JDCS
디지털콘텐츠학회논문지
Journal of Digital Contents Society
Vol. 18, No. 6, pp. 1165-1173, Oct. 2017

IoT 디바이스에서 다차원 디지털 신호 처리를 위한 신경망 최적화

최 권 택
강남대학교 소프트웨어응용학부 가상현실전공

Neural networks optimization for multi-dimensional digital signal processing in IoT devices

KwonTaeg Choi
Division of Software Application, Kangnam University, YongIn-si 16979, Korea

[요 약]

가장 대표적인 기계학습 알고리즘인 딥러닝 방법은 여러 응용 분야에서 활용성이 입증돼 디지털신호처리에 널리 사용되고 있다. 그러나 많은 학습데이터를 사용해 학습하는 과정에서 많은 메모리와 학습시간이 필요하기 때문에 CPU 성능과 메모리 용량이 제한된 IoT 디바이스에 딥러닝 기술을 적용하기는 어렵다. 특히 메모리 용량이 2K~8K 로 극히 적은 아두이노 기반의 디바이스를 사용한다면 알고리즘 구현에 많은 한계가 발생한다. 본 논문에서는 정확성과 효율성이 입증돼 여러 분야에서 활용되고 있는 ELM 알고리즘을 아두이노에서 최적화하는 방법을 제안하고, 실험을 통해 메모리 용량이 2KB인 아두이노 UNO와 메모리 용량이 8KB인 아두이노 MEGA에서 각각 15차원, 42차원의 다중 클래스 학습이 가능함을 보였다. 실험을 입증하기 위해 가우시안 혼합 모델링을 사용해 생성한 데이터셋과 범용적으로 사용하는 UCI 데이터셋을 사용해 제안한 알고리즘의 효율성을 입증하였다.

[Abstract]

Deep learning method, which is one of the most famous machine learning algorithms, has proven its applicability in various applications and is widely used in digital signal processing. However, it is difficult to apply deep learning technology to IoT devices with limited CPU performance and memory capacity, because a large number of training samples requires a lot of memory and computation time. In particular, if the Arduino with a very small memory capacity of 2K to 8K, is used, there are many limitations in implementing the algorithm. In this paper, we propose a method to optimize the ELM algorithm, which is proved to be accurate and efficient in various fields, on Arduino board. Experiments have shown that multi-class learning is possible up to 15-dimensional data on Arduino UNO with memory capacity of 2KB and possible up to 42-dimensional data on Arduino MEGA with memory capacity of 8KB. To evaluate the experiment, we proved the effectiveness of the proposed algorithm using the data sets generated using gaussian mixture modeling and the public UCI data sets.

색인어 : 신경망, 이엘엠, 기계학습, 아두이노
Key word : Neural Network, Extreme Learning Machine, Machine Learning, Arduino

http://dx.doi.org/10.9728/dcs.2017.18.6.1165

This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-CommercialLicense(http://creativecommons.org/licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

Received 28 September 2017; **Revised** 20 October 2017
Accepted 25 October 2017

***Corresponding Author; KwonTaeg Choi**

Tel: +82-02-2275-4435
E-mail: kwontaeg.choi@kangnam.ac.kr

Ⅰ. 서론

4차 산업 혁명을 이끌어갈 기술 중 가상현실, 증강현실관련 콘텐츠에 대한 연구가 그 어느 때보다 활발히 이루어지고 있다. 특히 인공지능 기술의 급격할 발전과 맞물려 스마트 콘텐츠에 대한 연구가 기술 개발 뿐만 아니라 인프라 구축 및 지원을 포함한 플랫폼 관점에서도 조명되고 있다. 특히 가상현실 콘텐츠는 소프트웨어뿐 만 아니라 사용자 경험을 극대화하기 위해서 모션 컨트롤러, 글러브 컨트롤러, 링 컨트롤러 등 다양한 하드웨어와 결합되고 있다[1][2][3].

최근의 센서디바이스는 센싱 기능만 하지 않고, 마이크로프로세서를 포함해 파라미터 설정 최적화, 출력 보정등 지능적인 프로세싱 기능을 포함하고 있다. 센서들도 실시간 데이터를 처리하고 해석할 수 있는 기능이 필요하기 때문이다. 그러나 이 모든 걸 중앙에서 조합해서 처리할 수 는 없다. 환경에 바로 적응해야 하거나 실시간 분석이 필요하거나, 스트리밍 데이터 처리가 필요할 경우 온보드(On-board)에서 최대한 지연시간 없이 디지털 신호를 지능적으로 처리할 필요가 있다.

디지털 신호를 효율적으로 처리하기 위해서는 기계학습 알고리즘이 주로 사용된다. 가장 대표적인 기계학습 알고리즘인 딥러닝 방법은 최근 여러 응용 분야에서 활용성이 입증돼 널리 사용되고 있다. 그러나 많은 학습데이터를 사용해 학습하는 과정에서 많은 메모리와 학습시간이 필요하기 때문에 CPU 성능과 메모리 용량이 충분하지 않은 온보드 디바이스에 딥러닝 기술을 적용하기는 어렵다. 예를 들어 메모리 용량이 2~8KB 인 아두이노 기반의 디바이스를 사용한다면 알고리즘 구현에 많은 한계가 발생한다.

본 논문에서는 정확성과 효율성이 입증돼 분류, 회귀, 군집화 관련 분야에서 널리 활용되고 있는 ELM(Extreme Learning Machine) [4][5][6][7][8][9][10] 알고리즘을 아두이노 UNO와 아두이노 MEGA에서 구현하고 다차원의 많은 학습데이터를 처리할 수 있도록 다양한 최적화 방법을 제안하고자 한다.

