|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 单位代码 | **10006** |
|  |  | 学 号 | **13061056** |
|  |  | 分类号 | **TP309** |
|  |  |

****

毕业设计(论文)

基于CUDA的遗传算法和神经网络在股票趋势问题中的应用研究

|  |  |
| --- | --- |
| 学院名称 | 计算机学院 |
| 专业名称 | 计算机科学与技术 |
| 学生姓名 | 王劭阳 |
| 指导教师 | 任健 |

2017年6月

北京航空航天大学

**本科生毕业设计（论文）任务书**

Ⅰ、毕业设计（论文）题目：

基于CUDA的遗传算法和神经网络在股票趋势问题中的应用研究

Ⅱ、毕业设计（论文）使用的原始资料（数据）及设计技术要求：

Ⅲ、毕业设计（论文）工作内容：

Ⅳ、主要参考资料：

计算机 学院（系） 计算机科学与技术 专业类 130616 班

学生 王劭阳

毕业设计（论文）时间： 2016 年 9 月 1 日至 2017 年 5 月 25 日

答辩时间： 年 月 日

成 绩：

指导教师： 任健

兼职教师或答疑教师（并指出所负责部分）：

系（教研室） 主任（签字）：

注：任务书应该附在已完成的毕业设计（论文）的首页。

**本人声明**

我声明，本论文及其研究工作是由本人在导师指导下独立完成的，在完成论文时所利用的一切资料均已在参考文献中列出。

作者：王劭阳

签字：

时间：2017年 6 月

基于CUDA的遗传算法和神经网络在股票趋势问题中的应用研究

|  |  |
| --- | --- |
| 学 生： | 王劭阳 |
| 指导老师： | 任健 |

摘要

摘要正文

关键词：关键词1，关键词2，关键词3，关键词4

Charity Platform Based on Private Blockchain

|  |  |
| --- | --- |
| Author: | Dong Meng Wei |
| Tutor: | Xia Yu Bing |

**Abstract**

Abstract Content

**Key words**: word1，word2，word3，word4

目 录

[1 绪论 1](#_Toc483280243)

[1.1 研究背景 1](#_Toc483280244)

[1.2 国内外研究现状 1](#_Toc483280245)

[1.2.1 国外研究及应用现状 1](#_Toc483280246)

[1.2.2 国内研究及应用现状 1](#_Toc483280247)

[1.3 研究目标与内容 1](#_Toc483280248)

[1.3.1 研究目标 1](#_Toc483280249)

[1.3.2 研究内容 1](#_Toc483280250)

[1.4 论文结构 1](#_Toc483280251)

[2 相关技术 2](#_Toc483280252)

[2.1 元启发式算法和遗传算法 2](#_Toc483280253)

[2.2 人工神经网络 2](#_Toc483280254)

[2.2.1 Theano 2](#_Toc483280255)

[2.2.2 Keras 2](#_Toc483280256)

[2.3 GPGPU并行计算 2](#_Toc483280257)

[2.3.1 CUDA 2](#_Toc483280258)

[2.3.2 PyCUDA 2](#_Toc483280259)

[3 算法设计 3](#_Toc483280260)

[3.1 问题定义 3](#_Toc483280261)

[3.1.1 问题描述 3](#_Toc483280262)

[3.1.2 数据格式 3](#_Toc483280263)

[3.1.3 问题抽象 3](#_Toc483280264)

[3.2 遗传算法 4](#_Toc483280265)

[3.2.1 问题建模 4](#_Toc483280266)

[3.2.2 基因定义 6](#_Toc483280267)

[3.2.3 遗传算法流程 6](#_Toc483280268)

[3.2.4 交叉操作 7](#_Toc483280269)

[3.2.5 变异操作 8](#_Toc483280270)

[3.2.6 适应度计算 9](#_Toc483280271)

[3.2.7 其他优化 9](#_Toc483280272)

[3.3 遗传算法的并行化 9](#_Toc483280273)

[3.4 人工神经网络 10](#_Toc483280274)

[3.4.1 问题建模 10](#_Toc483280275)

[3.4.2 网络结构 10](#_Toc483280276)

[3.4.3 神经元设计 11](#_Toc483280277)

[3.4.4 损失函数 12](#_Toc483280278)

[3.4.5 后向传播 13](#_Toc483280279)

[3.4.6 L2正则化 13](#_Toc483280280)

[3.4.7 Dropout优化 13](#_Toc483280281)

[3.5 人工神经网络的并行化 14](#_Toc483280282)

[4 算法实现 14](#_Toc483280283)

[4.1 开发环境 14](#_Toc483280284)

[4.2 系统架构及实现 14](#_Toc483280285)

[4.2.1 数据解析和预处理模块 14](#_Toc483280286)

[4.2.2 种群模块 15](#_Toc483280287)

[4.2.3 个体模块 15](#_Toc483280288)

[4.2.4 演化模块 15](#_Toc483280289)

[4.2.5 基于CPU的适应度计算模块 16](#_Toc483280290)

[4.2.6 基于GPU的适应度计算模块 16](#_Toc483280291)

[4.2.7 遗传算法训练模块 17](#_Toc483280292)

[4.2.8 人工神经网络训练模块 18](#_Toc483280293)

[4.2.9 测试模块 18](#_Toc483280294)

[5 实验验证 19](#_Toc483280295)

[5.1 验证方法 19](#_Toc483280296)

[5.2 算法评估 19](#_Toc483280297)

