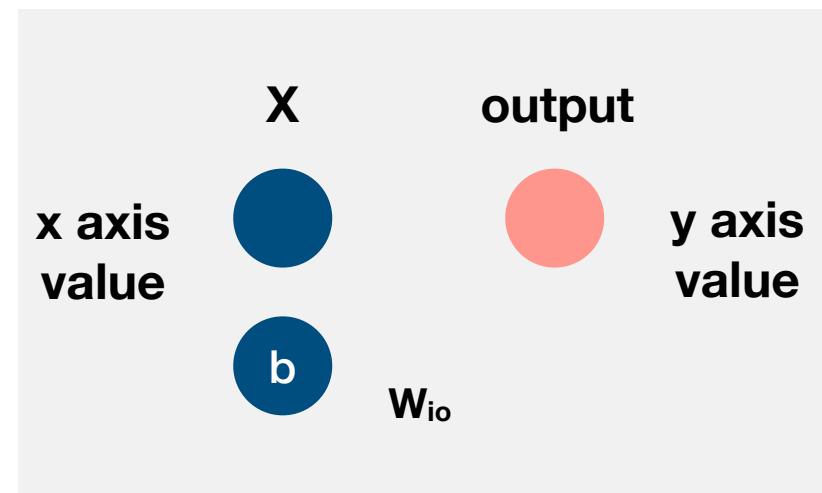
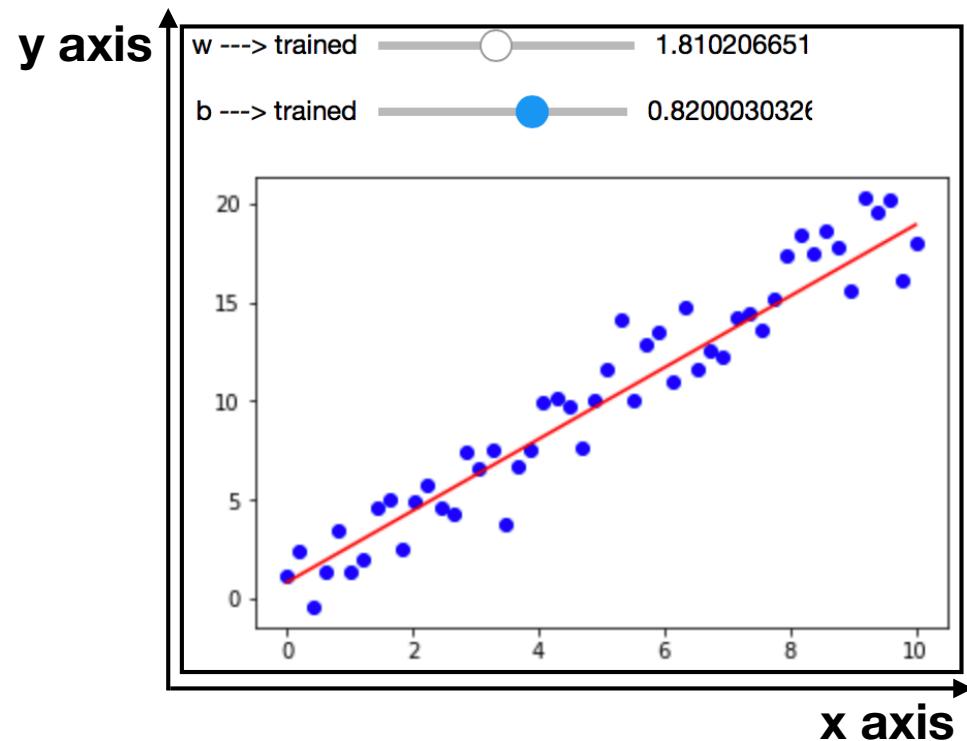


Neural Net basics

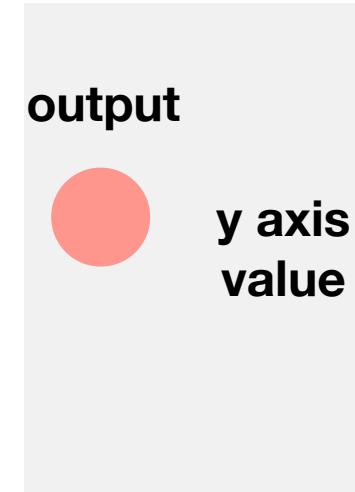
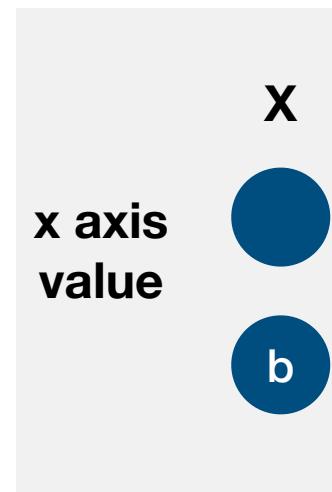
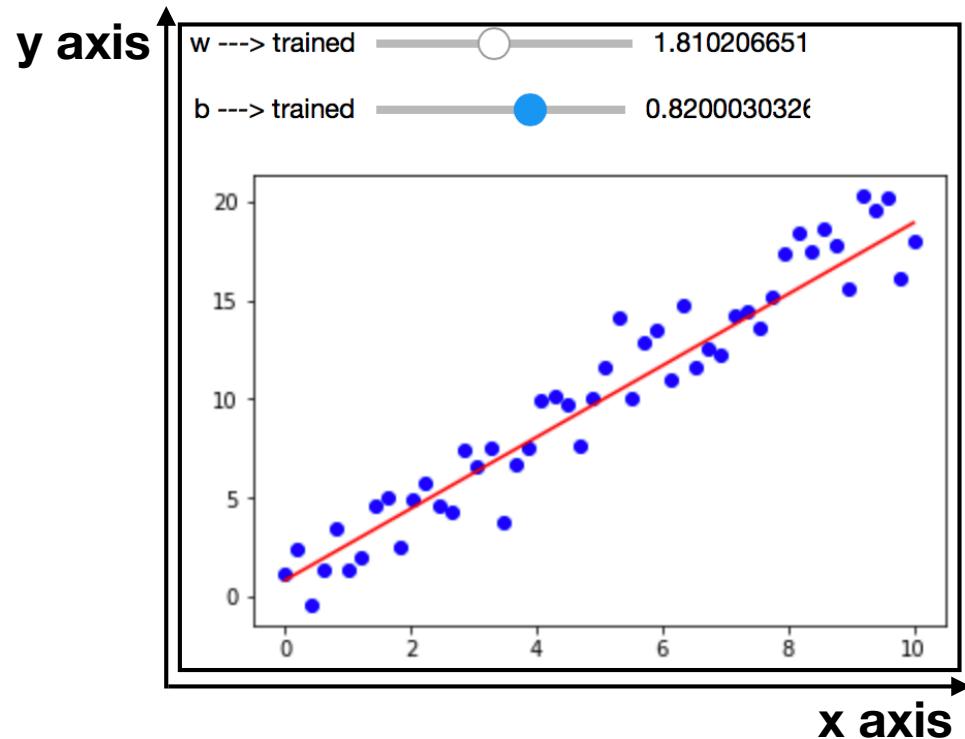
Least Square Regression

Least Square Regression



perceptrons를 이용한 least square regression graphical model

Least Square Regression



least square regression의 예시

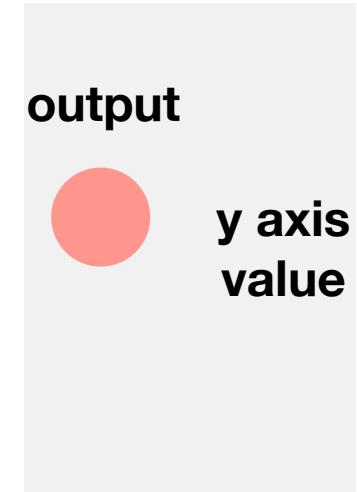
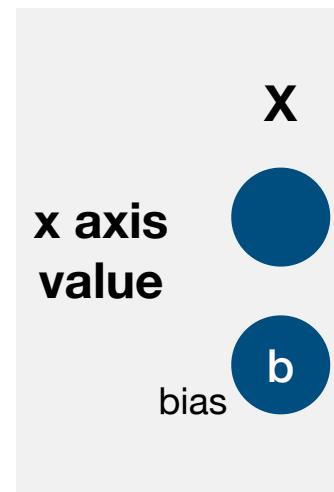
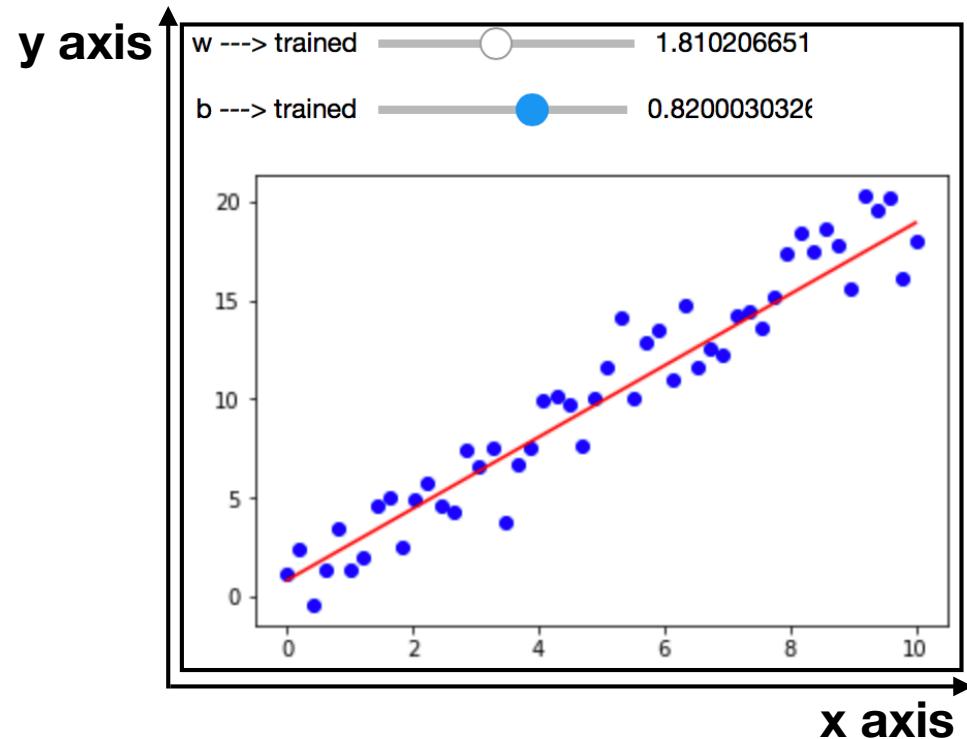
input: 파란 데이터 포인트의 x axis value

target: 파란 데이터 포인트의 y axis value

output: 빨간 선형 모델 상의 y axis value

목적: 파란 데이터를 가장 잘 나타내는 빨간 선형 모델을 그리는 것

Least Square Regression



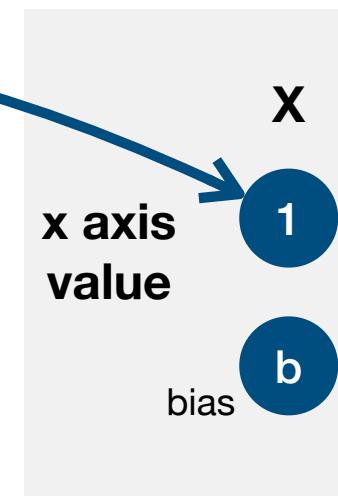
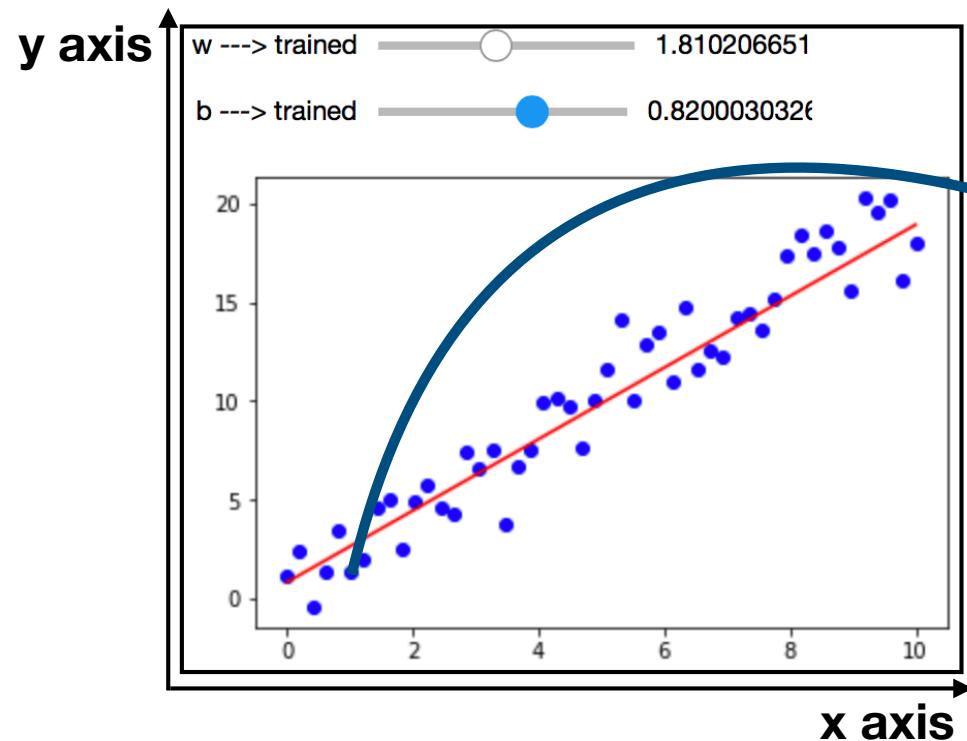
모델구조:

input layer (1 node),

output layer (1 node)

노드 숫자를 셀 때 bias node는 제외

Least Square Regression



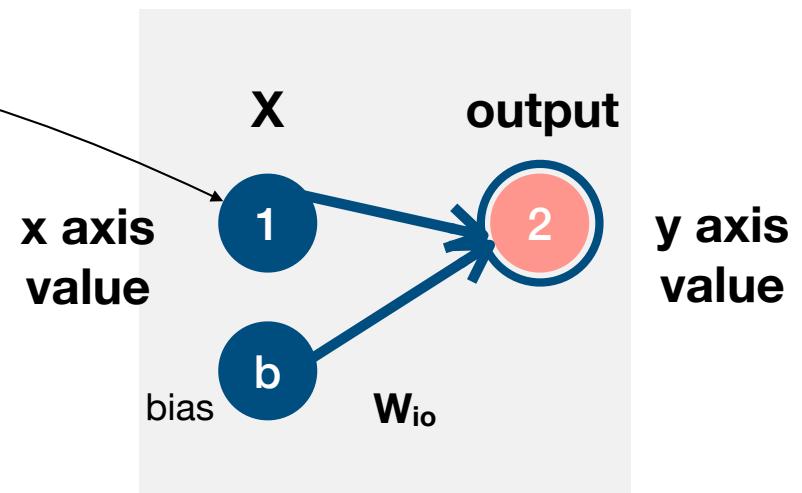
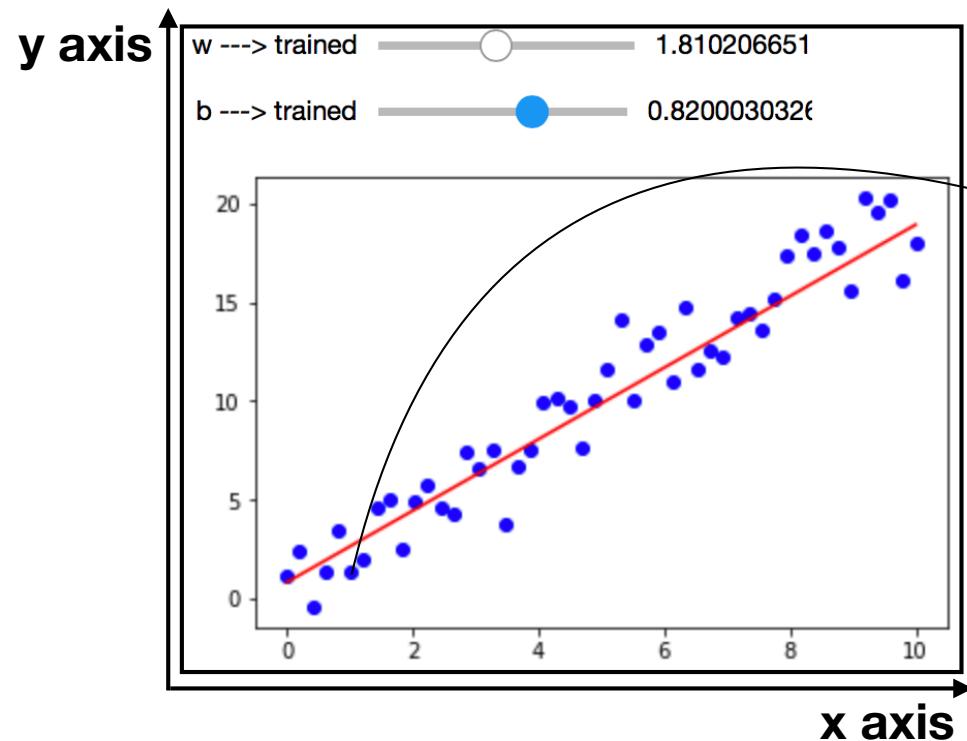
output
y axis value

w_{io}

훈련:

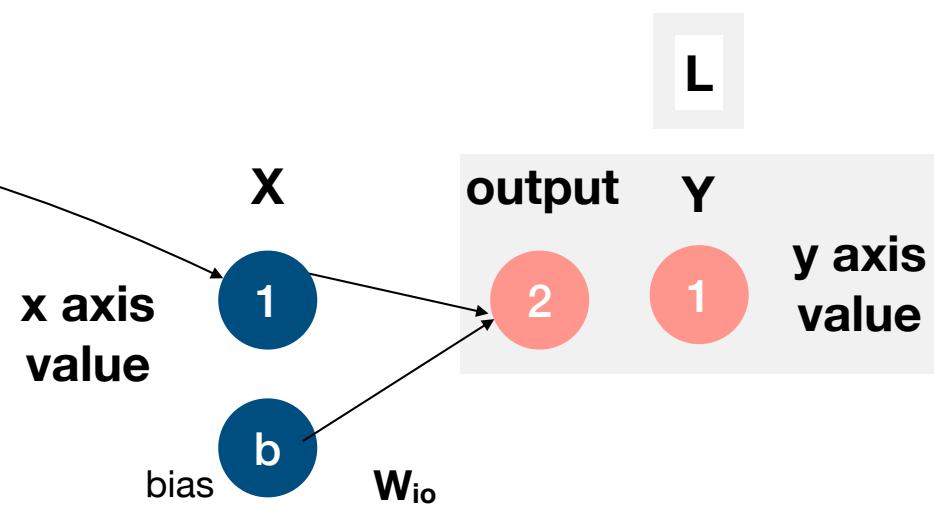
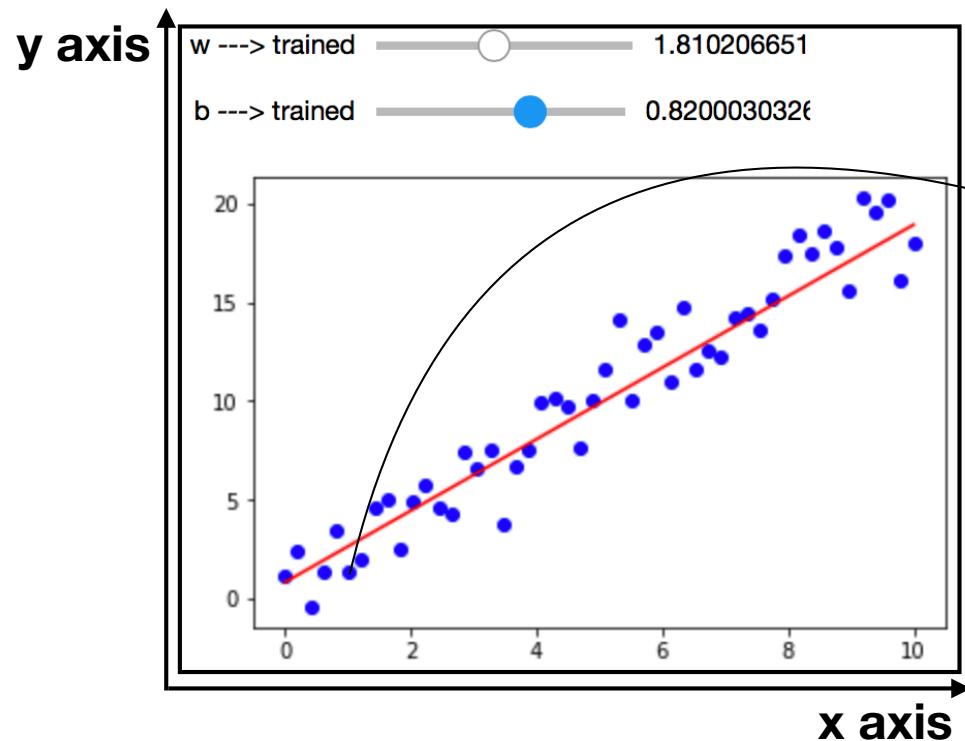
파란 데이터 포인트의 x axis value를 input layer의 node에 넣어줍니다.

Least Square Regression



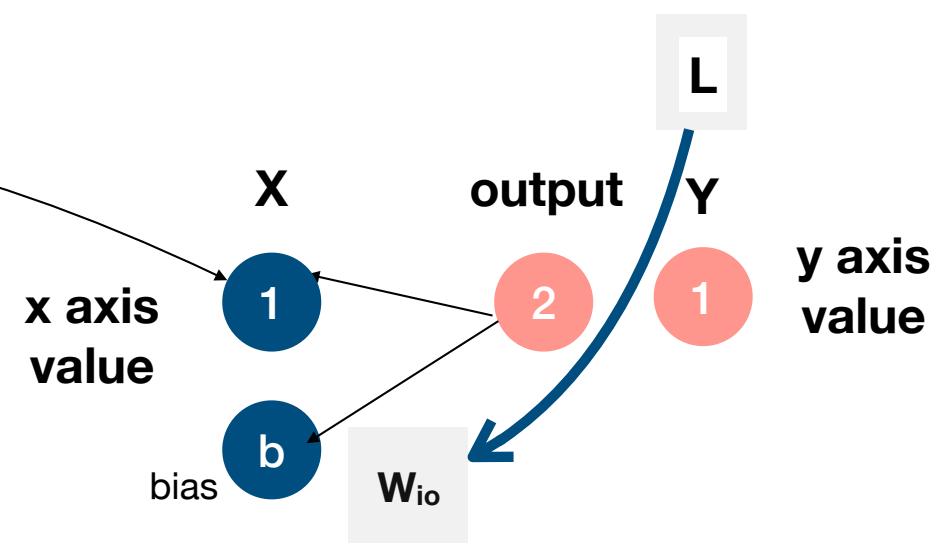
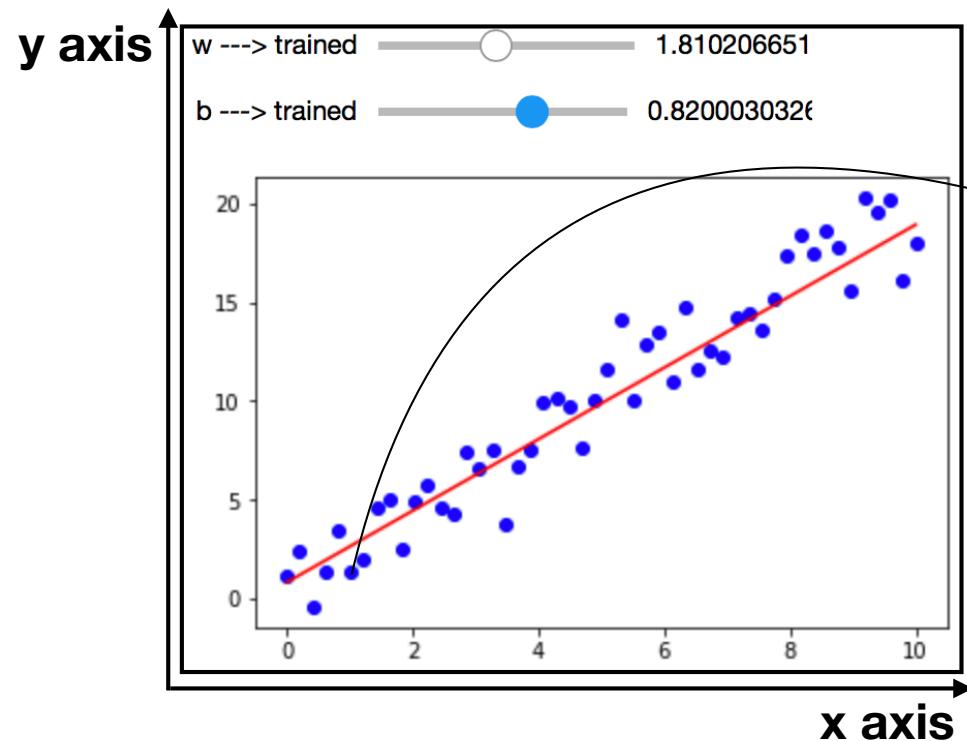
각 input layer의 node 값에 weight (진한 화살표)을 곱하고, 곱한 값을 모두 더해 output layer node에 넣어줍니다.

Least Square Regression



그렇게 구한 output과 정답 target (Y)을 비교하여 loss를 구합니다.

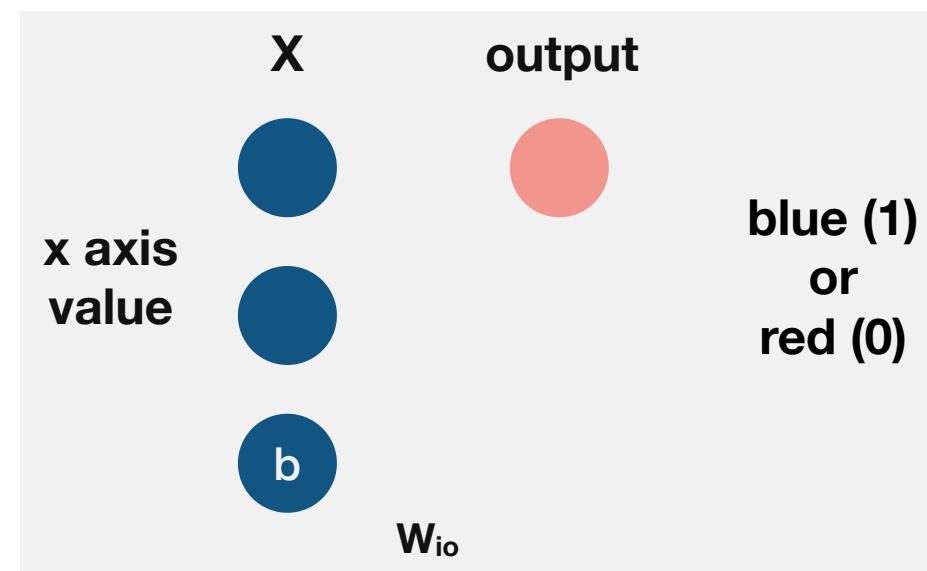
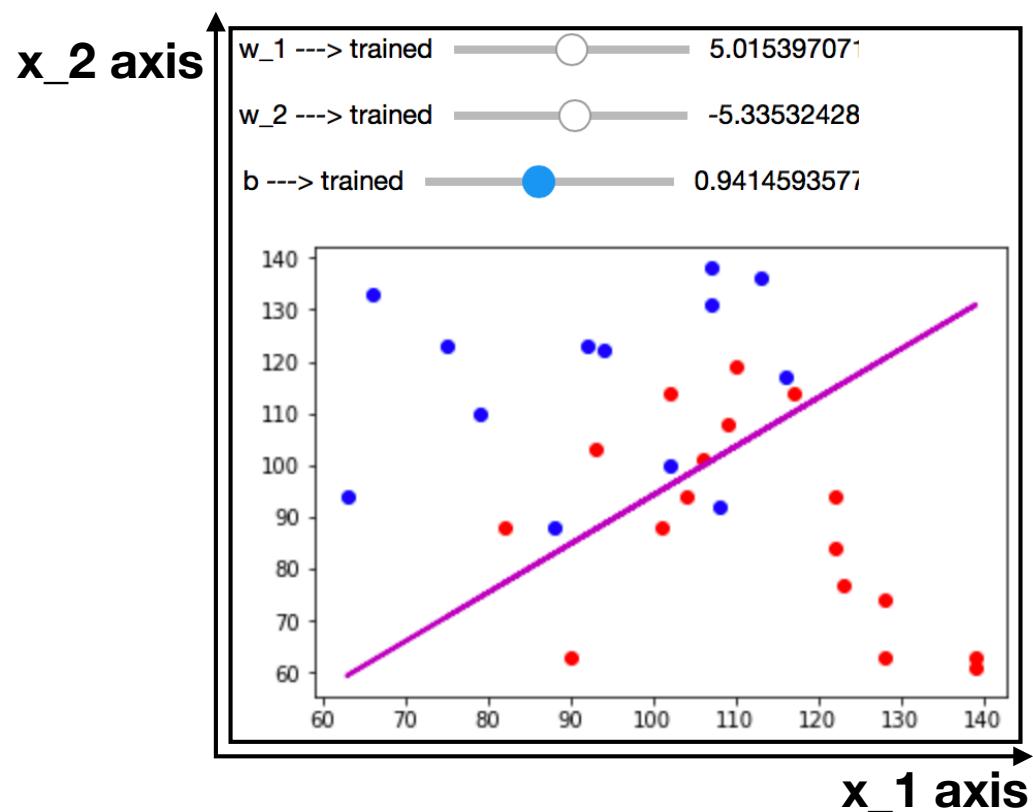
Least Square Regression



loss를 최소화하는 방향으로, weight를 update 합니다.

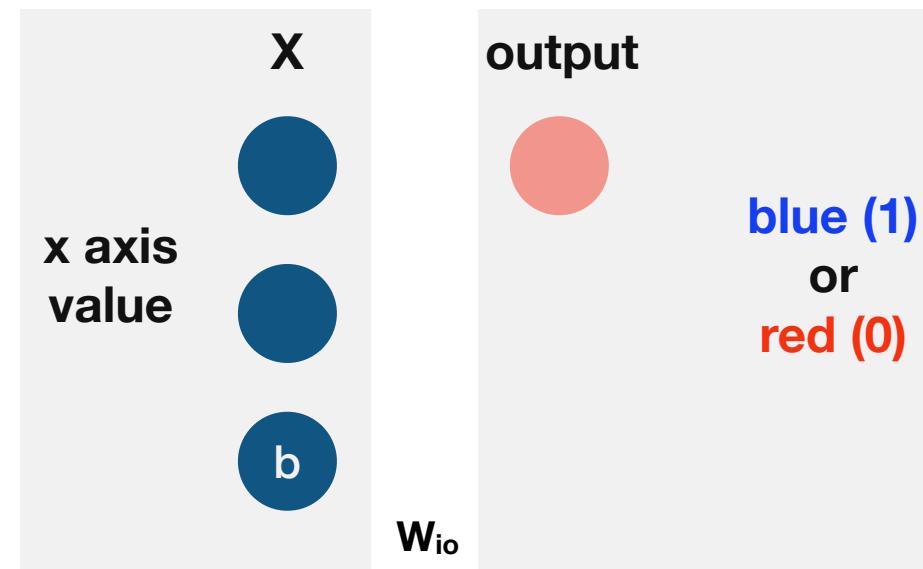
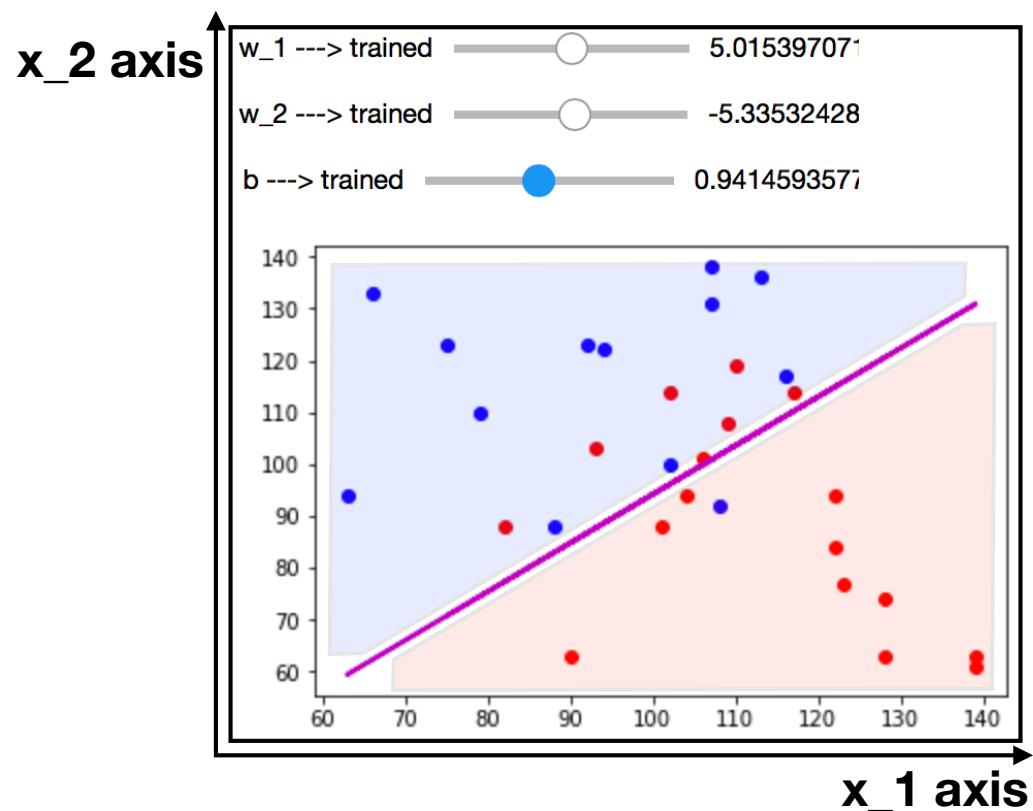
Logistic Regression

Logistic Regression



perceptrons를 이용한 logistic regression graphical model

Logistic Regression



logistic regression의 예시

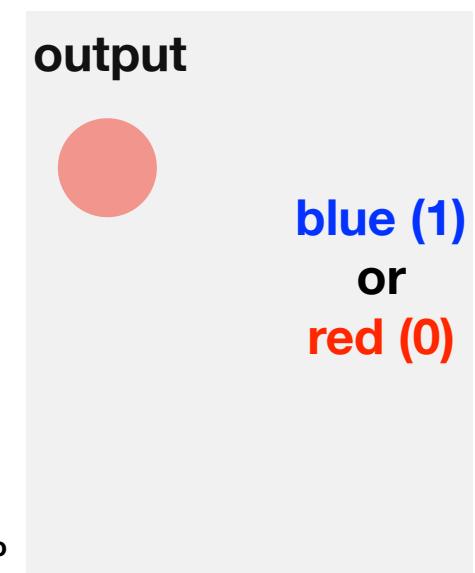
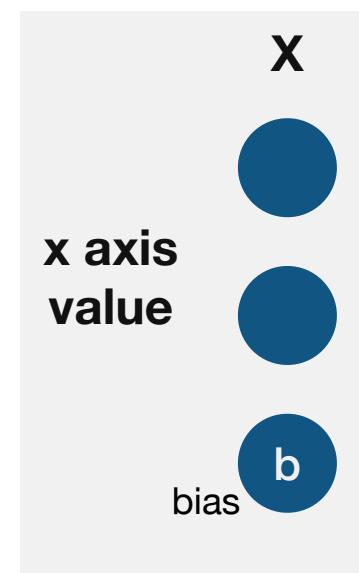
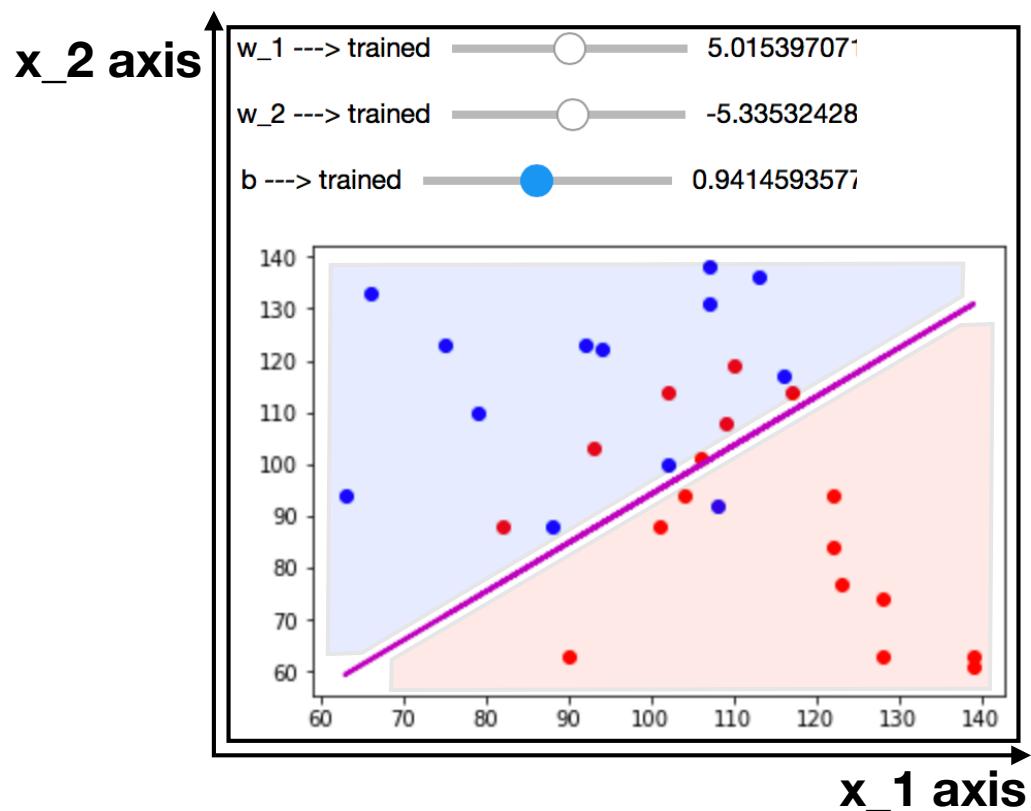
input: 데이터 포인트의 x_1 axis value, x_2 axis value

target: 데이터 포인트의 색깔, 파랑이면 1, 빨강이면 0

output: 선홍색 선형 모델을 기준으로 판별한 데이터 포인트의 색깔, 위는 파랑 1, 아래는 빨강 0

목적: 데이터를 가장 잘 구분하는 선형 모델을 그리는 것

Logistic Regression



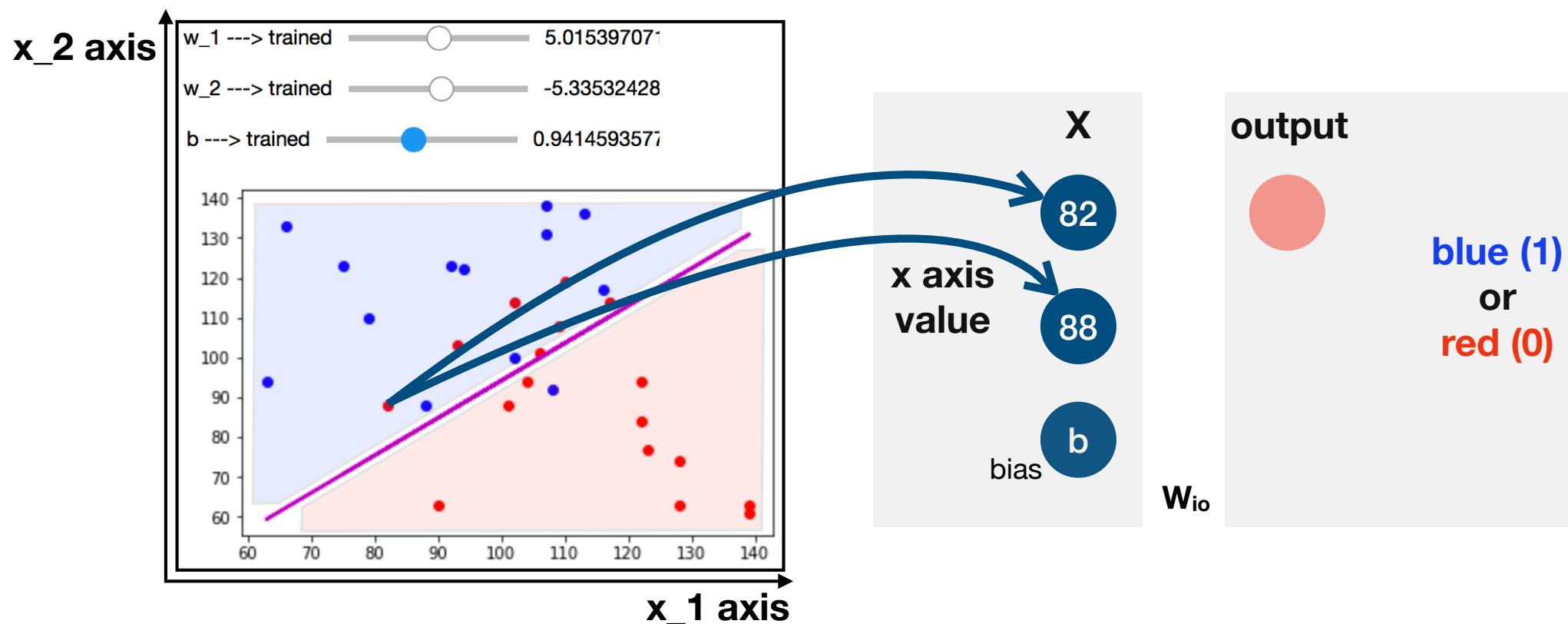
모델구조:

input layer (2 nodes),

output layer (1 node)

노드 숫자를 셀 때 bias node는 제외

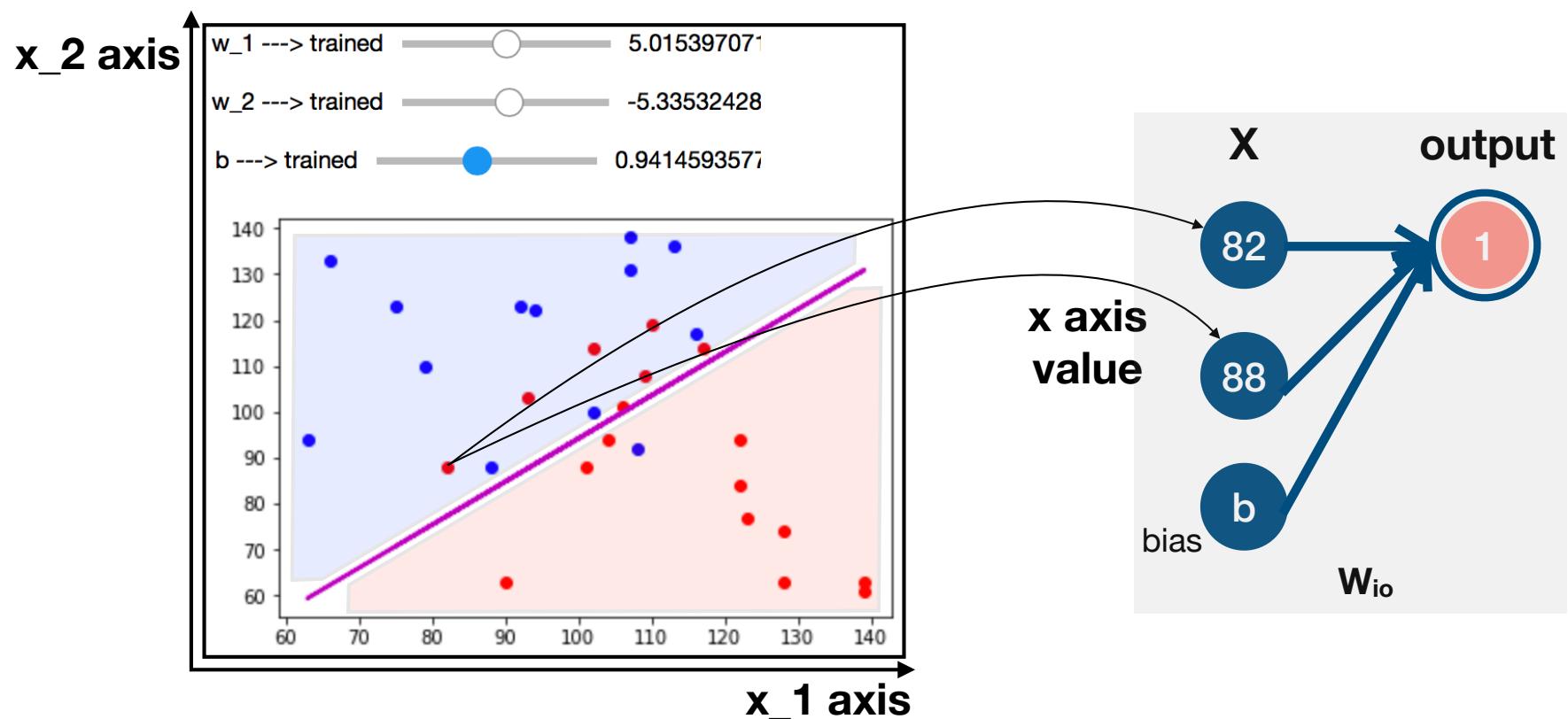
Logistic Regression



훈련:

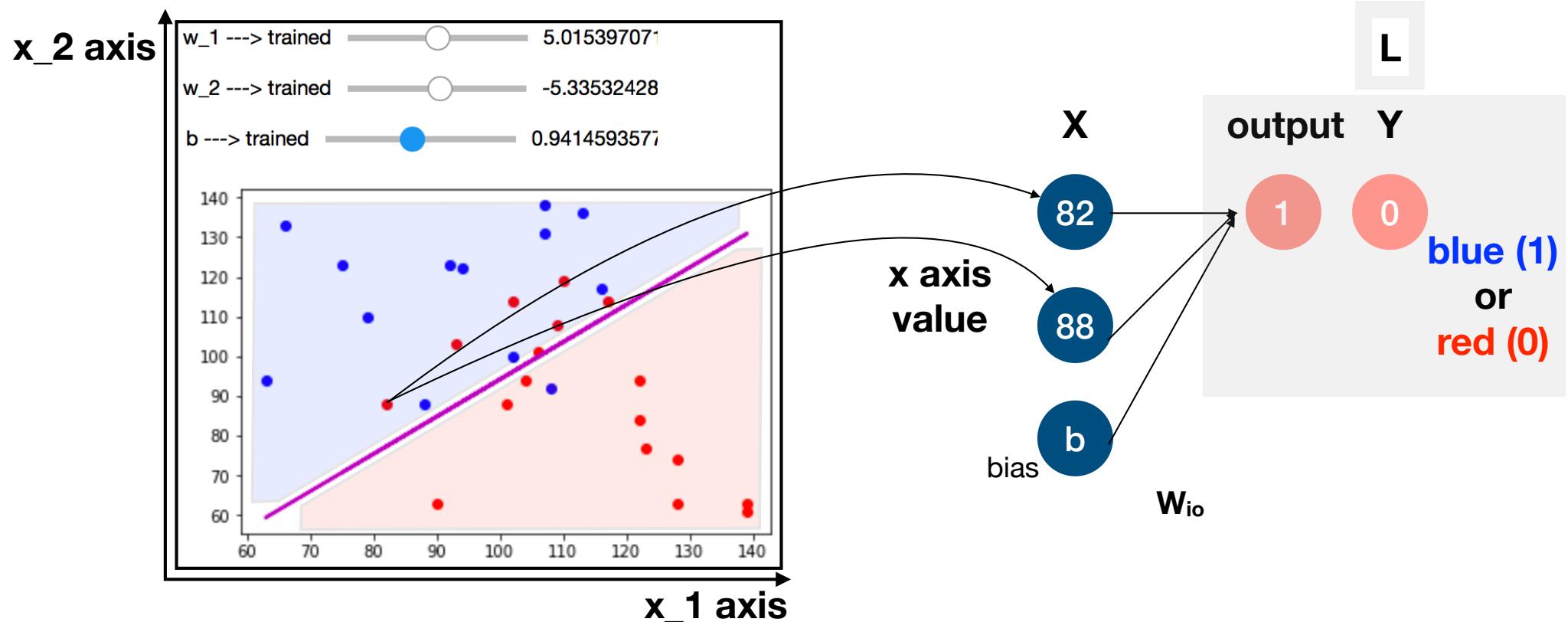
데이터 포인트의 x_1 axis value와 x_2 axis value를
input layer의 nodes에 각각 넣어줍니다.