아두이노, 라즈베리파이는 IoT 디바이스에 대한 프로토타입 연구 개발을 위해 많이 사용한다. 두 디바이스의 메모리 관련 주요 스펙은 표 1과 같다. 아두이노는 메모리가 작아 주로 마이크로 컨트롤러로 사용되고, 라즈베리파이는 복잡한 신호처리가 가능하다. 보조 메모리로 라즈베리파이는 대용량 범용 SD 카드를 아두이노는 EEPROM을 사용한다. 메모리 아키텍처 구조는 고려하지 않더라도 실시간으로 읽기/쓰기 가능한 메모리가 아두이노 2~8KB, 라즈베리파이는 1GB라는 매우 큰 차이가 있다. 따라서 CPU 처리 속도 보다는 메모리 용량이 지극히 적은 디바이스에서 성능이 뛰어난 기계학습 알고리즘을 구현해야 하는 문제가 발생한다.

아두이노와 라즈베리파이는 메모리 아키텍처가 다르기 때문에 기계학습 구현 방법도 달라진다. 메모리 공간이 작은 아두이노에서 효율적인 기계학습 알고리즘을 구현하기 위해 두 디바이스의 메모리 아키텍처를 살펴보면 그림 1과 같다.

표 1. 아두이노와 라즈베리파이 메모리 비교
Table 1. Memory specification comparisons of Arduino and Raspberry Pie

DB	아두이노 UNO	아두이노 MEGA	라즈베리파이
FLASH	32	256	---
SRAM	2	8KB	---
EEPROM	1	4K	---
SDRAM	---	---	1GB
STORAGE	SD card 모듈 필요	SD card 모듈필요	microSD

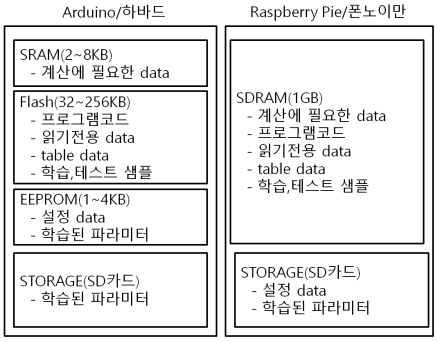

그림 1. 아두이노와 라즈베리파이의 메모리 구조 비교
Fig. 1. Memory architecture comparisons of Arduino and Raspberry Pie

라즈베리파이의 경우 충분한 공간의 하나의 큰 메모리 구조를 가지고 있으나 아두이노는 프로그램을 적재할 수 있는 FLASH 메모리와 계산과정에 필요한 SRAM, 제한적으로 읽고 쓰기가 가능한 EEPROM을 효과적으로 사용해야 기계학습 알고리즘을 구현할 수 있다. 메모리가 작기 때문에 속도 개선을 위해 주로 사용하는 테이블 매핑 기법은 사용할 수 없다.

아두이노에서 기계학습 알고리즘을 학습시키기 위해 알고리즘의 어떤 부분이 어느 메모리에 저장되는지 파악할 필요가 있다.

- FLASH : 프로그램이 들어가는 메모리이다. 통합 환경에서 코드 컴파일 후 업로드하면 FLASH 메모리에 복사되고 전원이 차단되어도 내용이 사라지지 않는다. 실행 중에는 쓰기가 불가능하고 읽기만 가능하기 때문에 이미 학습된 파라미터와 테이블 데이터는 이 영역에 저장할 수 있다.

- SRAM : 프로그램 실행 시 필요한 변수와 동적메모리 공간이다. PC의 DRAM 보다 매우 빠르게 동작하지만 2KB공간이기 때문에 알고리즘이 효율적으로 최적화 되어야 한다.

- EEPROM : 읽기/쓰기가 가능한 메모리이다. 10만 번 정도 지우고 다시 쓰기가 가능하나 속도 면에서 FLASH에 비해 매우 느리다. 따라서 설정 값이나 학습된 모델 데이터를 저장하는데 사용할 수 있다.

결국 기계학습을 아두이노에서 수행하기 위해 사용 메모리에 따라 다양한 최적화 방법이 필요하다. 본 논문에서는 정확성와 효율성이 입증돼 여러 분야에서 활용되고 있는 ELM 알고리즘을 아두이노에서 최적화하는 방법을 제안하고자 한다.

II. 제안 방법

2-1 아두이노에서의 ELM 알고리즘 학습

ELM은 SLFN(Single-hidden Layer Feedforwrad Network)의 하나로 은닉층의 가중치를 랜덤하게 설정하고 별도의 학습을 하지 않아 계산이 간단하고 다양한 데이터 셋에서 그 우수성이 입증된 알고리즘이다. ELM 출력은 식(1)처럼 계산된다.

$$y = H\alpha \tag{1}$$

여기서 H 행렬은 식 (2)에서 처럼 ELM에서 입력 X와 은닉층사이의 가중치 W, 바이어스 b의 합에 활성화 함수(activation function)를 적용한 값을 의미한다.

$$H = \frac{1}{(1 + \exp(-(WX + b))} \tag{2}$$

학습데이터와 학습데이터에 대한 목표 값 y 가 주어지면 가중치 α는 식 (3)을 통해 구할 수 있다.

$$\alpha = H^{+} y = (HH^{T})^{-1} Hy \tag{3}$$

비전 센서 같은 고차원 데이터가 아니더라도 기계학습에서는 학습데이터수가 많기 때문에 H 행렬은 매우 클 수 있다. H 행렬은 실시간으로 읽고 쓰기가 가능해야 하므로 반드시 SRAM에 위치해야 한다. 은닉층 노드수가 10개이고, 학습데이터의 수가 50개 일 때 H 행렬을 위해 필요한 메모리는 2000Bytes(=10 * 50 * sizeof(float))로 2KB SRAM공간에서는 대다수를 차지해 다른 계산을 할 수 없는 상태가 된다.

