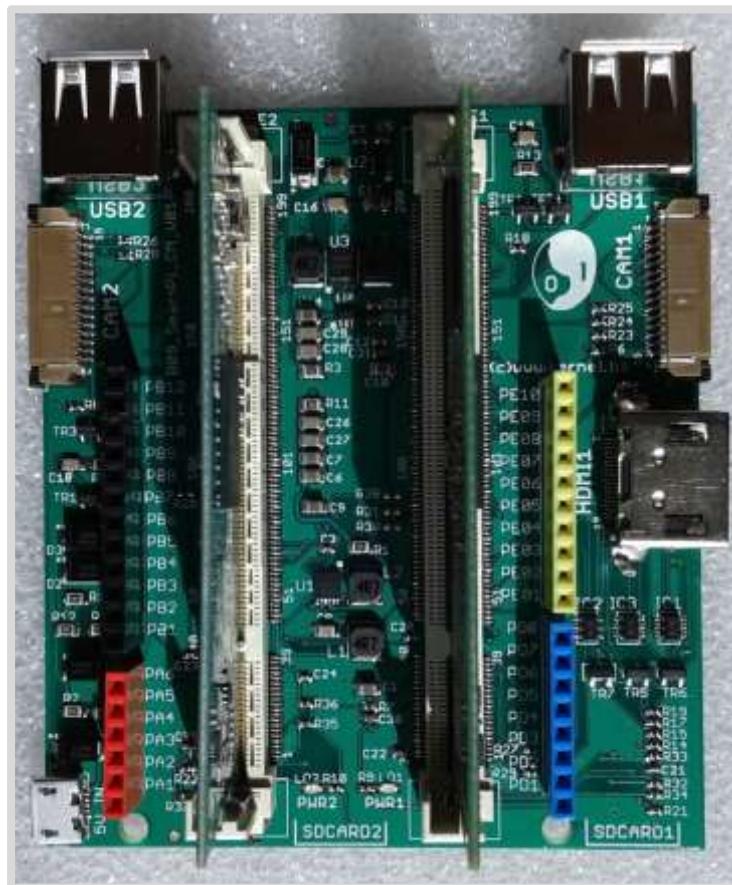


머신러닝/딥러닝

TensorFlow 실습 (강의용)

Third Edition



저작권

머신러닝/딥러닝 TensorFlow 실습(강의용)

저자 정재준

Copyright © 커널연구회(www.kernel.bz). All rights reserved.

Published by 커널연구회(www.kernel.bz).

경기도 남양주시 화도읍 비릉로 321, 104-501

커널연구회는 리눅스 커널과 자료구조 알고리즘을 연구하고 리눅스 시스템 프로그래밍 및 디바이스드라이버 개발을 통하여 창의적인 프로젝트를 수행하여 IoT 관련 제품들을 만들어 일상 생활을 풍요롭고 편리하게 하는데 가치를 두고 있습니다. 아울러 관련 기술들을 교육하여 여러사람들과 공유할 수 있도록 노력하고 있습니다. 커널연구회가 연구 개발한 결과물들은 체계적으로 문서화하여 온라인(www.kernel.bz)상에서 무료 혹은 유료로 제공하고 있습니다. 커널연구회가 제공하는 저작물에는 저작권을 표시하고 있으며 사용자는 저작권 표시를 보존해 주어야 합니다. 커널연구회가 유료로 제공하는 저작물은 사용자에게 개인키(암호)를 부여 하므로 개인키를 타인에게 공개 및 양도하는 일이 없도록 해야 합니다.

기타 자세한 내용들은 커널연구회 웹사이트(www.kernel.bz)를 방문해 주시기 바랍니다.
감사합니다.

발행인: 정재준

발행처: 커널연구회

출판사등록번호: 제2011-75호

출판사등록일: 2011년 09월 27일

전화및팩스: 031-594-5307

ISBN: 978-89-97750-10-8

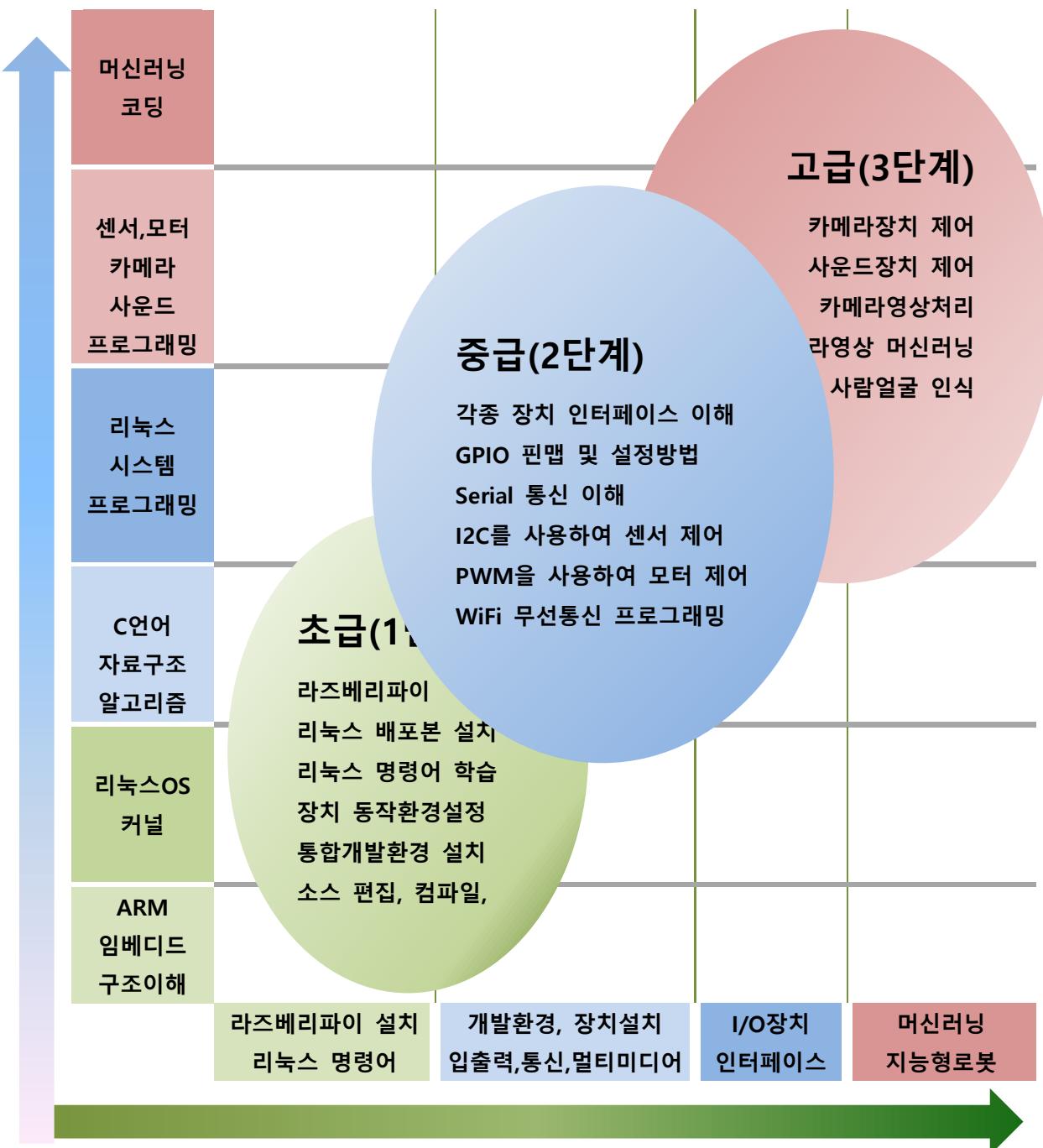
초판 발행일: 2016년 05월 06일

2판 발행일: 2017년 07월 20일



커널연구회 로드맵

연구개발 및 교육



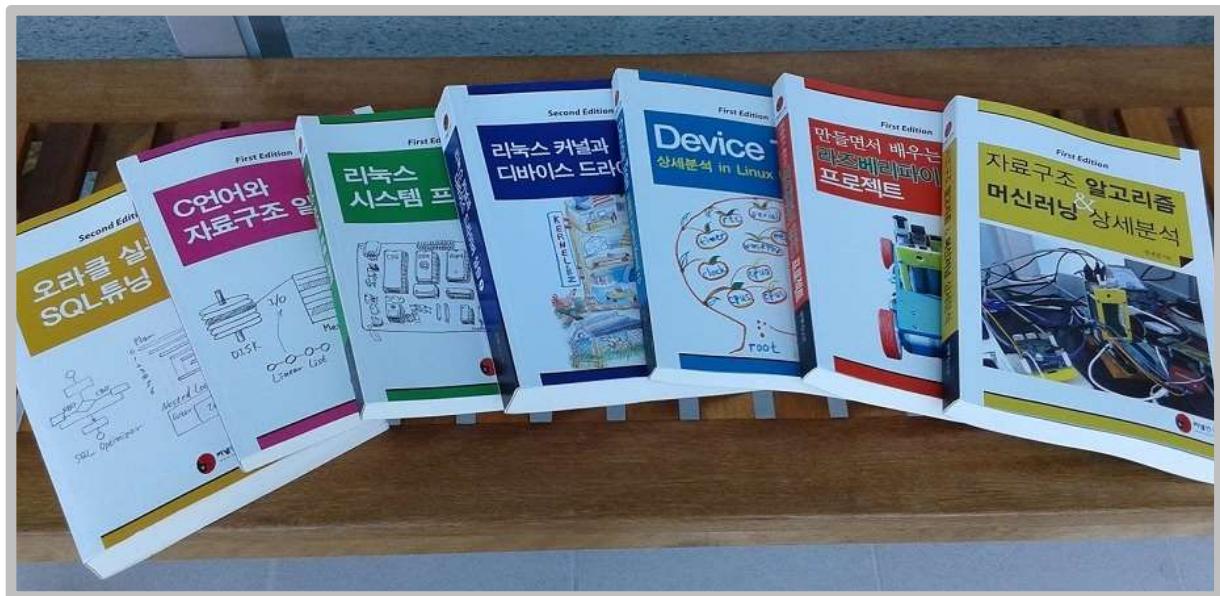
제품 개발 프로젝트

저자 소개



정재준 (rabi3307@nate.com) / 커널연구회(www.kernel.bz)

저자는 학창시절 마이크로프로세서 제어 기술을 배웠으며 리눅스 커널을 연구하고 있다. 15년 이상 쌓아온 실무 경험을 바탕으로 “C언어와 자료구조 알고리즘”, “리눅스 시스템 프로그래밍”, “리눅스 커널과 디바이스드라이버 실습2”, “자료구조 알고리즘 & 머신러닝 상세분석”등의 책을 집필하고, 월간임베디드월드 잡지에 다수의 글을 기고 하였다. 또한 “맞춤형 문장 자동 번역 시스템 및 이를 위한 데이터베이스 구축방법 (The System for the customized automatic sentence translation and database construction method)”라는 내용으로 프로그래밍을 하여 특허청에 특허등록 하였다. 최근에는 서울시 버스와 지하철 교통카드 요금결재 단말기에 들어가는 리눅스 커널과 디바이스 드라이버 개발 프로젝트를 성공적으로 수행했고 여러가지 임베디드 제품을 개발했다. 저자는 스탠포드대학교의 John L. Hennessy 교수의 저서 “Computer Organization and Design” 책을 읽고 깊은 감명을 받았으며, 컴퓨터구조와 자료구조 알고리즘 효율성 연구를 통한 기술서적 집필에 노력하고 있다. 저자는 커널연구회(<http://www.kernel.bz>) 웹사이트를 운영하며 연구개발, 교육, 관련기술 공유 등을 위해 노력하고 있다.



♣ 저자가 집필한 책들

목차

내용

머신러닝/딥러닝	1
저작권	2
커널연구회 로드맵.....	3
저자 소개.....	4
목차	5
머신러닝/딥러닝 TENSORFLOW 실습	8
1. 머신러닝 소개.....	9
1.1 인공지능(AI) 역사.....	9
1.2 인공지능 동작 환경	17
1.3 리눅스용 개발환경 설치	19
1.4 윈도우용 개발환경 설치	24
1.5 TENSORFLOW 기본	34
2. 머신러닝 알고리즘	36
2.1 알고리즘 소개.....	36
2.2 LINEAR REGRESSION	40
2.2.1 가설함수(학습 모델)와 비용함수	40
2.2.2 비용 줄이기(기울기 예측)	47
2.2.3 미분 함수(Convex)	49
2.3 LINEAR REGRESSION LEARNING	53
2.3.1 단항변수 기울기 학습1	53
2.3.2 단항변수 기울기 학습2	55
2.3.3 단항변수 기울기 학습3	57
2.3.4 다항변수 기울기 학습	59
2.3.5 다항변수 매트릭스 처리	62
2.3.6 다항변수 파일 읽기	65
2.4 LOGISTIC(BINARY) CLASSIFICATION	69

2.4.1 학습 모델(Hypothesis)	69
2.4.2 비용 함수.....	73
2.4.3 Logistic Regression	75
2.5 MULTINOMIAL(SOFTMAX) CLASSIFICATION.....	79
2.5.1 학습 모델(Hypothesis)	79
2.5.2 Softmax 함수.....	82
2.5.3 비용 함수.....	85
2.5.4 TensorFlow 실습.....	87
3. DEEP LEARNING.....	95
3.1 딥러닝 기본	96
3.1.1 행동 함수.....	98
3.1.2 XOR 문제.....	99
3.1.3 Neural Network.....	100
3.1.4 Back Propagation.....	107
3.2 XOR 문제 해결 실습	112
3.2.1 일반적인 XOR 문제.....	112
3.2.2 XOR Neural Network	114
3.2.3 XOR Deep Learning.....	117
3.2.4 XOR Deep Learning2.....	124
3.2.5 XOR ReLU.....	127
3.3 딥러닝 정확성 향상	129
3.3.1 ReLU	129
3.3.2 Good Weight (초기값)	131
3.3.3 Overfitting 조정.....	132
3.3.4 DropOut	133
3.3.5 Optimizer 성능 비교.....	135
3.4 딥러닝 실습	136
3.4.1 일반적인 softmax	137
3.4.2 ReLU	140
3.4.3 DropOut	142
3.4.4 초기값 설정.....	145
3.4.5 결과 정리.....	147
4. 영상 인지(CNN)	148
4.1 CONVOLUTION 연산	149

4.2 POOLING 연산	153
4.3 CNN 종류	156
4.3.1 AlexNet	156
4.3.2 GoogLeNet	156
4.3.3 ResNet	157
4.3.4 DeepMind AlphaGo	158
4.4 CNN 실습	159
4.4.1 Adam Optimizer	159
4.4.2 RMS Optimizer	164
4.4.3 결과 정리	168
5. 언어 인지(RNN)	169
5.1 언어 처리	169
5.1.1 전통적인 자료구조 알고리즘 특징	169
5.1.2 언어(문장) 유사도 측정	171
5.1.3 빠른 검색으로 문장 연결	173
5.2 언어 변환	180
5.3 언어 번역	180
5.4 언어 인식(RNN 알고리즘)	182
5.4.1 RNN 0/하//	182
5.4.2 RNN 활용 예	186
5.4.3 LSTM	187
5.4.4 GRU	192

머신러닝/딥러닝 TensorFlow 실습

Third Edition

텐서플로우/파이썬 기반 머신러닝

머신러닝/딥러닝 TensorFlow 실습



(사진설명) 라즈베리파이 CM 모듈을 병렬로 장착할 수 있는 보드(커널연구회 제작)

이책은 1 부와 2 부로 나누어져 있다. 제 1 부에서는 텐서플로우/파이썬 환경에서 실습하면서 머신러닝을 이해할 수 있도록 구성했다. 제 2 부에서는 C 언어로 머신러닝 알고리즘을 직접 코딩하면서 자세히 익힐 수 있도록 구성했다. 제 1 장에서는 머신러닝을 컴퓨터에서 실습할 수 있도록 각종 패키지 프로그램을 설치하고 환경설정하는 방법에 대해서 기술했다. 제 2 장에서는 머신러닝 기본 알고리즘을 이해할 수 있도록 했고, 제 3 장 부터는 Neural Network 을 어떻게 구성하여 딥러닝을 전개해 나가는지 자세히 설명한다. 제 4 장은 CNN 알고리즘에서 어떻게 영상이미지들을 인식하는지 설명하고, 제 5 장에서는 RNN 알고리즘을 사용하여 문자와 음성 데이터를 어떻게 인식하는지 설명한다.

♣ 책에 있는 모든 소스들은 아래 링크에서 다운로드 가능합니다.

<https://github.com/kernel-bz/ml>

<https://github.com/hunkim/DeepLearningZeroToAll>

머신러닝/딥러닝 TensorFlow 실습

Third Edition

1. 머신러닝 소개

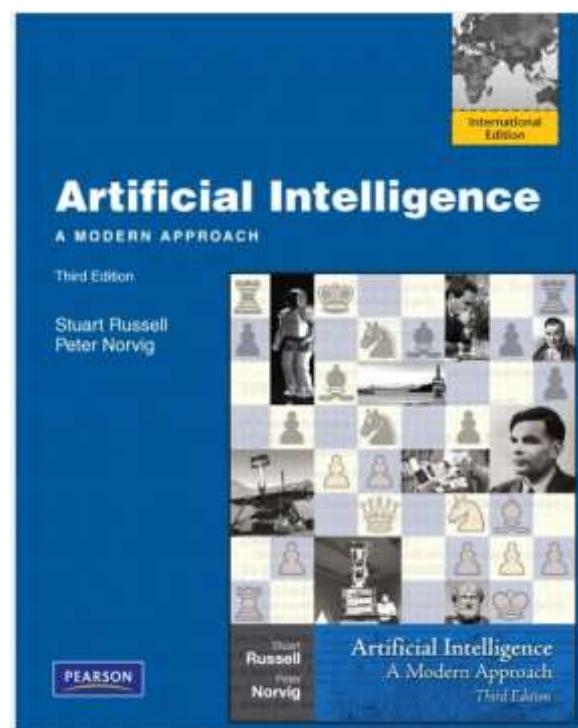
1. 머신러닝 소개

알고리즘은 컴퓨터 하드웨어가 시작되기 전부터 연구되어 왔다. 그러나 현대에 이르러 컴퓨터 하드웨어 성능이 비약적으로 발전하고 많은 데이터들이 축적됨으로 인해서 알고리즘과 인공지능 연구가 새로운 전환기를 맞이하고 있다. 그동안 컴퓨팅 역사에서 해결하지 못하고 있었던 문제들을 머신러닝/딥러닝에서 하나씩 해결하기 시작하면서 이것에 적용된 알고리즘 학습이 중요해졌다. 이번장에서는 먼저 알고리즘과 인공지능의 역사를 통해서 이것이 발전되고 있는 방향에 대해서 알아 보도록 하자.

1.1 인공지능(AI) 역사

인공지능 역사는 Stuart J. Russell and Peter Norvig 의 저서 Artificial Intelligence(Third Edition) 원서를 필자가 의역한 내용으로 다음과 같이 구성했다.

Stuart J. Russell and Peter Norvig의 저서 Artificial Intelligence



4가지 관점의 인공지능

사람처럼 생각하는 것: 인지학적인 접근방식

사람처럼 행동하는 것: 동작 테스트(Turing Test) 접근방식

합리적으로 생각하는 것: 생각의 법칙에 대한 접근

합리적으로 행동하는 것: 합리적인 행동(agent) 접근

인공지능(AI)은 철학(Philosophy), 수학(Mathematics), 경제학(Economics), 신경과학(Neuroscience), 심리학(Psychology), 컴퓨터공학(Computer engineering), 제어이론과 인공두뇌학(Control theory and cybernetics), 언어학(Linguistics)등 대부분의 학문들과 연관되어 있다.

철학(Philosophy)

- 정형적인 규칙들로 타당한 결론들을 이끌어낼 수 있는가?
- 마음(생각)은 우리의 두뇌에서 어떻게 형성되는가?
- 지식은 어디에서 오는가?
- 지식은 어떻게 행동을 유발시키는가?

아리스토텔레스(384-322 B.C.): 이성적인 법칙, 삼단논법

Ramon Lull(d. 1315): 기계적인 장치를 통하여 이성에 접근가능할 수도 있다는 생각.



Thomas Hobbes(1588-1679):

이성은 수치적인 계산으로 표현(생각을 더하고 빼는 것)과 유사.

레오나르도 다빈치(1452-1519): 기계적인 계산기 설계(당시 기술로는 제작하지 못함)

수학(Mathematics)

- 타당한 결론을 이끌어내기 위한 정형화된 규칙은 무엇인가?(논리)
- 무엇이 계산될 수 있는가?(계산)
- 불확실한 정보에 어떻게 의미를 부여하는가?(확률)

George Boole(1815-1864): 명제, Boolean, 논리등에 대해서 연구.

Gottlob Frege(1848-1925): Boole 의 논리를 좀 더 확장시켜 객체와 관계들을 처음으로 포함시킴.



Alfred Tarski(1902-1983): 실제 생활속의 객체들에 논리를 연관시키는 방법연구 → 알고리즘.



Alan Turing(1912-1954):
수학자, 컴퓨터 과학자, 알고리즘 성능 테스트 → Turing machine.

경제학(Economics)

- 급여(보상)를 최대화 하기위해서 우리는 어떻게 결정해야 하는가?
- 다른 사람들이 계속하지 않는다면 우리는 어떻게 해야 하는가?
- 급여가 앞으로 점점 불확실해 진다면 우리는 어떻게 해야 하는가?

Adam Smith(1723-1790): 경제를 과학적으로 접근.

Leon Walras(1834-1910): 경제를 보상의 관점에서 해석. → 게임 이론과 경제적 행동으로 발전



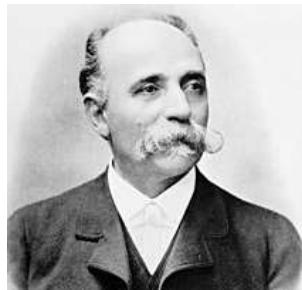
Herbert A. Simon(1916-2001): AI 영역을 개척하며 노벨 경제학상 수상.

"**적정만족추구자(Satisficer)**는 바늘을 찾는 순간 더 이상 건초 더미를 파헤치지 않는 사람이며, **최적만족추구자(Optimizer)**라면 건초 더미 전체를 파헤쳐서 최대한 많은 바늘을 찾은 다음 가장 예리한 바늘을 선택한다"

신경과학(Neuroscience)

- 두뇌는 어떻게 정보를 처리하는가?

Paul Broca(1824-1880): 해부학자. 뇌손상을 입은 환자의 실어증에 관한 연구에서 뇌에는 특정적인 것의 인식을 담당하는 부분이 존재함을 인지.



Camillo Golgi(1843-1926):
신경세포(뉴런)에 대해서 인지하고 연구, 관찰.

Nicolas Rashevsky(1899-1972): 신경 시스템 연구에 수학적 모델을 최초로 적용.

Han 과 Boyden(2007): 뉴런들은 전기적, 화학적 혹은 광학적으로 자극된다고 언급.



John Searle(1932-): 다음과 같은 간결한 표현으로 뇌가 마음을 형성한다고 했다.

"간단한 세포들의 집합으로 생각과 행동, 자각을 이끌어 낸다는 것은
실로 놀라운 결론이다."

심리학(Psychology)

- 사람과 동물은 어떻게 생각하고 행동하는가?



Hermann von Helmholtz(1821-1894): 심리학을 과학적으로 연구.
"사람을 물리적 및 생리학적인 시각으로 바라보는 한가지 매우 중요한 논문"

H. S. Jennings(1906): 저급 생물의 행동을 관찰.



Kenneth Craik(1914-1945): 본성에 대해서 믿음과 목표 지향적인 "정신"이라는 관점으로 재정립.

- (1)자극은 내부적인 표현으로 변환되어야 한다.
- (2)표현은 새로운 내부 표현들을 이끌어 내기 위해서 인지적인 방식으로 처리된다.
- (3)이러한 것들은 행동으로 다시 변환된다.

Donald Broadbent(1926-1993): 정보처리를 최초로 심리학적인 현상으로 모델링.

컴퓨터 공학(Computer engineering)

- 우리는 어떻게 효율적인 컴퓨터를 만들 수 있을까?

인공지능이 성공의 길로 가기 위해서는 2가지가 필요한데, 지능과 인공물이다. 컴퓨터는 인공물로 선택되어 왔다. 현대의 디지털 전자기기인 컴퓨터는 세계2차대전때 과학자들에 의해서 발명되었다. 1940년 Alan Turing 팀에 의해서 만들어진 컴퓨터는 전자기계적으로 너무 복잡했으며 단지 독일군의 암호문을 해독하고자 하는 한가지 목적을 위한 것이었다. 이들은 1943년에 진공관을 사용하여 일반 목적의 거대한 머신을 개발했다. 세계2차대전 이후, Turing은 이러한 컴퓨터들을 AI 연구에 사용하고자 했다. Turing은 1953년에 최초의 체스 게임 프로그램을 개발했다.

Konrad Zuse가 발명한 Z-3



동작을 프로그램할 수 있는 최초의 컴퓨터는 1941년 독일의 Konrad Zuse가 발명한 Z-3 이었다. Zuse는 또한 실수와 최초의 고급 프로그래밍 언어인 Plankalkül을 개발했다.



최초의 전자적인 컴퓨터인 ABC는 1940년과 1942년 사이에 아이오와 주립대학교에서 John Atanasoff와 그의 제자 Clifford Berry에 의해서 만들어졌다.

Atanasoff의 연구는 몇가지 지원과 인정을 받았는데 이것이 ENIAC 이었다. ENIAC은 펜실베니아 대학교에서 John Mauchly와 John Eckert가 포함된 팀에 의해서 비밀 군사 프로젝트로 진행된 것 이었으며, 이것은 현대 컴퓨터들에 막대한 영향을 준 선구자적인 프로젝트였다.

그이후로, 컴퓨터 하드웨어의 각 세대는 속도와 용량은 증가하고 가격은 감소하는 방향으로 발전 했다. 컴퓨터의 수행능력은 2005년까지 18개월마다 두배로 증가했다. 전력 소비 문제를 해결하기 위해서 제조사들은 CPU의 클럭 속도보다는 코어의 수를 널리기 시작했다. 현재는 사람의 두뇌 능력에 접근하기 위해서 CPU를 병렬화 시키는 문제로 인해서 전력 소모가 증가할 것이라고 예상하고 있다.

AI는 컴퓨터 과학의 소프트웨어적인 측면으로도 접근한다. 운영체제, 프로그래밍 언어, 현대적인 프로그램들을 작성하는데 필요한 도구들이 여기에 해당한다. 이것들은 AI 분야를 개척하는데 필요한 일부분이 될 것이다. AI는 컴퓨터 과학에 많은 아이디어를 제공하여 큰흐름을 개척했다. 예를들면, 시분할처리, 대화형 번역기, 마우스와 윈도우즈가 장착된 개인용 컴퓨터, 빠른 개발환경, 링크드 리스트 데이터형, 자동 저장 관리기, 기호화를 위한 핵심적인 개념, 함수, 선언, 객체지향 프로그래밍등이 여기에 해당한다.

제어이론과 인공두뇌학(Control theory and cybernetics)

- 인공물이 어떻게 스스로 동작할 수 있을까?

알렉산드리아의 Ktesibios(250 B.C.)는 최초로 스스로 제어하는 기계를 만들었다. 물시계 조절장치를 통하여 흐름 속도를 지속적으로 유지하도록 한 기계였다. 이 발명은 인공물이 하는 역할에 대한 정의를 재설정하는 계기가 되었다. 그 이전에는 오직 살아 있는 것만이 환경 변화에 대응하여

행동을 변경할 수 있다고 생각했다.



스스로 조절하는 능력이 있는 제어 시스템의 또 다른 예로 증기기관 엔진이 있는데, 이것은 James Watt(1736-1819)에 의해서 발명되었고, 온도조절 장치는 Cornelis Drebbel(1572-1633)가 발명했다.

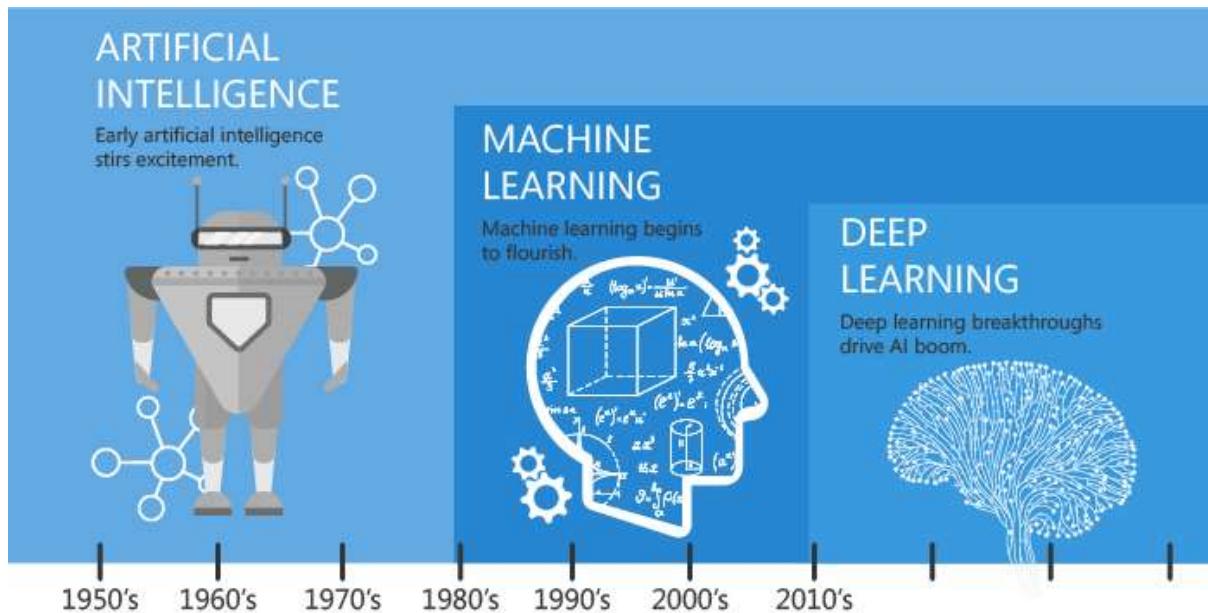
언어학(Linguistics)

- 언어는 어떻게 생각과 연관되어 있는가?
- 어린이들이 전에 들어보지도 못한 문장들을 어떻게 이해하고 만들어 내는가?

현대 언어학과 AI는 거의 동시대에 탄생했고 같이 성장하면서 계산적인 언어학 혹은 자연어 처리라는 유합된 분야에서 서로 교차되고 있다. 언어를 이해하는 문제는 1957년에 출현한 것보다도 더 복잡해 졌다. 언어 이해는 문장 구조를 이해하는 것보다 주제와 문맥 이해가 더 중요하다.

지금까지 인공지능의 역사와 앞으로의 발전방향을 아래의 그림을 통해서 요약할 수 있다.

요약 정리



Since an early flush of optimism in the 1950's, smaller subsets of artificial intelligence - first machine learning, then deep learning, a subset of machine learning - have created ever larger disruptions.

그림출처: <https://www.quora.com/What-are-the-differences-between-AI-machine-learning-deep-learning-and-neural-network>

머신러닝, 딥러닝의 선구자



Geoffrey Hinton (born 6 December 1947):

영국의 인지 심리학자이자 컴퓨터 과학자로 인공 신경망에 대한 연구로 유명합니다. 2015년 현재 그는 Google과 토론토 대학에서 일하고 있습니다. 그는 다층 신경망을 훈련하기 위해 일반화된 backpropagation 알고리즘의 사용을 시연한 최초의 연구자 중 한 명으로 딥러닝 공동체에서 중요한 인물이며 "AI의 대부"라고 불리지고 있습니다. 2012년 Imagenet 도전 대회에서 그의 극적인 이미지 인식 이정표는 컴퓨터 비전 분야에 혁명을 일으켰습니다.



Yann LeCun (born 1960 in France):

기계 학습, 컴퓨터 비전, 모바일 로봇 공학 및 컴퓨터 신경 과학에 기여한 컴퓨터 과학자이다. 그는 CNN(convolutional neural networks)을 사용하여 광학 문자 인식 및 컴퓨터 비전 작업을 하는 것으로 잘 알려져 있으며 회선 망(convolutional net)의 창립자이기도 합니다. 그는 또한 DjVu 이미지 압축 기술의 주요 제작자 중 하나입니다. 그는 Léon Bottou와 Lush 프로그래밍 언어를 공동 개발했습니다.



Sepp Hochreiter (born 1967):

독일의 컴퓨터 과학자입니다. 2006년부터 그는 Johannes Kepler University of Linz의 생물 정보학 연구소 소장을 지냈습니다. 그는 기계 학습 및 생물 정보학 분야에서 많은 공헌을했습니다. 그는 1991년 졸업 논문에서 첫번째 결과가 보고된 장기간 단기 기억(LSTM)을 개발했습니다. 주요 LSTM 논문은 1997년에 출간되었으며 기계 학습의 타임 라인에서 획기적인 발견으로 간주됩니다.

1.2 인공지능 동작 환경

환경 → 센서 → 인지 → 행동 → 구동기 → 환경

좋은 인공지능은 합리적인 인지와 행동을 한다.

인공지능의 여러가지 형태:

(1)택시 기사(자율주행):

수행능력: 안전, 신속, 준법, 편안, 최대이익

환경: 도로, 교통흐름, 보행자, 손님

센서: 속도계, GPS, 카메라, 엔진상태표시

구동기: 핸들, 가속페달, 브레이크, 경적

(2)의료진단 장비:

수행능력: 환자의 건강 점검, 비용절감

환경: 병원, 환자, 의료진

센서: 판독, 환자의 응답, 발견

구동기: 검사, 진단, 질문, 정보표시

(3)인공위성 이미지 분석:

수행능력: 정확한 이미지 선별, 분석

환경: 인공위성으로부터 이미지 다운로드

센서: 화소, 색상 배열

구동기: 이미지 표시 장치

(4)로봇 선별기:

수행능력: 선별의 정확성

환경: 바구니가 달린 컨베이어 벨트

센서: 카메라, 조립된 각도 센서

구동기: 조립된 팔과 손

(5)정유 제어장치:

수행능력: 순수성, 순도, 선별성, 안전성

환경: 정유 작동기

센서: 온도, 압력, 화학적 센서

구동기: 밸브, 펌프, 표시장치

(6) 대화형 영어 교사:

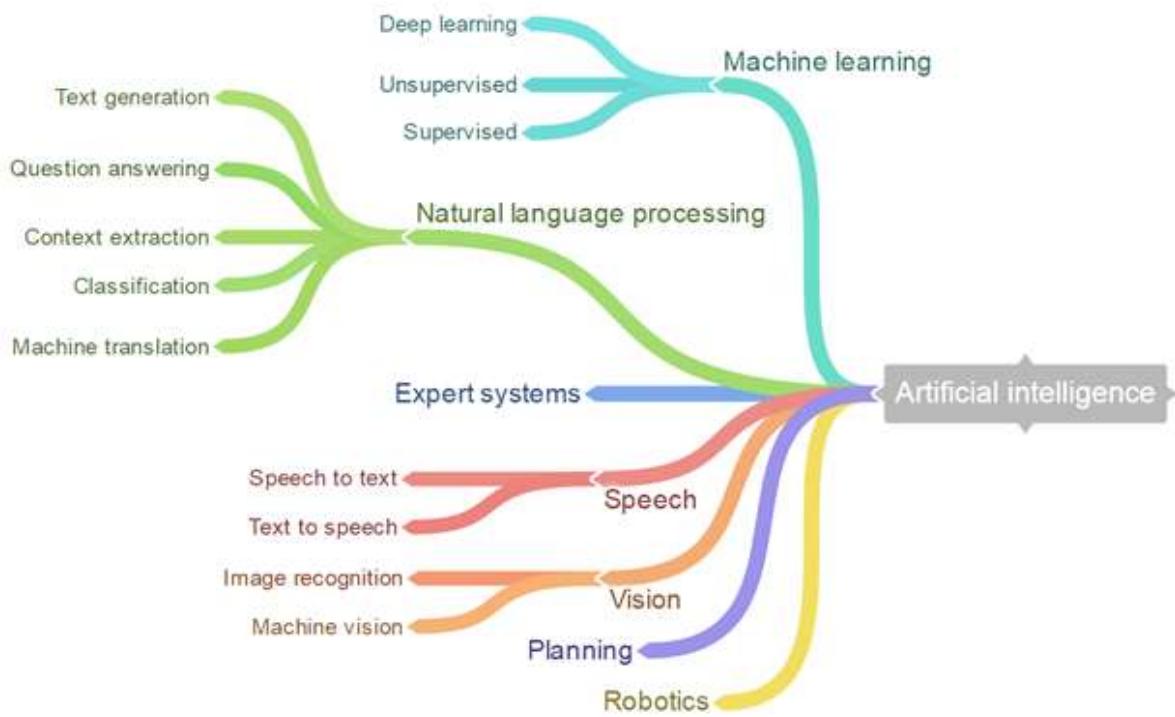
수행능력: 학생의 시험점수

환경: 학생들, 시험기관

센서: 키보드 입력, 발음

구동기: 연습문제, 제안, 교정

인공지능 환경 요약



그림출처: <https://hackernoon.com/jump-start-to-artificial-intelligence-f6eb30d624ec>

1.3 리눅스용 개발환경 설치

TensorFlow는 구글의 브레인(Brain)팀에서 머신러닝을 위해 개발한 오픈소스 라이브러리이다. 실제로 구글에서는 이것을 음성검색, 광고, 구글 포토, 구글 맵스, 스트리트뷰, 번역, 유튜브 등과 같은 실제 서비스에 활용하고 있다. TensorFlow는 다양한 플랫폼에서 활용가능하다. 필자는 라즈베리파이에서 실습하고 있으므로 여기에 python과 TensorFlow를 설치하여 머신러닝 알고리즘들을 실제로 실행하여 결과를 확인하는 방법으로 설명을 진행한다.

라즈베리파이3



라즈베리파이3 하드웨어 사양

- 1.2GHz quad-core ARM Cortex-A53 CPU
- 1GB RAM (built-in)
- 4 USB ports
- 40 GPIO pins
- Full HDMI port
- Ethernet port
- Combined 3.5mm audio jack and composite video
- Camera interface (CSI)
- Display interface (DSI)
- Micro SD card slot
- VideoCore IV 3D graphics core

TensorFlow 에 대해서는 아래의 웹사이트에서 좀더 많은 정보를 확인할 수 있다.

공식 웹사이트:

<https://www.tensorflow.org/>

위키피디아 문서:

<https://ko.wikipedia.org/wiki/TensorFlow>

TensorFlow 는 python 을 사용하여 실행할 수 있는 인터페이스(API)를 많이 제공하므로,

우선 python 을 다음과 같이 설치하도록 하자.

(좀더 자세한 설치 방법은 https://www.tensorflow.org/install/install_linux 참조 바람)

Python 설치

```
$ sudo apt-get install python-pip python-dev
```

아래와 같이 설치가 진행된다.

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
python-pip is already the newest version.
The following packages were automatically installed and are no longer required:
  libegl1-mesa-drivers libopenvg1-mesa
Use 'apt-get autoremove' to remove them.
The following extra packages will be installed:
  libpython-dev libpython2.7-dev python2.7-dev
The following NEW packages will be installed:
  libpython-dev libpython2.7-dev python-dev python2.7-dev
0 upgraded, 4 newly installed, 0 to remove and 188 not upgraded.
Need to get 18.2 MB of archives.
After this operation, 25.7 MB of additional disk space will be used.
Do you want to continue? [Y/n]
Get:1 http://mirrordirector.raspbian.org/raspbian/ jessie/main libpython2.7-dev armhf 2.7.9-2 [17.9 MB]
Get:2 http://mirrordirector.raspbian.org/raspbian/ jessie/main python-dev armhf 2.7.9-1 [1,188 B]
Get:3 http://mirrordirector.raspbian.org/raspbian/ jessie/main libpython-dev armhf 2.7.9-1 [19.6 kB]
Get:4 http://mirrordirector.raspbian.org/raspbian/ jessie/main python2.7-dev armhf 2.7.9-2 [281 kB]
Fetched 18.2 MB in 26s (688 kB/s)
Selecting previously unselected package libpython2.7-dev:armhf.
(Reading database ... 141536 files and directories currently installed.)
Preparing to unpack .../libpython2.7-dev_2.7.9-2_armhf.deb ...
Unpacking libpython2.7-dev:armhf (2.7.9-2) ...
Selecting previously unselected package libpython-dev:armhf.
Preparing to unpack .../libpython-dev_2.7.9-1_armhf.deb ...
Unpacking libpython-dev:armhf (2.7.9-1) ...
Selecting previously unselected package python2.7-dev.
```

```
Preparing to unpack .../python2.7-dev_2.7.9-2_armhf.deb ...
Unpacking python2.7-dev (2.7.9-2) ...
Selecting previously unselected package python-dev.
Preparing to unpack .../python-dev_2.7.9-1_armhf.deb ...
Unpacking python-dev (2.7.9-1) ...
Processing triggers for man-db (2.7.0.2-5) ...
Setting up libpython2.7-dev:armhf (2.7.9-2) ...
Setting up libpython-dev:armhf (2.7.9-1) ...
Setting up python2.7-dev (2.7.9-2) ...
Setting up python-dev (2.7.9-1) ...
```

pip 을 다음과 같이 설치한다..

```
$ sudo pip install -i https://pypi.python.org/simple/ --upgrade pip
```

리눅스 우분투(16.04 버전, 64 비트 아키텍쳐 권장)에서는 TensorFlow 를 다음과 같이 시스템 사양에 따라서 하나를 선별하여 설치한다.

```
$ pip install tensorflow      # Python 2.7; CPU support (no GPU support)
$ pip3 install tensorflow    # Python 3.n; CPU support (no GPU support)
$ pip install tensorflow-gpu # Python 2.7; GPU support
$ pip3 install tensorflow-gpu # Python 3.n; GPU support
```

그리나 라즈베리파이는 32 비트 아키텍처를 위한 것이므로 위와 같이 설치 할 수 없다. 인터넷을 검색해본 결과, 다행히 라즈베리파이의 32 비트 아키텍처용 TensorFlow 설치버전을 <https://github.com/samjabr ahams/tensorflow-on-raspberry-pi/> 에서 배포하고 있었다. 다음과 같이 다운로드 하면 된다.

```
$ wget https://github.com/samjabr ahams/tensorflow-on-raspberry-
pi/releases/download/v1.1.0/tensorflow-1.1.0-cp27-none-linux_armv7l.whl
```

다운로드가 완료되면 아래와 같이 TensorFlow 를 설치한다.

TensorFlow 설치

```
$ sudo pip install tensorflow-1.1.0-cp27-none-linux_armv7l.whl
```

아래처럼 설치 메시지가 표시되면서 설치 진행된다.

```
Unpacking ./tensorflow-0.7.1-cp27-none-linux_armv7l.whl
Requirement already satisfied (use --upgrade to upgrade): numpy>=1.8.2 in /usr/lib/python2.7/dist-
packages (from tensorflow==0.7.1)
Downloading/unpacking protobuf==3.0.0b2 (from tensorflow==0.7.1)
    Downloading protobuf-3.0.0b2-py3-none-any.whl (326kB): 326kB downloaded
Requirement already satisfied (use --upgrade to upgrade): wheel in /usr/lib/python2.7/dist-packages
(from tensorflow==0.7.1)
Downloading/unpacking six>=1.10.0 (from tensorflow==0.7.1)
    Downloading six-1.10.0-py2.py3-none-any.whl
Requirement already satisfied (use --upgrade to upgrade): setuptools in /usr/lib/python2.7/dist-
packages (from protobuf==3.0.0b2->tensorflow==0.7.1)
Installing collected packages: tensorflow, protobuf, six
    Found existing installation: six 1.8.0
    Not uninstalling six at /usr/lib/python2.7/dist-packages, owned by OS
Successfully installed tensorflow protobuf six
Cleaning up...
```

TensorFlow 가 설치되었으므로 이것을 python 에서 임포트하여 실행해 보도록 하자. 다음과 같이 파일 셈플 소스를 코딩한다.

TensorFlow 테스트용 파일 예제

```
''''
HelloWorld example using TensorFlow library.

Author: Aymeric Damien
Project: https://github.com/aymericdamien/TensorFlow-Examples/
''''

import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

import tensorflow as tf;

#Simple hello world using TensorFlow

# Create a Constant op
# The op is added as a node to the default graph.
#
# The value returned by the constructor represents the output
# of the Constant op.
hello = tf.constant('Hello, TensorFlow!');

# Start tf session
sess = tf.Session();

print (sess.run(hello));
```

위의 파이썬 예제 소스에서 `import tensorflow as tf` 구문이 TensorFlow 라이브러리를 파이썬으로 읽어들이는 역할을 한다. 위의 예제를 파이썬으로 실행하여 다음과 같이 "Hello, TensorFlow!"라는 문자열이 화면에 출력되면 정상적으로 동작하는 것이다.

실행결과

```
Hello, TensorFlow!
```

TensorFlow 버전 확인은 다음과 같이 파이썬 코드를 작성하여 실행하면 된다.

TensorFlow 버전표시

```
import os  
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'  
  
import tensorflow as tf;  
tf.__version__;
```

1.4 윈도우용 개발환경 설치

TensorFlow가 처음 나왔을 때는 윈도우용 버전을 지원하지 않았으나, 최근에는 윈도우 환경에서도 TensorFlow를 실행할 수 있는 배포본을 제공하고 있다. 필자는 아래 웹사이트를 참조하여 윈도우에 TensorFlow를 설치하고 개발환경을 구축했다.

<http://comajava.blogspot.kr/2017/01/windows-tensorflow-python-35-anaconda.html>

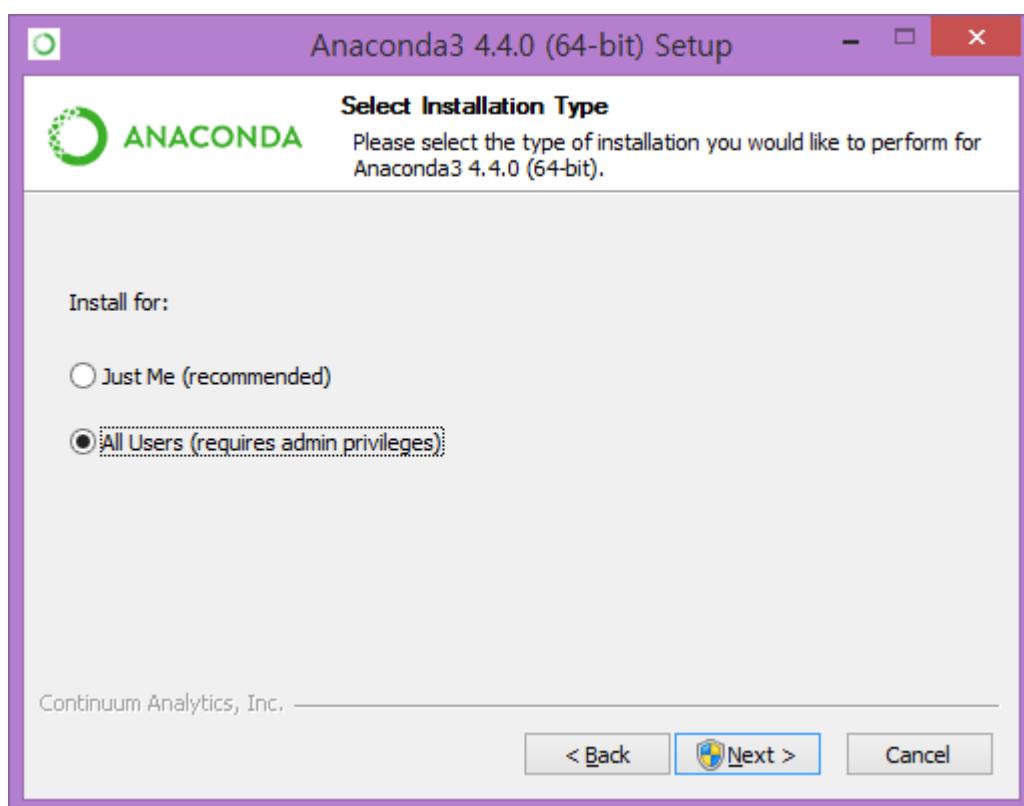
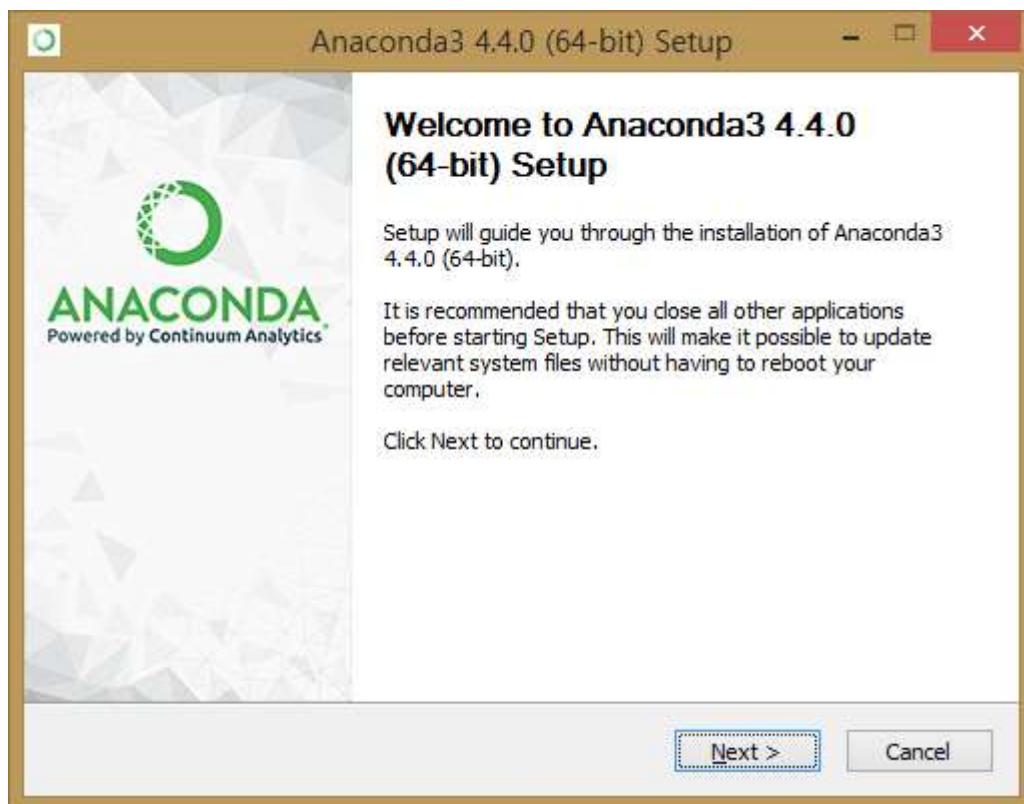
파이썬 개발 환경(아나콘다) 설치

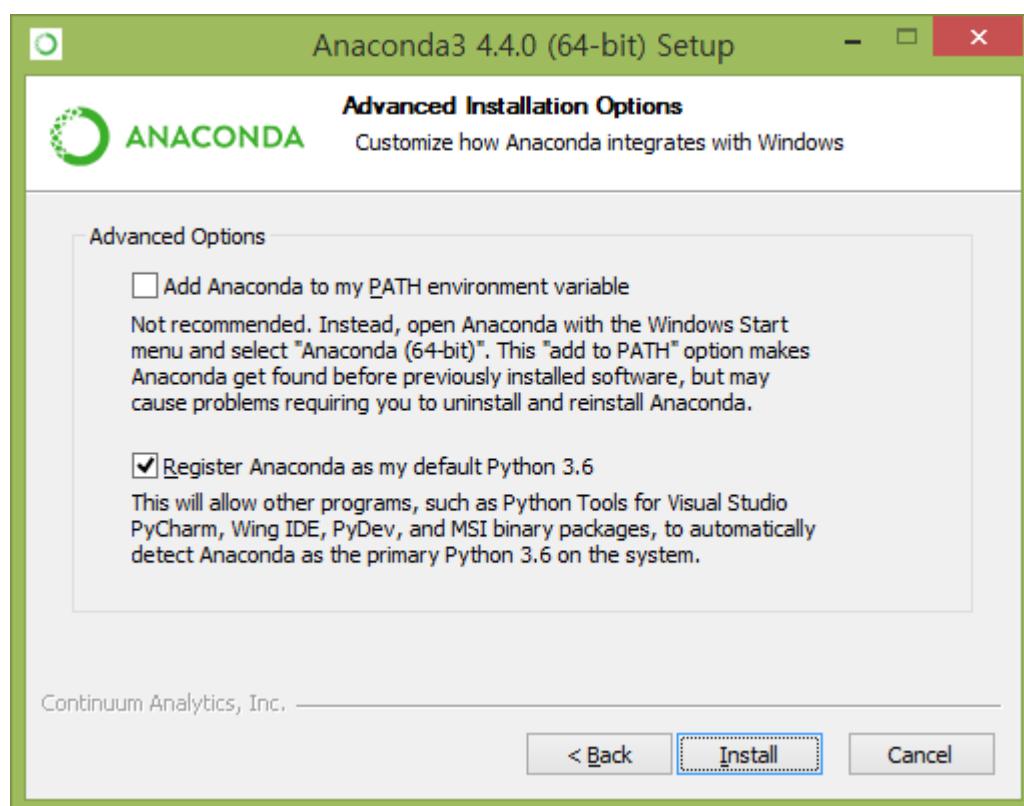
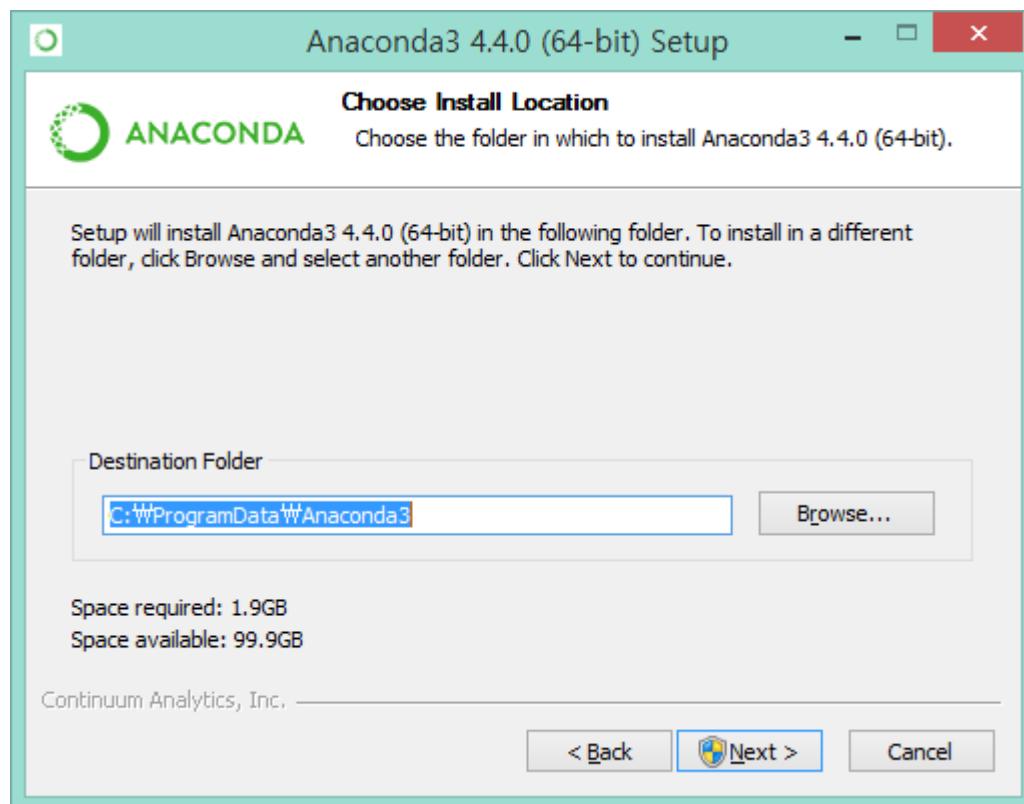
<https://www.anaconda.com/download/>

위의 링크에서 윈도우용 파이썬 3.6 버전 파일(파이썬 3.6.1 버전, 아나콘다 3 4.4.0 버전)을 다운로드 한다.

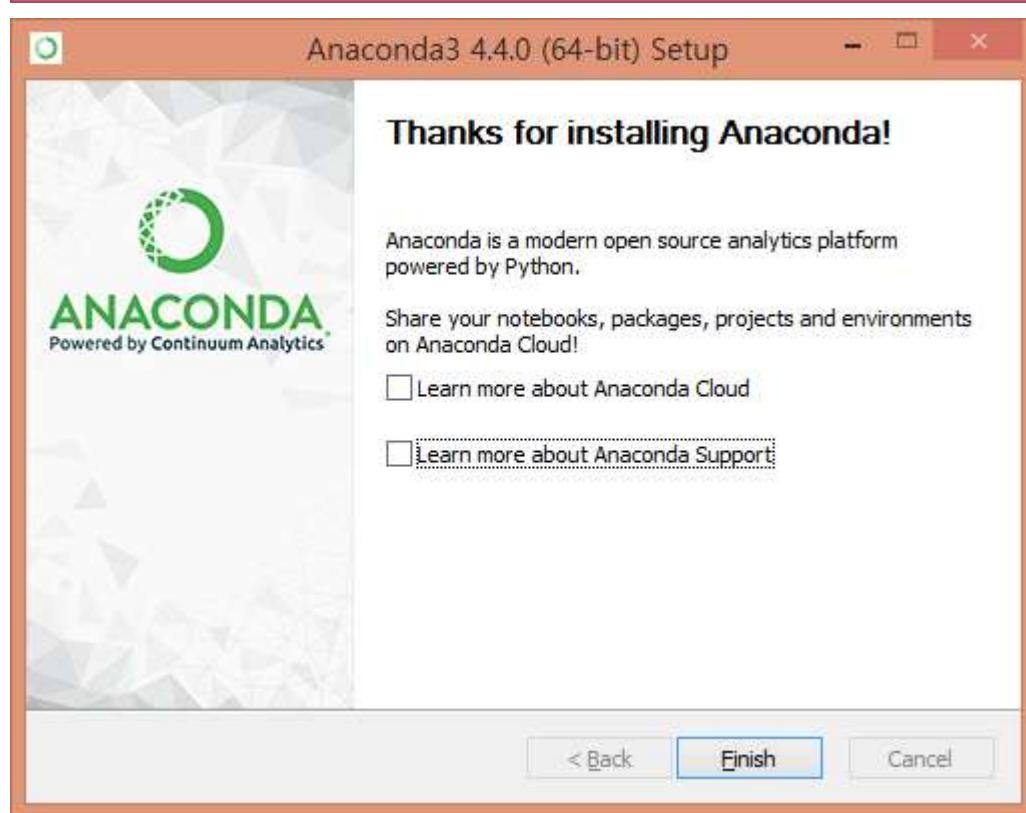
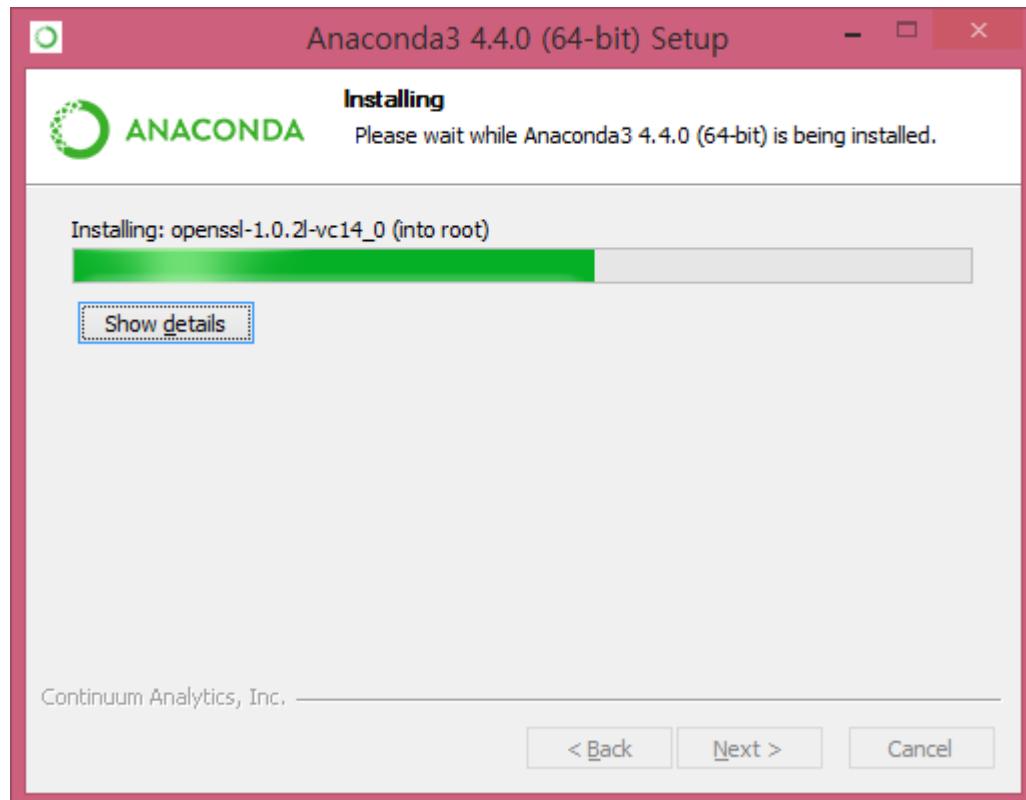


다운로드한 파일(Acnda3-5.2.0-Windows-x86_64.exe)을 마우스 오른쪽 클릭하여 반드시 관리자 권한으로 실행한다. 관리자 권한으로 설치하면 설치경로가 기본적으로 정해진 경로에 설치된다. 필자의 경우는 C:\ProgramData\Anaconda3에 설치 진행되었다. 이 경로는 윈도우에서 보안상 숨겨진 경로이다.





위와 같이 체크한 후 Install 버튼을 클릭하여 설치 진행한다. 설치시간이 꽤 오래 걸린다. (약 10~15 분 소요)



아나콘다(파이썬 3.6.1 버전, 아나콘다 3 4.4.0 버전)가 설치 완료 되었다. 이제 TensorFlow 를 설치하고 개발환경을 실행하기 위해서 윈도우 아이콘의 팝업메뉴에서 다음과 같이 명령 프롬프트를 관리자 권한으로 실행한다.



명령 프롬프트에서 다음과 같이 명령을 입력하여 아나콘다 3 가 설치된 경로로 이동한다.

