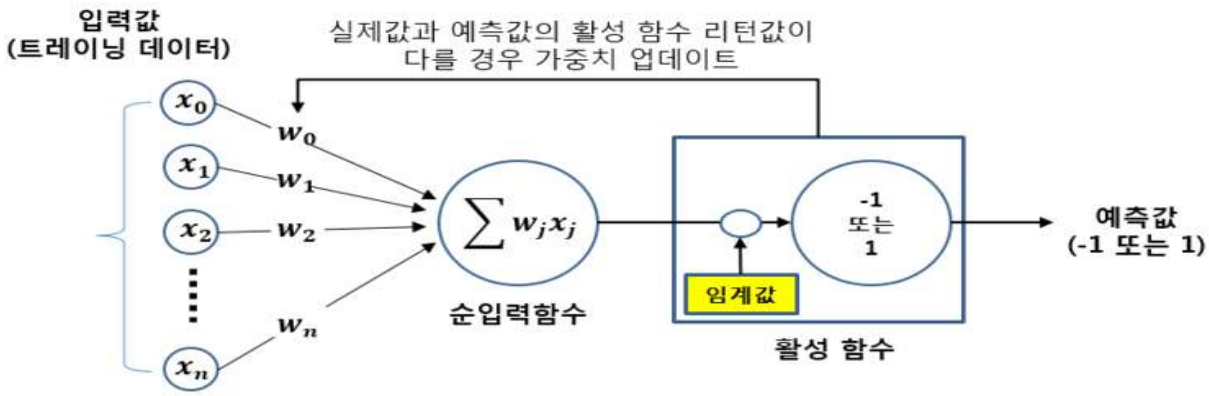
단층 퍼셉트론(SLP)를 구현 및, SIMD연산적용

컴퓨터공학과 2021302 장 두 혁

1. Abstract

컴퓨터구조특론 강의를 통해 SIMD기능들을 공부하였다. SIMD는 Single Instruction Multiple Data의 약자로, 하나의 명령어로 다량의 데이터를 처리하는 것을 말하며, 벡터 연산이라고도 한다. 수업시간에 배운대로, 행렬의 곱셈과, Transpose()함수, 병렬적으로 계산하는 부분을 SIMD기능으로 대체했다. 연구분야는 임베디드 시스템과 인공지능(기계학습)을 공부하는데, 단층 퍼셉트론을 구현을 하여, 위에 언급한대로 SIMD기능들을 적용시켰다.



2. 단층퍼셉트론 소개

단층 퍼셉트론(Single Layer Perceptron)은 뉴런에서 아이디어를 얻어 만들어진 연산 모델이다. 단층 퍼셉트론은 내부적으로 가중치와 바이어스를 저장한다. 가중치는 단층 퍼셉트론의 입력에 곱해지는 값으로, 가중치의 개수는 퍼셉트론에 전달되는 입력의 개수와 같으며, 바이어스는 1개이다. 퍼셉트론에 입력이 전달되면 각각의 입력에 대해 가중치가 곱해진다. 이렇게 곱해진 값의 합에 바이어스를 더하게 되어 가중합을 구해, 일정 수준 이상의 자극 받아야 활성화되는 것처럼 보이는 활성화 함수에 가중합을 입력값으로 활성화 함수를 출력값으로 단층 퍼셉트론 결과값이 된다.

3. 단층 퍼셉트론 구현 및 SIMD 적용

SIMD로 변환된 부분을 설명하기 위해서 퍼셉트론을 수식적으로 표현하면,

가중치	입력값	가중합
$\mathrm{W} = \begin{bmatrix} w_1 & \cdots & w_N \end{bmatrix}$	$\mathrm{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix}$	$u = \mathrm{WX} + b = \sum_{n=1}^{N} w_n x_n + b$

가중치와 입력값을 행 벡터와 열 벡터로 구성하여, 가중합(u)을 그림과 같은 수식으로 구하여

활성화 함수 입력값으로 넣는다.

