [0 1 0 0 1 1 0 1]

4 + 73 = 223

------------

Error: [ 3.5852713]

Pred: [0 0 0 0 1 0 0 0]

True: [0 1 0 1 0 0 1 0]

71 + 11 = 8

------------

Error: [ 2.53352328]

Pred: [1 0 1 0 0 0 1 0]

True: [1 1 0 0 0 0 1 0]

81 + 113 = 162

------------

Error: [ 0.57691441]

Pred: [0 1 0 1 0 0 0 1]

True: [0 1 0 1 0 0 0 1]

81 + 0 = 81

------------

Error: [ 1.42589952]

Pred: [1 0 0 0 0 0 0 1]

True: [1 0 0 0 0 0 0 1]

4 + 125 = 129

------------

Error: [ 0.47477457]

Pred: [0 0 1 1 1 0 0 0]

True: [0 0 1 1 1 0 0 0]

39 + 17 = 56

------------

Error: [ 0.21595037]

Pred: [0 0 0 0 1 1 1 0]

True: [0 0 0 0 1 1 1 0]

11 + 3 = 14

------------

## 生成对抗网络(GAN）

**1 GAN概念**

**GAN 主要包括了两个部分，即生成器 generator 与判别器 discriminator。**生成器主要用来学习真实图像分布从而让自身生成的图像更加真实，以骗过判别器。判别器则需要对接收的图片进行真假判别。在整个过程中，生成器努力地让生成的图像更加真实，而判别器则努力地去识别出图像的真假，这个过程相当于一个二人博弈，随着时间的推移，生成器和判别器在不断地进行对抗，最终两个网络达到了一个动态均衡：生成器生成的图像接近于真实图像分布，而判别器识别不出真假图像，对于给定图像的预测为真的概率基本接近 0.5（相当于随机猜测类别）。对于 GAN 更加直观的理解可以用一个例子来说明：造假币的团伙相当于生成器，他们想通过伪造金钱来骗过银行，使得假币能够正常交易，而银行相当于判别器，需要判断进来的钱是真钱还是假币。因此假币团伙的目的是要造出银行识别不出的假币而骗过银行，银行则是要想办法准确地识别出假币。

因此，我们可以将上面的内容进行一个总结。给定真 = 1，假 = 0，那么有：

对于给定的真实图片（real image），判别器要为其打上标签 1；

对于给定的生成图片（fake image），判别器要为其打上标签 0；

对于生成器传给辨别器的生成图片，生成器希望辨别器打上标签 1。

对于大多数的深度学习网络，还都是搭建好一个固定的网络，本着一定的目的通过数据去训练该网络直到学到一个很好地效果，以此为目的。如：Alexnet做分类，Lenet用来做mnist手写数字识别。但是Goodfellow大神在2014年对于一个无监督学习提出了一个非常innovative的想法——就是GAN 了。

GAN跟一般的深度学习网络不同，可以说他一共需要train两个网络，一个叫做生成网络（Generative Nets），另一个叫做判别网络（Discriminative Nets）,生成网络的输入是随机噪声（服从一定分布），通过该网络生成一幅与样本一样大小的图像；判别网络输入是一系列样本，也可以是生成网络的输出，用来判别输出是否是样本这一类。做一个形象的比喻：生成网络就好比是假货（fake）的制造者，判别网络好比是警察，每当生成网络生成一个新的假货，就交由判别网络来鉴别该货物是否属于正品（原始数据样本），刚开始G网络（即生成网络）的权重肯定是随机的，因此前几次迭代的生成值对于D网络（判别网络）来说很容易判别；而整个GAN网络就是根据最后D网络的输出的loss来进行优化更新权重的。直到最后D网络很难判断G网络的生成结果是否属于数据样本。

**2 GAN优点**

@模型只用到了反向传播,而不需要马尔科夫链

@训练时不需要对隐变量做推断

@理论上,只要是可微分函数都可以用于构建D和G,因为能够与深度神经网络结合做深度生成式模型

@G的参数更新不是直接来自数据样本,而是使用来自D的反向传播(这也是与传统方法相比差别较大的)

