目　录

遗传算法及神经网络在游戏开发中的应用

摘　要：在游戏开发中，游戏AI（人工智能）的编程是相当复杂的部分，程序员必须保证游戏AI能够满足玩家需求。现在国内外很多的游戏开发公司都运用遗传算法和神经网络对游戏AI进行编程。通过此种方法编出的游戏AI将变得更加灵活，并具有较强的智能性。相对于以前游戏AI编程中的大量条件判断来说，遗传算法和神经网络使得游戏AI的灵活性远高于前者。

关键词：遗传算法；神经网络；人工智能

Application of Genetic Algorithm and Neural Network in game programming

Abstract: Artificial intelligence is the most complex part in game programming . Programer must insure the AI meet players needs. Nowadays , many foreign companys uses genetic algorithm and neural network to encode the game AI . It is more superiority than lots of if-else encoding by GA and NN . Make the game AI become bright and has a strong intelligence. It’s more intelligent than former .

Key word: Genetic Algorithm; Neural Network;　Artificial Intelligence

1 绪论

1.1 什么是遗传算法

遗传算法是模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法[1]。它能解决的问题很多，譬如，数学方程的最大最小值，背包问题、装箱问题等等。在游戏开发中遗传算法的应用也十分频繁，不少的游戏 AI 都利用遗传算法进行编码。

1.2 遗传算法的原理

遗传算法是模拟了生物进化而演变出来的一种计算模型。生物在生存过程中需要生长、生殖、以及死亡几个阶段，遗传算法正式模仿了生长和生殖阶段，生殖阶段保证了生物体能够不断的延续下去。而生长阶段中生物体在自然环境不断变化的前提下，会因为环境的因素而产生一些突变。突变是无向的，它让一些生物体能够更好的适应环境，另一些生物体却因为不能够适应环境而死亡。突变同样也发生在生殖阶段。

生长：生物体从一个受精卵最终成长为一个成年个体得通过环境的考验——天敌、食物、自然环境、天气等因素都可能成为个体生存的威胁。强壮的个体能够逃脱天敌的追捕以及其他因素的束缚，得到了生存生殖的权利。因此它的优良基因能够传承给后代。

生殖：生物体进行交配长生后代，父代和母代各提供一半的染色体进行结合，最终子代就拥有父母代的基因。子代会继承父母代的一些性状。比如，长相、身材等等。一般来说，通过遗传产生的新个体对环境往往具有更好的适应性。

死亡：生物体最终的归宿。代表了本生物个体的死亡，该生物个体已经被淘汰。

用遗传算法解决问题时，每一个染色体代表了一个解决问题的方案，该方案相当于我们生物学上讲的某一个生物体，它可能适应环境，也可能不适应环境。遗传算法解决问题就是把能够适应环境的个体找出来，遗传算法需要找出种群中适应度较高的个体进行生殖产生后代，当后代达到原种群数量时进行适应度测试，当子代种群中某个个体能够达到要求，遗传算法就找到了一个问题的解决方案：否则，子代种群继续上一个步骤：生殖。如此循环直到程序找出达到要求的个体。见图 1.1。

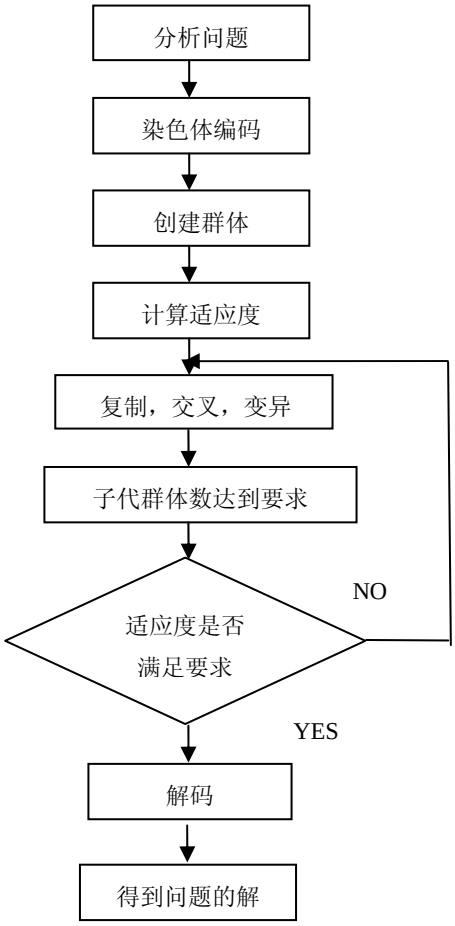

图 1.1 遗传算法流程

1.3 什么是神经网络

从生物学的角度来讲，神经网络是一个有很多个神经细胞组成的错综复杂的神经网。神经细胞是由细胞体、细胞核、树突、轴突和轴突末梢组成，也叫作神经元。神经元的树突连接着其他的神经元，信号通过树突传递给下一个神经元的轴突末梢，轴突末梢处理信号之后，通过轴突传递给了这个神经元。神经元之间的信息传递是单向的，即由一个神经元的树突传递给另一个神经元的轴突末梢。生物神经细胞如图 1.2 所示[2]：

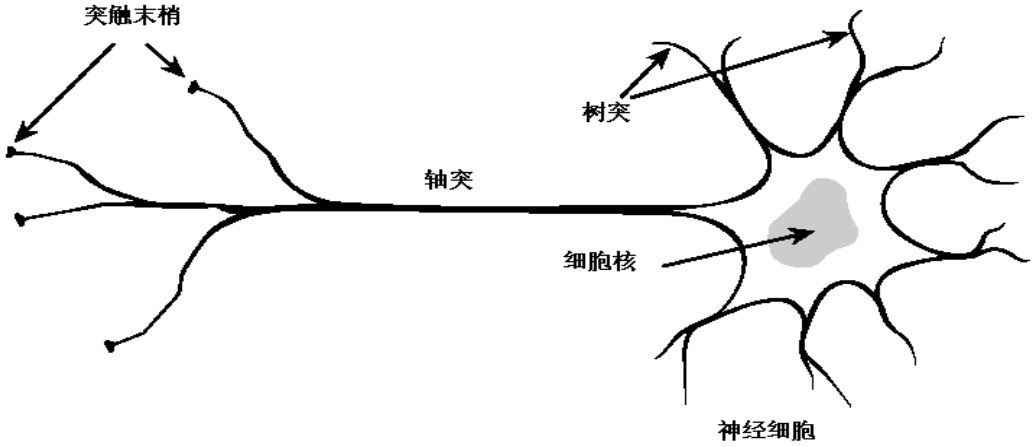

图 1.2 生物神经细胞

人类通过多年来的研究，仿照生物神经网络的模型，设计出了人工神经网络。人工神经网络是一种模仿动物神经网络行为特征，进行分布式并行信息处理的算法数学模型。这种网络依靠神经元权重之间的联系，权值的调整，从而达到处理信息的目的。人工神经网络具有自学习和自适应的能力[3]，可以通过预先提供的一批相互对应的输入和输出数据，分析掌握两者之间潜在的规律，最终根据这些规律，用新的输入数据来推算输出结果。各个神经元的权值恰到好处的表达了一个规律，就像一个物理公式一样。

