

本科毕业论文



|  |  |
| --- | --- |
| 论文题目： | 神经网络与大规模机器学习 |
| 院 系： | 计算机科学技术学院 |
| 专 业： | 计算机科学与技术 |
| 姓 名： | 戴涵俊 |
| 学 号： | 10300240030 |
| 指导教师： | 张军平 |

2014年 6月1日

神经网络与大规模机器学习

摘要：

关键词：

**Abstract:**

**Keyword:**

Table of Contents

1. 大规模神经网络平台IECABrain 6

1.1 神经网络综述 7

1.2 平台概览 9

1.2.1 动机 9

1.2.1.1 现有问题 9

1.2.1.2 需求 10

1.2.2 平台架构 11

1.2.2.1 架构预览 12

1.2.2.2 核心设计 14

1.2.3 平台特点 19

1.2 IECAMatrix矩阵库 21

1.2.1 动机 21

1.2.1.1 现有矩阵库 21

1.2.1.2 IECAMatrix特点 22

1.2.2 数据结构 22

1.2.3 模板框架 23

1.2.3.1 表达式引擎 24

1.2.3.2 表达式解析引擎 25

1.2.3.3 表达式执行引擎 25

1.2.4 性能对比 26

1.3 训练算法 28

1.3.1 反向传播 28

1.3.2 大规模训练方法 31

1.3.2.1 多模型并发 31

1.3.2.2 分布式训练 33

1.3.2.3 softmax近似 37

1.3.4 扩展功能 38

1.3.4.1 卷积神经网络 39

1.3.4.2 栈式自动编码器 40

1.5 平台性能测试 42

用递归神经网络进行搜索广告点击预测 43

神经网络与多义词的向量嵌入 44

# 1. 大规模神经网络平台IECABrain

## 神经网络综述

人工智能领域的研究一直具有十分重要的科学与社会意义。近年来，越来越多的学者致力于研究深度学习，并将其应用于语音识别、图像识别、自然语言处理等，并在各自的研究领域屡屡取得突破性进展。

深度学习是指利用计算机来创建深层次的人工神经网络计算模型，模拟生物神经网络的结构和功能，建立起复杂的模型进行学习与分析，从而令其可以按照人脑的机制识别图片、文字等信息。神经网络通常由大量的人工神经元联结构成，输入数据在其之间传导与计算，并且根据外界信息改变人工神经网络的内部结构，以此对输入和输出数据间复杂的关系进行建模。

1986年，Rumelhart和McCelland提出了误差反向传播算法，以此来对人工神经网络进行有监督的学习。自此，神经网络的研究与应用渐渐进入了人们的视野，一些科学家利用浅层的神经网络实现了汽车在公路上的自动驾驶[],之后又有科学家让计算机通过神经网络，学会了模仿人类的声音[]。慢慢的，人们发现人工神经网络具有一定的局限性，包括受限于当时硬件设备的计算能力，使训练神经网络耗时过长，并且不能处理规模过大的网络；同时网络加深层数后，由于当时神经元的激活函数普遍使用S型函数(Sigmoid)，误差在反向传播时随传播层数的增长呈指数级下降，所以通常多层神经网络只有最后几层学习到了有用的信息，导致其并未表现出比浅层神经网络更好的学习能力；另外，神经网络需要大量的数据进行学习，而当时的小数据集往往会出现过拟合现象，这些因素都导致了在整个90年代深度学习方面的研究进展很小。

1998年，Yann Lecun等人根据人眼感受野的工作原理，提出了首个进行了实际应用的卷积神经网络模型: LeNet-5。他将待识别的图片利用固定大小的卷积窗进行卷积以提取局部特征，并在卷积层后面添加了子采样层进行池化以增加神经网络对图像旋转、偏移的细微差别的鲁棒性。由于图片各个部位对应的卷积窗权值共享，所以极大的降低了参数的数量，加快训练速率，且不易发生过拟合。其卷积的特性在处理二维图像上对边缘信息特征的提取保证了其准确率。LeNet-5被广泛的应用在美国邮政编码与银行手写支票的识别。

2006年，Hinton等人为了克服深层网络误差传播的问题，提出了几种预训练的方法，包括受限玻耳兹曼机、栈式自编码、稀疏编码等方式，利用输入数据对神经网络的权值参数进行非监督的学习，使其预先学习到数据的分布信息，再利用标记数据进行监督学习以微调网络参数。利用这种方式训练出的多层人工神经网络具有非常好的表达。之后Hinton等人又提出了如Drop Out等在小数据集上抑制过拟合现象的方法，以及一些新的神经元激活函数如Rectified 函数等可以在误差反向传播时有效的传播梯度值。

在互联网时代，时刻都在产生大量的数据，而处理这些数据需要较高要求的硬件设备。近年来，图形处理单元(GPU)由于其优秀的计算能力以及并行度上的优势，已经逐渐代替CPU来进行大规模密集型科学计算。英伟达公司最新推出的Tesla K40 GPU，单精度浮点性能已达到4.29TFLOPS。Andrew Ng等人在论文中描述了他们将神经网络模型放到分布式GPU集群上以提高训练的迭代速度。至此深度学习的研究又进入了一个新的阶段。

随着人工神经网络的逐渐成熟，人们将其应用到图像识别、文本学习等领域，并取得了不错的成果。MNIST是一个手写数字数据集，其中包含共60,000个训练样本以及10,000个测试样本，每张图片大小为28\*28的0-255表示的灰度图。在2013年，Li Wan等人使用一种名为DropConnect的神经网络结构，在此数据集上达到了99.79%的分类正确率。CIFAR-10是一个彩色图片数据集，其中包含了10个物体种类共60,000张32\*32的RGB三色图片，其中50,000张训练样本, 10,000张测试样本。在2014年，Min Lin等人提出一种名为”Network In Network”的神经网络结构，在此数据集上达到了91.2%的分类正确率。Word2vec 是谷歌公司发行的一个用神经网络计算连续分布式词汇表达的工具。用word2vec在英文维基数据集上进行训练，并在单词相似度推理任务中可以达到61%的正确率。

为了进行深度学习方面的研究，搭建一个快速、准确、灵活的神经网络平台至关重要。Theano是一个Python的机器学习库，用户可以使用它进行神经网络方面的研究与应用。支持GPU, 开发快捷是Theano的优点。Matlab拥有自己的Neural Network Toolbox，其支持监督学习的前向传递、动态的网络等，同时也支持非监督学习的自组织映射等。使用这个工具箱，用户可以设计、训练神经网络，其最大的特点是具有可视化以及仿真功能。Cuda-convnet 是Hinton等人开发的卷积神经网络 C++/CUDA实现。其主要用于GPU，用户可自定义自己的网络结构，它的优点是训练速度较快。CAFFE平台与Cuda-convnet类似，也是用C++实现的卷积神经网络，不过其在CPU上也有很好的表现。Accord.NET是用C#实现的机器学习库，它支持深度置信网以及受限玻耳兹曼机。

## 1.2 平台概览

这一节主要介绍大规模神经网络平台的高层次信息。首先将阐述我们构建IECABrain的动机，通过对现今学术界和工业界的神经网络平台进行回顾，我们会发现他们具有各种不同的问题，而无法满足我们的需求；接下来，将概括性介绍我们开发的神经网络平台的设计架构和实线理念，具体的实现方法和算法将留在后面的章节中具体阐述；最后，我们总结新构建的平台的特点。

### 1.2.1 动机

构建一个通用的神经网络平台是一件需要大量精力与时间的工作。为了能够支持我们后续的研究工作，我们有必要构建适合自己的神经网络平台——IECABrain。以下将通过横向对比现有的神经网络平台、以及结合自身研究课题需求，阐述IECABrain诞生的意义，以及其所要完成的任务。

#### 1.2.1.1 现有问题

在当今的学术界和工业界，大数据和机器学习已经成为了热门关键词。而大数据带来的不仅仅是机遇，还有对现有机器学习算法以及软件开发人员的巨大挑战。神经网络在最近兴起的深度学习中，又再次引起了一波研究热潮。不过，由于多层神经网络的训练并不是一个凸优化问题，因而要训练好一个神经网络，即防止得到较差的局部最优解，需要很多的技巧与经验。

通过上文中对现有的神经网络平台的介绍，我们对其特点有了初步的掌握。在开发我们自己的平台之前，我们对这些现有的平台进行了详细的调查。通过安装并运行一些标准数据集，我们发现这些平台有如下的一些问题：

1. **Matlab**

Matlab一个主要的问题是非免费。它是个收费软件，因而要部署在集群上，就需要有多个许可证，而其收费也是非常高昂。Matlab程序编写容易所带来的代价便是效率的降低，尤其是和优化过的C++代码的效率相比。另外，在Matlab的神经网络工具包中，其包装的完整性也使得我们并不方便进行深层次的定制，而用其进行神经网络的研究，必须要研究、改变其底层结构和实现。这些原因使得Matlab不满足我们的研究需求。

1. **Theano**

虽然Theano可以编译成C代码执行，也可以方便地利用GPU运算，不过效率上还是不如优化过的C代码。另外，Theano更多的是可以看做一个符号运算的执行器，相对于神经网络来说，负责的是更底层的运算。考虑到后续开发和维护的代价，用高效的C++更为合适。

1. **Pybrain**

该项目已经多年没有更新了。虽然是一个完整的神经网络库，并且在设计方面有其独到之处，但是效率问题（并不支持MKL以及CUDA优化）以及扩展能力的不足（不能进行分布式运算），使得其离我们的需求还很远。

1. **FANN**

该平台虽然完全用C代码编写，但是由于没有用并行优化，也难以扩展到GPU平台和分布式框架，所以更多的适合于在单节点上小数据集的调试。另外，神经网络训练算法也固定在其设计中，导致批量梯度下降、随机梯度下降等方法不适用。

