SESSION #11

인공신경망

By TEAM1

목차 소개

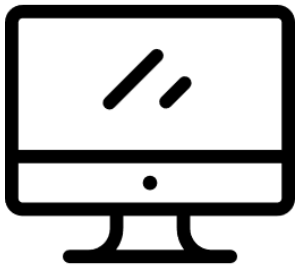
Chapter 1.1 인간에게는 쉽고 기계에게는 어려운

애벌레 or 무당벌레?

1231 x 13 x 515 + 12 =

Chapter 1.1 인간에게는 쉽고 기계에게는 어려운

질문 → 생각 → 대답

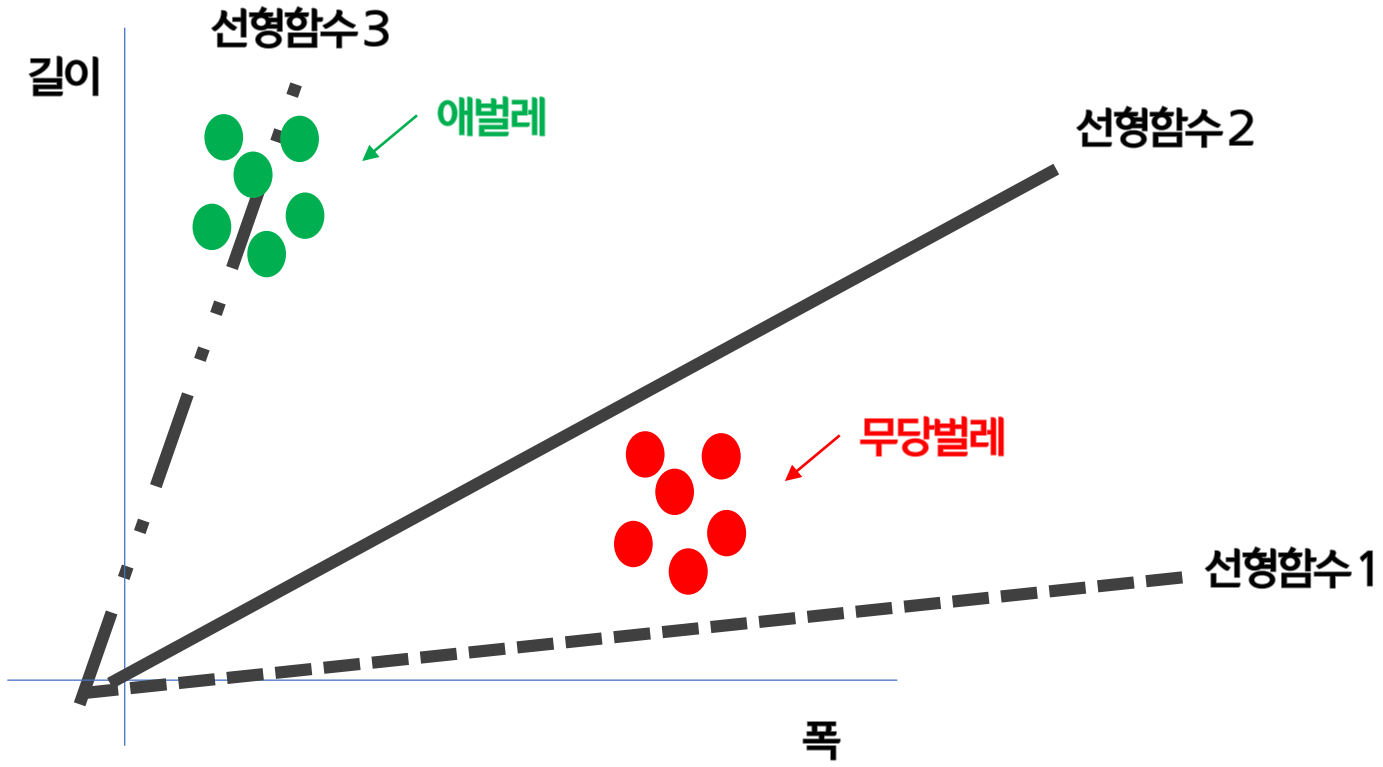


입력 → 처리 (연산) → 출력

↳ 인공신경망도 마찬가지

Chapter 1.2 애벌레와 무당벌레를 분류하는 컴퓨터

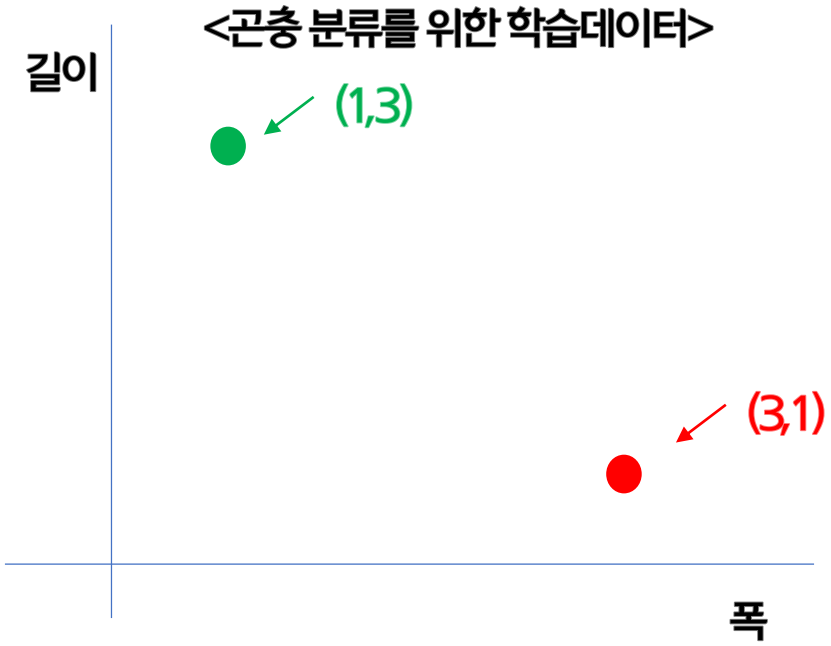
애벌레와 무당벌레를 분류하라는 명령을 컴퓨터에게 한다면?



Chapter 1.3 선형함수 학습시키기

선형함수 학습? 직선의 기울기만 결정하면 된다!

문제 풀고

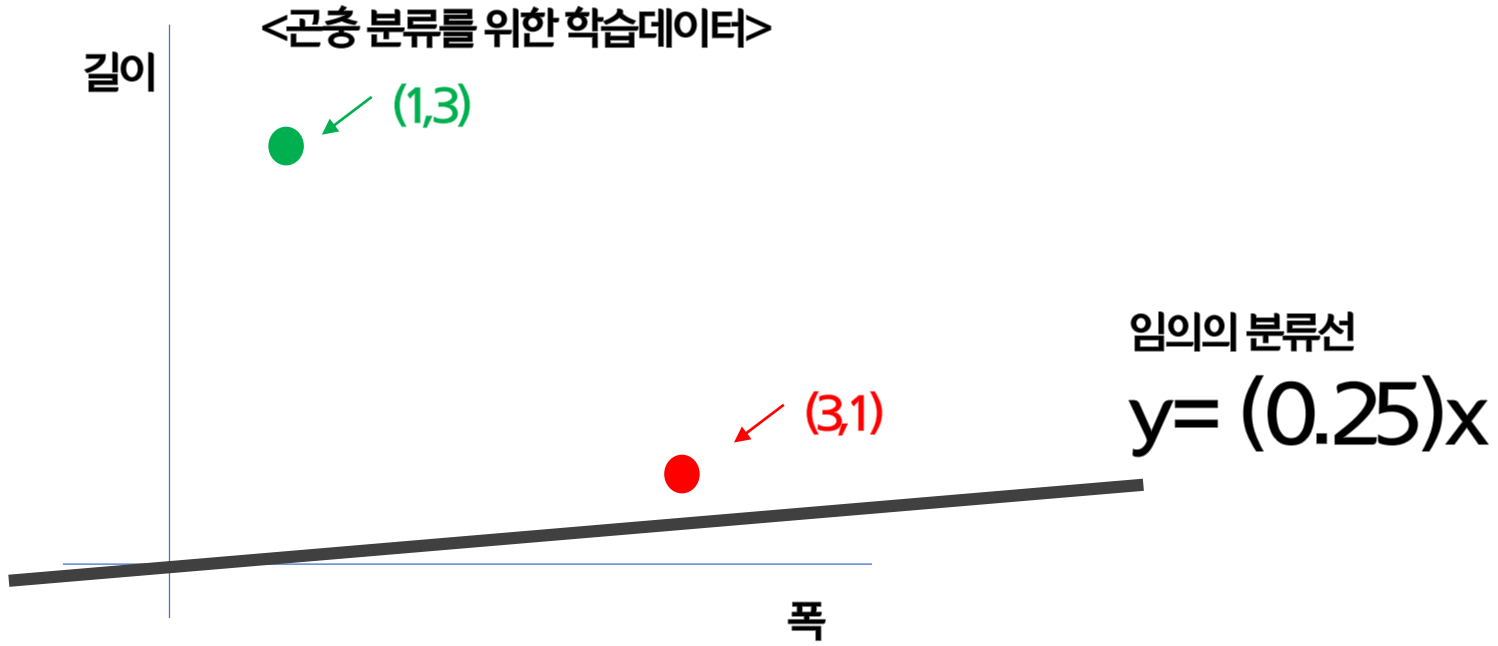


학습목표 -> (3, 1.1)을 지나면 좋겠다.

Chapter 1.3 선형함수 학습시키기

임의의 분류선을 정한 후 오차를 구한다.

채점 하고



(오차)E = (목표값)t − (실제값)y

Chapter 1.3 선형함수 학습시키기

$$y = Ax$$

$$t = (A + \Delta A)x$$

$$E = t - y$$

$$\Delta A = E / x$$

목표값을 t라고 표기하고
t를 얻기 위해서는 우리는 A를 조금씩 조정해 나가야 한다.
수학자들은 작은 변화를 나타낼 때 Δ (델타) 라는 기호를 사용한다.

오차 E와 ΔA가 매우 간단한 관계에 있다.

Chapter 1.3 선형함수 학습시키기

① 기울기 A에 임의의 값 0.25 대입 (x=3 일때 y=0.75)

② 목표값과의 오차를 계산한다.

X가 3 일 때 Y의 값은 1.1이어야 하므로,

오차 = 1.1 − 0.75 = 0.35

Chapter 1.3 선형함수 학습시키기

③ 오차 E와 ΔA의 관계를 활용한다. (ΔA = E/x)
0.35 / 3 = 0.1167

④ A를 0.1167 만큼 업데이트 해야 한다.
y= (0.25 + 0.1167) x

Chapter 1.3 선형함수 학습시키기

④ 이제 두 번째 학습데이터를 학습 시킨다.(x=1, y=3)

A=3.667로 업데이트한 선형함수에 대입하면
목표값 t를 이번엔 2.9로 잡고
$E = 2.9 - 0.3667 = 2.533$
$\Delta A = E / x = 2.533$
A는 2.9로 업데이트가 된다.

⑤ 문제점 : 그냥 마지막 학습 데이터에 맞춰지네?