활성화 함수	Relu 함수	Sigmoid 함수
$y = f(u) = f(\mathrm{WX} + b) = f\left(\sum_{n=1}^{N} w_n x_n + b\right)$	$\mathrm{ReLU}(u) = \max(0, u) = \begin{cases} u & \text{if } u \geq 0 \\ 0 & \text{otherwise} \end{cases}$	$\mathrm{Sigmoid}(u) = \frac{1}{1 + e^{-u}}$

여기서, **첫 번째, 가중합을 구할 때, 가중치 행렬(W)와 입력값행렬(X)의 곱과 바이어스(b)를 더하는 과정을 SIMD로 병렬적으로 변환했다.**
두 번째, 활성화함수 중 여러 함수가 있지만, OR, AND ,XOR연산을 위해서, Relu함수와 Sigmoid함수를 구현해놓았다. 사용은 Sigmoid함수만 사용한다. 활성화 함수 또한, 계산하는 과정에 병렬적인 지수함수 연산부분을 SIMD를 통해 구현했다.

SLP가 계산한 출력과 실제 정답과의 오차를 구하기 위해 손실 함수(Loss Function)를 사용합니다. 손실 함수의 종류는 여러가지가 있는데, SLP을 이용해 **회귀분석을 하기때문에 회귀분석에 적절한 MSE(Mean Squared Error) 손실 함수를 사용했다. 여기서, 세번째 연산적으로 행렬이 곱셈과 뺄셈 부분을 SIMD로 표현했다.**

손실 함수
$L = \frac{1}{I} \sum_{i=1}^{I} \left(f\left(\mathrm{WP}_i + b\right) - Q_i\right)^2$

역전파 Backward함수로 바이어스의 기울기를 구할 때, 발생하는 행렬의 곱셈과, 수업시간에 배운 Transpose()가 발생하는데, 네 번째로, Transpose() 함수 부분과 행렬의 곱셈 부분을 SIMD로 변환했다.
1000번 학습을 하여, 기존 연산으로 했을 때, 동일한 확률 결과값이 나오며, 가끔은 확률결과값이 미세하게 좋게 나올 경우도 나온다. 모든 부분을 SIMD로 변환하고 싶었지만, 이상하게 확률값이 다르게 나오는경우도 있어서, 최대한 병렬적으로 구현할 수 있는 부분을 변환했다.

4. 변환부분 코드설명

앞에서 언급한 순서대로, 코드 부분을 정리했다.(주요 변환 부분설명하고, 곱셈과 Transpose()의 상세한 코드는 뒤에 설명했음)

1. Forward 함수 부분

기존 함수

```
Matrix WeightedNeuron::Forward(const
Matrix& x) {

m_X = x;
// 입력값행렬
m_U = m_W * x;
//가중치행렬 * 입력값행렬

for (std::size_t i = 0; i <
m_U.GetColumn(); ++i) {
m_U(0, i) += m_B;
// 가중치*입력값으로 곱한 행렬에
바이어스를 //더한 연산
        }
return m_U;
}
```

SIMD 적용 함수

```
Matrix WeightedNeuron::Forward( Matrix& x) {

m_X = x;
// 입력값행렬
m_U = add_124(m_W, x);
//가중치행렬 * 입력값행렬(SIMD로 구현한함수/
//뒤에설명)

__m128 v,v1;
__m128 op1 = _mm_set_ps(m_B, m_B, m_B, m_B);
        v = _mm_set_ps(m_U(0, 0), m_U(0, 1), m_U(0,
2), m_U(0, 3));

v1 = _mm_add_ps(v, op1);
// 가중치*입력값으로 곱한 행렬에 바이어스를 //더한
연산
Matrix m_UU({ v1.m128_f32[3] ,v1.m128_f32[2]
,v1.m128_f32[1] ,v1.m128_f32[0] },1,4);

m_U = m_UU;
return m_U;
}
```

2. 활성화 함수(Sigmoid/Relu)