```
C:\Windows\system32>cd \ProgramData\Anaconda3  
C:\ProgramData\Anaconda3>
```

TensorFlow 를 설치할 때 사용하는 프로그램인 pip 을 다음과 같이 설치한다.

```
C:\ProgramData\Anaconda3>Scripts\easy_install.exe pip
```

pip 을 검색해서 설치하는데 다소 시간이 걸린다. 다음과 같이 설치 로그가 출력되면서 설치된다.

```
Searching for pip
Best match: pip 9.0.1
Adding pip 9.0.1 to easy-install.pth file
Installing pip-script.py script to C:\ProgramData\Anaconda3\Scripts
Installing pip.exe script to C:\ProgramData\Anaconda3\Scripts
Installing pip3-script.py script to C:\ProgramData\Anaconda3\Scripts
Installing pip3.exe script to C:\ProgramData\Anaconda3\Scripts
Installing pip3.6-script.py script to C:\ProgramData\Anaconda3\Scripts
Installing pip3.6.exe script to C:\ProgramData\Anaconda3\Scripts
```

```
Using c:\programdata\anaconda3\lib\site-packages
Processing dependencies for pip
Finished processing dependencies for pip
```

pip 이 설치되면 이것의 버전을 다음과 같이 확인한다.

```
C:\ProgramData\Anaconda3>Scripts\pip.exe --version
pip 9.0.1 from C:\ProgramData\Anaconda3\lib\site-packages (python 3.6)
```

pip 이 설치 되었으므로 이것을 사용하여 다음과 같이 TensorFlow 를 설치한다.

```
C:\ProgramData\Anaconda3>Scripts\pip.exe install tensorflow
```

TensorFlow 가 설치되는데 다소 시간이 걸린다. 다음과 같이 로그가 출력되면서 설치 진행된다.
필자의 경우 "tensorflow-1.2.1-cp36-cp36m-win_amd64.whl" 가 설치 되었다.

```
Collecting tensorflow
  Using cached tensorflow-1.2.1-cp36-cp36m-win_amd64.whl
```

```
Requirement already satisfied: six>=1.10.0 in c:\programdata\anaconda3\lib\site-packages (from tensorflow)
Collecting markdown>=2.6.8 (from tensorflow)
Collecting html5lib==0.9999999 (from tensorflow)
Collecting backports.weakref==1.0rc1 (from tensorflow)
    Using cached backports.weakref-1.0rc1-py3-none-any.whl
Requirement already satisfied: werkzeug>=0.11.10 in c:\programdata\anaconda3\lib\site-packages (from tensorflow)
Requirement already satisfied: numpy>=1.11.0 in c:\programdata\anaconda3\lib\site-packages (from tensorflow)
Collecting protobuf>=3.2.0 (from tensorflow)
Requirement already satisfied: bleach==1.5.0 in c:\programdata\anaconda3\lib\site-packages (from tensorflow)
Requirement already satisfied: wheel>=0.26 in c:\programdata\anaconda3\lib\site-packages (from tensorflow)
Requirement already satisfied: setuptools in c:\programdata\anaconda3\lib\site-packages\setuptools-27.2.0-py3.6.egg (from protobuf>=3.2.0->tensorflow)
Installing collected packages: markdown, html5lib, backports.weakref, protobuf, tensorflow
    Found existing installation: html5lib 0.999
    DEPRECATION: Uninstalling a distutils installed project (html5lib) has been deprecated and will be removed in a future version. This is due to the fact that uninstalling a distutils project will only partially uninstall the project.
    Uninstalling html5lib-0.999:
        Successfully uninstalled html5lib-0.999
Successfully installed backports.weakref-1.0rc1 html5lib-0.9999999 markdown-2.6.8 protobuf-3.3.0 tensorflow-1.2.1
```

TensorFlow 가 설치되면 다음과 같이 conda.exe 을 실행하여 관련되는 정보를 생성한다.

```
C:\ProgramData\Anaconda3>Scripts\conda.exe create -n tensorflow

Fetching package metadata .....
Solving package specifications:

Package plan for installation in environment C:\ProgramData\Anaconda3\envs\tensorflow
```

```
rfloow:  
  
Proceed ([y]/n)? y ← y 을 입력한다.  
  
#  
# To activate this environment, use:  
# > activate tensorflow  
#  
# To deactivate this environment, use:  
# > deactivate tensorflow  
#  
# * for power-users using bash, you must source  
#
```

다음과 같이 TensorFlow 을 활성화 시킨다.

```
C:\ProgramData\Anaconda3>Scripts\activate tensorflow  
  
(tensorflow) C:\ProgramData\Anaconda3>
```

위와 같이 실행하면 (tensorflow)라는 프롬프트가 추가 된다. 이 프롬프트에서 통합개발환경인 spyder 를 다음과 같이 실행한다.

```
(tensorflow) C:\ProgramData\Anaconda3> Scripts\spyder .exe
```

Spyder 가 다음과 같이 실행된다.



Spyder 파이썬 콘솔에서 다음과 같이 TensorFlow 버전을 확인한다.

TensorFlow 버전 확인

```
import tensorflow as tf;
tf.__version__;
```

The screenshot shows the Spyder Python console window. At the top, there are tabs for 'Python 1' and two 'test.py' files. A note at the top of the console window reads: 'NOTE: The Python console is going to be REMOVED in Spyder 3.2. Please start to migrate your work to the IPython console instead.' Below this, the Python interpreter output is shown:

```
Python 3.6.1 |Anaconda 4.4.0 (64-bit)| (default, May 11 2017, 13:25:24) [MSC v.1
900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> runfile('C:/Users/James/.spyder-py3/temp.py', wdir='C:/Users/James/.spyder-p
y3')
>>>
>>> import tensorflow as tf;
>>> tf.__version__;
'1.2.1'
>>>
```

At the bottom of the console window, there are tabs for 'Python console', 'History log', and 'IPython console'. The 'Python console' tab is currently selected.

이제 Spyder 편집창에서 아래의 파이썬 예제를 실행해 보자.

TensorFlow 파이썬 예제

```
...
HelloWorld example using TensorFlow library.

Author: Aymeric Damien
Project: https://github.com/aymericdamien/TensorFlow-Examples/
...

import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

import tensorflow as tf;
```

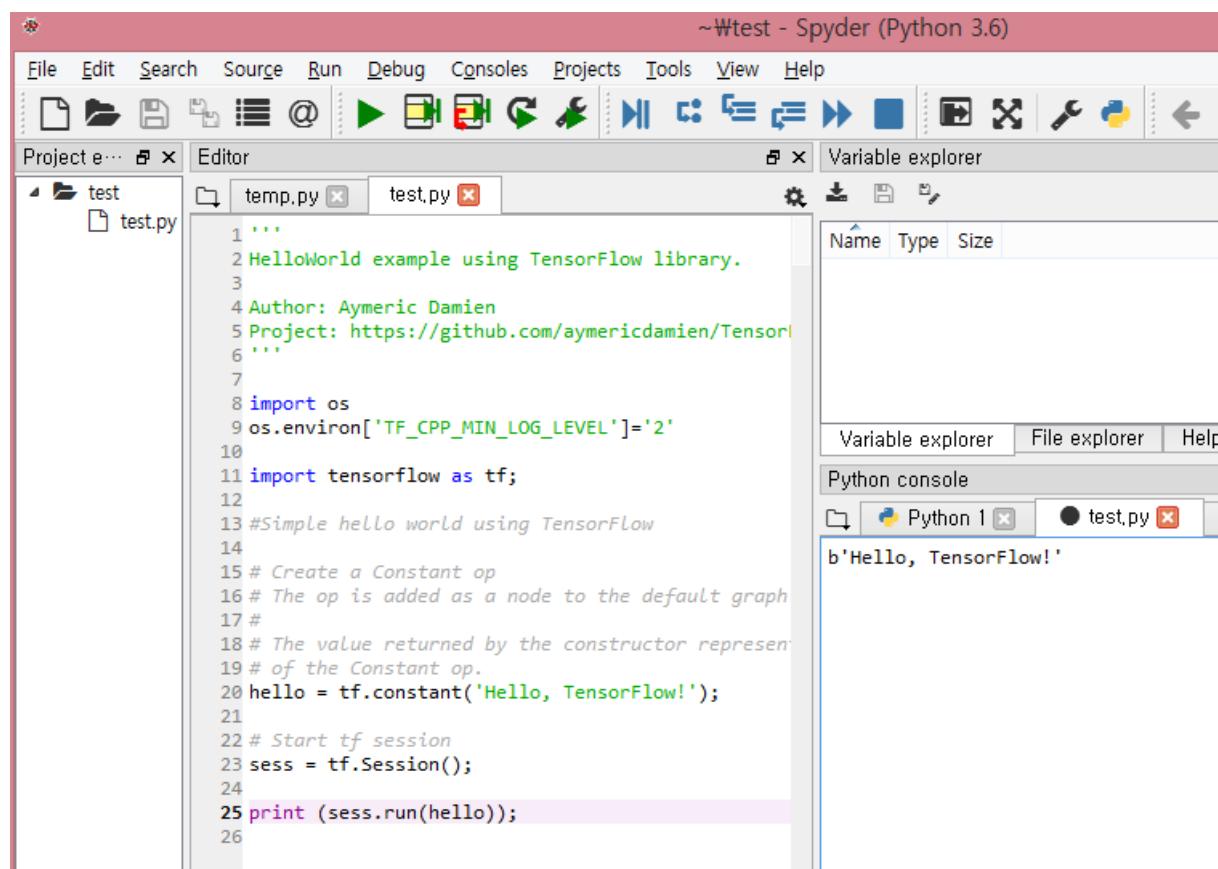
```
#Simple hello world using TensorFlow

# Create a Constant op
# The op is added as a node to the default graph.
#
# The value returned by the constructor represents the output
# of the Constant op.
hello = tf.constant('Hello, TensorFlow!');

# Start tf session
sess = tf.Session();

print (sess.run(hello));
```

Spyder 콘솔창에 "Hello, TensorFlow!"가 출력되면 개발환경 설치후 동작이 정상적으로 된다는 것이다.



1.5 TensorFlow 기본

아래는 파이썬에서 TensorFlow 라이브러리를 기본적으로 테스트 하는 코드를 연습하는 것이다. 파이썬에 TensorFlow 를 임포트하여 사용하는 방법을 간단히 예제 소스와 함께 정리한 것이다. 파이썬에 대해서는 TensorFlow 와 인터페이스하는 부분만 알아도 머신러닝 알고리즘을 충분히 테스트할 수 있다. 파이썬에 대한 좀더 깊이 있는 내용은 이책의 범위를 벗어나는 것으로 다른 참고서적이나 인터넷 검색정보를 활용하면 좋을듯 하다.

연산자 기본 예제

```
#Tensor Flow Basic

import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'

import tensorflow as tf

sess = tf.Session()

a = tf.constant(2)
b = tf.constant(6)

#c = 2 + 6 #error
c = a + b

print ("a + b: %i" % sess.run(c))
print ("a * b: %i" % sess.run(a * b))
```

실행 결과

```
a + b: 8
a * b: 12
```

TensorFlow 는 Placeholder 라는 데이터 저장소를 자주 사용한다. 아래 예제를 통하여 이것의 사용법을 익혀 두도록 하자.

Placeholder 기본 예제

```
#Tensor Flow Basic

import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
```

```
import tensorflow as tf

sess = tf.Session()

a = tf.placeholder(tf.int16)
b = tf.placeholder(tf.int16)

#define operation
add = tf.add(a, b)
mul = tf.mul(a, b)

#with tf.Session() as sess: <tab>
print ("add: %i" % sess.run(add, feed_dict={a:2, b:6}))
print ("mul: %i" % sess.run(mul, feed_dict={a:2, b:6}))
```

실행 결과

```
add: 8
mul: 12
```

머신러닝/딥러닝 TensorFlow 실습

Third Edition

2. 머신러닝 알고리즘

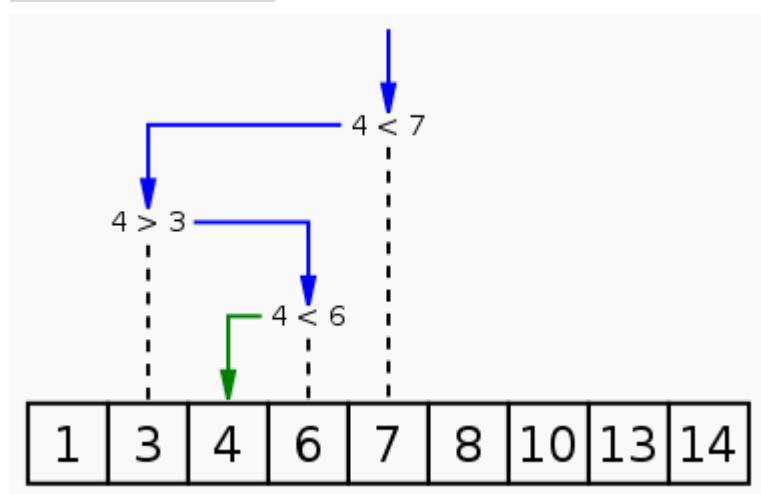
2. 머신러닝 알고리즘

2.1 알고리즘 소개

머신러닝은 약한 인공지능(특정분야에 맞추어진 인공지능)에서 강한 인공지능(인간과 유사한 판단 가능), 고전적 머신러닝(규칙을 판단하는 논리 기계)에서 현대적 Deep Learning 방식으로 발전하고 있다.

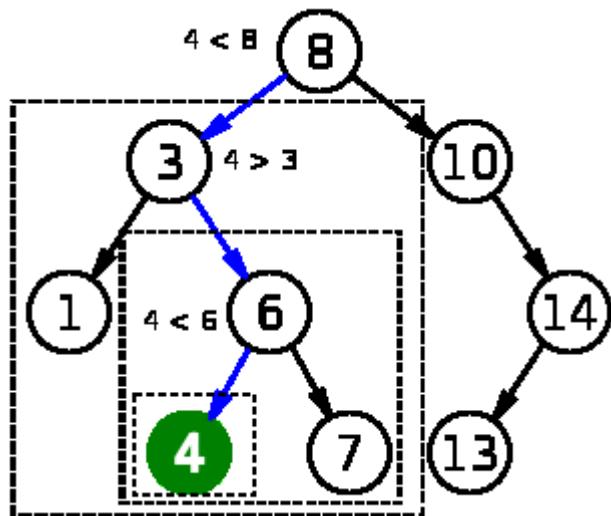
고전적 머신러닝은 정보들을 트리구조로 분류하여 탐색해 나가는 방식으로 정해진 답이 있는 전문가 시스템(expert system)에 적용 가능하다.

트리탐색 알고리즘



그림출처: <https://www.kullabs.com/classes/subjects/units/lessons/notes/note-detail/3861>

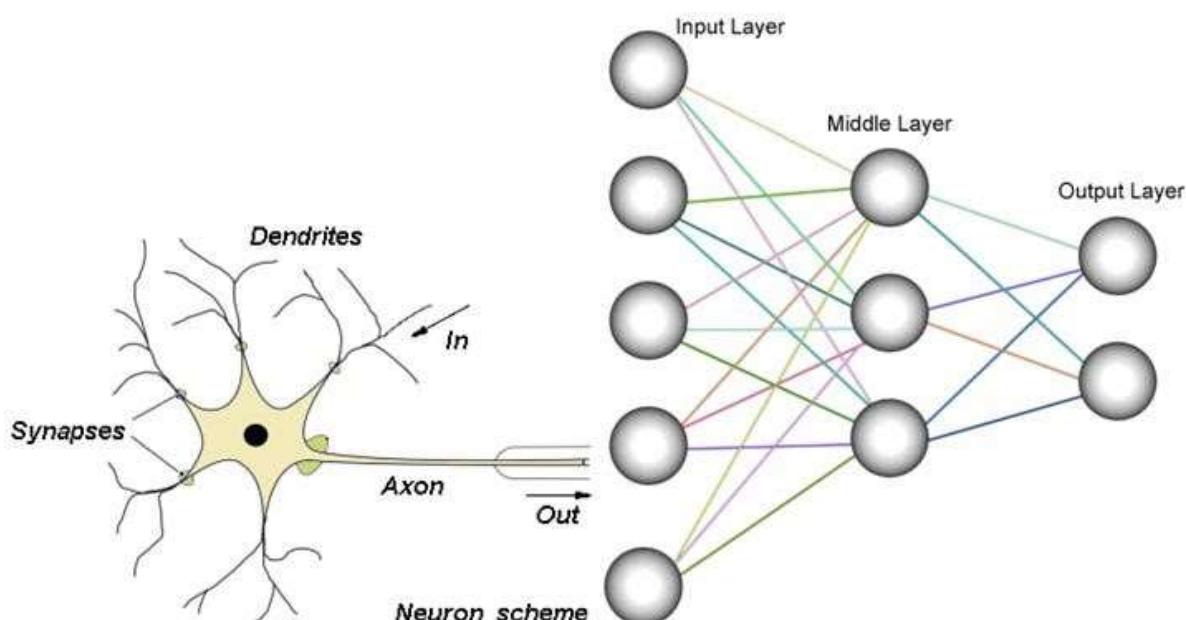
Binary Search Tree 탐색 알고리즘



그림출처: <https://www.kullabs.com/classes/subjects/units/lessons/notes/note-detail/3861>

그러나 애매한 조건들 속에서 이루어지는 종합적 판단을 하기 위해서는 단순한 트리구조 탐색으로는 부족하다. 그래서 현대의 머신러닝 알고리즘은 고전적인 알고리즘에서 벗어나서, 마치 인간이 학습해 나가는 방식과 유사하게 경험 데이터를 축적시키고 경험 데이터의 오류를 수정(보정)해 나가는 방향으로 발전하고 있다.

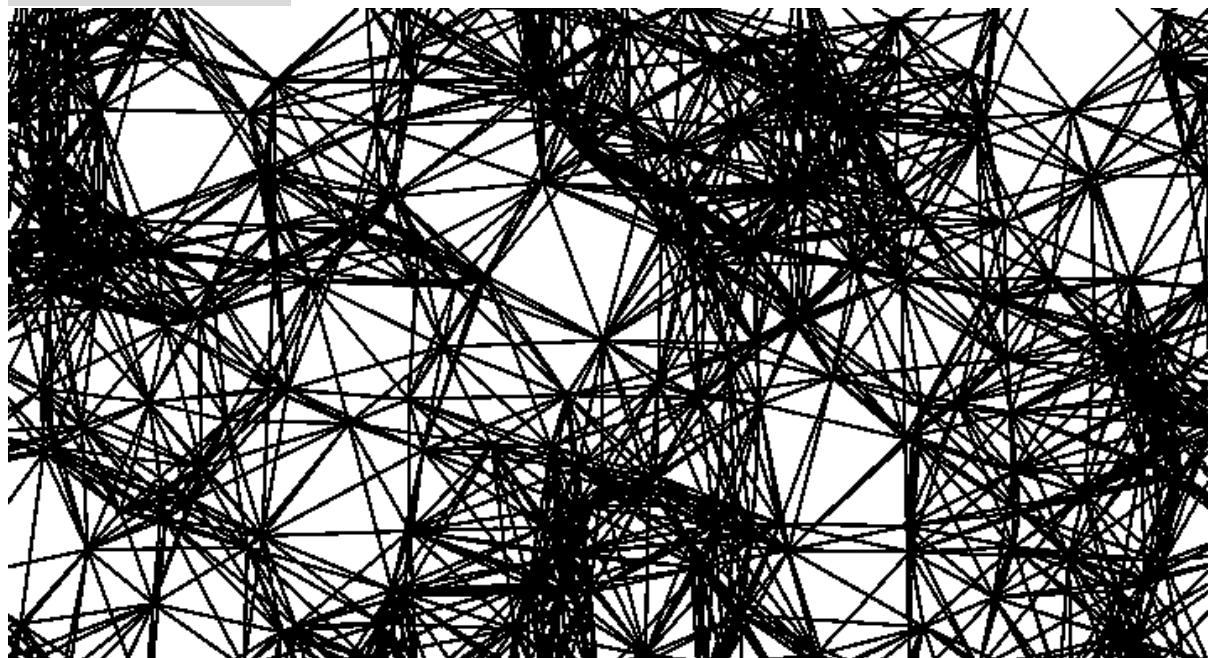
경험 데이터를 컴퓨터에서 자료구조로 표현



그림출처: <https://sites.google.com/site/mrstevensonstechclassroom/hl-topics-only/4a-robotics-ai/neural-networks-computational-intelligence>

위의 자료구조를 계속해서 연결해 나가면 다음과 같이 머신러닝에서 학습한 경험 데이터들을 거대하게 축적해 나갈 수 있다. 하지만 머신러닝은 학습데이터와 많은 메모리가 필요하다.

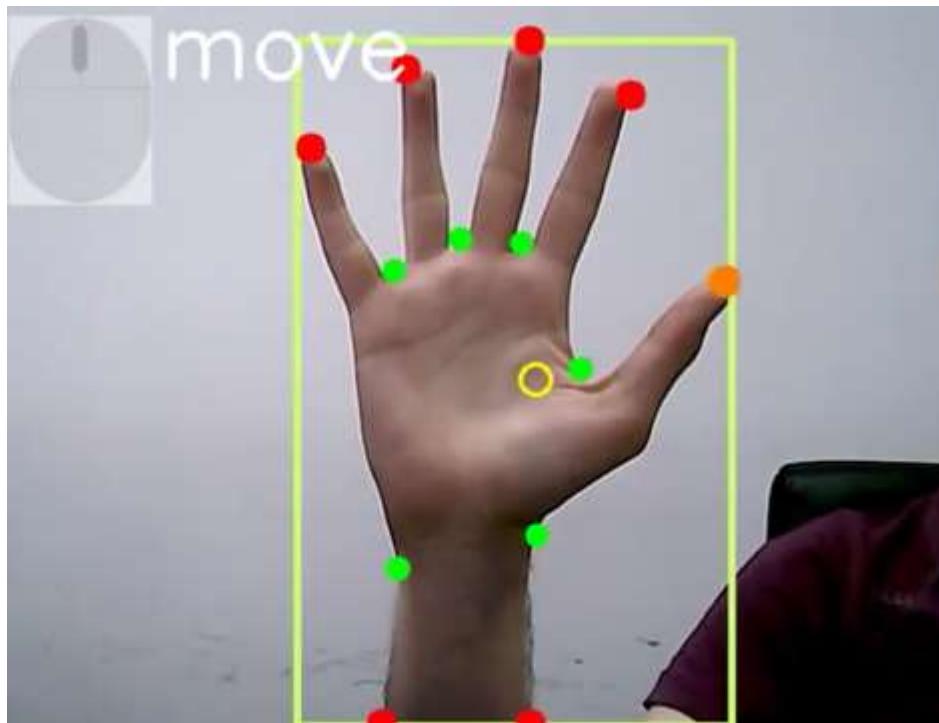
Neural Network 예



그림출처: http://jhnet.co.uk/projects/figures/neural_network

인간은 어떤 대상을 학습할 때, 특히 눈으로 보는 시각적인 대상은 많은 경험적 패턴을 통해서 판단을 내리듯이 오늘날의 머신러닝도 이러한 방식으로 접근하고 있다.

아래 그림은 인간의 손모양 패턴(특징값)을 학습해 가는 영상을 유튜브에서 캡쳐한 것이다.



출처: 유튜브: <https://www.youtube.com/watch?v=kQxiFaZbOfA>

머신러닝 알고리즘은 다음과 같이 크게 2 가지 종류로 구분할 수 있다.

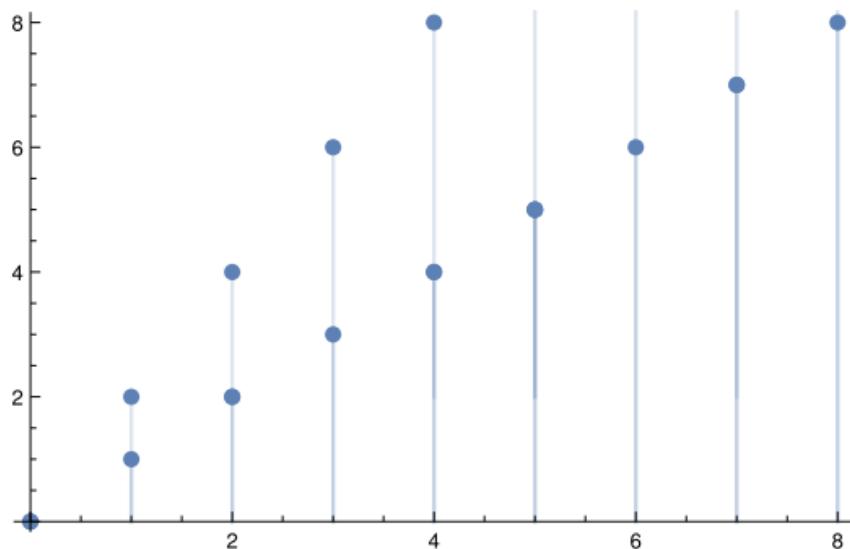
머신러닝 알고리즘 분류	요약 설명
Supervised Learning	지도학습, 정답을 미리 알려준 내용을 학습하는 방식
Unsupervised Learning	비지도학습, 정답을 미리 알려주지 않은 상태에서 그동안의 학습내용을 분류하여 스스로 정답을 찾아가는 방식

이번장은 Supervised Learning에 대해서 기술하고 Unsupervised Learning은 다음장부터 설명하는 Deep Learning에서 설명할 예정이다.

2.2 Linear Regression

Linear Regression은 선형적으로 변화하는 데이터를 반복적으로 학습한다는 것으로 해석할 수 있다. 선형적인 데이터는 우리가 직관적으로 가장 쉽게 이해할 수 있는 데이터이다. 머신러닝에서도 이것을 기본적인 학습 모델로하여 알고리즘을 전개하므로 이것부터 이해하도록 하자. 아래 그림에서 세로축(y)과 가로축(x)에 놓여 있는 점들은 선형적인(Linear) 규칙이 있다. 점들을 하나씩 따라 가보면 일정한 크기(기울기)로 증가하는 패턴임을 알 수 있다. 우리들은 직관적으로 점들에서 이러한 규칙성을 발견할 수 있으나, 이런 배경지식이 전혀 없는 기계(머신)에게는 이것을 계산적인 절차 즉 알고리즘으로 규칙성을 찾아가며 학습하는 방법을 알려 주어야 한다. 기계가 계산적인 절차(알고리즘)를 통하여 규칙성을 찾아내면 이것을 통하여 학습할 수 있으며 이것이 머신러닝의 기초가 된다.

2.2.1 가설함수(학습 모델)와 비용함수



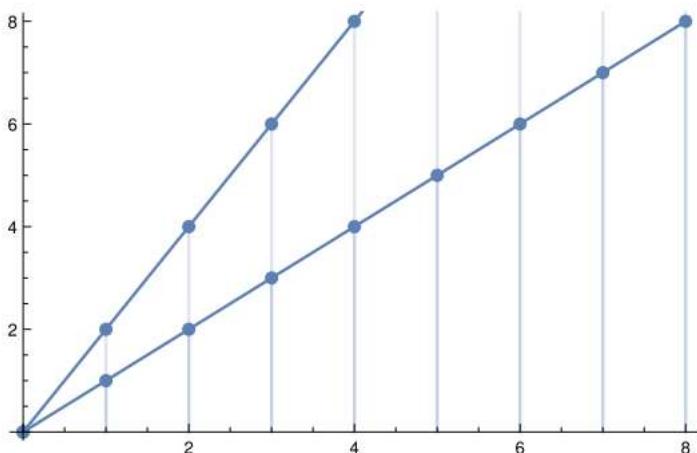
위의 데이터 분포 그래프를 수학식으로 일반화하면 아래와 같이 표현할 수 있다.

 $y = x$
 $y = 2x$
 $y = wx$

우리는 이미 중학교 시절 일차함수에 대해서 배웠기 때문에 위와 같이 선형적으로 변화하는 데이터에 대해서는 일차 함수식으로 표현하고, 데이터 변화의 규칙성을 함수의 기울기로 파악할 수 있다.

위의 일차 함수식에서 기울기를 w 로 표현 했다는 것을 기억해 두자. 우리는 일차함수에 대해서 이미 교육을 받아 알고 있기 때문에 위의 데이터를 보면 금방 아래와 같이 데이터에 대한 규칙성을 기울기 w 로 표현하는 일차 함수식을 세울 수 있다. 기울기를 w 로 표현하는 이유는 머신러닝에서 w 를 가중치(weight)로 표현하기 때문에 이 기호를 앞으로 계속 사용할 것이다.

$y = wx$ 그래프 (w 는 기울기)



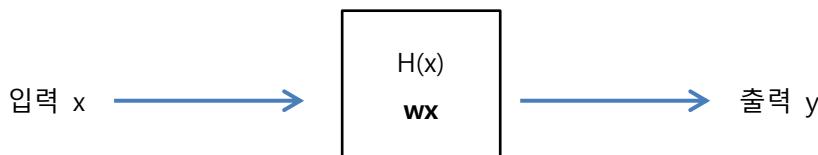
그런데 이런 배경지식이 전혀 없는 머신에게 위의 데이터를 학습 시킬 때, 먼저 학습 대상이 되는 모델에 대해서 알려 주고 시작해야 한다. 머신러닝에서는 이러한 학습 모델을 아래와 같이 가설함수로 알려준다. 머신러닝에서는 가설함수를 Hypothesis 라는 용어를 사용하고 $H(x)$ 라고 표현한다.

가설함수(Hypothesis)

$$y = wx$$

$$H(x) = wx$$

앞으로 우리의 머신은 위의 가설함수를 사용하여 학습을 전개해 나간다. 가설함수 $H(x)$ 에서 x 는 가설함수에 입력하는 입력 데이터가 되고, y 는 $H(x)$ 의 출력 결과가 된다.



우리는 지금까지 입력 데이터 x 에 대해서 출력된 y 를 보고 w 을 찾아가는 방식으로 학습을 했다. 이것이 선뜻 다가오지 않는 독자분들은 앞의 내용에서 기울기(w)를 어떤 방식으로 학습 했는지 다시 한번 확인해 보기 바란다. 우리의 머신도 이와 마찬가지로 “**입력 데이터 x 에 대해서 출력된 y 를 보고 w 을 찾아가는 방식으로 학습**”을 전개한다. 결국 머신러닝도 우리 사람이 학습하는 방식과 매우 유사한 방식으로 학습하는 알고리즘이다.

우리 사람은 이미 “**입력 데이터 x 에 대해서 출력된 y 를 보고 w 을 찾아가는 방식으로 학습**”하는 방법을 알고 있다. 우리 사람의 두뇌 세포는 과거 수백~수천만년의 진화 과정을 거쳐서 이러한 알고리즘을 유전적 물질로 전달받아 내재화 했다는 것이다. 앞으로 우리는 우리의 머신에게 이렇게 하는 방법을 알려 주고, 우리 사람의 삶을 널리 이롭게 하는 방향으로 발전시켜야 한다는 사명감으로 머신러닝을 배워야 할 것이다.

결국 우리의 머신은 입력 데이터 x 와 출력 데이터 y 에 의존해서 w 을 학습하기 때문에 사람과 같이 판단하기 위해서는 수많은 x 와 y 를 학습해야 한다는 난제들이 있지만, 이것을 학습하는 알고리즘은 이미 많은 학자들에 의해서 정립되고 있다. 일단, 우리는 이것을 이해하도록 하자.

그럼, 우리의 머신이 w 을 어떻게 학습하는지 알아보자.

우리의 머신에게 학습모델(가설함수) $y = wx$ 에 대해서 입력 데이터 x 가 아래와 같이 주어지고, 출력 y 가 아래와 같다고 알려주면 w 을 처음에 모르는 상태에서 이것을 어떻게 학습하는지 알아보자.

x	w	$y = wx$
1		2
2		4
3		6
4		8

처음에 우리의 머신은 w 가 어떤 값인지 모르기 때문에 임시 초기값으로 $w=5$ 라고 놓고 학습을 전개한다고 하자. (계산 편의상 소수이하는 반올림 했다)

x 는 1 일때 w 는 처음에 5이고 가설함수(학습모델)에서 산출하는 값은 x 와 w 을 곱하는 값이므로 $y = wx = 5$ 가 된다. 이때 우리는 머신에게 $y = 5$ 가 아니라 $y = 2$ 가 되어야 한다고 알려준다. 우리 머신은 사람과 마찬가지로 교육(지도)을 통해서 학습 한다. 우리 머신은 교육받은 내용을 바탕으로 처음에 계산한 5 와 지도 받은 2 사이의 차이값이 얼마인지 다음과 같이 계산한다.

차이값(오류) = 머신이 계산한값 – 교육받은 값 = $wx - y = 5 - 2 = 3$

우리의 머신은 위의 차이값을 사용하여 w 을 다음과 같이 보정(수정)한다.

$$w = w - (\text{차이값}/\text{보정율}) = 5 - (3 / 2) = 4$$

위에서 보정율을 2로 했다. 이것을 학습비율(Learning Rate)이라고 하는데, 이 수치에 대해서는 다음에 좀더 자세히 설명한다. 일단, 오차를 w 에 보정(수정)하면서 w 가 어떻게 학습되는지 이해하도록 하자.

우리 머신은 한번 학습을 진행하여 w 가 5보다는 4가 더 좋다는 것을 교육을 통해서 보정하고 w 는 4가 된다. 이렇게 학습한 w 을 가지고 다음 입력 데이터 $x=2$ 에 대해서 다음과 같이 또 학습한다.

$$y = wx = 8$$

이때 우리는 머신에게 $y = 4$ 가 되어야 한다고 교육한다. 그럼 우리 머신은 다음과 같이 차이값을 계산한다.

$$\text{차이값(오류)} = \text{머신이 계산한값} - \text{교육받은 값} = wx - y = 8 - 4 = 4$$

그리고 다음과 같이 w 을 다시 보정(수정)한다.

$$w = w - (\text{차이값}/\text{보정율}) = 4 - (4 / 2) = 2$$

우리 머신은 두번 학습을 진행하여 w 가 4보다는 2가 더 좋다는 것으로 오차를 보정하고 w 는 2가 되었다. 그 다음 입력 데이터 $x=3$ 에 대해서 다음과 같이 또 학습한다.

$$y = wx = 6$$

이때 우리는 머신에게 $y = 6$ 이 되어야 한다고 교육한다. 그럼 우리 머신은 다음과 같이 차이값을 계산한다.

$$\text{차이값(오류)} = \text{머신이 계산한값} - \text{교육받은 값} = wx - y = 6 - 6 = 0$$

그리고 다음과 같이 w 을 보정(수정)한다.

$$w = w - (\text{차이값}/\text{보정율}) = 2 - (0 / 2) = 2$$

이제 우리 머신은 w 을 2로 학습 했다. 그 다음 입력 데이터 $x=4$ 에 대해서 다음과 같이 또 학습한다.

$$y = wx = 8$$

이때 우리는 머신에게 $y = 8$ 이 되어야 한다고 교육한다. 그럼 우리 머신은 다음과 같이 차이값을 계산한다.

$$\text{차이값(오류)} = \text{머신이 계산한값} - \text{교육받은 값} = wx - y = 8 - 8 = 0$$

그리고 다음과 같이 w 을 보정(수정)한다.

$$w = w - (\text{차이값}/\text{보정율}) = 2 - (0 / 2) = 2$$

우리 머신은 w 을 2로 학습했다. 지금까지 내용을 보면 우리 머신은 w 을 차이값(오류)으로 보정하면서 학습을 전개하다가 차이값(오류)이 0이 되면 w 는 더 이상 보정(수정)되지 않는다는 것을 알 수 있다.

아래 표는 지금까지 내용을 요약 정리한 것이다.

x : 입력 데이터

w : 학습 데이터 (초기값을 5로 하여 학습 시작)

wx : 가설함수(학습모델을 통해 머신이 계산한 값)

y : 교육(지도) 데이터

c : 차이값($wx - y$)

r : 학습율(Learning Rate: 2로 설정)

e : 보정값(c / r)

u : 수정값($w - e$)

x	w	wx	y	c	r	e	u
1	5	5	2	3	2	1	4
2	4	8	4	4	2	2	2
3	2	6	6	0	2	0	2

4	2	8	8	0	2	0	2
---	---	---	---	---	---	---	---

위에서는 우리 머신이 학습을 시작할 때 w 초기값을 5로하여 시작했다. 그럼 이번에는 우리 머신이 w 초기값을 -5로하여 학습을 전개하면 다음과 같다. 학습 과정은 위에서 설명했으므로 테이블로만 정리한다.

x: 입력 데이터

w: 학습 데이터 (초기값을 -5로 하여 학습 시작)

wx : 가설함수(학습모델을 통해 머신이 계산한 값)

y: 교육(지도) 데이터

c: 차이값($wx - y$)

r: 학습률(Learning Rate: 2로 설정)

e: 보정값(c / r)

u: 수정값($w - e$)

x	w	wx	y	c	r	e	u
1	-5	-5	2	-7	2	-3	-2
2	-2	-4	4	-8	2	-4	2
3	2	6	6	0	2	0	2
4	2	8	8	0	2	0	2

결론적으로 우리 머신은 차이값 $c(wx-y)$ 가 0이 되면 w 를 더 이상 보정하지 않으므로 사실상 이 지점까지 학습한 w 에서 학습이 마무리 된다. 따라서 머신러닝의 기본 알고리즘을 다음과 같이 정리할 수 있다.

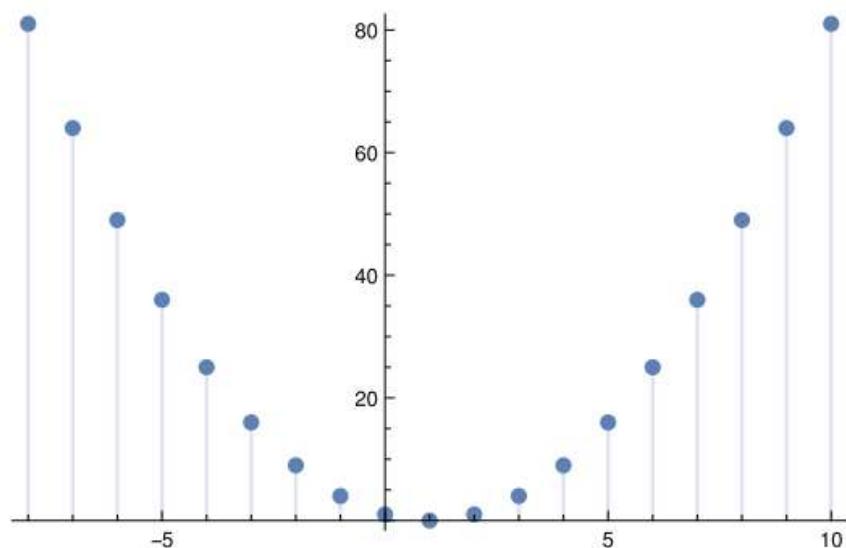
“계산한 차이값 ($wx-y$)으로 w 을 계속 보정하다가 차이값이 0이 되면 학습을 마무리 한다.”

머신러닝 알고리즘에서 위의 차이값을 비용(cost)이라고 하고 이것을 비용함수로 다음과 같이 정의 한다. 차이값이 크면 클수록 비용도 증가한다. 차이값은 양수 혹은 음수로 나타나는데 비용은 차이값의 절대치에 비례한다. 따라서 비용은 다음과 같이 차이값에 대해서 제곱을 하여 계산한다.

비용(Cost) 함수

$$\text{cost}(w) = (wx - y)^2$$

비용함수 그래프



그래프의 세로축은 $\text{cost}(w)$

그래프의 가로축은 w

비용함수를 사용하여 입력 데이터수 m 개에 대한 평균 비용을 계산하는 것을 수식으로 일반화하면 다음과 같다.

평균 비용

$$\text{cost}(W, b) = \frac{1}{m} \sum_{i=1}^m (H(x^{(i)}) - y^{(i)})^2 \quad \text{when } H(x) = Wx + b$$

Hypothesis and Cost	Simplified hypothesis
$H(x) = Wx + b$ $cost(W, b) = \frac{1}{m} \sum_{i=1}^m (H(x^{(i)}) - y^{(i)})^2$	$H(x) = Wx$ $cost(W) = \frac{1}{m} \sum_{i=1}^m (Wx^{(i)} - y^{(i)})^2$

위의 내용들은 머신러닝 알고리즘의 기본이며 앞으로 이것을 바탕으로 머신러닝이 전개 되므로 앞의 내용을 잘 숙지하기 바란다.

2.2.2 비용 줄이기(기울기 예측)

위의 비용함수를 사용하여 비용을 줄여 나가는 방식으로 반복연산을 수행하면 결국에는 최소 비용에 도달하게 된다. 이때 최소 비용에 해당하는 지점에서 우리가 찾고자하는 w 가 산출 된다. 아래의 파이썬 예제를 통하여 비용이 줄어드는 과정을 이해해 보자.

비용 줄이기 예제1

```
#Linear Regression(W=2)

import tensorflow as tf

X = [1., 2., 3.]
Y = [2., 4., 6.]
m = n_samples = len(X)

W = tf.placeholder(tf.float32)

hypothesis = tf.mul(X, W)

cost = tf.reduce_sum(tf.pow(hypothesis - Y, 2))/(m)

init = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init)

for i in range(-20, 30):
    print i, i*0.1, sess.run(cost, feed_dict={W: i*0.1})
```

실행결과

```
-20 -2.0 74.6667  
-19 -1.9 70.98  
-18 -1.8 67.3867  
-17 -1.7 63.8867  
-16 -1.6 60.48  
-15 -1.5 57.1667  
-14 -1.4 53.9467  
-13 -1.3 50.82  
-12 -1.2 47.7867  
-11 -1.1 44.8467  
-10 -1.0 42.0  
-9 -0.9 39.2467  
-8 -0.8 36.5867  
-7 -0.7 34.02  
-6 -0.6 31.5467  
-5 -0.5 29.1667  
-4 -0.4 26.88  
-3 -0.3 24.6867  
-2 -0.2 22.5867  
-1 -0.1 20.58  
0 0.0 18.6667  
1 0.1 16.8467  
2 0.2 15.12  
3 0.3 13.4867  
4 0.4 11.9467  
5 0.5 10.5  
6 0.6 9.14667  
7 0.7 7.88667  
8 0.8 6.72  
9 0.9 5.64667  
10 1.0 4.66667  
11 1.1 3.78  
12 1.2 2.98667  
13 1.3 2.28667  
14 1.4 1.68  
15 1.5 1.16667  
16 1.6 0.746666  
17 1.7 0.42  
18 1.8 0.186667  
19 1.9 0.0466667
```

20 2.0 0.0 //w=2일 때 최소비용에 도달함

```
21 2.1 0.0466666  
22 2.2 0.186667  
23 2.3 0.42  
24 2.4 0.746667  
25 2.5 1.16667  
26 2.6 1.68  
27 2.7 2.28667  
28 2.8 2.98667  
29 2.9 3.78
```

2.2.3 미분 함수(Convex)

지금까지 머신러닝에 대해서 설명한 것을 다시 정리해 보면 다음과 같다.

우리 머신은 학습모델(가설함수)을 통해서 계산한 값과 교육(지도)하는 값 사이에서 비용을 산출하여 그 비용이 최소화 되는(0으로 수렴) 방향으로 학습을 전개한다. 비용이 크면 클수록 보정(수정)하는 값은 크고, 비용이 작으면 작을수록 보정하는 값은 작아진다.

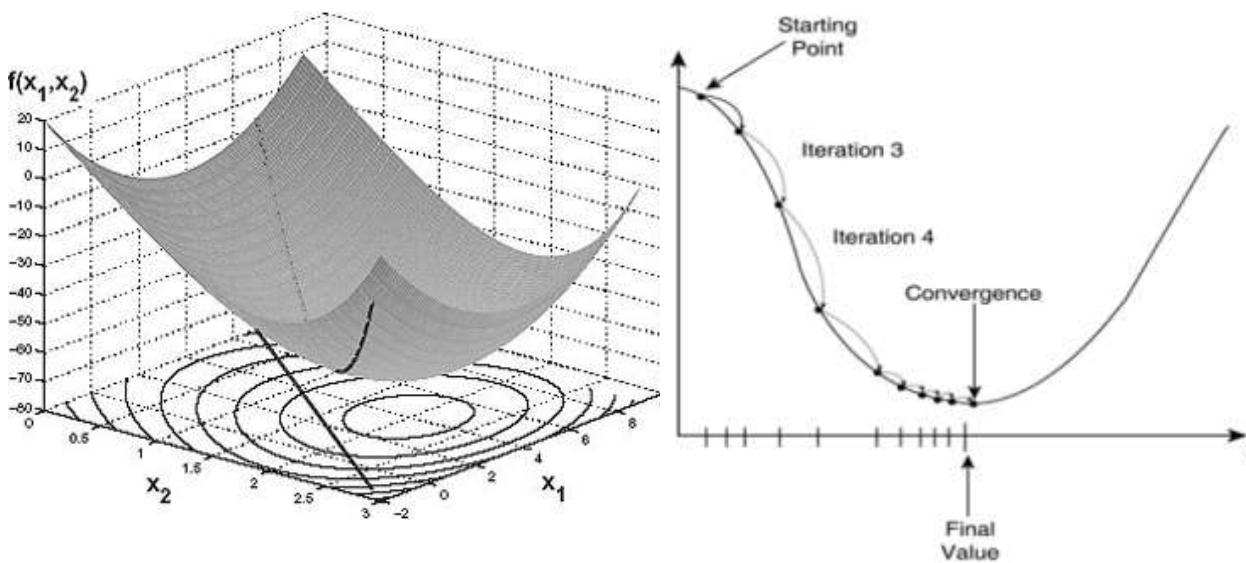
위에서 비용함수 그래프의 곡선을 보면 비용이 크면 클수록 그 지점에서 기울기가 가파르고 비용이 작으면 작을수록 기울기가 완만해진다는 것을 알 수 있다. 이를 바탕으로 우리는 보정하는 값이 비용함수의 기울기와 같다는 것을 판단할 수 있다. 우리는 고등학교 수학에서 기울기를 산출할 때 해당함수를 미분하면 된다는 것을 배웠다. 위에서 비용함수는 2 차함수 형태로 나타나고 이것을 미분하면 미분함수가 된다.

머신러닝에서는 비용함수 $\text{cost}(W)$ 를 통하여 미분함수를 다음과 같이 산출한다.

Formal definition	Formal definition
$\text{cost}(W) = \frac{1}{m} \sum_{i=1}^m (Wx^{(i)} - y^{(i)})^2$  $\text{cost}(W) = \frac{1}{2m} \sum_{i=1}^m (Wx^{(i)} - y^{(i)})^2$	$W := W - \alpha \frac{\partial}{\partial W} \frac{1}{2m} \sum_{i=1}^m (Wx^{(i)} - y^{(i)})^2$ $W := W - \alpha \frac{1}{2m} \sum_{i=1}^m 2(Wx^{(i)} - y^{(i)})x^{(i)}$ $W := W - \alpha \frac{1}{m} \sum_{i=1}^m (Wx^{(i)} - y^{(i)})x^{(i)}$

머신러닝에서는 위의 비용함수를 사용하여 비용이 최소화 되는 기울기 w 값을 찾아가도록 연산을 반복수행한다. 그러므로 비용함수는 아래와 같이 아래쪽으로 기울기값이 0으로 수렴하는 포물선 형태가 되어야 학습을 올바르게 전개할 수 있다. 따라서, 머신러닝에서 비용함수는 모두 이런 형태로 나타난다. 이러한 함수를 Convex Function 이라 한다. 아래는 Convex Function 을 그래프로 나타낸 것이다.

Convex Function



앞에서 예를 들어 설명한, 아래와 같은 학습데이터 모델에 대해서 지금까지 수학식으로 일반화한 내용들을 하나씩 적용해 보자.

x	w	y = wx
1		2
2		4
3		6
4		8

위의 데이터를 우리 머신에게 학습시킬 때 학습모델은 $y = wx$ 로 하고, 비용은 $(wx - y)^2$ 을 모두 더한 것에 대해서 $2m$ 으로 나누어 계산한다. 여기서 m 은 학습 데이터 개수이다. 위의 데이터는 학습 데이터가 4개 이므로 m 은 4가 된다.

$$cost(W) = \frac{1}{m} \sum_{i=1}^m (Wx^{(i)} - y^{(i)})^2$$

비용에 대해서 w 로 편미분을 하면 여러값이 되고 여기에 알파값(Learning Rate)을 곱한값을 가지고 w 를 다음과 같이 보정한다. (알파값은 w 가 발산되지 않도록 조정하는 값이다)

$$W := W - \alpha \frac{\partial}{\partial W} cost(W)$$

비용에 대한 수학수식을 w 로 편미분 하면 아래와 같다.

$$W := W - \alpha \frac{1}{m} \sum_{i=1}^m (Wx^{(i)} - y^{(i)})x^{(i)}$$

위의 내용을 학습데이터 모델에 대해서 적용하여 테이블로 정리하면 다음과 같다. w 는 초기값으로 5를 주었고, Learning Rate 인 알파값은 0.1로 했다. 계산의 편의상 소수 이하는 반올림 했다.

x	w	wx	y	wx-y	$(wx-y)^2$	$(wx-y)x$
1	5	5	2	3	9	3
2	5	10	4	6	36	12
3	5	15	6	9	81	27
4	5	20	8	12	144	48
				합산값	270	90
				비용값	34	23
보정: $w = 5 - 2 \leftarrow \dots \quad 2$					23 x 0.1	

위에서 $(wx-y)^2$ 으로 산출한 값들을 모두 더하면 270이 된다. 비용은 이 값을 $2m$ 으로 나누어야 한다. m 은 입력 데이터 개수 이므로 4가 된다. 따라서 비용값은 270을 8로 나누어 계산한 값인 34가 된다. 그리고 $(wx-y)x$ 는 비용을 미분한 것이고 이 값을 모두 합산하면 90이 된다. 이 값을 m 으로 나누면 23이 된다. 보정값은 23에 대해서 알파값 0.1을 곱하면 2가 되고 이것을 w 값에서 빼서 보정한다. 처음에 w 는 5이므로 5에서 2를 빼면 w 는 3이 된다.

이렇게 학습하여 산출된 $w=3$ 을 가지고 우리 머신은 위와 동일한 방법으로 또다시 학습한다.

x	w	wx	y	wx-y	$(wx-y)^2$	$(wx-y)x$
1	3	3	2	1	1	1
2	3	6	4	2	4	4
3	3	9	6	3	9	9
4	3	12	8	4	16	16
				합산값	30	30
				비용값	4	8
보정: $w = 3 - 1 \leftarrow \dots \quad 1$					8 x 0.1	

한번 더 학습한 결과 w 는 2가 되었다. 이 값을 가지고 또다시 학습하면,

x	w	wx	y	wx-y	$(wx-y)^2$	$(wx-y)x$
1	2	2	2	0	0	0
2	2	4	4	0	0	0
3	2	6	6	0	0	0
4	2	8	8	0	0	0
				합산값	0	0
				비용값	0	0
보정: $w = 2 - 0 \leftarrow \dots$					0 x 0.1	

w 를 2로 학습 했으므로 이때의 비용은 0이 되어 w 를 보정하는 값도 0이므로 w 는 변함없이 2가 된다. 즉, 비용이 0 될 때 w 값은 우리 머신이 학습한 최종값(정답)이 된다.

2.3 Linear Regression Learning

2.3.1 단항변수 기울기 학습1

이제 우리는 앞에서 확인한 비용 줄이기 함수를 사용하여 우리의 머신에게 최소 비용의 기울기를 찾아 가는 알고리즘을 적용 시킬 수 있다. 이렇게 계산적인 절차를 사용하여 최소비용을 찾아가는 진행을 머신러닝에서 학습이라고 한다. 학습대상이 단항변수인 경우에는 가중치(기울기)가 한 개로 수렴해 가는데 이것을 파이썬 예제를 통하여 확인해 보자.

단항변수 기울기 학습 예제1

```
#Linear Regression Learning(W=1)

import tensorflow as tf

x_data = [1., 2., 3.]
y_data = [1., 2., 3.]

