한국음향학회지 제36권 제6호 pp. 425~435 (2017)
The Journal of the Acoustical Society of Korea Vol.36, No.6 (2017)
https://doi.org/10.7776/ASK.2017.36.6.425

pISSN : 1225-4428
eISSN : 2287-3775

다중 코어 및 single instruction multiple data 기술을 이용한 심층 신경망 속도 향상

Improving the speed of deep neural networks using the multi-core and single instruction multiple data technology

정익주,[1†] 김승희[2]

(Ik Joo Chung[1 †] and Seung Hi Kim[2])

[1]강원대학교 IT 대학, [2]ETRI 음성지능연구그룹

(Received August 29, 2017; accepted November 29, 2017)

초 록: 본 논문에서는 다중 코어 ARM 프로세서의 NEON SIMD(Single Instruction Multiple Data) 병렬 명령어 및 다중 코어 병렬화를 통하여 심층 신경망의 피드포워드 네트워크 연산을 최적화하는 방안을 제시하였다. SIMD 병렬 명령어를 이용한 최적화의 경우에는 단계 별 최적화 과정에서의 속도 향상과 정밀도를 제시 하였다. 단일 코어 상에서 SIMD 병렬 명령어를 이용하여 구현된 결과는 C 컴파일러를 이용한 구현보다 2.6배의 속도 향상을 얻을 수 있었다. 또한 단일 코어 상에서 최적화된 코드를 다중 코어로 병렬화함으로써 5.7배~7.7배의 속도 향상을 얻을 수 있었다. 이상의 결과를 통하여 이동형 단말기에서도 연산량이 많은 심층 신경망 기술을 활용할 수 있는 가능성을 확인하였다.
핵심용어: SIMD (Single Instruction Multiple Data) 기술, 다중 코어, 병렬화, 심층 신경망

ABSTRACT: In this paper, we propose optimization methods for speeding the feedforward network of deep neural networks using NEON SIMD (Single Instruction Multiple Data) parallel instructions and multi-core parallelization on the multi-core ARM processor. As the result of the optimization using SIMD parallel instructions, we present the amount of speed improvement and arithmetic precision stage by stage. Through the optimization using SIMD parallel instructions on the single core, we obtain 2.6× speedup over the baseline implementation using C compiler. Furthermore, by parallelizing the single core implementation on the multi-core, we obtain 5.7×~7.7× speedup. The results we obtain show the possibility for applying the arithmetic-intensive deep neural network technology to applications on mobile devices.
Keywords: SIMD (Single Instruction Multiple Data) Technology, Multi-core, Parallelization, Deep neural network

PACS numbers: 43.72.Bs, 43.72.Ne

I. 서 론

기계 학습의 한 분야인 딥러닝이 주목을 받으면서, 여러 분야에서 딥러닝을 적용하기 위한 많은 연구가 진행되고 있다. 딥러닝은 매우 많은 양의 데이터를 이용하여 데이터에 포함된 구조 또는 표현을 심도 있게 학습하는 방식이다. 특히 기존의 방법으로는 잘 학습이 되지 않았던 비지도 학습에 매우 효과가 있는 것으로 알려져 있다.[1,2]

기존의 신경망의 경우 역전파를 이용하는 다층 퍼셉트론 방식이 도입되면서 부분적으로 좋은 성능을 보여주었으나, 신경망의 층 수가 증가되면서 발생하는 과적합과 기울기가 사라지는 문제로 인하여 그 한계를 드러내면서 잊혀지는 듯했다. 그러나 딥러닝이 위에서 언급한 문제를 어느 정도 해결하면서 소위 심층 신경망이 다시 주목을 받게 되었다.[3] 한편, 심층 신경망의 구조는 Fig. 1과 같이 기본적으로 다층 퍼셉

†**Corresponding author:** Ik Joo Chung (ijchung@kangwon.ac.kr)
Department of Electrical & Electronics Engineering, Kangwon National University, 1 Kangwondaehak-gil, Chuncheon, Gangwon 24341, Republic of Korea
(Tel: 82-33-250-6322, Fax: 82-33-256-6327)

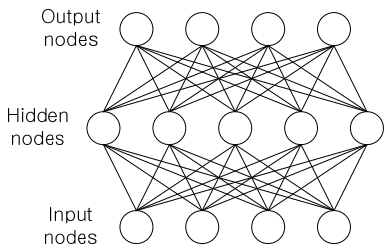

Fig.1. Neural network structure of the multi-layer perceptron.

트론과 유사한 구조를 가지고 있다. 하지만, 많은 양의 데이터를 이용하여 구조에 포함된 여러 파라메터들을 충분히 훈련 시키기 위해 기존의 다층 퍼셉트론보다 많은 은닉층을 가지고 있다는 특징이 있다.

딥러닝은 그 특성 상 많은 학습 데이터를 필요로 하며, 학습해야 할 구조가 복잡할 경우 이를 표현하기 위하여 여러 은익층과 각 층 당 많은 노드를 가지게 된다. 이럴 경우 많은 학습 데이터와 더불어 노드를 연결하는 수 많은 파라메터들의 증가는 곧 바로 훈련 및 인식 시의 연산량 증가로 이어진다. 연산 성능이 충분히 확보되지 않았던 심층 신경망 연구 초창기에는 이러한 연산량의 증가로 인하여 효과적인 연구를 진행하기 어려운 경우도 있었으나, 최근 GPGPU(General Purpose Graphics Processing Units)와 같은 고성능 연산 장비가 도입되면서 연산량 문제는 어느 정도 해결되었다. 현재 심층 신경망 관련 연구는 주로 GPGPU를 장착한 서버를 기반으로 이루어지고 있다.[4,5]

한편, 딥러닝의 응용이 활발해지면서 스마트폰과 같은 이동형 단말기에도 이 기술을 적용하고자 하는 시도가 이루어지고 있다.[6,7] 그러나 앞에서 언급한 딥러닝의 연산량을 이동형 단말기들이 감당하기에는 아직 이동형 단말기들의 연산 성능이 충분하지 못하다. 이런 문제점은 서버용 심층 신경망에 비하여 은닉층의 수나 층 당 노드 수를 줄이고, 이동형 단말기에 탑재된 응용 프로세서의 다양한 연산 기능을 충분히 활용함으로써 어느 정도 해결할 수 있다.

본 논문에서는 현재 가장 많이 사용되고 있는 응용 프로세서인 다중 코어 ARM 프로세서의 다중 코어를 이용한 병렬화와 NEON SIMD(Single Instruction Multiple Data) 병렬 명령어를 활용하여 심층 신경망의 피드포워드(Feedforward) 네트워크 연산을 최적화 하는 방안에 대하여 연구하였다. 논문의 구성은 2장에서 ARM 프로세서의 다중 코어와 NEON SIMD 명령어에 대하여 간략히 소개하고, 3장에서는 피드포워드 네트워크 연산을 구성하는 affine 매핑, tanh 활성화 함수, 소프트맥스 연산에 대한 단일 코어 상에서의 최적화 방안에 대하여 설명하고, 4장에서는 단일 코어 상에서 최적화된 코드를 다중 코어로 병렬화하는 방안에 대하여 설명한다. 5장에서 결론을 맺는다.