기존 함수

```
//시그모이드 함수(Sigmoid 함수)
Matrix SigmoidF(const Matrix& x) {
Matrix result(x.GetRow(),
x.GetColumn());
for (std::size_t i = 0; i < x.GetRow();
++i) {


for (std::size_t j = 0; j <
x.GetColumn(); ++j) {
result(i, j) = 1 / (1 + std::exp(-x(i,
j)));
//지수함수 & sigmoid 연산}
                }
return result;
}


Matrix SigmoidD(const Matrix& x) {
Matrix result(x.GetRow(),
x.GetColumn());
Matrix func = SigmoidF(x);

for (std::size_t i = 0; i < x.GetRow();
++i) {
for (std::size_t j = 0; j <
x.GetColumn(); ++j) {

result(i, j) = func(i, j) * (1 - func(i,
j));
                }
        }
return result;
}
```

SIMD 적용 함수

```
//시그모이드 함수(Sigmoid 함수)
Matrix SigmoidF(const Matrix& x) {
Matrix result(x.GetRow(), x.GetColumn());

__m128 v[1];
__m128 op1 = _mm_set1_ps(1);

v[0] = _mm_set_ps(-x(0, 0), -x(0, 1), -x(0, 2),
-x(0, 3));

//지수함수 SIMD 구현 & sigmoid 연산
v[0] = _mm_add_ps(FastExpSse(v[0]), op1);
v[0] = _mm_set_ps(1/v[0].m128_f32[0], 1/
v[0].m128_f32[1], 1 / v[0].m128_f32[2], 1 /
v[0].m128_f32[3]);

Matrix m_res({ v[0].m128_f32[0],v[0].m128_f32[1]
,v[0].m128_f32[2] ,v[0].m128_f32[3] },1,4);
result = m_res;
return result;
}


Matrix SigmoidD(const Matrix& x) {
Matrix result(x.GetRow(), x.GetColumn());
Matrix func = SigmoidF(x);

for (std::size_t i = 0; i < x.GetRow(); ++i) {

for (std::size_t j = 0; j < x.GetColumn(); ++j) {
result(i, j) = func(i, j) * (1 - func(i, j));
        }
        }
return result;
}
```

```
//Relu함수
Matrix ReLUF(const Matrix& x) {
Matrix result(x.GetRow(),
x.GetColumn());

for (std::size_t i = 0; i < x.GetRow();
++i) {
for (std::size_t j = 0; j <
x.GetColumn(); ++j) {
//1을 기준으로 임계값을 표현한 부분

//1보다 크면 출력
result(i, j) = std::max(0.0, x(i, j));
                          }
}
return result;
}


Matrix ReLUD(const Matrix& x) {
Matrix result(x.GetRow(),
x.GetColumn());
for (std::size_t i = 0; i < x.GetRow();
++i) {

for (std::size_t j = 0; j <
x.GetColumn(); ++j) {
result(i, j) = x(i, j) >= 0 ? 1 : 0;
}

    }
     return result;
        }
```

```
//Relu함수
Matrix ReLUF(const Matrix& x) {Matrix
result(x.GetRow(), x.GetColumn());

//1을 기준으로 임계값을 표현한 부분
//1보다 크면 출력
__m128 v[1];
__m128 op1 = _mm_set1_ps(1);

v[0] = _mm_set_ps(x(0, 0), x(0, 1), x(0, 2), x(0,
3));
v[0] = _mm_max_ps(op1,v[0]);

Matrix m_res({ v[0].m128_f32[0],v[0].m128_f32[1]
,v[0].m128_f32[2] ,v[0].m128_f32[3] },1,4);

result = m_res;
return result;
}

Matrix ReLUD(const Matrix& x) {
Matrix result(x.GetRow(), x.GetColumn());

for (std::size_t i = 0; i < x.GetRow(); ++i) {

for (std::size_t j = 0; j < x.GetColumn(); ++j) {
result(i, j) = x(i, j) >= 0 ? 1 : 0;
}
        }
return result;
        }
```

Exp()함수	Exp() 함수 SIMD 새로 구현

<math.h> 라이브러리 통해 사용	static inline __m128 FastExpSse(__m128 x)
	{
	__m128 a = _mm_set1_ps(12102203.2f); // (1 << 23) / ln(2)
	__m128i b = _mm_set1_epi32(127 * (1 << 23) - 486411);
	__m128 m87 = _mm_set1_ps(-87);
	// fast exponential function, x should be in [-87, 87]
	__m128 mask = _mm_cmpge_ps(x, m87);
	
