신경회로망 시뮬레이터
(Neural Network Simulator)

梁在宇 · 金斗賢
(한국전자통신연구소)

■ 차 례 ■

[1] 새로운 계산 모델 - 신경회로망

최근에 우리나라를 방문한 AT & T Bell 연구소의 Ross 소장은 90년대에 정보산업 분야의 중요한 연구개발 대상으로 반도체, 광장치, 소프트웨어, 시스템 공학 및 품질공학을 꼽았고 중요한 새로운 기술로는 음성 인식과 신경회로망을 들었다. 병렬분산처리 모델인 신경회로망은 최근에 괄목할 만한 발전을 이루었고 90년대에 가장 각광받는 신기술 중 하나로 손꼽히기에 이르렀다.

현재 컴퓨터가 지니는 한계를 극복하는데 검토해야 할 문제 중의 하나는 계산 모델이 적절한지를 알아보는 것이다. 만약 모델에 근원적인 잘못이 있으면 이를 재료 또는 소자를 개선시켜 극복하는 것은 불가능할 수 있기 때문이다. 현재의 디지털 컴퓨터로 수십~수백 MHz에서 동작하고, 부동소숫점 연산을 초당 10^9개 이상을 수행할 수 있다. 이는 인간의 뇌의 신경세포가 빨라야 수백 Hz로 동작하고 부동소숫점 연산은 일반인의 경우 초당 1개도 할 수 없는데 비해 놀라운 성능이다. 그럼에도 불구하고 컴퓨터는 아주 간단한 일, 예를 들면 사람을 알아본다던가 남의 이야기를 듣고 이해하는 일을 할 수 없다.

일반 컴퓨터가 10개 이내의 프로세서를 사용하여 계산하는 반면 인간의 두뇌는 10^{11}개의 신경세포가 10^{14}~10^{15}개의 선로를 통해 연결되어 있는 방대한 계산 망을 구성하고 있다. 이러한 모델의 차이가 현재 컴퓨터 성능의 한계를 유발시킨 것으로 가정하고, 사람의 이러한 신경계 계산 방식과 유사한 방법으로 문제를 해결하려는 것이 신경회로망이다. 즉, 신경회로망은 수백개에서 수백만개 이상의 간단한 프로세서를 상호연결하여 계산하는 것이다.

병렬분산처리 모델은 새로운 계산 방식으로 광범위한 분야에 적용될 수 있다. 즉, 패턴인식,

연상기억장치, 로보트제어, 지식처리 시스템 등에 이용될 수 있다. 그러나 아직 실용화 단계에 이르지 못했고 많은 연구를 필요로 한다. 그 중 제일 중요한 것이 모델 연구이다. 어떤 문제에 어떤 신경회로망 아키텍쳐가 적합한지를 알아내야 하는 것이다.

신경회로망을 설계하는 방법은 크게 두 가지로 나눌 수 있다. 첫째로는 분석적 방법(Analytic Method)이고 둘째로는 시뮬레이션에 의한 실험적 방법이다. 분석적 방법은 수학적인 증명에 기초하므로 그 결과가 분명하나, 현재까지는 신경회로망 학습 알고리즘의 수렴 여부, 복잡도의 분석 등 극히 일부에만 이용되고 있다. 따라서 실험적인 방법이 널리 쓰이는데, 신경회로망 실험은 시간이 많이 걸리는 것이 문제점이다. 그 이유는 시뮬레이션이 대용량 병렬 컴퓨터가 아닌 보통 일반 컴퓨터 상에서 수행되므로 계산 시간이 많이 소요되고, 그 구조 및 각종 파라미터에 이론적 제한이 별로 없으므로 경험적으로 이를 구해야 한다. 그러므로 신경회로망 설계에는 효율적인 시뮬레이션이 필수적이며 적절한 시뮬레이터의 사용이 중요하다.

[2] 기존 신경회로망 시뮬레이터

시뮬레이터의 기능은 크게 두가지로 나누어 생각할 수 있다. 즉, 비교적 많이 사용되는 기존의 신경회로망의 모델을 구현하여 라이브러리를 구성, 사용자로 하여금 선택사용이 가능하도록 하는 것과, 다른 한가지는 여러가지 신경회로망을 설계할 수 있는 프리미티브(primitive)를 제공하는 것 등이다. 대부분의 상업제품들은 사용자가 신경회로망에 관한 전문지식 없이 즉시 이용할 수 없도록 여러가지 신경회로망 라이브러리를 제공하는 것으로써 교육용으로는 적합하나 모델의 개선이 힘들고 본격적인 연구개발에 사용할 수 없다.

현재 많이 사용되는 대표적인 시뮬레이터의 예는 다음과 같다.

• P3 :

UCSD의 Zipser를 중심으로 개발된 시뮬레이터로서 Symbolics 환경을 이용하고 있다. P3는 크게, plan language, method language, constructor, simulation environment 요소로 구성되어 있다. plan language는 유니트와 연결 구조를 정의하는 언어이며, method language는 Lisp이 확장된 언어로 신경회로망의 동작을 프로그램하는 데에 사용된다. constructor는 plan과 method를 symbolics가 수행할 수 있도록 내부 코드로 변환하여 주는 요소이고, simulation environment는 시뮬레이션을 위한 대화형 인터페이스를 제공한다. P3의 특징은 시각화된 사용자 인터페이스를 제공하는 데에 있으며, 시뮬레이션 속도가 빠르지 않고, plan language와method language가 간결치 못한 단점이 있다.

• ANZA 시스템 :

IBM PC / AT와 386 PC에 장착할 수 있는 Neural Network Coprocessor Board인 ANZA에 사용되는 시뮬레이터이다. C언어와 UISL (User Interface Subroutine Library)를 이용하여 신경회로망을 프로그램할 수 있다. 이미 존재하는 몇가지 신경회로망 모델에 대하여 C 라이브러리로 제공되어 몇가지 파라미터 만을 조절함으로서 프로그램할 수 있다. 프로그래머가 독자적으로 고유의 신경회로망 모델을 개발하기 위해서는 먼저 라이브러리를 만들어야 하며 이를 위하여 AXON이란 모델 정의 언어가 제공되고 있다. Connell Scientific Graphics의 그래픽 패키지를 함께 사용함으로서 그래픽 인터페이스를 가능하게 하고 있다.