Chapter 1.3 선형함수 학습시키기

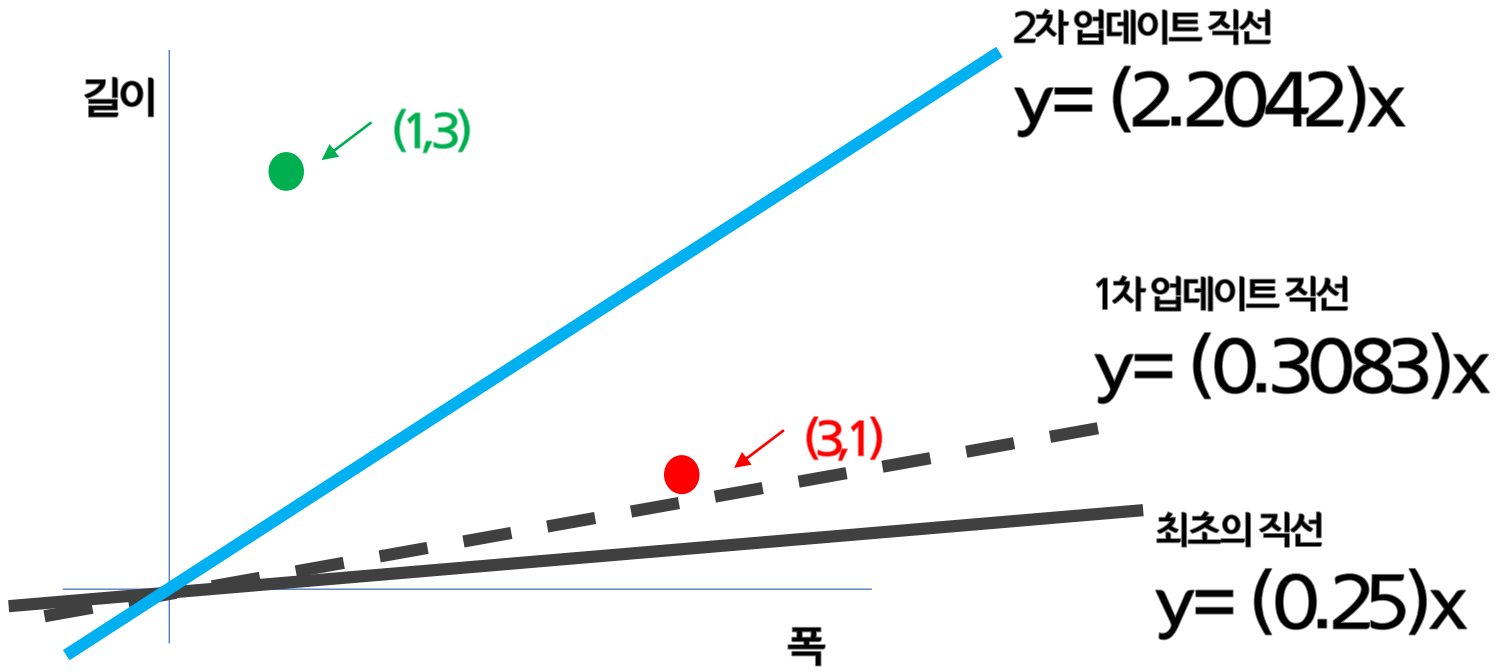
아 그럼 조금씩 조정(moderate)을 하자!

새로운 A값으로 바로 점프하는 것이 아니라
ΔA의 일부만 업데이트를 한다.

학습데이터가 제시하는 방향으로 조금씩 움직인다.
ΔA = L(E/x) (L은 학습률)

Chapter 1.3 선형함수 학습시키기

(학습률)L = 0.5 일 때…



오차를 계산의 반복을 통해서 C값을 조금씩 조정해 나가는 법,
이것이 인공신경망을 학습시키는 핵심과정

Chapter 1.3 선형함수 학습시키기

Chapter 1 정리

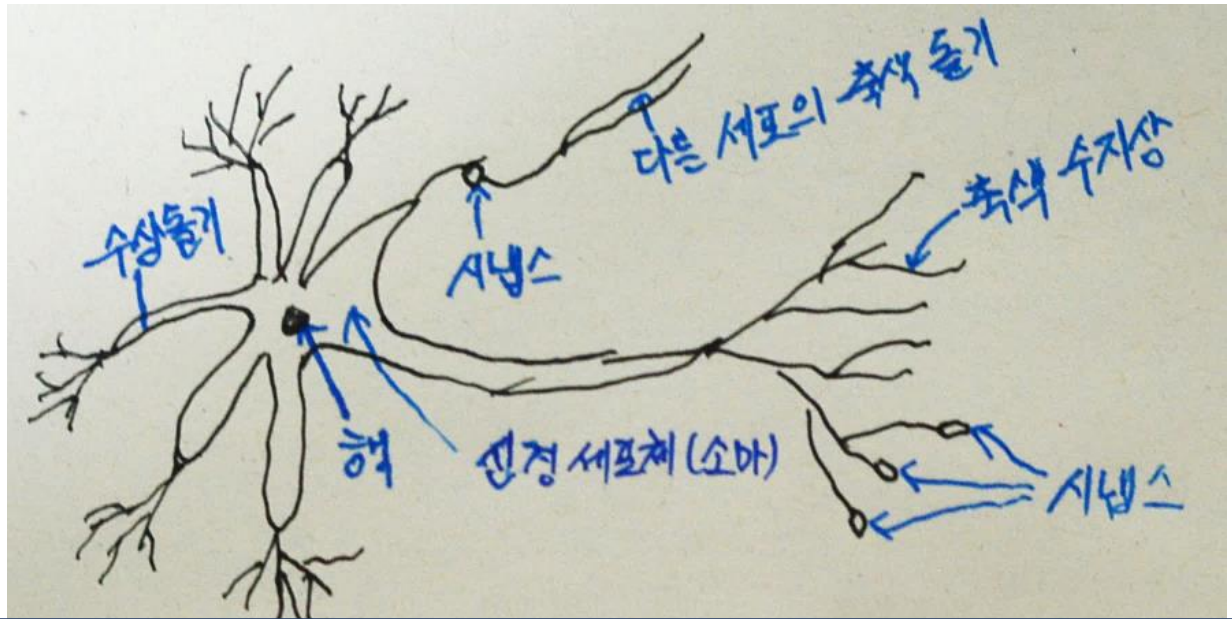
우리는 무당벌레와 애벌레를 분류하라는 명령을 했다.
가장 먼저 선형함수를 정의하고
기울기에 임의의 값을 부여하고 오차를 바탕으로 이를 업데이트해 나갔다.

오차와 기울기 간의 관계를 바탕으로 오차를 제거하기 위해
얼마만큼 기울기를 조정해야 하는지 알 수 있었다.
그러나 문제점은 이전의 학습 데이터는 무시하고 최종학습 데이터에만 맞춰 업데이트 된다는 것

이를 해결하기 위해 학습률을 도입해 업데이트의 정도를 조정
이를 통해 단일 학습 데이터가 학습에 지배적인 영향을 주는 것을 방지 할 수 있다.

Chapter 2
인공신경망 필수 개념

Chapter 2.1 대자연의 컴퓨터, 뉴런



뉴런이란? 한쪽 끝에서 다른 쪽 끝으로 전기 신호를 전송한다.

1. 무수히 상호 연결 되어있고

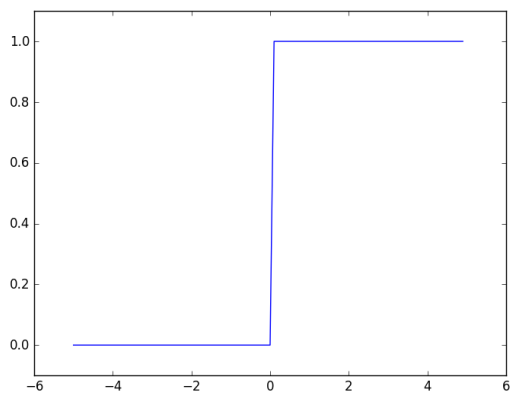
입력값이 2. 어떤 분계점에 도달해야 출력이 발생한다.

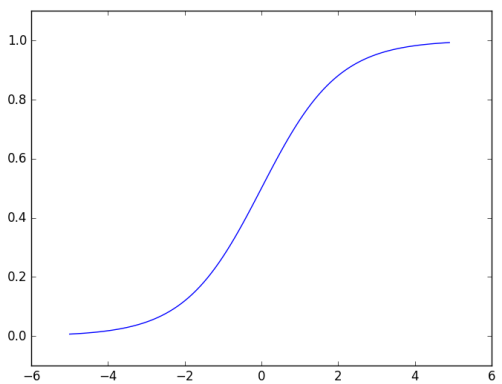
Chapter 2.1 대자연의 컴퓨터, 뉴런

어떤 분계점을 넘어야 반응한다. (활성화함수)

입력 신호를 받아 특정 분계점을 넘어서는 경우에 출력 신호를 생성해 주는 함수를 활성화 함수라고 한다.

EX) 계단함수, 시그모이드 함수

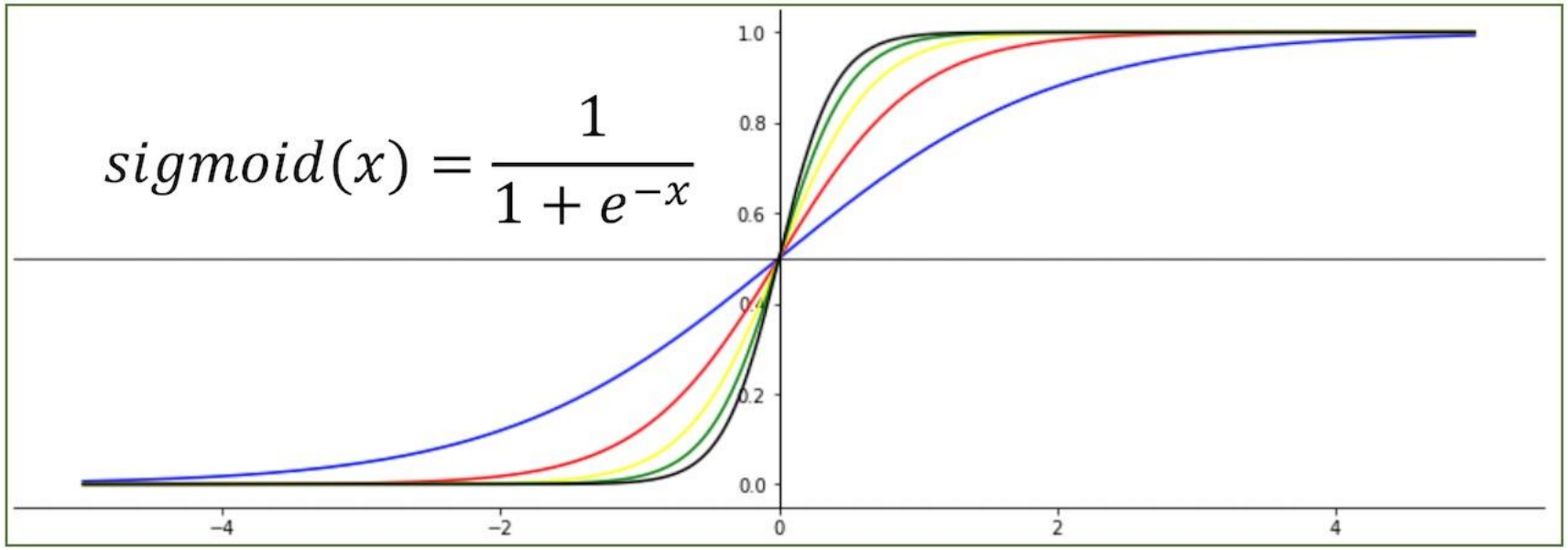




Chapter 2.1 대자연의 컴퓨터, 뉴런

시그모이드 함수를 사용하겠다. (계산이 편리하다.)

