2017 2학기
인공지능 강의노트

8주차

강사: 양일호

School of Computer Science
University of Seoul

강의 계획

주차	강의	실습
1	과목 소개 인공지능 및 python 소개 Python 프로그래밍 환경 조성 및 기본 문법	Python 개발 환경 구축 간단한 expert system 구현
2	Search (Blind Search, Heuristic Search)	DFS, BFS, A* 구현
3	Learning / Simulated Annealing	Simulated Annealing 구현
4	Genetic Algorithm	Genetic Algorithm 구현
5	보강 주간 (개천절 / 추석 연휴)	
6	Neural Network	Perceptron 구현
7	Neural Network	Single-Layer Perceptron 구현
8	Neural Network	Multi-Layer Perceptron 구현
9	Deep Neural Network	DNN 구현
10	Deep Neural Network	DNN 구현
11	DNN 심화 (혹은 Particle Swarm Optimization)	관련 내용 구현
12	DNN 심화 (혹은 Gaussian Mixture Model)	관련 내용 구현
13	DNN 심화 (혹은 Gaussian Mixture Model)	관련 내용 구현
14	DNN 심화 (혹은 Bayesian Networks)	관련 내용 구현
15	DNN 심화 (혹은 Decision Networks)	관련 내용 구현
16	기말고사	

Python 숙달을 위한 준비 기간

예비 기간 (대체 가능)

Cost를 줄이는 방향 찾기

- Delta rule(gradient descent를 이용한 perceptron 학습 방법)
  - 현재 파라미터에서의 gradient 계산
    - 각 차원별 기울기 벡터
  - gradient의 반대 방향으로 gradient 갱신

cost function 예시:

$$J(\vec{w}) = \frac{1}{2}\left(d_i - g\left(\vec{w}^{\mathrm{T}}\vec{x}_i\right)\right)^2$$

Update:

$$\vec{w} = \vec{w} - \rho\nabla J(\vec{w})$$

gradient:

$$\nabla J(\vec{w}) = \begin{bmatrix} \frac{\partial J(\vec{w})}{\partial w_0} \\ \cdots \\ \frac{\partial J(\vec{w})}{\partial w_m} \end{bmatrix}$$

Gradient descent

- Perceptron의 출력: $z = \vec{w}^{\mathrm{T}}\vec{x}$ — weighted sum

  $o = g(z)$ ---------- $g(z)$ — activation function

- Cost function:

$$J(\vec{w}) = \frac{1}{2}(d - o)^2$$

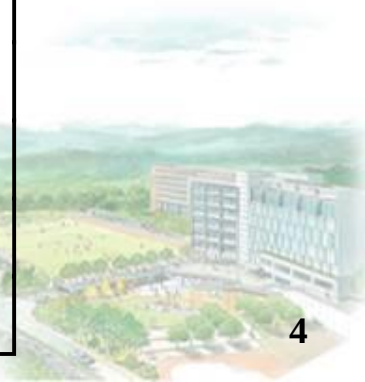
squared error

- 파라미터 업데이트(gradient descent):

$$\vec{w} = \vec{w} - \rho \nabla J(\vec{w})$$

learning rate　　gradient

$$\nabla J(\vec{w}) = \begin{bmatrix} \frac{\partial J(\vec{w})}{\partial w_0} \\ \cdots \\ \frac{\partial J(\vec{w})}{\partial w_m} \end{bmatrix}$$



Gradient descent

- 파라미터 업데이트(gradient descent) 수식 유도:

$$\frac{\partial J(\vec{w})}{\partial w_i} = -(d-o)\frac{\partial g(z)}{\partial w_i} = -(d-o)\frac{\partial g(z)}{\partial z}\frac{\partial z}{\partial w_i} \qquad z = \vec{w}^{\mathrm{T}}\vec{x}$$

$$= -(d-o)g'(z)\frac{\partial(\vec{w}^{\mathrm{T}}\vec{x})}{\partial w_i} \qquad \vec{w}^{\mathrm{T}}\vec{x} = \sum_{k=0}^{d} w_k x_k = w_0 x_0 + w_1 x_1 + \ldots + w_d x_d$$

$$= -(d-o)g'(z)x_i$$

$$\frac{\partial J(\vec{w})}{\partial w_i} = -(d-o)g'(z)x_i$$

만약 cost function이 바뀌면 다시 미분해서 유도해야 함

Gradient descent

- 파라미터 업데이트(gradient descent) 수식 유도:

$$\vec{w} = \vec{w} - \rho \nabla J(\vec{w})$$

$$\nabla J(\vec{w}) = \begin{bmatrix} \frac{\partial J(\vec{w})}{\partial w_0} \\ \cdots \\ \frac{\partial J(\vec{w})}{\partial w_m} \end{bmatrix} = \begin{bmatrix} -(d-o)g'(z)x_0 \\ \cdots \\ -(d-o)g'(z)x_m \end{bmatrix}$$

$$= -(d-o)g'(z)\begin{bmatrix} x_0 \\ \cdots \\ x_m \end{bmatrix} = -(d-o)g'(z)\vec{x}$$

$$\vec{w} = \vec{w} + \rho(d-o)g'(z)\vec{x}$$

Gradient descent

- Logistic function이 activation function인 경우:

$$\vec{w} = \vec{w} + \rho(d - o)g'(z)\vec{x}$$

$$g(z) = \frac{1}{1 + \exp(-z)}$$

Logistic function

$$g'(z) = \frac{d}{dz} g(z) = \frac{d}{dz} \frac{1}{1 + e^{-z}}$$

$$= \frac{d}{dz}\left(1 + e^{-z}\right)^{-1}$$

$$= -\left(1 + e^{-z}\right)^{-2}\left(-e^{-z}\right)$$

$$= -\frac{\left(-e^{-z}\right)}{\left(1 + e^{-z}\right)^2} = \frac{e^{-z}}{\left(1 + e^{-z}\right)^2}$$

$$\frac{d}{dx} e^{ax} = ae^{ax}$$

Gradient descent

- Logistic function이 activation function인 경우:

$$\vec{w} = \vec{w} + \rho(d - o)g'(z)\vec{x}$$

$$g(z) = \frac{1}{1 + \exp(-z)}$$

Logistic function

$$g'(z) = \frac{e^{-z}}{(1 + e^{-z})^2}$$

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

$$= g(z)\left(\frac{1 + e^{-z} - 1}{1 + e^{-z}}\right) = g(z)\left(\frac{1 + e^{-z}}{1 + e^{-z}} - \frac{1}{1 + e^{-z}}\right)$$

$$= g(z)(1 - g(z))$$

만약 activation function이 바뀌면
다시 미분해서 유도해야 함

Gradient descent

- Logistic function이 activation function인 경우:

$$\vec{w} = \vec{w} + \rho(d - o)g'(z)\vec{x}$$

$$g'(z) = g(z)(1 - g(z))$$

$$g(z) = \frac{1}{1 + \exp(-z)}$$

Logistic function

$$\vec{w} = \vec{w} + \rho(d - o)g(z)(1 - g(z))\vec{x}$$

$$= \vec{w} + \rho(d - o)o(1 - o)\vec{x}$$

Gradient descent

- 학습 샘플 1개마다 파라미터 업데이트
  - Stochastic gradient descent
- 일부 학습 샘플에 대해 파라미터 업데이트
  - Mini-batch gradient descent
  - Batch size: 한 번에 처리할 샘플 수
- 모든 학습 샘플에 대해 파라미터 업데이트
  - (Batch) gradient descent

(넓은 개념에서의) stochastic gradient descent

매 epoch 마다 학습 샘플 순서를 섞어 주면 좋음

Stochastic gradient descent

학습 반복 횟수(time step): $t = 0$

perceptron의 파라미터(weight+bias) $\vec{w}(t)$ 랜덤 초기화

학습 데이터세트(특징 벡터, 정답 레이블): $\{(\vec{x}_1, d_1), (\vec{x}_2, d_2), \ldots, (\vec{x}_n, d_n)\}$

for 충분히 오류가 감소할 때까지 반복 학습:

  for 각 학습 데이터 $(\vec{x}_k, d_k)$에 대해서 $(1 \le k \le n)$:

    $o_k = g(\vec{w}(t)^{\mathrm{T}} \vec{x}_k)$

    for 각 차원의 파라미터 $w_i(t)$ 에 대해서 $(0 \le i \le m)$:

      $w_i(t+1) = w_i(t) + \rho(d_k - o_k) o_k (1 - o_k) x_{k,i}$

  $t = t + 1$

학습 종료

1 epoch

m : 입력 특징의 차원

$x_{k,i}$: k번째 학습 특징 벡터의 i번째 원소

$w_i(t)$: t번째 반복 학습한 파라미터의 i번째 원소

Batch gradient descent

학습 반복 횟수(time step): $t = 0$

perceptron의 파라미터(weight+bias) $\vec{w}(t)$ 랜덤 초기화

학습 데이터세트(특징 벡터, 정답 레이블): $\{(\vec{x}_1, d_1), (\vec{x}_2, d_2), \ldots, (\vec{x}_n, d_n)\}$