1. OpenCV
2. **Caffe**

该平台更多的偏向于使用，而并不是二次开发。一方面，它并不支持分布式的神经网络训练；另外，虽然在Caffe平台下配置一个神经网络，只需要写少量的标记语言，但是，这也限制了用户能够修改的部分。对比我们对平台的要求：适于应对研究工作中各种修改、有分布式计算的能力，Caffe显然也不是一个合适我们当前任务的选择。

#### 1.2.1.2 需求

为了支持神经网络相关的研究工作，一个强大而高效的神经网络平台是必须的。具体而言，我们当前主要有两方面的任务：1）计算广告学中的搜索广告点击预测问题；2）自然语言处理中的词向量空间嵌入问题。以下分别针对这两个任务阐述其难点以及对神经网络平台的需求。

**任务一、计算广告学中的搜索广告点击预测问题**

作为新兴的分支学科，计算广告学涉及到了大量研究内容，如机器学习、信息检索等。而作为一个典型的计算广告学的应用，搜索广告占有了重要的地位。通过在搜索引擎中投放广告，进而通过用户的点击进行收费，这种模式已经成为了大量的互联网公司的重要盈利手段。而广告平台对广告主收费的重要依据就是广告的点击，因此，投放用户最可能点击的广告，是广告投放平台和广告主都关心的问题。而广告点击预测最为显著的特点，就是——数据量巨大。平均而言，主流的搜索引擎（如Google，百度，Bing等）每日的搜索量都在10亿级别。每次用户的搜索，系统都会投放一定数量的广告。如果我们将过去一个月的用户搜索广告点击行为数据收集起来，用机器学习的方法去训练，将是对所用的算法以及系统一个巨大的挑战。

另外，用户的行为也在随着时间变化。而且广告点击的行为不仅仅与用户相关，还与当前的热点事件、节假日等有密切关系。因此，训练出来的模型也需要经常更新。一般可以用在线学习的方式，也可以每两到三天更新一次。

我们不难从上面总结出，在这个问题中，如果用神经网络模型作为机器学习的工具，那么就需要训练网络的平台具有如下的特点：

1. 平台具有很好的扩展能力。这种扩展能力主要体现在当计算资源变多了之后，我们能够有办法利用大量的服务器去提升计算能力，从而在短时间内训练模型，以满足计算广告的问题中关于时效性的需求。因而，该平台能够有分布式计算的能力。
2. 在计算广告问题中，常常会需要用bag-of-words形式的向量，即所谓的ID特征。考虑到作为特征的对象的规模常常非常大，例如广告的数量、用户的数量。这些将导致一个问题——由于单个节点的内存大小有限，我们的模型甚至无法在单个运算节点（服务器）上面存下来！所以，平台需要有能力将模型的参数也分布式存储。

**任务二、自然语言处理中的词向量空间嵌入问题**

在传统的文本处理方法中，常常采用向量空间模型（Vector Space Model, VSM）。这种模型将文档表示为一个向量，而向量的维度就是词表的大小。向量中的每个元素对应了一个词，可以是这个词的词频，也可以是TF-IDF等特征。这种表达方式没有对单词进行细致的描述。

所谓的词向量空间嵌入，就是将单词也用向量来表示，进而，可以通过在向量空间中的操作，来寻找词与词之间的关系。譬如，我们可以用词向量的距离，来表征两个词的相近程度。在后文中，我们将详细阐述词向量空间嵌入的问题。

词向量嵌入有很多种方法。用神经网络训练的自然语言模型（Neural Network Language Model, NNLM）是最近较为流行的方法。当然，为了训练出较好的自然语言模型，大量的训练语料是必不可少的。一个大规模的语料库，除了篇幅巨大（例如数百GB的文学资料）之外，包含的词的种类也巨大（数以亿计的单词、词组、特殊意义的字符数字组合，等等）。所以，这种任务同样需要平台具有以下的能力：

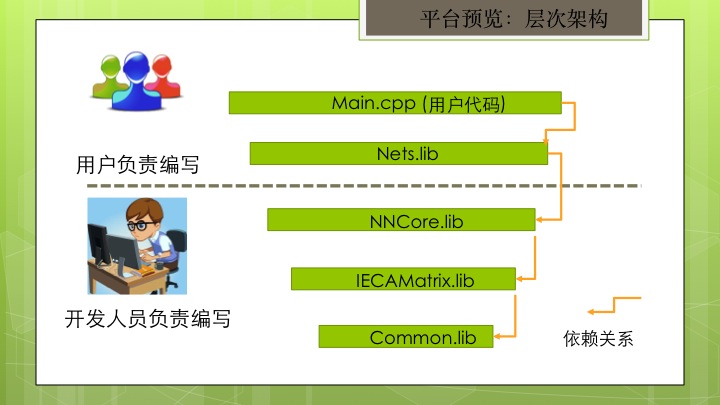
1. 平台能应对规模巨大的神经网络。该需求和任务一中的需求b类似，即需要能够分布式存储巨大的模型。
2. 平台需要高效。高效不仅仅需要计算上速度快，还要求其带来的额外冗余开销小。在神经网络自然语言模型中，单个词样本的训练计算代价非常小，但是由于训练样本数量巨大，因此总的代价也会很大。如果在单个词样本的训练上有多余的额外开销，那么整个训练过程的开销就非常可观了。另外，训练语言模型常常需要多次迭代训练一个语料库，这些问题都导致了对平台性能的高要求。

### 1.2.2 平台架构

在这一小节，我们将首先从大规模神经网络平台IECABrain的层次架构开始，介绍平台中各个部分的依赖关系；接下来会按照层次，从底层向高层介绍IECABrain，首先我们用一个小节介绍我们自主开发的矩阵运算库，这个矩阵库支撑了整个平台的核心运算，接着，便是核心部分的设计介绍，这包括了网络核心的各个部件，以及其组织逻辑。具体的大规模学习算法，以及神经网络的训练优化方法等，将留到第四章节详细阐述。

#### 1.2.2.1 架构预览

像众多软件的设计方法一样，IECABrain也采用多层次架构进行设计。具体来说，我们将架构分为五层，如下图：



可以看到，为了最大限度地让我们的平台易于扩展，需要用户编写少量的代码——也就是说，我们的平台不是一个傻瓜式的黑盒，只需要输入参数和数据即可，这不是我们的初衷。我们的目的是构建一个适合进行关于神经网络和深度学习的深层次研究的平台。而这就需要平台易于扩展，同时也要易于维护。

除了最顶层的用户代码，其余几层都被编译成了静态链接库。相较于将所有层次构建在一个项目编译输出，这种设计能够减少开发过程中编译的代价。同时，也易于之后的维护。

下面，我们由底层到顶层，分别介绍每个层次所负责的功能。

1. **Common.lib**

Common是通用的库，顾名思义，这个是最底层的，与神经网络几乎无关的库。该库主要包含许多的工具类，后文中将会直接使用到其中的一些类，由于其实现方法不是本文讨论的重点，所以统一在这里进行列举说明。主要用到的类有：

1. **Dictionary**

一种按照键-值对组织的数据结构。通过可以计算哈希编码的键来索引对应的值。核心部分是一个哈希表。这种实现方式平均情况下会比平衡树来的更高效，当然，是以一定空间代价作为交换的。

1. **HuffmanEncoder**

一个自动进行哈弗曼编码的工具。通过给定每个编码对象的字符串，以及该对象的权重（譬如单词的词频），该工具能够自动构建哈弗曼树，同时能够知道每个编码对象的编码、其在哈弗曼树上的路径等重要信息。

1. **CMultinomialSampler**

一个随机采样工具。该工具能够进行多项分布的采样。通过给定初始的项数和每项对应的概率，该工具能够很高效地对给定的多项分布进行采样，同时，其基于c++的伪随机数，方便调试。

1. **Log**

一个记录日志的工具。相比较工业界流行的glog之类的工具，这个工具更加轻量级，并且易于部署、管理。

更多的类和每个类各自的实现方式，请参考源代码。在后文中，这一个库就不会再进行更多的介绍。

1. **IECAMatrix.lib**

神经网络中，绝大多数操作都可以转化成矩阵运算。例如，两层网络之间的前向传播，就可以用一个矩阵乘法来表达。将这层运算用矩阵封装后，在更高层次，我们就只需要关注神经网络算法的逻辑，而不需要花费更多精力在具体运算实现上面了。

因此，我们有必要建立一个矩阵库，专门负责这些操作。IECAMatrix就是这样的一个通用矩阵库。之所以称之为通用，是因为它不仅可以服务于IECABrain这个神经网络平台，还可以用于其他的项目中。

然而，这个矩阵库有两个最主要的特点：统一性和优雅性。我们将在下一章节，**IECAMatrix矩阵库**中，进一步介绍。

1. **NNCore.lib**

该库包含了我们神经网络平台的核心部件。在设计的逻辑中，两个最主要的类——层（Layer）和连接（Connection）将在这一个层面实现。

除了构成神经网络的层和连接这些部件之外，还有配套的许多其他部件，例如流式数据读取、分布式计算、多线程并行，以及大规模神经网络训练的优化方法等，都会在这一层面实现。因而，这个库是整个神经网络平台的核心。

当然，这里不仅仅有一系列单独的部件，还涉及到这些部件如何协同工作的逻辑。这些内容将在下面的章节**核心设计**中进一步介绍。

1. **Nets.lib**

从这里开始，就要涉及到用户编写的代码了。有了NNCore这个核心部件，构建各种神经网络，就只需要组装这些部件了。而不同的神经网络的差别，大多也是在结构上、层和连接表现行为上的差异，因此，留给用户写的代码并不是很多。在**核心设计**这一节中，也会展示一个通用的抽象神经网络，即，通过配置不同的部件，同一个神经网络类可以展现完全不同的功能。