Logistic Regression



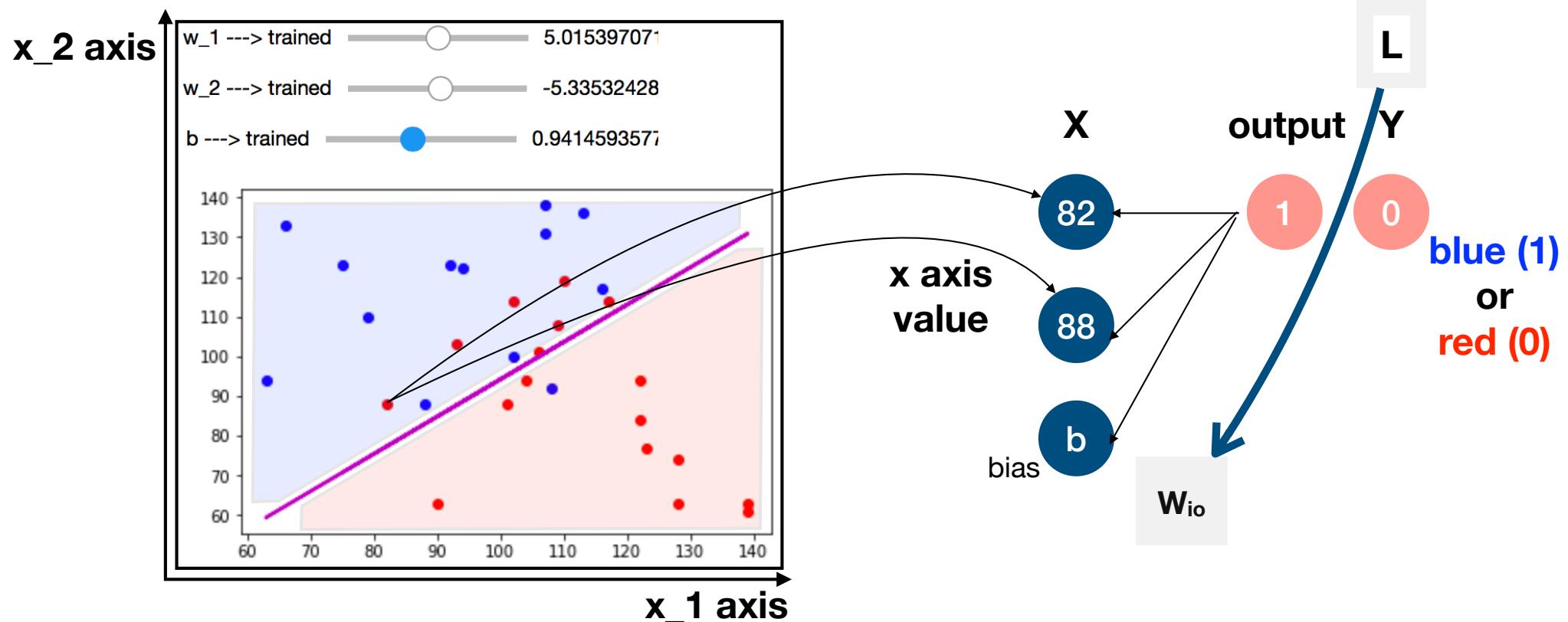
각 input layer의 node 값에 weight (진한 화살표)을 곱하고, 곱한 값을 모두 더해 output layer node에 넣어줍니다.

Logistic Regression



그렇게 구한 output과 정답 target (Y)을 비교하여 loss를 구합니다.

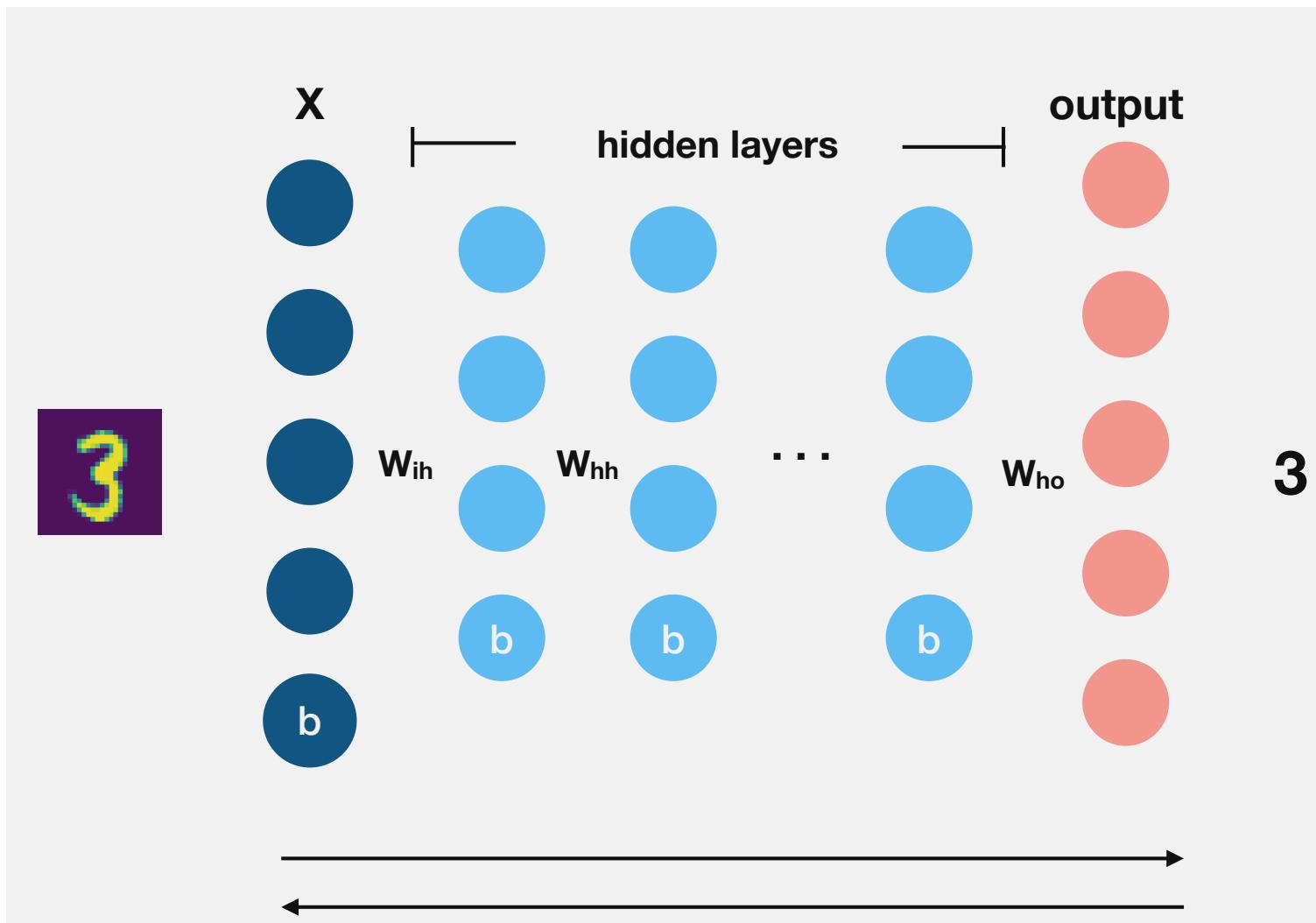
Logistic Regression



loss를 최소화하는 방향으로, weight를 update 합니다.

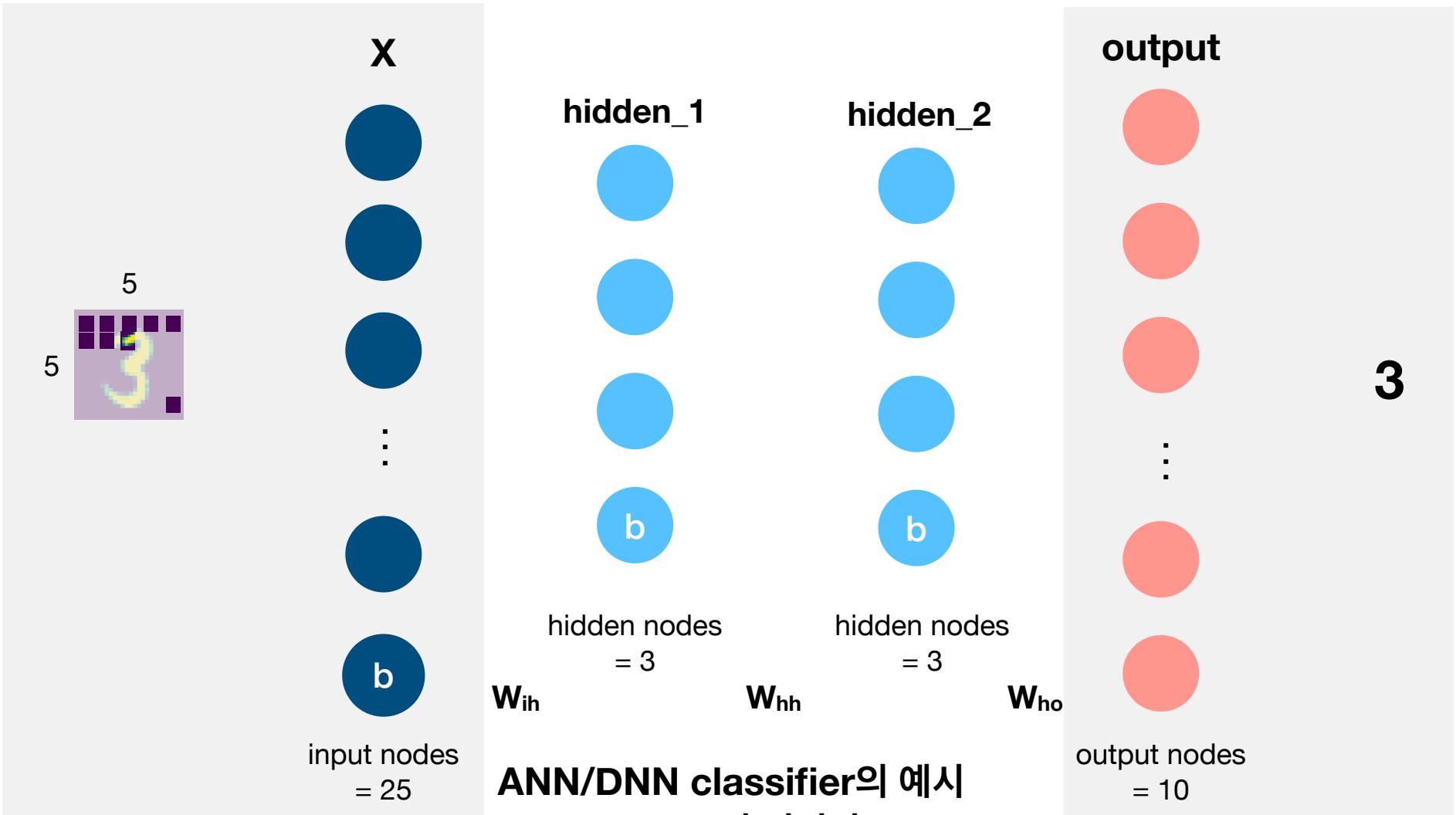
ANN / DNN

ANN DNN



perceptrons를 이용한 ANN/DNN classifier graphical model

ANN DNN



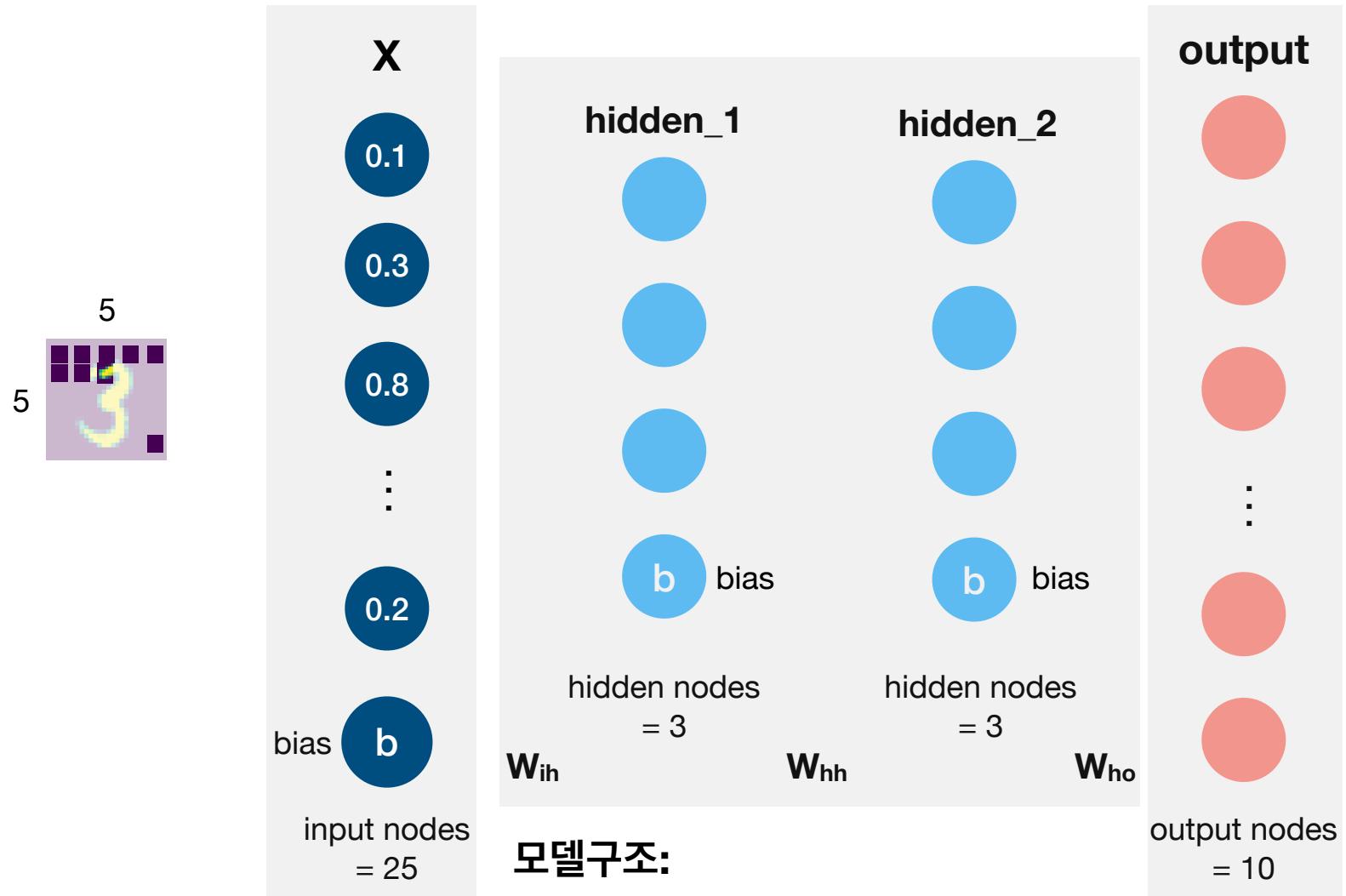
ANN/DNN classifier의 예시

input: 5 by 5의 이미지(X)

target: 0에서 9까지의 라벨(Y)

목적: input이 0~9 중 어떤 숫자인지 분류

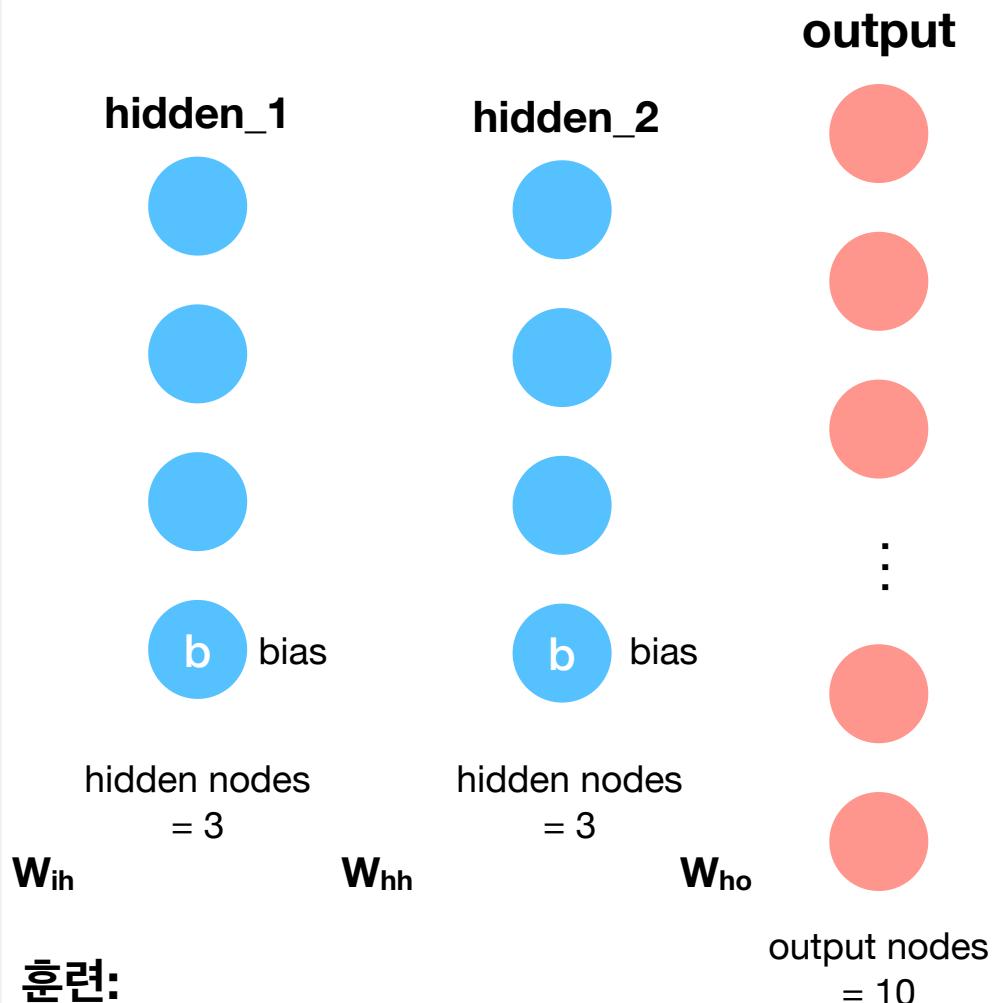
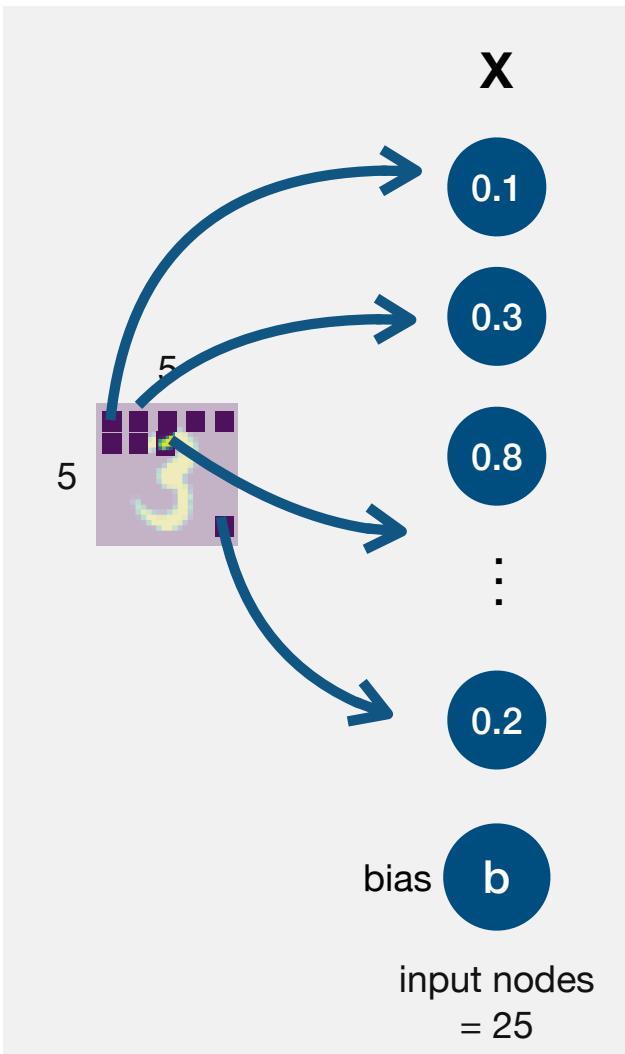
ANN DNN



모델구조:

**input layer (25 nodes),
2 hidden layers (3 nodes each),
output layer (10 nodes)**
노드 숫자를 셀 때 bias node는 제외

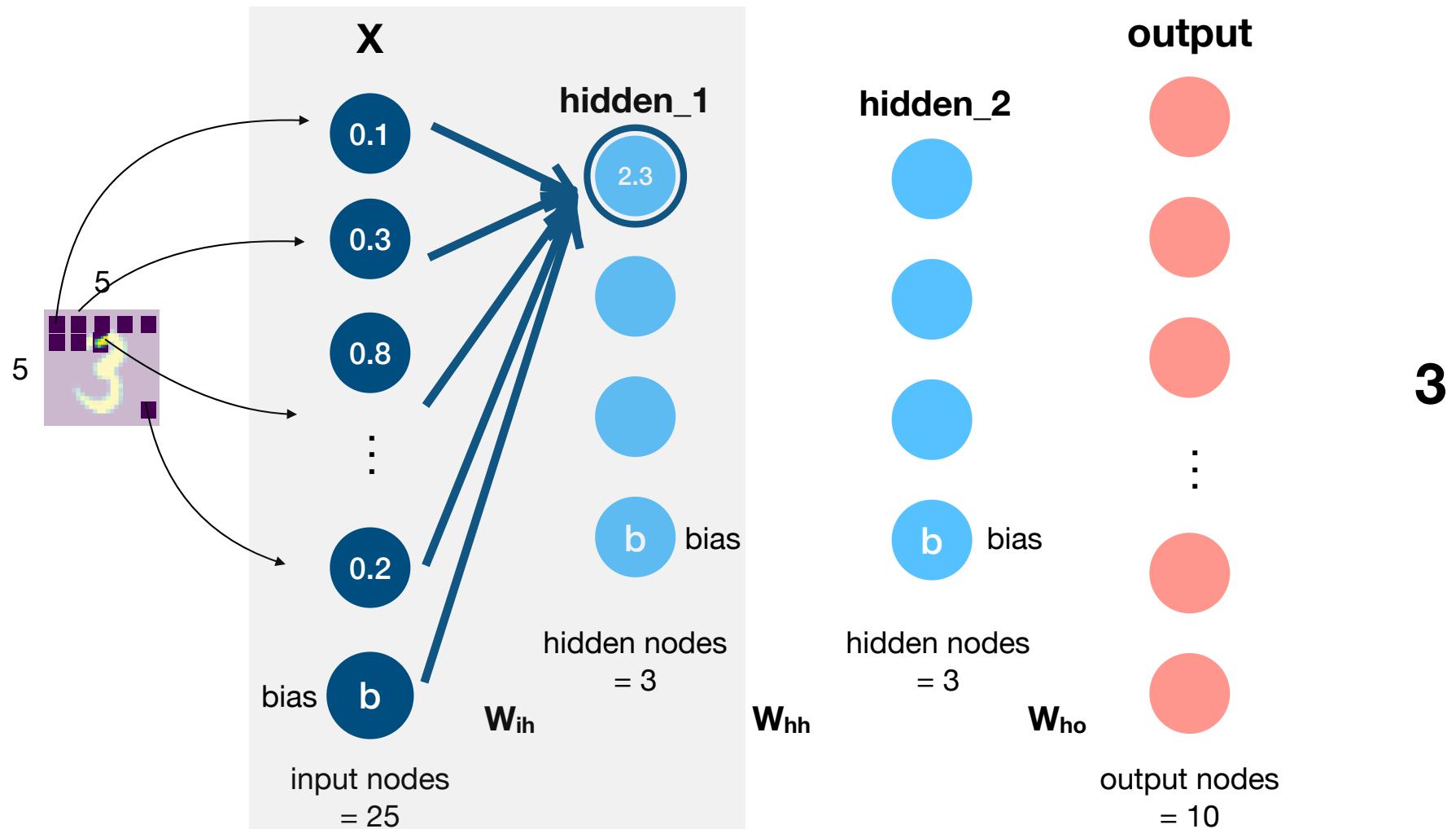
ANN DNN



훈련:

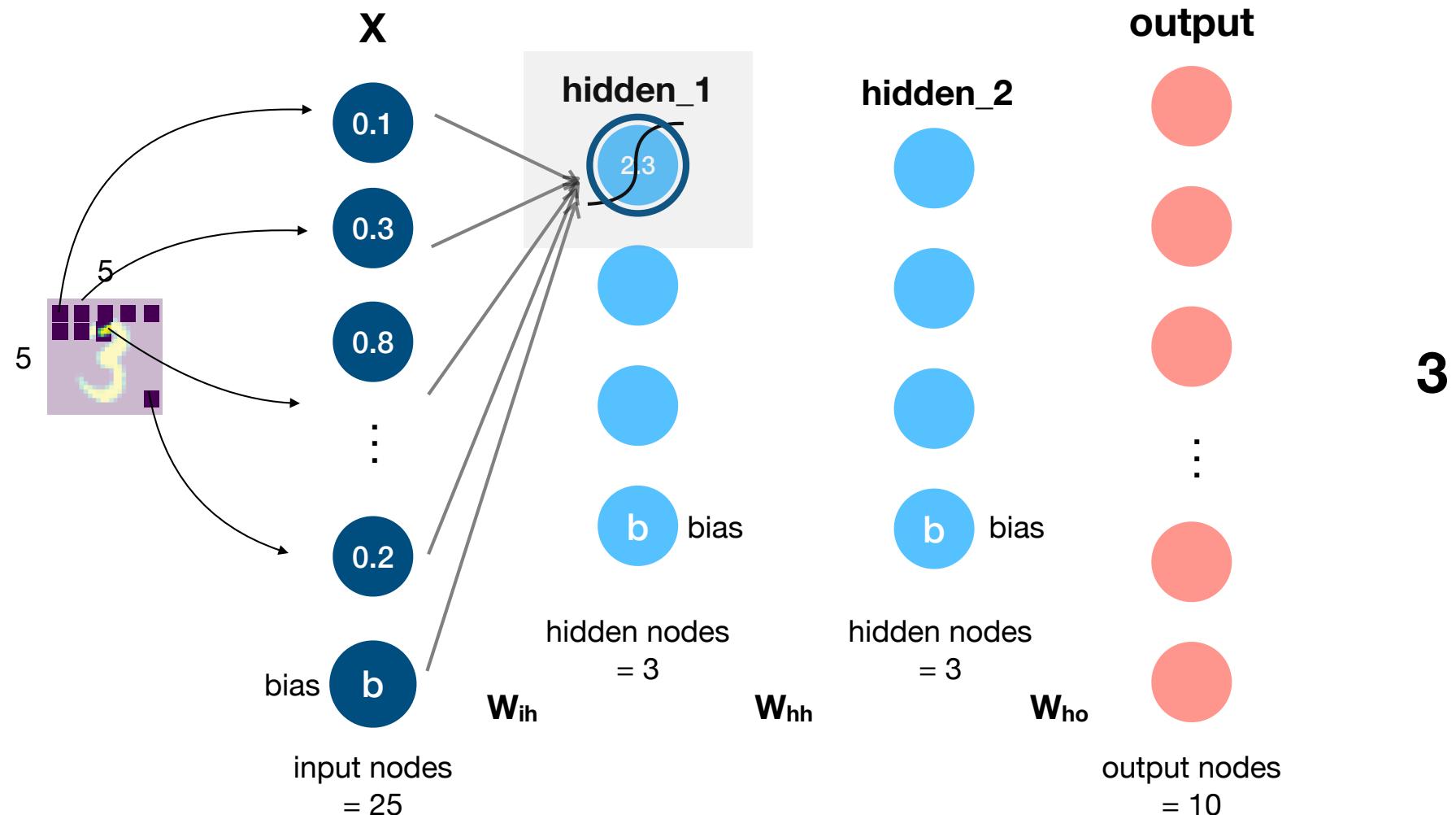
이미지를 pixel 단위로 쪼개서
그 값을 각각 input layer의 nodes에 넣어줍니다.

ANN DNN



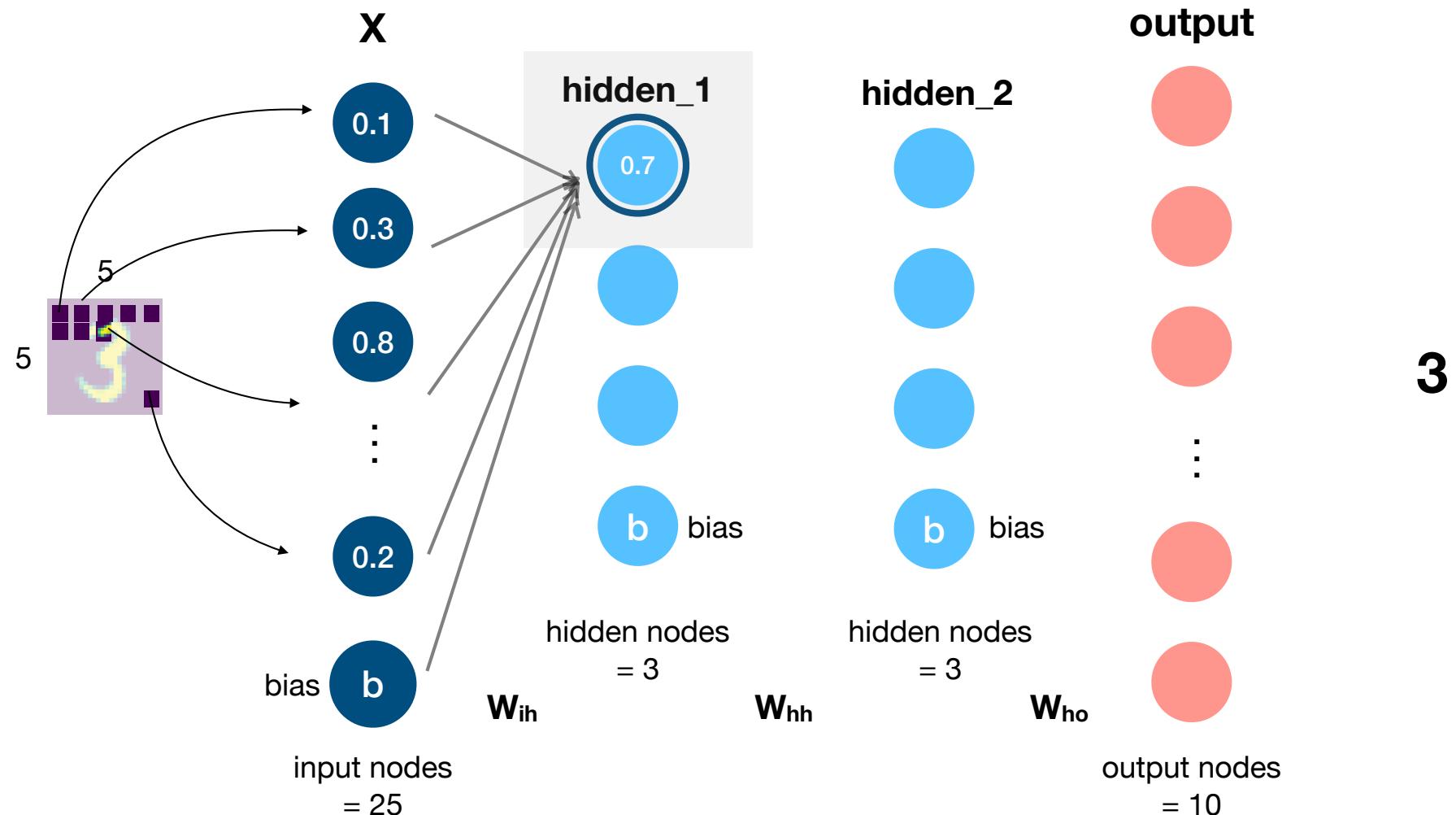
각 input layer의 node 값에 weight (진한 화살표)을 곱하고, 곱한 값을 모두 더해 첫 번째 hidden layer node에 넣어줍니다.

ANN DNN



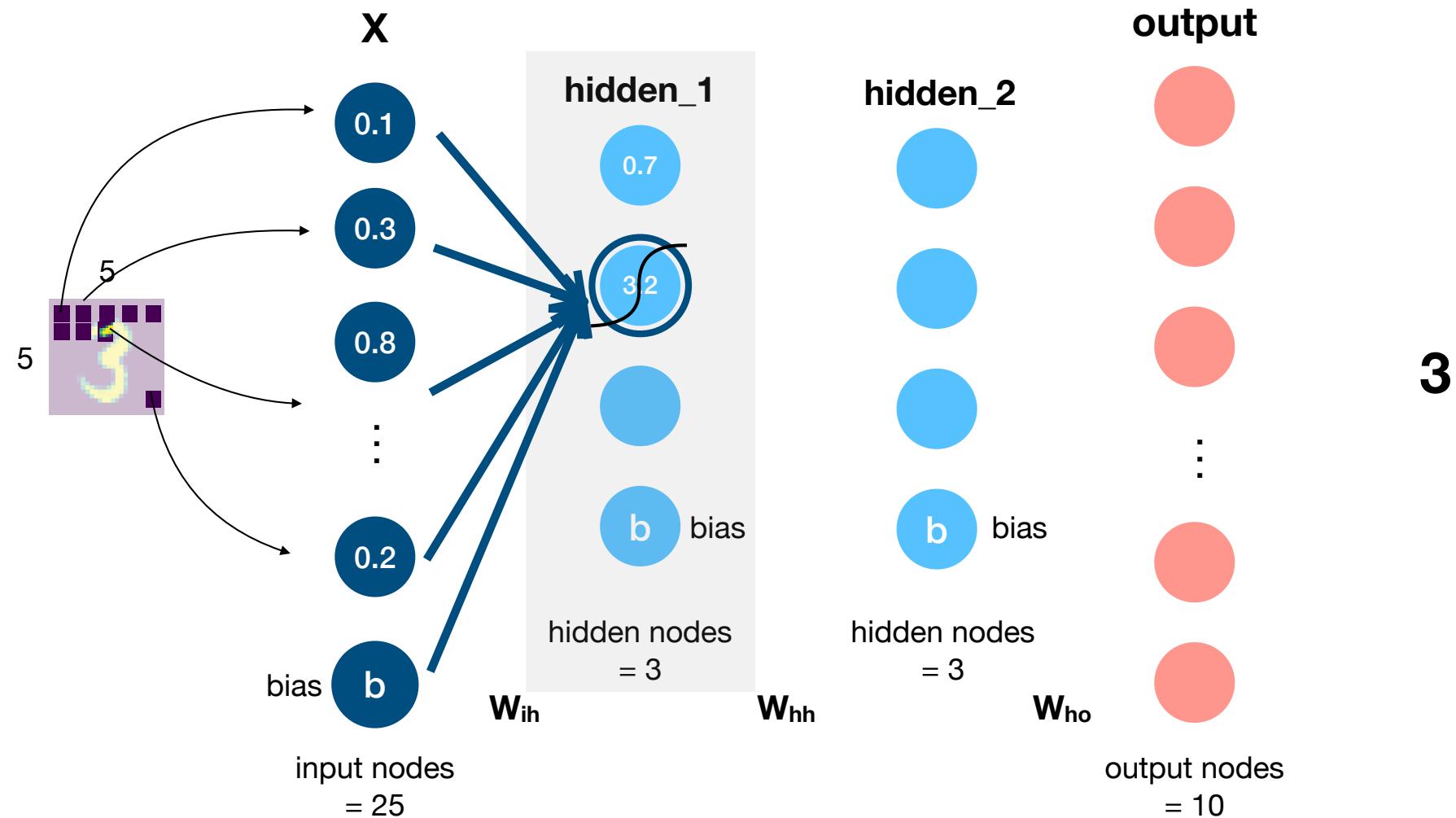
activation function을 통해서 node 값을
0과 1 사이의 값(sigmoid function)으로 바꿔줍니다.

ANN DNN



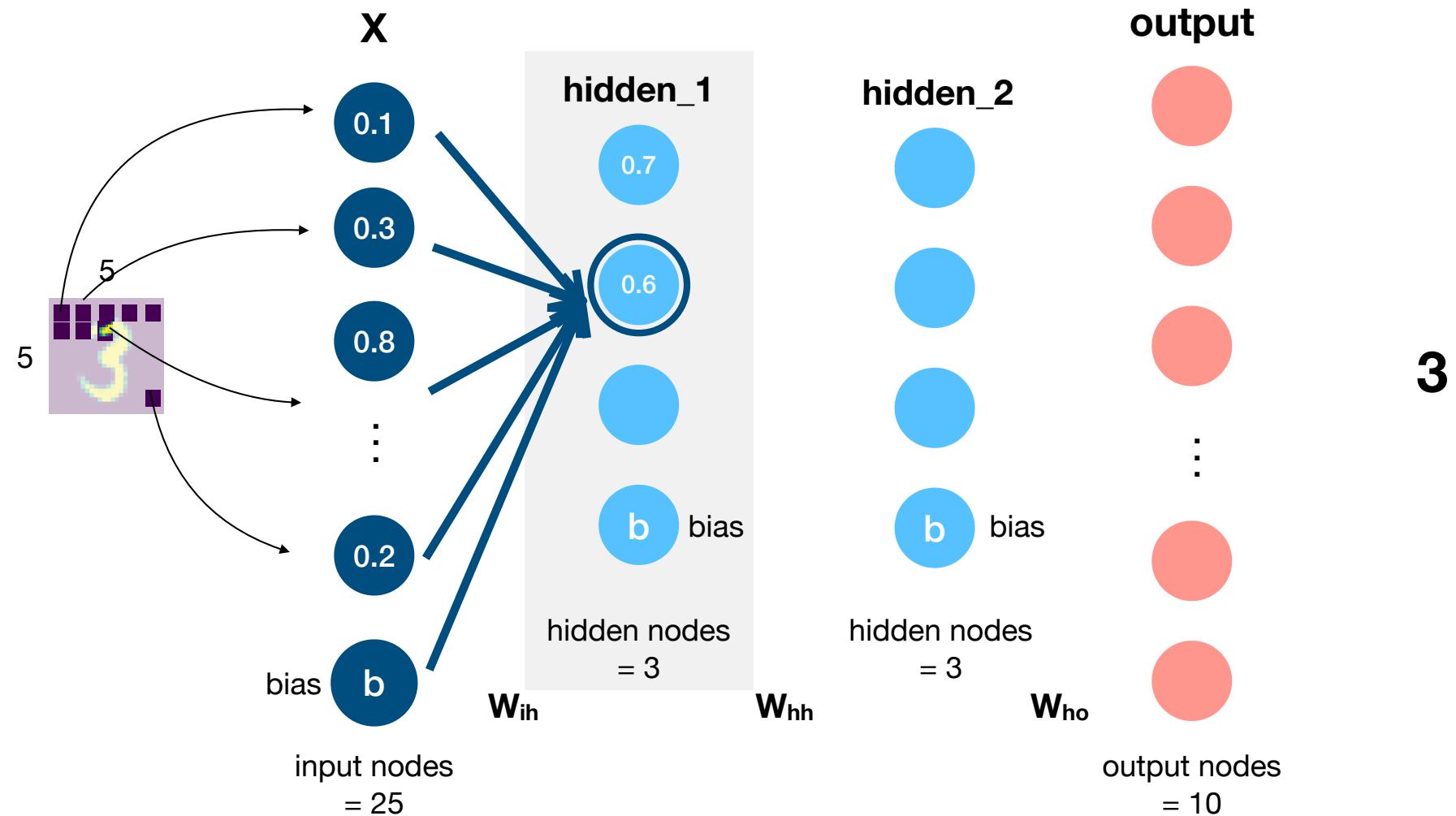
activation function을 통해서 node 값을
0과 1 사이의 값(sigmoid function)으로 바꿔줍니다.

ANN DNN



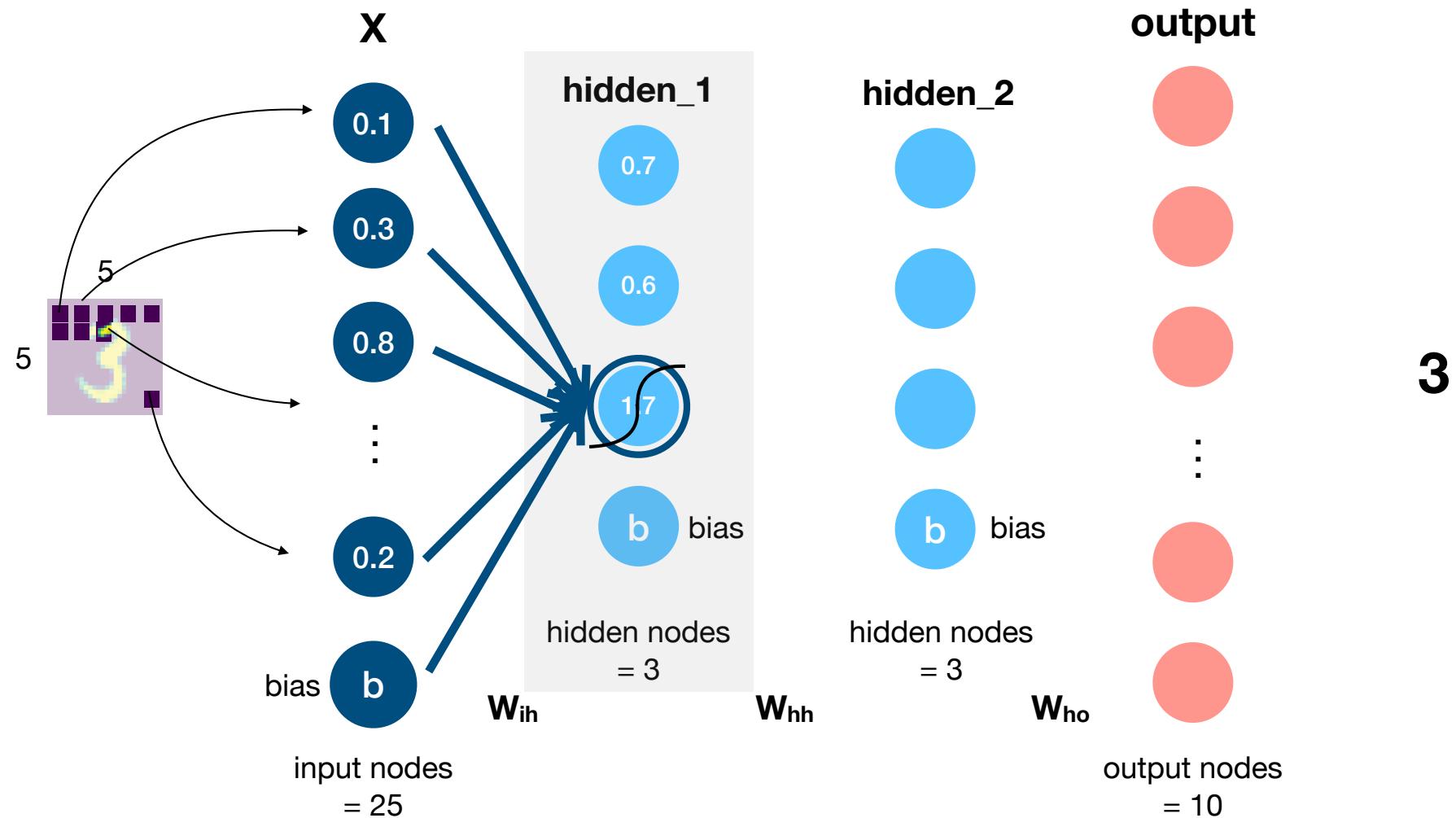
모든 hidden layer 1의 nodes에 대해 반복합니다.

