

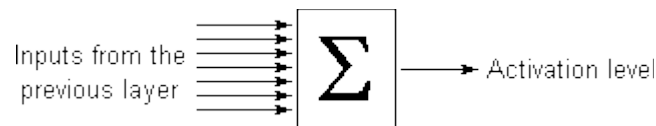
Web Site of Robert John Morton

Neural Networks: MLP Input Summation Function

[Outline](#), [precision](#), [speed](#), [loop optimisation](#), [testing](#), [conclusion](#), [final solution](#).

Outline

Each neuron in a multi-layer perceptron network receives a number of inputs. A neurone's summation function multiplies each of its inputs by a corresponding weight to form a weighted input. It then sums all the weighted inputs to produce a single output which is called the neurone's activation level:



This activation level for a 7-input neuron is calculated according to the formula:

$$\text{Activation Level} = \frac{1}{7} * \sum_{i=0}^{i=6} \text{input}_i * \text{weight}_i$$

The activation level is fed as the input to the neurone's Sigmoid Function to produce the neurone's output. See document file [sigmoid.htm](#).

Precision

To make our neural network as fast as possible, we will restrict all inputs and outputs to the bipolar range of a 16-bit interger, ± 32767 . We will also restrict the weights to this range. Suppose our neuron has 7 inputs as shown above. The full process for summing the 7 inputs and producing an activation-level output therefore will be:

$$\text{Activation Level} = \frac{1}{7 * 32768} * \sum_{i=0}^{i=6} \text{input}_i * \text{weight}_i$$

Two 16-bit integers produce a 32-bit product. Furthermore, the sum of the 32-bit products for a large number of inputs could be a number requiring far more than 32-bits. Suppose the inputs to our neuron are stored in an array `int I[]` where `int` is a 16-bit signed quantity. Suppose also that the corresponding weights are to be found in an array `int W[]`. The easy way out in computing the weighted sum while preserving the original precision would be therefore to use a floating-point double variable as the sum accumulator as follows:

```
int i;
double x;
for(x = 0, i = 0, i < 7; i++)
x += (double)*(I + i) * *(W + i);
x /= 7 * 32768;
```

The interger input value $*(I + i)$ must be typecast to a floating-point double before the multiply takes place so that the product is floating-point double precision. The weight will be typecast to a double automatically to match the forced typecasting of the input value. To scale the final output back to the input range of ± 32767 , we must, once the loop has terminated, divide by 32768 and by the number of inputs. This final value of the accumulator is then typecast automatically to a 16-bit signed interger when passed as input to the neurone's **int Sigmoid()** function.

Speed

Floating-point arithmetic is easy and trouble-free. But it is also very slow compared with interger arithmetic.

[In this document we shall assume that our neural network program will be running on a PC with an Intel 80286 16-bit CPU. The coding can easily be adapted to take advantage of longer word-lengths.]

We will find out the best form of 'C' coding for our fast interger version of our neural input summation program by investigating various methods as follows.

1) All-Interger Method

We will first try to compute the sum of the neurone's weighted inputs as follows:

```
int i, x;
for(x = 0, i = 0, i < 7; i++)
  x += *(I + i) * *(W + i);
x /= 7;
```

However, this truncates the 32-bit product immediately to 16-bit precision. Furthermore, the accumulator x could overflow with as few as two inputs.

2) Long Accumulator

We can prevent overflow in the accumulator x by defining it as a **long** (32-bit) variable. The summation loop would therefore become:

```
int i;
long x;
for(x = 0, i = 0, i < 7; i++)
  x += *(I + i) * *(W + i);
x /= 7 * 32768;
```

Notice that the final result, because it is 32-bits, has to be divided by 32768 as well as by the number of inputs in order to yield the final 16-bit 'activation level' we want. This final division could be made faster using a right-shift of 15:

```
x = (x >> 15) / 7;
```

Unfortunately, since we are adding together truncated products and then dividing the accumulated sum by the number of inputs, we are losing half the possible 32-bit precision of each input * weight product. The result will therefore not be a true 16-bit representation of the sum of the weighted inputs. We need to create and take account of the full 32-bit product.