1. **Main.cpp**

这里完全是用户自己编写的代码。主要涉及到用户的个性化配置，例如矩阵运算用CPU还是GPU实现，数据文件存放在哪里，训练神经网络的时候，各个可以调控的参数该如何设置，等等。

至此，我们按照层次结构从底层到高层，完整介绍了我们的平台概况。

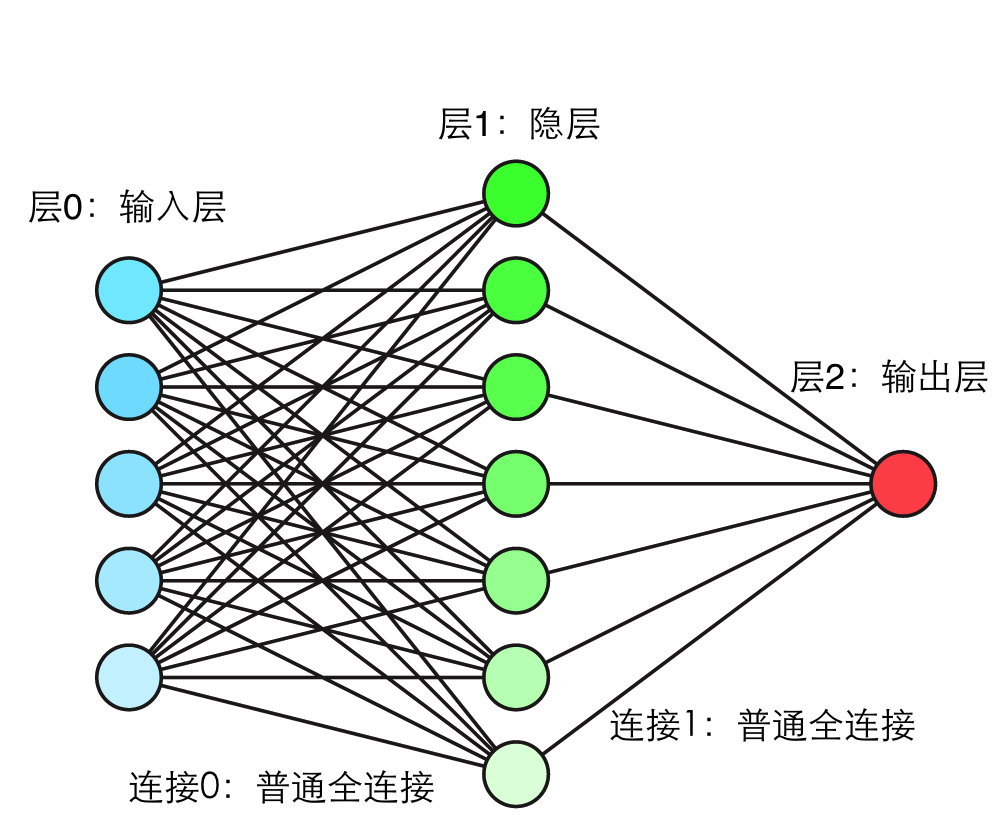
#### 1.2.2.2 核心设计

在这一节，我们主要介绍上文中提到的NNCore.lib，也就是神经网络平台的第三层。通过对各个部件的功能以及对其组织方式进行介绍，我们将从整体上把握整个平台的设计理念。在第三章**训练算法**中，我们会详细说明各个部件的实现方法，以及在各种标准数据集上，与现有平台的性能对比。

我们首先分门别类介绍各个部件。接下来，我们按照各个部分负责的逻辑上的功能，用示意图结合文字的形式，介绍它们组合运作的方式。

##### 1.2.2.2.1 层与连接

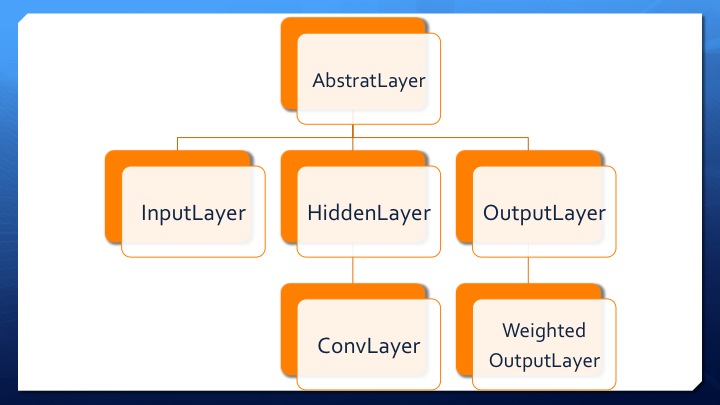
神经网络结构中，层和连接是最关键的两个组成元素。下图展示了一个普通的三层全连接神经网络，在这图上，我们定义了层和连接这两个概念。



上图中，一共有三层（层0~层2），两个连接（连接0和连接1）。因此，在我们的定义中，一层对应了一个神经元的集合，而一个连接，对应了连接两层之间的边。以上是数据结构层面的定义，而神经元和边连接又可以有多种表现行为，比如神经元的激活方式、边的前向传播方式以及反向更新方式等，都可以有变化。因此，我们将这两个元素——层和连接，用以下的数据结构来表示。

* **层**

如上所述，层是一个神经元集合的组织方式。而层的标号（0，1，2，等等）并不一定需要有顺序关系（即层1不一定代表着在层2之前），而仅仅是用来作为层的一个名字，用来唯一标识层。这种定义方式给我们平台带来了很大的灵活性——我们可以自己定义层的拓扑顺序。



上图是核心部件——层的类图。下面选取主要的类介绍。

* + **AbstractLayer**

是所有层的抽象父类。定义了层所拥有的基本数据成员，以及主要的前向和反向传播的运算逻辑。这里有必要介绍一下其所拥有的基本数据成员。

template<MatMode mode>

struct AbstractLayer

{

public:

IMatrix<mode>\* activations; //存放激活值

IMatrix<mode>\* errors; //存放反向传播误差

unsigned uThreadReplicaID; //多线程的ID

std::vector<Edge> inConns, outConns; //连接集合

const AbstractLayerInfo<mode>\* pLayerInfo;

//层的配置信息，如层的大小、编号，等等。

};

可以看到，层的数据用矩阵IMatrix来存储。

* + **InputLayer**

输入层负责将数据读入到当前的神经元中，作为输入层的激活值，并用来做前向传播。由于输入层不需要更新，因而不需要做反向传播。和隐层以及输出层最大的区别是，其神经元的激活值是从数据中读取而来的。

* + **HiddenLayer**

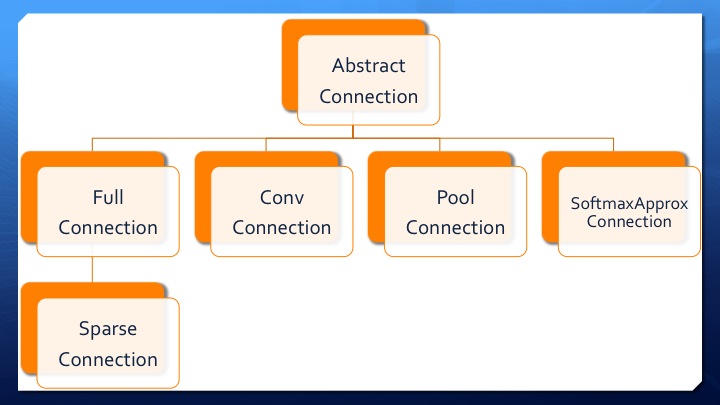
神经网络的隐层。负责前向传播和反向传播。

* + **OutputLayer**

神经网络的输出层。除了负责反向传播之外，还需要负责载入监督信息，以计算当前的输出值与标准值的偏差，用来反向传播误差。

* **连接**

故名意思，连接即是在两个层之间存在的边。连接可以有多种方式，比如全连接（Full Connection），也可以局部连接（Local Connection），或者相对于普通的前向传播，可以有随机丢失连接（Drop Connection），可以用卷积连接，等等。每种连接有着自己的特点，当然，他们也有着共同的地方，所以和层一样，我们也用子类继承方式组织这些不同的连接。另外，连接的编号也只是用来唯一区别，并不代表顺序或者其它一些信息。



这里只介绍最主要的抽象类。

* + AbstractConnection

是所有连接的抽象父类。同样包含了基本的数据成员。如下所示。同样，连接所对应的边用矩阵来存储。

template<MatMode mode>

struct AbstractConnection

{

public:

IMatrix<mode>\* pWeights; //连接的边权

IMatrix<mode>\* pBias; //神经网络的偏置项

unsigned uConnID; //连接的编号

BiasStatus biasStatus; //是否需要使用偏置

MatFormat format; //连接是稀疏还是稠密

};

其它一些类将在后文中具体介绍。

##### 1.2.2.2.2 有向无环图结构

IECABrain平台的一大特点，就是可以训练任何结构合法的神经网络。我们对结构合法的定义如下：用图论的观点，如果把层看做图中节点，把网络连接看做图中连接两个节点的有向边，其中边的方向就是神经网络前向传播的方向，那么，该图是个有向无环图（DAG）。

如果把神经网络看做图之后，图中存在环的话，那么这种训练行为是未定义的，必须得有额外的限制条件。所以，IECABrain在这一方面，已经是最大程度地支持灵活的网络结构了。

维护该图结构的是CDAGStructure这个类。该类的主要方法和数据结构如下：

template<MatMode mode>

class CDAGStructure

{

public:

//判断网络结构是否合法，并保存经过拓扑排序的层和连接

bool IsStructureValid();

//添加新的层

bool AddLayer(AbstractLayerInfo<mode>\* pLayerInfo);

//给两个层之间添加新的连接

bool AddConnection(unsigned uLayerFrom, unsigned uLayerTo, AbstractConnection<mode>\* pconn);

// 保存添加的层的信息

std::vector<AbstractLayerInfo<mode>\*> m\_layerinfo;