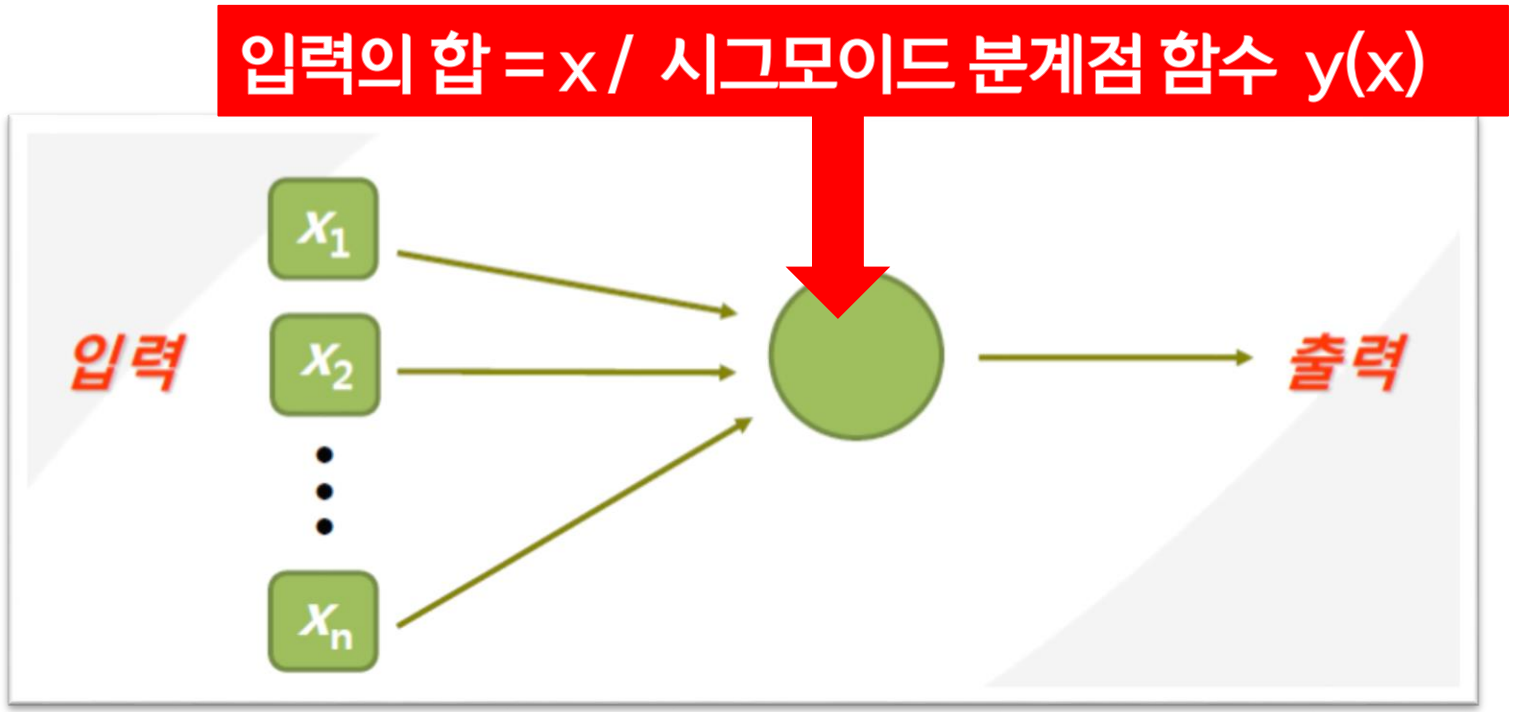
$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$



Chapter 2.1 대자연의 컴퓨터, 뉴런

(입력 –> 연산 –> 출력) 인공신경망ver.

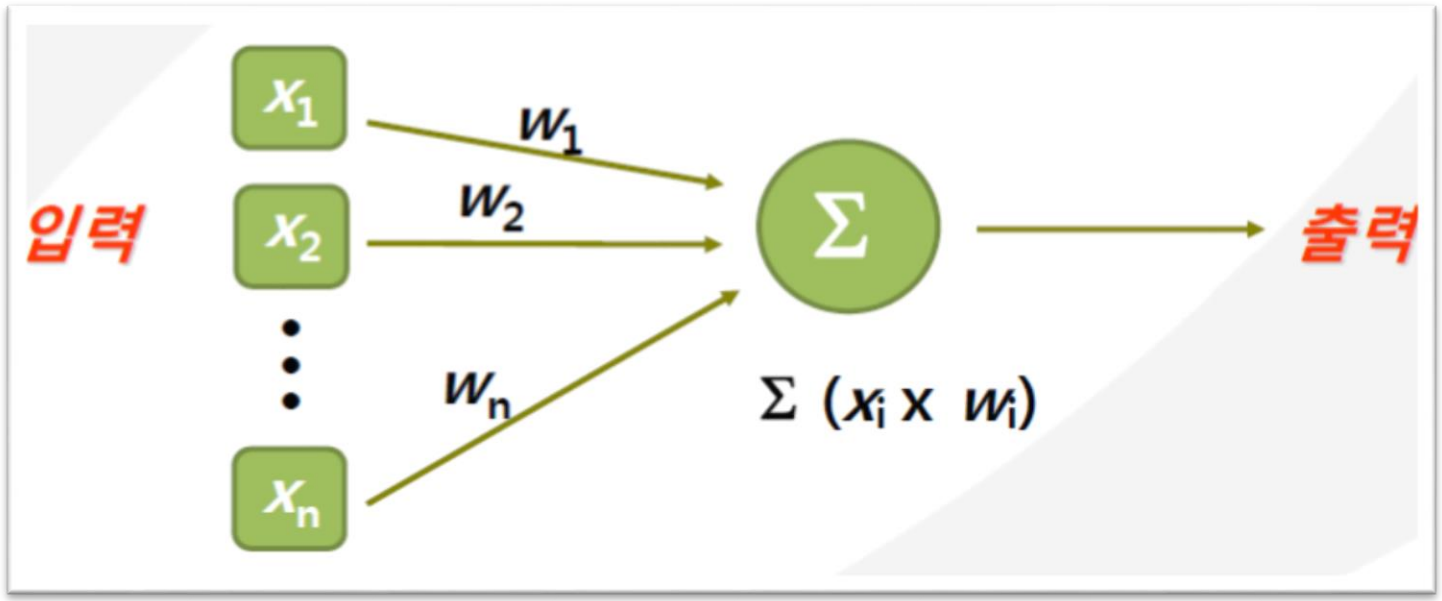
입력 신호를 측정
총 입력신호를 합산
출력 신호를 결정



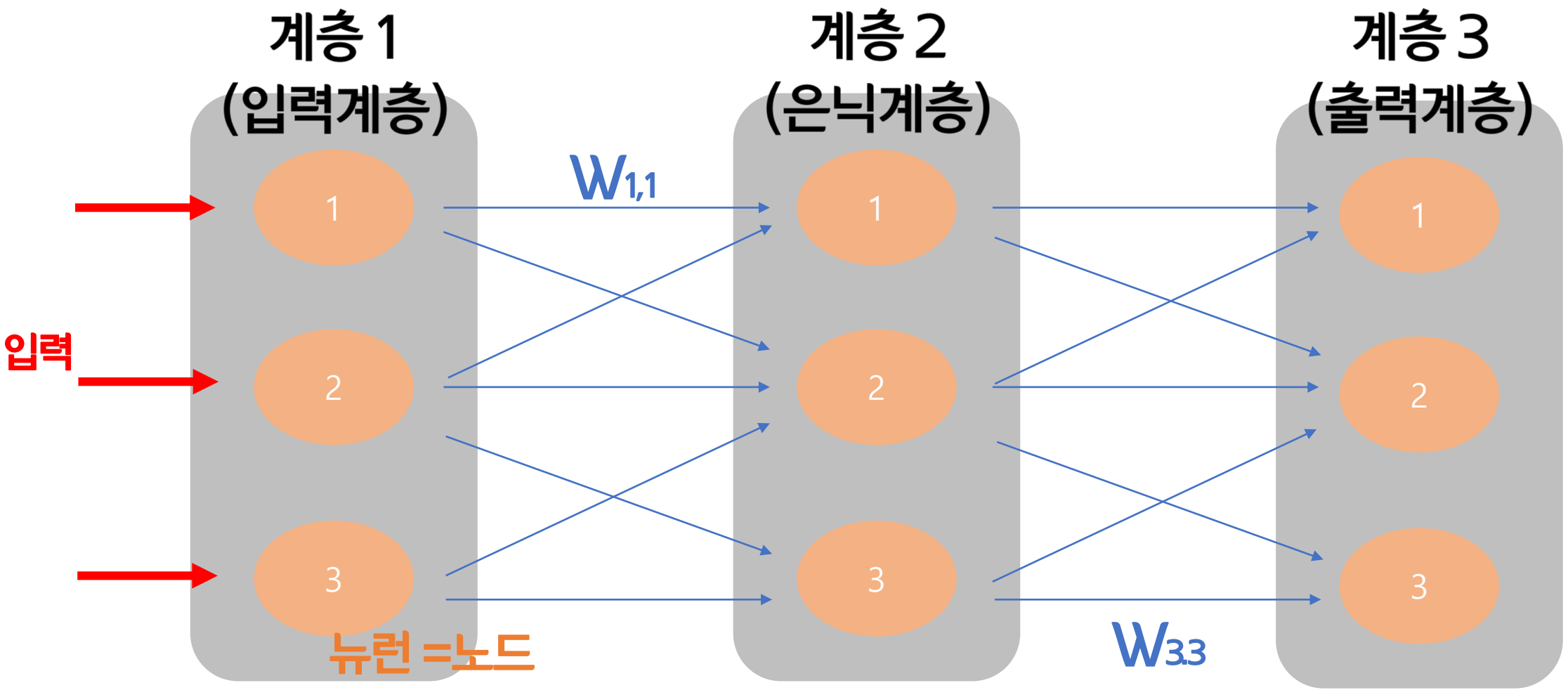
Chapter 2.1 대자연의 컴퓨터, 뉴런

(입력 –> 연산 –> 출력) 인공신경망ver.

