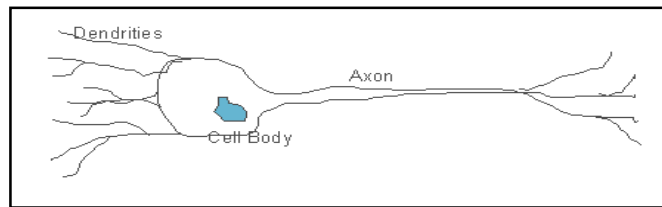


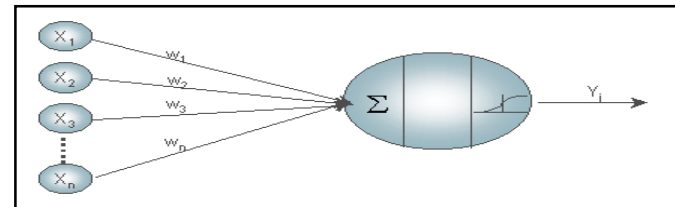
# Neural Networks

# 신경망(Neural Networks)의 개요

- 머신러닝 알고리즘 중 가장 많이 알려진 것이 신경망 분석이며, 보통 머신러닝에서 "신경망 분석 = 패턴을 찾아내는 것"이라고 연상할 만큼 잘 알려진 분석
- 인간 두뇌의 신경망(860억 개의 뉴런과 5000조 개의 시냅스로 구성)을 흉내 내어 데이터로부터의 반복적인 학습 과정을 거쳐 데이터에 숨어 있는 패턴을 찾아내는 모델링 기법



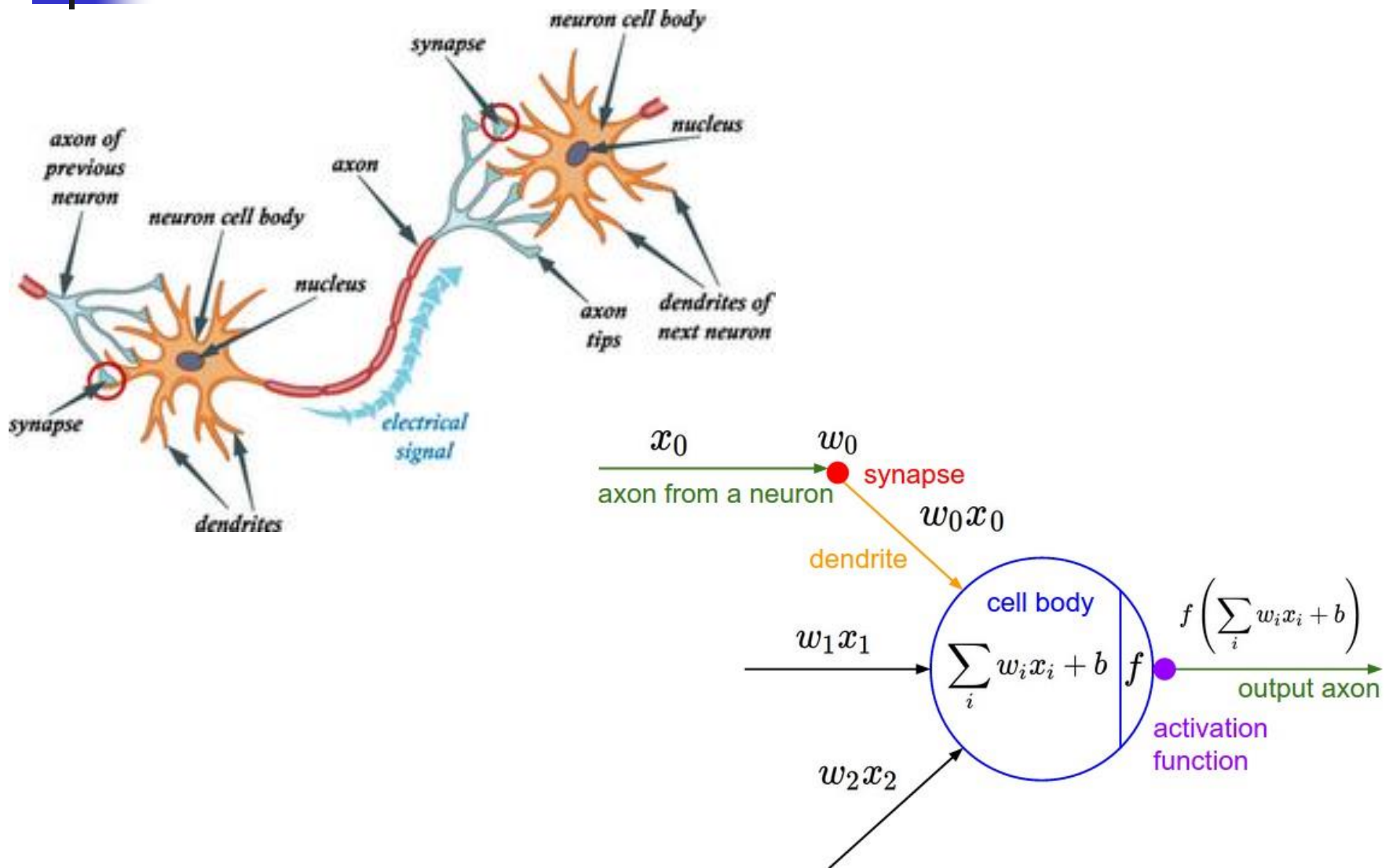
신경세포(neuron)



신경망(neural networks)

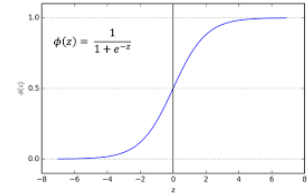
- 계층 구조를 갖는 수많은 프로세싱 요소로 이루어진 수학적 모형
  - 신경망 이론의 다양한 아키텍처를 기반으로 데이터로부터 패턴을 학습하여 최적해를 도출
- 장단점
  - 비선형 자료, 범주/연속형 혼합 자료 처리가 탁월하고 통계적 가정이 불필요
  - 설명변수들이 목표변수에 구체적으로 어떠한 영향을 주는지 해석하기 어렵고, Over-Fitting 가능성 높음

# Biological vs. Artificial neural network



# How neural networks work

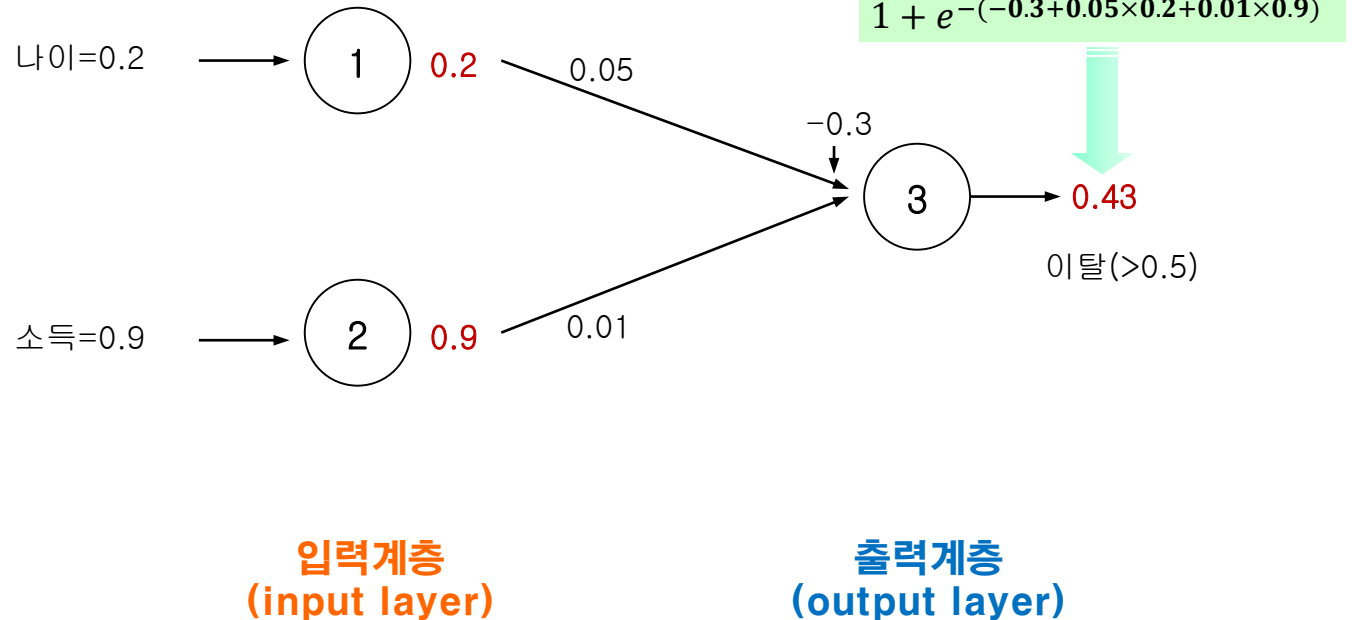
- Single layer neural network



나이	소득	이탈
0.2	0.9	1
0.3	0.5	0
0.7	0.9	1

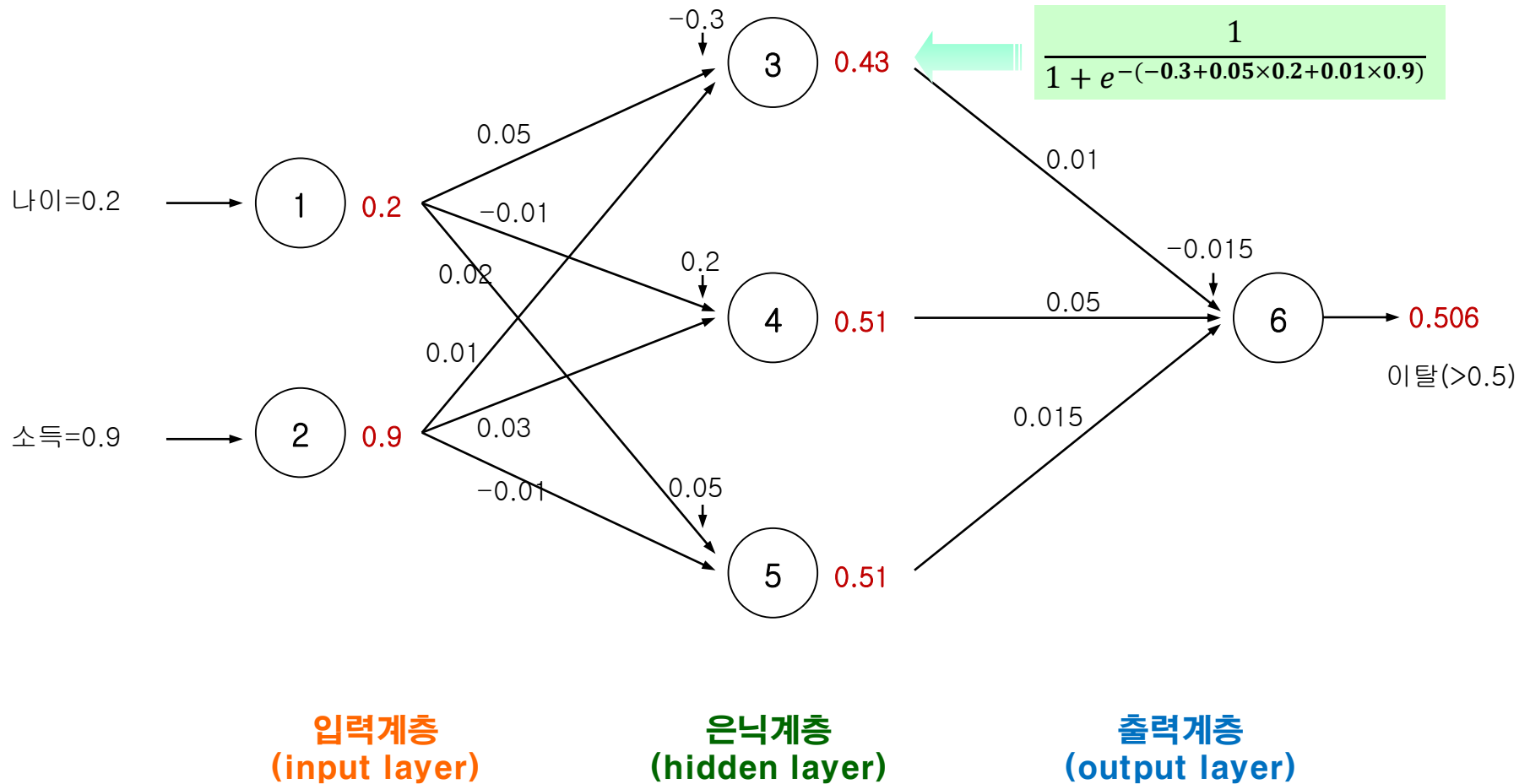
... ..

0.6	0.2	0
-----	-----	---



# How neural networks work

- Multi-layer neural network



# Mathematical notation

- Single layer neural network

- Neuron pre-activation (or input activation):

$$a(\mathbf{x}) = b + \sum_i w_i x_i = b + \mathbf{w}^T \mathbf{x}$$

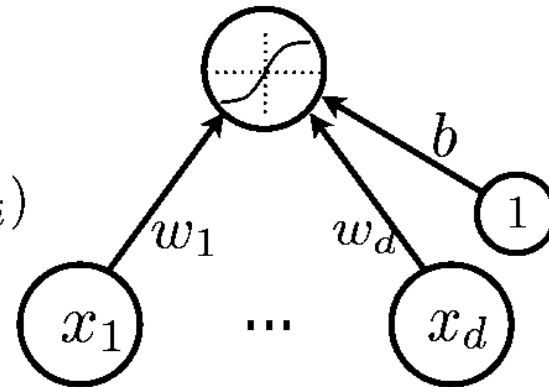
- Neuron (output) activation

$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$

$\mathbf{w}$  are the connection weights

$b$  is the neuron bias

$g(\cdot)$  is called the activation function



# Mathematical notation

- Multi-layer neural network

- Could have  $L$  hidden layers:

- layer pre-activation for  $k > 0$  ( $\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x}$ )

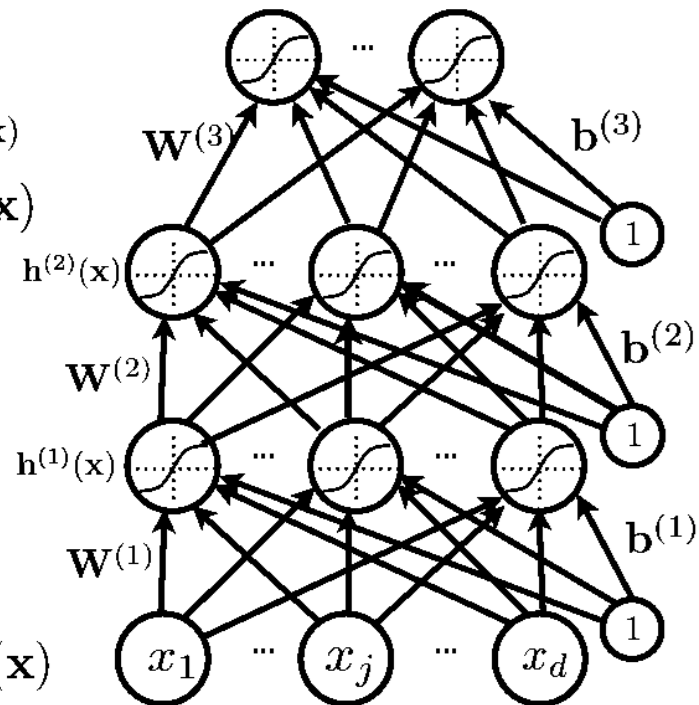
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})$$

- hidden layer activation ( $k$  from 1 to  $L$ ):