ANN DNN



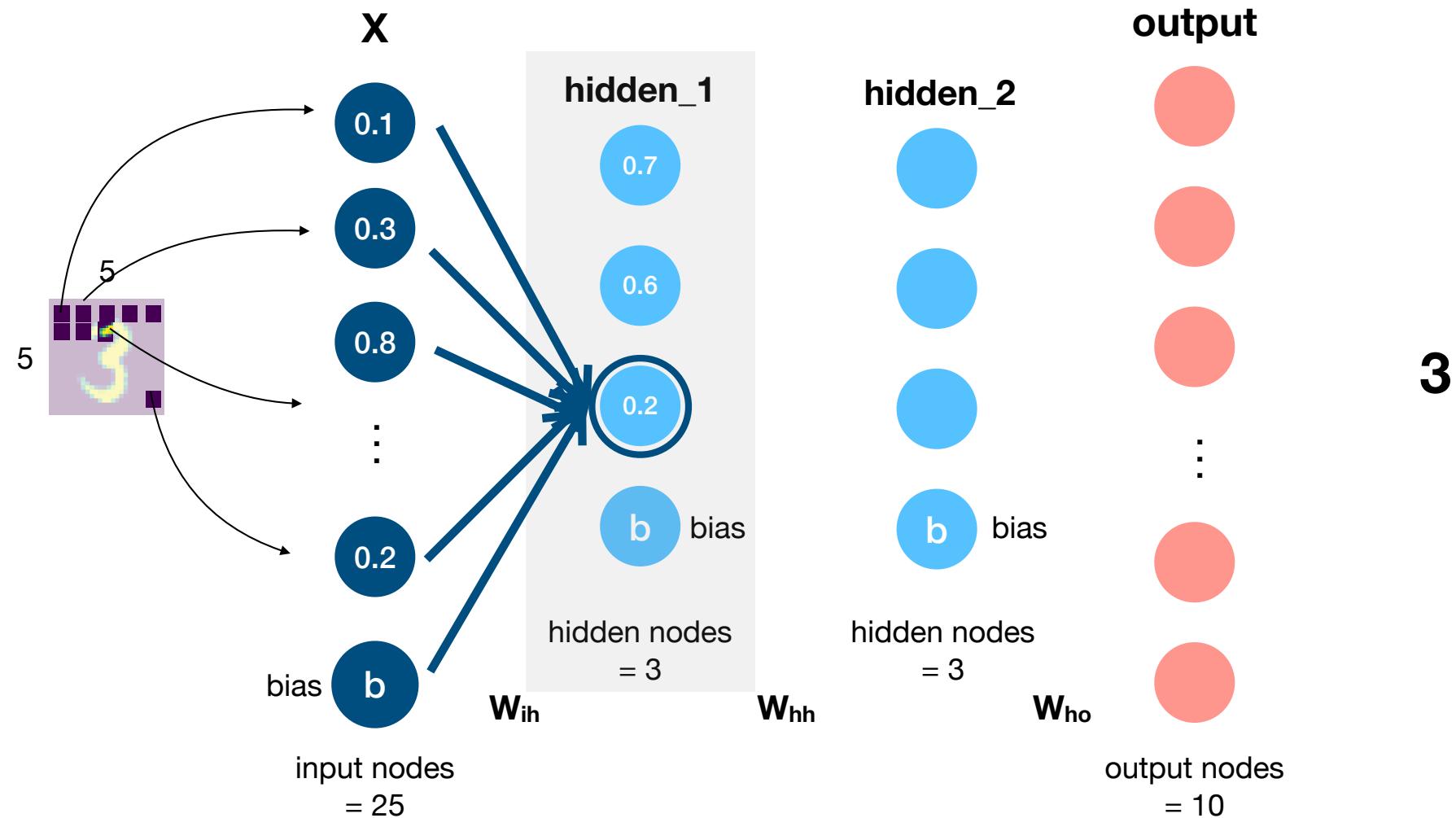
모든 hidden layer 1의 nodes에 대해 반복합니다.

ANN DNN



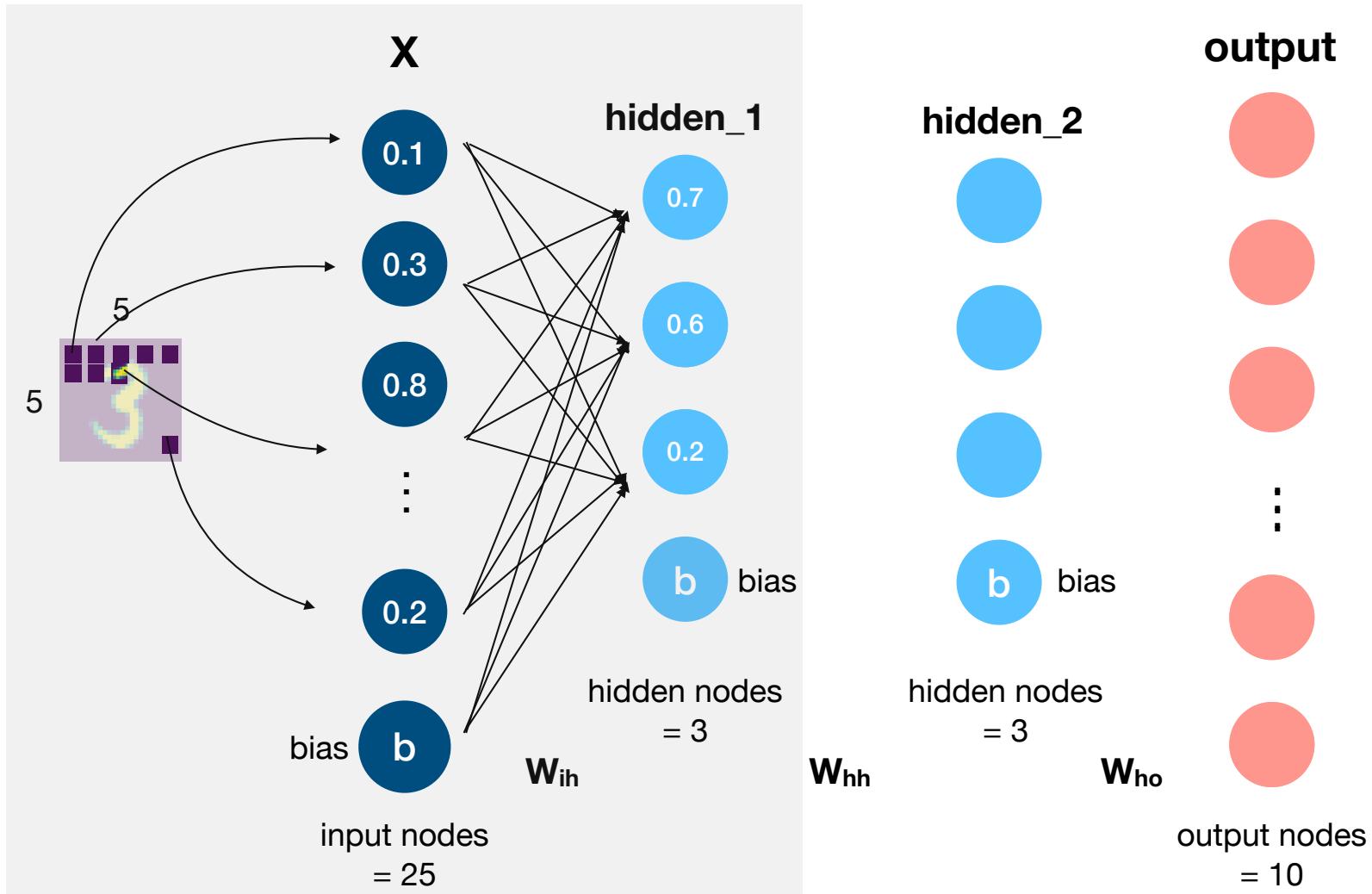
모든 hidden layer 1의 nodes에 대해 반복합니다.

ANN DNN



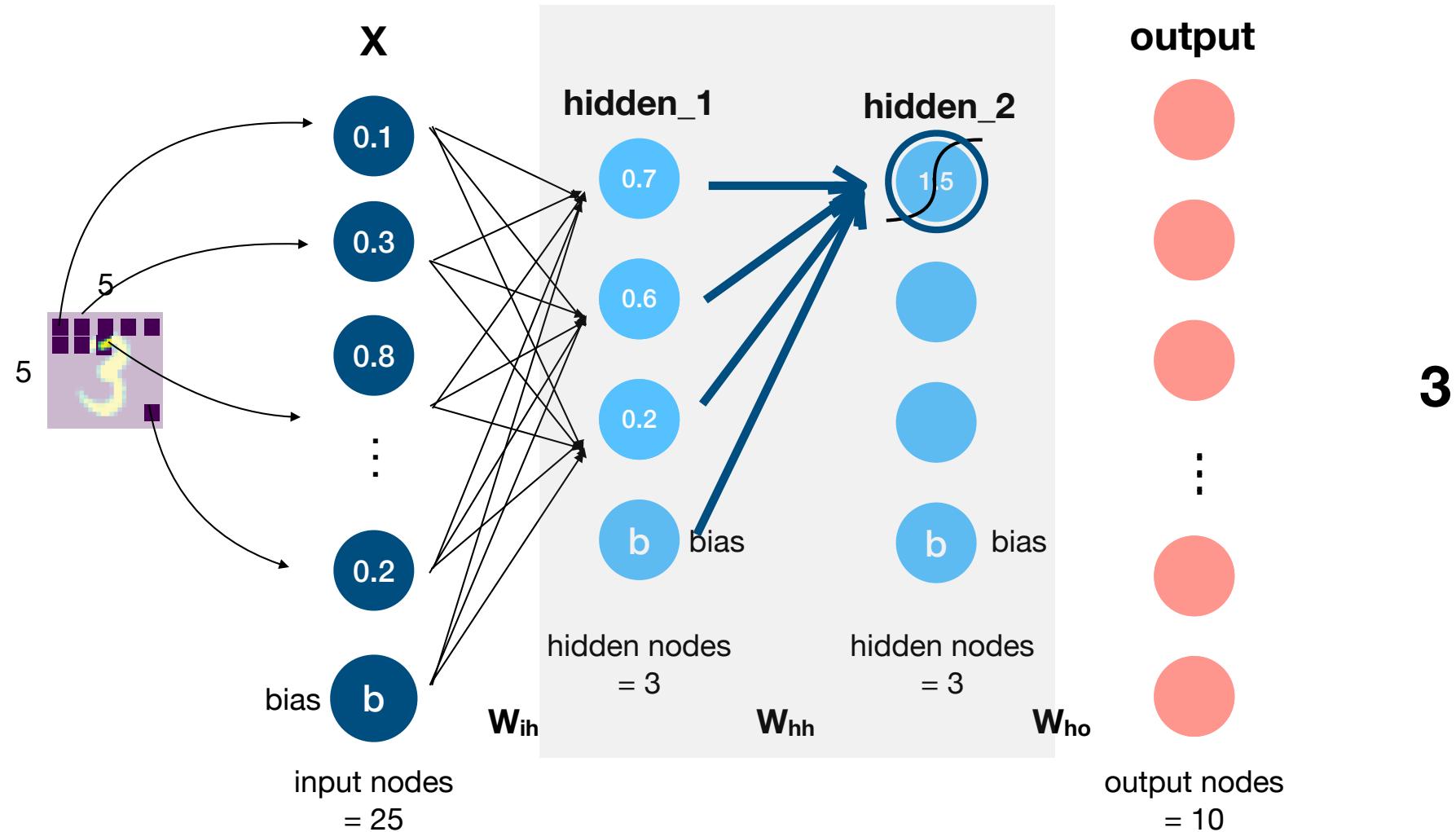
모든 hidden layer 1의 nodes에 대해 반복합니다.

ANN DNN



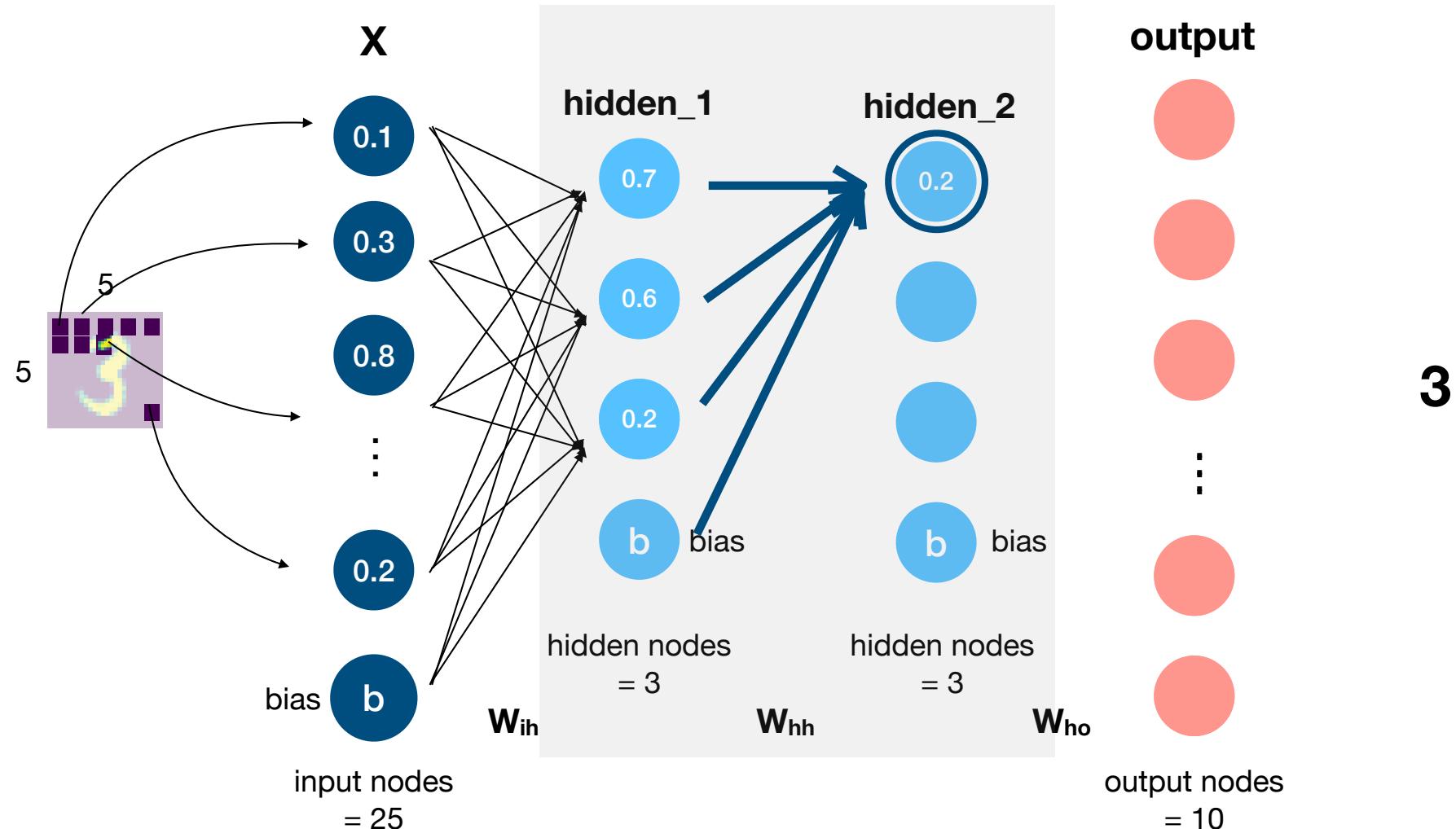
hidden layer 1까지의 nodes가 모두 계산되었습니다.
이제 hidden layer 2로 넘어갈 차례입니다.

ANN DNN



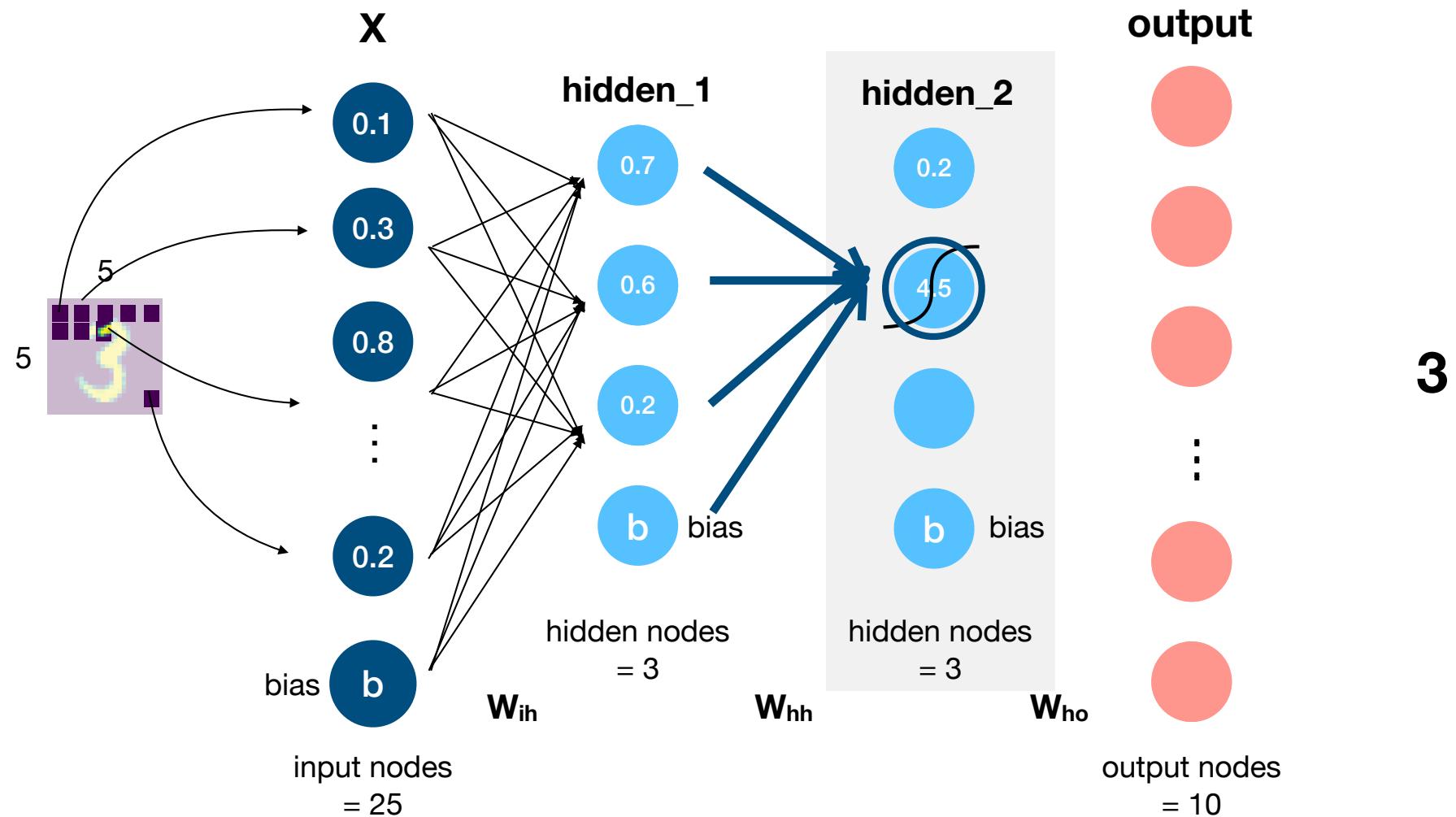
전과 같이, 각 hidden layer 1의 node 값에 weight (진한 화살표)을 곱하고 모두 더해 activation function을 건 후 hidden layer 2의 첫 번째 node에 넣어줍니다.

ANN DNN



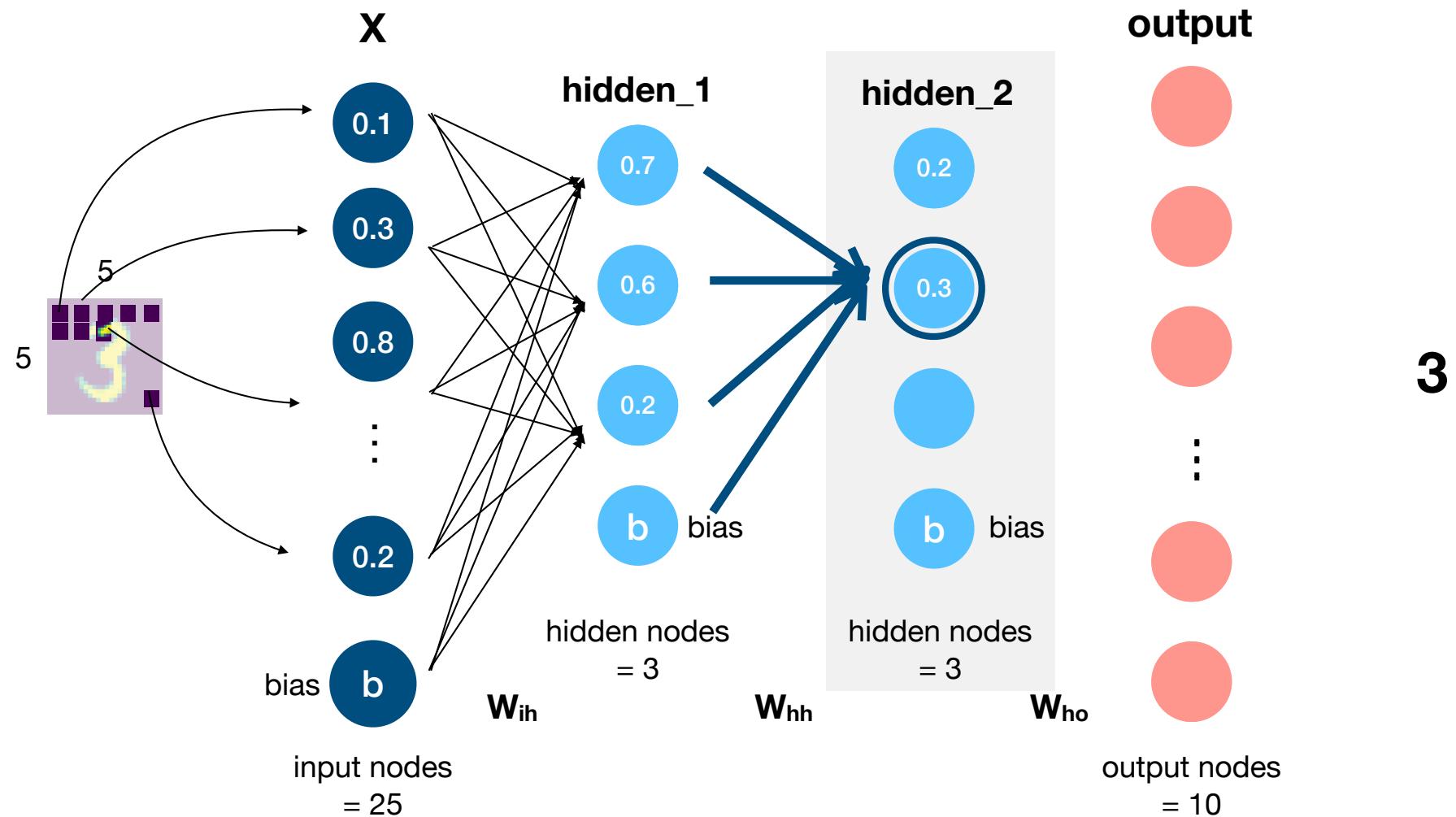
전과 같이, 각 hidden layer 1의 node 값에 weight (진한 화살표)을 곱하고 모두 더해 activation function을 건 후 hidden layer 2의 첫 번째 node에 넣어줍니다.

ANN DNN



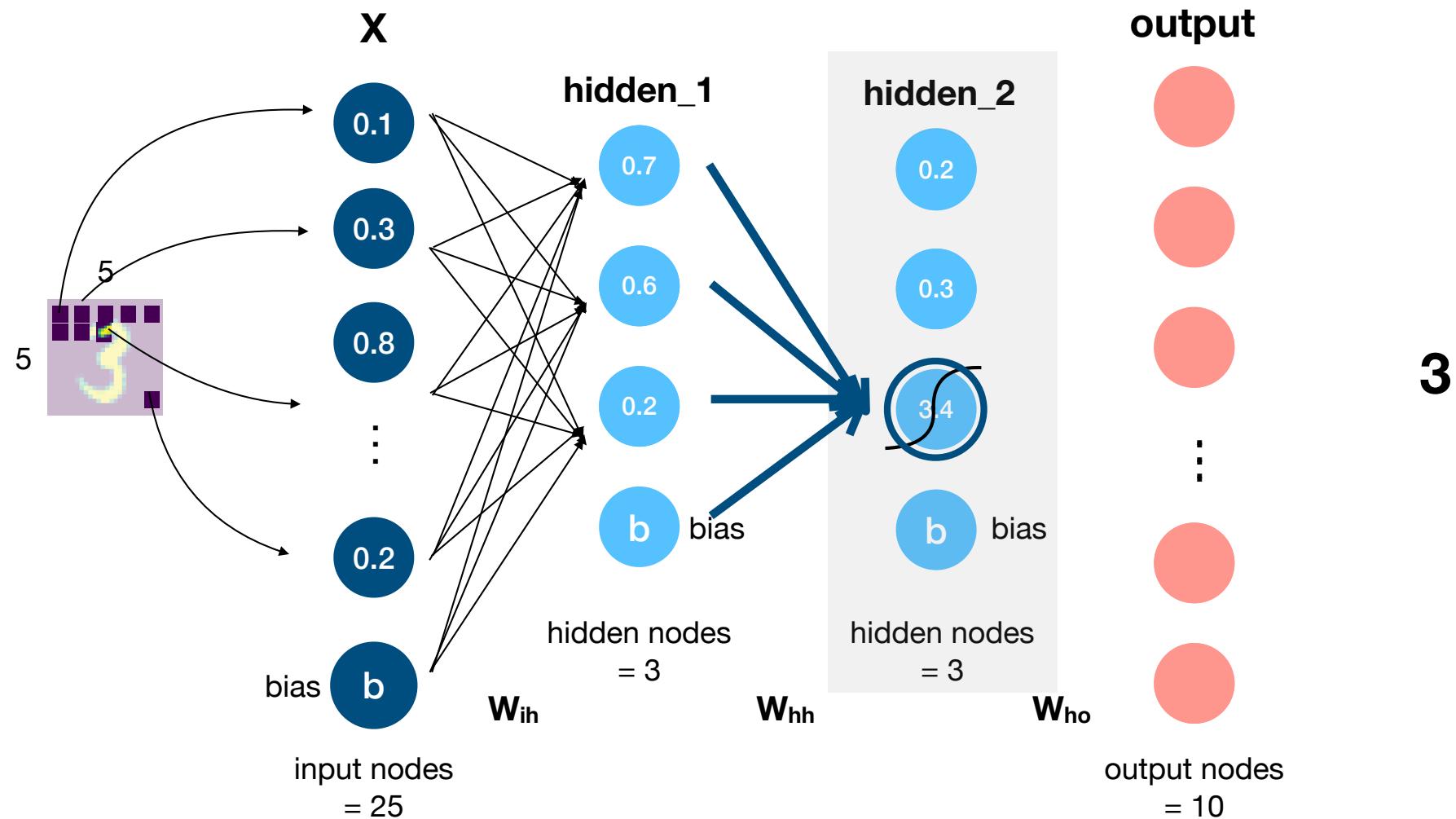
모든 hidden layer 2의 nodes에 대해 반복합니다.

ANN DNN



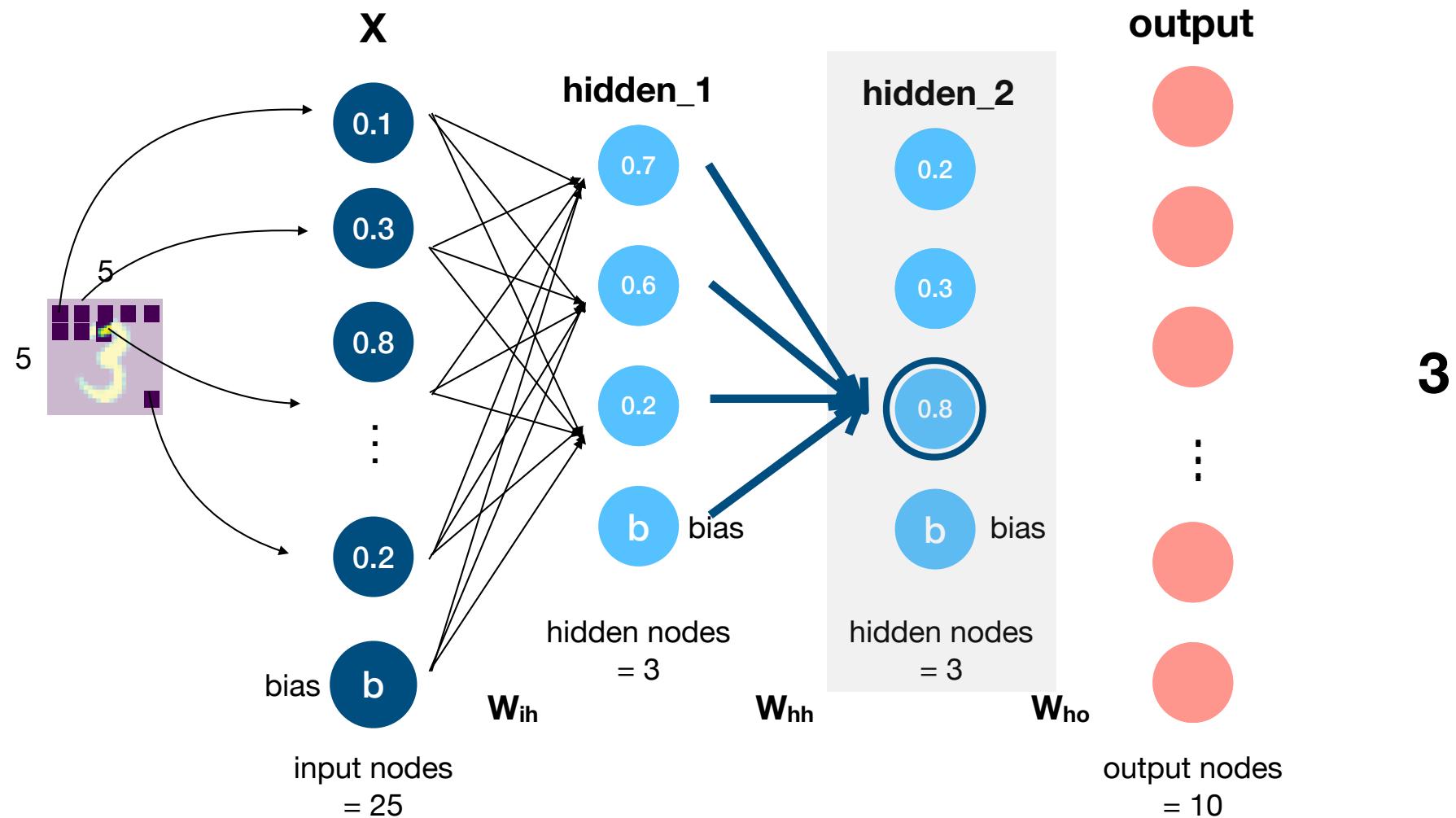
모든 hidden layer 2의 nodes에 대해 반복합니다.

ANN DNN



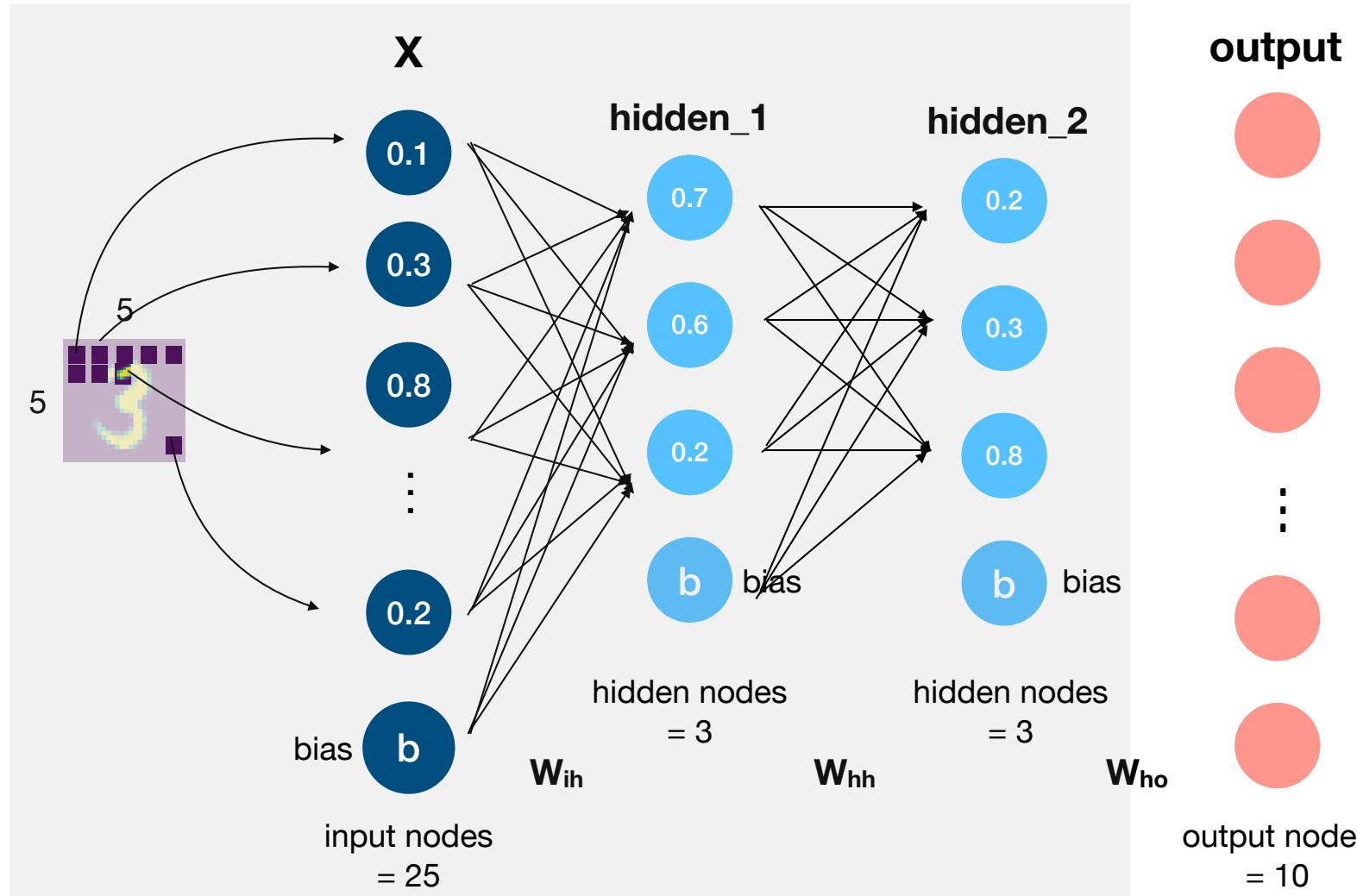
모든 hidden layer 2의 nodes에 대해 반복합니다.

ANN DNN



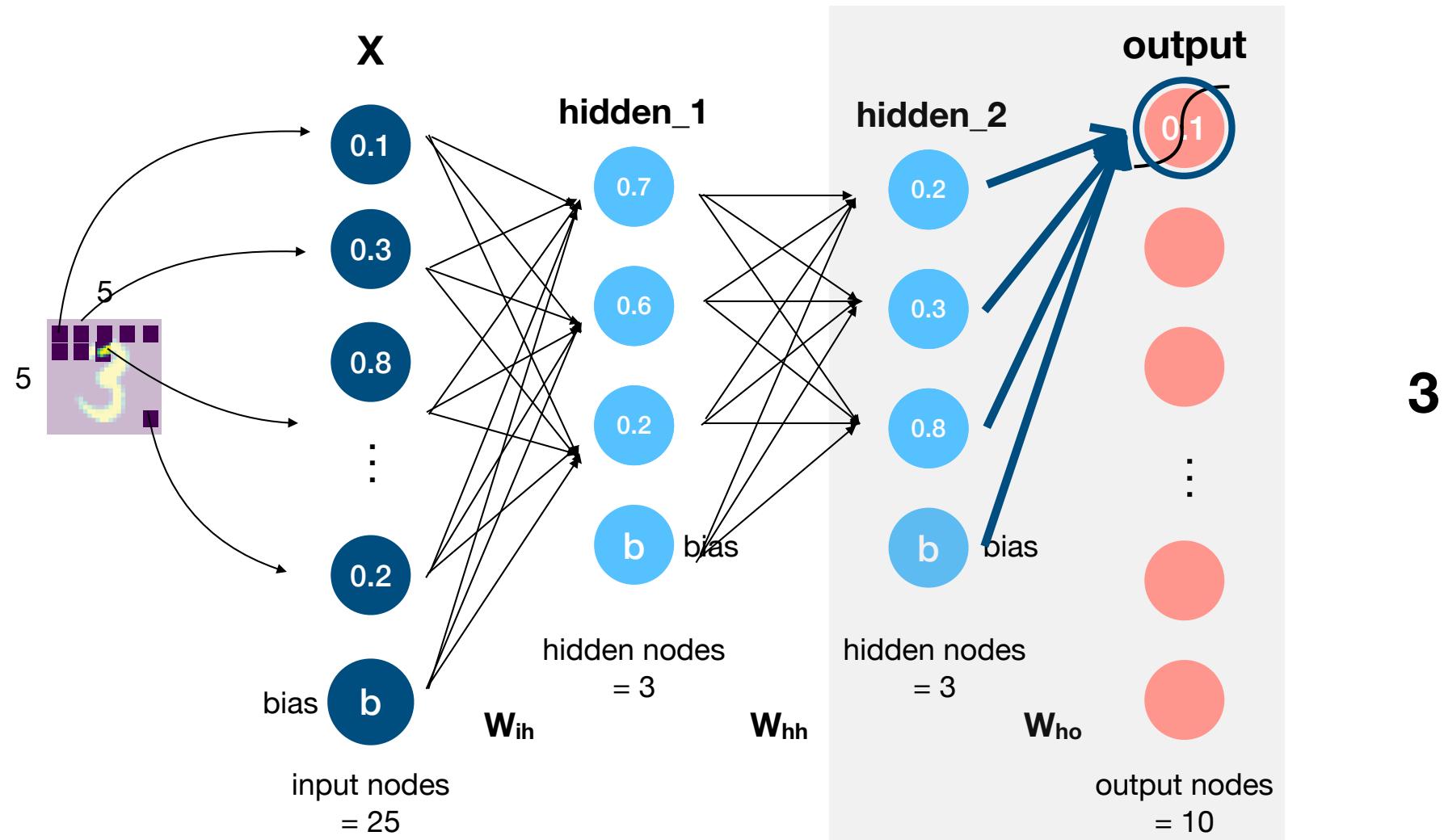
모든 hidden layer 2의 nodes에 대해 반복합니다.

ANN DNN



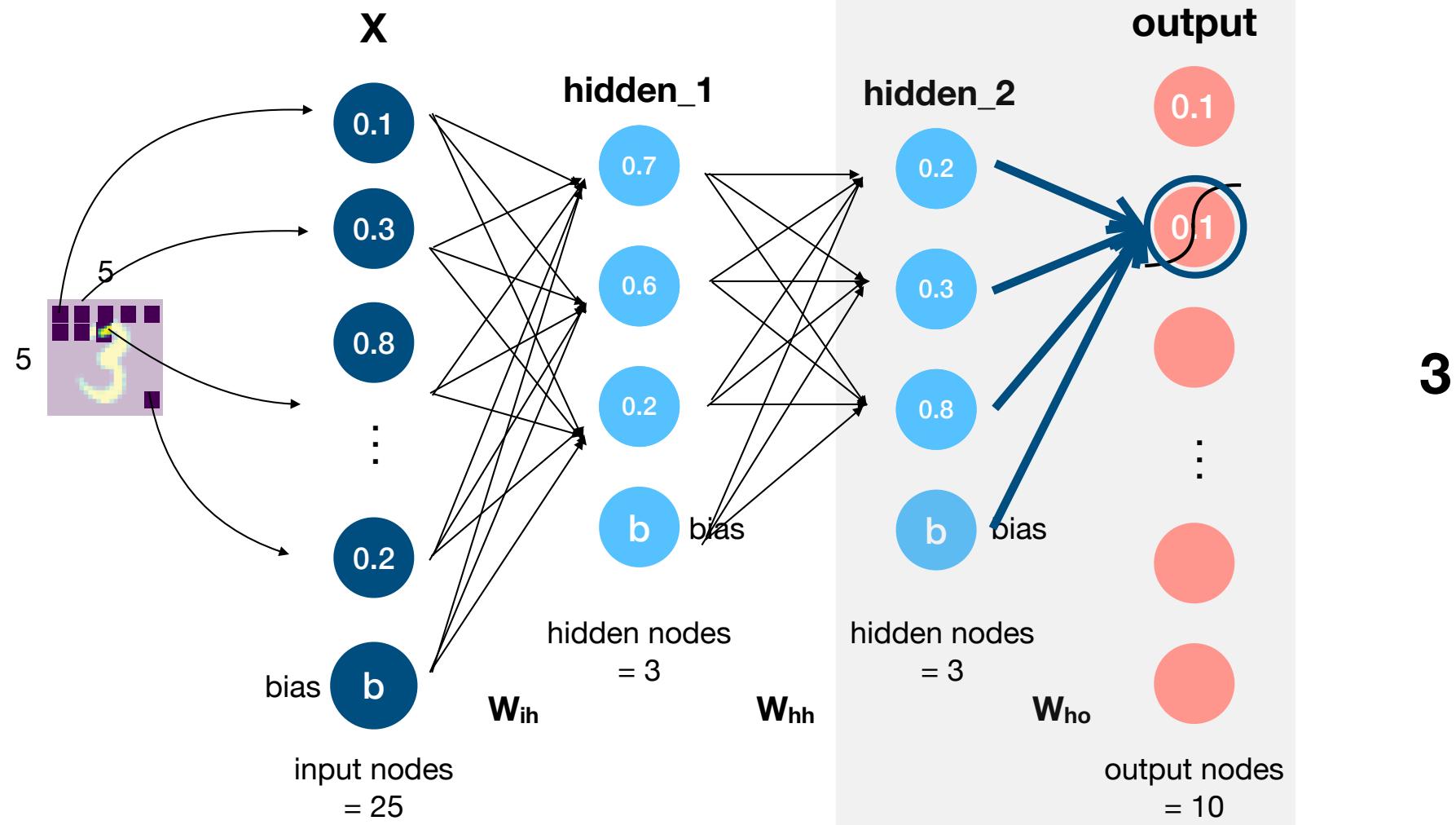
hidden layer 2까지의 nodes가 모두 계산되었습니다.
이제 output layer로 넘어갈 차례입니다.

ANN DNN



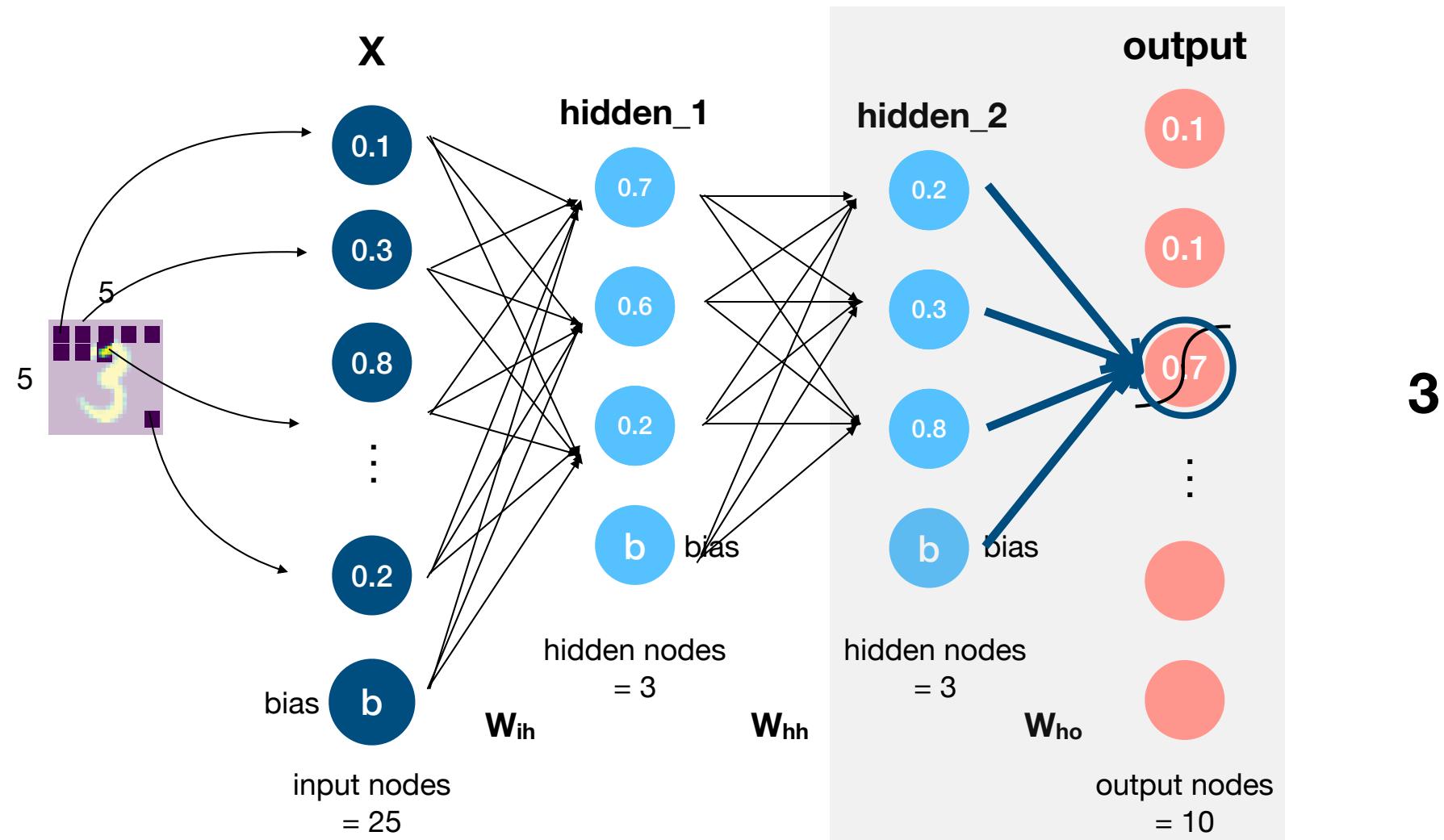
전과 같이, 각 hidden layer 2의 node 값에 weight (진한 화살표)을 곱하고 모두 더해 activation function을 건 후 output layer의 첫 번째 node에 넣어줍니다.

ANN DNN



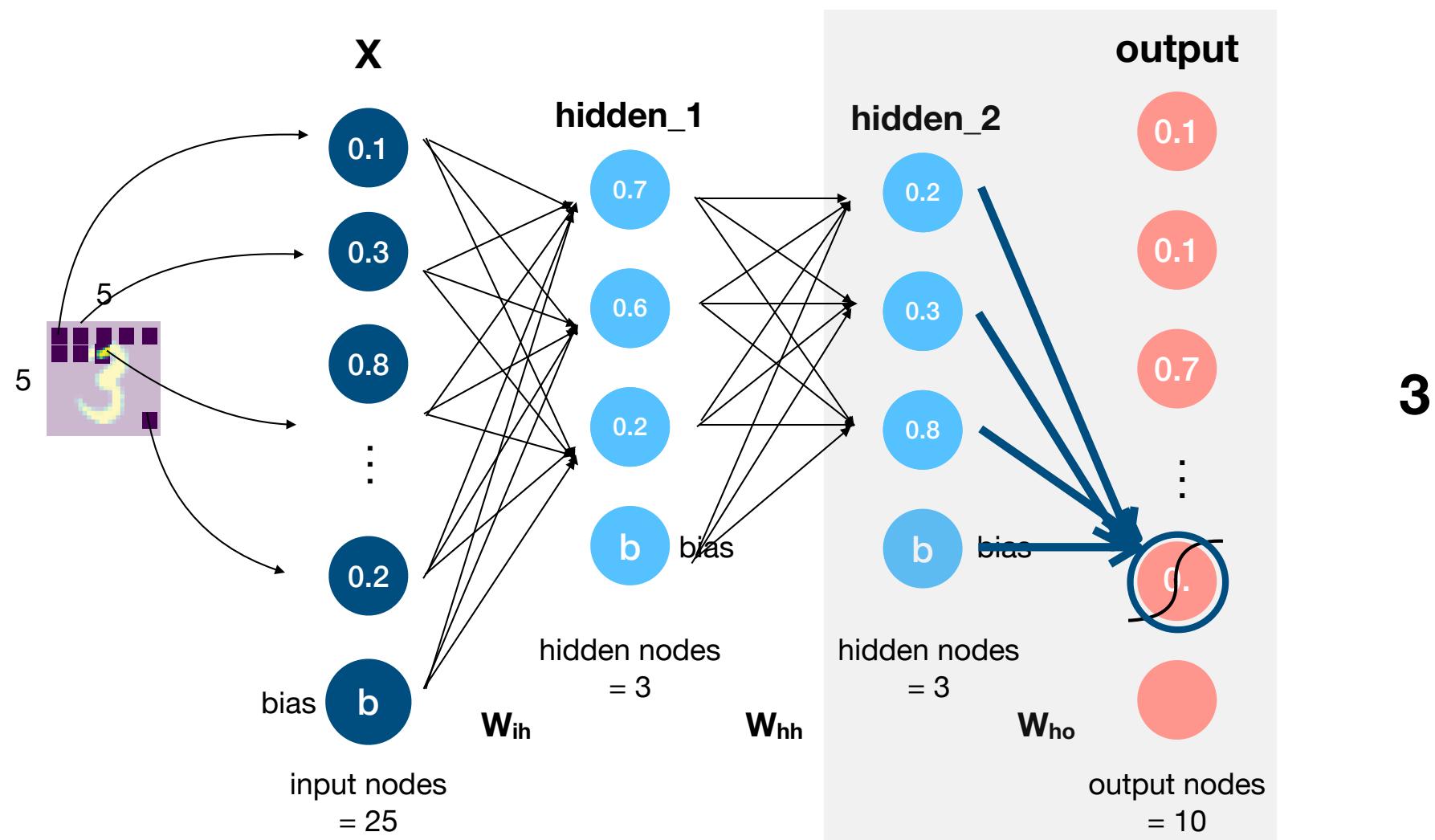
마찬가지 방식으로 반복합니다.

