

```
bool b;
```

```
uint64_
b = bit
```

根据中心极限定理，样本均值（也就是前面增加的系数1/...

--lyconghk

2. Re—文搞懂交叉熵损失

写得很好，花书里面第五章那个把训练数据极大似然最大和负数对数的期望那个式子困扰了我很久，博主的文章帮我解释了这一疑惑。上面文章有一句话的描述是不是有问题，“上式的最大化 $\theta\_ML$ 是和没有训练样本没有...

--lyconghk

3. Re-回归损失函数1: L1 loss, L2 loss以及Smooth L1 Loss的对比

我为了点推荐，注册了，结果提示新用户注册一天之内不让我推荐，离谱

--不可而为

4. Re—文搞懂交叉熵损失

写的很透彻啊

--wzz12135

5. Re:Ffmpeg学习6: 音视频同步

想问一下博主ffmpeg的版本是多少？

--13467

```
    b = bitStream.getBit();
}

std::bitset<64> bits;
bits.reset();
for (int i = 0; i < k; i++)
{
    b = bitStream.getBit();
    bits.set(i, b);
}

N = unary * m + bits.to_ulong();
```

Exponential Golomb 指数哥伦布编码

Rice的编码方式和Golomb的方法是大同小异的，只是选择m必须为2的次幂。而Exp-Golomb则有了一个很大的改进，不再使用固定大小的分组，而使组的大小呈指数增长。如下图：



Exp-Golomb的码元结构是：\*\* [M zeros prefix] [1] [Offset] \*\*，其中M是分组的编号GroupID，1可以看作是分隔符，Offset是组内的偏移量。

Exp-Golomb需要一个非负整数K作为参数，称之为K阶Exp-Golomb。其中当K = 0时，称为0阶Exp-Golomb，目前比较流行的H.264视频编码标准中使用的就是0阶的Exp-Golomb，并且可以将任意的阶数K转为0阶Exp-Golomb编码。

首先来看下0阶Exp-Golomb编码，如下图：

n	code	Group ID
0	0	0
1	100	1
2	101	
3	11000	2
4	11001	
5	11010	
6	11011	
7	1110000	3
8	1110001	
9	1110010	
10	1110011	
11	1110100	
12	1110101	
13	1110110	
14	1110111	

上图是0阶Exp-Golomb编码的前几个组的分组情况，可以看出编号为m的组，其组内的最小元素的值是 $2^{m-1}$ ，也就是说对于非负整数N，其在编号为m的组内的充要条件是： $2^{m-1} \leq N < 2^m$ 。所以可以由如下公式计算得到组号m以及组内的偏移量Offset

$$m = \lfloor \log_2(N+1) \rfloor$$

$$Offset = N - 2^{m-1} + 1$$

有了组号以及组内的偏移量后，其编码就比较简单了，具体过程如下：

- 首先使用公式计算组号m， $m = \lfloor \log_2(N+1) \rfloor$
- 对组号m进行编码，连续写入m个0，最后写入一个1作为结束。
- 计算组内偏移量offset， $offset = N - 2^{m-1} + 1$
- 取offset二进制形式的低m位作为offset码元

0阶Exp-Golomb的编码后的长度是： $2^m + 1$ ，其解码过程和上面的Rice码类似，读入bit流，是0则继续，1则停止，然后统计0的个数m；接着读入m位的bit，就是offset，最后解码后的数值是： $N = 2^{m-1} + offset - 1$ 。

k阶Exp-Golomb

前面提到任意的k阶Exp-Golomb可以转换为0阶Exp-Golomb进行求解，这是为何呢。Exp-Golomb的组的大小实际上是呈2的指数增长，不同的参数k，实际控制的是起始分组的大小，具体是什么意思呢。

- k = 0,其组的大小为1, 2, 4, 8, 16, 32, ...
- k = 1,其组的大小为2, 4, 8, 16, 32, 64, ...
- k = 2,其组的大小为4, 8, 16, 32, 64, ...
- ...
- k = n,其组的大小为 $2^n, 2^{n+1}, \dots$

不同的k造成了其起始分组的大小不同，所以对于任意的k阶Exp-Golomb编码都可以转化为0阶，具体如下：设待编码数字为N，参数为k

- 使用0阶Exp-Golomb编码  $N + 2^{k-1} - 1$
- 从第一步的结果中删除掉高位的k个0

以上的算法描述来自：[https://en.wikipedia.org/wiki/Exponential-Golomb\\_coding](https://en.wikipedia.org/wiki/Exponential-Golomb_coding)

在搜索得到中文资料中，对于K阶Exp-Golomb的算法描述大多如下：

- 将num以二进制的形式表示（若不足k位，则在高位补0），去掉其低k位（若刚好是k位，则为0）得到数字n
- 计算n + 1的最低有效位数lsb，则M = lsb - 1。就是prefix中0的个数
- 将第1步中去掉的k位二进制串放到(n + 1)的低位，得到[1][INFO]

