ML_6

Team BMS

INDEX

ML Study
6 Week

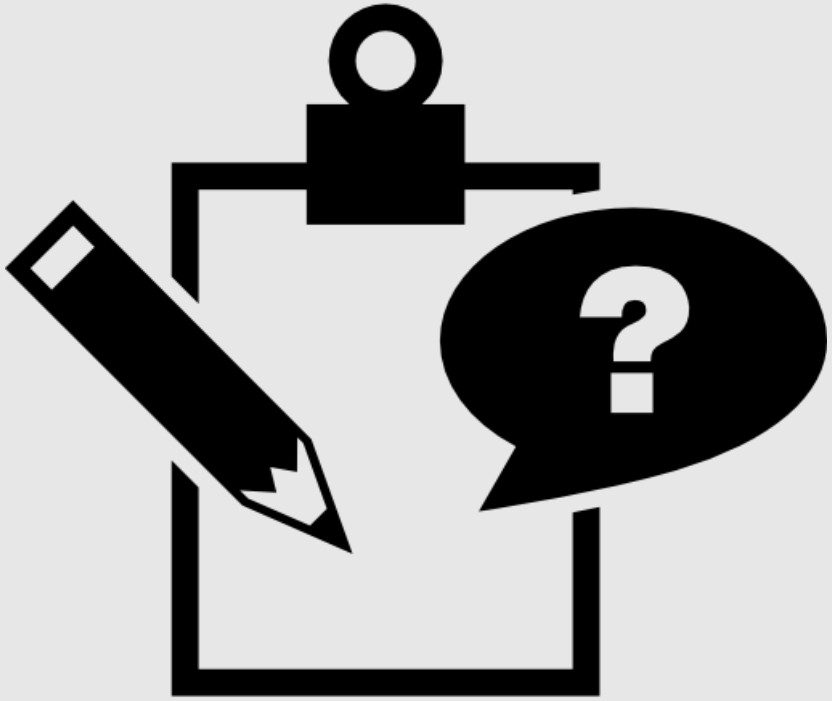
Ice Breaking

Ice Breaking

〈http://blog.naver.com/PostView.nhn?blogId=itkmj&logNo=220617363686&parentCategoryNo=&categoryNo=14&viewDate=&isShowPopularPosts=true&from=search〉

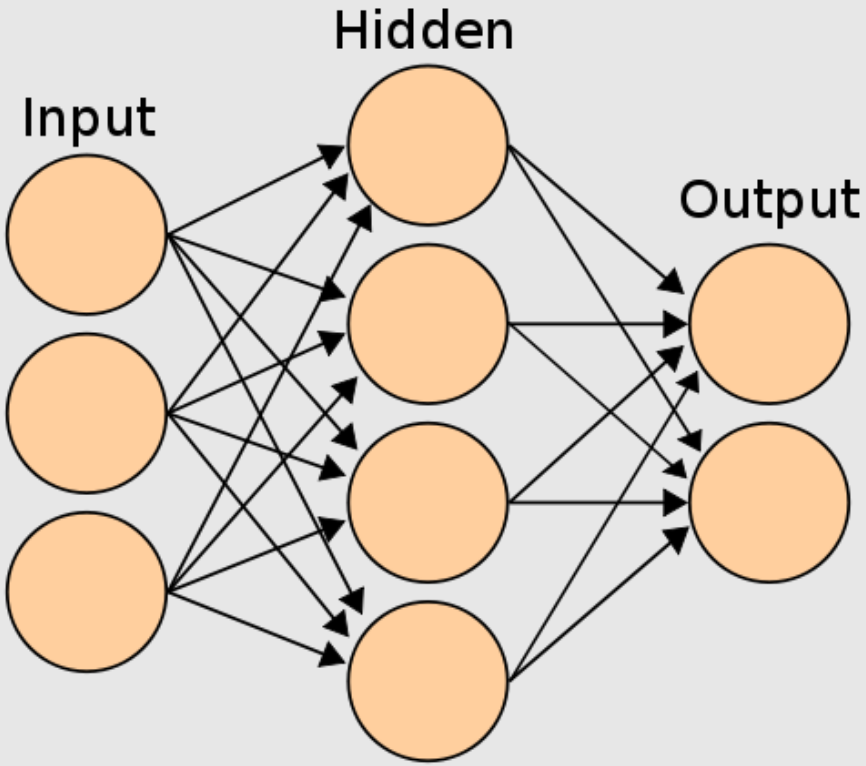
Ice Breaking

〈http://thegear.co.kr/16415?fbclid=IwAR1K-pGwlUPYfcRUK2kaa8hWX7iNuZbSPs3Cxir3RWMQzpH95ITpWclB3S4〉



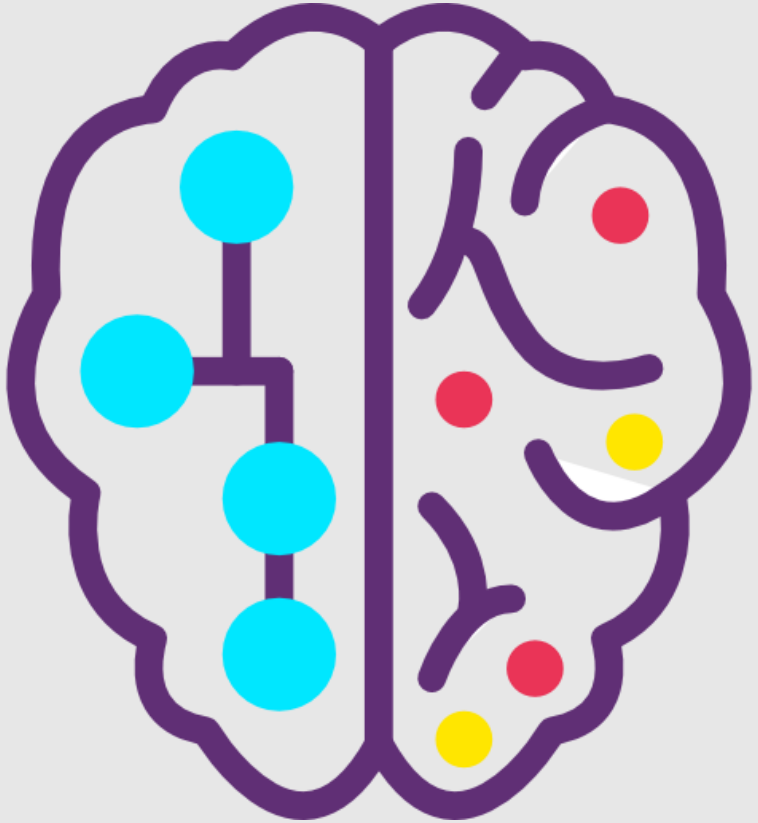
Introduction to Artificial Neural Networks

Artificial Neural Networks?



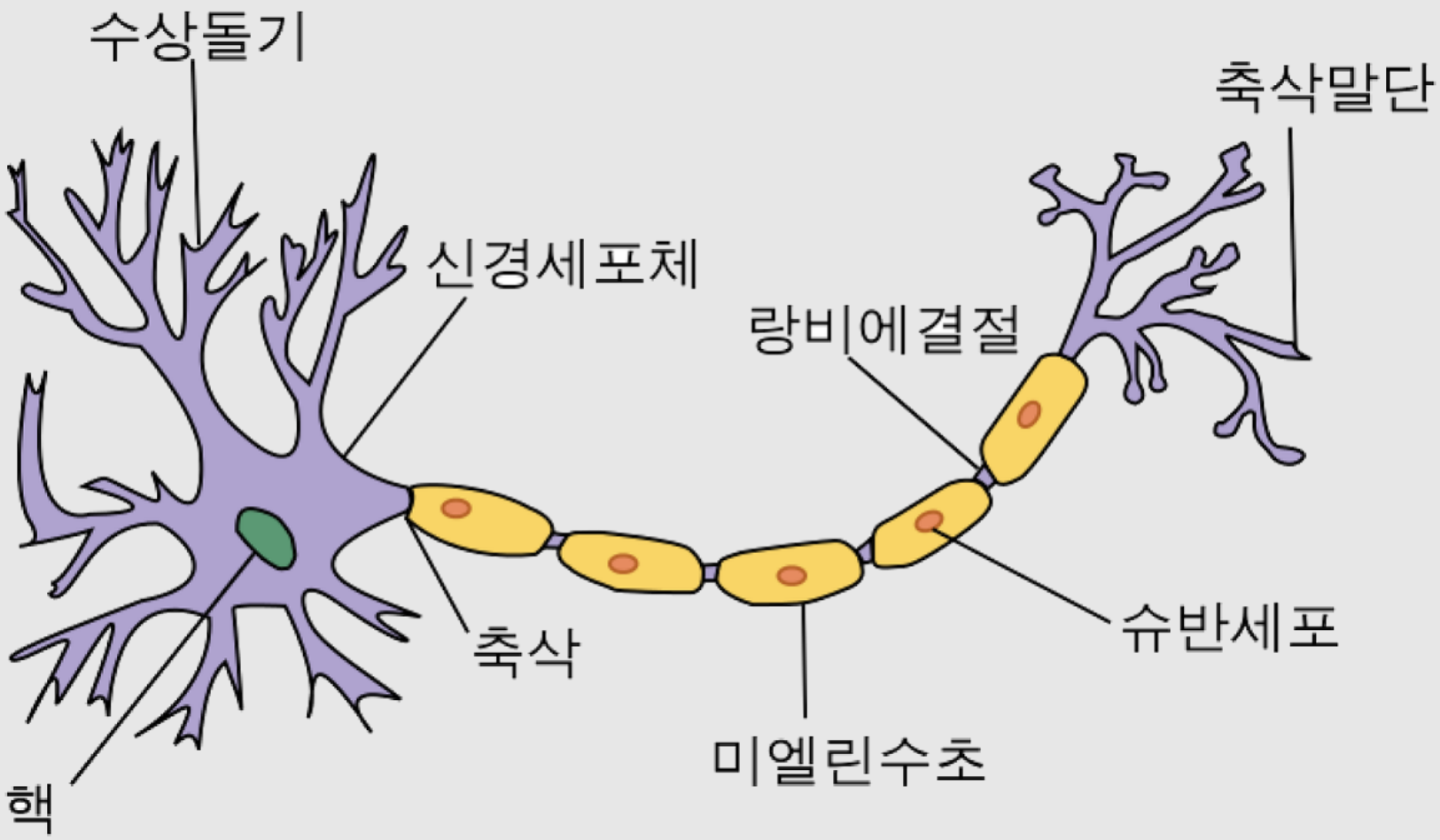
Artificial Neural Networks (ANN)?

Artificial Neural Networks?



AI에 대한 열망 ➔ 뇌 구조를 파악하자

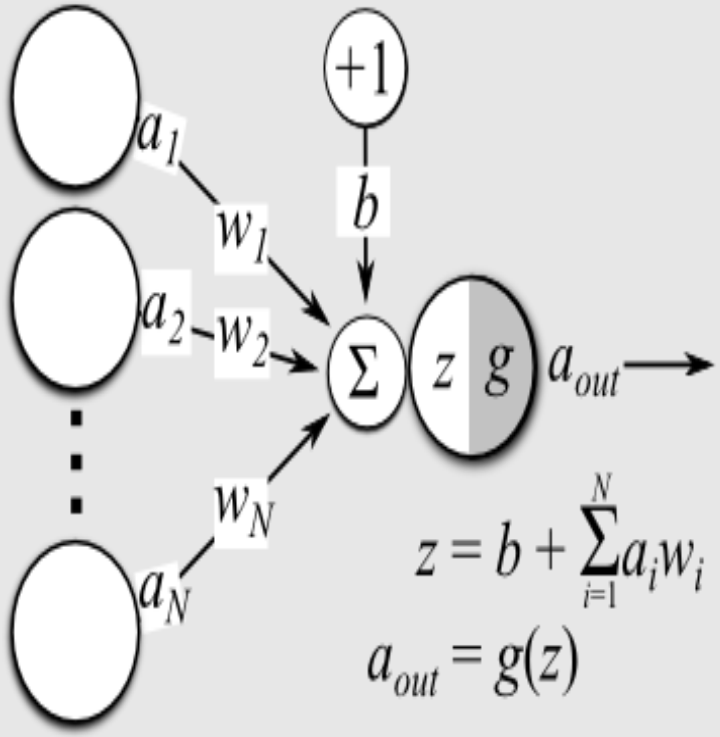
Neural?



〈https://brunch.co.kr/@cogsciin/16〉

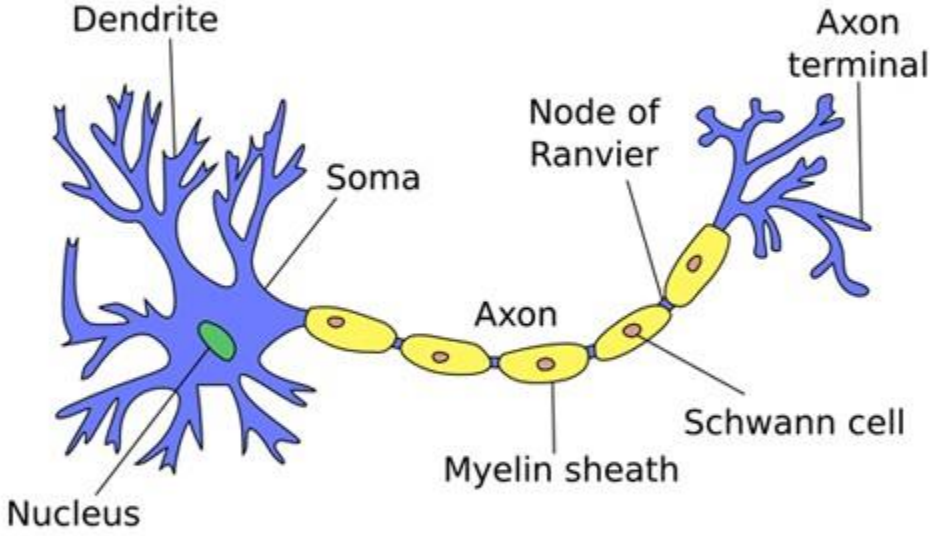
Artificial Neural Networks?

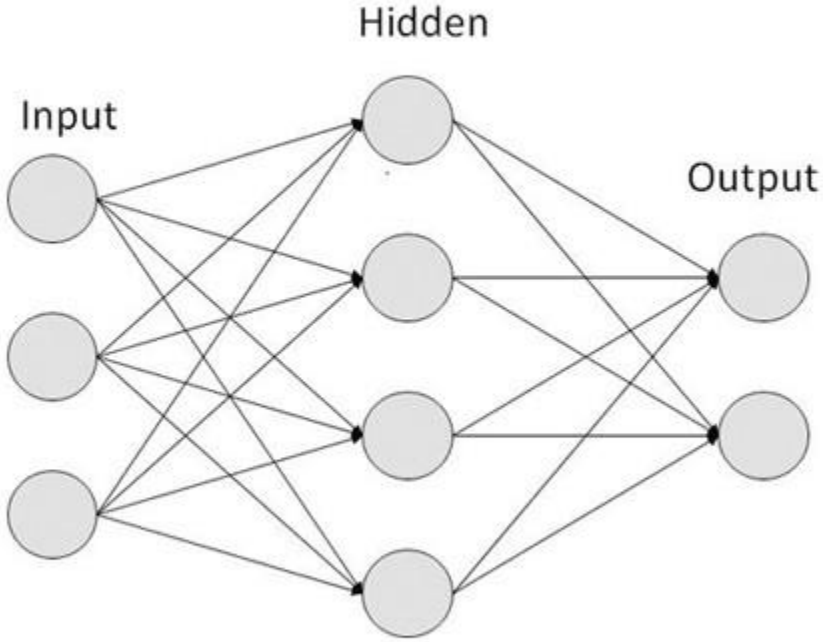
Neuroscience



Deeplearning

Neural & Neural Net

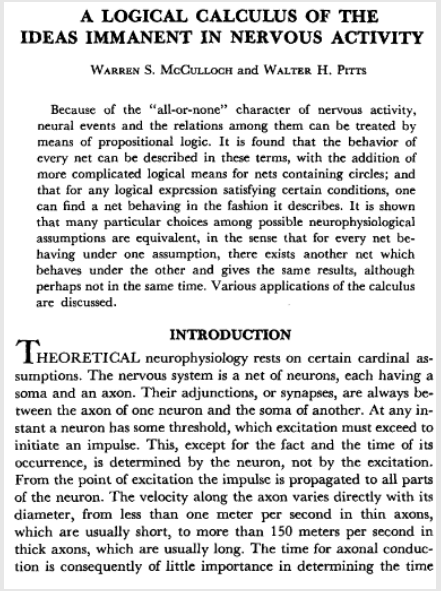




https://medium.com/@ismailghallou/build-your-perceptron-neural-net-from-scratch-e12b7be9d1ef

History

1943



A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY

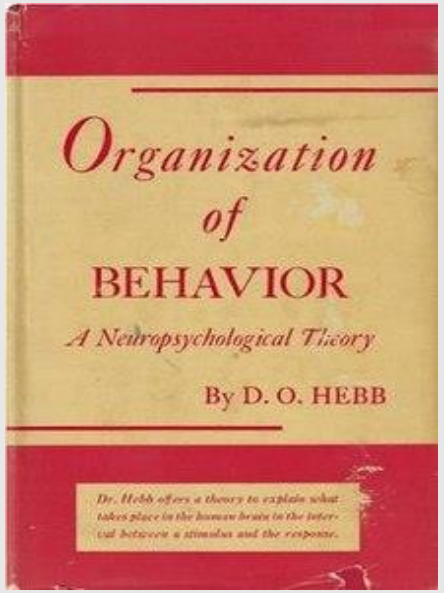
WARREN S. MCCULLOCH and WALTER H. PITTS

Because of the "all-or-none" character of nervous activity, neural events and the relations among them can be treated by means of propositional logic. It is found that the behavior of every net can be described in these terms, with the addition of more complicated logical means for nets containing circles; and that for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes. It is shown that many particular choices among possible neurophysiological assumptions are equivalent, in the sense that for every net behaving under one assumption, there exists another net which behaves under the other and gives the same results, although perhaps not in the same time. Various applications of the calculus are discussed.