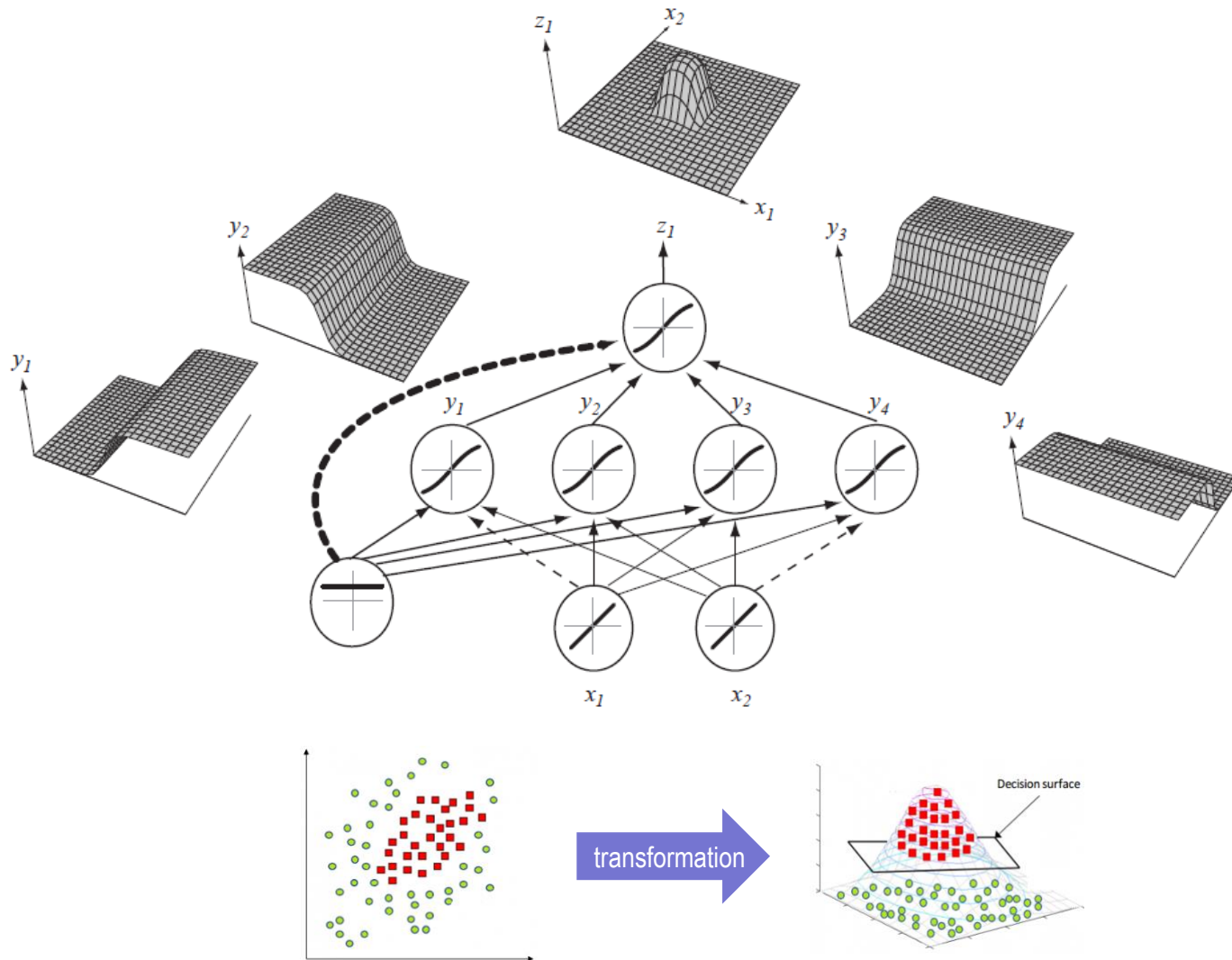
$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

- output layer activation ( $k=L+1$ ):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



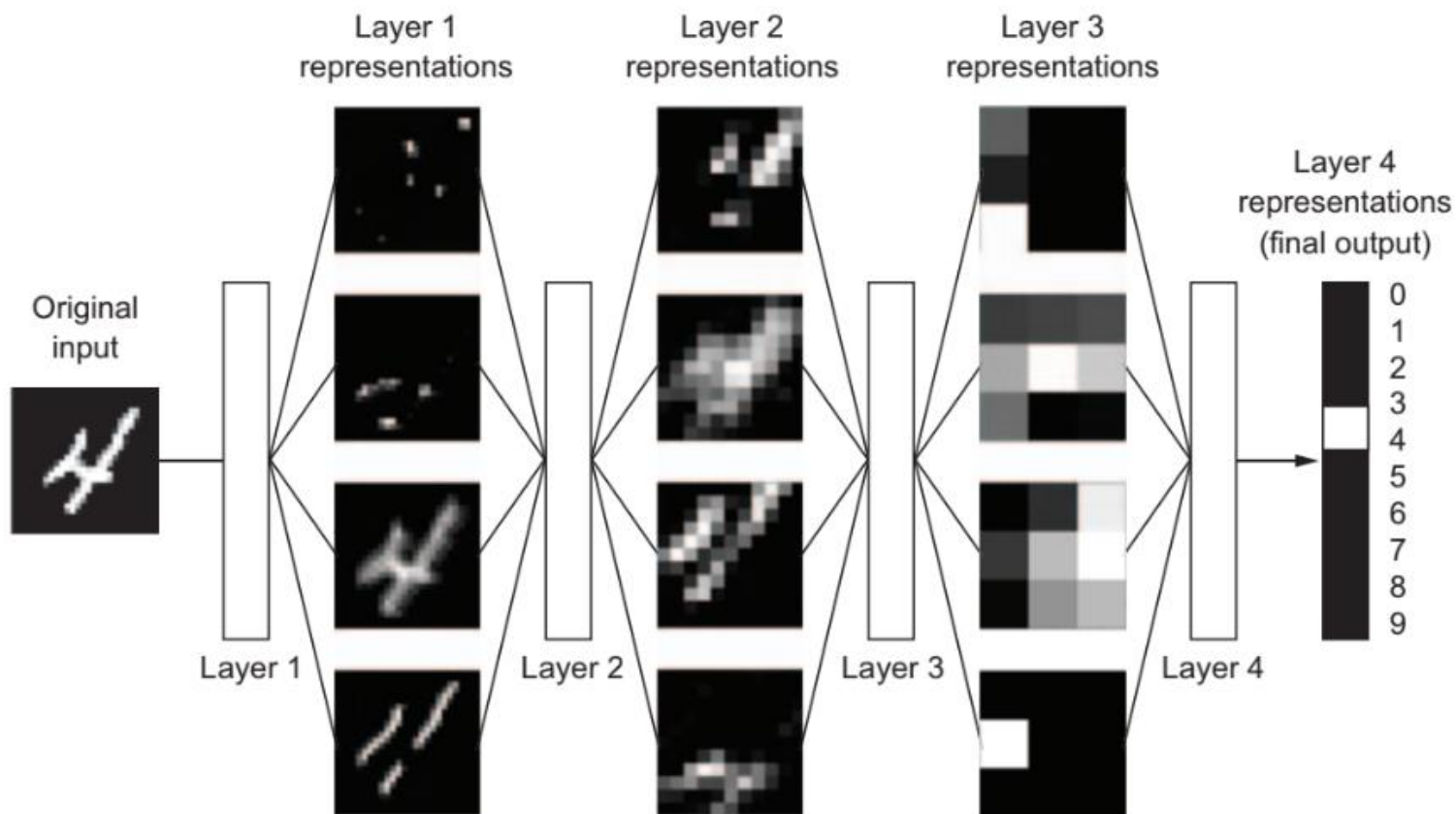
# Capacity of neural network



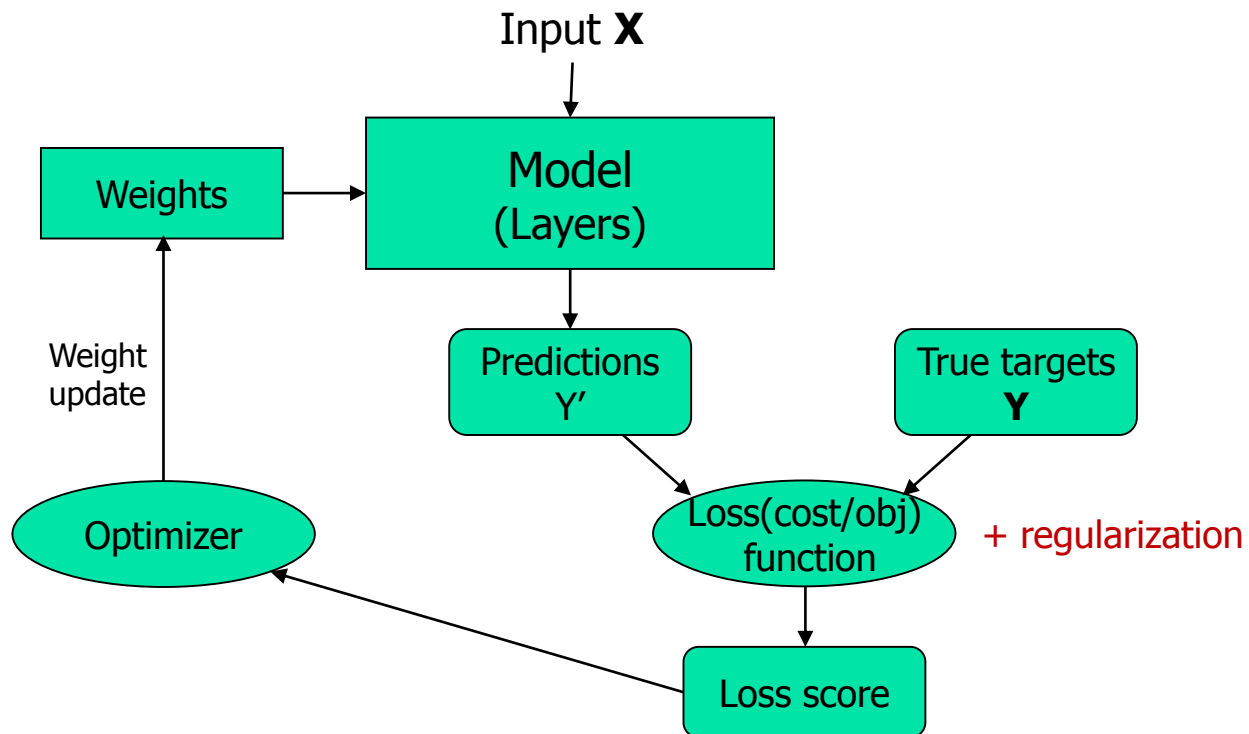


# Capacity of neural network

- Neural networks do input-to-target mapping via a deep sequence of simple data transformations

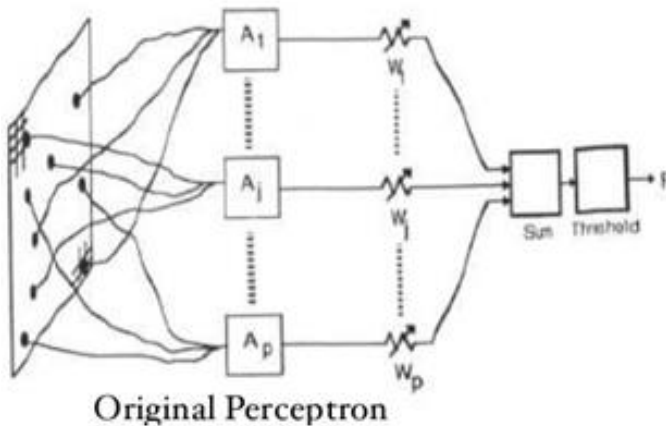


# Training loop in NN

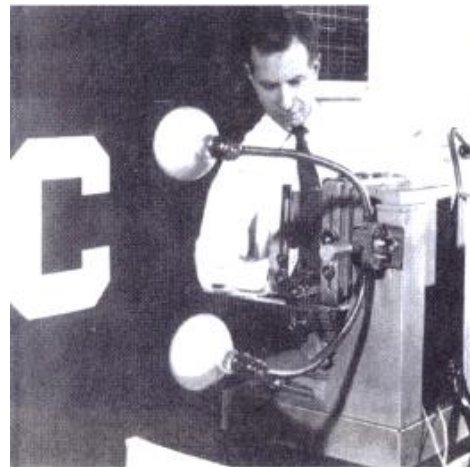


# Perceptron: the simplest NN architecture

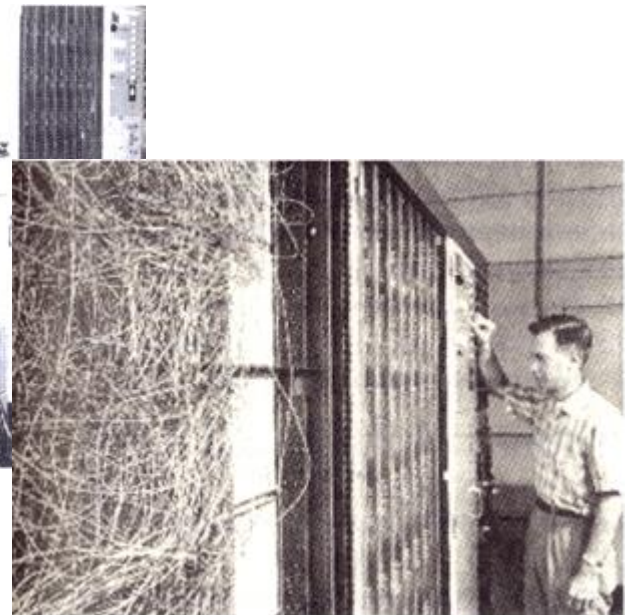
- 1943년, McCulloch와 Pitts는 인간의 두뇌를 수많은 신경세포들로 구성된 컴퓨터라 생각하고 최초로 신경망의 모델을 제안
- 1951년, Edmonds와 Minsky는 학습 기능을 갖는 최초의 신경망을 구축
- 1957년, Frank Rosenblatt는 Perceptron이라는 신경망모델을 제안하였는데, 이것은 패턴을 인식하기 위하여 학습 기능을 이용



Source: <https://www.slideshare.net/roelofp/220115dlmeetup>



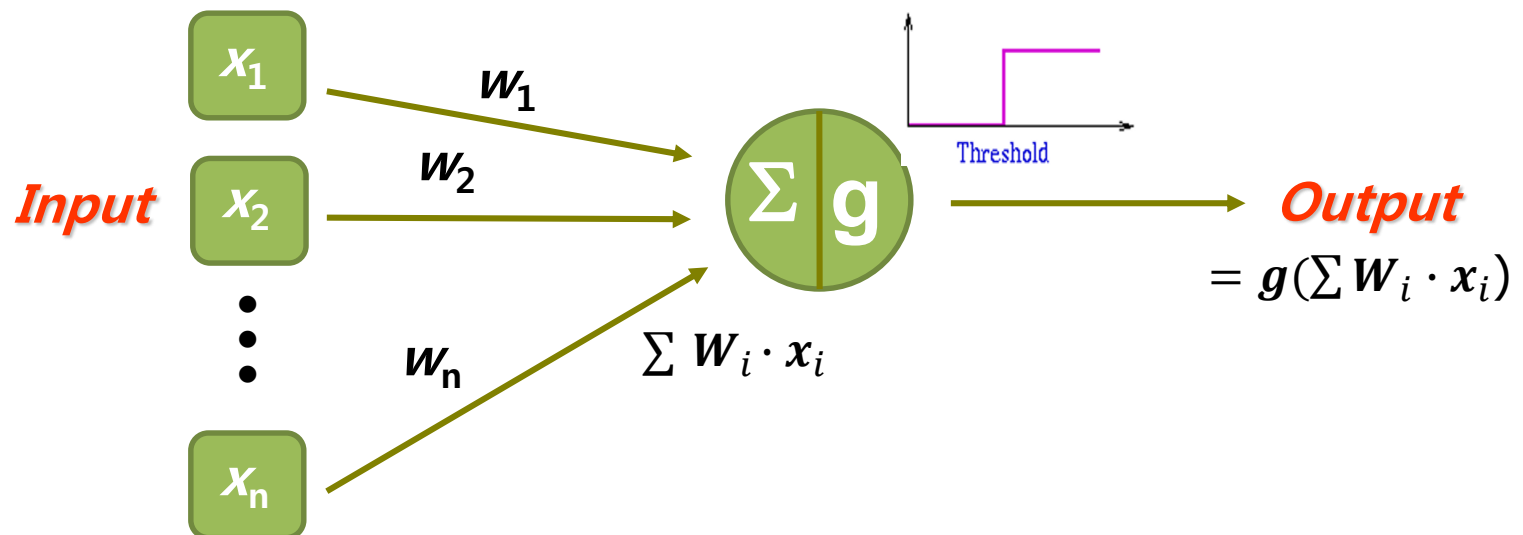
400( $20 \times 20$ ) 개의 pixel 인식용  
"Mark 1 Perceptron" machine



Source: [http://www.aistudy.com/neural/model\\_kim.htm](http://www.aistudy.com/neural/model_kim.htm)