其实现以及描述都不如wikipedia，故在下面的实现部分使用的是Wikipedia的方法。在资料搜索的过程中，对于Exp-Golomb算法描述不止上述的两种，还有其他的形式，但都是殊途同归，也许得到的编码是不一样的，但是其**编码的长度**是一样的，也就没有过多的计较。最后附上k = 0,1,2,3时前29个数字的编码：

Exp-Golomb-k coding examples														
x	k=0	k=1	k=2	k=3	x	k=0	k=1	k=2	k=3	x	k=0	k=1	k=2	k=3
0	1	10	100	1000	10	0001011	001100	01110	010010	20	000010101	00010110	0011000	011100
1	010	11	101	1001	11	0001100	001101	01111	010011	21	000010110	00010111	0011001	011101
2	011	0100	110	1010	12	0001101	001110	0010000	010100	22	000010111	00011000	0011010	011110
3	00100	0101	111	1011	13	0001110	001111	0010001	010101	23	000011000	00011001	0011011	011111
4	00101	0110	01000	1100	14	0001111	00010000	0010010	010110	24	000011001	00011010	0011100	00100000
5	00110	0111	01001	1101	15	000010000	00010001	0010011	010111	25	000011010	00011011	0011101	00100001
6	00111	001000	01010	1110	16	000010001	00010010	0010100	011000	26	000011011	00011100	0011110	00100010
7	0001000	001001	01011	1111	17	000010010	00010011	0010101	011001	27	000011100	00011101	0011111	00100011
8	0001001	001010	01100	010000	18	000010011	00010100	0010110	011010	28	000011101	00011110	000100000	00100100
9	0001010	001011	01101	010001	19	000010100	00010101	0010111	011011	29	000011110	00011111	000100001	00100101

注意1之前的0的个数就是该数字所在的组的编号，同一组内的编码长度是相同的。

实现

通过上面的描述可以发现，Golomb编码的实现是很简单的，唯一的难点在于bit的操作。编码过程是将bit进行操作，然后拼凑为byte，写入buffer；解码则是相反的过程，读取byte转化为bit stream，操作一个个的bit。具体来说就是以下两个功能：

- 将bit流转换为byte数组
- 将byte数组转换为bit流

而在C/C++中最小的数据类型也是8位的byte，这就造成了对bit的操作有一定的难度，好在C++中std::bitset结构能够在一定程度上简化对bit的操作。

BitBuffer / ByteBuffer

首先实现一个“底层”的库，实现bit流和byte之间的转换。在Golomb编码中，对bit和byte的操作只需要简单的get/put操作，因此封装了两个结构体BitBuffer和ByteBuffer，具体的声明如下：

```
//
//
// Bit buffer
```

- BitBuffer是一个bit的缓存，无论是将bit流转换为byte还是将byte转换为bit流，都将bit放在此结构体中进行缓存。
- ByteBuffer用来管理byte数组的缓存

## BitStream

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// Bit Output Stream
//  将bit stream转化为byte数组
//  这里也仅提供辅助功能。至于byte流存满的处理放到编码器中处理
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
class BitOutputStream
{
    ,
public:
    // 写入一个bit
    void putBit(bool b);

    // 写入多个相同的bit
    void putBit(bool b, int num);

    // 设置数据组
    void setBuffer(uint8_t *buffer, int len);
    void resetBuffer();

    /*
    判断byte buffer中是可用的bit长度
    */
    uint64_t freeLength();

    // Flush bit buffer to byte buffer
    bool flush();

    uint64_t getTotalCodeLength()
    {
        return bytes.pos;
    }

private:
    BitBuffer bits;
    ByteBuffer bytes;
};

class BitInputStream
{
public:
    // 读取一个bit
    bool getBit();

    // 设置byte buffer
    void setBuffer(uint8_t *buffer, int len);

    BufferState check();
};

private:
    BitBuffer bits;
    ByteBuffer bytes;
};

```

- 将BitBuffer中缓存的bit刷新到byte数组中
- 写入编码的编码终止符。编码终止符在解码过程中是一个很重要的判断标志，这里假定Golomb编码后码元的最大长度为64位，所以可设编码终止符为：**连续64bits的0**。在解码时，要判断接下来的是不是编码终止符。
- 将编码后输出的字节数填充为8 (8 bytes, 64 bits) 的倍数，在解码时以8 bytes为单位进行解码，并且每次判断是不是编码终止符时也需要至少8 bytes。

**编码**