본 논문에서는 학습데이터 수 n 이 커서 H행렬 크기가 아두이노 SRAM 보다 큰 경우 그림 2처럼 H행렬을 직접 계산에 사용하지 않고, 학습데이터를 순차적으로 계산할 수 있는 행렬 최적화 알고리즘을 제안하고자 한다.

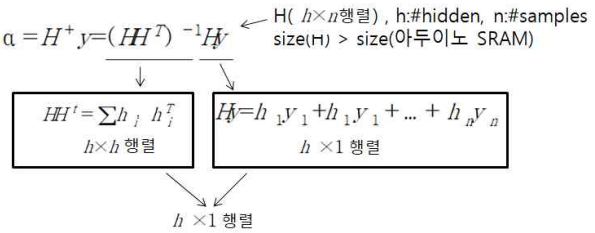

그림 2. 제안하는 ELM 메모리 최적화 알고리즘
Fig. 2. The proposed ELM memory optimization algorithm

2-2 순차적 역행렬 계산

H 행렬 계산을 위해 은닉층의 랜덤 가중치 W와 바이어스 b, 그리고 입력 행렬 X는 메모리 어딘가에 할당되어야 한다. W, b 의 경우 ELM에서는 학습 없이 랜덤하게 설정하기 때문에 읽기 전용 메모리에 위치해도 된다. W, b를 저장하기 위해 필요한 메모리 공간은 데이터 차원크기와 은닉층의 노드수에 의해 결정되기 때문에 학습 데이터 수와는 독립적이다. 본 논문에서는 W, b 를 FLASH 메모리에 저장해 동적 계산에 필요한 SRAM 공간을 최대한 확보하고자 한다.

ELM에서는 모든 학습데이터에 대한 은닉층 출력을 계산하고 닫힌 형태 방정식(closed-form equation)을 사용해 최적화된 해를(solution) 계산한다. 입력데이터의 경우는 실시간으로 학습데이터를 취득한다고 가정해 SD카드에 저장해 순차적으로 읽어 들이는 방식을 사용해 학습데이터 전체를 SRAM에 저장하지는 않았다.

그러나 문제는 H 행렬의 크기가 학습 데이터수에 의존하기 때문에 SRAM이 작아 H 행렬에 대한 공간을 할당할 수 없다면 식 (3)을 계산할 수 없다. 다행이도 계산이 필요한 부분은 H가 아닌 HH^{T}이다. 이 행렬은 학습데이터 수와 상관없이 은닉층 노드수의 정방형 행렬이다. 학습데이터를 하나씩 순차방식을 통해 은닉층 출력으로 계산하는 식 (4)를 이용해 H 행렬을 저장하지 않고 계산가능 하다.

$$HH^{t} = \sum h_i \cdot h_i^{T} \tag{4}$$

HH^{T} 행렬은 정방형 행렬로 여러 가지 방법으로 역행렬 계산이 가능하다. 행렬 크기가 2,3,4일 경우 역행렬 공식이 존재하며 5보다 클 경우는 여인수 전개(cofactor expansion)[11], 특이값 분해(SVD, Singular Value Decomposition)[12], 가우스-조던 소거법(Gauss Jordan Elimination)[13]을 사용해 계산가능하다. 메모리가 충분한 PC 급 디바이스에서는 어떤 방법을 사용해도 된다. SRAM이 부족한 아두이노에서는 각각의 알고리즘의 특성을 파악해야 한다.

- 여인수 전개 기반 역행렬 계산

식 (5)처럼 여인수 전개를 이용할 경우 행렬 A의 역행렬은 행렬식과 여인수 전개를 사용해 계산가능하다.

$$A^{-1} = \frac{1}{detA}(cofactor\ \ of\ A)^{T} \tag{5}$$

여인수 전개는 행렬식을 사용해 계산할 수 있다.따라서 역행렬은 행렬식 알고리즘에 전적으로 의존한다. 행렬식 계산 알고리즘은 재귀알고리즘으로 쉽게 구현가능하다. 예를 들어 det(4)는 4번의 det(3), det(3) 은 3번의 det(2)로 계산할 수 있다. det(2)는 단순 공식으로 계산 가능하다.

따라서 여인수 전개를 기반으로 역행렬을 계산할 경우 코드의 크기가 작아 FLASH 메모리 적재는 문제가 없다. 그러나 행렬식 알고리즘이 재귀방식이라 재귀 호출의 깊이가 깊어지면 아두이노에서는 스택 오버플로(stack overflow)가 쉽게 발생해 메모리 부족으로 역행렬 계산이 불가능할 수 있다.

또 다른 문제점은 여인수 전개과정에서 작은 행렬로 분할하는 과정에서 동적 메모리 할당/해지가 반복적으로 일어나기 때문에 메모리 파편화(fragmentation) 문제가 발생할 수 있다. 결국 여인수 전개 방법은 극히 적은 메모리를 가진 아두이노에서는 과도한 재귀 호출과 메모리 파편화 문제로 역행렬 계산이 불가능할 수 있다.