W = tf.Variable(tf.random_uniform([1], -10.0, 10.0))

X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

hypothesis = W * X

#Cost Function
cost = tf.reduce_mean(tf.square(hypothesis - Y))

#Minimize(gradient descent algorithm)
descent = W - tf.mul(0.1, tf.reduce_mean(tf.mul((tf.mul(W, X) - Y), X)))
update = W.assign(descent)

init = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init)

print 'Learning'
for step in range(1, 32):
    sess.run(update, feed_dict={X:x_data, Y:y_data})
    print step, sess.run(cost, feed_dict={X:x_data, Y:y_data}), sess.run(W)

print 'Answer'
print ' 5.0', sess.run(hypothesis, feed_dict={X:5.0})
print '10.0', sess.run(hypothesis, feed_dict={X:10.0})
```

실행 결과

```
Learning
1 85.4904 [ 5.28011465]
2 24.3173 [ 3.28272772]
3 6.91692 [ 2.21745491]
4 1.96748 [ 1.64930928]
5 0.559638 [ 1.34629822]
6 0.159186 [ 1.18469238]
7 0.0452796 [ 1.09850264]
8 0.0128795 [ 1.0525347]
9 0.0036635 [ 1.02801847]
10 0.00104207 [ 1.01494324]
11 0.000296409 [ 1.00796974]
12 8.43126e-05 [ 1.00425053]
13 2.3984e-05 [ 1.0022267]
14 6.82141e-06 [ 1.00120902]
15 1.94042e-06 [ 1.0006448]
16 5.52056e-07 [ 1.00034392]
17 1.5703e-07 [ 1.00018346]
18 4.47239e-08 [ 1.00009787]
19 1.27226e-08 [ 1.00005221]
20 3.63127e-09 [ 1.00002789]
21 1.03977e-09 [ 1.0000149]
22 2.95799e-10 [ 1.00000799]
23 8.59472e-11 [ 1.00000429]
24 2.34053e-11 [ 1.00000226]
25 6.63173e-12 [ 1.00000119]
26 1.80478e-12 [ 1.0000006]
27 5.16328e-13 [ 1.00000036]
28 2.65269e-13 [ 1.00000024]
29 9.9476e-14 [ 1.00000012]
30 0.0 [ 1.]
31 0.0 [ 1.]
```

Answer

```
5.0 [ 5.]
10.0 [ 10.]
```

위의 실행결과를 확인해 보면, 연산을 반복수행함에 따라서 비용은 0으로 수렴해 가고, 기울기(가중치 w: weight)는 1에 수렴해가는 것을 알 수 있다. 비용이 0으로 수렴하여 w가 1이 된다는 것을 학습한 것이 되므로 어떤 변수값에 대해서 질문을 하면, 우리의 머신은 w을 학습하여 알고 있기 때문에 이것을 가설함수에 대입하여 대답을 할 수 있게 된다.

2.3.2 단항변수 기울기 학습2

이번에는 학습 대상이 다음과 같이 주어 졌을 때, 기울기를 학습해 가는 과정을 실습해 보자.
가설함수에 b (바이어스)가 추가 되었다.

학습대상(가설, Hypothesis)

```
x_data = [1.0, 2.0, 3.0]
y_data = [1.0, 2.0, 3.0]

hypothesis = W * X + b
```

단항변수 기울기 학습 예제2

```
#Linear Regression Learning(W=1, b)

import tensorflow as tf

x_data = [1.0, 2.0, 3.0]
y_data = [1.0, 2.0, 3.0]

W = tf.Variable(tf.random_uniform([1], -10.0, 10.0))
b = tf.Variable(tf.random_uniform([1], -10.0, 10.0))

X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

hypothesis = W * X + b

#Cost Function
cost = tf.reduce_mean(tf.square(hypothesis - Y))

#Minimize(gradient descent algorithm)
descent = W - tf.mul(0.1, tf.reduce_mean(tf.mul((tf.mul(W, X) - Y), X)))
update = W.assign(descent)
a = tf.Variable(0.1)      #Learning rate, alpha
optimizer = tf.train.GradientDescentOptimizer(a)
train = optimizer.minimize(cost)

init = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init)

print 'Learning'
for step in range(1, 801):
    #sess.run(update, feed_dict={X:x_data, Y:y_data})
    sess.run(train, feed_dict={X:x_data, Y:y_data})
```

```

if step % 20 == 0:
    print step, sess.run(cost, feed_dict={X:x_data, Y:y_data}), sess.run(W), sess.run(b)

print 'Answer'
print ' 5.0', sess.run(hypothesis, feed_dict={X:5.0})
print '10.0', sess.run(hypothesis, feed_dict={X:10.0})

```

실행 결과

Learning

```

20 0.87753 [-0.0879951] [ 2.47326851]
40 0.331554 [ 0.33123532] [ 1.52025938]
60 0.12527 [ 0.58892614] [ 0.93446749]
80 0.0473305 [ 0.74732262] [ 0.57439506]
100 0.0178827 [ 0.84468526] [ 0.35306704]
120 0.00675658 [ 0.90453172] [ 0.21702197]
140 0.00255282 [ 0.94131792] [ 0.13339826]
160 0.000964524 [ 0.96392947] [ 0.08199676]
180 0.000364424 [ 0.97782832] [ 0.05040149]
200 0.000137689 [ 0.98637158] [ 0.03098061]
220 5.20231e-05 [ 0.99162292] [ 0.01904304]
240 1.96557e-05 [ 0.99485081] [ 0.01170526]
260 7.42605e-06 [ 0.99683499] [ 0.00719494]
280 2.80575e-06 [ 0.99805456] [ 0.00442255]
300 1.06001e-06 [ 0.99880415] [ 0.00271841]
320 4.00515e-07 [ 0.99926496] [ 0.00167097]
340 1.51337e-07 [ 0.9995482] [ 0.00102709]
360 5.71892e-08 [ 0.9997223] [ 0.00063134]
380 2.15914e-08 [ 0.99982935] [ 0.00038803]
400 8.16728e-09 [ 0.99989504] [ 0.0002385]
420 3.08039e-09 [ 0.99993557] [ 0.00014661]
440 1.16453e-09 [ 0.99996042] [ 9.01645981e-05]
460 4.41299e-10 [ 0.99997568] [ 5.54031649e-05]
480 1.65111e-10 [ 0.99998504] [ 3.40646984e-05]
500 6.35746e-11 [ 0.99999082] [ 2.09596292e-05]
520 2.34053e-11 [ 0.9999944] [ 1.29169739e-05]
540 9.08074e-12 [ 0.99999648] [ 7.93402523e-06]
560 3.51008e-12 [ 0.99999779] [ 4.89021431e-06]
580 1.25056e-12 [ 0.99999863] [ 2.99081353e-06]
600 4.78432e-13 [ 0.99999917] [ 1.85435147e-06]
620 1.94215e-13 [ 0.99999946] [ 1.16293813e-06]
640 6.15804e-14 [ 0.99999964] [ 7.65574157e-07]
660 6.15804e-14 [ 0.9999997] [ 6.22523146e-07]
680 1.89478e-14 [ 0.99999982] [ 4.23840987e-07]
700 4.73695e-15 [ 0.99999994] [ 2.01316880e-07]
720 0.0 [ 1.] [ 5.82656696e-08]
740 0.0 [ 1.] [ 5.82656696e-08]
760 0.0 [ 1.] [ 5.82656696e-08]
780 0.0 [ 1.] [ 5.82656696e-08]
800 0.0 [ 1.] [ 5.82656696e-08]

```

Answer

```

5.0 [ 5.]
10.0 [ 10.]

```

실행결과를 확인해 보면, 연산을 반복함에 따라서 비용이 0으로 수렴해 가고 기울기는 1에 수렴하고, 하나더 추가된 바이어스인 b 는 0으로 수렴하고 있음을 알 수 있다. 이때도 비용이 최소화 되는 기울기를 학습했으므로 질문에 대답을 할 수 있게 된다.

2.3.3 단항변수 기울기 학습3

이번에는 학습 데이터와 학습 대상이 다음과 같이 주어졌을 때, 기울기를 학습해 가는 과정을 실습해 보자.

학습대상(가설, Hypothesis)

```
x_data = [1., 2., 3.]  
y_data = [2., 4., 6.]
```

```
hypothesis = W * X
```

단항변수 기울기 학습 예제3

```
#Linear Regression Learning(W=2)  
  
import tensorflow as tf  
  
x_data = [1., 2., 3.]  
y_data = [2., 4., 6.]  
  
W = tf.Variable(tf.random_uniform([1], -10.0, 10.0))  
  
X = tf.placeholder(tf.float32)  
Y = tf.placeholder(tf.float32)  
  
hypothesis = W * X  
  
#Cost Function  
cost = tf.reduce_mean(tf.square(hypothesis - Y))  
  
#Minimize(gradient descent algorithm)  
descent = W - tf.mul(0.1, tf.reduce_mean(tf.mul((tf.mul(W, X) - Y), X)))  
update = W.assign(descent)  
  
init = tf.initialize_all_variables()  
  
sess = tf.Session()  
sess.run(init)
```

```
print 'Learning'
for step in range(1, 32):
    sess.run(update, feed_dict={X:x_data, Y:y_data})
    print step, sess.run(cost, feed_dict={X:x_data, Y:y_data}), sess.run(W)

print 'Answer'
print ' 5.0', sess.run(hypothesis, feed_dict={X:5.0})
print '10.0', sess.run(hypothesis, feed_dict={X:10.0})
```

실행 결과

```
Learning
1 4.4908 [ 2.98097634]
2 1.27738 [ 2.5231874]
3 0.363344 [ 2.27903318]
4 0.103351 [ 2.14881778]
5 0.0293978 [ 2.07936954]
6 0.00836205 [ 2.0423305]
7 0.00237856 [ 2.02257633]
8 0.000676557 [ 2.01204062]
9 0.000192437 [ 2.00642157]
10 5.47408e-05 [ 2.00342488]
11 1.55698e-05 [ 2.00182652]
12 4.42878e-06 [ 2.00097418]
13 1.25926e-06 [ 2.00051951]
14 3.58178e-07 [ 2.00027704]
15 1.0197e-07 [ 2.00014782]
16 2.90256e-08 [ 2.00007892]
17 8.33076e-09 [ 2.0000422]
18 2.34392e-09 [ 2.00002241]
19 6.63173e-10 [ 2.00001192]
20 1.90331e-10 [ 2.00000644]
21 5.19928e-11 [ 2.00000334]
22 1.22213e-11 [ 2.00000167]
23 4.24431e-12 [ 2.00000095]
24 1.06108e-12 [ 2.00000048]
25 3.97904e-13 [ 2.00000024]
26 0.0 [ 2.]
27 0.0 [ 2.]
28 0.0 [ 2.]
29 0.0 [ 2.]
30 0.0 [ 2.]
31 0.0 [ 2.]
```

Answer

```
5.0 [ 10.]
10.0 [ 20.]
```

이번에는 학습대상 데이터가 달라졌으나, 이것을 반복 연산하면서 비용이 최소화 되는 방향으로 학습한 결과, 비용은 0으로 수렴하고 기울기는 2에 수렴한다. 비용이 최소화되는 가중치를 학습 했으므로 이것도 역시 질문에 대답을 할 수 있게 된다.

2.3.4 다항변수 기울기 학습

학습대상 데이터가 변수 2개로 되어 있다면 기울기도 2개로 학습되어야 한다. 이번에는 이것에 대해서 실습해 보자.

학습대상(가설, Hypothesis)

```
x1_data = [1.0, 0.0, 3.0, 0.0, 5.0]
x2_data = [0.0, 2.0, 0.0, 4.0, 0.0]
y_data = [1.0, 2.0, 3.0, 4.0, 5.0]
```

```
hypothesis = W1*x1 + W2*x2 + b
```

다항변수 기울기 학습 예제

```
#Linear Regression Learning(Multi Variable)

import tensorflow as tf

x1_data = [1.0, 0.0, 3.0, 0.0, 5.0]
x2_data = [0.0, 2.0, 0.0, 4.0, 0.0]
#x1_data = [1.0, 2.0, 3.0, 4.0, 5.0]
#x2_data = [1.0, 2.0, 3.0, 4.0, 5.0]
y_data = [1.0, 2.0, 3.0, 4.0, 5.0]

W1 = tf.Variable(tf.random_uniform([1], -10.0, 10.0))
W2 = tf.Variable(tf.random_uniform([1], -10.0, 10.0))
b = tf.Variable(tf.random_uniform([1], -10.0, 10.0))

X1 = tf.placeholder(tf.float32)
X2 = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

hypothesis = W1*X1 + W2*X2 + b

#Cost Function
cost = tf.reduce_mean(tf.square(hypothesis - Y))

#Minimize(gradien descent algorithm)
#descent = W - tf.mul(0.1, tf.reduce_mean(tf.mul((tf.mul(W, X) - Y), X)))
```

```

#update = W.assign(descent)
a = tf.Variable(0.01)      #Learning rate, alpha
optimizer = tf.train.GradientDescentOptimizer(a)
train = optimizer.minimize(cost)

init = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init)

print 'Learning'
for step in range(1, 801):
    #sess.run(update, feed_dict={X:x_data, Y:y_data})
    sess.run(train, feed_dict={X1:x1_data, X2:x2_data, Y:y_data})
    if step % 20 == 0:
        print step, sess.run(cost, feed_dict={X1:x1_data, X2:x2_data, Y:y_data}), sess.run(W1),
        sess.run(W2), sess.run(b)

print 'Answer'
print ' 0.0, 6.0', sess.run(hypothesis, feed_dict={X1:0.0, X2:6.0})
print ' 7.0, 0.0', sess.run(hypothesis, feed_dict={X1:7.0, X2:0.0})
print ' 8.0, 8.0', sess.run(hypothesis, feed_dict={X1:8.0, X2:8.0})

```

실행결과

```

Learning
20 0.166593 [ 0.7456345] [ 0.69821054] [ 0.96759075]
40 0.0482358 [ 0.86313462] [ 0.83761036] [ 0.5206542]
60 0.0139663 [ 0.92635399] [ 0.91261953] [ 0.28015956]
80 0.00404383 [ 0.96037179] [ 0.95298129] [ 0.1507515]
100 0.00117086 [ 0.97867632] [ 0.97469962] [ 0.08111808]
120 0.000339016 [ 0.98852599] [ 0.98638612] [ 0.04364895]
140 9.81586e-05 [ 0.99382597] [ 0.99267447] [ 0.0234871]
160 2.8421e-05 [ 0.99667782] [ 0.99605823] [ 0.01263816]
180 8.22882e-06 [ 0.99821234] [ 0.99787897] [ 0.00680046]
200 2.38274e-06 [ 0.99903804] [ 0.99885869] [ 0.0036593]
220 6.8997e-07 [ 0.99948245] [ 0.99938583] [ 0.00196903]
240 1.99755e-07 [ 0.99972147] [ 0.99966955] [ 0.00105953]
260 5.78159e-08 [ 0.99985015] [ 0.99982214] [ 0.00057013]
280 1.67388e-08 [ 0.9999193] [ 0.99990433] [ 0.00030677]
300 4.85112e-09 [ 0.99995655] [ 0.9999485] [ 0.00016505]
320 1.40525e-09 [ 0.99997663] [ 0.99997228] [ 8.87643109e-05]
340 4.05146e-10 [ 0.99998748] [ 0.9999851] [ 4.77802823e-05]
360 1.16975e-10 [ 0.99999321] [ 0.99999195] [ 2.57075553e-05]
380 3.445e-11 [ 0.9999963] [ 0.99999571] [ 1.38057385e-05]
400 9.9277e-12 [ 0.99999809] [ 0.99999768] [ 7.47336253e-06]
420 2.87912e-12 [ 0.999999893] [ 0.999999875] [ 4.04014463e-06]
440 7.64544e-13 [ 0.9999994] [ 0.99999934] [ 2.16141143e-06]
460 2.50111e-13 [ 0.9999997] [ 0.99999964] [ 1.20297204e-06]
480 1.25056e-13 [ 0.99999982] [ 0.99999976] [ 7.07063009e-07]
500 2.27374e-14 [ 0.99999988] [ 0.99999988] [ 3.92351353e-07]
520 2.55795e-14 [ 0.99999988] [ 0.99999994] [ 2.44532060e-07]
540 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.82543744e-07]
560 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.63470759e-07]

```

```
580 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.63471185e-07]
600 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.63471611e-07]
620 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.63472038e-07]
640 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.63472464e-07]
660 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.63472890e-07]
680 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.63473317e-07]
700 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.63473743e-07]
720 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.63474169e-07]
740 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.63474596e-07]
760 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.63475022e-07]
780 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.63475448e-07]
800 4.83169e-14 [ 0.99999994] [ 0.99999994] [ 1.63475875e-07]

Answer
0.0, 6.0 [ 5.99999952]
7.0, 0.0 [ 6.99999952]
8.0, 8.0 [ 15.99999905]
```

실행결과를 확인해 보면, 비용은 0으로 수렴하고 기울기는 각각 1로 수렴해가며 학습하는 것을 알 수 있다.

2.3.5 다항변수 매트릭스 처리

학습대상 데이터가 많아지면 이것을 형렬 형태인 매트릭스로 처리할 수 있다. 이번에는 이것을 실습해 보자.

학습대상(가설, Hypothesis)

```
x_data = [[1.0, 0.0, 3.0, 0.0, 5.0],
           [0.0, 2.0, 0.0, 4.0, 0.0]]
y_data = [1.0, 2.0, 3.0, 4.0, 5.0]

hypothesis = tf.matmul(W, X) + b
```

위의 배열 데이터는 아래와 같이 Matrix 연산된다.

$$\begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} \end{bmatrix} = [y_1, y_2, y_3, y_4, y_5]$$

$$y_1 = w_1 x_{11} + w_2 x_{21} + b$$

$$y_2 = w_1 x_{12} + w_2 x_{22} + b$$

$$y_3 = w_1 x_{13} + w_2 x_{23} + b$$

$$y_4 = w_1 x_{14} + w_2 x_{24} + b$$

$$y_5 = w_1 x_{15} + w_2 x_{25} + b$$

파이썬 예제를 통하여 연산 과정을 확인해 보자.

단항변수 매트릭스 처리 예제

```
#Linear Regression Learning(Multi Variable Matrix + b)

import tensorflow as tf

x_data = [[1.0, 0.0, 3.0, 0.0, 5.0],
           [0.0, 2.0, 0.0, 4.0, 0.0]]
```

```

y_data = [1.0, 2.0, 3.0, 4.0, 5.0]

W = tf.Variable(tf.random_uniform([1,2], -10.0, 10.0))
b = tf.Variable(tf.random_uniform([1], -10.0, 10.0))

X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

hypothesis = tf.matmul(W, X) + b

#Cost Function
cost = tf.reduce_mean(tf.square(hypothesis - Y))

#Minimize(gradient descent algorithm)
descent = W - tf.mul(0.1, tf.reduce_mean(tf.mul((tf.mul(W, X) - Y), X)))
#update = W.assign(descent)
a = tf.Variable(0.1)      #Learning rate, alpha
optimizer = tf.train.GradientDescentOptimizer(a)
train = optimizer.minimize(cost)

init = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init)

print 'Learning'
for step in range(1, 801):
    #sess.run(train)
    sess.run(train, feed_dict={X:x_data, Y:y_data})
    if step % 20 == 0:
        print step, sess.run(cost, feed_dict={X:x_data, Y:y_data}), sess.run(W), sess.run(b)

print 'Answer'
print '[0.0, 6.0]', sess.run(hypothesis, feed_dict={X:[[0.0], [6.0]]})
print '[7.0, 0.0]', sess.run(hypothesis, feed_dict={X:[[7.0], [0.0]]})
print '[8.0, 8.0]', sess.run(hypothesis, feed_dict={X:[[8.0], [8.0]]})

```

실행 결과

```

Learning
20 0.507934 [[ 1.44412088  1.52695882]] [-1.6895442]
40 0.147068 [[ 1.23898375  1.28355253]] [-0.90912718]
60 0.0425825 [[ 1.12859511  1.15257716]] [-0.48919344]
80 0.0123294 [[ 1.06919587  1.08210039]] [-0.26323077]
100 0.00356989 [[ 1.03723371  1.04417753]] [-0.14164212]
120 0.00103364 [[ 1.02003515  1.02377152]] [-0.07621637]
140 0.000299282 [[ 1.01078081  1.01279128]] [-0.04101142]
160 8.66536e-05 [[ 1.00580108  1.00688279]] [-0.02206784]
180 2.50897e-05 [[ 1.0031215   1.00370359]] [-0.01187458]
200 7.26424e-06 [[ 1.00167966  1.00199294]] [-0.00638952]
220 2.1037e-06 [[ 1.00090384  1.00107229]] [-0.00343819]
240 6.09026e-07 [[ 1.00048637  1.00057709]] [-0.00185004]
260 1.76308e-07 [[ 1.00026166  1.00031042]] [-0.00099547]
280 5.10909e-08 [[ 1.00014091  1.00016713]] [-0.00053563]

```

```
300 1.4774e-08 [[ 1.0000757 1.00008988]] [-0.00028826]
320 4.29355e-09 [[ 1.00004077 1.0000484 ]] [-0.00015513]
340 1.24117e-09 [[ 1.00002193 1.00002611]] [-8.34778621e-05]
360 3.56574e-10 [[ 1.00001168 1.00001395]] [-4.48708670e-05]
380 1.02168e-10 [[ 1.00000632 1.00000751]] [-2.41428170e-05]
400 2.97895e-11 [[ 1.00000346 1.00000405]] [-1.29586369e-05]
420 8.57554e-12 [[ 1.00000179 1.00000215]] [-6.98150097e-06]
440 2.45919e-12 [[ 1.00000095 1.00000119]] [-3.73663329e-06]
460 7.56017e-13 [[ 1.00000048 1.0000006 ]] [-2.00095133e-06]
480 2.52953e-13 [[ 1.00000036 1.00000048]] [-1.19271522e-06]
500 6.89226e-14 [[ 1.00000024 1.00000024]] [-7.73100112e-07]
520 2.55795e-14 [[ 1.00000012 1.00000012]] [-3.48716100e-07]
540 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
560 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
580 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
600 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
620 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
640 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
660 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
680 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
700 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
720 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
740 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
760 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
780 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
800 2.06057e-14 [[ 1.00000012 1.00000012]] [-2.98648246e-07]
```

Answer

```
[0.0, 6.0] [[ 6.00000048]]
[7.0, 0.0] [[ 7.00000048]]
[8.0, 8.0] [[ 16.00000191]]
```

실행결과를 확인해 보면, 비용은 0으로 수렴하고 w1 과 w2 는 각각 1로 수렴해가며 학습하는 것을 알 수 있다.

2.3.6 다항변수 파일 읽기

학습데이터가 많아지면 이것을 모두 매트릭스에 담아두는 것이 불편하다. 이번에는 학습데이터를 파일에 저장하고 이것을 읽어서 학습하는 방법을 실습해 보자. 파이썬 라이브러리 중에서 numpy 을 활용하면 파일에 있는 데이터들을 간단히 읽을 수 있다.

학습 데이터 파일(train.txt)

#	x1	x2	x3	y
1	1	0	1	
1	0	2	2	
1	3	0	3	
1	0	4	4	
1	5	0	5	

위의 파일 데이터는 아래와 같이 Matrix 연산된다.

$$[w_1 \ w_2 \ w_3] \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} \end{bmatrix} = [y_1 \ y_2 \ y_3 \ y_4 \ y_5]$$

$$y_1 = w_1 x_{11} + w_2 x_{21} + w_3 x_{31}$$

$$y_2 = w_1 x_{12} + w_2 x_{22} + w_3 x_{32}$$

$$y_3 = w_1 x_{13} + w_2 x_{23} + w_3 x_{33}$$

$$y_4 = w_1 x_{14} + w_2 x_{24} + w_3 x_{34}$$

$$y_5 = w_1 x_{15} + w_2 x_{25} + w_3 x_{35}$$

파이썬 예제를 통하여 연산 과정을 확인해 보자.

다항변수 파일 읽기 예제

```
#Linear Regression Learning(Multi Variable in File)

import tensorflow as tf
import numpy as np

xy = np.loadtxt('train.txt', unpack=True, dtype='float32')
x_data = xy[0:-1]
y_data = xy[-1]

print 'x', x_data
print 'y', y_data

W = tf.Variable(tf.random_uniform([1,3], -10.0, 10.0))

X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

#hypothesis = tf.matmul(W, X) + b
hypothesis = tf.matmul(W, X)

#Cost Function
cost = tf.reduce_mean(tf.square(hypothesis - Y))

#Minimize(gradien descent algorithm)
#descent = W - tf.mul(0.1, tf.reduce_mean(tf.mul((tf.mul(W, X) - Y), X)))
#update = W.assign(descent)
a = tf.Variable(0.1)      #Learning rate, alpha
optimizer = tf.train.GradientDescentOptimizer(a)
train = optimizer.minimize(cost)

init = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init)

print 'Learning'
for step in range(1, 801):
    #sess.run(train)
    sess.run(train, feed_dict={X:x_data, Y:y_data})
    if step % 20 == 0:
        #print step, sess.run(cost), sess.run(W)
        print step, sess.run(cost, feed_dict={X:x_data, Y:y_data}), sess.run(W)

print 'Answer'
print '[0.0, 6.0]', sess.run(hypothesis, feed_dict={X:[[1.0], [0.0], [6.0]]})
print '[7.0, 0.0]', sess.run(hypothesis, feed_dict={X:[[1.0], [7.0], [0.0]]})
print '[8.0, 8.0]', sess.run(hypothesis, feed_dict={X:[[1.0], [8.0], [8.0]]})
```

실행 결과

```
x [[ 1.  1.  1.  1.  1.]
 [ 1.  0.  3.  0.  5.]
 [ 0.  2.  0.  4.  0.]]
y [ 1.  2.  3.  4.  5.]
Learning
20 0.560301 [[-1.77450132  1.46645796  1.55345726]]
40 0.162231 [[-0.95484257  1.25100112  1.29781103]]
60 0.0469727 [[-0.51379251  1.13506162  1.16024959]]
80 0.0136006 [[-0.27646732  1.07267547  1.08622885]]
100 0.00393794 [[-0.14876461  1.03910601  1.046399 ]]
120 0.0011402 [[-0.08004896  1.02104259  1.02496684]]
140 0.000330136 [[-0.04307358  1.01132286  1.01343453]]
160 9.55882e-05 [[-0.02317761  1.00609267  1.00722897]]
180 2.76764e-05 [[-0.01247162  1.00327837  1.0038898 ]]
200 8.01336e-06 [[-0.00671082  1.00176418  1.00209308]]
220 2.32034e-06 [[-0.00361106  1.00094926  1.00112629]]
240 6.71763e-07 [[-0.00194308  1.00051081  1.00060606]]
260 1.94432e-07 [[-0.0010455  1.0002749  1.00032616]]
280 5.63008e-08 [[ -5.62608009e-04  1.00014794e+00  1.00017548e+00]]
300 1.62919e-08 [[ -3.02658649e-04  1.00007963e+00  1.00009441e+00]]
320 4.71103e-09 [[ -1.62862343e-04  1.00004292e+00  1.00005078e+00]]
340 1.36012e-09 [[ -8.75866826e-05  1.00002313e+00  1.00002742e+00]]
360 3.96233e-10 [[ -4.71915373e-05  1.00001240e+00  1.00001466e+00]]
380 1.14907e-10 [[ -2.53333892e-05  1.00000668e+00  1.00000799e+00]]
400 3.28562e-11 [[ -1.36294566e-05  1.00000358e+00  1.00000429e+00]]
420 9.80904e-12 [[ -7.32092030e-06  1.00000191e+00  1.00000226e+00]]
440 2.73204e-12 [[ -3.89485558e-06  1.00000095e+00  1.00000119e+00]]
460 8.93152e-13 [[ -2.09480072e-06  1.00000048e+00  1.00000060e+00]]
480 2.52953e-13 [[ -1.20550237e-06  1.00000036e+00  1.00000048e+00]]
500 6.89226e-14 [[ -7.69197925e-07  1.00000024e+00  1.00000024e+00]]
520 2.55795e-14 [[ -3.44813913e-07  1.00000012e+00  1.00000012e+00]]
540 1.77636e-14 [[ -2.97130242e-07  1.00000012e+00  1.00000012e+00]]
560 1.77636e-14 [[ -2.97130242e-07  1.00000012e+00  1.00000012e+00]]
580 1.77636e-14 [[ -2.97130242e-07  1.00000012e+00  1.00000012e+00]]
600 1.77636e-14 [[ -2.97130242e-07  1.00000012e+00  1.00000012e+00]]
```

```
620 1.77636e-14 [[ -2.97130242e-07 1.00000012e+00 1.00000012e+00]]
640 1.77636e-14 [[ -2.97130242e-07 1.00000012e+00 1.00000012e+00]]
660 1.77636e-14 [[ -2.97130242e-07 1.00000012e+00 1.00000012e+00]]
680 1.77636e-14 [[ -2.97130242e-07 1.00000012e+00 1.00000012e+00]]
700 1.77636e-14 [[ -2.97130242e-07 1.00000012e+00 1.00000012e+00]]
720 1.77636e-14 [[ -2.97130242e-07 1.00000012e+00 1.00000012e+00]]
740 1.77636e-14 [[ -2.97130242e-07 1.00000012e+00 1.00000012e+00]]
760 1.77636e-14 [[ -2.97130242e-07 1.00000012e+00 1.00000012e+00]]
780 1.77636e-14 [[ -2.97130242e-07 1.00000012e+00 1.00000012e+00]]
800 1.77636e-14 [[ -2.97130242e-07 1.00000012e+00 1.00000012e+00]]
```

Answer

```
[0.0, 6.0] [[ 6.00000048]]
[7.0, 0.0] [[ 7.00000048]]
[8.0, 8.0] [[ 16.00000191]]
```

실행결과를 보면, 앞에서 매트릭스를 사용한 것과 동일하게 학습함을 알 수 있다.

2.4 Logistic(Binary) Classification

2.4.1 학습 모델(Hypothesis)

이번에는 입력 데이터를 학습하여 학습한 데이터를 두가지 형태로 분류하는 것에 대해서 알아보도록 하자. 예를들면, 어떤 학생이 보고서 제출 여부(x_1)와 공부한 시간(x_2), 수업에 참가한 회수(x_3)에 대한 데이터를 가지고 이 학생이 시험에 통과(Pass)할지 실패(Fail)할지를 예측하는 데이터 모델(가설)이 다음과 같은 테이블로 구성되어 있다고 하자.

분류학습 데이터

보고서제출 여부(X_1)	공부시간(x_1)	수업참가회수(x_2)	시험합격 여부(y)
1	2	1	0
1	3	2	0
1	3	3	0
1	5	5	1
1	7	5	1
1	2	5	1

위의 학습 데이터 모델은 입력 데이터를 학습하여 결과를 합격 혹은 불합격, 두가지 경우(0 또는 1)로 예측하면 되므로 가설(Hypothesis) 함수를 학습 데이터 모델에 맞도록 다음과 같이 정의 한다.

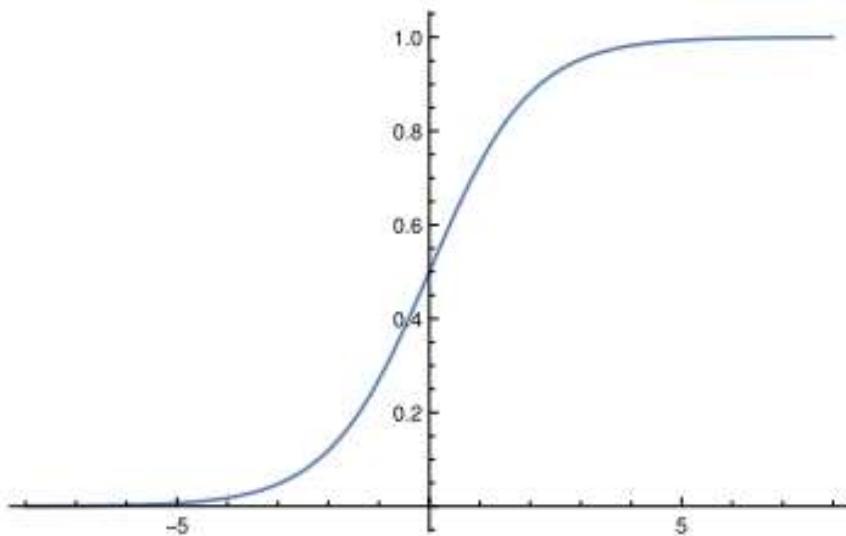
Logistic Function(Sigmoid)

$$h = wx$$

$$H(X) = \frac{1}{1 + e^{-h}}$$

위의 가설 함수는 입력 데이터 x 에 대해서 계산한 결과를 두가지 경우(0 또는 1)로 산출하는 함수이며 Logistic Function 혹은 “시그모이드” 함수라고 한다. h 을 가로축 $H(X)$ 를 세로축으로 하여 이 함수의 그래프를 그려 보면 다음과 같다.

Logistic Function 그라프



이 함수의 그래프를 보면 입력 데이터 $h=wx$ 에 대해서 출력값은 0과 1 사이 값임을 확인할 수 있다. Logistic Classification에서는 이 함수를 학습모델인 가설함수로 사용한다.

위의 학습모델 데이터들 중에서 제일 처음것을 가지고 가설함수가 연산되는 과정을 이해해 보자.

보고서제출 여부(x1)	공부시간(x1)	수업참가회수(x2)	시험합격 여부(y)
1	2	1	0

위의 데이터는 입력개수 3개에 대해서 출력데이터가 1개 산출되므로 매트릭스(행렬) 수식은 다음과 같다.

매트릭스 연산식

$$\begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = w_1x_1 + w_2x_2 + w_3x_3 = y_1$$

우리 머신이 처음에 w 매트릭스 값을 다음과 같이 가지고 있다고 가정하고 위의 연산을 수행해 보자.

매트릭스 연산 전개1

$$[1.0 \ 1.0 \ 1.0] \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} = 1 + 2 + 1 = 4$$

결과로 산출된 것에 대해서 시그모이드 함수를 적용 시키면 다음과 같다. 수학 상수인 exponential은 다음과 같은 값으로 정의 되어 있다.

수학 상수 exponential

$$e \approx 2.71828\dots$$

Sigmoid 연산1

$$\begin{aligned} \text{sigmoid}(h) &= \frac{1}{1+e^{-h}} = \frac{1}{1+e^{-4}} = \frac{1}{1+2.72^{-4}} \\ &= \frac{1}{1+0.018} = 0.98 \end{aligned}$$

$$\text{error} = 0.98 - 0 = 0.98$$

이번에는 우리 머신이 w 매트릭스 값을 다음과 같이 가지고 있다고 가정하고 위의 연산을 수행해 보자.

매트릭스 연산 전개2

$$\begin{bmatrix} -8.0 & 0.0 & 2.3 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} = -8.0 + 0 + 2.3 = -5.7$$

결과로 산출된 것에 대해서 시그모이드 함수를 적용 시키면 다음과 같다.

Sigmoid 연산2

$$\begin{aligned} \text{sigmoid}(-5.7) &= \frac{1}{1+e^{-5.7}} = \frac{1}{1+(2.72)^{-5.7}} \\ &= \frac{1}{1+300} = 0.003 \end{aligned}$$

$$\text{error} = 0.003 - 0 = 0.003$$

Sigmoid 연산 1에서는 결과가 0.98이고 Sigmoid 연산 2에서는 결과가 0.003이 산출되었다. 이것을 학습 데이터에 있는 정답과 비교해 보면 Sigmoid 연산 2가 좀 더 정답에 가깝게 산출되었다. 정답에 가깝게 산출된 Sigmoid 연산 2는 오차가 훨씬 작다. 오차가 작으면 비용(cost) 또한 적어진다.

이제 비용을 산출해 보자. 학습모델로 연산한 결과에 대해서 Sigmoid 함수를 적용했기 때문에 출력 결과는 0 와 1 사이의 값이 된다. 이렇게 값의 범위가 매우 축소 되었기 때문에, 우리는 조금 다른 비용함수를 도입해야 한다.

2.4.2 비용 함수

Logistic Classification 비용함수는 가설함수 $H(x)$ 에 Sigmoid 함수를 적용했기 때문에 산출되는 결과값이 0에서 1사이 범위 값이 된다. 이 결과값에 대해서 비용을 계산해야 하므로 우리는 아래와 같은 비용함수를 도입하기로 한다.

비용함수 정의

Cost function

$$cost(W) = \frac{1}{m} \sum c(H(x), y)$$

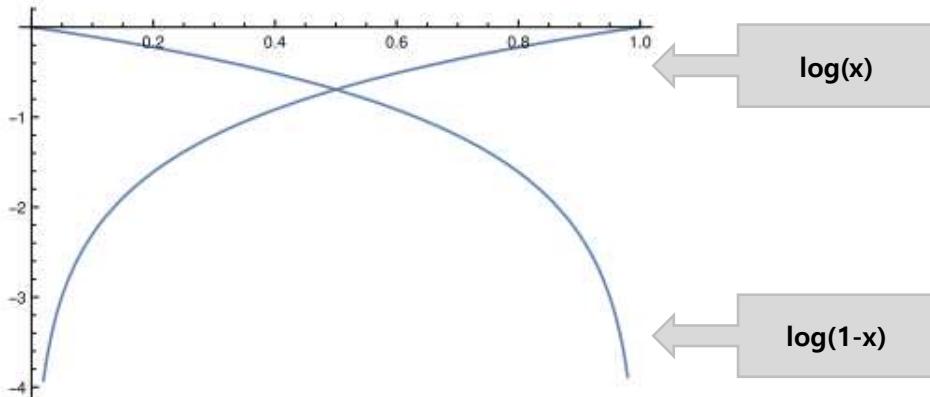
$$c(H(x), y) = \begin{cases} -\log(H(x)) & : y = 1 \\ -\log(1 - H(x)) & : y = 0 \end{cases}$$

$$c(H(x), y) = -y \log(H(x)) - (1 - y) \log(1 - H(x))$$

머신러닝에서 비용함수는 경사가 아래 방향으로 감소하는 매끈한 포물선 모양의 함수(Convex Function)가 되어야 한다. 앞의 Linear Regression 학습에서 확인해 본 것처럼, 우리의 머신이 weight 값을 학습해 나갈 때, 학습 모델에 주어진 데이터를 가설함수를 통하여 학습한 데이터와 실제 정답간의 차이(오차)로 인한 비용이 최소화 되는 방향으로 학습을 진행한다. 비용을 최소화하기 위해서는 비용함수를 미분한 경사도(변화치)가 0이 되는 지점을 찾아가야 하는데, 그러기 위해서는 비용함수의 모양이 경사가 아래 방향으로 감소하는 모양이 되어야 한다. 위의 비용함수가 이러한 모습이 되는지 아래에서 출력한 모양을 확인해 보자

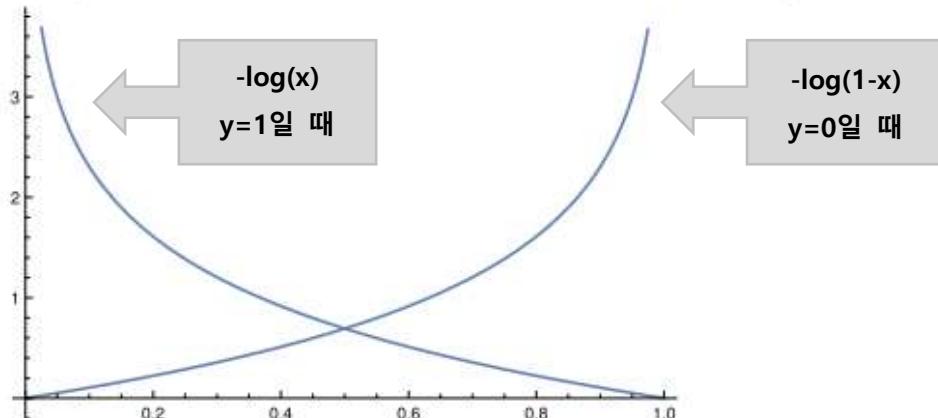
$\log(x)$ 와 $\log(1-x)$ 에 대한 그래프는 다음과 같다.

```
Show[Plot[Log[x], {x, 0, 1}], Plot[Log[1-x], {x, 0, 1}]]
```



$-\log(x)$ 와 $-\log(1-x)$ 에 대한 그래프는 다음과 같다.

```
show[Plot[-Log[x], {x, 0, 1}], Plot[-Log[1-x], {x, 0, 1}]]
```



$$C(H(x), y) = \begin{cases} -\log(H(x)) & : y = 1 \\ -\log(1 - H(x)) & : y = 0 \end{cases}$$

위의 비용함수는 정답 y 가 1 일때는 $-\log(x)$ 에서 오차값(미분, 변화율)을 산출하여 오차를 보정한다. 이 그래프 모양(기울기)을 보면, $H(x)$ 가 정답인 1에 가까워지면 비용은 0에 가까워지면서 경사가 완만해 진다. 그런데, $H(x)$ 가 오답인 0에 가까워지면 비용은 증가하면서 경사도 가파르게 상승함을 알 수 있다. 그만큼 오차를 많이 보정해야 한다.

정답 y 가 0 일때는 $-\log(1-x)$ 에서 오차값(미분, 변화율)을 산출한다. 이 그래프 모양(기울기)을 보면, $H(x)$ 가 정답인 0에 가까워지면 비용은 0에 가까워지면서 경사가 완만해 진다. 그런데, $H(x)$ 가 오답인 1에 가까워지면 비용은 증가하면서 경사도 가파르게 상승함을 알 수 있다. 그만큼 오차를 많이 보정해야 한다.

2.4.3 Logistic Regression

지금까지 Logistic Classification 에 도입된 가설함수와 비용함수에 대해서 알아봤다. 우리는 머신에게 가설함수와 비용함수를 알려주면, 우리 머신은 비용을 줄이는 방향으로 데이터를 학습할 수 있다.

Logistic Regression 은 위의 가설함수와 비용함수를 사용하여 머신러닝 알고리즘을 진행시키는 것이다. 이제 이것을 바탕으로 머신러닝 학습 알고리즘을 적용해 보자.

Logistic Regression(비용함수에서 오차 보정)

$$H(X) = \frac{1}{1 + e^{-W^T X}}$$

$$c(W) = -\frac{1}{m} \sum y \log(H(x)) + (1 - y) \log(1 - H(x))$$

$$W := W - \alpha \frac{\partial}{\partial W} cost(W)$$

지금까지 설명한 내용들을 TensorFlow 에 적용시켜 확인해 보자. 학습용 데이터는 다음과 같이 파일에 저장되어 있고, 이것을 읽어서 학습을 전개할 것이다.

학습 데이터 파일(train.txt)

#x1	x2	x3	y
1	2	1	0
1	3	2	0
1	3	4	0
1	5	5	1
1	7	5	1
1	2	5	1

학습모델(가설함수)

```
h = tf.matmul(W, X)      #matrix multiply
```

```
hypothesis = tf.div(1.0, 1.0 + tf.exp(-h))    #Logistic Function(Sigmoid)
```

위의 파일 데이터는 아래와 같이 Matrix 연산된다.

$$[w_1 \ w_2 \ w_3] \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} \end{bmatrix} = [y_1 \ y_2 \ y_3 \ y_4 \ y_5 \ y_6]$$

$$y_1 = w_1 x_{11} + w_2 x_{21} + w_3 x_{31}$$

$$y_2 = w_1 x_{12} + w_2 x_{22} + w_3 x_{32}$$

$$y_3 = w_1 x_{13} + w_2 x_{23} + w_3 x_{33}$$

$$y_4 = w_1 x_{14} + w_2 x_{24} + w_3 x_{34}$$

$$y_5 = w_1 x_{15} + w_2 x_{25} + w_3 x_{35}$$

$$y_6 = w_1 x_{16} + w_2 x_{26} + w_3 x_{36}$$

파이썬 예제를 통하여 연산 과정을 확인해 보자.

파이썬 예제

```
#Logistic(Binary, sigmoid) Classification

import tensorflow as tf
import numpy as np

xy = np.loadtxt('train.txt', unpack=True, dtype='float32')
x_data = xy[0:-1]
y_data = xy[-1]

print 'x', x_data
print 'y', y_data

X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

W = tf.Variable(tf.random_uniform([1, len(x_data)], -1.0, 1.0))
```

```

h = tf.matmul(W, X)      #matrix multiply
hypothesis = tf.div(1.0, 1.0 + tf.exp(-h))

#Cost Function
cost = -tf.reduce_mean(Y * tf.log(hypothesis) + (1-Y) * tf.log(1-hypothesis))

#Minimize
a = tf.Variable(0.1)      #Learning rate, alpha
optimizer = tf.train.GradientDescentOptimizer(a)
train = optimizer.minimize(cost)

init = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init)

print 'Learning'
for step in range(1, 2001):
    sess.run(train, feed_dict={X:x_data, Y:y_data})
    if step % 40 == 0:
        print step, sess.run(cost, feed_dict={X:x_data, Y:y_data}), sess.run(W)

print 'Answer'
print sess.run(hypothesis, feed_dict={X:[[1], [2], [2]]}) > 0.5
print sess.run(hypothesis, feed_dict={X:[[1], [5], [5]]}) > 0.5
print sess.run(hypothesis, feed_dict={X:[[1], [8], [3]]}) > 0.5

print sess.run(hypothesis, feed_dict={X:[[1,1], [4,3], [3,5]]}) > 0.5

```

실행 결과

```

x [[ 1.  1.  1.  1.  1.  1.]
 [ 2.  3.  3.  5.  7.  2.]
 [ 1.  2.  4.  5.  5.  5.]]
y [ 0.  0.  0.  1.  1.  1.]
Learning
40 0.598074 [[-0.26066899 -0.0005917  0.21602324]]
80 0.528374 [[-0.73799777 -0.08407902  0.41846961]]
120 0.480585 [[-1.16186607 -0.08699198  0.5244875 ]]
160 0.443575 [[-1.53959358 -0.06703199  0.59506941]]
200 0.413923 [[-1.87859535 -0.04189101  0.65070838]]
240 0.389688 [[-2.18527031 -0.01725524  0.69897068]]
280 0.369541 [[-2.4648881  0.00522749  0.74285603]]
320 0.352542 [[-2.72171998  0.02528252  0.78367263]]
360 0.338 [[-2.95921254  0.04307428  0.82207477]]
400 0.325409 [[-3.18015718  0.05887869  0.85845119]]
440 0.314387 [[-3.38682318  0.07297075  0.89306706]]
480 0.304645 [[-3.58107567  0.08559397  0.92612714]]
520 0.295958 [[-3.76445532  0.09695463  0.95779544]]
560 0.288153 [[-3.93824911  0.10722472  0.98821211]]
600 0.281091 [[-4.10353374  0.1165467  1.01749551]]
640 0.274658 [[-4.26122713  0.12504105  1.04574966]]
680 0.268767 [[-4.4121089  0.13280821  1.07306457]]
720 0.263344 [[-4.55685091  0.13993195  1.09951925]]

```

```
760 0.258328 [[-4.69602966 0.14648516 1.12518394]]
800 0.253668 [[-4.83015156 0.15252954 1.15012085]]
840 0.249321 [[-4.95965576 0.15811788 1.17438459]]
880 0.245253 [[-5.08492947 0.16329667 1.19802535]]
920 0.241432 [[-5.20631266 0.16810483 1.22108686]]
960 0.237832 [[-5.32410955 0.1725778 1.24360883]]
1000 0.234431 [[-5.4385891 0.17674589 1.26562619]]
1040 0.23121 [[-5.5499897 0.18063667 1.28717279]]
1080 0.22815 [[-5.65853167 0.18427329 1.30827689]]
1120 0.225239 [[-5.7644062 0.18767758 1.32896543]]
1160 0.222462 [[-5.86779165 0.19086836 1.34926331]]
1200 0.219807 [[-5.96884346 0.19386303 1.36919272]]
1240 0.217268 [[-6.06770992 0.19667664 1.38877368]]
1280 0.214829 [[-6.16451979 0.19932243 1.40802491]]
1320 0.212489 [[-6.25939465 0.20181312 1.42696369]]
1360 0.210236 [[-6.35244274 0.20416068 1.44560623]]
1400 0.208068 [[-6.44376755 0.20637429 1.46396649]]
1440 0.205973 [[-6.53345823 0.20846352 1.48205853]]
1480 0.20395 [[-6.62160254 0.21043691 1.49989498]]
1520 0.201994 [[-6.70828009 0.21230295 1.51748717]]
1560 0.2001 [[-6.79356146 0.21406741 1.53484595]]
1600 0.198265 [[-6.87751436 0.21573801 1.55198169]]
1640 0.196483 [[-6.96020174 0.2173201 1.56890368]]
1680 0.194753 [[-7.04168272 0.21881872 1.5856204 ]]
1720 0.193072 [[-7.12200975 0.22023965 1.60214067]]
1760 0.191436 [[-7.20123434 0.221588 1.61847186]]
1800 0.189843 [[-7.27940273 0.22286731 1.63462102]]
1840 0.188291 [[-7.35655928 0.22408247 1.65059519]]
1880 0.186777 [[-7.43274546 0.22523646 1.66640019]]
1920 0.1853 [[-7.50799942 0.22633235 1.68204272]]
1960 0.183857 [[-7.58235741 0.22737476 1.6975286 ]]
2000 0.182448 [[-7.65585423 0.22836529 1.7128619 ]]
```

Answer