INTRODUCTION

THEORETICAL neurophysiology rests on certain cardinal assumptions. The nervous system is a net of neurons, each having a soma and an axon. Their adjunctions, or synapses, are always between the axon of one neuron and the soma of another. At any instant a neuron has some threshold, which excitation must exceed to initiate an impulse. This, except for the fact and the time of its occurrence, is determined by the neuron, not by the excitation. From the point of excitation the impulse is propagated to all parts of the neuron. The velocity along the axon varies directly with its diameter, from less than one meter per second in thin axons, which are usually short, to more than 150 meters per second in thick axons, which are usually long. The time for axonal conduction is consequently of little importance in determining the time

1949



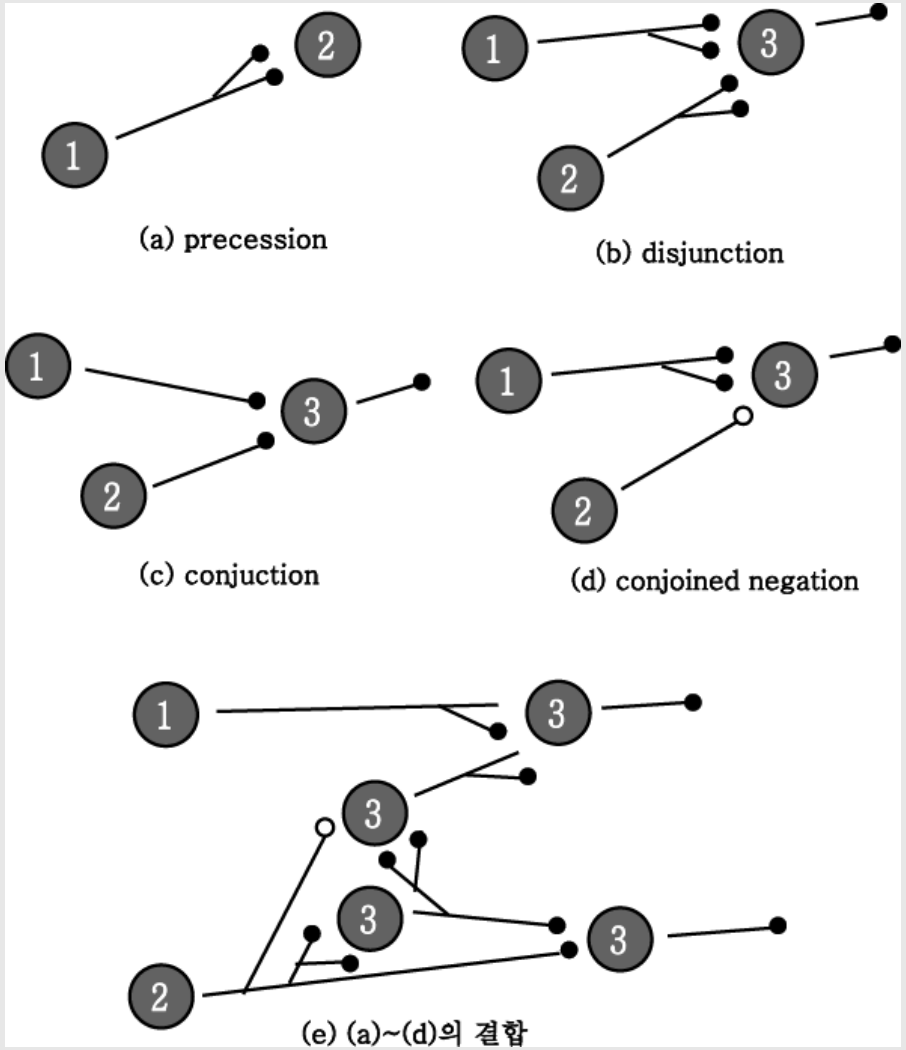
1957

1943: Pitts & McCulloh

신경 활동의 'All-or-Nothing'적인 특성 때문에 신경계의 일과 그들 사이의 관계들은 명제논리(Propositional logic)로 취급된다. 모든 망의 행동은 이러한 관점에서 기술될 수 있다. 어떠한 조건들을 만족시키려는 논리적 표현에 대하여, 우리는 그것이 기술하는 방법대로 행동하는 망을 찾을 수 있다.

- 뉴런의 활동은 All 아니면 Nothing이다.
- 어떤 뉴런을 흥분되게 하려면 2개 이상의 고정된 수의 시냅스가 일정한 시간 내에 활성화되어야 한다.
- 신경 시스템에서 유일하게 의미있는 시간지연(Delay)는 시냅스에서 지연(Synaptic Delay)이다.
- 어떠한 억제적인 시냅스는 그 시각의 뉴런의 활성화를 절대적으로 방지한다.
- 네트워크의 연결 구조는 시간에 따라 바뀌지 않는다.

1943: Pitts & McCulloh

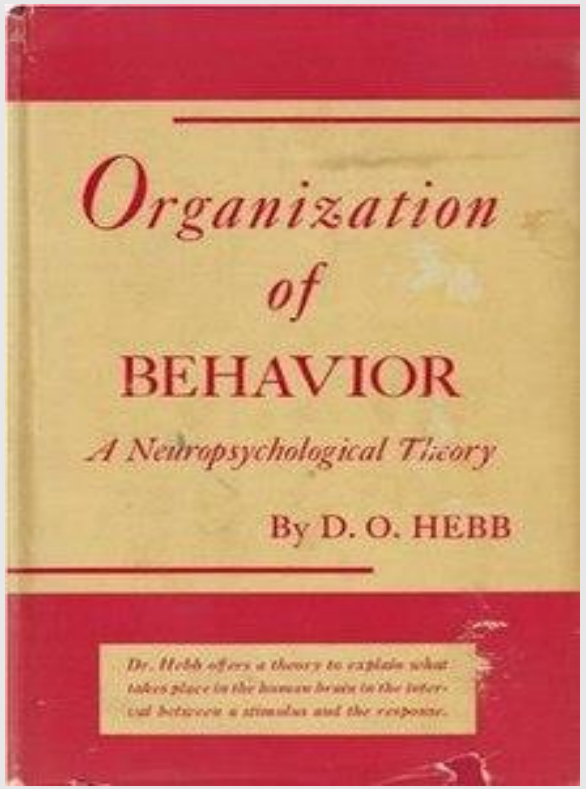


〈http://www.aistudy.com/neural/model_kim.htm〉

1949: Hebb's Rule

Let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability. When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B is increased.

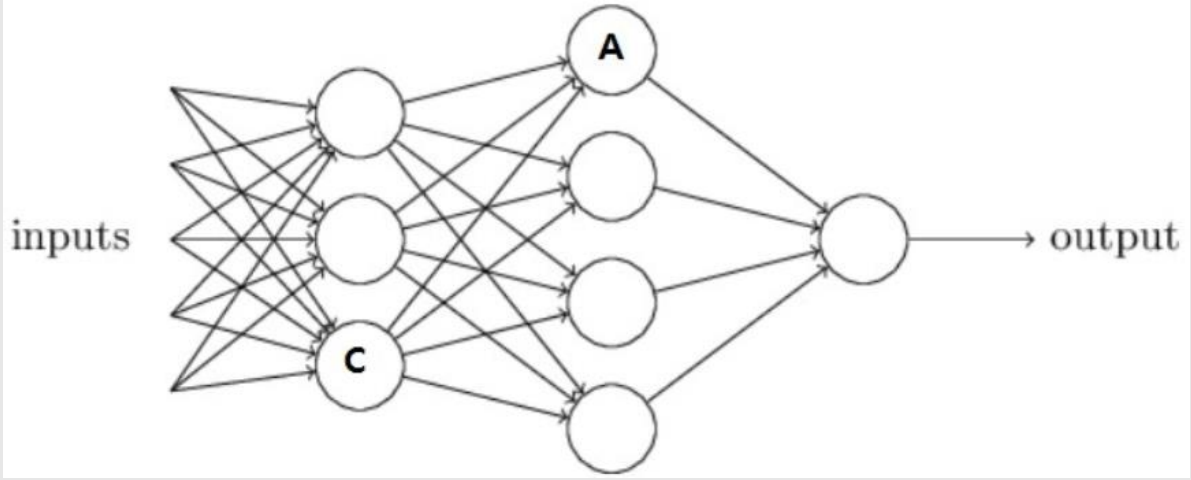
(Reverberatory(또는 "Trace")의 지속성이나 반복이 안정성을 증가시키는 지속적인 세포 변화를 유도한다고 가정해보자. 세포 A의 축색이 세포 B를 자극할 정도로 충분히 가까이 있고 반복적으로 혹은 지속적으로 그것을 발화하는 것에 참여하면, 어떤 성장 과정이나 신진대사 변화는 세포 B의 효율성이 증가된 하나의 세포에서 일어난다.)



➔ **두 개의 뉴런 A,B가 서로 반복적이고 지속적으로 점화하여 어느 한 쪽 또는 양 쪽 모두에 어떤 변화를 야기한다면 상호간의 점화의 효율은 점점 커지게 된다.**

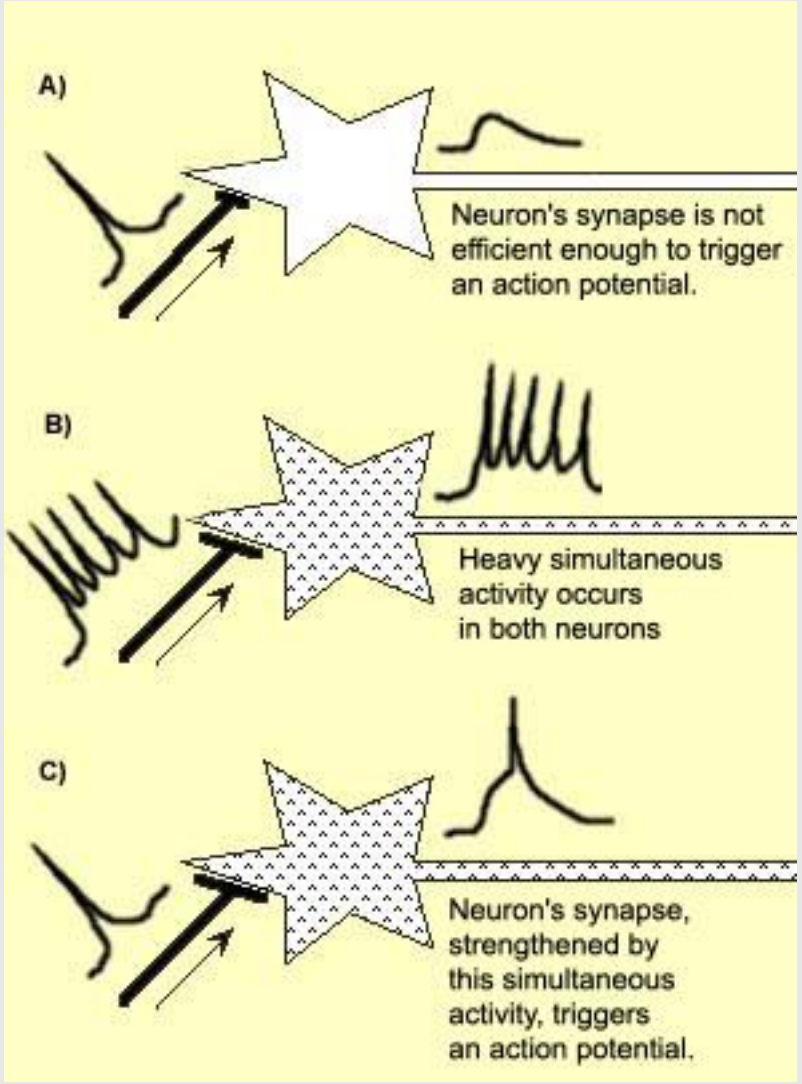
1949: Hebb's Rule

C가 출력을 냈을 때, A도 출력을 냈다면 C와 A는 인과관계
➔ 이 경우에만 연결 강도를 강화하고 그렇지 않다면 연결 강도를 약화



〈https://m.blog.naver.com/PostView.nhn?blogId=msnayana&logNo=220652975363&proxyReferer=https%3A%2F%2Fwww.google.com%2F〉

1949: Hebb's Rule



〈http://thebrain.mcgill.ca/flash/i/i_07/i_07_cl/i_07_cl_tra/i_07_cl_tra.html〉

1957: Perceptron (Frank Rosenblatt)

Perceptron (1957)

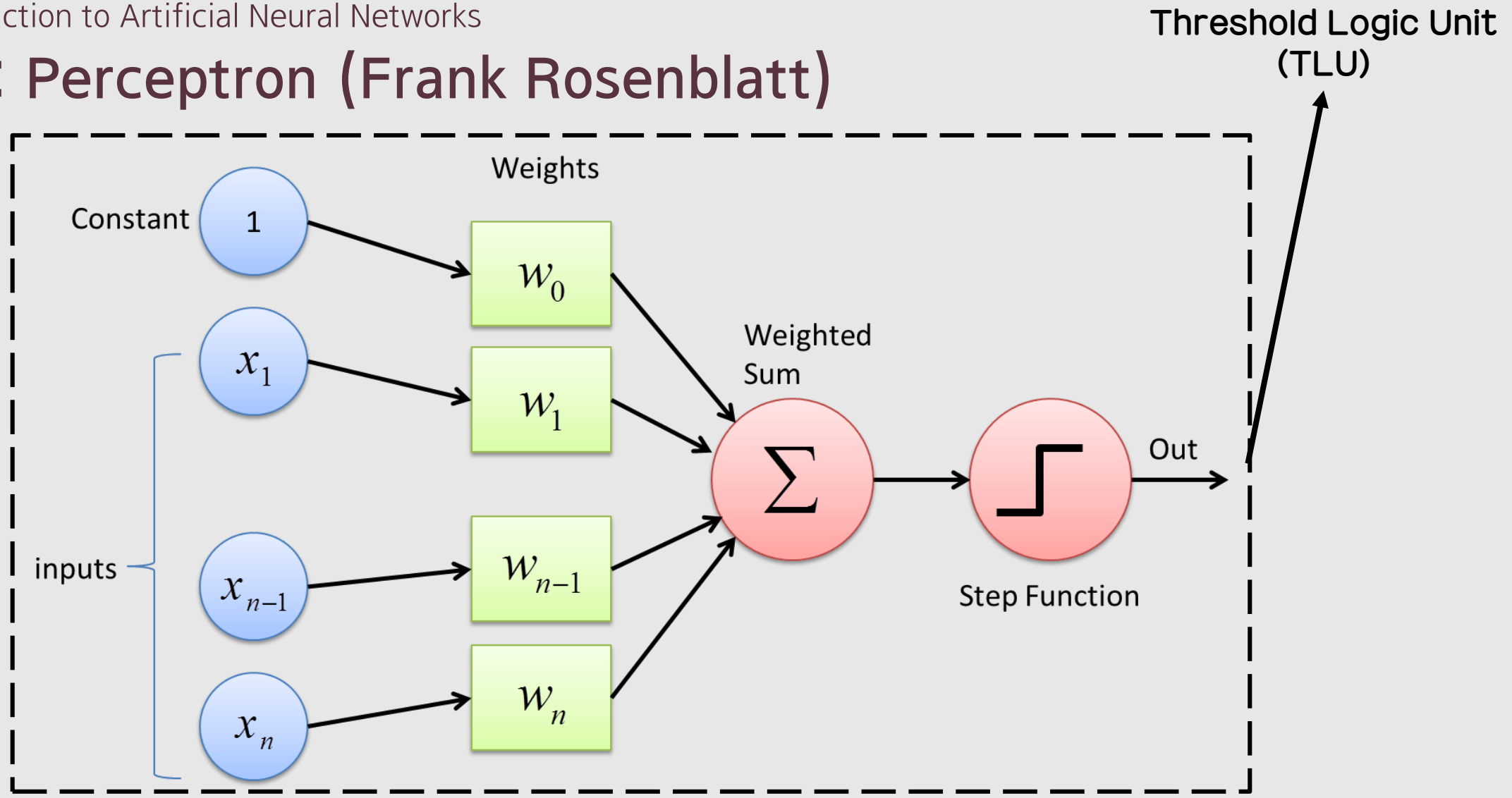
Frank Rosenblatt

Developed the learning algorithm.

Used his neuron (pattern recognizer = perceptron) for classification of letters.

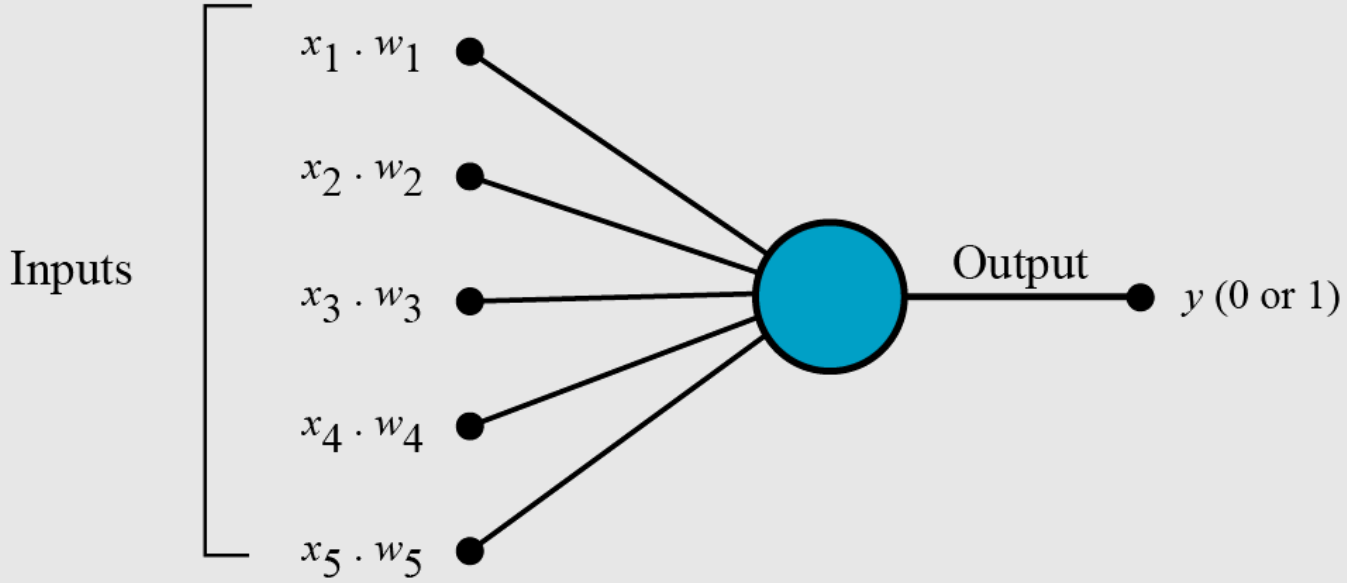
〈https://slideplayer.com/slide/11742740/〉

1957: Perceptron (Frank Rosenblatt)



〈https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53〉

1957: Perceptron (Frank Rosenblatt)

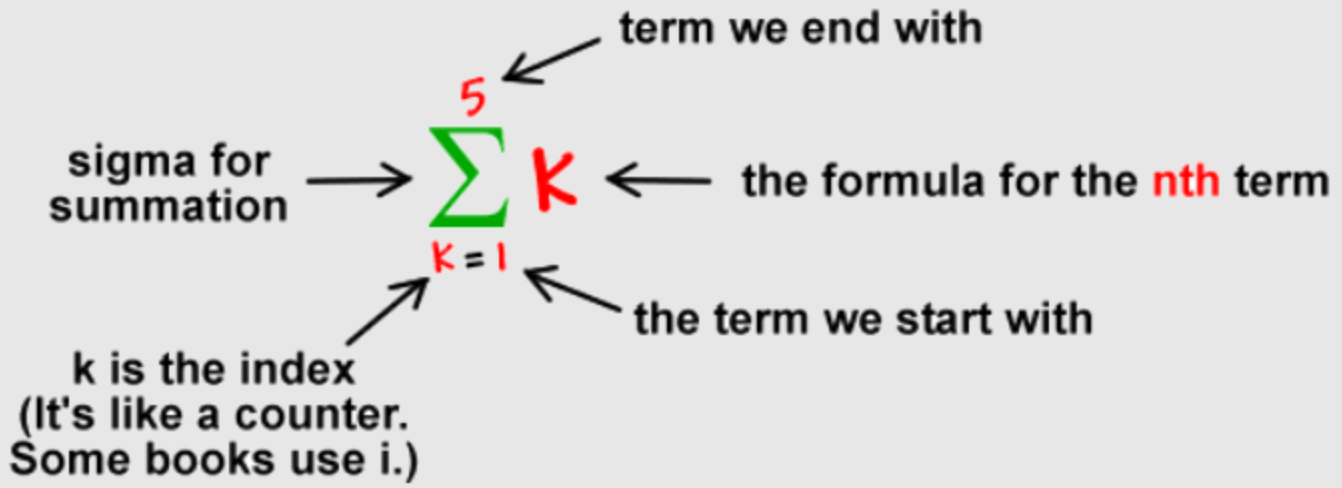


〈https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53〉

All the inputs x are multiplied with their weights w.
(In this figure, Bias is omitted)

➔ $K = w^T \cdot x$

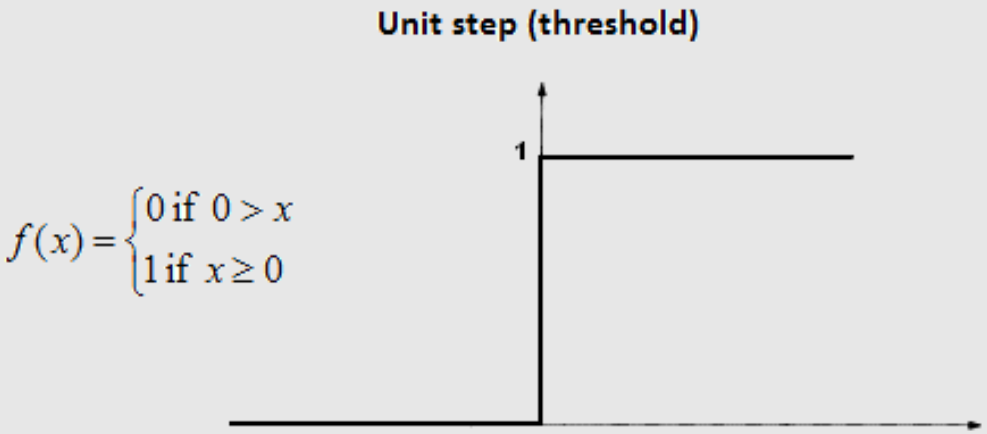
1957: Perceptron (Frank Rosenblatt)



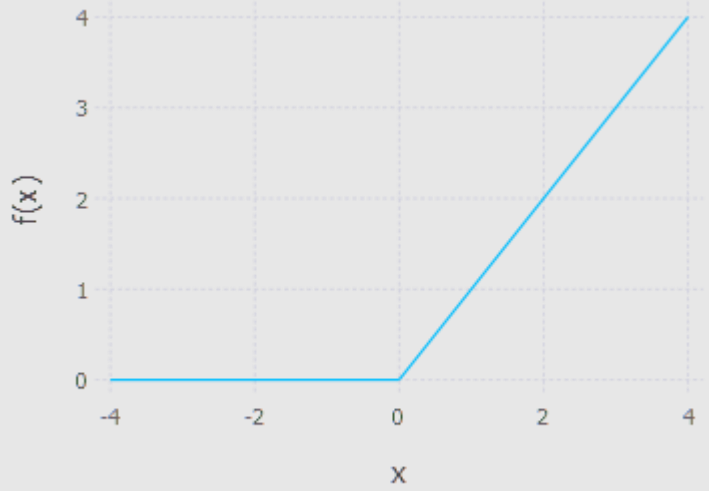
〈https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53〉

Add all the multiplied values and call them ‘Weighted Sum’.
And apply that weighted sum to the correct activation function.