• Neural Works :

Neural Ware Inc.에서 개발한 신경회로망 시뮬레이터로서 IBM PC, Macintosh, Sun Workstation 등에서 수행 가능하다. 다양한 모델을 구비하여 이를 메뉴로 제공하여 프로그래머가 이들 중에 적절한 것을 선택함으로서 프로그램을 개발할 수 있도록 되어있다. 그러나 신경회로망 모델을 프로그램할 수 있는 언어를 제공하지 않으므로 새로운 모델 설계에는 이용할 수 없

표 1　신경회로망 하드웨어 비교.

구 분	하드웨어이름	storage (K inter-cnts)	speed (K int/sec)	cost ($)
WORKSTATION				
Micro/Mini computer	PC/AT	100	25	5K
	SUN3	250	250	20K
	VAX	100	100	300K
Attached Processors	ANZA Plus	2500	6000	15K
	Sigma-1	3100	11000	25K
	Transputer	2000	3000	4K
Bus-Oriented Machine	Mark III	12000	500	80K
	Mark V	-	16000	-
	ODYSSEY	256	20000	15K
MASSIVELY PARALLEL	CM-2(64K)	64000	13000	2000K
	WARP(10)	320	10000	300K
	BUTTERFLY(64)	60000	8000	500K
SUPER-COMPUTER	CRAY XMP(1)	2000	50000	4000K

다. 대규모의 복잡한 신경회로망을 시뮬레이션 하기에는 속도가 늦은 단점이 있다.

• RCS :

RCS[1]는 Rochester 대학에서 개발된 신경회로망 시뮬레이터로서 Sun Workstation 환경 하에서 수행된다. C 라이브러리 함수로 신경회로망을 프로그램할 수 있게 되어 있고 Sun Workstation의 Sun Tools를 이용하여 그래픽 인터페이스를 제공한다. 각각의 뉴런이 다른 기능을 갖는 모델을 설계할 수 있고 사용자가 토플러지를 임의로 설정할 수 있다. 따라서 설계의 자유도는 높으나 속도는 높다.

시뮬레이션의 속도는 하드웨어에 의해 크게 좌우되는데 현재 많이 사용되는 하드웨어별 속도는 다음 표 1과 같다. 표에서 storage는 저장가능한 뉴런 및 시냅스의 합을 나타내고 속도는 1초에 몇개의 시냅스 값을 계산할 수 있는가를 나타내는 수치이다.[2]

3. 복합신경회로망 설계용 시뮬레이터 : nnSTUDIO

한국전자통신 연구소에서는 여러가지 신경회로망을 복합하여 새로운 신경회로망을 설계하기 쉽도록 복합신경회로망 설계용 시뮬레이터를 개발하였다.[3] 현재 신경회로망이 갖는 문제점 중의 하나는 한가지의 간단한 문제 해결 방법으로 실세계의 복잡한 문제를 해결하려는데 있다. 인공지능 분야에서는 과거 20년간의 경험을 통해 단일 문제 해결 방식으로 여러가지 복잡한 문제를 해결할 수 없다는 것이 정설로 되고 있다.

복합 신경회로망 모델을 실험하기 위해서는 기존의 시뮬레이터에 없는 다음과 같은 기능이 필요하다.

• 확장성

현재 널리 쓰이고 있는 신경회로망 모델을 미리 라이브러리로 내장하여 프로그래밍의 편이성과 질을 높일 수 있어야 한다. 그러나 응용 분야에 따라서 망의 구조, 파라미터, 학습 규칙 등을 변경하여야 하는 경우가 많다. 이를 위해 라이브러리에 있는 모델을 조금 변경하거나 확장하여 각기 응용 분야에 맞는 모델로 재구성하도록 도와 주는 기능이 필요하다. 따라서 다양한 신경회로망 모델에 대하여 양질의 프로그램을 라이브러리로 내장하고 있을 뿐만 아니라 라이브러리를 변경하여 자신의 문제에 맞는 프로그램을

개발할 수 있도록 하며, 새로운 모델을 설계하여 라이브러리에 손쉽게 추가할 수 있어야 한다.

• Enhanced Connection Mechanism :

신경회로망을 응용하여 시스템을 구성해야 할 경우 한가지 신경회로망 모델로만 전체 시스템을 구성하는 경우도 있지만 복잡한 문제 해결을 위해서는 몇가지의 신경회로망이 계층식이나 그래프식으로 복잡하게 연결된 복합신경 회로망을 필요로 한다. 이러한 신경회로망을 쉽게 프로그램하려면 프로그램 언어가 이종(heterogeneous) 의 신경회로망을 다양한 모양으로 접속시킬 수 있는 가능을 제공해야 한다.

• Reusability

프로그램하려는 복합신경회로망이 라이브러리나 과거에 작성해 놓은 프로그램하고 다르더라도 이를 구성하는 단위 신경회로망의 일부는 라이브러리나 기존 프로그램을 그대로 사용하거나 약간 수정보완하면 되는 경우가 있다. 이러한 경우 객체지향형 프로그램 방식과 같이 원하는 신경회로망을 상속받고 상속받지 않을 부분만을 프로그램하도록 하여 프로그램 시간을 줄이고 프로그램의 간결성과 재사용도를 높이도록 하여야 한다.

이러한 요구사항을 만족시키기 위하여 복합신경회로망 프로그램인 nnSTUDIO를 개발하였다. nnSTUDIO의 인터프리터는 Common Lisp[4]으로 구현되었는데 Common Lisp이 제공하는 기존 함수들을 모두 사용할 수 있고 Common Lisp 응용 프로그램에서 신경회로망 프로그램을 호출할 수 있다. 따라서 nnSTUDIO는 Common Lisp을 신경회로망용으로 확장하기 위한 일종의 Common Lisp 패키지로도 볼 수 있다. 이 nnSTUDIO는 Common Lisp이 탑재된 어떤 컴퓨터에서도 인터프리터가 수행되므로 컴퓨터의 기종이 달라도 호환성을 가지며, 현재 Sun, Macintish, Pyramid 등에 탑재되어 이용되고 있다.

nnSTUDIO는 객체지향형 프로그램 개념을 따라 뉴런과 층(layer)의 유형을 단계적으로 정의하고 다시 여러 유형의 층을 결합하여 복합신경회로망 모델을 프로그램하도록 되어 있다. 본 장에서는 nnSTUDIO의 신택스와 사용 방법에 대하여 뉴런, 층, 네트워크의 유형을 정의해 나가는 순으로 설명한다.