```
[[False]]
[[ True]]
[[False]]
[[False True]]
```

실행 결과를 보면 비용은 점점 줄어들면서 w1, w2, w3 가 각각 -7.65, 0.23, 1.71로 수렴해 감을 알수 있다.

2.5 Multinomial(Softmax) Classification

앞에서 설명한 Logistic Regression은 입력 데이터를 학습하여 실패 혹은 성공 2가지 경우만 결과를 산출했지만, 이번에는 여러 개로 결과를 산출하는 방법에 대해서 기술한다. 예를들면, 어떤 학생이 보고서 제출(x1변수) 여부와 공부한 시간(x2변수)과 수업에 참여한 회수(x3변수)에 대한 데이터가 다음과 같은 학습모델 가설(Hypothesis)로 주어 졌을 때, 이 데이터를 학습하여 이 학생이 받게되는 평점(A, B, C)을 예측하는 모델을 생각해 보자.

2.5.1 학습 모델(Hypothesis)

학습 모델 데이터를 테이블로 정리하면 다음과 같다. 학업 성적을 평가할 때 공부한 시간이 많고 수업에 참여한 회수도 많으면 비례하여 학업 성적도 좋아진다 라는 일반적인 가정으로 데이터를 다음과 같이 만들었다. 이러한 데이터를 우리 머신이 학습하게 되면 사람이 판단하는 것과 유사한 결과를 도출하는지 검증해 보도록 하자.

학습 데이터

x1(보고서제출)	x2(공부시간)	x3(수업참여)	y(평점)
1	2	1	C
1	3	2	C
1	3	4	C
1	5	5	B
1	7	5	B
1	2	5	B
1	6	6	A
1	7	7	A

그런데 우리는 A, B, C 학점을 우리 머신이 잘 분류할 수 있도록 다음과 같이 코딩하여 알려주어야 한다. 이것을 one-hot encoding 이라 한다.

one-hot encoding

평점	one-hot encoding		
	0	1	0
C	0	0	1
B	0	1	0
A	1	0	0

아래 테이블은 one-hot encoding 을 적용한 것이다.

학습 데이터(one-hot encoding)

x1	x2	x3	y		
1	2	1	0	0	1
1	3	2	0	0	1
1	3	4	0	0	1
1	5	5	0	1	0
1	7	5	0	1	0
1	2	5	0	1	0
1	6	6	1	0	0
1	7	7	1	0	0

다항변수 데이터를 학습할 때 매트릭스(행열) 형태로 연산이 진행된다는 것을 앞에서 이미 배웠다.(2 장 2 절 5 항 참조) 이것을 바탕으로 다음과 같이 매트릭스(행열) 연산을 수행해 보자.

입력 데이터 3 개에 대해서 출력 데이터 3 개를 산출하는 w 매트릭스(행열)는 다음과 같이 연산된다.

$$[W][X] = [Y]$$

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + w_{13}x_3 \\ w_{21}x_1 + w_{22}x_2 + w_{23}x_3 \\ w_{31}x_1 + w_{32}x_2 + w_{33}x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

[W] 행열과 [X] 행열의 곱하는 좌우 순서를 바꾸면, 다음과 같이 행렬을 Tanspose 하여 연산하면 위와 동일한 결과로 산출된다.

$$[X][W] = [Y]$$

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}^T = \begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix}$$

그럼 위의 학습데이터 중에서 첫번째 행에 대해서 매트릭스(행렬) 연산하는 과정을 진행해 보자.

x1	x2	x3	y		
1	2	1	0	0	1

우리 머신이 학습을 시작할 때, 처음에는 W 매트릭스(행렬) 안의 값을 임의로 정하기 때문에 W 행렬값을 아래와 같이 정해놓고 연산을 진행한다. 행렬을 Transpose 하여 연산해도 결과는 동일하다.

W 매트릭스(행렬) 초기값

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 1 & 3 \\ 0 & 1 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$

매트릭스(행렬) 연산

$$\begin{bmatrix} 2 & 1 & 3 \\ 0 & 1 & 0 \\ 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 + 1 \cdot 2 + 3 \cdot 1 \\ 0 \cdot 1 + 1 \cdot 2 + 0 \cdot 1 \\ 0 \cdot 1 + 1 \cdot 2 + 2 \cdot 1 \end{bmatrix} = \begin{bmatrix} 7 \\ 2 \\ 4 \end{bmatrix}$$

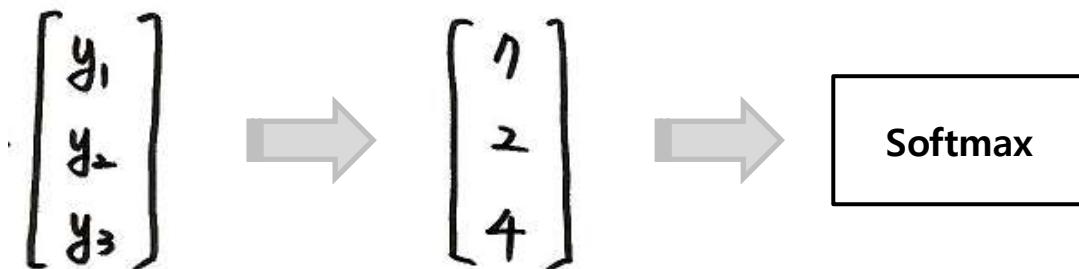
매트릭스(행렬) 연산(Transpose)

$$\begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 3 \\ 0 & 1 & 0 \\ 0 & 1 & 2 \end{bmatrix}^T = \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 1 & 1 & 1 \\ 3 & 0 & 2 \end{bmatrix}$$

$$= \begin{aligned} & 1 \cdot 2 + 2 \cdot 1 + 1 \cdot 3 \\ & 1 \cdot 0 + 2 \cdot 1 + 1 \cdot 0 \\ & 1 \cdot 0 + 2 \cdot 1 + 1 \cdot 2 \end{aligned}$$

$$= \begin{bmatrix} 7 & 2 & 4 \end{bmatrix}$$

이렇게 연산한 결과에 대해서 Logistic Classification에서는 Sigmoid 함수를 적용 시켰으나, Multinomial Classification에서는 Softmax 함수를 적용 시킨다.



2.5.2 Softmax 함수

Softmax 함수도 Sigmoid 함수처럼 출력 데이터를 0에서 1 사이의 값으로 변환하지만, 확률분포 형태로 변환한다는 점이 다르다. 아래의 Softmax 함수 정의를 통하여 Softmax 함수에서 계산되는 수치들을 산출해 보자.

Softmax 함수 정의

$$S(y_i) = \frac{e^{y_i}}{\sum_{j=1}^n e^{y_j}}$$

위의 출력결과에 대해서 Softmax 함수를 적용하면 다음과 같다.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix}$$

$$S(y_1) = S(1) = \frac{e^1}{e^1 + e^2 + e^4} = 0.95$$

$$S(y_2) = S(2) = \frac{e^2}{e^1 + e^2 + e^4} = 0.006$$

$$S(y_3) = S(4) = \frac{e^4}{e^1 + e^2 + e^4} = 0.046$$

좀 더 자세히 계산과정을 전개하면, 수학 상수인 exponential은 다음과 같은 값으로 정의되어 있다.

수학 상수 exponential

$$e \approx 2.71828\dots$$

그리므로 $e = 2.72$ (반올림값)로 하여 Softmax 함수를 적용하는 계산을 좀 더 자세히 전개하면 다음과 같다.

Softmax 함수를 적용하는 계산

$$\begin{aligned} S(y_1) &= S(1) = \frac{(2.72)^7}{(2.72)^7 + (2.72)^2 + (2.72)^4} \\ &= \frac{1130}{1130 + 7.4 + 54.7} = \frac{1130}{1192} = 0.95 \end{aligned}$$

$$S(y_2) = S(2) = \frac{7.4}{1192} = 0.006$$

$$S(y_3) = S(4) = \frac{54.7}{1192} = 0.046$$

$$\therefore S(1) + S(2) + S(4) = 0.95 + 0.006 + 0.046 = 1.0$$

Softmax 함수는 0에서 1.0 사이에서 값들이 얼마만큼의 비율(확률)로 분포되어 있는지를 나타내는 것이므로 Softmax 함수에서 계산한 결과값을 모두 합하면 위와 같이 1.0이 된다.

따라서 출력값 y 에 대해서 Softmax 함수를 적용시킨 최종 결과값은 다음과 같다.

Softmax 함수 적용값

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix} \xrightarrow{\text{Softmax}} \begin{bmatrix} 0.95 \\ 0.006 \\ 0.046 \end{bmatrix}$$

위의 출력결과에 대해서 정답 y 행렬을 대입하여 오류를 다음과 같이 계산할 수 있다.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix} \xrightarrow{\text{Softmax}} \begin{bmatrix} 0.95 \\ 0.006 \\ 0.046 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \xrightarrow{\text{error}} \begin{bmatrix} 0.95 \\ 0.006 \\ -0.954 \end{bmatrix}$$

그런데 우리는 오차를 보정할 때, 비용함수를 사용하여 보정하는 것을 이미 배웠다. 오차가 크면 비용 또한 커지고 오차를 보정하는 값(미분값)도 비례하여 커진다. 반대로 오차가 작으면 보정하는 값(미분값)도 작아진다.

2.5.3 비용 함수

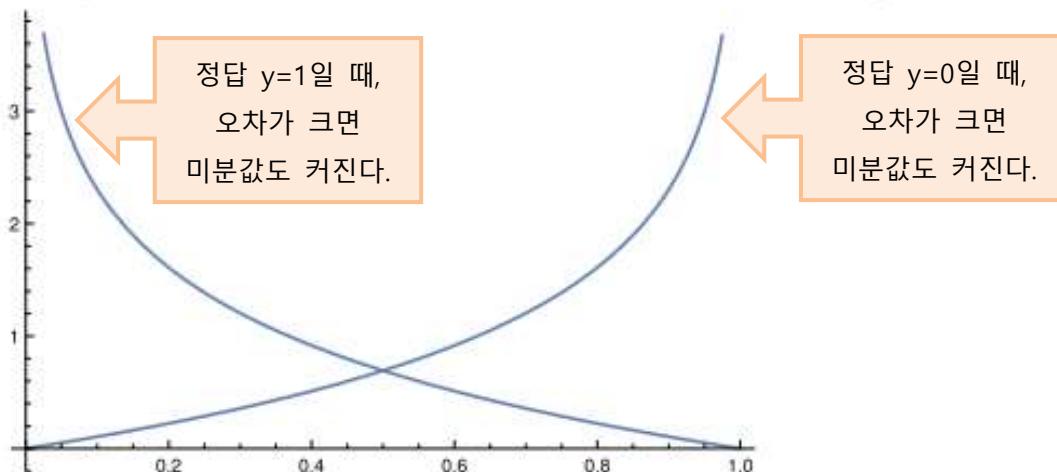
앞에서 이미 배운 Logistic Classification에서 사용한 비용함수를 Multinomial Classification에서도 그대로 사용한다.

비용 함수 정의

$$c(W) = -\frac{1}{m} \sum y \log(H(x)) + (1 - y) \log(1 - H(x))$$

비용함수 그래프

```
show[Plot[-Log[x], {x, 0, 1}], Plot[-Log[1-x], {x, 0, 1}]]
```



우리 머신에게 교육(지도)하는 정답이 $y=1$ 일 때, 우리 머신이 $y=0$ 에 가까운 값을 계산하면, 위의 비용함수 그래프에서 확인할 수 있듯이 비용에 대한 미분값(음수)이 커져서 오차도 그 만큼 많이 보정한다. 반대로, 우리 머신에게 교육(지도)하는 정답이 $y=0$ 일 때, 우리 머신이 $y=1$ 에 가까운 값을 계산하면, 위의 비용함수 그래프에서 확인할 수 있듯이 비용에 대한 미분값(양수)이 커져서 오차도 그 만큼 많이 보정한다.

미분값으로 오차를 보정하는 수식은 다음과 같다.

미분값으로 오차 보정

$$W := W - \alpha \frac{\partial}{\partial W} cost(W)$$

미분값이 음수이면 W 는 증가(더해짐)하는 방향으로 보정되고, 미분값이 양수이면 W 는 감소(빼짐)하는 방향으로 보정된다.

위와 같이 오차가 보정되는 과정은 사실 앞에서 이미 학습한 것을 한번 더 확인하는 것이다. 즉, 기본적인 머신러닝 알고리즘은 동일하다. 단지, 오차를 보정하는 w 가 매트릭스(행렬) 형태로 많아 졌다는 것이 차이점이다.

2.5.4 TensorFlow 실습

지금까지 배운 내용을 TensorFlow 를 사용하여 실습해 보자. 학습 데이터는 train.txt 파일에 다음과 같이 저장되어 있다.

학습 데이터 파일(train.txt)

#x0	x1	x2	y	[A B C]
1	2	1	0	0 1
1	3	2	0	0 1
1	3	4	0	0 1
1	5	5	0	1 0
1	7	5	0	1 0
1	2	5	0	1 0
1	6	6	1	0 0
1	7	7	1	0 0

학습모델(가설함수)

```

x_data = np.transpose(xy[0:3])
y_data = np.transpose(xy[3:])

hypothesis = tf.nn.softmax(tf.matmul(X, W)) #Softmax

W = tf.Variable(tf.zeros([3,3]))

```

위의 학습모델은 softmax 함수에서 Matrix 을 Transpose 하여 연산한다.

Matrix Transpose 예제

$$A = \begin{bmatrix} 2 \\ 8 \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} 2 & 8 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 4 \\ 1 & -1 \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} 0 & 2 & 1 \\ 1 & 4 & -1 \end{bmatrix}$$

매트릭스(행렬) 연산 요약

$$\begin{array}{c}
 X \\
 \left[\begin{array}{ccc} 1 & 2 & 1 \\ 1 & 3 & 2 \\ 1 & 3 & 4 \\ 1 & 5 & 5 \\ 1 & 1 & 5 \\ 1 & 2 & 5 \\ 1 & 6 & 6 \\ 1 & 1 & 7 \end{array} \right] \times W = Y
 \end{array}$$

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

$$Y = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

파이썬 Tensor Flow 예제

#Multi Classification(Softmax Classifier)

```

import tensorflow as tf
import numpy as np

xy = np.loadtxt('train.txt', unpack=True, dtype='float32')
x_data = np.transpose(xy[0:3])
y_data = np.transpose(xy[3:])

print 'x', x_data
print 'y', y_data

X = tf.placeholder("float", [None,3]) #x1, x2 and 1 (for bias)
Y = tf.placeholder("float", [None,3]) #A,B,C (for classes)

W = tf.Variable(tf.zeros([3,3]))

#hypothesis = tf.nn.softmax(tf.matmul(W, X)) #Softmax
hypothesis = tf.nn.softmax(tf.matmul(X, W)) #Softmax

#Cross-Entropy Cost Function
cost = tf.reduce_mean(-tf.reduce_sum(Y * tf.log(hypothesis), reduction_indices=1))
#cost = -tf.reduce_mean(Y * tf.log(hypothesis) + (1-Y) * tf.log(1-hypothesis))

#Minimize
#a = tf.Variable(0.1) #Learning rate, alpha
learning_rate = 0.001 #Learning rate, alpha

```

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
#train = optimizer.minimize(cost)

init = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init)

print 'Learning'

#with tf.Session() as sess:
#    sess.run(init)

for step in range(1, 2001):
    sess.run(optimizer, feed_dict={X:x_data, Y:y_data})
    if step % 40 == 0:
        print step, sess.run(cost, feed_dict={X:x_data, Y:y_data}), sess.run(W)

print 'Answer'
a = sess.run(hypothesis, feed_dict={X:[[1, 11, 7]]})
print a, sess.run(tf.argmax(a,1))

b = sess.run(hypothesis, feed_dict={X:[[1, 3, 4]]})
print b, sess.run(tf.argmax(b,1))

c = sess.run(hypothesis, feed_dict={X:[[1, 1, 0]]})
print c, sess.run(tf.argmax(c,1))

all = sess.run(hypothesis, feed_dict={X:[[1, 11, 7], [1, 3, 4], [1, 1, 0]]})
print all, sess.run(tf.argmax(all,1))
```

실행 결과

```
x [[ 1.  2.  1.]
 [ 1.  3.  2.]
 [ 1.  3.  4.]
 [ 1.  5.  5.]
 [ 1.  7.  5.]
 [ 1.  2.  5.]
 [ 1.  6.  6.]
 [ 1.  7.  7.]]
y [[ 0.  0.  1.]
 [ 0.  0.  1.]
 [ 0.  0.  1.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]]
```

```
[ 0.  1.  0.]  
[ 1.  0.  0.]  
[ 1.  0.  0.]]  
  
Learning  
40 1.07718 [[-0.00360971  0.0009908  0.00261897]  
 [ 0.00523967  0.00822053 -0.01345997]  
 [ 0.00525547  0.01322516 -0.01848039]]  
80 1.06892 [[-0.00757766  0.00104431  0.00653345]  
 [ 0.00864143  0.01168522 -0.02032618]  
 [ 0.00869319  0.02167628 -0.03036899]]  
120 1.06474 [[-0.01174343  0.00059366  0.01114993]  
 [ 0.01102422  0.01262065 -0.02364418]  
 [ 0.01112648  0.02755484 -0.03868061]]  
160 1.06195 [[-0.01603327 -0.00013361  0.0161671 ]  
 [ 0.01275215  0.01219122 -0.02494243]  
 [ 0.01291901  0.03201376 -0.0449318 ]]  
200 1.05967 [[-0.02040361 -0.0010163   0.02142017]  
 [ 0.014037    0.01101371 -0.02504958]  
 [ 0.01428254  0.03566487 -0.04994623]]  
240 1.0576 [[-0.02482544 -0.00198836  0.02681414]  
 [ 0.01501956  0.00942214 -0.02444021]  
 [ 0.0153576   0.03884017 -0.05419625]]  
280 1.05562 [[-0.02927888 -0.00301298  0.03229227]  
 [ 0.0157967   0.00760159 -0.02339643]  
 [ 0.01624058   0.04172418 -0.05796292]]  
320 1.05369 [[-0.03375032 -0.00406909  0.03781986]  
 [ 0.01643499  0.00565713 -0.02209004]  
 [ 0.01699754  0.04442208 -0.06141761]]  
360 1.05178 [[-0.03823057 -0.00514436  0.04337543]  
 [ 0.01697973  0.00364981 -0.02062725]  
 [ 0.01767327  0.04699533 -0.06466638]]  
400 1.0499 [[-0.04271348 -0.00623144  0.04894549]  
 [ 0.01746128  0.00161567 -0.01907442]  
 [ 0.0182977   0.04948055 -0.06777577]]  
440 1.04804 [[-0.04719502 -0.00732589  0.05452154]  
 [ 0.01789973 -0.00042375 -0.0174732 ]]
```

```
[ 0.01889044  0.05189994 -0.07078764]]  
480 1.04619 [[-0.05167255 -0.00842495  0.06009817]  
[ 0.01830834 -0.00245542 -0.01584989]  
[ 0.01946442  0.05426721 -0.07372863]]  
520 1.04436 [[-0.05614433 -0.00952692  0.06567197]  
[ 0.01869579 -0.0044715 -0.01422103]  
[ 0.02002799  0.05659093 -0.07661567]]  
560 1.04255 [[-0.06060927 -0.01063071  0.07124075]  
[ 0.01906763 -0.00646709 -0.01259706]  
[ 0.0205864   0.05887675 -0.07945966]]  
600 1.04075 [[-0.06506664 -0.01173561  0.0768031 ]  
[ 0.0194277 -0.00843936 -0.01098453]  
[ 0.02114318  0.06112823 -0.08226758]]  
640 1.03897 [[-0.06951602 -0.01284121  0.08235811]  
[ 0.01977823 -0.01038669 -0.00938752]  
[ 0.02170034  0.0633477 -0.08504403]]  
680 1.0372 [[-0.07395709 -0.0139472   0.08790522]  
[ 0.02012094 -0.01230817 -0.00780855]  
[ 0.02225934  0.06553682 -0.08779193]]  
720 1.03545 [[-0.07838969 -0.01505336  0.09344403]  
[ 0.02045681 -0.01420327 -0.00624907]  
[ 0.0228209   0.06769683 -0.09051328]]  
760 1.03371 [[-0.0828137 -0.01615958  0.09897431]  
[ 0.02078664 -0.01607191 -0.00470999]  
[ 0.02338562  0.06982855 -0.09320946]]  
800 1.03199 [[-0.08722907 -0.01726575  0.10449589]  
[ 0.02111084 -0.01791416 -0.00319175]  
[ 0.02395365  0.07193261 -0.0958813 ]]  
840 1.03028 [[-0.09163577 -0.01837179  0.11000868]  
[ 0.02142973 -0.01973009 -0.00169454]  
[ 0.02452511  0.07400963 -0.09852964]]  
880 1.02859 [[-0.09603376 -0.01947763  0.11551257]  
[ 0.02174366 -0.02151984 -0.00021837]  
[ 0.02510011  0.07606016 -0.10115486]]  
920 1.02691 [[-0.10042305 -0.02058324  0.12100752]  
[ 0.02205274 -0.02328383  0.00123676]]
```

```
[ 0.02567858  0.07808453 -0.10375748]]  
960 1.02524 [[-0.10480364 -0.02168856  0.1264935 ]  
[ 0.0223572  -0.02502229  0.00267096]  
[ 0.02626051  0.08008315 -0.10633783]]  
1000 1.02359 [[-0.10917556 -0.02279357  0.13197048]  
[ 0.02265713 -0.02673553  0.00408448]  
[ 0.02684582  0.08205638 -0.10889614]]  
1040 1.02195 [[-0.11353879 -0.02389823  0.13743845]  
[ 0.02295265 -0.02842382  0.00547753]  
[ 0.02743438  0.08400457 -0.11143266]]  
1080 1.02032 [[-0.11789338 -0.02500249  0.14289735]  
[ 0.02324388 -0.03008749  0.00685027]  
[ 0.02802613  0.08592818 -0.1139477 ]]  
1120 1.0187 [[-0.12223933 -0.02610631  0.14834721]  
[ 0.02353085 -0.03172678  0.00820295]  
[ 0.02862091  0.08782752 -0.11644145]]  
1160 1.0171 [[-0.12657665 -0.02720968  0.15378797]  
[ 0.02381369 -0.03334217  0.00953571]  
[ 0.02921864  0.08970276 -0.11891422]]  
1200 1.01551 [[-0.13090537 -0.02831259  0.15921964]  
[ 0.02409251 -0.03493391  0.01084882]  
[ 0.02981926  0.09155434 -0.1213662 ]]  
1240 1.01393 [[-0.13522549 -0.02941498  0.16464218]  
[ 0.02436746 -0.03650234  0.01214251]  
[ 0.03042271  0.0933825  -0.1237976 ]]  
1280 1.01237 [[-0.13953707 -0.03051682  0.17005566]  
[ 0.0246385  -0.03804773  0.01341703]  
[ 0.03102877  0.09518754 -0.12620853]]  
1320 1.01081 [[-0.14384009 -0.0316181   0.17546001]  
[ 0.02490574 -0.03957037  0.01467263]  
[ 0.03163737  0.0969699  -0.12859927]]  
1360 1.00927 [[-0.1481346  -0.03271877  0.18085523]  
[ 0.02516932 -0.0410706   0.01590945]  
[ 0.03224845  0.09872973 -0.13097005]]  
1400 1.00774 [[-0.15242061 -0.0338188   0.18624131]  
[ 0.02542936 -0.04254866  0.01712769]]
```

```
[ 0.032862  0.10046744 -0.13332106]]  
1440 1.00622 [[-0.15669812 -0.03491816  0.19161826]  
[ 0.02568586 -0.04400481  0.01832767]  
[ 0.03347781  0.10218336 -0.13565248]]  
1480 1.00471 [[-0.16096717 -0.03601684  0.19698605]  
[ 0.02593891 -0.04543947  0.01950958]  
[ 0.03409579  0.10387766 -0.13796446]]  
1520 1.00321 [[-0.16522779 -0.03711482  0.20234472]  
[ 0.02618856 -0.04685287  0.02067362]  
[ 0.03471585  0.10555065 -0.14025721]]  
1560 1.00173 [[-0.16947998 -0.03821205  0.2076942 ]  
[ 0.02643498 -0.04824532  0.02181999]  
[ 0.03533794  0.10720262 -0.14253093]]  
1600 1.00025 [[-0.17372379 -0.03930851  0.21303457]  
[ 0.02667816 -0.04961701  0.02294892]  
[ 0.03596193  0.10883391 -0.14478582]]  
1640 0.998783 [[-0.17795923 -0.04040418  0.21836576]  
[ 0.02691813 -0.05096832  0.02406058]  
[ 0.03658769  0.11044472 -0.14702207]]  
1680 0.997325 [[-0.18218634 -0.04149904  0.22368778]  
[ 0.02715505 -0.05229953  0.02515514]  
[ 0.03721523  0.11203532 -0.14923991]]  
1720 0.995879 [[-0.18640512 -0.04259307  0.22900064]  
[ 0.02738893 -0.05361091  0.02623278]  
[ 0.03784437  0.11360593 -0.15143953]]  
1760 0.994443 [[-0.19061561 -0.04368621  0.23430432]  
[ 0.02761984 -0.05490262  0.02729375]  
[ 0.03847508  0.11515696 -0.15362106]]  
1800 0.993016 [[-0.1948178  -0.04477847  0.23959883]  
[ 0.02784796 -0.05617502  0.02833821]  
[ 0.03910732  0.11668856 -0.15578473]]  
1840 0.991599 [[-0.19901177 -0.04586982  0.24488418]  
[ 0.0280732  -0.05742837  0.02936642]  
[ 0.0397409   0.118201   -0.15793066]]  
1880 0.990191 [[-0.20319749 -0.04696023  0.25016034]  
[ 0.02829576 -0.05866294  0.03037855]]
```

```
[ 0.04037589  0.11969453 -0.16005899]]  
1920 0.988793 [[-0.207375   -0.04804968  0.25542736]  
[ 0.02851567 -0.05987896  0.03137478]  
[ 0.04101212  0.12116939 -0.16216995]]  
1960 0.987404 [[-0.21154435 -0.04913814  0.26068521]  
[ 0.02873289 -0.0610766   0.0323553 ]  
[ 0.04164946  0.12262588 -0.16426368]]  
2000 0.986024 [[-0.21570554 -0.05022559  0.2659339 ]  
[ 0.02894764 -0.06225618  0.0333203 ]  
[ 0.04228799  0.12406422 -0.16634034]]
```

Answer

```
[[ 0.46269575  0.35486636  0.18243794]] [0]  
[[ 0.33820781  0.42100048  0.24078506]] [1]  
[[ 0.27005666  0.29087916  0.43906376]] [2]  
[[ 0.46269566  0.3548663   0.18243791]  
[ 0.33820781  0.42100048  0.24078506]  
[ 0.27005666  0.29087916  0.43906376]] [0 1 2]
```

♣ 책에 있는 모든 소스들은 아래 링크에서 다운로드 가능합니다.

<https://github.com/kernel-bz/ml>

머신러닝/딥러닝 TensorFlow 실습

Third Edition

3. Deep Learning

3. Deep Learning

최근에는 딥러닝(Deep Learning) 혹은 딥뉴럴네트워크(Deep Neural Network) 방식이 효과적인 머신러닝 알고리즘으로 인정을 받고 있다. 이 방식은 고전적인 인공신경망(Artificial Neural Network)을 좀더 개선하여, 사람의 뇌가 수많은 신경세포들에 의해 움직인다는 점에 착안하여 만들어졌는데, 많은 수의 노드들을 놓고 그들을 연결하여 이들의 연결값들을 훈련시켜 데이터를 학습시킨다. 즉, 관측된 데이터는 많은 요인들이 서로 다른 가중치로 기여하여 만들어졌다고 생각할 수 있는데, 인공신경망에서는 요인들을 노드로, 가중치들을 연결선으로 표시하여 거대한 네트워크를 만든 것이다. 딥러닝은 간략히 말해 이러한 네트워크 자료구조들을 층층히 쌓은 매우 깊은 학습 네트워크를 일컫는다. 이 방식은 이미지인식, 음성인식, 자연어처리 등 다양한 부분에 응용되고 있다.

딥러닝 연구를 주도한 토론토 대학교의 힌تون(Hinton) 교수



사진출처: 토론토대학교(<http://boundless.utoronto.ca/?s=our-supporters/geoffrey-hinton/>)

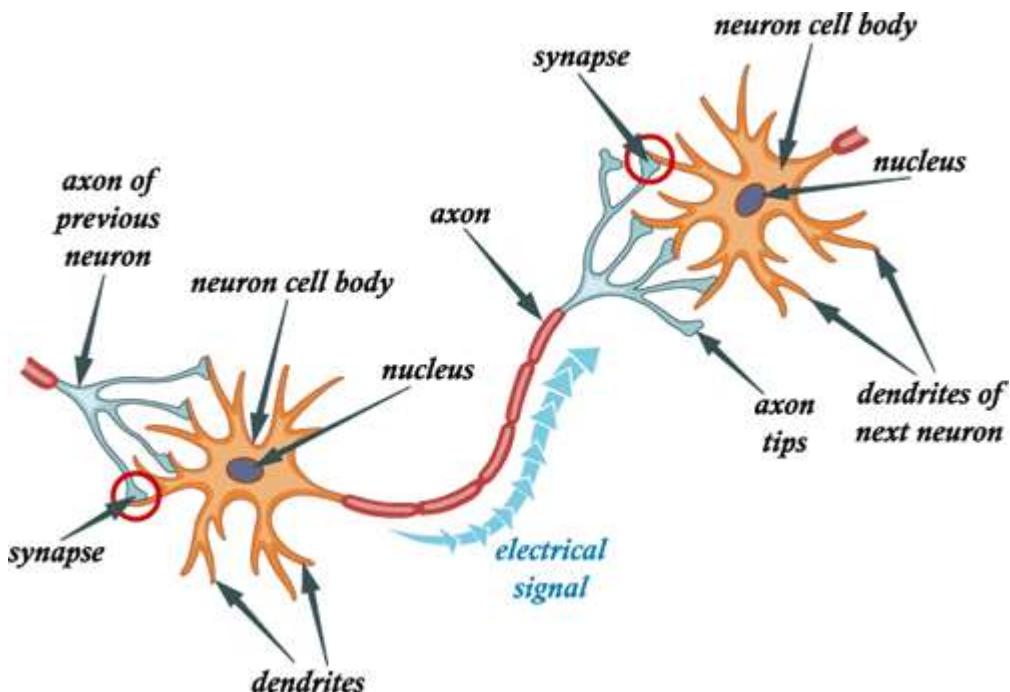
이번장은 머신러닝에서 오랫동안 해결하지 못했던 XOR 문제를 Neural Network로 어떻게 해결하는지 자세히 설명하다. 이것은 Neural Network를 구성하는 기본이 된다. Neural Network를 여러 계층으로 쌓아서 머신러닝을 좀더 깊게 할 수가 있는데, 이것이 Deep Learning이다. 이번장은 Deep Learning의 정확성을 향상시키는 방법은 어떤 것이 있는지 알아보고 TensorFlow에서 자세히 실습할 수 있도록 기술한다.

3.1 딥러닝 기본

최근에 딥러닝이 인정받는 이유중에 하나로 특징값 학습(Representation Learning)이라는 것이다. 기계학습의 단점 중 하나는 좋은 특징값(Weight)을 정의하기가 쉽지 않았다는 점이었는데, 딥러닝은 여러 단계의 계층적 학습과정을 거치며 적절한 특징값(Weight)을 스스로 생성해낸다. 이 특징값들은 많은 양의 데이터로부터 생성할 수 있는데, 이를 통해 기존에는 인간이 포착하지 못했던 특징값들까지 데이터에 의해 포착할 수 있게 되었다. 딥러닝은 마치 인간이 사물을 인식하는 방법처럼, 모서리, 변, 면 등의 하위 구성요소부터 시작하여 나중엔 눈, 코, 입과 같이 더 큰 형태로의 계층적 추상화를 가능하게 하였는데, 이는 인간이 사물을 인식하는 방법과 유사하다고 알려져 있다.

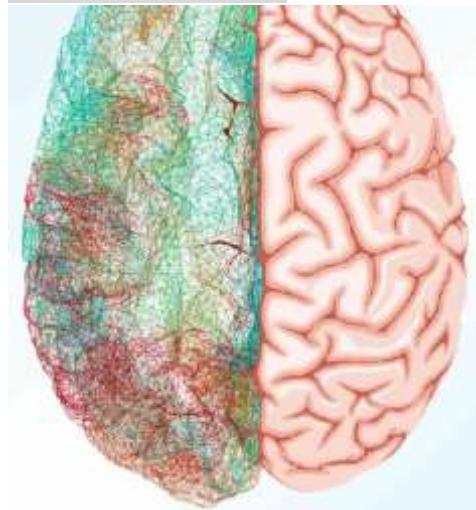
뉴런의 생물학적 연결 구조

그림출처: <https://blog.bufferapp.com/why-practice-actually-makes-perfect-how-to-rewire-your-brain-for-better-performance>



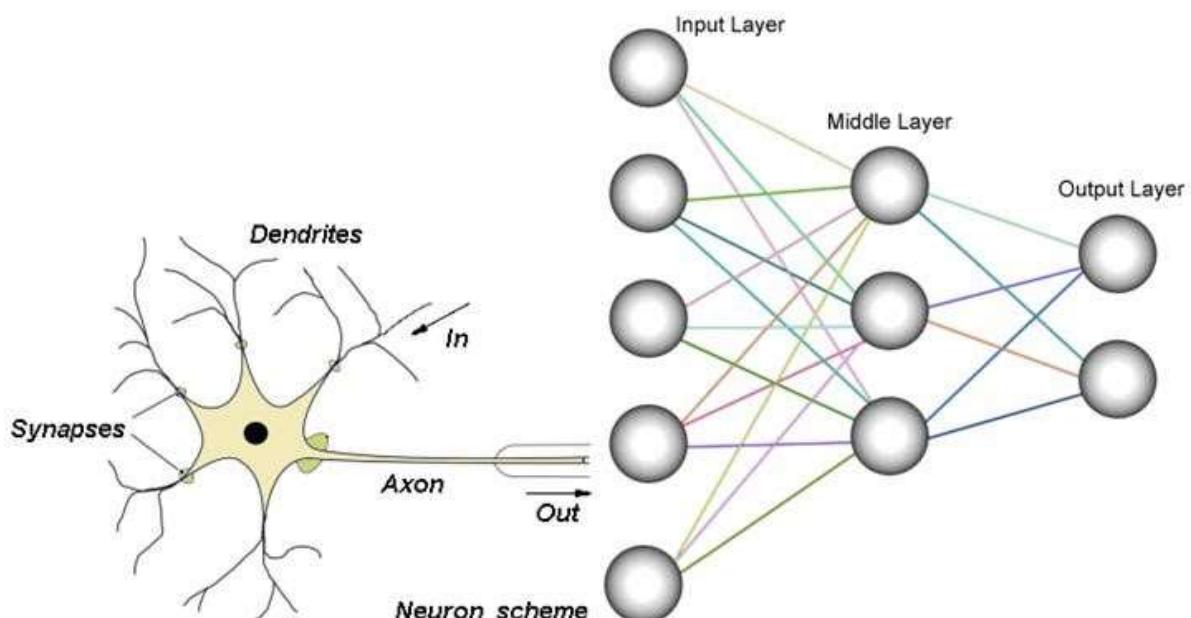
인간의 뇌세포를 형성하고 있는 뉴런(neuron)들은 외부에서 입력되는 신호들(오감: 시각, 청각, 후각, 미각 촉각)에 대해서 가중치를 부여해 가며 학습 한다고 알려져 있다. 이러한 뉴런들에서 학습한 가중치가 축색돌기(axon)을 거치며 행동함수값으로 변환되고 이 값이 다시 시냅스(synapse)들을 통하여 또 다른 뉴런들의 수상돌기(dendrites)들과 연결된다. 위와 같이 우리의 뇌는 거대하게 연결되어 학습 데이터가 구축된다. 참고로, 인간의 뇌는 수십조의 뉴런들과 수천조의 시냅스들로 연결되어 있다고 알려져 있다.

뇌세포 연결 이미지



그림출처: 토론토대학교(<http://boundless.utoronto.ca/impact/can-a-virtual-brain-repair-a-real-one/>)

뉴런을 컴퓨터에서 자료구조로 표현

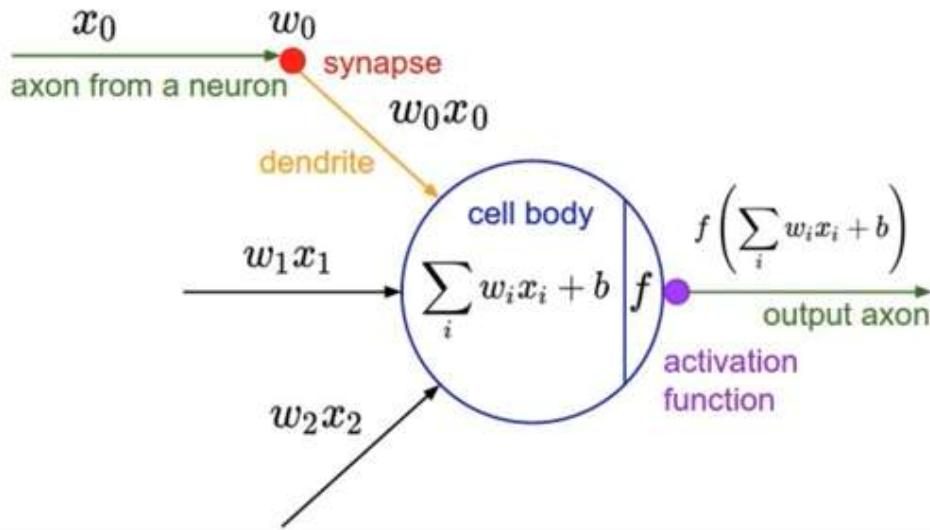


그림출처: <https://sites.google.com/site/mrstevensonstechclassroom/hl-topics-only/4a-robotics-ai/neural-networks-computational-intelligence>

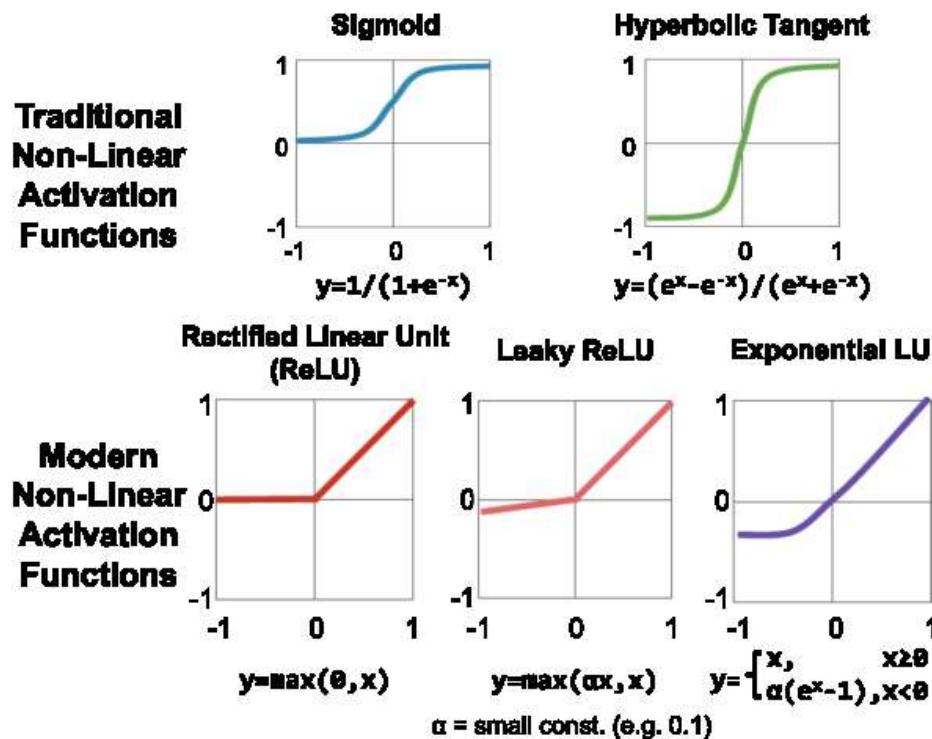
3.1.1 행동 함수

그동안 뉴런들이 연결되어 동작하는 방식을 많은 학자들이 연구하여 수학적으로 모델링하고 있으며, 다음과 같은 행동 함수 형태로 표현할 수 있다.

뉴런 행동함수의 수학적 모델링



위의 모델에서 cell body에서는 입력 데이터들을 가중치에 곱하여 합하는 작업을 하고, 이 합한 값에 대해서 output 쪽에서 행동함수(activation function)를 적용하여 출력한다. 행동함수는 수학적으로 Sigmoid 함수, Softmax 함수, tanh 함수들을 사용한다.



그림출처:

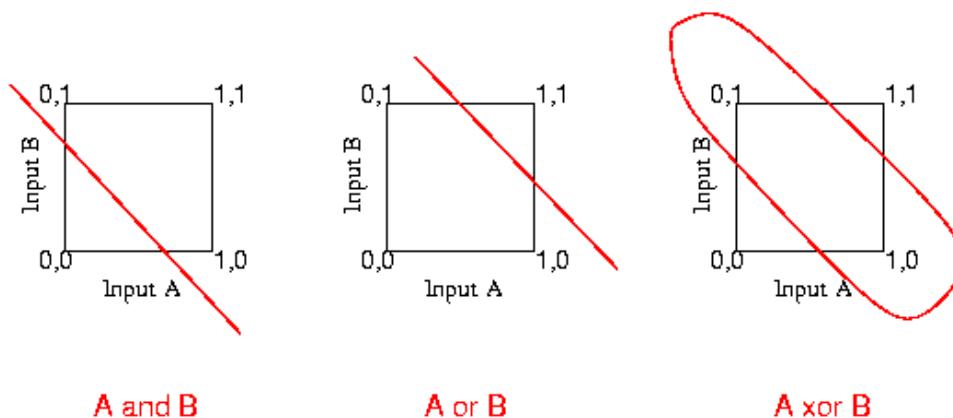
[http://gmelli.org/RKB/Rectified_Linear_Unit_\(ReLU\)_Activation_Function](http://gmelli.org/RKB/Rectified_Linear_Unit_(ReLU)_Activation_Function)

참조:

<https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>

3.1.2 XOR 문제

위의 수학적 모델에서 AND 논리나 OR 논리는 쉽게 해결이 되는데, XOR 논리는 쉽게 풀리지 않는 문제가 있었다.



아래는 AND, OR, XOR 논리가 매트릭스 행렬로 연산되는 값들을 예시한 것이다.

AND 논리 연산

x_1	x_2	AND
0	0	0
1	0	0
0	1	0
1	1	1

$$\begin{aligned}
 & \text{AND} \\
 & [w_1 \ w_2] \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \\
 & [0.4 \ 0.4] \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

OR 논리 연산

x_1	x_2	OR
0	0	0
1	0	1
0	1	1
1	1	1

$$w \quad X \quad Y$$

$$[w_1 \ w_2] \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \xrightarrow{\textcircled{S}} [0 & 1 & 1 & 1]$$

$$[0.6 \ 0.6] \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \xrightarrow{\textcircled{S}} [0 & 1 & 1 & 1]$$

그러나, XOR 논리는 매트릭스 행렬 연산에서 가중치 값을 쉽게 찾을 수 없다.

XOR 연산 ??

x_1	x_2	XOR
0	0	0
1	0	1
0	1	1
1	1	0

$$w \quad X \quad Y$$

$$[w_1 \ w_2] \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \xrightarrow{\textcircled{S}} [0 & 1 & 1 & 0]$$

$$[? \ ?]$$

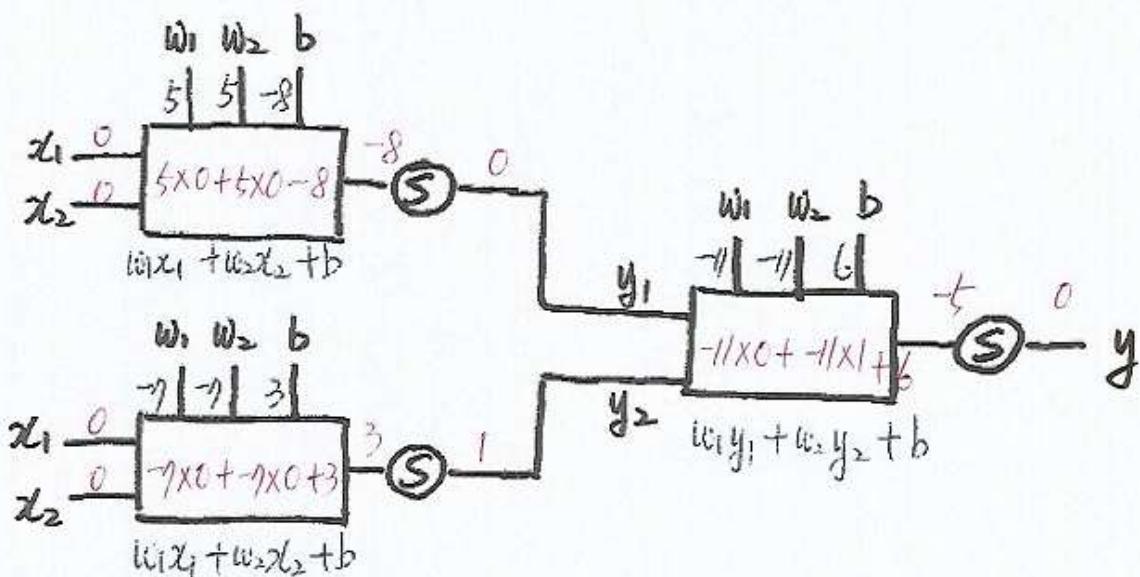
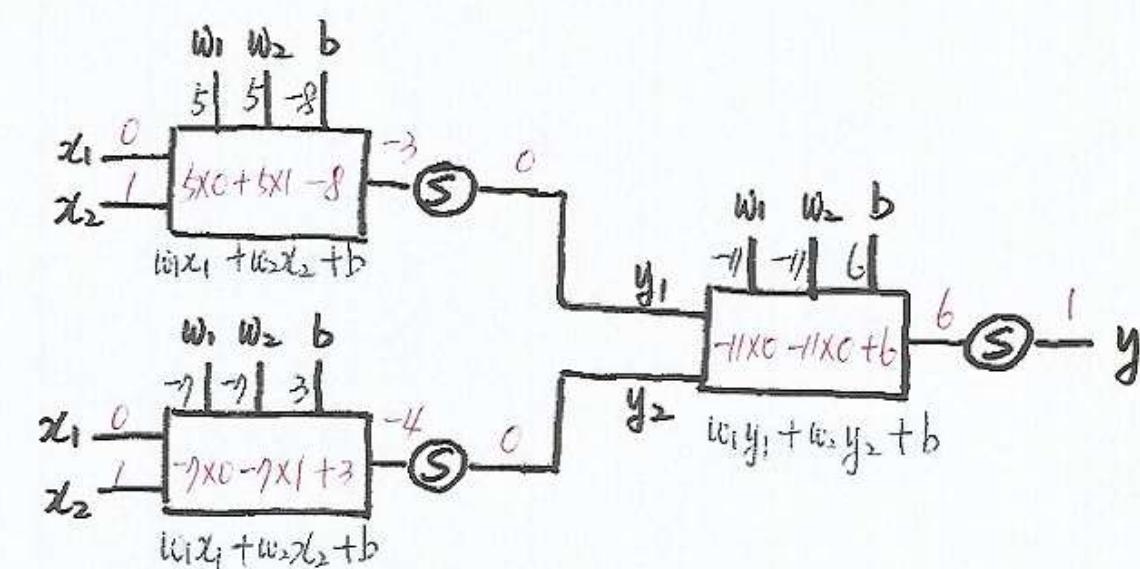
딥러닝에서 XOR 문제를 오랫동안 해결하지 못하고 있다가, 1974년~1986년도에 이르러 이 문제를 Neural Network으로 해결하면서 새로운 전환기를 맞이하게 된다. Neural Network에서 XOR 문제를 어떻게 해결했는지 자세히 알아보도록 하자.

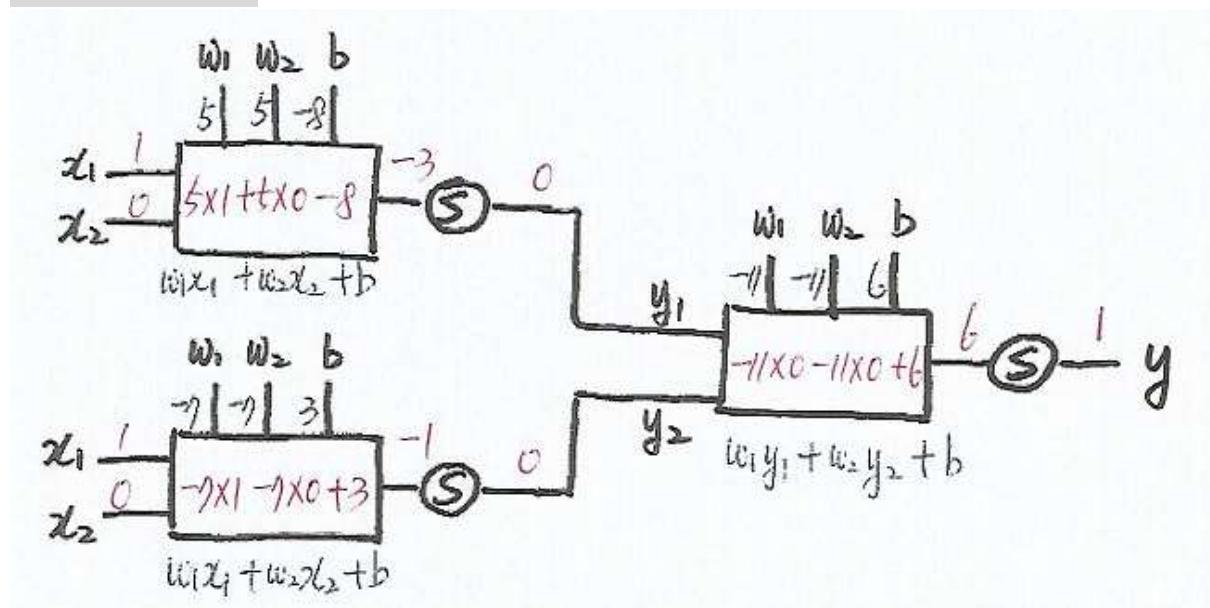
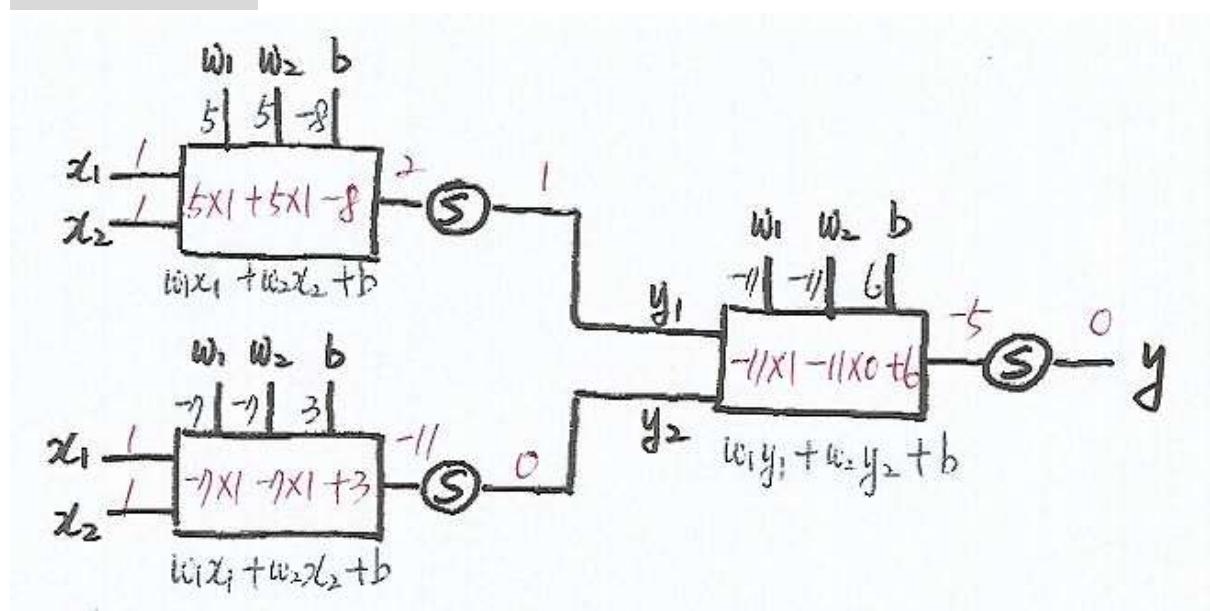
3.1.3 Neural Network

XOR 문제는 아래 그림과 같이 노드들을 연결하여 해결할 수 있다. 이것은 Neural Network을 구성하는 기본이 되므로 아래의 XOR 진리표를 참고하여 데이터가 계산되는 과정을 자세히 분석해 보자.

XOR 데이터 진리표

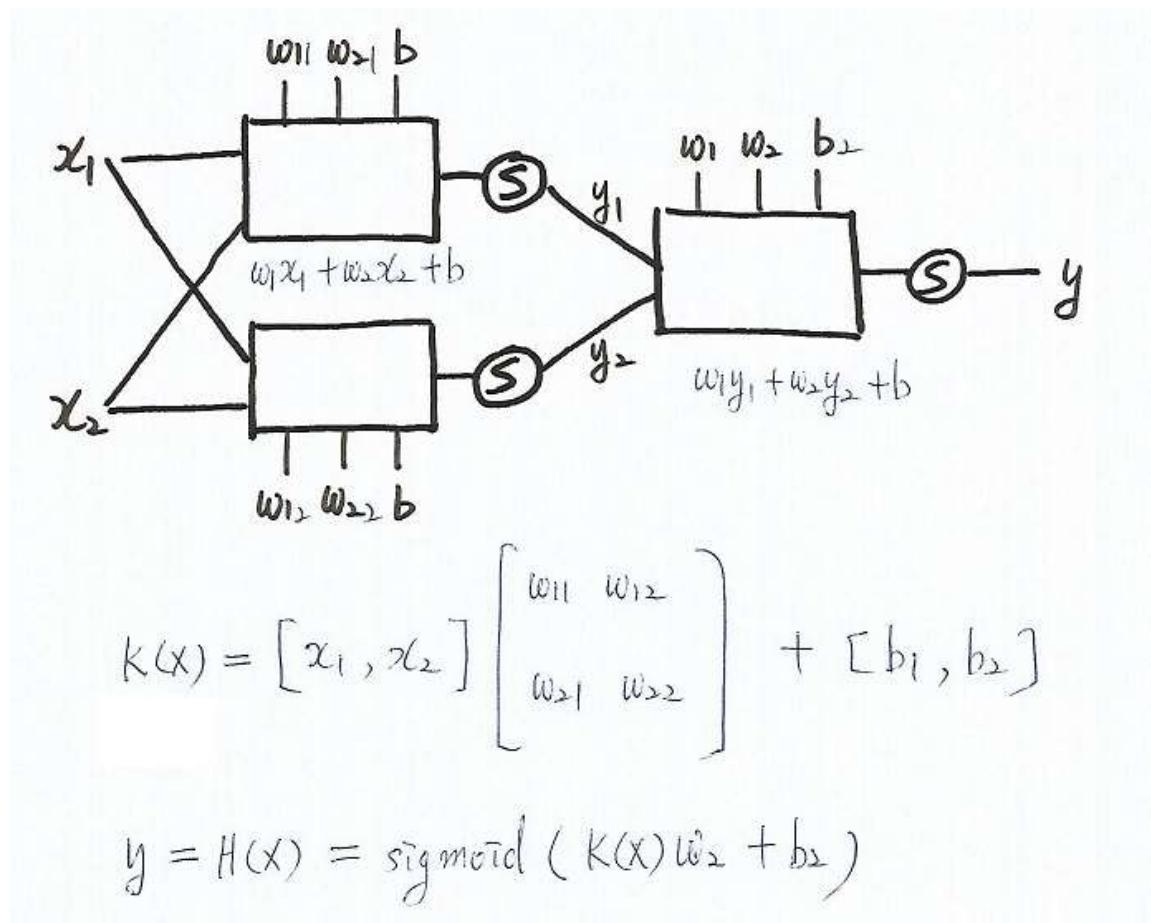
x1	x2	y1	y2	y	XOR
0	0	0	1	0	0
0	1	0	0	1	1
1	0	0	0	1	1
1	1	1	0	0	0

 $x_1=0, x_2=0$ 일때 $x_1=0, x_2=1$ 일때

x1=1, x2=0 일때**x1=1, x2=1 일때**

위와같이 XOR 문제는 Neural Network 을 다음과 같이 구성하여 해결할 수 있다.

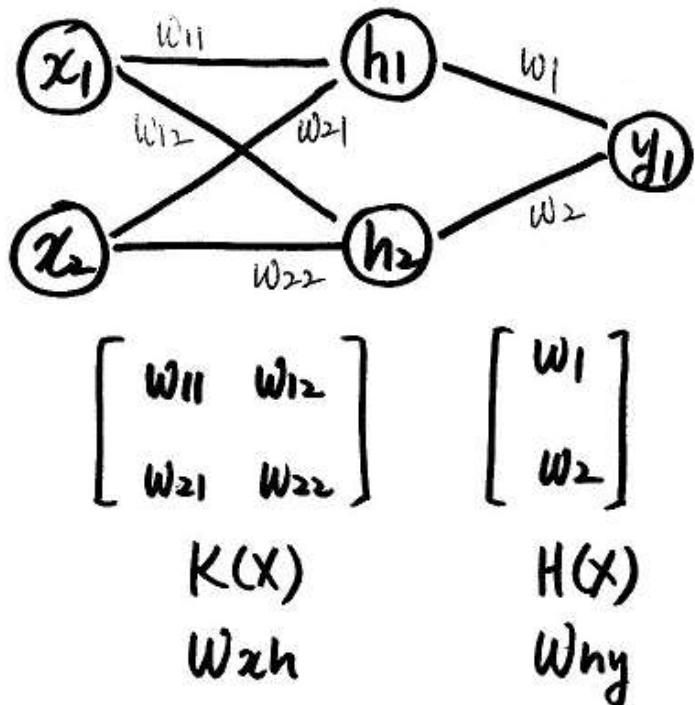
기본적인 Neural Network 블록 다이어그램



위에서 X로 주어진 입력값에 대해서 매트릭스 형태의 w 값을 곱하기 연산하고 b 값을 더하여 출력값인 Y가 산출된다. Neural Network는 입력값인 X에 대해서 정확한 Y값을 출력하기 위해서 w(weight)와 b(bias)값을 찾아갈 수 있도록 구성한다. 이것이 Neural Network를 구성하는 기본이 된다.

아래 그림은 위의 Neural Network를 좀더 간략히 표현하여 매트릭스 연산이 진행되는 과정을 설명하는 것이다.

Neural Network 매트릭스 연산



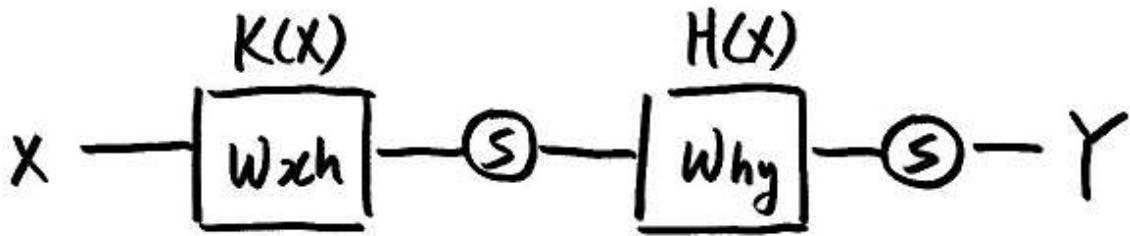
매트릭스 연산은 다음과 같이 진행된다.

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \stackrel{\textcircled{S}}{=} \begin{bmatrix} h_1 & h_2 \end{bmatrix}$$

$$\begin{bmatrix} h_1 & h_2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \stackrel{\textcircled{S}}{=} \begin{bmatrix} y_1 \end{bmatrix}$$

위의 Neural Network 을 좀더 단순화하여 블록 다이어그램으로 표현하면 다음과 같다.