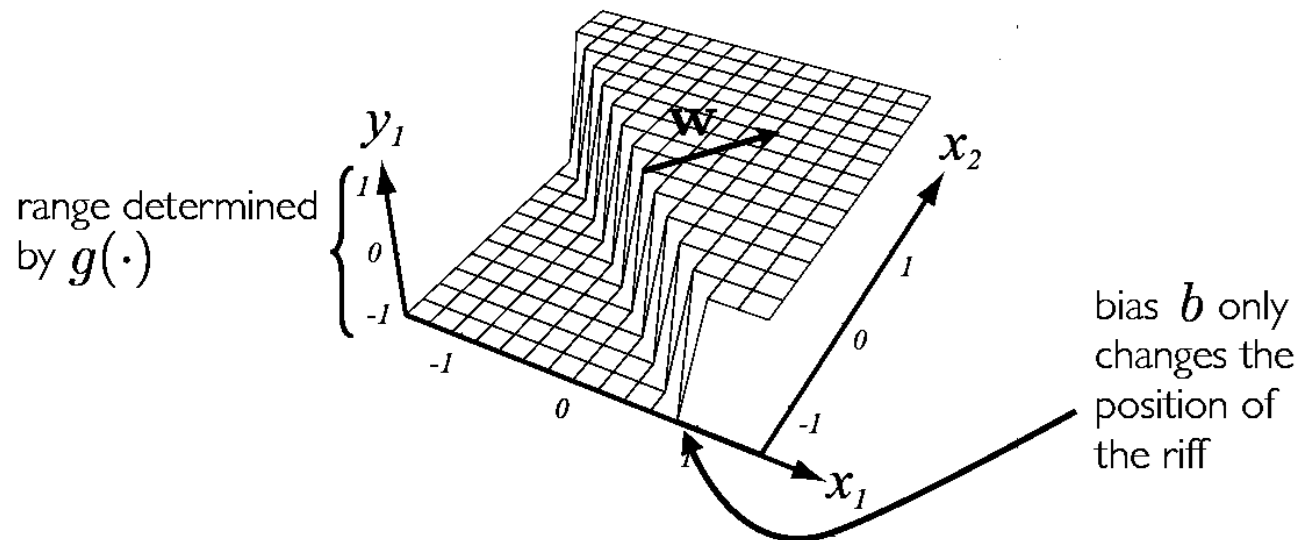
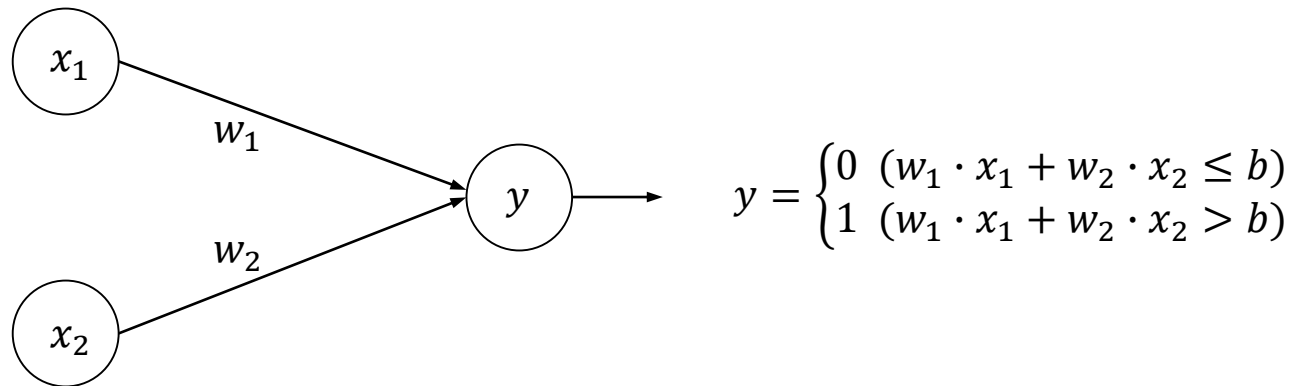
# Perceptron

- 가중치(weight)
  - 입력신호의 강도를 표현:  $W_1, W_2, \dots, W_n$
- 입력신호의 총합(total net input)
  - 각 입력신호에 해당 가중치를 곱하여 합한 값
  - $W_1 \cdot x_1 + W_2 \cdot x_2 + \dots + W_n \cdot x_n = \sum W_i \cdot x_i$
- 활성화 함수(activation/transfer function)
  - 입력신호의 총합이 활성화되는지(출력 값)를 결정하는 함수
  - 임계 값  $\theta$ 를 갖는 Step Function 사용



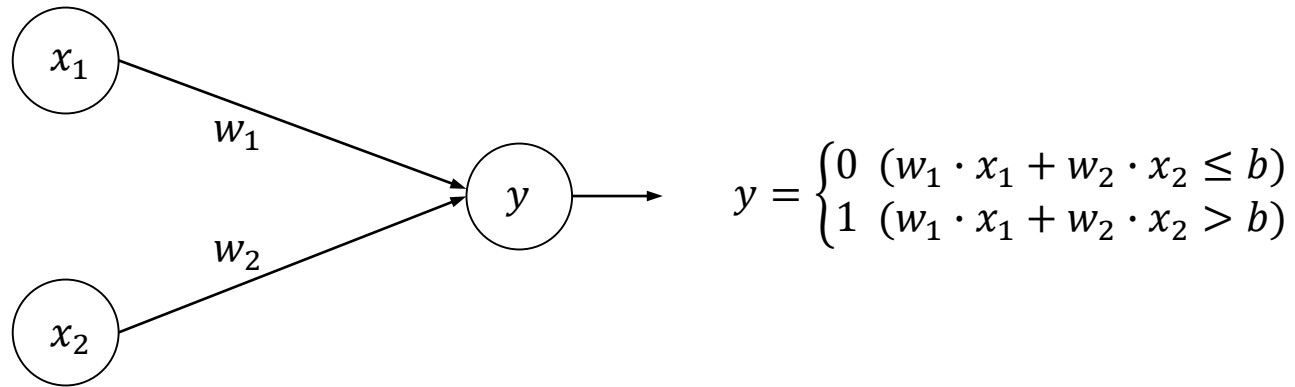
# Perceptron

- How a perceptron works

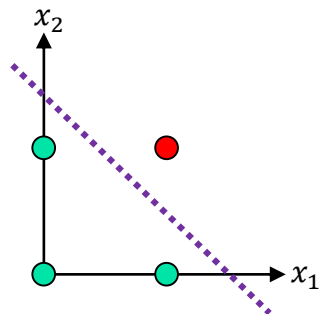


# Perceptron

- Limitation of (single layer) perceptron

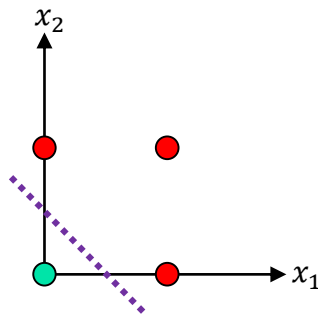


AND



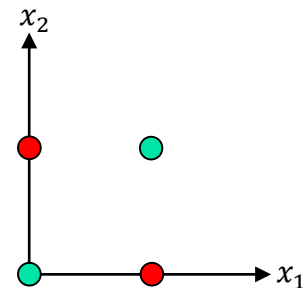
$$\begin{aligned} w_1 &= 1.0 \\ w_2 &= 1.0 \\ b &= 1.5 \end{aligned}$$

OR



$$\begin{aligned} w_1 &= 1.0 \\ w_2 &= 1.0 \\ b &= 0.5 \end{aligned}$$

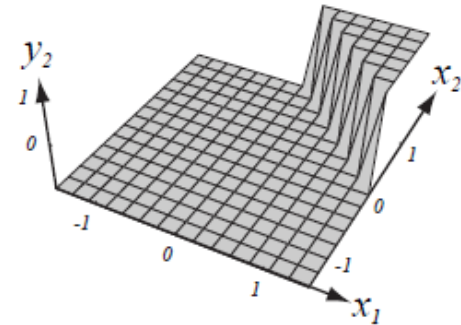
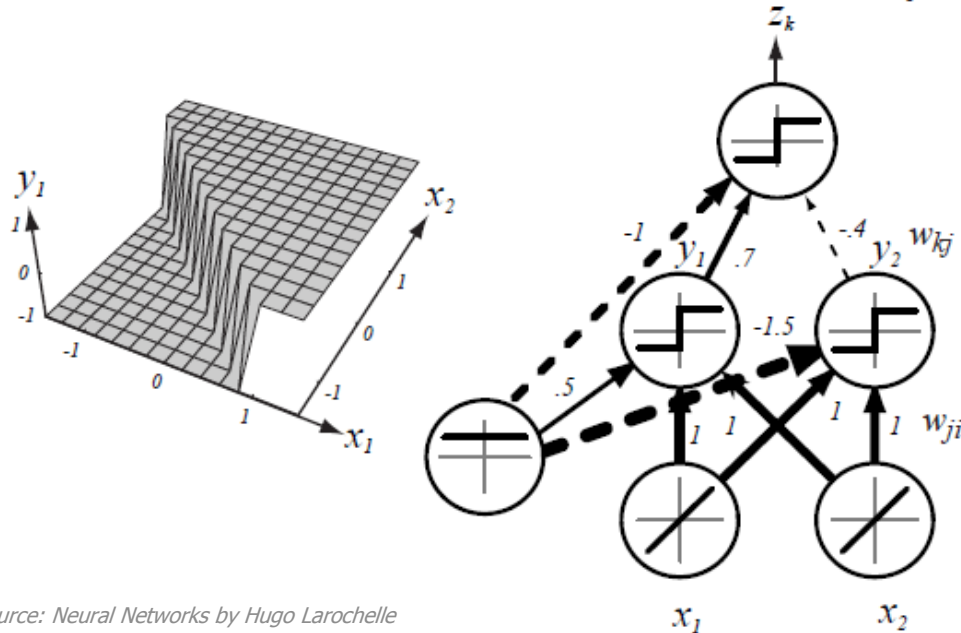
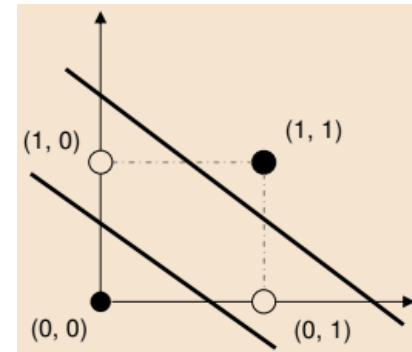
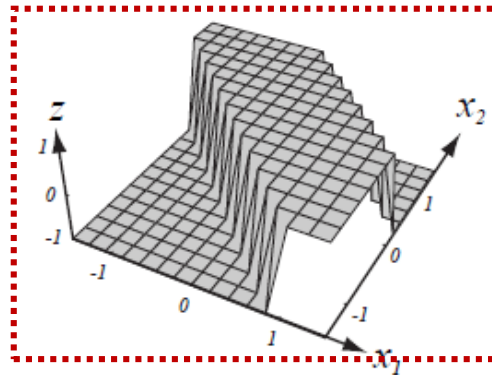
XOR



$$\begin{aligned} w_1 &= ? \\ w_2 &= ? \\ \theta &= ? \end{aligned}$$

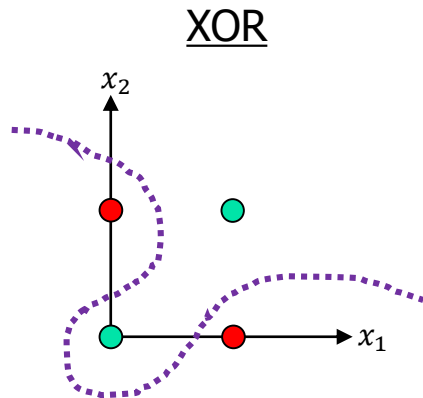
# Perceptron

- Capacity of multi-layer perceptron

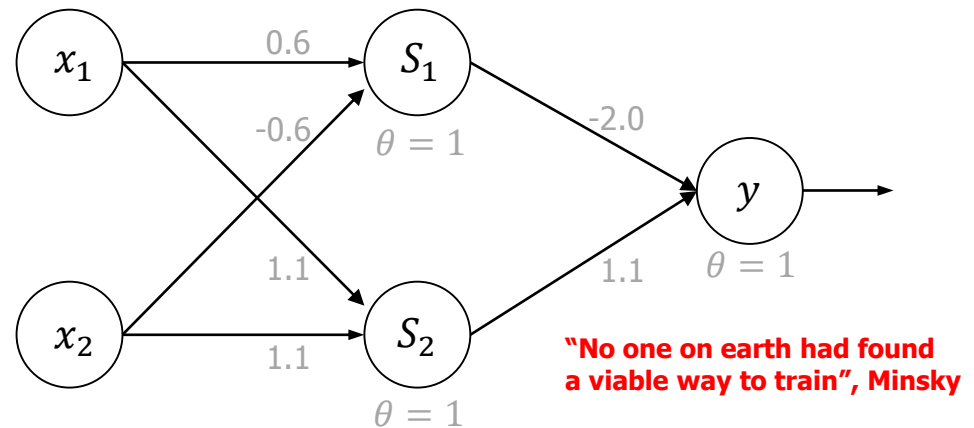


# Perceptron

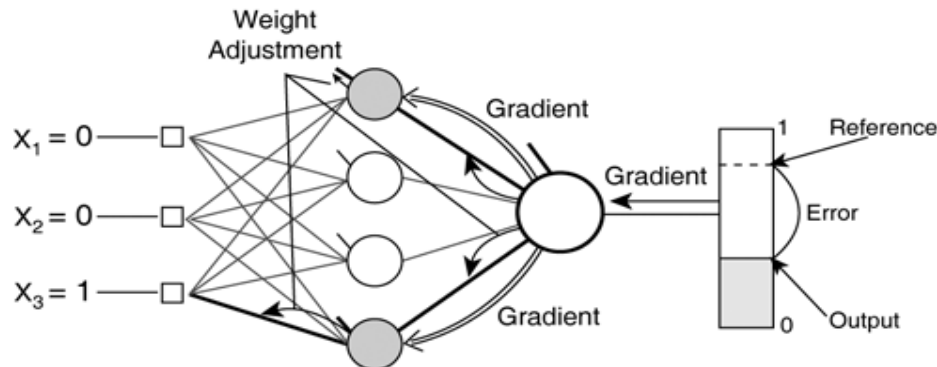
- 1969년, Minsky와 Papert가 그들의 저서 "Perceptrons"에서 퍼셉트론이 비선형 분리 문제를 풀 수 없음을 증명하여, 침체기에 들어감 (1st AI winter)



needs  
multi-layer  
perceptron  
(MLP)

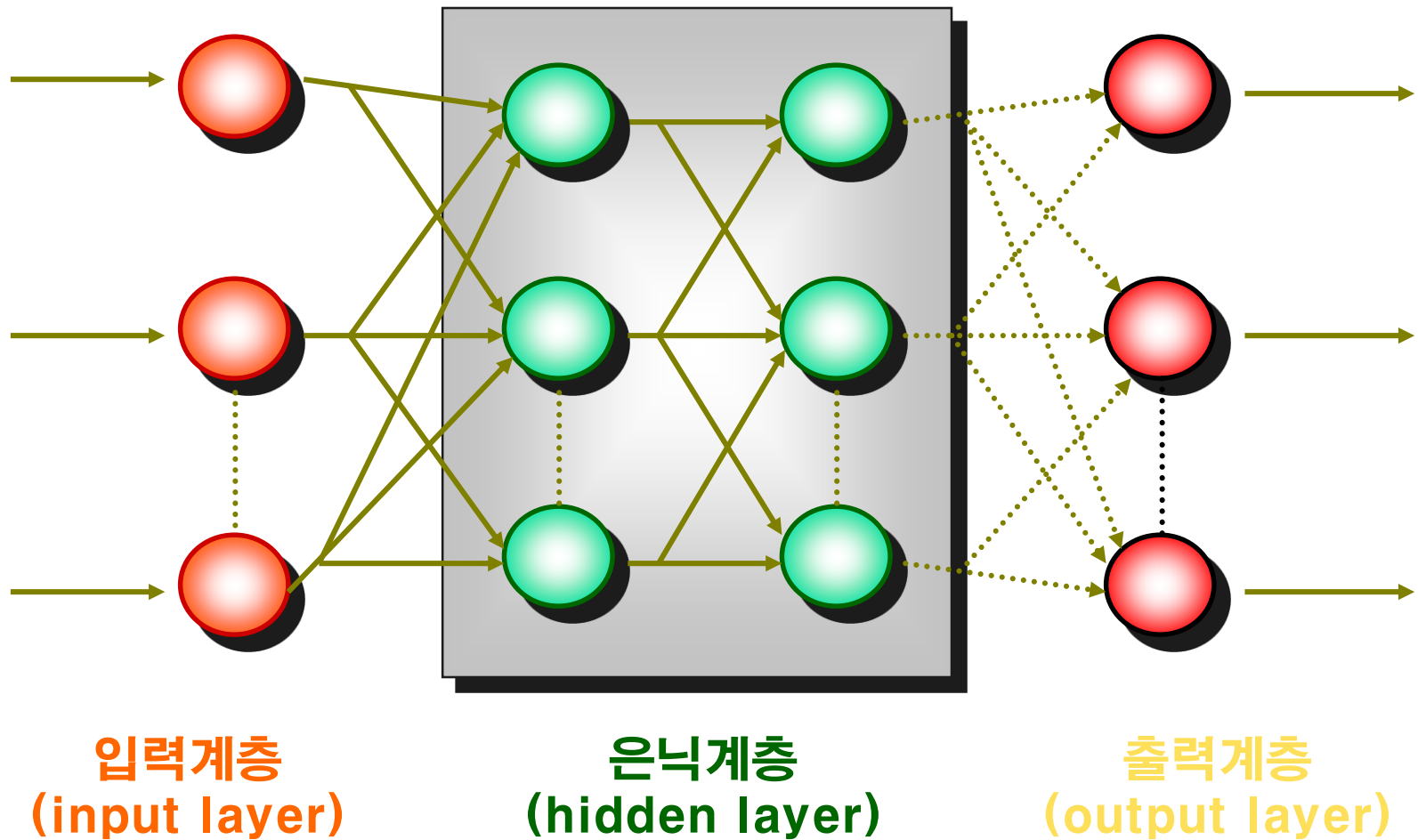


- 1986년, PDP그룹에 의해 Back-Propagation 알고리즘을 사용하는 MLP가 탄생되어 신경망의 다양한 분야에 대한 연구와 응용이 이루어짐.