```
bool GolombEncoder::encode(uint64_t num)
{
    uint64_t q = num >> k;
    uint64_t r = num & (m - 1);

    auto len = q + 1 + k; // 编码后码元的长度

    /*
    不会判断缓存是否已满，直接往里面放，不足的码缓存在bit buffer中
    */
    bitStream.putBit(1, q);
    bitStream.putBit(0);

    for (int i = 0; i < k; i++)
    {

```

```
        bitStream.setBit(static_cast<bool>(r & 0x01));
        r >>= 1;
    }

    return bitStream.freeLength() >= len; // 空间足够,存放编码后的码元则返回true,否则返回false
}
}
```

上述代码以Golomb-Rice编码为例。在putBit时候的不会判断缓存是否够用，直接存放，如果Byte Buffer不足以存放本次编码的bits，则将Byte Buffer填满后，余下的bits在BitBuffer中缓存，然后返回false，告诉调用者byte buffer已经填满，可以处理当前buffer的数据后调用resetBuffer后继续编码；也可以直接更换一个新的byte buffer。

解码

在每次解码前，先要调用check方法来判断byte buffer的状态，byte buffer中有以下几种状态

- 空，数据已读取完
- 编码终止符，buffer中的数据是编码终止符，解码结束
- 数据不足，buffer中的数据不足以完成本次解码，需要读取新的buffer
- 数据足够，继续解码

check的实现如下:

```
enum BufferState
{
    BUFFER_EMPTY, // buffer empty
    BUFFER_END_SYMBOL, // end_symbol 编码的终止符, 已经没有编码的数据
    BUFFER_LACK, // buffer数据不足以完成解码, 需要新的buffer
    BUFFER_ENOUGH // 数据足够, 继续解码
};

// 检测buffer的状态
// 在每次解码开始前调用
BufferState BitInputStream::check()
{
    // buffer中已无数据
    if (bits.count <= 0 && bytes.pos >= bytes.length)
        return BufferState::BUFFER_EMPTY;

    // buffer中还有数据. 分为两种情况: 不足64bits和有64bits
    auto count = (bytes.length - bytes.pos) * 8 + bits.count;

    // buffer中的数据足够64位
    if (count >= 64)
    {
        // bit buffer中数据就有64bits
        if (bits.count >= 64)
        {
            if (bits.data.none()) // 64 bits 0
                return BufferState::BUFFER_END_SYMBOL; // 编码终止符
            else
                return BufferState::BUFFER_ENOUGH; // 数据足够继续解码
        }
        // bit buffer中的数据不足64bit
        else
        {
            if (bits.data.none())
                return BufferState::BUFFER_ENOUGH;

            int count = ((64 - bits.count) / 8 + 1);
            int index = 0;
            while (index < count)
            {
                auto b = bytes.data[bytes.pos + index];
                index++;
                if (b != 0)
                    return BufferState::BUFFER_ENOUGH;
            }
            return BUFFER_END_SYMBOL;
        }
    }
    // buffer中数据不足64位. 不进行解码.
    // 将byte buffer中的数据取出放在bit buffer后. 返回BUFFER_LACK
    else
    {
        while (bytes.pos < bytes.length)
        {
            auto b = bytes.getByte();
            bits.putByte(b);
        }
        return BufferState::BUFFER_LACK;
    }
}
```

check的过程有些复杂，但代码中的注释已足够清晰，这里就不再详述了。

Golomb-Rice的解码过程如下:

```
////////////////////////////////////
//
// 解码
// 在每次解码前需要check buffer的状态. 根据不同的状态决定解码是否继续
//
////////////////////////////////////
BufferState GolombDecoder::decode(uint64_t& num)
{
    auto state = bitStream.check();

    // buffer中数据足够. 进行解码
    if (state == BufferState::BUFFER_ENOUGH)
    {
        bool b;
        uint64_t unary = 0;
        b = bitStream.getBit();
        while (b)
        {
            unary++;
            b = bitStream.getBit();
        }

        std::bitset<64> bits;
        bits.reset();
        for (int i = 0; i < k; i++)
        {
            b = bitStream.getBit();
            bits.set(i, b);
        }

        num = unary * m + bits.to_ulong();
    }

    return state;
}
```

解码完成后会返回当前byte buffer的状态，

- 状态是BUFFER\_END\_SYMBOL，则解码过程已经完成
- 状态是BUFFER\_EMPTY，byte buffer没有设置
- 状态是BUFFER\_LACK，byte buffer中的数据不足以完成一次解码，需要读入新的数据
- 状态是BUFFER\_ENOUGH，byte buffer中的数据足够，继续下一次的解码

测试

仍然以Golomb-Rice编码为例，测试代码如下

