华中农业大学
HUAZHONG AGRICULTURAL UNIVERSITY

Ch2:Data Representation

第二章 数据的机器级表示

2.1 数值数据的表示

- 定点数的表示
  - 进位计数制
  - 定点数的二进制编码
    - 原码、补码、移码表示
  - 定点整数的表示
    - 无符号整数、带符号整数
- 浮点数格式和表示范围
  - 浮点数的规格化
  - IEEE754浮点数标准
- C语言程序中的整数类型、浮点数类型

引例： - 5 62.547 D

1-数的符号如何表示？

2-数值大小如何表示？

3-小数点如何处理？

4-数据长度？

科学计数法(Scientific Notation)与浮点数

Example:

mantissa （尾数） *exponent（阶码、指数）*

6.02×10^{21}

decimal point （小数点） *radix （base, 基）*

- Normalized form（规格化形式）：小数点前只有一位非0数
- 同一个数有多种表示形式。例：对于数 1/1,000,000,000
  - Normalized（唯一的规格化形式）: 1.0×10^{-9}
  - Unnormalized（非规格化形式不唯一）：0.1×10^{-8}, 10.0×10^{-10}

……… for Binary Numbers?

4

科学计数法(Scientific Notation)与浮点数

for Binary Numbers:

mantissa（尾数） exponent（指数）

$0.101_{two} \times 2^{-10}$

binary point 基为2

只要对尾数和指数分别编码，就可表示一个浮点数（即：实数）

浮点（Floating Point）表示法：将一个数的*有效数字*和*数的范围*在一个存储单元中分别予以表示。

小数点的位置根据需要而浮动，这就是浮点数。

$$N=M\times r^E$$

5

浮点数(Floating Point)的原理

原理：若用32位表示浮点数，格式可设计如下图。

+/-0.1xxxxx × 2^E

0	1　　8	9　　31
S	阶码E	尾数M

第0位数符S；

第1～8位为8位移码表示阶码E；

第9～31位为24位原码小数表示的尾数M。

格式本质上是约定，早期的计算机各自定义浮点数格式。

机器之间传送数据时带来麻烦。

“Father” of the IEEE 754 standard

1970年代后期， IEEE成立委员会着手制定浮点数标准

1985年完成浮点数标准IEEE 754的制定

现在所有计算机都采用IEEE 754来表示浮点数

UC Berkeley math professor William Kahan.

www.cs.berkeley.edu/~wkahan/
ieee754status/754story.html

Prof. William Kahan

7

IEEE 754 Floating Point Standard

	符号	指数	尾数
单精度	1 bit	8 bits	23 bits
双精度	1 bit	11 bits	52 bits

规格化数：+/-1.xxxxxxxxxx x $2^{Exponent}$

- 规格化尾数最高位总是1，所以隐含表示，省1位
- **1 + 23 bits （ single）， 1 + 52 bits （double）**

指数Exponent范围：

SP：(-126~127),偏移127

0000 0001 (-126) ~ 1111 1110 (127)

DP：(-1022~1023)偏移1023

0000 … 0001 (-1022) ~ 1111 … 1110 (1023)

十进制数与IEEE754标准之间的转换

例1： IEEE 754 单精度浮点数 BEE00000H的十进制？

1	0111 1101	110 0000 0000 0000 0000 0000

$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

- 符号s: 1 => 负数
- 阶码:
  - $0111\ 1101_{two} = 125_{ten}$
  - 偏移值调整计算: e=125 - 127 = -2
- 尾数: 1.75

故，十进制表示：$-1.75_{ten} \times 2^{-2} = -0.4375$

9

例2：十进制数-12.75对应的IEEE 754 单精度格式数？

1. Denormalize: -12.75
2. Convert integer part:
   $12 = 8 + 4 = 1100_2$
3. Convert fractional part:
   $.75 = .5 + .25 = .11_2$
4. Put parts together and normalize:
   $1100.11 = 1.10011 \times 2^3$
5. Convert exponent: $127 + 3 = 128 + 2 = 1000\ 0010_2$