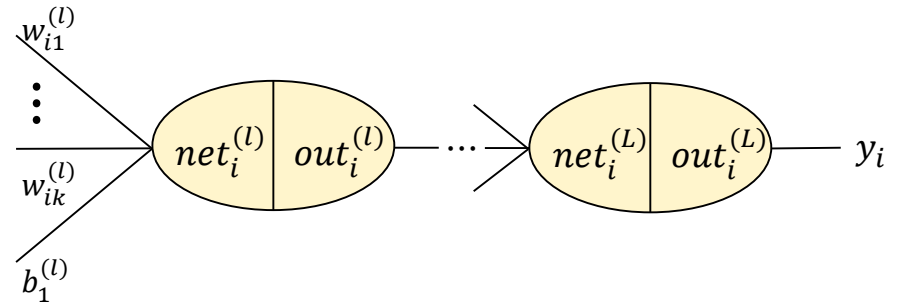
# MLP: multi-layer perceptron



# MLP

- Total net input에 편향(bias) 추가

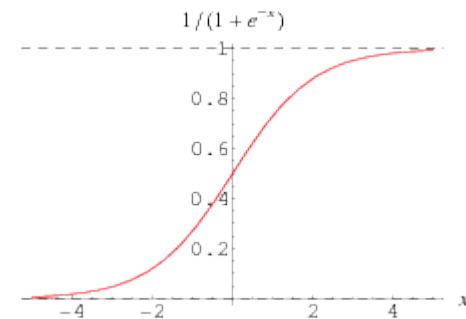
$$net_i^{(l)} = \sum_{j=1}^k w_{ij}^{(l)} \cdot out_j^{(l-1)} + b_1^{(l)}$$



- Activation 함수

- Hidden Node: 주로 시그모이드(sigmoid / logistic) 함수 사용

$$out_i^{(l)} = \frac{1}{1 + \exp(-net_i^{(l)})}$$

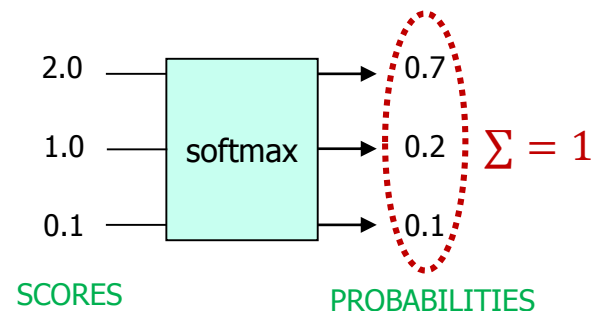


- Output Node

- 추정(y가 연속형) 문제에는 항등(identity) 함수 사용. 즉,  $y = net^{(L)}$
- 분류 문제에는 소프트맥스(softmax) 함수 사용

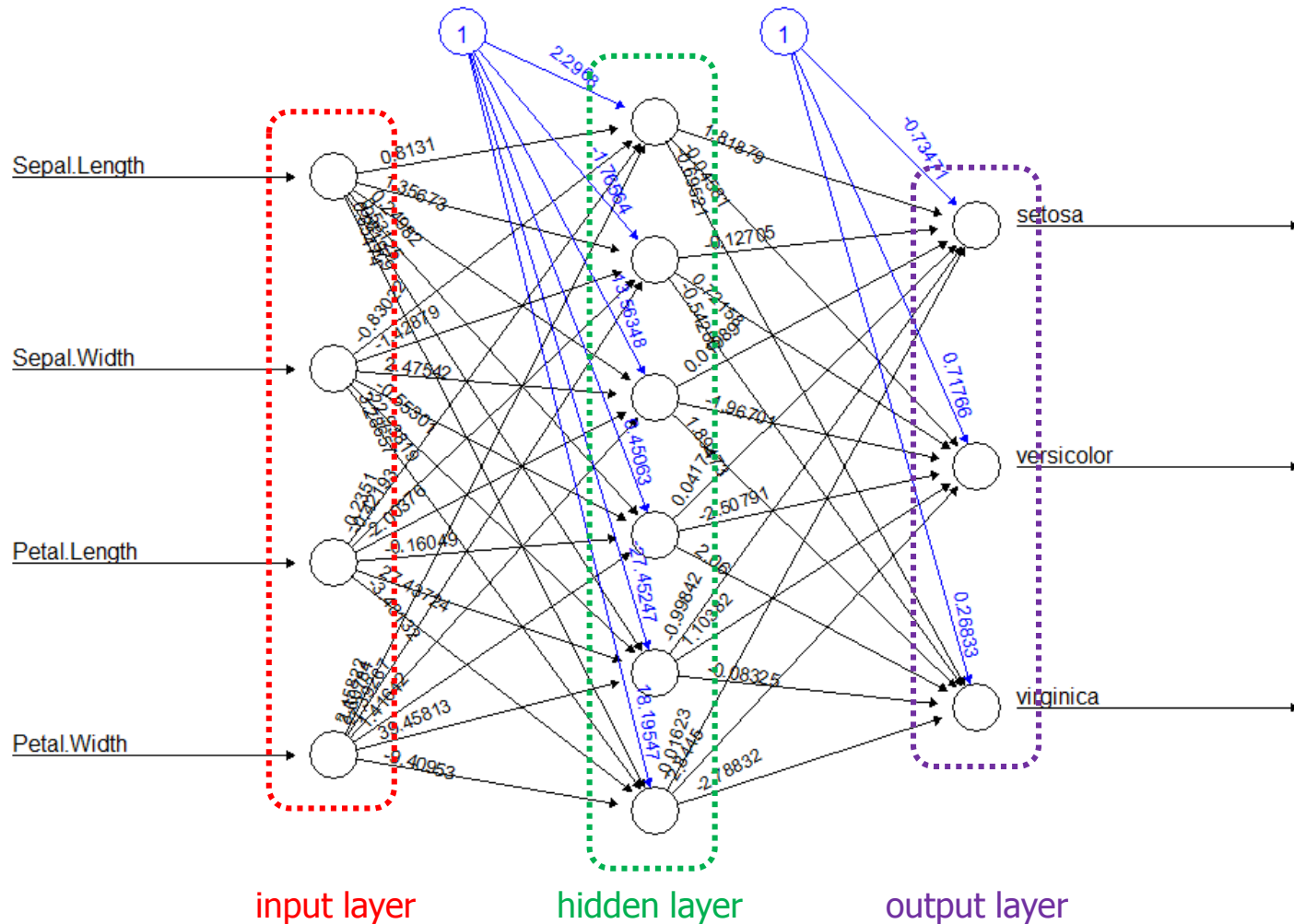
$$y_n = \frac{\exp(net_n^{(L)})}{\sum_{i=1}^N \exp(net_i^{(L)})}$$

$N$ : 출력층의 뉴런 수,  $0 \leq y_n \leq 1$



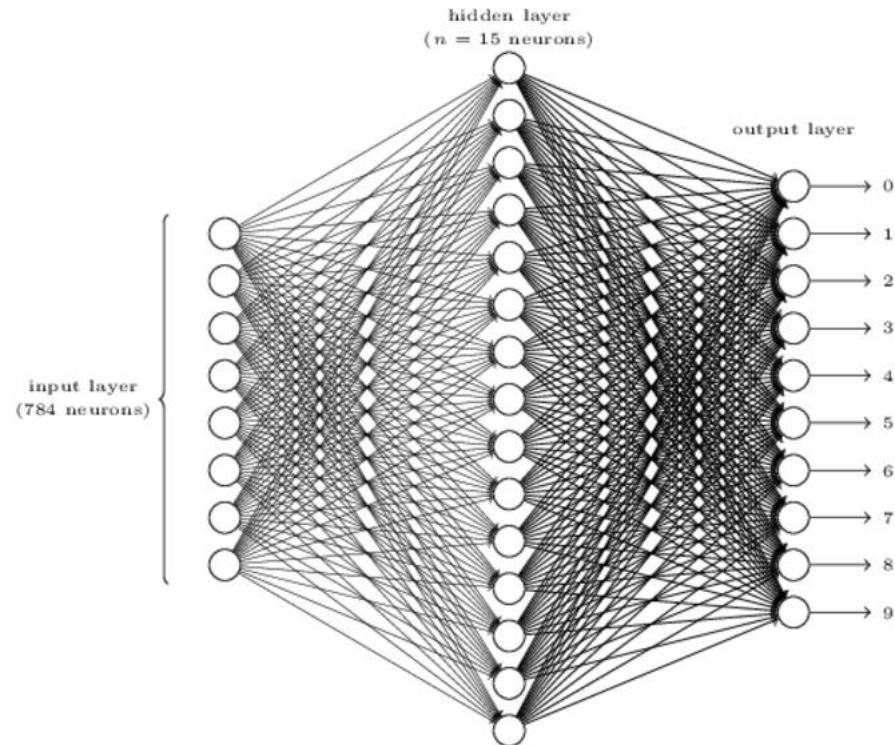
# MLP

- Classification Example for IRIS data by MLP



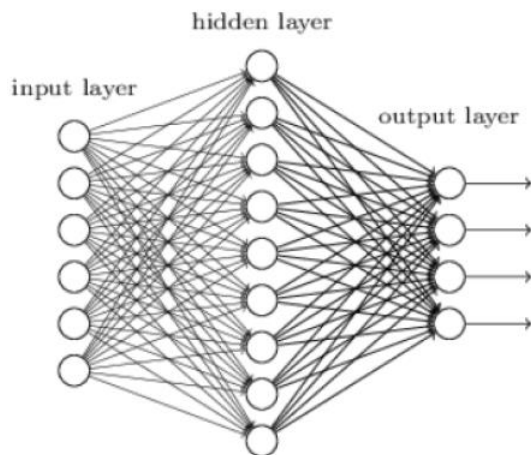
# MLP

- A simple network to classify handwritten digits
  - Training data: 28X28(784) pixel, greyscale(0.0~1.0)



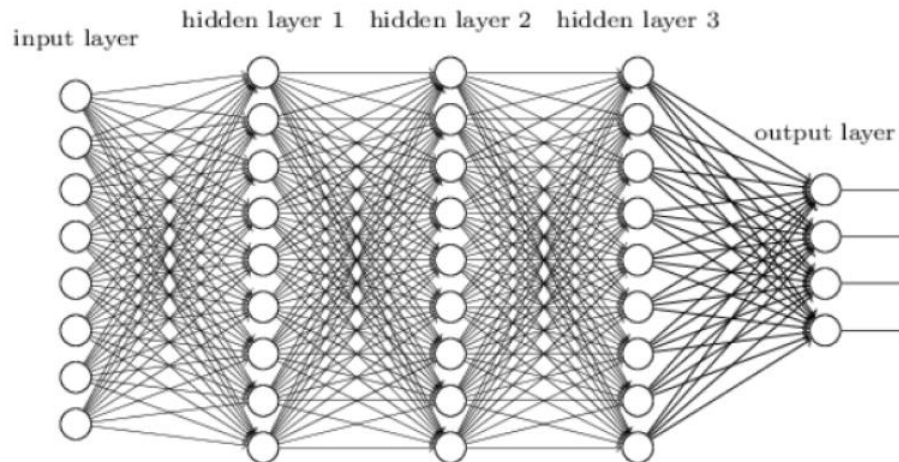
# Shallow NN vs. Deep NN

"Non-deep" feedforward neural network



- Input layer: 6개의 input neuron으로 구성 (dimension = 6)
- Hidden layer: 1개 layer, 9개 neuron(unit)
- Output layer: 4개 unit

Deep neural network



- Input layer: 8개의 input neuron으로 구성 (dimension = 8)
- Hidden layer: 3개 layer, 각 9개 unit
- Output layer: 4개 unit

# Learning algorithm

## ■ Error(cost) function

- Mean squared error:  $E(\mathbf{W}) = \frac{1}{2} \sum_{i=1}^N (y_i - \hat{y}_i)^2$
- Cross entropy error:  $E(\mathbf{W}) = - \sum_{i=1}^N y_i \log(\hat{y}_i)$

## ■ Learning network

- adjusting weights to minimize error ( $E$ )
- Iterative numerical procedure

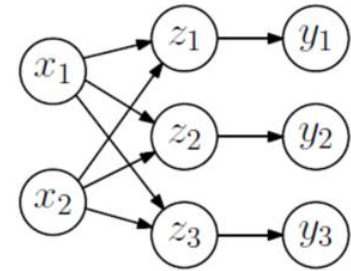
$$\mathbf{W}^{(t+1)} = \mathbf{W}^{(t)} + \nabla \mathbf{W}^{(t)}$$

## ■ Gradient descent optimization

$$\mathbf{W}^{(t+1)} = \mathbf{W}^{(t)} - \eta \frac{\partial E(\mathbf{W}^{(t)})}{\partial \mathbf{W}^{(t)}} \quad \eta: \text{learning rate } (0 \sim 1)$$

## ■ Mini-batch stochastic gradient descent (**mini-batch SGD**)

- Instead of the entire training data, work with **mini-batch** of  $m$  examples in each iteration (*note: **epoch***)



$$z_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1$$

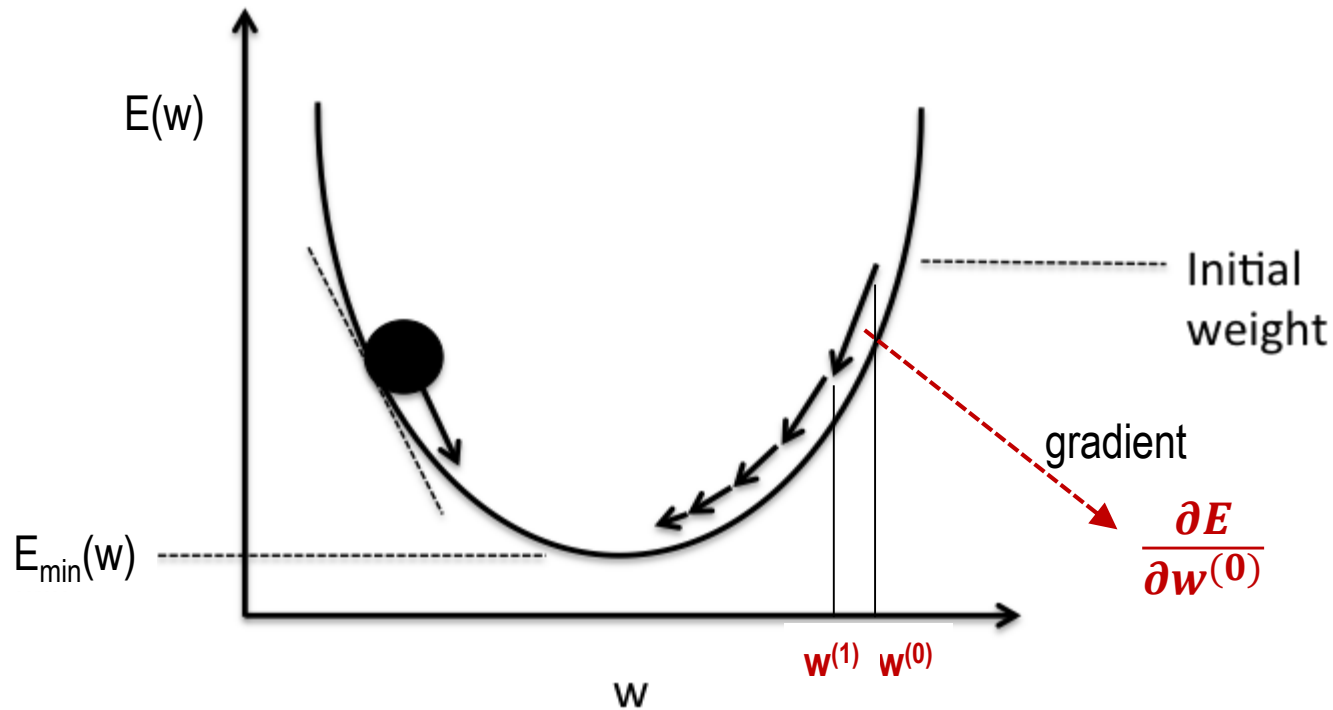
$$z_2 = w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2$$

$$z_3 = w_{31}x_1 + w_{32}x_2 + w_{33}x_3 + b_3$$

$$\mathbf{Z} = \mathbf{W}\mathbf{b}$$

$$y_i = \sigma(z_i)$$

# Schematics of gradient descent



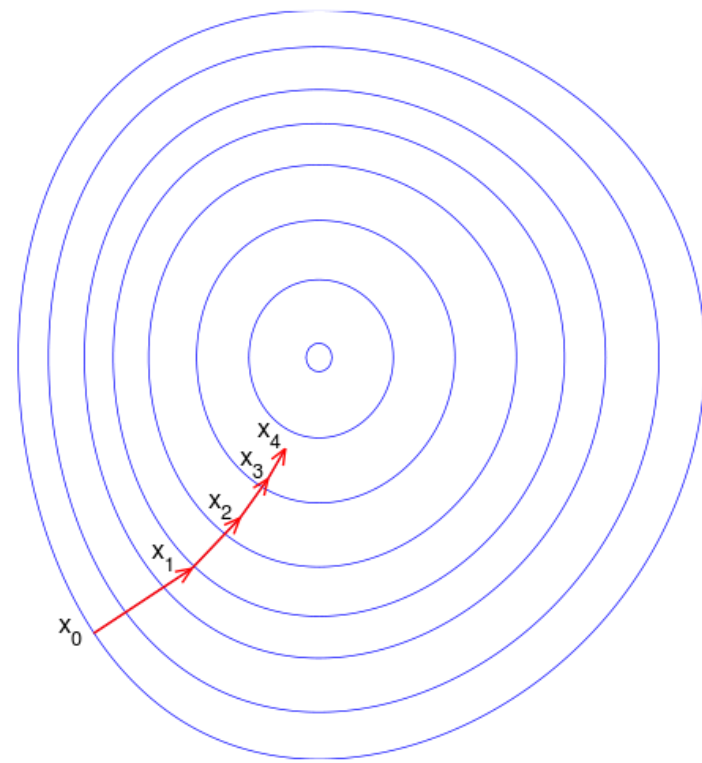
$$w^{(1)} = w^{(0)} - \eta \frac{\partial E}{\partial w^{(0)}}$$