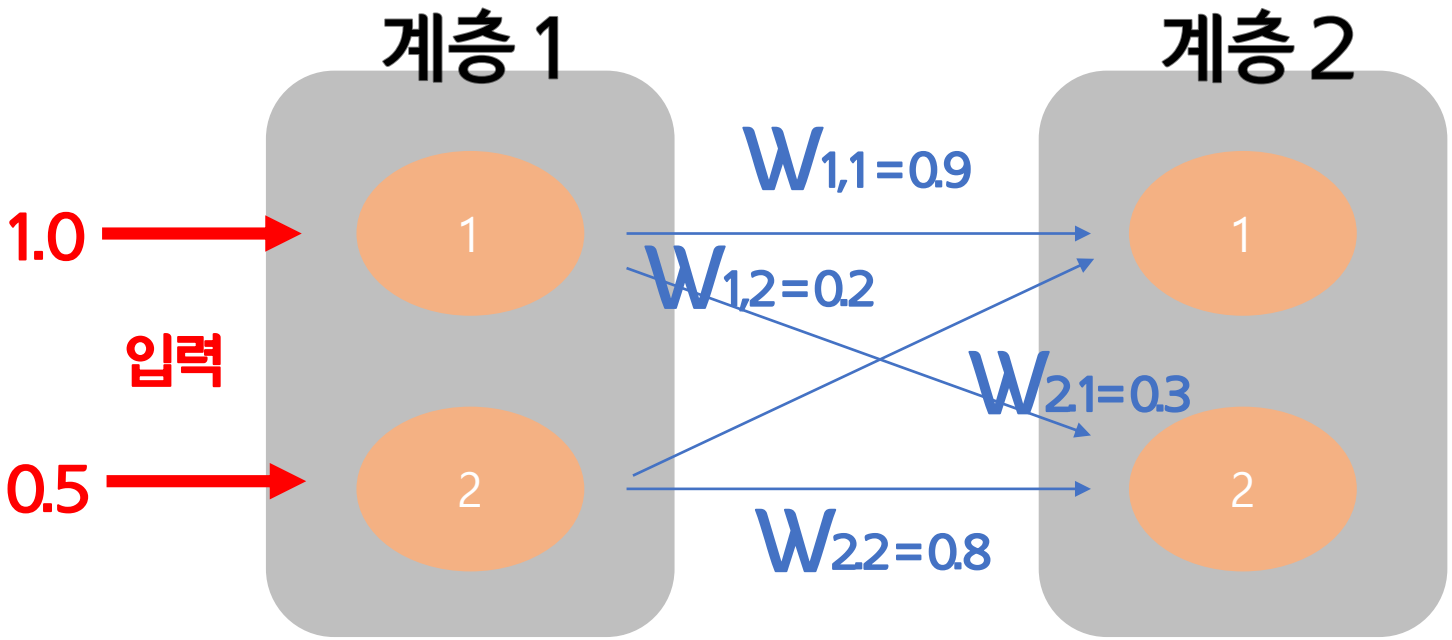
연결강도 가중치 (weight)
–입력 신호의 강도를 표현
–총 입력값 = X1*W1 + …… + Xn*Wn



Chapter 2.1 대자연의 컴퓨터, 뉴런

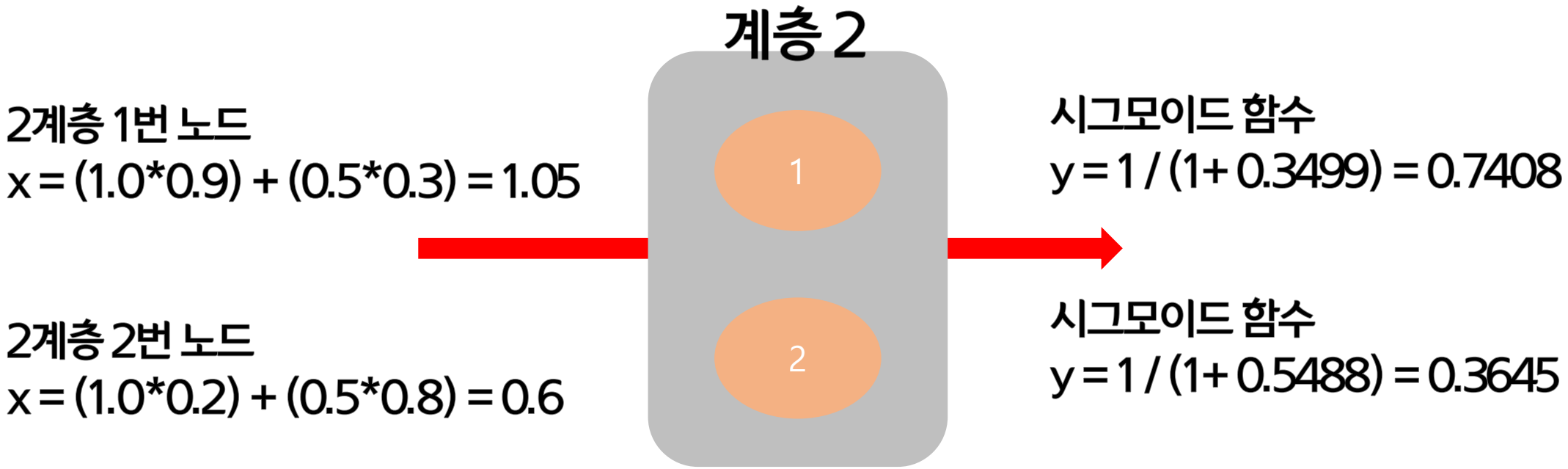


Chapter 2.2 신경망 내의 신호 따라가기



앞서 오차와의 계산을 통해 기울기를 업데이트했다.
인공신경망에서 업데이트 대상은 가중치

Chapter 2.2 신경망 내의 신호 따라가기



Chapter 2.2 신경망 내의 신호 따라가기

계산의 편의를 위해 행렬곱을 쓰자!

$$X = W*I$$
$$O = sigmoid(X)$$

I = input 행렬
O = output 행렬
W = 가중치 행렬

Chapter 2.3 가중치 학습하기

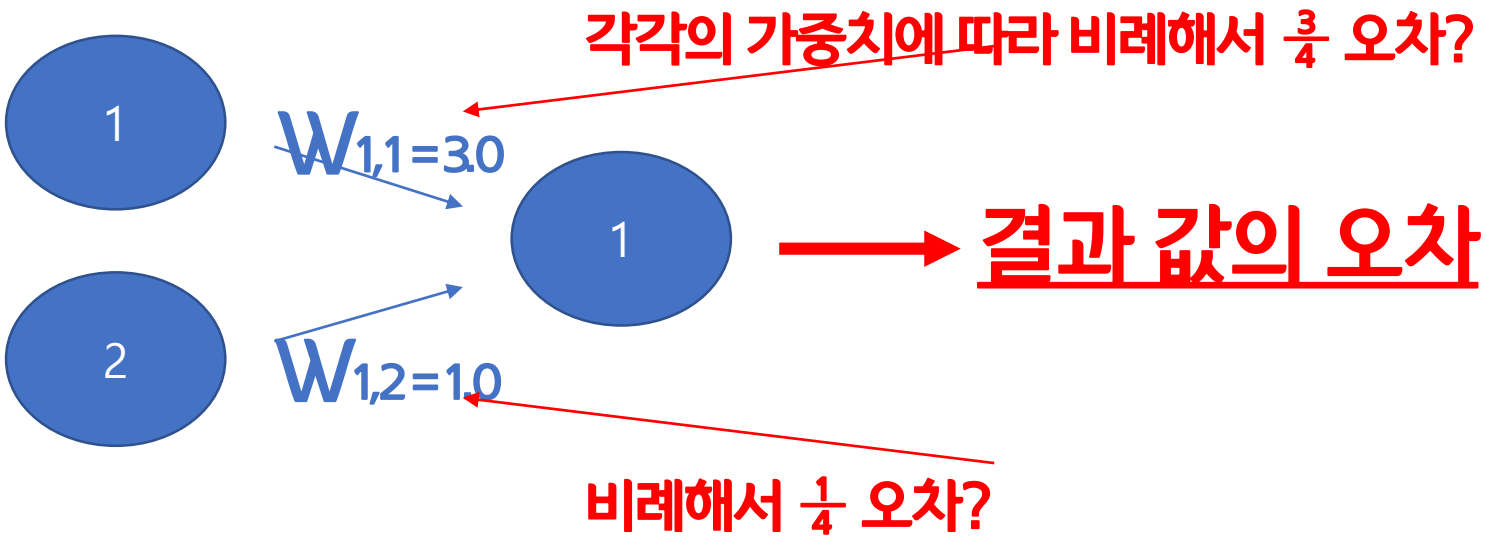
잠깐! 정리

앞서 기울기의 값을 조정해감으로써 선형 선형함수를 업데이트 했다.
우리는 이 오차에 기반을 두고 선형 선형함수를 정교해 나갔다.

그렇다면 여러 개의 노드가 결과 값과 오차에 영향을 주는 경우에는
가중치를 어떻게 학습시켜야 할까?

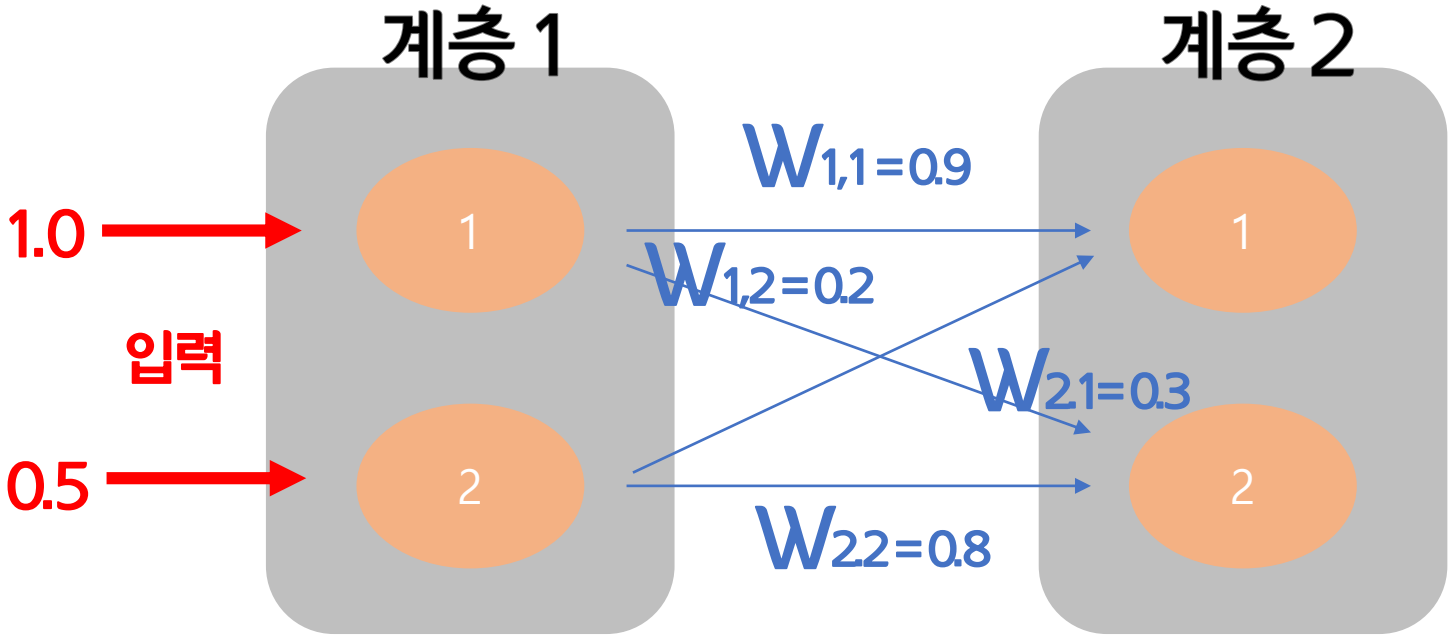
Chapter 2.3 가중치 학습하기

각각의 가중치에 비례해서 가중치를 학습?



오차를 하나의 계층에서 직전의 계층으로,
즉 역으로 전파하는 데에도 가중치를 이용한다. 그것이 바로 **역전파**

2.3 신경망의 학습: 가중치 업데이트



지금까지 오차와의 계산을 통해 무언가를 업데이트했다.
업데이트 대상은 가중치

Chapter 3
Backpropagation

3.1 역전파란?