- SVD 기반 역행렬 계산

SVD를 이용한 역행렬은 이래 공식으로부터 유도 된다.

$$A = U\Sigma V^T \tag{6}$$

여기 U, V는 직교 행렬이고, Σ는 특이값에 대한 대각행렬이다. 분해된 행렬 A에 대한 역행렬은 아래 수식으로 계산 가능하다.

$$A^{-1} = (V^T)^{-1}\Sigma^{-1}U^{-1} = V\frac{1}{\Sigma}U^T \tag{7}$$

SVD를 이용한 역행렬에서는 V, Σ, U 행렬에 대한 행렬곱셈을 최적화 하면 아두이노에서 어렵지 않게 역행렬을 구할 수 있다. SVD 알고리즘은 코드의 양이 많고, 반복연산을 통한 수렴 방식으로 계산된다. 가장 널리 쓰이는 SVD알고리즘을 보면 많은 반복문과 계산에 의해 U,V를 분해하는 과정이 구현되어 있고, 반복적인 메모리 할당/해지 코드가 없기 때문에 아두이노에 적합한 방법이다. 다만 코드가 여인수 전개 방법에 비해 복잡하기 때문에 컴파일되어 FLASH 공간에 적재 가능한지와 계산 시간에 대한 검증이 필요하다.

본 논문에서는 $V\Sigma^{-1}U^T$ 계산을 효율적으로 하기 위해 두 가지 방법을 사용했다. $V\Sigma^{-1}$에서 행렬곱 연산을 피하고자 $\overline{V} = V/\Sigma$에 대해서 먼저 계산하고, U도 별도의 전치행렬을 계산하지 않고 , $\overline{V}$ 와 U를 직접 계산함으로서 SVD 계산 이후 역행렬 계산 시 임시 메모리 생성을 최소화 하였다.

그리고 SVD의 경우 특이 값이 0이거나 매우 작은 값일 경우 랭크(Rank) 가 부족해 솔루션이 안정적이지 않을 수 있다. 이 문제는 통상적으로 식 (8)을 통해 해결할 수 있다[14].

$$A^{-1} = V\frac{1}{\Sigma + 0.0000001}U^T \tag{8}$$

- 가우스-조던 소거법

행렬로부터 직접적으로 역행렬을 구하는 방법으로 중첩 반복문으로 알고리즘 구현이 가능해 SVD 알고리즘에 비해 적은 코드로 역행렬 계산이 가능하다. 본 논문에서는 아두이노용으로 공개되어 있는 역행렬 라이브러리[13]를 사용하였다. SVD와 마찬가지로 반복적인 메모리 해당/해지 과정이 없기 때문에 메모리 파편화 문제도 발생하지 않는다.

2-3 행렬 결합법칙을 이용한 순차적 행렬 계산

역행렬이 구해지면 ELM 학습 파라미터는 행렬곱으로 계산 가능하다. 그러나 여기서 문제는 H 행렬과의 행렬곱이 다시 필요하다는 점이다. 은닉층의 출력을 계산할 때 학습 데이터가 많은 경우 메모리 문제가 발생하기 때문에 H 행렬을 저장하지 않았다. 본 논문에서는 최종 수식 $(HH^T)^{-1}Hy$ 을 계산할 때 H 행렬을 사용하지 않고 계산하기 위해 행렬 곱에 대한 결합 법칙을 사용해 식 (9)처럼 최적화 한다.

$$\begin{aligned}(HH^T)^{-1}Hy &= (((HH^T)^{-1}H)y) \\ &= ((HH^T)^{-1}(Hy))\end{aligned} \tag{9}$$

그리고 Hy는 식 (10)처럼 학습데이터 하나에 대해서 순차 방식을 통해 계산이 가능하다.

$$Hy = h_1 y_1 + h_1 y_1 + \ldots + h_n y_n \tag{10}$$

결합 법칙을 사용해 Hy를 먼저계산하고, 그 결과를 역행렬과 행렬곱을 하면 H 행렬을 저장할 필요 없이 ELM 솔루션을 계산할 수 있다. 이 과정에서 주의할 사항은 H 행렬을 보관하지 않기 때문에 HH^T계산할 때 Hy도 함께 계산해야 한다는 점이다.

Ⅲ. 실험

제안하는 방법의 우수성을 검증하기 위해 아두이노 UNO(2KB SRAM)와 아두이노 MEGA(8KB SRAM) 두 개의 보드에서 알고리즘을 구현하고, 수행시간을 측정하였다. 측정은 아두이노에서 기본적으로 제공하는 mills()함수를 사용해 밀리초(millisecond)로 측정하였다. 최적화 과정에서 근사화 기법을 사용하지 않기 때문에 기본 ELM 알고리즘과 동일하기 때문에 정확도 측정에 관한 실험은 진행하지 않았다.