1. 뉴런의 유형정의

DefNeuron과 **DefNeuron Method**를 사용하여 뉴런의 유형을 정의한다. 신택스는 다음과 같다.

• *neuron-slot- specifier*는 상용슬롯 (*static slot*) 고 선택슬롯 (*optional slot*)으로 나뉘어 진다. 상용슬롯에는 **Connection, Input, Output, SynapseIn, Weight**이 있고, 선택슬롯에는 **Bias, Difference, SynapseBack, Weight, Incidence** 슬롯이 있다. 상용슬롯은 프로그래머가 별다른 언급이 없어도 뉴런이 항상 갖고 있는 것으로서 단지 슬롯값의 타잎을 바꾸어 사용하기를 원할 때에 한하여 *value-type*의 형식에 따라 원하는 타잎으로 바꾸어 주면 된다. *static-slot*의 **connection**은 뉴런이 임의 층으로부터 입력을 받기 위하여 시냅스를 연결하여야 할 경우 그의 연결 양식을 지정하는 선택슬롯이다. 묵시값은 **fully**이며 표현의 완전성을 위해 **fully**라고 지정할 수도 있다. **random**이라 지정된 경우는 다음에 지정된 퍼센트만큼 무작위로 연결해준다.

• 선택슬롯은 프로그래머가 지정해야만 뉴런의 슬롯으로 할당되는 것으로 다양한 뉴런의 유형을 제공할 수 있도록 한다. **Teach** 슬롯은 supervised learning에서 입력에 대응하는 기대값을 저장하는데 사용하는 슬롯이다. **SynapseBack, Bias, Difference** 슬롯 등은 Multilayer Perceptron의 back- propagation 형 학습을 쉽게 프로그램 할 수 있도록 배려한 것으로 **SynapseBack**은 하위층의 **Difference** 슬롯에 입력될 값을 저장해 두기 위한 슬롯이고, 반면 **Difference** 슬롯 자체는 상위층의 오차를 전달 받는 데에 사용하는 슬롯이다. **InterConnection** 슬롯은 같은 층의 뉴런

```
(DefNeuron neuron_class_name ([neuron_superclass_name]*)
{(neuron_slot_specifier)}* )

(DefNeuronMethod (neuron_method_name neuron_class_name) (parameters)
method_body)

neuron_class_name ::= symbol
neuron_superclass_name ::= symbol

neuron_slot_specifier ::= static_slot | optional_slot

static_slot ::=     {Connection connection_type} |
                    {Input value_type} |
                    {Output value_type} |
                    {SynapseIn value_type} |
                    {Weight value_type} |

optional_slot ::=   {Teach value_type} |
                    {Bias value_type} |
                    {Difference value_type} |
                    {SynapseBack value_type} |
                    {Incidence value_type}
                    {InterConnection interconnection_type} |
                    {InterWeight value_type} |
interconnection_type ::= Fully | Neighbour integer
connection_type ::= Fully | Random integer %
value_type ::= { range } | Integer { range } | Binary | Bipolar
range ::= ({integer | real} {integer | real})
neuron_method_name ::= InFun | TransFun | OutFun
```

간의 연결 여부와 연결 형태를 지정한다. 이와 상응하여 **InterWeight**는 **InterConnection**이 지정되었을 때 연결 강도를 담는 변수이다. 이 두 가지 슬롯은 lateral inhibition이 일어나는 ART 등의 competitive network에 사용될 수 있다. **Incidence** 슬롯은 시냅스와 뉴런을 연결하는 **Weight**와 같은 크기의 벡터로서 **Connection**으로는 시냅스와 뉴런의 연결을 임의로 프로그램하기가 부족할 경우 연결 하나 하나에 대하여 그 연결여부나 가중치등을 임의로 지정하는데 사용할 수 있다.

- *neuron-method- name*의 **InFun**은 뉴런의 입력을 받아들이는 입력함수이고, **TransFun**은 뉴런의 입력을 전달하는 전달함수이며, **OutFun** 은 **TransFun**의 출력을 재차 변형하여 출력시키는 출력함수이다. 이들은 뉴런 유형이 갖고 있는 종속함수(method)로서 프로그래머가 원하는 대로 작성할 수 있다. 만약 종속함수가 프로그램 되어 있지 않다면 상위유형(superclass)에서 상속 받은 것을 사용하고 아무 곳에서도 상속되지 않으면 인터프리터에 내장되어 있는 묵시적인 종속함수를 사용한다. 종속함수를 프로그램할 때는 인터프리터가 제공하는 벡터처리함수를 이용하거나 Common Lisp 함수를 사용할 수 있다.

2. 층의 유형정의와 생성

DefLayer, DefLayer Method를 이용하여 같은 유형의 뉴런들로 구성된 층의 유형을 정의하고 **BuildLayer**로 객체를 생성한다. 신택스는 다음과 같다.

- *layer-slot- specifier*의 **Neuron** 슬롯은 현재 정의하고 있는 유형의 층이 사용할 뉴런의 유형을 지정한다. 프로그래머는 이 순간부터

(DefLayer *layer_class_name* ({*layer_superclass_name*}*) {(*layer_slot_specifier*)}*)

(DefLayerMethod *(layer_method_name layer_class_name) (parameters) method_body*)

(BuildLayer *layer_class_name* {**:number-of-neuron** *max_neuron*} {**:number-of-synapse** *max_synapse*})

layer_class_name ::= symbol
layer_superclass_name ::= symbol
layer_object_name ::= symbol
max_neuron ::= integer | integer_list
max_synapse ::= integer | integer_list

layer_slot_specifier ::= {**Neuron** *neuron_class_name*} |
{**Dimension** *integer*} |
{**Parameter** {(*parameter_name initial_value*)+}}

layer_method_name ::= **Run | Learn | Teach | Terminate | Initialize | Show | FeedIn**

를 지정하여 다른 층과 맞붙일 영역을 설정하고 이 영역에 맞붙일 B1의 **Output** 벡터의 *start-index*와 *end-index*를 지정한다. 만약 인덱스가 지정되지 않았으면 벡터 전체를 연결하는 것으로 간주된다. 같은 형식으로 B2, B3,… B_n을 연결할 수 있다. 단, A의 동일한 시냅스와 여러 층의 뉴런 출력과의 연결은 신경회로망의 원리에 어긋나므로 이를 허용치 않는다.