// 将层按照拓扑顺序存放

std::vector<AbstractLayerInfo<mode>\*> m\_layerInOrder;

};

从上面的结构可以看出，为了构造一个神经网络，我们将会像构建一个DAG一样，添加层和连接。经过拓扑排序，我们除了能够验证网络结构的合法性，还能得到按照拓扑顺序排列的层。不难想到，如果按照拓扑顺序对这些层进行前向传播以及逆拓扑序进行反向传播，我们就可以像训练普通的三层神经网络一样，训练一个多层的符合拓扑顺序的网络。

##### 1.2.2.2.3 流式数据提供

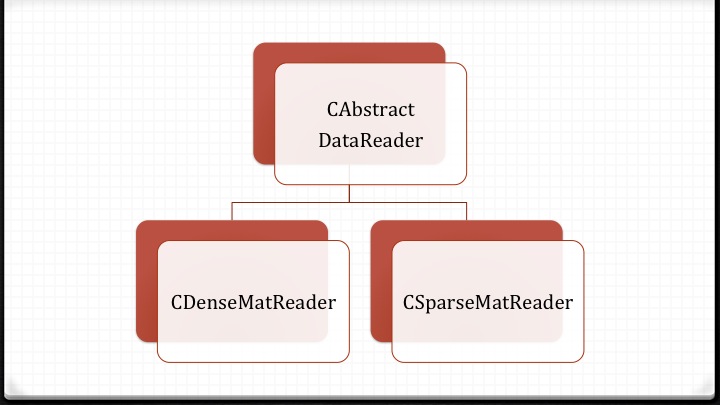
要让神经网络的训练过程收敛到较好的解，大量的训练数据是必不可少的。而数据量大了之后，我们便不可能将所有的数据都存放在内存中直接使用了，因此，我们设计了流式的数据提供方式——在这种方式下，只要时间允许，就可以训练无穷数量的数据。

这部分内容将会涉及到部分优化算法的问题，具体内容将在1.3节介绍。神经网络常常使用批量梯度下降进行训练。每次，一批数据会被读入内存，通过前向传播和反向传播对神经网络进行更新，接着下一批数据被读入。如此反复直到将所有的数据都通过神经网络，一个周期的训练便结束了。

用来提供流式数据的类是CDataSupplier。该类不负责读入数据，而只是将读入的数据进行缓存。输入层通过访问CDataSupplier来获取当前批次的数据。有了这个结构，在分布式框架和多模型训练情况下，输入层不需要考虑数据同步等问题，直接交由CDataSupplier即可。

而具体负责读入数据的，是DataReader所负责的。由于数据格式多种多样，为了使得能够适配我们的平台，需要有相应的数据适配器，这就是DataReader所负责的事情——把不同格式的数据都整理成神经网络能够接受的矩阵形式。

首先，来看看DataReader的组织结构以及对应接口的用法。



其中，CAbstractDataReader规定了所有数据适配器（Reader）必须实现的接口，以及一些共有的数据成员。两个具有代表性的子类CDenseMatReader和CSparseMatReader分别是读入稠密矩阵和稀疏矩阵的适配器。用户可以遵循这一原则，自主开发数据适配器来匹配自定义格式的数据。

class CAbstractDataReader

{

public:

//打开硬盘上的数据文件

virtual void Open(std::string filename) = 0;

//获取批量数据（转换成矩阵形式存储）

virtual unsigned LoadBatchSamples(IMatrix<MatMode::CPU> \*&batch, unsigned uBatchSize) = 0;

//关闭数据文件

virtual void Close() = 0;

unsigned m\_uID; //数据适配器编号

protected:

FileStorageMode m\_fStoreMode; //文件存储格式，二进制或者文本格式

DataShuffleMode m\_dShuffleMode;

std::default\_random\_engine m\_rand\_gen;

};

以上是数据适配器的抽象父类的接口定义。

在第1.3节算法部分，我们还将结合分布式以及多模型的情况，介绍如何用流式数据训练，以及这些类之间如何协作。

### 1.2.3 平台特点

IECABrain总结了现有的平台的优点，以及最新的学术界研究成果。它在如下几个方面具有突出的特点：

* **效率**

IECABrain完全使用高效的C++编写，其计算效率和算法效率都进行了深层次的优化。

* + **矩阵计算**

底层基于自主开发的IECAMatrix矩阵库。该矩阵库可以分别在CPU和GPU平台使用。在CPU平台利用MKL提供的针对Intel指令集和多核心优化，在GPU平台则依赖CUDA提供的并行运算，同时加上特有的延迟计算优化，使得矩阵库本身就比其它一些常见的矩阵库要高效。

* + **算法优化**

一方面，通过softmax近似算法，大大减少了分类问题中输出层计算代价；另外一方面，利用多模型并行训练，使得训练周期缩短。

* **能力**

IECABrain能够同时处理大数据和大模型，具体表现在：

* + **流式数据训练**

通过流式数据和批量梯度下降，IECABrain能够训练海量的数据，而仅需基本的内存需求。

* + **分布式计算**

通过将模型分布式存储，只要有足够的节点，IECABrain就可以训练任意规模的神经网络。

* **易用性**

在底层运算方面，IECAMatrix使用运算符重载，以最自然的算术表达式形式执行矩阵运算。

在神经网络层面，用户只需要像搭积木一般，将层和连接按照自己想要的方式组合起来，设置好需要的训练参数，即可以使用强大的IECABrain。

* **扩展性**

IECABrain本身的初衷便是成为一个适合研究和拓展的神经网络平台。平台中的每个部件几乎都可以由用户自己进行扩展。从底层的数据接口，到核心部分的层和连接的表现行为，再到高层的神经网络构建和训练方式，都可以按照约定的接口实现自己所需要的功能。

## 1.2 IECAMatrix矩阵库

为了支持IECABrain的底层运算，我们构建了自己的矩阵库——IECAMatrix。该矩阵库是通用的用于矩阵算术运算的库。首先，我们将从动机出发，阐述构建该库的原因，以及其所具有的特性；然后，我们从实现角度，介绍IECAMatrix的数据结构，以及其特性之一——C++模板框架。利用该特性，我们就能用少量的代码，实现符合自然语言习惯的矩阵算术运算表达。最后，为了验证其高效性和有效性，我们用常用的矩阵乘法，对比IECAMatrix和其他一些平台的性能。

### 1.2.1 动机

IECAMatrix是一个通用的矩阵运算库。其同时支持稠密矩阵、稀疏矩阵和稠密矩阵与稀疏矩阵之间的运算。另外，该库可以在CPU平台或者GPU平台运行。这些功能满足了依赖该矩阵库的IECABrain的各个需求：1）CPU与GPU平台通用；2）同时支持稠密/稀疏特征，以及稠密/稀疏网络连接。可以说，IECABrain的灵活特性，很大一部分依赖于IECAMatrix的灵活。

下面，我们首先对现有的流行的矩阵库进行介绍；接着，通过进一步介绍IECAMatrix，以及对比现有的矩阵库，将阐明构建IECAMatrix的动机。

#### 1.2.1.1 现有矩阵库

这一节，我们将介绍工业界和学术界常用的几个矩阵运算库的特点，同时对比各自的优缺点。

* **Matlab**

Matlab中同时支持稠密矩阵和稀疏矩阵，以及两者之间的转换和相互运算。另外，Matlab也可以同时在CPU平台和GPU平台进行运算，而且其矩阵算术运算的表达也符合自然语言习惯。但是，相比较高效的C++代码实现，Matlab的矩阵库就稍逊一筹。另外，如果混用C++和Matlab代码，中间的程序接口也会需要一定代价。

* **OpenCV**

OpenCV拥有自己的矩阵运算库。该库同样支持稠密矩阵的运算，并且在CPU上能够使用MKL加速。但是，在GPU上的运算接口与CPU中的不同，同时，稀疏矩阵并不像稠密矩阵那样获得足够的支持。实现的效率不够高也是缺点之一。

* **Eigen**

Eigen是一个开源的矩阵库。除了能够进行基本的矩阵算术运算之外，还实现了矩阵分解等算法。Eigen能够在CPU平台运行，并能够使用Intel的MKL加速，用C++实现的模板类也非常高效。不过，它并不支持GPU运算。同时，虽然Eigen同时支持稀疏矩阵和稠密矩阵，不过这两个矩阵之间并不方便进行抽象，如果使用Eigen，这个问题会给IECABrain的开发带来一定困难。

在后面的小节中，我们将把IECAMatrix和这几个矩阵库放在一起进行性能测试，通过对比来证明IECAMatrix的性能。

#### 1.2.1.2 IECAMatrix特点

IECAMatrix具有的特点如下：

* **统一**

在矩阵运算中，我们统一表达了CPU/GPU上的运算，以及稠密/稀疏矩阵的存储。

具体而言，IECAMatrix记录了模板变量MatMode，即矩阵的运算模式。模式可以有CPU运算模式和GPU运算模式。另外，通过C++的多态，我们能够用统一的接口访问具体的矩阵实例——即稠密矩阵或者稀疏矩阵。

通过这种方式，我们可以在仅修改配置参数的情况下，让IECAMatrix应对不同类型的计算。

* **优雅**

在矩阵算术运算方面，我们重载了C++的运算符，使得矩阵算术运算能够像一般变量的算术运算一样书写。通过重载C++的“+”，“-”，“\*”和“/”运算符，我们实现了矩阵的加法、减法、乘法、数乘，以及矩阵和矩阵之间逐元素的运算。

#define MODE MatMode::CPU

auto a = DenseMat<MODE>(10, 10);

auto b = SparseMat<MODE>(10, 5);

auto c = DenseMat<MODE>(5, 10);

a -= b \* c \* 0.001f;

以下的一段代码，实现了矩阵的乘法、数乘和减法。

* **高效**

在CPU平台，我们采用Intel的MKL运算库支持；而在GPU平台，使用CuBlas实现底层的矩阵运算。另外，由于使用了表达式模板引擎，使得我们能够对表达式本身进行优化，一方面减少临时变量的开销，另外一方面也减少了冗余运算。

