### 计算图说明文档

**一、基本功能**

本大作业中最重要的类是MyGraph，一个计算图对应一个MyGraph对象，它封装了计算图以及构建计算图、按计算图进行计算的相关函数。

提示：

若要使用该功能，请**注释掉main函数中的test\_MINST();**

（一）、计算图的储存

计算图中每个结点都是一个对象，对应的类为Node类的派生类。具体来说，定义了Node类，并派生出Operator\_0, Operator\_1, Operator\_2, Operator\_3，四个派生类，分别对应零、一、二、三元运算符。这四个派生类继续派生出具体运算符对应的类。

指向这些Node的指针都被封装在MyGraph类中。MyGraph中声明了结构体NodeInfo，一个Node对应一个NodeInfo, 封装了指向Node的指针和其是否被访问的记号（用于后续的计算）。MyGraph中的vector<NodeInfo> NodeInfoVec负责存储所有NodeInfo。在程序中，我们主要用NodeInfoVec的下标来表示、访问某个Node。MyGraph中还有std::map <std::string, int> StrToIntMap， 用于获取某个名称的Node的下标。

在类Node中，储存了结点名称（std::string name），节点值（double val）。其中节点值可以用成员函数Val()获取，用rev\_val(float x)修改。

Operator\_n (0<=n<=3)除继承了Node的成员外，还储存了其依赖的节点在NodeInfoVec中的下标。

（二）、计算图的构建

1．Placeholder, Var, Const节点的构建通过MyGraph的成员函数create\_root()完成。

分类讨论，分别调用create\_placeholder, create\_var, create\_const函数。

2. 其余运算符（即输入的第二部分）节点的构建通过MyGraph的成员函数create\_tree()完成。

定义了map，建立函数名与对应create\_...函数的函数指针的映射，从而根据函数名调用函数。其中ScanfMap1为函数符在第一位的函数，ScanfMap2为函数符在第二位的函数。

（三）计算图的运算

通过MyGraph的成员函数graph\_compute()完成

1．每次读入一行。再用stringstream读入字符串。若第一个字符串为“EVAL”进入运算的分支语句。

2. 读入待求节点，赋值个数，赋值具体情况（若后两项没有须特判）。将Placeholder的值记录在MyGraph的成员变量std::map <std::string, float > PlaceholderRev 中，（每次计算前都会清空），便于查看是否赋值、值是多少。

3．调用待求节点的Compt函数，递归计算每一个需要计算的节点的值，更新Node中的val。Compt函数的返回值用于提示错误信息。其中0为正常运行，可读取待求节点此时的val作为答案输出。结果记录在std::vector<float> myresult;中。

Compt函数的实现：递归调用其依赖节点的Compt函数，更新依赖节点的val。再调用自己的Cal函数更新自身的val。每次Compt之后调用MyGraph成员函数Mark(int x)，将NodeInfoVec中该节点的访问信息vis置为1，之后若递归到vis为1的节点不再递归，直接读取该节点的val（每次运算一行前，都要用erase\_mark()把所有vis置为0）。

（四）另外的功能

SETCONSTANT SETANSWER均在graph\_compute()中用分支语句实现。

**二、附加功能1：求导**

graph\_compute()中读取一行后，若第一个字符串为“DERIVATIVE”进入求导的分支语句。

输入格式：DERIVATIVE aim aim2 ParaNum 以及ParaNum个赋值。

输出：aim对aim2的偏导数

1. 与基本要求中的计算一样，递归计算aim依赖的所有节点的值。其中在Compt函数中，运行了成员函数push\_der(x), 在std::vector<int> DerVec中依次记录访问到的节点在NodeInfoVec中的下标
2. 计算aim对DerVec中所有节点的偏导数，记在Node的成员变量der中。将aim节点的der置为1。将DerVec倒置，对其中的每个节点逐一调用Derivate函数，用链式法则依次计算aim对各个节点的导数。Derivate函数的返回值代表错误信息。若为0则代表正常运行。
3. 调用aim2对应节点的Der()函数，输出待求偏导数的值。

**三、附加功能2：神经网络**

该部分基于前两部分的基础，在神经网络的搭建中用了计算图，在梯度下降法中使用了附加的求导功能。

如何使用：

在main函数中注释掉

i\_love\_compute.create\_root();

i\_love\_compute.create\_tree();

i\_love\_compute.graph\_compute();

使用

test\_MINST();

即可进入神经网络的功能（这里自动为训练MINST）。

提示：

若用Windows平台，在Neural\_network::save函数结尾请使用 freopen("CON","w",stdout);

若用Linux平台，请使用

freopen("/dev/tty","w",stdout);

神经网络相关代码主要在Neural\_network.cpp以及Layer文件夹里。test\_minst.cpp中有用MINST测试集测试神经网络的相关函数。

1. 神经网络的存储与访问

该部分最重要的类为Neural\_network类，储存了指向各层神经网络的指针（Layer\*）（在vector<Layer\*>seq中）。

神经网络的每一层都是一个Layer类的派生类的对象。Layer类派生出Input类，对应输入层；派生出Dense类，对应隐藏层和输出层。

每一层都可以看作一个计算图（这里我们实现的是全连接层），有上一层的输出与权值wij的乘法节点、各个乘法节点相加的节点、相加后求sigmoid值的节点。这些节点都储存在MyGraph类对象的NodeInfoVec中（与基本功能的实现类似）。而在每一个Layer的派生类中，我们用vector存了各个节点在NodeInfoVec中的下标，便于查找这些节点。

Neural\_network类及Layer类提供了公有函数int output(int n)，返回该网络（层）第n个输出节点在NodeInfoVec中的下标。

1. 神经网络的构建

Neural\_network提供了公有函数接口add\_Input和add\_Dense，分别用来构建神经网络的输入层即后续的各层。

每层的构建实际上是计算图搭建的过程（与权值相乘、求和、sigmoid）。另外，对于Dense类，在构造的过程中还需提供前一层（函数中用pre\_layer表示）的信息，作为这一层的输入节点。

1. 神经网络的训练

Neural\_network类的公有函数中，train用于输入训练集进行训练，test用于测试。

下面着重介绍train函数：

函数参数有InputData，为装有vector的vector，内层vector是一个测试图的数据，外层vector是这些侧视图的集合。每用一张图训练时，change\_input函数用于将图片的数据赋予输入层，之后调用Compt函数计算。遍历完一次InputData为一个epoch。

注意，为完成求导功能，我们在compt的时候自动把访问过的节点存入了Dervec中，因此，每次Compt结束后都要把DerVec清空，防止内存泄漏。

训练过程中记录了正确的次数。每训练完一张图片后，都会用梯度下降法进行修正。依次计算关于各个wij节点、bias节点的偏导，反向修正，修正量与偏导、学习速率（learn\_rate）成正比。

训练既定epoch（是函数参数，可以在调用traIn函数时调节）后，会自动调用test函数，用测试集进行检测。

1. 一些辅助函数

在Neural\_network.cpp中，有函数save和load，用于向txt文件输出/从txt文件读入训练过程中各个节点的val.

test\_minst.cpp中的函数用于从MINST训练集中读取数据，转化为需要用的形式。这部分代码是从网上摘录的。