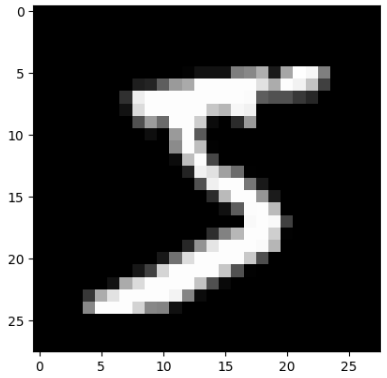
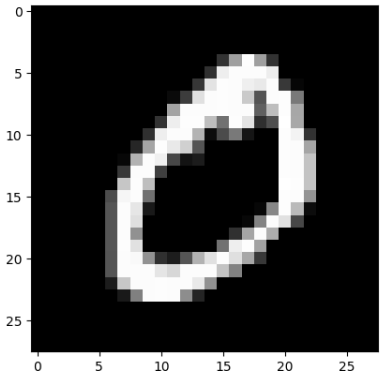
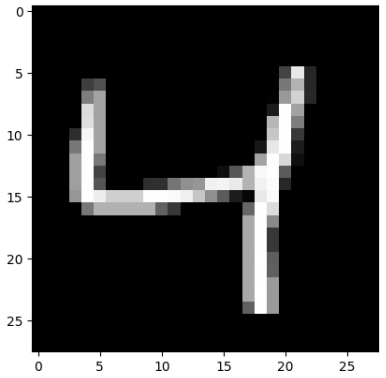
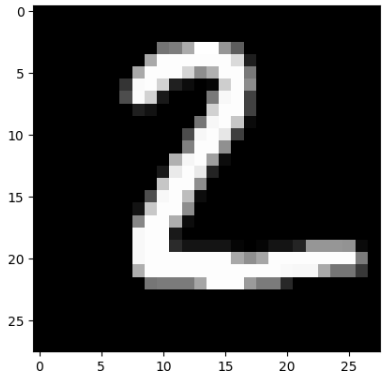
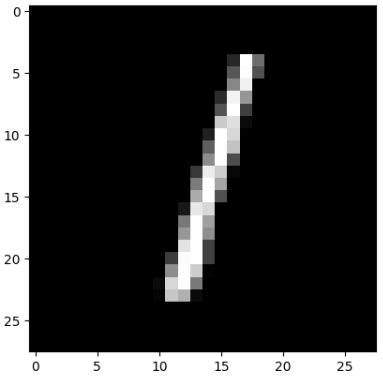
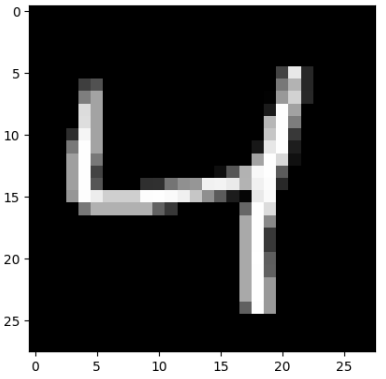
for 충분히 오류가 감소할 때까지 반복 학습:

- for 각 차원에 대해서 $(0 \le i \le m)$:
  - $\Delta w_i = 0$
- for 각 학습 데이터 $(\vec{x}_k, d_k)$에 대해서 $(1 \le k \le n)$:
  - $o_k = g(\vec{w}(t)^{\mathrm{T}} \vec{x}_k)$
  - for 각 차원에 대해서 $(0 \le i \le m)$:
    - $\Delta w_i += \rho(d_k - o_k) o_k (1 - o_k) x_{k,i}$
- for 각 차원에 대해서 $(0 \le i \le m)$:
  - $w_i(t+1) = w_i(t) + \Delta w_i$
- $t = t + 1$

1 epoch

MNIST

- THE MNIST DATABASE of handwritten digits
  - http://yann.lecun.com/exdb/mnist/

강의개요

- 강의
  - Artificial neural network

- 실습
  - Single layer perceptron 구현

강의

School of Computer Science
University of Seoul

강의 계획

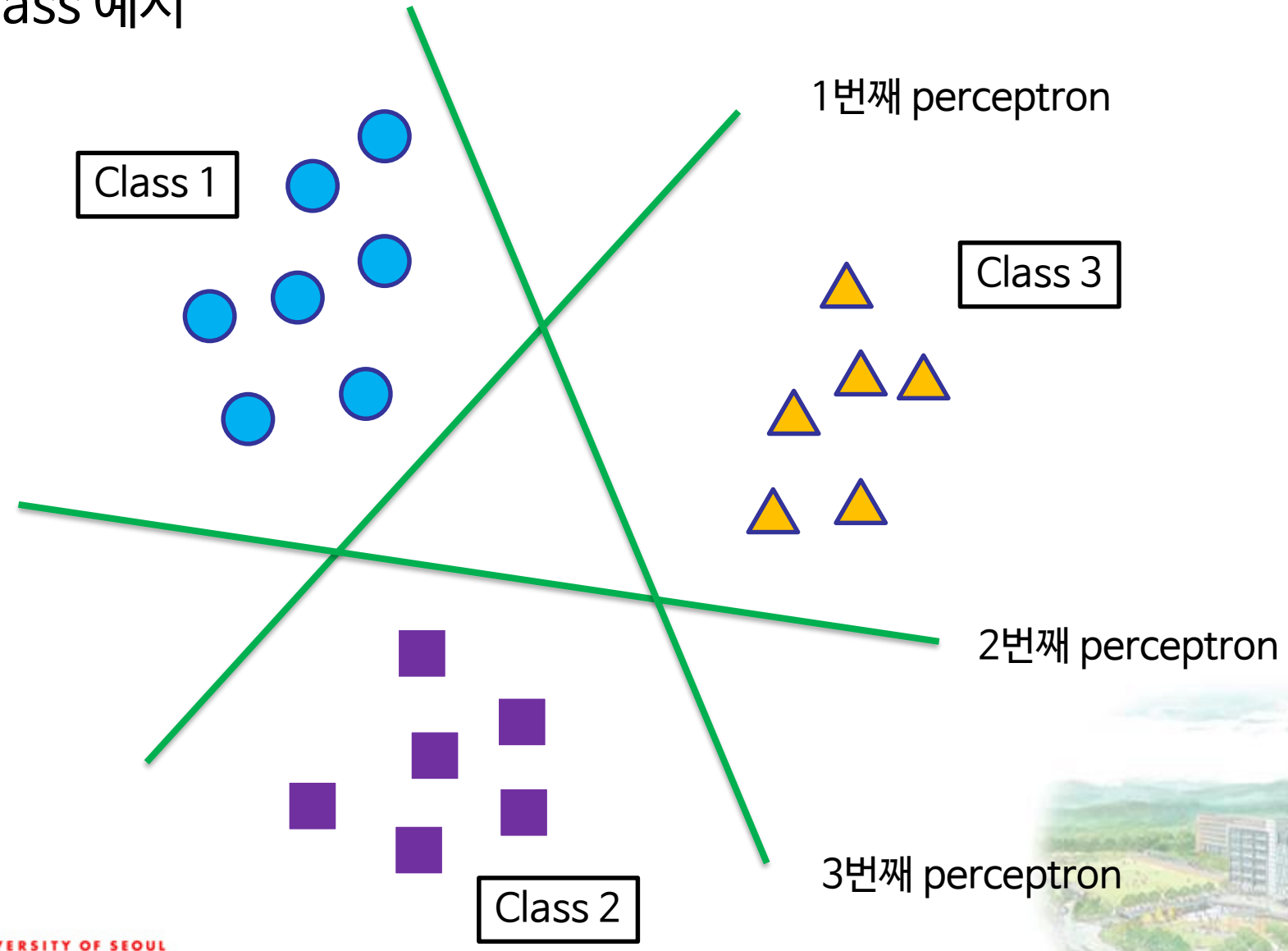
주차	강의	실습
1	과목 소개 인공지능 및 python 소개 Python 프로그래밍 환경 조성 및 기본 문법	Python 개발 환경 구축 간단한 expert system 구현
2	Search (Blind Search, Heuristic Search)	DFS, BFS, A* 구현
3	Learning / Simulated Annealing	Simulated Annealing 구현
4	Genetic Algorithm	Genetic Algorithm 구현
5	보강 주간 (개천절 / 추석 연휴)	
6	Neural Network	Perceptron 구현
7	Neural Network	(취소됨)
8	Neural Network	Single-Layer Perceptron 구현
9	Neural Network	Multi-Layer Perceptron 구현
10	Deep Neural Network	DNN 구현
11	DNN 심화 (혹은 Particle Swarm Optimization)	관련 내용 구현
12	DNN 심화 (혹은 Gaussian Mixture Model)	관련 내용 구현
13	DNN 심화 (혹은 Gaussian Mixture Model)	관련 내용 구현
14	DNN 심화 (혹은 Bayesian Networks)	관련 내용 구현
15	DNN 심화 (혹은 Decision Networks)	관련 내용 구현
16	기말고사	

Python 숙달을 위한 준비 기간

예비 기간 (대체 가능)