- 역전파 알고리즘의 등장 배경:

단층 신경망으로 해결되지 않는 수많은 Non-Linear Questions

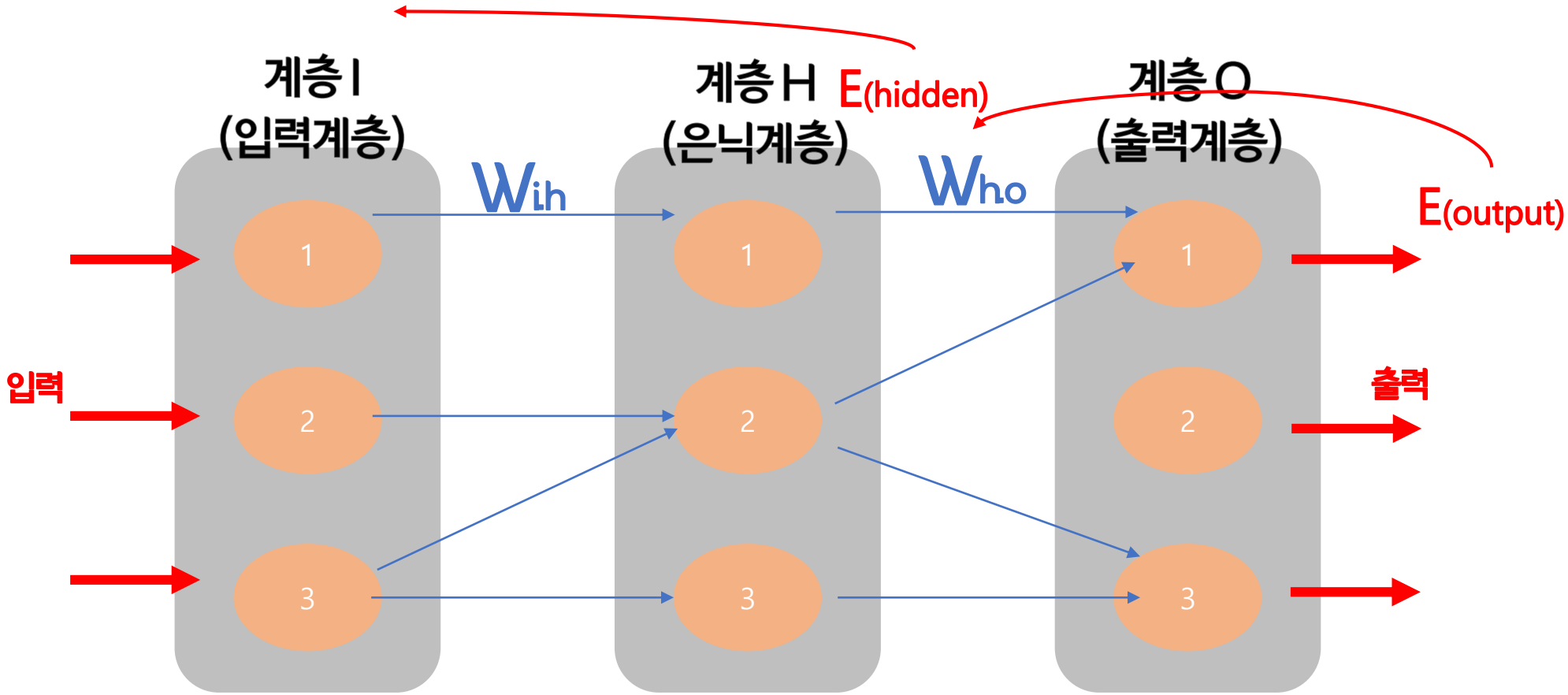
그러나 은닉층의 오차를 어떻게 정의할 것인가??

3.1 역전파란?

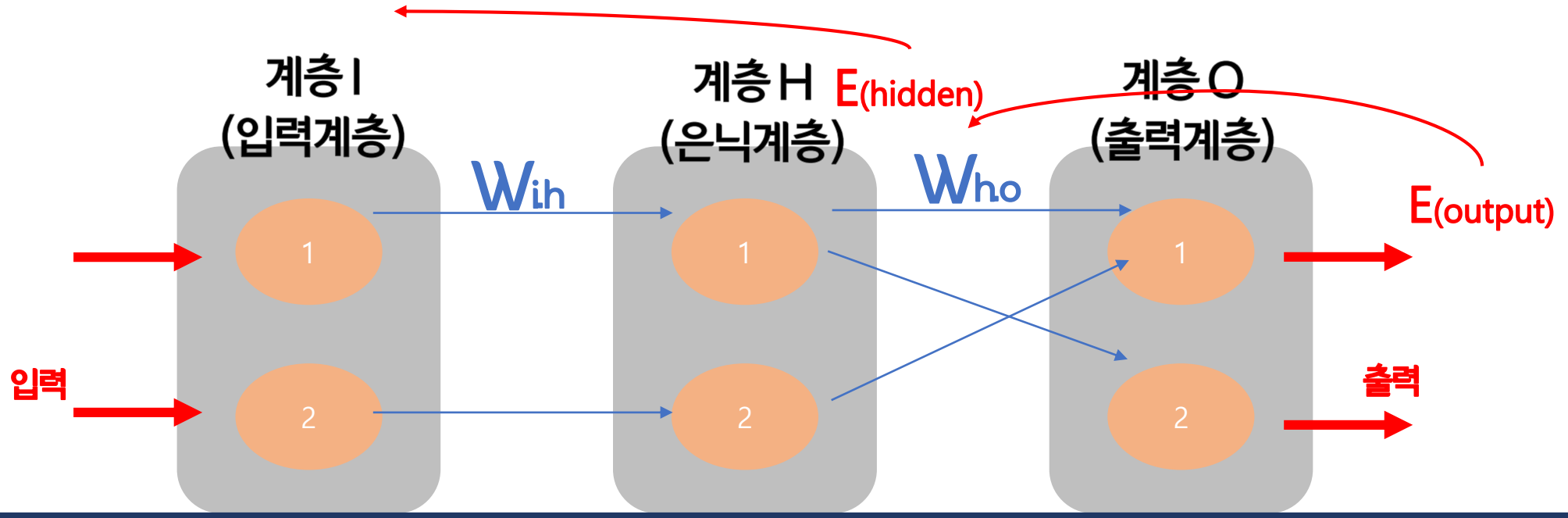
인공 신경망에서 학습 = 가중치를 업데이트 하는 과정
가중치의 업데이트 = 오차의 수정

-> BUT 중간계층에 존재하는 노드들의 오차는 명백하지 않다.
=>대안 : 출력계층의 노드들의 오차를 이와 연결된 가중치의 크기에 비례해 역전파 하여 계산한다.