실험에서 사용하는 데이터 셋은 GMM(Gaussian Mixture Model)을 사용해 가상적으로 생성한 데이터셋과 ELM성능을 테스트하기 위해서 사용했던 UCI 범용 데이터셋 [15]을 사용하였다. 제안하는 알고리즘의 성능 분석을 위해 가상의 GMM 데이터셋을 사용하는 방법이 타당한 이유는 학습 데이터 크기,

표 2. UCI 데이터셋 및 특성
Table 2. UCI dataset specification

DB	#train	#feature	#classes
Diabetes	512	8	2
Australian Credit	460	6	2
Liver	230	6	2
Banana	400	2	2
Brightdata	1000	14	2
Dimdata	1000	14	2
Mushroom	1500	22	2
Iris	100	4	3
Glass	142	9	6
Wine	118	13	3
Ecoli	224	7	8
Vowel	528	10	11
Vehicle	564	18	4
Segment	1540	19	7
Satimage	4435	36	6
Letter	13333	16	26
Shuttle	43500	9	7

학습데이터 수, 클래스 수를 자유롭게 변터 특징 차원 크기, 학습데이터 수, 클래스 수를 자유롭게 변경해가며 각각의 인자 변이에 대해 수행 속도 어느 정도 영향을 받는지 명확하게 분석할 수 있기 때문이다.

UCI 범용 데이터셋의 경우 제안하는 방법이 회귀(Regression) 문제가 아닌 분류(Classification) 문제만 다루기 때문에 분류 데이터셋 만 사용했고, 이 중 특징 차원의 수가 60 이하의 셋만 사용하였다. 표 2에 사용된 데이터셋의 이름(DB), 학습샘플 수(#train), 특징 차원 수(#feature), 클래스 수(#classes)를 표시하였다. 테스트데이터 에 대한 예측 값은 학습파라미터와 은닉층의 출력 값의 단순 내적(dot product)만으로 계산 가능해 별도의 수행시간을 측정하지 않았다.

3-1 순차적 역행렬 계산의 효율성 평가

첫 번째 실험으로 은닉층의 출력을 계산하는 시간을 측정하였다. GMM 데이터셋을 사용해 특징 차원은 10 ~ 30까지 변경하고, 샘플의 수는 10개 ~ 190까지 늘려가면서 계산 시간을 측정하여 표 3에 정리하였다. 메모리 부족으로 계산 불가능할 경우 error로 표기하였다. 실험 결과로부터 알 수 있는 사항은 다음과 같다.

제안하는 알고리즘인 식(4)는 아두이노 UNO에서는 15차원을 갖는 데이터를 아두이노 MEGA에서는 42 차원의 데이터셋에 대해서 처리가 가능하다. 반면 HH^T을 한번에 계산할 경우 아두이노 UNO에서 10차원 데이터셋 10개를 아두이노 MEGA에서는 10차원 데이터셋 170개까지는 가능하지만 여전

표 3. 은닉층 수행 시간 비교(측정 단위 ms)
Table 3. Comparison of running time for hidden layer(unit of measure ms)

	#feature	#train	HH^T	식 (4)
UNO	10	10	17	27
	10	30	53	83
	10	50	error	141
	10	100	error	283
	10	200	error	569
	10	300	error	1414
	15	100	error	574
	15	200	error	1149
	15	300	error	1721
	20	5	error	error
	20	10	error	error
	20	20	error	error
MEGA	10	90	168	258
	10	100	186	286
	10	150	280	431
	10	170	317	489
	10	190	error	546
	20	5	error	47
	20	10	error	94
	20	20	error	190
	30	40	error	817
	30	100	error	2061
	30	200	error	4126
	35	50	error	1368
	35	100	error	2753
	35	200	error	5515
	35	300	error	8253
	40	40	error	1404
	42	42	error	1619
	45	45	error	error

히 20차 원이 이상의 데이터에 대해서는 메모리 부족으로 계산이 불가능했다. 메모리가 부족이 발생한 상황은 두가지 였다. 식(3)의 경우 전체 데이터셋 저장을 위한 메모리 공간이 부족한 상황과 행렬곱 과정 중 SRAM이 부족한 상황이다. 식(4)의 경우는 행렬곱 과정에서 SRAM이 부족한 상황만 발생하였다.

계산 시간 관점에서는 HH^T 계산이 식 (4)에 비해 조금 빠른 것으로 나타났다. 이유는 기존 방법이 데이터를 한 번에 계산하는 반면 제안하는 알고리즘은 한 개씩 계산하기 때문이다. 특징 차원이 35이고, 학습샘플수가 300개 일 때 은닉층 계산만으로도 8초 이상(8253ms)이 걸린다. 많은 경우에 있어 신경망 학습을 자주하지 않고 간헐적으로 한다면 문제가 되지 않을 것이다. 평균적으로 1개의 학습데이터를 계산하는데 24ms가 필요하다. 표 3에서 얻을 수 있는 결과는 학습데이터가 2배로 늘어날 때 계산 시간도 거의 2배로 늘어남을 알 수 있다. 학습 샘플 수에 선형적으로 증가한다. 반면 특징 차원에 대해서는 데이터셋이 15/100(#feature/#train) 일 때 574ms, 30/100(#feature/#train) 일 때 2061ms으로 대략 특징 차원이 2배 증가하면 계산 시간은 4배로 늘어남을 알 수 있다.

표 4. 행렬식 호출 횟수 비교
Table 4. Comparison of the determinant call count

행렬식계산	det(2)호출횟수	여인수계산	det(2)호출횟수
det(3)	4	cofactor(3)	13
det(5)	86	cofactor(5)	511
det(7)	3620	cofactor(7)	28953
det(9)	260650	cofactor(9)	2606491
det(10)	2606501	cofactor(10)	28671501

ELM 알고리즘에서 메모리 복잡도와 계산 복잡도 관점에서 가장 큰 영향을 끼치는 부분은 역행렬 계산이다. 본 논문에서는 여인수 전개 알고리즘, SVD 기반 알고리즘, 가우스-조단 소거법 기반 알고리즘에 대한 성능을 비교하였다.