ANN DNN



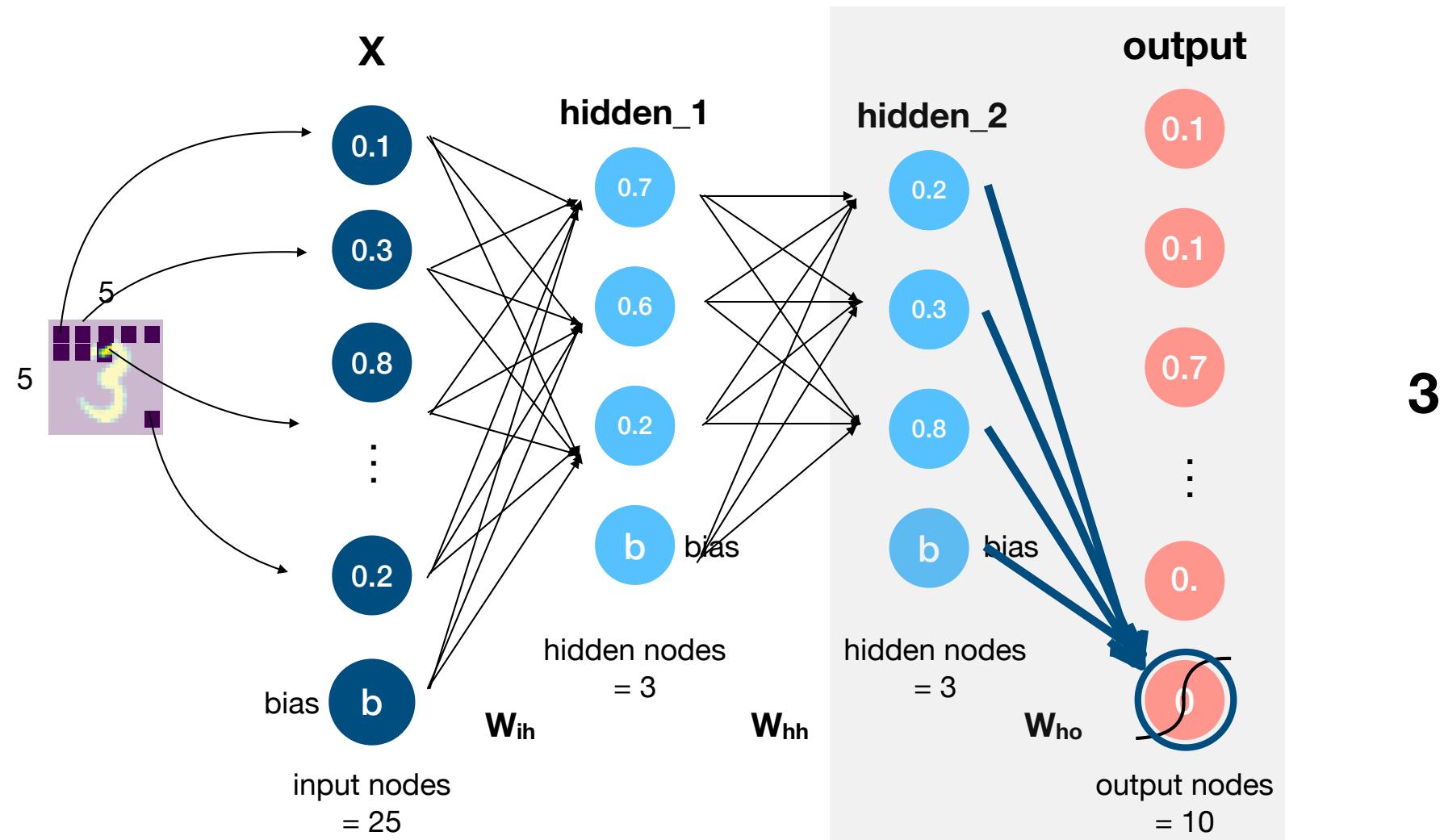
마찬가지 방식으로 반복합니다.

ANN DNN



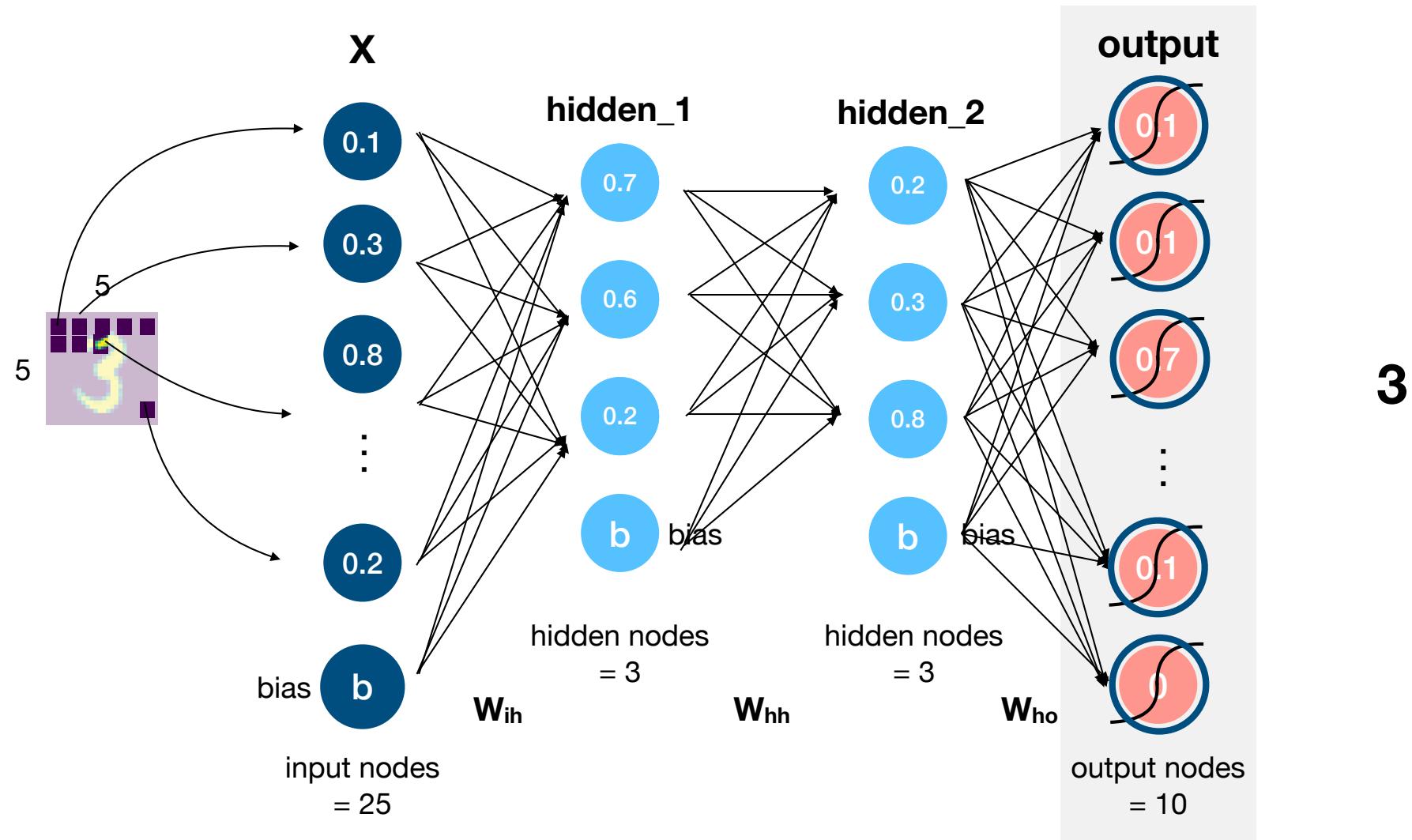
마찬가지 방식으로 반복합니다.

ANN DNN



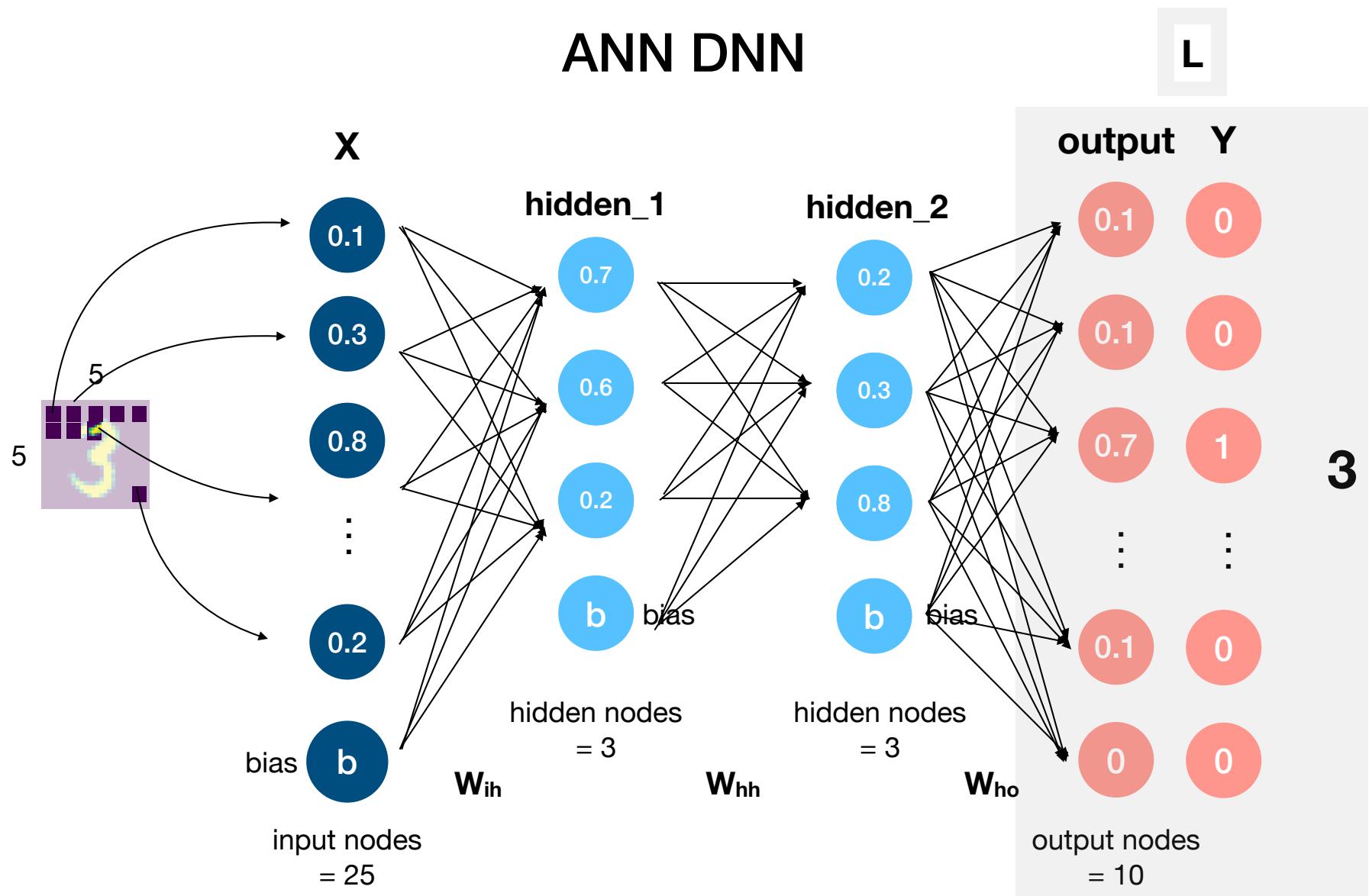
마찬가지 방식으로 반복합니다.

ANN DNN



hidden layers의 activation function과는 다르게
output layer에서는 activation function으로
softmax function을 사용하며 이는 input이 각 숫자일 확률을 계산해 줍니다.

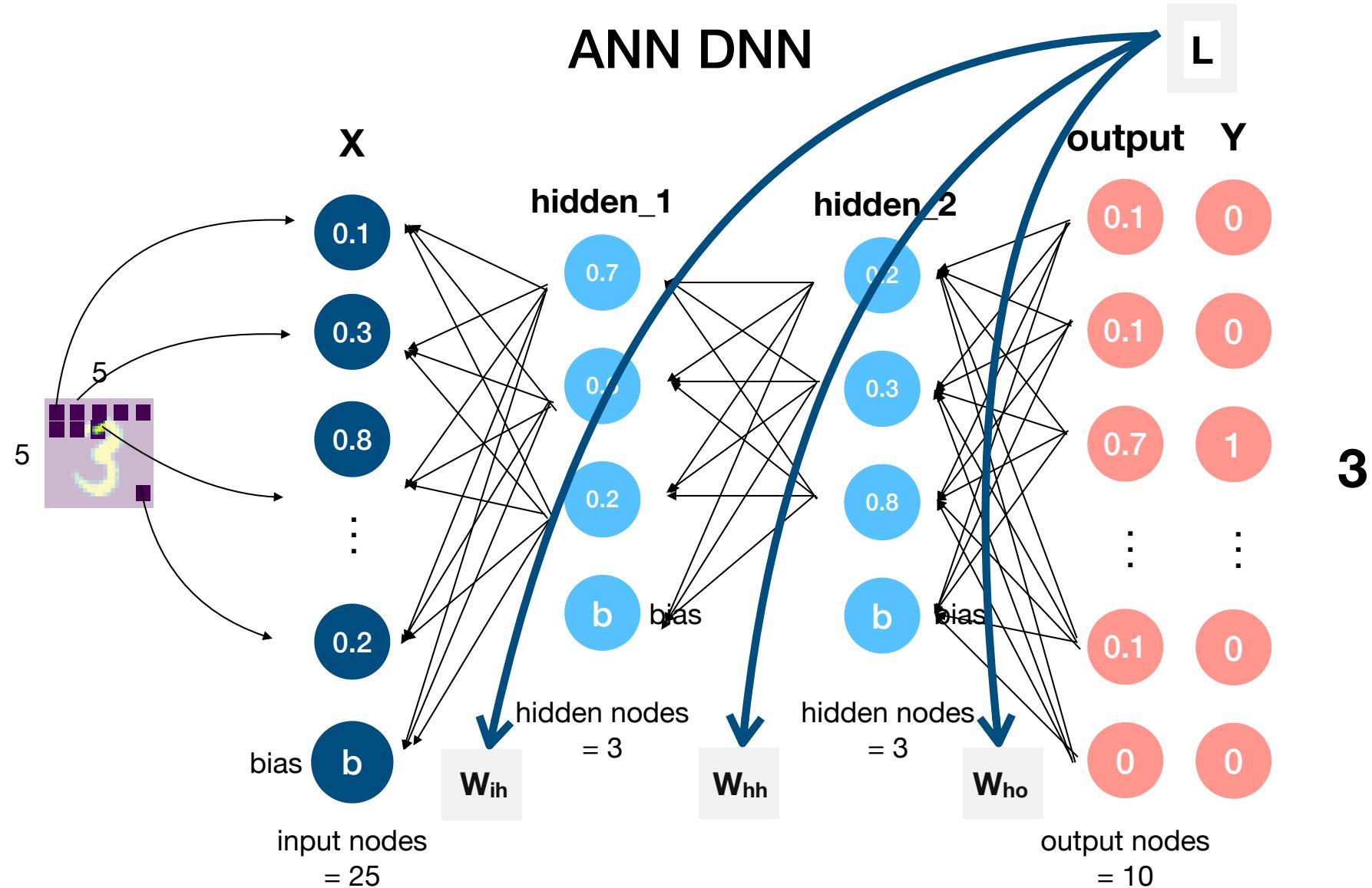
ANN DNN



그렇게 구한 output probability와 정답 target (Y)을 비교하여 loss를 구합니다.

이 때, target은 0~9 중에 정답인 3에 해당하는 세 번째 칸만 1, 나머지는 0인 one-hot vector

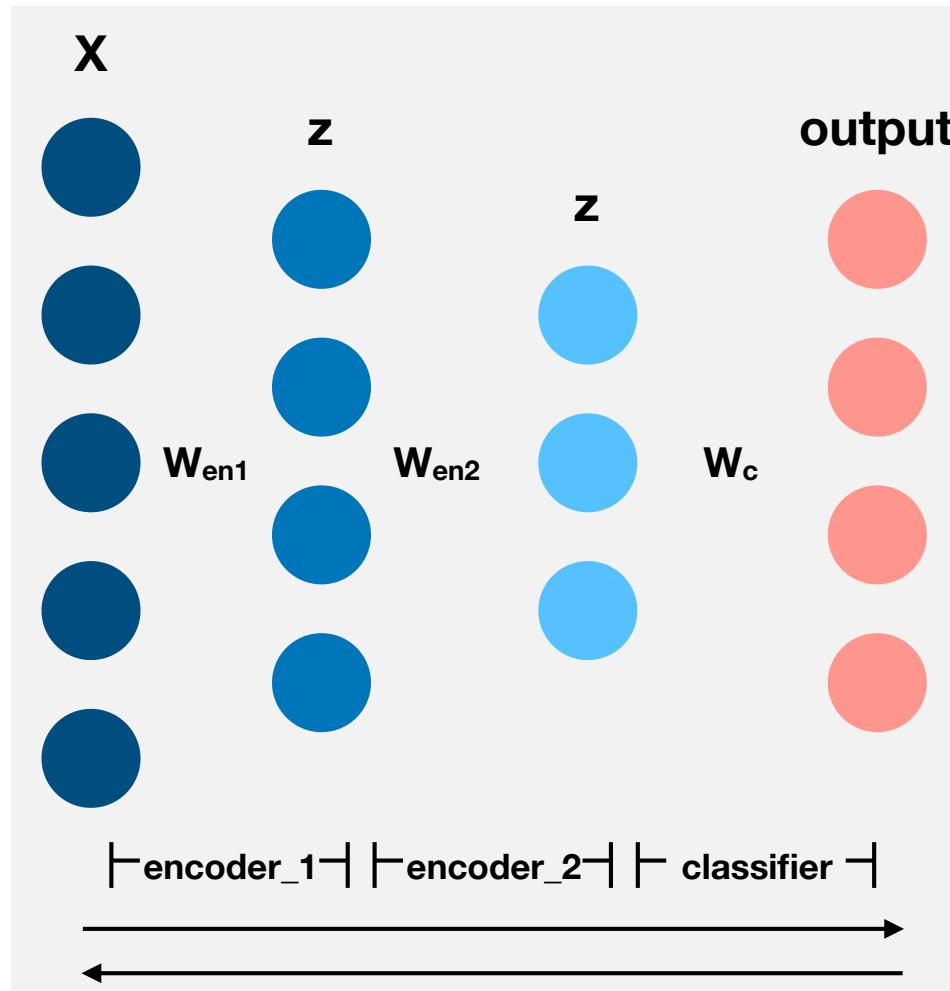
ANN DNN



loss를 최소화하는 방향으로, weight을 update 합니다.

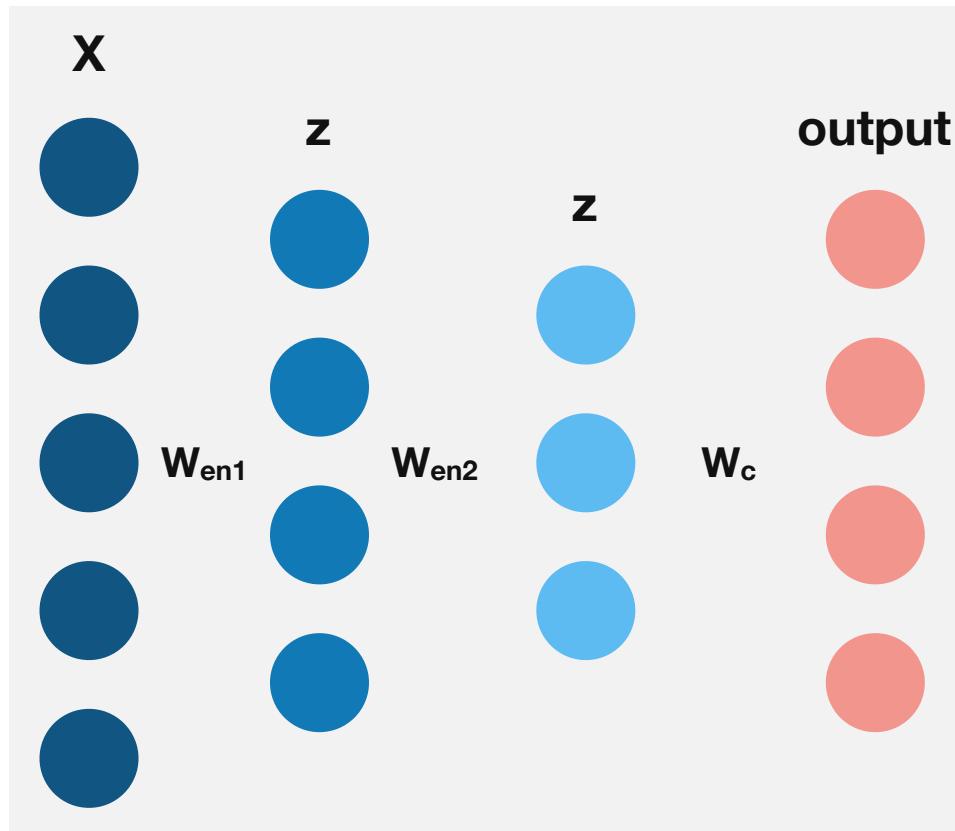
AE

Autoencoder



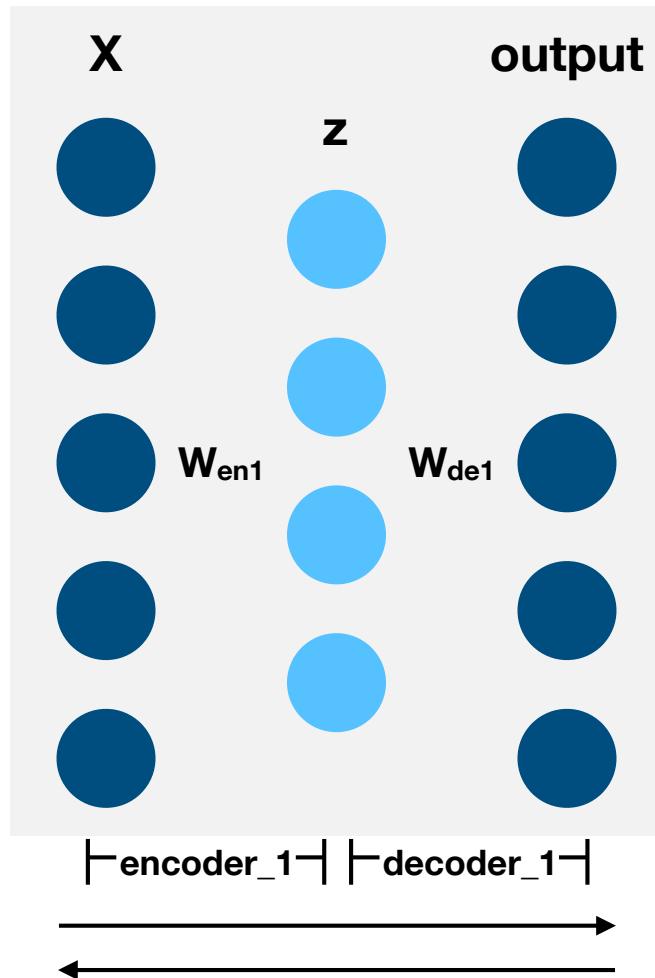
autoencoder classifier를 예시로 이해해 보겠습니다.

Autoencoder



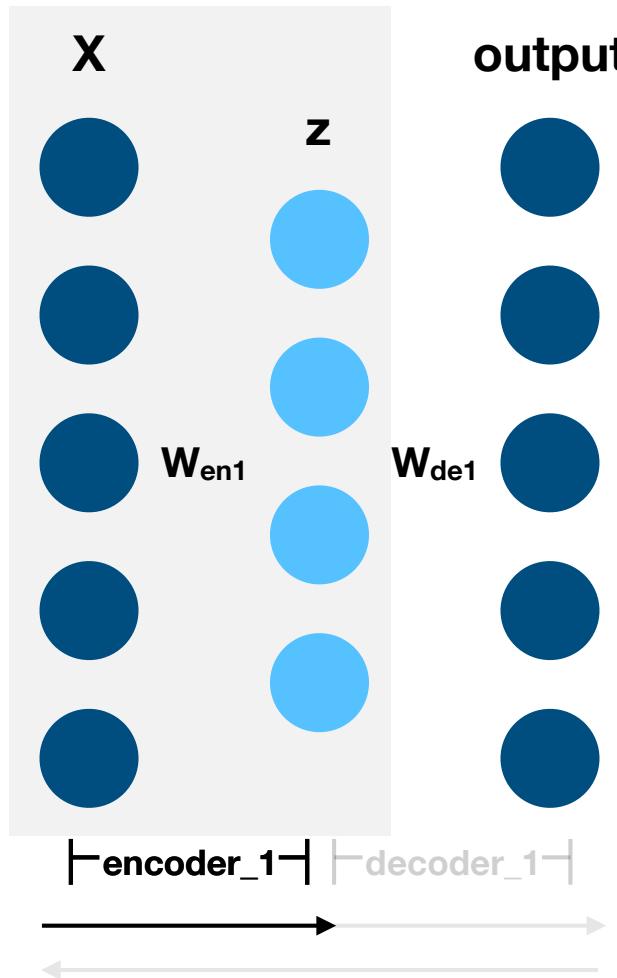
모델구조:
2 encoders
2 decoderes
1 classifier
1 fine-tuning

Autoencoder



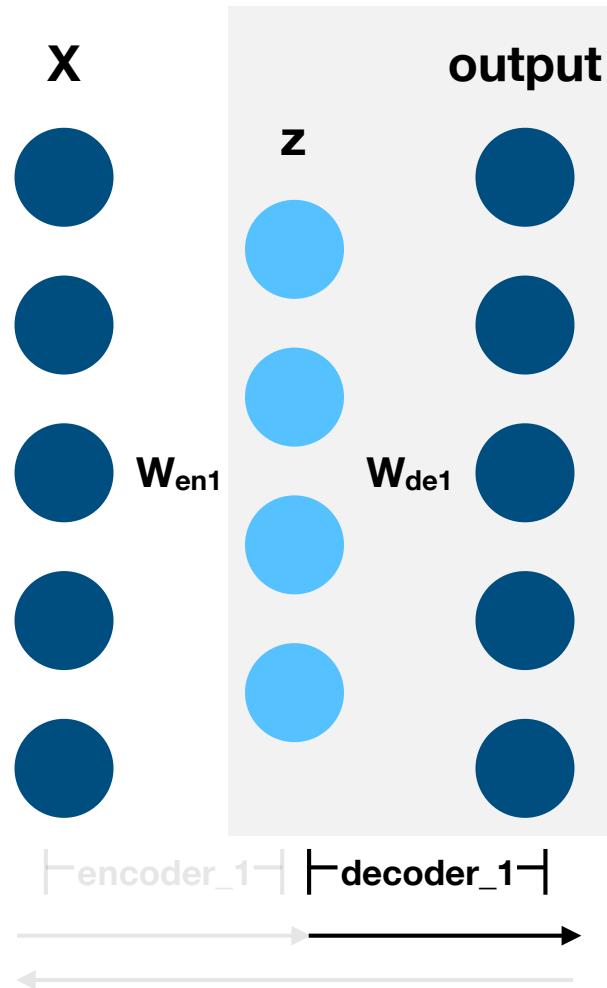
encoder_1과 decoder_1의 훈련과정

Autoencoder



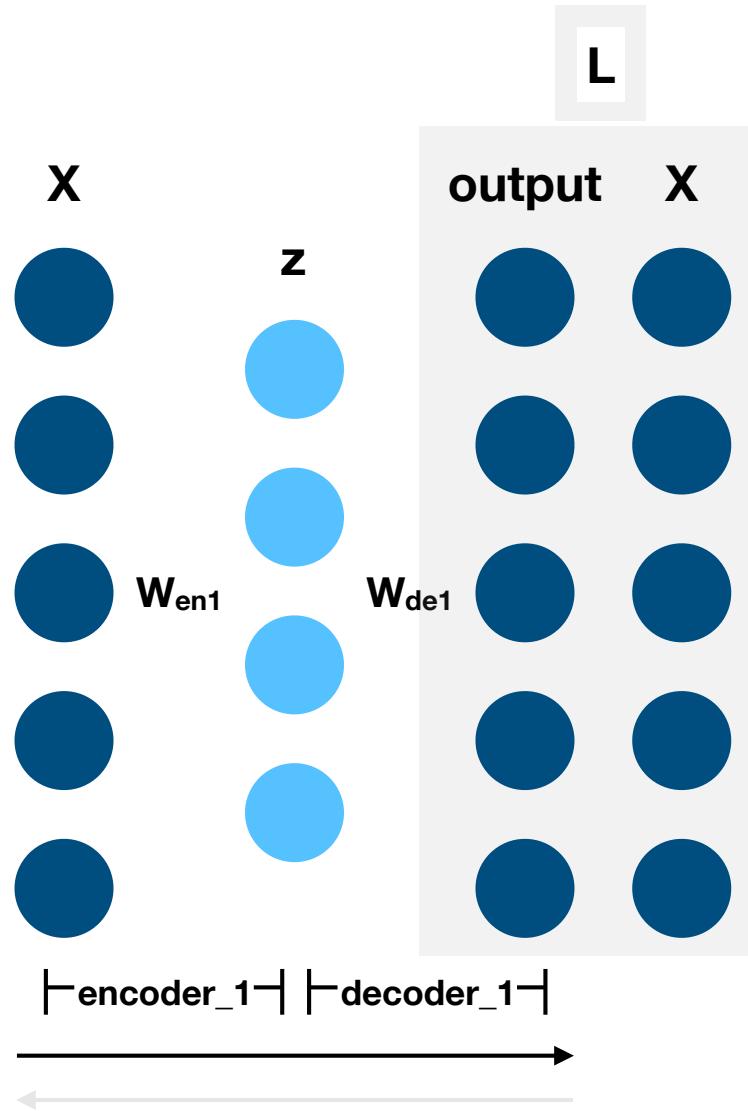
input layer에 encoder_1 weight (W_{en1})을 곱해서 z layer에 넣어줍니다.

Autoencoder



z layer에 **decoder_1** weight (**W_{de1}**)을 곱해서 **output** layer에 넣어줍니다.

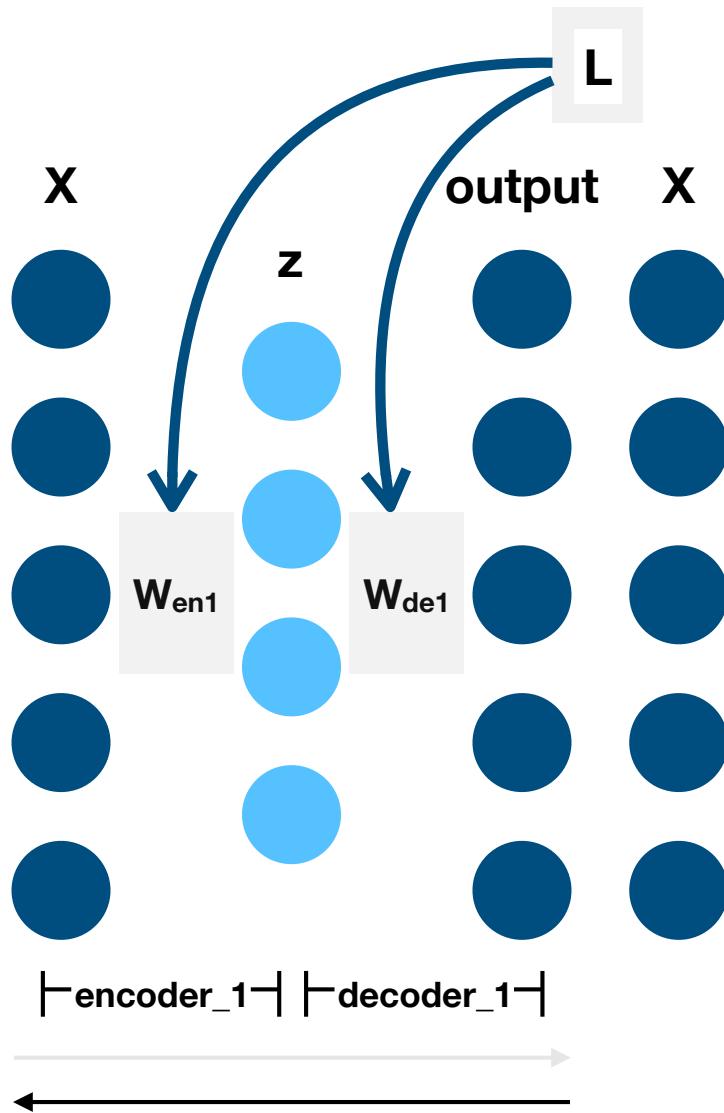
Autoencoder



target은 input 자기 자신과 동일합니다.

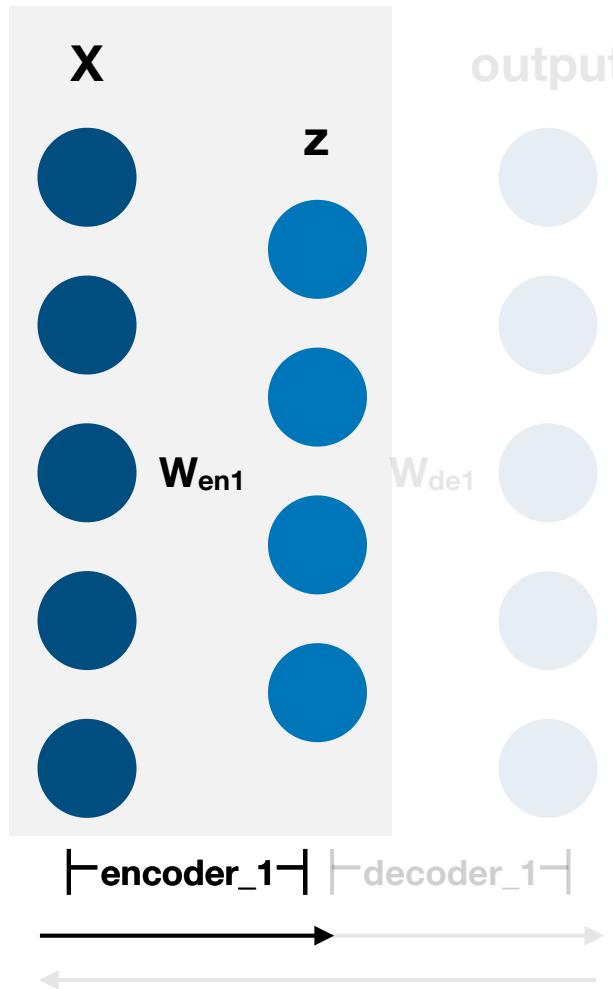
output과 정답 target (X)을 비교하여 loss를 구합니다.

Autoencoder



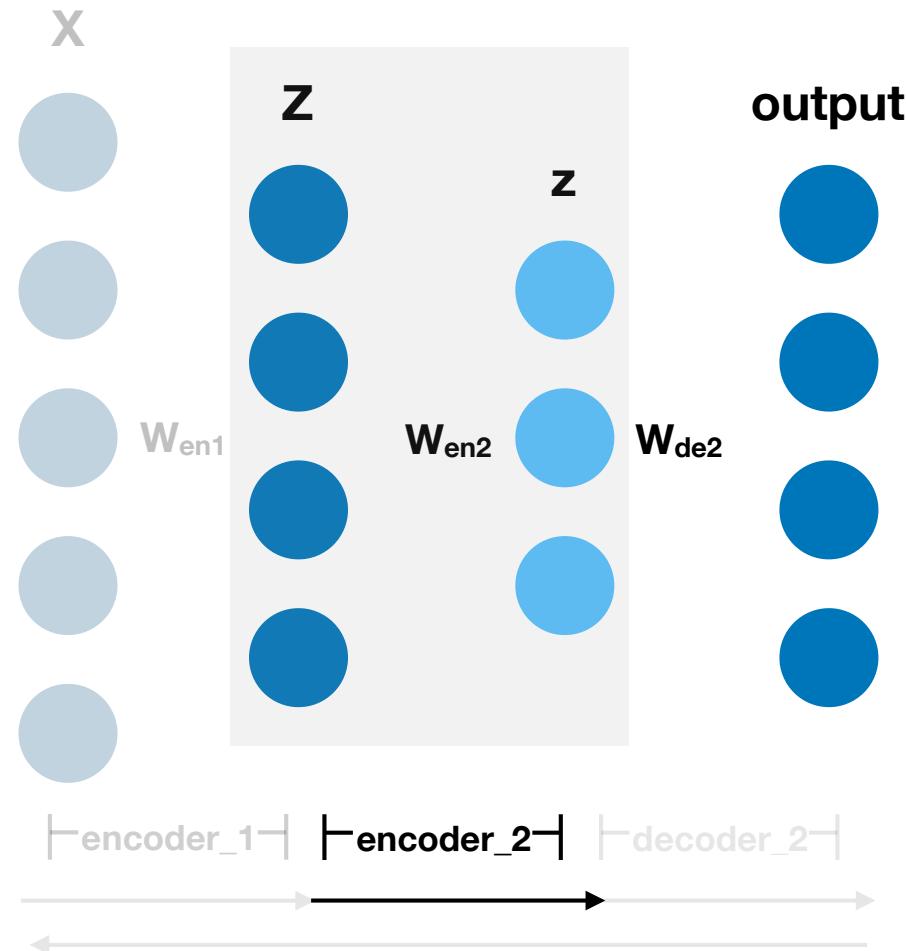
loss를 최소화하는 방향으로, weight를 update 합니다.

Autoencoder



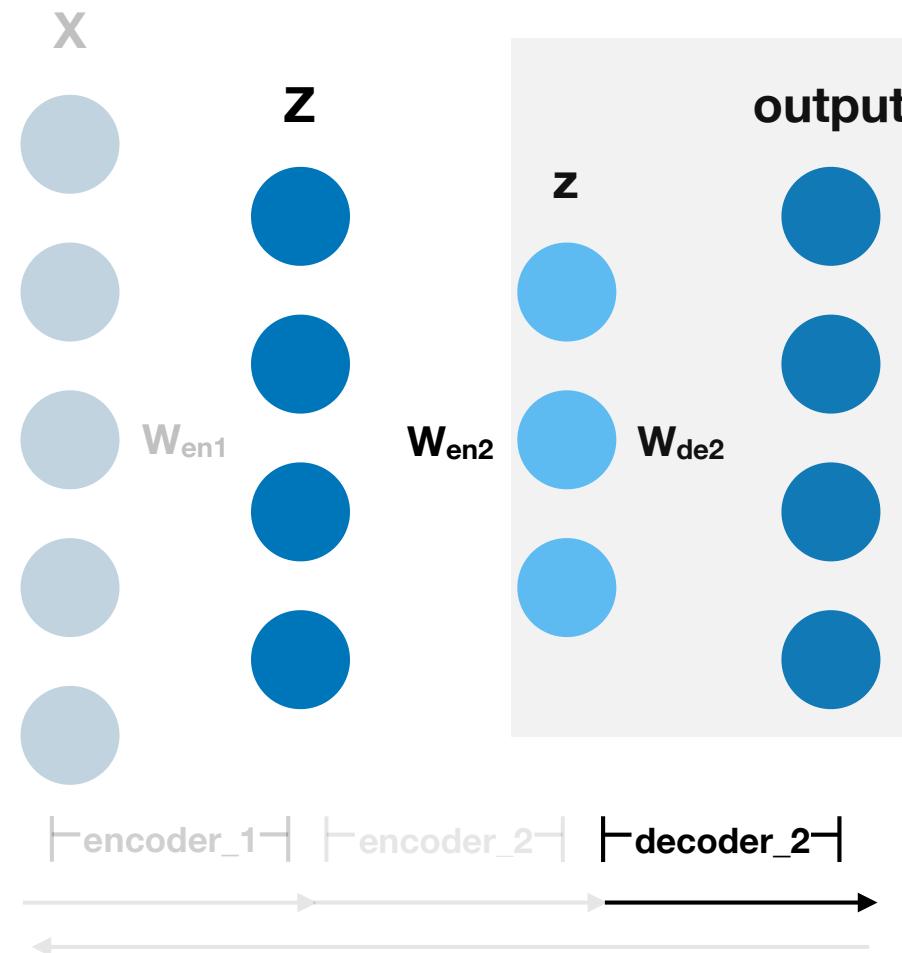
encoder_1의 훈련이 끝나면,
input을 encoder_1에 넣어 z를 구하고 이를 새로운 input Z로 삽습니다.

Autoencoder



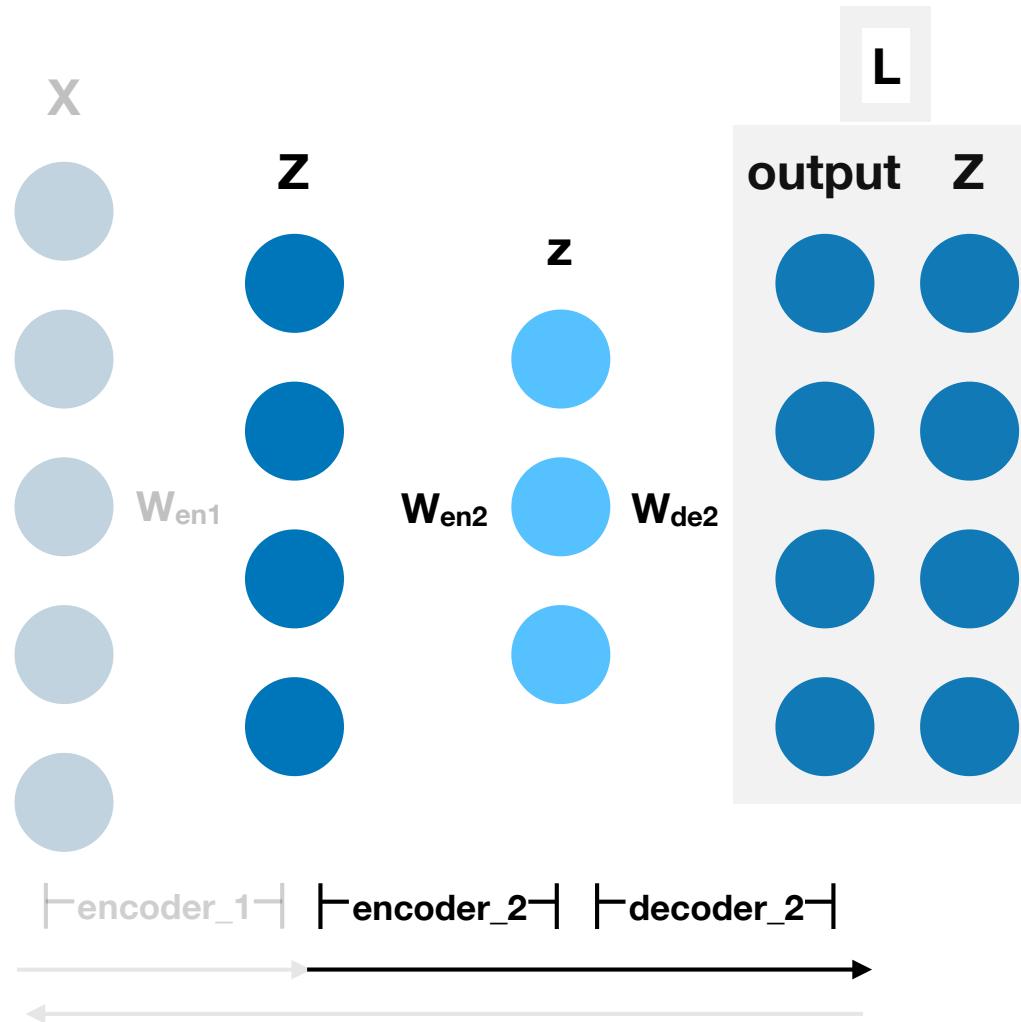
이 **Z** 값이 있는 layer에 **encoder_2 weight (W_{en2})**을 곱해서 **z layer**에 넣어줍니다.

Autoencoder



z layer에 **decoder_2** weight (W_{de2})을 곱해서 **output** layer에 넣어줍니다.

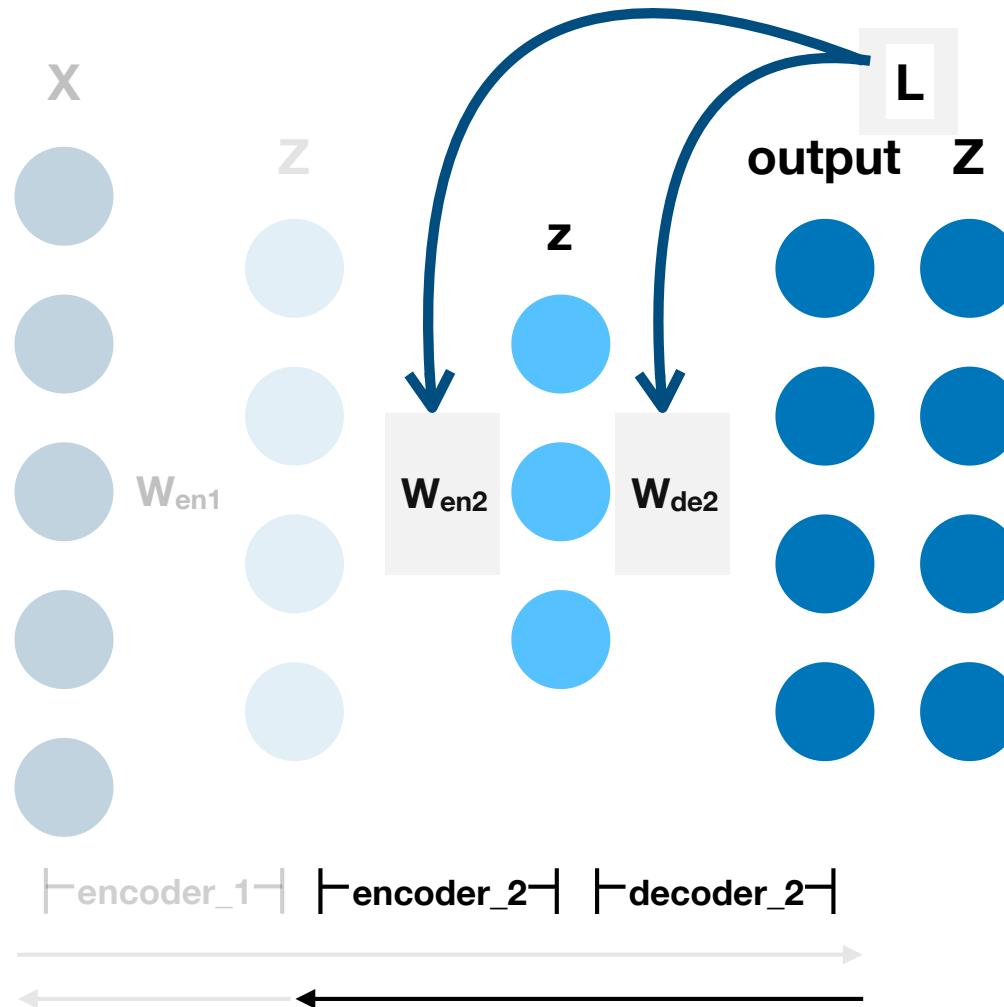
Autoencoder



target은 Z 자기 자신과 동일합니다.