1.4 人工神经网络原理

当我们看见一个图片的时候，我们可以很准确的告诉他人，这个图片描述的是什么。当然，在看这个图片之前，我们应该对该事物已经有所了解。人工神经网络其实也是如此，我们给出一个图片，如图 1.3 所示，我们能够清晰的看到这幅图片上显示的是一个阿拉伯数字“0”。但是计算机却不能跟我们人一样瞬间就识别这个数字。这是个 6*10 像素的图片，计算机接收的只是每一个像素的信息，这些像素就相当于输入。假设被着上黑色的代表 1，白色的为 0，那么当所有的 60 个信息输入人工神经网络的神经元，经过计算输出的结果是 1，就说明计算机认出了这个图片显示的是“0”。反之，计算机就没能认出这个图片是“0”。人工神经网络就是要通过调整神经元上的权值，让计算机能够顺利地识别出这张图片。

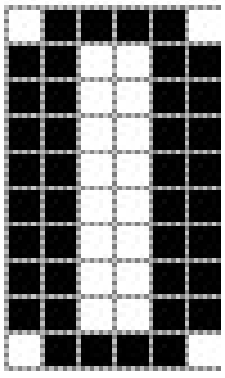

图 1.3 数字位图

神经网络会根据输入的值进行运算判断，如果网络作出错误的的判决，则通过网络的学习（遗传算法等进行权值优化），应使得网络减少下次犯同样错误的可能性。首先，给网络的各连接权值赋予(0，1)区间内的随机值，将“0”所对应的图象模式输入给网络，网络将输入模式加权求和、与门限比较、再进行非线性运算，得到网络的输出[4]。在此情况下，网络输出为“1”和“0”的概率各为 50%，完全是随机产生。当输出为“1”（正确结果）时，说明权值之间的关系是合理的。当输出为“0”（错误结果）时，进化算法[1]将改变各个权值，达到减小犯同样错误的可能性。一般说来，网络中所含的神经元个数越多，则它能记忆、识别的模式也就越多。以下是使用进化算法对神经元权值进行优化的流程，如图 1.4 所示。

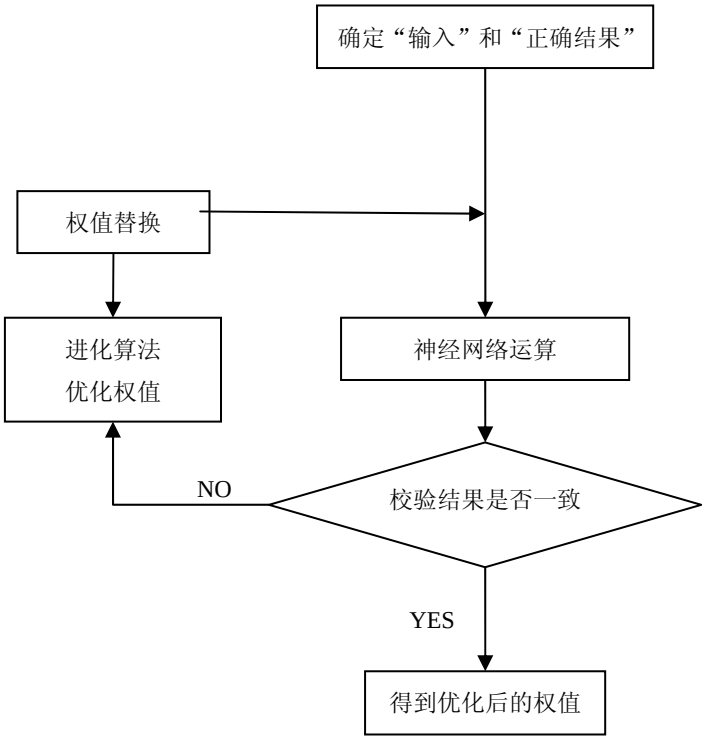

图 1.4 神经网络权值优化流程

2　遗传算法简介以及简单实例

通过对遗传算法的简单介绍，我们对遗传算法并没有多深刻的理解。下面给出一个程序，相信我们将对遗传算法的了解会变得深刻些。

2.1　遗传算法的一般步骤

遗传算法编程的一般步骤分为以下：

1. 计算染色体的适应度，记录下每条染色体的适应值。
2. 用轮盘法选出两条染色体。
3. 将这两条染色体进行组合交叉，用随机函数确定组合交叉进行的位置。
4. 将新组合成的个体进行突变测试。
5. 重复 2、3、4 步骤，直到新的种群个体数与原种群个体数的相同。

2.2　迷宫寻路问题

《帮助 Bob 找到回家的路》是一个寻路程序。图 2.1 是一个 2D 迷宫，Bob 要从右下的红方块，找到左上的红方块。

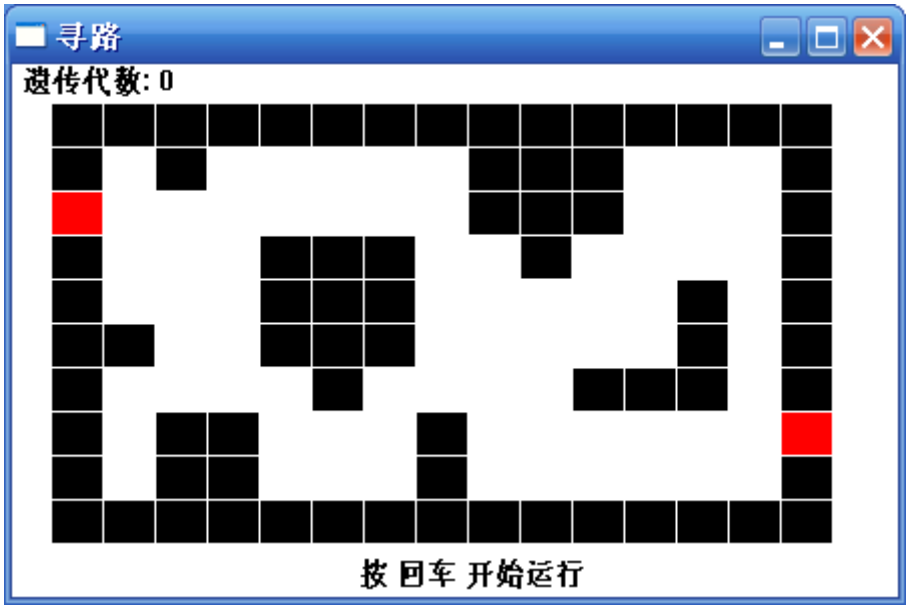

图 2.1 寻路程序效果

第一步我们得确定一个解决的方案，也就是一个个体。它代表了一个解决问题的方案，但不一定能够找到答案，应该说大有可能找不到答案。每一个个体都有很多的基因组成，很多基因组合在一起才能表现出一个个体的行为。在这个程序中，我们要解决的问题是——从一个点到另一个点。Bob 走出的每一步就相当于一个基因，Bob 可以选择往上、下、左、右四个方向移动。那么基因的种类就有四种。用 2 进制表示所有的可能只需要两位。见下表 2.1：