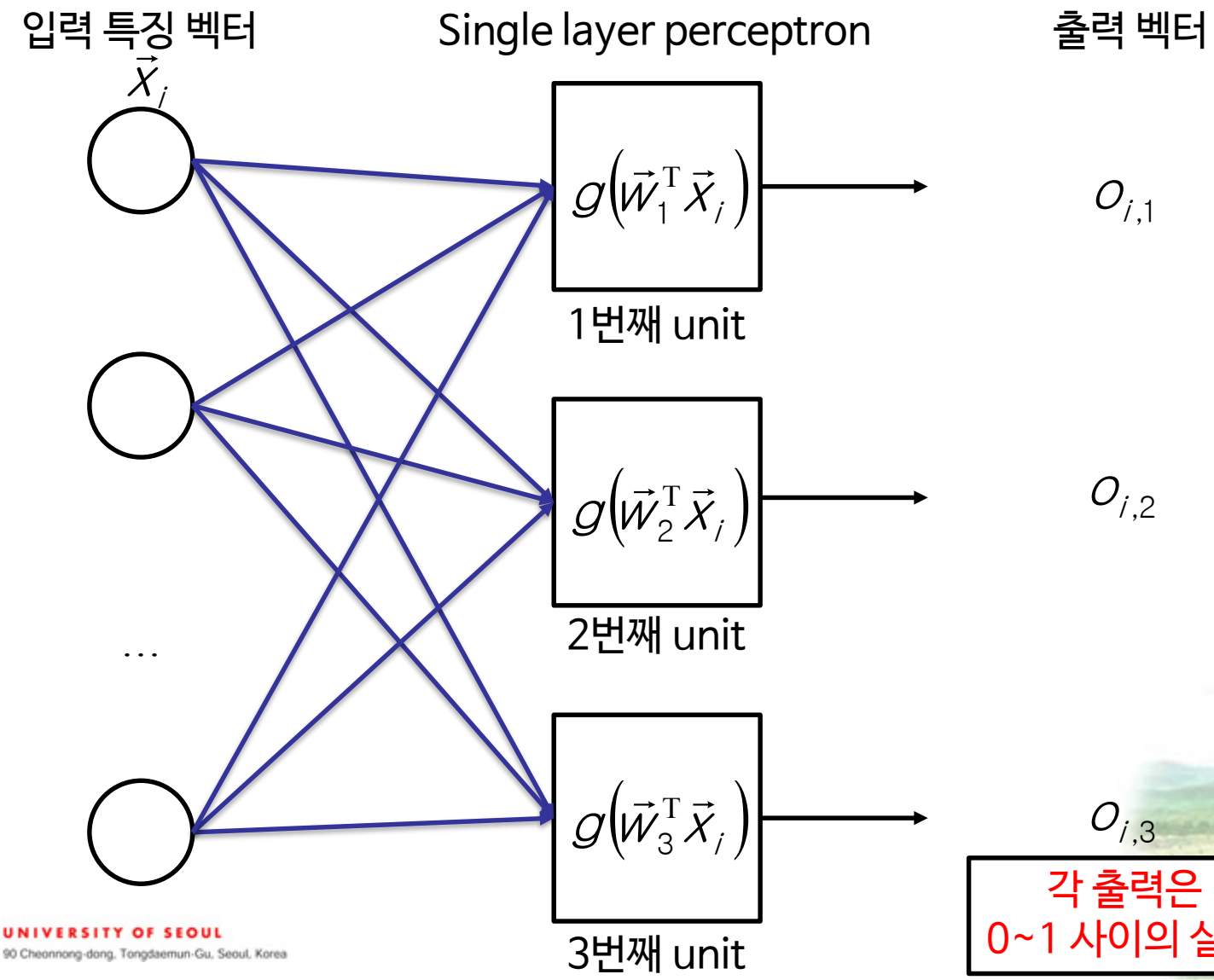
Single layer perceptron

- 3-class 예시



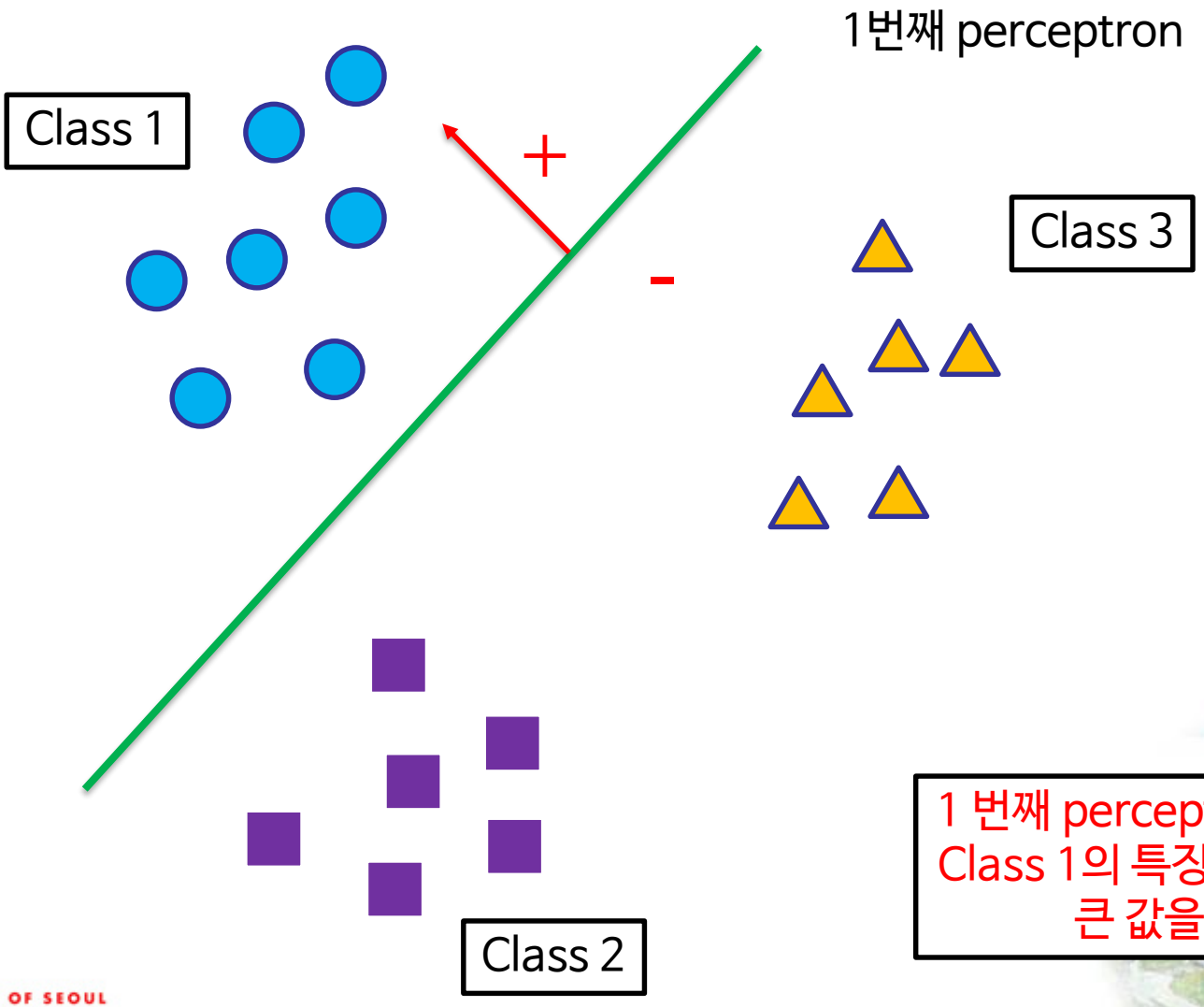
Single layer perceptron

- 3-class 예시



Single layer perceptron

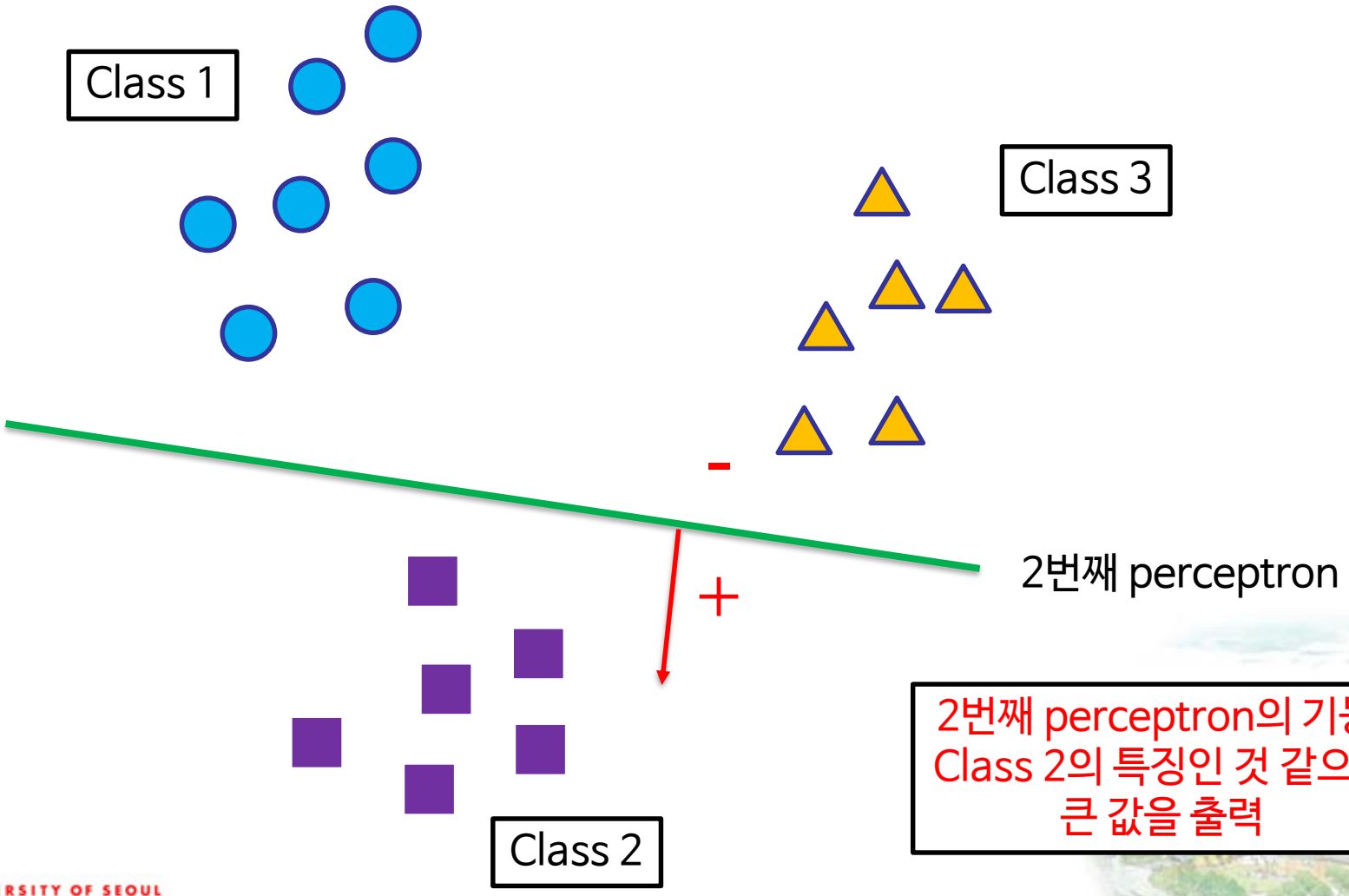
- 3-class 예시



1 번째 perceptron의 기능:
Class 1의 특징인 것 같으면
큰 값을 출력

Single layer perceptron

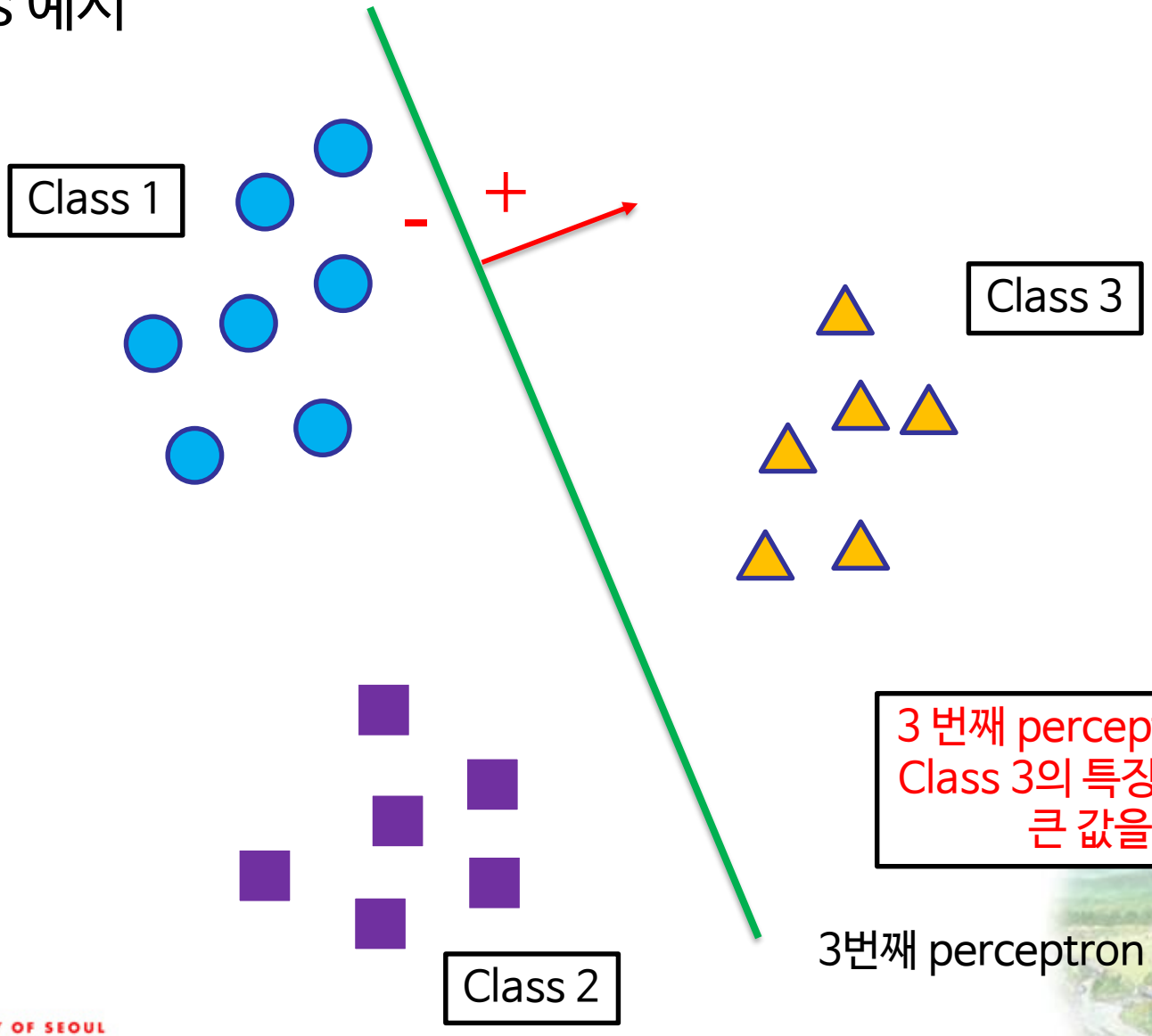
- 3-class 예시



2번째 perceptron의 기능:
Class 2의 특징인 것 같으면
큰 값을 출력

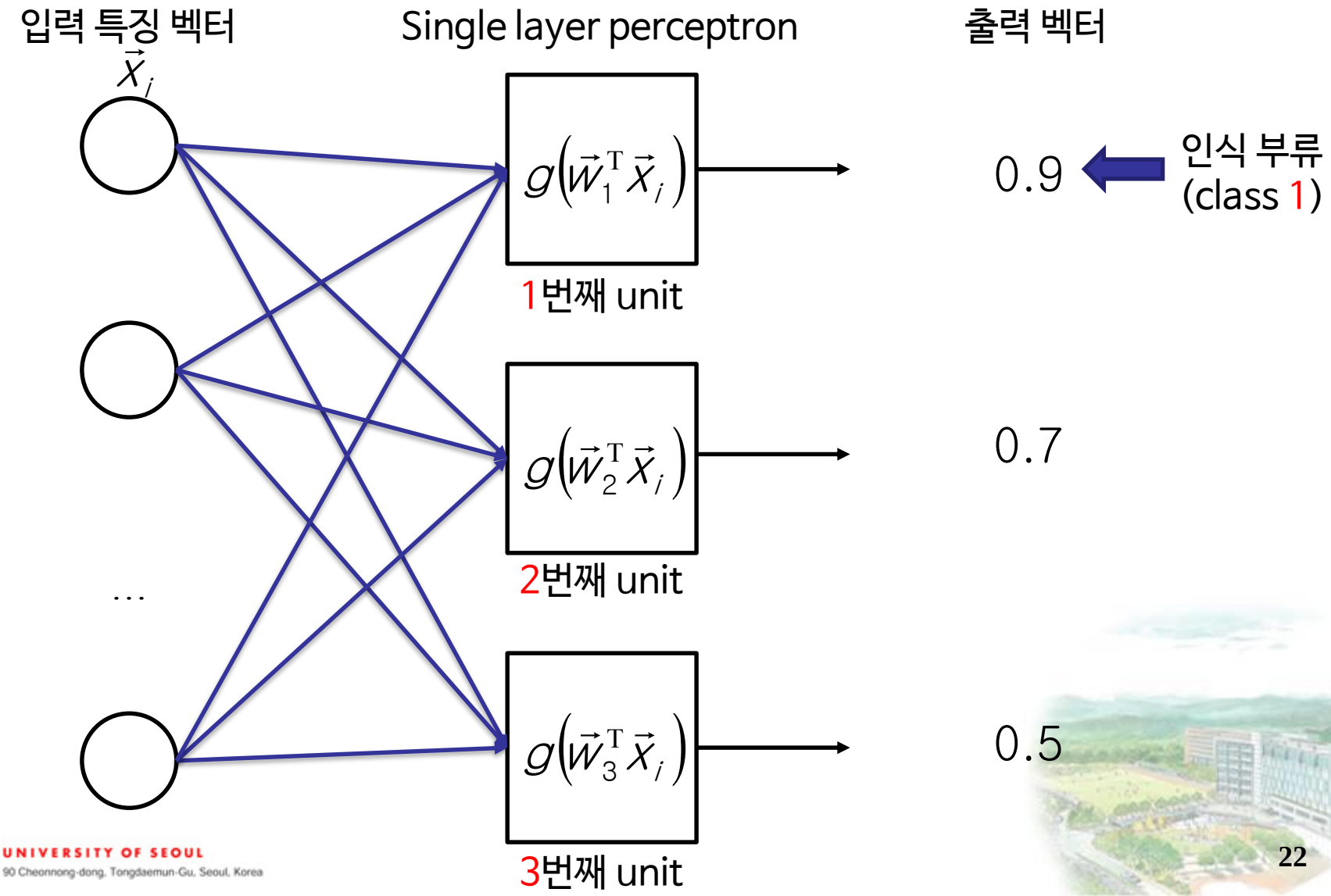
Single layer perceptron

- 3-class 예시



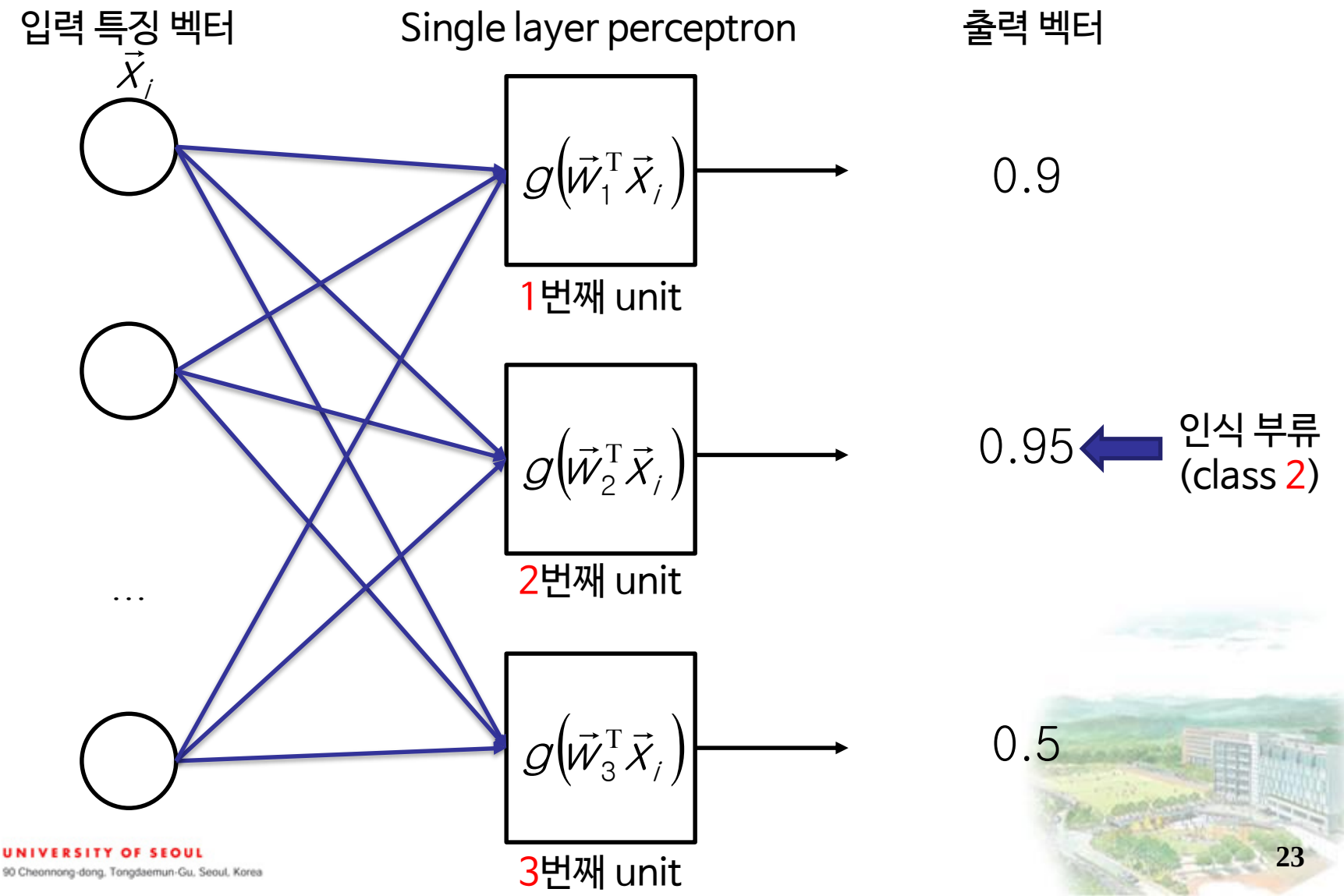
Single layer perceptron

- 테스트 과정: 최종 인식 부류 결정하기



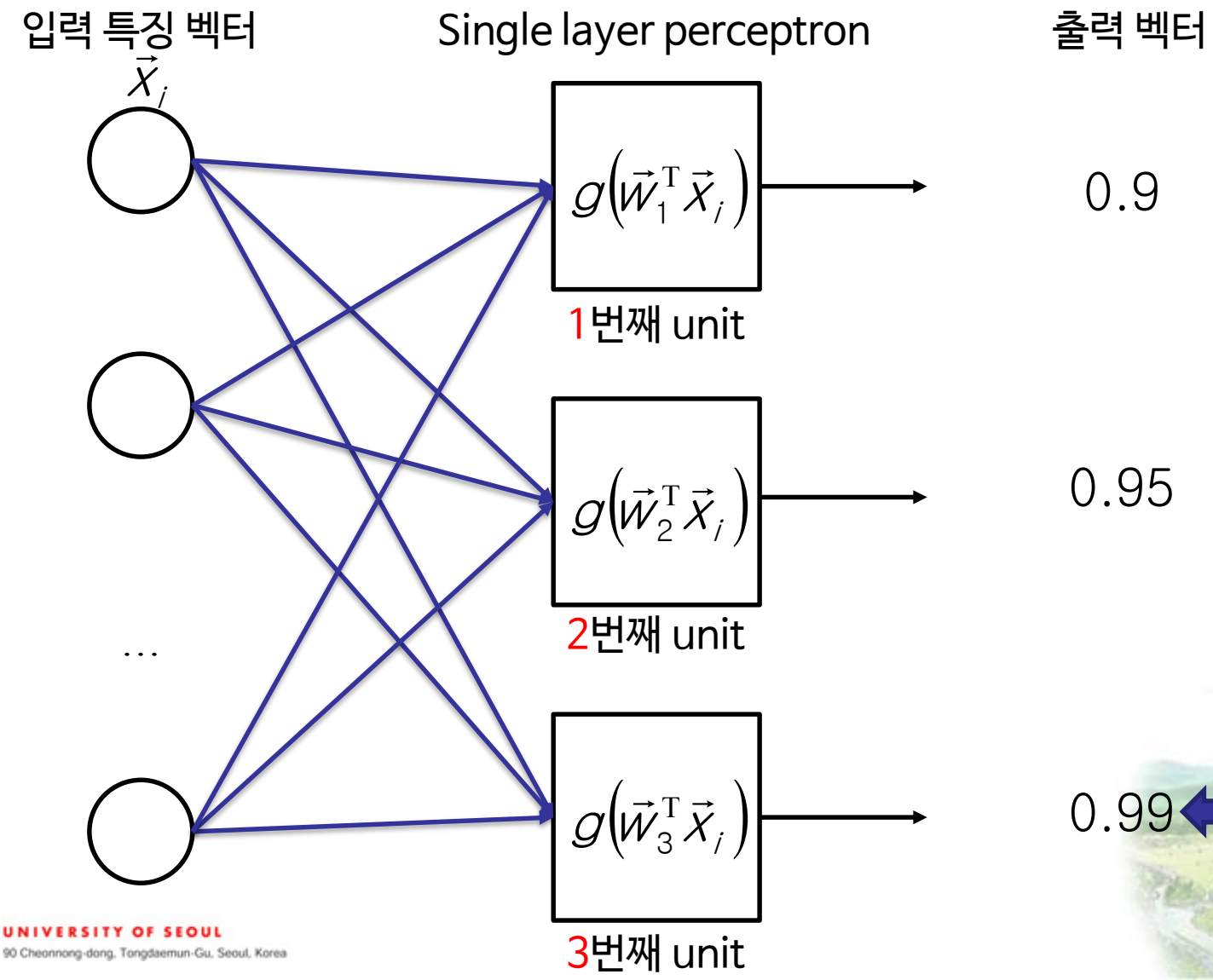
Single layer perceptron

- 테스트 과정: 최종 인식 부류 결정하기



Single layer perceptron

- 테스트 과정: 최종 인식 부류 결정하기



Single layer perceptron

- 테스트 과정: 최종 인식 부류 결정하기
  - 0~9의 숫자를 인식하는 10개의 perceptron으로 구성된 SLP
  - Perceptron의 index는 0부터 9까지
  - i-번째 테스트 데이터 $\vec{x}_i$에 대한 인식 결과(class):

$$\underset{0 \le j \le 9}{\arg\max}\, g\left(\vec{w}_j^{\mathrm{T}} \vec{x}_i\right)$$

$g\left(\vec{w}_j^{\mathrm{T}} \vec{x}_i\right)$ 가 제일 커지는 index를 반환

Single layer perceptron

