1. **伪随机数产生方法**
   1. **平方取中法**

产生伪随机数列最早的方法是平方取中法，它是由冯·诺依曼提出的。

此法开始取一个位的整数，称为种子，将其平方，得位整数(不足位的高位补0)，然后取此位的中间位作为下一个种子数，重复上述过程，即可得到一系列随机数。**平方取中法的递推公式为：**

产生伪随机数列：

平方取中法的优点为在计算机上易于实现，内存占用少，但仍存在对小数目偏倚的现象，均匀性不好，对初始数据的依赖很大，数列的长度和周期难以确定，很容易出现重复元素的短循环，而且，一旦某个元素是0，则后面所有的元素都将是0。

* 1. **线性同余法**

目前大多数的伪随机数发生器采用这种方法来生成随机数。它的优点是计算速度快、容易实现、代码可移植性强等，缺点是不能用于随机数要求高的场合。

**线性同余法的递推式：**

|  |  |  |  |
| --- | --- | --- | --- |
| x0种子(初值) | a乘子 | c增量 | m模 |

如果这些参数和种子（初值）都指定序列也就确定下来了。通常取作为区间(O，1)上均匀分布的随机数。（当，为乘同余法；当时，为加同余法，否则称为混合同余法。）

**线性同余法有如下特点:**

(1)。

(2)适当选取可使循环，无论取何值，其循环顺序相同，其循环

周期称为发生器周期，记为。若，则称之为满周期。

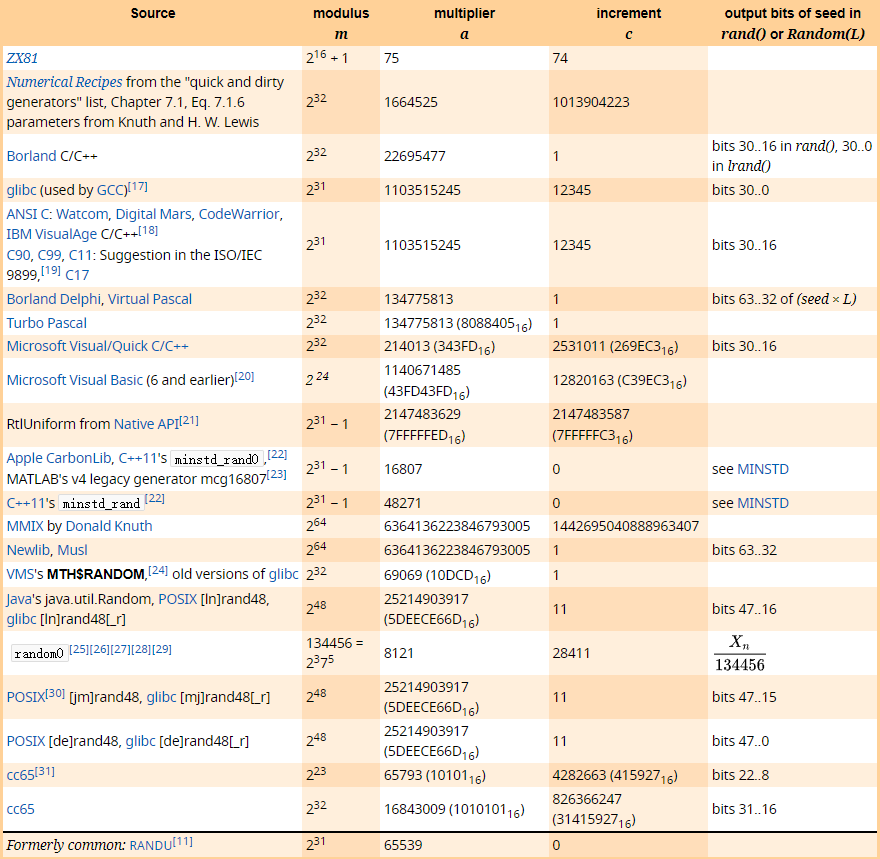
(3)适当选取，可保证在[0, 1]区间上一个周期内每个整数正好出现一次，从而保证了均匀性。