表 2.1 方向编码

二进制	十进制	方向
00	0	北
01	1	南
10	2	东
11	3	西

染色体长度与问题本身休戚相关，从这个迷宫来看，Bob 从起点到终点的最好方案需要 18 步。那么染色体的长度将大于或者等于 36，为什么是 36 而不是 18 呢，原因在于我们用 2 个二进制数表示了一个基因，也就是说基因长度为 2。

然后确定种群中的个体数量，这个数的取值直接关系到解决问题所需要的遗传代数。就像掷色子一样，你有 2 个色子，分别给你掷 10 次和 20 次，20 次能掷出 6.6 的组合的概率肯定比 10 次的大。但是由于在遗传算法中，虽然种群中的个体数越大，找出可行方案所需的遗传代数将会变小，但是，由于个体数量直接决定了遗传算法在每一代遗传中的运算量，所以在用遗传算法解决问题时，无限量地提高种群个体数将得不偿失。

遗传算法中，另有两个因素十分的重要。一个是发生组合交叉的概率，一个是突变的概率。组合交叉的概率决定了，父母代是否要进行组合交叉。为什么要在遗传算法中添加这一个变量呢？当随机数不能满足交叉组合条件的时候，程序将完成染色体复制的过程。在遗传算法中，父母都是筛选出来的，适应度越高的个体被选到的概率越大，换言之，能作为父代母代的都是比较能适应环境的。设置这个变量的好处：能够让遗传算法得到子代的种类更加的丰富，自然也能够节省一定的遗传代数。下一个是突变率，突变率的存在是必要的。现在地球上的生物都是通过突变而衍变成现在这样，他们都能够很好的适应环境。如果没有突变，世界将不堪设想。突变率的设置在程序中起到了重要的作用。以这个程序为例，如果我们在开始决定个体基因的时候，每一个个体的基因中都没有向北（00）的基因。那丢弃了突变率这个量，后果不堪设想。因为 Bob 不可能不往北找到终点。

2.3 迷宫寻路问题关键代码

根据遗传算法编程的一般步骤分析，其中有这样几个问题十分的重要。

1. 如何判断个体的适应程度；
2. 如何随机的找到两条适应度比较高的个体；
3. 怎样进行交叉组合。

在这个程序中，我们定义了一个迷宫类：CBobsMap；

计算个体的适应度由 CBobsMap 类完成，以下罗列了地图的基本信息：

```
static const int   map[MAP_HEIGHT][MAP_WIDTH];//存储地图信息数组
static const int   m_iMapWidth;                 //地图宽度
static const int   m_iMapHeight;                //地图高度
static const int   m_iStartX;                   //起始点 X 坐标
```

```
static const int    m_iStartY;                          //起始点 Y 坐标
static const int    m_iEndX;                            //终点 X 坐标
static const int    m_iEndY;                            //终点 Y 坐标
int   memory[MAP_HEIGHT][MAP_WIDTH]; //存储的当前地图信息
```

这些信息在遗传算法计算个体的适应度中使用到。

计算个体适应度是一个十分重要的问题，我们设计计算适应度程序时需要注意：每一个个体都有可能成为下一代个体的父母代。所以在计算个体适应度时，我们尽量不能让任何一个个体的适应度为 0。

适应度决定染色体符合要求的程度。我们将染色体翻译成移动方向的集合，从起点坐标开始对每一个方向进行比较，当将移动到的方位可以行走（值为 0），bob 向该方向移动；如果将移动到的方位不可行走（值为 1），bob 保持原地不动。当染色体上所有的基因全部测试完后 bob 的所在点为 P（x，y）。终点为 E（endx，endy）。

测试染色体适应度的函数：Fitness = $\frac{1}{|x-endx|+|y-endy|+1}$；

测试适应度代码如下：

```
double CBobsMap::TestRoute(const vector<int> &vecPath, CBobsMap &Bobs)
{//从起始点开始
    int posX = m_iStartX;
    int posY = m_iStartY;
    for (int dir=0; dir<vecPath.size(); ++dir)
    {
        int NextDir = vecPath[dir];
        switch(vecPath[dir])
        {
        case 0: //北
            //测试下一步能不能行走
            if ( ((posY-1) < 0 ) || (map[posY-1][posX] == 1) )
            {//不能行走呆在原位
                break;
            }
            else
            {      //可以行走
                posY -= 1;
            }
            break;
```

```
        case 1://南
                ……..break;
        case 2://东
                ……..break;
        case 3: //西
                ……..break;
        }
        Bobs.memory[posY][posX] = 1;
        //找到终点直接返回
        if(posY==m_iEndY && posX==m_iEndX)
            return 1;
    }
    //所有基因都被测试后
    //计算最终的位置与终点 X 轴 Y 轴的距离
    Int    DiffX = abs(posX - m_iEndX);
    int    DiffY = abs(posY - m_iEndY);
    //计算适应度
    return 1/(double)(DiffX+DiffY+1);
}
```

从这个函数来看，我们很容易得到了此程序推出循环的条件（找到一条答案）——当此函数返回 1 的时候。

第二个问题：如何找到适应度比较高的个体？

轮盘法可以解决这一问题，什么是轮盘法？简单的说就是高概率的被选到的概率大，低概率的相对较小。正如下面这条线段，这条线段分为几条长短不一的小线段，长的可以认为是适应度高的个体，而短的则是适应度低的。当电脑随机选择一个区间在 0 至 10 的数，这

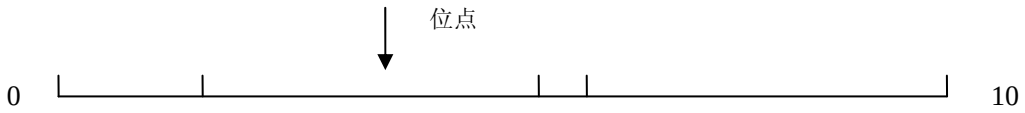

图 2.2 轮盘法示意图

个数落在长度较长的线段上的概率相对较高。如图 2.2 所示。

轮盘法选择适应度较高的个体函数为：

```
SGenome& CgaBob::RouletteWheelSelection()
{//选取一个点
    double fSlice     = RandFloat() * m_dTotalFitnessScore;
    double cfTotal    = 0.0;
```

```
        int   SelectedGenome = 0;
        //循环计算该点落在那个个体上
        for (int i=0; i<m_iPopSize; ++i)
        {
            cfTotal += m_vecGenomes[i].dFitness;
            if (cfTotal > fSlice)
            {//找到该点退出循环
                SelectedGenome = i;
                break;
            }
        }
        return m_vecGenomes[SelectedGenome];
    }
```