- 테스트 과정: 최종 인식 부류 결정하기

$$\underset{0 \le j \le 9}{\operatorname{argmax}}\, g\left(\vec{w}_j^{\mathrm{T}} \vec{x}_i\right)$$

$g\left(\vec{w}_j^{\mathrm{T}} \vec{x}_i\right)$ 가 제일 커지는 index를 반환

```
# -*- coding: utf-8 -*-

import numpy as np

if __name__ == '__main__':
    perceptronOutputList = [0.3, 0.5, 0.9, 0.1, 0.15,
                            0.2, 0.33, 0.87, 0.6, 0.4]

    print 'max =', np.max(perceptronOutputList)
    print 'argmax =', np.argmax(perceptronOutputList)
```

결과:

```
max = 0.9
argmax = 2
```

입력 특징 scaling

- Perceptron의 출력을 계산하는 과정에서 overflow가 날 때
  - 입력 특징의 scale 줄이기
    - 기존: 0~255 사이의 정수
    - 변경: 0~1 사이의 실수

```
dataList = []
for i in range(nData):
    dataRawList = fd.read(env._N_DIM)
    dataNumList = st.unpack('B' * env._N_DIM, dataRawList)

    # bias term에 해당하는 특징 추가
    dataNumList = [1.0] + list(dataNumList)

    dataArr = np.array(dataNumList)
    dataList.append(dataArr.astype('float32') / 255.0)
```

Learning rate 선택

- 큰 learning rate
  - 초기에 빠르게 학습이 진행됨
  - 어느 정도 수렴한 이후에는 정밀한 파라미터 탐색이 어려워 진동할 수 있음

- 작은 learning rate