@从实际结果来看,GAN看起来能产生更好的生成样本

@GAN框架可以训练任何一种生成器网络(理论上，然而在实践中,很难使用增强学习去训练有离散输出的生成器),大多数其他架构需要生成器有一些特定的函数形式,就像输出层必须是高斯化的.另外所有其他框架需要生成器整个都是非零权值(put non-zero mass everywhere),然而,GANs可以学习到一个只在靠近真实数据的地方(神经网络层)产生样本点的模型( GANs can learn models that generate points only on a thin manifold that goes near the data.)【指的是GAN学习到的分布十分接近真实分布，这里把分布密度函数看作高维流行当中的点，某个类型的真实分布，可能是这个高维空间中的低维流行，想象三维空间中一张卷曲的纸。GAN学习的G能够尽量的“收敛”到这张纸上，而别的生成模型不行，总是在真实的流行之外有一定的分布，不够收敛。非零的mass指的是分布的“密度”，或者分布的“微元”】

@没有必要遵循任何种类的因式分解去设计模型,所有的生成器和判别器都可以正常工作

@相比PixelRNN, GAN生成采样的运行时间更短,GANs一次产生一个样本,然而PixelRNNs需要一个像素一个像素的去产生样本

@相比VAE, GANs没有变分下界,如果鉴别器训练良好,那么生成器可以完美的学习到训练样本的分布.换句话说,GANs是渐进一致的,但是VAE是有偏差的

@相比深度玻尔兹曼机, GANs没有变分下界,也没有棘手的配分函数,样本是一次生成的,而不是重复的应用马尔科夫链来生成的

@相比GSNs, GANs产生的样本是一次生成的,而不是重复的应用马尔科夫链来生成的

@相比NICE和Real NVE,GANs没有对潜在变量(生成器的输入值)的大小进行限制

@GANs是一种以半监督方式训练分类器的方法.在你没有很多带标签的训练集的时候,你可以不做任何修改的直接使用我们的代码,通常这是因为你没有太多标记样本

@GANs可以比完全明显的信念网络(NADE,PixelRNN,WaveNet等)更快的产生样本,因为它不需要在采样序列生成不同的数据

@GANs不需要蒙特卡洛估计来训练网络,人们经常抱怨GANs训练不稳定,很难训练,但是他们比训练依赖于蒙特卡洛估计和对数配分函数的玻尔兹曼机简单多了.因为蒙特卡洛方法在高维空间中效果不好,玻尔兹曼机从来没有拓展到像ImgeNet任务中.GANs起码在ImageNet上训练后可以学习去画一些以假乱真的狗

@相比于变分自编码器, GANs没有引入任何决定性偏置( deterministic bias),变分方法引入决定性偏置,因为他们优化对数似然的下界,而不是似然度本身,这看起来导致了VAEs生成的实例比GANs更模糊

@相比非线性ICA(NICE, Real NVE等,),GANs不要求生成器输入的潜在变量有任何特定的维度或者要求生成器是可逆的

@相比玻尔兹曼机和GSNs,GANs生成实例的过程只需要模型运行一次,而不是以马尔科夫链的形式迭代很多次

**3 GAN缺点**

@可解释性差,生成模型的分布 Pg(G)没有显式的表达

@比较难训练,D与G之间需要很好的同步(例如D更新k次而G更新一次)，GAN模型被定义为极小极大问题，没有损失函数，在训练过程中很难区分是否正在取得进展。GAN的学习过程可能发生崩溃问题（collapse problem），生成器开始退化，总是生成同样的样本点，无法继续学习。当生成模型崩溃时，判别模型也会对相似的样本点指向相似的方向，训练无法继续。

@网络难以收敛，目前所有的理论都认为GAN应该在纳什均衡上有很好的表现，但梯度下降只有在凸函数的情况下才能保证实现纳什均衡。

@训练GAN需要达到纳什均衡,有时候可以用梯度下降法做到,有时候做不到.还没有找到很好的达到纳什均衡的方法,所以训练GAN相比VAE或者PixelRNN是不稳定的,但在实践中它还是比训练玻尔兹曼机稳定的多

@它很难去学习生成离散的数据,就像文本

@相比玻尔兹曼机,GANs很难根据一个像素值去猜测另外一个像素值,GANs天生就是做一件事的,那就是一次产生所有像素, 你可以用BiGAN来修正这个特性,它能让你像使用玻尔兹曼机一样去使用Gibbs采样来猜测缺失值