在1.2.4节性能对比这一节中，我们将进一步证明这一点。

### 1.2.2 数据结构

为了统一稠密矩阵和稀疏矩阵，我们用了一个抽象矩阵类IMatrix，它的两个子类，分别为SparseMat（稀疏矩阵）和DenseMat（稠密矩阵）。

通过IMatrix的data域，我们可以访问到具体子类的数据结构。

对于稠密矩阵，我们存储的数据结构如下：

template<MatMode mode>

struct DenseData

{

public:

int rows, cols;

matEleType\* pDense;

};

其中matEleType是预先定义的浮点数类型，可以是单精度，也可以是双精度。可以看到，实际存储二维矩阵元素的，是一维数组pDense。

对于稀疏矩阵，我们存储的数据结构如下：

template<MatMode mode>

struct SparseData

{

public:

matEleType\* pVal; //非零值

int\* p1BasedRowIdx; //非零元素行号

int\* p1BasedColIdx; //非零元素列号

int\* p1BasedCSCColPtr; //CSC格式列指针

int nnz; //非零元素个数

int ncol; //CSC格式列数

int nzCap; //非零元素个数上限

int colPtrCap; //CSC格式列数上限

};

这里，我们使用CSC的稀疏矩阵存储格式。同时，为了访问方便，也采用COO的存储方式。这些均是通用的稀疏矩阵存储格式，具体可以参考相关文档。

用户可以直接对这些域进行修改，也可以使用预先编写的常用函数，获取所需的矩阵运算功能。

### 1.2.3 模板框架

为了方便地实现符合自然语言习惯的矩阵算术运算表达，IECAMatrix采用了C++的模板。通过用模板抽象出表达式，再利用模板匹配，求解对应的具体表达式，一个从抽象到具体的自上而下的设计框架就呈现了出来。

整个模板框架自上而下分为：1）表达式引擎，负责表达式的构建；2）表达式解析引擎，负责解析构建好的表达式，调用具体的运算执行引擎；3）表达式执行引擎，对具体的运算表达式，判定运算操作数和操作符，并调用相应的库函数（MKL或者CuBlas，分别对应CPU和GPU的运算）。

#### 1.2.3.1 表达式引擎

表达式引擎，其主要职责是负责构建表达式树。具体而言，表达式引擎将一个完整的表达式按照其执行顺序（优先级）组装成一棵表达式树，再将该树交由表达式解析引擎进行解析。

在一个表达式中，有很多的子表达式。为了方便之后的解析，每个子表达式应该足够纯粹，使得每个子表达式都能用单一的解析引擎解析，而不需要有过多的依赖关系。下图是属于表达式引擎的各个元素之间的对应关系。

* **Expr**

表达式的抽象父类。该父类只有两个成员变量，即rows（行数）和cols（列数）。而Expr类最重要的两个函数，就是得到其衍生类的方法。

template<typename DerivedType>

struct Expr

{

public:

int rows, cols;

inline DerivedType& Derived()

{

return \*static\_cast<DerivedType\*>(this);

}

};

每个Expr的子类，会将自身的类型作为模板参数。这样，在仅知道抽象类的情况下，也能获得具体子类的实例。

* **LvalueExpr**

即左值表达式类型。该类型的表达式可以放在赋值符号的左边。典型的这种表达式类型有矩阵和子矩阵。

* **MultiplyExpr**

矩阵乘法表达式。这种表达式区别于矩阵加法和减法表达式。

* **BinaryCwiseExpr**

矩阵和矩阵之间逐元素的运算表达式。包括矩阵加法、减法和数乘。

通过重载运算符，我们可以根据运算符以及运算符两边的操作数，决定构建的子表达式类型。

#### 1.2.3.2 表达式解析引擎

在1.2.3.1节，我们介绍了如何构建表达式树。因此，真正的表达式计算，并不是在构建表达式树的过程中进行的，而是在赋值符号解析的时候。

解析表达式树的目的，是将封装好的表达式树按照运算优先级关系，将每个子表达式交予对应的表达式执行引擎运算，同时也为了便于充分对整棵表达式树进行运算和表达的优化。

表达式解析引擎分为三类：

* **UnaryCwiseEngine**

解析单元表达式。例如，A += B，A \*= 2.0。这种表达式的右边只有一个操作数，因此可以针对这种表达式做特殊的优化。

* **CwiseExprEngine**

解析二元逐元素运算表达式，例如A = B + C。这种情况下，我们可以直接将B + C的结果存入A，而不用先新建临时变量存储B + C的结果，再将结果赋值给予C。

* **CmplxExprEngine**

解析复杂表达式。例如矩阵的转置，以及矩阵乘法。值得注意的是，除非矩阵的转置会被赋予到新的矩阵中，否则，该转置操作不会被真正执行，只需要记录标记即可。这样可以免除新建临时变量，并减少不必要的操作。

#### 1.2.3.3 表达式执行引擎

表达式解析完成后，便交由表达式执行引擎进行计算。在这一层面，我们才会需要区分矩阵的类型，即稠密矩阵/稀疏矩阵，以及其所使用的模式，包括CPU和GPU两种。

##### 1.2.3.3.1 CPU引擎

Intel® Math Kernel Library 是英特尔公司发行的科学计算库。其提高了软件应用程序在进行大规模数学运算时的效率。英特尔MKL库包含了基础线性代数程序集(BLAS, Basic Linear Algebra Subprograms)、Fortran线性代数包(LAPACK, Linear Algebra PACKage)，快速傅里叶变换，量化数学函数以及随机数生成等接口。在本文所述平台中，主要在CPU模式下运用英特尔MKL库的基础线性代数程序集以及向量数学函数进行矩阵运算。

##### 1.2.3.3.2 GPU引擎

CUDA™ 是 NVIDIA® 公司的并行计算架构。The NVIDIA® CUDA® Toolkit 是其公司发行的基于CUDA架构的开发包，它为开发利用GPU加速的程序的开发者们提供了一个详尽的基于C/C++的开发环境。CUDA Toolkit包括NVIDIA GPU 的编译器，数学函数库以及一些用于调试和优化程序的工具。

在本文所述平台中，主要在GPU模式下运用NVIDIA CUDA Toolkit中的cuBlas库进行稠密矩阵的运算，cuSparse库进行稀疏矩阵的运算以及利用cuRand库作为GPU上的随机数生成器。

### 1.2.4 性能对比

下面将通过两个简单的矩阵算术运算实验，对比IECAMatrix和一些主流的矩阵运算库，分别是：Matlab，OpenCV，Eigen。

实验在我的笔记本上进行。具体的配置如下：Windows 7平台，4G内存，CPU为Intel Core i5-2410M双核，主频2.3GHz每个核。

我们将执行如下的矩阵运算：

A += B \* C

其中A、B、C均为N \* N的单精度浮点数矩阵。上面的运算同时涉及到矩阵的乘法和加法。我们按照[-0.1，0.1]的均匀分布随机初始化矩阵B和矩阵C（当然，在每个库的实验时，我们保证B和C按照同样的随机种子进行随机，以使得每组实验数据完全相同）。

* **小规模矩阵测试**

首先，我们将矩阵的大小N设置为100。我们将上述运算重复执行10000次，以使得计时更为准确。最后，我们统计执行一次运算的平均时间。

下面的表格是具体的实验结果。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 矩阵库 | IECAMatrix | Matlab | OpenCV | Eigen |
| 平均时间/s | 4.95\*10-5 | 1.06 \* 10-4 | 9.34\*10-4 | 7.17\*10-5 |

* **大规模矩阵测试**

在大规模矩阵测试中，我们将N设置为3000。这里的重复次数减少为10。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 矩阵库 | IECAMatrix | Matlab | OpenCV | Eigen |
| 平均时间/s | 0.76 | 1.66 | 27.63 | 0.99 |

从上面的两组实验结果可以看出，无论是在大矩阵运算还是在小矩阵运算，IECAMatrix都有着最高的效率，和IECAMatrix效率最接近的是Eigen矩阵运算库。其余两个矩阵库的差距是数量级上的。

在小规模测试中，IECAMatrix的优势更明显。其一是因为，通过IECAMatrix的表达式优化，可以直接将A + B \* C的结果写入A。而事实上，通过剖析Eigen的源码，其做法是将B \* C存入临时变量D，再执行A = A + D。这样做需要频繁开辟新的空间，这导致时间和内存开销均不是最优。

在大规模测试中，上述问题所带来的影响，相比较巨大规模的矩阵乘法所带来的开销而言，就小很多了。当然，这种情况下，IECAMatrix仍然具有很高的效率。

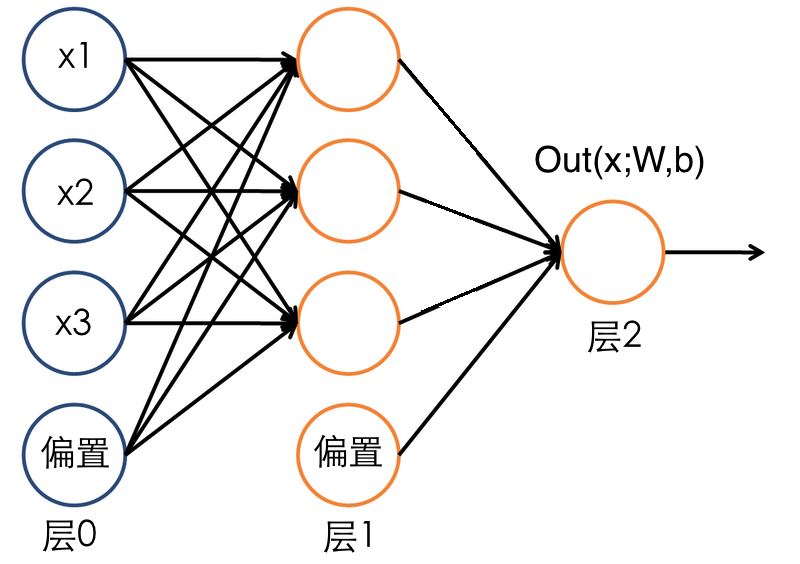
## 1.3 训练算法

在1.2节，我们已经从整体上介绍了平台的搭建，以及其各个部件所具有的功能。在这一章节，我们主要从算法的角度，阐述IECABrain所实现的功能，以及其所具有的特点。