역전파: Intuition

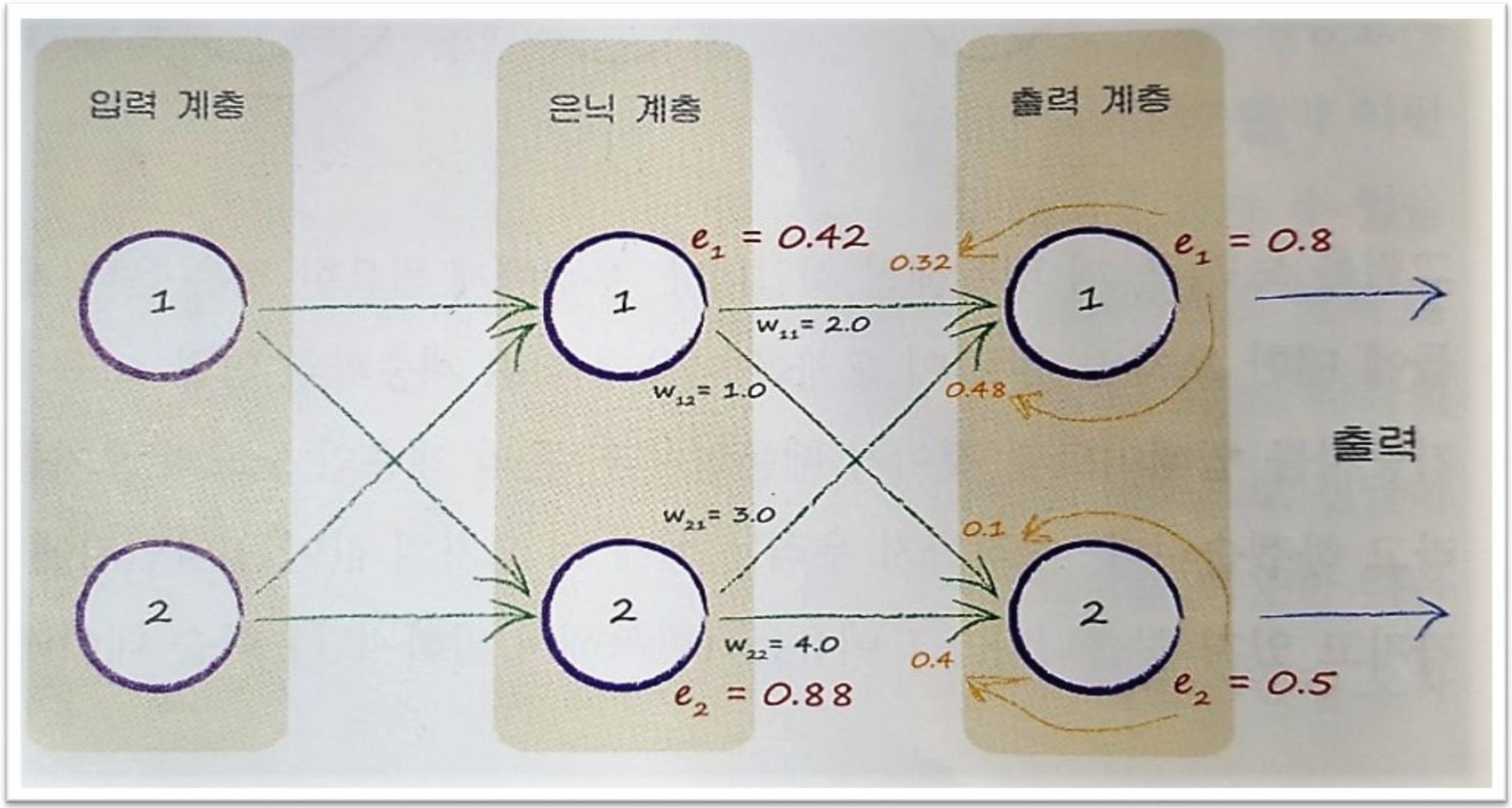


역전파: Intuition

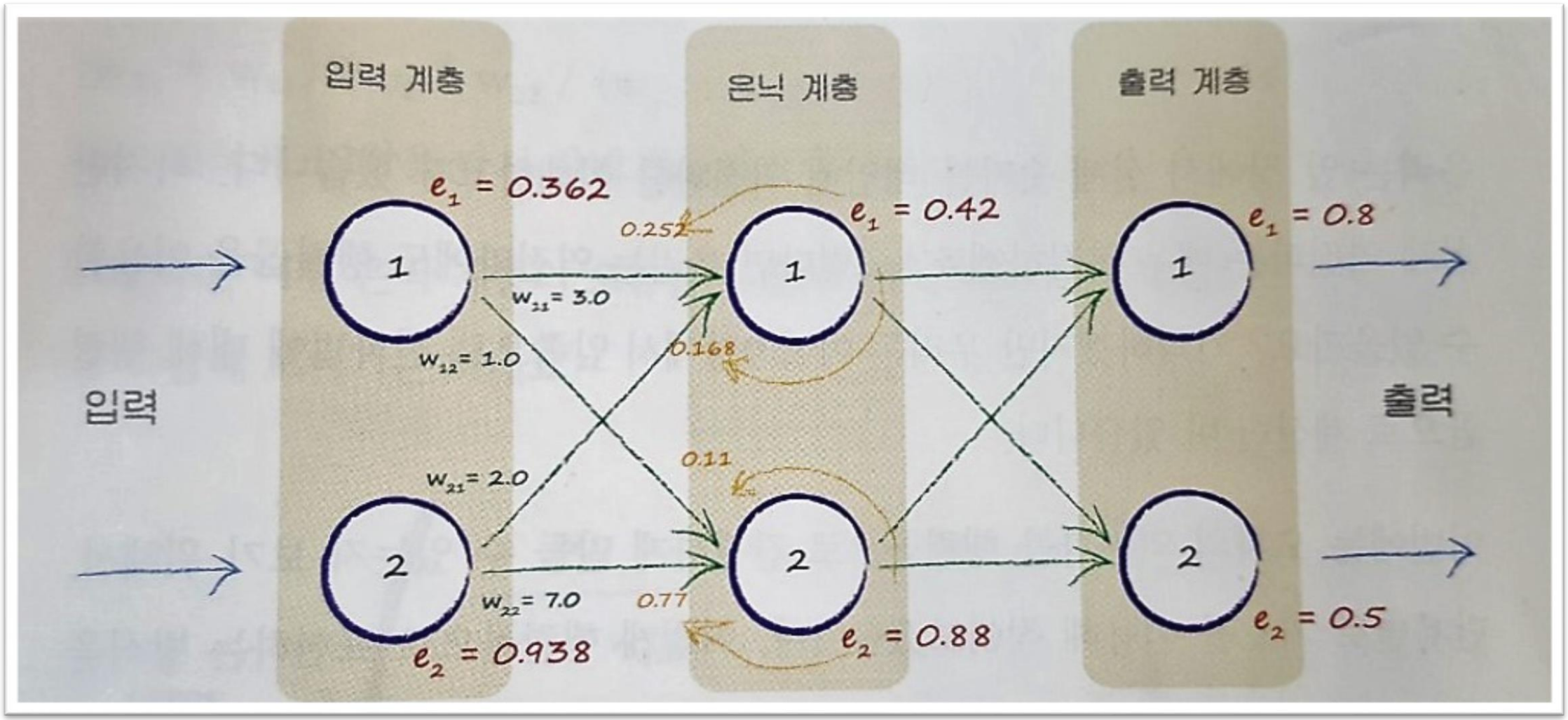


우리가 아는 것은 오직 출력값의 오차!
은닉계층의 오차는 구할 수 없다
=> 출력값의 오차를 가중치에 비례해서 뒤로 보내며 계산: 역전파

역전파: Intuition



역전파: Intuition



역전파: Intuition

그래서! 가중치를 어떻게 업데이트 할 건데?

대수학으로 하기에는 계산이 너무 복잡

그렇다면? **경사하강법!**

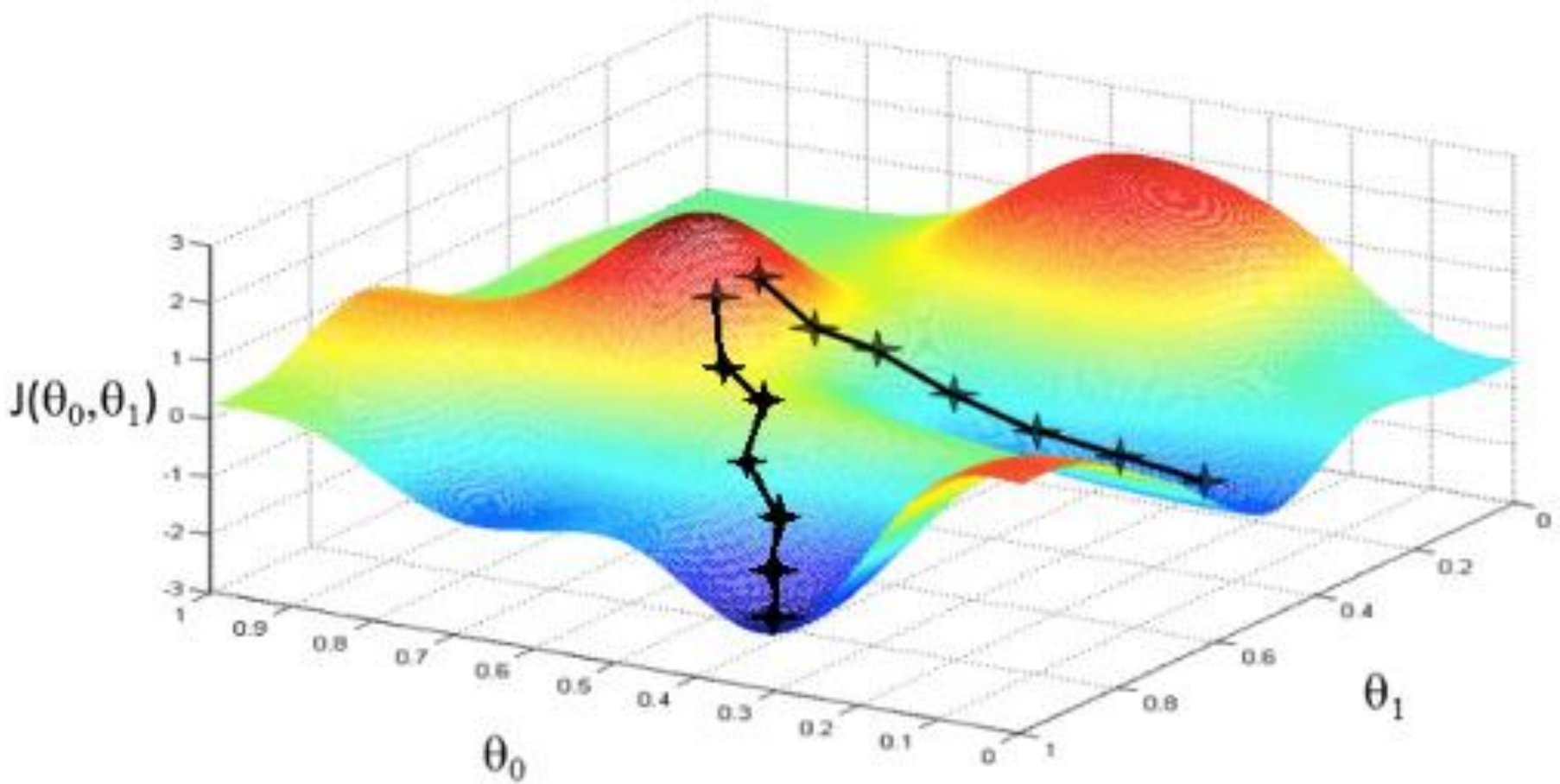
Recap : Gradient Descent

Optimization(최적화)

- 모든 Machine Learning 알고리즘의 목표
- 비용함수를 이용해서 모델을 학습하는 방법
- 비용함수의 결과를 최소화하는 모델의 인자를 찾는 것

예) 경사하강법, 뉴턴/준뉴턴 방법…무궁무진

Recap : Gradient Descent



Recap : Gradient Descent

3 Different Methods

$$J(W,b) = \frac{1}{m}\sum_{z=0}^{m} J(W,b,x^{(z)},y^{(z)})$$

- Batch

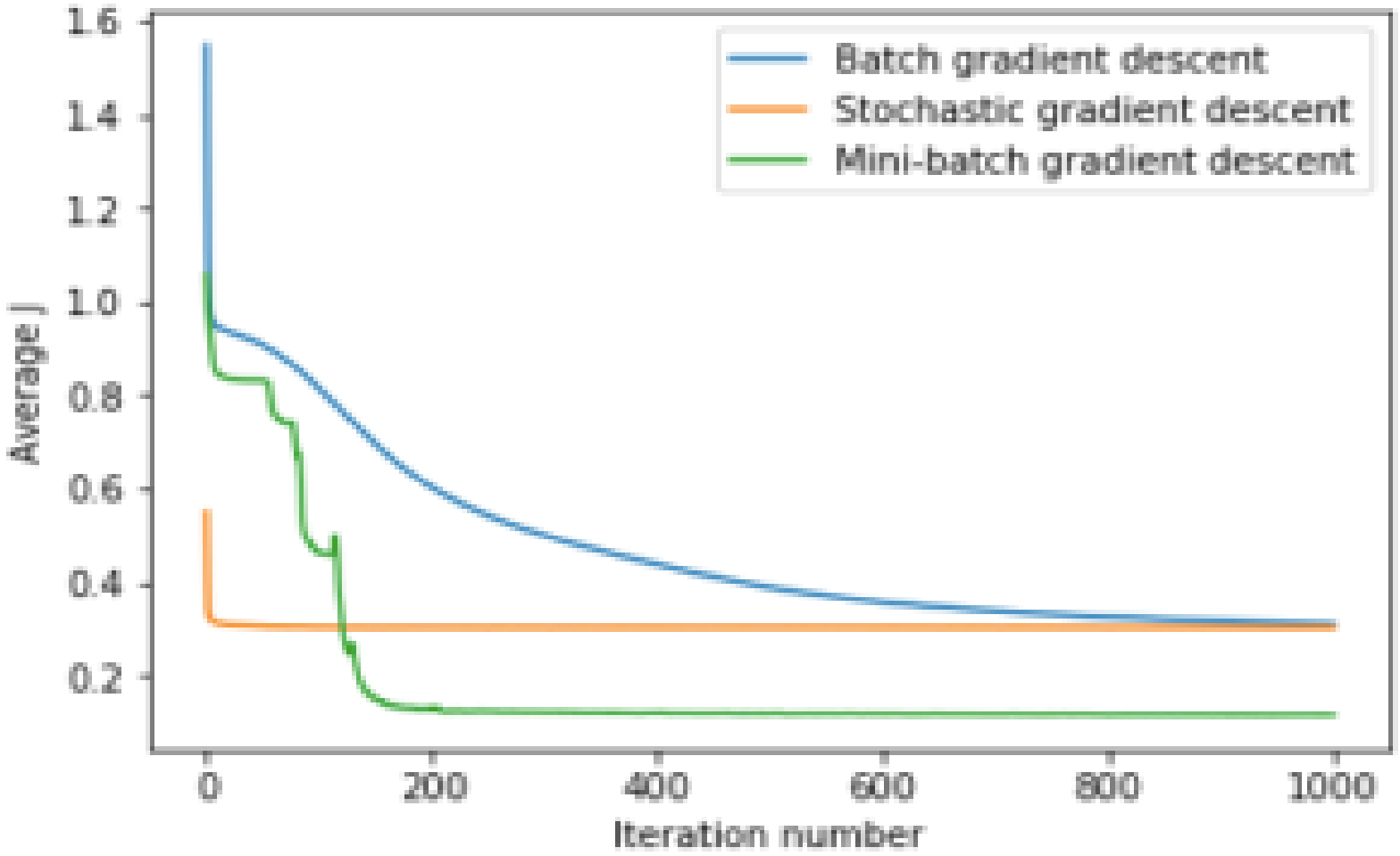
$$J(W,b,x^{(z:z+bs)},y^{(z:z+bs)}) = \frac{1}{bs}\sum_{z=0}^{bs} J(W,b,x^{(z)},y^{(z)})$$