- 임의 층이 다른 층과 연결하는 방법은 위에서 설명한 바와 같다. 그러나 층의 입출력이 항상 다른 층과만 일어나지 않고 외부로부터 주어지는 수도 있다. 이런 경우 데이터는 화일로 뉴런에 있는 변수들이 벡터로 확장된 것으로 생각하고 종속함수를 프로그램하는 데에 인터프리터가 제공하는 벡터처리함수를 이용할 수 있다. 이 슬롯이 지정되지 않은 경우에는 상속받은 층유형의 뉴런 유형을 사용한다.
- *layer-slot- specifier*의 **Dimension** 슬롯에는 뉴런의 차원을 정수로 지정한다.
- **Parameter** 슬롯에는 층을 학습시키거나 수행할 때 필요한 조절용 파라미터를 정의한다. 정의 시에 층의 객체가 생성될 때의 파라미터 초기값을 지정할 수도 있다. 만약 지정하지 않은 경우에는 **BuildLayer**나 **BuildNetwork**을 이용하여 층의 객체를 생성할 때 사용자에게 그 초기값을 묻도록 되어 있다.
- *layer-method- name*의 **Run, Learn, Teach, Terminate, Initialize**는 현재 정의되는 유형의 층이 가질 수 있는 종속함수이다. 프로그램되어 있지 않은 경우는 상속 받은 것을 이용하고 상속되는 것도 없을 경우엔 인터프리터가 내장하고 있는 가장 일반적인 종속함수를 적용한다.
- **Run**에는 층에 입력이 들어와서 출력될 때까지의 과정을 프로그램한다.
- **Learn**에는 층의 시냅스 강도(synaptic weight)를 조절하여 학습하는 과정과 규칙을 프로그램한다.
- **Teach**에는 층의 외부 입력에 대한 출력과 올바른 출력과의 오차를 계산하여 뉴런에 가르쳐주는 방법을 프로그램한다. 올바른 출력값은 **Teach** 슬롯에 저장하며 오차 값은 뉴런의 **Difference**에 저장하는 것을 원칙으로 한다.
- **Terminate**에는 학습을 종료시키는 조건을 프로그래밍한다.
- **Initialize**에는 **BuildLayer**가 수행되는 초기

에 프로그래머가 특별한 조작을 가하기 원할 때 그 내용을 프로그램한다.

- **Show**에는 층의 여러 상태를 그래픽으로 보여주기 위한 프로그램을 작성할 수 있다. 이 부분은 그래픽 도구에 따라 간단할 수도 있고 복잡할 수도 있으며 컴퓨터 기종에 따라 변경을 가해야 하는 부분이다.
- **FeedIn**은 내장되어 있는 종속함수이다. 이는 **DefNetwork**의 **Connect** 메카니즘에 의하여 신경회로망 프로그램 언어의 인터프리터가 자동으로 생성시켜준 종속함수로서 **FeedIn**을 호출하면 연결되어 있는 층의 출력들을 동시에 읽어 들여 **Connect**에서 지정된대로 **SynapseIn**에 입력시켜 준다.

3. 네트워크의 유형정의와 생성

DefNetwork, DefNetwork Method을 이용하여 동종 또는 이종의 층이 연결된 네트워트의 유형을 정의하고, **BuildNetwork**을 이용하여 네트워크를 생성한다. 신택스는 다음과 같다.

- **DefNetowrk**을 이용하면 다양한 유형의 층으로 구성된 네트워크를 디자인 할 수 있다. 먼저 네트워크를 구성할 층의 이름과 유형을 열거한다. 이 때 층의 뉴런 갯수(:number-of-neuron)나 시냅스의 갯수(:number-of- synapse)를 지정할 수도 있다 만약 지정하지 않았을 경우에는 **BuildNetwork** 함수가 수행될 때 뉴런용 메모리를 할당 받기 위하여 프로그래머에게 물어보도록 되어 있다.
- 여러개의 층으로 이루어진 일반적인 네트워크를 디자인할 수 있는 이유는 **Connect** 기능이 있기 때문이다. **Connect**에는 네트워크에 들어갈 각 층에 대하여 *location*을 이용하여 서로의 연결정보를 프로그램한다. 임의 *layer* A와 다른 **layer** B1과의 연결을 위하여 먼저 A의 **Synapsein** 벡터 중 *start-index*와 *end-index*

```
(DefNetwork net_class_name ({net_superclass_name}*) net_slot_specifier)

(DefNetwokMethod (net_method_name net_class_name) (parameters) method_body)

(BuildNetwork net_class_name)

net_class_name ::= symbol
net_superclass_name ::= symbol

net_slot_specifier ::= {(layer layer_class
                         {:number-of-neuron max-neuron}
                         {:number-of-synapse max-synapse})}+ |
                       Connect {(connect_specifier)}
connect_specifier ::= (location location)
location ::= (loc_slot layer {start_index} {end_index})

layer ::= symbol | LearnFile | RunFile | File
layer_class_name ::= symbol
max_neuron ::= integer_list | integer
max_synapse ::= integer_list | integer
start_index ::= integer_list | integer
end_index ::= integer_list | integer

loc_slot ::= SynapseIn | Output | Teach

net_method_name ::=Run | Learn | Initialize | Terminate | Show
```

지정되는데 이러한 입출력과의 연결도 **Connect** 로 가능하다. 우선 데이터 화일도 층의 일종으로 확장된 개념으로 생각하고 **LearnFile** 이나 **RunFile**에 대한 *location*을 이용할 수 있다. 단, *loc-slot*으로 입력 데이터에 해당하는 **SynapseIn**과 출력치의 기대값인 **Teach** 만을 사용할 수 있다.