# Schematics of gradient descent

- $x \in \mathbb{R}^n$
- 현재값  $x^{(k)}$ 에서 가장 빠르게  $f(x)$ 를 감소시키는 방향  $-\nabla f(x^{(k)})$ 으로  $\alpha$ 만큼씩 이동

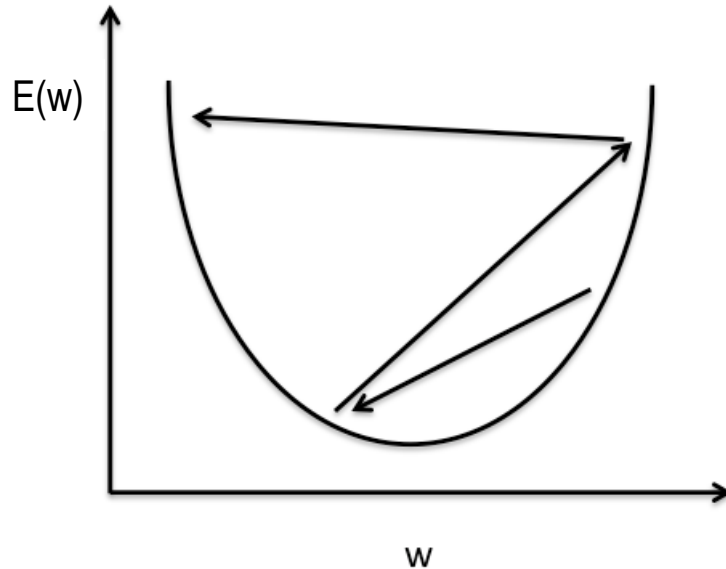
$$\begin{bmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \\ \vdots \\ x_n^{(k+1)} \end{bmatrix} = \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_n^{(k)} \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial}{\partial x_1} f(x^{(k)}) \\ \frac{\partial}{\partial x_2} f(x^{(k)}) \\ \vdots \\ \frac{\partial}{\partial x_n} f(x^{(k)}) \end{bmatrix}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \left\{ \nabla f \left( \mathbf{x}^{(k)} \right) \right\}^T$$

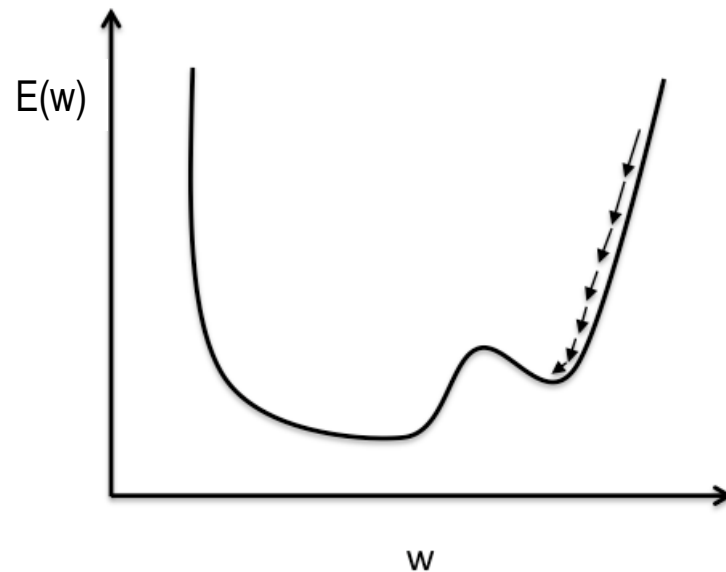




# Learning rate



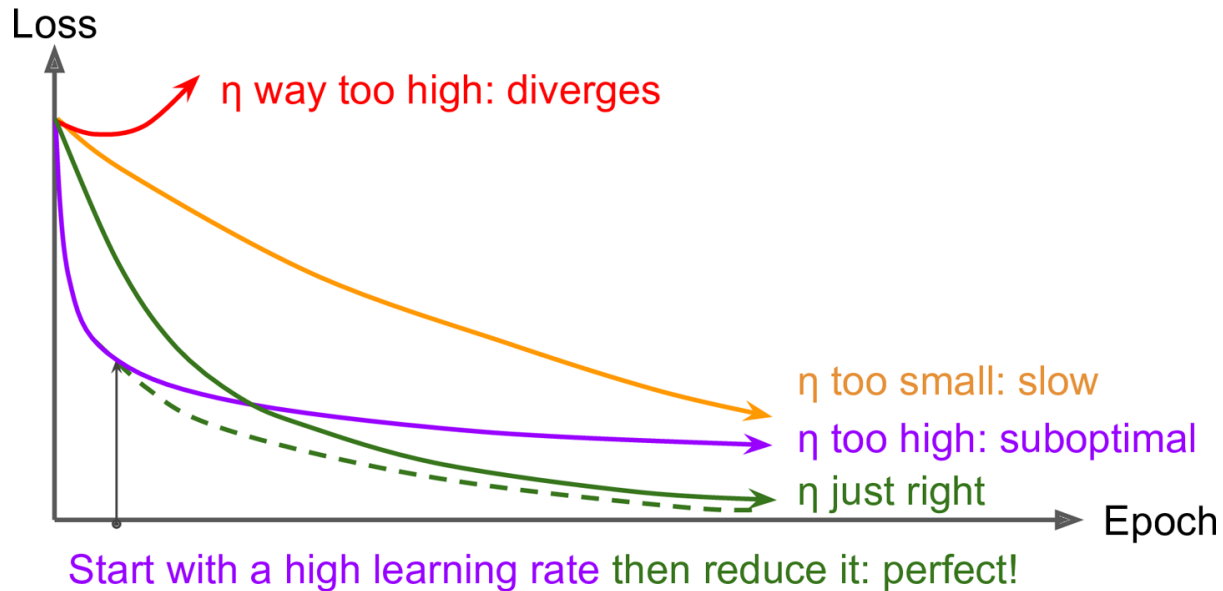
**Large learning rate: Overshooting.**



**Small learning rate: Many iterations until convergence and trapping in local minima.**

👉 recommended learning rate  $\approx 0.01 \sim 0.001$

# Learning rate Scheduling



- 미리 정의된 개별적인 고정 학습률
- 성능 기반 스케줄링
- 지수 기반 스케줄링 (Exponential Scheduling) \*
- 거듭제곱 기반 스케줄링 (Power Scheduling)



# Computing the gradient

---

## ■ Numerical gradient

- computes the partial derivative of each weight using the finite difference approximation

$$\frac{\partial E}{\partial w_{ji}} = \frac{E(w_{ji} + \epsilon) - E(w_{ji})}{\epsilon} + O(\epsilon)$$

- is very simple, but approximate and very computationally expensive;  $O(W^2)$

## ■ Back-propagation

- enables us to compute the gradients very efficiently;  $O(W)$
- uses the chain rule for gradient descent
- consists of a two-pass procedure:
  - **forward pass**: fix weights, evaluate  $y$  from  $x$
  - **backward pass**: compute the error  $E$  and back propagate it

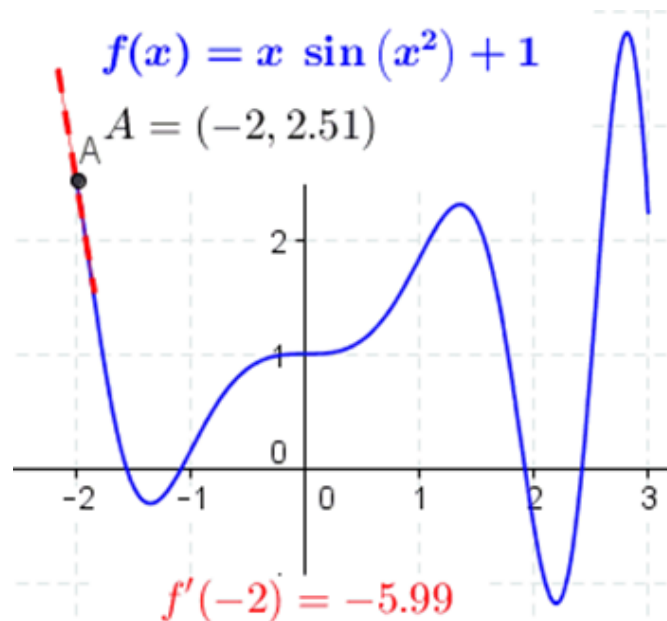
# [Ref] Scalar Differentiation

## ■ 미분의 정의

- 함수  $y = f(x)$ 의 정의역에서 임의의  $x$ 에서 미분은 다음과 같이 정의된다.

$$f'(x) = \frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

- 점  $x$ 에서  $f$ 가 변하는 순간 변화율





# [Ref] Scalar Differentiation: Formulas

- 많이 사용되는 미분 공식

$$\frac{d}{dx}(x^n) = nx^{n-1} \quad \frac{d}{dx}(\ln x) = \frac{1}{x}$$

$$\frac{d}{dx}(e^x) = e^x \quad \frac{d}{dx}(\sin x) = \cos x$$

$$\frac{d}{dx}(\cos x) = -\sin x \quad \frac{d}{dx}(\tanh x) = 1 - \tanh^2 x$$

$$\frac{d}{dx}\left(\frac{1}{1 + \exp(-x)}\right) = \left(\frac{1}{1 + \exp(-x)}\right)\left(1 - \frac{1}{1 + \exp(-x)}\right)$$



# [Ref] Scalar Differentiation: Rules

- Sum rule

$$(f(x) + g(x))' = f'(x) + g'(x) = \frac{df}{dx} + \frac{dg}{dx}$$

- Product rule

$$(f(x)g(x))' = f'(x)g(x) + f(x)g'(x) = \frac{df}{dx}g(x) + f(x)\frac{dg}{dx}$$

- Chain rule

$$(g \circ f)'(x) = (g(f(x)))' = g'(f(x))f'(x) = \frac{dg}{df} \frac{df}{dx}$$



# [Ref] Scalar Differentiation: Chain Rule

---

- Example

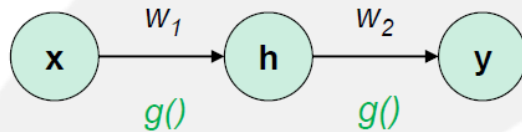
$$g(z) = \tanh(z)$$

$$z = f(x) = x^n$$

$$(g \circ f)'(x) = \underbrace{(1 - \tanh^2(x^n))}_{dg/df} \underbrace{nx^{n-1}}_{df/dx}$$

# Backpropagation (an easy case)

$$y = g(h \cdot w_2) = g(g(x \cdot w_1) \cdot w_2)$$



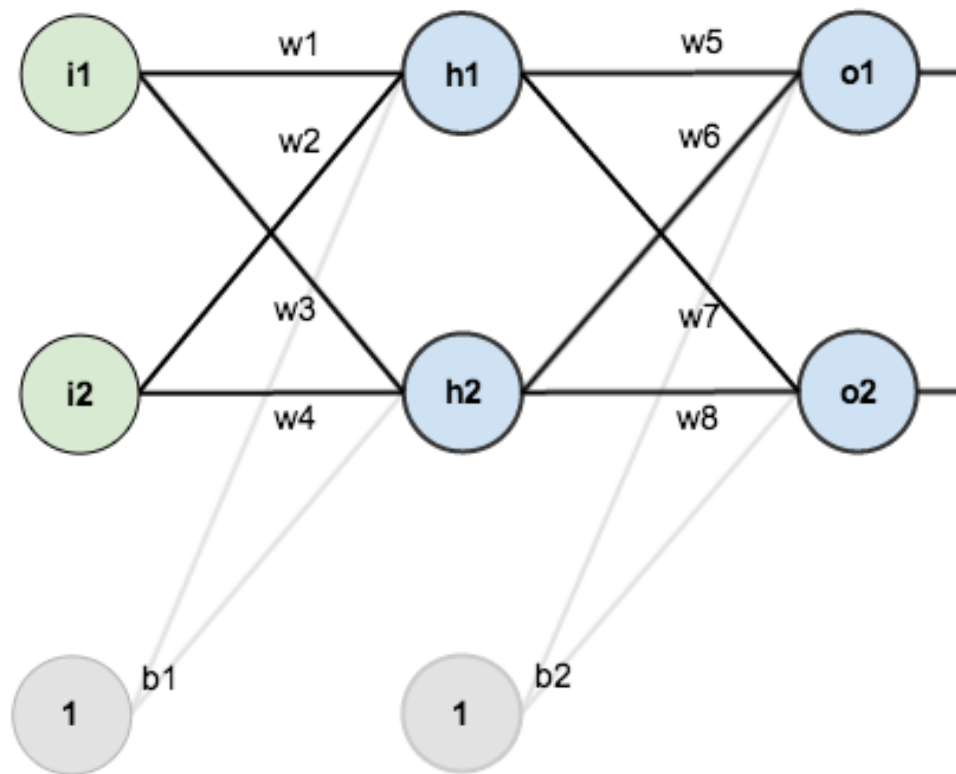
$E(\mathbf{w}) = \frac{1}{2}(y - y_i)^2$ , where  $y_i$  is a real value.  
 $g' = g(1 - g)$ , when  $g$  is a sigmoid ft.

$$\begin{aligned}\frac{\partial E(\mathbf{w})}{\partial w_2} &= (y - y_i) \cdot \frac{\partial y}{\partial w_2} \\ &= (y - y_i) \cdot \frac{\partial g(h \cdot w_2)}{\partial w_2} \\ &= (y - y_i) \cdot g(h \cdot w_2) \cdot (1 - g(h \cdot w_2)) \times \frac{\partial(h \cdot w_2)}{\partial w_2} \\ &= (y - y_i) \cdot y \cdot (1 - y) \cdot h = E_y \cdot h\end{aligned}$$

$$\begin{aligned}\frac{\partial E(\mathbf{w})}{\partial w_1} &= (y - y_i) \cdot \frac{\partial y}{\partial w_1} \\ &= (y - y_i) \cdot y \cdot (1 - y) \cdot \frac{\partial(h \cdot w_2)}{\partial w_1} \\ &= (y - y_i) \cdot y \cdot (1 - y) \cdot w_2 \cdot \frac{\partial h}{\partial w_1} \\ &= (y - y_i) \cdot y \cdot (1 - y) \cdot w_2 \cdot h \cdot (1 - h) \cdot \frac{\partial(x \cdot w_1)}{\partial w_1} \\ &= (y - y_i) \cdot y \cdot (1 - y) \cdot w_2 \cdot h \cdot (1 - h) \cdot x = E_h \cdot x\end{aligned}$$