1957: Perceptron (Frank Rosenblatt)



<https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53>

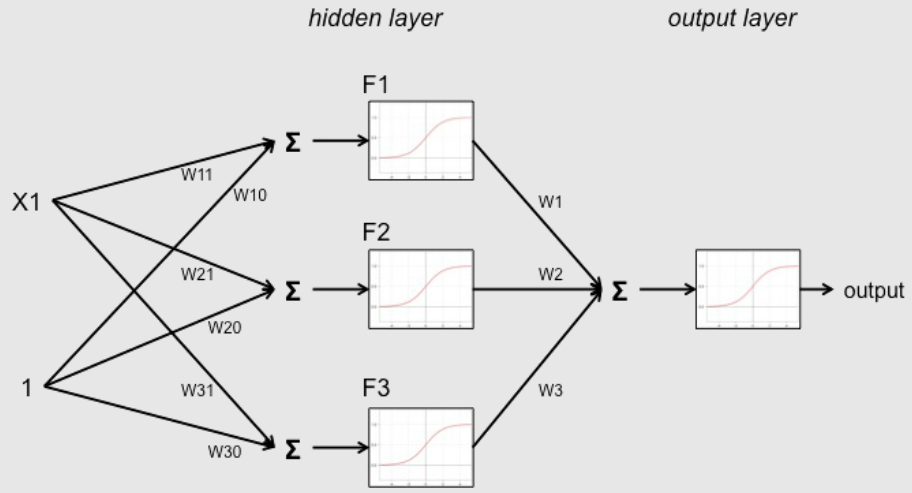


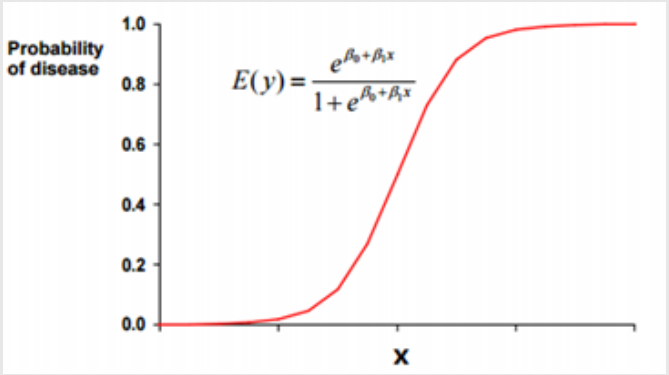
<https://int8.io/neural-networks-in-julia-hyperbolic-tangent-and-relu/>

It's just a thing (node) that you add to the output end of any neural network. It is also known as Transfer Function. It can also be attached in between two Neural Networks.

It is used to determine the output of neural network like yes or no. It maps the resulting values in between 0 to 1 or -1 to 1 etc. (depending upon the function).

1957: Perceptron





➔Feedforward Networks

These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. There are no feedback connections in which outputs of the model are fed back into itself.

(이러한 모델은 feedforward라고 불리며, 정보는 x에서 평가되는 함수를 통해 전달되고, f를 정의하는데 사용되는 중간 계산을 거쳐 최종적으로 출력 y로 전달되기 때문입니다. 모델의 출력이 자체로 피드백 되는 피드백 연결은 없습니다.)

1957: Perceptron

Feedforward networks are of extreme importance to machine learning practitioners. They form the basis of many important commercial applications. For example, the convolutional networks used for object recognition from photos are a specialized kind of feedforward network. Feedforward networks are a conceptual stepping stone on the path to recurrent networks, which power many natural language applications.

(Feedforward네트워크는 기계학습에서 매우 중요합니다. 그것들은 많은 중요한 상업적 응용의 기초를 형성합니다. 예를 들어 사진에서 물체를 인식하는데 사용되는 CNN네트워크는 피드포워드 네트워크의 특수케이스입니다. 피드포워드 네트워크는 많은 자연어 응용 프로그램을 구동하는 반복적인 네트워크 경로의 개념적 디딤돌입니다.)

Feedforward neural networks are called networks because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we might have three functions $f^{(1)}, f^{(2)}$ and $f^{(3)}$ connected in a chain, to form $f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(x)\right)\right)$. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is called the second layer, and so on. The overall length of the chain gives the length of the model. It is from this terminology that the name 'deep learning' arises. The final layer of a feedforward network is called the output layer. During neural network training, we drive $f(x)$ to match $f^*(x)$.

(피드포워드 신경망은 일반적으로 많은 다른 기능을 함께 구성함으로써 표현되기 때문에 네트워크라고 합니다. 모델은 함수가 함께 구성되는 방법을 설명하는 방향성이 있는 비순환적 그래프와 연관됩니다. 예를 들어 $f^{(1)}, f^{(2)}, f^{(3)}$ 이렇게 3개의 함수가 체인으로 연결되어 있을 때 $f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(x)\right)\right)$로 나타낼 수 있습니다. 이러한 체인 구조는 뉴럴 네트워크의 가장 일반적인 구조입니다. 이 경우에는 $f^{(1)}$는 첫 번째 층이 되며, $f^{(2)}$는 두 번째 층이 되는 식으로 이어집니다. 전체 길이는 모델의 층이 됩니다. 이것이 딥러닝이라는 용어가 나오게 된 계기입니다. 피드포워드 네트워크의 최종 층을 출력계층이라고 합니다. 신경 회로망 훈련 동안, 우리는 $f(x)$를 $f^*(x)$로 유도합니다.)

〈https://www.deeplearningbook.org/〉

1957: Perceptron

Perceptron Convergence Theorem

If $\mathcal{X}_m = \{(\mathbf{x}_1, \ell_1), \ldots, (\mathbf{x}_m, \ell_m)\}$ describes a linearly separable dichotomy, then the fixed-increment perceptron algorithm terminates after a finite number of weight updates.

A geometric picture of the perceptron algorithm emerges after transforming into *augmented* (or *homogeneous coordinates*). Let

$$\widehat{\mathbf{x}} = \begin{pmatrix} 1 \\ \mathbf{x} \end{pmatrix} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^{n+1}; \qquad \text{and let,} \qquad \widehat{\mathbf{w}} = \begin{pmatrix} w_0 \\ \mathbf{w} \end{pmatrix} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix} \in \mathbb{R}^{n+1}.$$

In the following, *hatted* vectors will always represent augmented vectors. Similarly, let

$$\widehat{\mathcal{X}}_m = \left\{ (\widehat{\mathbf{x}}_i, \ell_i) = ((1, \mathbf{x}_i^T)^T, \ell_i) \,\middle|\, i = 1, 2, \ldots, m \right\}$$

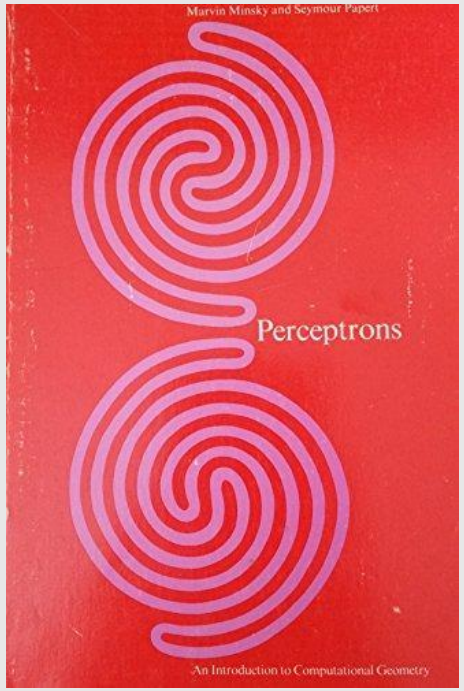
$$w_{i,j}^{(next\ step)} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

Perceptron Convergence Theorem

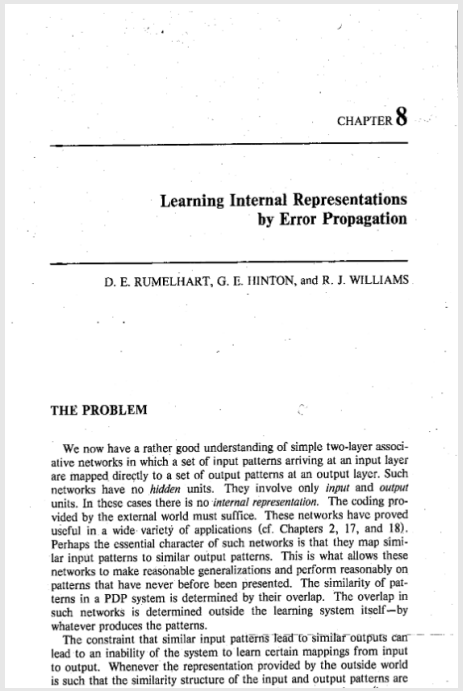
History

1995: Support-Vector Machine
1999: Convolution Neural Network
2006: Deep Belief Network

1969



1986



CHAPTER 8

Learning Internal Representations by Error Propagation

D. E. RUMELHART, G. E. HINTON, and R. J. WILLIAMS

THE PROBLEM

We now have a rather good understanding of simple two-layer associative networks in which a set of input patterns arriving at an input layer are mapped directly to a set of output patterns at an output layer. Such networks have no *hidden* units. They involve only *input* and *output* units. In these cases there is no *internal representation*. The coding provided by the external world must suffice. These networks have proved useful in a wide variety of applications (cf. Chapters 2, 17, and 18). Perhaps the essential character of such networks is that they map similar input patterns to similar output patterns. This is what allows these networks to make reasonable generalizations and perform reasonably on patterns that have never before been presented. The similarity of patterns in a PDP system is determined by their overlap. The overlap in such networks is determined outside the learning system itself—by whatever produces the patterns.

The constraint that similar input patterns lead to similar outputs can lead to an inability of the system to learn certain mappings from input to output. Whenever the representation provided by the outside world is such that the similarity structure of the input and output patterns are

2011

Deep Sparse Rectifier Neural Networks

Xavier Glorot
DIRO, Université de Montréal
Montréal, QC, Canada
glorotxa@iro.umontreal.ca

Antoine Bordes
Heudiasyc, UMR CNRS 6599
UTC, Compiègne, France
and
DIRO, Université de Montréal
Montréal, QC, Canada
antoine.bordes@hds.utc.fr

Yoshua Bengio
DIRO, Université de Montréal
Montréal, QC, Canada
bengioy@iro.umontreal.ca

Abstract

While logistic sigmoid neurons are more biologically plausible than hyperbolic tangent neurons, the latter work better for training multi-layer neural networks. This paper shows that rectifying neurons are an even better model of biological neurons and yield equal or better performance than hyperbolic tangent networks in spite of the hard non-linearity and non-differentiability at zero, creating sparse representations with true zeros, which seem remarkably suitable for naturally sparse data. Even though they can take advantage of semi-supervised setups with extra-unlabeled data, deep rectifier networks can reach their best performance without requiring any unsupervised pre-training on purely supervised tasks with large labeled datasets. Hence, these results can be seen as a new milestone in the attempts at understanding the difficulty in training deep but purely supervised neural networks, and closing the performance gap between neural networks learnt with and without unsupervised pre-training.

1 Introduction

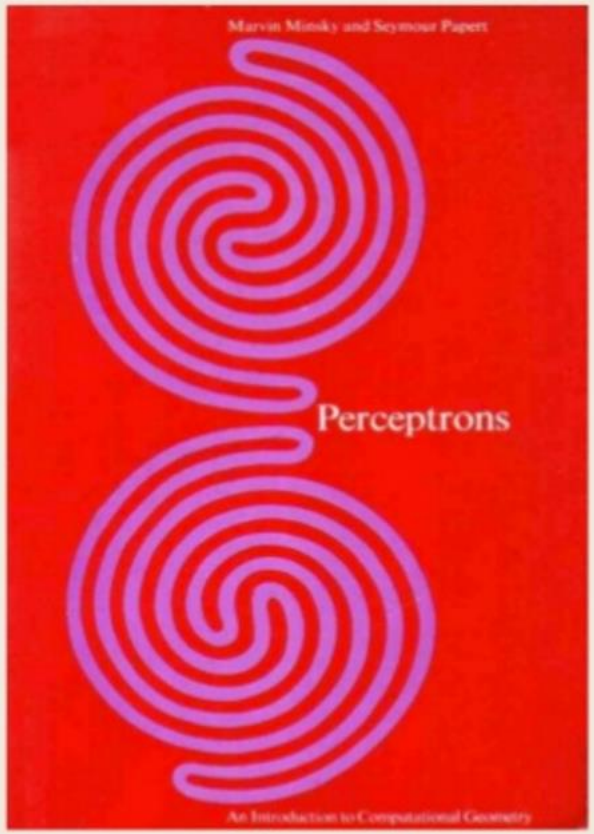
Many differences exist between the neural network models used by machine learning researchers and those used by computational neuroscientists. This is in part because the objective of the former is to obtain computationally efficient learners, that generalize well to new examples, whereas the objective of the latter is to abstract out neuroscientific data while obtaining explanations of the principles involved, providing predictions and guidance for future biological experiments. Areas where both objectives coincide are therefore particularly worthy of investigation, pointing towards computationally motivated principles of operation in the brain that can also enhance research in artificial intelligence. In this paper we show that two common gaps between computational neuroscience models and machine learning neural network models can be bridged by using the following linear by part activation : $\max(0, x)$, called the rectifier (or hinge) activation function. Experimental results will show engaging training behavior of this activation function, especially for *deep architectures* (see Bengio (2009) for a review), i.e., where the number of hidden layers in the neural network is 3 or more.

Recent theoretical and empirical work in statistical machine learning has demonstrated the importance of learning algorithms for deep architectures. This is in part inspired by observations of the mammalian visual cortex, which consists of a chain of processing elements, each of which is associated with a different representation of the raw visual input. This is particularly clear in the primate visual system (Serre *et al.*, 2007), with its sequence of processing stages: detection of edges, primitive shapes, and moving up to gradually more complex visual shapes. Interestingly, it was found that the features learned in deep architectures resemble those observed in the first two of these stages (in areas V1 and V2 of visual cortex) (Lee *et al.*, 2008), and that they become increasingly invariant to factors of variation (such as camera movement) in higher layers (Goodfellow *et al.*, 2009).

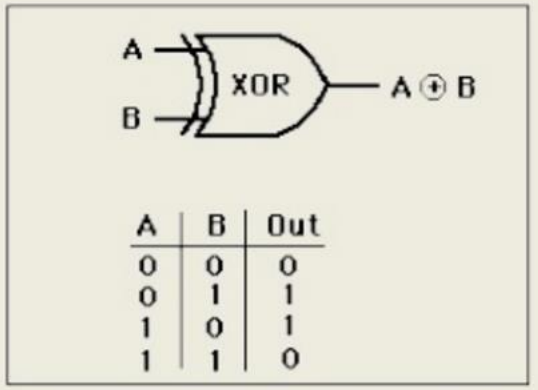
Appearing in Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS) 2011, Fort Lauderdale, FL, USA. Volume 15 of JMLR: W&CP 15.