- **LearnFile, RunFile, File** 등은 일종의 화일 변수로서 이를 이용하여 실제 화일 이름을 지정하지 않고도 화일 입출력에 관한 종속함수를 작성할 수 있다.
- **Run**에는 각 층에 존재하는 **Run** 등의 종속함수를 이용하여 네트워크 전체의 동작을 프로그램한다.
- **Learn**에는 각 층에 존재하는 **Learn, Run, Teach** 등을 이용하여 네트워크 전체의 학습과정을 프로그램한다.
- **Initialize**에는 **BuildNetwork**이 수행되는 초기에 프로그래머가 특별한 조작을 가하기 원할 때 그 내용을 지정한다.

4. 벡터처리함수

뉴런이 소유하고 있는 슬롯들은 여러 뉴런이 층을 형성하게 되면 자연히 벡터를 형성하게 된다. 따라서 층전체의 행위를 쉽고 개념적으로 프로그램할 수 있도록 하기 위한 벡터처리함수들을 제공한다.

```
(function_name {vector}+ {result_vector})
(function_name* {vector}+)
```

벡터처리함수의 일반적인 형식은 위와 같다. *result-vector*가 주어져 있으면 함수 수행의 결과를 그곳에 저장한다. 만약 주어져 있지 않을 경우 묵시적으로 *vector* 순의 맨 마지막을 destruct하여 그 결과를 저장한다. *function-name**을 사용한 경우는 *result-vector*가 주어져 있지 않더라도 맨 마지막 *vector*를 destruct하지 않고 함수의 호출값으로만 출력하여 준다. 이 경우 함수 수행의 결과를 사용키 위해서는 함수의 결과를 임의의 변수에 항상 저장하여야 한다. *functiion-name*에는 Add, Sub, Prod, OutProd, DotProd, EuclidDist, Div, Sigmoid, Threshold, Exp 등이 있다.

5. 데이터 입출력

화일 형식:

데이터 화일은 리스트 형식으로 제공되어야 한다. unsupervised learning의 경우처럼 입력만 담고있는 화일은 한번의 입력이 리스트로 표현되어 이를 단위로 화일에 연속적으로 기록되어 있어야 하고, 즉 {({*input-value* }+)}*, 입출력 쌍이 기록되는 화일은 입력 리스트와 출력 리스트를 한개의 리스트로 표현하여 이를 단위로 화일에 연속적으로 기록되어 있어야 한다. 즉 {(({*input-value*} +) ({*output-value* }+))}*.

FeedIn :

FeedIn은 **DefNetwork**의 **Connect** 정의에 따라 인터프리터가 각 층에 맞도록 자동으로 생성해준 종속함수이다. 임의 종속함수가 **FeedIn** 을 호출하면 연결된 다른 층의 **Output**을 읽어 **Connect**에서 지정한대로 **SynapseIn**에 입력시킨다. **Connect**에 **LearnFile**이나 **RunFile** 과의 관계가 지정되어 있으면 *file-variable*에서 지정한 화일로부터의 입력을 수행한다. 데이터 화일은 신경회로망의 성격에 따라 입력만 열거된 것이 있고 입력에 상응하는 출력값과의

```
(FeedIn layer {:File file_variable})
file_variable ::= {LearnFile | RunFile}
(Setfile layer_or_net {:LearnFile file_name} {:RunFile file_name})
```

쌍이 열거된 경우가 있다. 입력 데이터에 대하여는 지정된 **SynapseIn**의 위치로 입력하고, 만약 후자와 같이 출력이 제공된 경우는 이를 읽어서 **Teach**에 입력해 준다.

Setfile :

Setfile은 층이나 네트워크 객체의 학습이나 수행에 사용될 실제 데이터 화일명을 지정한다.

[4] 시뮬레이션

본 장에서는 제 Ⅲ장에서 설명한 신경회로망 프로그램 언어인 nnSTUDIO를 이용하여 실제로 신경회로망을 프로그램하고 이를 시뮬레이션 한다. 프로그램할 신경회로망은 3개의 서로 다른 유형의 층이 삼각관계로 연결되어 있는 복합 신경회로망(이하 KohPerGrap 네트워크라 칭함)으로 2개의 층은 설명한 Perceptron 유형의 층이고 나머지 하나는 설명한 Kohonen의 self organizing feature map 유형이다. 먼저 Kohonen 유형의 층과 Perceptron 유형의 층을 프로그램한다. 그리고 KohPerGrap를 프로그램하기 앞서 Kohonen 유형과 Perceptron 유형의 두 층이 계층식으로 접속되어 있는 신경회로망을 프로그램하여 음성데이터를 입력하여 실제로 수행된 예를 설명한 후 삼각관계의 그래프형 결합으로 이루어진 KohPerGrap 라는 유형의 네트워크를 프로그램하는 과정으로 설명한다.

시뮬레이션 Macintosh Ⅱ의 Allegro Common Lisp 위에서 수행하였다. 이에 따라 **Show** 함수는 Allegro Common Lisp이 제공하는 그래픽 유틸리티를 이용하여 프로그램하였고, 제시되는 디스플레이의 예시는 Macintosh Ⅱ의 Quick Draw 화면이다.