  - 전체적으로 학습이 느리게 진행됨
  - 어느 정도 수렴한 이후에는 정밀한 파라미터 탐색 가능

Learning rate 선택

- Learning rate decay
  - 초기에는 큰 learning rate로 시작
  - 학습이 진행되는 과정에서 점차 줄여 나감
  - 예시 1
    - 1 epoch이 지날 때마다 기존의 learning rate에 0.99를 곱하여 감쇄

decay factor
(변경 가능)

  - 예시 2
    - 기존의 learning rate를 계속 사용하다가 학습에 진전이 없으면 감쇄

Momentum

- 매 epoch마다 아주 조금씩만 cost가 개선될 때
  - 이전에 파라미터가 업데이트 되던 방향으로 계속 업데이트가 이루어지면…

Momentum 미사용: 

Δw_i : i-번째 perceptron을 업데이트하기 위한 변화량

$w_i(t+1) = w_i(t) + \Delta w_i$: i-번째 perceptron의 업데이트

  - 관성 속도를 받아 가속하여 이동

Momentum 사용: 

Δw_i : i-번째 perceptron을 업데이트하기 위한 변화량

$\Delta w_i(t) = \alpha \Delta w_i(t-1) + \Delta w_i$

$w_i(t+1) = w_i(t) + \Delta w_i(t)$

α : momentum 계수
(0이면 미사용과 동일)
(보통 0.9 정도 부여)

Hyper parameter 선택 문제

- SLP를 SGD로 학습할 때의 hyper parameter
  - 초기 파라미터 초기화 방법
  - Batch size
    - 작게 할까?
    - 크게 할까?
  - 초기 learning rate
    - 작게 할까?
    - 크게 할까?
  - Learning rate decay 방식
    - 계속 동일하게?
    - 매 epoch마다 조금씩 줄일까?
    - 학습이 정체될 때마다 줄일까?
  - Momentum 계수
    - Momentum을 쓰지 말까?
    - Momentum을 쓰면 얼마나 많이 반영할까?

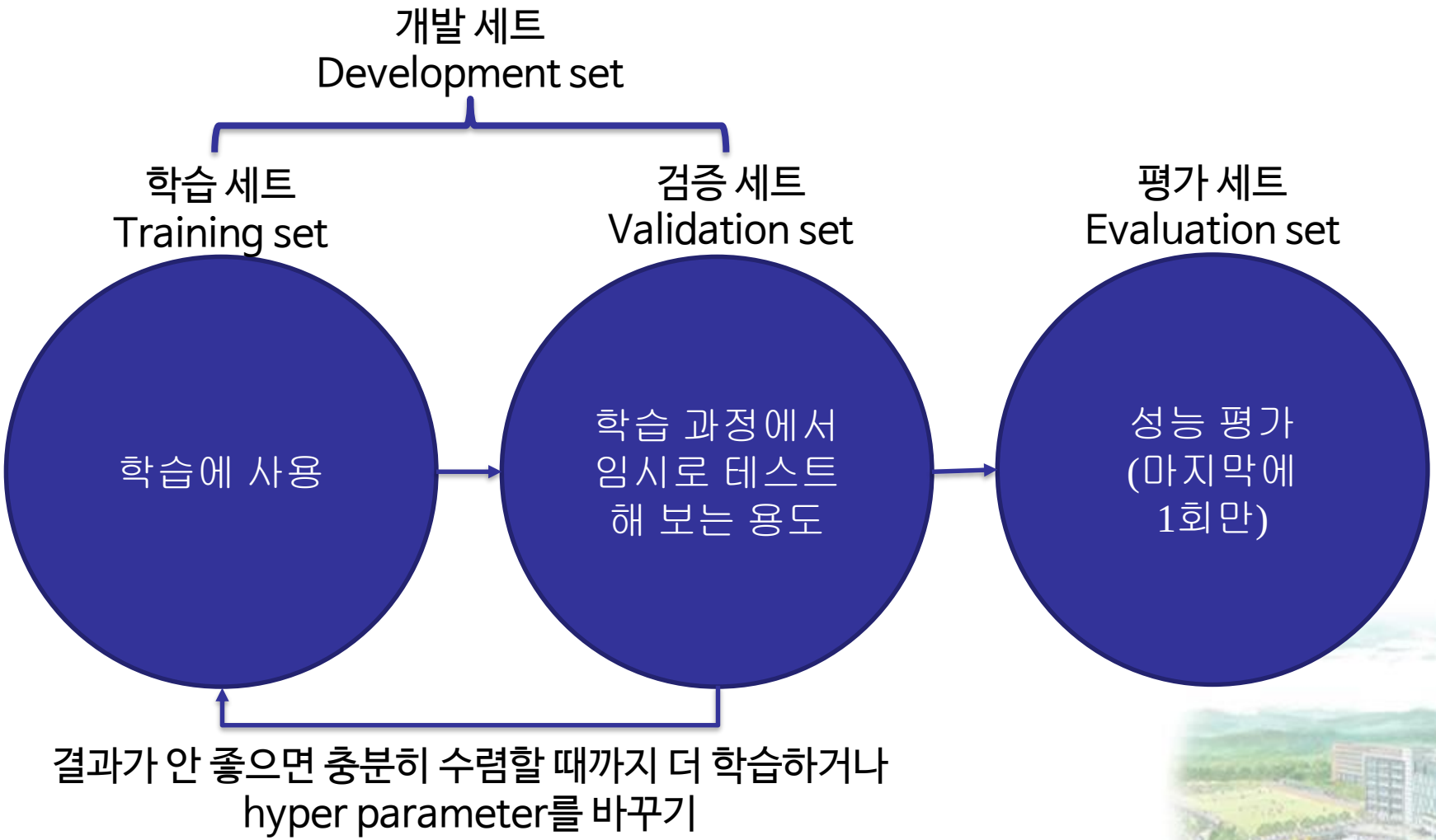
하나씩 바꿔보면서
재학습 해봐야 하나?