1969: Perceptron (Marvin Minsky)

1969: Perceptrons can't do XOR!



http://www.i-programmer.info/images/stories/BabBag/AI/book.jpg



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

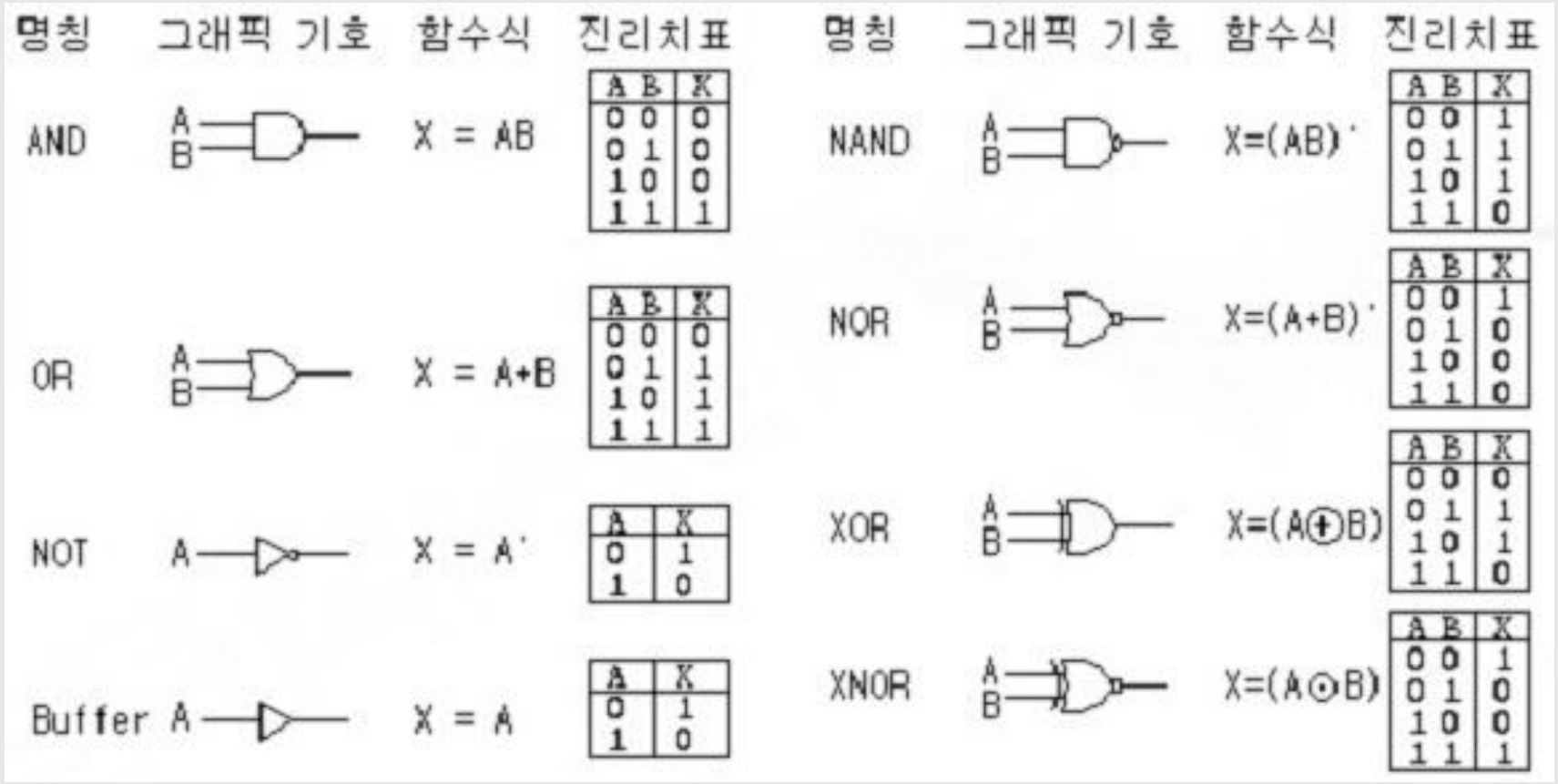
http://hyperphysics.phy-astr.gsu.edu/hbase/electronic/ietron/xor.gif

Minsky & Papert

https://constructingkids.files.wordpress.com/2013/05/minsky-papert-71-csolomon-x640.jpg

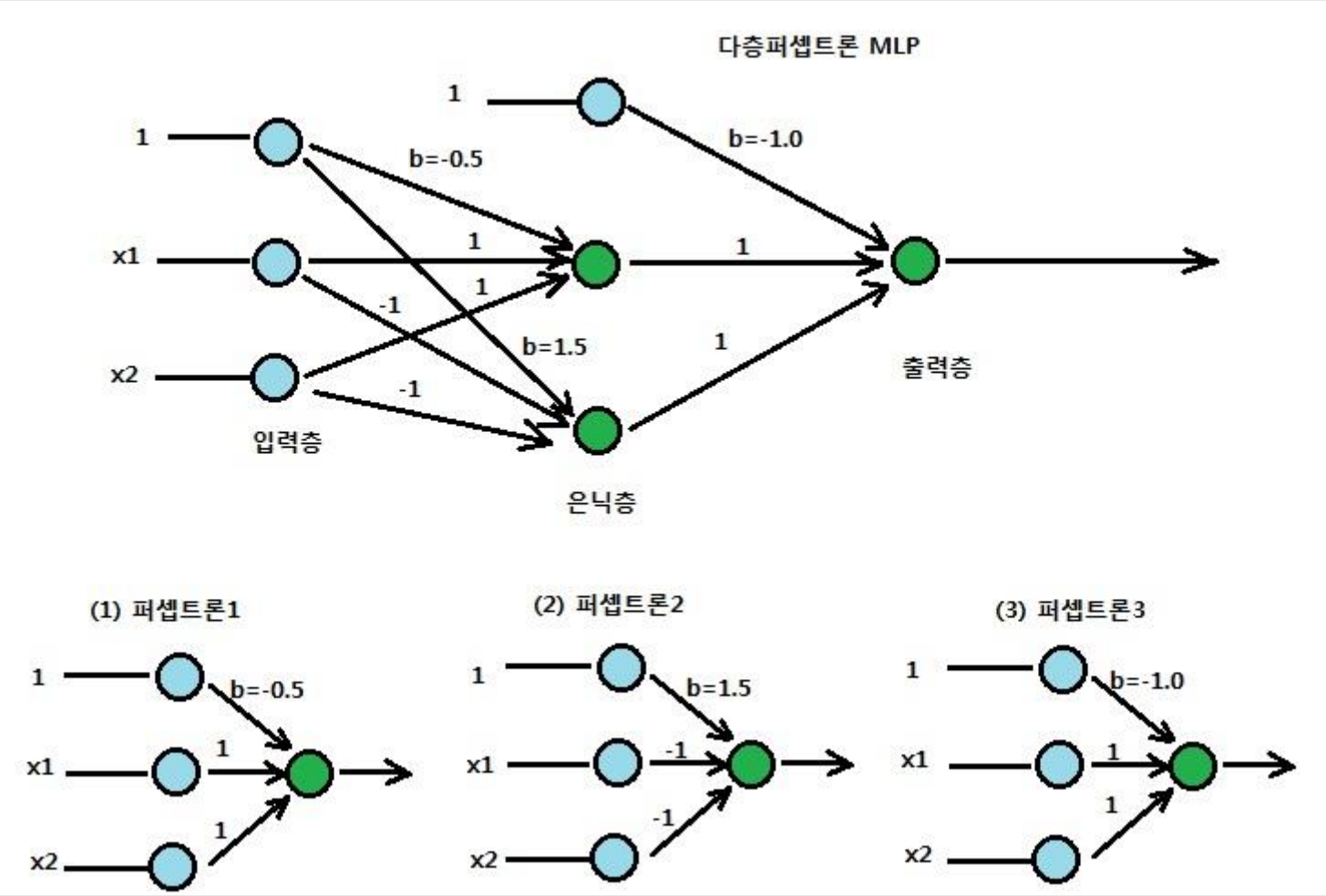
〈https://pmirla.github.io/2016/08/16/AI-Winter.html〉

1969: Perceptron (Marvin Minsky)



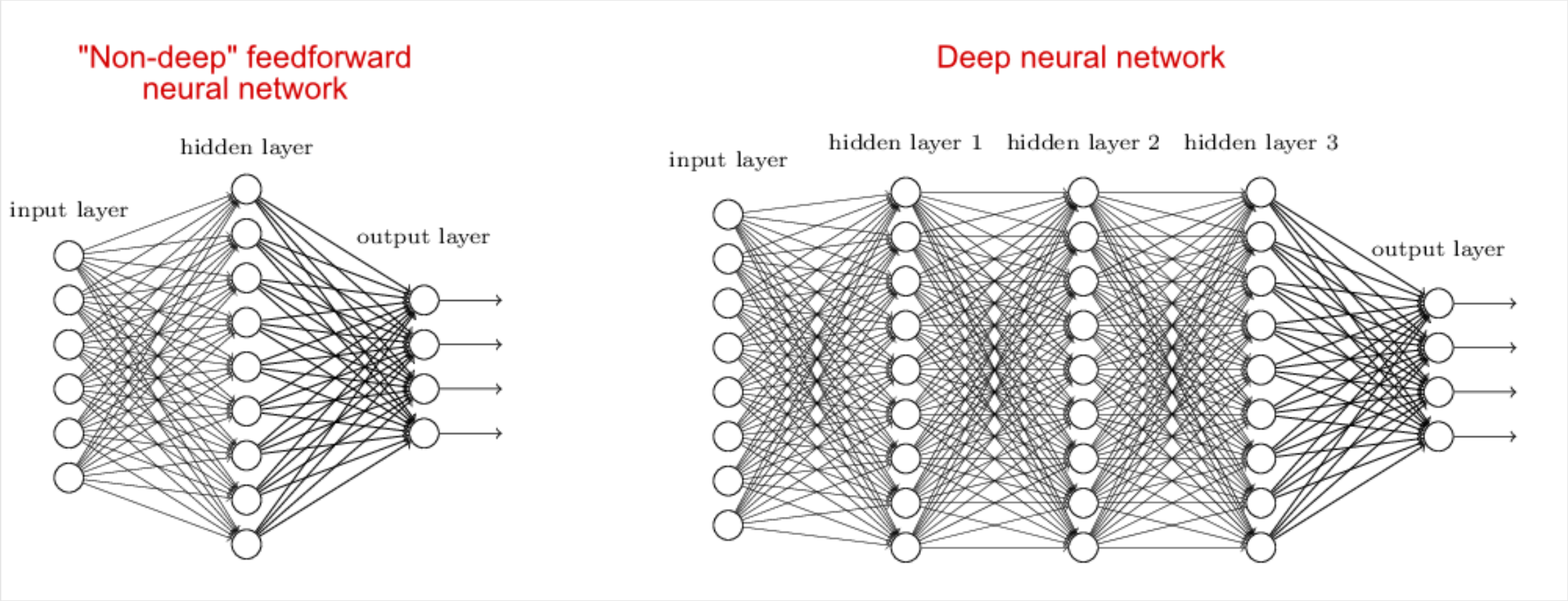
<http://tetrackis.tistory.com/entry/AND%EA%B2%8C%EC%9D%B4%ED%8A%B8-OR%EA%B2%8C%EC%9D%B4%ED%8A%B8-NOT%EA%B2%8C%EC%9D%B4%ED%8A%B8-NAND%EA%B2%8C%EC%9D%B4%ED%8A%B8-NOR%EA%B2%8C%EC%9D%B4%ED%8A%B8-XOR%EA%B2%8C%EC%9D%B4%ED%8A%B8%EC%97%90-%EB%8C%80%ED%95%B4-%EC%A1%B0%EC%82%AC%ED%95%98%EC%8B%9C%EC%98%A4>

1969: Perceptron (Marvin Minsky)



〈https://m.blog.naver.com/PostView.nhn?blogId=msnayana&logNo=80190275869&proxyReferer=https%3A%2F%2Fwww.google.com%2F〉

1969: Perceptron (Marvin Minsky)

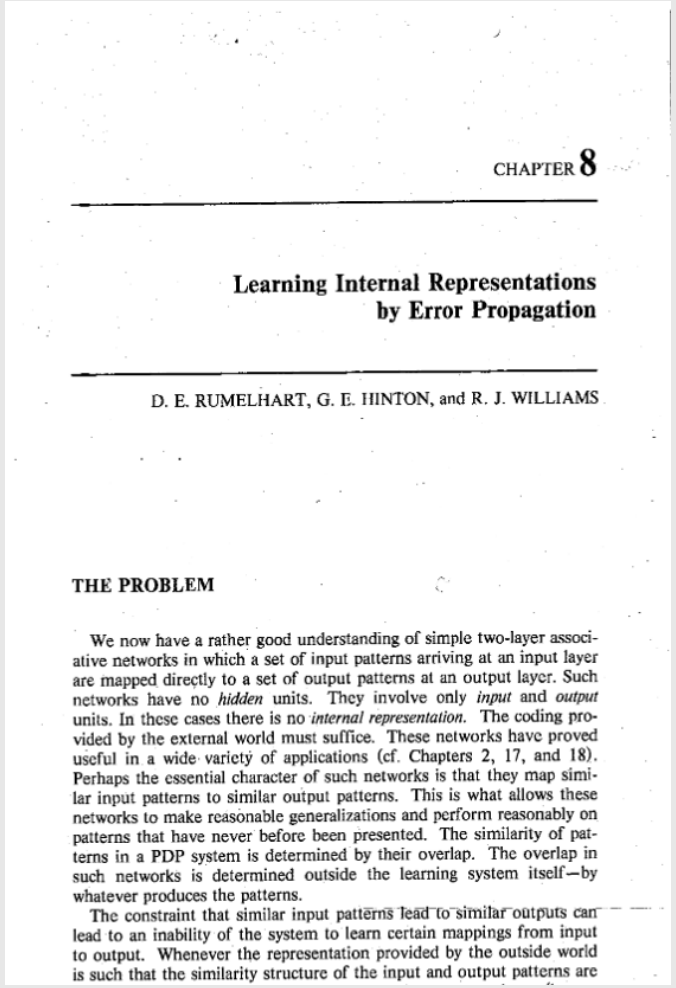


<https://stats.stackexchange.com/questions/182734/what-is-the-difference-between-a-neural-network-and-a-deep-neural-network-and-w>

1986: Backpropagation

단일 층이라면 Logistic과 같게 진행하면 된다.
➔Cross-entropy활용

다층일 땐?



CHAPTER 8

Learning Internal Representations by Error Propagation

D. E. RUMELHART, G. E. HINTON, and R. J. WILLIAMS

THE PROBLEM

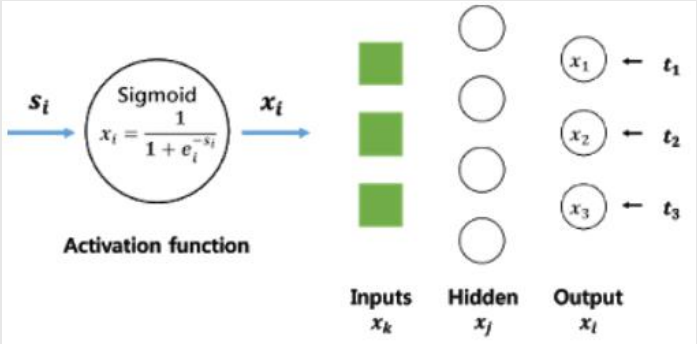
We now have a rather good understanding of simple two-layer associative networks in which a set of input patterns arriving at an input layer are mapped directly to a set of output patterns at an output layer. Such networks have no *hidden* units. They involve only *input* and *output* units. In these cases there is no *internal representation*. The coding provided by the external world must suffice. These networks have proved useful in a wide variety of applications (cf. Chapters 2, 17, and 18). Perhaps the essential character of such networks is that they map similar input patterns to similar output patterns. This is what allows these networks to make reasonable generalizations and perform reasonably on patterns that have never before been presented. The similarity of patterns in a PDP system is determined by their overlap. The overlap in such networks is determined outside the learning system itself—by whatever produces the patterns.

The constraint that similar input patterns lead to similar outputs can lead to an inability of the system to learn certain mappings from input to output. Whenever the representation provided by the outside world is such that the similarity structure of the input and output patterns are

1986: Backpropagation

“it is easy to fall into the trap of abstracting away the learning process – believing that you can simply stack arbitrary layers together and backprop will “magically make them work” on your data”
– Andrej Karpathy

1986: Backpropagation



앞서 문제를 설정한대로 우리의 error function, E는 다음과 같습니다.

$$E = -\sum_{i=1}^{nout}(t_i log(x_i) + (1 - t_i) log(1 - x_i))$$

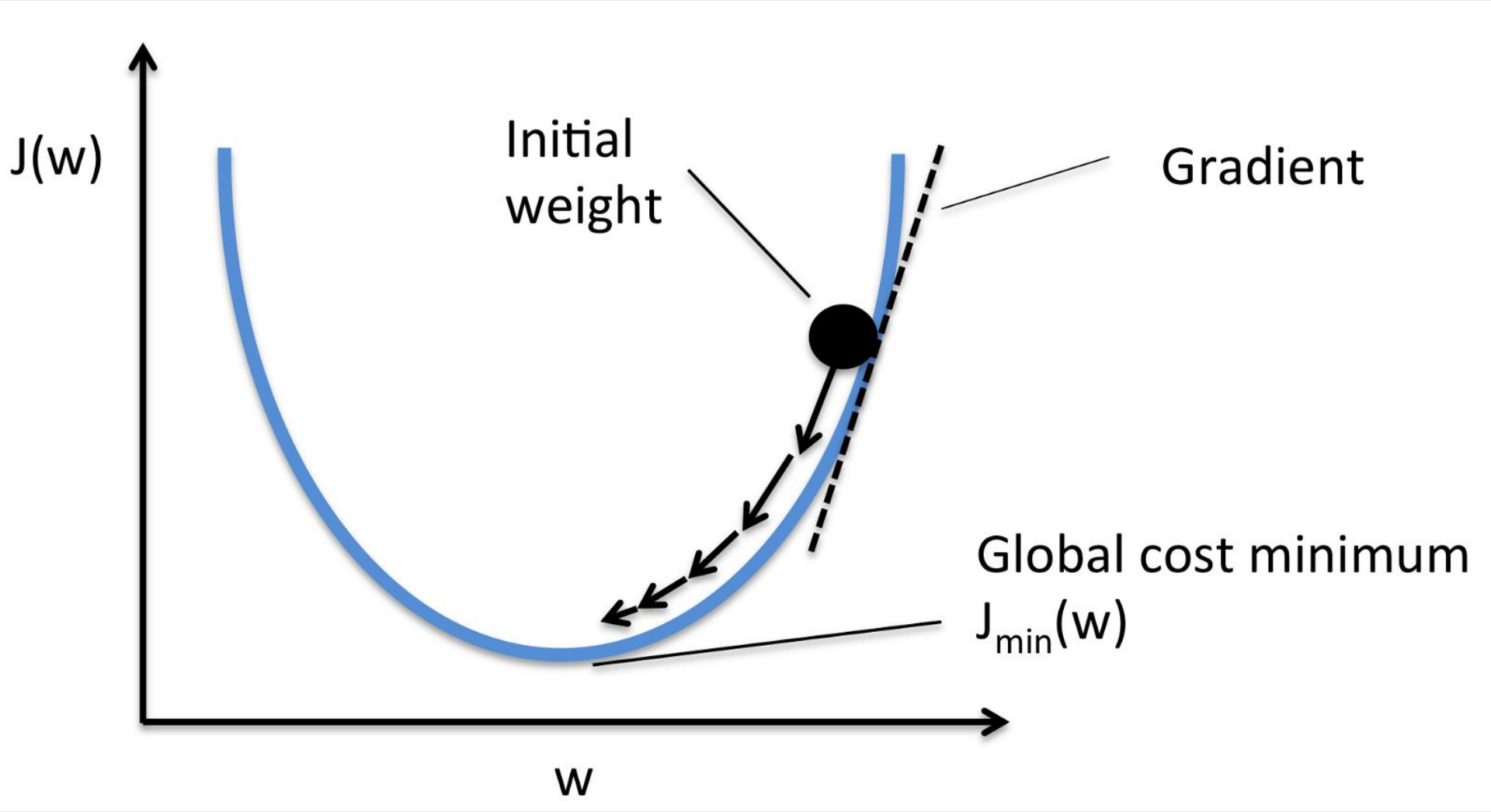
where, $x_i = \frac{1}{1+e^{-s_i}}$, $s_i = \sum_{j=1} x_j w_{ji}$.

수식이 무엇을 의미하는지 직관적으로 이해하려면 극단적인 예제를 넣어보는게 가장 빠르죠 ㅎㅎ 실제 class가 1일 때($t_i = 1$), 신경망이 계산한 답이 1일 확률이 0이라는($x_i = 0$) output 결과가 나왔다고 해보고 E 값을 계산해보면 쉽게 무한대가 나오는 것을 알 수 있습니다.
반대로 제대로 확률 값이 1 즉, target t_i의 class가 1일 때, 실제로 1이라고 계산할 확률이 100%라고($x_i = 1$) 신경망이 결과를 내었다면, E 값은 0으로 최소값을 갖습니다.