	__m128i tmp = _mm_add_epi32(_mm_cvtps_epi32(_mm_mul_ps(a, x)),b);
	
	return _mm_and_ps(_mm_castsi128_ps(tmp), mask);
	}

3. Loss function 함수 MSE

기존 함수

```
double MSEF(const Matrix& y, const Matrix&
a) {

double result = 0.0;
Matrix ya = y - a;

for (std::size_t i = 0; i <
ya.GetColumn(); ++i) {
result += std::pow(ya(0, i), 2);
        }

result /= ya.GetColumn();
return result;
        }

Matrix MSED(const Matrix& y, const Matrix&
a) {

Matrix ya = y - a;
ya *= (2.0 / y.GetColumn());

return ya;
}
```

SIMD 적용 함수

```
double MSEF(const Matrix& y, const Matrix& a) {
double result = 0.0;
Matrix ya;

float av[2][4];
__m128 v[2];
v[0] = _mm_set_ps(y(0, 0), y(0, 1), y(0, 2),
y(0,3));
v[1] = _mm_set_ps(a(0, 0), a(0, 1), a(0, 2), a(0,
3));

v[0] = _mm_sub_ps(v[0], v[1]);
v[0] = _mm_mul_ps(v[0], v[0]);

result = v[0].m128_f32[0] + v[0].m128_f32[1] +
v[0].m128_f32[2] + v[0].m128_f32[3];

result /= 4;
return result;
}

Matrix MSED(const Matrix& y, const Matrix& a) {

Matrix ya= y - a;
ya = ya *(2.0 / 4);

return ya;
}
```

4. Backward 함수

기존 함수

```
Matrix WeightedNeuron::Backward(const

Matrix& d) {
Matrix xT(m_X);

//Transpose() 사용
xT.Transpose();

m_DW = d * xT;

//행렬곱셈
m_DB = d * Matrix(1, m_X.GetColumn(), 1);

Matrix wT(m_W);

//Transpose() 사용
wT.Transpose();

return wT * d;
}
```

SIMD 적용 함수

```
Matrix WeightedNeuron::Backward( Matrix& d) {

Matrix xT(m_X);
Matrix newTr(4, 2);
Matrix newTr2(2, 1);
//Transpose() 사용 2*4행렬에대해 구현
newTr = Transpose24(m_X);

//SIMD로 (1,4) * (4*2) 행렬곱셈 구현
m_DW=add_142(d, newTr);

Matrix a(1, m_X.GetColumn(), 1);
//SIMD로 (1,4) * (4*1) 행렬곱셈 구현
m_DB = add_141(d,a);

Matrix wT(m_W);
//Transpose() 사용 1*2행렬에대해 구현
newTr2 = Transpose12(m_W);

//SIMD로 (2,1) * (1*4) 행렬곱셈 구현
return add_214(newTr2 , d);
}
```

5. 행렬의 곱셈

행렬곱셈마다, 예로들어 1*4행렬도 있지만, 2*1, 4*2 행렬 등 다양하고, Matrix클래스에서 값을 가져와야하며, 각각 행렬의 곱셈마다 필요한 연산들만 넣어서 각각 구현했다./ 기존은 연산 소요시간(단위 :밀리초)이 9였는데, SIMD구현된 함수는 6,7,8,10 이었다. 결코 코드 양이 많이 것이 아니라 각각 행렬크기마다 각각 사용함.