II. 다중 코어 및 SIMD 병렬 처리 기술

CPU의 성능을 향상시키기 위해 클럭을 높이는 방법이 한계에 다다르면서 이의 한 해결책으로 CPU에 포함된 코어의 수를 증가시키는 다중 코어 기술이 등장하였다. 다중 코어 기술은 데스크탑 및 서버용 CPU에 먼저 적용되기 시작하였고 현재는 대부분의 이동형 단말기용 응용 프로세서에도 적용되고 있다. 한편, 초기에는 단순히 기존의 CPU들을 하나의 다이에 집적하였으나, 이후 독자적인 소량의 (L1캐쉬 + 공유된 다량의 L2) 또는 L3 캐쉬의 구조를 갖춤으로서 캐쉬의 효용성을 높이고 있다. 또한, 최근에는 big-little 구조를 적용하여 고성능 코어와 저성능 코어를 동시에 집적하여, 실행되는 테스크의 성격에 따라서 코어를 효과적으로 활용할 수 있도록 하고 있다. 본 논문에서 사용한 ARM CPU의 경우에도 4개의 big 코어(Cortex-A57)와 4개의 little 코어(Cortex-A53)를 집적하고 있다.

CPU의 클럭을 향상시킬 경우에는 프로그램의 변경 없이도 성능의 향상을 얻을 수 있으나 다중 코어의 경우에는 성능 향상을 얻기 위해서는 기존의 프로그램을 병렬화할 필요가 있다. 이를 두고 "더 이상의 공짜 점심은 없다"라고 흔히 표현한다. 병렬화 작업은 비용과 시간을 상당히 요구하는 과정이기 때문에 값비싼 대가를 치루는 과정이기도 하다. 병렬 프로그램 방식으로 여러 기술이 존재하지만, 절대적으로 좋은 방법이 존재한다기 보다는 병렬화 하고자 하는 프로그램의 성격에 따라 적절한 기술이 적용되어야 한다. 본 논문에서는 다중 쓰레드 방식을 적용하여 다중 코어 병렬화를 시행하였다. 다중 쓰레드 방식은 본 논문에서 병렬화 하고자 하는 피드포워드 네트워크의 경

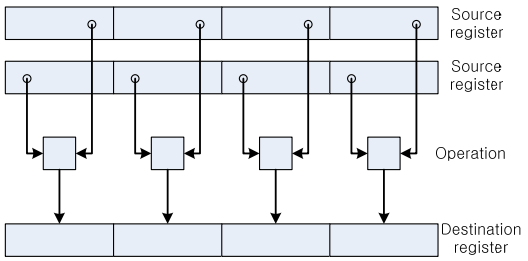

Fig. 2. Parallel operation using NEON SIMD instructions.

우처럼 동일 연산 구조를 다량으로 연산할 경우, 쓰레드를 이용하여 다중 코어에 골고루 분산하여 연산할 때 효과적인 방식이다.

SIMD는 명령어 레벨에서 병렬처리를 수행하는 기술로서 여러 데이터에 대하여 동일한 명령을 병렬로 처리할 수 있다. 인텔 CPU의 경우는 이전부터 MMX나 SSE와 같은 SIMD 계열의 명령어를 지원하고 있다. Cortex-A8 구조 이후의 ARM 프로세서는 NEON이라는 SIMD 유닛이 추가되었다. SIMD용 레지스터의 길이는 128비트이다. 8비트 데이터 16개, 16비트 데이터 8개, 32비트 데이터 4개, 64비트 데이터 2개를 동시에 연산할 수 있으며, 정수 연산뿐만 아니라 부동소수점 연산도 지원한다. Fig. 2는 4개의 데이터가 병렬로 연산되는 과정을 보여준다. 32비트 ARM 프로세서의 NEON 유닛은 128비트 레지스터를 16개 가지고 있었으나, Cortex-A57과 같은 64비트 ARM 프로세서부터는 레지스터의 수가 기존에 비해 2배로 확장되었다. 따라서 명령어 지연(latency)의 영향을 줄이기 위한 루프 unrolling이 용이해졌으며, 또한 레지스터를 저장하기 위한 문맥 저장(context save)의 필요성이 많이 줄었다.[8]

기본적으로 병렬 연산, 병렬 이동, 병렬 변환, 병렬 비트 조작이 지원되며, 테이블 인덱싱, zip & unzip과 같은 독특한 명령어도 지원된다. 대부분의 명령어 실행은 매 싸이클마다 이루어질 수 있으나, 결과가 최종 레지스터에 저장되는데 소요되는 기간인 명령어 지연은 명령어 따라 2~10 싸이클로 다양하며, 명령어 지연을 줄이기 위해서는 루프 unrolling이 필수적으로 활용되어야 한다.

III. SIMD 명령어를 이용한 병렬화

이번 절에서는 NEON SIMD 명령어를 이용하여 연산을 최적화는 방법을 제시한다. NEON 유닛은 모든 코어에 포함되어 있으므로 여기서는 단일 코어의 NEON SIMD 명령어를 이용하며 최적화할 것이며, 이렇게 최적화된 단일 코어 프로그램을 다중 코어로 확장하는 것은 다음 절에서 설명한다.

심층 신경망의 피드포워드 네트워크와 관련된 연산은 다음 세가지로 구성된다.

(1) Affine 매핑

각 층에서 입력, 웨이트(weight), 바이어스(bias)를 이용하여 Eq. (1)의 affine 매핑 출력을 계산한다.

$$y_j = b_j + \sum_i x_i w_{ij}, \tag{1}$$

여기서 i, j는 각각 입력 및 출력 노드의 인덱스이며, x_i 는 입력값, y_j 는 출력값, b_j 는 출력 노드의 바이어스 w_{ij} 는 i번째 입력 노드를 j번째 출력 노드로 연결하는 웨이트이다.

(2) tanh 비선형 활성화 함수

모든 은익층의 affine 매핑 출력은 Eq. (2)의 비선형 활성화 함수의 입력으로 사용된다.

$$z_j = \tanh(y_j) = \frac{1-e^{-2y_j}}{1+e^{-2y_j}}. \tag{2}$$

tanh 함수 연산 결과는 다음 층의 입력으로 사용된다.