〈http://jaejunyoo.blogspot.com/2017/01/backpropagation.html〉

➔ 즉, 우리의 목표는 E값을 줄이는 것

1986: Backpropagation

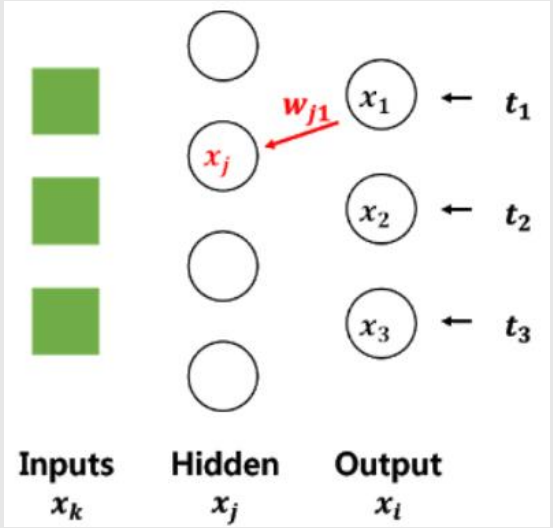


〈https://hackernoon.com/gradient-descent-aynk-7cbe95a778da〉

1986: Backpropagation

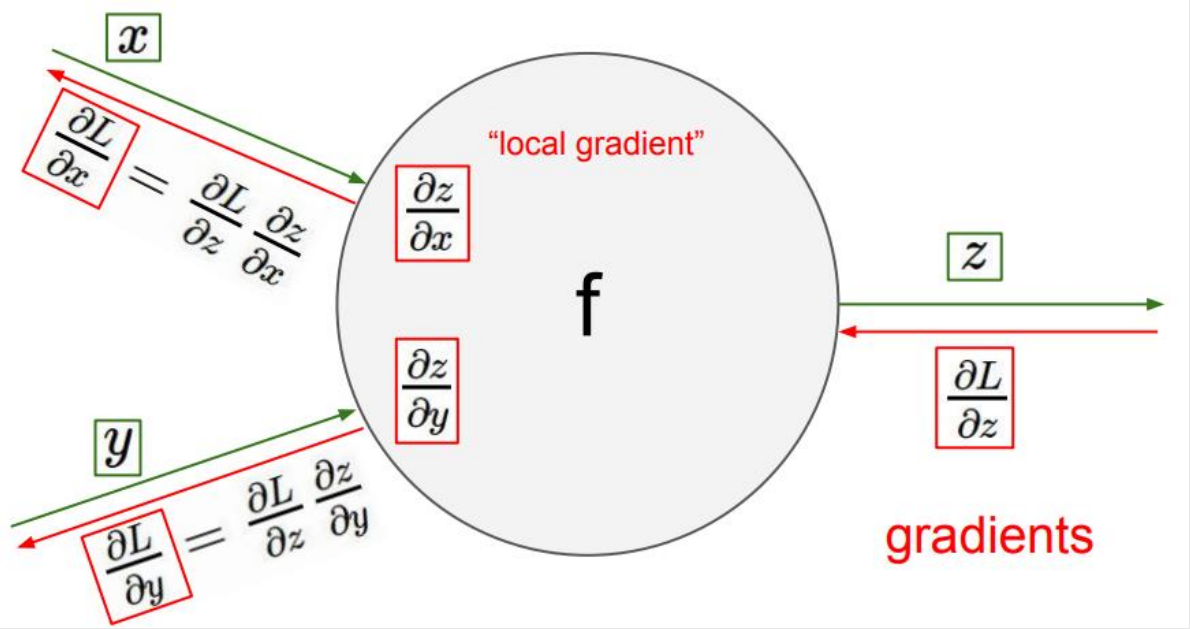
- 어떤 함수의 f의 임의의 점 x에서의 기울기(gradient)를 계산하면 x로부터 한 단위(Δx)만큼 움직였을 때, f값이 변화하는 양을 표현하는 벡터가 나온다.
- 이 벡터는 항상 가장 가파르게 f값이 증가하는 방향을 가리킨다.
- 고로 함수 f를 오류 식으로 놓고 기울기가 가리키는 반대방향(descent)으로 네트워크를 업데이트 해준다.

➔ 우리가 원하는 것은 Data의 Input의 변화가 아니기 때문에 바꾸는 것은 Weight와 Bias.
➔ 따라서 우리는 각 Layer의 Weight에 대한 함수의 Gradient를 계산하고,
이 변화량을 원래 w의 반대 반향으로 더해주는 것.



〈http://jaejunyoo.blogspot.com/2017/01/backpropagation.html〉

1986: Backpropagation



https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199

각 훈련 샘플에 대해 역전파 알고리즘이 먼저 예측을 만들고(정방향 계산), 오차를 측정하고, 그런 다음 역방향으로 각 층을 거치면서 각 연결이 오차에 기여한 정도를 측정합니다(역방향 계산). 마지막으로 이 오차가 감소하도록 가중치를 조금씩 조정합니다 (경사 하강법 스텝).

〈Hands-on Machine Learning〉

1986: Backpropagation

돌아갈 때 순서대로 가장 바깥 output과 hidden-layer사이의 weight부터 바꿔보겠다.
i번째 output인 x_i에 대한 오차 값에 대해 w_{ji}가 바뀌어야 하는 정도를 계산하려면 다음의 수식이 필요하다.

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial x_i}\frac{\partial x_i}{\partial s_i}\frac{\partial s_i}{\partial w_{ji}}$$

$$1.\ \frac{\partial E}{\partial x_i} = \frac{-t_i}{x_i} + \frac{1-t_i}{1-x_i} = \frac{x_i - t_i}{x_i(1-x_i)}$$

$$2.\ \frac{\partial x_i}{\partial s_i} = x_i(1-x_i)$$

$$3.\ \frac{\partial s_i}{\partial w_{ji}} = x_j$$

$$\therefore \frac{\partial E}{\partial w_{ji}} = x_j$$

〈http://jaejunyoo.blogspot.com/2017/01/backpropagation.html〉

1986: Backpropagation

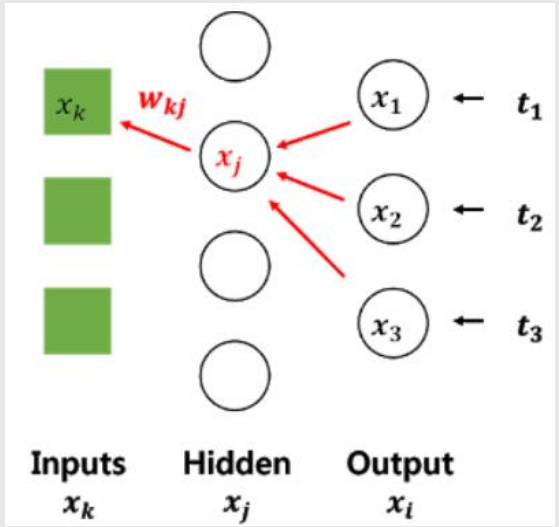
같은 논리로 그 다음 hidden과 input-layer간의 weight w_{kj}에 대해 변화량을 계산하면

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial s_j}\frac{\partial s_j}{\partial w_{kj}}$$

$$1.\ \frac{\partial E}{\partial s_j} = \sum_{i=1}^{nout} \frac{\partial E}{\partial s_i}\frac{\partial s_i}{\partial x_j}\frac{\partial x_j}{\partial s_j} = \sum_{i=1}^{nout}(x_i - t_i)(w_{ji})\left(x_j(1-x_j)\right)$$

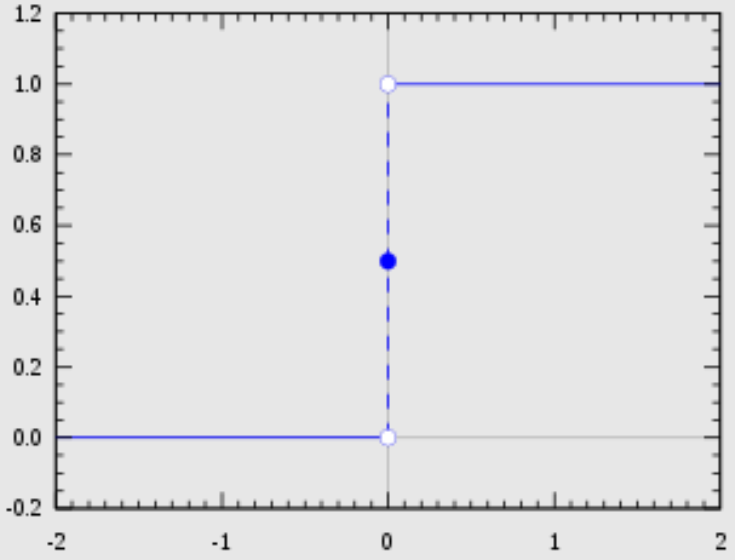
$$2.\ \frac{\partial s_j}{\partial w_{kj}} = x_k\left(\because s_j = \sum_{k=1} x_k w_{kj}\right)$$

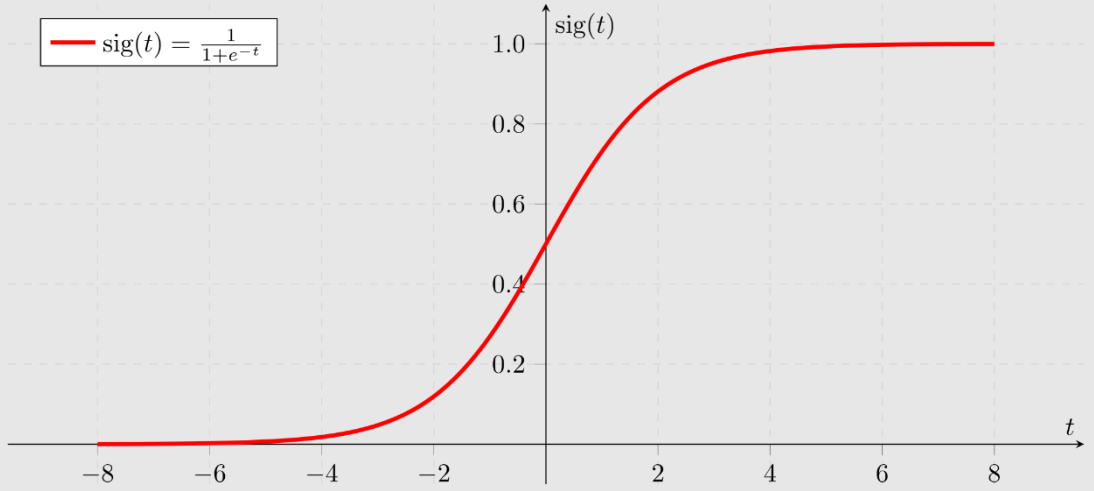
$$\therefore \frac{\partial E}{\partial w_{kj}} = \sum_{i=1}^{nout}(x_i - t_i)(w_{ji})\left(x_j(1-x_j)\right)(x_k)$$



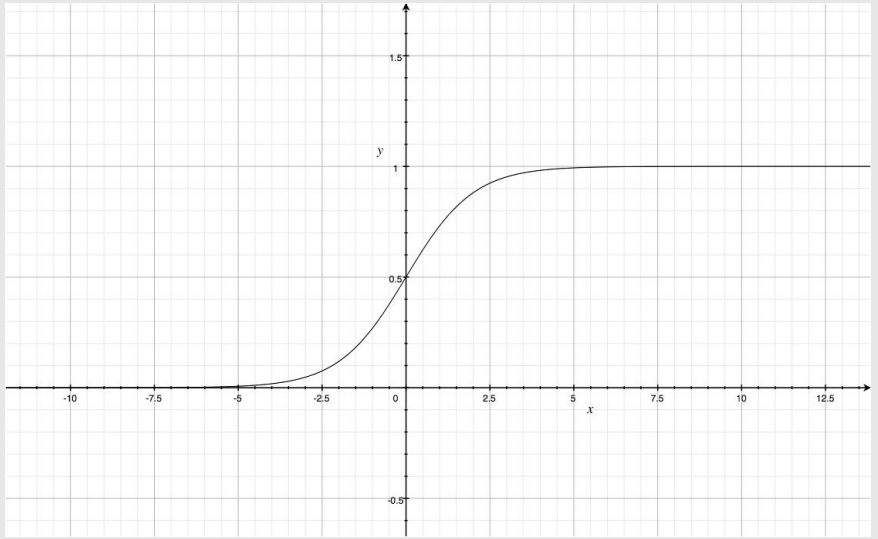
〈http://jaejunyoo.blogspot.com/2017/01/backpropagation.html〉

1986: Backpropagation





Activation Function: Sigmoid

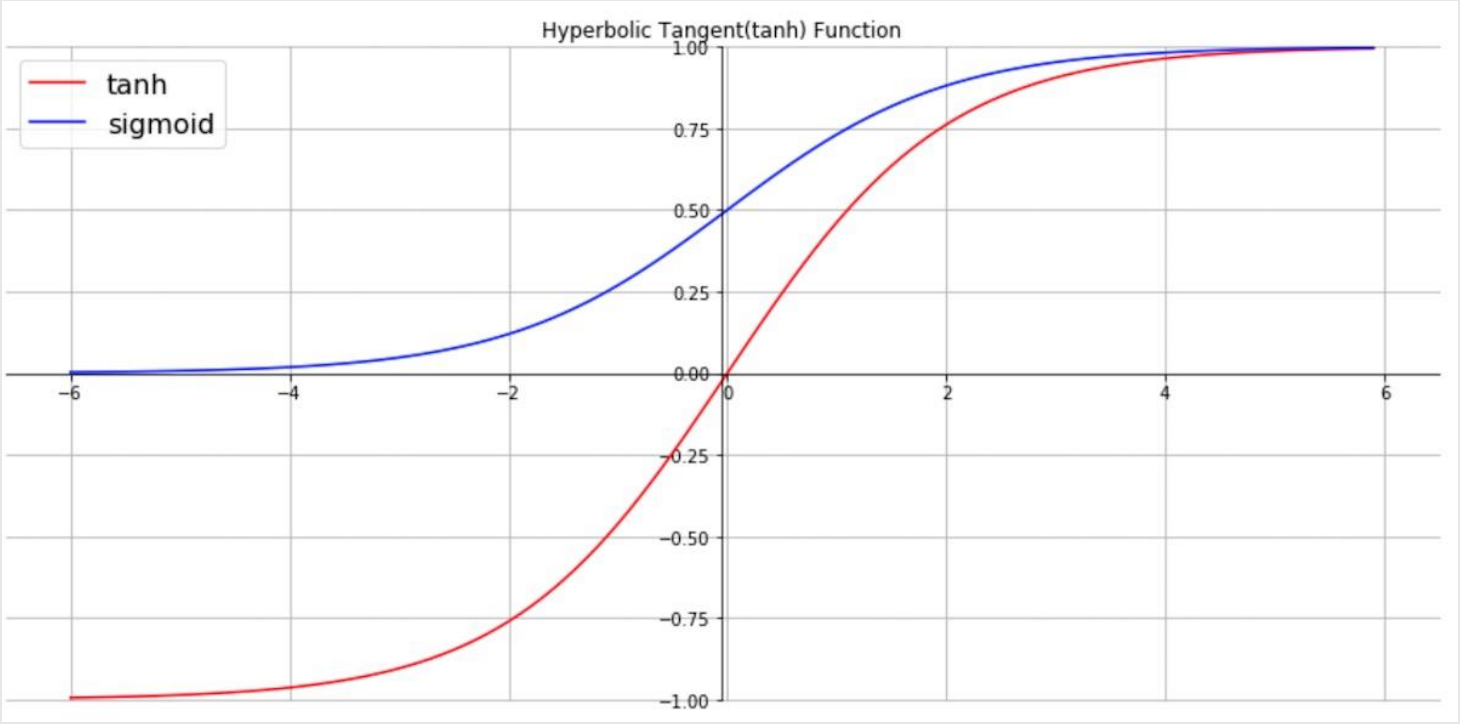


〈https://icim.nims.re.kr/post/easyMath/64〉

$$s(z) = \frac{1}{1 + e^{-z}}$$

- 성공과 실패를 구분하는 부분은 경사가 급하고 나머지 부분에서는 경사가 완만하다.
- y=1, y=0 두 평행선이 점근선이고 치역은 (0,1)이다.
  즉 위와 같은 Activation Function의 함수 값은 성공확률이라는 의미로 해석할 수 있다.
- **미분이 가능하다.**

Activation Function: Hyperbolic Tangent



〈http://taewan.kim/post/tanh_diff/〉

- Sigmoid의 대체제
- 출력 범위가 (0,1)이 아닌 (−1,1)
  ➔ 출력 범위가 더 넓고 경사면이 큰 범위가 더 크기 때문에 더 빠르게 수렴

Gradient Vanishing

다음은 반복 최적화 알고리즘(Iterative Optimization Algorithm)에 대해 알고있는 것입니다. 즉, 비용 함수의 출력이 감소하도록 그라디언트에서 추론 된 방향으로 가중치를 교란하여 로컬 최적 값으로 천천히갑니다. 그래디언트 디센트 알고리즘은 특히 0에서 1 사이의 작은 스칼라 값을 곱한 그래디언트의 음수로 가중치를 업데이트합니다.

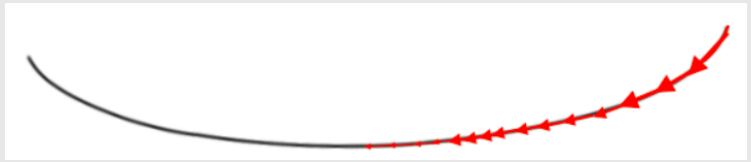
$$repeat\ \ until\ \ \frac{\partial J}{\partial W^{layer}_{ij}} \rightarrow 0:$$

$$\{\quad W^{layer}_{ij} := W^{layer}_{ij} - \alpha \frac{\partial J}{\partial W^{layer}_{ij}}$$

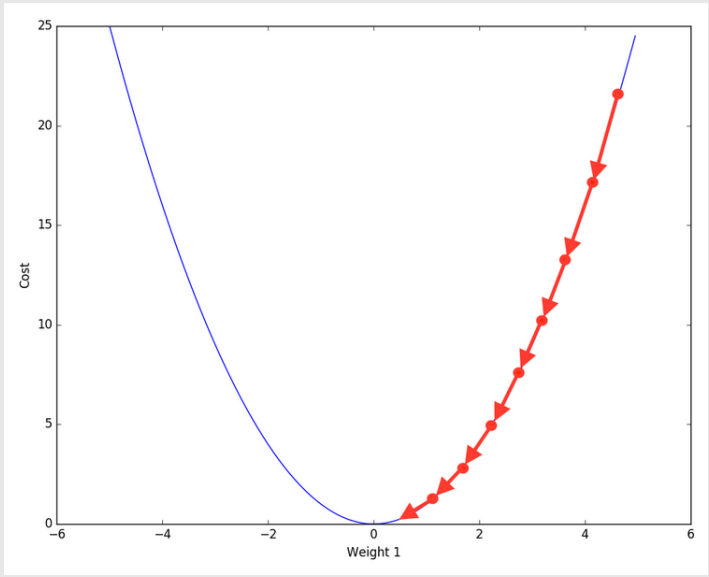
보시다시피 컨버전스까지 "반복"해야 합니다. 실제로는 최대 반복 횟수에 대한 하이퍼 파라미터를 설정했습니다. 특정 Deep Neural Network에 대한 반복 횟수가 너무 적으면 부정확 한 결과가 나타납니다. 숫자가 너무 많으면 훈련 기간이 비현실적으로 길어집니다. 그것은 훈련 시간과 정확성 사이의 불안정한 절충점입니다.