[5.2.1 遗传算法的正确率 19](#_Toc483280298)

[5.2.2 人工神经网络的正确率 20](#_Toc483280299)

[5.2.3 遗传算法和人工神经网络正确率的对比 21](#_Toc483280300)

[5.2.4 遗传算法CPU与GPU训练性能的对比 21](#_Toc483280301)

[5.2.5 人工神经网络CPU与GPU训练性能的对比 21](#_Toc483280302)

[结论 22](#_Toc483280303)

[致谢 23](#_Toc483280304)

[参考文献 24](#_Toc483280305)

[附录A 25](#_Toc483280306)

[附录B 26](#_Toc483280307)

# 绪论

## 研究背景

## 国内外研究现状

### 国外研究及应用现状

### 国内研究及应用现状

## 研究目标与内容

### 研究目标

### 研究内容

## 论文结构

# 相关技术

## 元启发式算法和遗传算法

在计算机科学与数学优化领域中，元启发式算法是一种更高层次的

## 人工神经网络

正文

### Theano

正文

### Keras

正文

## GPGPU并行计算

正文

### CUDA

正文

### PyCUDA

正文

# 算法设计

## 问题定义

本节将详细介绍本课题研究的问题、研究的数据集、预处理数据的方式和问题抽象。

### 问题描述

本课题研究的目标是，根据舆情数据来预测股票的涨跌情况。研究的数据集由欧洲一家企业提供。本课题实现的算法将通过一定天数的舆情数据和股票收盘价，训练出模型，之后用户就可以根据此模型和当天的舆情数据，来预测下一天的股票涨跌。

### 数据格式

我们得到的每份数据分为两部分。

一部分是2014~2016年每日该股票的价格（见图3.1），五列分别是日期、开盘价、最高价、最低价、收盘价。

另一部分是2014~2016年每日发生的与该股票相关的舆情（见图3.2）。其中每天的每条舆情数据有四个属性(event\_type\_id, scope\_id, polarity, count)，分别代表：事件类型ID、领域类型ID、极性（积极、中性或消极）、该事件在当天的发生次数。

### 问题抽象

要解决问题首先要对实际问题进行抽象。首先对数据进行预处理，目的是把每条数据处理成从特征映射到下一天股票涨跌情况的格式，即（特征1，特征2，…）->（涨跌情况）。

对于价格数据部分，计算出每日的股票收盘价相对于前一天是上涨还是下跌，作为涨跌情况。

对于舆情数据部分，统计所有数据中出现过的所有不同的三元组(event\_type\_id, scope\_id, polarity)，将每一组不同的三元组编号，给予一个从1开始的正整数ID，称作feature\_id，作为一个舆情特征。再把每天每个舆情特征的发生次数记录下来作为该特征的权重。举个例子，比如说2014年1月1日，某股票对应公司的微博（scope\_id）下出现大量用户的评论（event\_type\_id），评论内容为对该公司某产品投诉（polarity），这一条舆情数据就被我们抽象成一个特征，赋予一个特征ID。这样，把每个特征ID当作特征向量的每个维度，该特征的权重为向量分量的大小，就得到了每日舆情数据的特征向量。

设：m为样本总数（总天数），n为特征总数。

对于每日的数据：

，其xi(j)为第j天feature\_id = i的特征权重，设置x0恒为1。

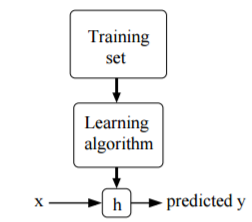
称一组 为一个样本。

对于全体数据：

需要提到的一点是，X和y是从总体Xtotal和ytotal（从古至今每一天的数据）抽样出的一组样本。

现实中存在一个函数f，满足，并且。

如图，我们的目标是找到一个函数h，使得h与f尽量相近，使我们可以通过函数h来预测股票的趋势。



## 遗传算法

本节介绍使用遗传算法来解决问题的算法设计。包括对问题的建模、遗传算法中基因的定义、交叉和变异操作的算法和整个遗传算法的流程。

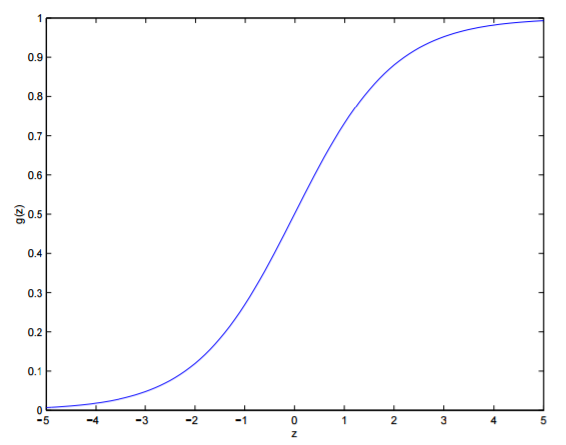
### 问题建模

设参数 为一个n维向量。

我们选定函数h的模型为 。

其中 为sigmoid函数，形状如图所示。当z < 0时，；当z > 0时，。

这个模型把特征向量x根据线性映射为一个实数，再将这个实数通过sigmoid函数映射到(0, 1)区间内。



由于x，y是随机变量，并且，所以我们做出如下假设：在给定x和的条件下，y = 1的概率为。

即：

即满足参数为的伯努利分布。

将两个等式写在一起得到：

如果能求出，那么函数h就求出来了。我们已经有了样本 (X,y)，要根据(X,y)来反求出满足样本分布的参数，我们使用极大似然估计的方法。假设m个样本相互独立，设L为参数的似然度，则根据似然函数的定义，有：