(4)为提高的均匀性，要求加大。

**参数选取方面：**

若S为计算机位数。当取中最大的数素，数论表明若是的原根（满足且与互素的数），则同余算法随机数发生器的循环周期，这样的随机数发生器达成了完成周期，而且其中不包括0。

若要便于计算，取可形式（；）这样在计算中可以使用移位和指令加法，提高计算速度，但是这种简单的选择，随机效果可能会差一点。下图为一些方法采用的参数选取：



<https://wiki2.org/en/Linear_congruential_generator>

* 1. **移位法**

计算机善于进行移位等逻辑运算，应用计算机的这个特点有一类产生伪随机数列的方法，该方法称为移位法。本方法是关于平方取中法的一种改进形式，它是移位运算与指令相加运算相结合的迭代过程。

它的思想是：首先选取一个初始值，使之左右移位，分别为和，然后指令相加得，再对进行上述过程，如此重复下去，便可得随机数序列。对于字长为32位的计算机，这一算法的递推公式为：

产生伪随机数列：

移位法运算速度快，但是对初始值的依赖性也很大，一般地初始值不能取得太小，选得不好会使伪随机数列长度较短。同时维均匀随机向量相关性大，即独立性较差。最后随机数序列周期与计算机的字长有关。

* 1. **斐波那契(Fibonacci)法**

斐波那契法是基于Fibonacci序列，其递推公式为:

显然，斐波那契法有两个初始种子和。方法的最大特点就是计算速度很快，且达到满周期。但是序列中数重复出现，独立性较差。此发生器没有乘法运算，产生速度快。但是它存在着令人不能容忍的不居中现象，即由前两个数得到的第三个数要不是同时大于就是同时小于前二者而永不居中。此序列的另一个缺点是显著的序列相关，即取小值的数后面出现也取小值的趋势。所有这些都说明它不是一个好的随机数序列。

* 1. **混沌函数生成方法**

混沌的定义很多，也有很多判断方法，数学角度Li-York定义被普遍接受。设f(x)在区间I上连续的自映射，若f(x)有3周期点，则对任何正整数n，f(x)有n周期点。如果满足下列条件，便确定它有混沌现象：

1. f(x)的周期点的周期无上界，f(x)具有任意正整数周期的周期点，即对任意自然数n，有x∈I，使；
2. 存在I的不可数子集S，使得对于任意x，y∈S，满足：