학습 데이터가 너무 많은데
모든 학습 데이터에 대한 cost를
다시 계산해야 되나?
(연산 시간/일반화 성능)

테스트 데이터에 대한
cost가 줄어들게 하면 되나?
(실험 공정성)

Validation 단계 두기

- 학습 데이터의 일부를 따로 빼 두고 임시 테스트용으로 쓰기



Validation 단계 두기

- 좋은 validation set 구성하기
  - Evaluation시 예상치 못한 데이터(unseen sample)가 나올 수 있음
  - → Training set과 다른 환경의 데이터로 validation set을 구성
    - 다른 환경의 데이터에 대한 일반화 성능을 높일 수 있음

- 데이터가 너무 적어서 validation set을 따로 빼기 어려울 때
  - K-fold validation
    - Development set을 k 등분
      1. 1개의 집합을 validation set으로 사용
      2. 나머지 k-1개의 집합을 training set으로 사용
      3. Validation set을 바꿔가며 1~2 과정을 거쳐 실험한 뒤 평균 성능 측정

파이썬 코드 연산 시간 단축

- 한 epoch에 소요되는 시간이 너무 길 때
  - 파이썬의 반복문은 시간이 많이 걸림
  - Numpy를 적극 이용할 것
    - 행렬 연산의 형태로 수식 정리(가능하다면…)
    - Numpy.array 형으로 변환한 뒤 행렬 연산

행렬 연산 (python 반복문)

```
# -*- coding: utf-8 -*-
import time
import numpy as np

_N_A_ROW = 2
_N_A_COL = 2

_N_B_ROW = 2
_N_B_COL = 2

if __name__ == '__main__':
    aElements = range(_N_A_ROW * _N_A_COL)
    aArr = np.array(aElements).reshape(_N_A_ROW, _N_A_COL)
    aList = aArr.tolist()

    bElements = range(_N_B_ROW * _N_B_COL)
    bArr = np.array(bElements).reshape(_N_B_ROW, _N_B_COL)
    bList = bArr.tolist()

    print 'aList'
    print aList
    print 'bList'
    print bList
```

```
aList
[[0, 1], [2, 3]]
bList
[[0, 1], [2, 3]]
```

행렬 연산 (python 반복문)

```
# dot(a, b)
before = time.time()
cList = []
for i in range(_N_A_ROW):
    rowList = []
    for j in range(_N_B_COL):
        s = 0.0
        for k in range(_N_A_COL):
            s += aList[i][k] * bList[k][j]

        rowList.append(s)

    cList.append(rowList)
now = time.time()
diffTime = now - before

print 'cList'
print cList
print 'time:', diffTime, 'sec'
```

```
aList
[[0, 1], [2, 3]]
bList
[[0, 1], [2, 3]]
```

```
cList
[[2.0, 3.0], [6.0, 11.0]]
time: 0.0 sec
```

행렬 연산 (numpy.dot)

```
# -*- coding: utf-8 -*-
import time
import numpy as np

_N_A_ROW = 2
_N_A_COL = 2

_N_B_ROW = 2
_N_B_COL = 2

if __name__ == '__main__':
    aElements = range(_N_A_ROW * _N_A_COL)
    aArr = np.array(aElements).reshape(_N_A_ROW, _N_A_COL)

    bElements = range(_N_B_ROW * _N_B_COL)
    bArr = np.array(bElements).reshape(_N_B_ROW, _N_B_COL)

    print 'aArr'
    print aArr
    print 'bArr'
    print bArr
```

```
aArr
[[0 1]
 [2 3]]
bArr
[[0 1]
 [2 3]]
```

행렬 연산 (numpy.dot)

```
# dot(a, b)
    before = time.time()
    cArr = np.dot(aArr, bArr)
    now = time.time()
    diffTime = now - before

    print 'cArr'
    print cArr
    print 'time:', diffTime, 'sec'
```

```
aArr
[[0 1]
 [2 3]]
bArr
[[0 1]
 [2 3]]
```

```
cArr
[[ 2  3]
 [ 6 11]]
time: 0.0 sec
```

행렬 연산 속도 비교

행렬 크기	행렬곱 연산 시간 (초)	
	python 반복문	numpy.dot
(2x2) x (2x2)	0.000	0.000
(20x20) x (20x20)	0.004	0.001
(200x200) x (200x200)	3.668	0.009
(2000x2000) x (2000x2000)	?	?

SLP 연산 과정 행렬화

- Perceptron 하나의 연산 과정

$\{(\vec{x}_1, d_1), (\vec{x}_2, d_2), \ldots, (\vec{x}_n, d_n)\}$: 학습 데이터 세트(특징 벡터, 정답 레이블)

$o_{i,j} = g(\vec{w}_j^{\mathrm{T}} \vec{x}_i)$: i번째 학습 데이터에 대한 j번째 perceptron의 출력

$$\underbrace{\begin{bmatrix} w_{j,0} & w_{j,1} & \cdots & w_{j,m} \end{bmatrix}}_{1 \times (m+1)} \underbrace{\begin{bmatrix} x_{i,0} \\ x_{i,1} \\ \cdots \\ x_{i,m} \end{bmatrix}}_{(m+1) \times 1} = \underbrace{\vec{w}_j^{\mathrm{T}} \vec{x}_i}_{1 \times 1}$$

SLP 연산 과정 행렬화

- Perceptron 하나의 연산 과정
  - i번째 학습 데이터 $\vec{x}_i$ 각각에 대한 연산

$$\begin{bmatrix} w_{j,0} & w_{j,1} & \cdots & w_{j,m} \end{bmatrix} \begin{bmatrix} x_{1,0} \\ x_{1,1} \\ \cdots \\ x_{1,m} \end{bmatrix} = \vec{w}_j^{\mathrm{T}} \vec{x}_1$$

$$\begin{bmatrix} w_{j,0} & w_{j,1} & \cdots & w_{j,m} \end{bmatrix} \begin{bmatrix} x_{2,0} \\ x_{2,1} \\ \cdots \\ x_{2,m} \end{bmatrix} = \vec{w}_j^{\mathrm{T}} \vec{x}_2$$

$$\cdots\cdots$$

$$\begin{bmatrix} w_{j,0} & w_{j,1} & \cdots & w_{j,m} \end{bmatrix} \begin{bmatrix} x_{n,0} \\ x_{n,1} \\ \cdots \\ x_{n,m} \end{bmatrix} = \vec{w}_j^{\mathrm{T}} \vec{x}_n$$

SLP 연산 과정 행렬화

- Perceptron 하나의 연산 과정
  - 모든 학습 데이터를 한 번에 처리하기

$$\begin{bmatrix} w_{j,0} & w_{j,1} & \cdots & w_{j,m} \end{bmatrix} \begin{bmatrix} x_{1,0} & x_{2,0} & \cdots & x_{n,0} \\ x_{1,1} & x_{2,1} & \cdots & x_{n,1} \\ \cdots & \cdots & \cdots & \cdots \\ x_{1,m} & x_{2,m} & \cdots & x_{n,m} \end{bmatrix}$$

1 x (m+1)

(m+1) x n

$$= \begin{bmatrix} \vec{w}_j^{\mathrm{T}} \vec{x}_1 & \vec{w}_j^{\mathrm{T}} \vec{x}_2 & \cdots & \vec{w}_j^{\mathrm{T}} \vec{x}_n \end{bmatrix}$$

1 x n

SLP 연산 과정 행렬화

- Perceptron 하나의 연산 과정
  - 모든 학습 데이터를 한 번에 처리하기

$$\begin{bmatrix} w_{j,0} & w_{j,1} & \cdots & w_{j,m} \end{bmatrix} \begin{bmatrix} x_{1,0} & x_{2,0} & \cdots & x_{n,0} \\ x_{1,1} & x_{2,1} & \cdots & x_{n,1} \\ \cdots & \cdots & \cdots & \cdots \\ x_{1,m} & x_{2,m} & \cdots & x_{n,m} \end{bmatrix}$$

1 x (m+1)

(m+1) x n

$$= \begin{bmatrix} \vec{w}_j^{\mathrm{T}} \vec{x}_1 & \vec{w}_j^{\mathrm{T}} \vec{x}_2 & \cdots & \vec{w}_j^{\mathrm{T}} \vec{x}_n \end{bmatrix}$$

1 x n

SLP 연산 과정 행렬화

- Perceptron 하나의 연산 과정
  - 모든 학습 데이터를 한 번에 처리하기

$$\begin{bmatrix} w_{j,0} & w_{j,1} & \cdots & w_{j,m} \end{bmatrix} \begin{bmatrix} x_{1,0} & x_{2,0} & \cdots & x_{n,0} \\ x_{1,1} & x_{2,1} & \cdots & x_{n,1} \\ \cdots & \cdots & \cdots & \cdots \\ x_{1,m} & x_{2,m} & \cdots & x_{n,m} \end{bmatrix}$$