여인수 전개를 이용한 방법의 경우 최종적으로 det(2)를 호출하기 때문에 몇 번의 det(2)를 호출하느냐에 따라 재귀호출 횟수가 결정된다. 재귀호출이 많을 경우 스택 메모리가 부족해진다. 따라서 이번 실험에서는 행렬식과 여인수 전개과정에서 det(2) 호출 횟수를 측정하여 표 4에 표시하였다. det(9)를 계산하는데도 det(2)를 20만번 이상 호출하고, cofactor(9)를 계산하는 det(2)를 80만 번 호출하게 되는데 이러한 과도한 재귀 호출은 스택 메모리 부족으로 이어지게 될 확률이 높다. 따라서 여인수 전개 기반 역행렬 알고리즘으로는 행렬의 크기가 매우 작은 경우만 유효하다.

다음 실험으로 다양한 크기의 행렬에 대해 역행렬 계산 시간을 측정해 표 5에 정리하였다. 우선 여인수 전개 방법은 9×9에서 오류가 발생했고, 수행시간도 92초 이상이 걸렸다. 아두이노에서는 실질적으로 사용할 수 없는 알고리즘이다. 이에 비해 SVD 알고리즘은 9×9에서 167ms로 고속 계산이 가능하다. 아두이노 UNO에서 10×10, 아두이노 MEGA에서 24×24 크기의 행렬에 대해서 역행렬 계산이 가능하다.

가장 빠르고 효율적인 알고리즘은 가우스-조단 소거법을 이용한 방법이다. 15×15에서도 64ms로 SVD 알고리즘에 비해서도 매우 빠르며 더 적은 메모리를 사용한다. 아두이노 UNO에서 19×19, 아두이노 MEGA에서 42×42 크기의 행렬에 대해 역행렬을 계산할 수 있다.

표 3과 표 5를 종합해 보면 아두이노 UNO에서는 15차원의 데이터를 아두이노 MEGA에서는 42차원 데이터에 대해 ELM 알고리즘 계산이 가능하다는 결론을 얻을 수 있다. 그리고 최대 계산 가능한 크기에서 역행렬 계산은 1.5초가 필요하고 이는 학습데이터 수에 상관없다. 오히려 식 (4)를 계산하는데 계산 비용이 큰 것을 알 수 있다.

3-2 메모리 최적화 성능 분석

Hy를 한 번에 계산하지 않고 한 개의 데이터에 대해 식 (10)처럼 누적해서 계산하는 방식의 장점을 검증하기 위한 비교 실험을 후 표 6에 정리하였다.

표 5. 역행렬 계산 시간 비교(측정 단위 ms)
Table 5. Computation time comparison of three inverse matrix methods(unit of measure ms)

	행렬크기	여인수	SVD	가우스-조단
UNO	4x4	99	20	1
	5x5	334	39	2
	6x6	1516	61	4
	7x7	10567	85	7
	8x8	92392	127	10
	9x9	error	167	15
	10x10	error	212	19
	11x11	error	error	25
	15x15	error	error	64
	17x17	error	error	93
	19x19	error	error	129
	20x20	error	error	error
MEGA	20x20	error	1461	154
	23x23	error	2160	232
	24x24	error	2621	263
	25x25	error	error	298
	30x30	error	error	513
	40x40	error	error	1212
	42x42	error	error	1404
	43x43	error	error	1504
	44x44	error	error	error

표 6. Hy 최적화 실험 결과(측정 단위 ms)
Table 6. Hy optimization experiment results(unit of measure ms)

	#feature	#train	Hy	식 (10)
UNO	10	10	1	27
	10	30	3	84
	10	50	error	142
	10	100	error	286
	10	200	error	575
	15	100	error	580
	15	200	error	1163
	20	10	error	94
	20	20	error	191
MEGA	20	20	4	190
	20	60	12	582
	20	80	17	778
	20	90	19	876
	30	10	error	198
	30	30	10	609
	30	50	16	1024
	30	60	20	1231
	30	70	error	202
	30	100	error	290
	30	200	error	578
	45	100	error	433
	45	200	error	865

제한된 메모리를 갖는 아두이노에서 모든 학습데이터를 메모리에 저장해 계산하는 Hy의 경우 아두이노 UNO에서 10/30(#feature/#train)의 매우 적은 데이터에 대해서 Hy계산이 가능하고, 아두이노 MEGA에서는 20/90(#feature/#train), 30/60(#feature/#train)에서만 가능하다. 반면 제안하는 방법은 45/200(#feature/#train)에서도 Hy 계산이 가능하다.

수행시간 관련해서는 Hy가 단순계산이기 때문에 매우 빠르다. 한 번에 계산하는 Hy 방식이 제안하는 식 (10)보다 조금 빠르기는 하지만 45/20(#feature/#train)에서 865ms인 경우 한 개 데이터 당 4.3ms(=865/200)로 학습이 가능하다.

다음 실험은 제안한 ELM 알고리즘의 전체 학습 시간을 측정하였다. 아두이노 UNO와 아두이노 MEGA에서 각각 측정하였고, 특징 차원의 수는 5,10,15,20,30,40개까지 늘려가고, 학습데이터 수는 10개에서 500개까지 늘려가며 제안하는 알고리즘이 특징 차원과 학습데이터 수에 따라 어떤 영향을 받지는 측정해 표 7에 정리하였다.