1	1000 0010	100 1100 0000 0000 0000 0000

The Hex rep. is C14C0000H

浮点数的表示范围

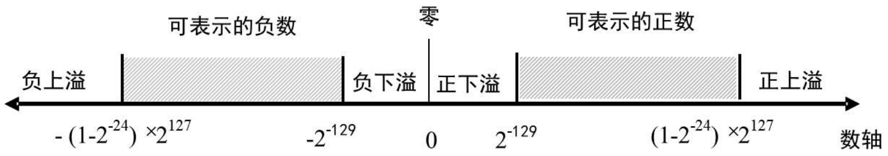
例：画出下述32位浮点数格式的规格化数的表示范围。

0	1 ... 8	9 ... 31
S	阶码E	尾数M

$+/-0.1xxxxx \times 2^E$

最大正数：$0.11\ldots1 \times 2^{11\ldots1} = (1-2^{-24}) \times 2^{127}$

最小正数：$0.10\ldots0 \times 2^{00\ldots0} = (1/2) \times 2^{-128}$



机器0：尾数为0 或 落在下溢区中的数

浮点数范围比定点数大，但数的个数没变多，故数之间更稀疏，且不均匀

11

Normalized numbers（规格化数）

前面的定义都是针对规格化数（normalized form）

How about other patterns?

Exponent	Significand	Object
1-254	**任意，隐含1**	**Norms**
0	**0**	**?**
0	**nonzero**	**?**
255	**0**	**?**
255	**nonzero**	**?**

0的表示

价码: all zeros
尾数: all zeros
符号位? 均有效

+0: 0 00000000 00000000000000000000000

-0: 1 00000000 00000000000000000000000

+∞/-∞ 的表示

In FP, 除数为0的结果是 +/- ∞, 不是溢出异常.（整数除0为异常）

如何表示+∞/-∞?

- 指数 ： 全1（11111111B = 255）
- 尾数： 全0

+∞ ： 0 11111111 00000000000000000000000

-∞ ： 1 11111111 00000000000000000000000

例：

5.0 / 0 = +∞, -5.0 / 0 = -∞

5+(+∞) = +∞, (+∞)+(+∞) = +∞

5 - (+∞) = -∞, (-∞) - (+∞) = -∞ etc

非数的表示（NaN）

Sqrt (– 4.0) = ?   0/0 = ?

NaN 的表示:

Exponent = 255

Significand: nonzero

如:

sqrt (-4.0) = NaN   0/0 = NaN

op (NaN,x) = NaN   +∞+(-∞) = NaN

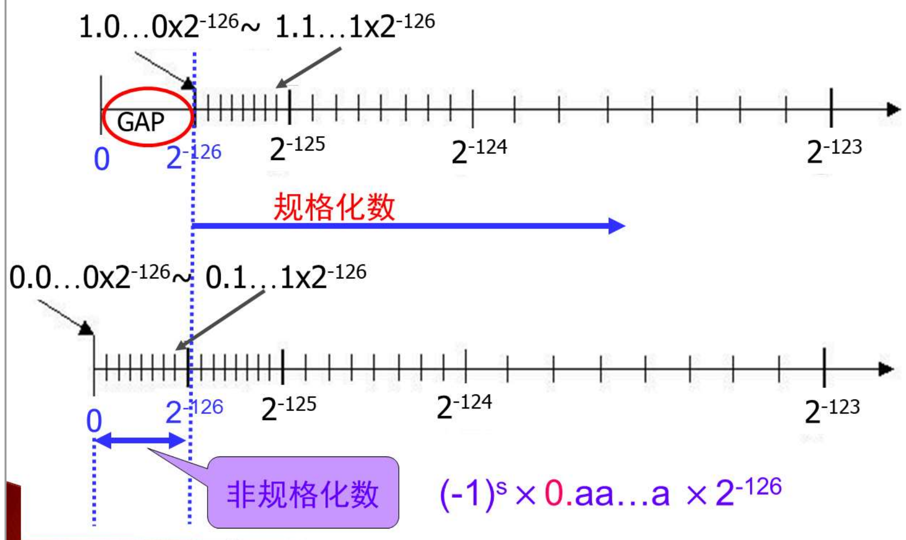
+∞- (+∞) = NaN   ∞/∞ = NaN

非规格化数的表示

指数	尾数	数
0	0	+/-0
0	非0	Denorms
1-254	任意 隐藏位1	规格化数
255	0	+/- 无穷
255	非0	非数

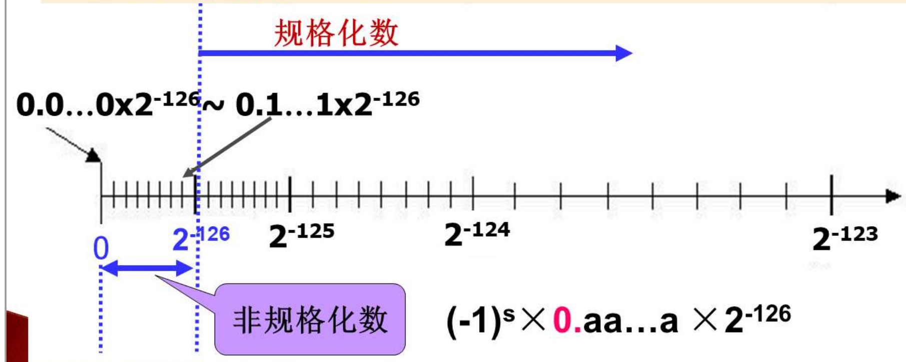
非规格化数

非规格化数的表示区域



浮点数在数轴的分布

- 浮点数范围比定点数大，但数的个数没变多，故数之间更稀疏不均匀。
- 指数将数轴分成不均匀的区间，最左是［2^{-126}，2^{-125}），向右依次扩大一倍，最右是[2^{127}，2^{128}）。
- 区间内部均匀分布2^{23}个数。



- 对应的十进制表示范围?

  单精度最大的数: $+1.11\ldots1\times 2^{127}$

  How about double?

约 $+3.4 \times 10^{38}$

约 $+1.8 \times 10^{308}$

小组讨论—C 语言中浮点数的精度

testsys — vi float.c — 74×35

```
//单精度浮点数的有效数字是十进制的7位
#include <stdio.h>
void main()
{
  float tem[10];
  float a=123456789;
  int* pTem;
  int i;
 printf("The size of float is %d\n",sizeof(float));

 pTem=(int*)tem;
 tem[0]=61.419996;
tem[1]=61.419997;
tem[2]=61.419998;
tem[3]=61.419999;
tem[4]=61.420000;
tem[5]=61.420001;
tem[6]=61.420002;
tem[7]=61.420003;
tem[8]=61.420004;
tem[9]=61.420005;

for(i=0;i<10;i++)
   printf("%.6f,0x%X\n",tem[i],*(pTem+i));
printf("%f\n",a);
return;
}
```

提问：

1、运行结果？

2、这十个数在机器中有几种表示？为什么？

计算机组成原理 信息学院 别丽华 20

浮点数的应用——类型转换问题

我们凭直觉一般认为：
浮点数转换为整数会损失精度，而整数转换为浮点数不会损失精度。 not always true!

P40 例2.25

```
int i;  float f;
if ( f == (float) ((int) f) )  {
        printf (“true”);
    }

    if ( i == (int) ((float) i) )  {
        printf (“true”);
    }
```

若f=1.3,转成int丢失小数

若i很大, i= $2^{31}-1$, 31>24,转成float只保留24位。

提问：**How about double?**

True，因31<53！

浮点数的应用——计算问题

◆浮点数加减运算（大数吃小数）

float $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, $z = 1.0$

$(x+y)+z = (-1.5\times10^{38}+1.5\times10^{38})+1.0 = 1.0$

$x+(y+z) = -1.5\times10^{38}+(1.5\times10^{38}+1.0) = 0.0$

为什么第二种方法计算不正确?

Any Questions?