根据极大似然原理，最可能的的值应为使取到最大的值。那么我们把作为遗传算法中的适应度函数，随着算法的进行，算法最终会找出使足够大的，则问题就得到解决了。

### 基因定义

如果要使用遗传算法来解决问题，按照遗传算法的工作流程，需要在要求解的和遗传算法中个体的基因建立双射，也就是确定编码和解码的方式。常用的编码解码方式有很多种，在本课题中，由于，即有n个分量，各分量均为实数，于是很自然的把编码为一个长度为n的实数数组就可以了。这种基因的定义方式有便于大空间搜索，并且结果的精度很高的优点。

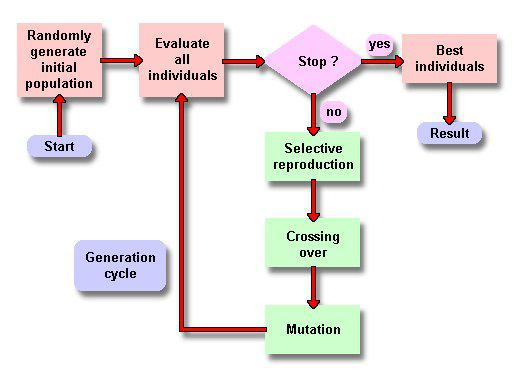
### 遗传算法流程

遗传算法的流程如图。

首先随机生成一个种群，种群中包括随机生成的初始个体，每个个体有自己的基因序列，然后不断迭代产生新的种群，直到达到迭代次数上限或者种群中的某个个体的适应度足够高。个体适应度是基因的函数，也就是上文提到的似然函数。

每次迭代产生新种群的过程包括两个步骤：交叉、变异。

首先进行交叉过程，每次随机选出两个较优秀的原种群个体交叉产生新的个体，直到充满新种群。然后进行变异过程，遍历新种群内的所有个体，按一定概率对其中的基因做突变。



### 交叉操作

遗传算法在繁衍新一代种群时，要不断根据上一代的两个个体（下称A和B）基因交叉产生一个新一代的个体。

下面介绍我们选择A和B让其繁衍的方式。遗传算法的核心思路是优胜劣汰、适者生存，我们要在交叉操作中体现这个启发式的优化。我们在老种群中先完全随机地选出一些个体（数量为tournament\_size，是算法的一个参数），然后在这些个体中取适应度最高的那个作为A去繁衍，再用同样的方式选出B。如下是个体选择的算法代码描述：

def tournament\_selection(pop):

tournament = Population(Algorithm.tournament\_size, False)

for i in xrange(Algorithm.tournament\_size):

tournament.append\_individual(pop.get\_individual(random.randint(0, pop.size() - 1)))

return tournament.get\_fittest()

算法 3.1 个体选择算法

然后我们将A和B的基因交叉，交叉方法是从头到尾遍历产生新个体的每一个基因，这个基因以一定的概率（概率为uniform\_rate，是遗传算法的一个参数）从A中相应位置复制或从B中相应位置复制。如下是交叉的算法代码描述：

def crossover(indiv1, indiv2):

new\_indiv = Individual()

for i in xrange(Individual.default\_genes\_len):

if random.random() < Algorithm.uniform\_rate:

new\_indiv.set\_gene(i, indiv1.get\_gene(i))

else:

new\_indiv.set\_gene(i, indiv2.get\_gene(i))

return new\_indiv

算法 3.2 交叉算法

经过这一步骤，我们就得到未经变异的新一代种群的所有个体了。

### 变异操作

接下来要对新种群中的每个个体进行变异操作。与最常见的二进制编码的遗传算法不同，由于我们的基因使用实数编码，所以变异操作要复杂一些。

对每个个体的变异操作流程如下：

遍历该个体的每个基因，以一定的概率（概率为mutation\_rate，是遗传算法的一个参数）决定这个基因是否需要变异。

如果需要变异，那么新的基因值 ，其中

* 为该位置原有的基因
* 为以50%概率随机产生的1或-1，用于控制新基因向上增长还是向下缩减
* 是一个设置好的实数步长，是遗传算法的参数
* t为当前种群的代次
* T为算法的总迭代数
* b是一个常量，用来控制算法的学习速度，是遗传算法的参数
* r为的随机数。

公式中的因子根据参数b和迭代的进度，控制了每个基因变异的速度，算法开始时变异速度很快，随着迭代次数增加，变异速度会逐渐减小。

### 适应度计算

对于每个新产生的个体，我们都要计算它基因的适应度。根据上文提到的问题抽象和问题建模，每个个体的基因都是不同的参数向量，而该基因的适应度就是似然度。其中的y和X就是我们的样本，也就是训练数据。

### 其他优化

在自然界中，生物繁衍到下一代后，老一代并不会全部死亡，会有一些精英个体由于适应度很高、生命力强盛等原因保留了下来。为了进一步优化算法，我加入了精英保留这个启发式的策略来模拟自然界的这个现象。策略是将上一代作为精英的适应度最高的个体不经交叉、不经变异，直接保留到新一代。这样就避免了由简单的杂交破坏掉较好的基因组合，达到优秀基因组合逐渐累积起来的目的，同时也可以加快最优解的收敛速度。