〈https://brunch.co.kr/@chris-song/39〉

Gradient Vanishing



VS

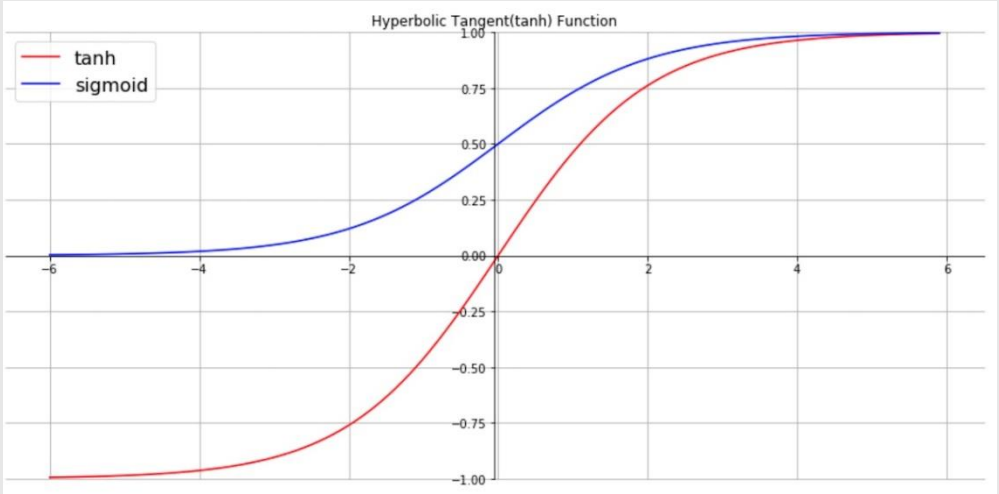


〈https://brunch.co.kr/@chris-song/39〉

- Vanishing gradient problem은 activation function을 선택하는 문제에 의존적으로 발생
- 각 단계의 기울기가 너무 작으면, 반복이 발생할 때 마다 Weight가 충분히 변하지 않기 때문에 더 큰 반복이 수렴 될 때까지 필요
- 설정된 반복 횟수에서 Weight가 최소에 근접하지 않을 수 있고 정말 작은 Gradient로는 훈련 자체가 불가능

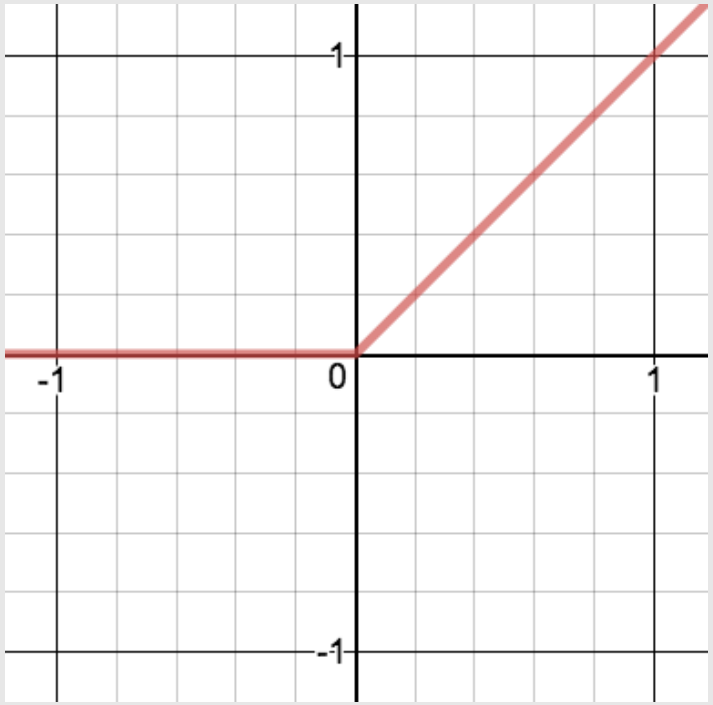
Gradient Vanishing

- Sigmoid, Tanh, 등 요즘 많이 사용하는 activation function은 매우 비선형적인 방식으로 그들의 input을 매우 작은 output range로 squash해서 넣음
  ➔ 이렇게 되어 버린 input space에서는 큰 변화가 있더라도, output에서 작은 변화를 보인다. (Gradient가 작기 때문)
- 이러한 현상은 여러 레이어로 쌓을 때 더욱 악화된다.

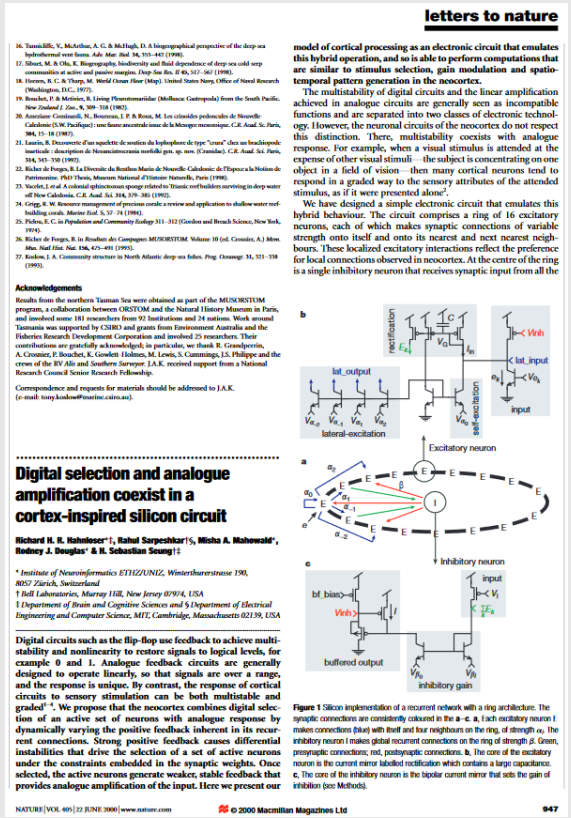


〈http://taewan.kim/post/tanh_diff/〉

Activation Function: Rectified Linear Units (ReLU)



〈https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html〉



letters to nature

Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit

Richard H. R. Hahnloser*†, Rahul Sarpeshkar†§, Misha A. Mahowald*, Rodney J. Douglas* & H. Sebastian Seung†‡

"Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit"
〈Hahnloser et al. 2000〉

Deep Sparse Rectifier Neural Networks

Xavier Glorot
DIRO, Université de Montréal
Montréal, QC, Canada
glorotxa@iro.umontreal.ca

Antoine Bordes
Heudiasyc, UMR CNRS 6599
UTC, Compiègne, France
and
DIRO, Université de Montréal
Montréal, QC, Canada
antoine.bordes@hds.utc.fr

Yoshua Bengio
DIRO, Université de Montréal
Montréal, QC, Canada
bengioy@iro.umontreal.ca

Abstract

While logistic sigmoid neurons are more biologically plausible than hyperbolic tangent neurons, the latter work better for training multi-layer neural networks. This paper shows that rectifying neurons are an even better model of biological neurons and yield equal or better performance than hyperbolic tangent networks in spite of the hard non-linearity and non-differentiability at zero, creating sparse representations with true zeros, which seem remarkably suitable for naturally sparse data. Even though they can take advantage of semi-supervised setups with extra-unlabeled data, deep rectifier networks can reach their best performance without requiring any unsupervised pre-training on purely supervised tasks with large labeled datasets. Hence, these results can be seen as a new milestone in the attempts at understanding the difficulty in training deep but purely supervised neural networks, and closing the performance gap between neural networks learnt with and without unsupervised pre-training.

1 Introduction

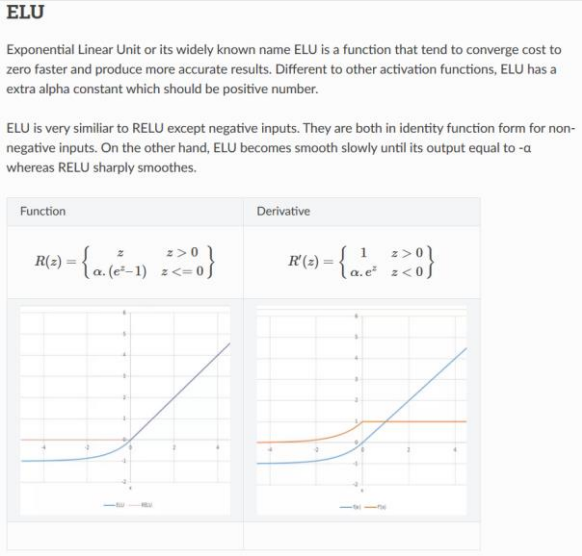
Many differences exist between the neural network models used by machine learning researchers and those used by computational neuroscientists. This is in part because the objective of the former is to obtain computationally efficient learners, that generalize well to new examples, whereas the objective of the latter is to abstract out neuroscientific data while obtaining explanations of the principles involved, providing predictions and guidance for future biological experiments. Areas where both objectives coincide are therefore particularly worthy of investigation, pointing towards computationally motivated principles of operation in the brain that can also enhance research in artificial intelligence. In this paper we show that two common gaps between computational neuroscience models and machine learning neural network models can be bridged by using the following linear by part activation : $\max(0, x)$, called the rectifier (or hinge) activation function. Experimental results will show engaging training behavior of this activation function, especially for *deep architectures* (see Bengio (2009) for a review), i.e., where the number of hidden layers in the neural network is 3 or more.

Recent theoretical and empirical work in statistical machine learning has demonstrated the importance of learning algorithms for deep architectures. This is in part inspired by observations of the mammalian visual cortex, which consists of a chain of processing elements, each of which is associated with a different representation of the raw visual input. This is particularly clear in the primate visual system (Serre *et al.*, 2007), with its sequence of processing stages: detection of edges, primitive shapes, and moving up to gradually more complex visual shapes. Interestingly, it was found that the features learned in deep architectures resemble those observed in the first two of these stages (in areas V1 and V2 of visual cortex) (Lee *et al.*, 2008), and that they become increasingly invariant to factors of variation (such as camera movement) in higher layers (Goodfellow *et al.*, 2009).

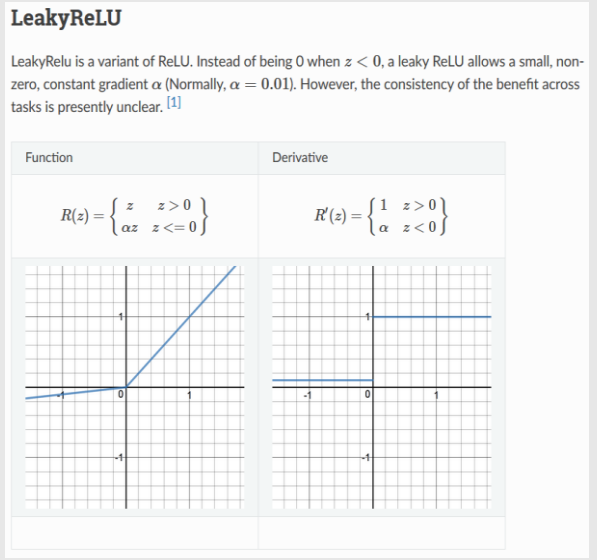
Appearing in Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS) 2011, Fort Lauderdale, FL, USA. Volume 15 of JMLR: W&CP 15. Copyright 2011 by the authors.

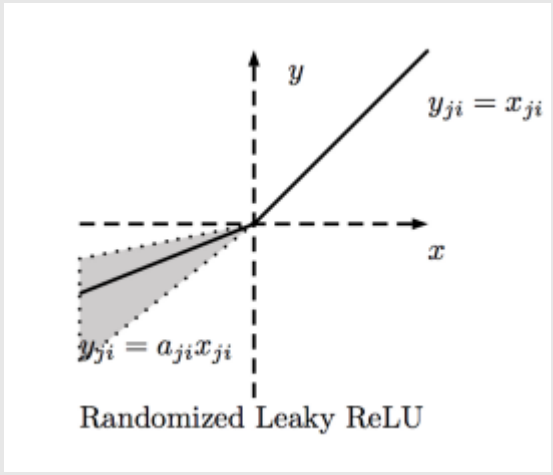
"Deep Sparse Rectifier Neural Networks"
〈Xavier et al. 2011〉

Activation Function: Rectified Linear Units (ReLU)



〈https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html〉





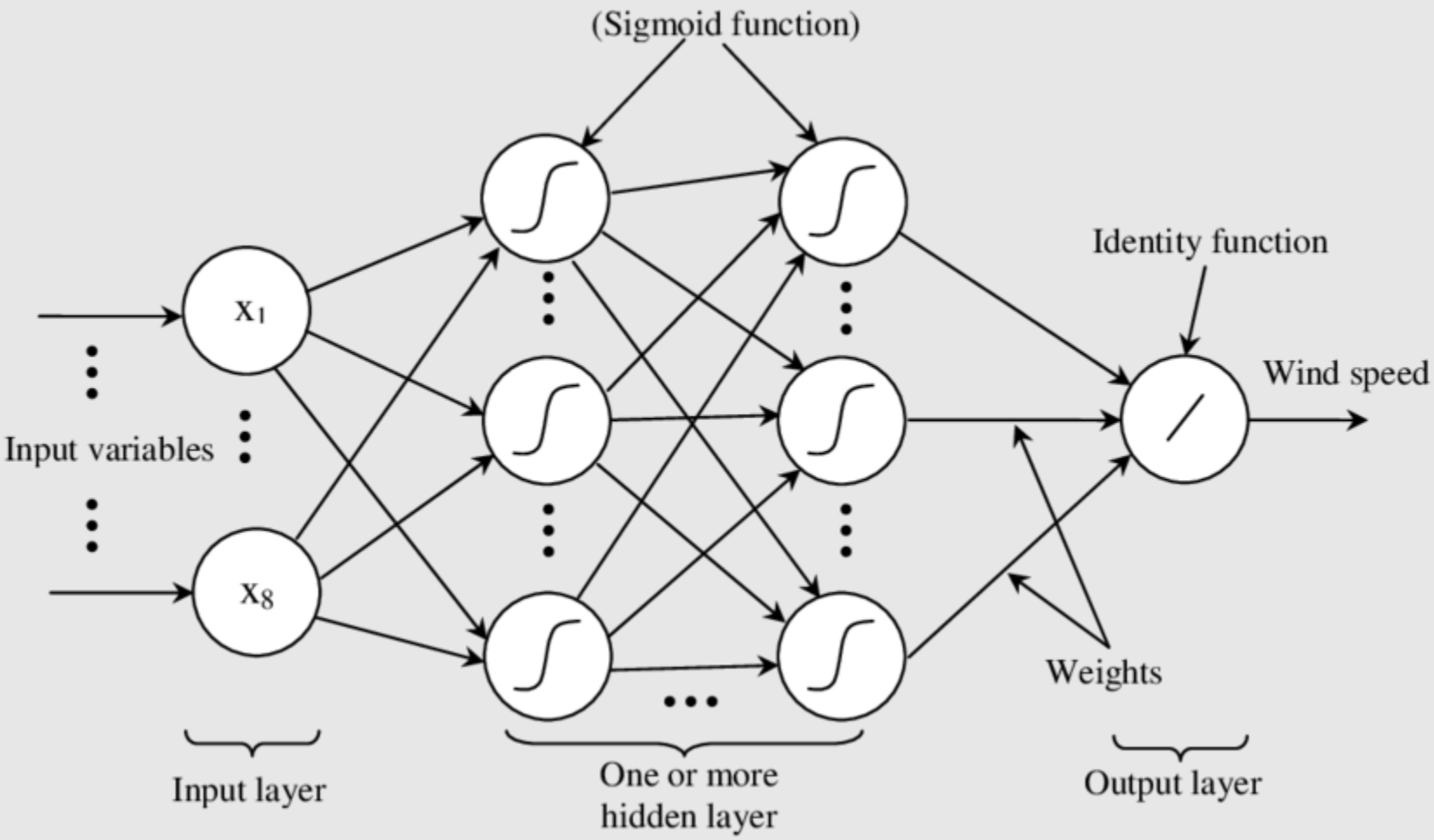
〈https://www.gabormelli.com/RKB/Randomized_Leaky_Rectified_Linear_Activation_Function〉

- Strong alternative to ReLU
- ELU can produce negative outputs
- Output range of [0, inf]

- Attempt to fix the 'dying ReLU'
- As it possess linearity, it can't be used for the complex computation

- Same as 'Leaky ReLU'

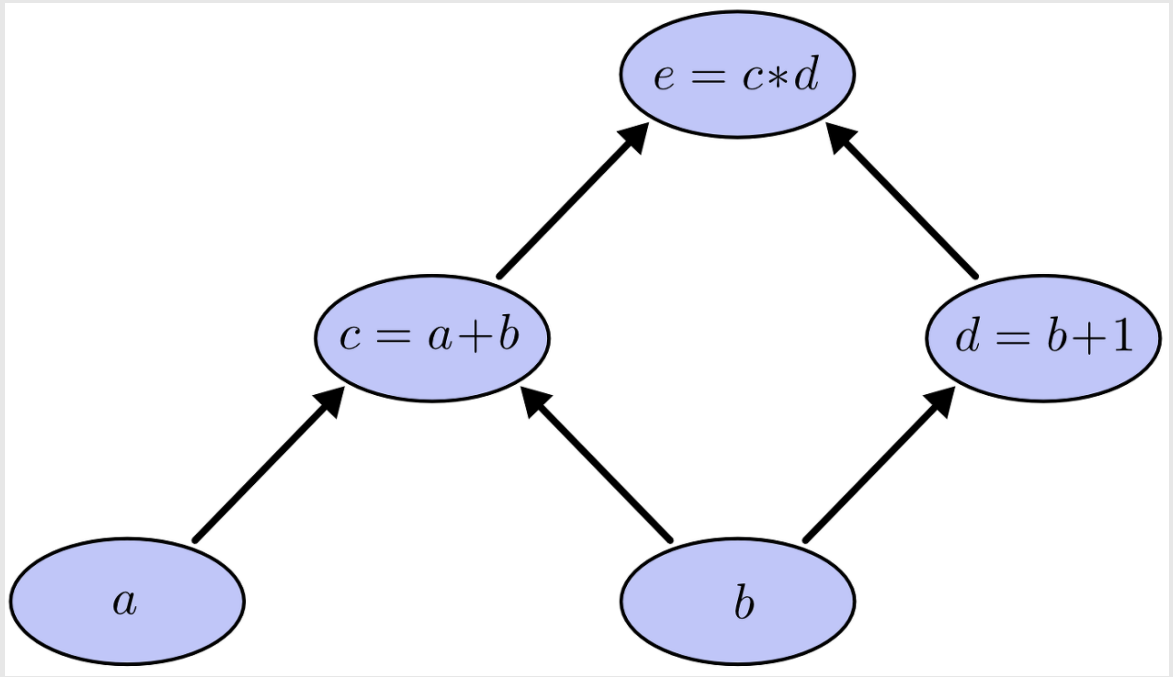
Modern Multi-layer Perceptron



〈https://www.researchgate.net/figure/Architecture-of-a-multilayer-perceptron-neural-network_fig5_316351306〉

Before Implementation

연산 그래프 이해 필요



〈https://brunch.co.kr/@chris-song/22〉

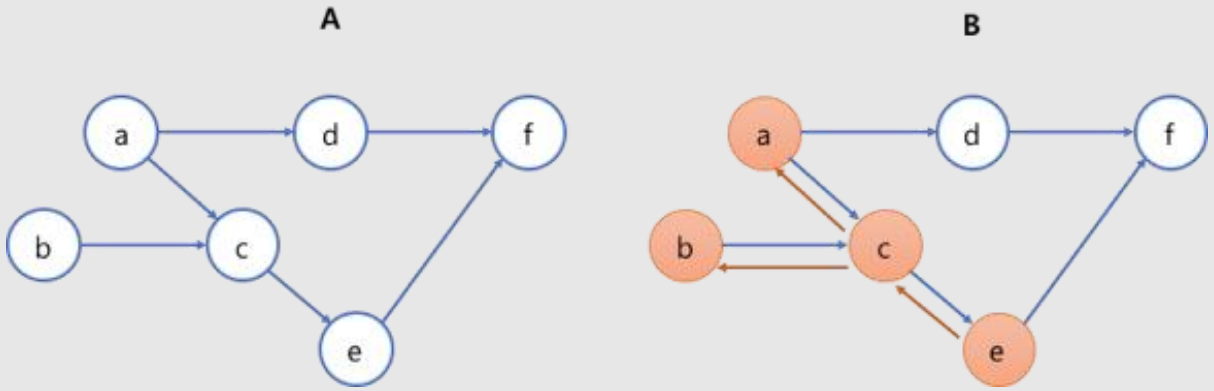
Before Implementation

그래프:

- 노드(Node)나 꼭지점(Vertex)이라고 부르는 서로 연결된 개체(Entity)의 집합을 부르는 용어
  - 노드들은 변을 통해 서로 연결
- 데이터 흐름 그래프에서의 변은 어떤 노드에서 다른 노드로 흘러가는 데이터의 방향 지정

'텐서플로에서 그래프의 각 노드는 하나의 연산을 나타내는데, 입력값을 받을 수 있으며 다른 노드로 전달할 결괏값을 출력할 수 있다. 비유하자면 그래프를 계산한다는 것은 각각의 설비(노드)가 원자재(입력)를 가져오거나 생성하여, 원자재를 가공한 후 다른 설비에 전달하는 과정을 순서대로 수행하여 부품을 만들고, 이러한 부분 생산 과정을 모아 최종 제품을 만들어 내는 것과 같다.'

〈'Learning Tensorflow' by Tom Hope et al.〉



〈http://excelsior-cjh.tistory.com/151〉

Before Implementation

Graph / Session / Fetch

‘텐서플로의 동작은 그래프를 (1)만들고
(2)실행하는 두 단계로 크게 나눌 수 있다.’