아두이노 UNO에서는 최대 15차원의 특징을 갖는 데이터에 대해 학습이 가능했고, 아두이노 MEGA에서는 최대 42차원의 특징을 갖는 데이터를 학습이 가능했다. 5/100(#feature/#train)에서 5/200(#feature/#train)으로 학습데이터수가 2배 증가하면 학습시간은 104ms 에서 206ms으로 대략 2배정도 증가한다. 40/100(#feature/#train)에서 40/500(#feature/#train)으로 학습데이터수가 5배 증가하면 4916ms에서 19449ms로 대략 5배 정도 증가한다. 결론적으로 보면 학습 데이터 수에 대한 o(n)에 비례하는 것으로 보인다.

특징 차원에 관련해서는 5/10(#feature/#train)에서 10/10(#feature/#train)으로 2배 늘어날 때 13ms에서 50ms 로 4.6배 늘어났고. 10/100(#feature/#train)에서 40/100(#feature/#train)으로 4배 늘어날 때 322ms에서 4916ms으로 15.2배 늘어났다. 대략적으로 특징 차원에 대해서는 $o(n^2)$수준으로 볼 수 있다.

3-3 다중 클래스 실험

기본적으로 ELM 알고리즘은 이진분류기 이다. 다중 클래스 문제를 다루기 위해서 one-vs-rest 방법을 사용해 식 (11)처럼 다중 클래스로 확장할 수 있다.

$$[\alpha_1 \alpha_2 \alpha_3 \ldots \alpha_k] = (HH^T)^{-1} HY \quad (11)$$

여기서 Y는 지시자 행렬로(Indicator Matrix) 자신의 클래스에 속하면 1, 그렇지 않으면 -1을 원소로 갖는 행렬이다. 가장 많은 연산이 필요한 $(HH^T)^{-1}$에서 클래스에 대한 정보가 필요하지 않기 때문에 ELM 알고리즘은 클래스 수에 대해 계산 복잡도가 높지 않다.

다중 클래스의 경우 이진 분류 보다 많은 메모리를 필요로 하기 때문에 실질적으로 UNO에서는 의미가 없다고 판단해 측정하지 않았다.

표 7. 특징차원, 샘플수에 따른 성능 측정 결과(측정 단위 ms)
Table 7. Performance measurement results(#feature, #train, unit of measure ms)

	#feature	#train	계산 시간
UNO	5	10	13
	5	20	23
	5	30	32
	5	100	104
	5	200	206
	5	500	507
	10	10	50
	10	100	322
	10	500	1518
	15	15	156
	15	100	671
	15	500	3066
MEGA	20	100	1178
	20	500	5222
	30	100	2669
	30	500	11167
	40	100	4916
	40	500	19494
	42	100	5464
	42	500	21455
	43	43	error

특징 차원과 클래스 수에 따라 학습시간이 어떻게 영향 받는지 분석하기 위해 특징 수는 20, 30, 35, 40개로 늘려갔고, 클래스 수는 3개에서 36개까지 다양하게 변경해 가면 서 학습 시간을 측정하였다. 아두이노 MEGA라 하더라도 메모리가 충분하지 않기 때문에 고차원으로 갈수록 클래스 수를 줄여서 최대로 가능한 클래스수를 측정하였다. 샘플 수는 100개로 고정하였다. 표 8은 실험결과를 보여준다.

우선 20차원의 데이터에 대해서는 35개의 클래스까지 2초(2229ms)의 시간으로 학습이 가능했다. 30차원인 경우는 15개의 클래스까지, 40차원인 경우에는 3개의 클래스 분류만 가능했다. 클래스 수가 20/10(#feature/#classes)에서 20/20(#feature/#classes) 으로 2배 증가할 때 학습시간은 1.2배(=1754/1437)증가되었다. 30/5(#feature/#classes)에서 30/10(#feature/#classes)으로 2배 증가했을 때도 1.08배 (= 3055/2803), 35/4(#feature/#classes)에서 35/8(#feature/#classes)로 2배 증가했을 때 1.06배(=4025/3782)증가했다. 고차원으로 갈수록 클래스 수에 대해 매우 효율적임을 알 수 있다. 이는 주로 ELM이 특징 차원 수에 따라 계속 복잡도가 결정되기 때문이다.

3-4 UCI 데이터셋을 사용한 성능 평가

마지막 실험으로 UCI 데이터셋에 대한 실험을 수행하였다. 데이터셋이 크기 때문에 아두이노 MEGA에서만 학습시간을 측정하였다. 기존 ELM과 최적화된 ELM을 비교하였다.

표 8. 특징차원, 샘플수, 클래스수에 따란 성능 측정 결과(측정 단위 ms)
Table 8. Performance measurement results(#feature, #train, #classes, unit of measure ms)

#feature	#train	#classes	계산 시간
20	100	5	1277
20	100	10	1437
20	100	15	1597
20	100	20	1754
20	100	30	2071
20	100	35	2229
20	100	36	error
30	100	5	2803
30	100	10	3055
30	100	15	3307
30	100	16	error
35	100	4	3782
35	100	5	3844
35	100	8	4025
35	100	9	4085
35	100	10	error
40	100	3	4936
40	100	4	error

표 9. 제안하는 ELM 성능(UCI 공개용 데이터셋, 측정 단위 ms)
Table 9. Performance of the proposed ELM : UCI public datasets(unit of measure ms)

DB	ELM	Proposed ELM
Diabetes	error	1085
Australian Credit	error	617
Liver	error	313
Banana	113	105
Brightdata	error	5415
Dimdata	error	5412
Mushroom	error	18259
Iris	121	95
Glass	809	485
Wine	error	645
Ecoli	error	590
Vowel	error	2444
Vehicle	error	5134
Segment	error	16324
Letter	error	165276
Shuttle	error	152084

학습 데이터셋이 4만개 이상인 데이터셋도 있기 때문에 실험에서는 SD카드에 학습데이터를 저장하고, 이때 SD카드 모듈에 따라 읽는 속도가 다를 수 있기 때문에 이 시간을 제외하고 전체 학습 시간을 측정하였다. 또한 Shuttle의 학습데이터수가 43500개인데 아두이노의 int 범위가 −32,768 to 32,767이다. 따라서 unsinged int로 이용해 43500번 루프를 돌면서 SD카드에서 하나씩 데이터를 읽도록 했다. 실험 결과를 표 9에 표시하였다.