第三个问题是如何进行交叉组合？

给出两条染色体

染色体 1：　　0100110100000110

染色体 2：　　1101100101101101

首先要选择交叉组合的位点，比如位点选择在了 8 这个位置，两条染色体进行交叉组合。

染色体 1：　　**11011001** 00000110

染色体 2：　　**01001101** 01101101

字体加粗部分为交叉组合的基因段。

我们将交叉组合的函数命名为 Crossover（……）；原型如下：

```
void CgaBob::Crossover( const vector<int> &mum,
                        const vector<int> &dad,
                        vector<int>       &baby1,
                        vector<int>       &baby2)
{   //是否需要进行交叉组合
    if ( (RandFloat() > m_dCrossoverRate) || (mum == dad))
    {//父母直接作为下一代的个体
        baby1 = mum;
        baby2 = dad;
        return;
    }
    //选择交叉组合位点
    int cp = RandInt(0, m_iChromoLength - 1);
    //进行染色体段的交换
```

```
    for (int i=0; i<cp; ++i)
    {
        baby1.push_back(mum[i]);
        baby2.push_back(dad[i]);
    }
    for (i=cp; i<mum.size(); ++i)
    {
        baby1.push_back(dad[i]);
        baby2.push_back(mum[i]);
    }
}
```

2.4 遗传算法的时间复杂度

遗传算法是一个需要大量运算的算法，它的不确定性十分的大。就像这个程序一样，Bob 根本就不知道回家的路，他通过摸索才能回到家。

让我们计算一下这个程序的时间复杂度。

在程序开始之前，我们要定义几个量。种群的个体数量，个体的染色体长度，以及基因的长度。根据上文得到基因长度为 2。种群的个体数量和个体染色体长度，我们假设他们为 pop_size，chromo_length，简称 P 和 C。

程序开始时，种群中没有任何个体，程序给这个种群增加这些个体。假设给每个个体赋值所有用的时间复杂度为 C，初期赋值所用的时间复杂度为 t*P*C。接下来就是将染色体的基因转化成 10 进制，这次操作需要消耗 n*C*P 的时间复杂度，之后是计算适应度，适应度计算需要时间复杂度为 m*C*P。轮盘法选择个体耗费 x*P，交叉组合为 y*C*P，突变为 z*C*P。第一次循环结束，进入下次循环。一次循环需要的时间复杂度为：n*C*P+m*C*P+x*P+y*C*P+z*C*P=(n+m+y+z)*C*P+x*P，假设找到最佳答案的循环次数为 G。那么整个程序找到答案的时间复杂度为：t*P*C+((n+m+y+z)*C*P+x*P)*G
程序的时间复杂度为 O（P*C*G）。

这个程序的时间复杂度取决于三个因素：染色体长度，个体的数量，遗传的代数。这三个因素的确定有很大的客观性。染色体的长度可以选择 36 以上的任何一个，当你把染色体个数设置为 36 的时候，这样看来这个个体将是十分完美的，它就是一条到达终点的最短路径。在这个程序开始时，每个个体的基因都是由计算机随机取的，个体的不确定因素十分多。如果每个个体的基因没有那条基因是通向最短的那条路径，那么一切将不堪设想。唯一的可能就是等待突变。而在这程序中突变率，只有千分之一，可想而知这样低的突变率，什么时候才能突变出适合的基因呢？当然提高突变几率在这样的情况下可以增加遗传找到终点的概率。好吧，让我们将染色体的长度加大。染色体的长度直接的决定了 Bob 回家的路径的多少。染色体越大，路径越多，找到终点的概率相对也较大。个体的数量直接的决定了个体中基因

的多样性，个体数越大对答案的产生越有利。个体数超大也不能够体现出程序的优化，毕竟P*C*G 的值会很大。最后就是遗传代数，这个因素并不是由程序员来决定的，当然也不好确定，只能够根据多次测试才能够将其近似值确定下来。

2.5 遗传算法的总结

通过这个程序，我们对遗传算法有了更深入的了解。遗传算法可以解决一切求取最优解的问题，但是在具体问题面前，也许会变得不可取。大家也许会提出这样的疑问：这只是个简单的寻路程序为什么要用遗传算法来做的？在利用遗传算法来解决这个问题之前，我们可以通过对所有路径的遍历得到一条最优的解，而且在运算效率方面也大大的优于遗传算法。再设想出口是被封死的，那通过遗传算法来计算最优解又会是什么样的结果呢？

我们知道遗传算法能够确定一些不确定的因素，遗传算法的这一特性正好适合来找神经网络算法中各个神经元的权值。如何用遗传算法优化神经元的权值？将会在下一章介绍。

3 基于遗传算法的神经网络优化实例

通过上一章对遗传算法的研究，我们知道遗传算法可以找到问题中不确定因素。这一特性正好在这一章中得到应用。

3.1 神经网络介绍

神经网络是生物大脑思维的形式。生物的大脑有很多的神经元组成，当生物受到刺激，电流信息将会从突出末梢通过轴突传递到神经细胞中，然后神经细胞对信息进行处理后从树突传递给下一个神经元。大量的神经元之间进行传递之后，最终生物就做出了反应。

3.2 神经元的数学模型

人工神经网络（ANN）是模仿生物大脑而建立起来的，它是由很多个神经元相互的链接形成的一种结构。每一个人工神经元是程序模拟的简化了的生物神经元，用 ANN 进行编程所使用的神经元个数的多少取决于程序的解决问题的复杂程度，让我们来看看 ANN 中的神经元的庐山真面目。如图 3.1 所示：