3) Long Product

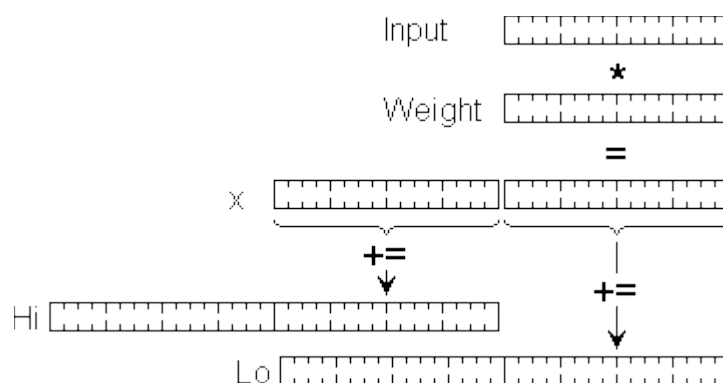
We can preserve the 32-bit precision of the product by typecasting one of the operands to a long as follows:

```
int i;
long x;
for(x = 0, i = 0, i < 7; i++)
    x = (long)*(I + i) * *(W + i);
x /= 7;
```

Since we typecast the input to a long, the weight is automatically typecast to a long before the multiplication is performed. We therefore get a 32-bit long product. However, we cannot simply add this long product to a long accumulator. It could overflow after only two inputs.

4) Split Long Accumulator

A 48-bit accumulator on the other hand could accommodate 32-bit input * weight products from 32k inputs. This is more than we are likely to need. We can effectively create a 48-bit accumulator by using two separate 32-bit **long** accumulators as follows:



We add the top half of the 32-bit product to the Hi accumulator and the bottom half to the Lo accumulator. We can write this in 'C' as follows:

```
int i, AL;
long x, Lo, Hi;
for(Hi = 0, Lo = 0, i = 0, i < 7; i++) {
```

```

    x = (long)*(I + i) * *(W + i);
    Hi += x >> 16;
    Lo += x & 0xFFFF;
}
AL = ((Hi << 1) + (Lo >> 15)) / 7;

```

Once the loop has terminated, the lower half of Lo is discarded. The upper half of Lo is added into Hi. Be aware that because x is right-shifted by 16 before it is added to Hi, Hi holds the sum of the products already divided by 65536. We therefore multiply it by 2 (left-shift it by 1). It has thus been effectively divided by 32768. We then divide Lo by 32768 by right-shifting it by 15. We then add the two results together and divide by the number of inputs. The long result is then assigned to the int variable AL.

This indeed works perfectly with only a binary rounding error on the least significant bit of the resulting 16-bit activation level, AL. Furthermore, it performs its task in one thirty-seventh of the time taken by the equivalent floating-point double computation.

5) In-line Assembler

Although it makes a small part of a program specific on a particular type of CPU chip, in-line assembler can in some cases add extra vital speed at the speed-critical heart of a program. For the small amount of re-coding needed to migrate to another processor, the speed advantage of in-line assembler can sometimes be worth it. But not always.

When multiplying two integers on the Intel 80x86 family of processors, you put one operand in the AX register and multiply by the other operand. The 32-bit product then appears in the two registers DX and AX. In theory this saves the overhead of typecasting (in effect both operands) to **long** before the multiplication is performed. It is coded as follows:

```

int i, ni, w, AL;    //AL = Activation Level
unsigned int q;
long    Lo, Hi;
for(Hi = 0, Lo = 0, i = 0, i < 7; i++) {
    ni = *(I + i);    //neural input
    w = *(W + i);    //weight
    _asm {
        mov ax, ni    ;load input into AX register
        imul w        ;signed multiply with weight
        mov ni, dx    ;put ms word into ni
        mov q, ax     ;put ls word into unsigned q
    }
    Hi += ni;        //add ms word into High accum
    Lo += q;        //add ls word into Low accum
}
AL = ((Hi << 1) + (Lo >> 15)) / 7;

```

Note that ordinary 'C' int variables can be used within in-line assembler.