十进制数的表示

- 数值数据（numerical data）的两种表示

Binary（二进制数）
- 定点整数：Fixed-point number（integer）
  - Unsigned and signed int
- 浮点数：Floating-point number（real number）

Decimal（十进制数）
- 用ASCII码表示
- 用BCD（Binary coded Decimal）码表示

计算机中为什么要用十进制数表示数值?
- 日常使用十进制数，所以计算机外部用十进制，内部用二进制。在极少数大数据输入/出的系统中，为减少二进制数和十进制数之间的转换，可在计算机内部直接用十进制数表示。

24

用ASCII码表示十进制数

- 前分隔数字串
  - 符号位单独用一个字节表示，位于数字串之前。
  - 正号用 “+”的ASCII码(2BH)表示；负号用 “-”的ASCII码(2DH)表示
  - 例：十进制数+236表示为: 2B 32 33 36H
    0010 1011 0011 0010 0011 0011 0011 0110B
    十进制数-2369表示为: 2D 32 33 36 39H
    0010 1101 0011 0010 0011 0011 0011 0110 0011 1001B
- 后嵌入数字串
  - 符号位嵌入最低位数字的ASCII码高4位中。比前分隔方式省一个字节。
  - 正数不变；负数高4位变为0111.
  - 例：十进制数+236表示为: 32 33 36H
    0011 0010 0011 0011 0011 0110B
    十进制数-2369表示为: 32 33 36 79H
    0011 0010 0011 0011 0011 0110 0111 1001B

缺点：占空间大，且需转换成二进制数或BCD码才能计算。

用BCD码表示十进制数

- 编码思想：每个十进数位用4位二进制表示。
- 编码方案
  1. 十进制有权码
     - 4个二进制位都有一个权。8421码是最常用的十进制有权码。也称自然BCD（NBCD）码。
  2. 十进制无权码
     - 各位没有确定的权。如格雷码(两相邻代码间只有一位数码不同)。
- 符号位的表示：
  - “+”：1100； “-”：1101
  - 例：$+236=(1100\ 0010\ 0011\ 0110)_{8421}$ （占2个字节）
    $-2369=(1101\ 0000\ 0010\ 0011\ 0110\ 1001)_{8421}$ （占3字节）

补0以使数占满一个字节

小 结

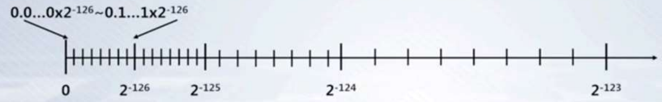
- 在机器内部编码后的数称为机器数，其值称为真值
- 定义数值数据有三个要素：进制、定点/浮点、编码
- 整数的表示
  - 无符号数：正整数，用来表示地址等；带符号整数：用补码表示
- C语言中的整数
  - 无符号数：unsigned int (short / long)；带符号数： int (short / long)
- 浮点数的表示
  - 符号；尾数：定点小数；指数（阶）：定点整数（基不用表示）
- 浮点数的范围
  - 正上溢、正下溢、负上溢、负下溢；与阶码的位数和基的大小有关
- 浮点数的精度：与尾数的位数和是否规格化有关
- 浮点数的表示（IEEE 754标准）：单精度SP（float）和双精度DP（double）
  - 规格化数(SP)：阶码1~254，尾数最高位隐含为1
  - “零” (阶为全0，尾为全0)
  - ∞ (阶为全1，尾为全0)
  - NaN (阶为全1，尾为非0)
  - 非规格化数 (阶为全0，尾为非0，隐藏位为0)（P.41）
- 十进制数的表示：用ASCII码或BCD码表示

第二章第一次作业：

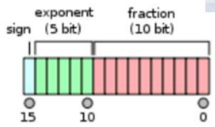
- 题4. 6. 7. 9. 11. 14.
- 复习消化书上例题
- 预习非数值信息的表示