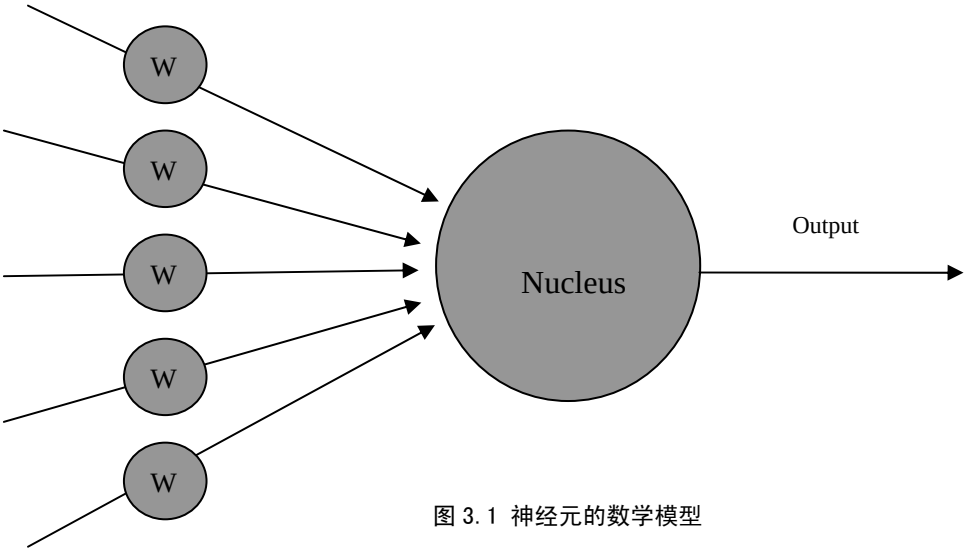

图 3.1 神经元的数学模型

图中标有 w 的小圈代表了 ANN 中神经元中突出末梢的权重（weight）。ANN 中的输入（input）和这些重量决定了神经网络的整体活动。所有这些权重都设置为-1 到 1 之间的随机值。这些权重直接的影响了神经元的输出。他们起到的作用可能的正面的也可能是负面的。那这些权重是如何联系起来决定整个神经元的输出（output）的呢？我们用一个数学公式来解决：

$$a=w_1x_1+w_2x_2+w_3x_3+.....+w_nx_n$$

用希腊字母表示 $\sum_{i=0}^{i=n} w_i x_i$ ；

在人工神经网络编程中，这些是远远不够的。通过对各个输入和权值乘积的累加运算后，得出的值（activation）并不是能够直接作为输出，成为下一个神经元的输入因子，正如人工神经网络工作原理中写的，输出将被限制为 0 或者 1。在人工神经网络中，定义了一个临界

值来限制输出值。如果输出的值大于临界值，则输出值为 1，反之为 0。用数学坐标如图 3.2 所示：

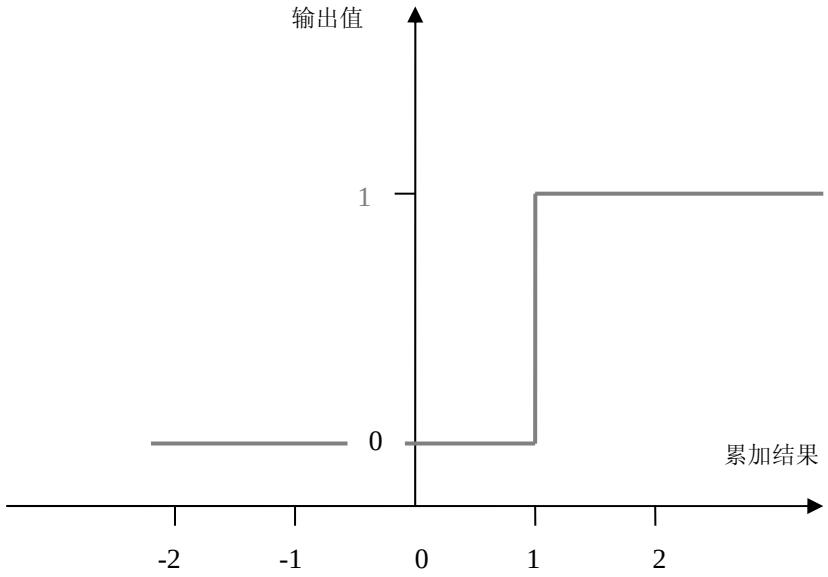

图 3.2 输入输出的数学坐标

将这个临界值设置为 1，那么当输出值大于 1 的时候，输出值为 1。小于的时候输出为 0。

3.3 神经网络结构

在生物大脑内，神经元与神经元直接是相互连接的，他们以某种特定的形式连接。如下图 3.3 所示，这是一个前馈网络，输入值通过输入层传递到隐含层，图中圆圈表示神经元，输入层输入的值，经过隐含层神经元的计算，隐含层的输出作为下一层输出层的输入，经过输出层处理后得到的结果才是最终的值。

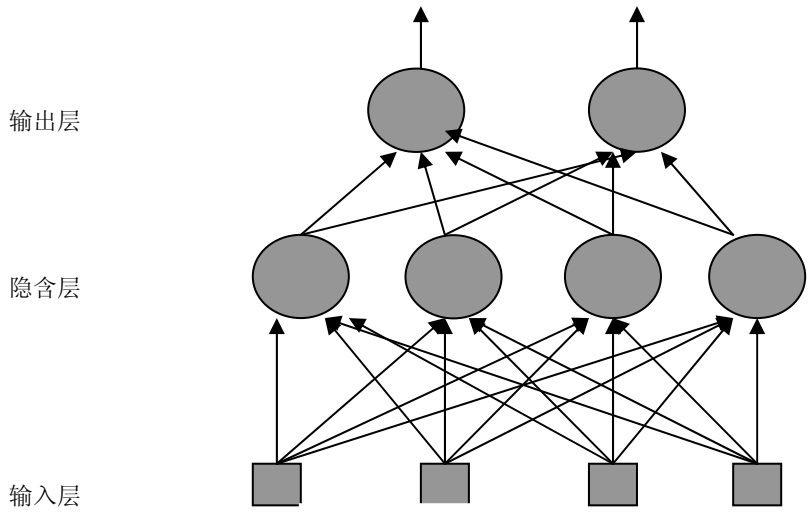

图 3.3 神经网络结构

神经网络并不局限于这个模型，隐含的数量也可以根据问题需要增加或者减少。输入输出的个数也是按照问题需要而异。接下来我将介绍一个关于神经网络的实例。

3.4 神经网络实例

接下来介绍一个用神经网络做的智能采矿程序。我们用神经网络来实现小车自动采矿，通过遗传算法对小车神经元权值进行调整，使小车能够更加灵活地采矿。程序运行效果如图 3.4 所示：