对于任意x,y∈S且x≠y时有：

对于任意x,y∈S时有：

对于任意x∈S和f的任意周期点y有：

则此f(x)是混沌的。

**几种常见的混沌模型：**

1）Logistic映射模型

其中为分形参数，。当时，Logistic模型处于混沌状态。

2）Lorenz混沌模型

当参数取典型值为时，系统处于混沌状态。

1. Kent映射模型

当初始值=0.32时，系统处于混沌状态。

在随机数生成算法中可使用混沌映射，并配合物理墒作为初始状态，能够使整个生成系统的输出体现得更“混乱”。

1. **网络初始化**
   1. **均匀分布初始化**

均匀分布的概率密度函数为：

通常用U(a,b)表示。均匀分布的均值为,方差为。

均匀分布随机数：利用混合同余法产生（0，1）区间的随机数，再通过变换产生（a,b）区间上的随机数。

* 1. **正态分布初始化**

正态分布的概率密度函数为：

通常用N（,）表示。式中是均值，是方差。

正态分布的随机数：设为（0，1）上n个相互独立的均匀分布的随机数，由于，，根据中心极限定理可知，当n充分大时

的分布近似正态分布N（0，1）。通常取n=12，此时有

最后，再通过变换，便可得到均值、方差为的正态分布随机数。

* 1. **Xavier初始化和kaiming初始化**

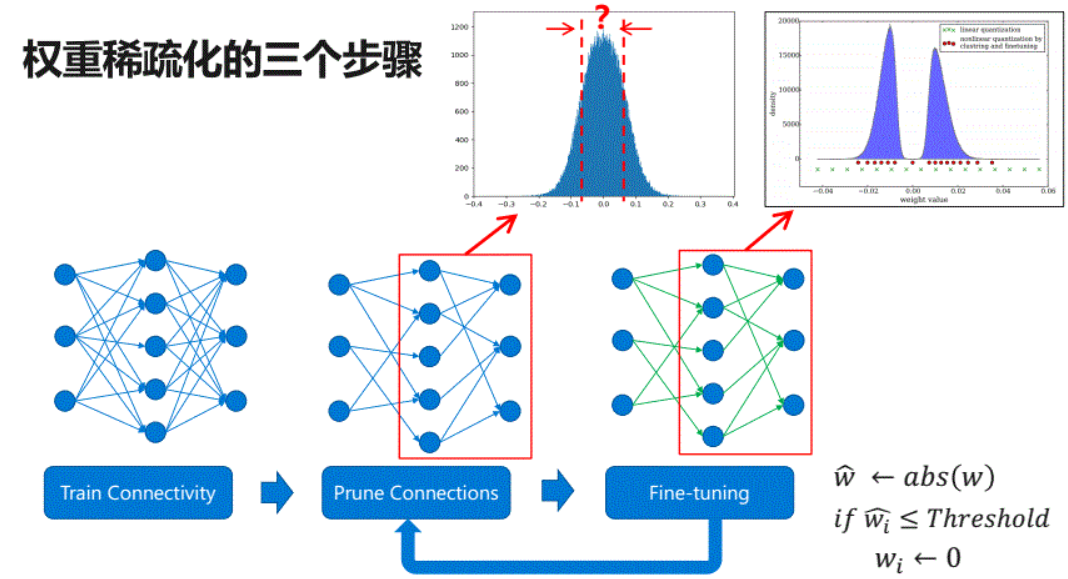
表示输入神经元的数量，表示输出神经元的数量。

|  |  |  |  |
| --- | --- | --- | --- |
| 初始化方法 | 激活函数 | 均匀分布[-r,r] | 正太分布N（0，） |
| Xavier初始化 | Logistic |  |  |
| Xavier初始化 | Tanh |  |  |
| kaiming初始化 | ReLu |  |  |

1. **稀疏化**
   1. **权重稀疏化（权重剪枝）**

从剪枝的粒度来划分，可以分为结构化剪枝和非结构化剪枝，2个剪枝结构方法。以下为4种具体的剪枝方法：

**细粒度剪枝**(fine-grained)：剪枝的操作单元是单个权重本身，是粒度最小的剪枝。属于对全连接权重的剪枝。（流程图示如下）



**向量剪枝**(vector-level)：它相对于细粒度剪枝粒度稍大，属于对卷积核内部(intra-kernel)的剪枝。

**核剪枝**(kernel-level)：即去除某个卷积核，丢弃对输入通道中对应卷积核的计算。

**滤波器剪枝**(Filter-level)：对整个卷积核组进行剪枝，推理过程中输出特征通道数会改变。

细粒度剪枝、向量剪枝、核剪枝在参数量与模型性能之间取得了一定的平衡，但是网络模型单层的神经元之间的组合结构发生了变化，需要专门的算法或者硬件结构来支持稀疏的运算，这种叫做结构化剪枝（Unstructured Pruning）。

滤波器剪枝（Filter-level）主要改变网络中的滤波器组和特征通道数目，所获得的模型不需要专门的算法和硬件就能够运行，被称为结构化剪枝（Structured Pruning）。

1. **随机数判断标准**

目前仅常用的随机性测试工具就有200多个，但每一个测试工具也仅仅可以代表随机性的一个方面，绝对无法代表随机性的所有方面。即使由任何由有限个检验组成的子集也无法代表，所以不管是谁，都无法设计出一个检验工具，只要某个随机序列通过了这个工具的检验就可以说这个序列是随机的了。