〈‘Learning Tensorflow’ by Tom Hope et al.〉

Before Implementation

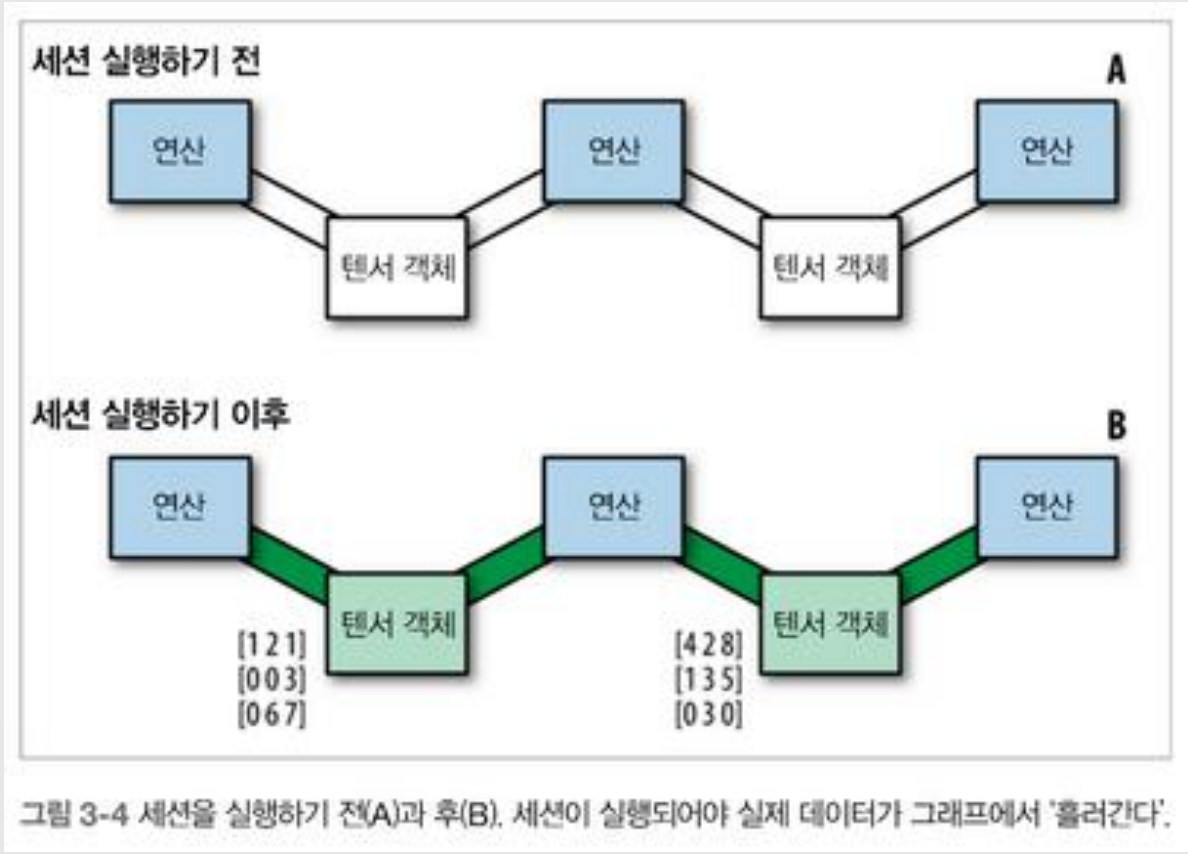


그림 3-4 세션을 실행하기 전(A)과 후(B). 세션이 실행되어야 실제 데이터가 그래프에서 '흘러간다'.

〈'Learning Tensorflow' by Tom Hope et al.〉

Implementation

```
import tensorflow as tf

n_inputs = 28*28  # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
```

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int32, shape=(None), name="y")
```

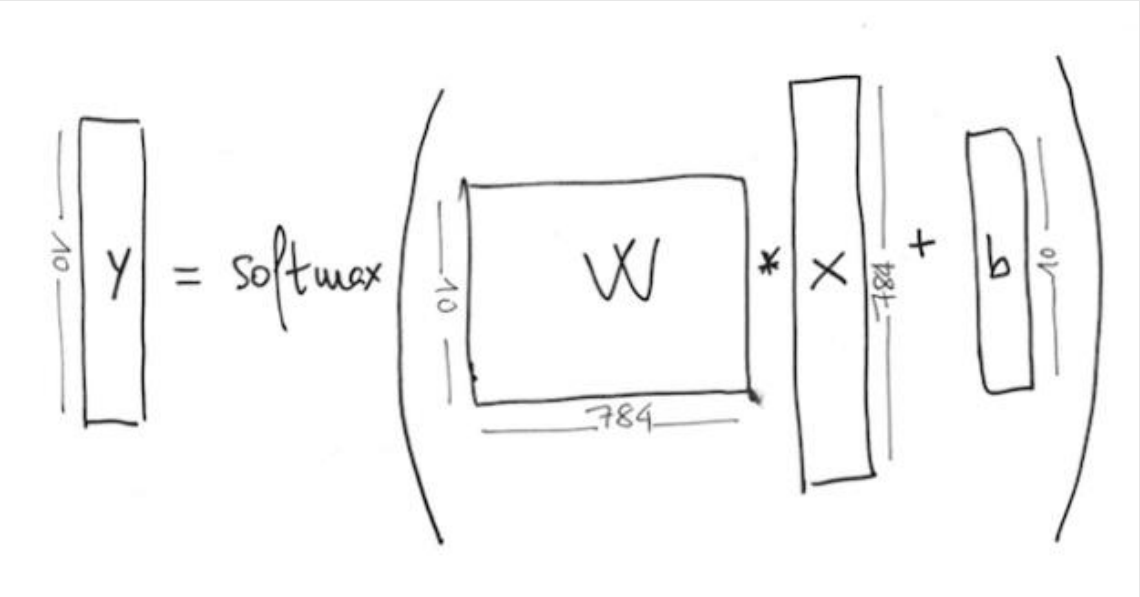
Placeholder? vs Variable?

Placeholder:
선언과 동시에 초기화하는 것이 아닌
일단 선언 후 그 다음 값을 전달
➔ 자리표시자
➔ Input에 주로 사용

Variable:
현재 값(할당한 값)을 출력하는 상태 저장 노드
➔ 변수에 값 할당 필요
➔ 이름대로 변수에 사용

Implementation

```python
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="kernel")
        b = tf.Variable(tf.zeros([n_neurons]), name="bias")
        Z = tf.matmul(X, W) + b
        if activation is not None:
            return activation(Z)
        else:
            return Z
```



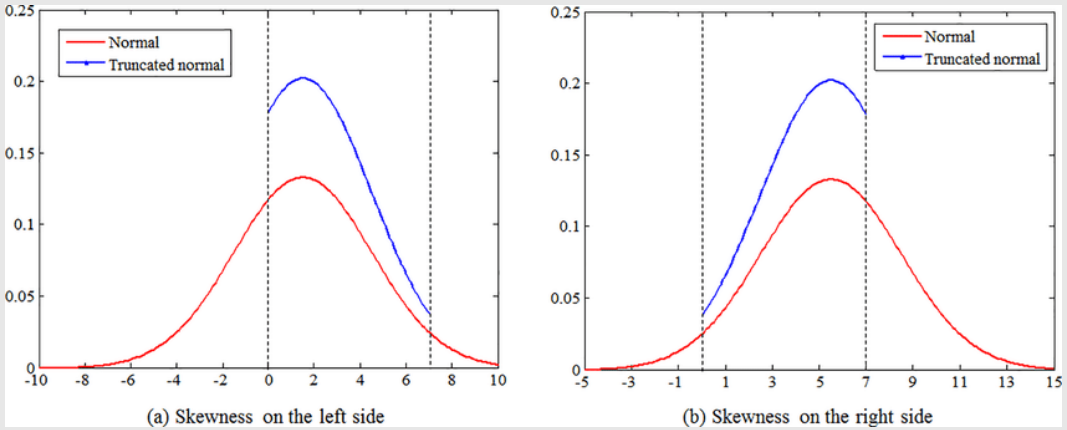
〈https://tensorflow.blog/4-%ed%85%90%ec%84%9c%ed%94%8c%eb%a1%9c%ec%9a%b0-%ec%8b%b1%ea%b8%80-%eb%a0%88%ec%9d%b4%ec%96%b4-%eb%89%b4%eb%9f%b4-%eb%84%a4%ed%8a%b8%ec%9b%8c%ed%81%ac-first-contact-with-tensorflow/〉

Implementation

```python
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="kernel")
        b = tf.Variable(tf.zeros([n_neurons]), name="bias")
        Z = tf.matmul(X, W) + b
        if activation is not None:
            return activation(Z)
        else:
            return Z
```

truncated_normal?

Truncated Normal Distribution:
절단 정규분포
➔ 큰 가중치가 생기지 않아 훈련이 느려지지 않는다.
➔ 가중치가 크면 계산된 출력값이 커져
로지스틱의 양 극단에 가까워지고 이는
gradient vanishing문제로 귀결될 수 있다.



〈https://www.researchgate.net/figure/Truncated-normal-distribution-The-concept-of-truncated-normal-distribution-is-able-to_fig3_318991090〉

Implementation

```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, name="hidden1",
                           activation=tf.nn.relu)
    hidden2 = neuron_layer(hidden1, n_hidden2, name="hidden2",
                           activation=tf.nn.relu)
    logits = neuron_layer(hidden2, n_outputs, name="outputs")
```

=

```
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1",
                              activation=tf.nn.relu)
    hidden2 = tf.layers.dense(hidden1, n_hidden2, name="hidden2",
                              activation=tf.nn.relu)
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
```

with?

With은 파이썬2.5에 도입된 기능으로 context manager에 의해서 실행되는 __enter__()와 __exit—()를 정의하여, with구문 body의 앞부분과 뒷부분에 실행되는 코드를 대신할 수 있다.

```
>>> f = open("x.txt")
>>> f
<open file 'x.txt', mode 'r' at 0x00AE82F0>
>>> f.__enter__()
<open file 'x.txt', mode 'r' at 0x00AE82F0>
>>> f.read(1)
'X'
>>> f.__exit__(None, None, None)
>>> f.read(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

```
with open("x.txt") as f:
    data = f.read()
    do something
```

〈https://cjh5414.github.io/python-with/〉

Implementation

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                              logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

softmax_cross_entropy_with_logits?

softmax_cross_entropy_with_logits:
Cross-entropy의 로짓 계산을 다 해주는 함수.
(물론, Softmax로 계산해줘서 나온 값을
로짓에 넣는 식으로 하나하나 해도 무관)

sparse_softmax_cross_entropy_with_logits:
언더플로 방지를 위한 방법
(단순한 ε이 아닌 $\log(\varepsilon)$로 계산)
➔ 식은 책 P.345참조

softmax_cross_entropy_with_logits_v2:
Adversarial learning같은 특수 상황을 고려하여 나온
새로운 버전으로 기존 버전과 동일
(기존 방식은 Deprecated로 사라질 예정)

Implementation

```
with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

global_variables_initializer?

➔ 변수 값들은 초기화를 꼭 해주어야 함을 잊지 말자.
➔ 안 하면 처음에는 괜찮을지언정
추가적으로 진행될 때 마다 오류 발생

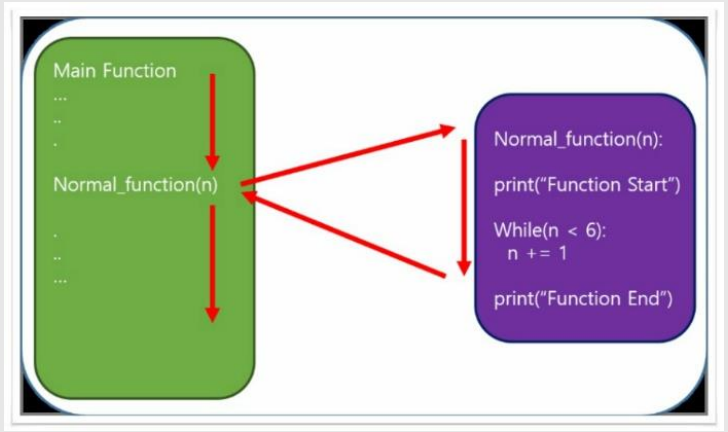
reduce_mean?

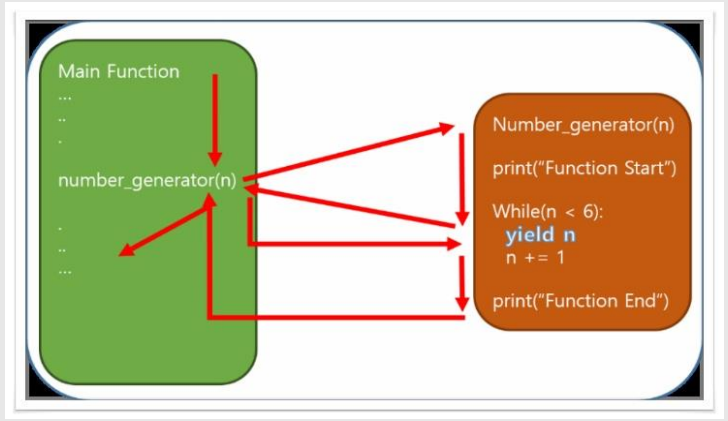
➔ Tensorflow의 모든 자료형은 Tensor
➔ 차원을 줄여서 나온 평균
Ex) [1,2],[4,5]의 reduce_mean은 6

Implementation

```
n_epochs = 40
batch_size = 50

def shuffle_batch(X, y, batch_size):
    rnd_idx = np.random.permutation(len(X))
    n_batches = len(X) // batch_size
    for batch_idx in np.array_split(rnd_idx, n_batches):
        X_batch, y_batch = X[batch_idx], y[batch_idx]
        yield X_batch, y_batch

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for X_batch, y_batch in shuffle_batch(X_train, y_train, batch_size):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_batch = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_val = accuracy.eval(feed_dict={X: X_valid, y: y_valid})
        print(epoch, "Batch accuracy:", acc_batch, "Val accuracy:", acc_val)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

yield?