**4 WGAN**

与DCGAN不同，WGAN主要从损失函数的角度对GAN做了改进，损失函数改进之后的WGAN即使在全链接层上也能得到很好的表现结果，WGAN对GAN的改进主要有：

◆ 判别器最后一层去掉sigmoid

◆ 生成器和判别器的loss不取log

◆ 对更新后的权重强制截断到一定范围内，比如[-0.01，0.01]，以满足论文中提到的lipschitz连续性条件。

◆ 论文中也推荐使用SGD， RMSprop等优化器，不要基于使用动量的优化算法，比如adam，但是就我目前来说，训练GAN时，我还是adam用的多一些。

从上面看来，WGAN好像在代码上很好实现，基本上在原始GAN的代码上不用更改什么，但是它的作用是巨大的

◆ WGAN理论上给出了GAN训练不稳定的原因，即交叉熵（JS散度）不适合衡量具有不相交部分的分布之间的距离，转而使用wassertein距离去衡量生成数据分布和真实数据分布之间的距离，理论上解决了训练不稳定的问题。

◆ 解决了模式崩溃的（collapse mode）问题，生成结果多样性更丰富。

◆ 对GAN的训练提供了一个指标，此指标数值越小，表示GAN训练的越差，反之越好。

总的来说，GAN中交叉熵（JS散度）不适合衡量生成数据分布和真实数据分布的距离，如果通过优化JS散度训练GAN会导致找不到正确的优化目标，所以，WGAN提出使用wassertein距离作为优化方式训练GAN，但是数学上和真正代码实现上还是有区别的，使用Wasserteion距离需要满足很强的连续性条件—lipschitz连续性，为了满足这个条件，作者使用了将权重限制到一个范围的方式强制满足lipschitz连续性，但是这也造成了隐患，接下来会详细说。另外说实话，虽然理论证明很漂亮，但是实际上训练起来，以及生成结果并没有期待的那么好。

**注：Lipschitz限制是在样本空间中，要求判别器函数D(x)梯度值不大于一个有限的常数K，通过权重值限制的方式保证了权重参数的有界性，间接限制了其梯度信息。**

**5 BEGAN: (不是EBGAN)**

BEGAN全称是Boundary Equilibrium GANs。BEGAN的主要贡献：

◆ 提出了一种新的简单强大GAN，使用标准的训练方式，不加训练trick也能很快且稳定的收敛

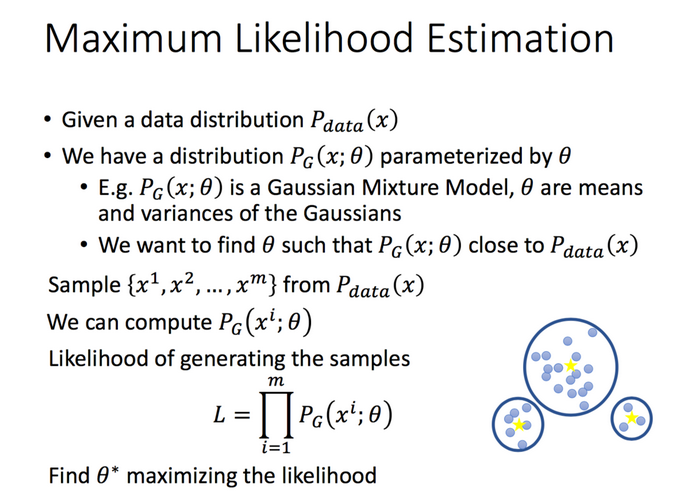
◆ 对于GAN中G，D的能力的平衡提出了一种均衡的概念（GAN的理论基础就是goodfellow理论上证明了GAN均衡点的存在，但是一直没有一个准确的衡量指标说明GAN的均衡程度）

◆ 提出了一种收敛程度的估计，这个机制只在WGAN中出现过。作者在论文中也提到，他们的灵感来自于WGAN，在此之前只有wgan做到了

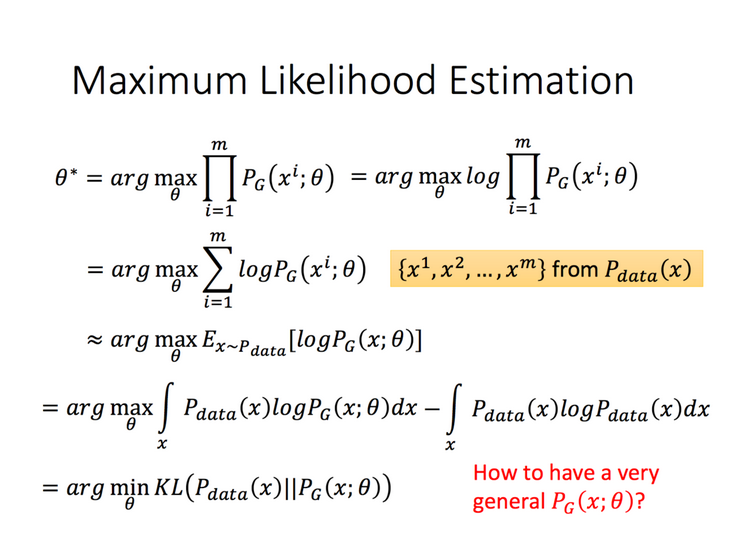
◆ 提出了一种收敛程度的估计，这个机制只在WGAN中出现过。作者在论文中也提到，他们的灵感来自于WGAN