기존 함수

```
// (1,2) *(2,4) 행렬 (1,4) *(4,2) 행렬
// (2,1) *(1,4) 행렬 // (1,4) *(4,1)
행렬을 사용
Matrix   Matrix::operator*(const   Matrix&
matrix) const {

assert(m_Column == matrix.m_Row);

Matrix result(m_Row, matrix.m_Column);


for (std::size_t i = 0; i < m_Row; ++i) {

for   (std::size_t   j   =   0;   j   <
matrix.m_Column; ++j) {

for (std::size_t k = 0; k < m_Column; ++k)
{

result(i, j) += (*this)(i, k) * matrix(k,
j);
        }
}
}
        return result;
}
```

SIMD 적용 함수

```
// (1,2) *(2,4) 행렬
Matrix add_124(Matrix& matrixb, Matrix& matrixc)
{
float matrix1[2][4] = {
matrixc(0,0),matrixc(0,1),matrixc(0,2),matrixc(0,
3),matrixc(1,0),matrixc(1,1),matrixc(1,2),matrixc
(1,3) };

float matrix2[1][2] ={matrixb(0,0),matrixb(0,1)};

__m128 b = _mm_load1_ps(&matrix2[0][0]);
__m128 b1 = _mm_load1_ps(&matrix2[0][1]);
__m128 c = _mm_loadu_ps(&matrix1[0][0]);
__m128 d = _mm_loadu_ps(&matrix1[1][0]);
__m128 mul1 = _mm_mul_ps(b, c);
__m128 mul2 = _mm_mul_ps(b1, d);

__m128 add1 = _mm_add_ps(mul1, mul2);

Matrix out({add1.m128_f32[0] , add1.m128_f32[1]
,add1.m128_f32[2] ,add1.m128_f32[3]}, 1, 4);

return out;
}

// (1,4) *(4,2) 행렬
Matrix add_142(Matrix& matrixb, Matrix& matrixc)
{

float matrix1[4][2] = {
matrixc(0,0),matrixc(1,0),matrixc(2,0),matrixc(3,
0),matrixc(0,1),matrixc(1,1),matrixc(2,1),matrixc
(3,1) };

float matrix2[1][4] = {
matrixb(0,0),matrixb(0,1),matrixb(0,2),matrixb(0,
3) };

__m128 b = _mm_loadu_ps(&matrix1[0][0]);
__m128 b1 = _mm_loadu_ps(&matrix1[2][0]);
__m128 c = _mm_loadu_ps(&matrix2[0][0]);
__m128 mul1 = _mm_mul_ps(c, b);
__m128 mul2 = _mm_mul_ps(c, b1);

float sum1 = mul1.m128_f32[0] + mul1.m128_f32[1]
+ mul1.m128_f32[2] + mul1.m128_f32[3];

float sum2 = mul2.m128_f32[0] + mul2.m128_f32[1]
+ mul2.m128_f32[2] + mul2.m128_f32[3];

Matrix out({sum1,sum2}, 1, 2);
return out;
}
```

```
// (2,1) *(1,4) 행렬
Matrix add_214(Matrix& matrixb, Matrix& matrixc)
{

float matrix1[1][4] = {
matrixc(0,0),matrixc(0,1),matrixc(0,2),matrixc(0,
3)};

float matrix2[2][1] = { matrixb(0,0),matrixb(1,0)
};

__m128 b = _mm_load1_ps(&matrix2[0][0]);
__m128 b1 = _mm_load1_ps(&matrix2[1][0]);
__m128 c = _mm_loadu_ps(&matrix1[0][0]);
__m128 mul1 = _mm_mul_ps( b,c);
__m128 mul2 = _mm_mul_ps( b1,c);

Matrix out({ mul1.m128_f32[0] , mul1.m128_f32[1]
,mul1.m128_f32[2]
,mul1.m128_f32[3],mul2.m128_f32[0] ,
mul2.m128_f32[1] ,mul2.m128_f32[2]
,mul2.m128_f32[3] },2, 4);

return out;
}

// (1,4) *(4,1) 행렬
Matrix add_141(Matrix& matrixb, Matrix& matrixc)
{

float matrix1[1][4] = {
matrixb(0,0),matrixb(0,1),matrixb(0,2),matrixb(0,
3) };
float matrix2[1][1] = { matrixc(0,0)};

__m128 b = _mm_loadu_ps(&matrix1[0][0]);
__m128 c = _mm_load1_ps(&matrix2[0][0]);

__m128 mul1 = _mm_mul_ps(b, c);
float sum1 = mul1.m128_f32[0] + mul1.m128_f32[1]
+ mul1.m128_f32[2] + mul1.m128_f32[3];

Matrix out({ sum1 },1,1);

return out;
}
```

6. Transpose() 함수

(1,2) >> (2,1)행렬과 (2,4) >> (4,2) 행렬로 Transpose하는 함수 구현(양이많은 것이)

기존은 연산 소요시간(단위 :밀리초)이 13였는데, SIMD구현된 함수는 7,8 이었다.