3.4.1 程序运行效果

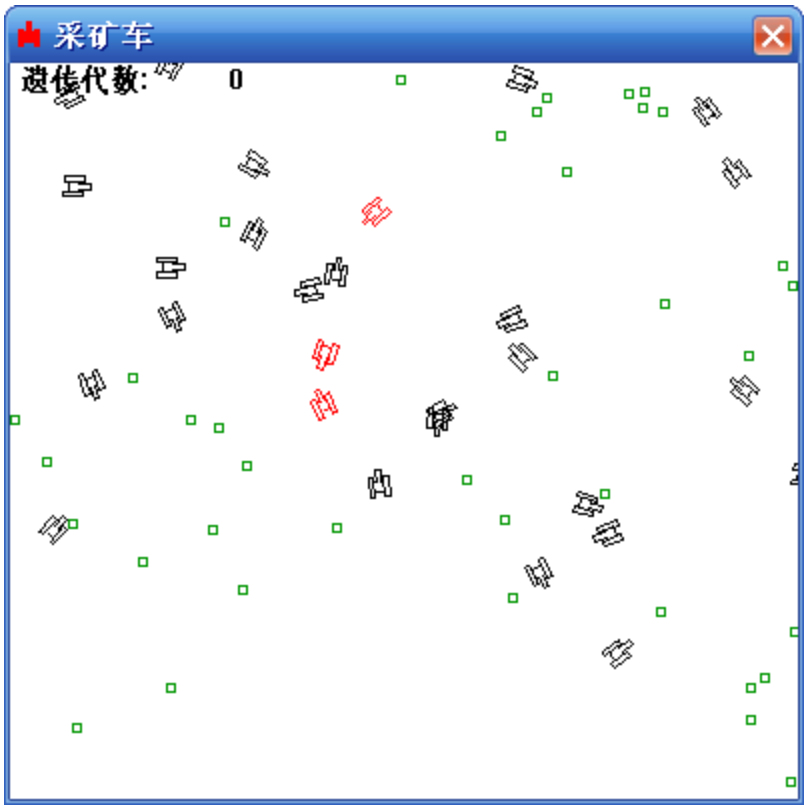

图 3.4 采矿车效果

图中有 2 种颜色的小车，这两种小车都是整个种群中的个体。红色的小车为适应度比较高的。绿色的方块代表了矿。

3.4.2 确定神经网络的输入输出

在程序开始前，我们得确定输入输出的数量以及他们的内容。首先，我们要知道小车自身的位置（坐标），最近矿的位置也要作为输入。这样两个坐标就有 4 个输入，分别是小车坐标（x_1，y_1）以及矿的坐标（x_2，y_2）。但这些就足够了吗？当然不是，小车有自己的速度，速度用一个矢量表示（x，y）。如此我们就有了 6 个输入。在神经网络中，输入的多少决定了每个神经元的输入个数，从而时间复杂度将大大的增加。在神经网络优化中，如何减少输入的多少是最重要的。

在这个程序中，其实小车的坐标和矿的坐标可以合成一个矢量——小车指向矿的矢量（y_1-x_1,y_2-x_2）。如图 3.5 所示：

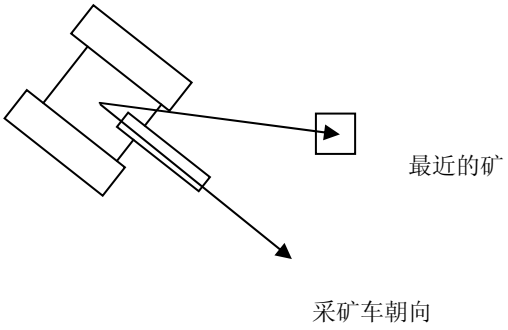

图 3.5 小车输入变量

输入减少到了 4 个。

输入已经找出，那接下来就是决定输出的个数。输出的个数在程序中只影响到输出层神经元的个数，如果神经网络庞大的话，输出层所消耗的 CPU 时间往往可以忽略。在决定输出之前，让我们看下小车是如何设计的。如图 3.6 所示：

神经网络的输出来控制小车的移动。小车的左右连个轮子动力的大小可以决定小车移动的速度以及小车的转弯。如此看来这两个量就可以决定小车的移动。那么，该神经网络的输出为 2 个，一个左轮的力，另一个是右轮的力。小车的转向值通过左轮力减去右轮力而来，小车速度则是他们的和。

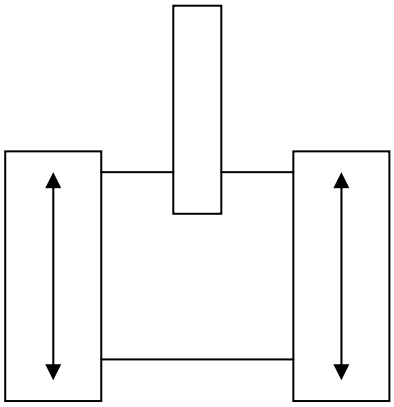

图 3.6 采矿车模型

```
RotForce = m_lTrack - m_rTrack;          //转向力度
m_dSpeed = m_lTrack + m_rTrack;          //速度
```

或许大家会有疑问，神经网络的输出为 0 和 1，但是在这个问题中，如果输出为 0 和 1 的，那么小车的移动将只有 4 种情况。在这个程序中，我们不把神经元的输出限定为 0 和 1，取而代之，它的输出将被限定在 0 到 1 之间的小数。这样小车就能够很自然的移动了。既然每个神经元的输出被限定在了 0 到 1 之间的小数，它如何来实现呢。下图是一个数学坐标图（如图 3.7 所示）；它能够很形象的解决这个问题。

这个曲线的数学方程式为 $output = \frac{1}{1+e^{-a/p}}$ 。e 为常量，e 的近似值为 2.7183。p 值通常被设置为 1。

那我们需要多少个隐含层呢？先来说说隐含层的好处吧。隐含层顾名思义它并没有在实际的输出上引用，但是他对输出有着决定性作用。就譬如这个问题吧。当没有隐含层的时候，小车会做出一些十分古怪的动作，即使是遗传很多次的个体同样存在着这样的问题。众所周知，人类大脑很发达，其神经元的个数是其他动物的千万倍。为什么人类能统治地球，其原因可想而知。在这个问题上，我们将隐含层的个数设置为 1。这样小车的行为看起来会更加的自然。