(3) 소프트맥스 로그 함수

최종 출력층의 affine 매핑 출력은 Eq. (3)의 소프트맥스 함수의 입력으로 사용된다.

$$z_j = \frac{e^{y_j}}{\sum_i e^{y_i}}, \tag{3}$$

한편, 음성인식 응용을 포함한 많은 응용에서 로그 확률 및 비용 함수로 cross-entropy를 사용하기 때문에

최종적으로 소프트맥스 출력 값의 로그 값 $\log(z_j)$ 을 계산하게 된다.

먼저 tanh 함수와 소프트맥스 로그 함수에 대하여 먼저 설명하고 가장 많은 연산을 차지하는 affine 매핑은 마지막에 설명한다.

3.1 tanh 비선형 활성화 함수

C언어 표준 라이브러리에서 제공하는 수학 함수는 배정도(double precision) 연산을 수행한다. 따라서 최적화 결과의 정밀도 및 연산 속도는 C 언어 표준 라이브러리에서 제공하는 함수의 결과와 비교할 것이다. tanh 함수 및 소프트맥스 함수는 지수 함수로 이루어져 있으므로 먼저 지수 함수를 효율적으로 구하는 방식에 대하여 설명한다. NEON SIMD명령어를 이용한 지수 함수는 Reference [9]에서 제공하고 있다. 그러나 지수 함수를 구현하는 연산 과정이 실행 순서와 밀접한 인과 관계를 가지게 되기 때문에 연산 과정에서 많은 명령어 지연이 발생하게 된다. Reference [9]의 경우도 NEON SIMD 명령어를 사용하여 연산 속도를 향상시키기는 하였지만, 여전히 많은 명령어 지연이 발생한다. 한편, 심층 신경망 연산에서는 은닉층의 모든 노드에 대하여 활성화 함수가 적용되기 때문에 동시에 많은 횟수의 지수 함수 연산을 하게 된다. 이점에 착안하여 NEON SIMD 명령어를 최대한 활용하고 명령어 실행 순서를 조정하여 명령어 지연을 최대한 줄이도록 루프 unrolling을 수행하여 프로세서의 성능을 극대화할 수 있다. 이렇게 명령어 실행 효율을 높이도록 루프 안에서 한 번에 4개씩 지수 연산을 수행하는 어셈블리 함수를 구현하였으며 Table 1은 그 성능을 비교한 것이다.

실험은 음성인식에 적합한 환경에서 실시하였다. 피드포워드 네트워크의 입력으로 사용되는 특징벡터로는 40차의 필터뱅크 분석값을 사용하였는데 이

Table 1. Performance comparison of exponential function.

Implementation	Relative time to process	Arithmetic error (%)
(float) exp	100	2.19e-6
expf	63.1	2.30e-6
exp_neon	25.5	3.69e-6
exp4_neon	10.8	3.69e-6

럴 경우, 피드포워드 네트워크의 affine 매핑 결과값은 대부분이 -10~10에 분포하고 있기 때문에 -10~10 사이 값을 1000000 등분한 값들에 대하여 1000번 반복 실시한 정밀도와 수행 시간을 구하였다. **(float)exp**은 C언어 표준 라이브러리의 배정도 지수 함수의 결과를 단정도(single precision)인 float형으로 캐스팅한 것이며, **expf**는 단정도 C언어 표준 라이브러리 함수를 사용한 결과이다. 일부 임베디드 프로세서의 C언어 표준 라이브러리의 경우에는 배정도 및 단정도 수학 함수를 모두 제공하는데, ARM용 GNU 컴파일러(GCC)의 경우도 두 함수를 모두 지원한다. 오차는 다음과 같이 계산하였다.

$$\frac{Average\ Error}{Average\ Magnitude}\times 100\ \% = \frac{\sum_i |y_i - \tilde{y}_i|}{\sum_i |y_i|}\times 100\ \%, \quad (4)$$

여기서 y_i 는 참값에 해당하는 배정도 연산 결과이고 $\tilde{y}_i$ 는 해당 구현 방식의 결과이다. 수행 시간은 가장 느린 **(float)exp**을 100으로 했을 때의 상대 속도이다. Table 1에서 보듯이 배정도 함수를 float형으로 캐스팅하는 것은 단정도 함수에 비해 수행시간은 오래 걸리는 반면 정밀도 향상은 미미하다. 따라서 임베디드 시스템에서는 단정도 함수를 사용하는 것이 효율적이다. 한편, NEON SIMD 어셈블리 명령어를 이용하여 구현한 **exp_neon**가 C언어 표준 라이브러리 함수에 비하여 훨씬 빠른 속도를 보이며, 특히 본 논문에서 어셈블리 함수로 구현한 **exp4_neon**은 기존의 **exp_neon**에 비하여 2.5배 정도 빠른 수행 속도를 보인다. 연산 오차는 모든 방식이 무시할 정도로 작았다.

tanh 함수는 앞서 구현한 **exp4_neon**을 이용하여 최적화할 수 있다. 본 논문에서는 **exp4_neon**을 이용한 tanh 함수의 구현뿐만 아니라, 룩업(lookup) 테이블을 이용한 tanh 함수 역시 NEON SIMD 명령어를 이용하여 구현해 보았다. tanh 함수는 그 값의 절대값이 1로 제한되기 때문에 룩업 테이블을 이용하여 구현하기 적합하다. 룩업 테이블을 작성하기 위한 입력의 범위는 4.28 Q-포멧의 고정 소수점 표현을 사용하기 위해 -8~8로 설정하였고, 테이블의 크기는 65536으로 설정하였

Table 2. Performance comparison of tanh function.

Implementation	Relative time to process	Arithmetic error (%)
(float) tanh	100	1.55e-6
tanhf	66.4	1.67e-6
tanh4_neon	10.2	3.81e-6
tanh4_table	3.4	6.56e-4
tanh16_table	2.3	6.56e-4

다. tanh함수의 경우 입력의 절대값이 4정도만 되어도 출력의 절대값이 거의1이 되기 때문에 입력 범위를 -8~8로 설정해도 포화에 의한 오차는 거의 없다. 룩업 테이블 방식에서는 입력 값을 테이블의 인덱스로 변환하는 과정이 중요한데, NEON 유닛은 이 과정을 한 번에 4개씩 동시에 변환할 수 있도록 하는 효율적 명령어를 제공한다. 변환 과정은 다음과 같다.

[1단계] **FCVTZS** 명령어를 이용하여 4개의 float를 4개의 4.28 Q-포맷으로 변환

[2단계] **SQXTUN** 명령어를 이용하여 4개 4.28Q를 4개의 4.12 Q-포맷으로 변환

[3단계] **SADDW** 명령어를 이용하여 4개의 4.12Q를 0~65535의 인덱스로 변환

위와 같이 단 3개의 어셈블리 명령어로 4개의 float형 변수를 4개의 테이블 인덱스로 변환할 수 있다. 또한 이러한 명령어들의 장점은 변환 과정에서 입력값이 테이블 변환이 수용하는 범위를 넘어서도 자동으로 포화(saturation)시키는 기능을 가지고 있기 때문에 오차를 최소화 하게 된다. Table 2는 구현된 tanh 함수의 성능을 비교한 것이다.