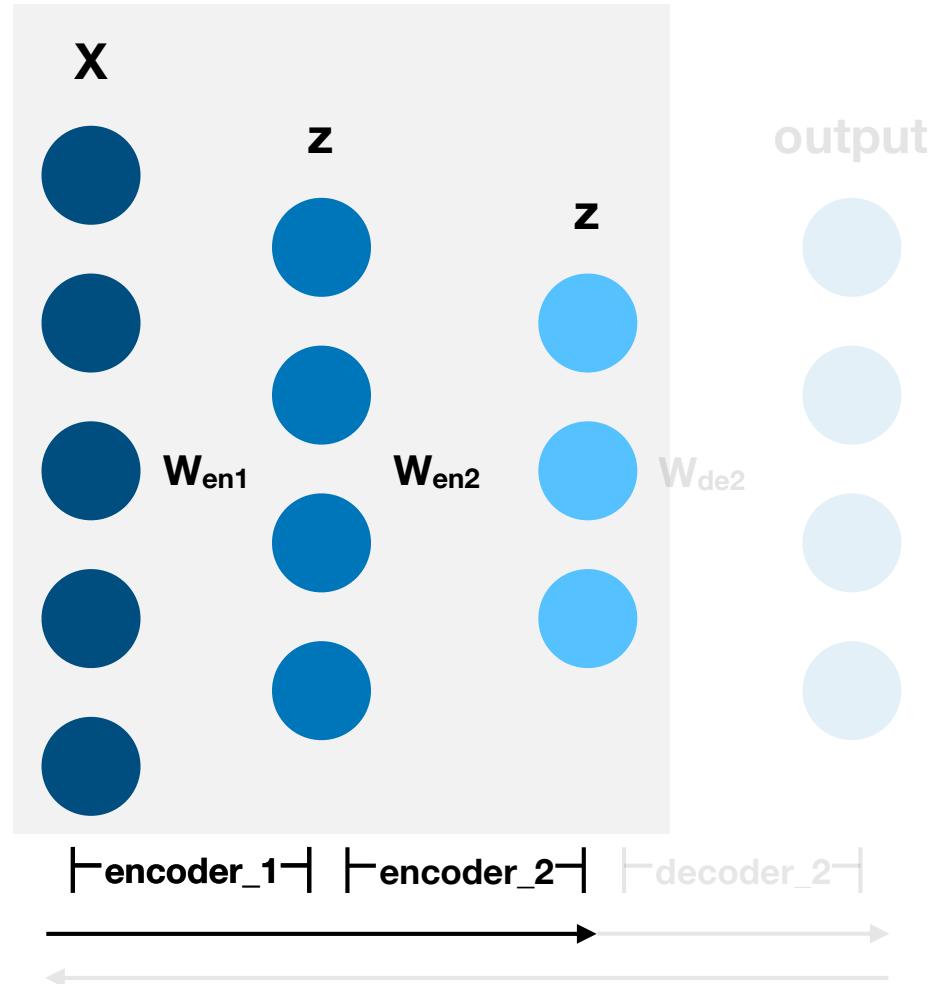
output과 정답 target (Z)을 비교하여 loss를 구합니다.

Autoencoder



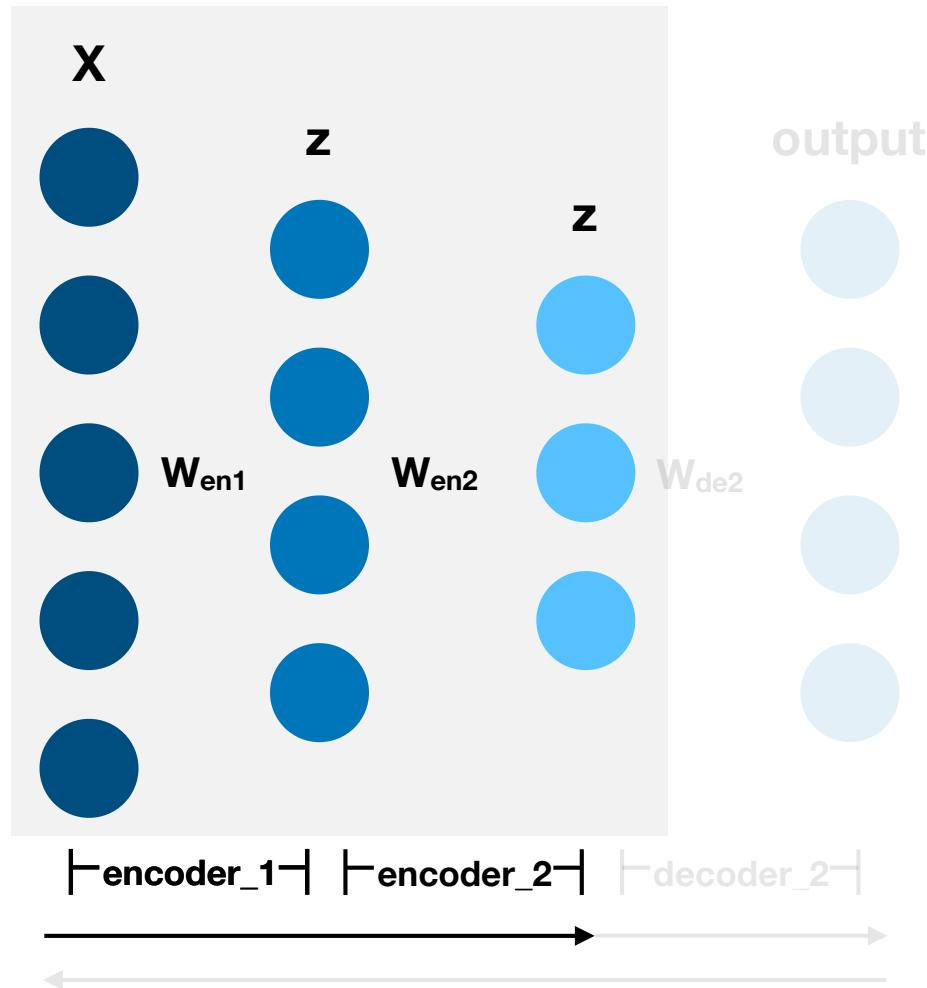
loss를 최소화하는 방향으로, weight을 update 합니다.

Autoencoder



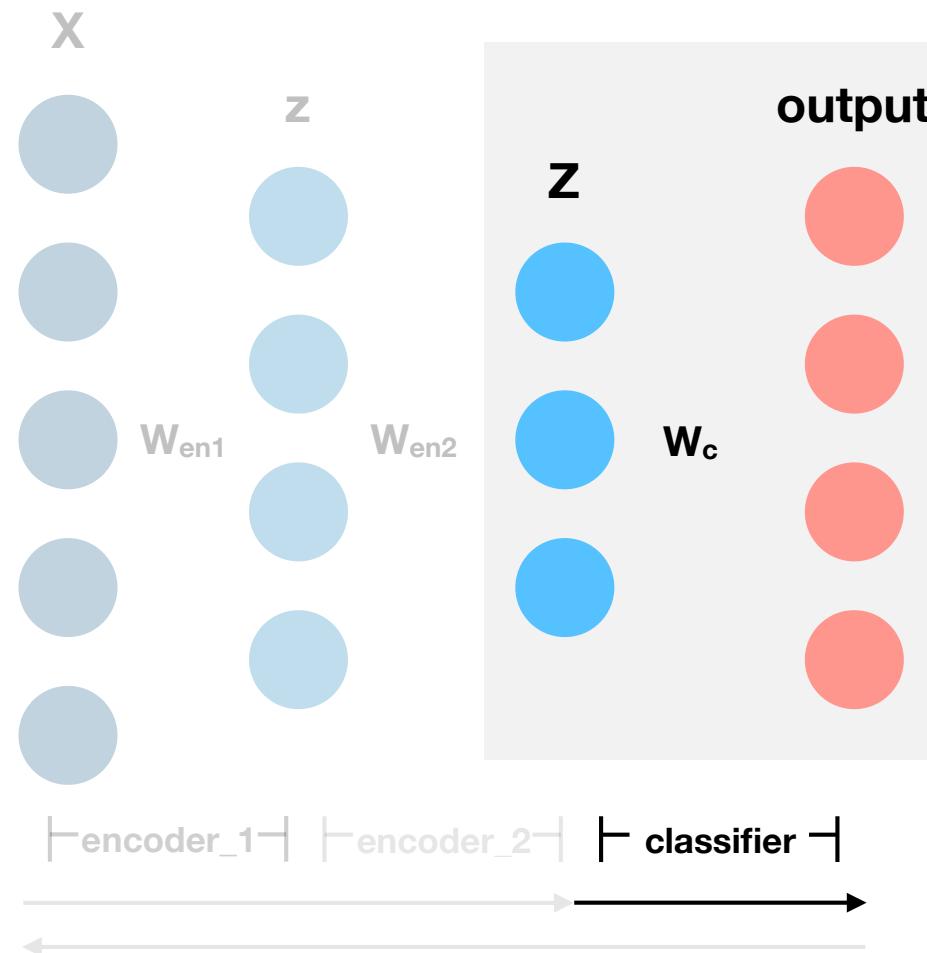
encoder 훈련이 모두 끝났습니다.
classifier 훈련을 하겠습니다.

Autoencoder



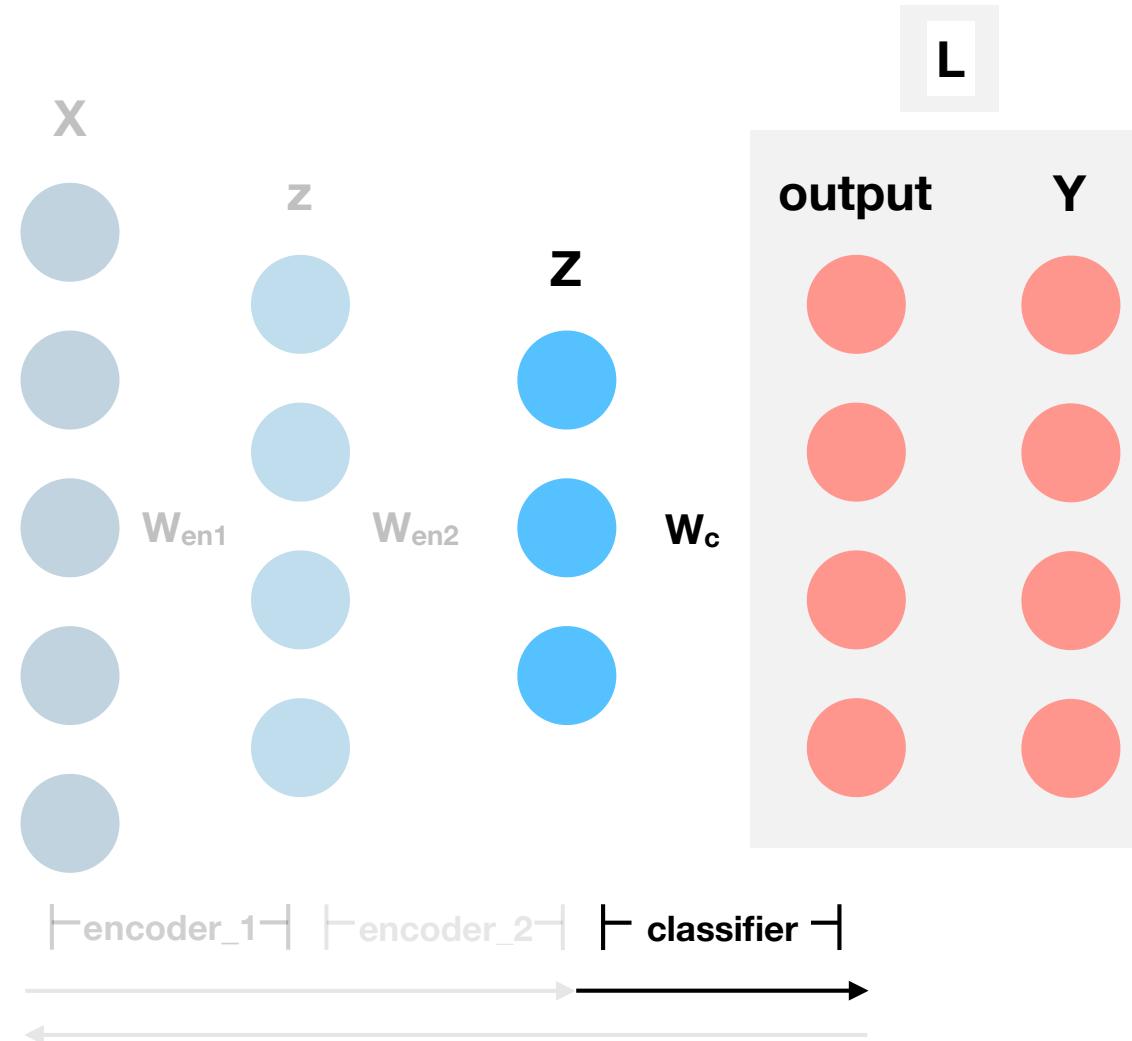
input을 **encoder_1**에 넣어 **z**를 구한 후, 다시 그 **z**를 **encoder_2**에 넣어 다음 **z** 값을 구합니다.
이전과 마찬가지 방식으로, 이 마지막 **z**를 새로운 **input Z**로 삼습니다.

Autoencoder



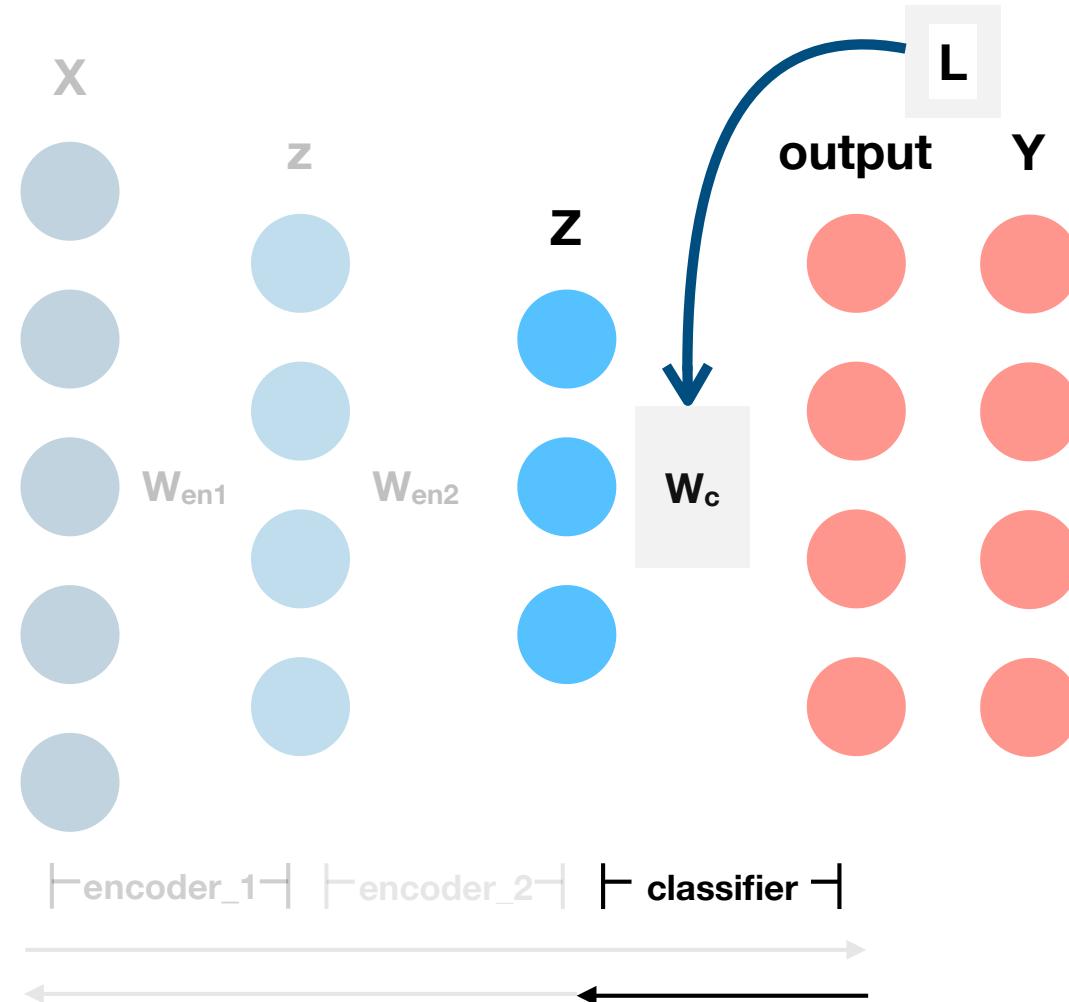
이 **z** 값이 있는 layer에 classifier weight (W_c)을 곱해서 output layer에 넣어줍니다.

Autoencoder



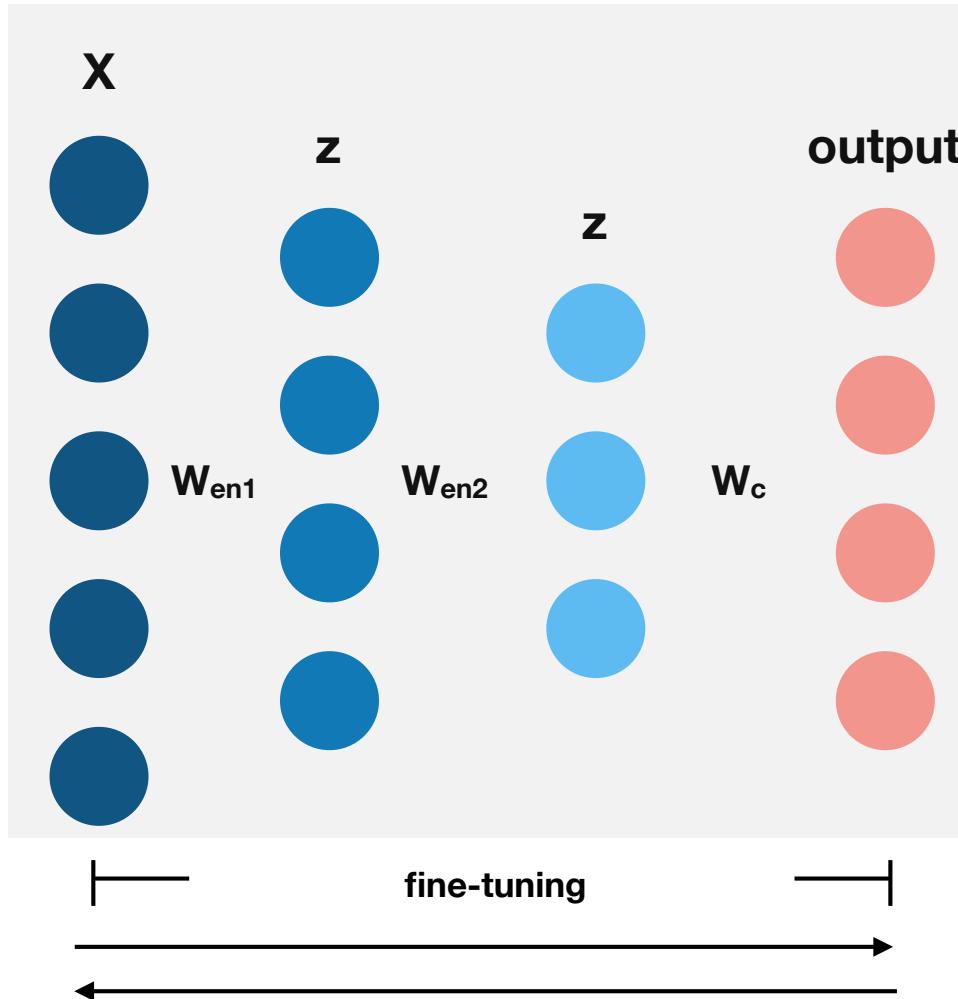
output과 정답 target (Y)을 비교하여 loss를 구합니다.

Autoencoder



loss를 최소화하는 방향으로, weight을 update 합니다.

Autoencoder



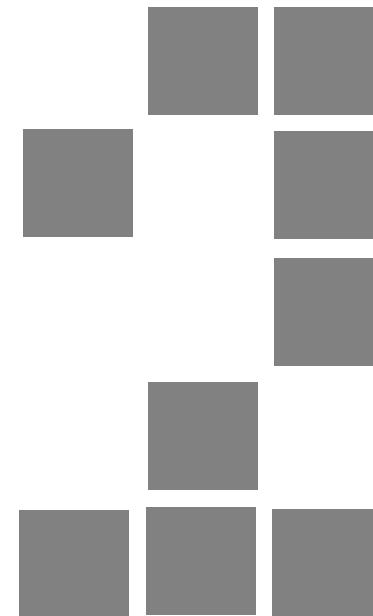
지금까지 각 layer의 weight을 따로 훈련했기 때문에,
마지막으로 모든 weight을 한꺼번에 조정해주는 fine-tuning을 해줍니다.
이 과정은 ANN 훈련 과정과 동일

CNN

Convolution

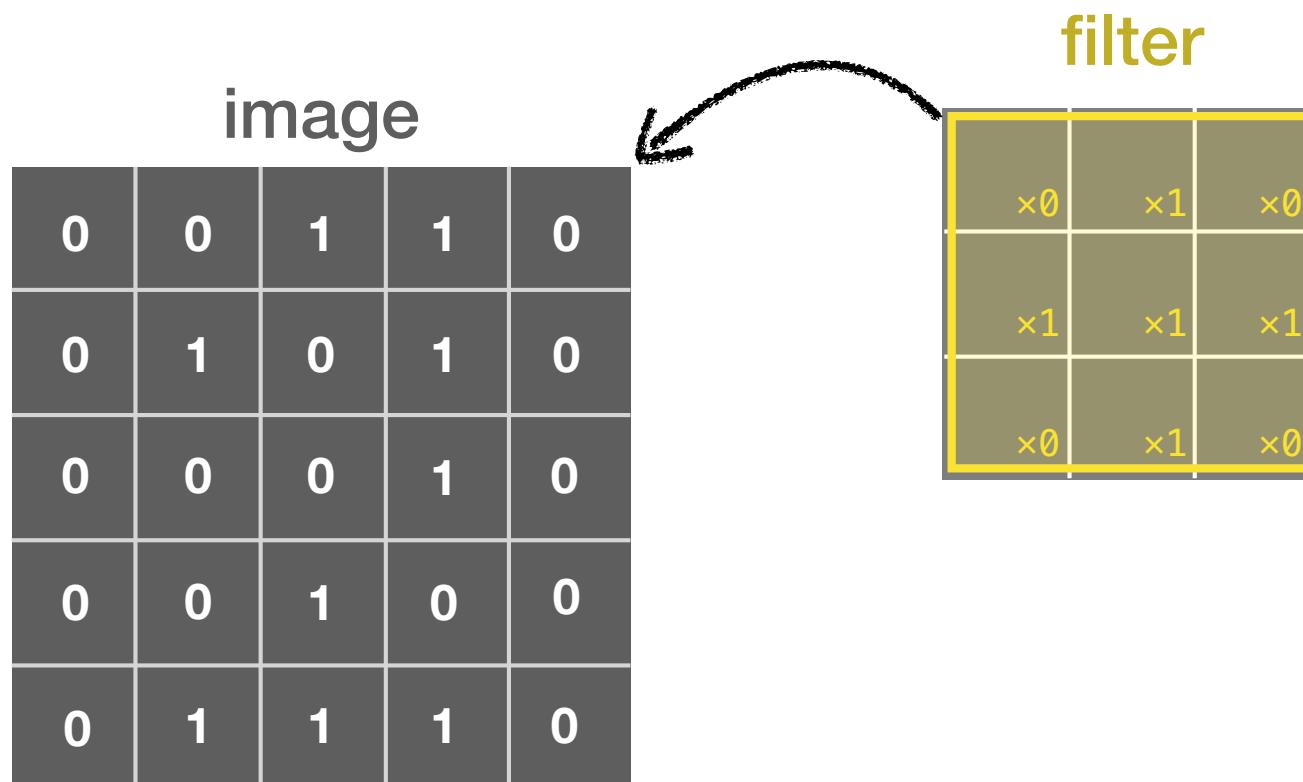
image

0	0	1	1	0
0	1	0	1	0
0	0	0	1	0
0	0	1	0	0
0	1	1	1	0



숫자 2의 이미지

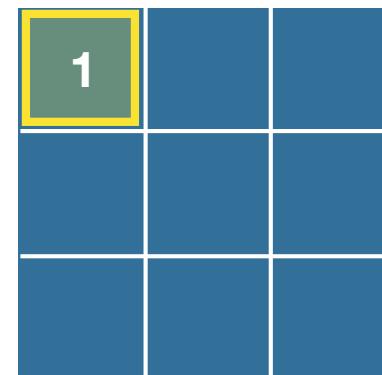
Convolution



이미지 위에 필터를 씌웁니다.

Convolution

0 $\times 0$	0 $\times 1$	1 $\times 0$	1	0
0 $\times 1$	1 $\times 1$	0 $\times 1$	1	0
0 $\times 0$	0 $\times 1$	0 $\times 0$	1	0
0	0	1	0	0
0	1	1	1	0



필터는 ANN에서의 weight과 같은 역할을 합니다.

필터의 각 노란색 weight 값과 이미지의 각 회색 pixel 값을 곱해서 합을 구합니다.

$$[0 \ 0 \ 1; 0 \ 1 \ 0; 0 \ 0 \ 0] \quad \circ \quad [0 \ 1 \ 0; 1 \ 1 \ 1; 0 \ 1 \ 0] = 1$$

Convolution

0	$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	0
0	$1_{\times 1}$	$0_{\times 1}$	$1_{\times 1}$	0
0	$0_{\times 0}$	$0_{\times 1}$	$1_{\times 0}$	0
0	0	1	0	0
0	1	1	1	0

1	3	

필터를 한 칸씩 옮겨가면서 값을 구해 새로운 파란색 matrix를 채워갑니다.

여기선 필터를 한 칸씩 옮겼지만 몇 칸씩 옮길지를 설정할 수 있는데, 이것을 `stride`라고 합니다.

Convolution

0	0	$1_{\times 0}$	$1_{\times 1}$	$0_{\times 0}$
0	1	$0_{\times 1}$	$1_{\times 1}$	$0_{\times 1}$
0	0	$0_{\times 0}$	$1_{\times 1}$	$0_{\times 0}$
0	0	1	0	0
0	1	1	1	0

1	3	3

이 과정을 반복합니다.

Convolution

0	0	1	1	0
0×0	1×1	0×0	1	0
0×1	0×1	0×1	1	0
0×0	0×1	1×0	0	0
0	1	1	1	0

1	3	3
1		

이 과정을 반복합니다.

Convolution

0	0	1	1	0
0	1 $\times 0$	0 $\times 1$	1 $\times 0$	0
0	0 $\times 1$	0 $\times 1$	1 $\times 1$	0
0	0 $\times 0$	1 $\times 1$	0 $\times 0$	0
0	1	1	1	0

1	3	3
1	2	

이 과정을 반복합니다.

Convolution

0	0	1	1	0
0	1	0×0	1×1	0×0
0	0	0×1	1×1	0×1
0	0	1×0	0×1	0×0
0	1	1	1	0

1	3	3
1	2	2

이 과정을 반복합니다.

Convolution

0	0	1	1	0
0	1	0	1	0
0×0	0×1	0×0	1	0
0×1	0×1	1×1	0	0
0×0	1×1	1×0	1	0

1	3	3
1	2	2
2		

이 과정을 반복합니다.

Convolution

0	0	1	1	0
0	1	0	1	0
0	0×0	0×1	1×0	0
0	0×1	1×1	0×1	0
0	1×0	1×1	1×0	0

1	3	3
1	2	2
2	2	

이 과정을 반복합니다.

Convolution

0	0	1	1	0
0	1	0	1	0
0	0	0×0	1×1	0×0
0	0	1×1	0×1	0×1
0	1	1×0	1×1	0×0

1	3	3
1	2	2
2	2	3

완성된 오른쪽 파란색 박스를 **feature map0**이라고 합니다.

Max Pooling

3	7	2	3
3	2	4	1
0	1	3	1
0	4	1	0

7	

convolution의 결과로 나온 feature map이 너무 클 때, feature map의 크기를 줄여줘야 합니다. feature map의 구획을 나눠서 각 구획에서 가장 큰 값을 뽑는 방법을 max pooling이라고 합니다.

Max Pooling

3	7	2	3
3	2	4	1
0	1	3	1
0	4	1	0

7	4

convolution의 결과로 나온 feature map이 너무 클 때, feature map의 크기를 줄여줘야 합니다. feature map의 구획을 나눠서 각 구획에서 가장 큰 값을 뽑는 방법을 max pooling이라고 합니다.

Max Pooling

3	7	2	3
3	2	4	1
0	1	3	1
0	4	1	0

7	4
4	

convolution의 결과로 나온 feature map이 너무 클 때, feature map의 크기를 줄여줘야 합니다. feature map의 구획을 나눠서 각 구획에서 가장 큰 값을 뽑는 방법을 max pooling이라고 합니다.

Max Pooling

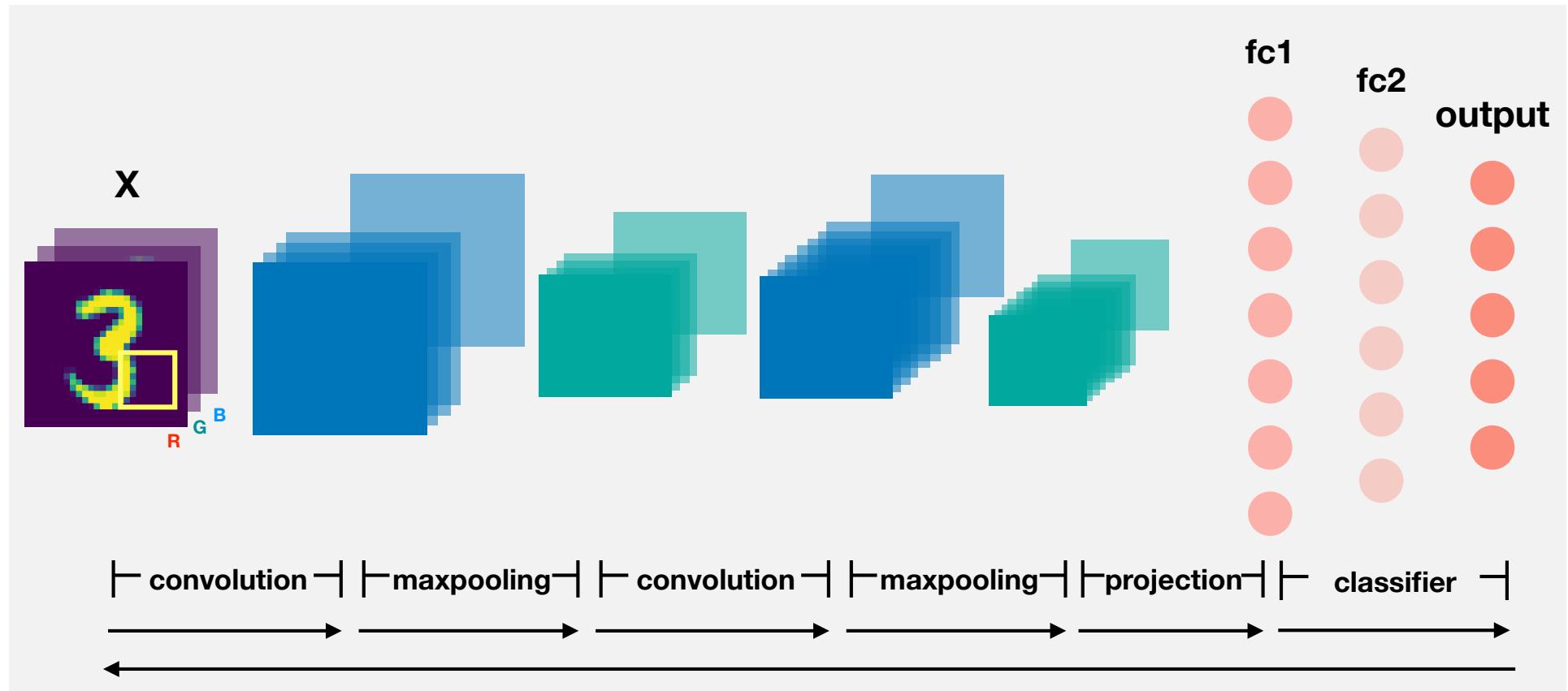
3	7	2	3
3	2	4	1
0	1	3	1
0	4	1	0

7	4
4	3

convolution의 결과로 나온 feature map이 너무 클 때, feature map의 크기를 줄여줘야 합니다.
feature map의 구획을 나눠서 각 구획에서 가장 큰 값을 뽑는 방법을 max pooling이라고 합니다.

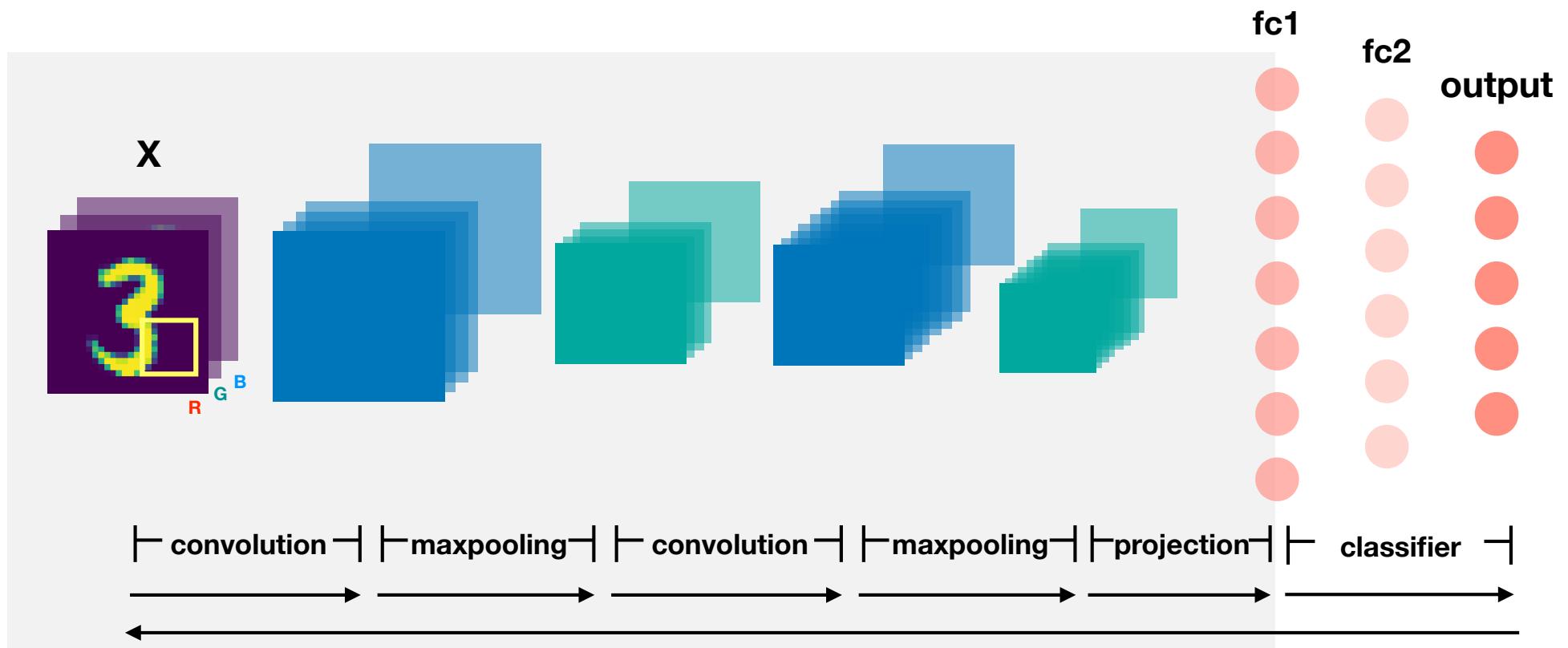
max pooling 또한 stride를 조정할 수 있음

CNN



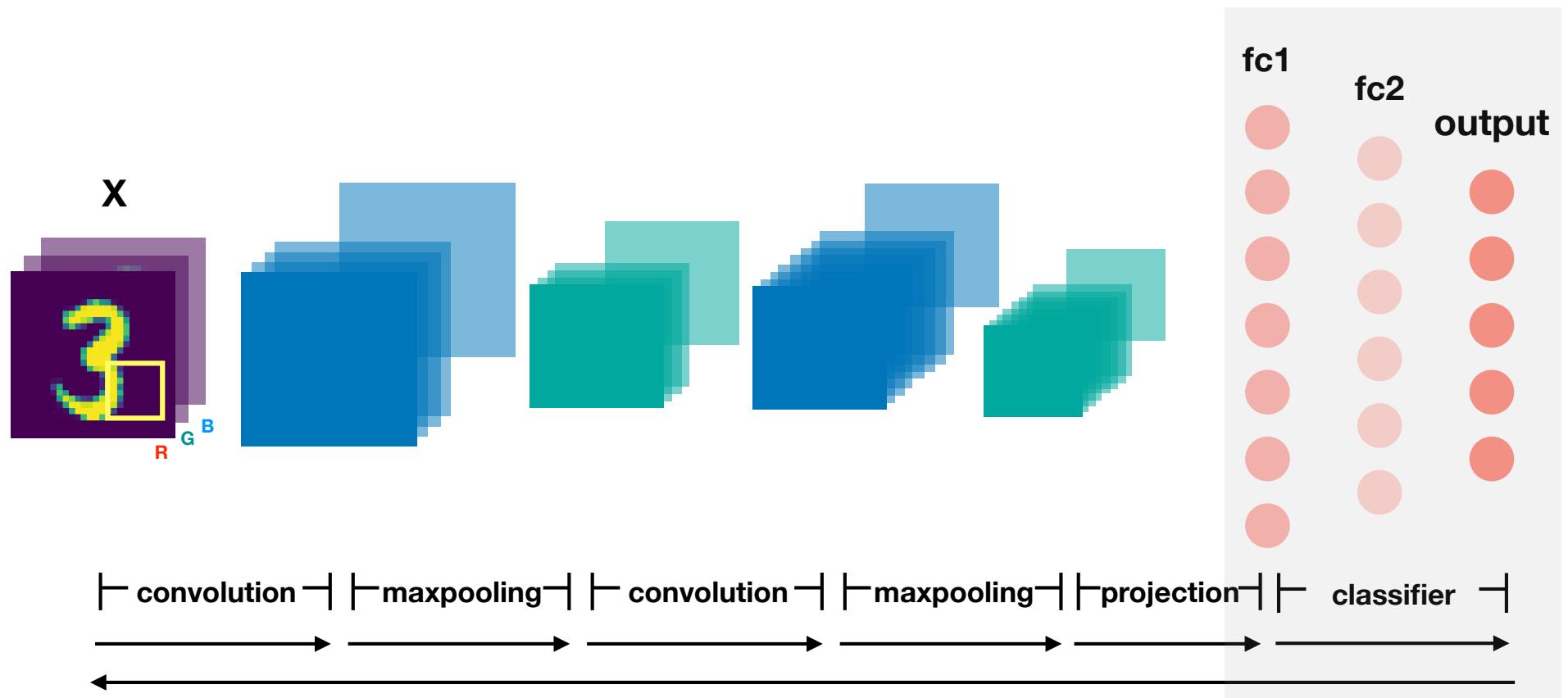
이 개념을 토대로 CNN을 이해해 보겠습니다.

CNN



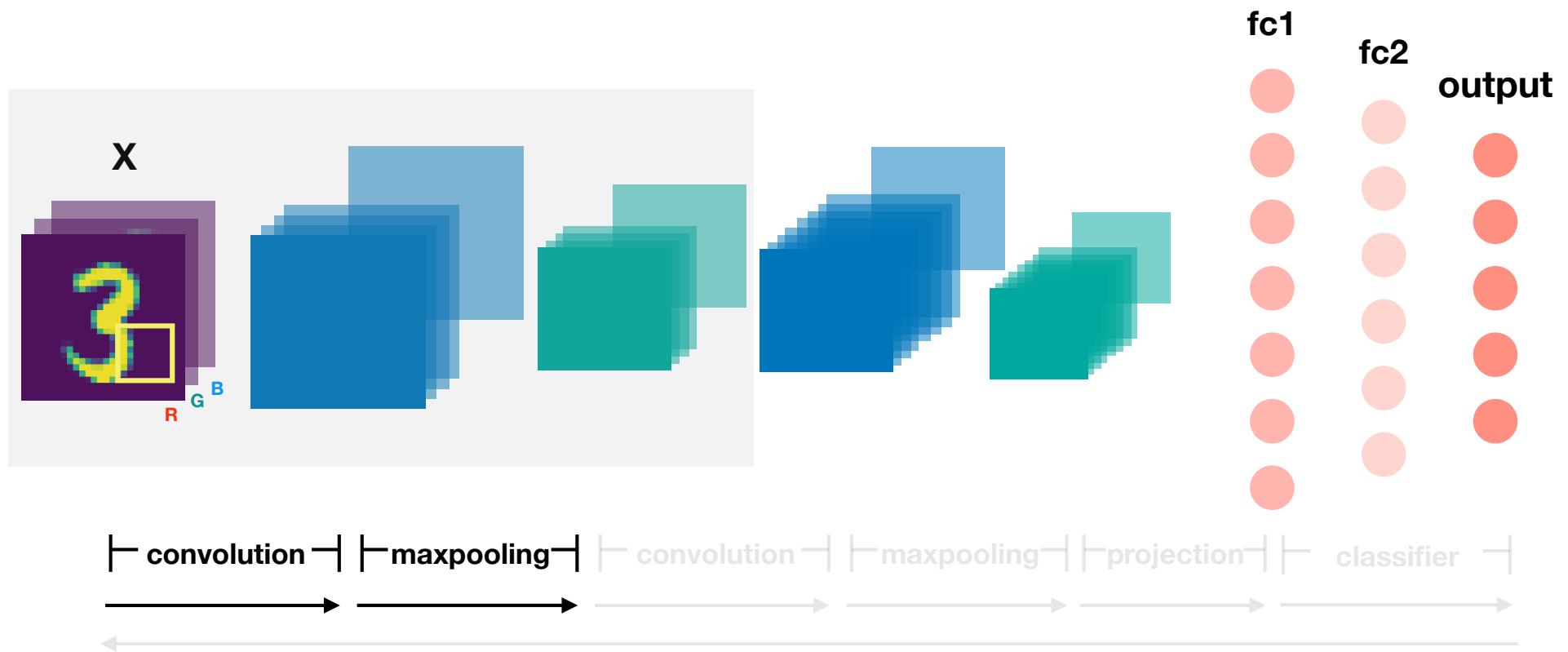
CNN은 input의 특징을 잘 반영하도록 해주는 준비과정과

CNN



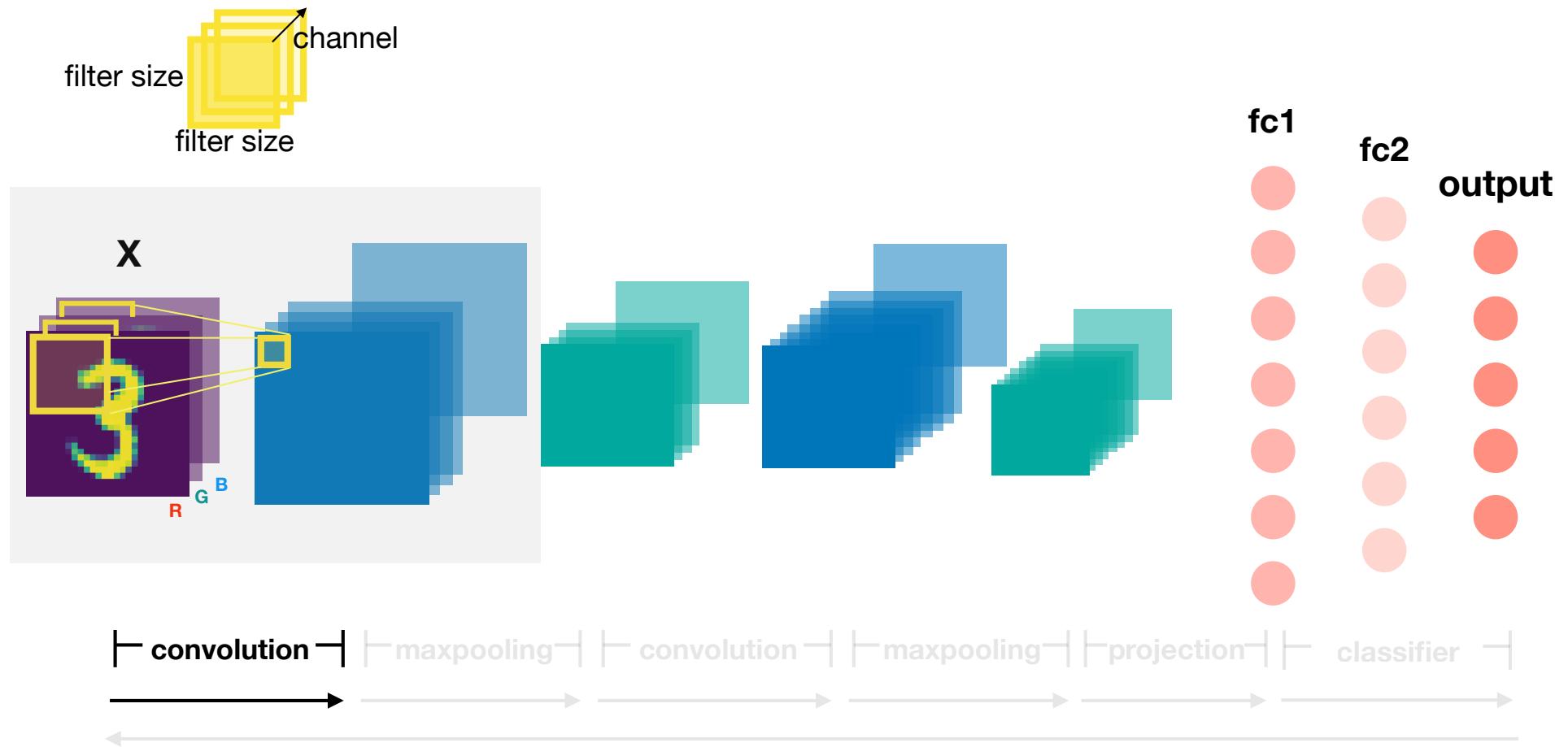
그 결과물을 분류해주는 **classifier** 부분으로 나누어져 있습니다.