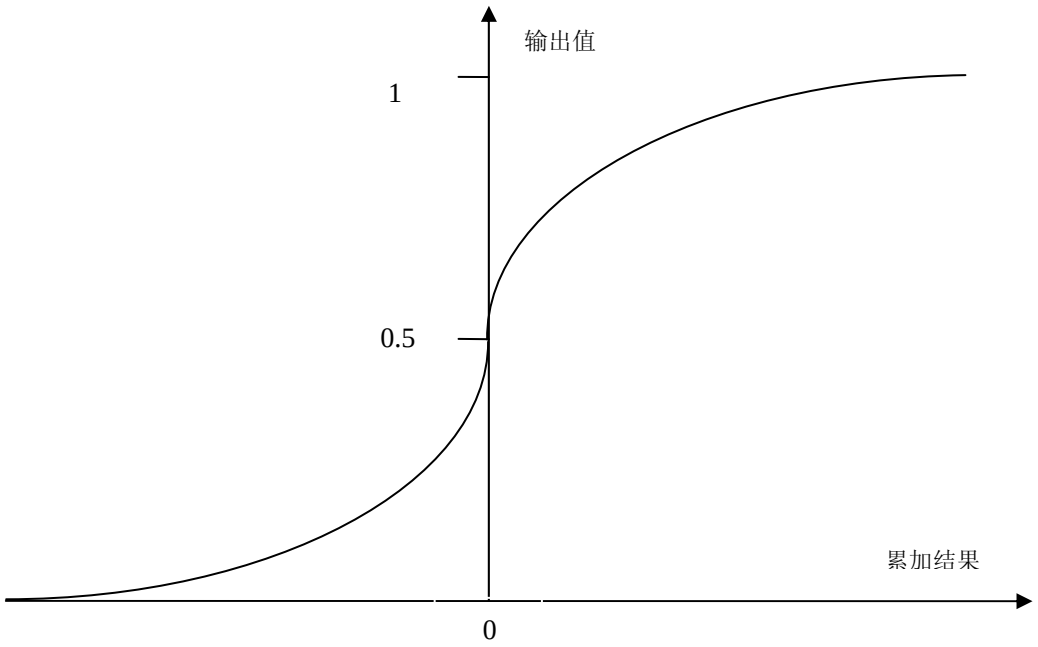

图 3.7 神经元输出值曲线

3.4.3 程序介绍

介绍了那么多，让我们看看神经元、隐含层以及神经网的定义吧。

神经元：struct SNeuron

```
{
  //输入的数量
  int                  m_NumInputs;
  //每个输入的权重
   vector<double>   m_vecWeight;
   SNeuron(int NumInputs);
};
```

隐含层：struct SNeuronLayer

```
{
   //隐含层中神经元的数量
   int                   m_NumNeurons;
   //隐含层中所有神经元
   vector<SNeuron>   m_vecNeurons;
   SNeuronLayer(int NumNeurons, int NumInputsPerNeuron);
};
```

神经网：class CNeuralNet

```
{
private:
```

```
int                     m_NumInputs;            //输入个数
int                     m_NumOutputs;           //输出个数
int                     m_NumHiddenLayers;      //隐含层数量
int                     m_NeuronsPerHiddenLyr;  //隐含层的神经元数
vector<SNeuronLayer>    m_vecLayers;//隐含层
CNeuralNet();
  //创建神经网络
  void    CreateNet();
  //获取神经网络的所有神经元的权重
  vector<double>     GetWeights()const;
  //获取神经网络所有神经元的权重个数
  int                GetNumberOfWeights()const;
  //替换所有权重
  void               PutWeights(vector<double> &weights);
  //S 型曲线函数
  inline double     Sigmoid(double activation, double response);
  //计算输出
  vector<double>    Update(vector<double> &inputs);
  };
```

介绍了神经网络的一些重要的函数之后，遗传算法如何加入到神经网络中，成了接下来要解决的问题。其实，遗传算法并没有直接的跟神经网络进行挂钩，它还是很平常的遗传算法一样完成自己的使命（上一章寻路问题）。唯一不同的就是他的基因染色体不是简单的 0 和 1 序列了。神经网络中所有神经元的权重将成为遗传算法中的染色体。那么，遗传算法中的染色体长度将由神经网络中神经元来决定。

如何来测试个体的适应度才是这个程序比较核心的问题。这并不像上一章中的寻路问题一样简单了，在这个问题中，对个体适应度的测试并不是一步就能解决的。用观察的方式来测试个体适应度，我们给每个小车 2000 步（游戏 2000 帧）的机会，当小车吃到一个矿的时候小车的适应度加 1，直到 2000 步结束。假定小车一步能够采到的矿个数为 m，那么适应度函数如下：

$$\text{Fitness} = \sum_{i=1}^{2000} m_i$$

```
struct SGenome
{
   vector <double>    vecWeights;
   double              dFitness;
   SGenome():dFitness(0){}
```

```
    SGenome( vector <double> w, double f): vecWeights(w), dFitness(f){}
    //重载 '<' 用于排序
     friend bool operator<(const SGenome& lhs, const SGenome& rhs)
     {
        return (lhs.dFitness < rhs.dFitness);
     }
};
```

变异程序：void CGenAlg::Mutate(vector<double> &chromo)

```
{
   for (int i=0; i<chromo.size(); ++i)
   {
      if (RandFloat() < m_dMutationRate)
      {
         chromo[i] += (RandomClamped() * CParams::dMaxPerturbation);
      }
   }
}
```

在突变程序中，我们并没有看到像寻路问题中直接将基因取反操作，而是将该基因（权重）加上了一个很小的数（可正可负），该数完全由电脑给出的随机数。

小车类（遗传算法中的个体）：

```
class CMinesweeper
{
private:
   //小车的大脑（神经网络）
   CNeuralNet    m_ItsBrain;
   //小车所处在的位置坐标
   SVector2D    m_vPosition;[4]
   //小车的朝向
   SVector2D    m_vLookAt;
   //小车速度（速度包含了方向和速度大小，第一个变量代表了转弯的大小）
   double       m_dRotation;
   double       m_dSpeed;
   //存储神经网络输出，左轮力量以及右轮力量
   double       m_lTrack,
                m_rTrack;
   //小车适应度
```

```
    double        m_dFitness;
    //小车显示比例大小（游戏中应用较广）
    double        m_dScale;
    //最近矿的编号
    int           m_iClosestMine;
    CMinesweeper();
    //测试适应度的核心，在此程序中每次遗传需要运行 2000 次
    bool          Update(vector<SVector2D> &mines);
    //世界坐标的转变
    void          WorldTransform(vector<SPoint> &sweeper);
    //获取最近矿的坐标
    SVector2D   GetClosestMine(vector<SVector2D> &objects);
    //查看是否已经找到了一个矿
    int           CheckForMine(vector<SVector2D> &mines, double size);
    void          Reset();
    SVector2D   Position()const{return m_vPosition;}
    void          IncrementFitness(double val){m_dFitness += val;}
    double        Fitness()const{return m_dFitness;}
    void          PutWeights(vector<double> &w){m_ItsBrain.PutWeights(w);}
    int           GetNumberOfWeights()const{return m_ItsBrain.GetNumberOfWeights();}
};
```