具体算法是每次产生新一代时，在交叉过程之前，遍历选择上一代适应度最大的个体，复制进下一代的种群，同时将需要交叉产生的个体数目减一，变异操作时也跳过这个个体。

## 遗传算法的并行化

我们希望通过使用GPU大规模并行计算来提高遗传算法的运行效率。由于GPU的核心数量比CPU多得多，并行度就会高很多。我们的遗传算法中对每一个新产生的个体都要计算其适应度，在计算适应度时为了计算，由于，我们要计算m个，而这刚好是的m个分量。那么这个频繁进行的矩阵乘法就成为了我们优化的重点。

在矩阵乘法中，因为乘积的每个分量的计算都是独立的，之间互不相干，如果把乘积的每个分量放在一个单独的核上做计算，各个核没有互斥资源的竞争，不需要锁，而且运算所需要的资源都是一致的，只需要从内存拷贝一个副本到显存上，所以矩阵乘法天生具有容易并行的性质。

是一个的矩阵，是一个的矩阵，乘积是一个的矩阵。所以这样一个矩阵相乘的并行度就是m。现在家用CPU内核数量一般都小于8个，而GPU的内核数量普遍破千，如Nvidia GTX1080Ti的内核数就有3584个之多。如果m远大于8，那么使用GPU并行加速的优势就会很大。

并行化的计算算法如下：

* 算法初始化
  + 在显存中分配三块空间，一块用来存储X，一块用来存储，一块用来存储乘积，并记录下三块空间的地址
  + 将矩阵X从内存拷贝到显存中
* 每次计算
  + 将向量从内存拷贝到显存中
  + 根据m的大小设置线程块数量和线程数量
  + 启动所有的计算线程，并行计算乘积的每个分量，结果存储在预先分配好的空间中
  + 将乘积从显存拷贝到内存中

由于每次计算时矩阵X都是不变的，所以每次计算都只需要把从内存拷贝到显存，X只需要使用最初拷贝的那份就可以，这样可以节省大量的IO时间。

## 人工神经网络

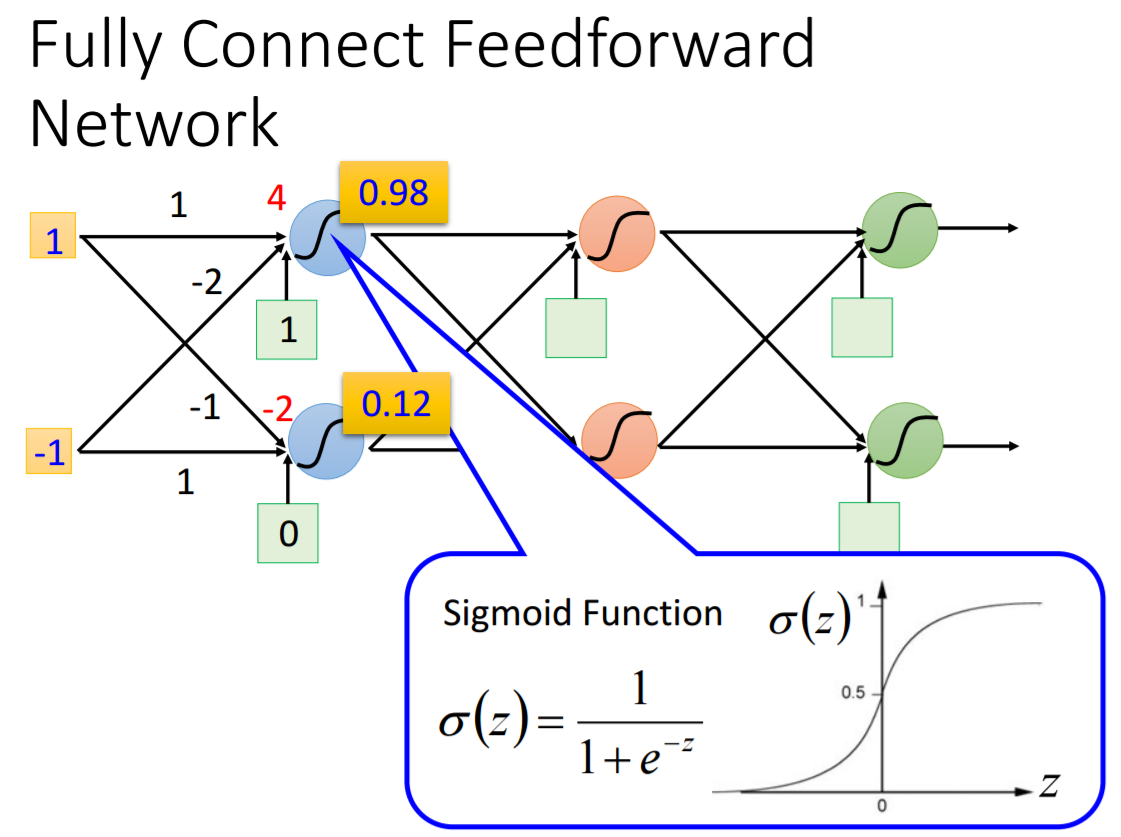
本节介绍使用人工神经网络来解决问题的算法设计。包括对问题的建模、神经网络结构的设计、神经元的设计、损失函数的设计、训练方法和其他优化。

### 问题建模

在3.1.3小节中我们讨论了问题的抽象：现实中存在一个函数f，满足，并且。我们的目标是从假设空间H中找到一个函数h∈H，使得h与f尽量相近。我们将建立一个人工神经网络，并使用样本集训练出网络的各个参数，该网络就是函数h，将特征向量x映射到预测值h(x)。