The neural input ni is placed in the AX register and a signed multiply is then performed with the weight w.

The most-significant (the upper) half of the product is then copied into `ni`. Integer `ni` is used here because at this point it has finished serving its primary purpose of carrying the neural input, and using it for the upper half of the product saves defining a separate variable for this purpose.

The least-significant (the lower) half of the product is copied into `q`. This is an *unsigned* integer so that the bits in `AX` are copied literally without any complementing. This is because the arithmetic sign of the whole product is effectively conveyed within the upper half of the product in `ni`.

For all this, testing showed the in-line assembler version of the weighted input summation loop to be 41.6% slower than the split accumulator 'C' version. We must bear in mind that the compiler has to push registers on to the stack before dropping into in-line assembler and popping them off again afterwards. This extra chore obviously outweighs the small saving gained by being able to lift the two halves of the double-precision product directly from the `DX` and `AX` registers.

Loop Optimisation

The final way of making sure the code was as fast as possible was to try a different method of accessing the inputs and weights. Instead of adding the index `i` to the start addresses of the inputs and weights arrays, two pointers `pi` and `pw` were defined to point at the current input and current weight respectively and incremented each pass of the loop. Two versions were tried. The first increments the pointers in the **for** statement:

```
int i, AL, *pi = I, *pw = W;
long x, Lo = 0, Hi = 0;
for(i = 0, i < 7; i++, pi++, pw++) {
    x = (long) *pi * *pw;
    Hi += x >> 16;
    Lo += x & 0xFFFF;
}
AL = ((Hi << 1) + (Lo >> 15)) / 7;
```

The second version increments the pointers in the multiply statement:

```
int i, AL, *pi = I, *pw = W;
long x, Lo, Hi;
for(Hi = 0, Lo = 0, i = 0, i < 7; i++) {
    x = (long) *pi++ * *pw++;
    Hi += x >> 16;
    Lo += x & 0xFFFF;
}
AL = ((Hi << 1) + (Lo >> 15)) / 7;
```

Both were found to be 4.6% slower than the original so were abandoned.

Testing

Before deciding on Method 4 above as the optimum way of implementing the weighted inputs summation function, all the above methods were fully tested and exercised. Three types of test were performed. Each was done by embedding the code for each method in an exerciser program.


```

int LongAccum(void) { //int product with long accumulator
    int i;
    long x;
    for(x = 0, i = 0; i < NI; i++)
        x += *(I + i) * *(W + i);
    return(x / NI);
}

int LongProd(void) { //long product with long accumulator
    int i;
    long x;
    for(x = 0, i = 0; i < NI; i++)
        x += (long)*(I + i) * *(W + i);
    return((x >> 15) / NI);
}

int SplitLong(void) { //long product with 2 long accumulators
    int i;
    long x, Hi, Lo;
    for(Hi = 0, Lo = 0, i = 0; i < NI; i++) {
        x = (long)*(I + i) * *(W + i);
        Hi += x >> 16; //add upper half of long product to Hi
        Lo += x & 0xFFFF; //add lower half of long product to Lo
    }
    return(((Hi << 1) + (Lo >> 15)) / NI);
}

int SplitASM(void) { //in-line assembler version of above
    int i, ni, w;
    unsigned int q;
    long Hi, Lo;
    for(Hi = 0, Lo = 0, i = 0; i < NI; i++) {
        ni = *(I + i); //current input amplitude
        w = *(W + i); //corresponding input weight
        _asm {
            mov ax, ni //load integer input into AX register
            imul w //do signed mpy with interger weight
            mov ni, dx //put ms word into unsigned interger p
            mov q, ax //put ls word into unsigned variable q
        }
        Hi += ni; //add ms word into high accumulator
        Lo += q; //add ls word into low accumulator
    }
    return(((Hi << 1) + (Lo >> 15)) / NI);
}

void Test(int flag) {
    int c, AL1, AL2, AL3, AL4, AL5;
    char *r;
    double AL0;
    AL0 = Benchmark(); //double computation as benchmark
    AL1 = Interger(); //all-interger method
    AL2 = LongAccum(); //interger product with long accumulator
    AL3 = LongProd(); //long product with long accumulator
    AL4 = SplitLong(); //long product with 2 long accumulators
    AL5 = SplitASM(); //in-line assembler version of the above
    i += sprintf(s + i,
        " Benchmark using floating-point double %+9.2f\n"