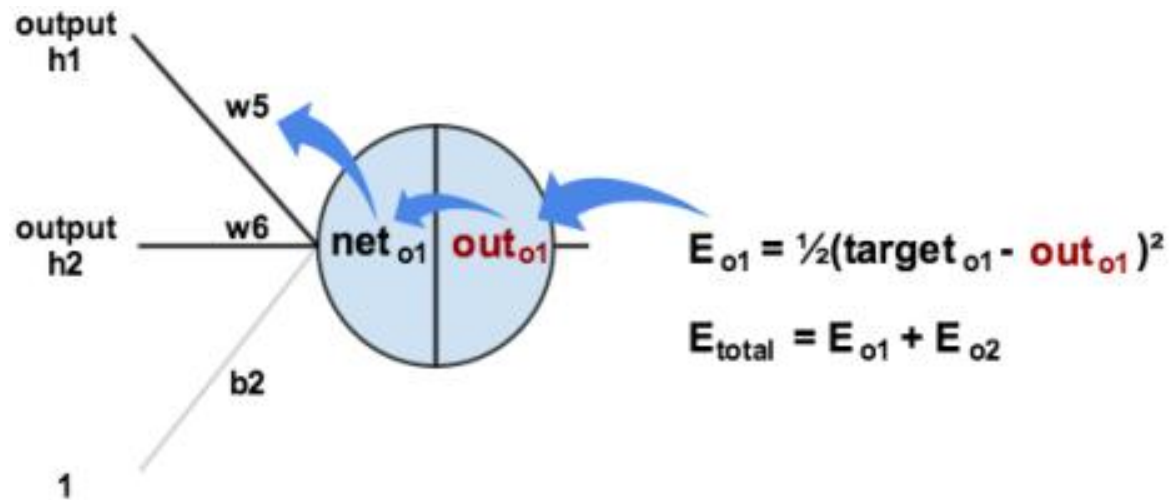
# Backpropagation



*Assume the hidden and output neurons use the sigmoid function for activation*

# Backpropagation

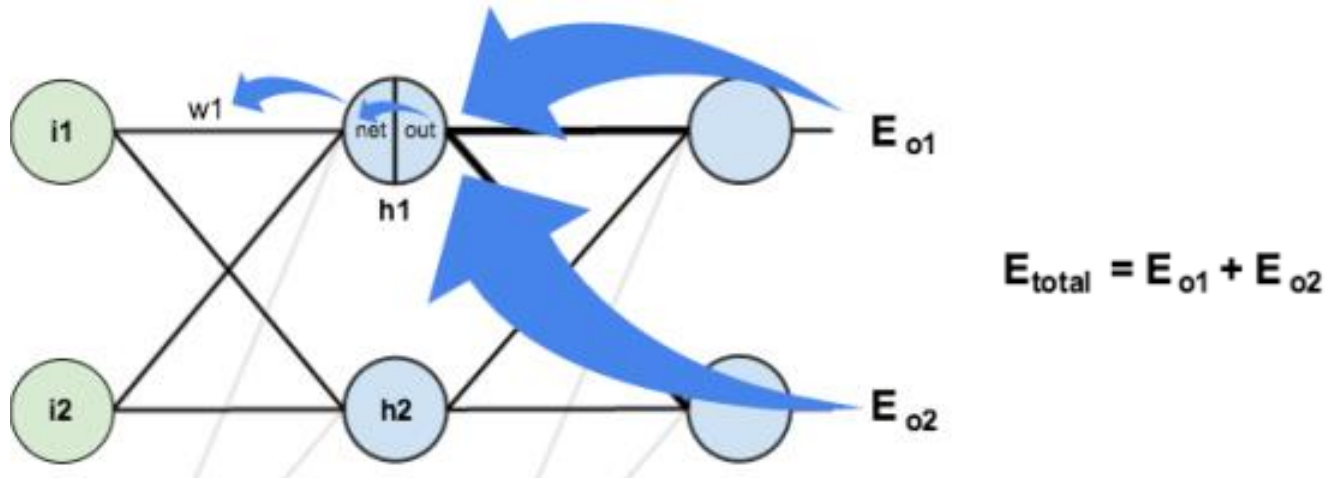
- Output layer



$$\begin{aligned}\frac{\partial E_{\text{total}}}{\partial w_5} &= \frac{\partial E_{o1}}{\partial \text{out}_{o1}} \times \frac{\partial \text{out}_{o1}}{\partial \text{net}_{o1}} \times \frac{\partial \text{net}_{o1}}{\partial w_5} \\ &= -(target_{o1} - out_{o1}) \times out_{o1}(1 - out_{o1}) \times out_{h1} \\ &= \delta_{o1} \times out_{h1}\end{aligned}$$

# Backpropagation

- Hidden layer



$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} \times \frac{\partial out_{h1}}{\partial net_{h1}} \times \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

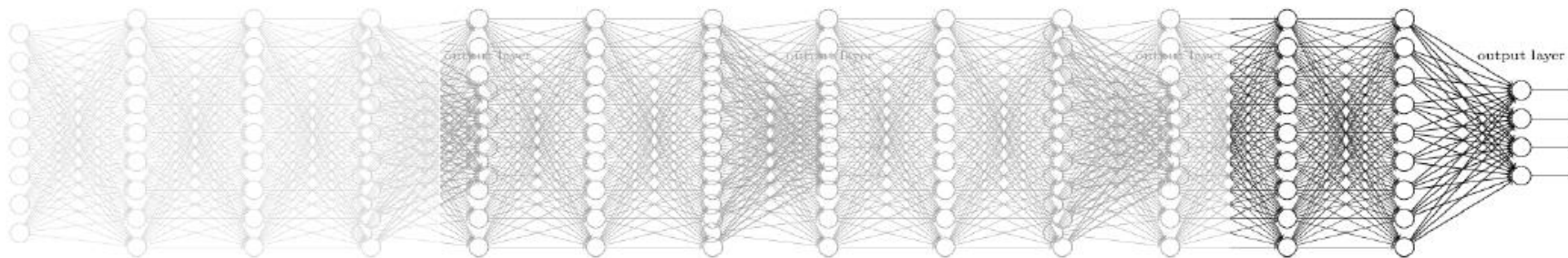
$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial out_{h1}}$$

$$= \left( \sum_o \frac{\partial E_o}{\partial out_o} \times \frac{\partial out_o}{\partial net_o} \times \frac{\partial net_o}{\partial out_{h1}} \right) \times \frac{\partial out_{h1}}{\partial net_{h1}} \times \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial net_{o1}}$$

$$= \left( \sum_o \delta_o \times w_{ho} \right) \times out_{h1} (1 - out_{h1}) \times i_1$$

# Vanishing gradient problem

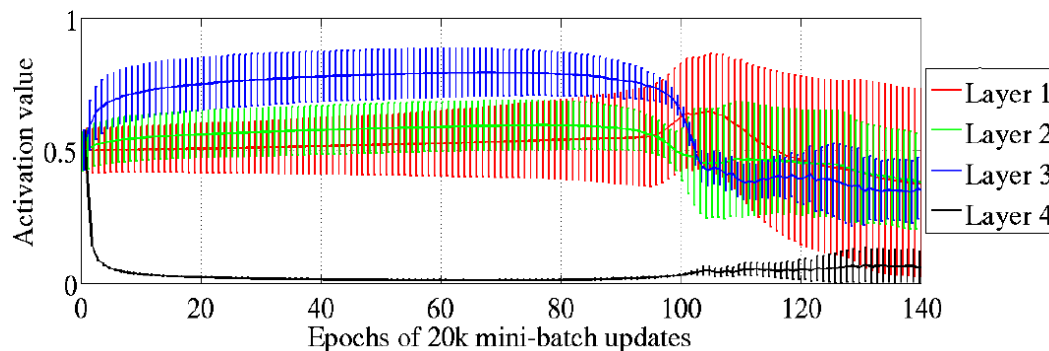
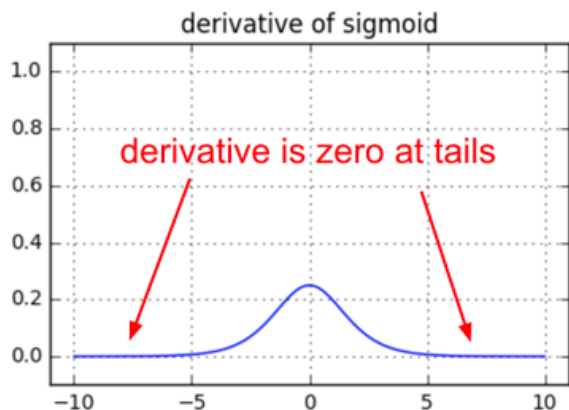


Source: <http://hunkim.github.io/ml/>

- meaning that the gradient (error signal) decreases exponentially with  $n$  and the front layers train very slowly.
- why?
  - Initializing the weights in a stupid way
  - Using wrong type of non-linearity




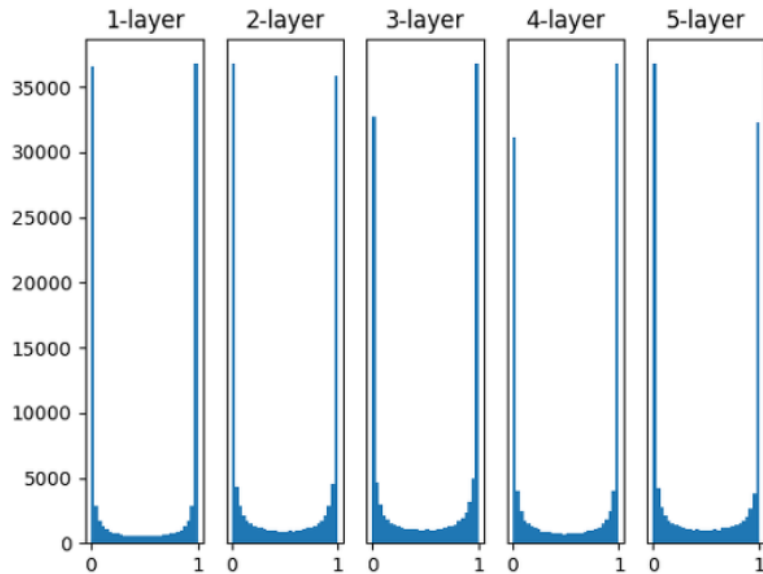
**2nd AI winter**



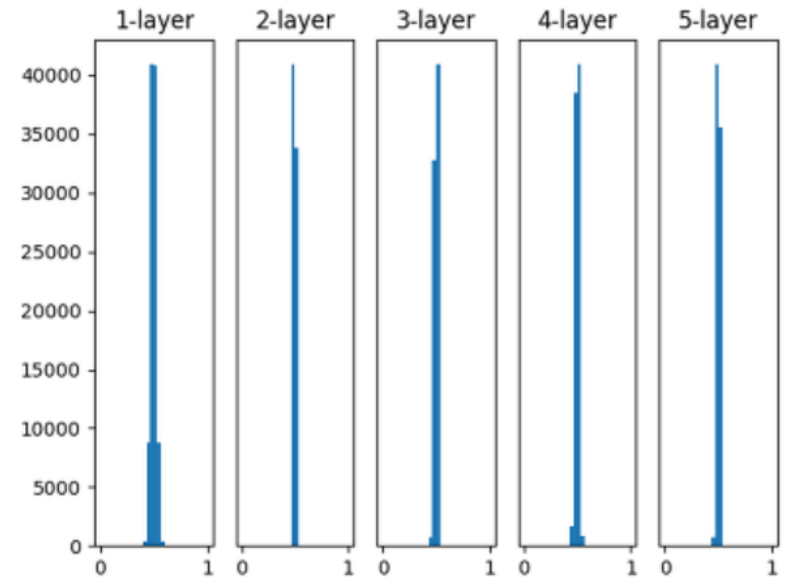
Source: *Understanding the difficulty of training deep feedforward neural networks*

# 초기 가중치에 따른 활성화 값 분포

- Activation function = sigmoid,   
init.  $W \sim N(0, 1)$



- Activation function = sigmoid,  
init.  $W \sim N(0, 0.01)$

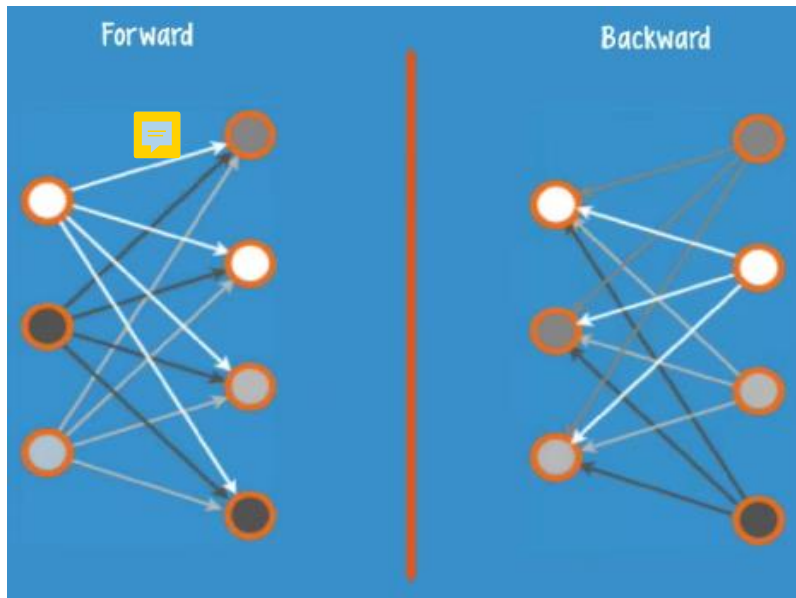


Source: "밑바닥부터 시작하는 딥러닝", 사이토 고키

# 단순한 신경망일수록 초기 가중치에 민감하다.

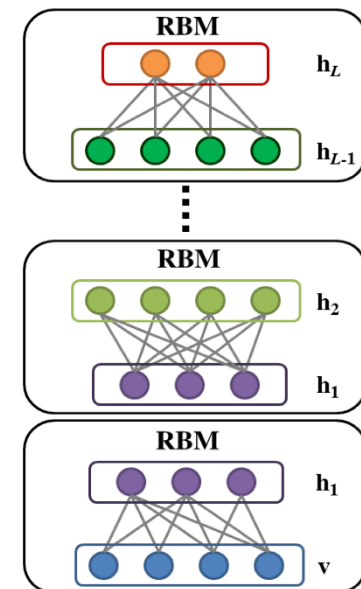
# Deep learning begins

- 2006년, Geoffrey Hinton에 의해 RBM 기반의 pre-training으로 Deep Neural Network의 학습이 가능해 지면서 "Deep Learning"이라는 새로운 이름으로 다시 주목을 받기 시작함.
- Deep Belief Networks
  - Hinton et al. (2006) "A Fast Learning Algorithm for Deep Belief Nets"
  - Apply the RBM idea on adjacent two layers as a pre-training step, and continue the first process to all layers => This will initialize good weight values !



*RBM(Restricted Boltzman Machine) structure*

Source: <http://hunkim.github.io/ml/>



*DBN*

Source: Maximum Entropy Learning with Deep Belief Networks



# Modern weight value initialization

*Makes sure the weights are 'just right', not too small, not too big*

## ■ Xavier initialization

- Glorot & Bengio (2010) "Understanding the difficulty of training deep feedforward neural networks"
- initializes the weights by drawing them from a distribution with zero mean and a specific variance,

$$Var(\mathbf{W}) = \frac{1}{n_{in}} \img alt="comment icon" data-bbox="550 505 575 540"/>$$

where  $\mathbf{W}$  is the initialization distribution for the neuron in question, and  $n_{in}$  is the number of neurons feeding into it. The distribution used is typically Gaussian or uniform.

## ■ He's initialization

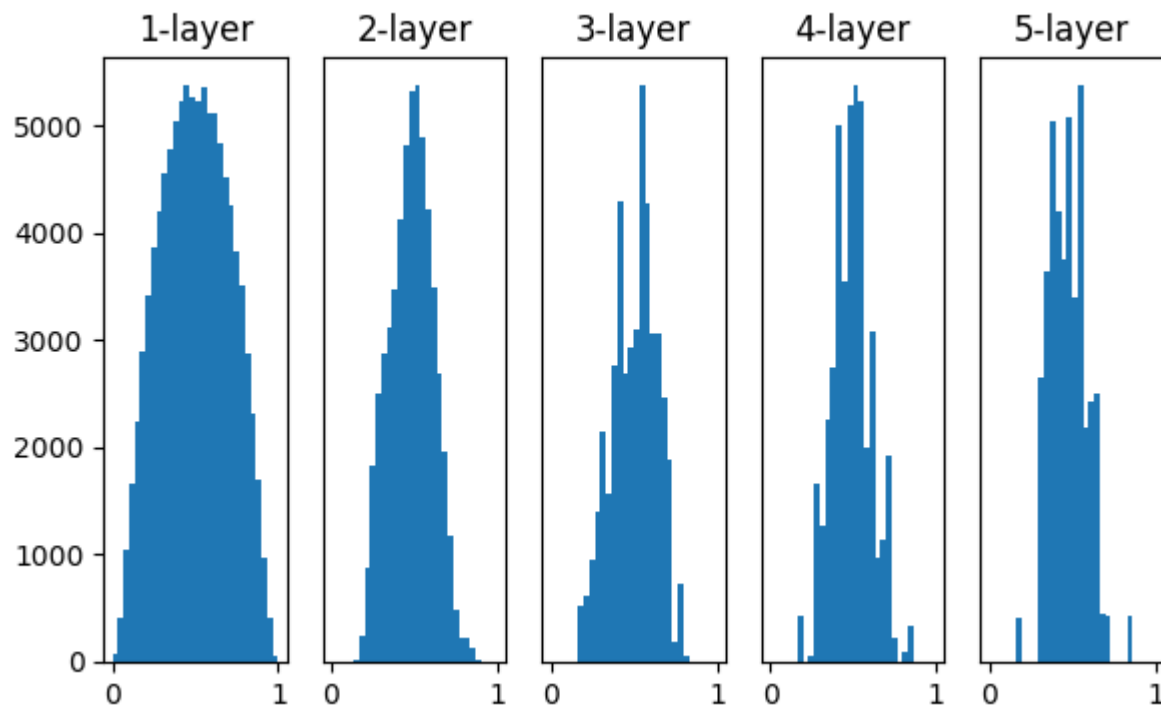
- He et al. (2015) "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification"
- For ReLU, it is recommended using



$$Var(\mathbf{W}) = \frac{2}{n_{in}}$$

# Modern weight value initialization

- Xavier initialization 적용 시의 활성화 값 분포

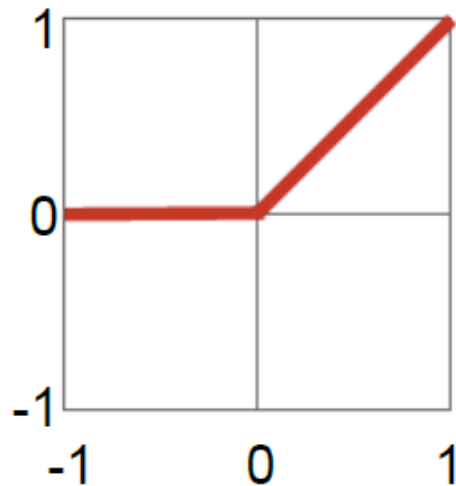




# Modern activation functions

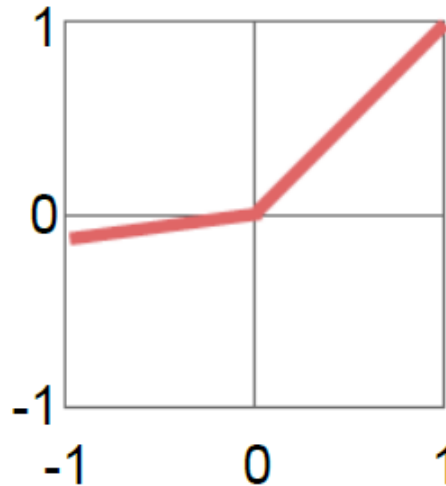
Source: *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*