실험 결과를 보면 최적화되지 않은 ELM은 학습 차원과 학습데이터 수가 작은 Banana, Iris, Glass 셋에 대해서만 학습이 가능했고, 대부분의 데이터셋에서 학습이 불가능했다. 반면 제안하는 알고리즘은 16/26/13333(#feature, #classes, #train) 크기의 Letter 데이터셋에 대해서 대략 165초, 9/7/43500(#feature, #classes, #train)크기의 Shuttle 데이터셋에 대해서 대략 152초로 학습이 가능했다.

Ⅳ. 결론

본 논문에서는 정확성과 효율성이 입증돼 여러 분야에서 활용되고 있는 ELM 알고리즘을 메모리 제약이 심한 아두이노 UNO와 아두이노 MEGA에서 학습할 있는 방법을 제안하였다. 학습데이터 전체를 메모리에 저장하지 않고, 순차적 방식으로 행렬 계산을 할 수 있는 알고리즘을 통해 학습데이터 수에 상관없이 ELM 학습이 가능하도록 했다. 실험을 통해 메모리 용량이 2KB인 아두이노 UNO와 메모리 용량이 8KB인 아두이노 MEGA에서 각각 15차원, 43차원의 다중 클래스 학습이 가능함을 보였다.

감사의 글

본 연구는 2016년도 강남대학교 교내 연구비 지원 사업에 의하여 이루어진 연구로서, 관계부처에 감사드립니다.

참고문헌

[1] Faure, Bouyer, Mercier, Robitaille, McFadyen, Fortin Côté, Cardou, Gosselin, Bonenfant, Laurendeau, "Development of a virtual-reality system with large-scale haptic interface and accessible motion capture for rehabilitation", 2017 International Conference on Virtual Rehabilitation (ICVR), Pages: 1 - 2.

[2] Sukun Li, Avery Leider, Meikang Qiu, Keke Gai, Meiqin Liu, "Brain-Based Computer Interfaces in Virtual Reality", 2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud), Pages: 300 - 305.

[3] S.-h. Kim and D.-H. Shin, "Effects of whole body movements in using virtual reality headsets on visually induced motion sickness", Journal of Digital Contents Society, Vol. 18, No. 2, pp. 283-291, Apr. 2017.

[4] Adnan O. M, Abuassba, Dezheng Zhang, Xiong Luo, Ahmad Shaheryar, Hazrat Ali, "Improving Classification Performance through an Advanced Ensemble Based Heterogeneous Extreme Learning Machines", Comp. Int. and Neurosc. 2017: 3405463:1-3405463:11 (2017).

[5] G. Huang, S. Song, and K. You, "Trends in extreme learning machines: a review", Neural Networks, vol. 61, pp. 32.

[6] Z. Bai, G.-B. Huang, D. Wang, H. Wang, M.B. Westover, "Sparse extreme learning machine for classification", IEEE Transactions on Cybernetics (2014).

[7] G. Huang, S. Song, J. Gupta, C. Wu, "Semi-supervised and unsupervised extreme learning machines", IEEE Transactions on Cybernetics (2014).

[8] F. Cao, B. Liu, D.S. Park, "Image classification based on effective extreme learning machine", Neurocomputing, 102 (2013), pp. 90-97.

[9] K. Choi, K.A. Toh, H. Byun, "Incremental face recognition for large-scale social network services", Pattern Recognition, 45 (8) (2012), pp. 2868-2883.

[10] G.-B. Huang, Q.-Y. Zhu, C.-K. Siew, "Extreme learning machine: theory and applications", Neurocomputing, 70 (1) (2006), pp. 489-501.

[11] http://paulbourke.net/miscellaneous/determinant/

[12] http://www.public.iastate.edu/~dicook/JSS/paper/code/svd.c

[13] https://playground.arduino.cc/Code/MatrixMath.

[14] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman, "Applied Linear Regression Models", 3rd ed. Chicago, IL: Irwin, 1996.

[15] G.-B. Huang, H. Zhou, X. Ding, and R. Zhang, "Extreme learning machine for regression and multiclass classification", IEEE Trans. Syst. Man Cybern. Part B: Cybernetics, 42:513–529, 2012.

최권택(KwonTaeg Choi)

2001년 : 한림대학교 컴퓨터공학과 (공학학사)

2006년 : 연세대학교 컴퓨터과학과 (공학석사)

2011년 : 연세대학교 컴퓨터과학과 (공학박사-컴퓨터비전,패턴인식)

2011년～2015년: LG전자

2016년～현 재: 강남대학교 컴퓨터미디어정보공학부 교수

2017년～현 재: 강남대학교 소프트웨어응용학부 교수

※관심분야 : 가상현실, 모바일컴퓨팅, 기계학습, HCI