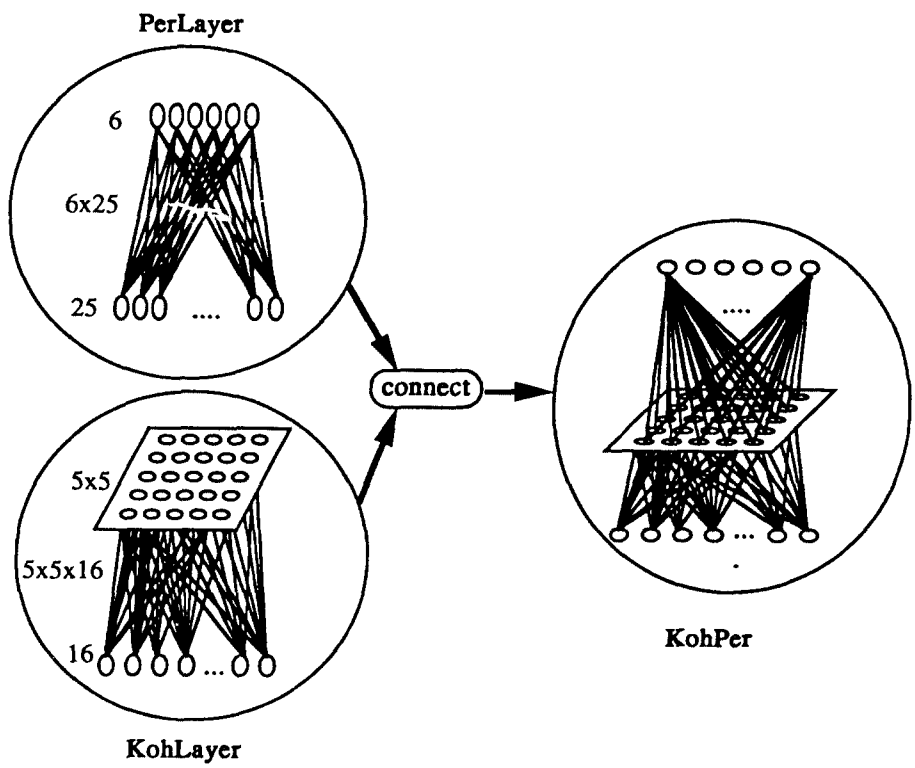

그림 4.1. Kohper 네트워크의 구성도

```
(DefNet KohPer()
  (KohLayer Kohonen :number-of-neuron (5 5) :number-of-synapse 16)
  (PerLayer Perceptron :number-of-neuron 6 :number-of-synapse (5 5))
  (Connect (KohLayer PerLayer)
           ((Teach File) (Teach PerLayer))
           ((SynapseIn File) (SynapseIn KohLayer))))

(DefNetMethod (Run KohPer)()
  (Run KohLayer RunFile)
  (Run PerLayer RunFile)
  (Show Self))

(DefNetMethod (Learn KohPer)(&optional (KohTimes 1) (PerTimes 1))
  (dotimes (i KohTimes)
           (Learn KohLayer))
  (dotimes (i PerTimes)
           (Run KohLayer LearnFile)
           (Run PerLayer LearnFile)
           (Teach PerLayer)
           (Learn PerLayer)))
```

1. 이종층의 단순 결합

Kohonen[5] 층 위에 Perceptron[6] 층이 연결되어 있는 이종 다층 네트워크를 프로그램한다. 네트워크의 유형명은 KohPer이고 Kohonen 유형의 층을 KohLayer라 하고 Perceptron 유형의 층을 PerLayer라 한다. Kohonen 유형과 Perceptron 유형은 nnSTUDIO로 이미 작성이 되어 있고 본 절에서는 이들을 상속하여 네트워크를 구성하는 부분만 예시한다. 이들 두 유형의 nnSTUDIO 프로그램은 [3]에 자세히 설명되어 있다. KohLayer 는 5×5의 뉴런이 이차원 배열을 하고 각 뉴런은 16개의 시냅스를 갖는다. PerLayer는 6개의 뉴런이 일차원 배열을 하고 각 뉴런은 KohLayer의 출력과 접속된 5×5개의 시냅스를 갖는다. KohPer 네트워크의 구성도가 그림 4.1에 나타나 있다.

Connect 슬롯에 KohLayer위에 PerLayer가 연결되어 있음을 지정하고, **File** (**LearnFile**과 **RunFile**을 모두 지칭)도 **Teach** 값과 **SynapseIn** 값을 생성하여 주는 일종의 층으로 보아 **Teach**는 교사학습을 할 PerLayer에 접속하고 SynapseIn은 자율학습을 할 KohLayer에 접속시킨다. 종속함수인 **Learn**과 **Run**은 KohLayer와 PerLayer의 **Learn, Run**을 호출하면서 네트워크 전체의 학습 및 실행을 진행하도록 한다.

KohPer 유형의 네트워크의 프로그램이 끝났으므로 **BuildNet** 함수를 이용하여 이의 객체를 생성하여 akohper 변수에 바인드시킨다. 객체를 생성할 때 층의 초기화 되지 않은 파라미터가 있으면, 예를 들면 PerLayer의 Eta, 그 초기값을 묻는다. 만약 **DefNetwork**에서 **:number-of-neuron**이나 **:number-of- synapse**가 지정되지 않았으면 역시 그 값을 묻는다. akohper를 수행시키기 전에 **SetFile** 함수를 이용하여 **LearnFile** 과 **RunFile**에 실제로 화일명인 "aeiouw-.tch"와 "aeiouw.tst"를 바인드시킨다. 화일의

```
(setq akohper (BuildNet 'KohPer))
(SetFile akohper :learn "aeiouw.tch" :run "aeiouw.tst")
(Learn akohper 5000)
(Run akohper)
```

지정이 끝났으므로 Learn 종속함수를 호출하여 akohper를 학습시킨다. 총 학습 횟수는 5,000번으로 한다. 학습이 끝난 후 **Run**을 호출하여 학습 결과를 시험한다.

실험에 사용된 "aeiou.tch"와 "aeiou.tst" 화일은 한국어 모음 '아', '에', '이', '오', '우', '으'의 실제 음성 입력에 몇단계의 전처리[3]를 가한 데이터 화일이다.

학습이 끝난 후 '아'에 대한 KohPer 네트워크의 화면이 그림 4.2에 나타나 있다. 화면의 우편은 전처리된 음성 입력을 그래프로 나타낸 것으로 모음 입력이므로 low frequency 부분의 값이 큰 것을 볼 수 있다. 좌편 하단은 KohLayer에 5×5로 나열된 EuclidNeuron 뉴런의 Output 상태를 5 grey level로 나타낸 것으로(0에서 1 사이를 0.2씩 균등 분할) 검을수록 뉴런이 활성화되어 있음을 나타낸다. 좌편상단은 PerLayer의 6개 NonLinear 뉴런의 Output 상태를 나타낸 것으로 밝을수록 뉴런이 활성화된 것이다. 그림을 자세히 보면 KohLayer가 self organization의 결과로 '아' 모음에 대하여 우측 상단에 활성화된 뉴런 그룹이 형성되어져 있는 것을 볼 수 있으며 PerLayer는 학습된 KohLayer의 출력 패턴을 6가지 유형으로 분리하여 첫번째 뉴런이 '아'에 대하여 활성화되어 백색인 것을 볼 수 있다.