**Rectified Linear Unit  
(ReLU)**



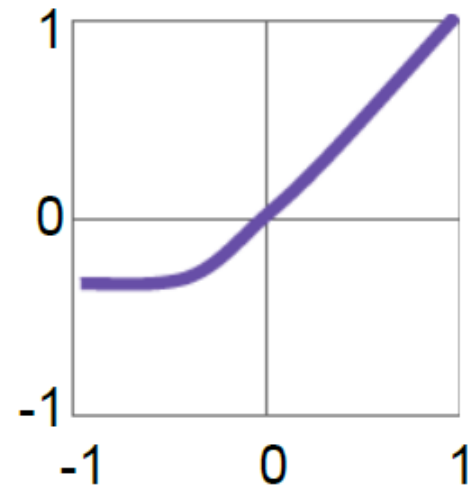
$$y = \max(0, x)$$

**Leaky ReLU**



$$y = \max(\alpha x, x)$$

**Exponential LU**

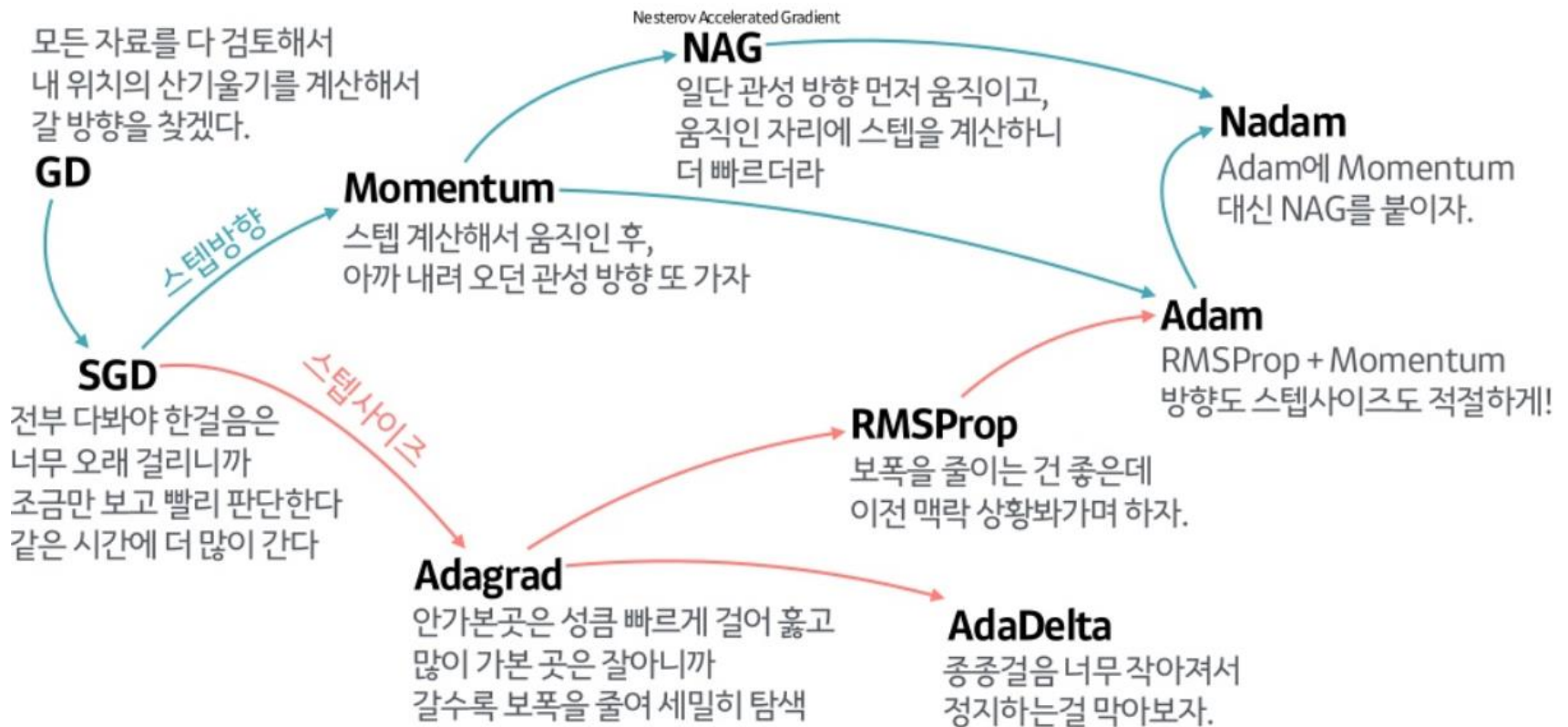


$$y = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$$

# *scikit-learn*에서는 *relu*(기본값), *logistic*, *tanh*, *identity*를 지원

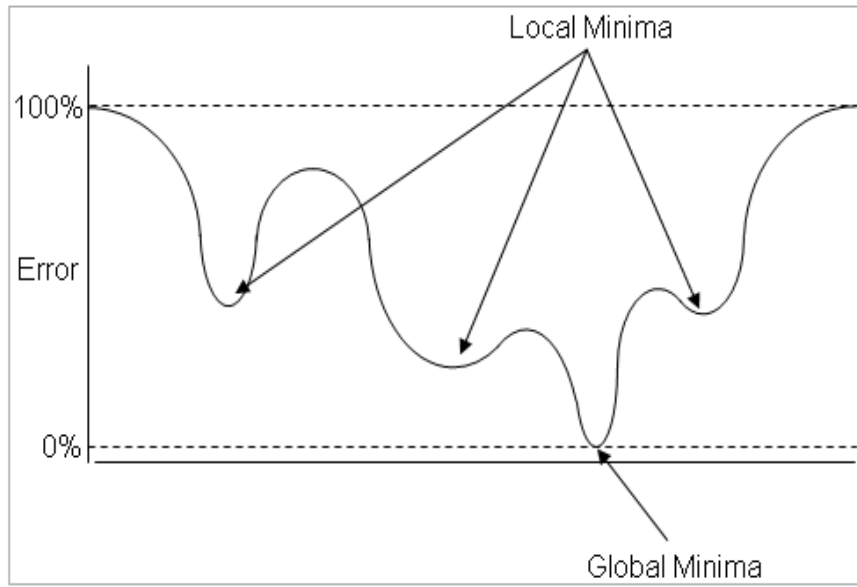
# Optimization techniques

Source: <https://www.slideshare.net/yongho/ss-79607172>

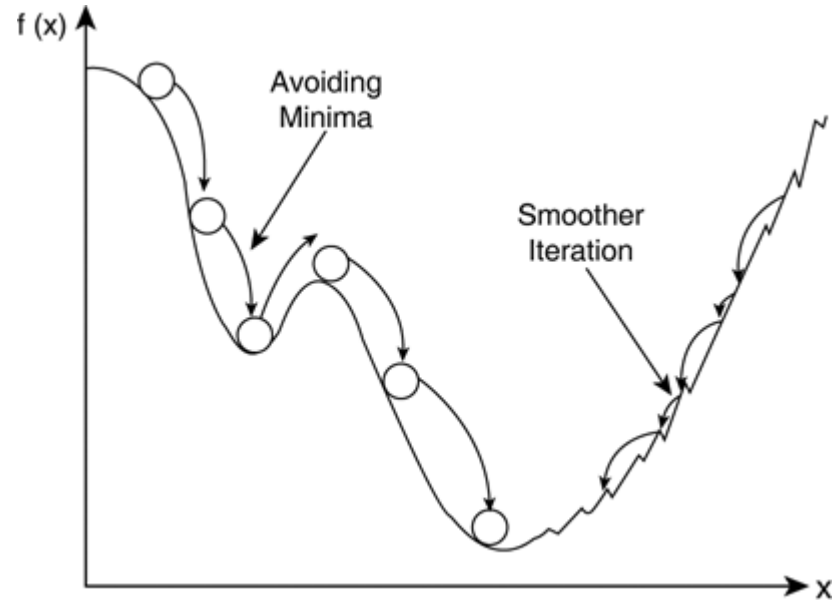


# *scikit-learn*에서는 *adam*(기본값), *sgd*, *lbfgs*를 지원

# Momentum



Source: <http://mnemstudio.org/neural-networks-backpropagation.htm>



Source: <http://www.yaldex.com>

기존 업데이트에 사용했던 기울기의 일정 비율을 남겨서 현재의 기울기와 더하여 업데이트함.

$$\mathbf{v}^{(t+1)} = \alpha \mathbf{v}^{(t)} + \eta \frac{\partial E}{\partial \mathbf{W}^{(t)}}$$
$$\mathbf{W}^{(t+1)} = \mathbf{W}^{(t)} - \mathbf{v}^{(t+1)}$$

recommended  $\alpha \approx 0.9$

# Modern optimizers

- **Adagrad**  
(Adaptive Gradient)

$$G_t = G_{t-1} + (\nabla_{\theta} J(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \nabla_{\theta} J(\theta_t)$$

- **RMSProp**

$$G = \gamma G + (1 - \gamma)(\nabla_{\theta} J(\theta_t))^2$$
$$\theta = \theta - \frac{\eta}{\sqrt{G + \epsilon}} \cdot \nabla_{\theta} J(\theta_t)$$

- **Adam**  
(Adaptive Moment Estimation)

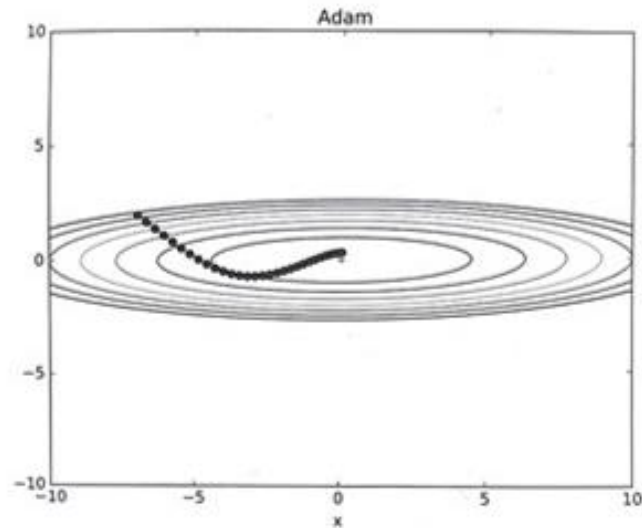
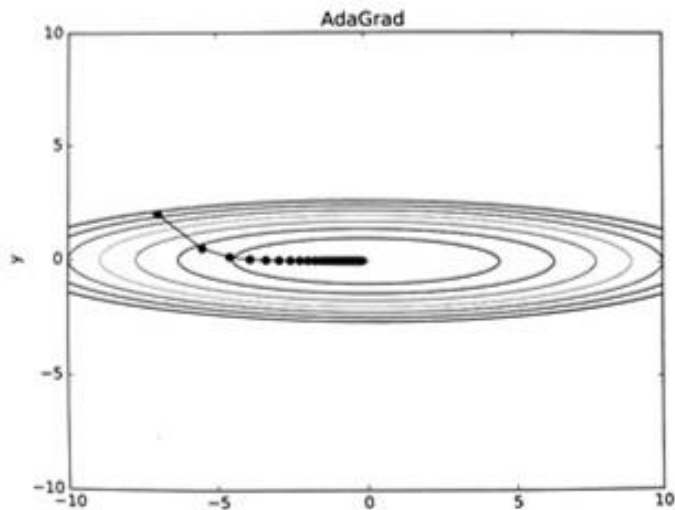
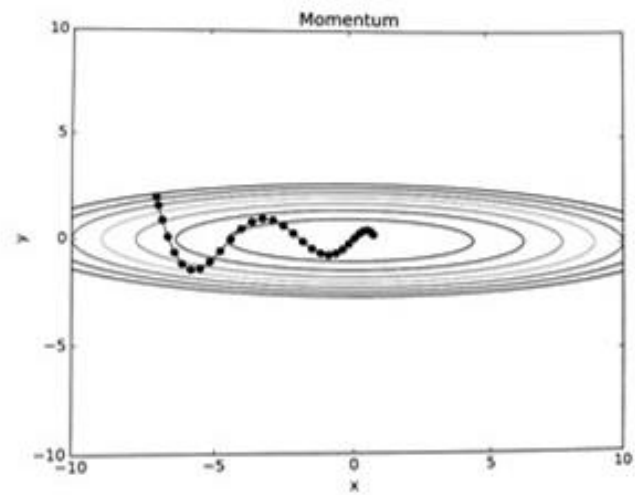
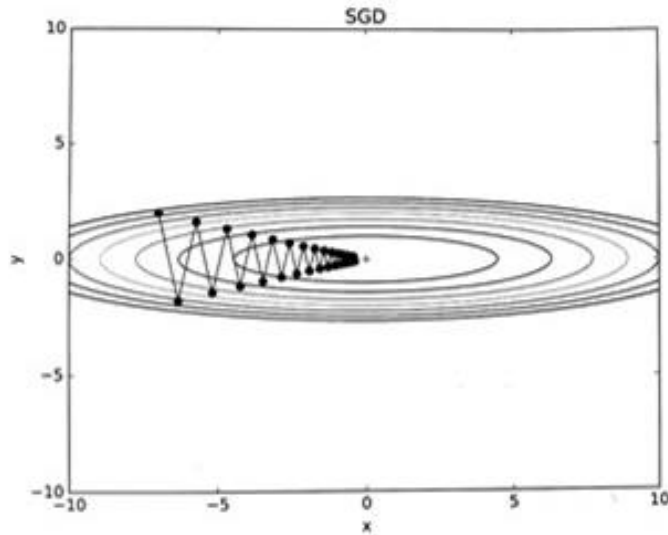
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{\theta} J(\theta) \quad \leftarrow \text{Momentum}$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) (\nabla_{\theta} J(\theta))^2 \quad \leftarrow \text{RMSProp}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta = \theta - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

# Optimizer Comparison



Source: "밑바닥부터 시작하는 딥러닝", 한빛미디어, 2017.

# Solutions for overfitting

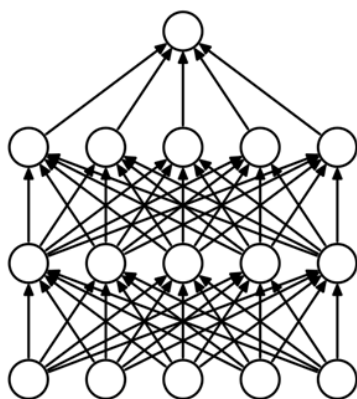
## ■ Regularization (L2 penalty)

- Let's not have too big numbers in the weight

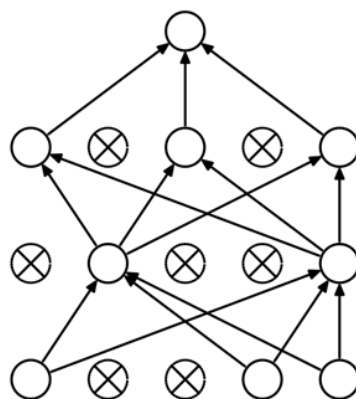
$$E(W) + \frac{1}{2}\lambda W^2$$

## ■ Dropout

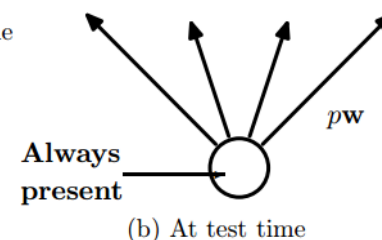
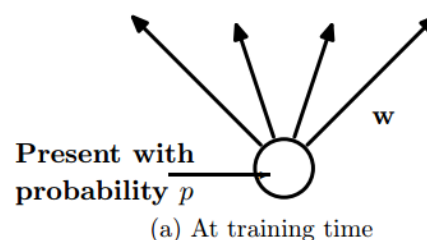
- Srivastava et al. (2014) "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"
- randomly selected neurons are ignored during training
- If a unit is retained with probability  $p$  during training, the outgoing weights of that unit are multiplied by  $p$  at test time



(a) Standard Neural Net



(b) After applying dropout.