```
GolombEncoder encoder(m);
encoder.setBuffer(buffer, 1024);

ofstream ofs;
ofs.open("golomb_gi", ios::binary);
```

```
for (int i = 0; i < <length; i++)
{
    auto b = encoder.encode(nums[i]);
    if ( !b)
    {
        cout << "Lack of buffer space,write the data to file" << endl;
        cout << "reset buffer" << endl;
        ofs.write((const char*)buffer, encoder.getToalCodeLength());

        encoder.resetBuffer();
        break;
    }
}
encoder.close();

ofs.write((const char*)buffer, encoder.getToalCodeLength());

ofs.close();

cout << "Golomb finished coding" << endl;
```

- 实例编码器时，需要设定编码的参数m和以及存放编码后数据的buffer；
- 编码时，判断编码的的返回值，如果为true则继续编码，为false则buffer已满，将buffer写入文件后，resetBuffer继续编码。
- 编码结束后，调用close方法，写入编码终止符，并将整个编码后的数据填充为8的倍数。

下面代码Golomb-Rice的解码调用过程

```
ifstream ifs;
ifs.open("golomb.gl", ios::binary);

memset(buffer, 0, 1024);

ifs.read((char*)buffer, 664);

ofstream encodeOfs;
encodeOfs.open("encode.txt");

GolombDecoder decoder(m);
decoder.setBuffer(buffer, 1024);
uint64_t num;
auto state = decoder.decode(num);

int index = 0;
while (state != BufferState::BUFFER_END_SYMBOL)
{
    encodeOfs << num << endl;
    state = decoder.decode(num);

    index++;
}

ifs.close();
encodeOfs.close();

cout << "decode finished" << endl;
```

编码是也需要根据返回的状态，来处理byte buffer，在上面已详述。

## 总结

终于完成了这篇博文，本文主要对Golomb编码进行了一个比较详尽的描述，包括Golomb编码的两个变种：Golomb-Rice和Exp-Golomb。在编码实现部分，难点有三个：

- byte数组和bit流之间的转换
- 需要一个唯一的编码终止符
- 解码时，byte buffer中剩余数据不足以完成一次解码

针对上述问题，做了如下工作：

- 实现了一个简单的BitStream库，能够方便在bit流和byte数组之间进行转换
- 对编码后的码元长度做了一个假设，其最长长度不会超过64位，这样就使用64比特的0作为编码的终止符
- 在编码的时，会将编码后的总字节数填充为8的倍数，解码的过程中就以8字节为单位进行，当Byte buffer中的数据不足8字节时，可以判定当前buffer中的数据并不是全部的数据，需要继续读入数据已完成解码

本文所使用的源代码，

- Github <https://github.com/brookicv/GolombCode>
- CSDN <http://download.csdn.net/detail/brookicv/9740838>

2017年的第一篇博文，完。

如果您觉得阅读本文对您有帮助，请点一下“**推荐**”按钮。您的“**推荐**”将是我最大的写作动力！欢迎各位转载，但是未经作者本人同意，转载文章之后**必须在文章页面明显位置给出作者和原文连接**，否则保留追究法律责任的权利。

分类: 05-图像处理



- 上一篇: [生成特定分贝的音频波形](#)
- 下一篇: [图像处理基础\(1\): 噪声的添加和过滤](#)

posted @ 2017-01-18 18:29 Brook\_icv 阅读(23866) 评论(3) 编辑 收藏 举报

弹尽粮绝，会员救园：会员上线，命悬一线

[刷新评论](#) [刷新页面](#) [返回顶部](#)

登录后才能查看或发表评论，立即 [登录](#) 或者 [逛逛](#) 博客园首页

【推荐】阿里云-云服务器省钱攻略：五种权益，限时发放，不容错过

编辑推荐：

- [「WPF」使用 HLSL 实现百叶窗动效](#)
- [程序员的产品思维](#)
- [如何正确实现一个自定义 Exception](#)
- [dotnet SemanticKernel 入门：自定义变量和技能](#)
- [浅谈基于 QT 的截图工具的设计与实现](#)

阅读排行：

- [【故障公告】今年第五次：数据库服务器 CPU 100%](#)
- [基于ASP.NET ZERO，开发saaS版供应链管理系統](#)
- [\[WPF\]使用HLSL实现百叶窗动效](#)
- [升讯威在线客服系统的开发高性能数据处理技术：高性能TCP服务器技术](#)
- [MQTT vs. XMPP，哪一个才是IoT通讯协议的正解](#)

### 本文目录

- [Golomb编码的基本原理](#)
- [Golomb-Rice 编码](#)
- [Exponential Golomb](#)
- [k阶Exp-Golomb](#)
- [实现](#)
- [BitBuffer / ByteStream](#)
- [BitStream](#)
- [编码/解码](#)
- [编码](#)
- [解码](#)
- [测试](#)
- [总结](#)

☆ 关注一下博主吧

11

推荐

1

反对