(float)tanh은 C언어 표준 라이브러리 함수의 배정도 함수를 이용하여 수행한 결과를 float형으로 캐스팅한 것이고 **tanf**는 단정도 함수를 이용하여 구한 결과이다. **tanh4_neon**은 앞에서 구현한 **exp4_neon**을 이용하여 tanh 값을 루프 안에서 한 번에 4개씩 구하는 함수이며, **tanh4_table**는 앞에서 설명한 룩업 테이블 방식으로 구현한 것이다. **tanh16_table**은 NEON 명령어 지연을 최소화하기 위하여 루프 안에서 한 번에 16개씩 변환하도록 보다 많은 루프 unrolling을 적용한 함수이다. Table 2를 통하여 NEON SIMD명령어를 적용한 방식이 기존의 C언어 표준 라이브러리 함수들의 비하여 상당히 빠르다는 것을 알 수 있다. 한편, 룩업 테이블 방식의 경우는 연산 속도가 가장 빠른 반면 연산의 특성 상 연산 오차가 다소 증가하게 된다.

3.2 소프트맥스 로그 함수

음성인식 응용에서는 최종적으로 로그 확률을 활용하기 때문에 Eq. (3)으로부터 Eq. (5)를 얻게 된다.

$$\log\left(\frac{e^{y_j}}{\sum_i e^{y_i}}\right) = y_i - \log\left(\sum_i e^{y_i}\right). \quad (5)$$

Eq. (5)에서 $\log\left(\sum_i e^{y_i}\right)$ 은 미리 한 번만 계산해 놓으면 되기 때문에 전체 출력 노드의 로그 값을 계산하기 위해서는 한 번의 로그 연산만을 필요로 한다. 한편, Eq. (5)에서는 지수 값들을 누적하여 더하기 때문에 앞서 구현한 **exp4_neon** 방식을 약간 변경하여 4개의 지수 값을 구함과 동시에 이를 누적하여 더하는 별도의 어셈블리 함수를 작성하였다. 로그 함수의 경우에는 전체 심층 신경망 연산에서 한 번만 적용되기 때문에 최적화에 큰 영향을 미치지 않으므로 C언어 표준 라이브러리 함수를 그대로 사용해도 되지만, 본 논문에서는 로그 함수 역시 NEON SIMD 명령어로 작성된 함수를 사용하였다. 사용된 로그 함수의 성능은 다음과 같다.

(float)log은 C언어 표준 라이브러리 함수의 배정도 이용하여 수행한 결과를 float형으로 캐스팅한 것이고 **logf** 는 단정도 함수를 이용하여 구한 결과이다. NEON SIMD 명령어를 이용한 **log_neon**의 결과는 배정도C함수보다는4.5배, 단정도C함수 보다는2.5배 정도 빠르다.

지수 함수 자체는 범위가 제한되어 있지 않기 때문에 tanh함수와 달리 소프트 맥스 함수를 룩업 테이블 방식으로 구할 수는 없다. 따라서 기본적으로는 앞에서 설명한 **exp4_neon** 방식을 이용하여 지수 함수를 효과적으로 계산함으로써 소프트맥스 함수를 빠르게

Table 3. Performance comparison of log function.

Implementation	Relative time to process	Arithmetic error (%)
(float) log	100	2.23e-6
logf	54.6	2.24e-6
log_neon	22.3	4.98e-6

계산할 수 있다. 한편, 소프트맥스 함수가 가지는 shift invariant 특성을 이용하면 소프트맥스 로그 함수를 룩업 테이블 방식으로 구할 수 있다. 소프트맥스 함수는 임의의 k에 대하여 Eq. (6)이 성립한다.[10]

$$\frac{e^{y_j}}{\sum_i e^{y_i}} = \frac{e^{(y_j-k)}}{\sum_i e^{(y_i-k)}}. \tag{6}$$

또한, Eq. (6)을 Eq. (5)에 적용하면,

$$\log\left(\frac{e^{y_j}}{\sum_i e^{y_i}}\right) = (y_i - k) - \log\left(\sum_i e^{(y_i-k)}\right). \tag{7}$$

본 논문에서는 Eqs. (6)과 (7)의 성질을 이용하여 룩업 테이블 방식의 소프트맥스 로그 함수를 구하는 방법을 제안한다. 제안된 룩업 테이블 방식은 다음과 같다.

(1) 우선 Fig. 3처럼 지수 함수의 입력 범위를 정하여 지수 함수 테이블을 작성한다. 본 논문에서는 입력의 범위 $-M \sim M$에 대하여 테이블의 크기를 65536의 지수 함수 룩업 테이블을 생성하였다. M은 8로 설정하였다.

(2) 출력 노드의 affine 매핑 값 y_i 들의 최고치 $y_{\max}$를 구한다.

(3) $M = y_{\max} - k$ 에 해당하는 k 를 구하여 모든 i에 대하여 $y_i' = y_i - k$ 를구한다. 이렇게 되면 $-\infty < y_i' \le M$을만족한다.

(4) 룩업 테이블을 이용하여 $e^{y_i'}$ 을 구한다. 이때 y_i' 이 $-M$보다 작을 경우 테이블 인덱스는 0이 되고,

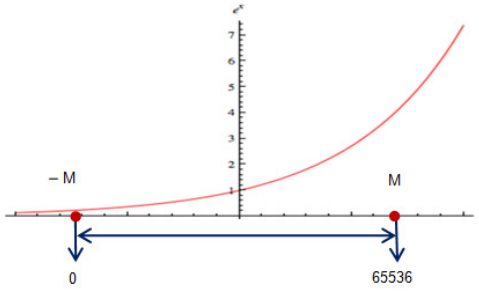

Fig. 3 Exponential function.

Table 4. Performance comparison of softmax-log function.

Implementation	Relative time to process	Arithmetic error (%)
(float) (double)	100	2.13e-6
(single)	63.2	2.46e-5
softmax_neon	12.3	3.45e-6
softmax_table	6.9	1.20e-5
softmax_table2	6.9	1.10e-5

e^{-M} 으로 포화된다. 룩업 테이블을 이용하여 구한 $e^{y_i'}$ 를 이용하여 $\sum_i e^{y_i'}$ 을 계산하면, 최종적으로 Eq. (7)을 계산할 수 있다.

Eq. (7)에서 필요로 하는 것은 각각의 지수값이 아니라 모든 지수값의 합이므로 최대값인 e^{M} 와 최소값인 e^{-M} 의 비를 고려할 때, e^{-M} 보다 작은 값들이 e^{-M} 로 포화되어도 최종 지수 합에 거의 영향을 주지 못한다. 이는 실험을 통하여 좀 더 살펴볼 것이다. Table 4는 구현된 소프트맥스 로그 함수의 성능을 보여준다.