# How to tune hyperparameter

## ■ 주요 파라미터

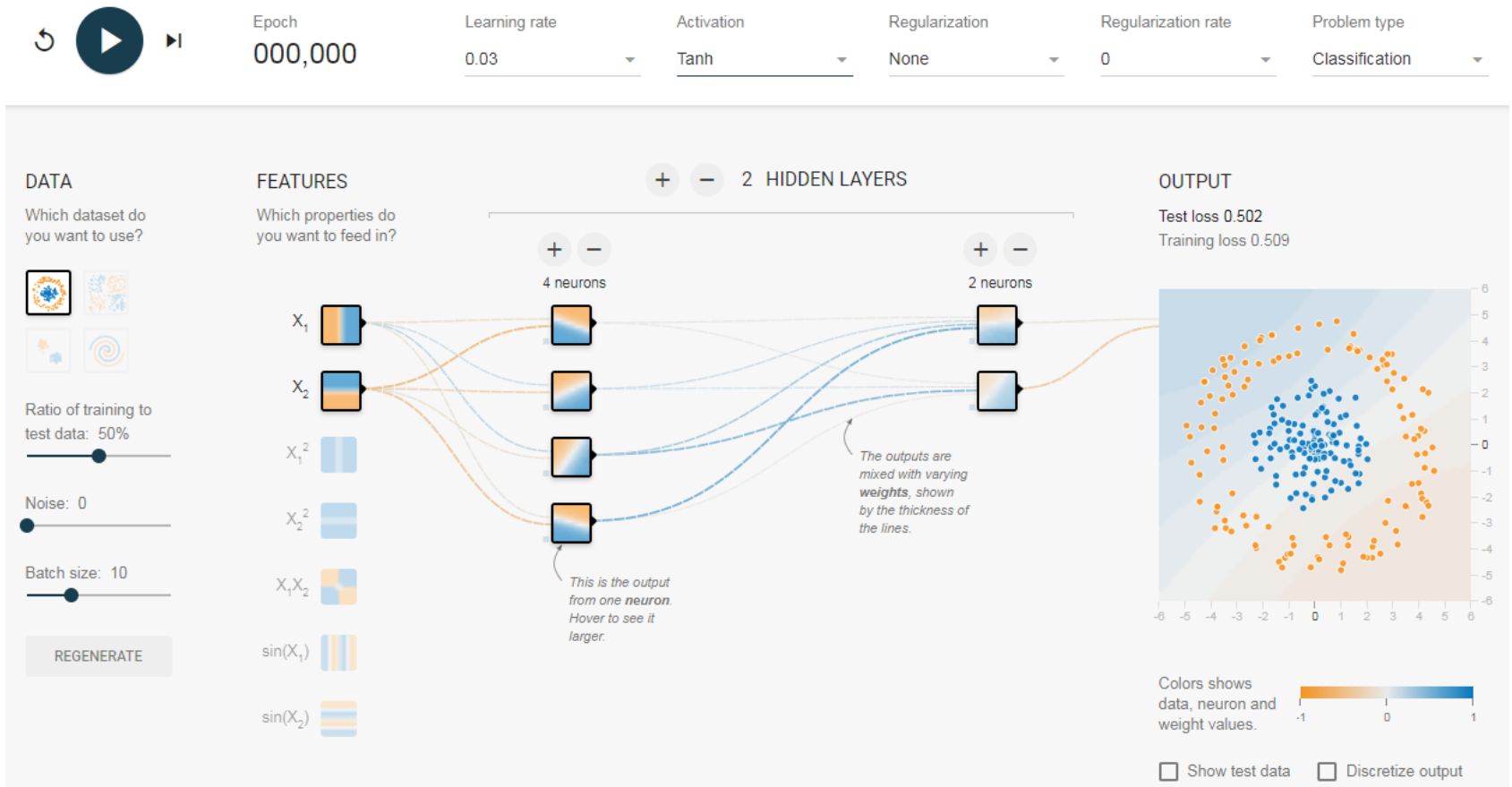
- 은닉층과 은닉노드의 개수
- 활성화함수
- Regularization
- 최적화 기법
- 가중치 초기화 방식
- learning rate와 mini-batch size

## ■ 일반적인 방법

- 'Stretch pants' 접근법  
  - 충분히 과적합되어 원하는 성능이 나올만한 복잡한 모델을 만든 후 신경망 구조를 줄이거나 regularization을 강화하여 일반화 성능을 향상
  - Deep network이 shallow network 보다 파라미터 효율성이 좋음 - 훨씬 적은 수의 뉴런 사용
  - 최근에는 고려할 hyperparameter의 수를 줄이기 위해 모든 은닉층에서 같은 수의 은닉노드를 갖는 추세
- Random Search 활용
  - CV를 통해 적절한 hyperparameter를 찾고자 할 때 grid search 보다 random search가 더 효율적

# Neural Network Playground:

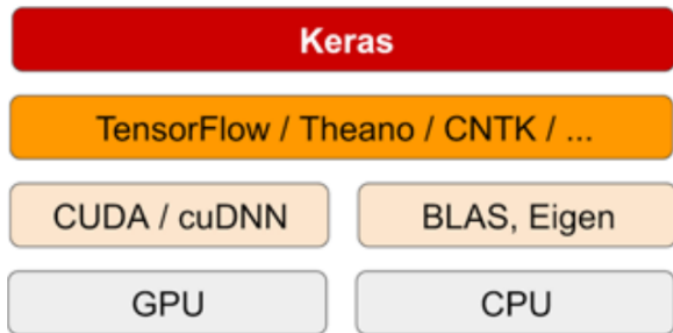
## An interactive tool for learning neural networks



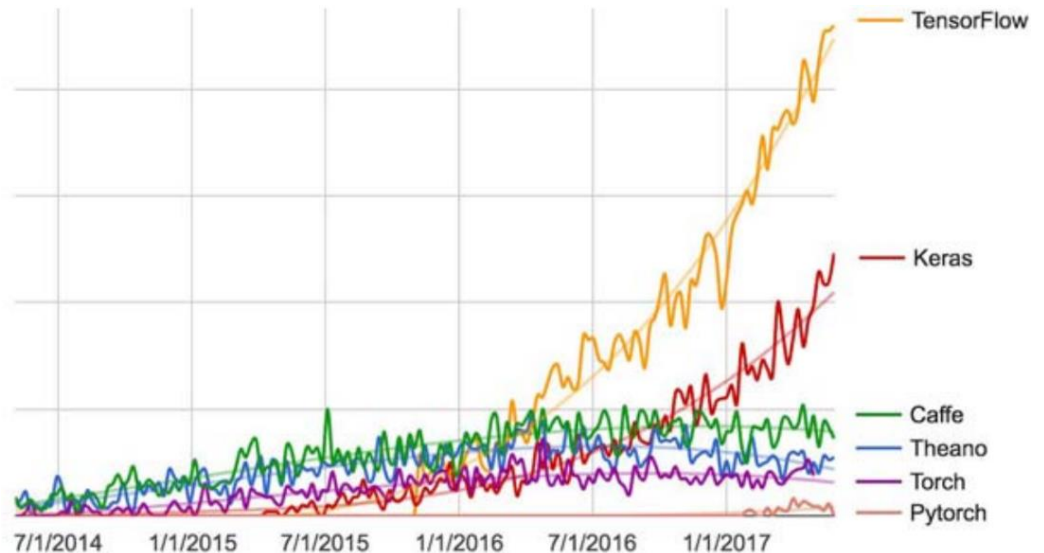


# Keras Framework

- What is Keras
  - Tensorflow, CNTK, Theano를 사용하기 편하게 만들어 놓은 high-level API
  - Documentation: <http://keras.io/>



Source: Deep Learning with Python by François Chollet



## Model building steps (Sequential API)

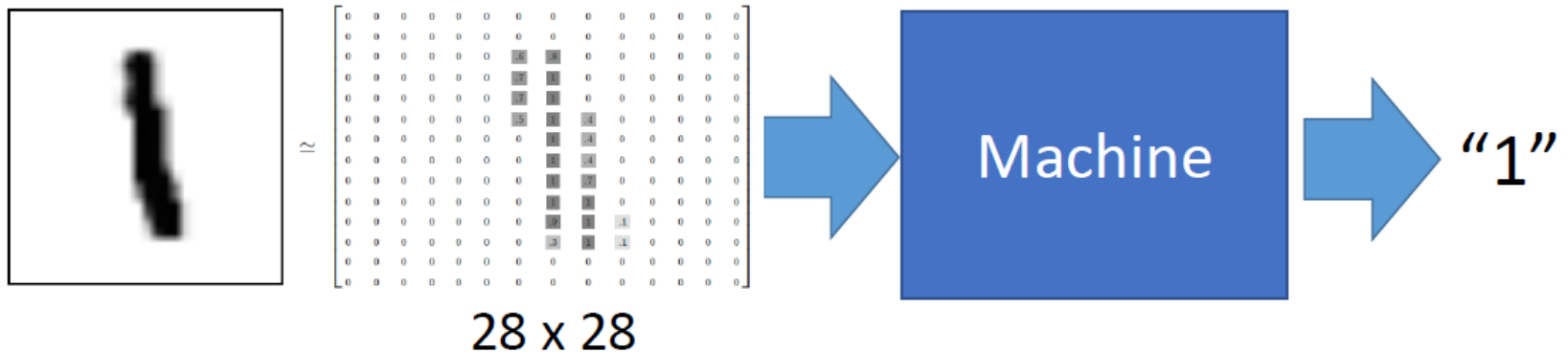
- Specify Architecture
  - `model = Sequential()`: 모형 생성
  - `model.add()`: 레이어 추가
- Compile
  - `model.compile()`: 목적함수 및 최적화 방법 지정
- Fit
  - `model.fit()`: 입력, 출력 데이터를 사용하여 가중치 계산
- Predict
  - `model.predict()`

## ■ Core layers

- `Dense()`
- `Activation()`
  - `softmax()`: multi-category output
  - `sigmoid()`: binary output
- `Dropout()`

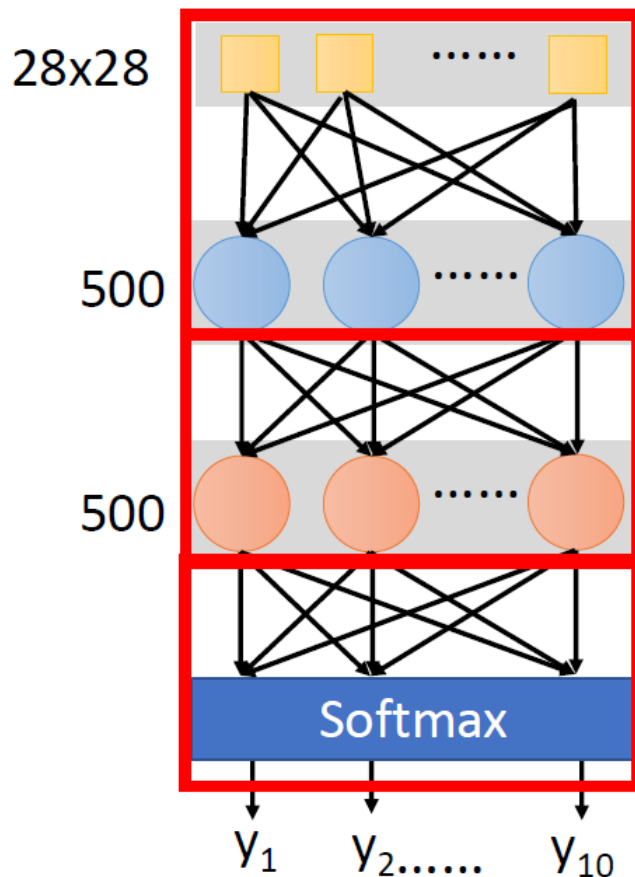
# Training DNN with keras

Application – Handwriting Digit Recognition



# Training DNN with keras

Step 1. define a model



```
model = Sequential()
```

```
model.add( Dense( input_dim=28*28,  
                  output_dim=500 ) )  
model.add( Activation( 'sigmoid' ) )
```

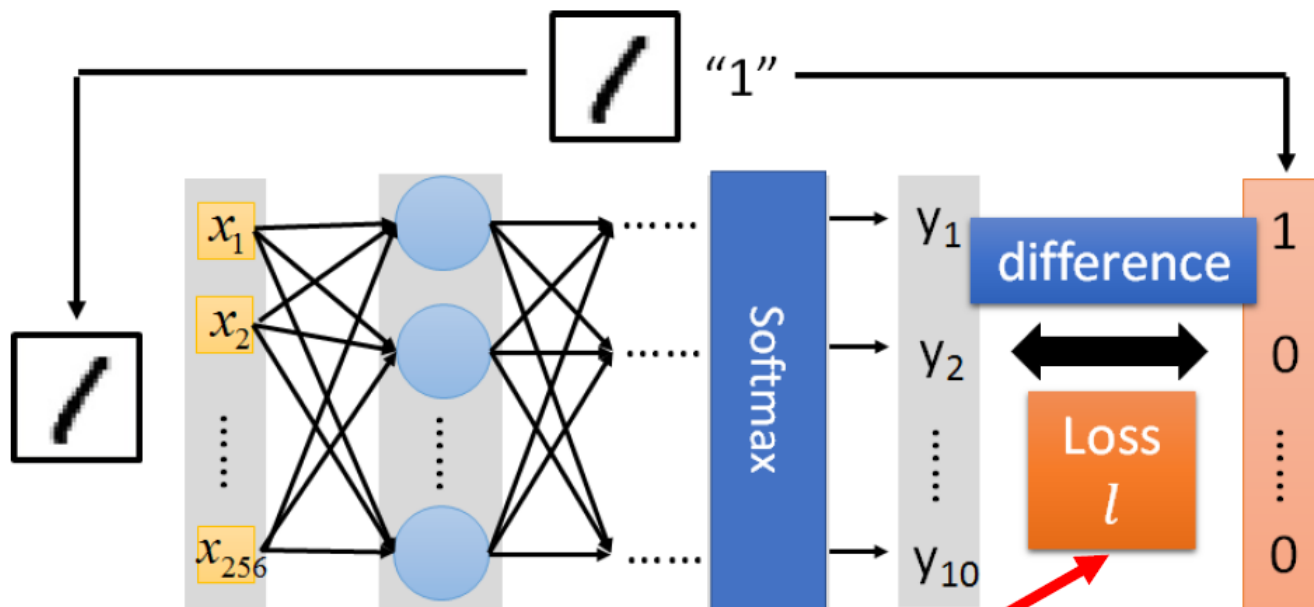
softplus, softsign, relu, tanh,  
hard\_sigmoid, linear

```
model.add( Dense( output_dim=500 ) )  
model.add( Activation( 'sigmoid' ) )
```

```
model.add( Dense( output_dim=10 ) )  
model.add( Activation( 'softmax' ) )
```

# Training DNN with keras

Step 2. configure the learning process

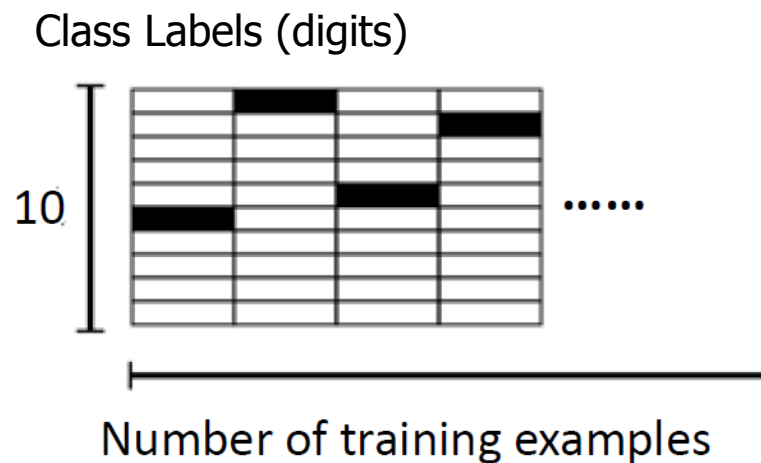
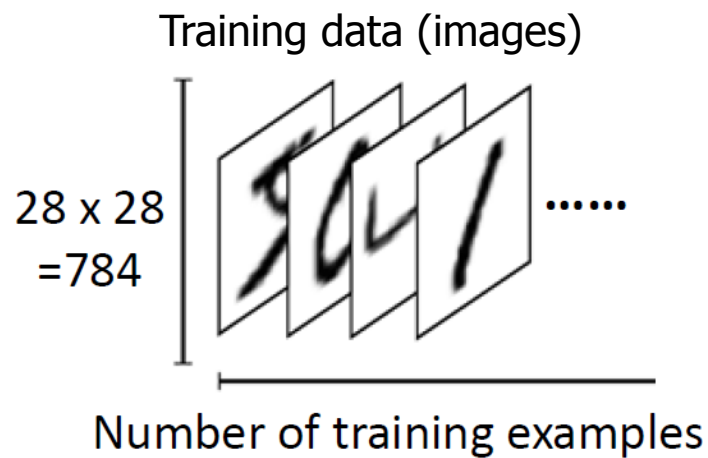


```
model.compile(loss='categorical_crossentropy',  
              optimizer='adam',  
              metrics=['accuracy'])
```

# Training DNN with keras

Step 3. iterate on the training data

```
model.fit(x_train, y_train, batch_size=100, nb_epoch=20)
```





# Training DNN with keras

---

Step 4. evaluate & use the model

```
score = model.evaluate(x_test,y_test)
print('Total loss on Testing Set:', score[0])
print('Accuracy of Testing Set:', score[1])
```

```
result = model.predict(x_test)
```