### 网络结构

算法采用全连接的人工神经网络。如图，网络包括一个输入层，多个隐层（隐层数量为算法的参数）和一个输出层。输入层包含n个神经元，每个神经元输出样本的一个分量。每个隐层包含一定量的神经元（单隐层神经元个数为算法的参数）。输出层包含1个神经元，输出对当前输入数据的预测结果。各层之间的神经元全连接。

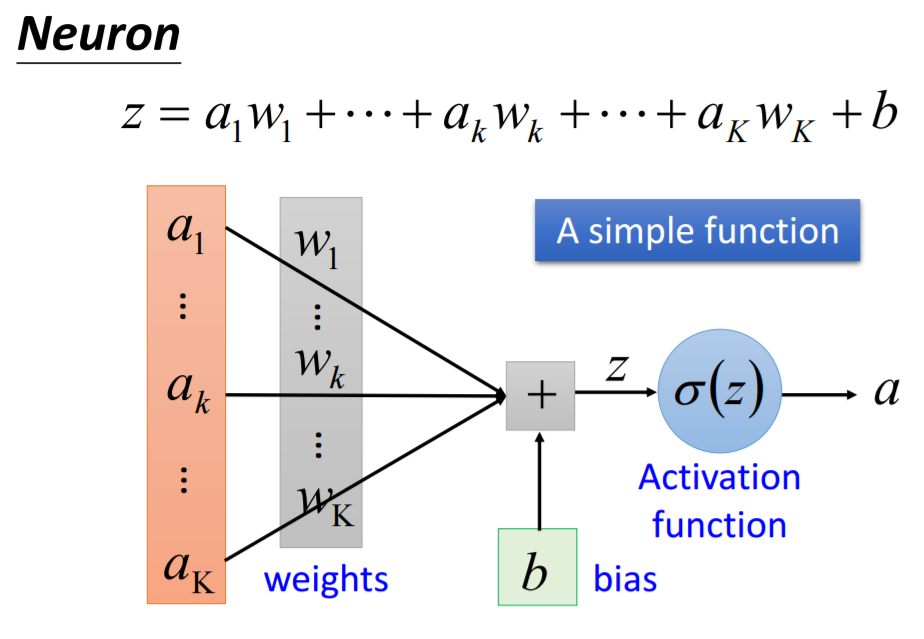


### 神经元设计

在每个神经元内设置一个参数。如图，神经元的输入是上一层的所有输出，设为向量，然后把和相乘得到一个实数，再把这个数做一个sigmoid函数映射，映射到(0, 1)区间内，将这个值输出。

可以看出，我们人工神经网络算法中的每个神经元的设计与3.2.1小节讨论的遗传算法问题模型十分相似，上文讨论的遗传算法模型实际上可以看做是一个无隐层、只有输入层和输出层的人工神经网络。

对于每一层而言，设和分别是该层的输入输出向量，是该层的参数矩阵（将每个神经元的参数拼在一起），是矩阵为的线性变换，则。可以看出神经网络的每一层对输入向量先进行一个线性变换（升降维、放大缩小、旋转），然后平移，然后再进行一个非线性变化（弯曲），将输入空间投向另一个空间。多层神经网络组合，可以产生十分复杂的函数模型，使假设空间H很大。



### 损失函数

我们的神经网络的训练目的是最小化每个神经元的损失函数。在3.2.1小节中，我们希望最大化似然函数。同样的道理，在这里对于输出神经元，我们希望最大化似然函数。将取对数后得到

将损失函数设置为对数似然函数的相反数：。那么随着算法的进行，各神经元的损失函数值会越来越小，最终就会得到我们想要的函数h。

在信息论中，常用交叉熵来衡量两个概率分布p、q之间的相似性。交叉熵越小，说明p和q越相似。从这个角度来考虑的话，根据定义，样本分布y和预测分布y’的交叉熵

显然，我们的损失函数刚好正比于y和y’的交叉熵。

### 后向传播

我们使用后向传播的方法来训练出每个神经元的参数。

首先初始化所有为随机值。每次迭代中，首先算出输出层的损失函数值，然后更新该神经元的参数，其中为设置好的参数，用于控制学习速率，为loss对于的梯度。

对于隐层，我们定义函数

其中k为当前层数，为k+1层与k层该神经元连接的那个分量。接着从后向前迭代每一层，按照更新每个神经元的参数。直到达到最大的迭代次数。

### L2正则化

由于人工神经网络需要大量的数据来训练，而我们的训练数据不够多，比较容易导致过拟合。为了防止这一现象，我们在损失函数上添加了L2正则项：

其中为设置好的控制正则化力度的参数。可以看出，L2正则项其实是一个惩罚项，的各分量越大，惩罚越严重。这样可以使模型变得更简单，而简单的模型一般更有范化的能力。

算法思路很朴素：通过最小化损失函数来拟合训练集，又通过加入正则项来防止过分拟合训练集。

### Dropout优化

为了进一步的防止过拟合，算法使用了dropout的优化策略，也就是说，在算法的每次迭代过程中，都暂时丢弃一部分（比例为算法的参数dropout\_rate）随机的神经元。也就相当于我们训练出了很多个部分结构的神经网络，每一个部分神经网络都可以给出一个分类结果，这些结果有的是正确的，有的是错误的。随着训练的进行，大部分部分神经网络都可以给出正确的分类结果，那么少数的错误分类结果就不会对最终结果造成大的影响，同时又因为每次迭代减少了部分神经元，也就是减少了参数的维度，而减少了算法过拟合的风险。