扩展——半精度fp16（自学）

IEEE754单精度浮点数

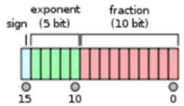


说明： 在实数轴上，同一个阶码范围内，能够表示2^{23}个不同的数。
所以并不是所有在表示范围内的数都可以准确表示。

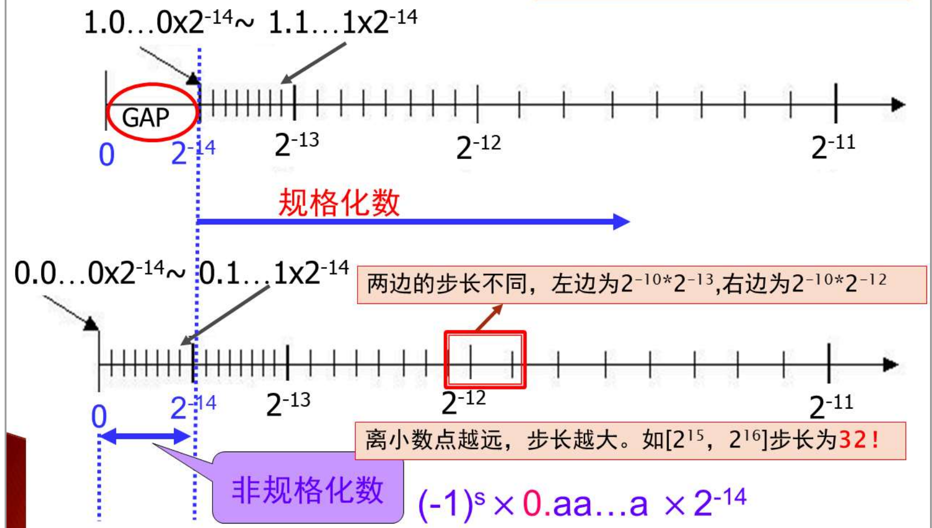


半精度可以表示的最大值：
0 11110 1111111111 计算方法为：
(-1) ^0×2^(30-15)×1.1111111111 = 1.1111111111(b)×2^15 = 1.9990234375(d)×2^15 = 65504
半精度可以表示的最小值（除了subnormal value）：
0 00001 0000000000 计算方法为：（-1）^（-1）×2(1-15)=2^(-14),约等于十进制的 6.104×10^(-5)

半精度数fp16的精度问题



指数范围E：0-32
规格化数E：1-31
偏移量：15
故规格化数指数e范围：[-14，16]



关于半精度f16

```
0 01111 0000000000 = 1
0 01111 0000000001 = 1 + 2-10 = 1.0009765625 (next smallest float after 1)
1 10000 0000000000 = -2

0 11110 1111111111 = 65504  (max half precision)

0 00001 0000000000 = 2-14 ≈ 6.10352 × 10-5 (minimum positive normal)
0 00000 1111111111 = 2-14 - 2-24 ≈ 6.09756 × 10-5 (maximum subnormal)
0 00000 0000000001 = 2-24 ≈ 5.96046 × 10-8 (minimum positive subnormal)

0 00000 0000000000 = 0
1 00000 0000000000 = -0

0 11111 0000000000 = infinity
1 11111 0000000000 = -infinity

0 01101 0101010101 = 0.333251953125 ≈ 1/3
```


f32到f16的一种特殊的转换

- NVidia在2002年提出了半精度浮点数，只使用2个字节16位，包括1位符号、5位指数和10位尾数，能表示的最大数值是 $(2-2^{-10}) \times 2^{15} = 65504$， 最小数值$2^{-14} \approx 6.10 \times 10^{-5}$ 。NVidia的方案已经被IEEE-754采纳。
- cuda的ptx中cvt是可以转换f32到f16的，不过如果没有这样的支持，在操作寄存器时，我们可以直接将32位寄存器的高16位mov到新寄存器的高或低16位中。
- 因为f32前16位中：1位符号、8位指数、7位尾数，再使用该数时，可以使用随机数或者0补全后面丢失的16位尾数。
- 在转换过程中会丢失精度，结果是否符合要求则根据应用程序不同有不同的标准。