**6 GAN原理深入**

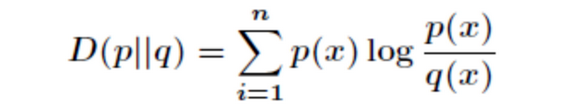
先来回顾一下我们的极大似然估计，假设我们有一大堆data，他的分布是Pdata(x)，我们可以认为这里的data就是一大堆图片，但是，我们有了这一大堆东西，再想生成一个新的data是不容易的，因为我们不知道这个分布的具体参数，所以，我们就想估计这堆数据的所服从的参数。那么，我们可以从Pdata(x)产生一大堆sample，然后，我们就希望找一组参数，使得服从这组参数的分布产生这堆sample的可能性最大。



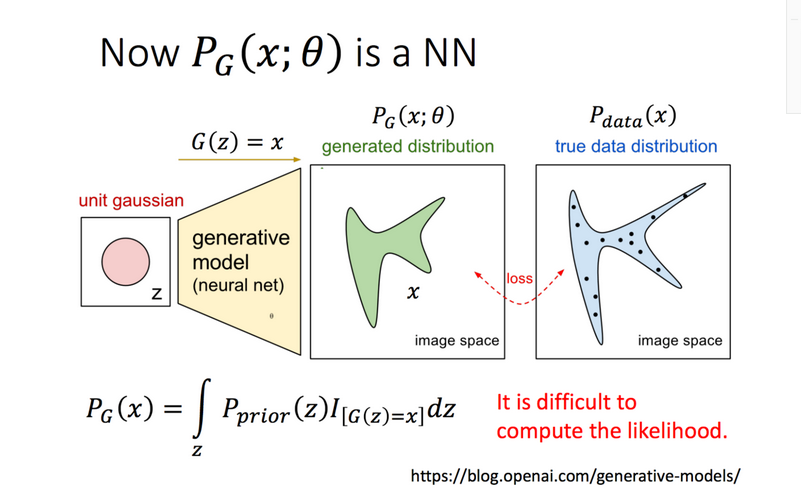
下面是极大似然估计的化简过程，因为这堆sample都是从Pdata(x)里面出来的，所以，我们可以进行下面的约等转换。然后我们把期望转换为积分，同时加上后面一项（后面一项是一个常数，只是为了更简单的表示KL散度）



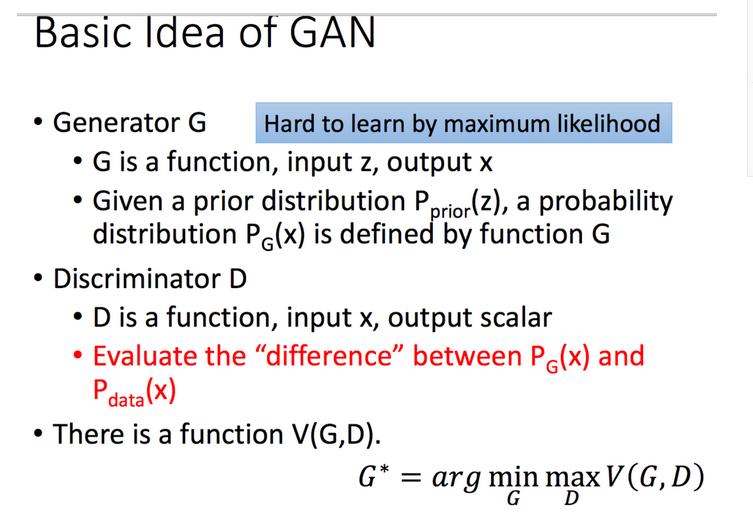
KL散度又称相对熵。设P(x)和Q(x)是X取值的两个概率概率分布，则对的相对熵为:



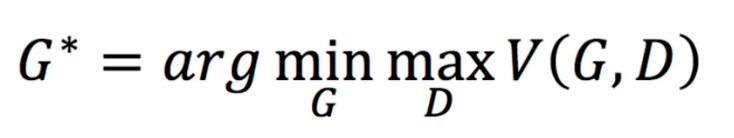
在一定程度上，熵可以度量两个随机变量的距离。KL散度是两个概率分布P和Q差别的非对称性的度量。KL散度是用来度量使用基于Q的编码来编码来自P的样本平均所需的额外的位元数。 典型情况下，P表示数据的真实分布，Q表示数据的理论分布，模型分布，或P的近似分布。那么，在GAN中，我们用NN的参数表示PG的参数θ：



那么 ，GAN的基本原理如下：

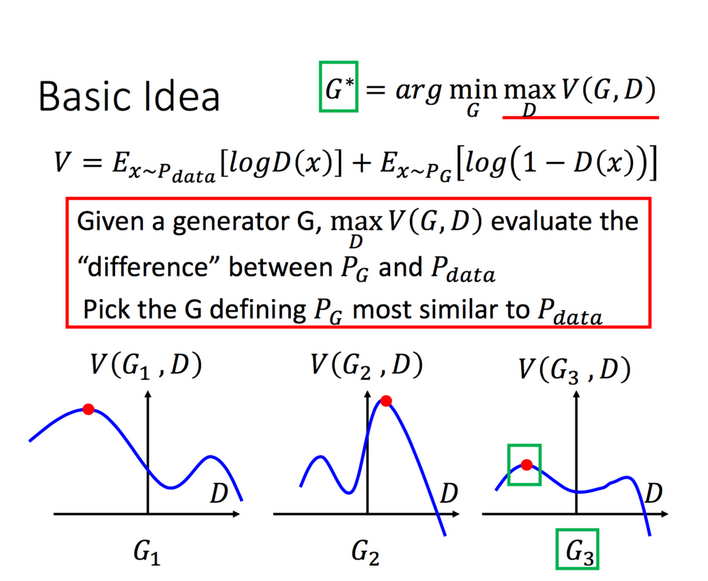


所以，我们最终的求解目标是：

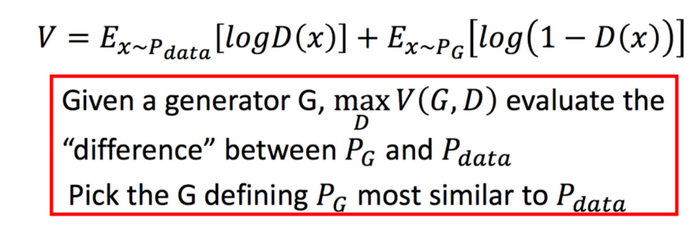


我们可以用下面的图来形象的表示上面的求解问题，看下面的图，我们可以很清楚的知道，我们要找的G就是G3，而D则是最高点的D。

那么 ，V是什么，V写作下面的式子，你不必管这个东西怎么来的，我只想说，能想到这个的人真的是太可怕了。当我们的V写成下面这样子的时候，我们取maxV(G,D)就饿能表示Pg和Pdata的差异。



为什么V要写成这样能表示二者的差异呢？这个可以通过严格的数学证明推导出，这里，对于一个给定的G，我们来求解maxV(G,D)：



对于任何一个常数，因为Pdata和G这里都是给定的，我们可以认为这里是常数，那么D取什么可以得到最大呢，很简单，导数为0的情况下。

**7 实际中的GAN**

刚才我们讲的是理论部分的内容，但是在实际中，Pdata和Pg我们是不知道的，我们没办法穷举所有的x，所以，我们只能采用采样的方法，同时可以采用我们二分类的思路，我们把Pdata(x)中产生的样本当作正例，把Pg(x)产生的样本当作负例，那么，下面V可以看作是我们二分类的一个损失函数的相反数(少了负号嘛）。也就是说，最大化V的话，其实就是最小化我们二分类的损失，下面的Minimize少了一个负号，所以我们要找的D，就是能使二分类的损失最小的D，也就是能够正确分辨Pdata和Pg(x)的D，这也正符合我们想要找的discriminator的定义，是不是很神奇！

所以，总结一下，实践中我们的GAN基于如下的步骤：

上面的步骤很好的解释了我们刚才对于GAN的解释：首先，我们又一个第一代的Generator，然后他产生一些图片，然后我们把这些图片和一些真实的图片丢到第一代的Discriminator里面去学习，让第一代的Discriminator能够真实的分辨生成的图片和真实的图片，然后我们又有了第二代的Generator，第二代的Generator产生的图片，能够骗过第一代的Discriminator，此时，我们在训练第二代的Discriminator，依次类推。

在实际中，我们有可能做下面的变换，可以加快我们的训练速度：