```

```

    , AL0);
i += sprintf(s + i,
    " 1) int products with int accumulator      %+6d \n"
    , AL1);
i += sprintf(s + i,
    " 2) int products with long accumulator     %+6d \n"
    , AL2);
i += sprintf(s + i,
    " 3) long products with long accumulator    %+6d \n"
    , AL3);
i += sprintf(s + i,
    " 4) long products with 2 long accumulators %+6d \n"
    , AL4);
i += sprintf(s + i,
    " 5) In-line assembler version of the above %+6d\n\n"
    , AL5);
if(flag) sprintf(s + i, "\n");
for(r = s; (c = *r) > '\0'; r++)
    putc(c, fh);                //write it to file
}

main() {
    fh = fopen("results.txt", "a");    //open file for appending
    i = sprintf(s,
        "TEST RESULTS FOR THE 5 COMPUTATION METHODS\n\n"
    );
    i += sprintf(s + i, "For positive inputs...\n");
    Test(0);                          //test for positive inputs
    for(i = 0; i < NI; i++)            //reverse inputs
        I[i] = -I[i];
    i = sprintf(s, "For negative inputs...\n");
    Test(1);                          //test for negative inputs
    fclose(fh);                       //close the file
}

```

The results produced in the file [results.txt](#) by running the above exerciser ([wsum1.exe](#)) are as follows:

TEST RESULTS FOR THE 5 COMPUTATION METHODS

For positive inputs...

Benchmark using floating-point double	+13485.75
1) int products with int accumulator	+189
2) int products with long accumulator	-8321
3) long products with long accumulator	-132
4) long products with 2 long accumulators	+13485
5) In-line assembler version of the above	+13485

For negative inputs...

Benchmark using floating-point double	-13485.75
1) int products with int accumulator	-189
2) int products with long accumulator	+8321
3) long products with long accumulator	+132
4) long products with 2 long accumulators	-13485
5) In-line assembler version of the above	-13485

Clearly, methods 1, 2 and 3 do not work. Register overflow has obviously occurred.


```

double x, t = GetTime();
for(k = 0; k < 10000; k++) {
    for(x = 0, i = 0; i < NI; i++)
        x += (double)*(I + i) * *(W + i);
    x /= NI * 32768;
}
return(GetTime() - t);
}

double Lfun1(void) {
int k, i, a;
long x, Hi, Lo;
double t = GetTime();
for(k = 0; k < 10000; k++) {
    for(Hi = 0, Lo = 0, i = 0; i < NI; i++) {
        x = (long)*(I + i) * *(W + i);
        Lo += x & 0xFFFF;
        Hi += x >> 16;
    }
    a = ((Hi << 1) + (Lo >> 15)) / NI;
}
return(GetTime() - t);
}

double Lfun2(void) {
int k, i, a, *pi = I, *pw = W;
long x, Hi, Lo;
double t = GetTime();
for(k = 0; k < 10000; k++) {
    for(Hi = 0, Lo = 0, i = 0; i < NI; i++) {
        x = (long)*(pi + i) * *(pw + i);
        Lo += x & 0xFFFF;
        Hi += x >> 16;
    }
    a = ((Hi << 1) + (Lo >> 15)) / NI;
}
return(GetTime() - t);
}

double Lfun3(void) {
int k, i, a, *pi = I, *pw = W;
long x, Hi, Lo;
double t = GetTime();
for(k = 0; k < 10000; k++)
    for(Hi = 0, Lo = 0, i = 0; i < NI; i++) {
        x = (long)*pi++ * *pw++;
        Lo += x & 0xFFFF;
        Hi += x >> 16;
    }
    a = ((Hi << 1) + (Lo >> 15)) / NI;
}
return(GetTime() - t);
}

double Lfun4(void) {
int k, i, a, *pi = I, *pw = W;
long x, Hi, Lo;
double t = GetTime();
for (k = 0; k < 10000; k++) {