(float)(double)은 연산 과정에서 배정도 C언어 표준 라이브러리의 지수 함수 및 로그 함수를 사용하여 연산한 후, 최종적으로 float형으로 캐스팅한 것이며, **(single)**은 단정도 C 언어 표준 라이브러리 함수를 사용하여 연산한 결과이다. **softmax_neon**은 Eq. (7) 연산에서 앞서 구현한 **exp4_neon** 및 **log_neon** 을 이용하여 계산한 결과이고, **softmax_table**은 앞에서 설명한 룩업 테이블을 이용하여 얻은 결과이다. 실험을 위하여 입력 값으로 포화가 발생하는 경우를 고려하여 테이블의 입력 범위 -8~8보다 넓은 -10~10사이의 값을 1000000등분한 값을 이용하여 소프트맥스 로그 확률을 구하였다. NEON SIMD명령어로 구현된 결과들이 매우 빠른 속도를 보인다. 뿐만 아니라 제안된 **softmax_table은** 룩업 테이블 방식이면서도 비교적 높은 정밀도를 보인다. **softmax_table2**는 앞에서 언급한 테이블의 입력 범위를 넘어서는 입력들이 성능에 어떤 영향을 미치는 지를 실험한 것이다. **softmax_table2**는 포화가 보다 많이 발생하도록 -16~16에 해당하는 입력을 사용하였다. Table 4의 결과를 보면 **softmax_table**과 **softmax_table2**의 결과가 거의 차이가 없음을

알 수 있다. 이를 통해 앞서 언급한 바와 같이 제안된 룩업 테이블 방식이 입력의 포화와 관계 없이 거의 일정한 정밀도를 보임을 알 수 있다.

3.3 Affine 매핑

Eq. (1)의 affine 매핑은 전체 심층 신경망 피드포워드 네트워크 연산의 대부분을 차지한다. 연산의 핵심적인 부분은 입력 값과 웨이트를 읽어 들인 후 이를 이용하여 MAC 연산을 수행하는 것이다. 바이어스는 MAC 연산에서 누적기로 사용할 레지스터의 초기값으로 설정함으로써 덧셈 연산을 줄일 수 있다. NEON SIMD명령어를 이용하여 4개의 입력, 4개의 웨이트를 각각 한 번의 로드 명령어를 이용하여 읽어 들인 후, FMLA 명령어를 이용하여 4개의 MAC 연산을 동시에 수행할 수 있다. 다음은 가장 안쪽 루프의 핵심 어셈블리 코드의 일부이다.

```
ld1   {v1.4s}, [x5],16       // load 4 inputs
ld1   {v2.4s}, [x6],16       // load 4 weights
flma  v0.4s, v1.4s, v2.v4    // 4 MAC operations
```

FMLA 명령어 지연이 10이기는 하지만, 메모리 로드에 상당한 시간이 소요되기 때문에 루프 수행 과정에서 FMLA 명령어 지연 영향이 크지는 않다. 그래도 충분한 NEON 레지스터를 활용하고, 루프의 반복 횟수 및 메모리 로드 명령어 실행 횟수를 줄이기 위하여 루프 unrolling은 필수적이다. 따라서 실제 구현에서는 16개의 입력을 한 루프에서 처리하도록 구현하였다. 즉, 아래와 같이 한 번의 루프 연산에서 16개의 MAC 연산을 수행한다.

```
ld1 {v18.4s,v19.4s,v20.4s,v21.4s},[x5],64   // load 16 inputs
ld1 {v22.4s,v23.4s,v24.4s,v25.4s},[x6],64   // load 16 weights
fmla v2.4s, v18.4s, v22.4s                  // 4 MAC operations
fmla v3.4s, v19.4s, v23.4s                  // 4 MAC operations
fmla v4.4s, v20.4s, v24.4s                  // 4 MAC operations
fmla v5.4s, v21.4s, v25.4s                  // 4 MAC operations
```

실험을 위하여 비교적 규모가 큰 서버용 심층 신경망 피드포워드 네트워크(DNN_0)과 임베디드용으로 규모를 줄인 심층 신경망 피드포워드 네트워크(DNN_1) 두 종류의 심층 신경망 피드포워드 네트워크을 사용하였다. 각각의 심층 신경망 모델은 Table 5에 나와 있다.

괄호는 DNN_1의 모델을 의미한다. **# of Blocks**는 블록 affine 매핑이 적용되었을 경우, 블록의 수이다. Table 6은 각 구조에 필요한 연산 횟수를 보여준다. 이번 절에서는 심층 신경망 피드포워드 네트워크 전체 연산에서 affine 매핑을 포함하여 각 부분이 얼마나 연산에 영향을 미치는 지를 통하여 최적화의 정도를 살펴볼 것이다 연산 속도는 실제 단말기에서 실행 시 측정한 소요 시간을 이용하여 RTF(Real Time Factor)로 표기하였다. 실험에 사용한 단말기는 갤럭시 S6이다.

Tables 7과 8의 **Optimization Level** 컬럼 부분에서 + 기호는 이전 단계의 최적화에 추가되어 최적화가 이루어진다는 것을 표기한 것이다. 표를 통해 tanh 및 소프트맥스 로그는 전체 연산에서 차지하는 비중이 크지 않음을 알 수 있다. 따라서 정밀도에 좀 더 비중을 두기 위하여 tanh 및 소프트맥스 로그는 룩업 테이블 방식이 아닌 **exp4_neon** 기반으로 구현한 방식을 사용하였다. 예상대로 affine 매핑 최적화에서 가장 속도가 많이 향상되는 것을 볼 수 있다. 표에서 **Optimization Level** 컬럼의 **C Baseline**은 C언어로 구현된 baseline 코

Table 5. Models of the deep neural network.

Layer	# of Input nodes	# of Output nodes	# of Blocks
0	600	900	1
1	150	3762(1254)	6
2	627(209)	3762(1254)	6
3	627(209)	3762(1254)	6
4	3762(1254)	1536(512)	1
5	1536(512)	1536(512)	1
6	1536(512)	1536(512)	1
7	1536(512)	7508(3997)	1

Table 6. Number of arithmetic operations by the deep neural network model.

Model	# of Affine Mapping MAC operations	# of tanh operations	# of Softmax-Log operations
DNN_0	27,851,160	15,894	7,508
DNN_1	4,465,072	5,298	3,997

Table 7. Stage-by-stage optimization results of DNN_0.