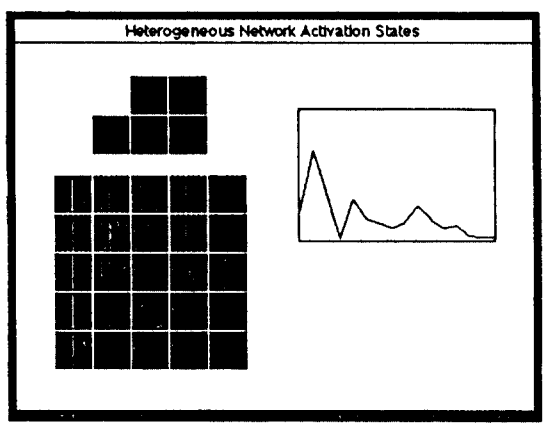

그림 4.2 모음 '아'에 대한 KohPer 네트워크의 **Output** 상태

2. 이종층의 그래프식 결합

Kohonen 유형의 층 KohLayer와 Perceptron 유형의 두 층 CVLayer 및 PerLayer가 그림 4.3과 같이 연결된 KohPerGrap을 프로그램한다. 이 KohPerGrap은 제1절의 KohPer 네트워크가 모음만을 입력받아 인식하던 것을 확장하여 모음과 자음이 혼합되어 있는 실제 연속음성 중에서 모음 부분만 감지하여 인식하는 것을 목표로 하고 있다. KohPerGrap 모델이 모임인식에 실용적인지는 미지수이나 본 절에서는 다만 복합형 신경회로망의 프로그램을 실험하기 위한 것이다.

KohLayer와 PerLayer는 제1절에 설명된 것과 같으나 단지 PerLayer의 시냅스 갯수가 두 개 늘어나 27개인 것은 CVLayer의 두 개 **Output**을 추가로 입력받기 때문이다. CVLayer는 두 개의 뉴런이 KohLayer의 출력을 입력받아 이를 모음과 모음이 아닌 것 두 가지의 유형으로 분류하고 그 결과를 PerLayer에 전달해 주는 역할을 한다. PerLayer는 KohLayer의 출력과 CVLayer의 출력을 입력받아 모음인 경우11개의 단모음으로 분류하여 해당 뉴런이 활성화되고 비모음인 경우는 12번째 뉴런만 활성화된다.

이와 같은 기능을 할 수 있도록 KohPerGrap의 객체를 학습시키기 위해서는 학습데이터가 필요한데 **LearnFile**의 SynapseIn 부문에는 전처리된 16개 채널의 값을 기록하고 **Teach** 부분 중 12개는 PerLayer를 학습시키기 위하여 **SynapseIn**의 값에 상응하는 PerLayer 교사값(teaching value)을 기록하고(11개는 각 모음을 지시하고 12번째는 비모음을 지시함) 나머지 두개는 CVLayer가 모음과 비모음을 분류하도록 학습시키기 위한 CVLayer의 교사값을 기록한다.

KohPerGrap의 학습은 **Learn** 종속함수와 같이 먼저 KohLayer를 학습하고 그 다음 CVLayer 를 학습한 후 마지막으로 PerLayer를 학습한다. 학습후 실행도 **Run**과 같이 학습할 때의 순서대로 한다.