기존 함수

```
void Matrix::Transpose() {
        Matrix temp(m_Column, m_Row);
        for (std::size_t i = 0; i < m_Row;
++i) {
                for (std::size_t j = 0; j
< m_Column; ++j) {
                        temp(j, i) =
(*this)(i, j);
                }
        }
        Swap(temp);
}
```

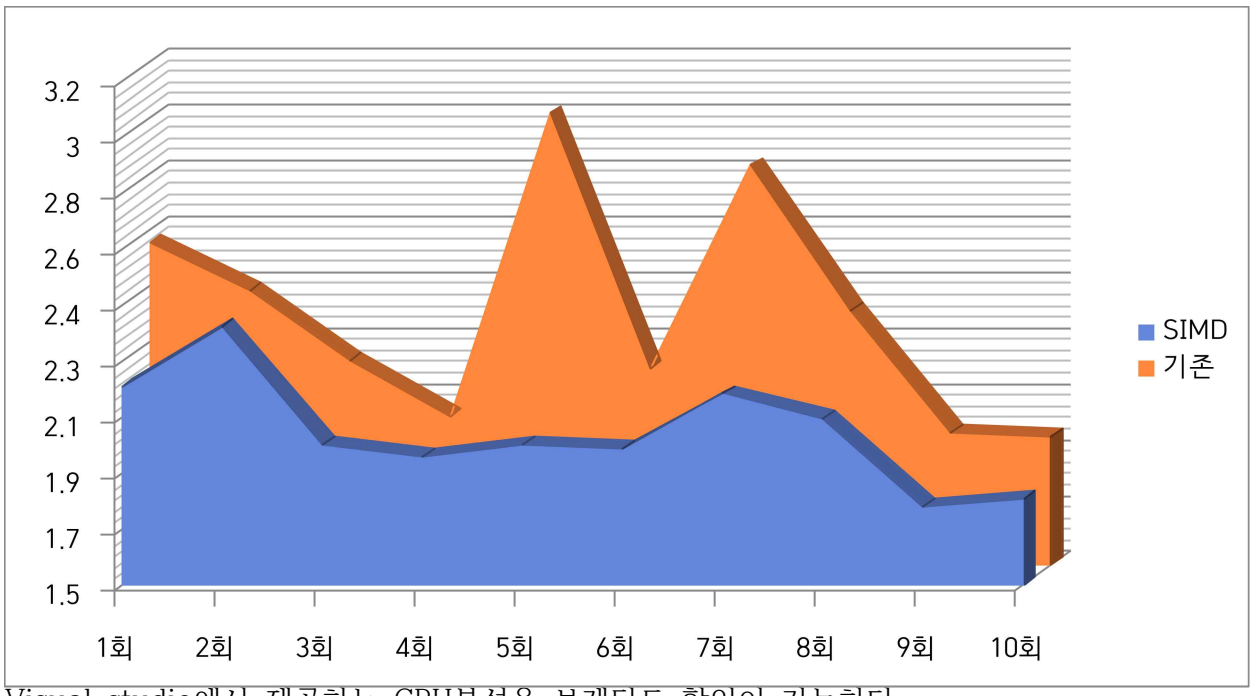
SIMD 적용 함수

```
// (1,2) >> (2,1) Transpose()구현
Matrix Transpose12(Matrix& matrix) {
        float matrix1[1][2] = {
matrix(0,0),matrix(0,1)};
        float matrix2[1][4] = { 0,0 };

        __m128 a = _mm_loadu_ps(&matrix1[0][0]);
        __m128 c = _mm_loadu_ps(&matrix2[0][0]);
        __m128 aa = _mm_unpacklo_ps(a, c);
        __m128 bb = _mm_unpackhi_ps(aa, c);

        Matrix out({ aa.m128_f32[0] ,
bb.m128_f32[0]}, 2,1);

        return out;
}


// (2,4) >> (4,2) Transpose()구현
```

```
Matrix Transpose24(Matrix& matrix) {
        float matrix1[2][4] =
{matrix(0,0),matrix(0,1),matrix(0,2),matrix(0,3),
matrix(1,0),matrix(1,1),matrix(1,2),matrix(1,3)
};
        float matrixset[1][4] = { 0,0,0,0 };

        __m128 a = _mm_loadu_ps(&matrix1[0][0]);
        __m128 b = _mm_loadu_ps(&matrix1[1][0]);
        __m128 c =
_mm_loadu_ps(&matrixset[0][0]);

        __m128 aa = _mm_unpacklo_ps(a, c);
        __m128 bb = _mm_unpackhi_ps(a, c);
        __m128 cc = _mm_unpacklo_ps(b, c);
        __m128 dd = _mm_unpackhi_ps(b, c);


        __m128 aaa = _mm_unpacklo_ps(aa, cc);
        __m128 bbb = _mm_unpackhi_ps(aa, cc);
        __m128 ccc = _mm_unpacklo_ps(bb, dd);
        __m128 ddd = _mm_unpackhi_ps(bb, dd);

        Matrix out({ aaa.m128_f32[0] ,
aaa.m128_f32[1] ,bbb.m128_f32[0]
,bbb.m128_f32[1],ccc.m128_f32[0],ccc.m128_f32[1],
ddd.m128_f32[0] ,ddd.m128_f32[1]},4,2);

        return out;
}
```