CNN



먼저 input에 convolution과 max pooling을 합니다.
이 과정을 좀 더 자세히 살펴보겠습니다.

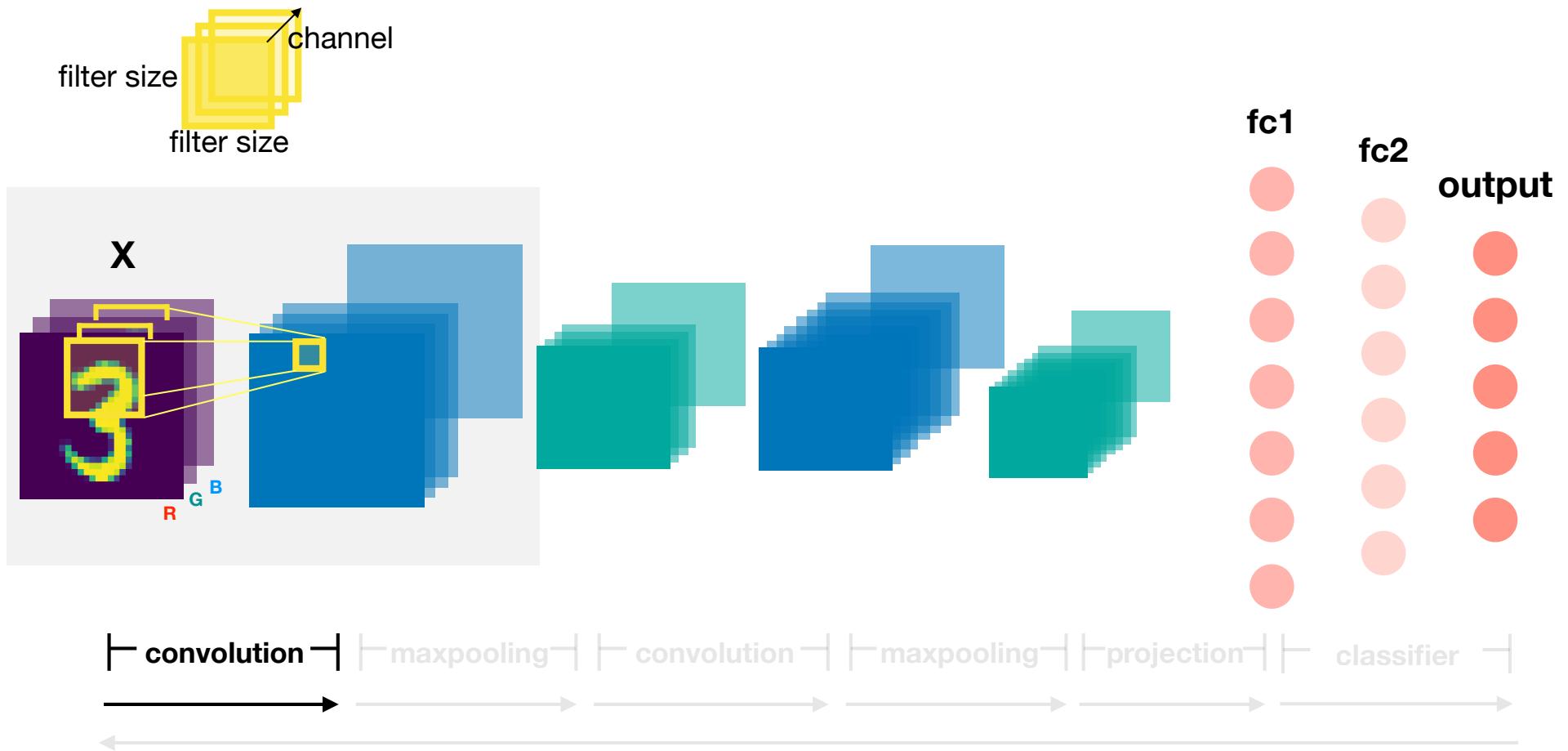
CNN



우선 첫 번째 필터(노란색)를 input의 각 channel마다 씌웁니다.

그렇게 convolution한 값을 모두 더해 첫 번째 feature map (제일 앞의 파란색)을 만듭니다.

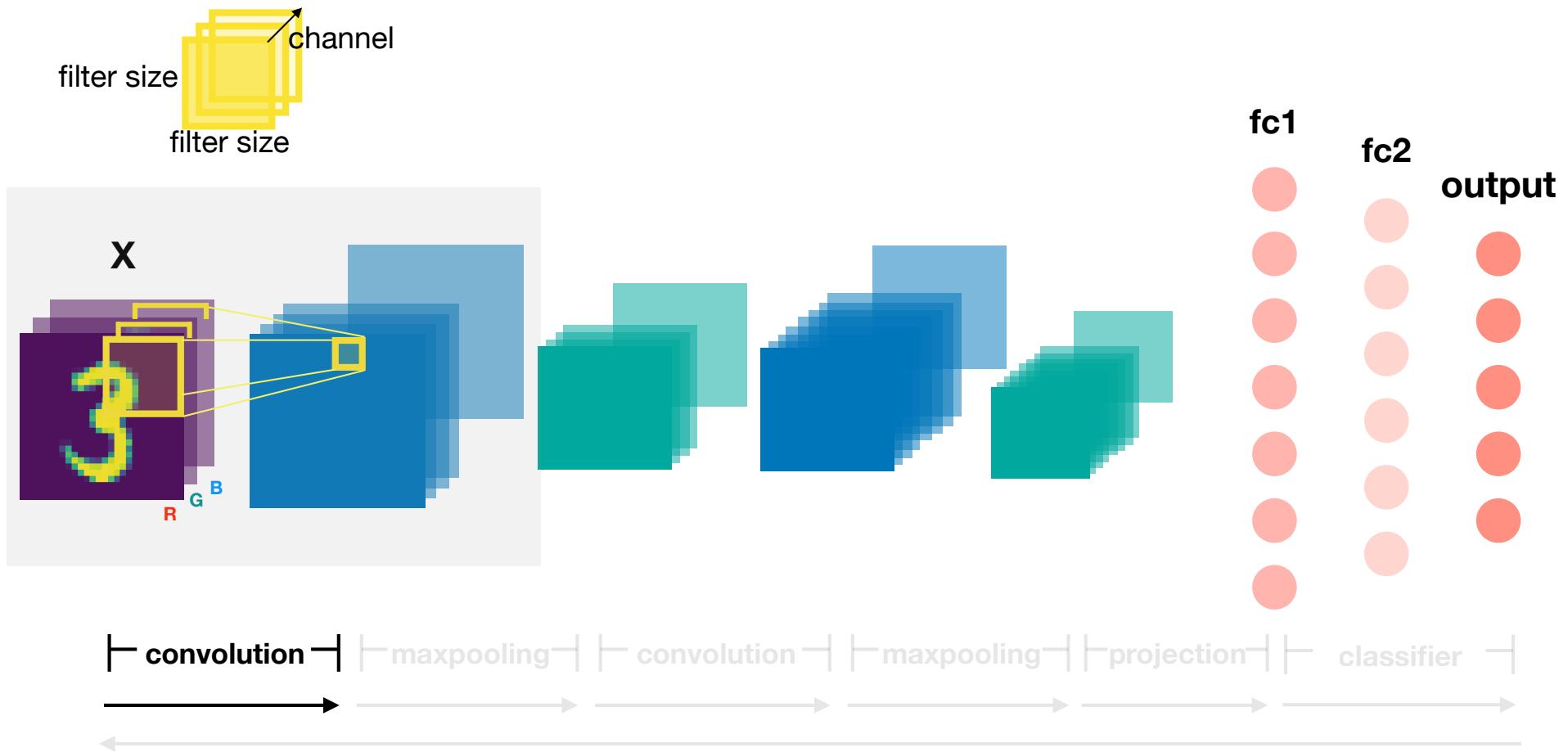
CNN



우선 첫 번째 필터(노란색)를 input의 각 channel마다 씌웁니다.

그렇게 convolution한 값을 모두 더해 첫 번째 feature map (제일 앞의 파란색)을 만듭니다.

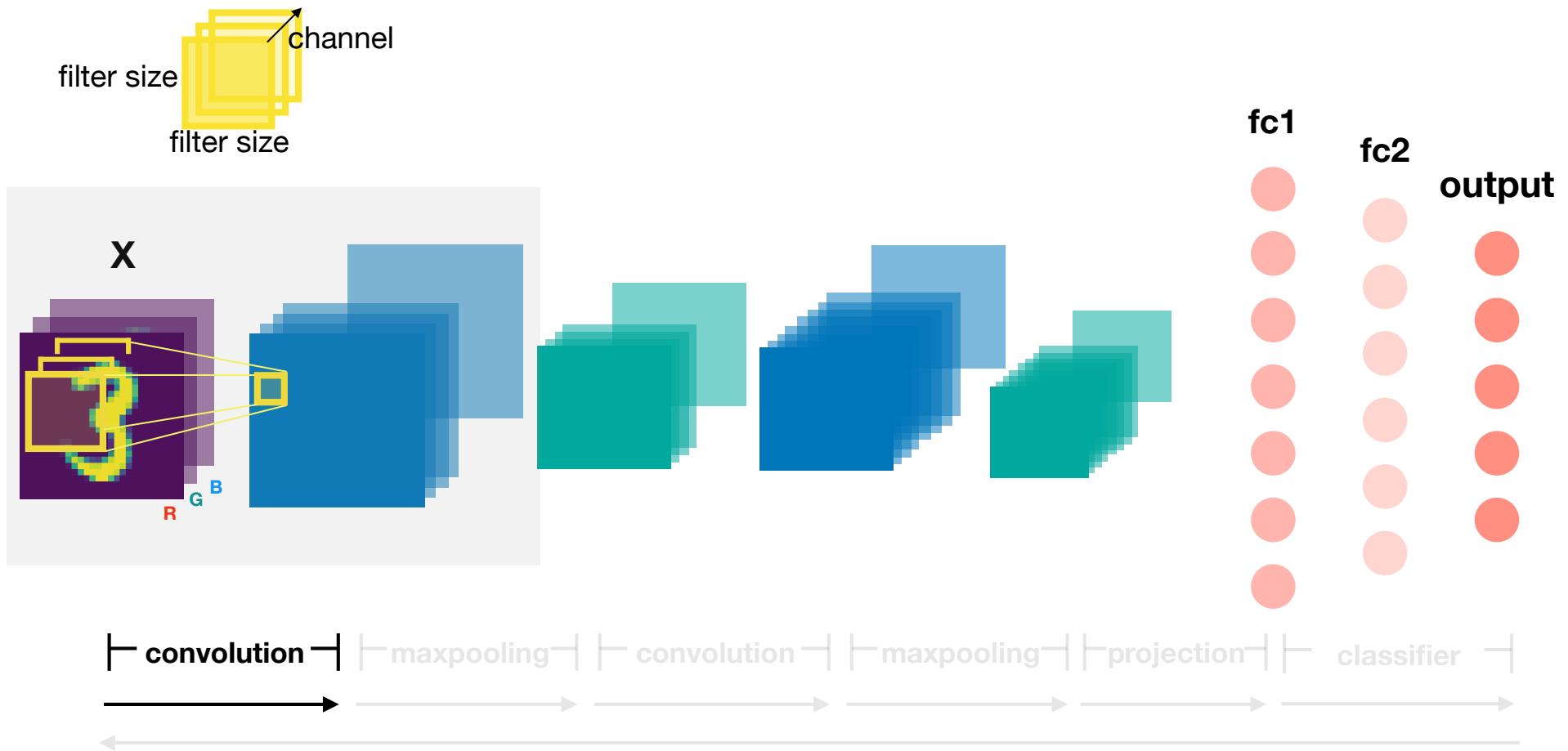
CNN



우선 첫 번째 필터(노란색)를 input의 각 channel마다 씌웁니다.

그렇게 convolution한 값을 모두 더해 첫 번째 feature map (제일 앞의 파란색)을 만듭니다.

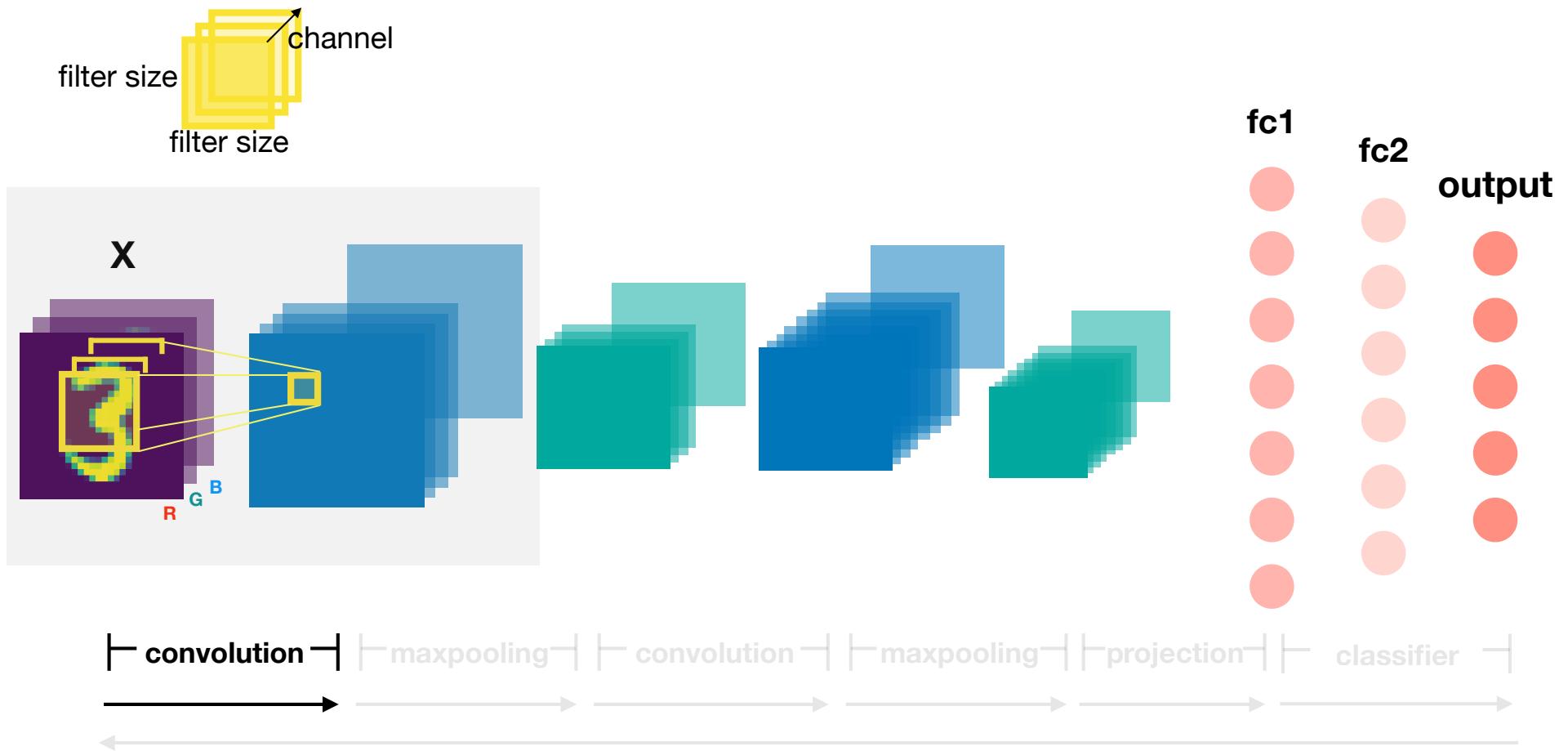
CNN



우선 첫 번째 필터(노란색)를 input의 각 channel마다 씌웁니다.

그렇게 convolution한 값을 모두 더해 첫 번째 feature map (제일 앞의 파란색)을 만듭니다.

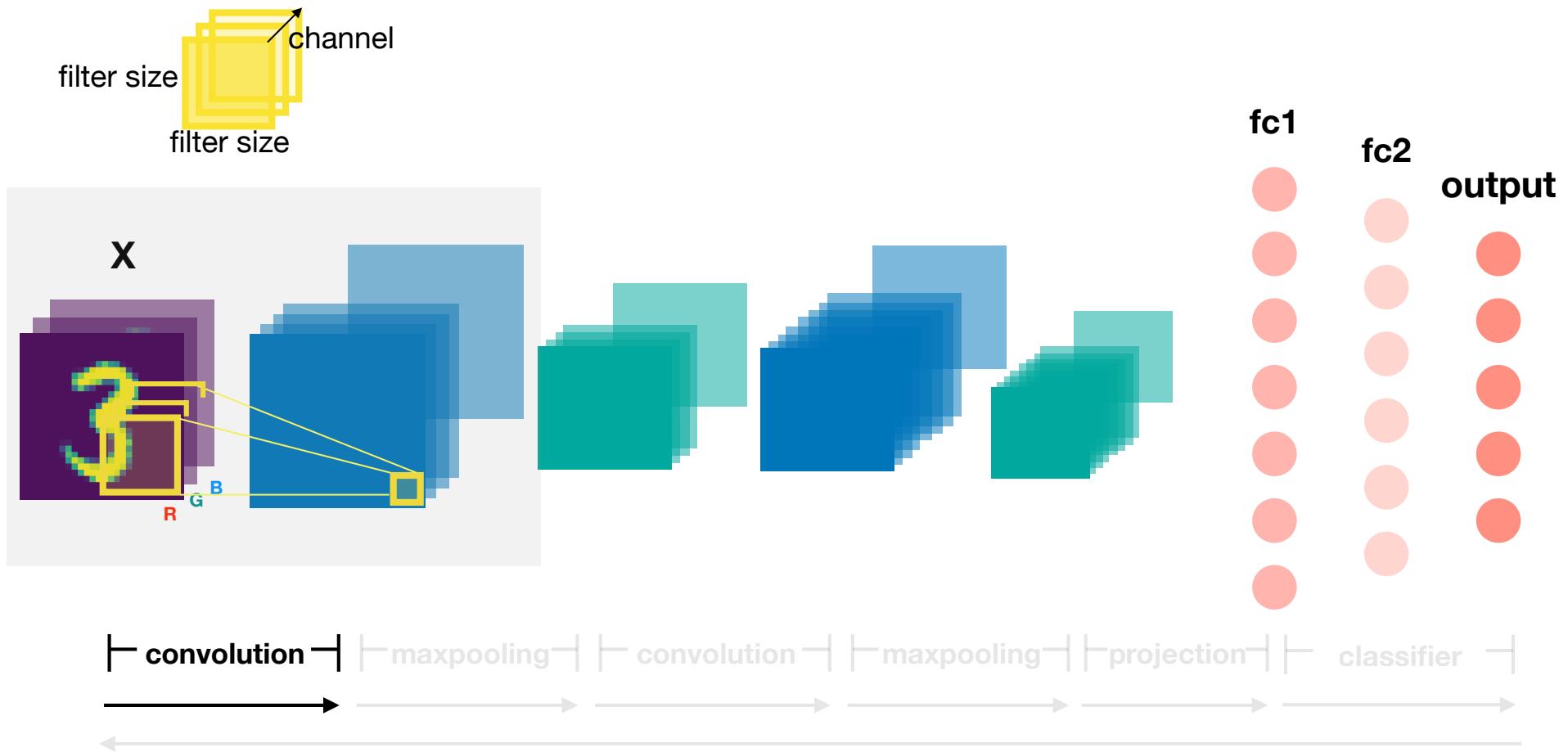
CNN



우선 첫 번째 필터(노란색)를 input의 각 channel마다 씌웁니다.

그렇게 convolution한 값을 모두 더해 첫 번째 feature map (제일 앞의 파란색)을 만듭니다.

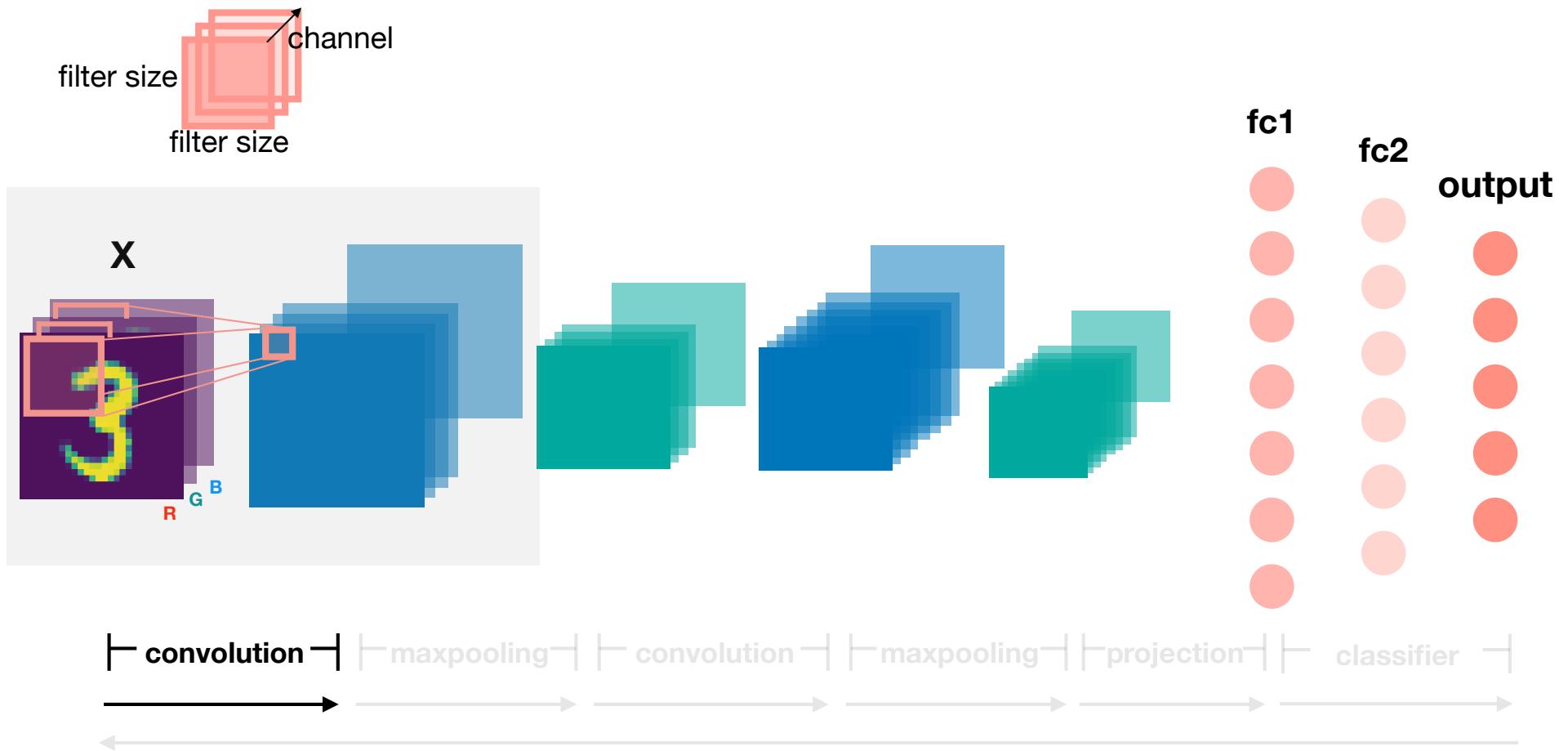
CNN



우선 첫 번째 필터(노란색)를 input의 각 channel마다 씌웁니다.

그렇게 convolution한 값을 모두 더해 첫 번째 feature map (제일 앞의 파란색)을 만듭니다.

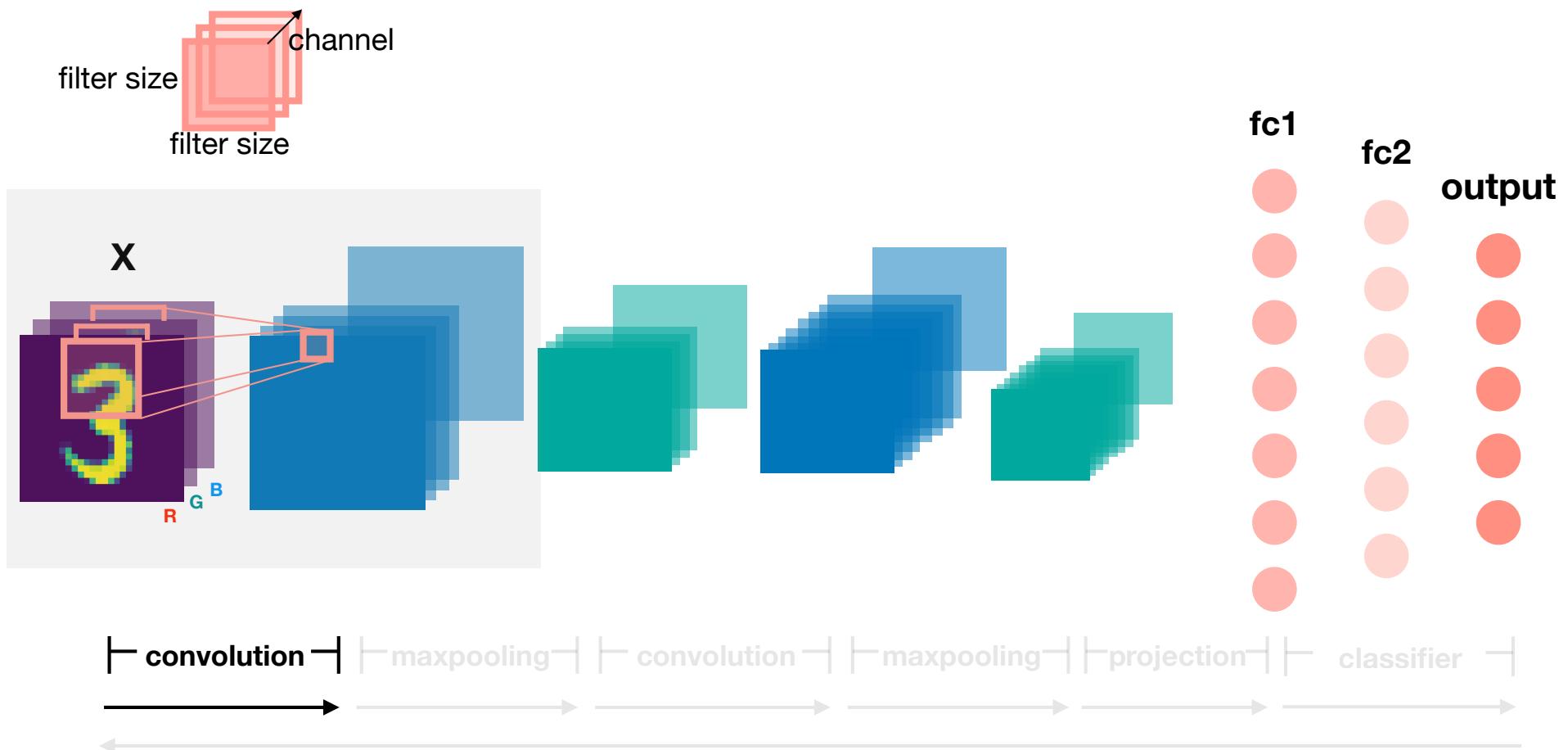
CNN



다음은 두 번째 필터(분홍색)를 input의 각 channel마다 씌웁니다.

그렇게 convolution한 값을 모두 더해 두 번째 feature map (두 번째 앞의 파란색)을 만듭니다.

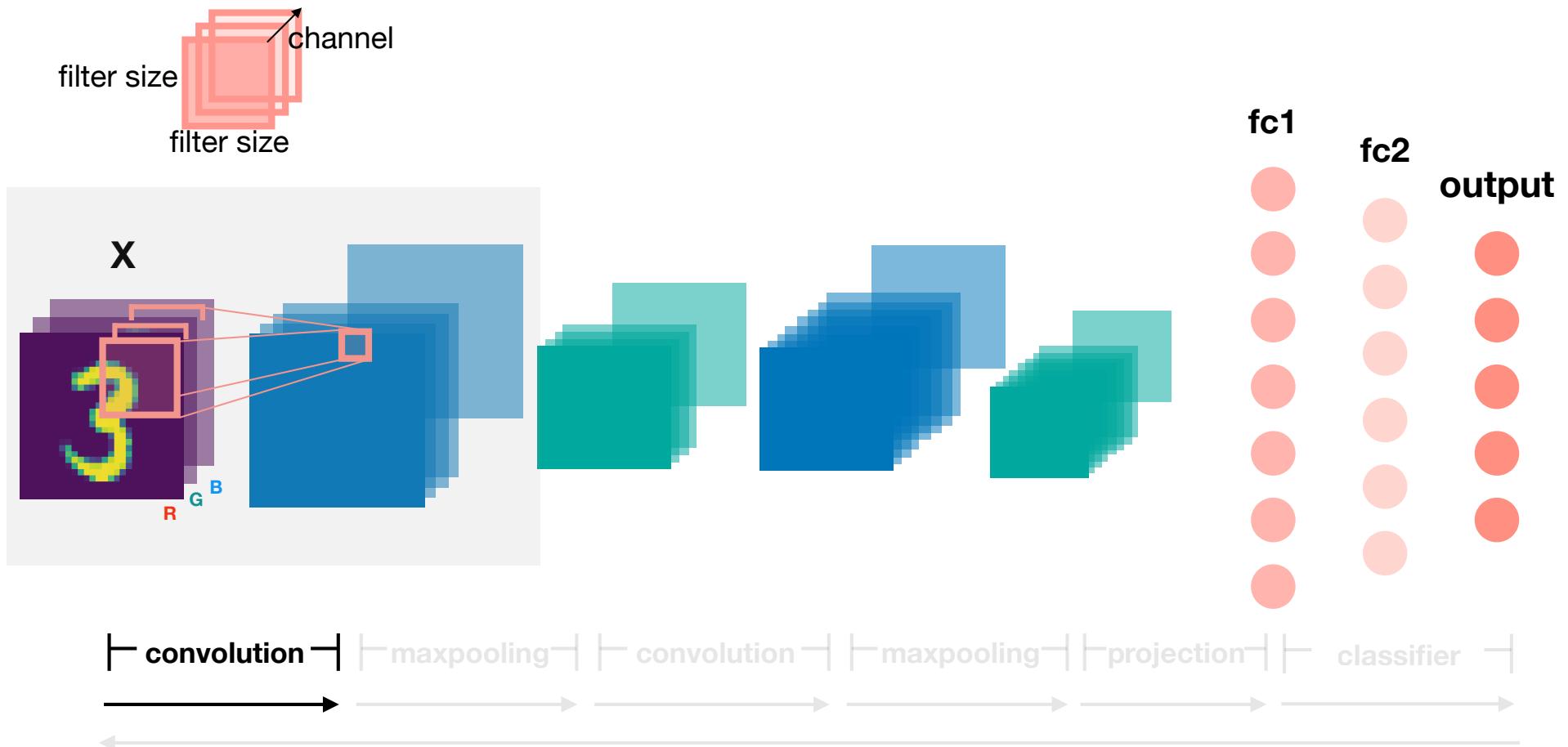
CNN



다음은 두 번째 필터(분홍색)를 input의 각 channel마다 씌웁니다.

그렇게 convolution한 값을 모두 더해 두 번째 feature map (두 번째 앞의 파란색)을 만듭니다.

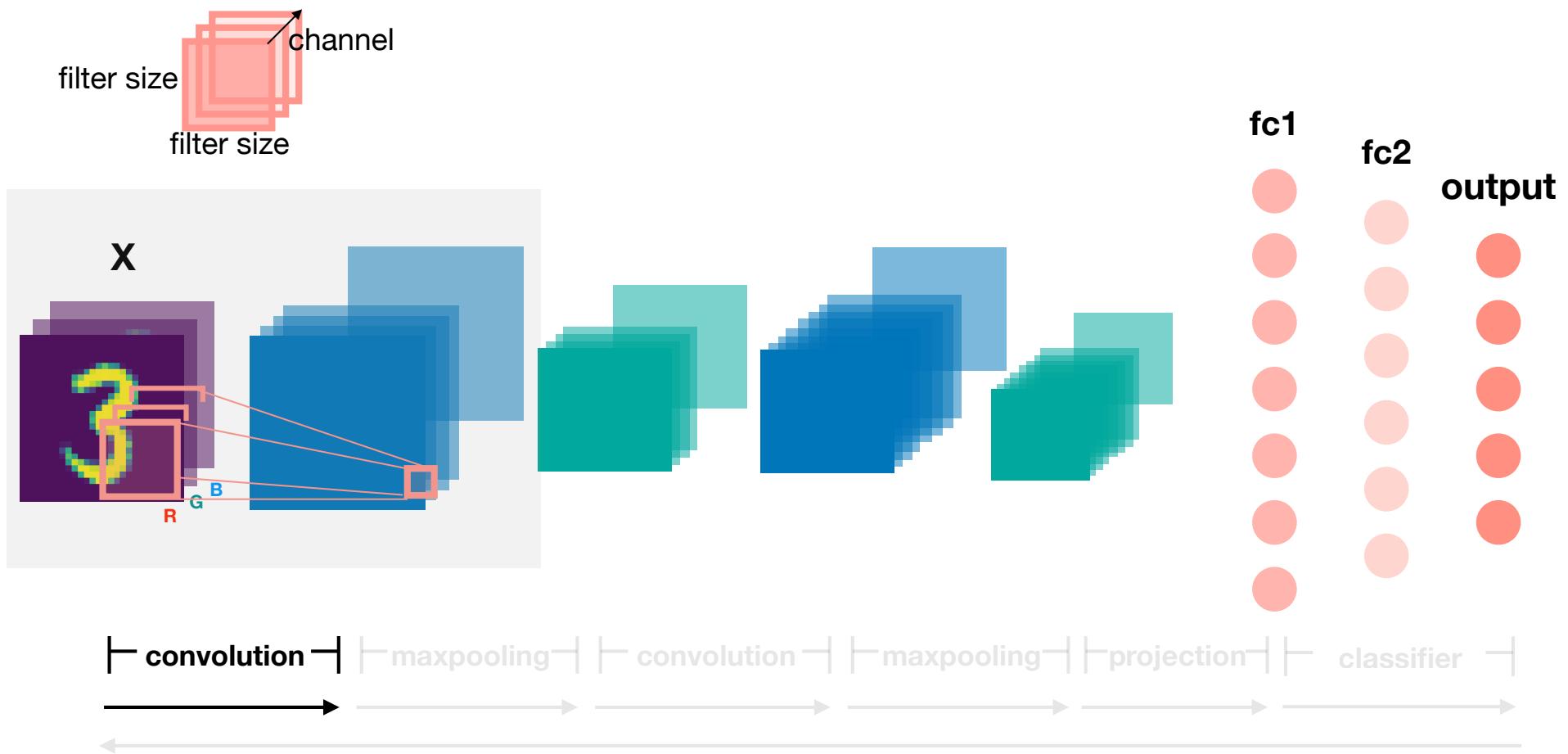
CNN



다음은 두 번째 필터(분홍색)를 input의 각 channel마다 씌웁니다.

그렇게 convolution한 값을 모두 더해 두 번째 feature map (두 번째 앞의 파란색)을 만듭니다.

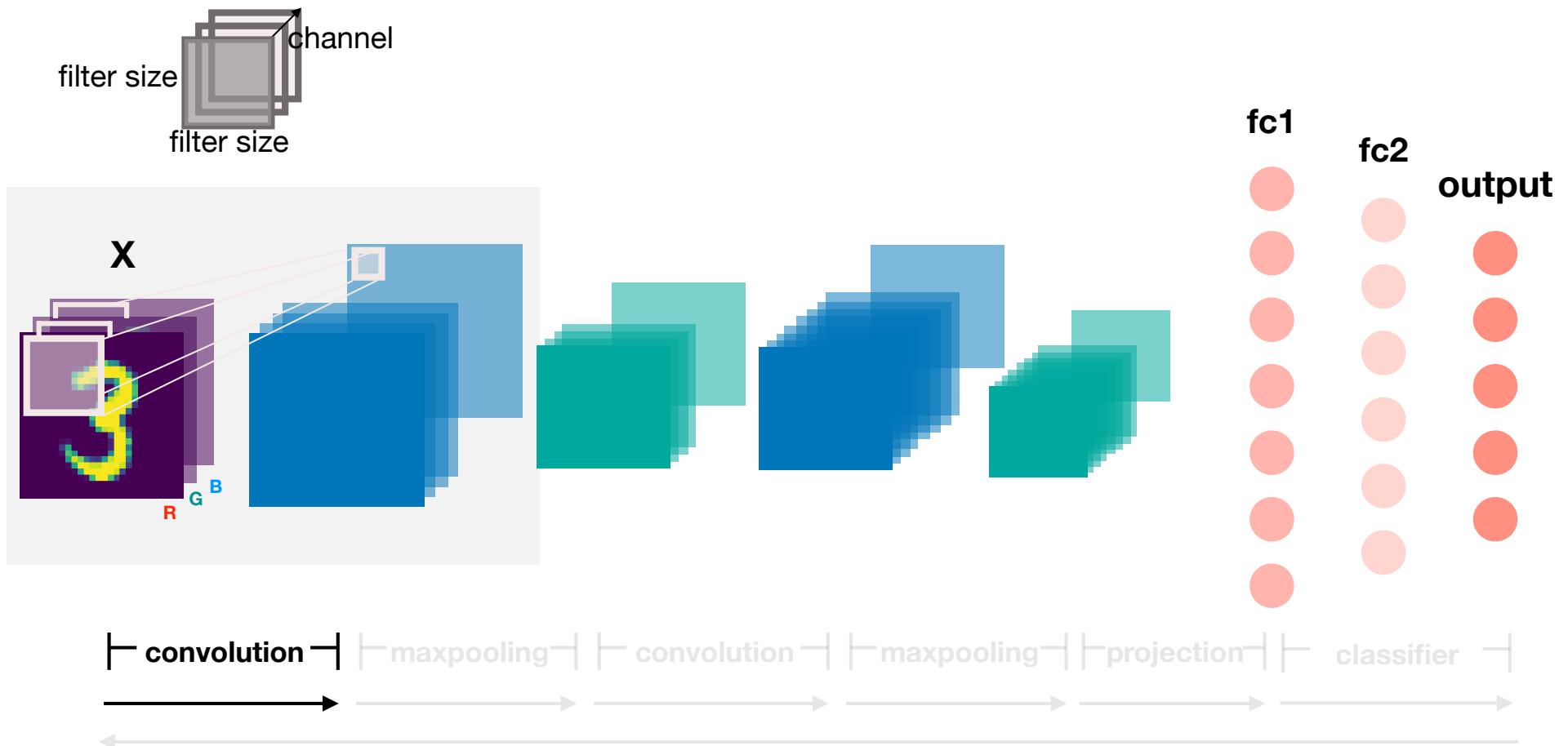
CNN



다음은 두 번째 필터(분홍색)를 input의 각 channel마다 씌웁니다.

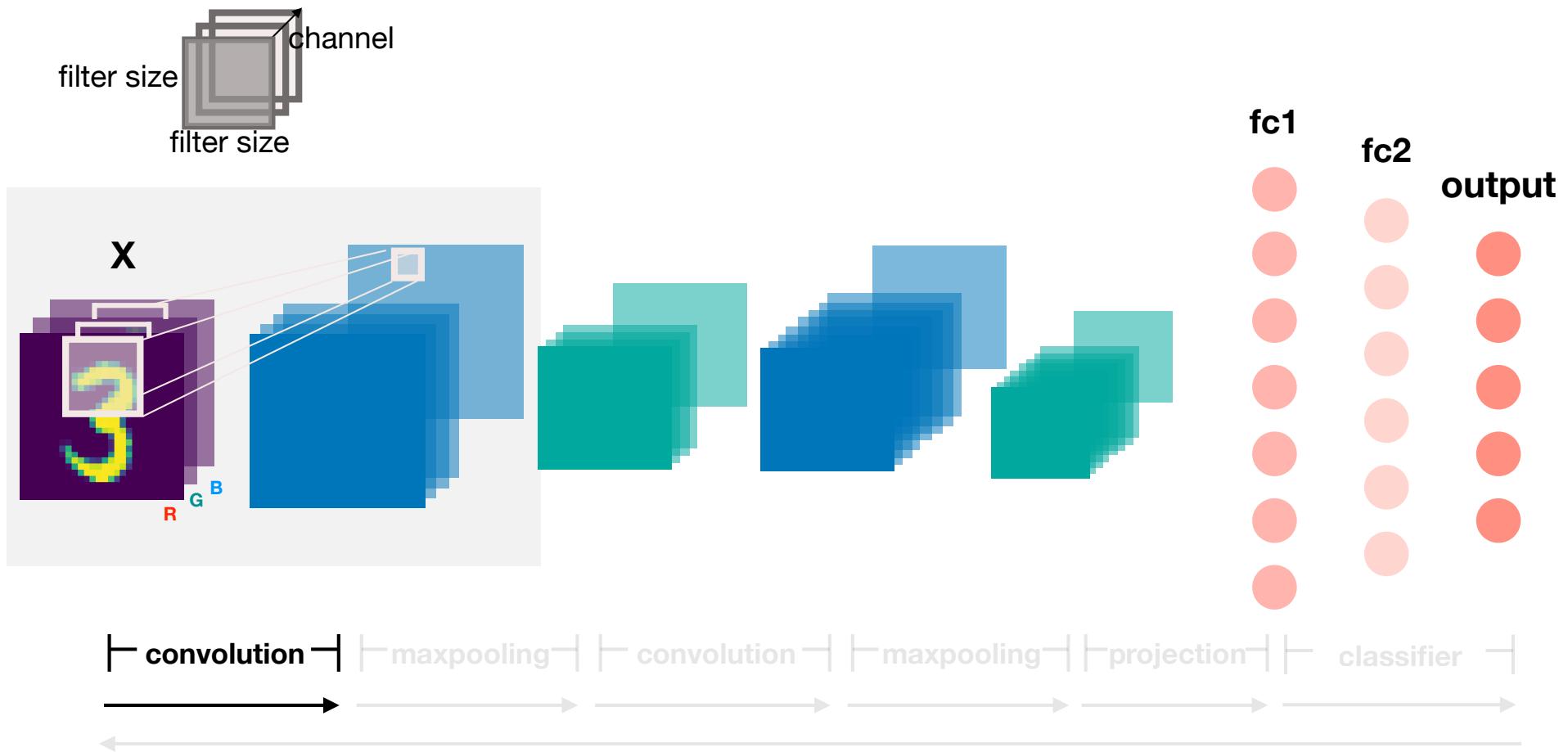
그렇게 convolution한 값을 모두 더해 두 번째 feature map (두 번째 앞의 파란색)을 만듭니다.

CNN



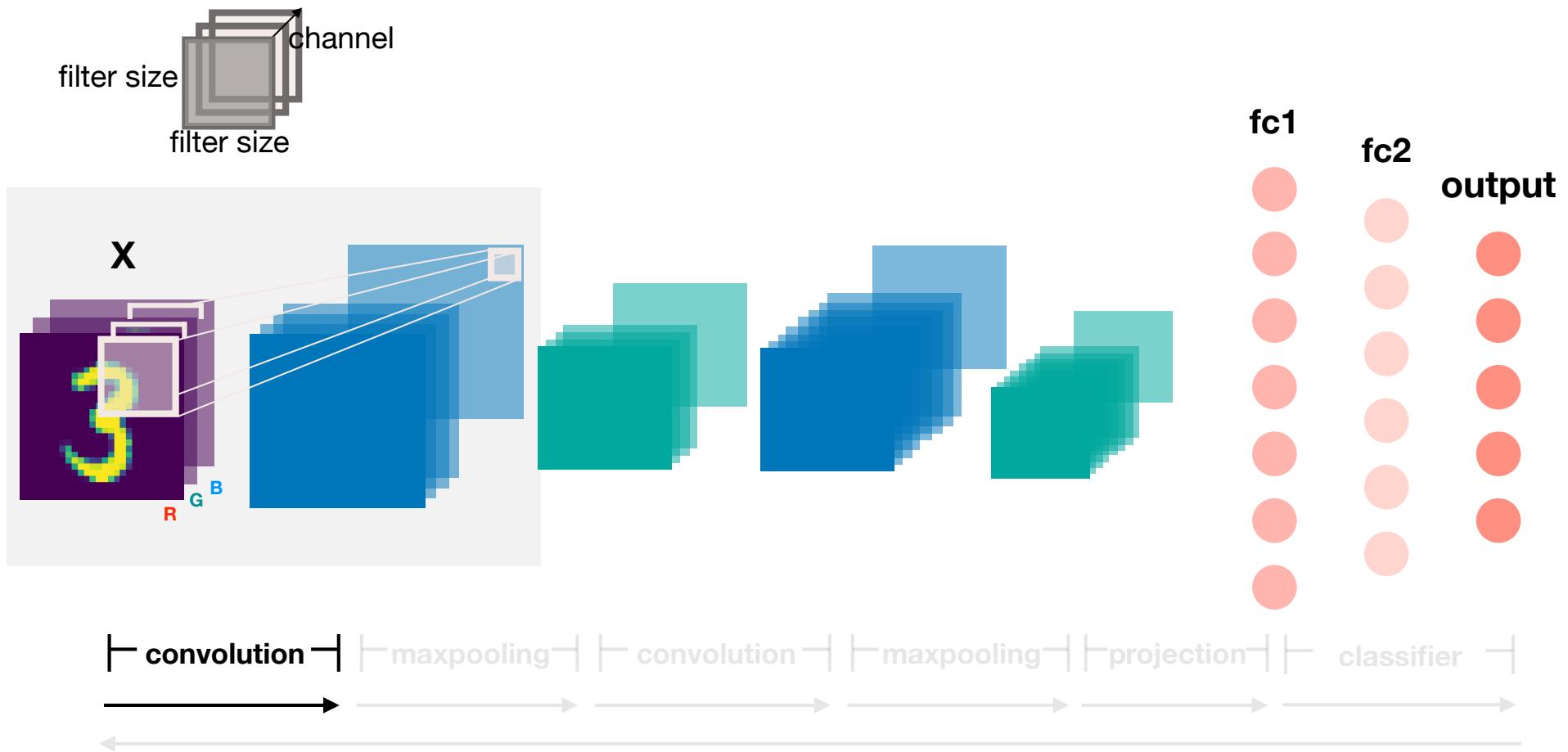
마지막 feature map (제일 뒤의 파란색)을 만들 때까지 반복합니다.