- Mini-batch

$$W = W - \alpha \nabla J(W,b,x^{(z)},y^{(z)})$$

- Stochastic

Batch vs Stochastic vs Mini-batch



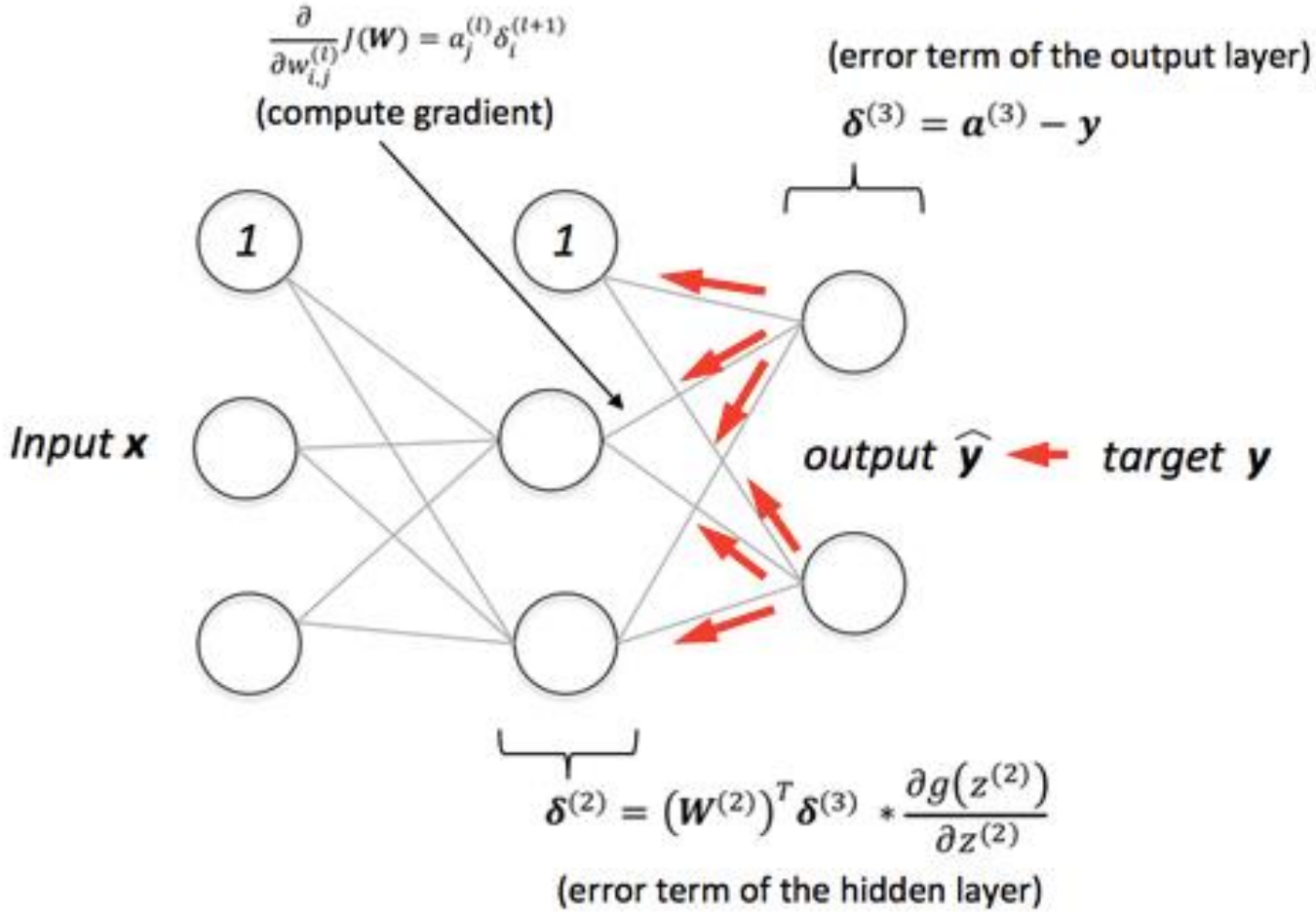
3.2 역전파 알고리즘(원리)

1. 훈련 데이터(Training Data)를 Input으로 넣어서 전방향 연산(Forward propagate)을 수행한다.

2. 이 결과로 나온 신경망 예측값(Prediction)과 실제값(목표값) 차이인 오류(Error)를 계산한다.

3. 오류(Error)를 신경망 각각의 노드에 역전파(Backpropagate)한다.

3.2 역전파 알고리즘(원리)



3.3 실전 알고리즘: 기초

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\partial C}{\partial w^l_{j\,k}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

3.3 실전 알고리즘: 과정

1. **Input** x: Set the corresponding activation a^1 for the input layer.

2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

3. **Output error** δ^L: Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

3.3 실전 알고리즘: 가중치 업데이트

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = W_{ij}^{(l)} - \alpha \cdot a_j^{(l)} \delta_i^{(l+1)}$$

$$b_{ij}^{(l)} = b_{ij}^{(l)} - \alpha \frac{\partial}{\partial b_{ij}^{(l)}} J(W, b; x, y) = b_{ij}^{(l)} - \alpha \cdot \delta_i^{(l+1)}$$

Chapter 3 정리

신경망을 개선한다는 것은 **가중치의 변화를 통해 오차를 줄인다는 뜻이다.**
최적의 가중치를 직접 찾는 것은 매우 어렵고 이를 대체하는 접근 방법은
작은 발걸음으로 오차 함수를 줄여가면서 반복적으로 가중치를 개선해 나가는 방법

각 발걸음은 현재 위치에서 볼 때 가장 급격히 낮아지는 경사의 방향으로 취해진다.

오차의 기울기는 미분을 이용해 계산한다.

Chapter 4
코딩

4.1 신경망 building

```
from random import seed, randrange, random
from csv import reader
from math import exp, log


#Single-layered NN

# Initialize
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    #list of weights for each layer

    hidden_layer = [{'weights':[random() for i in range(n_inputs +1)]} for i in range(n_hidden)]
    #for each nodes(nuerons) in hidden layer, we give a 'vector'(in a list format) of (n_inputs+1)X1

    network.append(hidden_layer)
    output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
    network.append(output_layer)
    return network
```

4.1 신경망 Forward propagate

```
# Forward Propagate

#1.Neuron Transfer
#transfer = sum(weight_i * input_i) + bias

def transfer(weights, inputs):
    transferred = weights[-1]
    for i in range(len(weights)-1):
        transferred+= weights[i]*inputs[i]
    return transferred


#2. Neuron Activate
#output = 1/(1+e^(-transfer))

def activate(transferred):
    return 1.0/(1.0+exp(-transferred))
```

4.1 신경망 Forward propagate

```
#3. Feed forward

def feedforward(network, row):
    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
            transferred = transfer(neuron['weights'],inputs)
            neuron['output']=activate(transferred)
            new_inputs.append(neuron['output'])

        inputs = new_inputs
    return inputs
```

RECAP

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\partial C}{\partial w^l_{j\,k}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

4.1 신경망 Backpropagate

```
#4. Backpropagation

def transfer_derivative(output):
    return output*(1.0-output)


def backprop_error(network, expected):
    for i in reversed(range(len(network))):
        #because we start from the BACK
        layer = network[i]
        errors = list()

        if i!= len(network)-1:
        #hidden layer
            for j in range(len(layer)):
                error = 0.0
                for neuron in network[i+1]:
                    error+=(neuron['weights'][j]*neuron['delta'])
                errors.append(error)
```

4.1 신경망 Backpropagate

```
else:
#output layer
    for j in range(len(layer)):
        neuron = layer[j]
        errors.append(expected[j]-neuron['output'])

for j in range(len(layer)):
    neuron = layer[j]
    neuron['delta']=errors[j]*transfer_derivative(neuron['output'])
```

4.2 훈련시키기

```
#Train Network
#1. Update Weights with Error(delta)
#weight = weight + learning_rate*error*input

def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[:-1]
        if i!=0:
            inputs = [neuron['output'] for neuron in network[i-1]]
            #if not the first entry, input equals to the former output(i-1th)

        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j]+=l_rate*neuron['delta']*inputs[j]
            neuron['weights'][-1]+=l_rate*neuron['delta']
```

4.2 훈련시키기

```
#2. Train a network for a fixed number of epochs

def train_network(network, train, l_rate, n_epoch, n_outputs):
    for epoch in range(n_epoch):
        sum_error = 0

        for row in train:
            outputs = feedforward(network, row)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]]=1
            #when row[-1] = 4 --> data format converts into [0 0 0 0 1 0 0..]

            sum_error+= sum([(expected[i]*log(outputs[i])+(1.0-expected[i])*log(1.0-outputs[i]))
                             for i in range(len(expected))])
            backprop_error(network, expected)
            update_weights(network, row, l_rate)
        print('>epoch=%d, lrate=%.3f, error=%.3f' %(epoch, l_rate, sum_error))
```

4.3 예측치 확인

```
#Testing backprop

seed(1)

dataset = [[2.7810836,2.550537003,0],
        [1.465489372,2.362125076,0],
        [3.396561688,4.400293529,0],
        [1.38807019,1.850220317,0],
        [3.06407232,3.005305973,0],
        [7.627531214,2.759262235,1],
        [5.332441248,2.088626775,1],
        [6.922596716,1.77106367,1],
        [8.675418651,-0.242068655,1],
        [7.673756466,3.508563011,1]]
n_inputs = len(dataset[0]) - 1
n_outputs = len(set([row[-1] for row in dataset]))
network = initialize_network(n_inputs, 2, n_outputs)
train_network(network, dataset, 0.5, 20, n_outputs)
for layer in network:
        print(layer)
```

4.3 예측치 확인

```
#Making predictions
def predict(network, row):
    outputs = feedforward(network, row)
    return outputs.index(max(outputs))

for row in dataset:
    prediction = predict(network,row)
    print('Expected:%d, Got=%d' %(row[-1], prediction))
```

4.4 ScikitLearn: MLP classifier

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import load_digits
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from scipy.special import expit

import numpy as np

digits = load_digits()
digits_data = digits.data/16

X_train, X_test, y_train, y_test = train_test_split(digits_data, digits.target, test_size = 0.1)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

4.4 ScikitLearn: MLP classifier

```
#OneHotEncoder
#convert it into vectorized version
ohe = OneHotEncoder(n_values = 10)
y_train_enc = ohe.fit_transform(y_train.reshape(-1,1)).toarray()

mlp = MLPClassifier(solver='sgd', learning_rate_init=0.1, alpha=0,
                    batch_size=1617, activation='logistic',
                    random_state=10, max_iter=2000,
                    hidden_layer_sizes=100, momentum=0)

mlp.fit(X_train, y_train)
score = mlp.score(X_test, y_test)

print(score)
```

Quest: 신경망으로 분류하기

데이터셋: Animals

Quest: Animals

```
from random import seed
from random import randrange
from random import random
import csv
from math import exp
import nn_func as nn

# Load a CSV file
def load_csv(filename):
        dataset = list()
        with open(filename, 'r') as file:
                csv_reader = csv.reader(file)
                for row in csv_reader:
                        if not row:
                                continue
                        dataset.append(row)
        return dataset
```

Quest: Animals

```
# Convert string column to float
def str_to_float(dataset, column):
        for row in dataset:
                row[column] = row[column].strip()
                row[column]=float(row[column].strip())

# Convert string column to integer
def str_to_int(dataset, column):
        class_values = [row[column] for row in dataset]
        unique = set(class_values)
        lookup = dict()
        for i, value in enumerate(unique):
                lookup[value] = i
        for row in dataset:
                row[column] = lookup[row[column]]
        return lookup
```

Quest: Animals

순서를 참고하며 신경망으로 어떤 동물인지 맞추는 코드를 작성해 봅시다

1. 데이터를 **Normalize** 한다. (각 **entry** 가 **0-1** 사이 값이 되도록)
2. **Cross-validation-split** (몇% 만큼을 떼 올 것인가?) 코드 작성
3. 정확도 계산하는 코드 작성
4. 데이터를 적절히 전처리 + 우리가 이때까지 작성한 신경망 코드에 넣어 정확도를 계산해 본다.
5. **n_folds, l_rate, n_epoch** 바꿔가면서 정확도를 테스트해 본다.
6. **Mini-batch, batch** 경사하강법 코드로 바꿔 테스트해 본다.