```

```

    for (Hi = 0, Lo = 0, i = 0; i < NI; i++, pi++, pw++) {
        x = (long)*pi * *pw;
        Lo += x & 0xFFFF;
        Hi += x >> 16;
    }
    a = ((Hi << 1) + (Lo >> 15)) / NI;
}
return(GetTime() - t);
}

double Afun(void) {
    int k, i, AL, ni, w;
    unsigned int q;
    long Hi, Lo;
    double t = GetTime();
    for (k = 0; k < 10000; k++) {
        for (Hi = 0, Lo = 0, i = 0; i < NI; i++) {
            ni = *(I + i);
            w = *(W + i);
            _asm {
                mov ax, ni    ;load integer input into AX register
                imul w       ;do signed multiply with interger weight
                mov ni, dx   ;put ms word into unsigned interger p
                mov q, ax    ;put ls word into unsigned variable q
            }
            Hi += ni;        //add ms word into High accumulator
            Lo += q;         //add ls word into Low accumulator
        }
        AL = ((Hi << 1) + (Lo >> 15)) / NI;
    }
    return(GetTime() - t);
}

main() {
    FILE *fh;
    char s[1024], *p = s;
    long o;
    int c, i;
    double T1, T2, T3, T4, T5, T6;
    for(T1 = GetTime() + 10; GetTime() < T1); //wait 10 seconds
    printf("\n\nTiming...\n");
    T1 = Dfun(); T2 = Lfun1(); T3 = Lfun2();
    T4 = Lfun3(); T5 = Lfun4(); T6 = Afun();

    i = sprintf(s, "TIME TRIALS AGAINST FLOATING-POINT \
DOUBLE BENCHMARK:\n");
    i += sprintf(s + i, "    Floating-point double    %6.2f \
secs (Benchmark)                \n", T1);
    i += sprintf(s + i, "    4) 'C' using *(I+i)    %6.2f \
secs (%6.2f times as fast)\n", T2, T1/T2);
    i += sprintf(s + i, "    4) 'C' using *(pi+i)   %6.2f \
secs (%6.2f times as fast)\n", T3, T1/T3);
    i += sprintf(s + i, "    4) 'C' using *pi++     %6.2f \
secs (%6.2f times as fast)\n", T4, T1/T4);
    i += sprintf(s + i, "    4) 'C' for(..., pw++)  %6.2f \
secs (%6.2f times as fast)\n", T5, T1/T5);
    i += sprintf(s + i, "    5) asm using *(I+i)    %6.2f \
secs (%6.2f times as fast)\n", T6, T1/T6);
    sprintf(s + i, "Note: time trials were only done on the \

```

```

methods that worked.\n\n\n");

    fh = fopen("results.txt","a");    //open for appending
    for (p = s; (c = *p) > '\0'; p++) //write it to file
        putc(c, fh);
    fclose(fh);                        //close the file
}

```

The results produced in the file [results.txt](#) by running the above exerciser ([wsum2.exe](#)) are as follows:

```

TIME TRIALS AGAINST FLOATING-POINT DOUBLE BENCHMARK:
Floating-point double  345.15 secs  (Benchmark)
4) 'C' using *(I+i)    9.34 secs  ( 36.95 times as fast)
4) 'C' using *(pi+i)  9.39 secs  ( 36.76 times as fast)
4) 'C' using *pi++    9.72 secs  ( 35.51 times as fast)
4) 'C' for(..., pw++) 9.61 secs  ( 35.92 times as fast)
5) asm using *(I+i)   15.99 secs  ( 21.59 times as fast)
Note: time trials were only done on the methods that worked.