首先，我们介绍训练神经网络的函数优化算法——梯度下降，按照反向传播推导的公式逐层进行更新；接下来将着重介绍IECABrain应对大规模机器学习时所采用的训练方法；之后，还会介绍常见的神经网络的技巧是如何方便地添加到我们的平台中，以及将从卷积神经网络和栈式自动编码器这两种神经网络出发，探讨如何对IECABrain进行扩展。

### 1.3.1 反向传播

我们以一个简单的三层全连接神经网络处理二值分类问题为例，回顾反向传播算法。网络结构如下图所示：



假设现在我们的任务是判断输入图像是不是一个人脸。我们有训练集

，其中x代表输入图像，y代表监督信息，即该图像是不是一个人脸。一共有m个训练样本。神经网络的权值用W来表示，而偏置用b来表示。那么，我们的目标函数可以写成：

其中，LogLoss(p,y)是在分类问题中常用的损失函数，其可以写作：

因此，对于整个训练数据集而言，其训练目标函数为：

我们的目标就是最小化上面的函数。考虑到神经网络模型的参数是W和b，如果采用梯度下降，那么W和b应该按照如下的偏导进行更新：

上面W和b的上标（l）代表其所在的层的编号。

为了求得对应的偏导，采用反向传播算法。反向传播算法的本质就是分部求偏导。首先求得输出层的偏导：

上式中，代表第l层第i个神经元的线性激活值（即前向传播所得到的值）。我们用代表第l层第j个神经元的非线性激活值，那么，权值W和偏置b可以按照如下的公式更新：

为了能够将后一层的误差（偏导）向前传播，以计算前面层的偏导，反向传播按照如下的公式进行：

其中，是第l+1层的神经元个数。而f则是神经网络的非线性激活函数。

综合上面的推导，我们将前向传播和反向传播结合起来写入算法中，并用矩阵运算来代替上面的求和符号，那么，算法可以按照如下的流程进行：

1. 对于当前读入的数据，进行前向传播，计算出除了输入层之外的每一层的非线性激活值。
2. 对于输出层，用当前层的输出和标准答案计算出当前的偏导：

其中，nl为最后一层的编号。“\*”为逐元素乘法，区别于矩阵乘法。

1. 按照反向传播公式，计算从输出层往前一直到第一层的偏导。
2. 计算W和b的偏导。
3. 更新W和b。

上式中，是学习率，用来控制梯度下降对参数更新的速率。

然而，相对于梯度算法，实际应用中，常常采用批量梯度下降，或者是随机梯度下降。所谓批量梯度下降，是每次从数据集中读入m个训练样本，进行上述的前向传播和反向传播，并更新参数，然后再读入下m个训练样本，如此反复直到训练完成。这种方式相对于梯度下降的好处是：

* 适合实际

我们当前讨论的是大规模机器学习，因而，训练数据可以是海量的。这么大规模的数据不可能一次性读入到内存中，所以标准的梯度下降根本无法完成这种工作。

* 收敛更快

这里讨论的收敛，是根据训练数据集迭代的次数来计算的。我们把用完整的数据集训练一次神经网络叫做一次迭代。如果用梯度下降，显然需要很多次对整个数据集的迭代。而用批量梯度下降，由于每次都会更新神经网络的参数，因此，有些时候，将整个数据集迭代一次即可。

* 流式数据在线更新

用批量梯度下降，可以在事先不知道整个数据集的情况下进行。即，训练数据以数据流的方式输入到神经网络训练算法中。这种方式赋予了神经网络两个重要的能力：1）可以训练任意大规模的数据，而不用担心存储空间的问题（当然，要考虑时间问题）；2）可以在线训练，给予神经网络在线学习的能力，使得模型在使用（测试）的过程中也能每时每刻都在更新。

所以，我们平台采用批量梯度下降进行训练。

实际上，批量梯度下降与随机梯度下降和传统的梯度下降之间的区别，只是在于每个批次的数据量不同。如果每个批次数据量为整个数据集的大小，那么就是梯度下降了；如果数据量为1（一个训练样本），那么便是随机梯度下降。所以采用这种方式，也能灵活调整批次大小，适应不同问题。

### 1.3.2 大规模训练方法

在这一节，我们着重介绍IECABrain在应对大规模机器学习问题时候所采用的算法。主体内容将分为三个部分：1）如何同时用多份训练数据训练同一个模型，来加速对数据集的迭代；2）如何将模型分布式存储，以应对大规模神经网络模型；3）如何对分类问题常用的Softmax函数进行近似，从而加速有监督分类问题的训练。

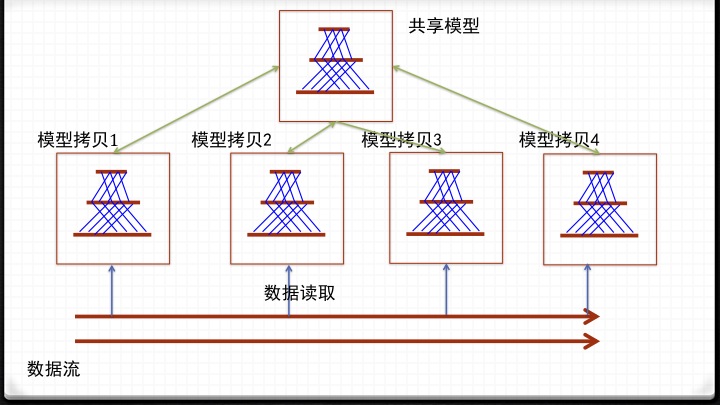
除了神经网络本身的算法，我们还将讨论配套的相关技术。例如，在分布式计算中，我们使用MPI（Message Passing Interface）框架；我们的流式数据读取方式如何在多模型和分布式情况下使用。

#### 1.3.2.1 多模型并发

在上文中，我们介绍了批量梯度算法，用来训练神经网络。由于数据是按照流的形式输入到神经网络中，也就是说，一个批次的数据对应了一次更新。所以一个很自然的想法——我们可以通过并行化这些更新，来减少整个训练时间。

##### 1.3.2.1.1 模型拷贝

在IECABrain中，我们实现了多线程的多模型训练。下图展示了该训练框架。



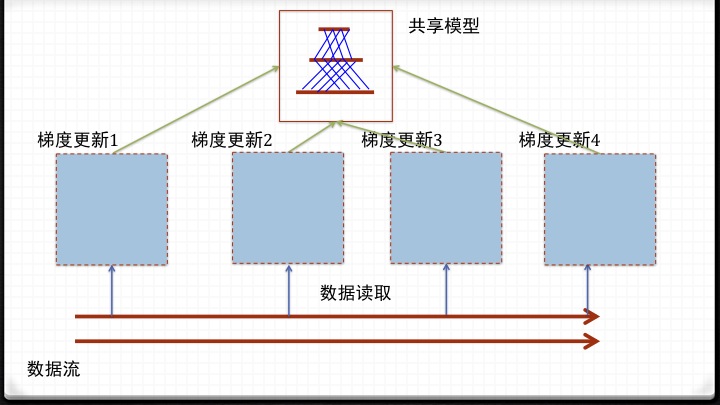
图中，每个模型拷贝都由一个线程（或者是进程）维护。每个模型拷贝都是一个完整的神经网络。这些模型拷贝是从同一个模型复制而来——即共享模型。每个模型拷贝都会从数据流中读入数据，因此，在上图中，整个训练数据集实际上有四个通道流出，平均而言，每个模型拷贝会用到整个训练数据集四分之一的数据来训练自身。

和仅仅用四分之一的训练数据训练一个神经网络模型不同，上面的四个模型拷贝，时刻保持同步。绿色箭头代表了模型拷贝之间会进行同步。因此，将整个训练数据训练完成后，期望的结果是和原本单独一个模型训练的效果等价。

##### 1.3.2.1.2 无锁更新

上述模型拷贝如果存在于不同的进程中，那么这些模型拷贝之间需要相互的进程之间的通信。如果存在于不同的线程，那么也需要相应的通信代价。

有通信就会带来额外的代价，而通信代价过高的时候，甚至效率不如单一模型，所以得不偿失。为了最大限度地提升效率，减少通信代价，IECABrain采用了一种多线程无锁异步更新的方式，如下图所示：



可以看到，与上图相比，主要的变化有：

1. 没有了模型拷贝。由于是多线程训练，所有的线程共享一份模型数据（即共享神经网络模型的权重和偏置）。
2. 仍然是多线程训练，不过每次训练的结果——即对权重和偏置的梯度更新，将会直接更新到共享模型上。

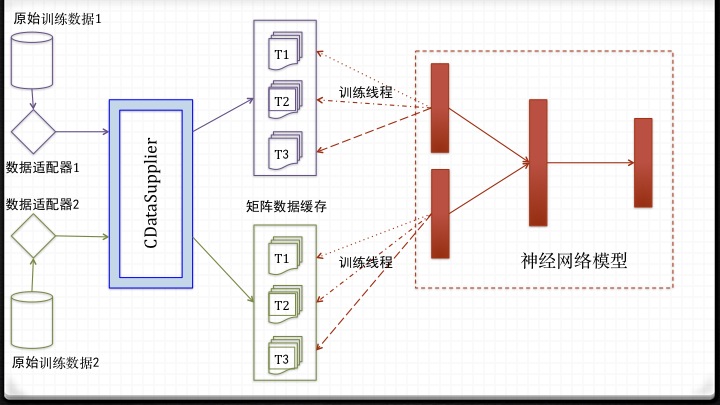
这样的方法主要的好处有：1) 没有冗余的模型拷贝，节省内存；2) 多线程训练的结果将直接反应到全局的模型参数中，没有额外的通信代价。