CNN



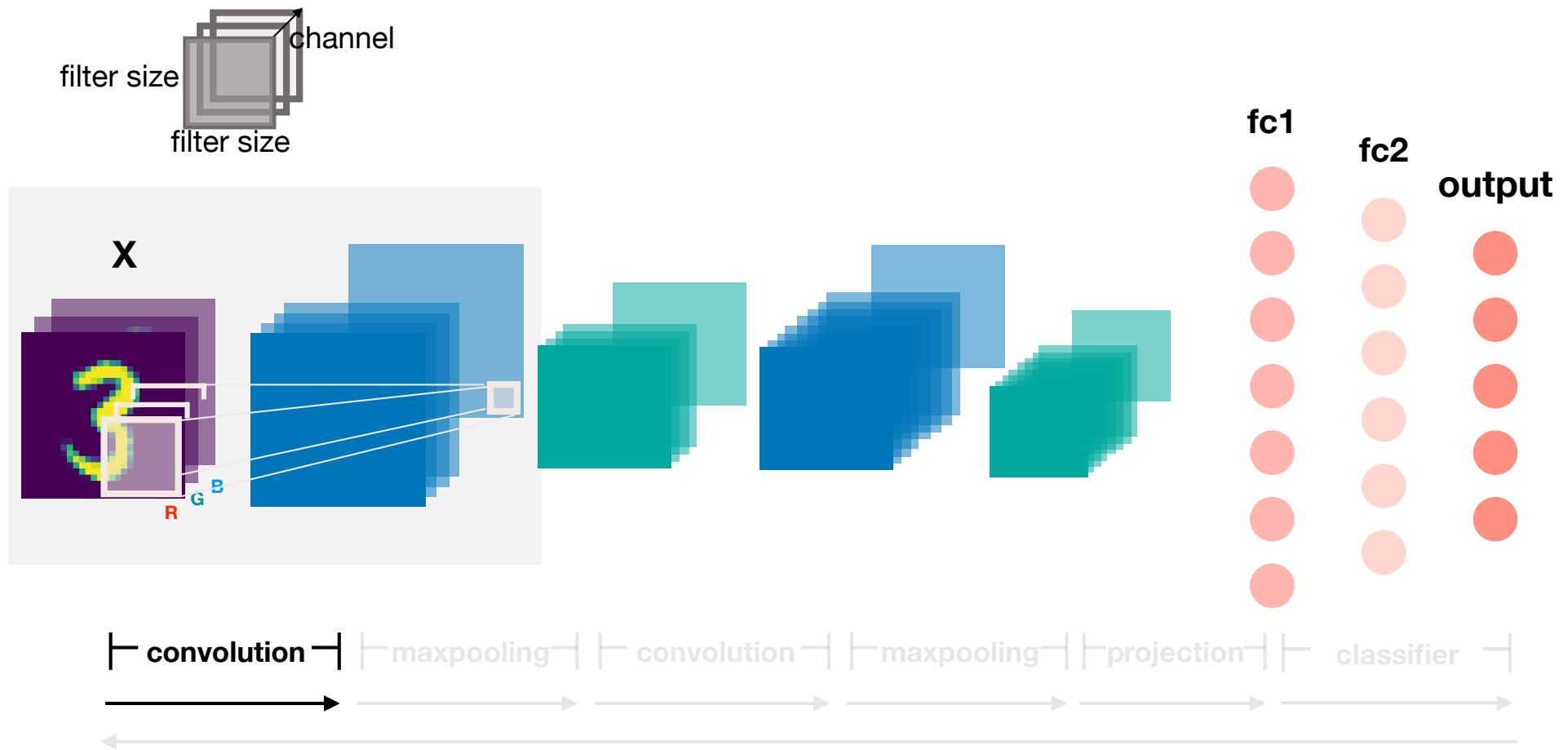
마지막 feature map (제일 뒤의 파란색)을 만들 때까지 반복합니다.

CNN



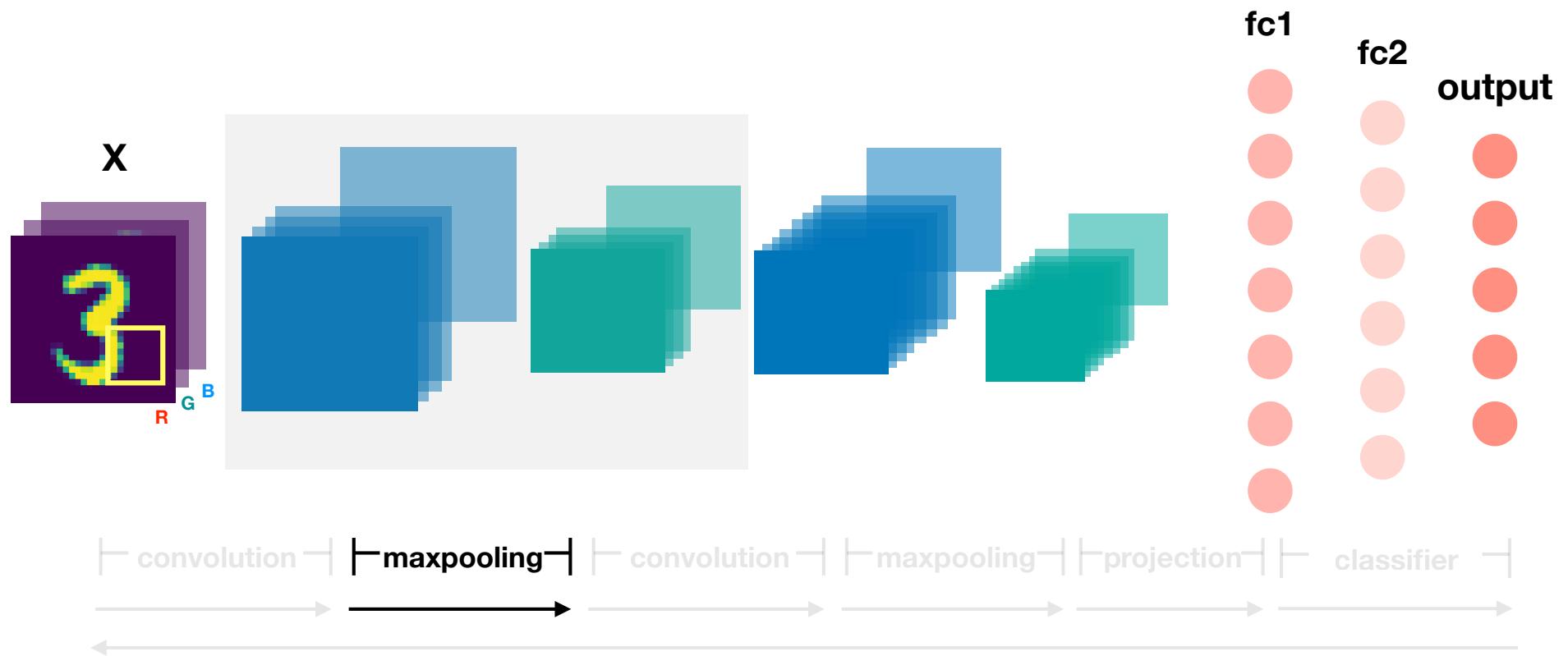
마지막 feature map (제일 뒤의 파란색)을 만들 때까지 반복합니다.

CNN



마지막 feature map (제일 뒤의 파란색)을 만들 때까지 반복합니다.

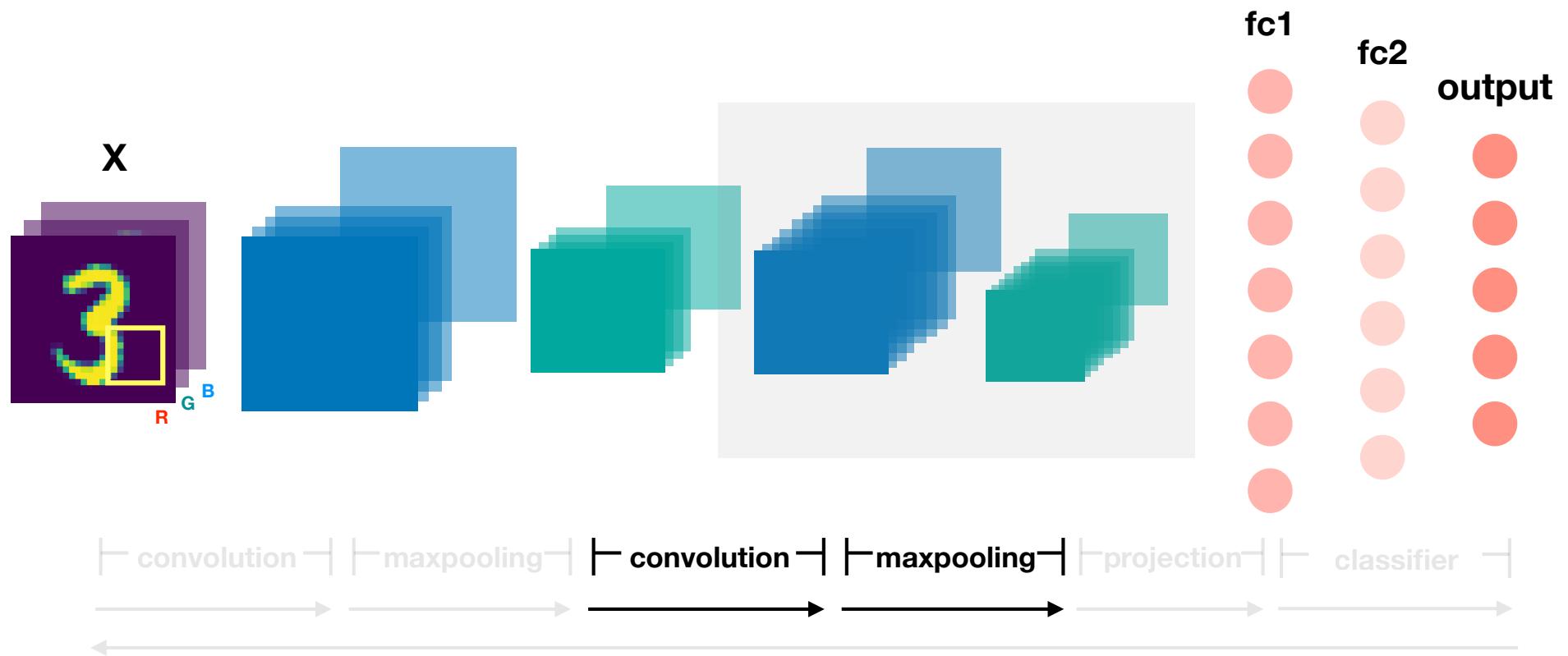
CNN



max pooling을 합니다.

이 때 feature map (파란색)과 pooling된 feature map (초록색)은 개수가 동일합니다.

CNN



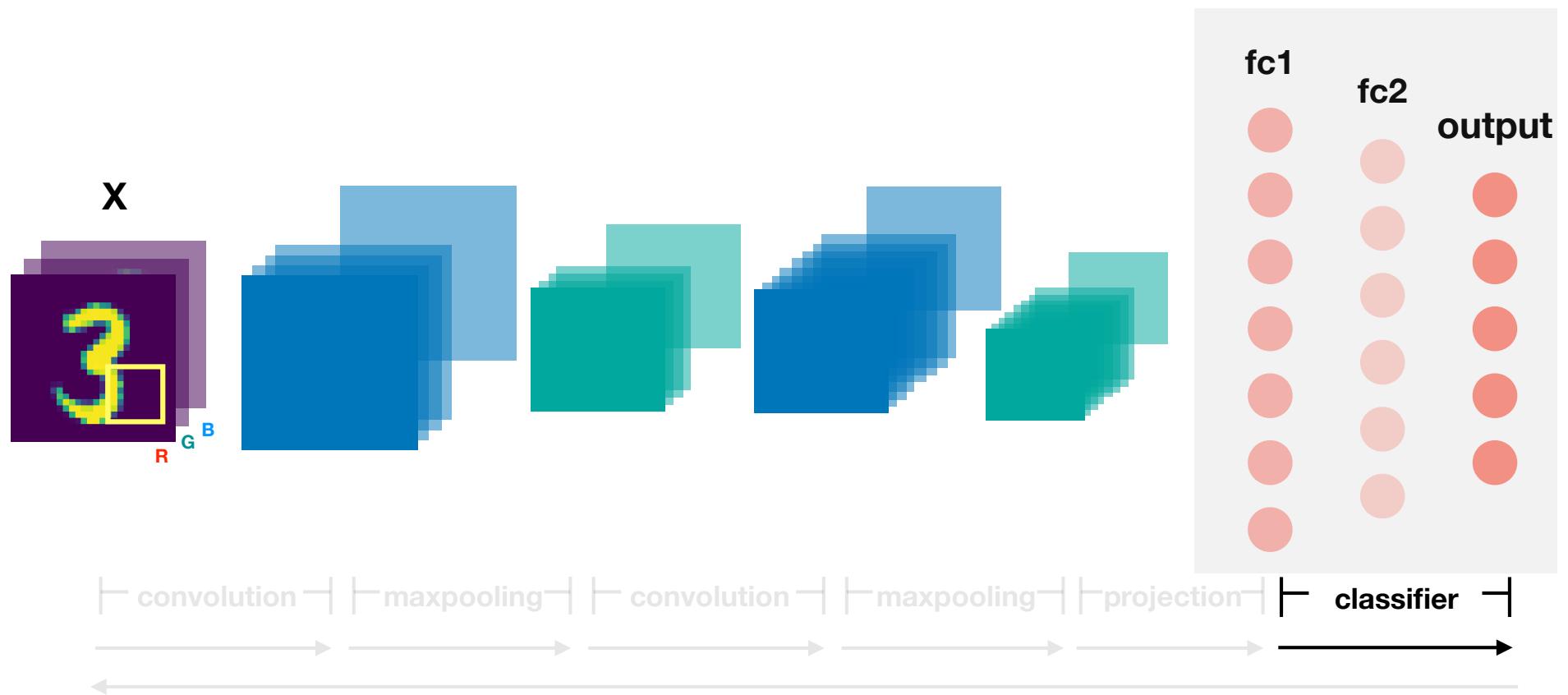
convolution과 max pooling을 반복합니다.
(원하는 만큼 하면 됩니다. 단, 이 예시에선 각각 2회만 하고 있습니다.)

CNN



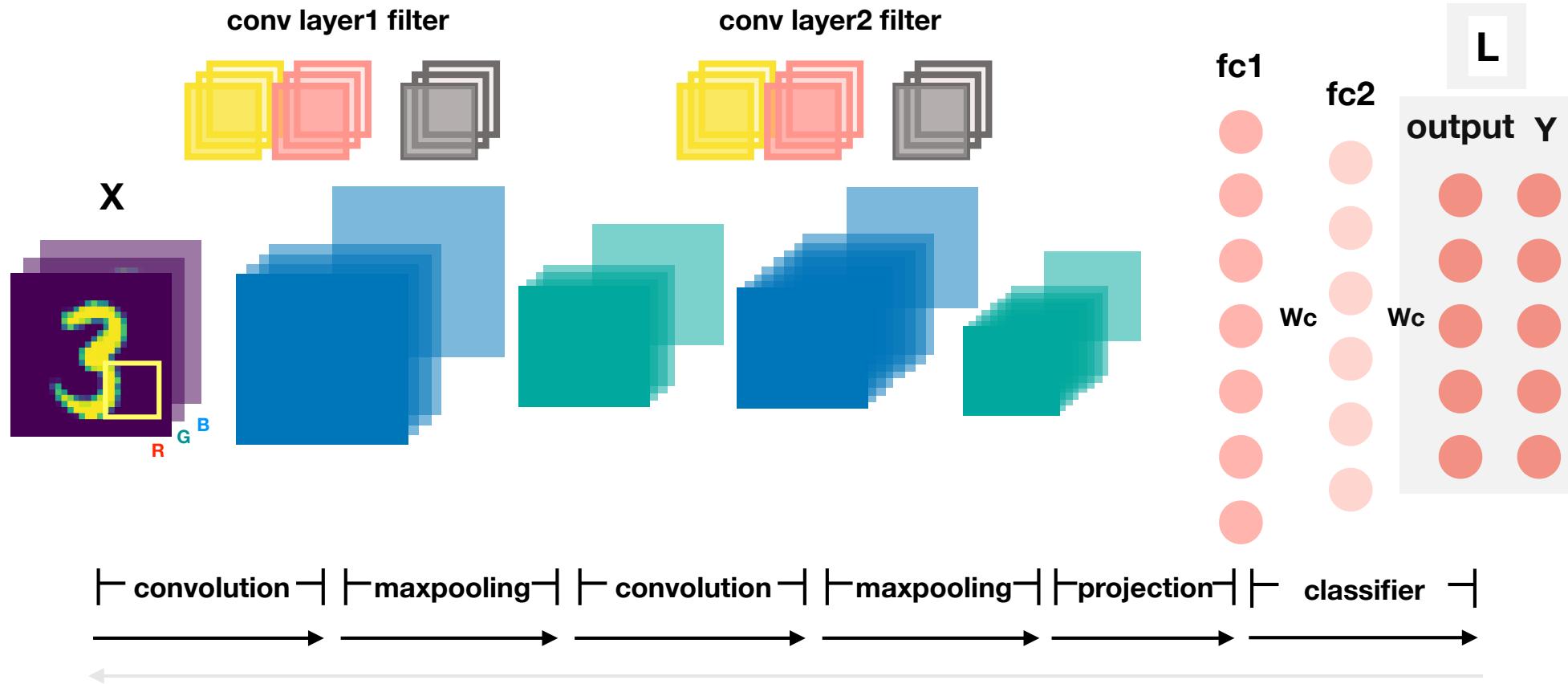
이제 완성된 feature map을 모두 모아 한 줄로 나열한 뒤 fc1 layer에 넣습니다.

CNN

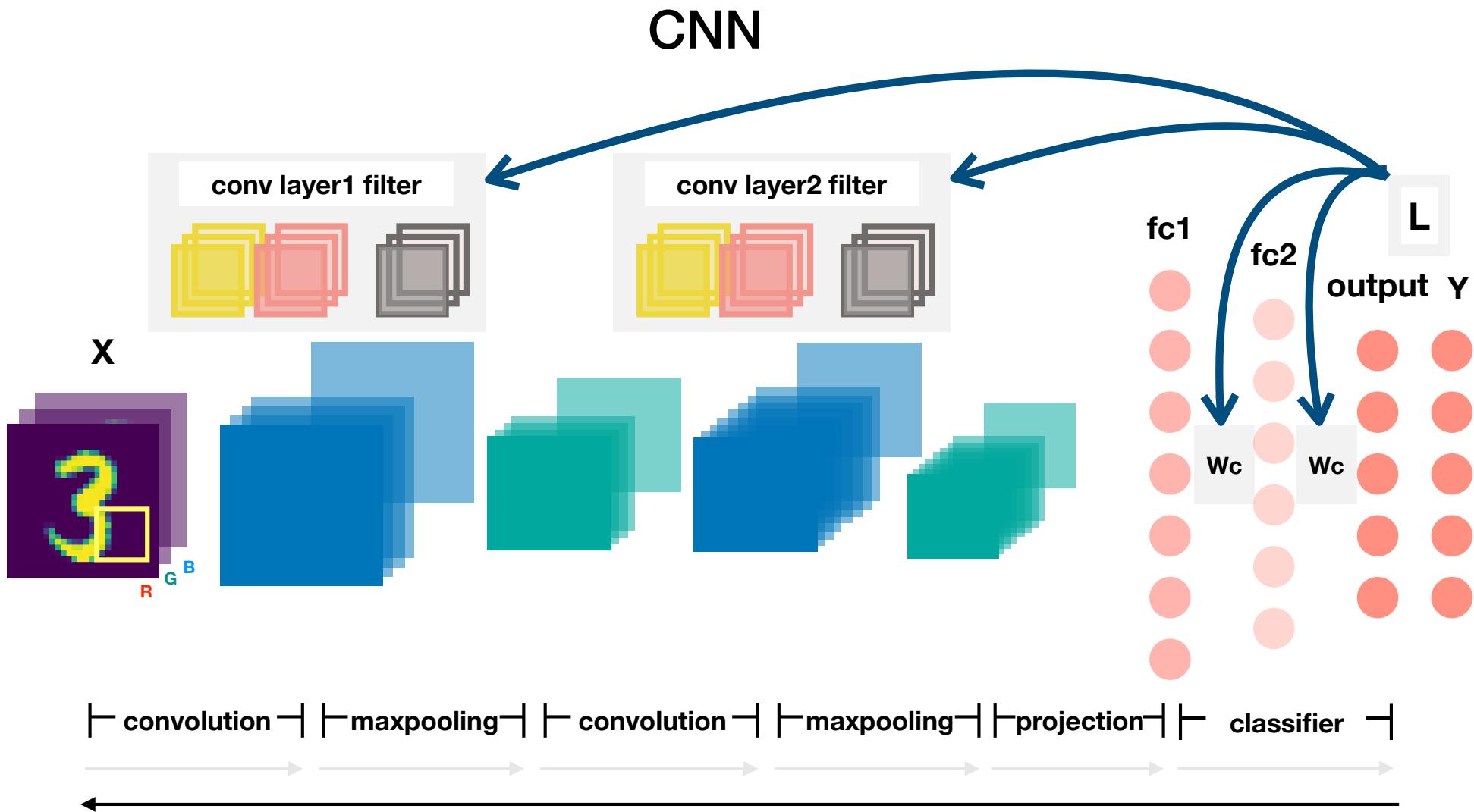


classifier 부분은 ANN과 동일합니다.

CNN



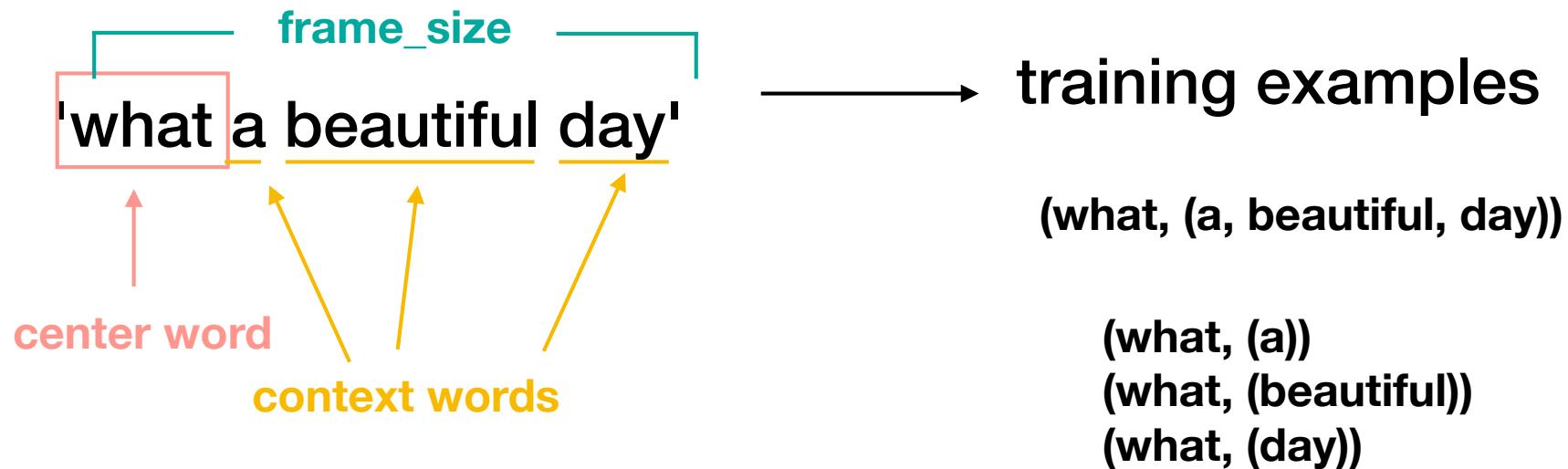
output과 정답 target (Y)을 비교하여 loss를 구합니다.



loss를 최소화하는 방향으로, weight을 update 합니다.

WORD2VEC

Word2Vec



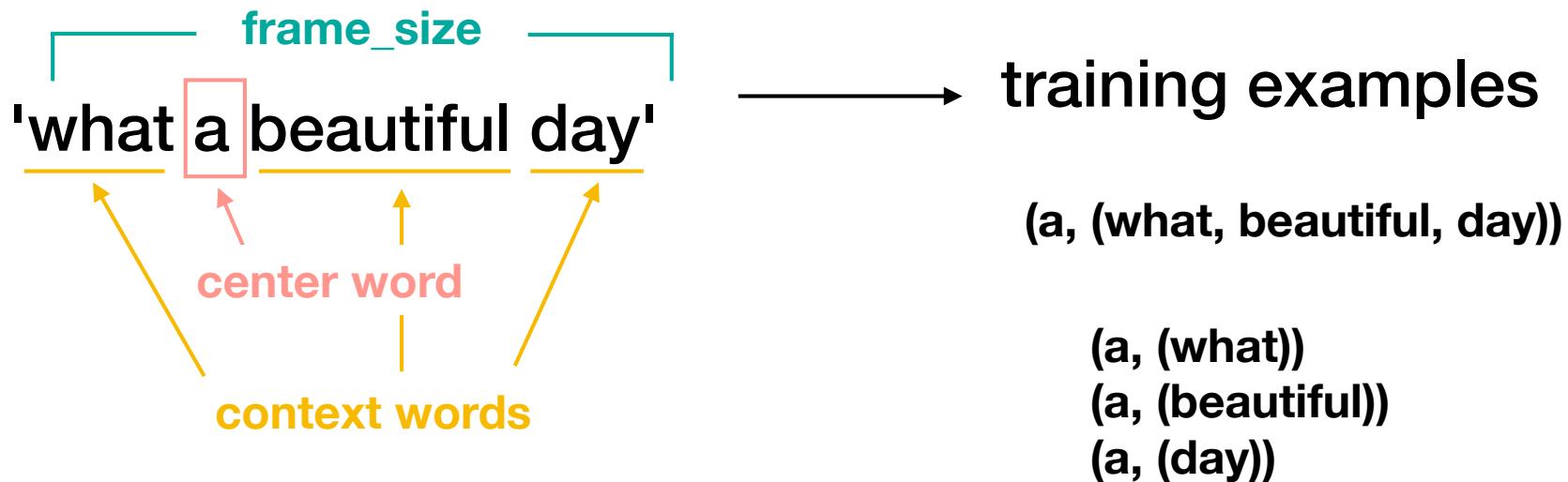
word2vec의 training data 준비방식은 다음과 같습니다.

문장의 각 단어를 center word (분홍색)로 정하고

frame size만큼의 단어를 context word (노란색)로 지정합니다.

하나의 center word가 input0이 되고 그 center word에 대한 context word 각각이 target0이 됩니다.

Word2Vec



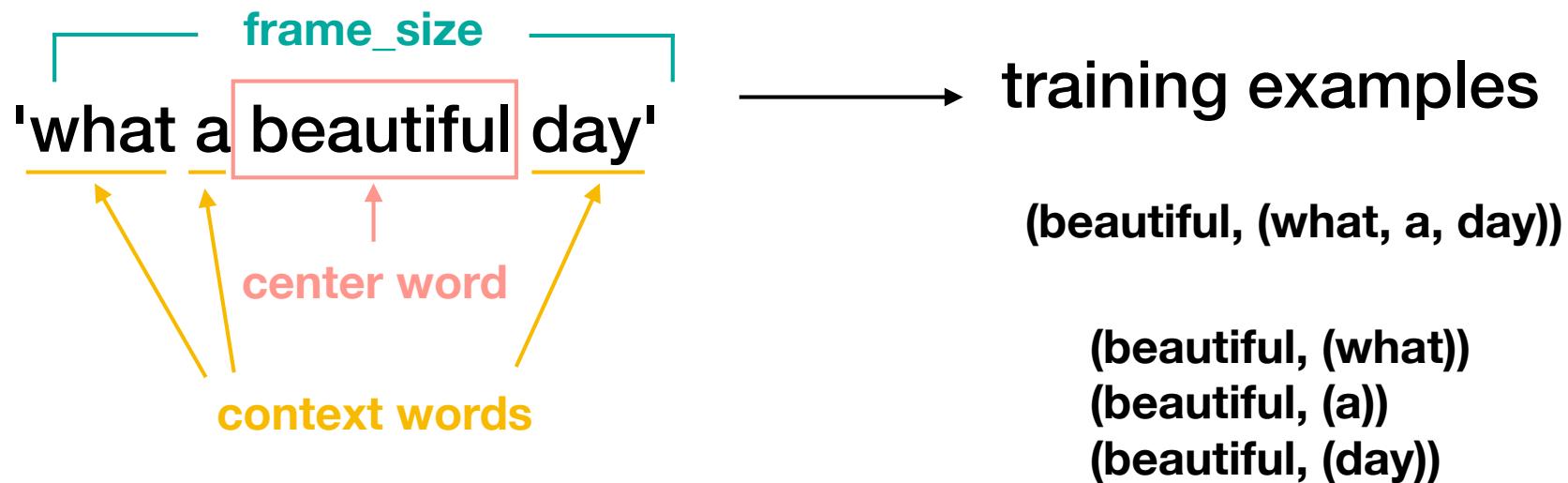
word2vec의 training data 준비방식은 다음과 같습니다.

문장의 각 단어를 center word (분홍색)로 정하고

frame size만큼의 단어를 context word (노란색)로 지정합니다.

하나의 center word가 input0이 되고 그 center word에 대한 context word 각각이 target0이 됩니다.

Word2Vec



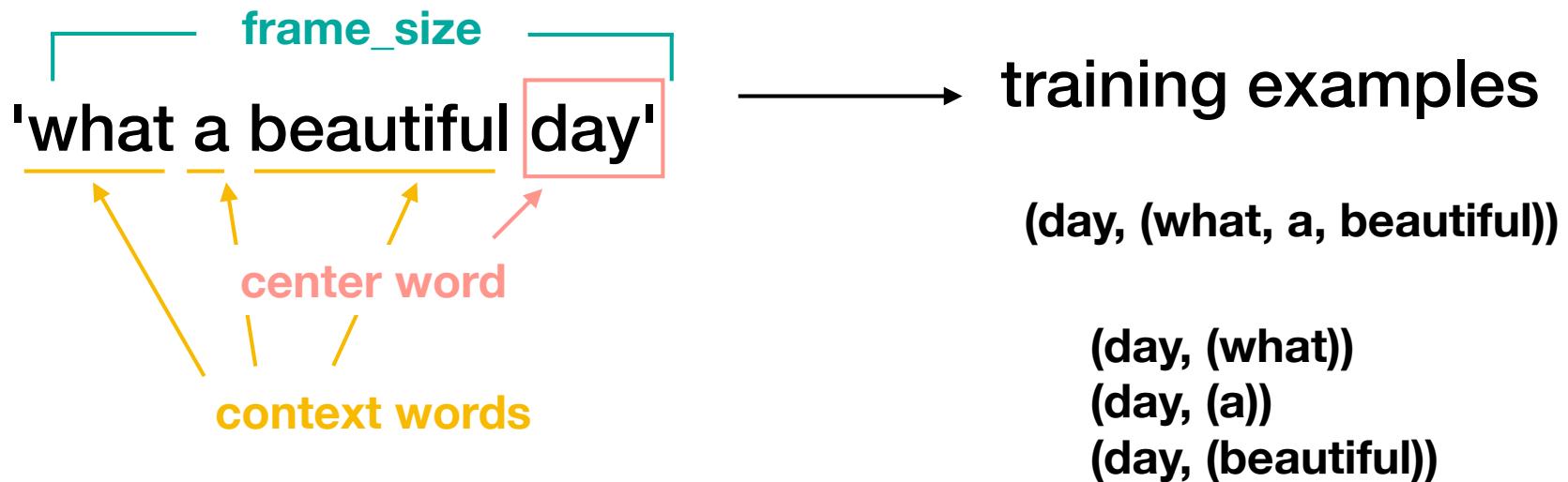
word2vec의 training data 준비방식은 다음과 같습니다.

문장의 각 단어를 center word (분홍색)로 정하고

frame size만큼의 단어를 context word (노란색)로 지정합니다.

하나의 center word가 input0이 되고 그 center word에 대한 context word 각각이 target0이 됩니다.

Word2Vec



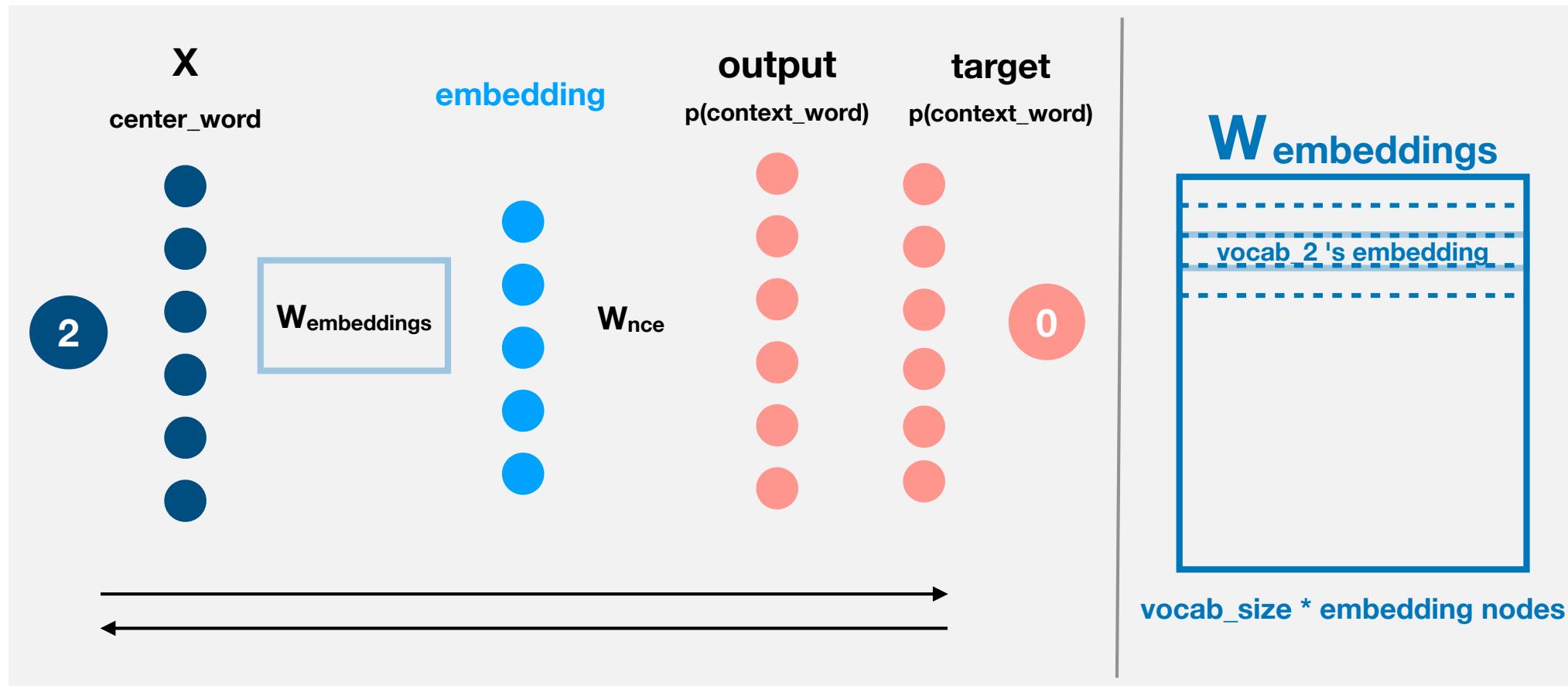
word2vec의 training data 준비방식은 다음과 같습니다.

문장의 각 단어를 center word (분홍색)로 정하고

frame size만큼의 단어를 context word (노란색)로 지정합니다.

하나의 center word가 input0이 되고 그 center word에 대한 context word 각각이 target0이 됩니다.

Word2Vec



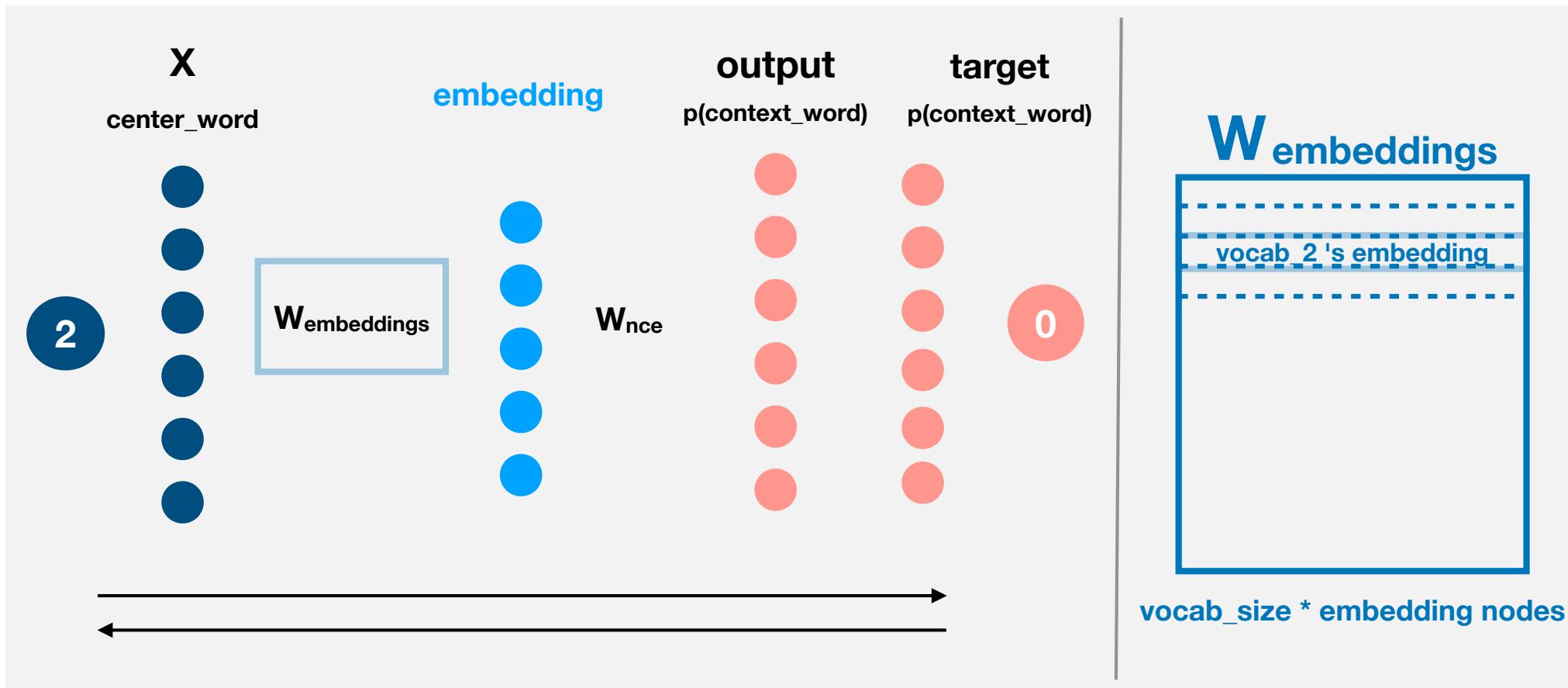
모델구조:

기본적으로 ANN classifier와 같음

input, output, target nodes는 각각 전체 단어의 수

이 예시에서는 이해의 편의를 위하여 nodes 숫자를 제한적으로 표기함

Word2Vec

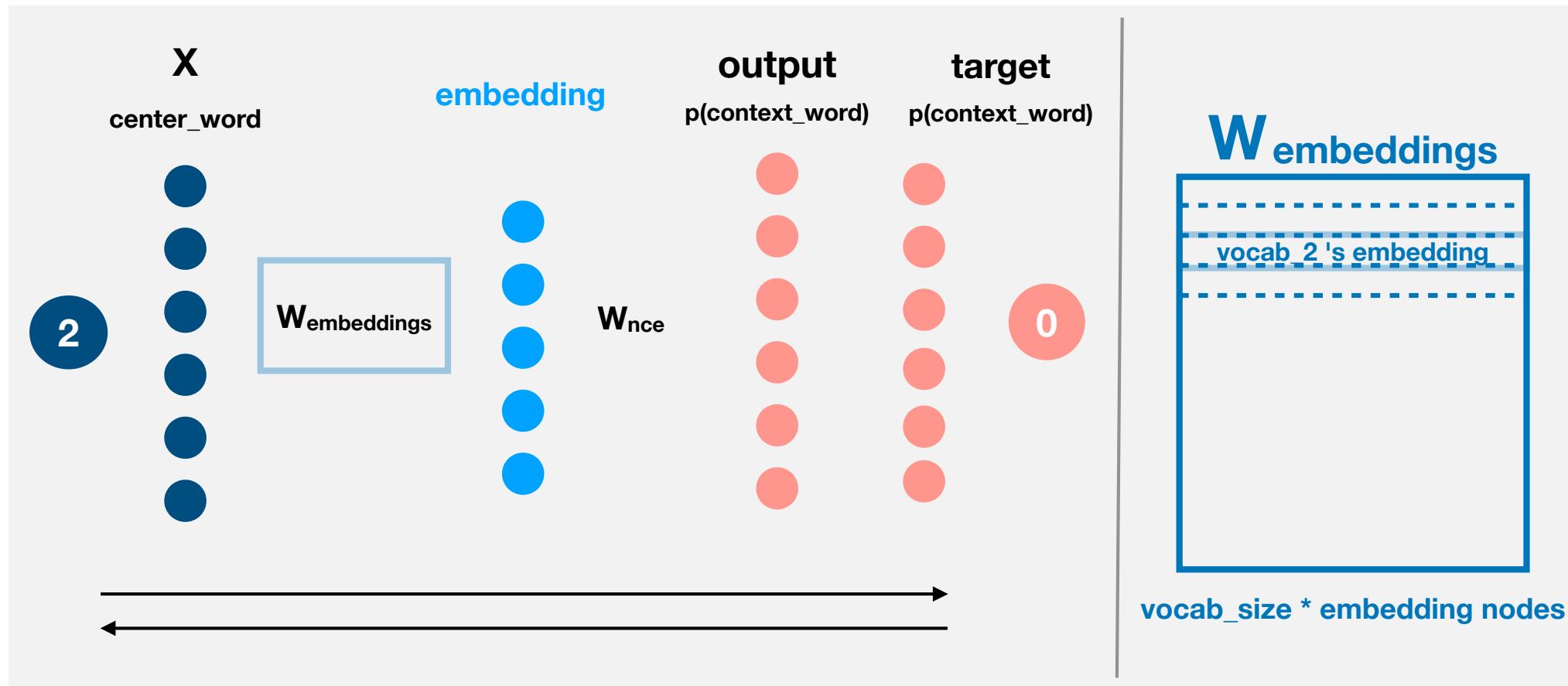


input에 들어간 단어가

hidden layer를 거쳐서

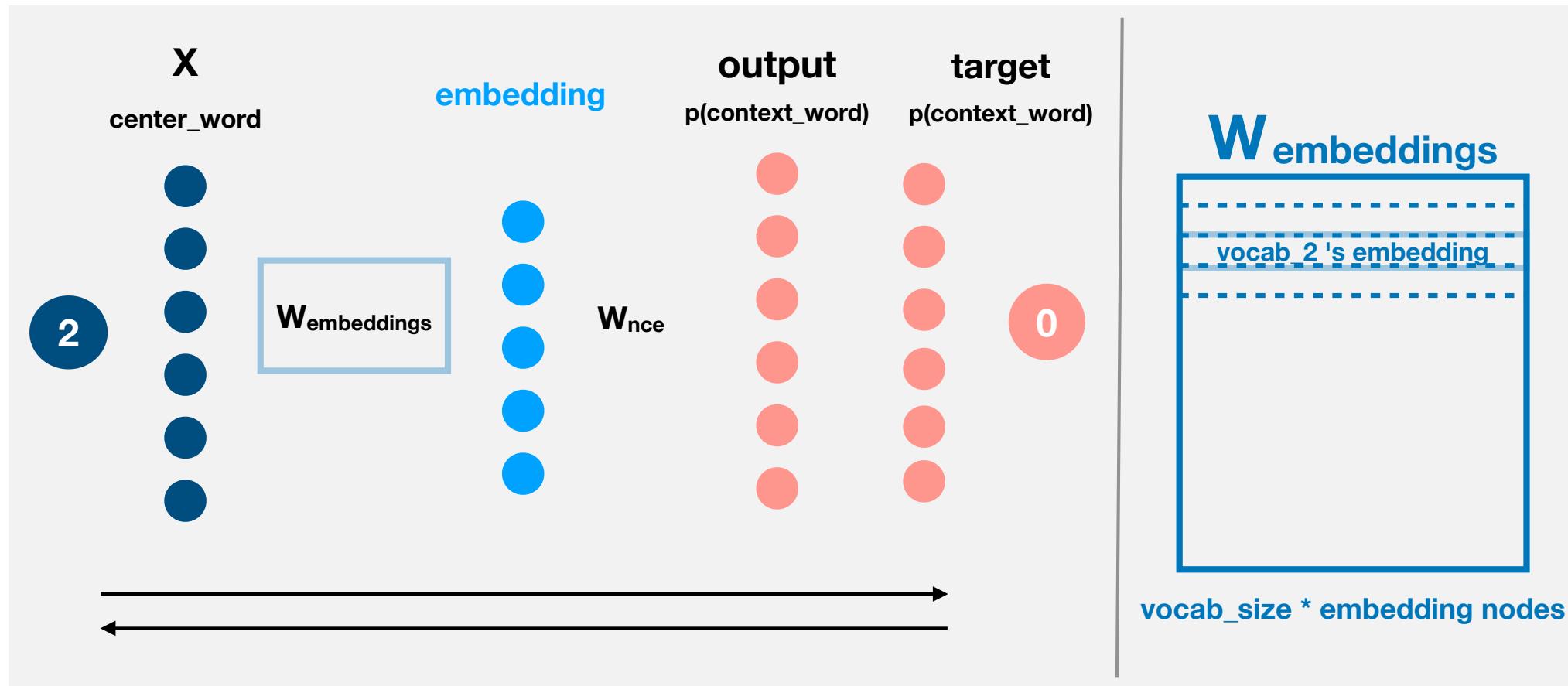
output으로 나온 값과,
target 간의 차이를 최소화하는 것이 목표입니다.