Quest Hint: SGD

```
# Backpropagation Algorithm With Stochastic Gradient Descent
def back_propagation(train, test, l_rate, n_epoch, n_hidden):
        n_inputs = len(train[0]) - 1
        n_outputs = len(set([row[-1] for row in train]))
        network = initialize_network(n_inputs, n_hidden, n_outputs)
        train_network(network, train, l_rate, n_epoch, n_outputs)
        predictions = list()
        for row in test:
                prediction = predict(network, row)
                predictions.append(prediction)
        return prediction
```

Additional Topics in Neural Network

- Activation function: Sigmoid vs Softmax
- Gradient descent with Momentum

보충1. Softmax: Intuition

(예) (강아지, 고양이, 토끼)를 분류하려고 한다. 2개의 Training Data 각각의 마지막 layer output이

1. (0.9, 0.8, 0.7)
2. (0.5, 0.2, 0.3)

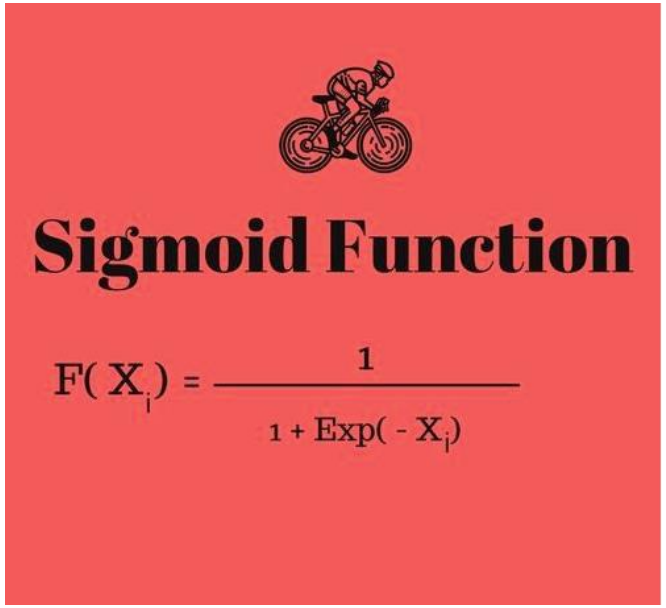
우리는 강아지가 (참)인 경우에 대해 (1,0,0)을 기대한다.

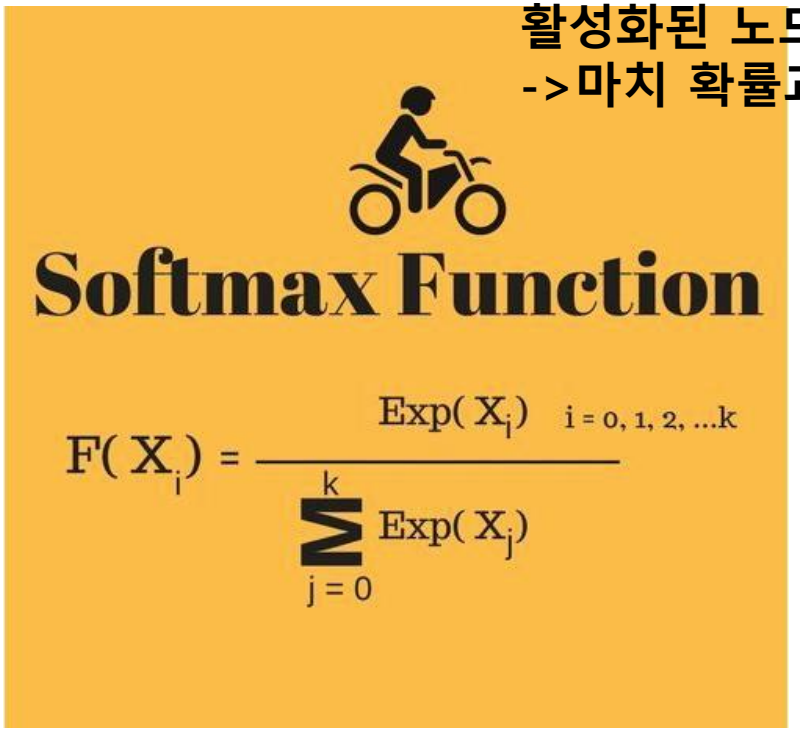
둘 중 어떤 데이터가 강아지에 더 가까운가?

보충1. Softmax: Intuition

Sigmoid vs Softmax

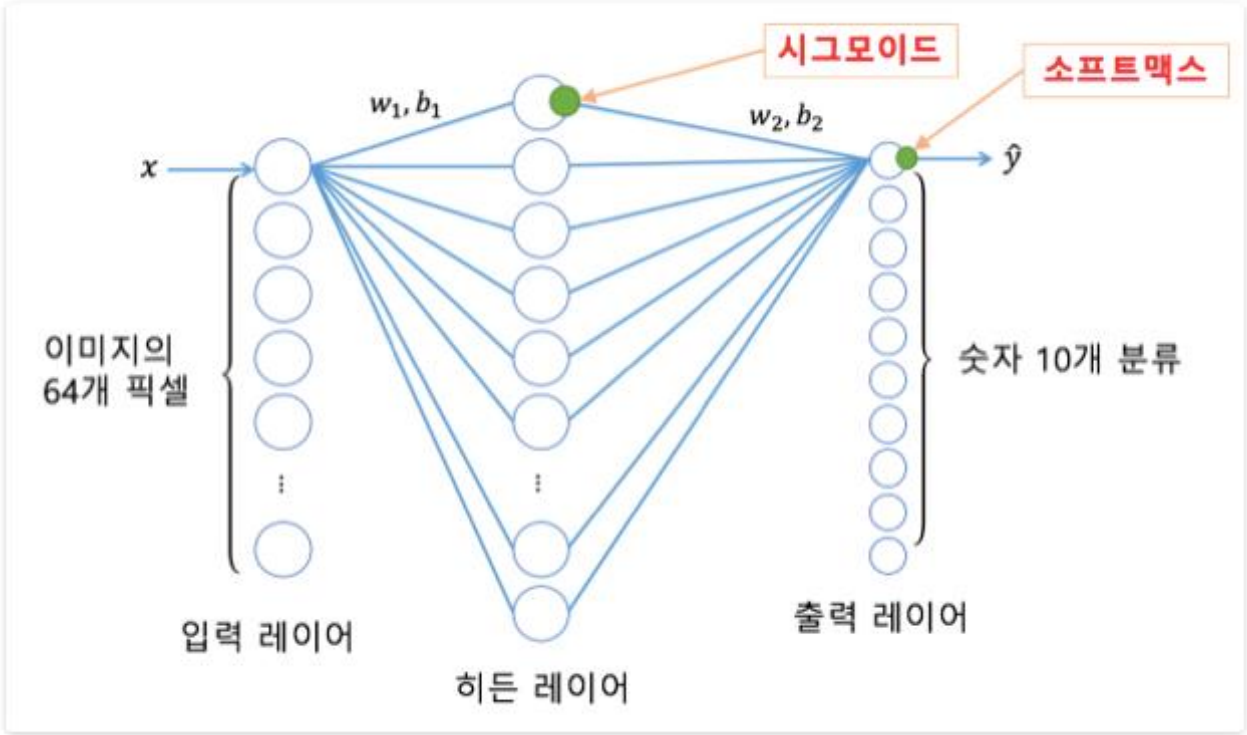
활성화된 노드 값은 0~1사이
활성화된 노드들의 합은 1
->마치 확률과 같다.





보충1. Softmax: Intuition

Softmax: Multi-classification의 성능개선



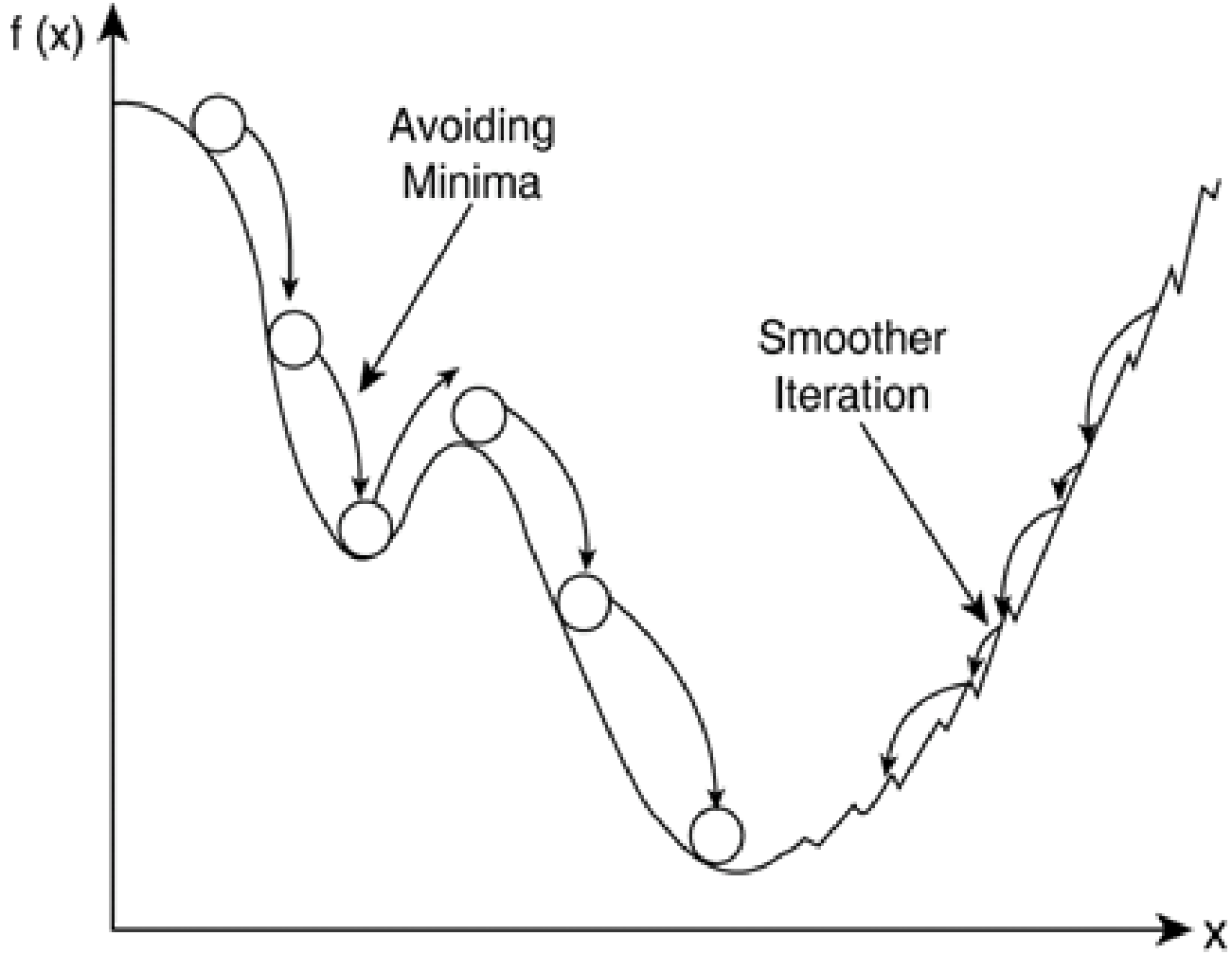
보충2. Gradient Descent with Momentum

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

$$v_t = \eta \nabla_\theta J(\theta)_t + \gamma \eta \nabla_\theta J(\theta)_{t-1} + \gamma^2 \eta \nabla_\theta J(\theta)_{t-2} + \dots.$$

보충2. Gradient Descent with Momentum



Chapter 3. 기타 자료

학구열에 불타는 당신을 위해…

- http://ruder.io/#open (Sebastian Ruder의 머신러닝 블로그)

다양한 머신러닝 최적화 기법/이론 소개(실제 코드는 없지만 최신 기법을 쉽게 잘 설명해 놓음)

- http://neuralnetworksanddeeplearning.com/index.html

신경망/딥러닝 특화. 이론 + Python 코드를 담아놓은 free e-book.

수학적 증명 및 심화된 설명이 있음.

Growth Hackers

Thank you