Optimization level	RTF	Arithmetic error (%)
C Baseline	6.32	4.04e-5
Tanh Optimization	6.22	4.06e-5
+Softmax-Log Optimization	6.08	1.99e-4
+Affine Mapping Optimization	2.94	2.09e-4
+prefetch using PRFM	2.44	2.09e-4
+Half-precision Weight	1.52	5.81e-2
+prefetch using PRFM (Multi-core)	1.10	2.09e-4

Table 8. Stage-by stage optimization results of DNN_1.

Optimization level	RTF	Arithmetic error (%)
C Baseline	1.09	3.32e-5
Tanh Optimization	1.04	3.49e-5
+Softmax-Log Optimization	0.99	5.51e-5
+Affine Mapping Optimization	0.49	7.00e-5
+prefetch using PRFM	0.40	7.00e-5
+Half-precision Weight	0.25	4.16e-2
+prefetch using PRFM (Multi-core)	0.14	7.00e-5

드를 GCC 최적화 레벨 3으로 컴파일한 결과이다(즉, C컴파일러를 이용하여 얻을 수 있는 가장 빠른 속도를 baseline 코드의 성능으로 삼았다. 참고로 최적화 레벨 3은 컴파일러가 최적화를 수행할 때 NEON SIMD 명령어를 이용하여 나름대로의 최적화를 수행한다). 반면, **Arithmetic Error** 부분은 C 언어로 구현된 baseline 코드를 GCC 최적화 레벨 0로 컴파일한 결과와 비교한 것이다(최적화 레벨 0으로 컴파일할 때 주어진 연산의 순서를 충실하기 유지하기 때문에 가장 높은 정밀도를 얻을 수 있다).

한편, 실제 작성된 어셈블리 코드 상에서 affine 매핑 최적화에 소요되는 명령어 횟수와 해당 명령어들의 실행 클럭수를 고려할 때, 연산 소요 시간이 예상 소요 시간보다 훨씬 긴데, 이는 로드 명령 시 메모리 접근 시간이 길기 때문이다. 로드 명령은 데이터가 캐쉬에 없을 경우 외부 메모리에서 읽어오게 되는데, 이 경우 100 클럭 이상 소요가 된다. Affine 매핑 연산에서 입력 데이터는 반복하여 읽혀지게 되는데, 캐쉬가 되기 때문에 문제가 되지 않으나 웨이트는 단발성으로 읽혀지게 되고 그 양이 방대하기 때문에 거의 캐쉬가 되지 않다. 특히, 임베디드용 프로세서의 경우에는 많아야 1MB 정도의 L2 캐쉬를 가지기 때문에 웨이트가 캐쉬되는 것은 현실적으로 거의 불가능하다. 그런데 affine 매핑 연산 부분을 살펴보면 입력과 웨이트를 읽은 후, 이렇게 읽은 데이터들을 이용하여 단지 한 번의 MAC 연산을 할 뿐이다. 즉, affine 매핑 연산은 메모리-접근 집중적인 성질을 가지고 있다. 실제로 NEON SIMD 명령어를 이용하여 최적화한 코드에서 입력과 웨이트를 로드하는 부분만을 남기고 나머지 부분을 모두 주석처리 한 상태로 수행할 경우 전체 소요 시간의 90 % 이상이 소요되었다. 이는 대부분의 시간이 메모리를 읽어 들이는데 소요되며 더 이상 연산 최적화의 여지는 별로 남아 있지 않다는 것을 의미한다. ARM 프로세서의 경우는 이런 메모리 접근의 문제점을 해결하기 위하여 prefetch 명령어를 제공한다. 다만, prefetch는 향후 사용될 데이터를 예측하여 캐쉬에 미리 가져다 놓는 기능인데, 캐쉬 miss 문제를 완전히 해결하는 것은 아니고 어느 정도 보완하는 기능을 제공한다.[8] 또한 얼마나 앞에 있는 코드나 데이터를 prefetch해야 하는지는 다소 경험적이기 때문에 시행착오를 거쳐 최적의 prefetch를 결정해야 한다. Tables 7과 8에서 prefetch를 수행하는 PRFM 명령을 추가했을 때, 속도가 상당히 향상되는 것을 볼 수 있다. 그러나 근본적으로 읽어 들이는 웨이트의 양을 줄이거나, GPU와 같이 다중 버스 구조를 이용하여 동시에 다량의 웨이트를 읽어 들이지 않는 한, 더 이상의 최적화는 한계가 있게 된다.

과도한 메모리 접근 문제를 해결하기 위한 방법으로 웨이트를 표현하는 워드의 길이를 줄이는 방법이 있다. 앞의 모든 구현에서 웨이트는 32비트의 단정도 부동소수점 방식으로 표현되었다. 최근 딥러닝을 고정소수점 방식으로 구현하는 방법들이 제안되고 있는데, 이 경우 통상 웨이트는 32비트보다 작은 워드 길이의 고정소수점 방식으로 표현된다. 이럴 경우 웨이트를 읽어들이는 총량은 32비트의 단정도 부동소수점 방식에 비하여 줄어들기 때문에 메모리 접근 문제가 완화될 수 있다. 고정소수점 방식의 구현이 최종 인식 결과에 큰 영향을 주지 않는 것으로 알려져 있지만, 부동소수점 방식에 비하여 구현이 복잡하다는 근본적인 문제가 있다.[6,11] 또한 피드포워드 네트워크

연산에서 활성화 함수 연산을 포함한 모든 부분을 고정소수점 방식으로 연산하기에는 어려움이 따르기 때문에, 경우 따라서는 고정소수점을 부동소수점으로, 부동소수점을 고정소수점으로 변환하는 작업이 수행되어야 한다. 본 논문에는 웨이트의 워드 길이를 줄이는 방법으로 최근 IEEE754 규격에 포함된 반정도(half-precision) 부동소수점 방식을 이용하였다. 반정도 부동소수점은 부호를 위해 1비트, 지수를 위해 5비트, 가수를 위해 10비트, 총 16비트로 표현되기 때문에 단정도 부동소수점에 비하여 웨이트를 위한 메모리를 절반으로 줄일 수 있으며, 메모리 접근 문제도 절반으로 줄어든다. 최근의 몇몇 CPU및 GPU들은 반정도 부동소수점 연산을 지원한다. 뿐만 아니라 GPU의 경우는 반정도 부동소수점 연산 지원이 딥러닝 연산에서 가장 효과를 볼 것으로 기대하고 있다.[12] NEON 유닛은 반정도 부동소수점 연산을 지원하지는 않지만 반정도 부동소수점과 단정도 부동소수점 사이의 변환을 해주는 SIMD 명령어를 지원한다. 본 논문에서는 이 기능을 이용하여, 최소한의 정도(precision) 변환 추가만으로 기존 코드의 변경 없이 메모리 접근 문제를 상당 부분 해소할 수 있는 방안을 제안하였다. 제안된 방식은 다음의 두 과정이 필요하다.