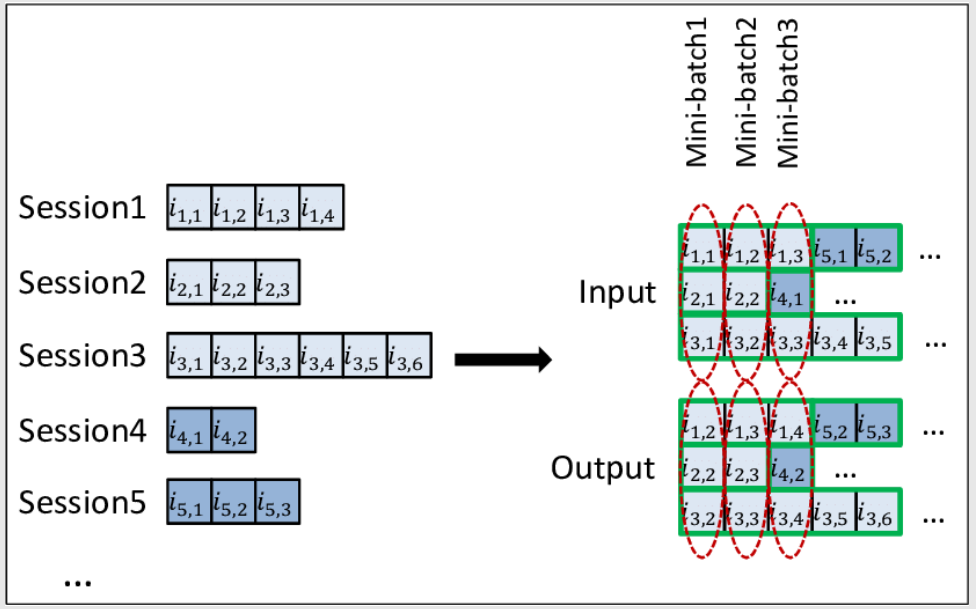


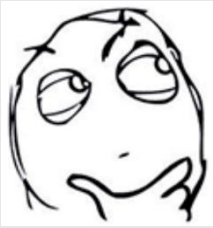


Implementation

```
n_epochs = 40
batch_size = 50

def shuffle_batch(X, y, batch_size):
    rnd_idx = np.random.permutation(len(X))
    n_batches = len(X) // batch_size
    for batch_idx in np.array_split(rnd_idx, n_batches):
        X_batch, y_batch = X[batch_idx], y[batch_idx]
        yield X_batch, y_batch

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for X_batch, y_batch in shuffle_batch(X_train, y_train, batch_size):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_batch = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_val = accuracy.eval(feed_dict={X: X_valid, y: y_valid})
        print(epoch, "Batch accuracy:", acc_batch, "Val accuracy:", acc_val)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

Mini batch?





Do you remember?

Batch Gradient Descent

$$\frac{\partial}{\partial \theta_j} MSE(\theta) = \frac{2}{m} \sum_{i-1}^{m} \left(\theta^T \cdot x^{(i)} - y^{(i)}\right) x_j^{(i)}$$

$$\nabla_\theta MSE(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} MSE(\theta) \\ \frac{\partial}{\partial \theta_1} MSE(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} MSE(\theta) \end{pmatrix} = \frac{2}{m} X^T \cdot (X \cdot \theta - y)$$

- 매 스텝마다 훈련 데이터 전체를 사용한 위의 식 계산 ➔ 매우 많은 연산
- 하지만 특성 수에 영향을 받지 않으므로 때에 따라서 정규방정식보다 빠르기도 하다.

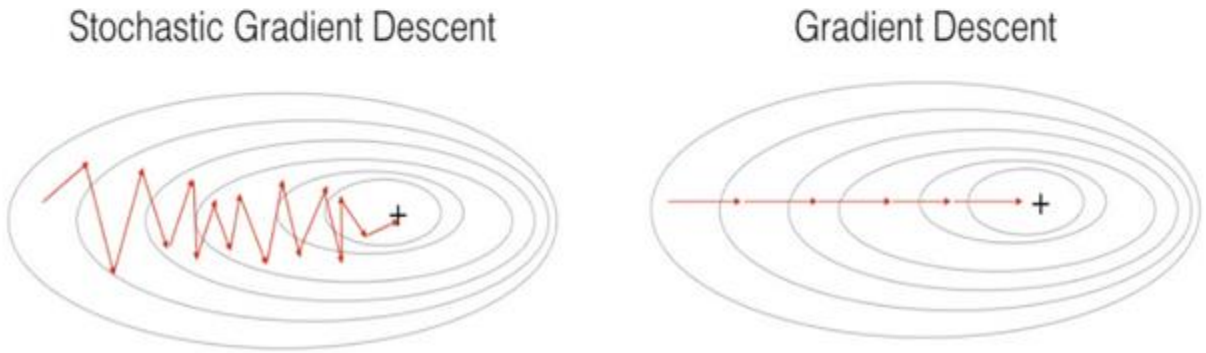
Stochastic Gradient Descent

SGD (Stochastic Gradient Descent)이란?

배치 크기가 1인 경사하강법 알고리즘 입니다.

즉, 확률적 경사하강법은 데이터 세트에서 무작위로 균일하게 선택한 하나의 예를 의존하여 각 단계의 예측 경사를 계산합니다.

< 최소값을 찾는 과정 >



〈https://engmrk.com/mini-batch-gd/〉

Stochastic Gradient Descent

Pros

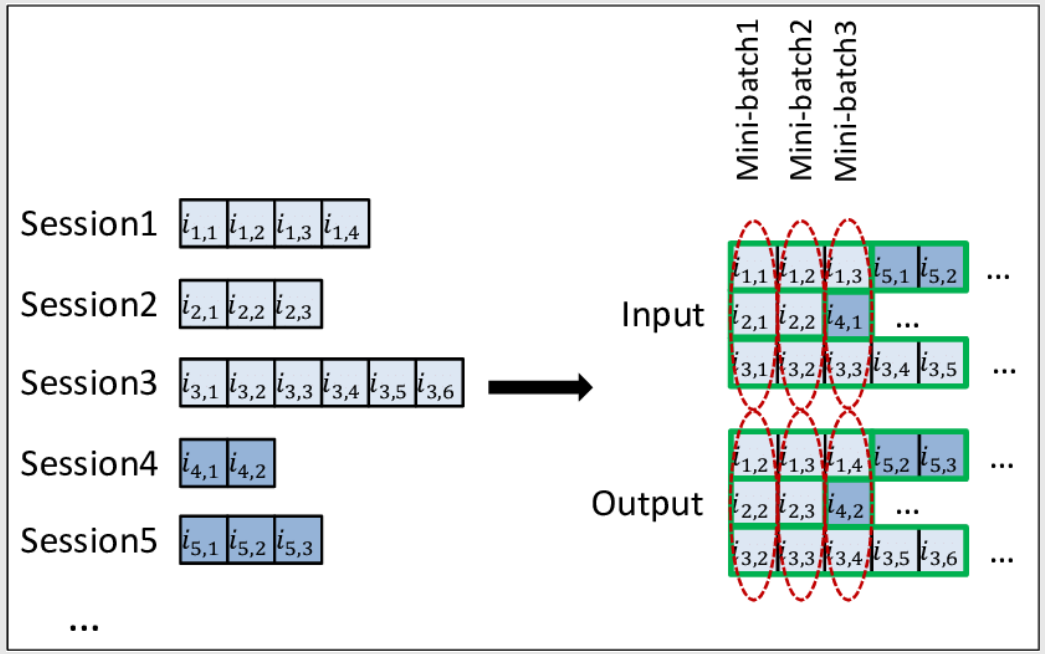
- 적은 메모리 사용
- 무작위성이 추가되어 다양성 증가
- Local Minima 탈출이 용이함

Cons

- 노이즈에 취약
- 무작위성이 추가되어 불안정성 증가
- Global Minima의 접근이 힘듬

머신러닝 분야의 Trend 중 하나
➔ 장단점이 있는 알고리즘은 절충안이 꼭 나옴

Mini-batch Gradient Descent

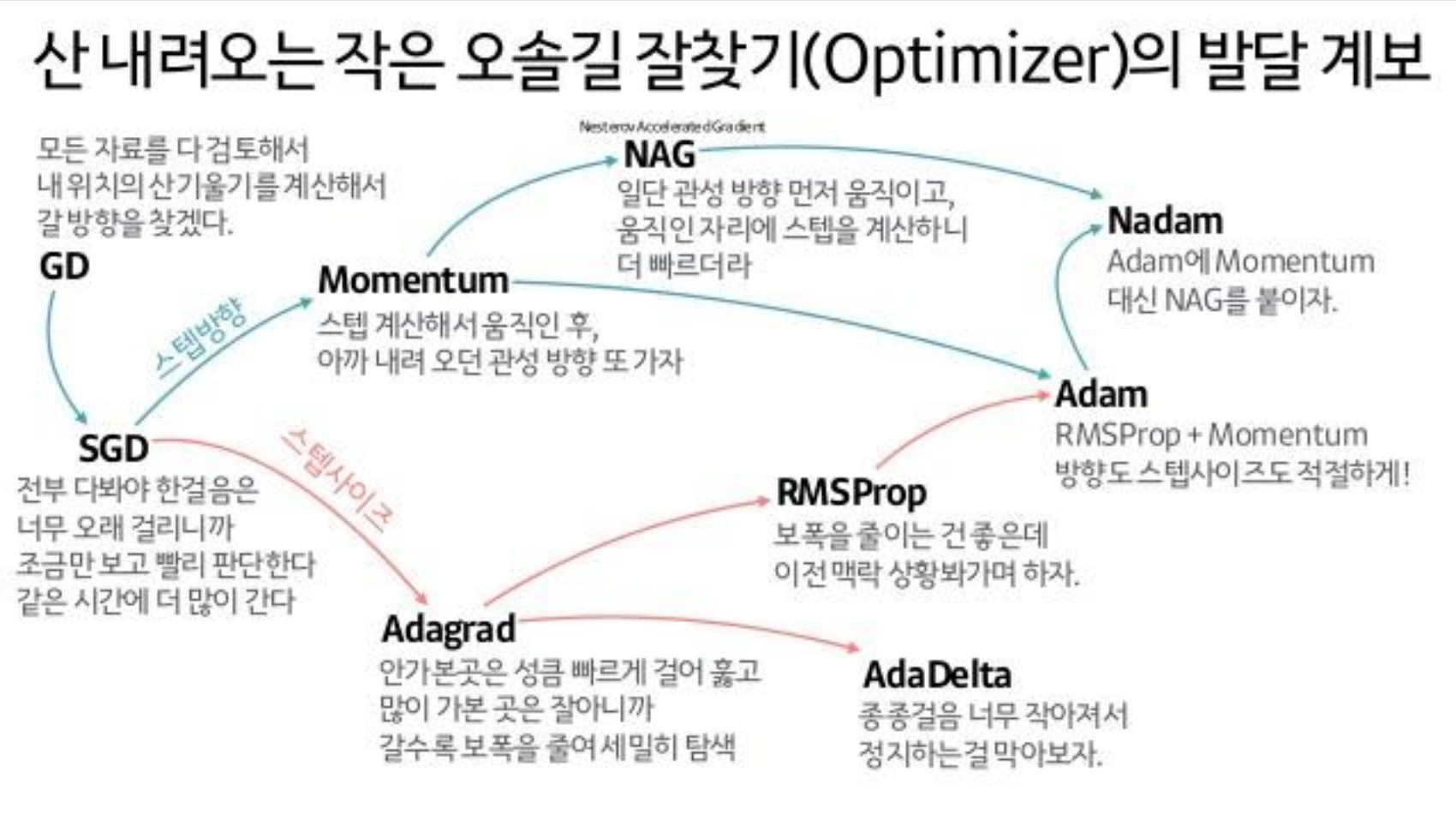


〈https://yq.aliyun.com/articles/236684〉

결국 데이터를 한개 쓰면 빠르지만 너무 헤매고, 전체를 쓰면 정확하지만 너무 느립니다. 즉 적당히 빠르고 적당히 정확한 길을 찾기 위해 **mini-batch를 사용합니다.** 적게는 수십개부터 많게는 수백개의 데이터를 한 그룹으로하여 처리함으로써 iteration 한번 수행하는데 소요되는 시간을 최대한 줄이면서 전체 데이터를 최대한 반영합니다. 동시에 보통은 가능한 한도 내에서 batch 크기를 최대한 크게 잡아 하드웨어에 부담을 주지 않는 선에서 하드웨어를 최대한 활용합니다. 이와 같은 이유로 거의 당연하게 mini-batch를 사용하게 됩니다.

〈http://dambaekday.tistory.com/1〉

Optimizer



〈https://www.slideshare.net/yongho/ss-79607172〉

Easy Implementation

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

import tensorflow as tf

x = tf.placeholder("float", [None, 784])
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))

matm=tf.matmul(x,W)
y = tf.nn.softmax(tf.matmul(x,W) + b)
y_ = tf.placeholder("float", [None,10])

cross_entropy = -tf.reduce_sum(y_*tf.log(y))
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)

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

for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels}
```

Universal Approximation Theorem

뉴런 수만 무한하다면 은닉층 하나로 어떤 함수
도 근사할 수 있다.

➔ Universal Approximation Theorem

1989년 시벤코(Cybenko)가 발표한 **시벤코 정리**(Cybenko's theorem)는 다음과 같다.

φ를 시그모이드 형식의 연속 함수라 하자(예, $\varphi(\xi) = 1/(1+e^{-\xi})$). $[0,1]^n$ 또는 R^n의 부분집합에서 실수의 연속 함수 f와 $\epsilon > 0$가 주어지면, 다음을 만족하는 벡터 $\mathbf{w_1}, \mathbf{w_2}, \ldots, \mathbf{w_N}, \alpha, \theta$와 매개 함수 $G(\cdot, \mathbf{w}, \alpha, \theta) : [0,1]^n \to R$이 존재한다.

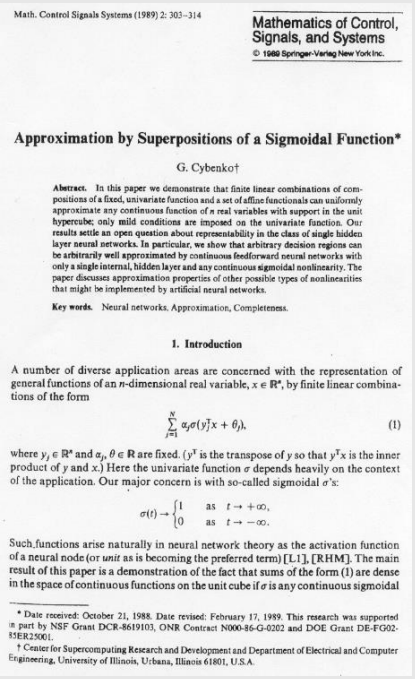
$$|G(\mathbf{x}, \mathbf{w}, \alpha, \theta) - f(x)| < |\epsilon| \text{ for all } \mathbf{x} \in [0,1]^n$$

이때,

$$G(\mathbf{x}, \mathbf{w}, \alpha, \theta) = \sum_{i=1}^{N} \alpha_j \varphi(\mathbf{w}_j^T \mathbf{x} + \theta_j)$$

이고, $\mathbf{w}_j \in R^n, \alpha_j, \theta_j \in R, \mathbf{w} = (\mathbf{w}_1, \mathbf{w}_2, \ldots \mathbf{w}_N), \alpha = (\alpha_1, \alpha_2, \ldots, \alpha_N), \theta = (\theta_1, \theta_2, \ldots, \theta_N)$이다.

〈https://ko.wikipedia.org/wiki/%EC%8B%9C%EB%B2%A4%EC%BD%94_%EC%A0%95%EB%A6%AC〉



Math. Control Signals Systems (1989) 2: 303–314

Mathematics of Control, Signals, and Systems

© 1988 Springer-Verlag New York Inc.

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse application areas are concerned with the representation of general functions of an n-dimensional real variable, $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j), \quad (1)$$

where $y_j \in \mathbb{R}^n$ and $\alpha_j, \theta \in \mathbb{R}$ are fixed. (y^T is the transpose of y so that $y^T x$ is the inner product of y and x.) Here the univariate function σ depends heavily on the context of the application. Our major concern is with so-called sigmoidal σ's:

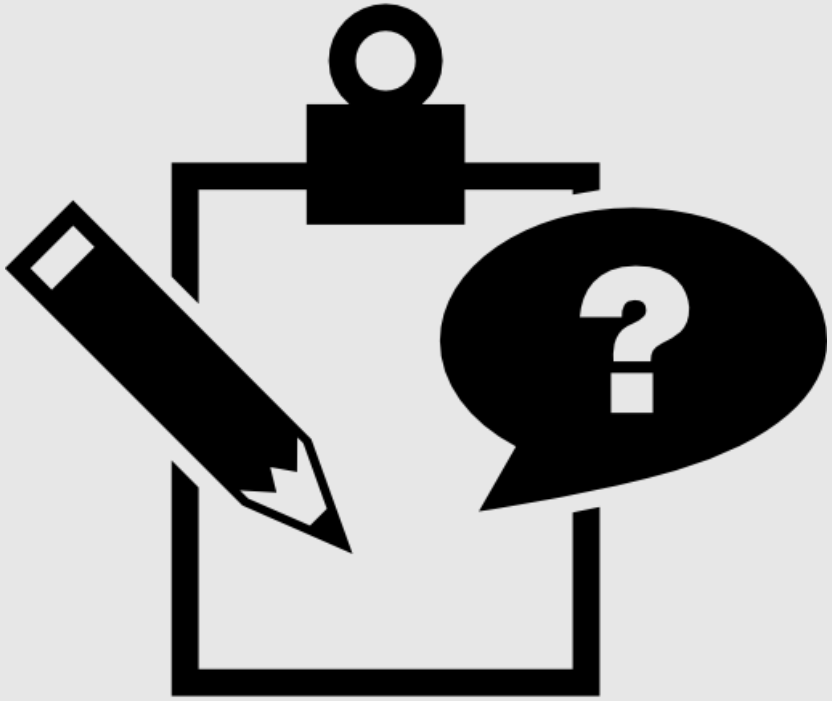
$$\sigma(t) \to \begin{cases} 1 & \text{as } t \to +\infty, \\ 0 & \text{as } t \to -\infty. \end{cases}$$

Such functions arise naturally in neural network theory as the activation function of a neural node (or *unit* as is becoming the preferred term) [L1], [RHM]. The main result of this paper is a demonstration of the fact that sums of the form (1) are dense in the space of continuous functions on the unit cube if σ is any continuous sigmoidal

* Date received: October 21, 1988. Date revised: February 17, 1989. This research was supported in part by NSF Grant DCR-8619103, ONR Contract N000-86-G-0202 and DOE Grant DE-FG02-85ER25001.

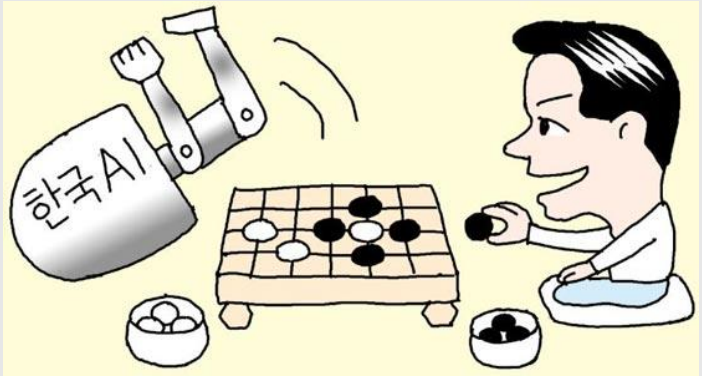
† Center for Supercomputing Research and Development and Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois 61801, U.S.A.

Implementation



Before Finish Class

Og-Go



이날 가장 큰 관심거리는 한국 랭킹 2위(현재 1위) 신진서 9단과 국내 대표 인터넷 기업 카카오가 자체 개발한 인공지능(AI) '오지고(Og-Go)'의 대국이었습니다. 지난해 카카오 AI 연구 자회사인 카카오브레인은 한국기원의 대국 데이터를 기반으로 AI를 개발한다고 발표했고 이날은 오지고의 실력이 공개되는 날이었습니다. 오지고는 국내 IT(정보기술) 대기업이 개발한 첫 바둑 AI입니다.

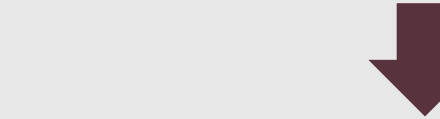
그러나 대국은 싱겁게 끝났습니다. 오지고가 바둑 축(逐)을 파악하지 못해 신 9단이 단 83수(手) 만에 불계승을 거둔 것입니다. 축은 계속 돌을 두어도 어차피 상대방의 돌에 잡힐 수밖에 없는 모양을 말합니다. '축 모르고 바둑 두지 말라'는 격언이 있을 정도로 기본적인 수읽기에 해당합니다.

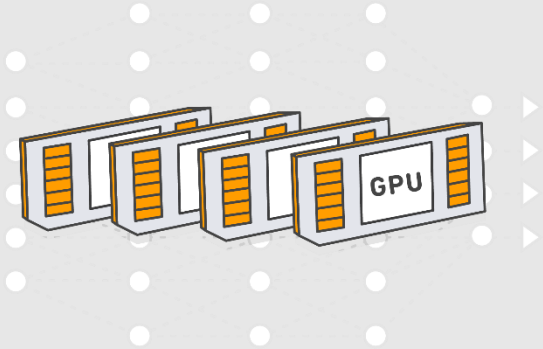
이번 대국을 앞두고 신 9단이 AI를 이기기 어려울 것이라는 예상이 많았습니다. 2016년과 2017년 구글 딥마인드 AI '알파고'가 이세돌·커제 9단을 차례로 꺾었고, 일본 '딥젠고', 중국 '줴이'도 프로바둑 기사보다 앞선 기력을 선보였기 때문입니다. 오지고도 이날 초반엔 매끄러운 수순으로 국면을 이끌었다고 합니다. 그러다 어이없는 오류로 83수 만에 패하고 말았습니다.

<https://news.v.daum.net/v/20181106031045814>

AlphaGo

AlphaGo

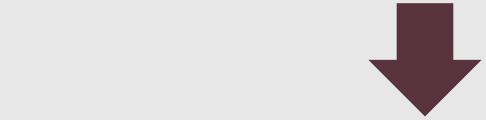


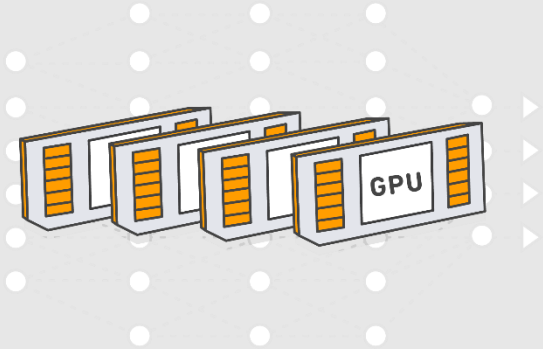


X ?

AlphaGo

AlphaGo





X 176

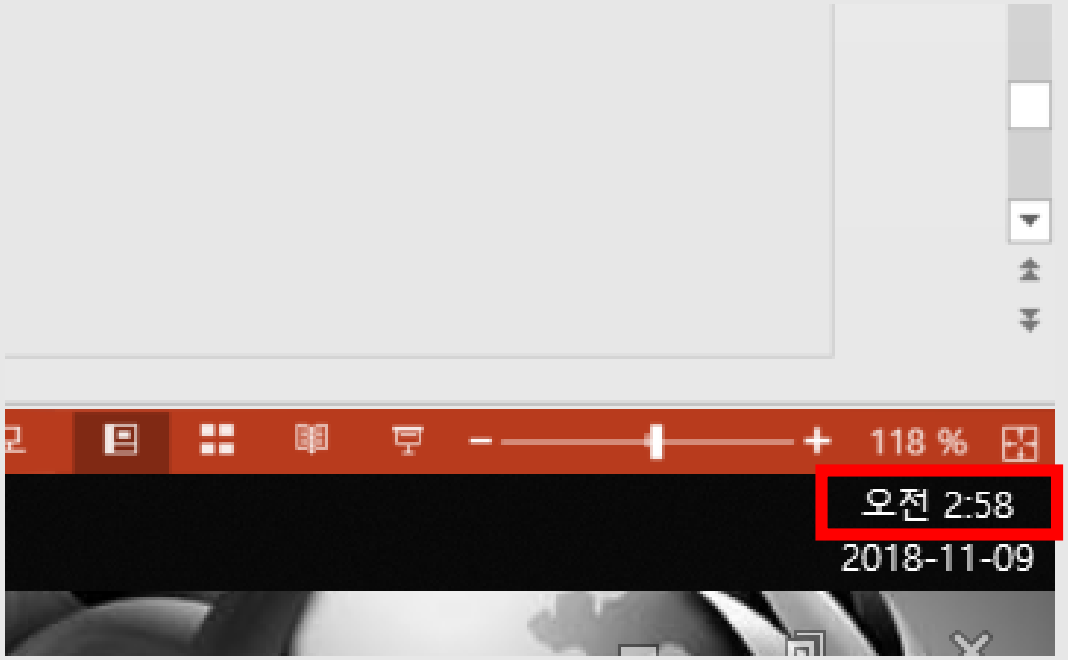
Next Plan

11월 15일 ➔ 11월 23일
11장(심층 신경망 훈련) & 복습(복습해오세요)

11월 16일
학술제 파이팅!

11월 17일
지스타(G-Star)

Give Me Coffee



커피라도 사주세요


Ideas worth spreading
– TED Talks

고생하셨습니다.