Google的TensorFlow则比较简单粗暴，把单精度的后16位砍掉，也就是1位符号、8位指数和7位尾数。

测试半精度浮点数的使用效果

typedef unsigned short half;　　// 先定义unsigned short为half

按NVidia方案，**把单精度浮点数转成半精度**，可以这么做(0值的处理有问题！)：

```
half Float2Half(float m)
{
    unsigned long m2 = *(unsigned long*)(&m);    // 强制把float转为unsigned long
    // 截取后23位尾数，右移13位，剩余10位；符号位直接右移16位；
    // 指数位麻烦一些，截取指数的8位先右移13位(左边多出3位不管了)
    // 之前是0~255表示-127~128, 调整之后变成0~31表示-15~16
    // 因此要减去127-15=112(在左移10位的位置).
    unsigned short t = ((m2 & 0x007fffff) >> 13) | ((m2 & 0x80000000) >> 16)
        | (((m2 & 0x7f800000) >> 13) - (112 << 10));
    if(m2 & 0x1000)
        t++;                    // 四舍五入(尾数被截掉部分的最高位为1, 则尾数剩余部分+1)
    half h = *(half*)(&t);     // 强制转为half
    return h ;
}
```


从半精度转回单精度比较好办，按格式取出符号位、指数和尾数，再按定义计算，结果保存为float即可

```
float Half2Float(half n)
{
    unsigned short frac = (n & 0x3ff) | 0x400;
    int exp = ((n & 0x7c00) >> 10) - 25;
    float m;

    if(frac == 0 && exp == 0x1f)
        m = INFINITY;
    else if (frac || exp)
        m = frac * pow(2, exp);
    else
        m = 0;

    return (n & 0x8000) ? -m : m;
}
```


最后把实际数据从单精度转成半精度，再转回单精度，计算误差：

```
for(float n = 4e-5; n < 6e4; n *= 1.001) {
    printf("%f, %f,  %.4f\n", n, Half2Float(Float2Half(n)),
        ((double)n - Half2Float(Float2Half(n))) / n * 100.0);
}
```

实测最大误差0.048%(也就是1/2048)，平均绝对误差0.018%，似乎还不错。

Google TensorFlow的方案验证起来就非常简单了，砍掉后16位即可，四舍五入还是要的。

```
#include <stdio.h>

int main(void)
{
    for(float n = 1e-8; n < 1e8; n *= 1.001){
        unsigned long k, l;
        k = *(unsigned long*)(&n);
        l = k & 0xffff0000;
        if(k & 0x8000)
            l += 0x10000;         // 四舍五入
        float m = *(float*)(&l);

        printf("%f, %f, %f\n", n, m, (n - m) / n);
    }
    return 0;
}  //最大误差0.39%(也就是1/256)，平均绝对误差0.14%。许多场合其实主要关心的只是数量级，用这个也不错。
```

TensorFlow使用[illegible]brain float16（BF16），符号指数尾数位分别占1、8、7位。Tensor Float32（TF32）对应的符号指数尾数分别占1、8、10位，精度高于BF16

关于fp16的溢出和舍入误差

- fp16 的有效的动态范围约为 （ 2^{-24}～65504)，比单精度的float要狭窄很多。对于深度学习而言，最大的问题在于 Underflow（下溢出），在训练后期，例如激活函数的梯度会非常小， 甚至在梯度乘以学习率后，值会更加小。

何为舍入误差：

OK　FP16 weight = 2^{-3} (0.125)

OK　FP16 gradient = 2^{-14} (约等于0.000061)

FP16 weight = weight + gradient

Error　$= 2^{-3} + 2^{-14}$

$= 2^{-3}$

舍入错误（Rounding Error）：

$[2^{-3}, 2^{-2}]$间，FP16表示的固定间隔为2^{-13}

即比2^{-3}大的下一个数为$2^{-3} + 2^{-13}$

半精度FP16舍入误差的例子，来自引用[2]

fp16 各个区间的最小gap：

Precision limitations on decimal values in [0, 1] [edit]