所以，在程序效率上，这种方法显然是更为优秀的。不过也会有相应的代价。1) 首先，多线程的无锁更新会带来读写冲突。然而，这种冲突会造成影响的情况，也只是在多个线程同时更新共享模型的时候，所以一种方式，可以允许并发读、顺序写，另外一种方式就是直接忽略这个问题——因为写冲突导致的结果，只是会丢失一次更新，这是可以容忍的；2) 其次，每个训练线程在做前向传播和反向传播之间，模型参数就有可能被其他线程所更新，导致前后不一致。然而，这种问题所导致的结果也要视情况而言。大多数问题中，上述问题并不会使得训练结果在精度上有明显的损失。

##### 1.3.2.1.3 流式数据提供

在前文中，我们介绍了IECABrain的流式数据提供方CDataSupplier。这里将阐述如何为多个线程提供同一个数据流，并且在保证多线程读取的情况下，保证数据的顺序和避免数据不一致。

在下面的例子中，假设在多模型训练中，开启三个线程进行训练。另外，由于神经网络只要满足有向无环的条件即可训练，所以，我们也必须支持存在多个输入层的情况（例如多任务学习，协同训练等任务）。



上图描述了在多模型的情况下，流式数据如何提供服务，具体如下：

1. 两个原始训练数据文件，对应了神经网络的两个输入层。因而，需要两个数据适配器从原始文件中读入数据（因为这两个原始数据可以是完全不同的格式，如音频、图像）。
2. CDataSupplier会将两个适配器读入的数据分配到两个不同的缓存仓库。每个缓存仓库对应了一个原始数据文件。同时，每个缓存仓库缓存的数据份数正好是训练线程的数量（多模型训练的模型数量）。另外，这里的缓存已经是处理好的矩阵数据，即可以直接放入神经网络中训练，不需要额外的格式转换。
3. 在训练的过程中，每个线程各自从CDataSupplier的对应缓存仓库读取已经准备好格式的矩阵（数据）。当缓存被清空的时候，CDataSupplier会自动调用数据适配器，装入下一个批次的数据。如此反复，直到所有的数据文件都读入到末尾。

#### 1.3.2.2 分布式训练

在拥有多于一个运算节点（服务器或者PC）的时候，IECABrain能够同时利用多个运算资源，从而进一步提升运算性能，以及能够处理的问题的规模。这一节，我们来关注IECABrain的分布式计算能力。

具体而言，IECABrain的分布式体现在如下两个方面：

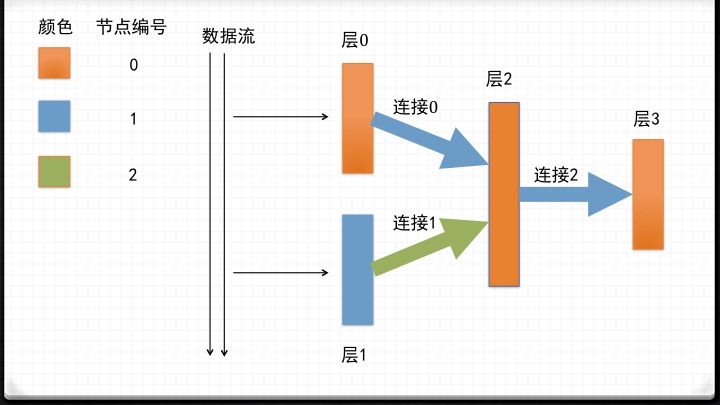
1. **数据分布式**

在拥有多个运算节点的时候，每个节点可以独自用一部分数据训练一个神经网络模型。只需要在合适的时候，将这些节点之间训练的结果进行同步，最终就能够得到与单个运算节点等价的训练结果。

由于流式数据同时被多个运算节点所需求，因此，这大大加快了训练一整个数据集的效率。如果除去运算节点之间的通信代价，其所带来的扩展能力是相当可观的。即，只要有充足的运算节点，便可以让训练效率进一步提升——这便是上一节提到的多模型训练。在下文中，我们便不再赘述这种分布式的做法了。

1. **模型分布式**

所谓模型分布式，就是将一个神经网络模型拆分成若干部分，每个部分存放在一个运算节点上。下图便是一个直观的例子：



图中展示了一个具有4个层，3个连接的神经网络，网络的拓扑深度是3。由于神经网络的主要内存开销主要来自于层中的神经元激活值存储，以及连接中的边和偏置的权值存储，因此，层和连接都可以位于不同的计算节点中。图中一共有四个运算节点，分别用不同颜色来表示。相同颜色的层和连接代表它们存在于同一个运算节点。

以上的模型分布式使得IECABrain具有如下的能力：

* 1. **训练大规模神经网络**

在一些需求中，神经网络的输入层可能非常大（例如，数十亿维度的图像数据或者搜索广告数据）。如果隐层维度也高的话，那么这两层之间的全连接的边数规模就会非常大，导致无法在一台机器上的内存中完整存储（如果用虚拟内存，效率会有数量级的差别，也不可取）。而如果能够将模型进行切分，进而将其存放在不同的运算节点上，那么单个节点的内存压力就会大大降低，同时，我们付出的代价仅仅是多个节点之间的通信开销。在具有InfiniBand连接的服务器集群中，这个开销非常小，与高昂的计算时间相比可以忽略。

* 1. **分担节点计算量，加速神经网络训练**

在上图中可以看出，层0和层1的前向传播和反向传播之间没有依赖关系，因此可以并行计算。又由于层0和层1存在于不同的运算节点中，因此，其使用的计算资源也不冲突。通过并行计算，在分担运算节点的计算量的同时，也加速了神经网络的训练。

下文将着重阐述IECABrain是如何实现模型分布式的。

##### 1.3.2.2.1 模型分布式的实现

在分布式计算的实现中，有两种主流的方式。其一是MapReduce，其二则是MPI。下面对比这两种方式，以阐述我们选择MPI作为分布式计算实现的动机。

* **MapReduce**

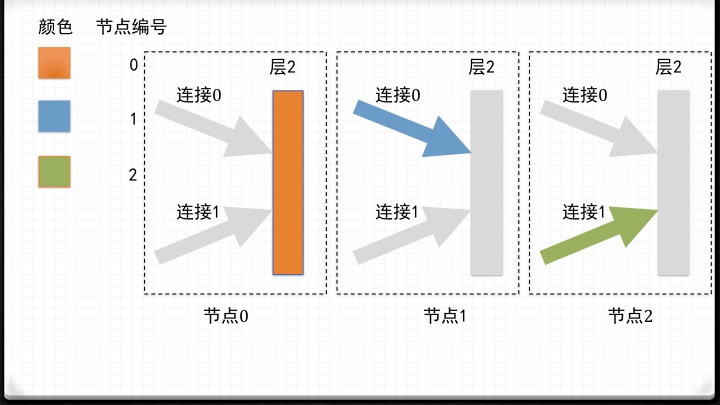
该框架通过map和reduce这两种抽象操作，在计算机集群上实现了分布式的计算。Map操作将运算分布式，reduce操作再将具有相同键值的数据归约到同一计算节点进行归约操作。而MapReduce有两个主要的问题，不适合用于神经网络计算中。一方面，神经网络优化采用的是迭代式的优化算法，而MapReduce并不适合；另外一方面，常见的MapReduce服务将数据存放于硬盘，而对于时效性较高的神经网络训练任务，这种做法效率偏低。

* **MPI**

MPI是个消息传递接口（Message Passing Interface），通过统一的接口，将常用的进程间通信封装。用户不需要关注具体的实现（并且事实上，有多套MPI的实现，各有优劣），只需遵循MPI接口。另外，无论多个进程是存在于同一个节点，还是存在于不同的节点，对于MPI程序没有区别。MPI程序将数据存于内存中，因而效率较高。综合以上的原因，我们采用MPI作为分布式计算的底层通信依赖。

现在，我们以上图中的网络分割方案为例，阐述分布式实现方法。

为了让运算节点之间协同工作，每个运算节点都将会知道完整的网络结构（但是存储只在必须的运算节点上完成）。如下图所示：

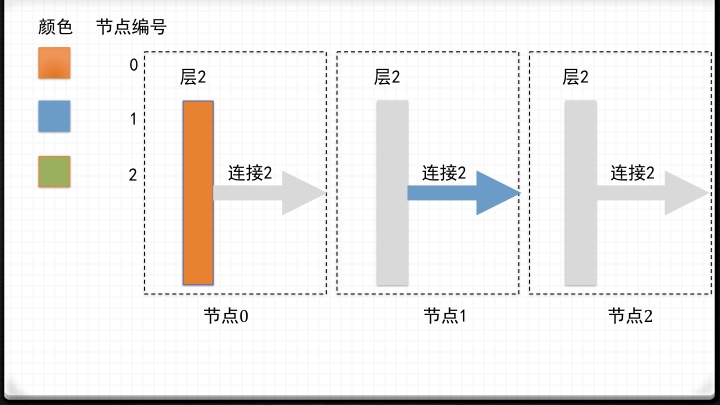


上图展示了进行层2的前向传播时候，各个节点上的存储情况。

由于层2存放于节点0，而前向传播所需要的连接0和连接1都在别的节点（节点1和节点2）上，所以，为了让节点0上具有实际应该有的前向传播的激活值，需要让节点1和节点2各自计算出来的激活值传递给节点0。