Neural Network 블록 다이어그램



TensorFlow에서는 위의 Neural Network 을 다음과 같이 코딩한다.

TensorFlow에서 Neural Network 코딩1

```
X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

#Neural Network
W1 = tf.Variable(tf.random_uniform([2, 2], -1.0, 1.0))
W2 = tf.Variable(tf.random_uniform([2, 1], -1.0, 1.0))

b1 = tf.Variable(tf.zeros([2]), name="bias1")
b2 = tf.Variable(tf.zeros([1]), name="bias2")

#Neural Network hypothesis
H = tf.sigmoid(tf.matmul(X, W1) + b1)
hypothesis = tf.sigmoid(tf.matmul(H, W2) + b2)
```

TensorFlow에서 Neural Network 을 다음과 같이 코딩하면,

TensorFlow에서 Neural Network 코딩2

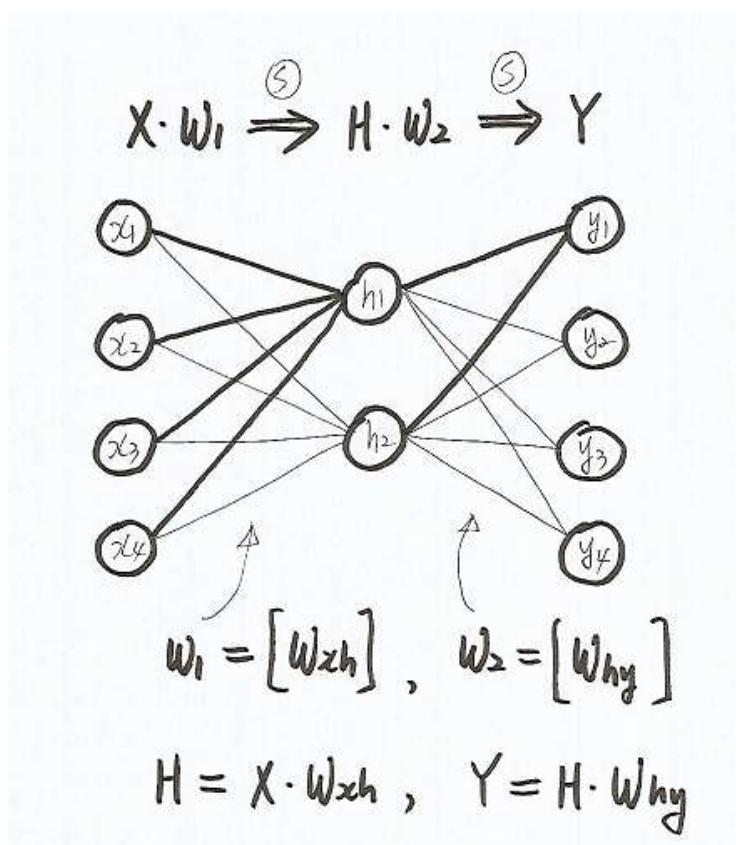
```
X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

#Neural Network
W1 = tf.Variable(tf.random_uniform([4, 2], -1.0, 1.0))
W2 = tf.Variable(tf.random_uniform([2, 4], -1.0, 1.0))

b1 = tf.Variable(tf.zeros([2]), name="bias1")
b2 = tf.Variable(tf.zeros([4]), name="bias2")

#Neural Network hypothesis
H = tf.sigmoid(tf.matmul(X, W1) + b1)
hypothesis = tf.sigmoid(tf.matmul(H, W2) + b2)
```

Matrix 행렬 연산은 다음과 같이 수행된다.



위에서 w_1 과 w_2 가 우리의 머신이 학습하는 목표이다. 이 Weigt 값은 입력과 출력 사이에서 다음과 같이 Matrix 연산된다.

$$\begin{array}{c}
 X \\
 \left[x_1 \ x_2 \ x_3 \ x_4 \right] \\
 \times \\
 \left[\begin{array}{cc} w_{xh} \\ w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{array} \right]
 \end{array}$$

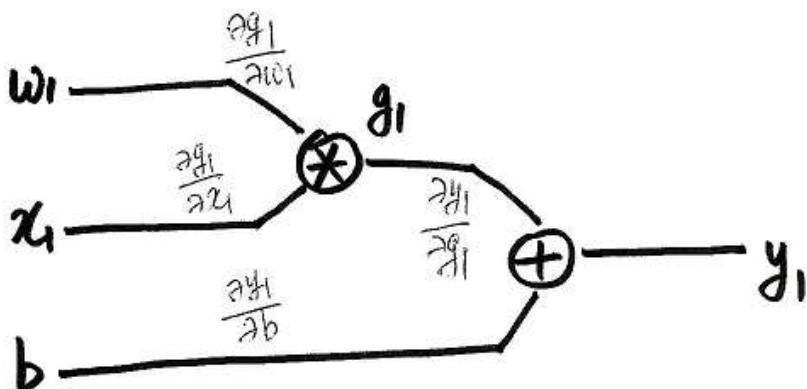
$$\Rightarrow H \xrightarrow{\textcircled{S}} \left[h_1 \ h_2 \right] \left[\begin{array}{cccc} w_{hy} \\ w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{array} \right] \xrightarrow{\textcircled{S}} Y \left[y_1 \ y_2 \ y_3 \ y_4 \right]$$

지금까지 Neural Network 을 구성하는 기본에 대해서 기술했으나, 이제는 Neural Network 내부의 w (weight)값과 b (bias)값을 어떻게 찾아가는지 알아 보도록 하자. 다음에 설명하는 Back Propagation 이 이 문제를 해결한다.

3.1.4 Back Propagation

아래 그림은 Back Propagation의 동작 원리를 설명하는 것이다. 입력값인 w_1, x_1, b 에 대해서 출력값 y_1 이 산출될 때 y_1 의 오류(cost)가 최소화 되도록, 출력에서 입력쪽(Back)으로 Chain Rule을 적용하여 w_1 과 b 를 조정해 나가는 방법으로 미분 연산을 반복수행(Propagation) 하면, 정답 y_1 으로 수렴해 갈 수 있다.

1단계 네트워크 예제



Back Propagation (Chain Rule)

$$y_1 = w_1 x_1 + b, \quad g_1 = w_1 x_1, \quad y_1 = g_1 + b$$

$$\frac{\partial g_1}{\partial w_1} = x_1, \quad \frac{\partial g_1}{\partial x_1} = w_1, \quad \frac{\partial y_1}{\partial g_1} = 1, \quad \frac{\partial y_1}{\partial b} = 1$$

$$\frac{\partial y_1}{\partial w_1} = \frac{\partial y_1}{\partial g_1} \cdot \frac{\partial g_1}{\partial w_1} = 1 \cdot x_1 = x_1$$

$$\frac{\partial y_1}{\partial x_1} = \frac{\partial y_1}{\partial g_1} \cdot \frac{\partial g_1}{\partial x_1} = 1 \cdot w_1 = w_1$$

위의 연산식에서, y_1 을 w_1 으로 미분을 하면 y_1 출력에 대해서 w_1 이 얼마만큼 영향을 준것인지 알 수 있고, 연산식을 보면 x_1 만큼 영향을 주었다. 그렇다면 아래의 간단한 데이터를 통해서 Back Propagation이 진행되는 과정을 이해해 보자.

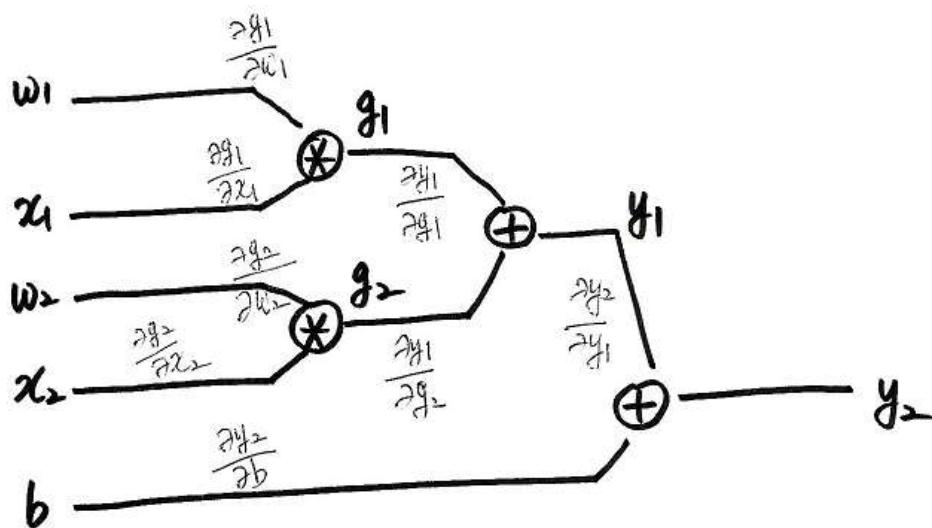
출력값인 y_1 은 정답으로 20이 되어야 한다고 가정하자. x_1 입력값은 2라고 하고, 초기에 w_1 값이 2이고 b 값이 1이라면, g_1 은 w_1 과 x_1 을 곱한 값이므로 4가 되고 y_1 은 g_1 에 b 를 더한 값이므로 5가 된다. 출력 y_1 은 20(정답)이 되어야 하는데 5가 출력 되었으므로 오차를 보정하기 위해서 Back Propagation을 진행한다. 보정하는 대상은 w_1 이고, 이 w_1 이 출력 y_1 에 영향을 준 것은 x_1 이다. (위의 미분식에 의해서) 그러므로 Back Propagation을 진행할 때마다 w_1 에 x_1 만큼 더하여 오류 보정해 가면, 아래와 같이 정답 $y_1=20$ 으로 수렴해 간다.

Back Propagation 연산 데이터 예제

반복순번	w_1	x_1	b	g_1	y_1
1	2	2	1	4	5
2	4	2	2	8	10
3	6	2	3	12	15
4	8	2	4	16	20

이번에는 아래 그림과 같이 네트워크를 2 단계로 구성해도 같은 방식의 Back Propagation(Chain Rule)을 적용할 수 있다.

2단계 네트워크 예제



Back Propagation (Chain Rule)

$$y_2 = w_1x_1 + w_2x_2 + b, \quad g_1 = w_1x_1, \quad g_2 = w_2x_2$$

$$y_1 = g_1 + g_2, \quad y_2 = y_1 + b$$

$$\frac{\partial y_2}{\partial y_1} = 1, \quad \frac{\partial y_2}{\partial b} = 1, \quad \frac{\partial y_1}{\partial g_1} = 1, \quad \frac{\partial y_1}{\partial g_2} = 1$$

$$\frac{\partial y_2}{\partial g_1} = \frac{\partial y_2}{\partial y_1} \cdot \frac{\partial y_1}{\partial g_1} = 1 \cdot 1 = 1$$

$$\frac{\partial y_2}{\partial g_2} = \frac{\partial y_2}{\partial y_1} \cdot \frac{\partial y_1}{\partial g_2} = 1 \cdot 1 = 1$$

$$\frac{\partial y_2}{\partial w_1} = \frac{\partial y_2}{\partial g_1} \cdot \frac{\partial g_1}{\partial w_1} = 1 \cdot x_1 = x_1$$

$$\frac{\partial y_2}{\partial w_2} = \frac{\partial y_2}{\partial g_2} \cdot \frac{\partial g_2}{\partial w_2} = 1 \cdot x_2 = x_2$$

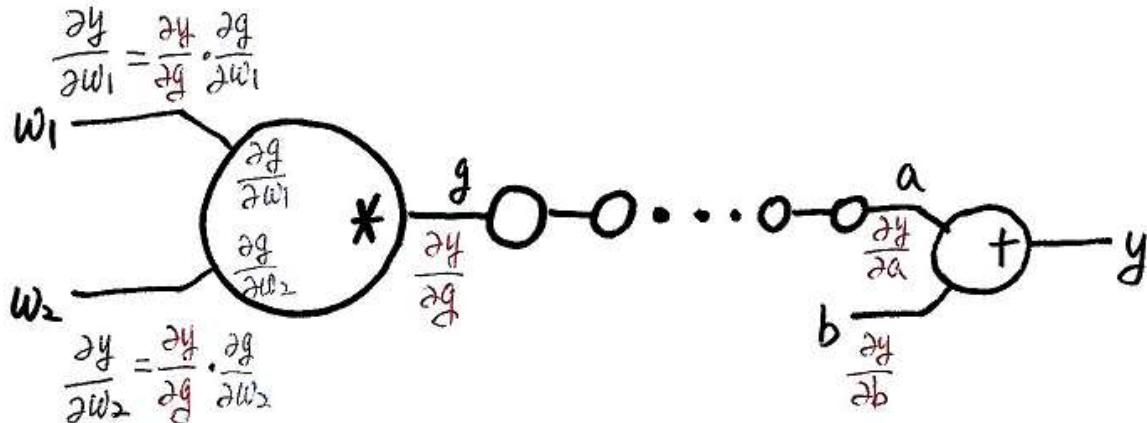
Back Propagation 연산 데이터 예제

반복순번	w1	x1	w2	x2	b	g1	g2	y1	y2
1	1	2	2	3	1	2	6	8	9
2	3	2	5	3	2	6	15	21	23
3	5	2	8	3	3	10	24	34	37
4	7	2	11	3	4	14	33	47	51

최종 출력인 y_2 의 목표치(정답)가 40이라고 한다면, Back Propagation(Chain Rule)을 적용하여 반복수행할 때, 순번 3에서 가장 오류(비용)가 최소화된다는 것을 알 수 있다.

위와 같이 Back Propagation은 네트워크를 다단계로 깊이 구성해도 동일한 방식으로 적용할 수 있다. 아래는 네트워크를 깊게 Deep Neural Network로 구성했을 때 Back Propagation이 그대로 적용된다는 것을 설명해 주는 것이다.

Deep Neural Network 예제

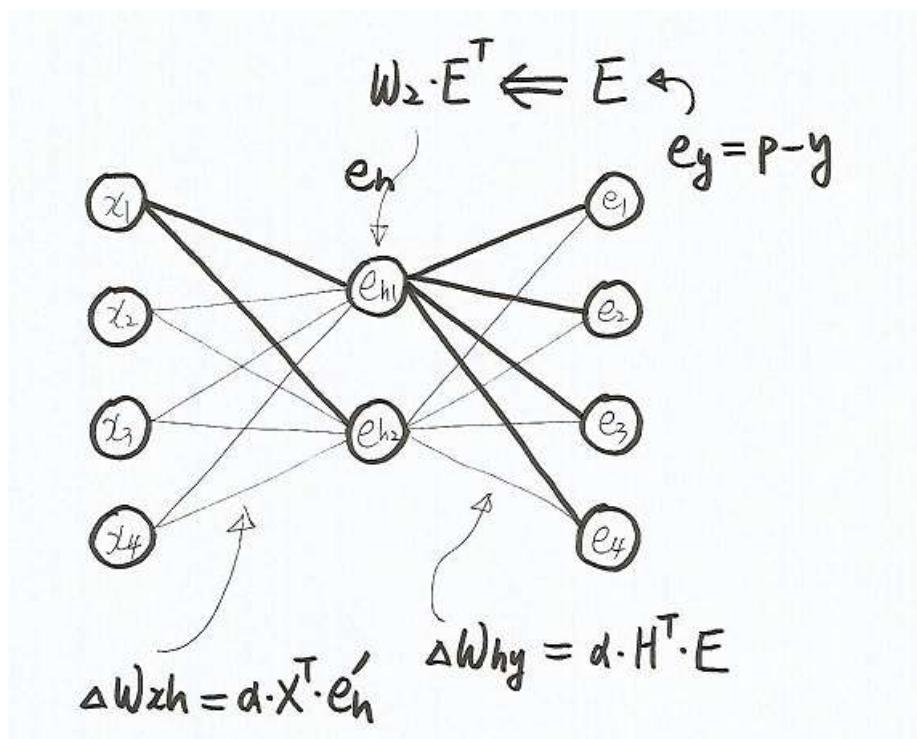


지금까지 Back Propogation(Chain Rule)을 적용하여 출력값에 대한 정답을 찾아가는 방법에 대해서 설명했다. 즉, Back Propogation 을 반복수행 할수록 출력값이 정답에 대한 오류가 최소화 되도록 w (weight)값과 b (bias)값을 조정해 나가는 방향으로 학습이 진행된다.

그런데, 머신러닝 알고리즘을 컴퓨터에 구현하고 테스트할때 TensorFlow 를 많이 사용 하고 있지만, TensorFlow 는 위의 Back Propogation 연산 과정이 라이브러리 안에 숨겨져 있기 때문에 자세히 분석하기 힘들다. 필자는 위의 Back Propogation 연산 과정을 자세히 분석하여 C 언어 소스로 코딩작업을 했다. 그 내용은 이 책의 제 2 부에서 기술된다.

참고로, 아래 그림은 Back Propogation 이 내부적으로 Matrix 연산되는 과정을 설명하는 것이다. 좀더 자세한 내용은 제 2 부에서 설명된다.

Back Propogation Matrix 연산 과정



$$H = X \cdot W_{xh}, \quad Y = H \cdot W_{yh}$$

$$\Delta W_{yh} = \alpha \cdot H^T \cdot e_y = \alpha \cdot \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} \begin{bmatrix} e_1 & e_2 & e_3 & e_4 \end{bmatrix}$$

$$e_h = W_{yh} \cdot e_y^T = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix}$$

$$e'_h = e_h \odot H' = [e'_h; e'_s] \odot [h'_1; h'_2]$$

$$\Delta W_{xh} = \alpha \cdot X^T \cdot e'_h$$

3.2 XOR 문제 해결 실습

머신러닝에서 난제중에 하나인 XOR 문제를 Neural Network로 풀어가는 방법에 대해서 이론적으로 설명했으나, 지금부터는 파이썬에서 TensorFlow를 사용하여 직접 실습해 보도록 하자.

먼저 아래와 같은 XOR 논리의 데이터를 파이썬에서 행렬로 읽는다.

XOR 입력 데이터

```
# XOR
# x1    x2      y
0      0      0
0      1      1
1      0      1
1      1      0
```

3.2.1 일반적인 XOR 문제

위의 데이터를 일반적인 Gradient Descent 알고리즘을 사용하여 실행 결과를 확인해 보자. 아직 Neural Network로 구성하지는 않았다.

일반적인 XOR 문제 예제

```
#XOR with Logistic Classification
#Editted by JungJaeJoon(rgb13307@nate.com) on the www.kernel.bz

import tensorflow as tf
import numpy as np

xy = np.loadtxt('xor_train.txt', unpack=True, dtype='float32')
x_data = xy[0:-1]
y_data = xy[-1]

print 'x', x_data
print 'y', y_data

X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

W = tf.Variable(tf.random_uniform([1, len(x_data)], -1.0, 1.0))

h = tf.matmul(W, X)      #matrix multiply
```

```

hypothesis = tf.div(1.0, 1.0 + tf.exp(-h))

#Cost Function
cost = -tf.reduce_mean(Y * tf.log(hypothesis) + (1-Y) * tf.log(1-hypothesis))

#Minimize
a = tf.Variable(0.1)      #Learning rate, alpha
optimizer = tf.train.GradientDescentOptimizer(a)
train = optimizer.minimize(cost)

init = tf.initialize_all_variables()

with tf.Session() as sess:
    sess.run(init)

    print 'Learning'
    for step in range(1, 1001):
        sess.run(train, feed_dict={X:x_data, Y:y_data})
        if step % 100 == 0:
            print step, sess.run(cost, feed_dict={X:x_data, Y:y_data}), sess.run(W)

    print 'Test Model'
    prediction = tf.equal(tf.floor(hypothesis+0.5), Y)
    #Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(prediction, "float"))
    print sess.run([hypothesis, tf.floor(hypothesis+0.5), prediction, accuracy],
feed_dict={X:x_data, Y:y_data} )
    print "Accuracy:", accuracy.eval({X:x_data, Y:y_data})

```

실행 결과

```

x [[ 0.  0.  1.  1.]
 [ 0.  1.  0.  1.]]
y [ 0.  1.  1.  0.]
Learning
100 0.695182 [[ 0.05906953 -0.20268884]]
200 0.693476 [[ 0.05912896 -0.08080875]]
300 0.693236 [[ 0.03575107 -0.03901689]]
400 0.693171 [[ 0.01972787 -0.02021541]]
500 0.693153 [[ 0.01063846 -0.0107018 ]]
600 0.693149 [[ 0.00569867 -0.00569848]]
700 0.693147 [[ 0.00304735 -0.00304721]]
800 0.693147 [[ 0.00162732 -0.00162713]]
900 0.693147 [[ 0.00086757 -0.00086745]]
1000 0.693147 [[ 0.00046172 -0.0004616 ]]

Test Model
[array([[ 0.49999809,  0.49988317,  0.50011539,  0.5       ]], dtype=float32),
 array([[ 0.,  0.,  1.,  1.]], dtype=float32),
 array([[ True, False,  True, False]], dtype=bool), 0.5]
Accuracy: 0.5

```

위의 실행결과를 확인해 보면, 정확도(Accuracy)가 0.5(50%)로 산출됨을 알 수 있다. XOR 문제를 50% 정도만 맞출 수 있다는 것이다.

3.2.2 XOR Neural Network

위의 XOR 문제를 이제는 파이썬에서 TensorFlow 를 사용하여 Neural Network 로 구성하여 실행해 보도록 하자.

XOR Neural Network 예제

```
#XOR Neural Network with Logistic Classification
#Editted by JungJaeJoon(rgb13307@nate.com) on the www.kernel.bz

import tensorflow as tf
import numpy as np

xy = np.loadtxt('xor_train.txt', unpack=True, dtype='float32')

#xy.transpose()

x_data = xy[0:-1]
y_data = xy[-1]

print 'x', x_data
print 'y', y_data

X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

#W = tf.Variable(tf.random_uniform([1, len(x_data)], -1.0, 1.0))
#Neural Network
W1 = tf.Variable(tf.random_uniform([4, 2], -1.0, 1.0))
W2 = tf.Variable(tf.random_uniform([2, 4], -1.0, 1.0))

b1 = tf.Variable(tf.zeros([2]), name="bias1")
b2 = tf.Variable(tf.zeros([4]), name="bias2")

#h = tf.matmul(W, X)      #matrix multiply
#hypothesis = tf.div(1.0, 1.0 + tf.exp(-h))
#Neural Network hypothesis
L2 = tf.sigmoid(tf.matmul(X, W1) + b1)
hypothesis = tf.sigmoid(tf.matmul(L2, W2) + b2)

#Cost Function
cost = -tf.reduce_mean(Y * tf.log(hypothesis) + (1-Y) * tf.log(1-hypothesis))

#Minimize
a = tf.Variable(0.1)    #Learning rate, alpha
optimizer = tf.train.GradientDescentOptimizer(a)
train = optimizer.minimize(cost)
```

```

init = tf.initialize_all_variables()

with tf.Session() as sess:
    sess.run(init)

    print 'Learning'
    for step in range(1, 1001):
        sess.run(train, feed_dict={X:x_data, Y:y_data})
        if step % 100 == 0:
            print step, sess.run(cost, feed_dict={X:x_data, Y:y_data}), sess.run(W1), sess.run(W2)

    print 'Test Model'
    prediction = tf.equal(tf.floor(hypothesis+0.5), Y)
    #Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(prediction, "float"))
    print sess.run([hypothesis, tf.floor(hypothesis+0.5), prediction, accuracy],
feed_dict={X:x_data, Y:y_data} )
    print "Accuracy:", accuracy.eval({X:x_data, Y:y_data})
```

실행 결과

```

x [[ 0.  0.  1.  1.]
 [ 0.  1.  0.  1.]]
y [ 0.  1.  1.  0.]
Learning
100 0.25095 [[ 0.03660297  0.01244998]
 [ 0.49042973 -0.04401221]
 [-0.08756048 -0.2778239 ]
 [-0.04936086  0.47328007]] [[-0.94034404  0.08567502  0.87725097 -0.66278332]
 [ 0.27895248  0.03625461 -0.41631523 -0.23795699]]
200 0.122168 [[ 0.03660297  0.01244998]
 [ 0.59755951 -0.04441736]
 [ 0.05649288 -0.27860844]
 [ 0.20182233  0.47209036]] [[-1.19569445  0.39566734  1.14289808 -0.91848022]
 [ 0.09379384  0.26079082 -0.22369429 -0.42326099]]
300 0.0761536 [[ 0.03660297  0.01244998]
 [ 0.65850848 -0.02743955]
 [ 0.14667243 -0.26011112]
 [ 0.35295078  0.50756538]] [[-1.36417925  0.59711844  1.3168354 -1.08946252]
 [-0.01563703  0.39164406 -0.11071981 -0.53431273]]
400 0.0541189 [[ 0.03660297  0.01244998]
 [ 0.69756305 -0.00887111]
 [ 0.20638539 -0.2395222 ]
 [ 0.45171857  0.54672247]] [[-1.48663139  0.73970228  1.44275737 -1.21333146]
 [-0.09399148  0.48289567 -0.03014046 -0.61356854]]
500 0.0415177 [[ 0.03660297  0.01244998]
 [ 0.72517455  0.0086433 ]
 [ 0.24913427 -0.21987627]
 [ 0.52207875  0.58388239]] [[-1.58179247  0.84790009  1.54039097 -1.30896628]
 [-0.15554847  0.55289596  0.03302094 -0.67542529]]
600 0.0334579 [[ 0.03660297  0.01244998]
 [ 0.74605018  0.02465449]
 [ 0.28161857 -0.20175146]]
```

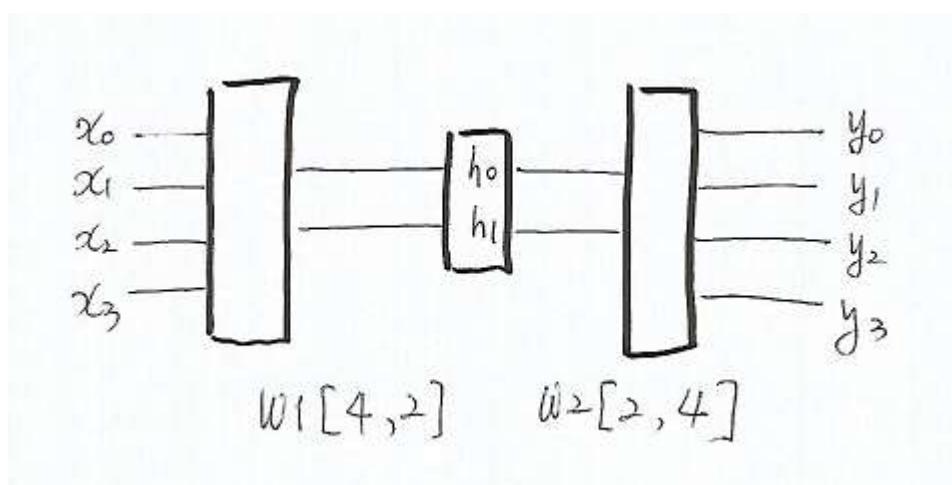
```
[ 0.57543892  0.61801851] ] [[-1.65916491  0.93416822  1.61965823 -1.3862108 ]
 [-0.20651552  0.60972989  0.08524041 -0.7263006 ]]
700 0.0278936 [[ 0.03660297  0.01244998]
 [ 0.76258999  0.03919911]
 [ 0.30740216 -0.18516079]
 [ 0.61776233  0.64915359]] ] [[-1.72411072  1.00543785  1.68612635 -1.45066547]
 [-0.25014359  0.65761006  0.12989475 -0.76959169]]
800 0.0238425 [[ 0.03660297  0.01244998]
 [ 0.77614778  0.05242574]
 [ 0.32854319 -0.16997536]
 [ 0.65246117  0.67756575]] ] [[-1.77992141  1.06589139  1.74320304 -1.50577044]
 [-0.28835517  0.69900262  0.16897608 -0.80731392]]
900 0.0207702 [[ 0.03660297  0.01244998]
 [ 0.78755051  0.06449608]
 [ 0.34631464 -0.1560401 ]
 [ 0.68163514  0.7035712 ]] ] [[-1.82875407  1.11821735  1.79311335 -1.55377412]
 [-0.32238805  0.73547119  0.20376223 -0.84076345]]
1000 0.0183669 [[ 0.03660297  0.01244998]
 [ 0.79733527  0.07555865]
 [ 0.36154911 -0.14320733]
 [ 0.70665431  0.72746652]] ] [[-1.87209404  1.16423798  1.83738911 -1.59621966]
 [-0.35308921  0.76807183  0.23512857 -0.87082654]]
```

Test Model

```
[array([[ 0.01866032,  0.98030055,  0.98100561,  0.01838661],
       [ 0.01723054,  0.98174787,  0.98234093,  0.01670825]], dtype=float32),
array([[ 0.,  1.,  1.,  0.],
       [ 0.,  1.,  1.,  0.]], dtype=float32),
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool), 1.0]
```

Accuracy: 1.0

위의 실행결과를 확인해 보면, Accuracy 가 1.0 즉 100% 모두 올바르게 학습하여 결과를 예측함을 알 수 있다. 따라서 XOR 문제를 TensorFlow 에서 Neural Network 로 구성하여 해결한다는 것을 증명하고 있다. 아래 그림은 위의 Neural Network 연결을 이해하기 쉽도록 블록도로 설명하는 것이다.



3.2.3 XOR Deep Learning

이제는 Neural Network 연결을 좀더 많이 해보도록 하자. 연결을 많이, 깊게 한다는 것은 Deep Learning 을 점차적으로 구현해 간다는 것이다.

XOR Deep Learning 예제

```
#XOR Neural Network with Logistic Classification
#Edited by JungJaeJoon(rgb13307@nate.com) on the www.kernel.bz

import tensorflow as tf
import numpy as np

xy = np.loadtxt('xor_train.txt', unpack=True, dtype='float32')

#xy.transpose()

x_data = xy[0:-1]
y_data = xy[-1]

print 'x', x_data
print 'y', y_data

X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

#W = tf.Variable(tf.random_uniform([1, len(x_data)], -1.0, 1.0))
#Neural Network
W1 = tf.Variable(tf.random_uniform([4, 8], -1.0, 1.0))
W2 = tf.Variable(tf.random_uniform([8, 2], -1.0, 1.0))
W3 = tf.Variable(tf.random_uniform([2, 6], -1.0, 1.0))
W4 = tf.Variable(tf.random_uniform([6, 4], -1.0, 1.0))

b1 = tf.Variable(tf.zeros([8]), name="bias1")
b2 = tf.Variable(tf.zeros([2]), name="bias2")
b3 = tf.Variable(tf.zeros([6]), name="bias3")
b4 = tf.Variable(tf.zeros([4]), name="bias4")

#h = tf.matmul(W, X)      #matrix multiply
#hypothesis = tf.div(1.0, 1.0 + tf.exp(-h))
#Neural Network hypothesis
L2 = tf.sigmoid(tf.matmul(X, W1) + b1)
L3 = tf.sigmoid(tf.matmul(L2, W2) + b2)
L4 = tf.sigmoid(tf.matmul(L3, W3) + b3)
hypothesis = tf.sigmoid(tf.matmul(L4, W4) + b4)

#Cost Function
cost = -tf.reduce_mean(Y * tf.log(hypothesis) + (1-Y) * tf.log(1-hypothesis))

#Minimize
a = tf.Variable(0.1)    #Learning rate, alpha
```

```

optimizer = tf.train.GradientDescentOptimizer(a)
train = optimizer.minimize(cost)

init = tf.initialize_all_variables()

with tf.Session() as sess:
    sess.run(init)

    print 'Learning'
    for step in range(1, 1001):
        sess.run(train, feed_dict={X:x_data, Y:y_data})
        if step % 100 == 0:
            print step, sess.run(cost, feed_dict={X:x_data, Y:y_data}), sess.run(W1), sess.run(W2),
            sess.run(W3), sess.run(W4)

        print 'Test Model'
        prediction = tf.equal(tf.floor(hypothesis+0.5), Y)
        #Calculate accuracy
        accuracy = tf.reduce_mean(tf.cast(prediction, "float"))
        print sess.run([hypothesis, tf.floor(hypothesis+0.5), prediction, accuracy],
        feed_dict={X:x_data, Y:y_data} )
        print "Accuracy:", accuracy.eval({X:x_data, Y:y_data})
    
```

실행 결과

```

x [[ 0.  0.  1.  1.]
 [ 0.  1.  0.  1.]]
y [ 0.  1.  1.  0.]
Learning
100 0.114419 [[-0.53696656  0.09492016 -0.46433854  0.33826256  0.45726037 -0.54607201
 0.47275686 -0.31685877]
 [ 0.30497411  0.23785822  0.89911723  0.95741385  0.00513542 -0.40189067
 -0.25889775 -0.68945903]
 [ 0.57287711 -0.75749326  0.27753583 -0.15163743  0.3686246   0.07325713
 0.98045665  0.44173077]
 [ 0.7311554 -0.40420601  0.43343782 -0.46672651 -0.39110404  0.4342263
 -0.63856757  0.66231889]] [[-1.01391912 -0.2047261 ]
 [ 0.72797835 -0.79294288]
 [-0.03056335  0.22873099]
 [-0.1082243   0.13853502]
 [ 0.53236443 -0.576958 ]
 [ 0.74733788  0.45275301]
 [ 0.0377837   0.82423723]
 [-0.11081205 -0.15412927]] [[-0.31446156 -0.3012099   0.02435048 -0.15778133  0.33663949 -
 0.17149268]
 [-0.4433651 -0.36458912  0.80303204  0.75777429 -0.24388967  0.75733691]] [[-0.93854433
0.54819727  0.65918219 -0.82576787]
 [ 0.60507047  0.94005984  0.13484281 -0.01735144]
 [-0.73367298  0.44321746  1.14395416 -0.04057798]
 [-1.30177617  1.06318212  1.22790396  0.36861777]
 [ 0.20369822  0.59830463 -0.53798568 -1.16561735]
 [-0.08807652  0.85622299 -0.52008486 -0.94112593]]
200 0.0568369 [[-0.53696656  0.09492016 -0.46433854  0.33826256  0.45726037 -0.54607201
0.47275686 -0.31685877]]
```

```
[ 0.30541489  0.23244627  0.90003717  0.95834851  0.00134086 -0.4005138
-0.25514477 -0.69005382]
[ 0.57328069 -0.76112247  0.27866873 -0.15079862  0.36495364  0.07438917
 0.98447382  0.44132817]
[ 0.73199999 -0.41324705  0.43549082 -0.4649528 -0.39856952  0.43673545
-0.63079798  0.66132128]] [[[-1.02340364 -0.17258033]
[ 0.72369808 -0.77831805]
[-0.03967149  0.25969306]
[-0.11429631  0.15928051]
[ 0.5268563 -0.55833548]
[ 0.74031299  0.47652099]
[ 0.03233069  0.8425836 ]
[-0.11851063 -0.1281589 ]] [[[-0.23909928 -0.29426244  0.07813801 -0.09112109  0.36308205 -
0.14030971]
[-0.35773847 -0.35661045  0.86407363  0.83339429 -0.21373869  0.79284483]] [[[-1.04082358
0.61218536  0.76898909 -0.93991488]
[ 0.52427161  0.99052131  0.22159897 -0.1075173 ]
[-0.87013549  0.5285297   1.29047358 -0.19290283]
[-1.4373771   1.14797604  1.37349582  0.21725273]
[ 0.09862386  0.66395676 -0.42516729 -1.28287768]
[-0.20932916  0.93201071 -0.38989389 -1.07647514]]
300 0.0362762 [[[-0.53696656  0.09492016 -0.46433854  0.33826256  0.45726037 -0.54607201
0.47275686 -0.31685877]
[ 0.30532482  0.22904028  0.90075845  0.95902717 -0.00101515 -0.39896345
-0.25233665 -0.69046587]
[ 0.57323498 -0.76331747  0.27953163 -0.15020779  0.36274064  0.07565936
 0.98736411  0.44105592]
[ 0.73186415 -0.4188481   0.43707475 -0.4636834 -0.40313861  0.43955585
-0.62509948  0.66063684]] [[[-1.0264653 -0.14963904]
[ 0.72233135 -0.76794761]
[-0.0426152   0.28184903]
[-0.11626117  0.17417648]
[ 0.52509409 -0.54517084]
[ 0.73803973  0.49351323]
[ 0.0305574   0.8557294 ]
[-0.12099152 -0.10969009]] [[[-0.19282588 -0.28600627  0.11110616 -0.05127419  0.38289705 -
0.11762097]
[-0.30097076 -0.34646663  0.90446079  0.88219637 -0.18940043  0.82068187]] [[[-1.10383081
0.65759581  0.83598036 -1.01016593]
[ 0.4782339   1.02368772  0.2705487 -0.1588486 ]
[-0.9512198   0.58695966  1.37668884 -0.2833285 ]
[-1.51875412  1.20661902  1.46002162  0.1265016 ]
[ 0.03745786  0.70802736 -0.36013234 -1.35107803]
[-0.28089884  0.98358399 -0.31379485 -1.15629232]]
400 0.0261393 [[[-0.53696656  0.09492016 -0.46433854  0.33826256  0.45726037 -0.54607201
0.47275686 -0.31685877]
[ 0.30509543  0.22663614  0.90134519  0.95955902 -0.00265996 -0.39752901
-0.25012586 -0.69076478]
[ 0.57306832 -0.76482141  0.28021953 -0.149755   0.36123121  0.07682398
 0.98957801  0.44086096]
[ 0.7314682   -0.42275614  0.43834919 -0.46269876 -0.40629283  0.44215494
-0.62067479  0.66014326]] [[[-1.02724624 -0.13214555]
[ 0.72198504 -0.76006967]
[-0.04336667  0.29878163]
[-0.11676324  0.18558718]
[ 0.52464736 -0.53519166]
```

```
[ 0.73745859  0.5065015 ]
[ 0.03010273  0.86578888]
[-0.12162349 -0.09563771]] [[-0.16010705 -0.27862167  0.13426939 -0.02371282  0.39825791 -
0.10036441]
[-0.25893217 -0.3369745   0.93417686  0.917548   -0.16965011  0.84284484]] [[-1.1488992
0.69247633  0.88367081 -1.05995119]
[ 0.44667599  1.04810858  0.30394271 -0.1937087 ]
[-1.00795472  0.63086736  1.43672633 -0.34601092]
[-1.57595062  1.25088477  1.52054763  0.06330861]
[-0.00502636  0.74090457 -0.31517622 -1.39800858]
[-0.33096328  1.02233028 -0.2608155  -1.21160734]]
500 0.0202171 [[-0.53696656  0.09492016 -0.46433854  0.33826256  0.45726037 -0.54607201
0.47275686 -0.31685877]
[ 0.30482972  0.22481655  0.90183824  0.95999616 -0.00389316 -0.3962408
-0.24831381 -0.6909914 ]
[ 0.57286769 -0.76593274  0.28078863 -0.14938933  0.36012122  0.07786165
0.99135482  0.44071427]
[ 0.73100162 -0.42568737  0.43941128 -0.46189588 -0.40863612  0.44448087
-0.61708611  0.65977001]] [[-1.02704108 -0.11814729]
[ 0.72207522 -0.75378168]
[-0.0431689  0.31235626]
[-0.11663112  0.19475083]
[ 0.52476418 -0.52723908]
[ 0.73761195  0.51691961]
[ 0.0302232  0.87386334]
[-0.12145734 -0.08440979]] [[-0.13506538 -0.27216446  0.15188514 -0.00299262  0.41071388 -
0.08658511]
[-0.2257406 -0.32841456  0.95748979  0.94496435 -0.15313195  0.86109787]] [[-1.18382752
0.72067875  0.92052287 -1.09815395]
[ 0.42287731  1.06732345  0.32905164 -0.21973756]
[-1.05124342  0.66582012  1.48239934 -0.39336386]
[-1.6196996  1.28620982  1.56670702  0.01545173]
[-0.03736017  0.76701057 -0.28106207 -1.43337286]
[-0.36921659  1.05321741 -0.22045465 -1.25345314]]
600 0.0163763 [[-0.53696656  0.09492016 -0.46433854  0.33826256  0.45726037 -0.54607201
0.47275686 -0.31685877]
[ 0.30456084  0.22337399  0.90226275  0.9603672  -0.00486261 -0.3950851
-0.246784  -0.69116932]
[ 0.57265961 -0.76679659  0.28127289 -0.14908332  0.35926297  0.07878653
0.9928301  0.4405995 ]
[ 0.73052633 -0.42799354  0.44031999 -0.46121892 -0.41046375  0.44656143
-0.61408097  0.65947729]] [[-1.02636206 -0.10655351]
[ 0.72237349 -0.74858332]
[-0.04251363  0.32361722]
[-0.11619306  0.20236279]
[ 0.52515048 -0.52067286]
[ 0.73812062  0.52556753]
[ 0.03062299  0.88056862]
[-0.12090758 -0.07512055]] [[-0.11491321 -0.26647636  0.1659815   0.01343793  0.42116466 -
0.07517479]
[-0.19842395 -0.32070398  0.97656846  0.96719819 -0.13895996  0.87655401]] [[-1.21227694
0.74429888  0.95047456 -1.12898469]
[ 0.4038606  1.08311105  0.34907231 -0.24034539]
[-1.08608305  0.69474548  1.51907897 -0.4311237 ]
[-1.65496278  1.31548643  1.60383296 -0.02276724]
[-0.06337412  0.7886079  -0.25367448 -1.4615643 ]]
```

```

[-0.40006143  1.07882643 -0.18798019 -1.28688478]]
700 0.0137032 [[-0.53696656  0.09492016 -0.46433854  0.33826256  0.45726037 -0.54607201
  0.47275686 -0.31685877]
[ 0.30430079  0.22219218  0.90263516  0.96068937 -0.00565088 -0.3940424
 -0.24546355 -0.69131291]
[ 0.57245809 -0.76749218  0.28169349 -0.14882071  0.35857531  0.07961654
  0.99408615  0.44050708]
[ 0.73006427 -0.42987093  0.44111311 -0.46063417 -0.41193953  0.4484342
 -0.61150444  0.65924114]] [[-1.02544332 -0.0967032 ]
[ 0.72277606 -0.74417281]
[-0.04162624  0.33319777]
[-0.11559965  0.20884594]
[ 0.52567172 -0.51510757]
[ 0.7388103   0.53293002]
[ 0.03116575  0.88627934]
[-0.12016387 -0.06723484]] [[-0.09812602 -0.26140651  0.1776672   0.02695738  0.43015766 -
0.06546785]
[-0.17527971 -0.31371459  0.9926545   0.98580521 -0.12655701  0.88992697]] [[-1.23624456
0.76458901  0.97566533 -1.1547426 ]
[ 0.38806549  1.09648287  0.3656739   -0.25732034]
[-1.11515117  0.7193554   1.54963267 -0.46236819]
[-1.68441308  1.34041977  1.63478816 -0.0544223 ]
[-0.08509598  0.80699742 -0.23084308 -1.48490989]
[-0.42584813  1.10065806 -0.16087529 -1.31460321]]
800 0.0117436 [[-0.53696656  0.09492016 -0.46433854  0.33826256  0.45726037 -0.54607201
  0.47275686 -0.31685877]
[ 0.30405325  0.2211998   0.90296668  0.96097374 -0.00630816 -0.39309484
 -0.24430418 -0.69143134]
[ 0.57226443 -0.76806742  0.28206494 -0.14859107  0.35800949  0.08036731
  0.99517661  0.44043082]
[ 0.72962314 -0.43143839  0.44181606 -0.46011999 -0.41316253  0.4501327
 -0.60925442  0.65904647]] [[-1.0244031  -0.08816957]
[ 0.72323084 -0.74035579]
[-0.04062074  0.3415077 ]
[-0.11492706  0.21447456]
[ 0.52626121 -0.51029509]
[ 0.73959285  0.53932047]
[ 0.03178218  0.89123702]
[-0.11932183 -0.06040766]] [[-0.08378591 -0.25683704  0.18760766  0.03838568  0.4380472  -
0.05703844]
[-0.15524554 -0.30733094  1.00652051  1.00174415 -0.11553163  0.90169406]] [[-1.2569319
0.78235638  0.99737728 -1.17681122]
[ 0.37457946  1.10806525  0.37982789 -0.27170676]
[-1.14004183  0.74073333  1.5757575  -0.48892432]
[-1.70964777  1.36209297  1.661273   -0.08134495]
[-0.10372196  0.82299417 -0.2112945  -1.50477993]
[-0.44797212  1.11965978 -0.13765462 -1.33820784]]
900 0.0102512 [[-0.53696656  0.09492016 -0.46433854  0.33826256  0.45726037 -0.54607201
  0.47275686 -0.31685877]
[ 0.30381912  0.22035046  0.90326524  0.96122849 -0.00686709 -0.39222786
 -0.24327207 -0.69153082]
[ 0.57208049 -0.76855308  0.28239715 -0.14838718  0.35753423  0.08105176
  0.99613756  0.44036686]
[ 0.72920489 -0.43277347  0.44244692 -0.45966157 -0.41419673  0.45168421
 -0.60726142  0.65888309]] [[-1.02330542 -0.08066157]
[ 0.72370964 -0.73700029]]

```

```

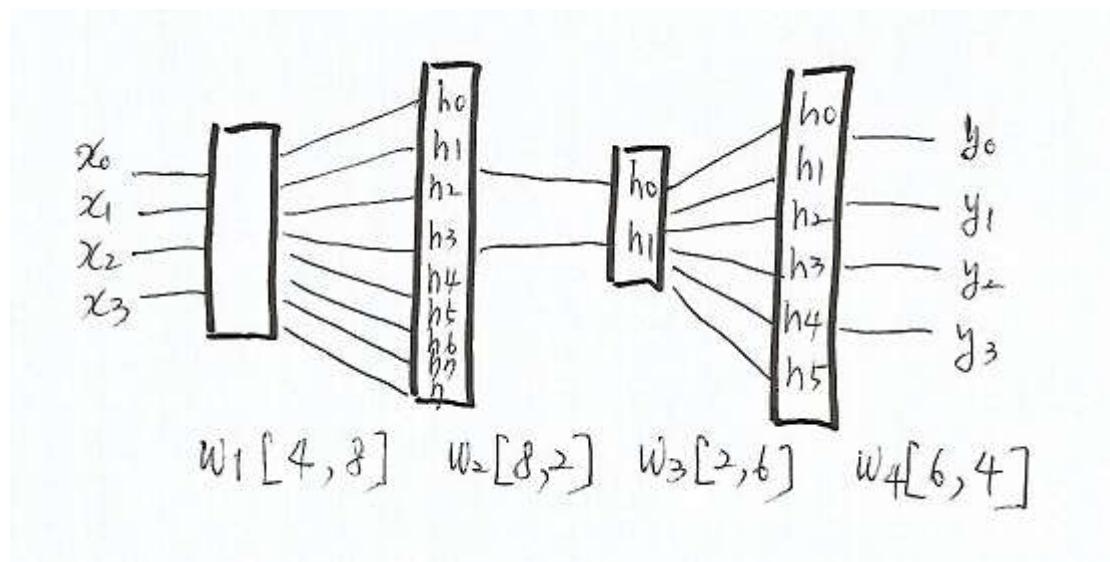
[-0.03955868  0.34882665]
[-0.11421649  0.21943586]
[ 0.52688235 -0.50606787]
[ 0.74042046  0.54495275]
[ 0.03243463  0.89560717]
[-0.11843303 -0.05440411]] [[-0.0712989 -0.25267777  0.19623266  0.048248   0.44507265 -
0.04960033]
[-0.13761424 -0.30145827  1.01868021  1.01564634 -0.10560909  0.91218787]] [[-1.27511907
0.79814643  1.01644564 -1.1960845 ]
[ 0.36282456  1.11827171  0.39215219 -0.28416309]
[-1.1617763   0.75960332  1.59854531 -0.51195896]
[-1.73169255  1.38123167  1.68438637 -0.10470781]
[-0.12001392  0.83713824 -0.19421332 -1.52204382]
[-0.46732673  1.13646364 -0.1173618 -1.35871994]]
1000 0.00908017 [[-0.53696656  0.09492016 -0.46433854  0.33826256  0.45726037 -0.54607201
0.47275686 -0.31685877]
[ 0.30359814  0.21961239  0.90353686  0.96145868 -0.00734985 -0.39142942
-0.2423432 -0.69161528]
[ 0.57190543 -0.76896983  0.2826975 -0.1482041   0.35712838  0.08168004
0.99699491  0.44031236]
[ 0.72880906 -0.43392828  0.44301876 -0.45924816 -0.41508541  0.45311099
-0.60547507  0.65874386]] [[-1.02218568 -0.07397336]
[ 0.72419804 -0.73401326]
[-0.03847487  0.35535261]
[-0.11349128  0.22386263]
[ 0.52751416 -0.50230736]
[ 0.74126542  0.54997814]
[ 0.03310166  0.89950693]
[-0.11752655 -0.04905843]] [[-0.06026125 -0.24885917  0.20383254  0.05689743  0.45140514 -
0.04295104]
[-0.1218936 -0.29601991  1.02948749  1.02794504 -0.09658732  0.92165017]] [[-1.29133952
0.81235194  1.03343058 -1.21317029]
[ 0.35241324  1.12738955  0.40305409 -0.29512945]
[-1.18104744  0.77648008  1.61872447 -0.53225899]
[-1.75124335  1.39835441  1.70485926 -0.12530354]
[-0.13448566  0.84981257 -0.17905958 -1.53728795]
[-0.48451659  1.15151858 -0.0993612 -1.37682855]]
Test Model
[array([[ 0.00892645,  0.99212331,  0.99063337,  0.00946936],
       [ 0.00925983,  0.99188703,  0.99033308,  0.00963214]], dtype=float32), array([[ 0.,  1.,  1.,
0.],
[ 0.,  1.,  1.,  0.]], dtype=float32), array([[ True,  True,  True,  True],
[ True,  True,  True,  True]], dtype=bool), 1.0]
Accuracy: 1.0

```

위의 예제는 출력데이터가 많다. 이들은 모두 W(Weight) 행렬(Matrix)을 print 명령으로 출력한 것이다.

```
print step, sess.run(cost, feed_dict={X:x_data, Y:y_data}), sess.run(W1), sess.run(W2), sess.run(W3),
sess.run(W4)
```

위의 값들은 다음과 같은 블로도로 행렬을 표시할 수 있다.



Deep Neural Network은 결국 위의 행렬(Matrix)안에 있는 Weight 값을 학습하는 방향으로 알고리즘이 진행된다.

3.2.4 XOR Deep Learning2

이제는 Neural Network 연결을 많이 해보도록 하자. 연결을 더 많이, 더 깊게 한다는 것은 Deep Learning 을 더 많이 한다는 것이고 학습하는 w(weight)와 b(bias)의 개수 또한 증가한다.

XOR Deep Learning 예제2

```
#XOR Neural Network with Logistic Classification
#Editted by JungJaeJoon(rgb13307@nate.com) on the www.kernel.bz

import tensorflow as tf
import numpy as np

xy = np.loadtxt('xor_train.txt', unpack=True, dtype='float32')

#xy.transpose()

x_data = xy[0:-1]
y_data = xy[-1]

print 'x', x_data
print 'y', y_data

X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

#W = tf.Variable(tf.random_uniform([1, len(x_data)], -1.0, 1.0))
#Deep Neural Network
W1 = tf.Variable(tf.random_uniform([4, 8], -1.0, 1.0))
W2 = tf.Variable(tf.random_uniform([8, 2], -1.0, 1.0))
W3 = tf.Variable(tf.random_uniform([2, 6], -1.0, 1.0))
W4 = tf.Variable(tf.random_uniform([6, 4], -1.0, 1.0))
W5 = tf.Variable(tf.random_uniform([4, 8], -1.0, 1.0))
W6 = tf.Variable(tf.random_uniform([8, 2], -1.0, 1.0))
W7 = tf.Variable(tf.random_uniform([2, 6], -1.0, 1.0))
W8 = tf.Variable(tf.random_uniform([6, 4], -1.0, 1.0))
W9 = tf.Variable(tf.random_uniform([4, 8], -1.0, 1.0))
W10 = tf.Variable(tf.random_uniform([8, 2], -1.0, 1.0))
W11 = tf.Variable(tf.random_uniform([2, 6], -1.0, 1.0))
W12 = tf.Variable(tf.random_uniform([6, 4], -1.0, 1.0))
W13 = tf.Variable(tf.random_uniform([4, 8], -1.0, 1.0))
W14 = tf.Variable(tf.random_uniform([8, 2], -1.0, 1.0))
W15 = tf.Variable(tf.random_uniform([2, 10], -1.0, 1.0))
W16 = tf.Variable(tf.random_uniform([10, 10], -1.0, 1.0))
W17 = tf.Variable(tf.random_uniform([10, 4], -1.0, 1.0))

b1 = tf.Variable(tf.zeros([8]), name="bias1")
b2 = tf.Variable(tf.zeros([2]), name="bias2")
b3 = tf.Variable(tf.zeros([6]), name="bias3")
b4 = tf.Variable(tf.zeros([4]), name="bias4")
b5 = tf.Variable(tf.zeros([8]), name="bias5")
b6 = tf.Variable(tf.zeros([2]), name="bias6")
b7 = tf.Variable(tf.zeros([6]), name="bias7")
```

```

b8 = tf.Variable(tf.zeros([4]), name="bias8")
b9 = tf.Variable(tf.zeros([8]), name="bias9")
b10 = tf.Variable(tf.zeros([2]), name="bias10")
b11 = tf.Variable(tf.zeros([6]), name="bias11")
b12 = tf.Variable(tf.zeros([4]), name="bias12")
b13 = tf.Variable(tf.zeros([8]), name="bias13")
b14 = tf.Variable(tf.zeros([2]), name="bias14")
b15 = tf.Variable(tf.zeros([10]), name="bias15")
b16 = tf.Variable(tf.zeros([10]), name="bias16")
b17 = tf.Variable(tf.zeros([4]), name="bias17")

#h = tf.matmul(W, X)      #matrix multiply
#hypothesis = tf.div(1.0, 1.0 + tf.exp(-h))
#Neural Network hypothesis
L2 = tf.sigmoid(tf.matmul(X, W1) + b1)
L3 = tf.sigmoid(tf.matmul(L2, W2) + b2)
L4 = tf.sigmoid(tf.matmul(L3, W3) + b3)
L5 = tf.sigmoid(tf.matmul(L4, W4) + b4)
L6 = tf.sigmoid(tf.matmul(L5, W5) + b5)
L7 = tf.sigmoid(tf.matmul(L6, W6) + b6)
L8 = tf.sigmoid(tf.matmul(L7, W7) + b7)
L9 = tf.sigmoid(tf.matmul(L8, W8) + b8)
L10 = tf.sigmoid(tf.matmul(L9, W9) + b9)
L11 = tf.sigmoid(tf.matmul(L10, W10) + b10)
L12 = tf.sigmoid(tf.matmul(L11, W11) + b11)
L13 = tf.sigmoid(tf.matmul(L12, W12) + b12)
L14 = tf.sigmoid(tf.matmul(L13, W13) + b13)
L15 = tf.sigmoid(tf.matmul(L14, W14) + b14)
L16 = tf.sigmoid(tf.matmul(L15, W15) + b15)
L17 = tf.sigmoid(tf.matmul(L16, W16) + b16)
hypothesis = tf.sigmoid(tf.matmul(L17, W17) + b17)

#Cost Function
cost = -tf.reduce_mean(Y * tf.log(hypothesis) + (1-Y) * tf.log(1-hypothesis))

#Minimize
a = tf.Variable(0.1)      #Learning rate, alpha
optimizer = tf.train.GradientDescentOptimizer(a)
train = optimizer.minimize(cost)

init = tf.initialize_all_variables()

with tf.Session() as sess:
    sess.run(init)

    print 'Learning'
    for step in range(1, 1001):
        sess.run(train, feed_dict={X:x_data, Y:y_data})
        if step % 100 == 0:
            print step, sess.run(cost, feed_dict={X:x_data, Y:y_data})

    print 'Test Model'
    prediction = tf.equal(tf.floor(hypothesis+0.5), Y)
    #Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(prediction, "float"))
    print sess.run([hypothesis, tf.floor(hypothesis+0.5), prediction, accuracy]),

```

```
feed_dict={X:x_data, Y:y_data} )
    print "Accuracy:", accuracy.eval({X:x_data, Y:y_data})
```

실행 결과

```
x [[ 0.  0.  1.  1.]
 [ 0.  1.  0.  1.]]
y [ 0.  1.  1.  0.]
Learning
100 0.0841887
200 0.0376717
300 0.0234386
400 0.0167628
500 0.0129443
600 0.0104911
700 0.00879241
800 0.00754854
900 0.00660142
1000 0.00585832
Test Model
[array([[ 0.00578441,  0.9942928 ,  0.99480999,  0.00668259],
       [ 0.00578441,  0.9942928 ,  0.99480999,  0.00668259]], dtype=float32), array([[ 0.,  1.,  1.,
0.],
       [ 0.,  1.,  1.,  0.]], dtype=float32), array([[ True,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)], 1.0]
Accuracy: 1.0
```

3.2.5 XOR ReLU

이번에는 sigmoid 함수 대신에 ReLU 를 사용해 보도록 하자.