(1) 기존의 단정도 부동소수점 방식으로 저장되어 있는 웨이트를 반정도 부동소수점으로 변환하여 저장한다. 이 부분은 미리 수행되는 과정이다.

(2) 실행 시에는 반정도 부동소수점 방식의 웨이트를 읽어들인 후 이를 NEON SIMD 명령어를 이용하여 단정도 부동소수점 방식으로 변환한다. 이렇게 되면 이후 단정도 부동소수점 방식으로 계산하는 기존의 코드에서 동일하게 동작하게 된다.

SIMD 명령어인 **fcvtl** 및 **fcvtl2** 명령어를 연속 실행하므로서 아래의 예와 같이 8개의 반정도 부동소수점을 8개의 단정도 부동소수점으로 바꿀 수 있다. 두 명령이 연속 실행되는데 소요되는 클럭은 3클럭이다.

```
fcvtl    v22.4s, v30.4h
fcvtl2   v23.4s, v30.8h
```

따라서 실제로 1개의 반정도 부동소수점을 변환하는데는 3/8클럭 밖에 소요되지 않으며 이는 전체 웨이트들을 로드하는 시간에 비하면 무시할 정도이다. 최적화 결과는 Tables 7과 8의 **+Half-precision Weight**에 나와 있다. RTF가 이전 최적화에 비해 약 63 % 정도로 줄었음을 보여주고 있다. 이는 앞서 설명한 바와 같이 읽어 들이는 웨이트의 총량이 절반으로 줄은 데서 기인하는 것이다. 반면, 웨이트를 반정도로 표현하였기 때문에 오차는 증가한다. 최근 연구에 의하면 딥러닝의 연산 정밀도가 떨어져도 인식률에는 큰 영향은 주지 않는 것으로 알려져 있으며,[13] 따라서 고정소수점 연산 또는 반정도 부동소수점 연산을 통해 하드웨어의 비용을 줄이려는 시도가 이루어지고 있다. 본 논문에서 제안한 방법은 전적으로 반정도 부동소수점 연산하는 방법보다는 높은 연산 정밀도를 유지한다. 왜냐하면 제안된 방식은 메모리 접근을 줄이기 위해 웨이트를 반정도 표현으로 저장한 것이며 실제 연산에서는 이 반정도 웨이트를 읽어들인 후 단정도 표현으로 변환한 후 사용하기 때문에 이후 연산에는 단정도 부동소수점 연산을 하기 때문에 웨이트를 반정도로 표현하고 연산 자체를 반정도 부동소수점 연산을 하는 경우에 비해서는 보다 정확한 연산 결과를 주기 때문이다.

IV. 다중 코어를 이용한 병렬화

다중 코어 응용 프로세서의 경우는 프로그램의 병렬화를 통하여 하드웨어 자원을 충분히 활용할 수 있다. 음성인식의 경우 프레임 단위로 피드포워드 네트워크를 수행하게 되기 때문에 이를 병렬화하는 것은 비교적 간단하다. 즉, 각각의 프레임을 서로 다른 코어에서 수행되도록 하면 된다. 다만, 단일 채널을 처리하는 음성인식기에서 한 프레임을 하나의 코어에서 충분히 실시간 처리할 수 있는 경우라면 다중 코어 병렬화는 큰 의미가 없다. 왜냐하면, 한 코어에서 해당 프레임을 처리하는 동안 다른 코어들이 처리할 프레임들이 아직 입력되고 있지 않기 때문이다. 그러나 다음과 같은 경우 다중 코어를 이용한 병렬화가 필요하다.

(1) 하나의 코어에서 해당 프레임을 실시간 처리하지 못할 경우
(2) 일정한 길이의 음성데이터가 버퍼링되어 있는 경우
(3) 다채널 음성 데이터를 처리하는 경우
(4) 오프라인 방식으로 처리하는 경우

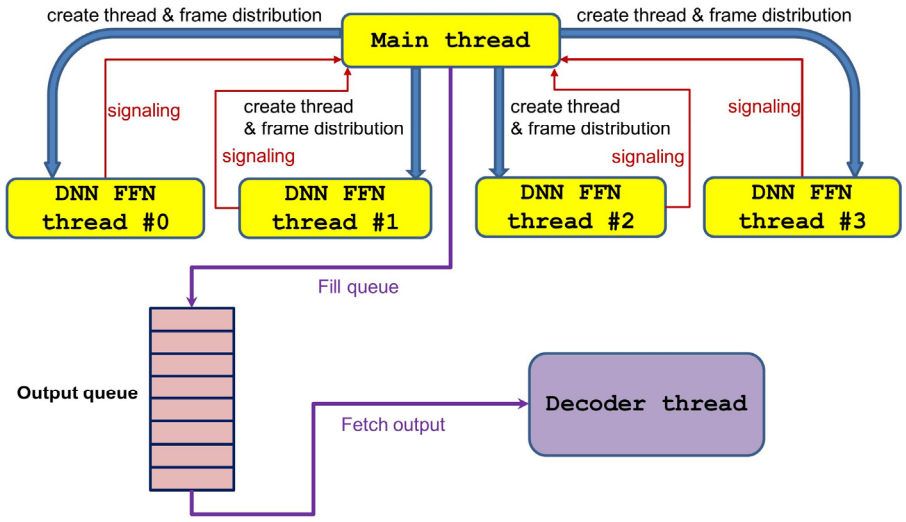

Fig. 4. Parallelization of feedforward network using multi-core.

한 프레임의 피드포워드 네트워크 연산 자체가 실시간 처리가 가능하다고 하더라도 피드포워드 네트워크 연산 이후의 디코더 단의 연산이 필요하고 다른 응용프로그램이 동시에 동작하고 있는 상황을 고려한다면, 피드포워드 네트워크의 연산 자체의 실시간 처리와 무관하게 다중 코어를 이용하도록 병렬화하는 것이 전체 인식 시스템의 전반적인 속도 향상과 안정성을 높일 수 있다. 본 논문에서 제안하는 피드포워드 네트워크의 다중 코어를 이용한 병렬화는 Fig. 4에 나와 있다.