- Decimals between 2^{-24} (minimum positive subnormal) and 2^{-14} (maximum subnormal): fixed interval 2^{-24}
- Decimals between 2^{-14} (minimum positive normal) and 2^{-13}: fixed interval 2^{-24}
- Decimals between 2^{-13} and 2^{-12}: fixed interval 2^{-23}
- Decimals between 2^{-12} and 2^{-11}: fixed interval 2^{-22}
- Decimals between 2^{-11} and 2^{-10}: fixed interval 2^{-21}
- Decimals between 2^{-10} and 2^{-9}: fixed interval 2^{-20}
- Decimals between 2^{-9} and 2^{-8}: fixed interval 2^{-19}
- Decimals between 2^{-8} and 2^{-7}: fixed interval 2^{-18}
- Decimals between 2^{-7} and 2^{-6}: fixed interval 2^{-17}
- Decimals between 2^{-6} and 2^{-5}: fixed interval 2^{-16}
- Decimals between 2^{-5} and 2^{-4}: fixed interval 2^{-15}
- Decimals between 2^{-4} and 2^{-3}: fixed interval 2^{-14}
- Decimals between 2^{-3} and 2^{-2}: fixed interval 2^{-13}
- Decimals between 2^{-2} and 2^{-1}: fixed interval 2^{-12}
- Decimals between 2^{-1} and 2^{-0}: fixed interval 2^{-11}

解决办法

- FP32 权重备份

这种方法主要是用于解决舍入误差的问题。其主要思路，可以概括为：weights, activations, gradients 等数据在训练中都利用FP16来存储，同时拷贝一份FP32的weights，用于更新。 如图：

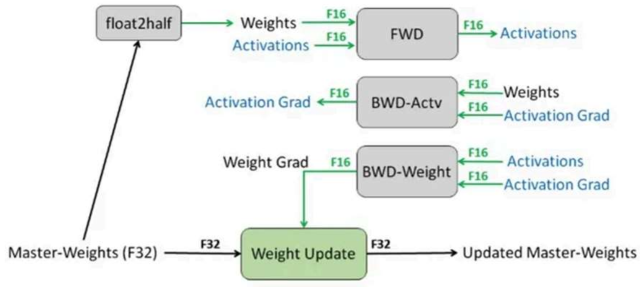
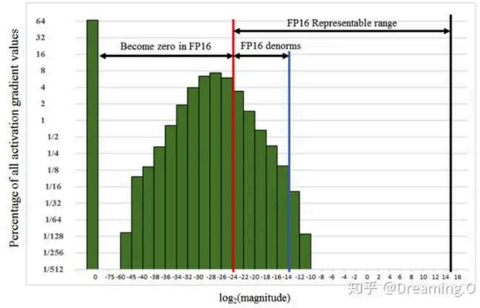


Figure 1: Mixed precision training iteration for a layer.

Loss Scale

- Loss Scale 主要是为了解决 fp16 underflow 的问题。刚才提到，训练到了后期，梯度（特别是激活函数平滑段的梯度）会特别小，fp16 表示容易产生 underflow 现象。下图展示了 SSD 模型在训练过程中，激活函数梯度的分布情况：可以看到，有67%的梯度小于 2^{-24} ，如果用 fp16 来表示，则这些梯度都会变成0。



C语言中的类型转换

例2.24 假定变量i、f、d 的类型分别是int、float 和double，它们可以取除+∞、–∞和NaN 以外的任意值。请判断下列每个C 语言关系表达式在32 位机器上运行时是否永真。

A. i = = (int) (float) i

B. f = = (float) (int) f

C. i = = (int) (double) i

D. f = = (float) (double) f

E. d = = (float) d

F. f = = –(–f)

G. (d+f) – d = = f

补充：C语言中字面量的比较

ISO C90标准下，在32位系统上以下C表达式的结果是什么?

-2147483648 < 2147483647

False!

为什么？

以下关系表达式结果是什么?

int i = -2147483648;

i < 2147483647

True !

1、ISO C90中如何处理字面量

2、编译器对比较运算的处理

编译器处理常量时默认的类型

- C90

范围	类型
$0\sim2^{31}-1$	int
$2^{31}\sim2^{32}-1$	unsigned int
$2^{32}\sim2^{63}-1$	long long
$2^{63}\sim2^{64}-1$	unsigned long long

$2147483648=2^{31}$

- C99

范围	类型
$0\sim2^{31}-1$	int
$2^{31}\sim2^{63}-1$	long long
$2^{63}\sim2^{64}-1$	unsigned long long

41

-2147483648 < 2147483647

无符号整型　　带符号整型

机器数：0x80000000

int i = -2147483648;

i < 2147483647　按照带符号整型比较

将2147483648转换为带符号整数后赋给变量i

机器数：0x80000000

真值：-2147483648

42