XOR ReLU 예제

```
#XOR Neural Network with ReLU
#Editted by JungJaeJoon(rgb3307@nate.com) on the www.kernel.bz

import tensorflow as tf
import numpy as np

# Parameters
learning_rate = 0.1

xy = np.loadtxt('xor_train.txt', unpack=True, dtype='float32')

#xy.transpose()

x_data = xy[0:-1]
y_data = xy[-1]

print 'x', x_data
print 'y', y_data

X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

#W = tf.Variable(tf.random_uniform([1, len(x_data)], -1.0, 1.0))
#Deep Neural Network
W1 = tf.Variable(tf.random_uniform([4, 8], -1.0, 1.0))
W2 = tf.Variable(tf.random_uniform([8, 6], -1.0, 1.0))
W3 = tf.Variable(tf.random_uniform([6, 2], -1.0, 1.0))
W4 = tf.Variable(tf.random_uniform([2, 4], -1.0, 1.0))

b1 = tf.Variable(tf.zeros([8]), name="bias1")
b2 = tf.Variable(tf.zeros([6]), name="bias2")
b3 = tf.Variable(tf.zeros([2]), name="bias3")
b4 = tf.Variable(tf.zeros([4]), name="bias4")

#h = tf.matmul(W, X)      #matrix multiply
#hypothesis = tf.div(1.0, 1.0 + tf.exp(-h))

#Neural Network hypothesis
L2 = tf.nn.relu(tf.add(tf.matmul(X, W1), b1))
L3 = tf.nn.relu(tf.add(tf.matmul(L2, W2), b2))
L4 = tf.nn.relu(tf.add(tf.matmul(L3, W3), b3))
#hypothesis = tf.add(tf.matmul(L2, W2), b2)
hypothesis = tf.sigmoid(tf.matmul(L4, W4) + b4)

#Cost Function
#cost = -tf.reduce_mean(Y * tf.log(hypothesis) + (1-Y) * tf.log(1-hypothesis))
#cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(hypothesis, Y))
```

```
#optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

#Cost Function
cost = -tf.reduce_mean(Y * tf.log(hypothesis) + (1-Y) * tf.log(1-hypothesis))
#Minimize
a = tf.Variable(0.1)      #Learning rate, alpha
optimizer = tf.train.GradientDescentOptimizer(a)
train = optimizer.minimize(cost)

init = tf.initialize_all_variables()

with tf.Session() as sess:
    sess.run(init)

    print 'Learning'
    for step in range(1, 1001):
        #sess.run(optimizer, feed_dict={X:x_data, Y:y_data})
        sess.run(train, feed_dict={X:x_data, Y:y_data})
        if step % 100 == 0:
            print step, sess.run(cost, feed_dict={X:x_data, Y:y_data})

    print 'Test Model'
    prediction = tf.equal(tf.floor(hypothesis+0.5), Y)
    #Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(prediction, "float"))
    print sess.run([hypothesis, tf.floor(hypothesis+0.5), prediction, accuracy],
feed_dict={X:x_data, Y:y_data} )
    print "Accuracy:", accuracy.eval({X:x_data, Y:y_data})
```

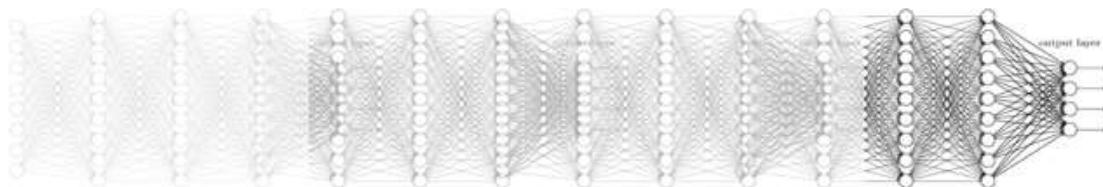
실행 결과

```
x [[ 0.  0.  1.  1.]
 [ 0.  1.  0.  1.]]
y [ 0.  1.  1.  0.]
Learning
100 0.329186
200 0.200388
300 0.140567
400 0.107165
500 0.0861554
600 0.0718323
700 0.0614879
800 0.0536879
900 0.0476078
1000 0.0427423
Test Model
[array([[ 0.04184143,  0.95815796,  0.95815796,  0.04184143],
       [ 0.04184143,  0.95815796,  0.95815796,  0.04184143]], dtype=float32), array([[ 0.,  1.,  1.,
0.], [ 0.,  1.,  1.,  0.]], dtype=float32), array([[ True,  True,  True,  True],
[ True,  True,  True,  True]], dtype=bool), 1.0]
Accuracy: 1.0
```

3.3 딥러닝 정확성 향상

머신러닝에서 첫번째 난제였던 XOR 문제를 1986년에 BackPropagation을 사용하여 Neural Network로 해결하였지만, 또다른 난제인 Vanishing Gradient 문제가 발생하게 된다. 이 문제는 학습이 진행될 때마다 sigmoid 함수를 곱하는 것이 원인 이었다. Sigmoid는 0에서 1 사이의 값을 계속 곱하는 특징이 있으므로, 학습을 깊게 진행하면 할수록 W 값의 변화량이 점점 작아져서 결국에는 변화량이 없어지는 문제가 발생한다.

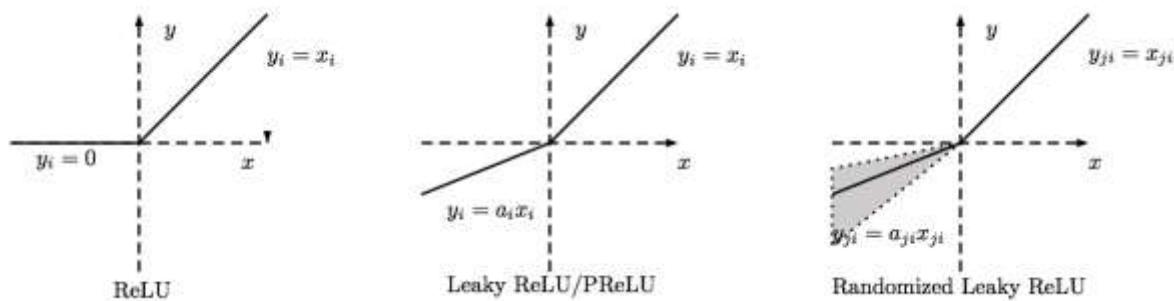
Vanishing gradient (NN winter2: 1986-2006)



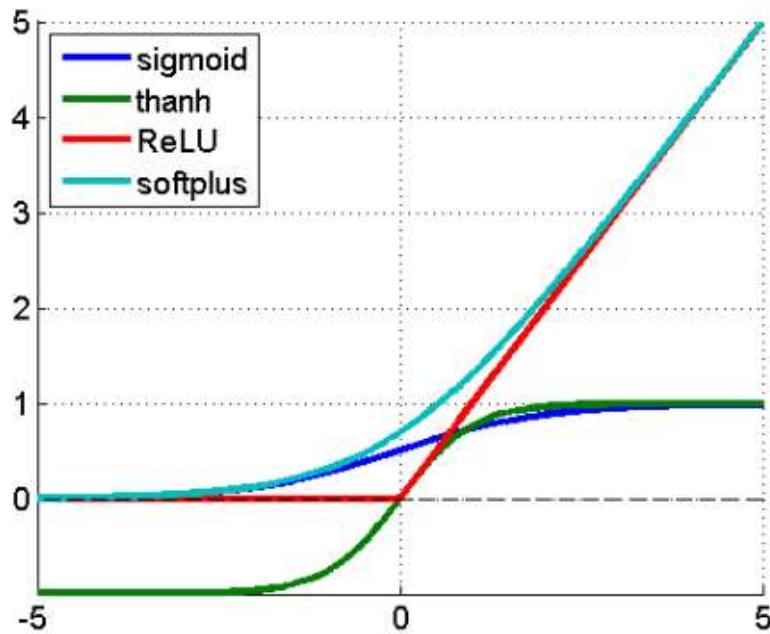
이것을 Vanishing Gradient 문제라고 하는데, 이 문제도 2006년에 ReLU 함수를 도입하면서 해결하게 된다. 이때부터 Deep Learning 용어가 도입되고 이것이 인정받기 시작한다. 아래부터 딥러닝의 정확성을 어떻게 향상시켜 왔는지 자세히 알아보도록 하자.

3.3.1 ReLU

ReLU: Rectified Linear Unit



출처: <http://www.kdnuggets.com/2016/03/must-know-tips-deep-learning-part-2.html>



출처: <https://imiloainf.wordpress.com/2013/11/06/rectifier-nonlinearity/>

sigmoid: $g(x) = 1 / (1 + \exp(-x))$. sigmoid 의 미분함수 $g'(x) = (1-g(x))g(x)$.

tanh : $g(x) = \tanh(x) = (\exp(x) - \exp(-x)) / (\exp(x) + \exp(-x))$

ReLU 함수는 다음과 같이 max 함수로 표현할 수 있다.

$$g(x) = \max(0, x)$$

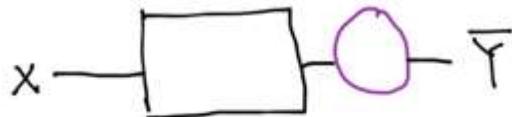
ReLU 함수는 아래와 같이 표현되기도 한다.

Noise ReLU $\max(0, x + N(0, \sigma(x)))$

이것의 미분함수는 다음과 같이 log 형태로 나타난다.

$$g'(x) = \log(1 + \exp(x))$$

아래는 TensorFlow에서 sigmoid 와 ReLU 함수를 사용하는 코드이다.

TensorFlow for ReLU

$$L1 = \text{tf.sigmoid}(\text{tf.matmul}(X, W1) + b1)$$

$$L1 = \text{tf.nn.relu}(\text{tf.matmul}(X, W1) + b1)$$
3.3.2 Good Weight (초기값)

학습할 때 산출되는 Weight 값을 초기에 어떻게 선정하는가에 따라서 학습의 정확성에 많은 영향을 준다. 아래는 TensorFlow에서 W에 대한 초기값을 결정하는 예를 보여 주는 것이다. 입력 데이터의 개수에 해당하는 fan_in과 출력 데이터의 개수에 해당하는 fan_out 사이에서 random 한 수를 선택하여 이것을 $\text{sqrt}(\text{fan_in})$ 로 나누어서 초기값을 결정하는 예제이다.

Good Weight (초기값) 선택

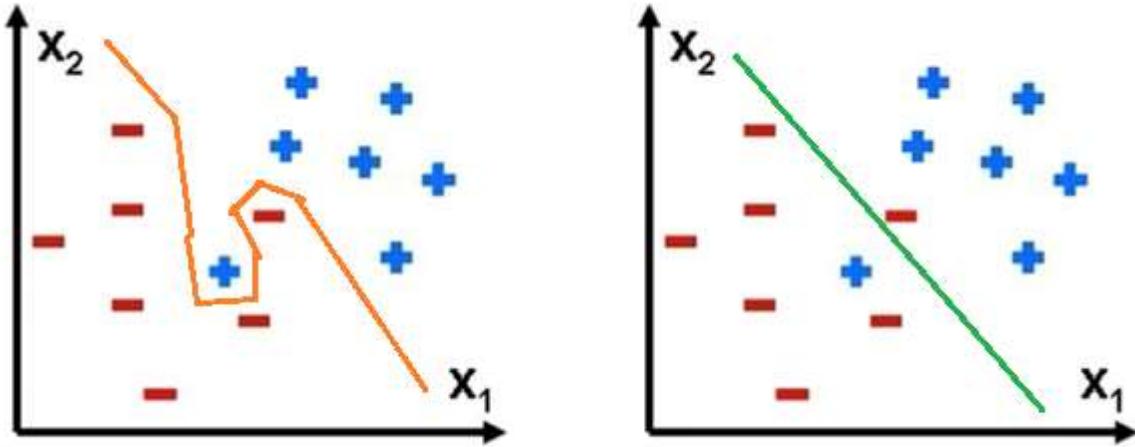
```
# Xavier initialization
# Glorot et al. 2010
W = np.random.randn(fan_in, fan_out)/np.sqrt(fan_in)

# He et al. 2015
W = np.random.randn(fan_in, fan_out)/np.sqrt(fan_in/2)
```

위의 예제는 다음 절에 있는 실습 코드에서 xavier_init 함수로 만들어서 자세히 알아볼 것이다.

3.3.3 Overfitting 조정

Overfitting은 학습을 지나치게 많이 하면 오히려 학습의 정확성이 떨어지는 문제가 있기 때문에 학습량을 적절히 조절하여 정확성을 신뢰할 수 있는 수준으로 높여주는 방법이다.



Regularization

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2 \right]$$

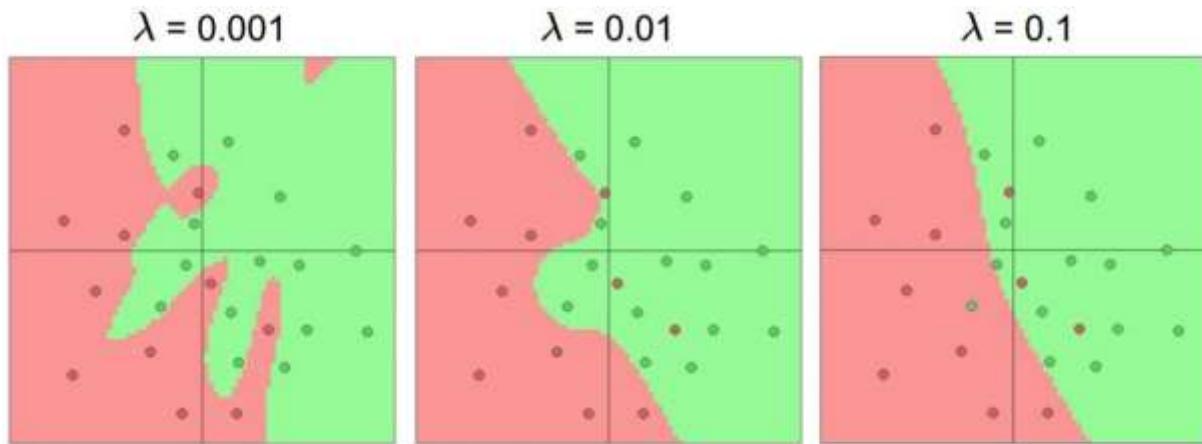
Regularization

TensorFlow에서는 위와 같이 학습한 W 을 제곱한 값을 더하여 적절한 값(0.001)을 곱하여 정형화(Regularization) 한다.

$$\text{cost} + \lambda \sum W^2$$

```
l2reg = 0.001 * tf.reduce_sum(tf.square(W))
```

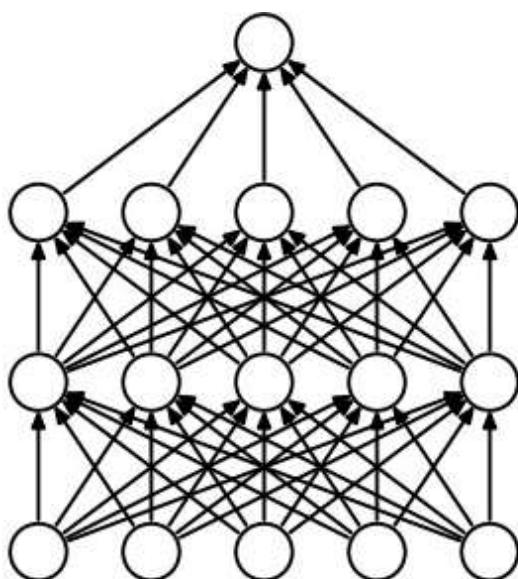
아래 그림은 곱해지는 값에 따라서 정형화(Regularization) 되는 모습을 그림으로 보여주는 것이다.



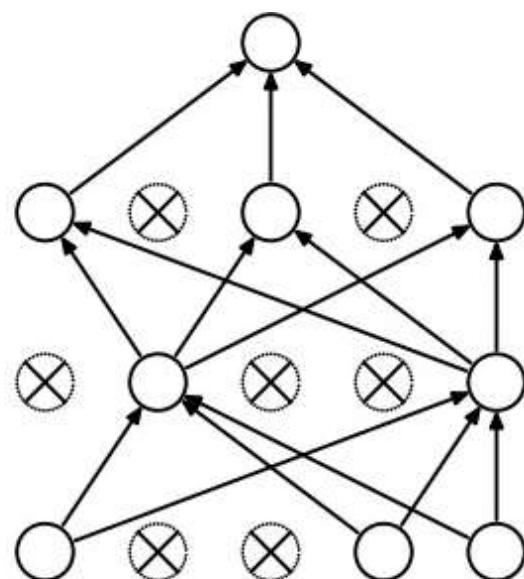
출처: <http://cs231n.github.io/neural-networks-1/>

3.3.4 DropOut

DropOut은 Neural Network에서 불필요한 뉴런들을 제거하여 학습 효율을 향상 시키는 방법이다. 아래 그림에서 왼쪽은 모든 뉴런들이 학습에 참여하는 것이고, 오른쪽에 있는 그림은 불필요한 뉴런들을 DropOut으로 제거한 것이다.



(a) Standard Neural Net



(b) After applying dropout.

TensorFlow 에서는 아래와 같이 dropout 함수의 매개변수로 dropout_rate 를 전달하면 되는데, 이때 dropout_rate 가 0.7 이면 학습에 참여하는 뉴런들은 70%가 되고 나머지 30%는 DropOut 된다.

```
dropout_rate = tf.placeholder("float")
_L1 = tf.nn.relu(tf.add(tf.matmul(X, W1), B1))
L1 = tf.nn.dropout(_L1, dropout_rate)
```

TRAIN:

```
sess.run(optimizer, feed_dict={X: batch_xs, Y: batch_ys,
dropout_rate: 0.7})
```

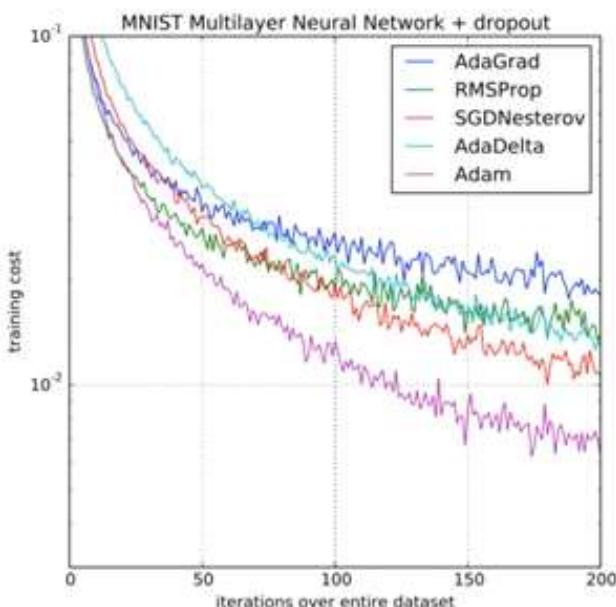
EVALUATION:

```
print "Accuracy:", accuracy.eval({X: mnist.test.images, Y:
mnist.test.labels, dropout_rate: 1})
```

3.3.5 Optimizer 성능 비교

아래는 Deep Learning 옵티마이저별로 성능을 평가한 그래프이다. 얼마나 빨리 비용(cost)이 줄어드는가로 성능을 평가한 것이며, 현재, Adam 옵티마이저가 가장 성능이 좋은 것으로 평가되고 있다.

Optimizer 종류별 성능평가



TensorFlow에 구현되어 있는 옵티마이저들

```
class tf.train.GradientDescentOptimizer  
class tf.train.AdadeltaOptimizer  
class tf.train.AdagradOptimizer  
class tf.train.MomentumOptimizer  
class tf.train.AdamOptimizer  
class tf.train.FtrlOptimizer  
class tf.train.RMSPropOptimizer
```

3.4 딥러닝 실습

지금까지 기술한 딥러닝 기법들을 실습 예제를 통하여 익혀 보도록 하자. 아래 그림은 손으로 쓴 숫자 이미지이다. 이미지 한 개의 크기는 가로 28 픽셀, 세로 28 픽셀이다. 이 이미지는 아래 링크에서 다운로드 가능하다.

<http://yann.lecun.com/exdb/mnist/>

파이썬 예제에서도 이미지를 다운로드하여 사용할 것이다. 손글씨인 숫자를 딥러닝으로 학습하여 이것을 인식하는 예제들을 익혀 보도록 하자.

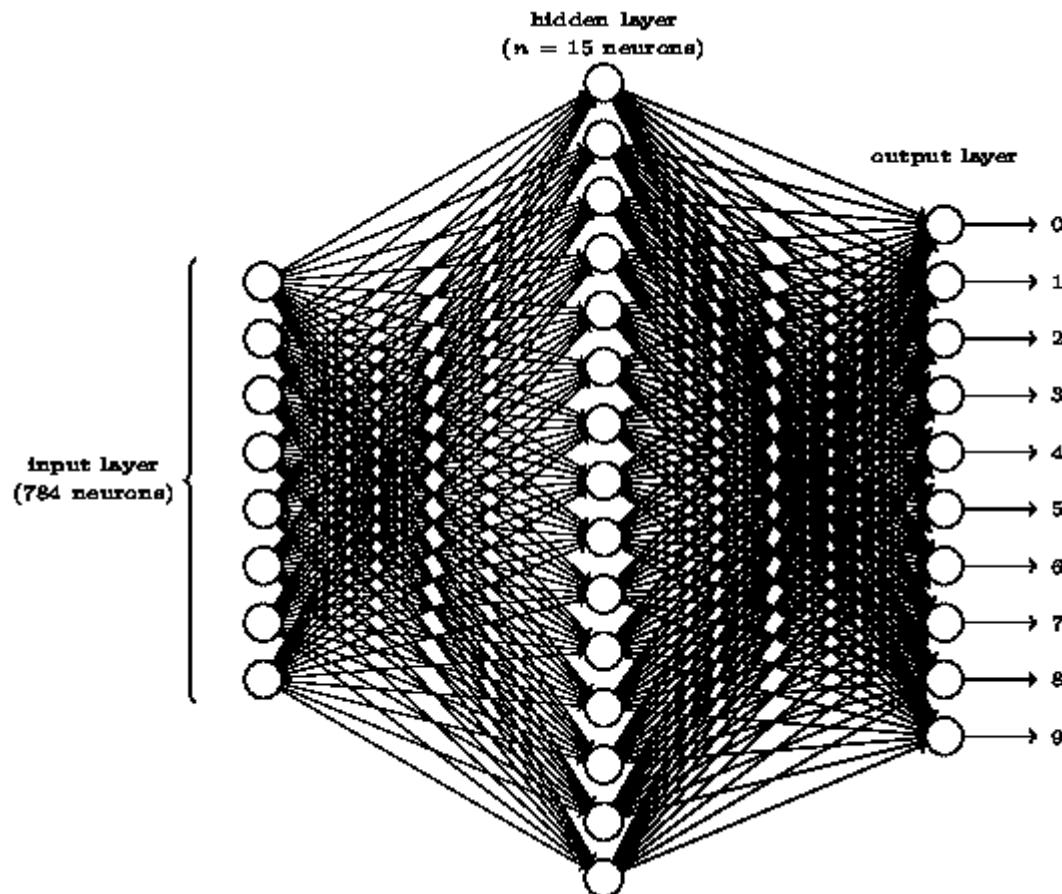
손글씨 숫자 이미지



출처: <http://neuralnetworksanddeeplearning.com/chap1.html>

위의 이미지는 가로 28 픽셀, 세로 28 픽셀이므로, 학습해야하는 픽셀은 가로와 세로를 곱한 784 픽셀이다. 784 픽셀마다 w 가중치들을 각각 학습하여 0부터 9 까지 10 가지 숫자를 인식하도록해야 한다. 따라서 784 개의 입력 데이터에 대해서 Neural Network 을 구성하여 10 개의 출력(학습결과)이 나오도록 해야 한다. 아래 그림은 이것을 구현하는 Neural Network 이다.

손글씨 숫자 인식용 Neural Network



출처: <http://neuralnetworksanddeeplearning.com/chap1.html>

3.4.1 일반적인 softmax

먼저, Neural Network 을 구성하지 않고, 일반적인 softmax 방식으로 위의 데이터를 학습시키는 파이썬 예제를 실행해 보도록 하자.

일반적인 softmax 예제

```
'''  
A logistic regression learning algorithm example using TensorFlow library.  
This example is using the MNIST database of handwritten digits (http://yann.lecun.com/exdb/mnist/)
```

```
Author: Aymeric Damien  
Project: https://github.com/aymericdamien/TensorFlow-Examples/  
'''
```

```
# Import MINST data  
import input_data
```

```
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

import tensorflow as tf
import random

# Parameters
learning_rate = 0.01
training_epochs = 25
batch_size = 100
display_step = 1

# tf Graph Input
x = tf.placeholder("float", [None, 784]) # mnist data image of shape 28*28=784
y = tf.placeholder("float", [None, 10]) # 0-9 digits recognition => 10 classes

# Create model

# Set model weights
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

# Construct model
activation = tf.nn.softmax(tf.matmul(x, W) + b) # Softmax

# Minimize error using cross entropy
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(activation), reduction_indices=1)) # Cross entropy
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost) # Gradient Descent

# Initializing the variables
init = tf.initialize_all_variables()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Fit training using batch data
            sess.run(optimizer, feed_dict={x: batch_xs, y: batch_ys})
            # Compute average loss
            avg_cost += sess.run(cost, feed_dict={x: batch_xs, y: batch_ys})/total_batch
        # Display logs per epoch step
        if epoch % display_step == 0:
            print "Epoch: ", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg_cost)

    print "Optimization Finished!"

    # Test model
    correct_prediction = tf.equal(tf.argmax(activation, 1), tf.argmax(y, 1))
    # Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

```
print "Accuracy:", accuracy.eval({x: mnist.test.images, y: mnist.test.labels})  
  
r = random.randint(0, mnist.test.num_examples - 1)  
print "Label: ", sess.run(tf.argmax(mnist.test.labels[r:r+1], 1))  
print "Prediction: ", sess.run(tf.argmax(activation,1), {x: mnist.test.images[r:r+1]})
```

실행결과

```
Extracting /tmp/data/train-images-idx3-ubyte.gz  
Extracting /tmp/data/train-labels-idx1-ubyte.gz  
Extracting /tmp/data/t10k-images-idx3-ubyte.gz  
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz  
Epoch: 0001 cost= 1.174410056  
Epoch: 0002 cost= 0.662088700  
Epoch: 0003 cost= 0.550482203  
Epoch: 0004 cost= 0.496659856  
Epoch: 0005 cost= 0.463734650  
Epoch: 0006 cost= 0.440835492  
Epoch: 0007 cost= 0.423904836  
Epoch: 0008 cost= 0.410614374  
Epoch: 0009 cost= 0.399880962  
Epoch: 0010 cost= 0.390925939  
Epoch: 0011 cost= 0.383348579  
Epoch: 0012 cost= 0.376748794  
Epoch: 0013 cost= 0.371020428  
Epoch: 0014 cost= 0.365902853  
Epoch: 0015 cost= 0.361321656  
Epoch: 0016 cost= 0.357280480  
Epoch: 0017 cost= 0.353479697  
Epoch: 0018 cost= 0.350177354  
Epoch: 0019 cost= 0.346985189  
Epoch: 0020 cost= 0.344120319  
Epoch: 0021 cost= 0.341458275  
Epoch: 0022 cost= 0.338966686  
Epoch: 0023 cost= 0.336628434  
Epoch: 0024 cost= 0.334487694  
Epoch: 0025 cost= 0.332427301  
Optimization Finished!  
Accuracy: 0.9136
```

실행결과를 확인해 보면 정확도는 약 0.91(91%) 정도 된다.

3.4.2 ReLU

이번에는 Neural Network 을 구성한후 LeLU 연산을 수행하는 TensorFlow 라이브러리를 파일 예제에서 실행해 보도록 하자.

ReLU 예제

```
'''  
A Multilayer Perceptron implementation example using TensorFlow library.  
This example is using the MNIST database of handwritten digits (http://yann.lecun.com/exdb/mnist/)  
  
Author: Aymeric Damien  
Project: https://github.com/aymericdamien/TensorFlow-Examples/  
Editted by JungJaeJoon(rgb13307@nate.com) on the www.kernel.bz  
'''  
  
# Import MINST data  
import input_data  
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)  
  
import tensorflow as tf  
  
# Parameters  
learning_rate = 0.001  
training_epochs = 15  
batch_size = 100  
display_step = 1  
  
# Network Parameters  
n_hidden_1 = 256 # 1st layer num features  
n_hidden_2 = 256 # 2nd layer num features  
n_input = 784 # MNIST data input (img shape: 28*28)  
n_classes = 10 # MNIST total classes (0-9 digits)  
  
# tf Graph input  
X = tf.placeholder("float", [None, n_input])  
Y = tf.placeholder("float", [None, n_classes])  
  
# Weight  
W1 = tf.Variable(tf.random_normal([n_input, n_hidden_1]))  
W2 = tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2]))  
W3 = tf.Variable(tf.random_normal([n_hidden_2, n_classes]))  
  
b1 = tf.Variable(tf.random_normal([n_hidden_1]))  
b2 = tf.Variable(tf.random_normal([n_hidden_2]))  
b3 = tf.Variable(tf.random_normal([n_classes]))  
  
# Layers  
L1 = tf.nn.relu(tf.add(tf.matmul(X, W1), b1))  
L2 = tf.nn.relu(tf.add(tf.matmul(L1, W2), b2))  
hypothesis = tf.add(tf.matmul(L2, W3), b3)  
  
# Define loss and optimizer
```

```

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(hypothesis, Y)) # Softmax loss
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost) # Adam Optimizer

# Initializing the variables
init = tf.initialize_all_variables()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Fit training using batch data
            sess.run(optimizer, feed_dict={X:batch_xs, Y:batch_ys})
            # Compute average loss
            avg_cost += sess.run(cost, feed_dict={X:batch_xs, Y:batch_ys})/total_batch
        # Display logs per epoch step
        if epoch % display_step == 0:
            print "Epoch: ", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg_cost)

    print "Optimization Finished!"

# Test model
correct_prediction = tf.equal(tf.argmax(hypothesis, 1), tf.argmax(Y, 1))
# Calculate accuracy
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
print "Accuracy:", accuracy.eval({X:mnist.test.images, Y:mnist.test.labels})

```

실행결과

```

Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Epoch: 0001 cost= 151.206853309
Epoch: 0002 cost= 36.956441101
Epoch: 0003 cost= 23.312868608
Epoch: 0004 cost= 16.085679977
Epoch: 0005 cost= 11.531727971
Epoch: 0006 cost= 8.392710752
Epoch: 0007 cost= 6.116542267
Epoch: 0008 cost= 4.590167932
Epoch: 0009 cost= 3.276341141
Epoch: 0010 cost= 2.363842059
Epoch: 0011 cost= 1.717204026
Epoch: 0012 cost= 1.189641714
Epoch: 0013 cost= 0.841942948
Epoch: 0014 cost= 0.571519489
Epoch: 0015 cost= 0.427100243

```

```
Optimization Finished!
Accuracy: 0.9468
```

실행결과를 확인해 보면 정확도는 약 0.95(95%) 정도 된다. Neural Network으로 구성하면 확실히 정확도가 높아짐을 알 수 있다.

3.4.3 DropOut

이번에는 Neural Network 안에서 몇가지 뉴런들을 제외(DropOut) 시키는 방법을 실습해 보도록 하자. 아래의 예제는 DropOut Rate 을 0.7로 해서 30% 정도의 뉴런들을 제외 시키는 것이다.

DropOut 예제

```
'''  
A Multilayer Perceptron implementation example using TensorFlow library.  
This example is using the MNIST database of handwritten digits (http://yann.lecun.com/exdb/mnist/)
```

```
Author: Aymeric Damien  
Project: https://github.com/aymericdamien/TensorFlow-Examples/  
Edited by JungJaeJoon(rgb13307@nate.com) on the www.kernel.bz  
'''
```

```
# Import MINST data
import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

import tensorflow as tf

def xavier_init(n_inputs, n_outputs, uniform=True):
    if uniform:
        init_range = tf.sqrt(6.0 / (n_inputs + n_outputs))
        return tf.random_uniform_initializer(-init_range, init_range)
    else:
        stddev = tf.sqrt(3.0 / (n_inputs + n_outputs))
        return tf.truncated_normal_initializer(stddev=stddev)

# Parameters
learning_rate = 0.001
training_epochs = 15
batch_size = 100
display_step = 1

# Network Parameters
n_hidden_1 = 256 # 1st layer num features
n_hidden_2 = 256 # 2nd layer num features
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)
```

```

# tf Graph input
X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_classes])

# Weight
W1 = tf.get_variable("W1", shape=[n_input, n_hidden_1], initializer=xavier_init(n_input,
n_hidden_1))
W2 = tf.get_variable("W2", shape=[n_hidden_1, n_hidden_2], initializer=xavier_init(n_hidden_1,
n_hidden_2))
W3 = tf.get_variable("W3", shape=[n_hidden_2, n_hidden_2], initializer=xavier_init(n_hidden_2,
n_hidden_2))
W4 = tf.get_variable("W4", shape=[n_hidden_2, n_hidden_2], initializer=xavier_init(n_hidden_2,
n_hidden_2))
W5 = tf.get_variable("W5", shape=[n_hidden_2, n_classes], initializer=xavier_init(n_hidden_2,
n_classes))

b1 = tf.Variable(tf.random_normal([n_hidden_1]))
b2 = tf.Variable(tf.random_normal([n_hidden_2]))
b3 = tf.Variable(tf.random_normal([n_hidden_2]))
b4 = tf.Variable(tf.random_normal([n_hidden_2]))
b5 = tf.Variable(tf.random_normal([n_classes]))

# Layers for dropout
dropout_rate = tf.placeholder("float")
_L1 = tf.nn.relu(tf.add(tf.matmul(X, W1), b1))
L1 = tf.nn.dropout(_L1, dropout_rate)
_L2 = tf.nn.relu(tf.add(tf.matmul(L1, W2), b2))
L2 = tf.nn.dropout(_L2, dropout_rate)
_L3 = tf.nn.relu(tf.add(tf.matmul(L2, W3), b3))
L3 = tf.nn.dropout(_L3, dropout_rate)
_L4 = tf.nn.relu(tf.add(tf.matmul(L3, W4), b4))
L4 = tf.nn.dropout(_L4, dropout_rate)

hypothesis = tf.add(tf.matmul(L4, W5), b5)

# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(hypothesis, Y)) # Softmax loss
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost) # Adam Optimizer

# Initializing the variables
init = tf.initialize_all_variables()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Fit training using batch data
            sess.run(optimizer, feed_dict={X:batch_xs, Y:batch_ys, dropout_rate:0.7})
            # Compute average loss

```

```
avg_cost += sess.run(cost, feed_dict={X:batch_xs, Y:batch_ys,
dropout_rate:0.7})/total_batch
# Display logs per epoch step
if epoch % display_step == 0:
    print "Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg_cost)

print "Optimization Finished!"

# Test model
correct_prediction = tf.equal(tf.argmax(hypothesis, 1), tf.argmax(Y, 1))
# Calculate accuracy
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
print "Accuracy:", accuracy.eval({X:mnist.test.images, Y:mnist.test.labels, dropout_rate:1})
```

실행결과

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Epoch: 0001 cost= 0.567883993
Epoch: 0002 cost= 0.212641549
Epoch: 0003 cost= 0.164437731
Epoch: 0004 cost= 0.132774231
Epoch: 0005 cost= 0.113458113
Epoch: 0006 cost= 0.104297687
Epoch: 0007 cost= 0.091792821
Epoch: 0008 cost= 0.086032183
Epoch: 0009 cost= 0.077678566
Epoch: 0010 cost= 0.074236836
Epoch: 0011 cost= 0.071206722
Epoch: 0012 cost= 0.064238651
Epoch: 0013 cost= 0.062537259
Epoch: 0014 cost= 0.062307651
Epoch: 0015 cost= 0.055458841
Optimization Finished!
Accuracy: 0.9785
```

실행결과를 보면, 정확도가 0.978 정도가 나온다. 정확도가 좀더 향상됨을 알 수 있다.

3.4.4 초기값 설정

이번에는 가중치 초기값을 적절하게 주어서 실행해 보자.

초기값 설정 예제

```
'''  
A Multilayer Perceptron implementation example using TensorFlow library.  
This example is using the MNIST database of handwritten digits (http://yann.lecun.com/exdb/mnist/)  
  
Author: Aymeric Damien  
Project: https://github.com/aymericdamien/TensorFlow-Examples/  
Editted by JungJaeJoon(rgb13307@nate.com) on the www.kernel.bz  
'''  
  
# Import MINST data  
import input_data  
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)  
  
import tensorflow as tf  
  
def xavier_init(n_inputs, n_outputs, uniform=True):  
    if uniform:  
        init_range = tf.sqrt(6.0 / (n_inputs + n_outputs))  
        return tf.random_uniform_initializer(-init_range, init_range)  
    else:  
        stddev = tf.sqrt(3.0 / (n_inputs + n_outputs))  
        return tf.truncated_normal_initializer(stddev=stddev)  
  
# Parameters  
learning_rate = 0.001  
training_epochs = 15  
batch_size = 100  
display_step = 1  
  
# Network Parameters  
n_hidden_1 = 256 # 1st layer num features  
n_hidden_2 = 256 # 2nd layer num features  
n_input = 784 # MNIST data input (img shape: 28*28)  
n_classes = 10 # MNIST total classes (0-9 digits)  
  
# tf Graph input  
X = tf.placeholder("float", [None, n_input])  
Y = tf.placeholder("float", [None, n_classes])  
  
# Weight  
W1 = tf.get_variable("W1", shape=[n_input, n_hidden_1], initializer=xavier_init(n_input,  
n_hidden_1))  
W2 = tf.get_variable("W2", shape=[n_hidden_1, n_hidden_2], initializer=xavier_init(n_hidden_1,  
n_hidden_2))  
W3 = tf.get_variable("W3", shape=[n_hidden_2, n_classes], initializer=xavier_init(n_hidden_2,  
n_classes))
```

```

b1 = tf.Variable(tf.random_normal([n_hidden_1]))
b2 = tf.Variable(tf.random_normal([n_hidden_2]))
b3 = tf.Variable(tf.random_normal([n_classes]))

# Layers
L1 = tf.nn.relu(tf.add(tf.matmul(X, W1), b1))
L2 = tf.nn.relu(tf.add(tf.matmul(L1, W2), b2))
hypothesis = tf.add(tf.matmul(L2, W3), b3)

# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(hypothesis, Y)) # Softmax loss
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost) # Adam Optimizer

# Initializing the variables
init = tf.initialize_all_variables()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Fit training using batch data
            sess.run(optimizer, feed_dict={X:batch_xs, Y:batch_ys})
            # Compute average loss
            avg_cost += sess.run(cost, feed_dict={X:batch_xs, Y:batch_ys})/total_batch
        # Display logs per epoch step
        if epoch % display_step == 0:
            print "Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg_cost)

    print "Optimization Finished!"

    # Test model
    correct_prediction = tf.equal(tf.argmax(hypothesis, 1), tf.argmax(Y, 1))
    # Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print "Accuracy:", accuracy.eval({X:mnist.test.images, Y:mnist.test.labels})

```

실행결과

```

Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Epoch: 0001 cost= 0.303844176
Epoch: 0002 cost= 0.104028660
Epoch: 0003 cost= 0.065244118
Epoch: 0004 cost= 0.044454055
Epoch: 0005 cost= 0.030983394

```

```

Epoch: 0006 cost= 0.021459243
Epoch: 0007 cost= 0.015217349
Epoch: 0008 cost= 0.011872460
Epoch: 0009 cost= 0.008130962
Epoch: 0010 cost= 0.008212744
Epoch: 0011 cost= 0.006703014
Epoch: 0012 cost= 0.003906108
Epoch: 0013 cost= 0.004232739
Epoch: 0014 cost= 0.004422310
Epoch: 0015 cost= 0.002568416
Optimization Finished!
Accuracy: 0.9801

```

정확도가 0.98(98%)로 더욱 향상되는 결과가 나온다.

3.4.5 결과 정리

지금까지 실습한 예제에서 산출되는 정확도를 테이블로 정리하면 다음과 같다.

딥러닝 기법	실행 시간	정확도
일반적인 softmax	8 분 10 초	0.914
ReLU	21 분 40 초	0.945
DropOut	38 분 10 초	0.978
초기값 설정	22 분 50 초	0.980

따라서 초기값을 얼마나 적절하게 주는가에 따라서 딥러닝의 정확도가 많이 영향을 받는다는 것을 알 수 있다.

♣ 책에 있는 모든 소스들은 아래 링크에서 다운로드 가능합니다.

<https://github.com/kernel-bz/ml>

머신러닝/딥러닝 TensorFlow 실습

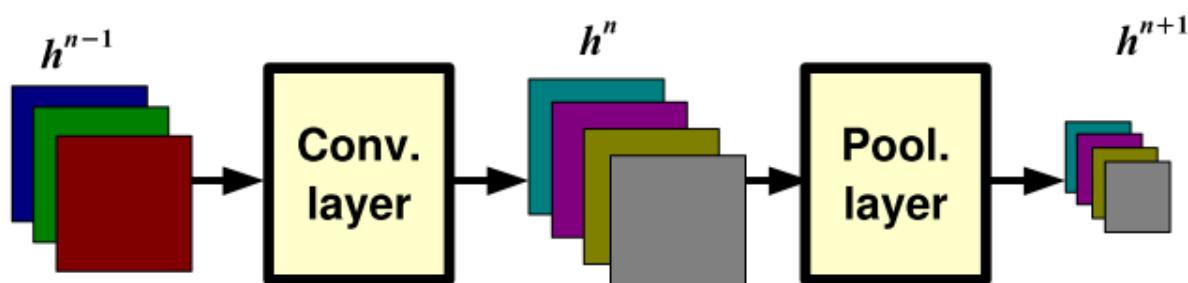
Third Edition

4. 영상 인지(CNN)

4. 영상 인지(CNN)

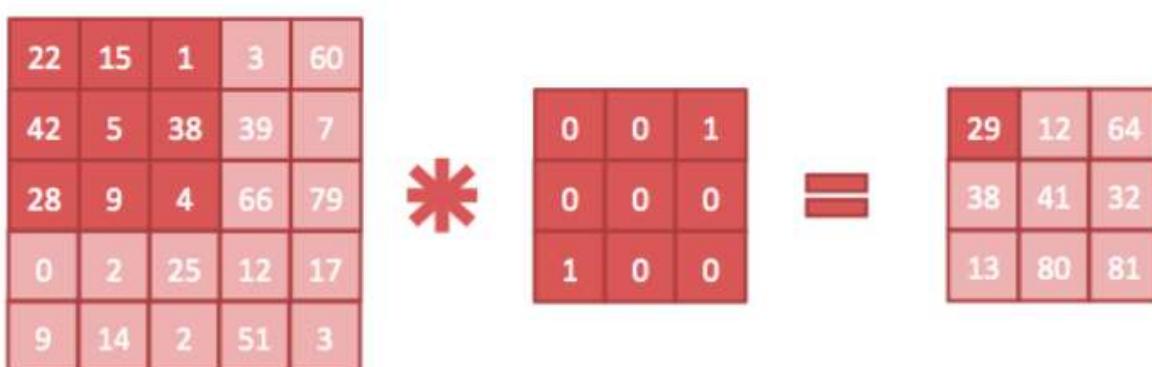
CNN(Convolutional Neural Network)은 사람 눈의 시각 피질에서 영감을 얻었다. 우리가 무언가를 볼 때마다 시각 세포와 뇌세포의 뉴런 레이어가 활성화되고 각 레이어는 선, 가장자리와 같은 시각 특징들을 분류하여 인지한다. 높은 수준의 레이어는 우리가 본 것을 인식하기 위해 더 복잡한 기능을 감지하기도 한다. CNN은 이러한 시각 인지를 수학적으로 모델링하여 컴퓨터 연산에 적용한 알고리즘이다. 학습(Conv, Relu, Pool) → 분류(Fully Connection, Softmax)

Convolutional Neural Network



그림출처: <http://www.cs.toronto.edu/~ranzato/>

Convolution 연산



그림출처: <https://www.quora.com/How-can-I-explain-the-dimensionality-reduction-in-convolutional-neural-network-CNN-from-this-image>

Convolution 연산 예



$$\text{Input Image} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \text{Output Image}$$

(a) Identity kernel



$$\text{Input Image} * \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \text{Output Image}$$

(b) Edge detection kernel



$$\text{Input Image} * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \text{Output Image}$$

(c) Blur kernel



$$\text{Input Image} * \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} = \text{Output Image}$$

(d) Sharpen kernel



$$\text{Input Image} * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 12 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \text{Output Image}$$

(e) Lighten kernel



$$\text{Input Image} * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \text{Output Image}$$

(f) Darken kernel



$$\text{Input Image} * \begin{bmatrix} -1 & -2 & -1 \\ -2 & 12 & -2 \\ -1 & -2 & -1 \end{bmatrix} = \text{Output Image}$$

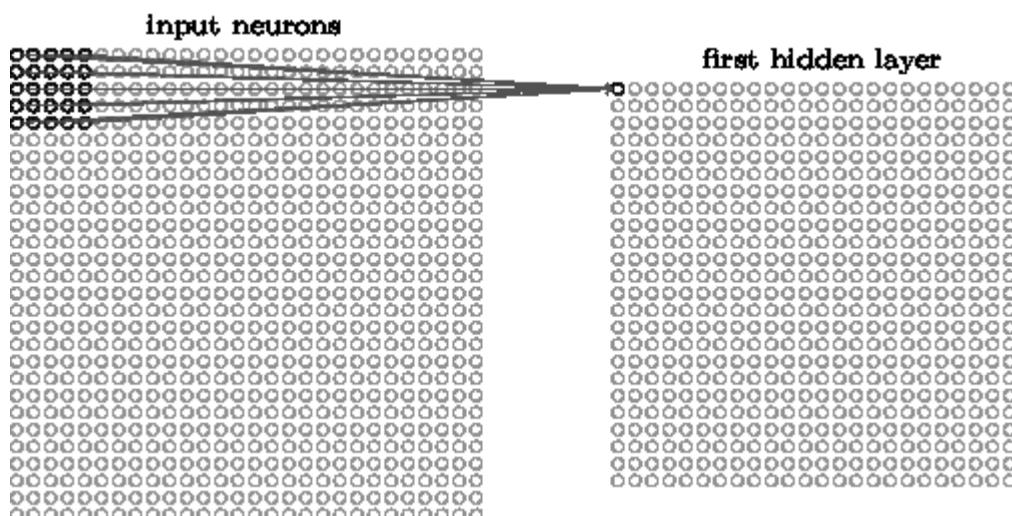
(g) Random kernel 1

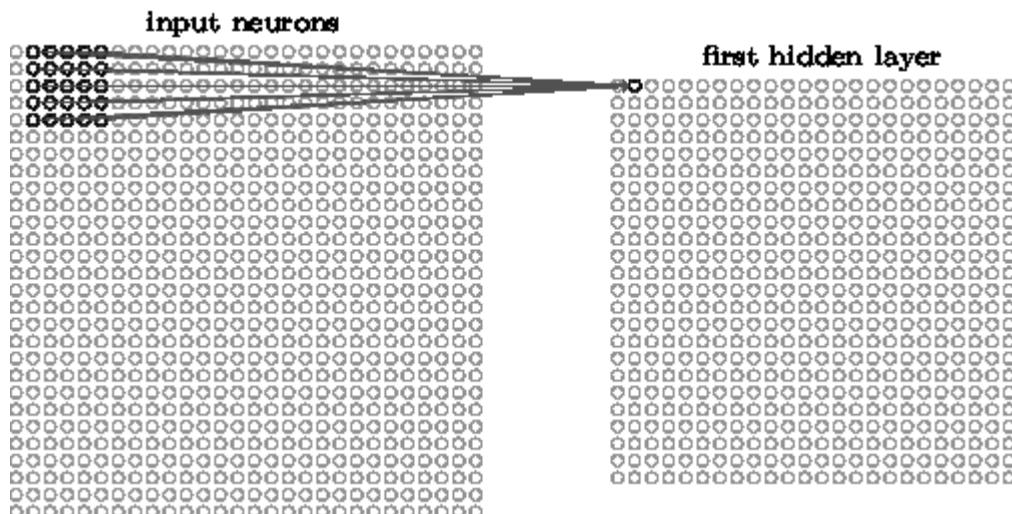


$$\text{Input Image} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \text{Output Image}$$

(h) Random kernel 2

4.1 Convolution 연산



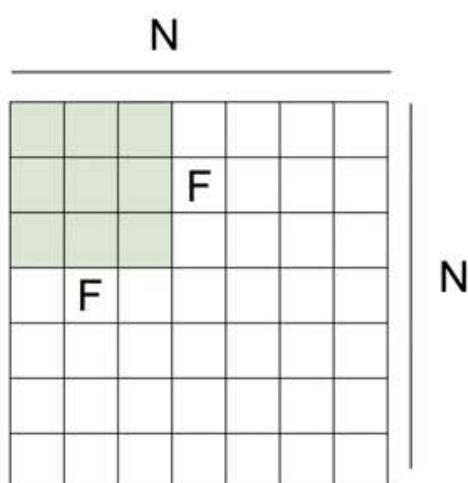


그림출처: <http://neuralnetworksanddeeplearning.com/chap6.html>

Convolution 수학적 정의

$$\left(b + \sum_{l=0}^4 \sum_{m=0}^4 w_{l,m} a_{j+l,k+m} \right)$$

출력 크기 연산식



Output size:
 $(N - F) / \text{stride} + 1$

e.g. $N = 7$, $F = 3$:
 stride 1 => $(7 - 3)/1 + 1 = 5$
 stride 2 => $(7 - 3)/2 + 1 = 3$
 stride 3 => $(7 - 3)/3 + 1 = 2.33$

Padding

0	0	0	0	0	0			
0								
0								
0								
0								

e.g. input 7x7

3x3 filter, applied with **stride 1**

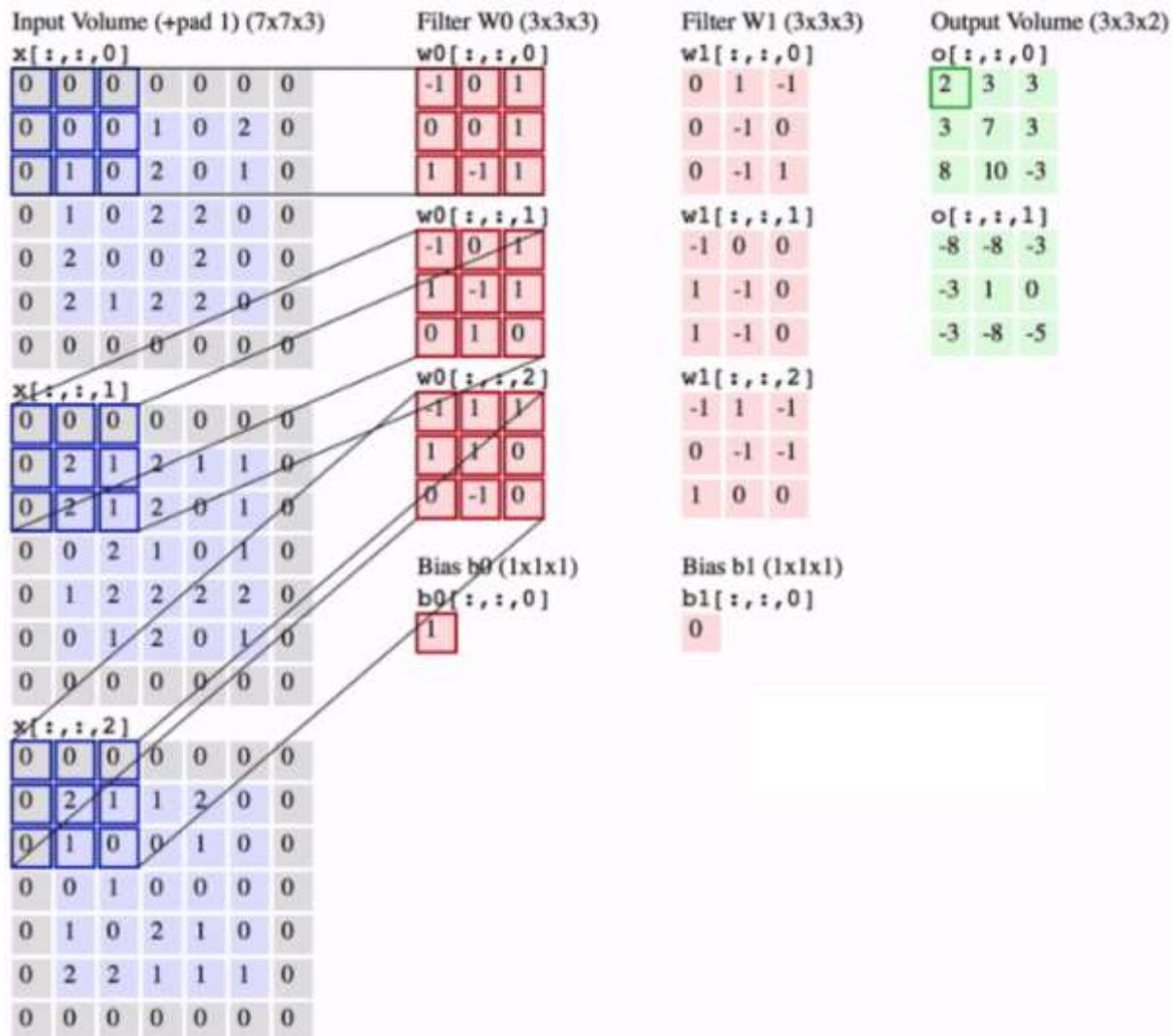
pad with 1 pixel border => what is the output?