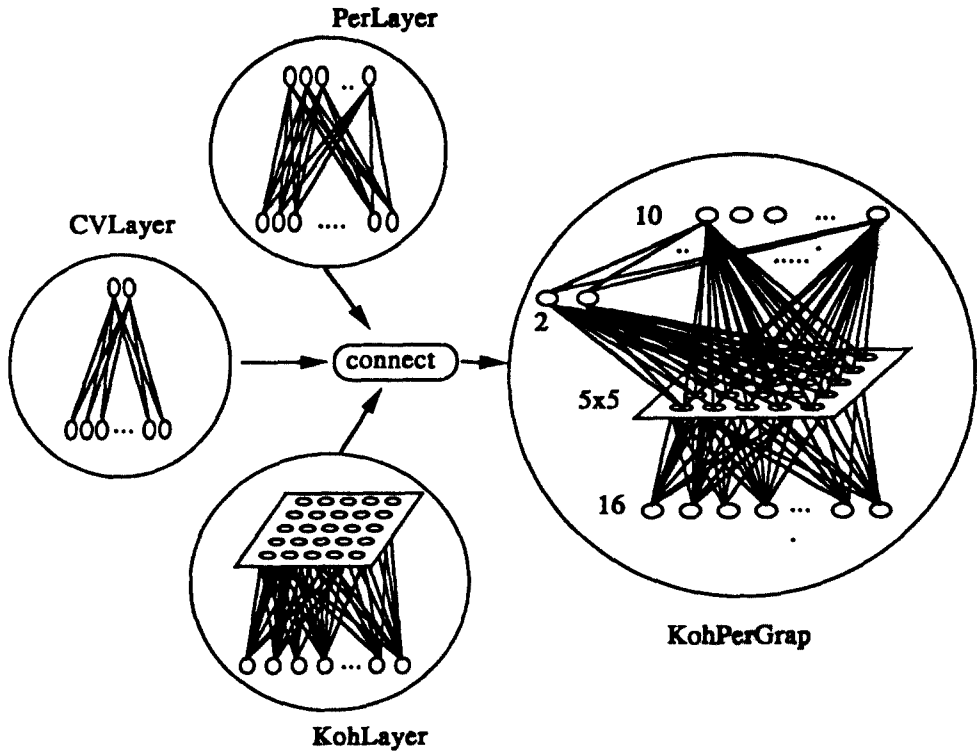

그림 4.3 KohPerGrap 네트워크의 구성도

```
(DefNetwork KohPerGrap()
  (KohLayer Kohonen :number-of-neuron (5 5) :number-of-synapse 16)
  (CVLayer Perceptron :number-of-neuron 2 :number-of-synapse 25)
  (PerLayer Perceptron :number-of-neuron 12 :number-of-synapse 27)
  (Connect ((SynapseIn PerLayer 1 25) (Output KohLayer (1 1) (5 5)))
           ((SynapseIn PerLayer 26 27) (Output CVLayer))
           ((SynapseIn CVLayer) (Output KohLayer))
           ((Teach LearnFile 1 12) (Teach PerLayer))
           ((Teach LearnFile 13 14) (Teach CVLayer))
           ((SynapseIn LearnFile) (SynapseIn KohLayer))
           ((SynapseIn RunFile) (SynapseIn KohLayer))))
```

```
(DefNetworkMethod (Learn KohPerGrap)
                  (&optional (KohTimes 1)(CVTimes 1)(PerTimes 1))
  (dotimes (i KohTimes)
           (Learn KohLayer LearnFile))
  (dotimes (i CVTimes)
           (Run KohLayer LearnFile)
           (Learn CVLayer LearnFile))
  (dotimes (i PerTimes)
           (Run KohLayer LearnFile)
           (Learn PerLayer LearnFile)))

(DefNetworkMethod (Run KohPerGrap)()
  (Run KohLayer RunFile)
  (Run CVLayer)
  (Run PerLayer))
```

3. 유형 상속

nnSTUDIO는 객체지향성 프로그램 언어의 개념을 수용하고 있기 때문에 유형의 상속체계가 제공된다. 이러한 상속체계는 프로그램의 생산성과 간결성을 증대시켜주는 nnSTUDIO의 중요한 특징 중의 하나이다. nnSTUDIO에서의 유형상속의 예를 들어 본다. 아래의 프로그램된 Perceptron-B 유형은 nnSTUDIO에 내장된 Perceptron 유형의 변종으로 Perceptron이 사용하는 NonLinear 뉴런의 종속함수인 InFun이 Sigmoid에서 **Threshold** 함수로 바뀐 예이다. 프로그램에서 볼 수 있듯이 Perceptron을 상속받고 **InFun** 종속함수를 변경하면 모든 프로그램이 끝난다.

```
(DefLayer Perceptron-B (Perceptron))

(DefNeuronMethod (InFun Perceptron-B)()
  (Prod Weight SynapseIn Input)
  (Threshold Input Bias Output))
```

[5] 결론 및 전망

본 논문에서는 복합신경회로망 설계를 위한 프로그램 언어인 nnSTUDIO에 대하여 설명하고 복합신경회로망 시뮬레이션의 실례를 들어 보았다.

nnSTUDIO는 기존의 시뮬레이션 언어가 제공하지 않는 몇가지 특징을 갖고 있다. 첫째, 객체주의 프로그램 방식에 따라 nnSTUDIO에 내장되어 있는 기존 모델을 상속하여 보다 개선된 모델을 설계할 수 있도록 하였고, 이미 개발된 다양한 모델을 임의의 형태로 접합시킬 수 있도록 하여 복합 신경회로망을 구성하기에 용이하도록 하였다. 현재 nnSTUDIO는 Sun, Macintosh, Pyramid 등에 탑재되어 있는데 병렬처리 하드웨어를 이용하지 않았으므로 속도가 느리다. 그러나 내장된 벡터함수를 Array Processor 등의 병렬처리 하드웨어에 구현하면 손쉽게 성능을 향상시킬 수 있을 것이다. nnSTUDIO는 Common Lisp에 패키지로 구현되어 있어서 호환성이 높고 Common Lisp과 상호 호출이 가능하며 인공지능의 심볼릭 프로그램 개념과 신경회로망 프로그램 개념이 동시에 존재함으로 양 개념의 적절한 융합에서 취득되는 잉여 기능을 통하여 보다 지능적인 프로그램을 작성할 수 있을 것이다.

신경회로망의 응용면에 있어서 단일 종류의 모델보다는 여러 종류의 모델을 적절히 연결지어 문제를 해결하는 것이 보다 유용함에도 불구하고, 기존의 시뮬레이터나 신경회로망 프로그램

언어들은 이미 개발된 다양한 종류의 신경회로망 모델을 접합시켜 복합 신경회로망을 구성하는 데에 필요한 기능을 제공하고 있지 않다. 그러나, 추세로 보아 복합신경회로망의 사용도는 계속 높아질 것으로 예상되므로 복합신경회로망 프로그램 기능은 시뮬레이터의 중요 요구사항이 될 것이다. 반면 신경회로망이 복합화됨에 따라 망의 크기가 커지고 계산량도 많아져 속도가 심각한 문제로 대두될 것이다. 이러한 점에 비추어 볼 때 nnSTUDIO와 같이 복합 신경회로망 프로그램 기능을 제공하면서 병렬하드웨어화하기 쉬운 시뮬레이션 언어를 개발 사용하는 것이 바람직할 것이다.

參 考 文 獻

1. J. A. Feldman, et. al., "Computing with Structured Neural Networks", *IEEE Computer,* vol. 21. no. 3, 1988.
2. Defense Advanced Research Project Agency, *DARPA Neural Network Study,* Virginia : AFCEA Onternational Press, 1988.
3. 한국전자통신 연구소, 실시간 패턴인식을 위한 병렬분산처리모델 연구, 최종연구보고소, 9SC3 300286113F, 1990. 5.
4. G. L. Steele Jr., S. E. Fahlman, R. P. Gabriel, D. A. Moon, and D.L. Weinreb, *Common Lisp : The Language,* Digital Press, 1984.
5. T. Kohonen, *Self- Organization and Associative Memory : 2nd Ed.,* Information Sciences, 8, Springer- Verlag, Berlin-Heidelberg- New York--Tokyo, 1988.
6. R. Rosenblatt, "The Perceptron : a probabilistic model for information storage in the brain, " *Psychological Review,* vol. 65, pp. 386~408, 1958.

梁 在 宇

- 1952년 7월 9일생
- 1978년 2월 : 서울공대 전기공학과 졸업
- 1982년 7월 : 서울대학원 제어계측과 졸업(석사)
- 1978년 2월~1979년12월 : 삼성전자(주) 근무
- 1980년 1월~현재 : 한국전자통신연구소 소프트웨어공학 실장

金 斗 賢

- 1962년 7월28일생
- 1985년 2월 : 서울공대 전자계산기공학과 졸업
- 1987년 2월 : 한국과학기술원 전산학과졸업(석사)
- 1987년 2월~현재 : 한국전자통신연구소 인공지능연구실