目前有许多随机性测试工具软件，比较常用的如澳大利亚昆士兰理工大学信息安全研究中心设计的**CryPt-XS**随机性测试工具;美国佛罗里达州州立大学设计的**DIEHARD**随机性测试工具;美国国家标准技术研究所(NIST)推出的**FIPS PUB 140-1、140-2**随机性测试工具等等。

**SP800-22 标准（NIST提供的测试包）15个大项**有些测试项被分成了多个小的子测试项，每个测试项的结果中均有P-value值和通过率Propotion两个指标。示例：



NIST SP 800-22随机数测试指南：

<https://developers.goodix.com/zh/bbs/blog_detail/76c7e154cbfb4b18b8a00c67debc5eb6>

1. **简单随机算法测试**

线性反馈移位寄存器（LFSR）一种移位寄存器电路：

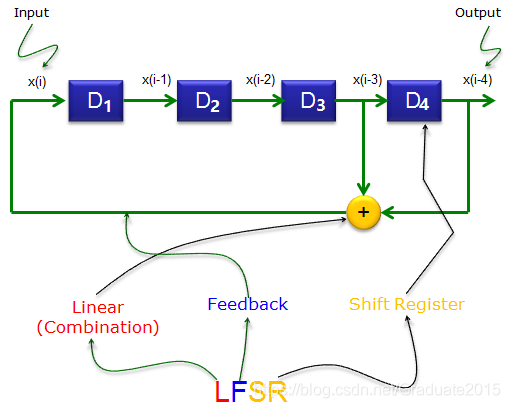


图 示例

LFSR是最著名的伪随机数发生器之一，具有非常广泛的应用。LFSR效率高，产生序列的随机性能良好，而且m序列的周期可以计算。

原理：<https://blog.csdn.net/Graduate2015/article/details/113133919>

**Mersenne Twister（梅森旋转）算法**：Mersenne Twister算法是利用线性反馈移位寄存器(LFSR)产生随机数的。

Mersenne Twister有以下优点：**随机性好，在计算机上容易实现，占用内存较少**(mt19937的C程式码执行仅需624个字的工作区域)，与其它已使用的伪随机数发生器相比，产生随机数的速度快、周期长，可达到2^19937-1，且具有623维均匀分布的性质，对于一般的应用来说，足够大了，序列关联比较小，**能通过很多随机性测试**。

算法详细：<https://wiki2.org/en/Mersenne_Twister>

Python 实现：

1. Index = 624
2. MT = [0]\*index
3. # MT[0] ->seed
5. **def** \_int32(t):
6. **return**(0xFFFFFFFF & t) #取最后32位->t
8. **def** twister():
9. **global** index
10. **for** i **in** range(624):
11. y = \_int32((MT[i] & 0x80000000) +(MT[(i + 1) % 624] & 0x7fffffff))
12. MT[i] = MT[(i + 397) % 624] ^ y >> 1
13. **if** y % 2 != 0:
14. MT[i] = MT[i] ^ 0x9908b0df
15. index = 0
17. **def** exnum():
18. **global** index
19. **if** index >= 624:
20. twister()
21. y = MT[index]
22. y = y ^ y >> 11
23. y = y ^ y << 7 & 2636928640
24. y = y ^ y << 15 & 4022730752
25. y = y ^ y >> 18
26. index = index + 1
27. **return** \_int32(y)
29. **def** mainset(seed):
30. MT[0] = seed #seed
31. **for** i **in** range(1,624):
32. MT[i] = \_int32(1812433253 \* (MT[i - 1] ^ MT[i - 1] >> 30) + i)
33. **return** exnum()
35. **def** main():
36. num = mainset(int(time())) / (2\*\*32-1)
37. **print**("产生的随机数为：",num)
38. main()