5. 실행시간 비교 & 단층 퍼셉트론 확률결과

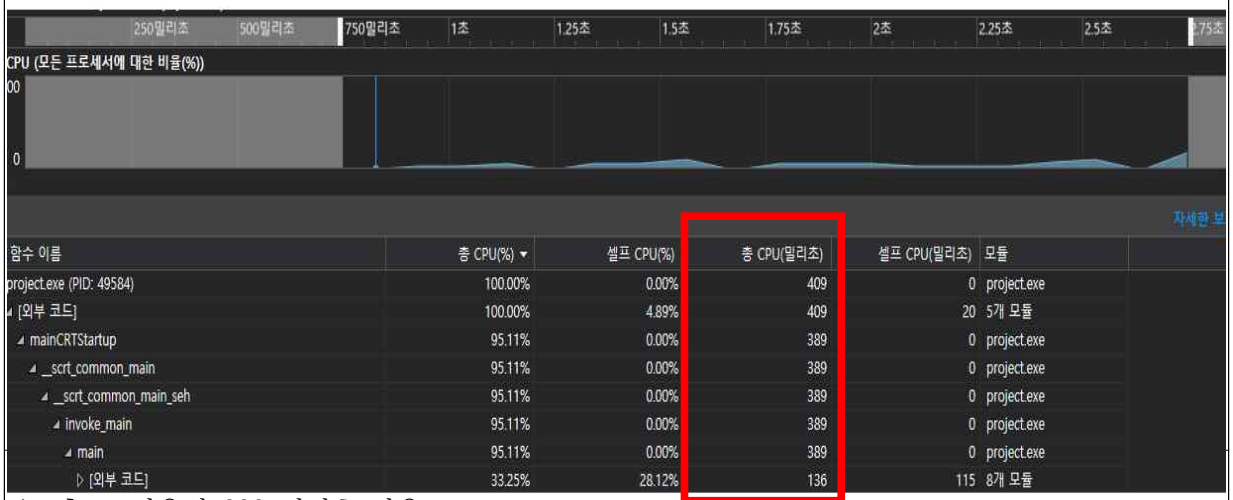
총 10번를 비교하여, 실행하여 평균적으로 SIMD 연산 시간은 2.0142초/ 기존 함수 연산은 2.3491초이다. 약 15% 빠르게 소요되었다. 만약에 임베디드 보드에 SIMD를 사용한다면, intel cpu가 장착된다면, 연산부분과 메모리 부분에서 효율적으로 사용될 것을 예상한다.
아래는 비교 표이다. (컴퓨터마다 사양이 다르기 때문에, 시간보다는 15%향상된것에 의미가 있다.)