我们把上面的过程叫做合并激活值。

当激活值合并以后，层2就可以对其进行非线性变换（例如使用Sigmoid函数，或者Tanh函数）。为了能够将层2的状态继续向前传播，我们需要将层2的状态进行分发，如下图所示：



由于在之后的前向传播过程中，只有连接2需要用到层2的状态，因此，存放于节点0的层2需要将其状态传输给存放连接2的节点1。由于层2和连接2都与节点2无关，所以在这一局部的前向传播过程中，不需要将数据与节点2进行交换。

我们把上面的过程叫做分发激活值。

反向传播的过程与前向传播类似，我们就不再赘述其分布式算法了。

综上，我们将前向传播和反向传播的伪代码总结如下：

前向传播

对于每个传播到当前层的连接：

调用该连接的前向传播

合并每个节点上计算出来的当前层的激活值

计算非线性激活值

将当前层的激活值分发到相关的运算节点

反向传播

对于每个当前层的后继连接：

调用该连接的反向传播

合并每个节点上计算出来的当前层的累计误差

根据当前层的激活函数计算当前层的偏导

将当前层的偏导分发到相关的运算节点

#### 1.3.2.3 softmax近似

在许多分类问题中，常常采用Softmax作为神经网络输出层的激活函数。

其中，K为输出层的神经元个数，zj是第j个神经元的线性激活值。可以看到，这个函数的计算代价与K成正比。当K很大的时候，一方面，该函数的计算会需要一定代价，另外更重要的一点是，最后一层的前向传播和反向传播会非常耗费时间。实际的分类问题确实会碰到K很大的情况。譬如说语言模型的问题中，输出层的大小就是整个字典的大小，可以是数十万，也可以上亿。

在IECABrain中，实现了两种Softmax近似算法。为了对已有的框架进行扩充，我们新构建了SoftmaxApproxConnection和SoftmaxApproxOutputLayer，用来进行Softmax近似。具体可以参考代码部分。以下将对这两种方法的原理进行逐一介绍。

##### 1.3.2.3.1 层次Softmax

层次Softmax将所有的类别标注作为最底层。通过高层到底层逐层分类，最终确定所预测的类别编号。而这种分类方式所需要计算的量是和层次的深度有关，所以，一种很自然的想法，便是将类别标注用树的方式组织。下面展示一种用哈弗曼树实现层次Softmax的方式，这也是在IECABrain中实现的。

具体的步骤如下：

1. 统计每个类别编号出现的频率，以此作为构建哈弗曼树的权重。
2. 将每个类别标记作为哈弗曼树的叶子节点，并记录这些叶子节点的哈弗曼编码。
3. 记录下从根节点到每个叶子节点的路径。

对于用哈弗曼编码实现的层次Softmax函数，令

w代表类别标号对应的哈弗曼编码。而Lw则是编码长度。同时，需要记录w对应的路径，用下面的标记：

那么，这时候的层次Softmax对应的概率解释为：

即从根节点到w节点路径的概率。如果把路径上经过的每个内部节点看做一个二值分类器，那么每个分类器的概率为：

其中是对w节点的线性激活值zw计算Sigmoid函数得到的。上面的经过哈夫曼编码压缩后，只需要次计算即可得到结果。

##### 1.3.2.3.2 负样本采样

负样本采样的基本思想是利用少量的负样本采样以及标记的正样本来计算目标函数以减小计算量。对比Softmax函数，事实上，除了唯一的正例（即我们希望预测的正确样本），其余的标记都是负例。负样本采样的思想非常简单，只是将其余的负例按照一定概率采样一部分进行前向传播和反向传播，而不是用所有的样本。

这样在计算时，可以用以下公式代替：

计算次数降为k+1次(k为所采样的负样本个数)。

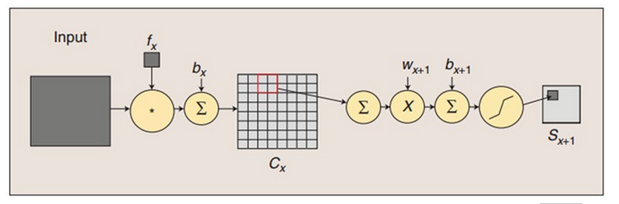
### 1.3.4 扩展功能

IECABrain的一大特点，就是方便在上面进行扩充，从而支持多种神经网络以及多种算法，利于今后的研究工作。下面就展示两种扩充而来的神经网络——卷积神经网络和栈式自动编码器如何在IECABrain平台上实现。

#### 1.3.4.1 卷积神经网络

卷积神经网络（CNN）在计算机视觉等领域有着突出的表现。而计算机视觉中处理的问题规模常常都非常巨大。因此，有必要针对CNN的实现方式花一定篇幅阐述。CNN相关理论工作由于与本文主题无关，就不在这里赘述了。下面主要阐述其在IECABrain上的实现方式。

卷积神经网络最特别的操作有两个：卷积与池化，如下图所示。



因此，在以层-连接为核心的IECABrain中，我们额外实现了ConvConnection和PoolConnection，分别对应了卷积和池化这两种操作。由于池化操作较为简单，我们以卷积连接为例阐述：

* **卷积连接中的前向传播。**

1. 前层神经元的输入：使用I.N×I.C×I.H×I.W长度的向量表示输入图像，I.N表示批处理图像时一次训练的样本数量， I.C表示前层卷积特征图像的个数（对于输入层即输入图像通道数），I.H表示输入特征图像的高度， I.W表示输入特征图像的宽度。
2. 卷积连接的权值矩阵：F.N×F.C×F.S×F.S，F.N表示本层卷积操作所需卷积模板的个数(即生成特征图像个数)， F.C=I.C，表示前层卷积特征图像的个数，F.S×F.S是卷积窗口的大小。
3. 后层神经元的输出：O.N×O.C×O.H×O.W，O.N=I.N，表示批处理图像时一次训练的样本数量，O.C=F.N，表示输出特征图像的个数（也即本层卷积连接的卷积模板个数），O.H×O.W表示输出图像的大小。
4. 依次对每张输入图片进行采样展开(im2col) 将每次采样得到的F.C×F.S×F.S个数值依次填入Col\_data矩阵的一列。采样完毕后，可得到将输入图像按照卷积顺序展开的完整Col\_data矩阵。
5. 将权值矩阵二维化，使其行向量表示一个卷积模板的所有权重，行数表示本层卷积模板的数量。
6. 计算后层神经元的输出，将权值矩阵与展开后的输入图像相乘，得到输出的特征图像。

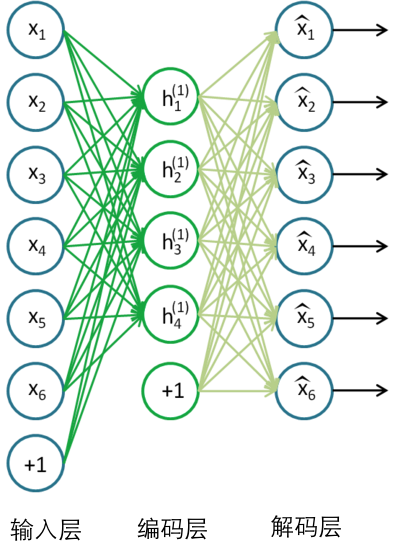
* **卷积连接中的误差反向传播**

1. 误差反向传播时的误差信号维度与输入图像相同。
2. 首先传播误差信号，按照公式，计算前层神经元采样展开后的误差输出。
3. 对于计算所得前层误差输出，需进行反采样展开(col2im)得到前层神经元对应的结果再输出。
4. 利用后层神经元误差信号与前层神经元前向传播时的激活值对权重进行更新。首先对前层神经元激活值进行采样展开，得到展开后矩阵。
5. 根据公式 ，对权值矩阵进行更新。

#### 1.3.4.2 栈式自动编码器

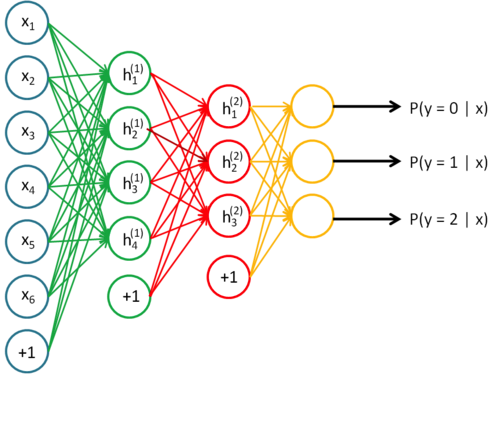
在深度神经网络中，栈式自动编码器（SAE）起到了至关重要的作用——预训练神经网络，从而将深度的神经网络初始化到一个更优的解。因此，为了支持深度神经网络，我们采用SAE进行预训练。在现有的框架中，我们很容易为IECABrain实现一个SAE。

最基本的自动编码器的结构如下图所示：



其中，输出是输入数据的拟合。即通过隐层将输入进行编码。就其本质而言，是个三层的神经网络。因此，自动编码器本身已经可以直接在IECABrain中实现。

至于用来预训练深度神经网络的栈式自动编码器，只是由多个自动编码器叠加而成，高层的自动编码器用低层自动编码器的输出作为输入，又将自身的输出作为更高层的自动编码器的输入。所以，我们需要处理的只是如何将多个三层神经网络的输入输出相衔接——而流式数据提供方CDataSupplier已经具备这个功能，只要将前层的输出神经元的值保存到文件中，用CDataSupplier重定向输入源到该文件，即可为后面层的训练提供输入数据了。下图展示了四层神经网络的栈式自动编码过程：



如上图所示，我们一共需要建立三个三层自动编码器。每个自动编码器负责预训练一个全连接，所以三种颜色的全连接在训练完整个栈式自动编码器之后，就得到了较好的初始解。在之后，只要对整个深度神经网络进行梯度下降训练即可。

## 1.5 平台性能测试

# 用递归神经网络进行搜索广告点击预测

# 神经网络与多义词的向量嵌入