(recall:)

$$(N - F) / \text{stride} + 1$$

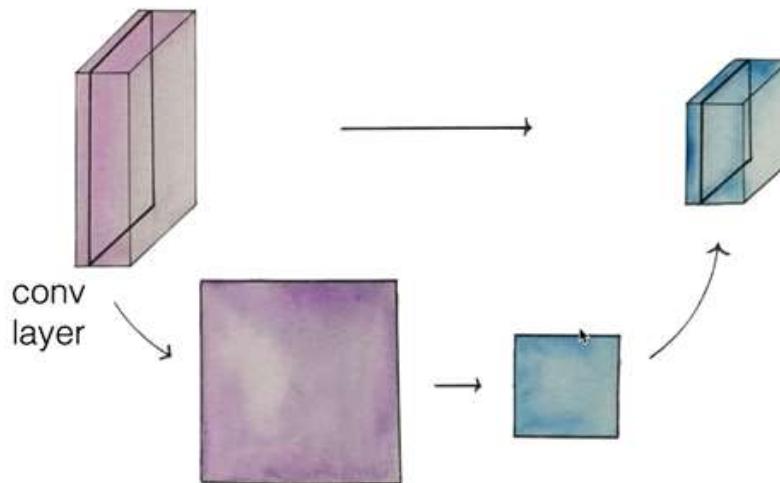
위에서 패딩이 추가 되었으므로 $N = 7 + 2 = 9$ 가 되고 필터의 크기 $F = 3$, Stride 는 1 이 되므로 출력의 크기는 다음과 같이 계산된다.

$$\text{Output} = (N - F) / \text{stride} + 1 = (9 - 3) / 1 + 1 = 7$$

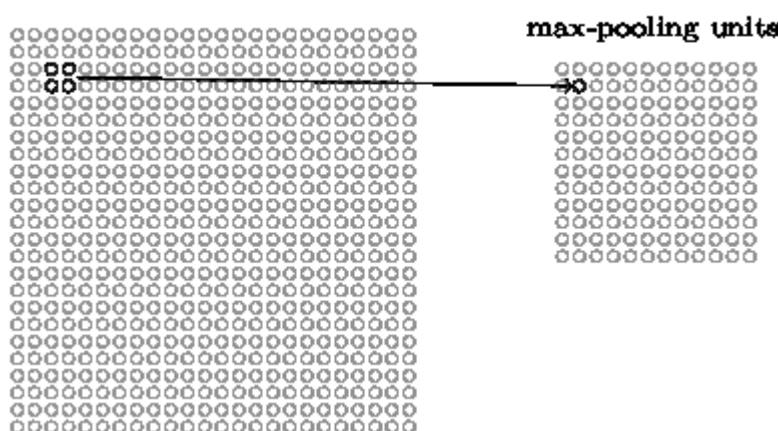
N=7 (패딩 P=1), F=3, S=2, Conv Layer K=2인 경우

그림참조: <https://medium.com/@phidaouss/convolutional-neural-networks-cnn-or-convnets-d7c688b0a207>

4.2 Pooling 연산



hidden neurons (output from feature map)



Pooling 연산(Max)

Single depth slice

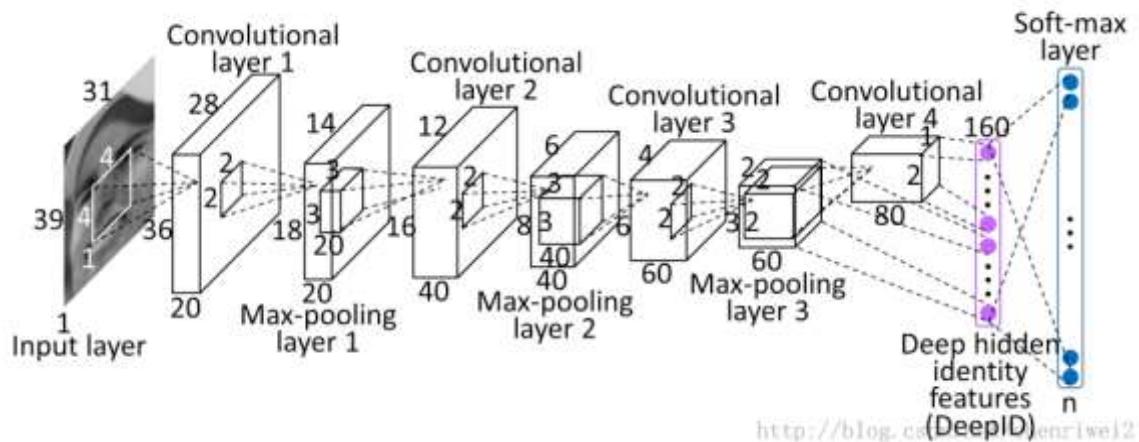
1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2x2 filters
and stride 2

6	8
3	4

아래의 그림은 지금까지의 내용을 전체적으로 설명해 주는 것이다.

Convolutional Neural Network(CNN)



그림출처: <http://www.cnblogs.com/yxwlf/p/3831310.html>

Output 크기 계산 과정

Input layer: $31 \times 39 \times 1$

Filter: $4 \times 4 \times 1 \rightarrow (31-4) / 1 + 1 = 28, (39-4) / 1 + 1 = 36 \rightarrow 28 \times 36$

Convolutional layer1: $28 \times 36 \times 20$

Filter: $2 \times 2 \times 20 \rightarrow (28-2) / 2 + 1 = 14, (36-2) / 2 + 1 = 18 \rightarrow 14 \times 18 \times 20$

Max-pooling layer1: $14 \times 18 \times 20$

Filter: $3 \times 3 \times 20 \rightarrow (14-3) / 1 + 1 = 12, (18-3) / 1 + 1 = 16 \rightarrow 12 \times 16 \times 40$

Convolutional layer2: $12 \times 16 \times 40$

Filter: $2 \times 2 \times 40 \rightarrow (12-2) / 2 + 1 = 6, (16-2) / 2 + 1 = 8 \rightarrow 6 \times 8 \times 40$

Max-pooling layer2: $6 \times 8 \times 40$

Filter: $3 \times 3 \times 40 \rightarrow (6-3) / 1 + 1 = 4, (8-3) / 1 + 1 = 6 \rightarrow 4 \times 6 \times 60$

Convolutional layer3: $4 \times 6 \times 60$

Filter: $2 \times 2 \times 60 \rightarrow (4-2) / 2 + 1 = 2, (6-2) / 2 + 1 = 3 \rightarrow 2 \times 3 \times 60$

Max-pooling layer3: $2 \times 3 \times 60$

Filter: $2 \times 2 \times 60 \rightarrow (2-2) / 1 + 1 = 1, (3-2) / 1 + 1 = 2$

Convolutional layer4: $1 \times 2 \times 80 \rightarrow 160 \rightarrow$ Fully Connection, Softmax

위의 Convolutional Neural Network(CNN)는 n 개의 입력 개체에 대해서 n-way softmax 분포의 예측 가능한 출력 결과가 나타난다. 위의 출력결과는 다음과 같은 수식(Softmax 계산)로 표현된다.

$$y_i = \frac{\exp(y'_i)}{\sum_{j=1}^n \exp(y'_j)}$$

여기서 아래의 값은

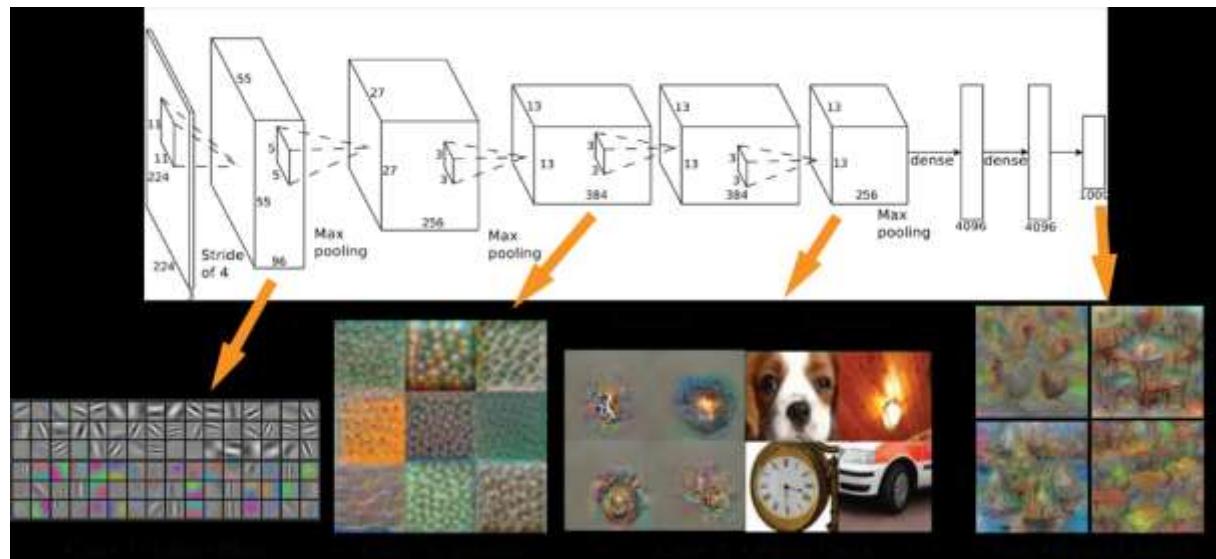
$$y'_j = \sum_{i=1}^{160} x_i \cdot w_{i,j} + b_j$$

160 개의 DeepID 들을 선형적으로 뮤어주는(곱하고 더함) 수식이다.

4.3 CNN 종류

4.3.1 AlexNet

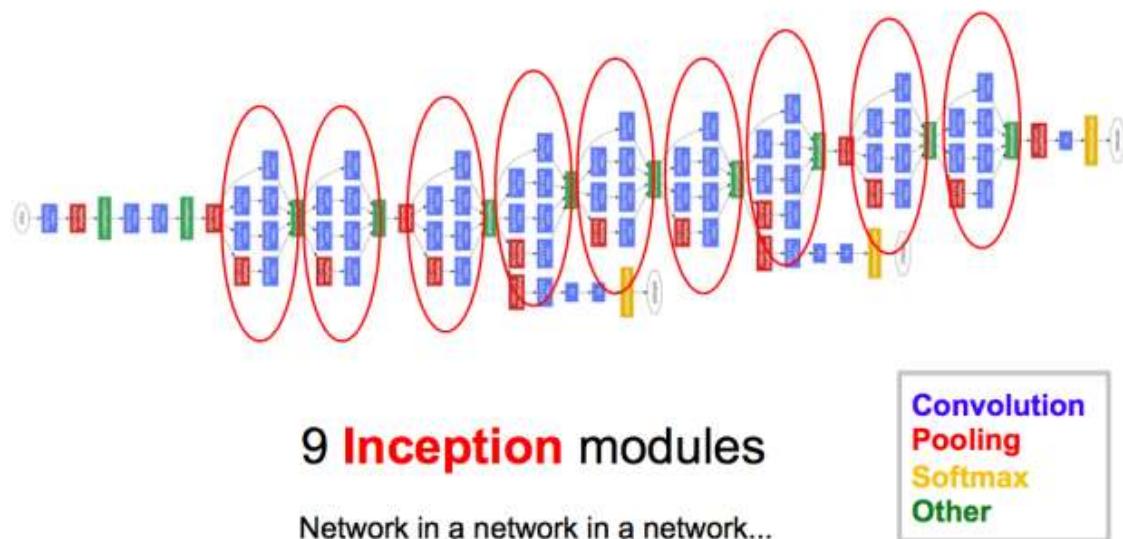
2012년 (오류율: 15.4%)



출처: <http://www.cc.gatech.edu/~hays/compvision/proj6/>

4.3.2 GoogLeNet

2014년 (오류율: 6.7%)

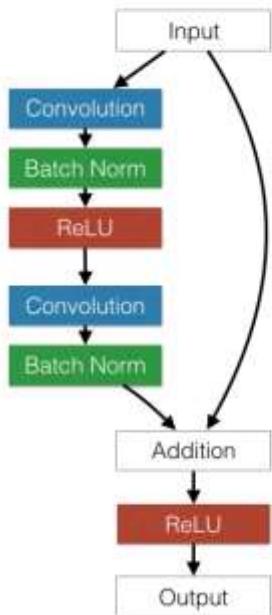


출처: <http://homes.cs.washington.edu/~jmschr/lectures/bioinformatics.html>

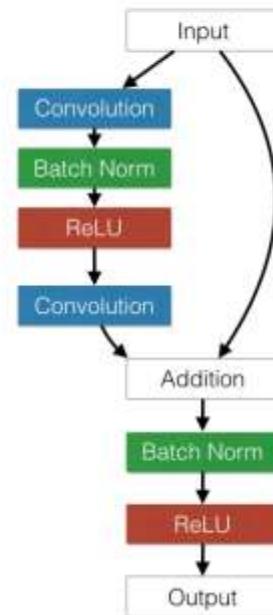
4.3.3 ResNet

2015년 (오류율: 3.6%)

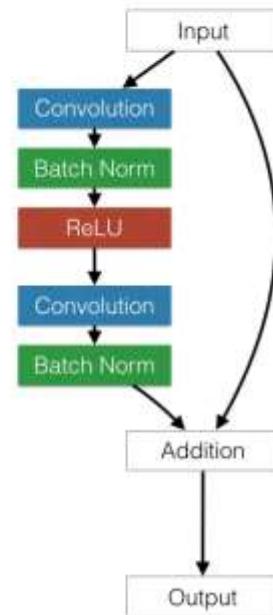
Reference paper



Batch Norm after add



No ReLU



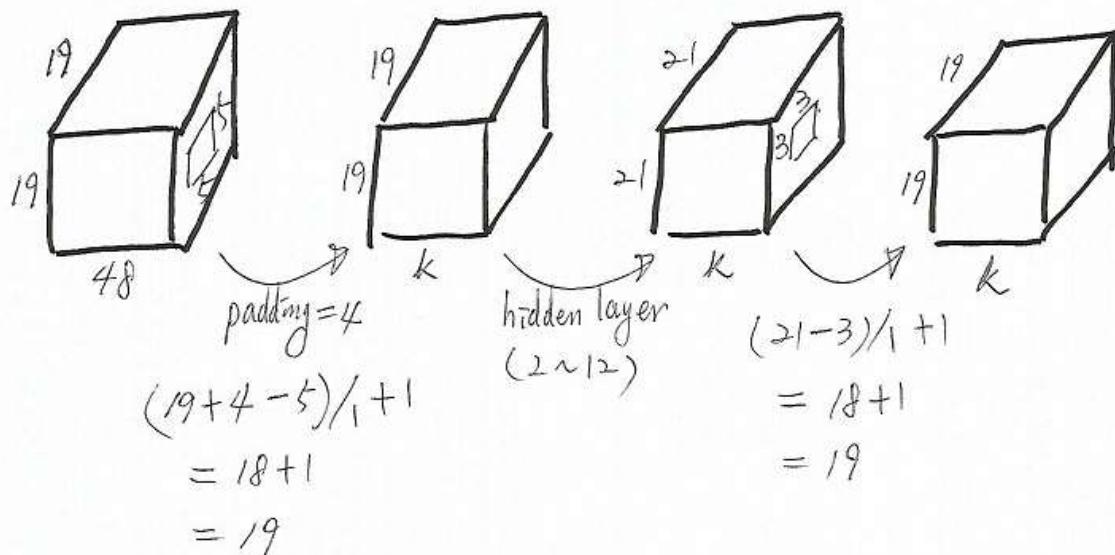
출처: <http://torch.ch/blog/2016/02/04/resnets.html>

4.3.4 DeepMind AlphaGo

2016년 3월, 머신러닝으로 무장한 알파고와 세계랭킹 1위인 이세돌 9단과의 바둑대결에서 알파고가 승리하면서 머신러닝/딥러닝이 다시 부각되기 시작했다. 아래의 내용은 필자가 인터넷에 있는 네이처 논문을 한글로 번역한 것이다. 알파고에 대해서 모든 내용을 상세히 분석해 볼 수는 없지만, 아래의 내용으로 바둑판에 펼쳐진 바둑돌들을 알파고가 CNN으로 어떻게 학습하는지 개념적으로 이해할 수 있다.

알파고의 정책망에 입력하는 이미지들은 $19 \times 19 \times 48$ 이며 48개의 feature planes들을 사용한다. 첫번째 hidden layer는 입력을 zero 패딩한 23×23 이미지이다. 이것을 stride 1로 한 5×5 크기의 필터 k 개로 convolve 연산하고 비선형적인 rectifier에 적용시킨다. 2에서 12까지 각각의 hidden layer는 순서대로 zero 패딩하여 21×21 이미지로 만든다. 그런 다음 stride 1로 한 3×3 크기의 필터 k 개로 convolve 연산하고 다시 비선형적인 rectifier에 적용시킨다. 마지막 layer는 stride 1로 한 1×1 크기의 필터 1개로 convolve 연산한다. 이때 각각의 위치에 대해서 다른 bias를 적용하여 softmax 함수를 수행한다. 경기용 버전의 알파고는 필터 $k=192$ 를 사용한다. 그리고 훈련할 때는 필터 $k=128, 256, 384$ 을 사용한다.

아래 그림은 위의 내용을 CNN으로 구성해본 것이다.



4.4 CNN 실습

위에서 기술한 Convolutional Neural Network 을 TensorFlow 라이브러리를 활용하는 파이썬 예제를 통하여 실습해 보자. 학습용 입력 데이터는 앞장에서 사용한 손글씨 데이터를 그대로 사용한다.

4.4.1 Adam Optimizer

아래의 예제는 Adam Optimizer 를 실습하는 코드이다.

파이썬 예제

```
'''  
A Convolutional Network implementation example using TensorFlow library.  
This example is using the MNIST database of handwritten digits (http://yann.lecun.com/exdb/mnist/)  
  
Author: Aymeric Damien  
Project: https://github.com/aymericdamien/TensorFlow-Examples/  
'''  
  
# Import MINST data  
import input_data  
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)  
  
import tensorflow as tf  
  
# Parameters  
learning_rate = 0.001  
training_iters = 100000  
batch_size = 128  
display_step = 10  
  
# Network Parameters  
n_input = 784 # MNIST data input (img shape: 28*28)  
n_classes = 10 # MNIST total classes (0-9 digits)  
dropout = 0.75 # Dropout, probability to keep units  
  
# tf Graph input  
x = tf.placeholder(tf.float32, [None, n_input])  
y = tf.placeholder(tf.float32, [None, n_classes])  
keep_prob = tf.placeholder(tf.float32) #dropout (keep probability)  
  
# Create model  
def conv2d(img, w, b):  
    return tf.nn.relu(tf.nn.bias_add(tf.nn.conv2d(img, w, strides=[1, 1, 1, 1], padding='SAME'), b))  
  
def max_pool(img, k):  
    return tf.nn.max_pool(img, ksize=[1, k, k, 1], strides=[1, k, k, 1], padding='SAME')  
  
def conv_net(_X, _weights, _biases, _dropout):
```

```

# Reshape input picture
_X = tf.reshape(_X, shape=[-1, 28, 28, 1])

# Convolution Layer
conv1 = conv2d(_X, _weights['wc1'], _biases['bc1'])
# Max Pooling (down-sampling)
conv1 = max_pool(conv1, k=2)
# Apply Dropout
conv1 = tf.nn.dropout(conv1, _dropout)

# Convolution Layer
conv2 = conv2d(conv1, _weights['wc2'], _biases['bc2'])
# Max Pooling (down-sampling)
conv2 = max_pool(conv2, k=2)
# Apply Dropout
conv2 = tf.nn.dropout(conv2, _dropout)

# Fully connected layer
dense1 = tf.reshape(conv2, [-1, _weights['wd1'].get_shape().as_list()[0]]) # Reshape conv2
output to fit dense layer input
dense1 = tf.nn.relu(tf.add(tf.matmul(dense1, _weights['wd1']), _biases['bd1'])) # Relu
activation
dense1 = tf.nn.dropout(dense1, _dropout) # Apply Dropout

# Output, class prediction
out = tf.add(tf.matmul(dense1, _weights['out']), _biases['out'])
return out

# Store layers weight & bias
weights = {
    'wc1': tf.Variable(tf.random_normal([5, 5, 1, 32])), # 5x5 conv, 1 input, 32 outputs
    'wc2': tf.Variable(tf.random_normal([5, 5, 32, 64])), # 5x5 conv, 32 inputs, 64 outputs
    'wd1': tf.Variable(tf.random_normal([7*7*64, 1024])), # fully connected, 7*7*64 inputs, 1024
outputs
    'out': tf.Variable(tf.random_normal([1024, n_classes])) # 1024 inputs, 10 outputs (class
prediction)
}

biases = {
    'bc1': tf.Variable(tf.random_normal([32])),
    'bc2': tf.Variable(tf.random_normal([64])),
    'bd1': tf.Variable(tf.random_normal([1024])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

# Construct model
pred = conv_net(x, weights, biases, keep_prob)

# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(pred, y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

# Evaluate model
correct_pred = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

```

```

# Initializing the variables
init = tf.initialize_all_variables()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)
    step = 1
    # Keep training until reach max iterations
    while step * batch_size < training_iters:
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        # Fit training using batch data
        sess.run(optimizer, feed_dict={x: batch_xs, y: batch_ys, keep_prob: dropout})
        if step % display_step == 0:
            # Calculate batch accuracy
            acc = sess.run(accuracy, feed_dict={x: batch_xs, y: batch_ys, keep_prob: 1.})
            # Calculate batch loss
            loss = sess.run(cost, feed_dict={x: batch_xs, y: batch_ys, keep_prob: 1.})
            print "Iter " + str(step*batch_size) + ", Minibatch Loss= " + "{:.6f}".format(loss) + ", "
            Training Accuracy= " + "{:.5f}".format(acc)
        step += 1
    print "Optimization Finished!"
    # Calculate accuracy for 256 mnist test images
    print "Testing Accuracy:", sess.run(accuracy, feed_dict={x: mnist.test.images[:256], y:
mnist.test.labels[:256], keep_prob: 1.})

```

실행 결과

```

Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /tmp/data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Iter 1280, Minibatch Loss= 39571.574219, Training Accuracy= 0.12500
Iter 2560, Minibatch Loss= 16663.375000, Training Accuracy= 0.35156
Iter 3840, Minibatch Loss= 12026.269531, Training Accuracy= 0.46094
Iter 5120, Minibatch Loss= 7629.645020, Training Accuracy= 0.57812
Iter 6400, Minibatch Loss= 5540.701660, Training Accuracy= 0.69531
Iter 7680, Minibatch Loss= 8531.413086, Training Accuracy= 0.67188
Iter 8960, Minibatch Loss= 6600.549805, Training Accuracy= 0.68750
Iter 10240, Minibatch Loss= 4401.302734, Training Accuracy= 0.74219
Iter 11520, Minibatch Loss= 2229.770508, Training Accuracy= 0.82812
Iter 12800, Minibatch Loss= 4814.633301, Training Accuracy= 0.71094
Iter 14080, Minibatch Loss= 1784.971191, Training Accuracy= 0.85156
Iter 15360, Minibatch Loss= 2439.172363, Training Accuracy= 0.78906
Iter 16640, Minibatch Loss= 3421.355957, Training Accuracy= 0.80469
Iter 17920, Minibatch Loss= 2168.399902, Training Accuracy= 0.85938
Iter 19200, Minibatch Loss= 2291.311523, Training Accuracy= 0.86719
Iter 20480, Minibatch Loss= 725.317810, Training Accuracy= 0.92969
Iter 21760, Minibatch Loss= 4109.961426, Training Accuracy= 0.77344

```

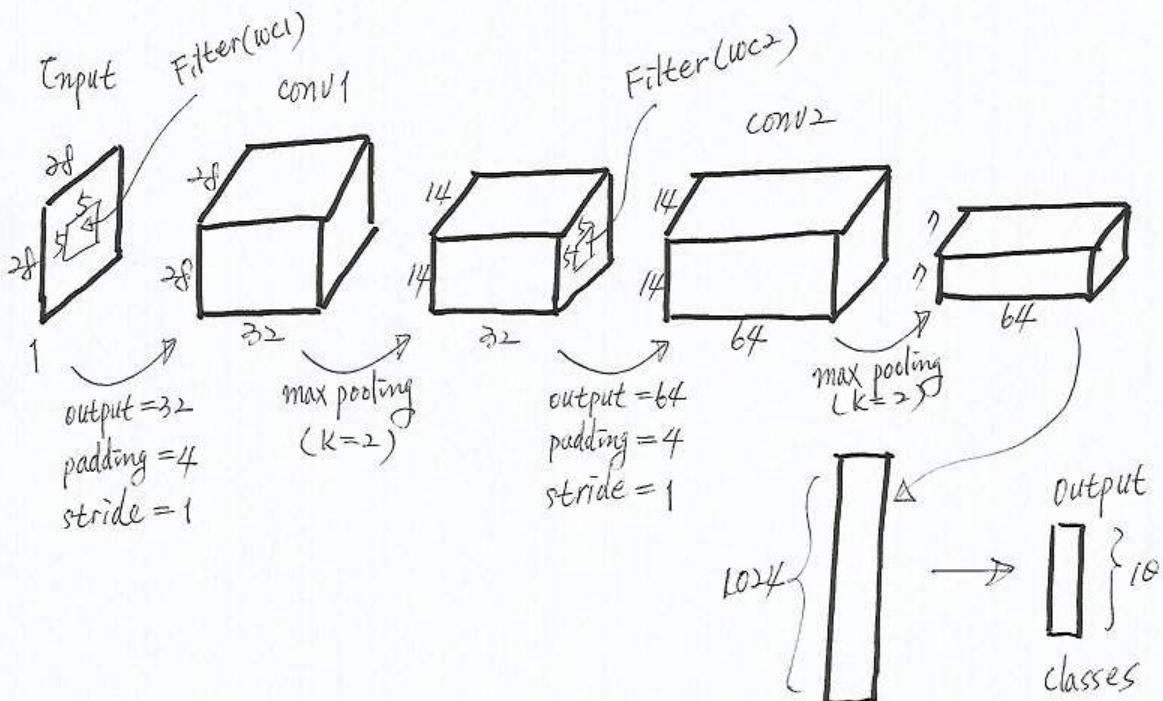
```
Iter 23040, Minibatch Loss= 1099.244385, Training Accuracy= 0.90625
Iter 24320, Minibatch Loss= 2067.941406, Training Accuracy= 0.86719
Iter 25600, Minibatch Loss= 2190.412109, Training Accuracy= 0.83594
Iter 26880, Minibatch Loss= 2738.618164, Training Accuracy= 0.83594
Iter 28160, Minibatch Loss= 1099.682129, Training Accuracy= 0.87500
Iter 29440, Minibatch Loss= 2316.953125, Training Accuracy= 0.85938
Iter 30720, Minibatch Loss= 1272.965332, Training Accuracy= 0.87500
Iter 32000, Minibatch Loss= 1216.170776, Training Accuracy= 0.89844
Iter 33280, Minibatch Loss= 1803.709351, Training Accuracy= 0.82812
Iter 34560, Minibatch Loss= 1358.294434, Training Accuracy= 0.84375
Iter 35840, Minibatch Loss= 598.053223, Training Accuracy= 0.93750
Iter 37120, Minibatch Loss= 1950.213013, Training Accuracy= 0.86719
Iter 38400, Minibatch Loss= 332.973206, Training Accuracy= 0.96875
Iter 39680, Minibatch Loss= 557.572266, Training Accuracy= 0.93750
Iter 40960, Minibatch Loss= 1528.193115, Training Accuracy= 0.89062
Iter 42240, Minibatch Loss= 1081.945801, Training Accuracy= 0.89844
Iter 43520, Minibatch Loss= 647.879456, Training Accuracy= 0.95312
Iter 44800, Minibatch Loss= 361.200653, Training Accuracy= 0.96094
Iter 46080, Minibatch Loss= 954.414917, Training Accuracy= 0.89844
Iter 47360, Minibatch Loss= 1267.340820, Training Accuracy= 0.84375
Iter 48640, Minibatch Loss= 1475.328613, Training Accuracy= 0.86719
Iter 49920, Minibatch Loss= 991.829285, Training Accuracy= 0.88281
Iter 51200, Minibatch Loss= 757.160095, Training Accuracy= 0.90625
Iter 52480, Minibatch Loss= 371.265869, Training Accuracy= 0.93750
Iter 53760, Minibatch Loss= 330.319244, Training Accuracy= 0.96094
Iter 55040, Minibatch Loss= 1315.269043, Training Accuracy= 0.85938
Iter 56320, Minibatch Loss= 1555.724609, Training Accuracy= 0.91406
Iter 57600, Minibatch Loss= 591.706787, Training Accuracy= 0.92969
Iter 58880, Minibatch Loss= 485.850128, Training Accuracy= 0.92969
Iter 60160, Minibatch Loss= 700.377136, Training Accuracy= 0.92188
Iter 61440, Minibatch Loss= 919.180054, Training Accuracy= 0.89844
Iter 62720, Minibatch Loss= 942.675964, Training Accuracy= 0.89844
Iter 64000, Minibatch Loss= 1475.131592, Training Accuracy= 0.89062
Iter 65280, Minibatch Loss= 1042.180420, Training Accuracy= 0.89844
Iter 66560, Minibatch Loss= 975.437012, Training Accuracy= 0.89844
Iter 67840, Minibatch Loss= 617.484192, Training Accuracy= 0.92188
Iter 69120, Minibatch Loss= 529.599976, Training Accuracy= 0.92969
Iter 70400, Minibatch Loss= 851.615662, Training Accuracy= 0.90625
Iter 71680, Minibatch Loss= 811.685059, Training Accuracy= 0.89062
Iter 72960, Minibatch Loss= 1222.854614, Training Accuracy= 0.89062
Iter 74240, Minibatch Loss= 829.171814, Training Accuracy= 0.91406
Iter 75520, Minibatch Loss= 402.435944, Training Accuracy= 0.93750
Iter 76800, Minibatch Loss= 248.807281, Training Accuracy= 0.93750
Iter 78080, Minibatch Loss= 392.773804, Training Accuracy= 0.94531
Iter 79360, Minibatch Loss= 696.895386, Training Accuracy= 0.89844
Iter 80640, Minibatch Loss= 292.426575, Training Accuracy= 0.96094
Iter 81920, Minibatch Loss= 1126.807617, Training Accuracy= 0.89844
Iter 83200, Minibatch Loss= 652.764160, Training Accuracy= 0.91406
Iter 84480, Minibatch Loss= 609.673096, Training Accuracy= 0.92969
Iter 85760, Minibatch Loss= 652.681702, Training Accuracy= 0.90625
Iter 87040, Minibatch Loss= 890.243774, Training Accuracy= 0.91406
Iter 88320, Minibatch Loss= 1083.500366, Training Accuracy= 0.92188
Iter 89600, Minibatch Loss= 356.112671, Training Accuracy= 0.93750
Iter 90880, Minibatch Loss= 299.382935, Training Accuracy= 0.92969
Iter 92160, Minibatch Loss= 495.023956, Training Accuracy= 0.93750
Iter 93440, Minibatch Loss= 729.896057, Training Accuracy= 0.89844
```

```

Iter 94720, Minibatch Loss= 480.458221, Training Accuracy= 0.94531
Iter 96000, Minibatch Loss= 570.205078, Training Accuracy= 0.89062
Iter 97280, Minibatch Loss= 870.419128, Training Accuracy= 0.89844
Iter 98560, Minibatch Loss= 313.707184, Training Accuracy= 0.93750
Iter 99840, Minibatch Loss= 624.721680, Training Accuracy= 0.92188
Optimization Finished!
Testing Accuracy: 0.949219

```

위의 TensorFlow 예제는 다음과 같이 CNN 연산을 수행한다.



4.4.2 RMS Optimizer

아래의 예제는 RMS Optimizer를 실습하는 코드이다.

파이썬 예제

```
'''  
A Convolutional Network implementation example using TensorFlow library.  
This example is using the MNIST database of handwritten digits (http://yann.lecun.com/exdb/mnist/)  
  
Author: Aymeric Damien  
Project: https://github.com/aymericdamien/TensorFlow-Examples/  
'''  
  
# Import MINST data  
import input_data  
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)  
  
import tensorflow as tf  
  
# Parameters  
learning_rate = 0.001  
training_iters = 100000  
batch_size = 128  
display_step = 10  
  
# Network Parameters  
n_input = 784 # MNIST data input (img shape: 28*28)  
n_classes = 10 # MNIST total classes (0-9 digits)  
dropout = 0.75 # Dropout, probability to keep units  
  
# tf Graph input  
x = tf.placeholder(tf.float32, [None, n_input])  
y = tf.placeholder(tf.float32, [None, n_classes])  
keep_prob = tf.placeholder(tf.float32) #dropout (keep probability)  
  
# Create model  
def conv2d(img, w, b):  
    return tf.nn.relu(tf.nn.bias_add(tf.nn.conv2d(img, w, strides=[1, 1, 1, 1], padding='SAME'), b))  
  
def max_pool(img, k):  
    return tf.nn.max_pool(img, ksize=[1, k, k, 1], strides=[1, k, k, 1], padding='SAME')  
  
def conv_net(_X, _weights, _biases, _dropout):  
    # Reshape input picture  
    _X = tf.reshape(_X, shape=[-1, 28, 28, 1])  
  
    # Convolution Layer  
    conv1 = conv2d(_X, _weights['wc1'], _biases['bc1'])  
    # Max Pooling (down-sampling)  
    conv1 = max_pool(conv1, k=2)  
    # Apply Dropout  
    conv1 = tf.nn.dropout(conv1, _dropout)  
    ...
```

```
# Convolution Layer
conv2 = conv2d(conv1, _weights['wc2'], _biases['bc2'])
# Max Pooling (down-sampling)
conv2 = max_pool(conv2, k=2)
# Apply Dropout
conv2 = tf.nn.dropout(conv2, _dropout)

# Fully connected layer
dense1 = tf.reshape(conv2, [-1, _weights['wd1'].get_shape().as_list()[0]]) # Reshape conv2
output to fit dense layer input
dense1 = tf.nn.relu(tf.add(tf.matmul(dense1, _weights['wd1']), _biases['bd1'])) # Relu
activation
dense1 = tf.nn.dropout(dense1, _dropout) # Apply Dropout

# Output, class prediction
out = tf.add(tf.matmul(dense1, _weights['out']), _biases['out'])
return out

# Store layers weight & bias
weights = {
    'wc1': tf.Variable(tf.random_normal([5, 5, 1, 32])), # 5x5 conv, 1 input, 32 outputs
    'wc2': tf.Variable(tf.random_normal([5, 5, 32, 64])), # 5x5 conv, 32 inputs, 64 outputs
    'wd1': tf.Variable(tf.random_normal([7*7*64, 1024])), # fully connected, 7*7*64 inputs, 1024
outputs
    'out': tf.Variable(tf.random_normal([1024, n_classes])) # 1024 inputs, 10 outputs (class
prediction)
}

biases = {
    'bc1': tf.Variable(tf.random_normal([32])),
    'bc2': tf.Variable(tf.random_normal([64])),
    'bd1': tf.Variable(tf.random_normal([1024])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

# Construct model
pred = conv_net(x, weights, biases, keep_prob)

# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(pred, y))
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate).minimize(cost)

# Evaluate model
correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initializing the variables
init = tf.initialize_all_variables()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)
    step = 1
    # Keep training until reach max iterations
    while step * batch_size < training_iters:
```

```

batch_xs, batch_ys = mnist.train.next_batch(batch_size)
# Fit training using batch data
sess.run(optimizer, feed_dict={x: batch_xs, y: batch_ys, keep_prob: dropout})
if step % display_step == 0:
    # Calculate batch accuracy
    acc = sess.run(accuracy, feed_dict={x: batch_xs, y: batch_ys, keep_prob: 1.})
    # Calculate batch loss
    loss = sess.run(cost, feed_dict={x: batch_xs, y: batch_ys, keep_prob: 1.})
    print "Iter " + str(step*batch_size) + ", Minibatch Loss= " + "{:.6f}".format(loss) + ", "
Training Accuracy= " + "{:.5f}" .format(acc)
step += 1
print "Optimization Finished!"
# Calculate accuracy for 256 mnist test images
print "Testing Accuracy:", sess.run(accuracy, feed_dict={x: mnist.test.images[:256], y:
mnist.test.labels[:256], keep_prob: 1.})

```

실행 결과

```

Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Iter 1280, Minibatch Loss= 15044.337891, Training Accuracy= 0.30469
Iter 2560, Minibatch Loss= 7866.206055, Training Accuracy= 0.55469
Iter 3840, Minibatch Loss= 5286.229004, Training Accuracy= 0.60156
Iter 5120, Minibatch Loss= 4336.242188, Training Accuracy= 0.65625
Iter 6400, Minibatch Loss= 2423.458008, Training Accuracy= 0.78125
Iter 7680, Minibatch Loss= 5367.664062, Training Accuracy= 0.68750
Iter 8960, Minibatch Loss= 2561.865723, Training Accuracy= 0.80469
Iter 10240, Minibatch Loss= 2611.284424, Training Accuracy= 0.77344
Iter 11520, Minibatch Loss= 1460.441162, Training Accuracy= 0.85938
Iter 12800, Minibatch Loss= 3867.540771, Training Accuracy= 0.72656
Iter 14080, Minibatch Loss= 1045.453857, Training Accuracy= 0.85156
Iter 15360, Minibatch Loss= 859.744629, Training Accuracy= 0.92188
Iter 16640, Minibatch Loss= 1653.588379, Training Accuracy= 0.86719
Iter 17920, Minibatch Loss= 1299.167358, Training Accuracy= 0.88281
Iter 19200, Minibatch Loss= 870.486572, Training Accuracy= 0.91406
Iter 20480, Minibatch Loss= 512.209473, Training Accuracy= 0.95312
Iter 21760, Minibatch Loss= 2658.354492, Training Accuracy= 0.79688
Iter 23040, Minibatch Loss= 1012.323364, Training Accuracy= 0.90625
Iter 24320, Minibatch Loss= 990.037354, Training Accuracy= 0.90625
Iter 25600, Minibatch Loss= 1193.083008, Training Accuracy= 0.86719
Iter 26880, Minibatch Loss= 898.115662, Training Accuracy= 0.90625
Iter 28160, Minibatch Loss= 980.036255, Training Accuracy= 0.92188
Iter 29440, Minibatch Loss= 1118.619385, Training Accuracy= 0.89844
Iter 30720, Minibatch Loss= 863.524597, Training Accuracy= 0.91406
Iter 32000, Minibatch Loss= 646.047241, Training Accuracy= 0.92188
Iter 33280, Minibatch Loss= 780.657959, Training Accuracy= 0.93750
Iter 34560, Minibatch Loss= 191.474686, Training Accuracy= 0.96094
Iter 35840, Minibatch Loss= 248.128632, Training Accuracy= 0.96094
Iter 37120, Minibatch Loss= 916.254883, Training Accuracy= 0.88281
Iter 38400, Minibatch Loss= 291.823303, Training Accuracy= 0.96875

```

```
Iter 39680, Minibatch Loss= 190.094574, Training Accuracy= 0.96094
Iter 40960, Minibatch Loss= 1092.242432, Training Accuracy= 0.92188
Iter 42240, Minibatch Loss= 225.435638, Training Accuracy= 0.95312
Iter 43520, Minibatch Loss= 306.406036, Training Accuracy= 0.94531
Iter 44800, Minibatch Loss= 157.786072, Training Accuracy= 0.96875
Iter 46080, Minibatch Loss= 163.582169, Training Accuracy= 0.97656
Iter 47360, Minibatch Loss= 542.392700, Training Accuracy= 0.93750
Iter 48640, Minibatch Loss= 795.955566, Training Accuracy= 0.91406
Iter 49920, Minibatch Loss= 323.505341, Training Accuracy= 0.91406
Iter 51200, Minibatch Loss= 144.104416, Training Accuracy= 0.95312
Iter 52480, Minibatch Loss= 198.293732, Training Accuracy= 0.94531
Iter 53760, Minibatch Loss= 144.998291, Training Accuracy= 0.97656
Iter 55040, Minibatch Loss= 269.541962, Training Accuracy= 0.96094
Iter 56320, Minibatch Loss= 162.895889, Training Accuracy= 0.96094
Iter 57600, Minibatch Loss= 134.568771, Training Accuracy= 0.96875
Iter 58880, Minibatch Loss= 305.590485, Training Accuracy= 0.94531
Iter 60160, Minibatch Loss= 233.371307, Training Accuracy= 0.92969
Iter 61440, Minibatch Loss= 313.154907, Training Accuracy= 0.96094
Iter 62720, Minibatch Loss= 509.048218, Training Accuracy= 0.94531
Iter 64000, Minibatch Loss= 124.118073, Training Accuracy= 0.95312
Iter 65280, Minibatch Loss= 262.470703, Training Accuracy= 0.95312
Iter 66560, Minibatch Loss= 494.242615, Training Accuracy= 0.89844
Iter 67840, Minibatch Loss= 202.761673, Training Accuracy= 0.95312
Iter 69120, Minibatch Loss= 498.557678, Training Accuracy= 0.92188
Iter 70400, Minibatch Loss= 254.279037, Training Accuracy= 0.94531
Iter 71680, Minibatch Loss= 604.011780, Training Accuracy= 0.94531
Iter 72960, Minibatch Loss= 264.961914, Training Accuracy= 0.92969
Iter 74240, Minibatch Loss= 228.085358, Training Accuracy= 0.94531
Iter 75520, Minibatch Loss= 389.408661, Training Accuracy= 0.92188
Iter 76800, Minibatch Loss= 509.473572, Training Accuracy= 0.89844
Iter 78080, Minibatch Loss= 146.448151, Training Accuracy= 0.96094
Iter 79360, Minibatch Loss= 522.951355, Training Accuracy= 0.92188
Iter 80640, Minibatch Loss= 589.985962, Training Accuracy= 0.90625
Iter 81920, Minibatch Loss= 84.166092, Training Accuracy= 0.98438
Iter 83200, Minibatch Loss= 335.588135, Training Accuracy= 0.92188
Iter 84480, Minibatch Loss= 307.818512, Training Accuracy= 0.92969
Iter 85760, Minibatch Loss= 512.624939, Training Accuracy= 0.94531
Iter 87040, Minibatch Loss= 138.497787, Training Accuracy= 0.96094
Iter 88320, Minibatch Loss= 436.226044, Training Accuracy= 0.91406
Iter 89600, Minibatch Loss= 444.379669, Training Accuracy= 0.93750
Iter 90880, Minibatch Loss= 197.094330, Training Accuracy= 0.95312
Iter 92160, Minibatch Loss= 241.629578, Training Accuracy= 0.92969
Iter 93440, Minibatch Loss= 280.125793, Training Accuracy= 0.96875
Iter 94720, Minibatch Loss= 454.368347, Training Accuracy= 0.92969
Iter 96000, Minibatch Loss= 438.901794, Training Accuracy= 0.92969
Iter 97280, Minibatch Loss= 171.418945, Training Accuracy= 0.95312
Iter 98560, Minibatch Loss= 435.348389, Training Accuracy= 0.92188
Iter 99840, Minibatch Loss= 162.014038, Training Accuracy= 0.97656
Optimization Finished!
Testing Accuracy: 0.964844
```

4.4.3 결과 정리

지금까지 실습한 예제에서 산출되는 정확도를 테이블로 정리하면 다음과 같다.

CNN 옵티마이저	실행 시간	정확도
Adam Optimizer	90 분	0.949
RMS Optimizer	84 분	0.965

따라서 RMS Optimizer 를 사용한 학습이 좀더 정확도가 높다는 것을 알 수 있다.

♣ 책에 있는 모든 소스들은 아래 링크에서 다운로드 가능합니다.

<https://github.com/kernel-bz/ml>

머신러닝/딥러닝 TensorFlow 실습

Third Edition

5. 언어 인지(RNN)

5. 언어 인지(RNN)

5.1 언어 처리

5.2 언어 변환

5.3 언어 번역

5.4 언어 인식

5.1 언어 처리

단어 카운팅, 단어 분류, 통계 처리, 문장 유사도 측정.

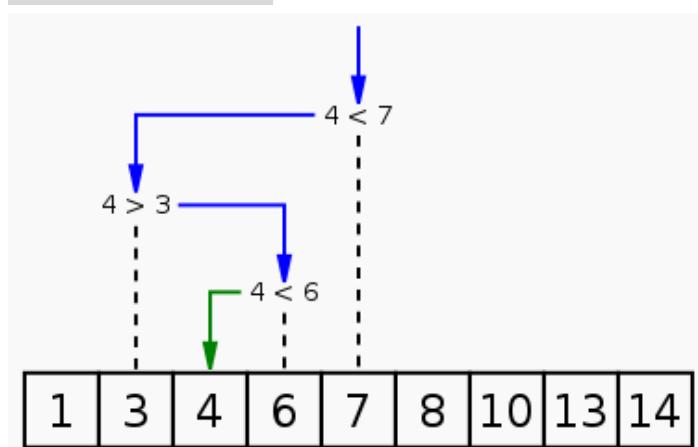
전통적인 자료구조 알고리즘으로 가능, 인덱싱으로 빠른 처리, 정확성이 높음.

5.1.1 전통적인 자료구조 알고리즘 특징

전통적인 자료구조 알고리즘은 데이터들을 트리구조로 인덱스 분류하여 탐색해 나가는 방식이다.

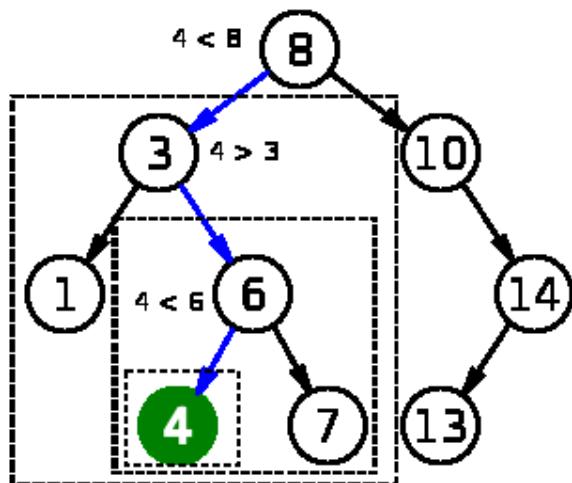
이 알고리즘은 정해진 답이 있는 것을 빠르게 검색하는 방식으로 동작한다.

트리탐색 알고리즘



그림출처: <https://www.kullabs.com/classes/subjects/units/lessons/notes/note-detail/3861>

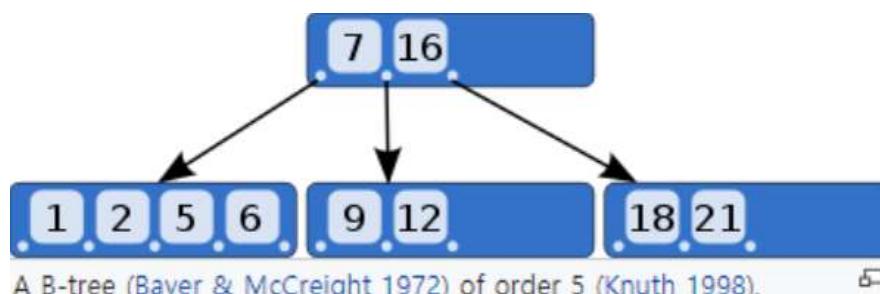
Binary Search Tree 탐색 알고리즘



그림출처: <https://www.kullabs.com/classes/subjects/units/lessons/note-detail/3861>

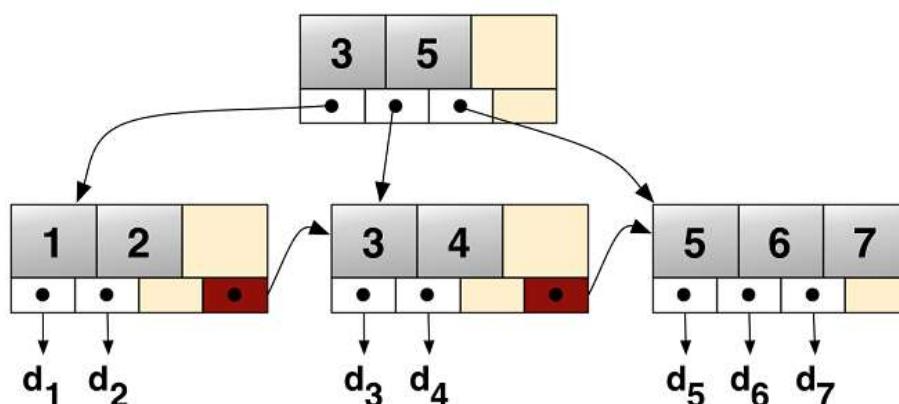
B-Tree 자료구조 알고리즘

<https://en.wikipedia.org/wiki/B-tree>



B+Tree 자료구조 알고리즘

https://en.wikipedia.org/wiki/B%2B_tree



5.1.2 언어(문장) 유사도 측정

n-gram: 문장내에서 단어들의 빈도수 측정.

TF-IDF: 문서 내에서 어떤 단어가 중요한가를 측정.

참고 문서::

위키피디아 문서: <https://en.wikipedia.org/wiki/Tf%E2%80%93idf>

파이썬 소스: <http://secmem.tistory.com/670>

C 언어 소스: <https://github.com/brendano/bow/blob/master/tfidf.c>

n-gram 알고리즘은 n 개의 문자열을 원쪽에서 오른쪽으로 한 단위씩 움직이며 추출되는 시퀀스의 집합의 출현 빈도수를 기록한다. 이때 n 은 얼마만큼의 단위로 잘라낼지를 나타내는 지표인데, 이 값이 1 이면 unigram, 2 이면 bigram, 3 이면 trigram 이라 부르며, 그 값은 더 커질 수 있다.

ex) "I am a boy" 라는 문장을 문자 단위의 3-gram 으로 만든다면,

"I a", " am", "am ", "m a", " a", "a b", " bo", "boy" 로 만들 수 있다.

같은 작업을 단어 단위로도 할 수 있는데, 만약 2-gram 으로 만든다면, "I am", "am a", "a boy" 가 된다. 여기서 띄어 쓰기를 포함할지 말지, 단어단위로 할지 문자단위로 할지는 전적으로 설계자가 정한 규칙대로 하면 된다.

is 와 have 와 같이 모든 문서에 공통적으로 자주 나오는 단어라면 중요한 단어로 판단하기가 힘들다. 이러한 문제를 해결하기 위해서 해당 문서군을 통째로 검사하여, 해당 단어의 중요도를 낮추고, 문서 내에서 유일하게 많이 나오는 키워드의 중요성을 나타내 주는 기법이 TF-IDF 방법이다.

TF 는 단어빈도(Term Frequency)를 의미한다. 특정 단어가 문서 내에 얼마나 빈도로 등장하는지를 나타낸다. 당연히 문서에 자주 등장하는 단어라면 그만큼 중요도가 높다고 예상할 수 있다.

DF 는 문서빈도(Document Frequency)로서 문서군 내에서 해당 단어가 나타난 문서의 수 이다. 따라서 각 단어가 그 문서에서 얼마나 나타났는지는 중요하지 않고, 몇개의 문서에서 나타났는지가 중요하다. 해당 문서군의 기준은 보통 폴더를 기준으로 한다. (이건 설계자 마음대로) df 값이 높다는 것은 해당 문서군 내에서 많은 문서에서 등장하는 것이므로, 별로 중요한 단어가 아니라는 것을 나타낸다. 즉 is, have 등과 같이 어느 곳에서나 쓰이는 단어일 수록 DF 값이 높기 마련이다.

IDF 는 Inverse Document Frequency 로 앞서 설명한 df 에 역수를 취한 것이다. 즉 앞서 설명드렸듯, df 는 값이 클 수록 중요하지 않은 단어를 나타내는 것이다. 이 값을 역수로 취하면 $(1/DF)$ DF 값이 클수록 값이 작아지게 되는 것이다. 즉 IDF 값이 클수록 다른 문서에서 잘 등장하는 않는 단어라는 것을 뜻한다. 참고로 DF 값은 굉장히 넓은 범위로 구성되어 있어 보통 \log 를 취한다.

계산식 요약 설명

TF = 문서내 단어의 개수 / 문서내 모든 단어수

DF = 단어를 포함한 문서의 수 / 문서 전체 개수

IDF = $\log(1/DF)$

TF-IDF = TF * IDF

공통 키워드 유사성:

코사인 유사도(Cosine Similarity)

계산 예제

아래의 위키피디아 예제 참조:

<https://en.wikipedia.org/wiki/Tf%E2%80%93idf>

5.1.3 빠른 검색으로 문장 연결

아래 내용은 필자가 “맞춤형 문장 자동 번역 시스템 및 이를 위한 데이터베이스 구축방법”을 C 언어로 개발한 것에 관한 것이며, 2012년도에 특허청에 특허 등록 및 공개 되어 있다.

특허청(<http://www.kipo.go.kr>) 참조:

출원번호 (날짜): 1020110045634 (2011.05.16)

등록공개번호 (날짜): 1020120127876 (2012.11.26)

아래 내용은 위의 알고리즘을 컴퓨터 프로그래밍 언어인 C 언어로 구현하여 증명하는 내용으로 구성한다. 이 알고리즘은 영어 문장과 한글 문장을 상호 번역하는 방법에 관한 것이다. 번역된 문장을 구성하는 단어들에 고유번호를 부여하고 이것을 색인사전에 저장한 것을 빠르게 검색하는 방식으로 문장을 번역한다. 문장을 상호 번역하기 위해서 영어단어 색인번호와 한글단어 색인번호를 서로 맞춤형 자료구조로 저장하고 이것을 빠르게 검색하는 알고리즘이 새로운 발명이며 특히 출원 대상이다. 이것을 “번역 맞춤형 단어색인 검색 알고리즘”이라 하고 “TWI”라는 명칭을 부여한다. TWI는 C 언어로 프로그래밍 되었으며 B+Tree 구조의 단점을 보완하여 문장 번역을 위한 맞춤형으로 최적화 시켰다. 개발한 프로그램의 명칭은 “TransWorks”이다. TransWorks에서 TWI의 수행 시간은 산출되었으며 아래와 같은 수식이 도출되었다.

$$M = (\log(n) \times 10) / 2$$

$$T = M \times c$$

$$TS = w \times T$$

$$TWI \text{ 검색시간(초)} : TWS = 2TS + T$$

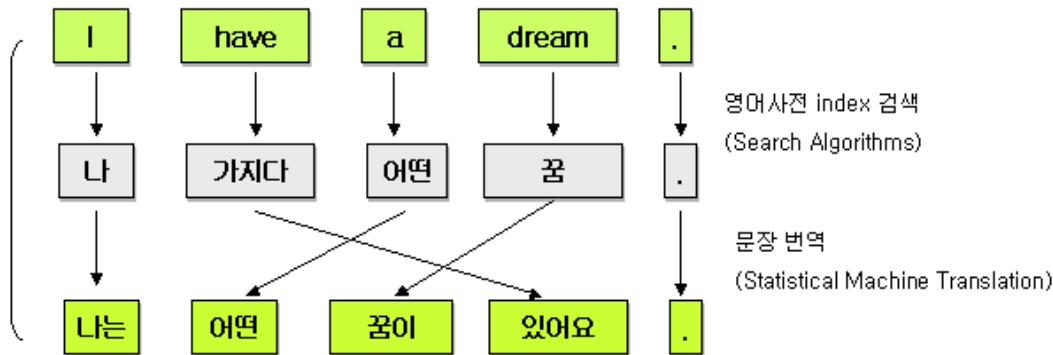
여기서 n은 색인번호의 개수, M은 비교회수, w는 문장안의 단어개수, \log 함수의 밑수는 10이고, c는 단어 하나를 검색하는데 소요되는 시간으로 컴퓨터 처리속도에 의존하는 값이다. 수식의 의미와 산출 결과값들은 아래에서 자세히 설명된다.

핵심용어 정리:

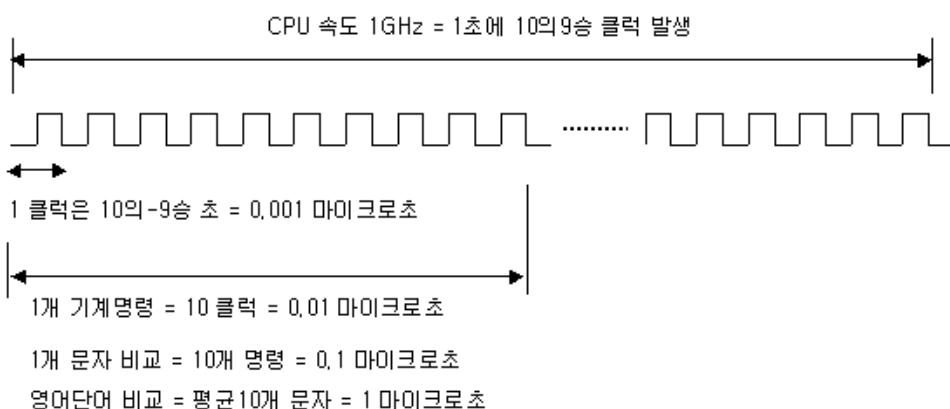
- * 색인번호: 문장을 구성하는 단어들에게 부여한 고유번호.
- * 색인사전: 색인번호를 저장한 사전.
- * 번역 맞춤형 단어색인: 영어단어 색인번호와 한글단어 색인번호를 서로 연결하는 자료구조.
- * TWI (TransWorks Index): 번역 맞춤형 단어색인 검색 알고리즘.
- * TransWorks: TWI를 C언어로 개발한 프로그램.