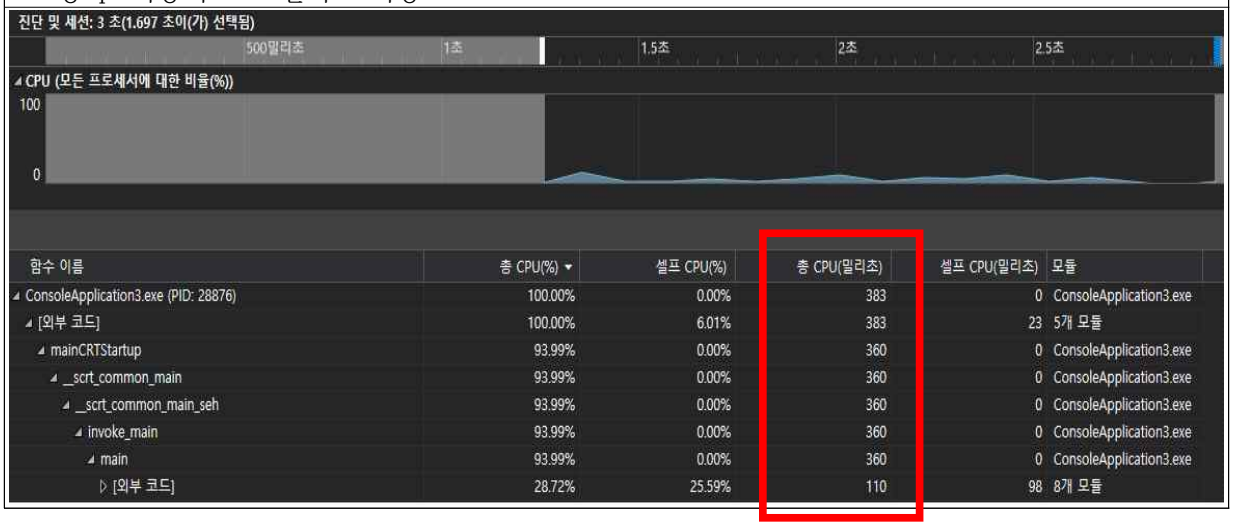
Visual studio에서 제공하는 CPU분석을 보게되도 확인이 가능하다.

기존 함수 사용

▶ 총cpu사용이 409 밀리초 사용



▶ 총cpu사용이 383 밀리초 사용



단층 퍼셉트론의 정확도 비교

오른쪽이 기존 함수로 구현된 정확도이며, 왼쪽이 SIMD 함수로 구현된 정확도인데, 정확도면에서, 다를 바가 없으며, 오히려 0 이라고 예상하는 부분에서 0.01 정도 정확했으며, 나머지는 차이가 미미했다.

input행렬 (2,4)행렬로 (0,1,0,1) 와 (0,0,1,1) 에 대해 OR연산을 한결과 out행렬(1,4) (0,1,1,1) 1000번 학습을 했을 경우 아래 같은 정확도가 나옴

```
Matrix input({ 0, 1, 0, 1,
    0, 0, 1, 1 }, 2, 4);
Matrix output({ 0, 1, 1, 1 }, 1, 4);
```

```
Epoch 1000: MSE 0.00551211
[ 0.0903529 0.072053 0.942089 0.927099 ]
일반함수 소요된 시간 : 2.851
계속하려면 아무 키나 누르십시오 . . .
```

```
Epoch 1000: MSE 0.00540675
[ 0.0888918 0.0719729 0.94324 0.927205 ]
SSE함수 소요된 시간 : 2.146
계속하려면 아무 키나 누르십시오 . . .
```