遗传算法核心类：

```
class CGenAlg
{
private:
    //所有个体的染色体
    vector <SGenome>      m_vecPop;
    //个体数量
    int m_iPopSize;
    //个体基因长度
    int m_iChromoLength;
    //种群的适应度
    double m_dTotalFitness;
    //最佳个体适应度
    double m_dBestFitness;
    //平均适应度
```

```
    double m_dAverageFitness;
    //最差个体适应度
    double m_dWorstFitness;
    //最佳个体编号
    int         m_iFittestGenome;
    //突变概率
    double m_dMutationRate;
    //交叉组合率
    double m_dCrossoverRate;
    //记录遗传代数
    int      m_cGeneration;
    void      Crossover(const vector<double> &mum,
                        const vector<double> &dad,
                        vector<double>          &baby1,
                        vector<double>          &baby2);
    void      Mutate(vector<double> &chromo);
    SGenome     GetChromoRoulette();
    void    GrabNBest(int                   NBest,
                      const int          NumCopies,
                      vector<SGenome>  &vecPop);
    void    FitnessScaleRank();
    void    CalculateBestWorstAvTot();
    void    Reset();
public:
    CGenAlg(int        popsize,
            double   MutRat,
            double   CrossRat,
            int        numweights);
    //遗传一代
    vector<SGenome>     Epoch(vector<SGenome> &old_pop);
    vector<SGenome>     GetChromos()const{return m_vecPop;}
    double                  AverageFitness()const{return m_dTotalFitness / m_iPopSize;}
    double                  BestFitness()const{return m_dBestFitness;}
};
```

当遗传算法的问题解决之后，但是我们发现神经网络和遗传算法之间似乎没有任何联系，所有我们继续一个类来将他们进行有机的结合。我们又编写了这么一个控制类，来把两者联系。接下来

让我们揭开这个程序的神秘面纱：

```
class CController
{
private:
    //存储所有个体基因
    vector<SGenome>            m_vecThePopulation;
    //个体
    vector<CMinesweeper> m_vecSweepers;
    //矿
    vector<SVector2D>        m_vecMines;
    //遗传算法类指针
    CGenAlg*                    m_pGA;
    int                              m_NumSweepers;
    int                              m_NumMines;
    int                              m_NumWeightsInNN;
    //绘制小车所需
    vector<SPoint>             m_SweeperVB;
    vector<SPoint>             m_MineVB;
    //存储平均适应度
    vector<double>             m_vecAvFitness;
    //存储最佳适应度
    vector<double>             m_vecBestFitness;
    //用于画图
    HPEN                       m_RedPen;
    HPEN                       m_BluePen;
    HPEN                       m_GreenPen;
    HPEN                       m_OldPen;
    //窗口句柄
    HWND                       m_hwndMain;
    //选择模式
    bool                  m_bFastRender;
    //小车步数，测试适应度所需
    int                        m_iTicks;
    //遗传代数记录
    int                        m_iGenerations;
    //窗口大小
```

```
    int             cxClient, cyClient;
    void    PlotStats(HDC surface);
public:
    CController(HWND hwndMain);
    ~CController();
    void    Render(HDC surface);
    void    WorldTransform(vector<SPoint> &VBuffer,
                                        SVector2D       vPos);
    bool    Update();
    bool    FastRender()const       {return m_bFastRender;}
    void    FastRender(bool arg){m_bFastRender = arg;}
    void    FastRenderToggle()  {m_bFastRender = !m_bFastRender;}
};
```

3.4.4 问题的优化

从这个程序来看，神经网络的确是个消耗 CPU 时间的大户。如果我们要找出一个十分适应的个体并将它的各个神经元的权重记录下来，我们得等待程序遗传几百代，或许更多。优化这个问题有很多的方法，其中之一就是遗传过程中的交叉组合。在这个程序中，交叉组合的位点是程序自动选择的随机值，这样选择其实并不合理，原因很简单。程序中的基因代表了一个权重，我们所说的位点是以基因为单位的。本来这个神经元的各个权重配合相当的完美，但事实恰恰将位点落到了这个神经元各个权重之间，因此神经元中的完美配合将无形地被摧毁。如果将这个程序的交叉组合函数进行修改，将其改成以神经元为单位进行交叉组合的话，在某种情况下可以对程序做出优化。对于这个程序的优化还有更多的方法，比如优化适应度测试方法。正如上一章所说，我们可以增加个体数来减少遗传的代数。当然一切都以实际情况出发。

总结

遗传算法和神经网络在游戏开发中的应用越来越广泛。在游戏开发中游戏运行时的效率问题越来越受到了重视，本文在讲述遗传算法和神经网络在游戏开发中应用的同时，也对游戏开发中程序的优化做了些许讲述。遗传算法的编程过程中要注意染色体的长度和个体个数的组合。也可以通过改变突变的几率、交叉组合几率、适应度函数选择等几个方面来达到算法的优化。在神经网络编程开始前，我们必须把神经网络的结构确定，把问题的输入输出决定。然后才进行进一步的分析。神经网络在游戏人工智能方面的贡献十分明显，如何设计神经网络才是最最重要的。这方面的应用很多都能在日常生活中得到启发。

参考文献

[1] 云庆夏. 遗传算法和遗传规划 ：一种搜索寻优技术[M]. 冶金工业出版社. 1997

[2] Andre LaMothe. AI TECHNIQUES FOR GAME PROGRAMMING[M].Stacy L.Hiquet.2002

[3] 候媛彬，杜京义，汪　梅. 神经网络[M] . 西安电子科技大学出版社.

[4] 倪明田, 吴良芝. 计算机图形学[M].北京大学出版社: 1999.11

[5] 谭浩强. C++面向对象程序设计[M].清华大学出版社：2006.01

[6] 2007.01 柏拉图. 遗传算法的数学基础[M].西苑出版社. 2003.05

[7] 周　明，孙东栋.遗传算法原理及应用[M].国防工业出版社.2002.05

[8] 张良均，曹晶，蒋世忠. 神经网络实用教程[M].机械工业出版社.2008.02

[9] 张青贵. 人工神经网络导论[M].中国水利水电出版社.2004.10

[10] 韩力群.人工神经网络教程[M].北京邮电大学出版社.2006.06

[11] 叶 涛，朱学峰.关于过程神经网络的理论探讨[J].智能系统学报.2007. (05)

[12] 杨新敏 孙静怡 钱育渝.城市交通流配流问题的遗传算法求解[J].城市交通.2002.(02)

[13] 张　铃，张　钹. 统计遗传算法[M].安徽大学人工智能所，清华大学计算机系.1997.(05)

[14] 郭崇慧. 若干最优化方法的收敛性分析及应用研究[D].大连理工大学, 2002

[15] 李盼池. 过程神经网络模型及学习算法研究[D]大庆石油学院, 2004

[16] 邢贞相. 遗传算法与 BP 模型的改进及其在水资源工程中的应用[D]东北农业大学, 2004

[17] 裴浩东. 基于神经网络的稳态优化和控制研究[D]浙江大学, 2001

[18] 王晓琳. 遗传算法与神经网络在汇率预测中的应用[D]青岛大学, 2006

[19] 廖　平. 基于遗传算法的形状误差计算研究[D]中南大学, 2002

[20] 吴　昊. 并行遗传算法的研究与应用[D]安徽大学, 2001