TWI 소개

일반적으로 문장 번역은 아래 그림과 같은 방식으로 수행하며 이때 문장을 구성하고 있는 단어들을 검색하는 방식이 중요한 평가 요소가 된다.



옥스포드 영어사전에 등재되어 있는 영어단어 수는 약30만(3×10^5)개 라고 알려져 있다. 일상생활에서 통용되는 단어(방언, 신조어 등.)들까지 합하면 약100만(10^6)개의 단어가 있다고 예상된다. 영어단어 100만개, 한글단어 100만개를 서로 연결시킨 순차적인 자료구조로 컴퓨터 메모리에 영한-한영 사전을 구축했을 때, 일반적인 단어 검색 시간에 대해서 먼저 정리한다.



요즘 컴퓨터의 CPU 처리속도는 보통 수 GHz이다. 즉, CPU 내부에서 1 클럭(디지털 신호가 On/Off로 변화되는 클럭)은 약 10의-9승초(0.001마이크로초)가 걸린다는 것이다. CPU 제조사마다 조금씩 다르겠지만, 1개의 기계명령이 10클럭으로 설계되었다면, 하나의 명령을 수행하는데

0.01마이크로초가 소요된다. 여기서, 하나의 문자를 비교하는데 10개의 기계명령이 필요하다면, 하나의 문자 비교에 0.1마이크로(10의-7승)초가 소요된다. 영어단어가 평균 10개의 문자로 구성되었다고 보면, 하나의 영어단어를 비교하는데 약 1마이크로(10의-6승)초가 걸린다고 계산할 수 있으며 이것이 상수 c 값(컴퓨터 처리속도에 따라 달라짐)이다. 그렇다면, 컴퓨터 메모리에 100만개의 단어가 일반적인 형태의 순차 배열로 저장되어 있다고 할 때, 단어 검색 시간을 아래와 같은 수식으로 계산할 수 있다.

$$N = n$$

$$A = N \times c$$

여기서, n 은 단어 사전안의 단어개수, N 은 단어비교 회수이며, c 는 단어 하나를 비교하는데 소요되는 시간으로 위에서 산출한 1마이크로(10의-6승)초 이다.

- (1) **최선의 경우:** 영어단어 1번 비교($N = 1$): $A_1 = 1 \times c = 1$ 마이크로(10의-6승)초
- (2) **최악의 경우:** 영어단어 100만번 비교($N = 10^6$):

$$A_2 = (10^6) \times c = (10^6) \times (10^{-6}) = 1$$
초
- (3) **평균:** $A_3 = A_2 / 2 = 1$ 초 / 2 = 0.5초

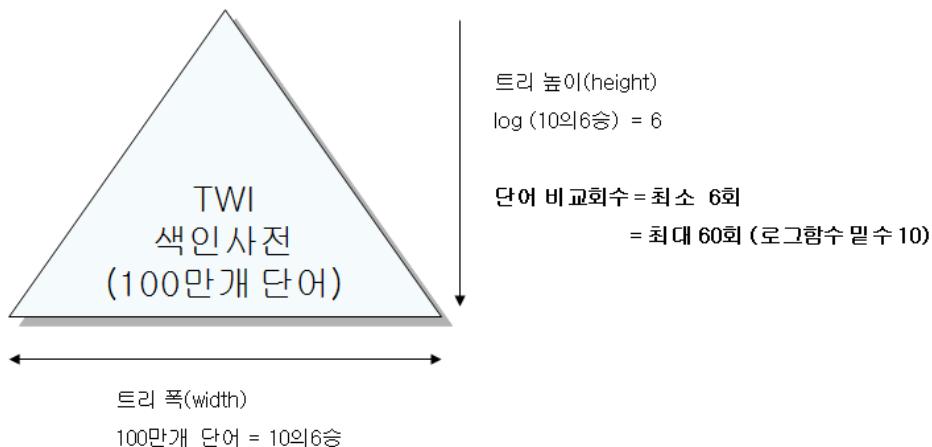
위의 결과는 100만개의 단어가 컴퓨터 메모리에 순차적인 배열형태로 나열되어 있는 자료구조에서 하나의 단어를 순차 검색했을때의 결과이다. 영어문장은 보통 수십개의 단어들로 구성되어 있다. 따라서, 문장안에 있는 단어들을 모두 검색하는데 소요되는 시간은 아래와 같은 수식으로 계산할 수 있다.

$$AS = w \times N \times c = w \times A_3$$

여기서, w 는 문장안의 단어개수, N 은 단어비교 회수이며, c 는 단어 하나를 비교하는데 소요되는 시간으로 위에서 산출한 1마이크로(10의-6승)초 이다.

예를들어 10개의 단어들로 구성된 문장의 단어들을 모두 검색한다면, $w=10$, $A_3 = 0.5$ 초 이므로 $AS = 10 \times 0.5$ 초 = 5초가 된다. 5초라는 시간은 요즘의 컴퓨팅 환경에서 상당히 긴 시간이다. 따라서, 문장 번역을 빠르게 하기 위해서는 좀더 효율적인 자료구조와 알고리즘이 필요하다.

TWI는 Tree구조를 바탕으로 한 B+Tree 구조를 개선한 “번역 맞춤형 단어색인 검색 알고리즘”이다. TWI의 단어 검색속도는 아래와 같이 산출된다.



100만개의 단어에 색인번호를 부여하여 TWI 색인사전에 저장하면 Tree의 높이(height)는 $\log(10의6승)$, 즉 6이 된다. (\log 함수의 밑수는 10). 높이(height)는 100만개의 단어중에서 한 개 단어를 검색할 때 단어 비교 회수 N이며, 검색 시간은 아래와 같이 계산된다.

$$M_1 = \log(n)$$

$$T_1 = M_1 \times c$$

$$M_2 = \log(n) \times 10$$

$$T_2 = M_2 \times c$$

$$(1) \text{ 최소: } T_1 = \log(10의6승) \times 1\text{마이크로초} = 6 \times 1\text{마이크로초} = 6\text{마이크로초} = 0.006\text{미리초}$$

$$(2) \text{ 최대: } T_2 = \log(10의6승) \times 10 \times 1\text{마이크로초} = 60\text{마이크로초} = 0.06\text{미리초}$$

$$(3) \text{ 평균: } T = T_2 / 2 = 0.03\text{미리초}$$

문장 안에 단어가 10개로 구성되어 있다면, 문장 안의 단어를 모두 검색하는 시간은 아래의 수식으로 계산할 수 있다.

$$M = (\log(n) \times 10) / 2$$

$$T = M \times c$$

$$TS = w \times M \times c = w \times T$$

위의 수식에서 M은 평균비교 회수이며 아래와 같이 계산된다.

$$M = (\log(10의6승) \times 10) / 2 = 60 / 2 = 30$$

$$T = 30 \times 1\text{마이크로초} = 30\text{마이크로초} = 0.03\text{미리초}$$

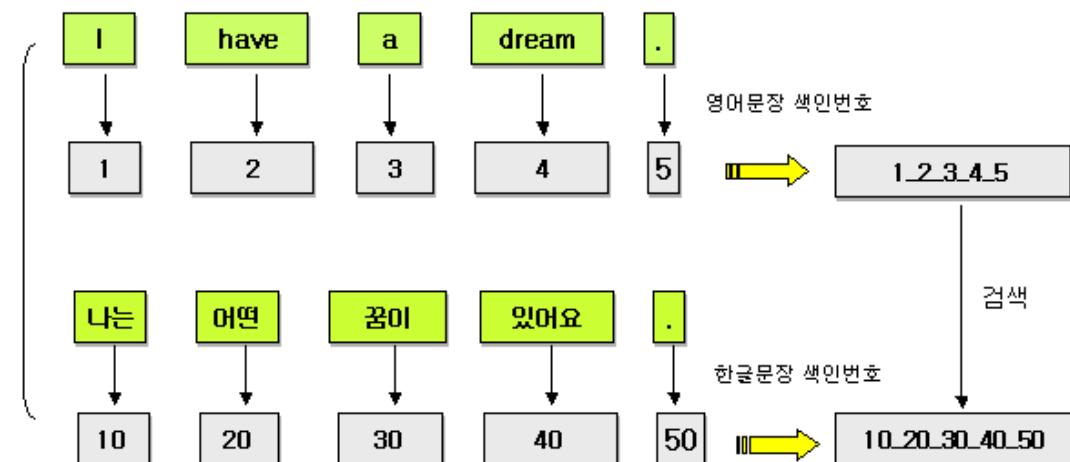
$$TS = 10 \times 0.03\text{미리초} = 0.3\text{미리초}$$

문장안의 단어 개수 w 을 10으로 했을때 TS는 0.3미리초가 된다. 이것은 앞의 순차탐색 평균 검색시간(AS=5초)에 비해서 약16000배가 빨라지는 결과이다. 위와 같이 TS의 검색속도가 빠른 이유는 단어비교 회수가 로그함수로 증가하기 때문이다. 아래 테이블은 순차탐색의 단어비교 회수 $N = n$, TWI 색인탐색의 단어비교 회수 $M = \log(n)$ 를 상호 비교한 것이다.

$n = \text{단어개수}$	$n = 10$	$n = 1000(10\text{의}3\text{승})$	$n = \text{백만}(10\text{의}6\text{승})$	$n = \text{십억}(10\text{의}9\text{승})$
$N = n (\text{비교회수})$	10	1000	1000000	1000000000
$N = n / 2 (\text{평균회수})$	5	500	500000	500000000
$M1 = \log(n)$	1	3	6	9
$M = (\log(n) \times 10) / 2$	5	15	30	45

TWI 번역 방식

TWI 번역 방식은 사용자가 번역한 문장에 색인번호를 부여한후 이것을 TWI 색인사전에 저장하여 TS 검색 시간으로 수행되는 방식이다.

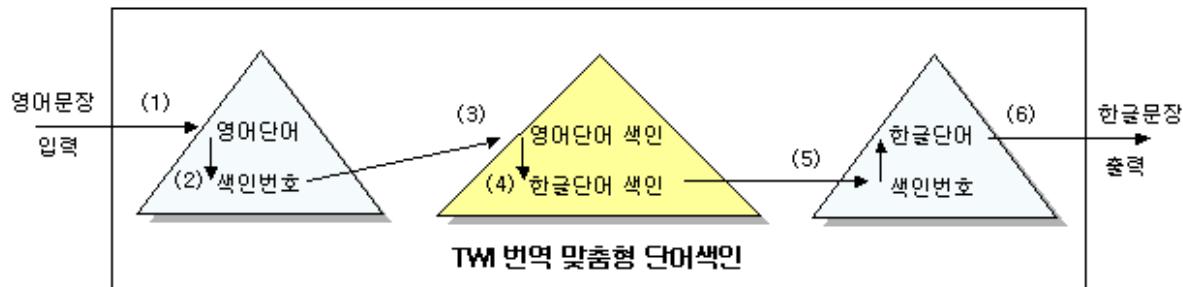


TWI는 아래와 같은 방식으로 문장을 번역한다.

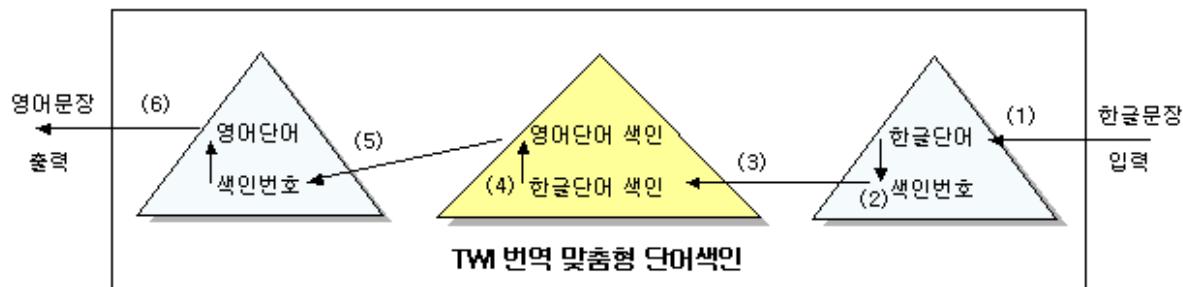
- (1) 문장을 단어 단위로 분리.
- (2) 분리된 각각의 단어를 색인사전에서 검색.
- (3) 검색된 색인번호를 조합하여 번역 맞춤형 단어색인에서 검색.
- (4) 번역대상 색인번호 선택.
- (5) 번역대상 색인번호로 색인사전을 검색.

(6) 검색된 각각의 번역단어를 문장으로 조합.

영한번역: 영어문장 → 한글문장



한영번역: 한글문장 → 영어문장



따라서, TWI를 통한 문장번역 시간 TWS는 아래와 같은 수식으로 계산할 수 있다.

$$M = (\log(n) \times 10) / 2$$

$$T = M \times c$$

$$TS = w \times M \times c = w \times T$$

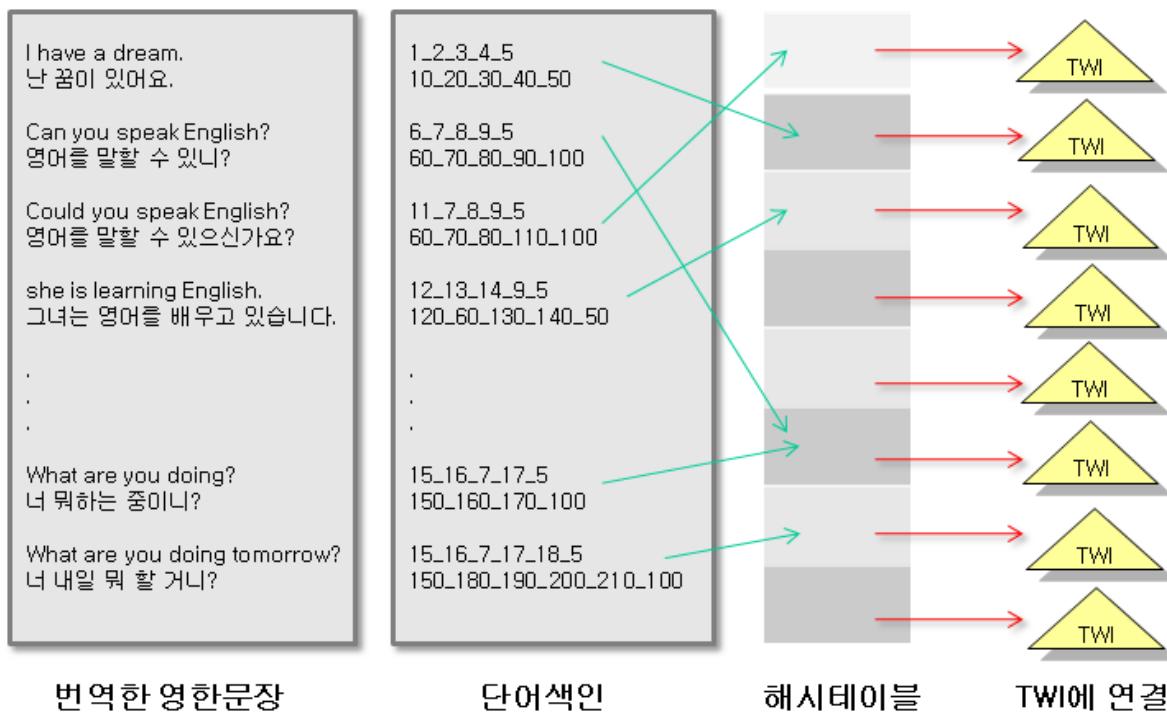
$$TWS = 2TS + T$$

여기서, M은 TWI 색인사전 검색회수, T는 검색시간, TS는 w개의 단어로 구성된 문장 검색시간, TWS는 TWI의 전체 검색시간이다. 아래의 테이블은 단어개수(n)에 따라서 순차검색 시간(AS)과 TWI의 색인사전 검색시간(TWS)을 표로 정리한 것이다.

w = 10	n = 10	n = 1000(10의3승)	n = 백만(10의6승)	n = 십억(10의9승)
AS = w × N × c	50 마이크로초	5 미리초	5 초	5000 초
TWS = 2TS+T	105 마이크로초	345 마이크로초	630 마이크로초	945 마이크로초

TWI 번역 맞춤형 단어색인

TWI 번역에서 TWI 번역 맞춤형 단어색인은 중요한 역할을 한다. 번역한 문장의 단어 색인들을 연결하여 빠르게 검색할 수 있도록 하며 번역문장 인덱스를 컴퓨터 메모리가 허용하는 범위내에서 무한히 할당해도 속도가 별로 떨어지지 않으므로 번역문장 인덱스 개수를 무한히 저장할 수 있다. TWI는 번역문장 인덱스에서 산출한 해시값을 해시 테이블에 연결 시키는 방식을 사용한다.



해시 테이블은 배열형태로 컴퓨터의 기억장소가 허용하는 한 무한히 할당할 수 있으며, 해시 테이블의 배열 요소마다 TWI 단어색인을 연결시킬 수 있다. TWI 단어색인은 32비트 컴퓨터에서 42억개까지 저장할 수 있고, 해시테이블의 배열 요소가 n개라면 영어문장 번역 인덱스는 $n \times 42$ 억개만큼 저장 가능하다.

C언어 소스는 아래의 커널연구회 GitHub 페이지에 공개 되어 있다.

<https://github.com/kernel-bz/linux-kernel-struct/tree/master/btree>

5.2 언어 변환

전통적인 자료구조 알고리즘으로 가능, 인덱싱으로 빠른 처리, 정확성이 높음.

(변환) 문장 → 음성

(변환) 음성 → 문장

5.3 언어 번역

구글 번역:

<https://translate.google.co.kr>

구글 번역 테스트:

구분	영문	옳은 번역	구글 번역
생활 관용 표현	beats me.	몰라.	나를 때린다.
	sort of.	그런 셈이야.	일종의
	you are telling me.	동감이야.	너는 나에게 말하고 있다.
	big deal.	그래서 어쩌라고.	큰 거래.
	it's a deal.	그렇게 해요.	그것은 거래의.
	it's my shout.	내가 쓸게.	내 외침이야.
	i'll keep in touch.	자주 연락할게.	나는 계속 연락할 것이다.
전문 지식	A recurrent neural network (RNN) is a class of artificial neural network where connections between units form a directed graph along a sequence.	반복적 신경망 (RNN)은 단위 간의 연결이 순서적인 절차를 따라서 방향성을 형성하는 인공 신경망의 한 종류입니다.	반복적 신경망 (RNN)은 단위 간의 연결이 시퀀스를 따라 방향성 그래프를 형성하는 인공 신경망의 클래스입니다.
과학 기술	An axon (from Greek ἀξων áxōn, axis) or nerve fiber, is a long, slender projection of a nerve cell,	축색돌기 또는 신경 섬유는 길고 가느다란 모양의 신경 세포로서 신경 세포 몸체로부터	축삭은 (그리스어 ἀξων áxōn 축에서) 또는 신경 섬유는 전형적으로 신경 세포 몸에서 떨어져 활동 전위로 알려진

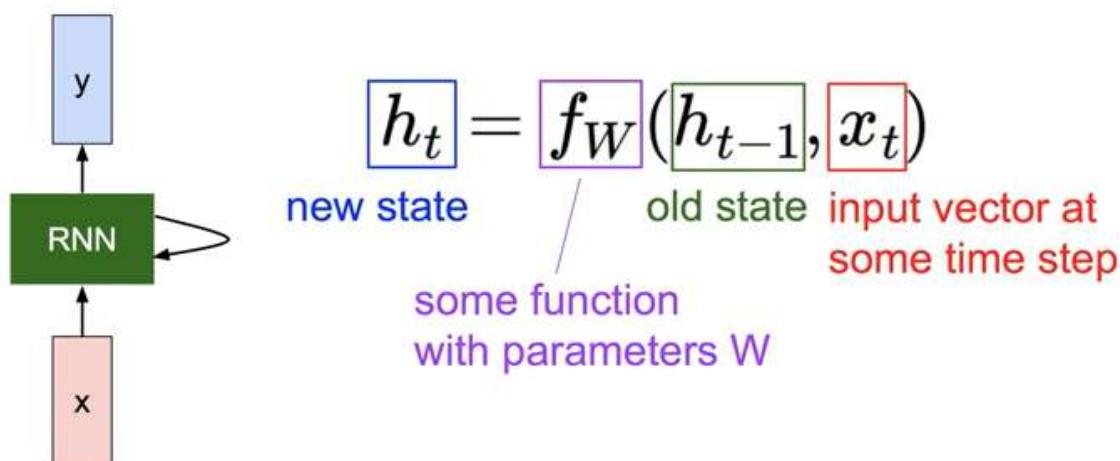
	or neuron, that typically conducts electrical impulses known as action potentials, away from the nerve cell body. The function of the axon is to transmit information to different neurons, muscles, and glands.	떨어져서, 행동 전위로 알려져 있는 전기적 자극을 전달합니다. 축색돌기의 역할은 다른 뉴런, 근육 및 땀샘에 정보를 전달하는 것입니다.	전기적 자극을 수행하는 신경 세포 또는 신경 세포의 길고 가느다란 투사입니다. 축색돌기의 기능은 다른 뉴런, 근육 및 땀샘에 정보를 전송하는 것입니다.
소설 어린 왕자	I have serious reason to believe that the planet from which the little prince came is the asteroid known as B-612.	나에게는 어린왕자가 B-612라고 알려진 소행성이에서 왔다는 것을 믿도록 하는 중요한 이유가 있습니다.	나는 어린 왕자가 태어난 행성이 소행성이 B-612로 알고 있다고 믿을만한 심각한 이유가 있습니다.
소설 메밀 꽃 필 무렵	YounghShin decides to make the classroom as quickly so that children can study freely. She goes to the house of Han-NangChung who has financial resources and begs to pay the donation of 50 won.	영신은 아이들이 자유롭게 공부할 수 있도록 하루빨리 교실을 만들기로 결심한다. 그녀는 재력가인 한낭청의 집에 가서 기부금 50 원을 지원해 달라고 간청한다.	영신은 아이들이 자유롭게 공부할 수 있도록 교실을 빨리 만들기로 결심합니다. 그녀는 재정적 자원이 있는 한남 청의 집에 가서 기부금으로 50 원을 지불하기를 간청한다.

5.4 언어 인식(RNN 알고리즘)

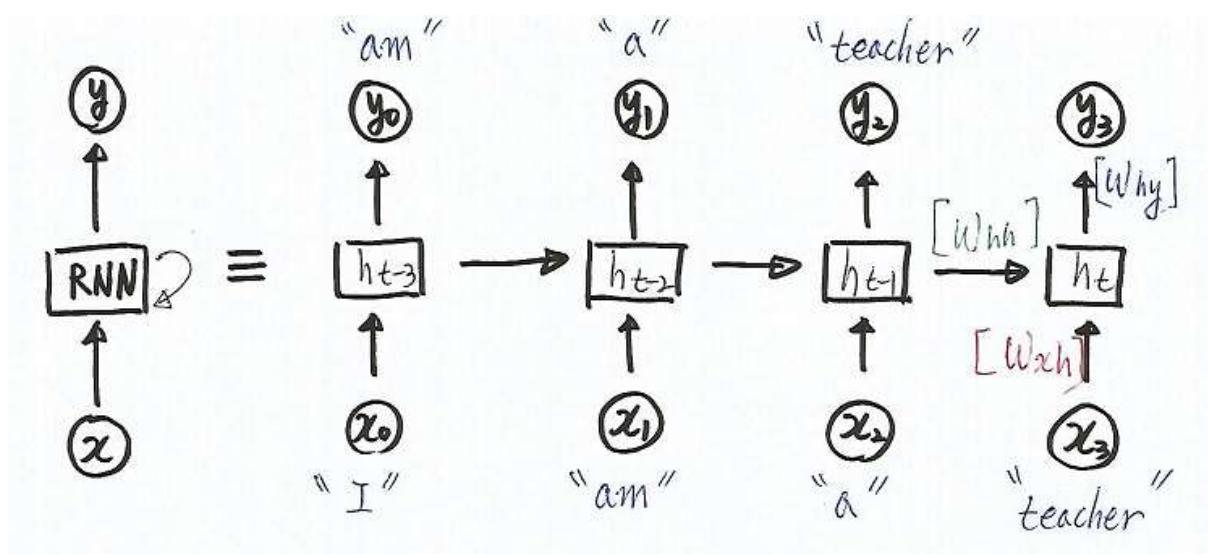
RNN은 어떤 시점에 주어진 입력 데이터를 학습하여 다음 시점에서 출력하는 값들을 예측하는 Neural Network의 한 종류이다. RNN은 데이터가 순서대로 연관관계를 가지며 입력되는 모델에 대해서 학습을 잘하도록 설계되었다. 예를들면, 우리가 일상생활에서 사용하는 언어(문장)는 앞뒤의 문맥(상황)에 맞도록 표현되어야 한다. "I am a teacher"라는 문장은 I 다음에 am 다음에 a 다음에 teacher라는 단어의 흐름을 순서대로 표현하는 것이다. RNN은 이렇게 시간 순서대로 흘러가는 데이터들을 학습하는 모델이다.

5.4.1 RNN 이해

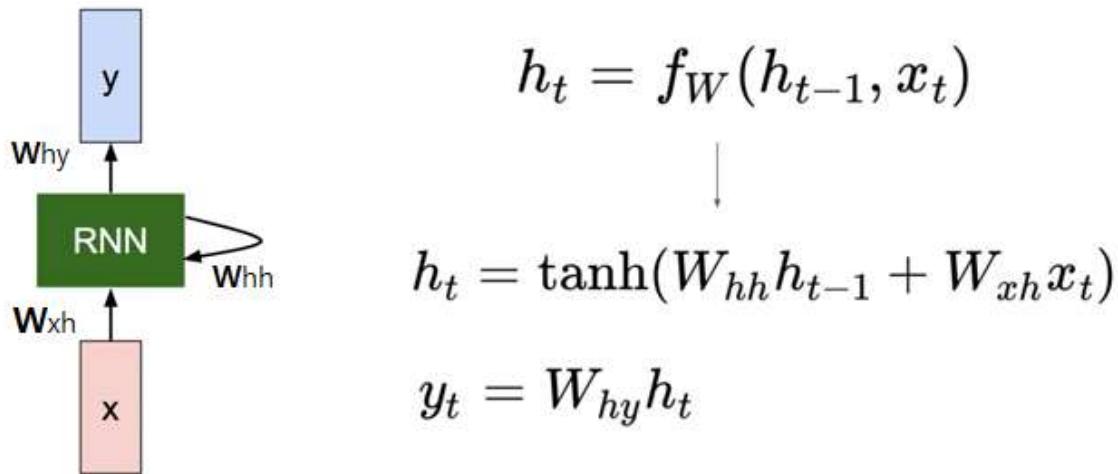
RNN을 수식으로 일반화하여 아래와 같이 표현할 수 있다.



위에서 h_t 는 현재 시점의 상태를 나타내는 데이터이고, h_{t-1} 는 이전 시점의 상태를 나타내는 데이터이다. f_w 는 우리의 머신이 학습하는 데이터인 weight이다. 입력 데이터인 x_t 가 주어지면 이것에 대해서 이전 상태(h_{t-1})값을 곱하여 현재 상태(h_t)값을 결정한다. 이렇게 현재와 이전의 상태값을 반복적(Recurrent)으로 계산하면서 f_w 를 학습한다. 아래 그림은 이러한 과정을 좀더 자세히 설명해 주는 것이다.

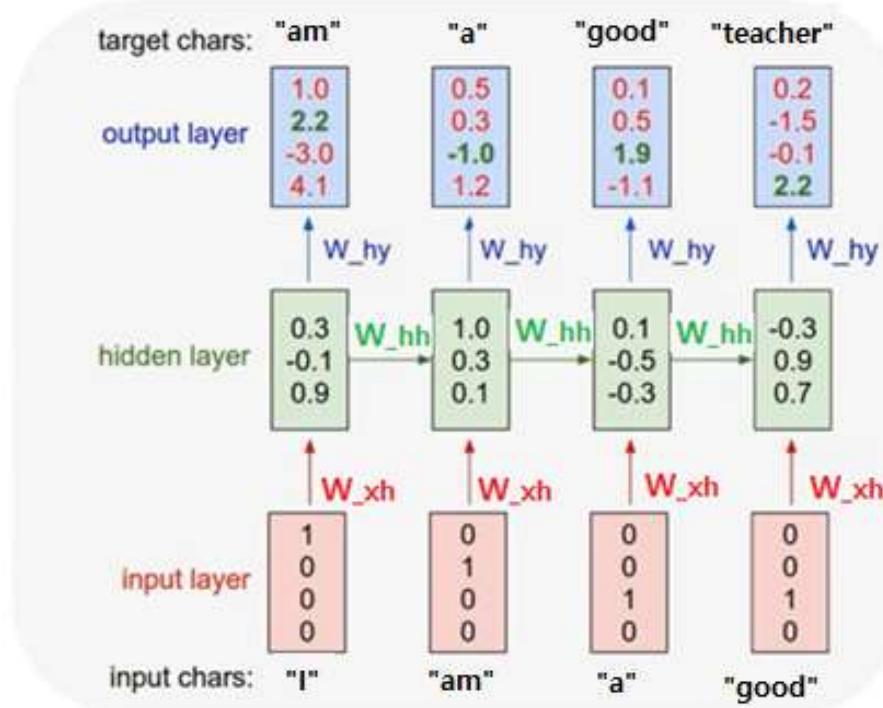


아래는 RNN에서 Weight을 어떻게 학습하는지, 그 과정을 수학수식으로 좀더 자세히 설명하는 것이다.



아래 그림은 "I am a good teacher"라는 문장에 대해서 각각의 단어들이 입력되는 순서, 즉 이전 상태와 현상태와의 관계를 RNN으로 학습하는 과정을 개념적으로 설명하는 것이다.

"I am a good teacher" 문장 학습(개념적 설명)



그림출처: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

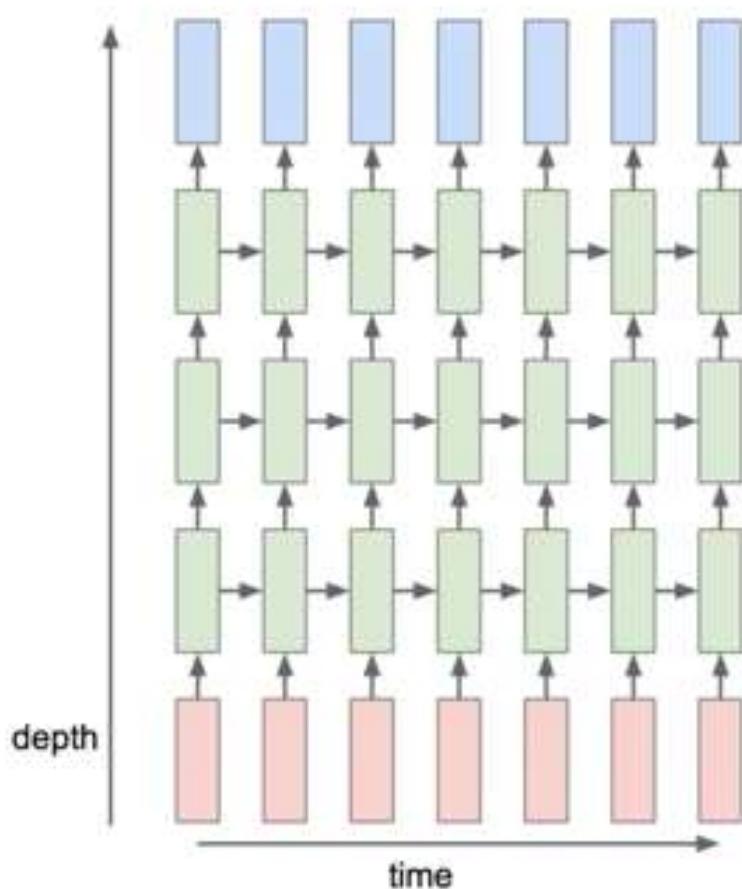
연산과정을 행렬(Matrix)로 표현하면 다음과 같다.

$$\begin{bmatrix} x_t \\ x_0 \ x_1 \ x_2 \ x_3 \end{bmatrix} \begin{bmatrix} w_{xh} \\ w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix} = \begin{bmatrix} x_{h0} \ x_{h1} \ x_{h2} \end{bmatrix} = xh$$

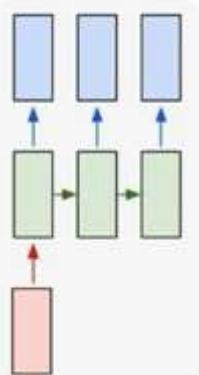
$$\begin{bmatrix} h_{t-1} \\ h_0 \ h_1 \ h_2 \end{bmatrix} \begin{bmatrix} w_{hh} \\ w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} = \begin{bmatrix} h_{h0} \ h_{h1} \ h_{h2} \end{bmatrix} = hh$$

$$h_t = \tanh(xh \odot hh) = [h_0 \ h_1 \ h_2]$$

$$\begin{bmatrix} h_t \\ h_0 \ h_1 \ h_2 \end{bmatrix} \begin{bmatrix} w_{hy} \\ w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix} = \begin{bmatrix} y_0 \ y_1 \ y_2 \ y_3 \end{bmatrix}$$

Multi-Layer Deep RNN

5.4.2 RNN 활용예

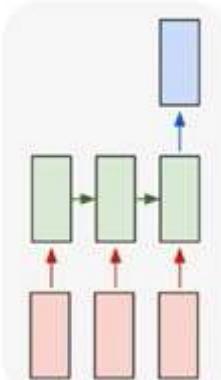
one to many

입력된 그림을 학습하여 설명문 출력하기.

그림 → 이 그림에서는 여러 사람이 운동하고 있습니다.

사진 → 이 사진은 많은 새들이 하늘을 날고 있는 모습입니다.

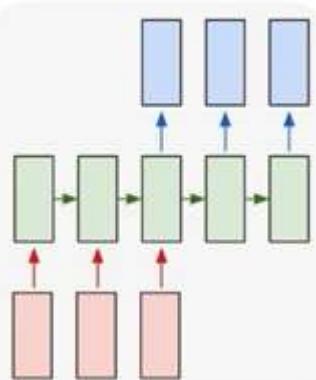
그림 → 이것은 들판에서 농부들이 곡식을 수확하는 모습을 그린 그림입니다.

many to one

입력된 문장을 학습하여 감정 표현하기.

나는 어제 친구들과 한강에서 자전거를 탔다. → 즐거워요.

오늘 저녁 뉴스에서 여객기 납치사건을 들었다. → 슬퍼요.

many to many

문장 번역하기.

I have a dream. → 나는 꿈이 있어요.

I'd like to drink a cup of coffee. → 나는 커피 한잔하고 싶어요.

나는 훌륭한 과학자가 되고 싶어요. → I want to be a great scientist.

5.4.3 LSTM

Long Short-Term Memory (LSTM)는 RNN 의 계층들을 형성하기 위한 단위들(blocks).

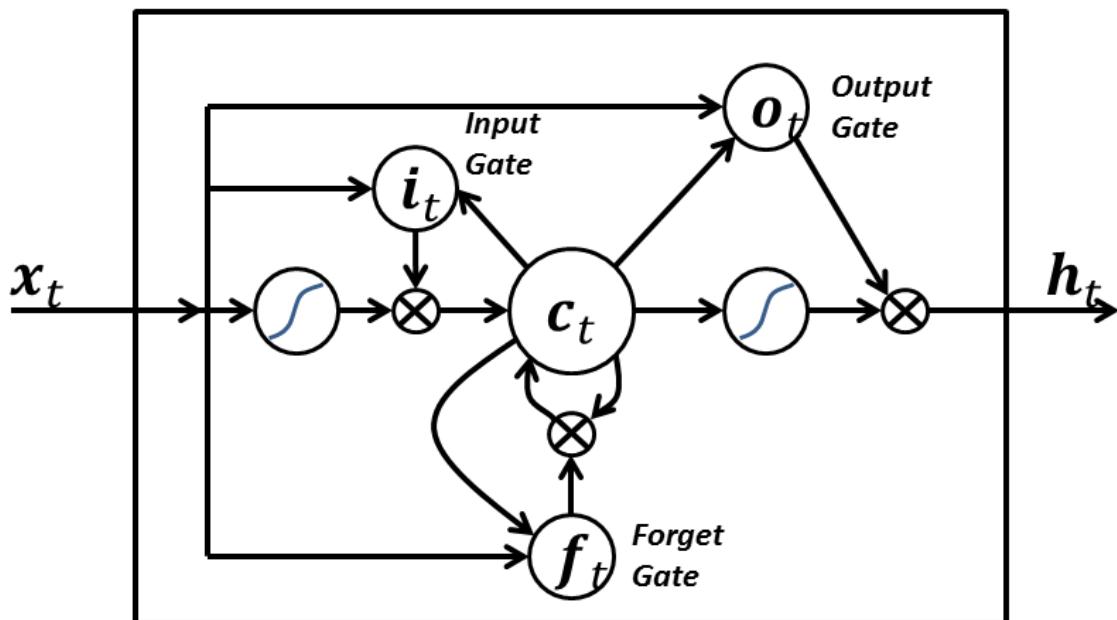
LSTM 단위들로 구성된 RNN 을 LSTM 네트워크라 한다.

LSTM 단위들은 Cell, Input Gate, Output Gate, Forget Gate 로 구성된다.

Cell(C_t)은 어떤 주어진 시간동안 값을 기억(Memory)하는 역할을 한다.

Input/Output/Forget Gate 들은 특정시간(t)에서 가중치 합의 행동(activation)을 계산하기 위한 gate 역할을 한다.

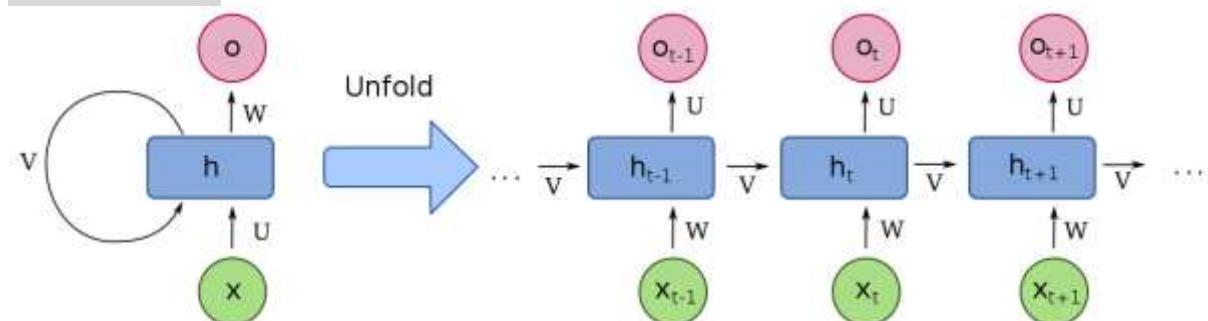
LSTM 단위들(blocks)



그림출처: https://en.wikipedia.org/wiki/Long_short-term_memory

https://commons.wikimedia.org/wiki/File:Long_Short_Term_Memory.png

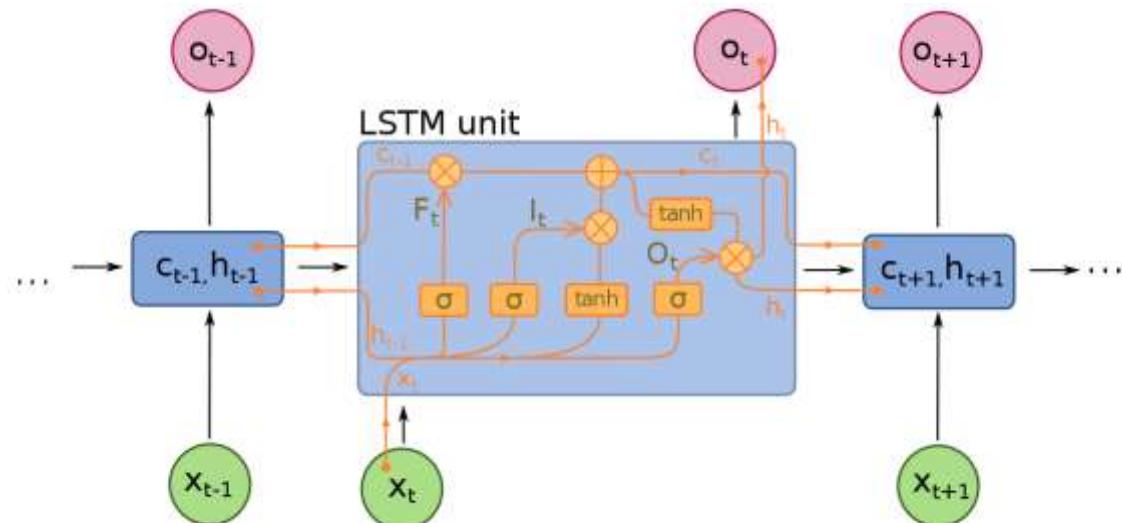
기본적인 RNN



그림출처: https://en.wikipedia.org/wiki/Recurrent_neural_network

https://commons.wikimedia.org/wiki/File:Recurrent_neural_network_unfold.svg

LSTM 단위들 상세 블록도



그림출처: https://en.wikipedia.org/wiki/Recurrent_neural_network

https://en.wikipedia.org/wiki/Recurrent_neural_network#/media/File:Long_Short-Term_Memory.svg

변수들

- $x_t \in R^d$: input vector to the LSTM unit
- $f_t \in R^h$: forget gate's activation vector
- $i_t \in R^h$: input gate's activation vector
- $o_t \in R^h$: output gate's activation vector
- $h_t \in R^h$: output vector of the LSTM unit
- $c_t \in R^h$: cell state vector
- $W \in R^{h \times d}$, $U \in R^{h \times h}$ and $b \in R^h$: weight matrices and bias vector parameters

행동 함수들(Acivation Functions)

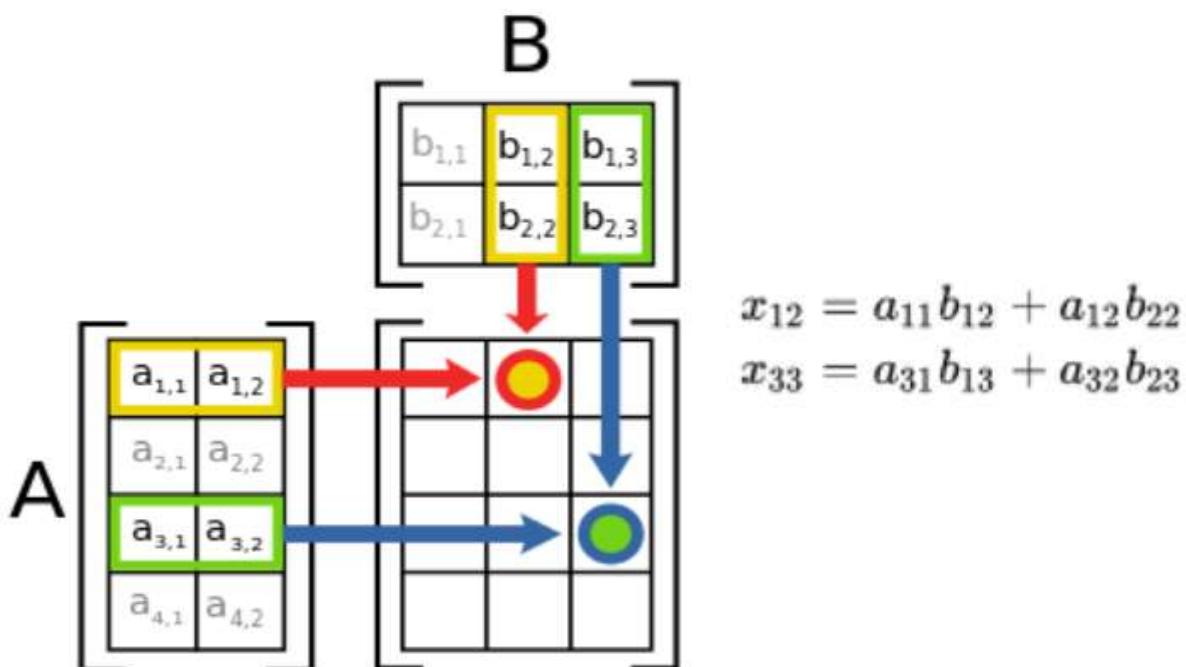
- σ_g : sigmoid function.
- σ_c : hyperbolic tangent function.
- σ_h : hyperbolic tangent function or, as the peephole LSTM

forward pass 연산식들

$$\begin{aligned}
 f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
 i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
 o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
 c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
 h_t &= o_t \circ \sigma_h(c_t)
 \end{aligned}$$

Matrix product (AB)

$$\begin{matrix} 4 \times 2 \text{ matrix} \\ \left[\begin{array}{cc} a_{11} & a_{12} \\ \cdot & \cdot \\ a_{31} & a_{32} \\ \cdot & \cdot \end{array} \right] \end{matrix} \begin{matrix} 2 \times 3 \text{ matrix} \\ \left[\begin{array}{ccc} \cdot & b_{12} & b_{13} \\ \cdot & b_{22} & b_{23} \end{array} \right] \end{matrix} = \begin{matrix} 4 \times 3 \text{ matrix} \\ \left[\begin{array}{ccc} \cdot & x_{12} & x_{13} \\ \cdot & \cdot & \cdot \\ \cdot & x_{32} & x_{33} \\ \cdot & \cdot & \cdot \end{array} \right] \end{matrix}$$



그림출처: https://en.wikipedia.org/wiki/Matrix_multiplication#Matrix_product_.28two_matrices.29

Hadamard product

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \circ \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} & a_{12}b_{12} & a_{13}b_{13} \\ a_{21}b_{21} & a_{22}b_{22} & a_{23}b_{23} \\ a_{31}b_{31} & a_{32}b_{32} & a_{33}b_{33} \end{pmatrix}$$

Peephole LSTM 연산식들

$$\begin{aligned} f_t &= \sigma_g(W_f x_t + U_f c_{t-1} + b_f) \\ i_t &= \sigma_g(W_i x_t + U_i c_{t-1} + b_i) \\ o_t &= \sigma_g(W_o x_t + U_o c_{t-1} + b_o) \\ c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c c_{t-1} + b_c) \\ h_t &= o_t \circ \sigma_h(c_t) \end{aligned}$$

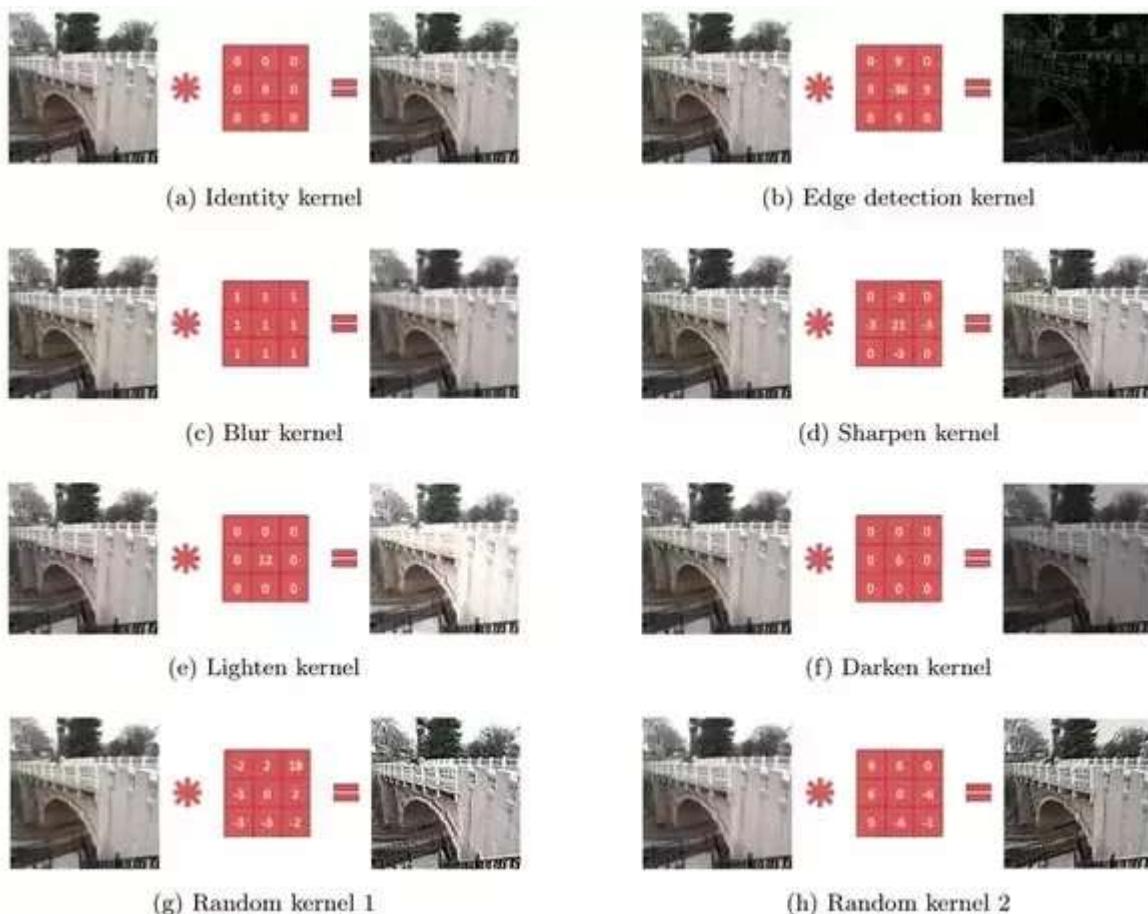
Convolutional LSTM

$$\begin{aligned} f_t &= \sigma_g(W_f * x_t + U_f * h_{t-1} + V_f \circ c_{t-1} + b_f) \\ i_t &= \sigma_g(W_i * x_t + U_i * h_{t-1} + V_i \circ c_{t-1} + b_i) \\ o_t &= \sigma_g(W_o * x_t + U_o * h_{t-1} + V_o \circ c_{t-1} + b_o) \\ c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c * x_t + U_c * h_{t-1} + b_c) \\ h_t &= o_t \circ \sigma_h(c_t) \end{aligned}$$

Two dimension convolution operation

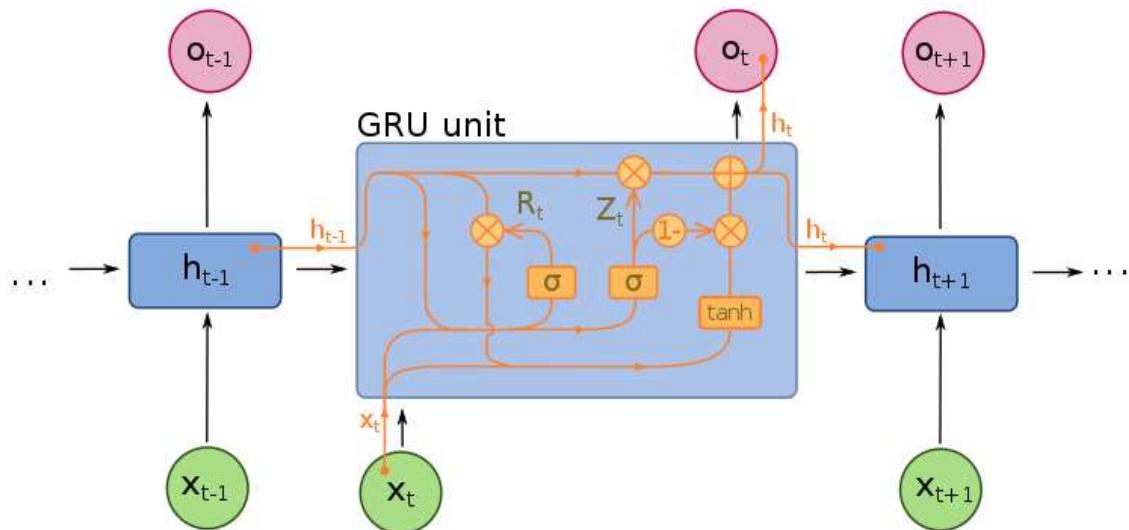
The diagram shows a 5x5 input matrix on the left, a 3x3 kernel matrix in the center, and a 5x3 output matrix on the right. The input matrix has values: 22, 15, 1, 3, 60; 42, 5, 38, 39, 7; 28, 9, 4, 66, 79; 0, 2, 25, 12, 17; 9, 14, 2, 51, 3. The kernel matrix has values: 0, 0, 1; 0, 0, 0; 1, 0, 0. The output matrix has values: 29, 12, 64; 38, 41, 32; 13, 80, 81.

그림출처: <https://www.quora.com/How-can-I-explain-the-dimensionality-reduction-in-convolutional-neural-network-CNN-from-this-image>



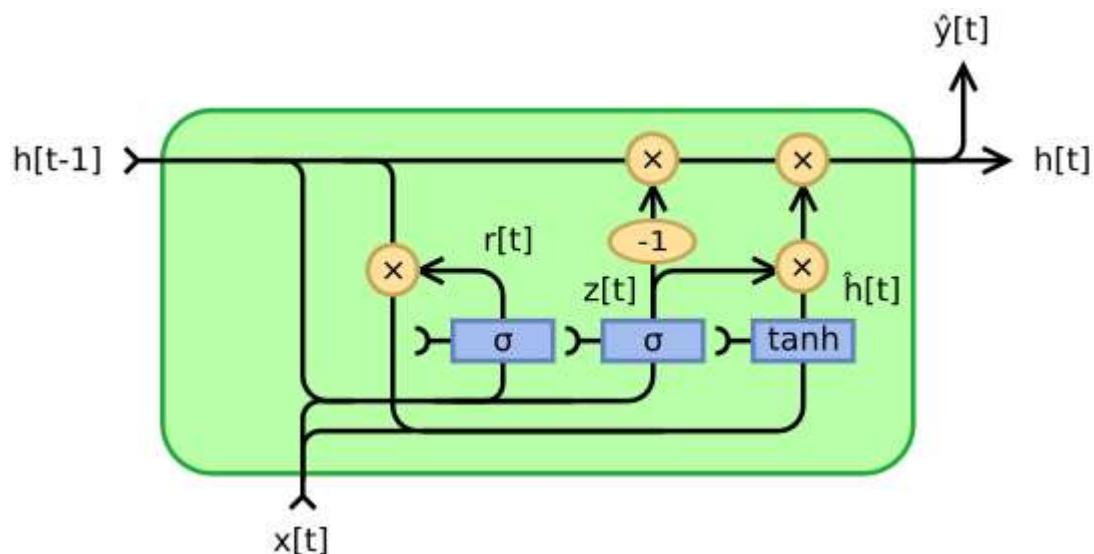
5.4.4 GRU

Gated Recurrent Unit (GRU)은 2014년에 코넬대 조경현 박사에 의해 소개된 RNN 게이팅 메커니즘입니다. 완전한 형태와 몇가지 단순화 된 변형으로 사용됩니다. 다성 음악 모델링 및 음성 신호 모델링에서의 LSTM과 유사하다는 것이 밝혀졌습니다. 이들은 출력 게이트가 없기 때문에 LSTM 보다 매개 변수가 적습니다.



그림출처: https://en.wikipedia.org/wiki/Recurrent_neural_network

https://en.wikipedia.org/wiki/Recurrent_neural_network#/media/File:Gated_Recurrent_Unit.svg



그림출처: https://en.wikipedia.org/wiki/Gated_recurrent_unit

변수들

- x_t : input vector
- h_t : output vector
- z_t : update gate vector
- r_t : reset gate vector
- W , U and b : parameter matrices and vector

행동 함수들(Aactivation Functions)

- σ_g : The original is a sigmoid function.
- σ_h : The original is a hyperbolic tangent.

forward pass 연산식들

$$\begin{aligned} z_t &= \sigma_g(W_z x_t + U_z h_{t-1} + b_z) \\ r_t &= \sigma_g(W_r x_t + U_r h_{t-1} + b_r) \\ h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \sigma_h(W_h x_t + U_h (r_t \circ h_{t-1}) + b_h) \end{aligned}$$