## 人工神经网络的并行化

与遗传算法一样，人工神经网络中也需要进行大量的矩阵运算，我们使用theano框架提供的功能，将数据转为32精度的浮点数，然后把矩阵运算等操作挪到GPU中做，来优化算法的执行效率。

# 算法实现

## 开发环境

编程语言：Python 2.7

操作系统：Windows 10 64bit

包管理工具：conda、pip

IDE：Spyder 3

CPU：Intel Core i5 6500

GPU：NVIDIA GeForce GTX 1060

依赖库：NVIDIA CUDA Toolkit 8.0、NumPy 1.11.3、Keras 1.0.7、Theano 0.9.0、PyCuda

## 系统架构及实现

实验系统整体的架构如图所示，我们依照之前的设计，把需要实现的程序分为了三个文件：ga.py用于实验基于CPU的遗传算法的效果和性能；ga\_cuda.py用于实验基于GPU的遗传算法的效果和性能；dl.py用于实验人工神经网络算法的效果和性能。

我们使用NumPy库来支持矩阵和向量的计算；在遗传算法部分使用PyCuda来编译c++的CUDA并行计算代码并插入到我们的Python程序中；在人工神经网络部分利用Keras和Theano框架方便地建立神经网络、训练并验证数据集，也可以通过改变配置，使程序方便地切换到GPU上运行。

### 数据解析和预处理模块

该模块对数据进行解析和预处理。本课题研究的数据集是以csv格式提供的，所以要使用python内部的csv库来解析数据文件。首先解析价格数据，逐行读取文件，从第二行开始，读取该行的日期和收盘价，然后判断该日期相较于前一天价格是上涨还是下跌来得出该日趋势，再以日期为键、趋势为值存入字典trends里面。接着解析舆情数据，逐行读取文件，对每一行读取该行的日期、类型ID、领域ID、极性和次数，把类型ID、领域ID、极性组成三元组，若还未给此三元组赋予特征ID就生成一个特征ID并把关联信息存入一个字典中，然后在二维数组sentiments中保存该日该特征的权重为次数。

数据解析完之后开始建立矩阵和向量。初始化矩阵X和向量y为空的numpy对象，然后按日期对trends和sentiments排序，对每个日期把趋势和价格写入X和y中。

### 种群模块

该模块把种群概念抽象为类class Population。支持如下方法：

* \_\_init\_\_：用于建立新种群，如果need\_init参数为真，则不断添加随机生成的个体来填满这个种群。
* get\_individual：按序号获取种群中的某个个体。
* get\_individuals：获取种群中的所有个体。
* append\_individual：在种群的末尾添加个体。
* size：获取种群的大小。
* get\_fittest：获取种群中适应度最高的个体。

### 个体模块

该模块把个体概念抽象为类class Individual。支持如下方法：

* \_\_init\_\_：用于初始化新个体，生成空的基因序列，将适应度置为0。
* generate\_individual：使用均一分布的±10000间的随机值填充个体的基因序列。
* set\_default\_genes\_len：静态方法，用于设置默认的基因长度。
* get\_gene：获取特定位置的单个基因。
* set\_gene：设置特定位置的单个基因。
* get\_genes：获取整条基因。
* get\_fitness：获取个体的适应度。首先检查该个体的适应度之前有没有计算过，没有的话进行计算并把结果保存下来。

### 演化模块

该模块class Alogorithm实现了各种演化过程需要的所有算法，包括交叉算法、变异算法等。支持如下静态方法：

* evolve\_population：根据传入的种群，进行一次完整的繁衍过程。先不断的调用tournament\_selection来选出两个个体进行交叉产生新个体填入新种群中，然后遍历新种群的每个个体进行变异操作，返回新产生的种群。
* tournament\_selection：用于选出要交叉的个体。传入种群，随机选出该种群中一定量的个体，返回这些个体中适应度最高的个体。
* crossover：用于交叉两个个体。首先初始化一个新个体，然后遍历生成新个体的基因序列，每次生成一个[0,1)的随机数，判断其是否小于设置好的交叉率，如果小于则该基因从个体1复制，否则从个体2复制。
* non\_uniform\_degree：用于计算突变的程度。
* mutate：用于让个体进行变异。遍历传入的个体基因序列，每次生成一个[0,1)的随机数，判断其是否小于设置好的突变率，如果小于则计算突变的程度并对该单个基因进行变异。

### 基于CPU的适应度计算模块

该模块class FitnessCalc实现了基于CPU的适应度计算的算法。支持如下静态方法：

* h：用于计算。输入的x既可以是全体样本的矩阵，也可以是单个样本的向量。
* l：用于计算似然函数。
* get\_fitness：用于计算个体的适应度。输入一个个体，调用方法l来计算似然度作为该个体的适应度。

### 基于GPU的适应度计算模块

该模块class FitnessCalc实现了基于GPU的适应度计算的算法。

如算法4.1，首先该模块用c++实现了在单线程上计算矩阵乘法中的一个元素并做sigmoid映射的算法：先根据当前线程块号和线程号计算出要计算的元素在乘积矩阵中的偏移，然后读取在显存中的乘数矩阵和乘数向量，计算相应位置的元素值，顺便做sigmoid映射，之后填在显存相应的位置中。

算法 4.1 在GPU单核上运行的算法

\_\_global\_\_ void h(const int \*X, const float \*theta, float \*rst, int n, int m) {

int i = threadIdx.x + blockIdx.x \* 1024;

if (i >= m) {

return;