안드로이드 운영 체제의 하부는 리눅스 OS를 기반으로 하기 때문에 쓰레드 기반의 병렬화는 가장 먼저 고려해 볼 수 있는 방법이다. 다중 코어의 경우에는 생성된 쓰레드를 OS가 가장 효율적인 방법으로 코어들에게 분배함으로써 손쉽게 여러 코어를 활용할 수 있게 해준다. 따라서 쓰레드를 다중화하는 것만으로도 다중 코어를 이용한 병렬화를 수행할 수 있다. 쓰레드를 이용하는 방법이 상대적으로 용이한 병렬화 이기는 하지만, 쓰레드 간의 통신과 테스크의 분배는 기본적으로 수행되어야 하기 때문에 기존의 프로그램을 병렬화하는 것은 생각만큼 간단하지 않다. Fig. 4는 1개의 메인 쓰레드와 심층 신경망 피드포워드 네트워크 연산을 수행하는 4개의 연산 쓰레드를 보여준다. 메인 쓰레드는 버퍼에 저장되어 있는 입력 프레임들을 읽어서 4개의 연산 쓰레드에 분배하고 연산이 종료된 쓰레드의 결과를 출력 큐에 저장하는 역할을 담당한다. 각각의 피드포워드 네트워크 연산 쓰레드가 연산을 모두 마쳤는지에 대한 시그널링은 뮤텍스를 통해 수행되며 특정 피드포워드 네트워크 연산 쓰레드가 피드포워드 네트워크 연산을 마치면 메인 쓰레드는 그 결과를 출력 큐에 저장하고 다음 피드포워드 네트워크 연산을 위해 해당 쓰레드에 새로운 입력 프레임을 할당한다. 이렇게 출력 큐에 저장된 결과는 이후 단에서 활용된다. 음성인식의 경우에는 디코더 쓰레드가 출력 큐의 결과를 꺼내서 디코딩을 수행하게 된다. 출력 큐의 크기는 디코더 단의 처리율을 고려하여 적절하게 설정된다.

다중 코어 구현은 단일 코어에서 구현한 +prefetch using PRFM을 다중 코어로 병렬화하였고 그 결과가 Tables 7과 8에 나와 있다. 4개의 피드포워드 네트워크 연산 쓰레드를 사용한 경우 단일 코어에 비하여 DNN_0의 경우는 약 2.2배, DNN_1의 경우는 약 2.8배 정도 속도가 향상됨을 볼 수 있다. 4개의 연산 쓰레스를 사용한 이유는 사용한 응용 프로세서가 4개의 big 코어를 가지고 있기 때문이다. 4개의 little 코어가 있으나 메인 쓰레드 외에 리눅스 상위에 있는 안드로이드 앱에서 추가 쓰레드를 생성하였고 그 외의 OS 상의 여러 서비스등으로 고려할 때, 연산 쓰레드의 수를 big 코어 수와 동일하게 설정하는 것이 합리적이다. 연산 쓰레드를 6개, 8개 등으로 변경하여 추가 실험을

해보았으나 성능은 거의 개선되지 않고 오히려 성능 저하가 발행한다. 이를 통해 연산 쓰레드가 big 코어 위주로 할당 됨을 알 수 있다. 다만, 어떤 코어에 할당 되는가는 전적으로 OS에 의해 결정 되어진다.

V. 결 론

본 논문에서는 NEON SIMD 병렬 기술 및 다중 코어를 이용하여 심층 신경망 피드포워드 네트워크 연산을 최적화하는 방안을 제안하고 구현한 결과의 성능 향상을 제시 하였다. 단일 코어 상에서 NEON SIMD 명령어를 이용하여 최적화 한 결과 약 2.6배의 속도 향상을 얻었다. 또한 메모리 접근을 최소화 하기 위하여 웨이트를 반정도 부동소수점 방식으로 저장했을 경우 4배 이상의 속도 향상을 얻었다. 또한 단일 코어 상에서 개발된 최적화 코드를 다중 코어를 이용하여 병렬화함으로서 심층 신경망 규모에 따라 전체적으로 약 5.7배~7.7배의 속도 향상을 얻을 수 있었다. 본 연구를 통해 이동형 단말기의 자원을 최대한 활용하여 연산 속도가 상당히 개선될 수 있음을 보였고, 따라서 많은 연산량으로 인하여 모바일 환경에서 활용이 어려웠던 딥러닝 응용이 이동형 단말기에서도 활용이 가능하리라 예상된다.

감사의 글

본 연구는 2015년도 강원대학교 대학회계 학술연구조성비(관리번호-520150458) 및 국가과학기술연구회 지원으로 ETRI에서 수행한 '언어장벽 없는 국가구현을 위한 자동통번역 산업경쟁력 강화 사업(17ZS1200)'을 통해 수행되었습니다.

References

1. Y. Bengio, "Learning deep architectures for AI," Foundations and Trends in Machine Learning **2**, 1-127 (2009).
2. Y. LeCunm, Y. Bengio, and G. Hinton, "Deep learning," Nature **521**, 436-444 (2015).
3. G. E. Hinton, S. Osindero, and Y. W. Teh, "A fast learning algorithm for deep belief nets," Neural Computation **18**, 1527-1554 (2006).
4. A. Coates, B. Huval, T. Wang, D. Wu, A. Ng, and B. Catanzaro, "Deep learning with COTS HPS systems," J. Mach. Learn. Res. **28**, 1337-1345 (2013).
5. *NVIDA cuDNN – GPU Accelerated Deep Learning*, https://developer.nvidia.com/cudnn, 2017.
6. X. Lei, A. Senior, A. Gruenstein, and J. Sorensen, "Accurate and compact large vocabulary speech recognition on mobile devices," Proc. Interspeech, 662-665 (2013).
7. I. J. Chung and S. H. Kim "Improving the speed of deep neural network using NEON instructions" (in Korean), J. Acoust. Soc. Kr. Suppl.2(s) **34**, 39-44 (2015).
8. *ARM, ARMv8 Instruction Set Overview reference manual*, PRD03-GENC-010197 15, 2011.
9. *Google Code Archive*, https://code.google.com/archive/p/math-neon/
10. *Quora*, https://www.quora.com/Why-is-softmax-invariant-to-constant-offsets-to-the-input
11. S. Anwar, K. Hwang, and W. Sung, "Fixed point optimization of deep convolutional neural networks for object recognition," ICASSP, IEEE, 1131-1134 (2015).
12. *Mixed-Precision Programming with CUDA 8*, https://devblogs.nvidia.com/parallelforall/mixed-precision-programming-cuda-8, 2016.
13. M. Courbariaux, J. P. David, and Y. Bengio, "Training deep neural networks with low precision multiplications," arXiv:1412.7024v5 (2015).

저자 약력

▸ 정 익 주 (Ik Joo Chung)

1986년 2월: 서울대학교 전자공학과 공학사
1988년 2월: 서울대학교 전자공학과 공학석사
1992년 2월: 서울대학교 전자공학과 공학박사
1992년 8월 ~ 현재 : 강원대학교 전기전자공학부 교수
<주관심분야> 음성신호처리, 적응신호처리

▸ 김 승 희 (Seung Hi Kim)

1997년 2월: 부산대학교 전자공학과 공학사
1999년 2월: 부산대학교 전자공학과 공학석사
1999년 3월 ~ 현재: 한국전자통신연구원 음성지능연구그룹 책임연구원

<주관심분야> 음성인식, 자동통역, 딥러닝