1 x (m+1)

(m+1) x n

$$= \begin{bmatrix} \vec{w}_j^{\mathrm{T}} \vec{x}_1 & \vec{w}_j^{\mathrm{T}} \vec{x}_2 & \cdots & \vec{w}_j^{\mathrm{T}} \vec{x}_n \end{bmatrix}$$

1 x n

SLP 연산 과정 행렬화

- Perceptron 하나의 연산 과정
  - 모든 학습 데이터를 한 번에 처리하기

$$\begin{bmatrix} w_{j,0} & w_{j,1} & \cdots & w_{j,m} \end{bmatrix} \begin{bmatrix} x_{1,0} & x_{2,0} & \cdots & x_{n,0} \\ x_{1,1} & x_{2,1} & \cdots & x_{n,1} \\ \cdots & \cdots & \cdots & \cdots \\ x_{1,m} & x_{2,m} & \cdots & x_{n,m} \end{bmatrix}$$

1 x (m+1)

(m+1) x n

$$= \begin{bmatrix} \vec{w}_j^{\mathrm{T}} \vec{x}_1 & \vec{w}_j^{\mathrm{T}} \vec{x}_2 & \cdots & \vec{w}_j^{\mathrm{T}} \vec{x}_n \end{bmatrix}$$

1 x n

SLP 연산 과정 행렬화

- 모든 perceptron/모든 학습 데이터를 한 번에 처리하기
  - Perceptron이 c개 있을 때:

$$\begin{bmatrix} w_{1,0} & w_{1,1} & \cdots & w_{1,m} \\ w_{2,0} & w_{2,1} & \cdots & w_{3,m} \\ \cdots & \cdots & \cdots & \cdots \\ w_{c,0} & w_{c,1} & \cdots & w_{c,m} \end{bmatrix} \begin{bmatrix} x_{1,0} & x_{2,0} & \cdots & x_{n,0} \\ x_{1,1} & x_{2,1} & \cdots & x_{n,1} \\ \cdots & \cdots & \cdots & \cdots \\ x_{1,m} & x_{2,m} & \cdots & x_{n,m} \end{bmatrix}$$

c x (m+1) (m+1) x n

혹은 perceptron별로는 multiprocessing을 해도 됨

SLP 연산 과정 행렬화

$$o_{i,j} = g\left(\vec{w}_j^{\mathrm{T}} \vec{x}_i\right)$$

Activation function을 각 행렬의 원소별로 적용하는 방법?

```
# -*- coding: utf-8 -*-
import numpy as np
import numpy.random as nr

# logistic activation
def activation(z):
    return 1.0 / (1.0 + np.exp(-z))

if __name__ == '__main__':
    nr.seed(12345)
    z = nr.rand(2, 2)

    print 'z'
    print z
    print 'g(z)'
    print activation(z)
```

```
z
[[ 0.92961609  0.31637555]
 [ 0.18391881  0.20456028]]
g(z)
[[ 0.71699739  0.57844069]
 [ 0.54585053  0.55096248]]
```

QnA

실습

School of Computer Science
University of Seoul

실습과제 (1 / 3)

- MNIST DB 다운로드 받기
  - http://yann.lecun.com/exdb/mnist/

- 학습 과정(train.py)
  - Single-layer perceptron 숫자인식기 학습
    - 엄격하게 validation을 할 필요는 없음(테스트 오류가 적당히 줄어들면 중단)
    - 테스트 오류율이 20% 이내로는 줄어들도록
  - 매 학습 반복(epoch)마다 오류율 혹은 코스트 콘솔 출력 + 텍스트 파일로 저장
    - "train_log.txt"
  - 가장 좋은 모델 파라미터를 파일로 저장
    - "best_param.pkl"

실습과제 (2 / 3)

- 테스트 과정(test.py)
  - 파일로 저장한 가장 좋은 모델 파라미터 읽기
    - “best_param.pkl”
  - 분류 결과를 출력 + 텍스트 파일로 저장
    - 각 샘플별 분류 결과는 너무 많으니 콘솔 출력하지 말고 텍스트 파일로만 저장
    - 마지막에 오류율 계산하여 보여주기
    - “test_output.txt”

- 주의사항
  - 학습 스크립트와 테스트 스크립트를 분리할 것
  - 가장 좋은 모델 파라미터를 파일로 저장하여 동봉할 것
  - 테스트 스크립트에서는 학습이 완료된 모델 파라미터를 불러와 사용할 것

실습과제 (3 / 3)

- 제출 형식
  - 입력 파일은 굳이 같이 제출할 필요 없음(필요한 경우는 함께 제출)
    - 단, 한 단계 상위 폴더에서 로드하기
      - 학습 데이터
        - ≫ ../train-images.idx3-ubyte
        - ≫ ../train-labels.idx1-ubyte
      - 테스트 데이터
        - ≫ ../t10k-images.idx3-ubyte
        - ≫ ../t10k-labels.idx1-ubyte
  - 출력 파일과 함께 제출
    - 학습 스크립트(train.py) / 테스트 스크립트(test.py)
    - 학습 로그(“train_log.txt”) / 테스트 결과(“test_output.txt”)
    - 가장 좋은 모델 파라미터 저장 파일(“best_param.pkl”)

SLP 연산 과정 행렬화

- 모든 perceptron/모든 학습 데이터를 한 번에 처리하기
  - Perceptron이 c개 있을 때:

$$W^T \qquad X$$

$$\begin{bmatrix} w_{1,0} & w_{1,1} & \cdots & w_{1,m} \\ w_{2,0} & w_{2,1} & \cdots & w_{3,m} \\ \cdots & \cdots & \cdots & \cdots \\ w_{c,0} & w_{c,1} & \cdots & w_{c,m} \end{bmatrix} \begin{bmatrix} x_{1,0} & x_{2,0} & \cdots & x_{n,0} \\ x_{1,1} & x_{2,1} & \cdots & x_{n,1} \\ \cdots & \cdots & \cdots & \cdots \\ x_{1,m} & x_{2,m} & \cdots & x_{n,m} \end{bmatrix}$$

c x (m+1) (m+1) x n

$$= W^T X$$

c x n

n: 전체 학습 샘플 수

SLP 연산 과정 행렬화

- Batch size에 따른 주의사항
  - 한 (mini-) batch안에 포함된 학습 데이터에 대해서만 한번에 처리할 것
    - 파라미터를 업데이트한 뒤에 다음 batch를 처리해야 하므로…

$$W^T \qquad X_{batch}$$

$$\begin{bmatrix} w_{1,0} & w_{1,1} & \cdots & w_{1,m} \\ w_{2,0} & w_{2,1} & \cdots & w_{3,m} \\ \cdots & \cdots & \cdots & \cdots \\ w_{c,0} & w_{c,1} & \cdots & w_{c,m} \end{bmatrix} \begin{bmatrix} x_{1,0} & x_{2,0} & \cdots & x_{n_b,0} \\ x_{1,1} & x_{2,1} & \cdots & x_{n_b,1} \\ \cdots & \cdots & \cdots & \cdots \\ x_{1,m} & x_{2,m} & \cdots & x_{n_b,m} \end{bmatrix}$$

c x (m+1) (m+1) x n_b

$$= W^T X_{batch} \quad c \times n_b$$

n_b: 한 batch 안에 포함될 학습 샘플 수 (batchSize)

Mini-batch 학습시 주의점

- 매 epoch마다 학습 샘플 순서 바꿔주기
  - 하나의 batch에 포함되는 샘플들이 골고루 섞이도록

- 파라미터 업데이트시 누적치 대신 평균치 사용하기
  - 하나의 batch에 포함된 샘플의 수가 달라질 수 있음

batchSize: 25000

Python 실습

- random.shuffle

```
# -*- coding: utf-8 -*-
import random as ra

_N_SAMPLE = 6

if __name__ == '__main__':
    sampleIndexList = range(_N_SAMPLE)        # 전체 학습 샘플 인덱스 목
록
    print 'sampleIndexList', sampleIndexList
    ra.shuffle(sampleIndexList)               # 인덱스 목록 순서 섞기
    print 'sampleIndexList', sampleIndexList
```

출력:

```
sampleIndexList [0, 1, 2, 3, 4, 5]
sampleIndexList [1, 5, 4, 3, 2, 0]
```

Python 실습

- numpy.array의 slicing

```
# -*- coding: utf-8 -*-
import numpy as np

_N_DIM = 2      # 특징 차원
_N_SAMPLE = 6   # 학습 샘플 수

if __name__ == '__main__':
    trData = np.array(range(_N_DIM * _N_SAMPLE)).reshape(_N_DIM, _N_SAMPLE)
    print trData
    print
    print trData[:, 0]
    print trData[:, 0].reshape(-1, 1)
    print
    print trData[:, [0, 1]]
    print trData[:, [0, 2, 3]]
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]

[0 6]
[[0]
 [6]]

[[0 1]
 [6 7]]
[[0 2 3]
 [6 8 9]]
```

Python 실습

- random.shuffle을 이용한 numpy.array의 random slicing

Python 실습

```
# -*- coding: utf-8 -*-
import random as ra
import numpy as np

_N_DIM = 2      # 특징 차원
_N_SAMPLE = 6   # 학습 샘플 수
_BATCH_SIZE = 2 # mini-batch 크기
_N_BATCH = int(_N_SAMPLE / _BATCH_SIZE) # batch 수

if __name__ == '__main__':
    trData = np.array(range(_N_DIM * _N_SAMPLE)).reshape(_N_DIM, _N_SAMPLE)
    print trData

    sampleIndexList = range(_N_SAMPLE)      # 전체 학습 샘플 인덱스 목록
    print 'sampleIndexList', sampleIndexList
    ra.shuffle(sampleIndexList)             # 인덱스 목록 순서 섞기
    print 'sampleIndexList', sampleIndexList

    for batchIndex in range(_N_BATCH):
        start = batchIndex * _BATCH_SIZE
        end = np.min([_N_SAMPLE, (batchIndex + 1) * _BATCH_SIZE])
        batchSampleIndexList = sampleIndexList[start:end]
        batchTrData = trData[:, batchSampleIndexList]   # 배치 샘플 선택
        print 'batch', batchIndex
        print batchTrData
```

Python 실습

- random.shuffle을 이용한 numpy.array의 slicing

출력:

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
sampleIndexList [0, 1, 2, 3, 4, 5]
sampleIndexList [0, 4, 2, 1, 3, 5]
batch 0
[[ 0  4]
 [ 6 10]]
batch 1
[[2 1]
 [8 7]]
batch 2
[[ 3  5]
 [ 9 11]]
```

Hyperparameter 찾기

- Automatic hyperparameter optimization algorithm
  - Grid search
    - 일정 간격으로 하이퍼 파라미터를 바꾸어가며 실험

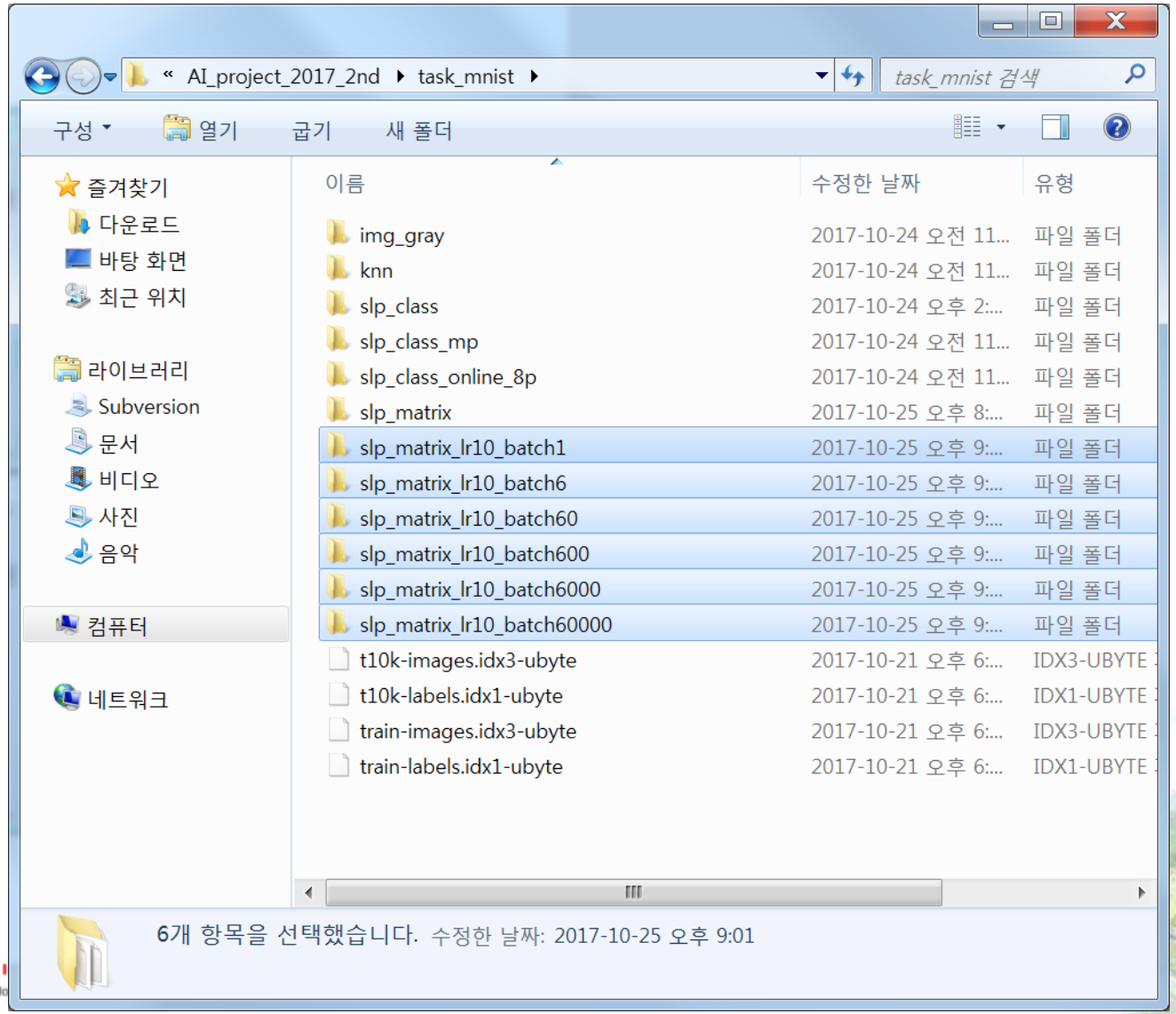
learningRate: 0.1 batchSize: 60	learningRate: 1 batchSize: 60	learningRate: 10 batchSize: 60
learningRate: 0.1 batchSize: 600	learningRate: 1 batchSize: 600	learningRate: 10 batchSize: 600
learningRate: 0.1 batchSize: 6000	learningRate: 1 batchSize: 6000	learningRate: 10 batchSize: 6000

  - Random search
    - 하이퍼 파라미터를 랜덤하게 바꾸어가며 실험

learningRate: 0.15 batchSize: 600	learningRate: 1.3 batchSize: 60	learningRate: 0.78 batchSize: 6000

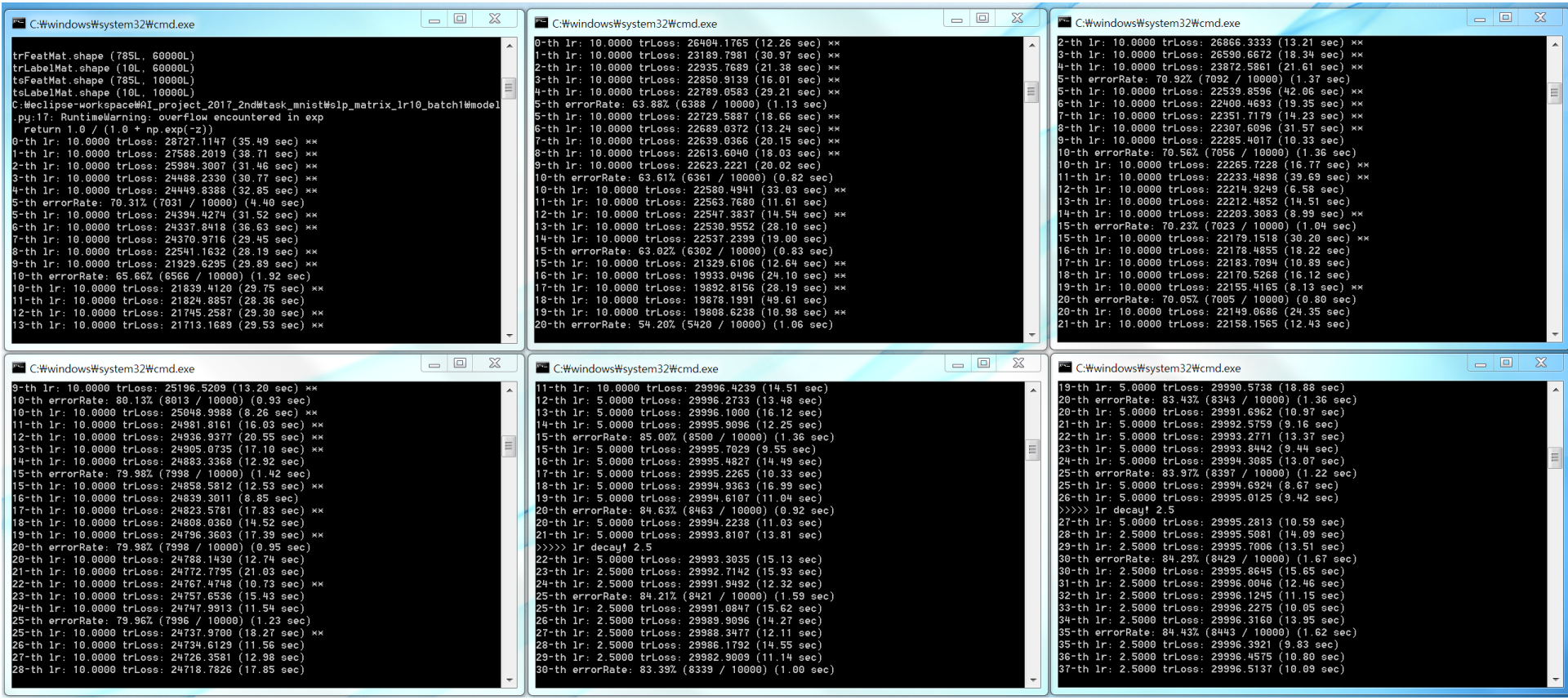
Hyperparameter 찾기

- Grid search 예시



Hyperparameter 찾기

- Grid search 예시



Warm-up

- 매우 낮은 학습률로 몇 epoch 정도 초벌 학습
  - Ex〉
    - 처음 10 epoch 동안 학습률 0.01로 학습(warm-up)
    - Warm-up 종료 후 학습률을 10으로 올려 진행

Python 실습

- 특정 정수 레이블의 one-hot representation 구현 예시

```
# -*- coding: utf-8 -*-
import numpy as np

_N_CLASS = 10          # 0~9까지 총 10개 클래스

if __name__ == '__main__':
    label = 3          # 0~9 사이의 레이블
    oneHot = np.zeros(_N_CLASS).astype('float32')
    print 'oneHot =', oneHot
    oneHot[label] = 1.0
    print 'oneHot =', oneHot
```

출력:

```
oneHot = [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
oneHot = [ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
```

Python 실습

- one-hot representation을 특정 정수 레이블로 바꾸기

```
# -*- coding: utf-8 -*-
import numpy as np

_N_CLASS = 10        # 0~9까지 총 10개 클래스

if __name__ == '__main__':
    oneHot = np.array([0.0, 0.0, 0.0, 1.0, 0.0,
                       0.0, 0.0, 0.0, 0.0, 0.0])
    label = np.argmax(oneHot)
    print 'label =', label
```

출력:

```
label = 3
```

Python 실습

- Model parameter 저장 / 로드 예시

```
# -*- coding: utf-8 -*-
import pickle as pkl

def save(fn, obj):
    fd = open(fn, 'wb')
    pkl.dump(obj, fd)
    fd.close()

def load(fn):
    fd = open(fn, 'rb')
    obj = pkl.load(fd)
    fd.close()
    return obj

if __name__ == '__main__':
    param = [0.5, -1.5, 1.0, 0.0]
    save('best_param.pkl', param)
    loadedParam = load('best_param.pkl')
    print 'loadedParam:', loadedParam
```

출력:

```
loadedParam: [0.5, -1.5, 1.0, 0.0]
```

QnA