}

int j = 0;

float t = 0;

for (j = 0; j < n; j++) {

t += X[i \* n + j] \* theta[j];

}

rst[i] = 1 / (1 + exp(-t / n \* 0.0001));

}

在模块初始化时，会调用PyCuda来把这个c++程序编译并作为该模块的python函数h加载到算法进程中。模块还支持如下静态方法：

* l：用于计算似然函数。与基于CPU的适应度计算模块中的方法l不同，这里先判断是否已经在显存中分配好了空间，若没有则先分配空间并把指针保存起来，然后如果显存中不存在矩阵X则把X复制到显存相应空间中。再复制到显存相应空间中，建立大量GPU线程运行函数h计算出结果，再把结果从显存拷贝回内存中。
* get\_fitness：用于计算个体的适应度。输入一个个体，调用方法l来计算似然度作为该个体的适应度。

### 遗传算法训练模块

该模块调用种群、个体、演化、适应度计算等模块，串联出整个遗传算法的流程，在样本子集上做训练，最终得到训练出的参数。同时此模块使用计时器记录了训练过程所使用的时间。

模块首先设置好总迭代数、种群大小、基因长度等参数，然后把训练集交付给适应度计算模块，再使用种群模块初始化生成一个随机的种群，不断迭代。迭代时记录每次繁衍新种群需要的时间。最终记录下最末代种群中最优个体的基因序列作为返回值。

### 人工神经网络训练模块

该模块建立并设置整个人工神经网络，并使用训练集将网络训练完成。

此模块利用Keras和Theano框架实现。首先实例化一个Keras的Sequential对象，然后构建网络，添加输入层、隐层和输出层，添加dropout层，设置神经元使用sigmoid作为激活函数，使用L2正则化的方法，使用后向传播的训练方法，使用交叉熵作为损失函数，使用二分类来评估结果。然后调用Keras模型的fit方法，输入训练集，设置迭代次数，自动训练网络。训练时使用计时器计时，保存所用的时间。

Keras底层采用Theano框架实现，要使用GPU优化性能只需要在配置文件中做如配置4.1的设置。

配置 4.1 theanorc文件

[global]

device=cuda

floatX=float32

allow\_gc=False

mode=FAST\_RUN

compute\_test\_value = ignore

exception\_verbosity = high

### 测试模块

该模块用于测试算法产生的模型的正确率。输入测试集和训练好的模型（对于遗传算法是，对于人工神经网络是网络对象），使用模型对测试集的每条数据做预测，再与正确结果作比较，得到预测正确的天数，最后输出正确率（）。

# 实验验证

## 验证方法

为了保证结果的可靠性，本课题采用了两个数据集进行实验验证：

1. 2014年8月11日至2016年8月9日间德国DAX股票指数（下称DAX）的价格和相关的舆情数据，共包含505天的价格和51654条舆情。DAX（德语： Deutscher Aktienindex）是德国重要的股票指数，是由德意志交易所集团（Deutsche Börse Group）推出的一个蓝筹股指数。该指数中包含有30家主要的德国公司。DAX指数是欧洲的重要证券指数，也是世界证券市场中的重要指数之一。
2. 2014年6月2日至2016年6月2日间国际黄金价格（下称GLD）和相关的舆情数据，共包含518天的价格和38770条舆情。

对于每个数据集，按照3:1:1的比例划分为训练集、验证集和测试集。使用训练集进行模型训练，使用验证集对算法进行调参，使用测试集进行正确率的测试。单独划分验证集是为了确保测试的可靠性，防止为了拟合测试集而调参。

在对比基于CPU的（无CUDA优化）遗传算法训练性能和基于GPU的（有CUDA优化）遗传算法训练性能，以及基于CPU的（无CUDA优化）人工神经网络训练性能和基于GPU的（有CUDA优化）人工神经网络训练性能时，采用DAX数据集来分析。

## 算法评估

### 遗传算法的正确率

按表格 5.1设置可调参数，分别使用两个数据集训练模型并测试。

表格 5.1 遗传算法参数设置

|  |  |
| --- | --- |
| 可调参数 | 值 |
| 总迭代数（产生种群数） | 1000 |
| 种群内个体数 | 100 |
| 交叉率uniform\_rate | 0.85 |
| 变异率mutation\_rate | 0.05 |
| 个体选择数tournament\_size | 30 |
| 变异速度b | 3 |
| 变异步长d | 10000 |

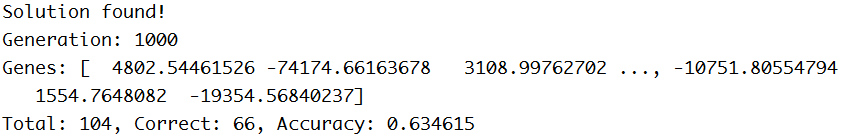
如图 5.1，算法在DAX数据集上测试了104个数据点，其中预测正确66个，正确率为63.4615%。

图 5.1 遗传算法DAX测试结果

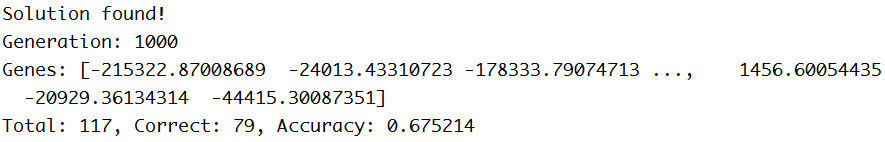
如图 5.2，算法在GLD数据集上测试了117个数据点，其中预测正确79个，正确率为67.5214%。