```

Test 3

This was to check that the function behaved properly at the extremities of its numeric range. The test was done only on Method 4. With all weights set to 32767, the computation was again done on all 77 inputs and their corresponding weights, first with all inputs set to +32767, then with all inputs set to zero, then with all inputs set to -32767. The results were then further appended to the [results.txt](#) file.

Below is the exerciser for Test 3 with the embedded code fragment to be tested shown in bold. This source code is in the file [wsum3.c](#):

```

#include <stdio.h>
#define NI 77//number of inputs to the neuron
int I, W = 32767;
long SplitLong(void) { //long product with 2 long accumulators
    int i;
    long x, Hi, Lo;
    for(Hi = 0, Lo = 0, i = 0; i < NI; i++) {
        x = (long)I * W;
        Hi += x >> 16;
        Lo += x & 0xFFFF;
    }
    return(((Hi << 1) + (Lo >> 15)) / NI);
}

main() {
    int i, c;
    long x;
    char s[512], *r;
    FILE *fh;
    fh = fopen("results.txt","a");    //open file for appending
    i = sprintf(s
        "TESTING WITH MAXIMUM INPUT AND WEIGHT\n"

```

```

);
I = 32767; x = SplitLong(); c = x;
i += sprintf(s + i,
"   For max positive inputs  %+6d  %9lX\n", c, x
);
I = 0; x = SplitLong(); c = x;
i += sprintf(s + i,
"   For zero inputs...      %+6d  %9lX\n", c, x
);
I = -32767; x = SplitLong(); c = x;
i += sprintf(s + i,
"   For max negative inputs  %+6d  %9lX\n\n", c, x
);
for (r = s; (c = *r) > '\0'; r++)
    putc(c, fh);
fclose(fh);                                //close the file
}

```

The results produced in the file [results.txt](#) by running the above exerciser ([wsum3.exe](#)) are as follows:

```

TESTING WITH MAXIMUM INPUT AND WEIGHT
For max positive inputs  +32766   00007FFE
For zero inputs...      +0       00000000
For max negative inputs  -32766   FFFF8002

```

Notice that the output of the summation can never reach 32767. However, this could only become a problem in a network comprising an impracticably large number of neural layers. If it does become a problem, arrangements will have to be made for setting weights to 32768 which would overflow a 16-bit interger by one bit.

Conclusion

The best coding for the weighted inputs summation function is as follows:

```

int i, al;
long x, Hi, Lo;
for (Hi = 0, Lo = 0, i = 0; i < NI; i++) {
    x = (long)*(I + i) * *(W + i);
    Hi += x >> 16;    //add upper half of long product to Hi
    Lo += x & 0xFFFF; //add lower half of long product to Lo
}
al = ((Hi << 1) + (Lo >> 15)) / NI;

```

where NI is the externally-defined number of inputs, I is the start address of the interger inputs array, W is the start address of the interger weights array, al is the resulting neural activation level. This code performs the task at 37 times the speed of a floating-point double computation.

Final Solution

When implemented within a neural network, the base addresses of the input and weight arrays are unlikely to be available directly as constants I and W as they were in the exerciser programs. These

addresses will probably be passed to the summation function in pointer-variables, say, *pi and *pw. Finally, therefore, the above coding was adapted as follows:

```
int i, a1, *pi, *pw;
long x, Hi, Lo;
for(pi = I, pw = W, Hi = 0, Lo = 0, i = 0; i < NI; i++) {
    x = (long)*(pi + i) * *(pw + i);
    Hi += x >> 16;    //add upper half of long product to Hi
    Lo += x & 0xFFFF; //add lower half of long product to Lo
}
a1 = ((Hi << 1) + (Lo >> 15)) / NI;
```

This was then added to the time trials exerciser and was found to be only half a percent slower than using I and W directly.

This is therefore the code that was recommended for use in the neurone's weighted input summation function.

© December 1997 Robert John Morton

© This content is free and may be reproduced unmodified in its entirety or as “fair usage” quotations that are attributed as follows: “ - [article name] by Robert John Morton
<http://robmorton.20m.com/>”