图 5.2 遗传算法GLD测试结果

### 人工神经网络的正确率

按表格 5.2设置可调参数，分别使用两个数据集训练模型并测试。

表格 5.2 人工神经网络参数设置

|  |  |
| --- | --- |
| 可调参数 | 值 |
| 总迭代数 | 10000 |
|  | 0.01 |
| 隐层数 | 1 |
| 隐层神经元数 | 8 |

C:\Users\SyW\AppData\Local\Temp\1495485770(1).png如图 5.3，算法在DAX数据集上测试了104个数据点，其中预测正确59个，正确率为56.7308%。

图 5.3 人工神经网络DAX测试结果

如图 5.4，算法在GLD数据集上测试了117个数据点，其中预测正确77个，正确率为65.8120%。

### C:\Users\SyW\AppData\Local\Temp\1495486051(1).png遗传算法和人工神经网络正确率的对比

图 5.4 人工神经网络GLD测试结果

两种算法在两个测试集上的正确率对比见表格 5.3。

表格 5.3 算法正确率

|  |  |  |
| --- | --- | --- |
|  | DAX | GLD |
| 遗传算法 | 63.4615% | 67.5214% |
| 人工神经网络 | 56.7308% | 65.8120% |

可以看出，遗传算法的表现要比人工神经网络更加优秀。

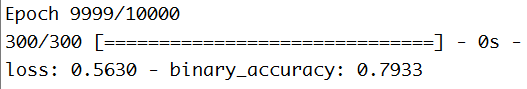
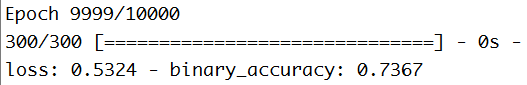
不难理解，从直觉上考虑，舆情和股票价格应当是线性相关的。比如说在微博上对某公司的负面评价越多，该公司的股票价格越可能下跌，并且评价数和下跌概率直觉上应该呈线性。遗传算法的问题模型就使用了一个简单的线性模型，sigmoid函数只是把结果映射到(0,1)区间内，不影响模型的线性。而人工神经网络的问题模型很复杂，上文提到多层多神经元的网络可以表示相当复杂的函数，而复杂的函数模型就容易过分拟合训练集。如图 5.5和图 5.6，使用人工神经网络训练时，在最后的迭代中，对训练集的预测正确率高达79.33%和73.67%，然而在测试集上正确率却只有56.7308%和65.8120%，说明此模型的确过拟合了。

图 5.5 人工神经网络对GLD训练集的预测正确率

图 5.6 人工神经网络对DAX训练集的预测正确率

### 遗传算法CPU与GPU训练性能的对比

DAX训练集共包括300条数据。分别使用基于CPU和基于GPU的遗传算法训练该训练集，并记录花费的时间。结果如图 5.7和图 5.8，基于CPU的算法花费了86.9秒，基于GPU的算法花费了73.4秒C:\Users\SyW\AppData\Local\Temp\1495489704(1).png。通过使用CUDA的优化，在训练集大小为300时，遗C:\Users\SyW\AppData\Local\Temp\1495491335(1).png传算法的训练性能提高了18.4%。

图 5.7 CPU遗传算法训练DAX数据的耗时

图 5.8 GPU遗传算法训练DAX数据的耗时

### 人工神经网络CPU与GPU训练性能的对比

C:\Users\SyW\AppData\Local\Temp\1495492319(1).pngC:\Users\SyW\AppData\Local\Temp\1495492007(1).png分别使用基于CPU和基于GPU的人工神经网络训练DAX训练集，并记录花费的时间。结果如和，基于CPU的算法花费了6.8秒，基于GPU的算法花费了13.4秒。我们意外地发现使用GPU比使用CPU还多花了接近一倍的时间。

图 5.9 CPU神经网络训练DAX数据的耗时

图 5.10 GPU神经网络训练DAX数据的耗时

在使用CUDA优化算法性能时，theano框架把矩阵运算等计算挪到了GPU上做，以矩阵乘法为例，乘法的并行度是min{GPU核心数，m\*1}。实验环境中虽然GPU的核心数是1280个，而训练集却只有300条数据，乘法的并行度就只有300。由于训练集小，并行度不够大，并行计算节约的时间还没有内存显存间拷贝数据带来的额外开销多，所以造成使用GPU比CPU还要慢的情况。

我们使用复制的手段扩大训练集，发现当训练集大小为1700时，基于CPU和GPU的算法耗时十分接近，均为23秒左右。当训练集大小为2400时，基于CPU的算法训练耗时27.4秒，基于GPU的算法训练耗时25.7秒。

# 结论

相较于人工神经网络，遗传算法更好地解决了通过舆情数据来预测股票涨跌的问题。对于遗传算法，我们成功地使用CUDA优化了它的训练性能，提高了训练速度；对于人工神经网络，我们了解了在本课题的实验环境下，如果扩大训练集到1700条数据以上，使用CUDA优化的算法就会比只使用CPU的算法性能高、速度快。

# 致谢

内容

# 参考文献

[1]

[2]

[3]

[4]

[5]

[6]

[7].

[8]

[9]

[10]

[11]

[12]

[13]

[14]

[15]

[16]

[17]

[18]

[19]

[20]

[21]

# 附录A

# 附录B