Word2Vec



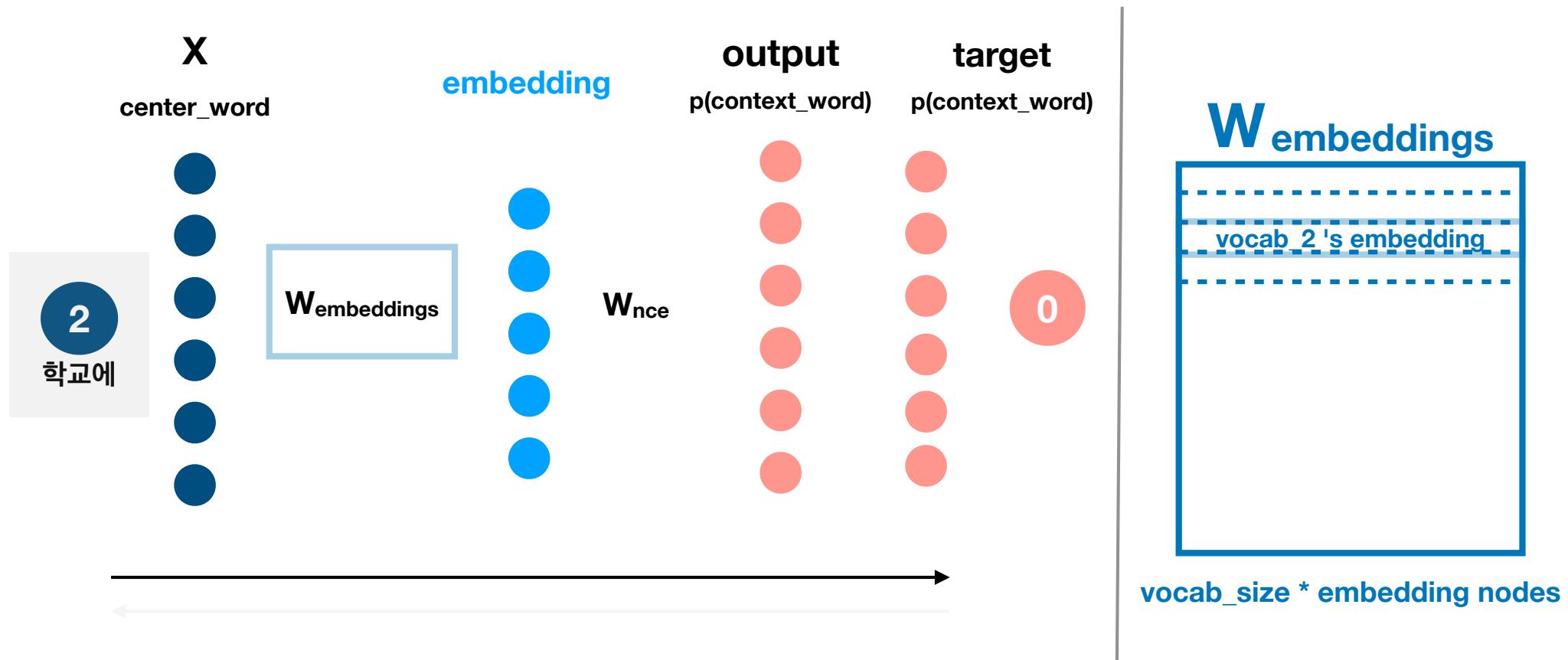
이 과정을 통해 center word와 context word 간의 관계를 vector로 표현하는 것이 목표입니다.

Word2Vec



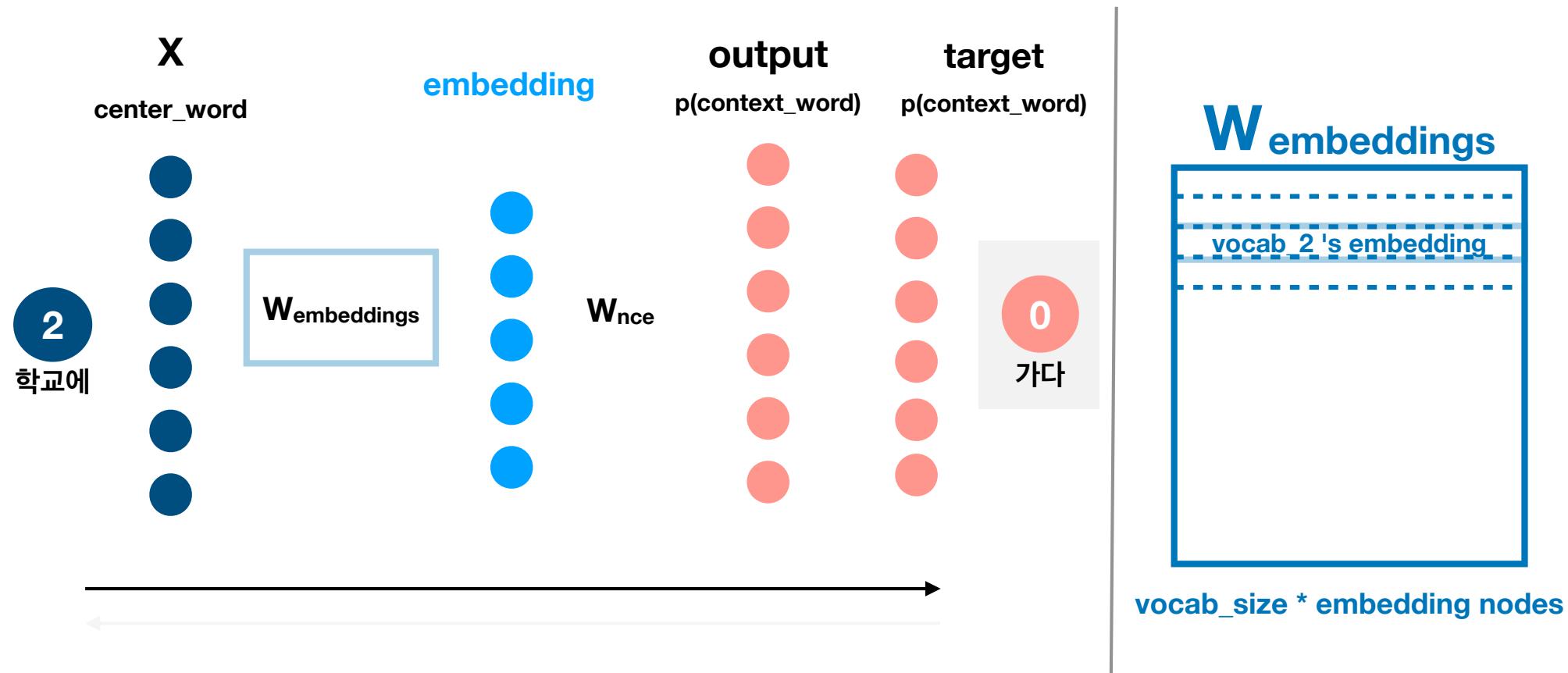
구체적 예시로 훈련을 해봅시다.

Word2Vec



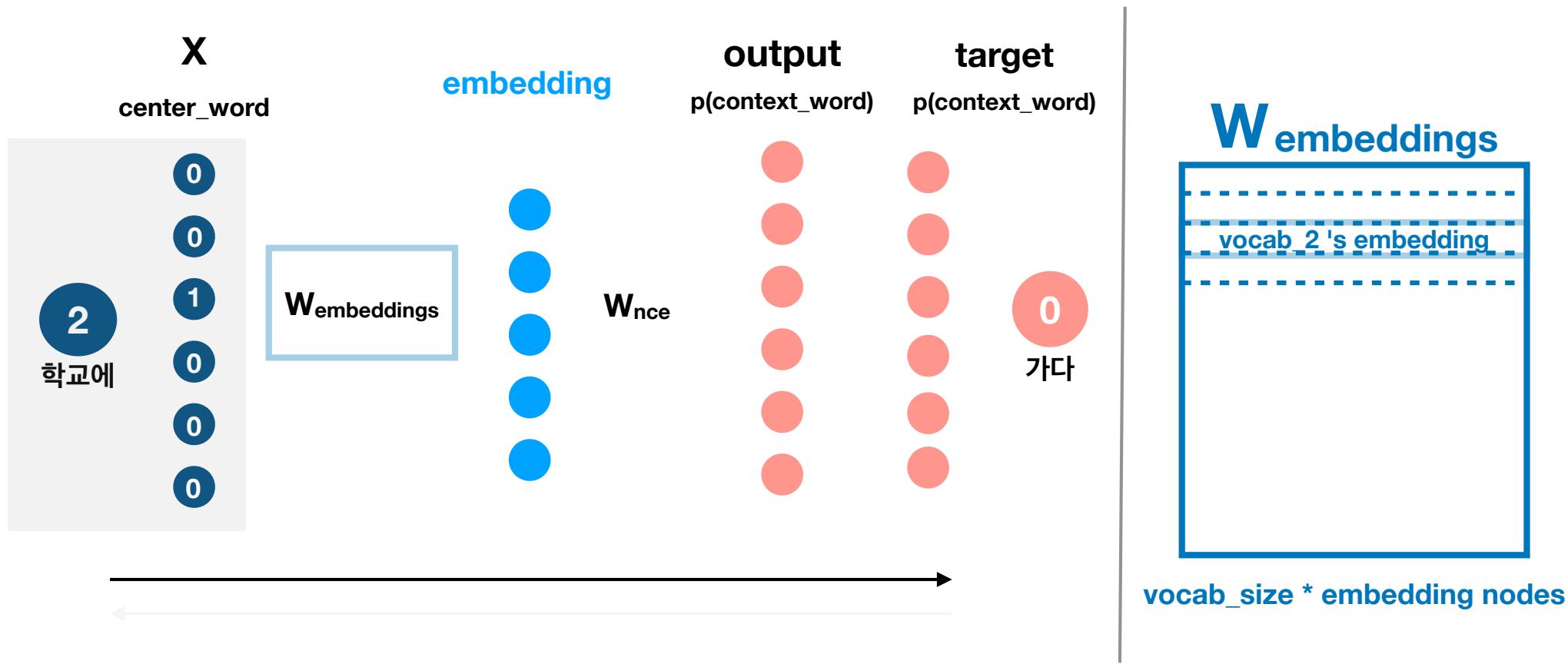
input에 '학교에'라는 center word가 들어가면

Word2Vec



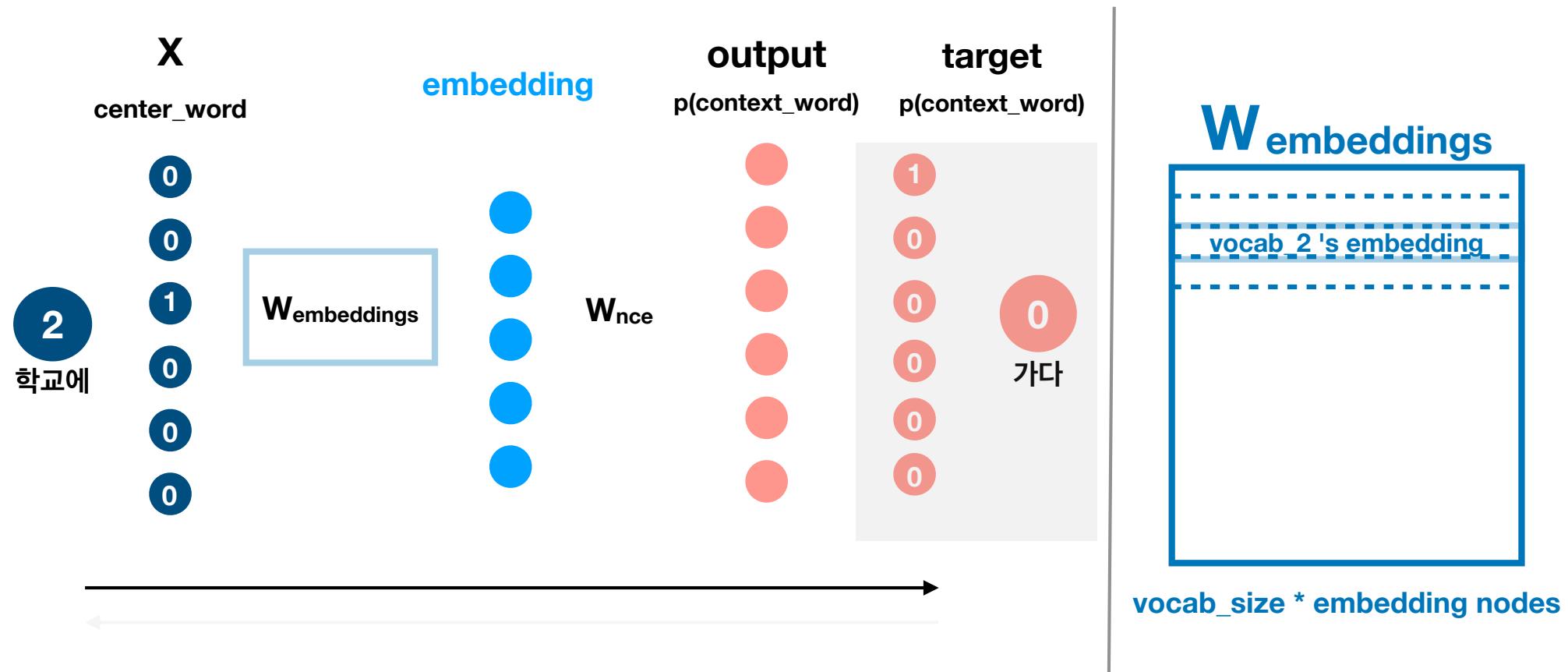
target은 '학교에'와 함께 쓰이는 context word '가다'가 됩니다.

Word2Vec

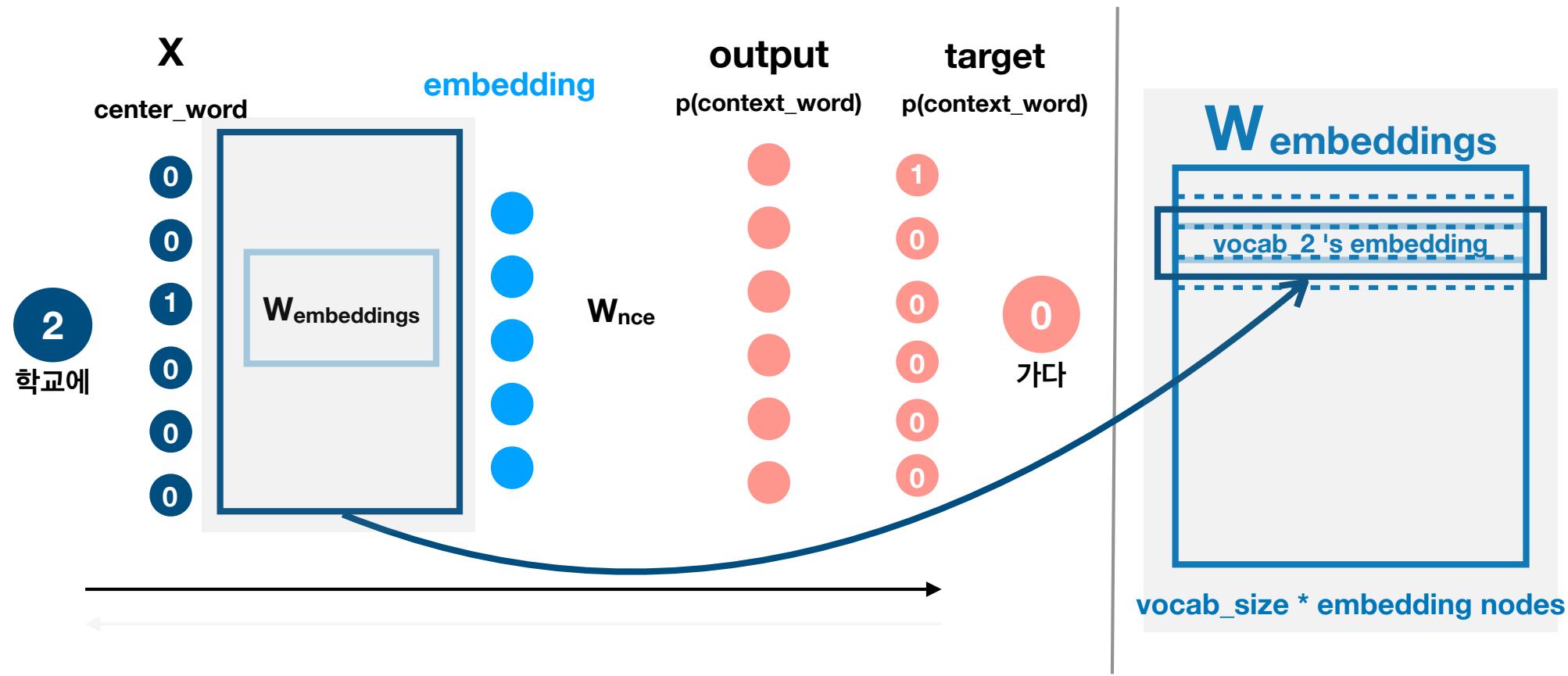


'학교에'라는 단어의 index가 2였다면,
input을 세 번째 칸의 값만 1이고 나머지는 0인 one-hot vector로 만들어줍니다.

Word2Vec

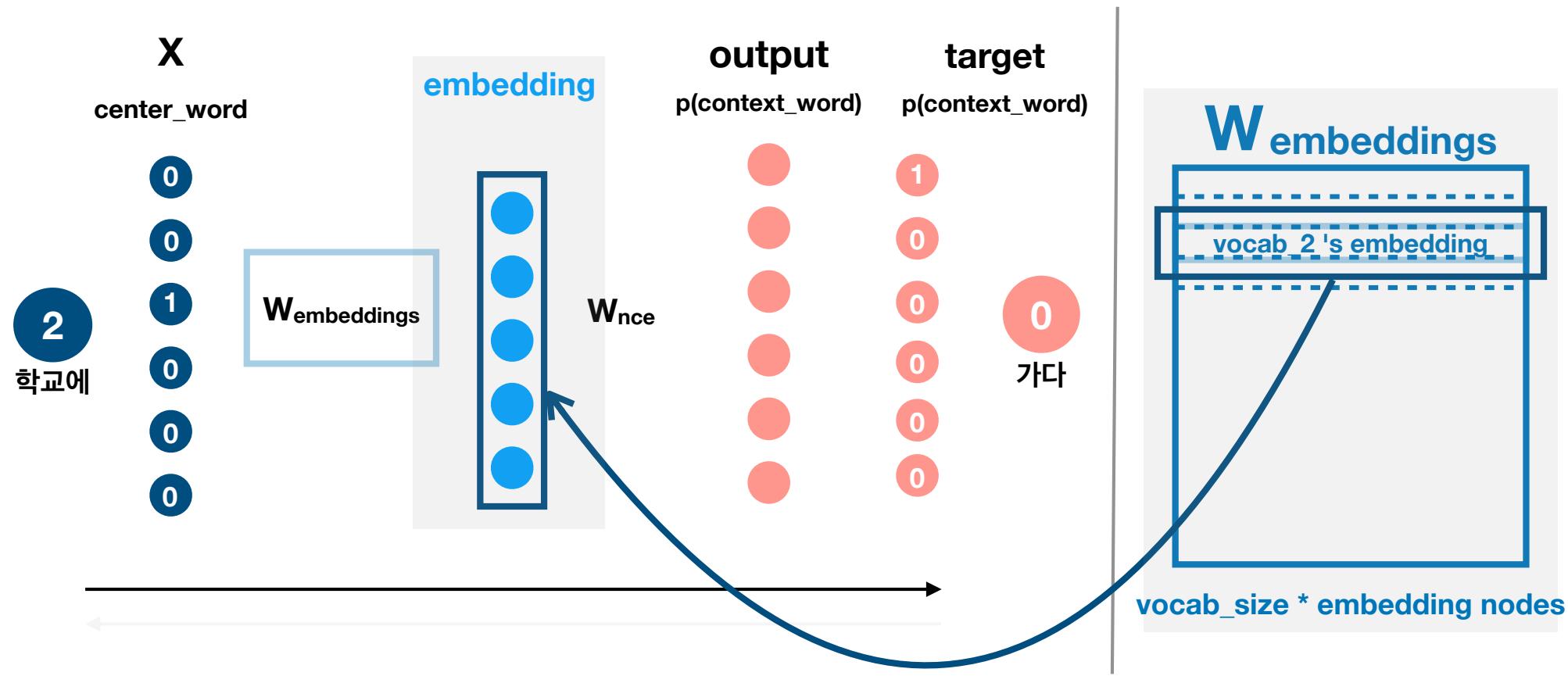


Word2Vec



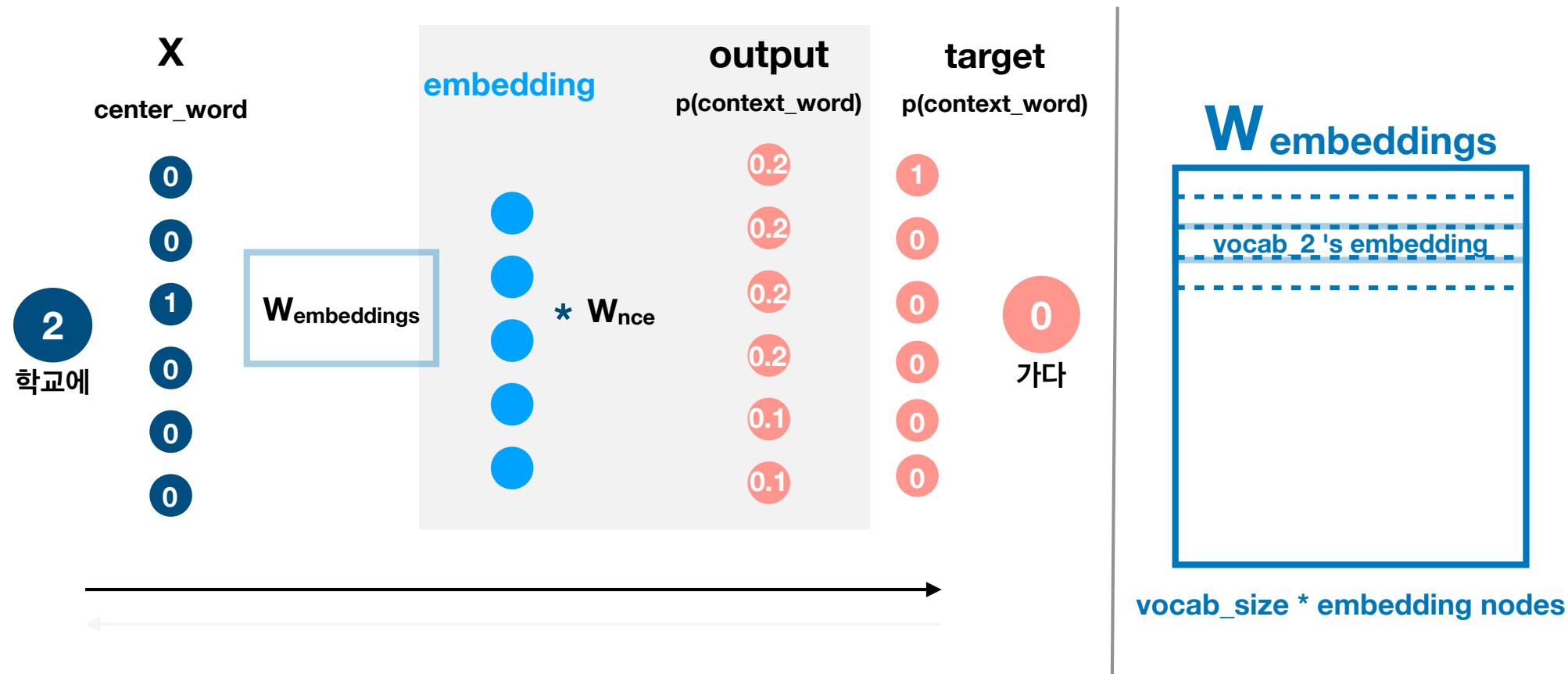
'학교에'라는 index에 해당하는 word vector를 $W_{\text{embeddings}}$ 란 표에서 찾습니다.

Word2Vec



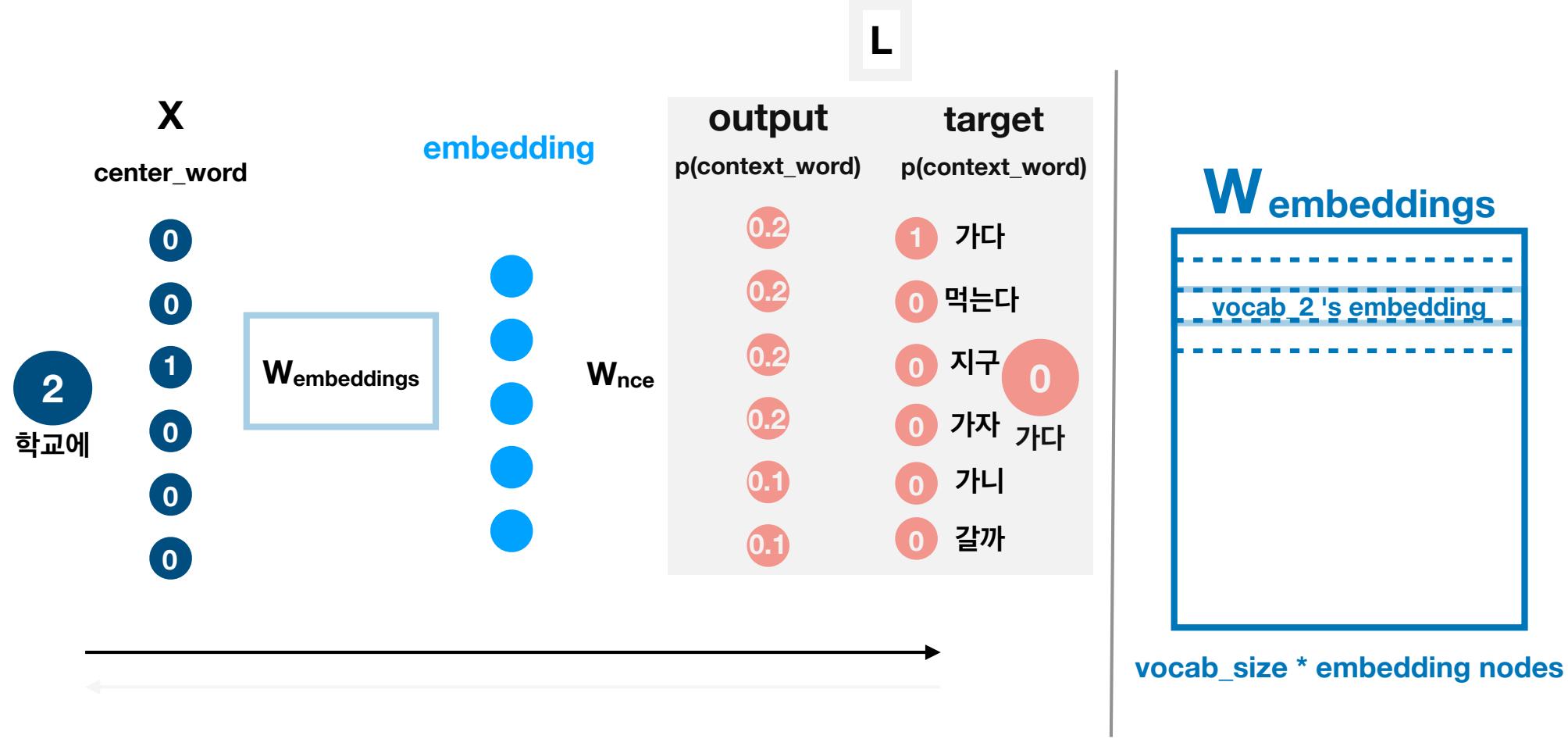
그 word vector가 embedding layer에 들어갑니다.

Word2Vec



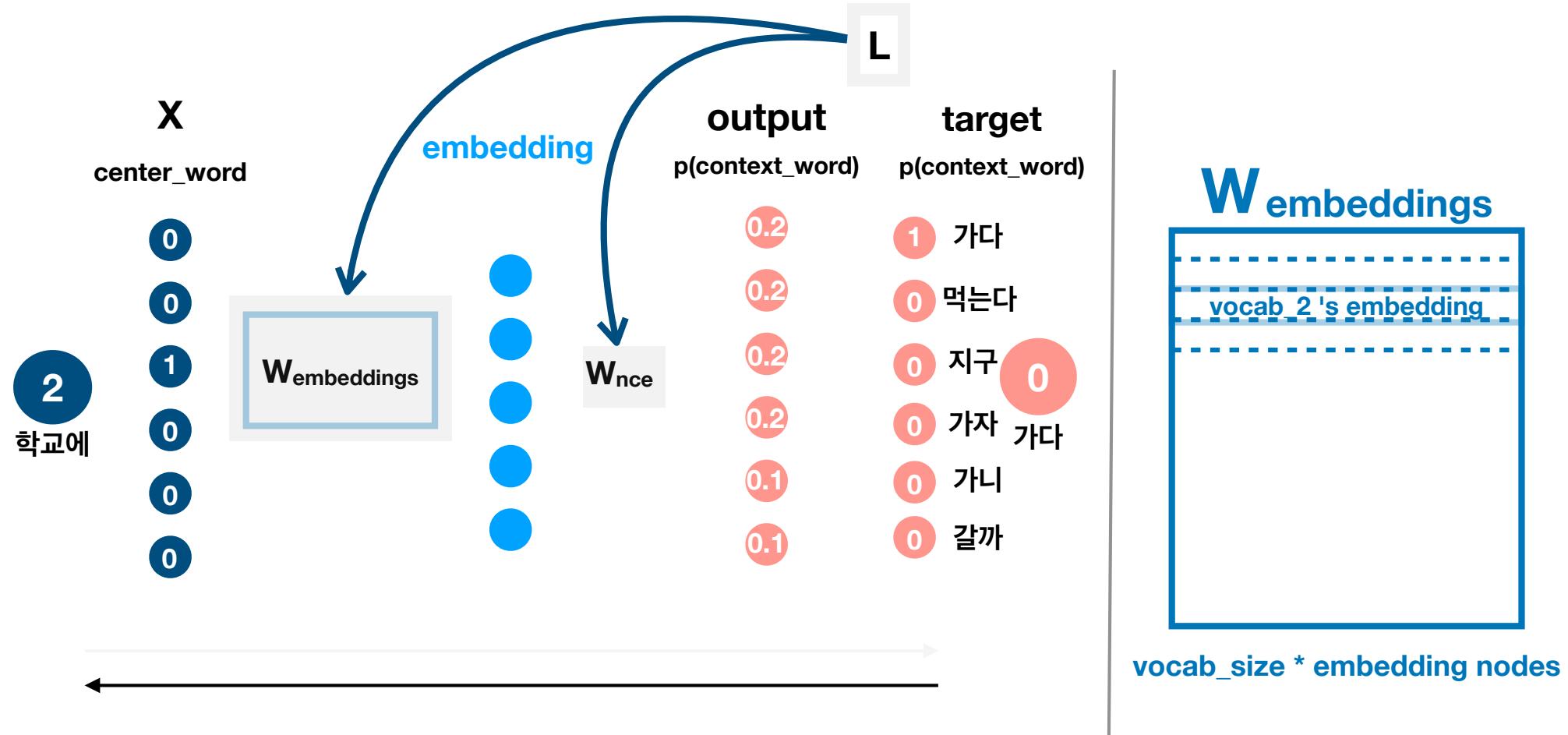
embedding layer의 값에 W_{nce} 를 곱하여 output 값을 구합니다.
output 값은 각 단어가 '학교에'의 context word일 확률입니다.

Word2Vec



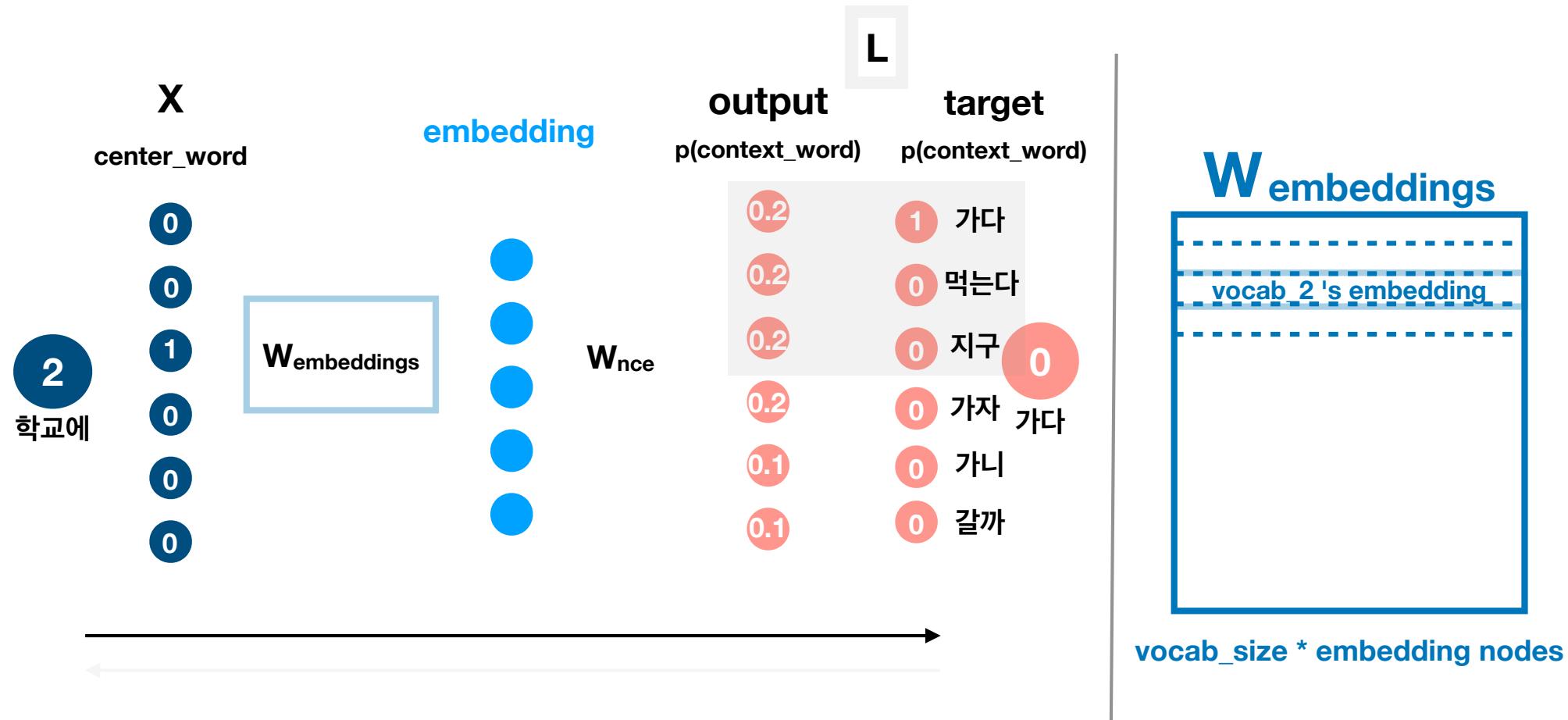
output과 정답 target (context_word)을 비교하여 loss를 구합니다.

Word2Vec



loss를 최소화하는 방향으로, weight을 update 합니다.

Word2Vec



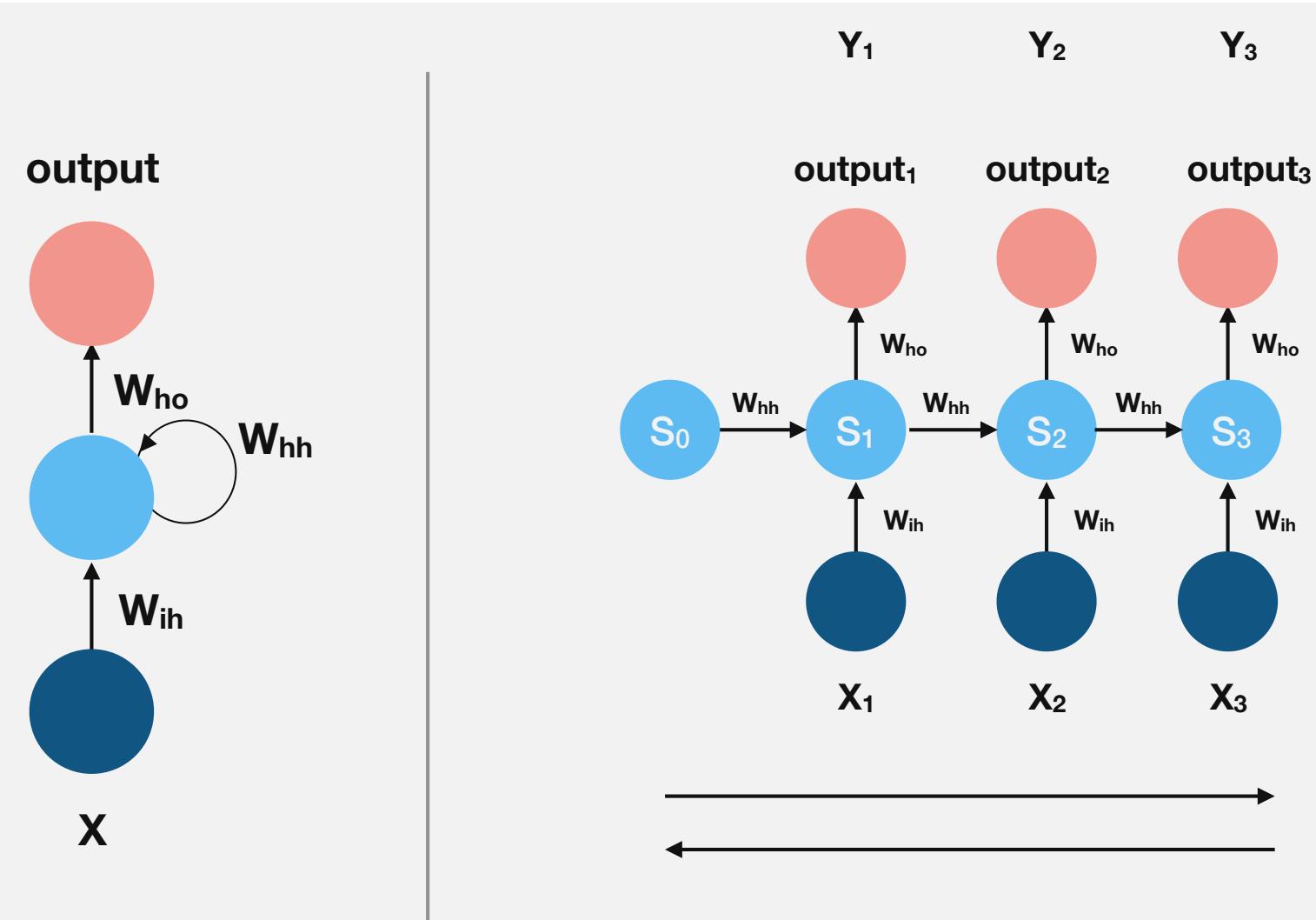
그러나, 훈련 대상이 되는 단어의 총 개수가 너무 많기 때문에,

실제로는 단어의 일부만 가지고 loss를 구합니다.

구체적으로는 정답인 '가다'와 정답과 거리가 먼 '먹는다', '지구' 등의 negative sample을 선별해 loss를 구하게 되는데 이런 방식을 negative sampling이라고 합니다.

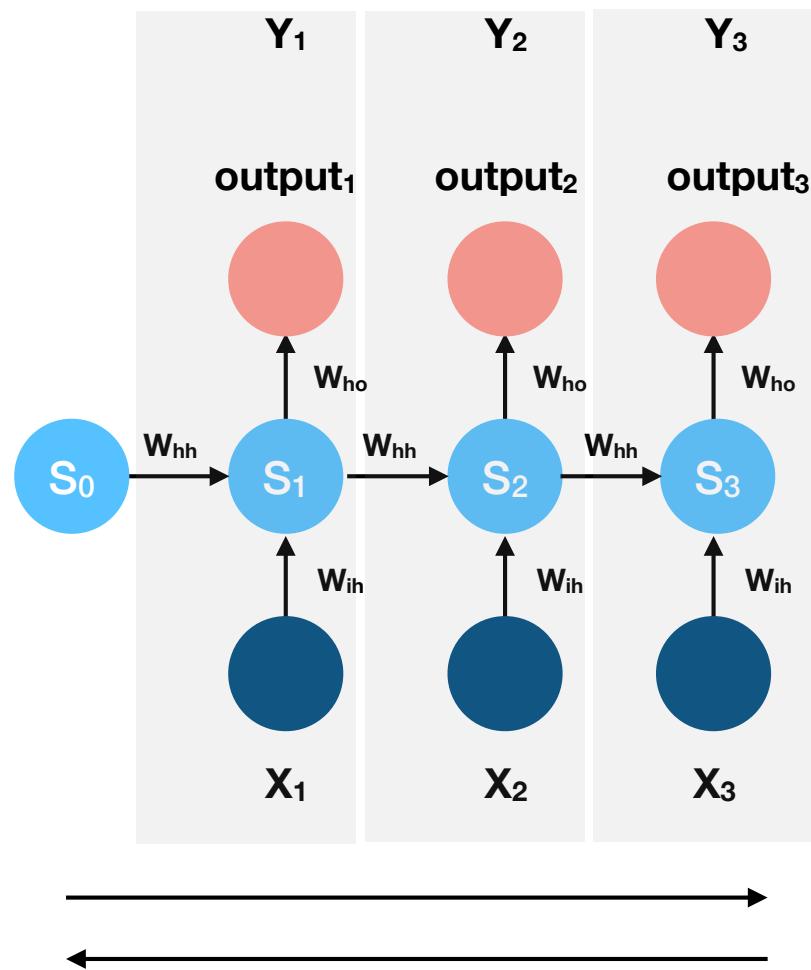
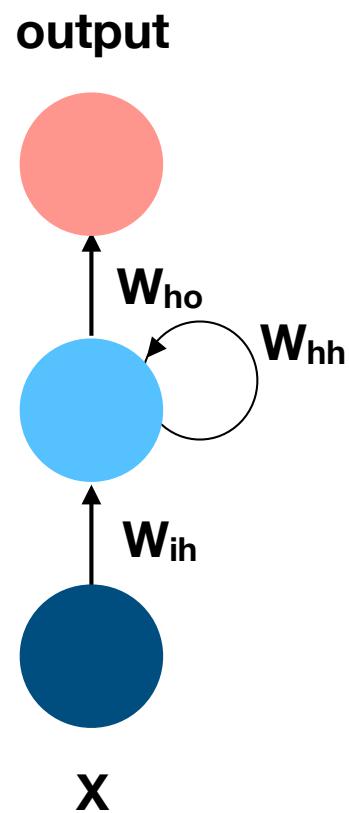
RNN

RNN



'나는 오늘 집에 간다'라는 문장을 단어 단위로 훈련해봅시다.

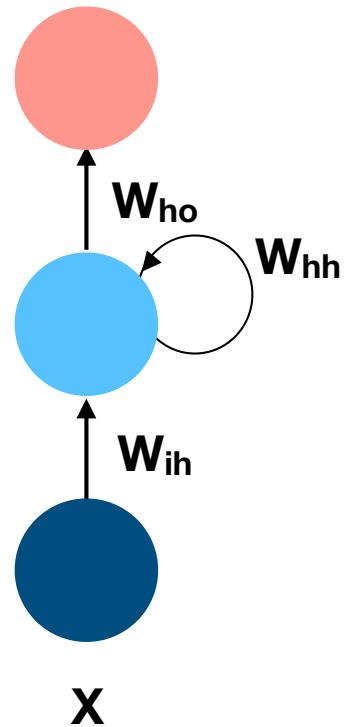
RNN



이 예시에서는 총 3개의 time step을 가지는 RNN을 훈련할 것입니다.

RNN

output



Y_1
오늘

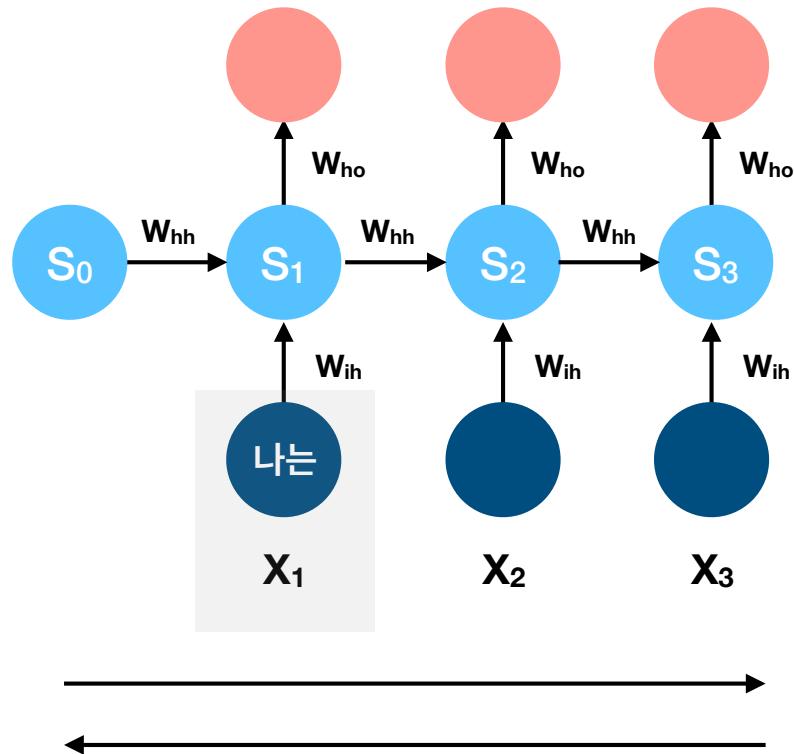
output₁

Y_2

output₂

Y_3

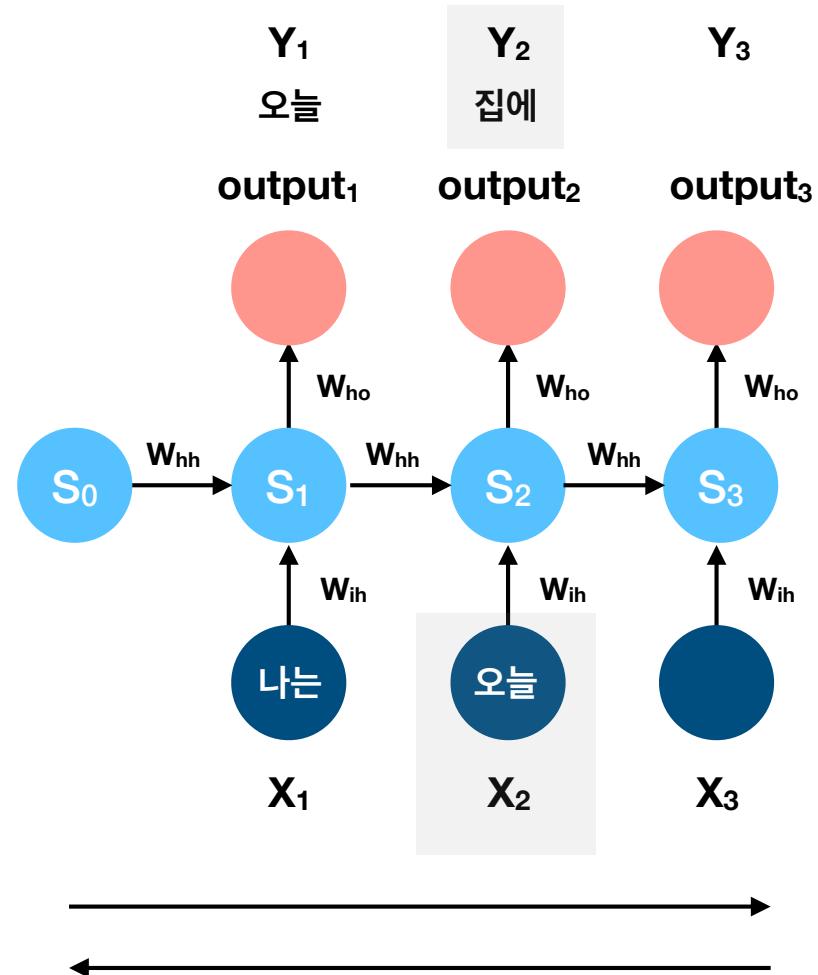
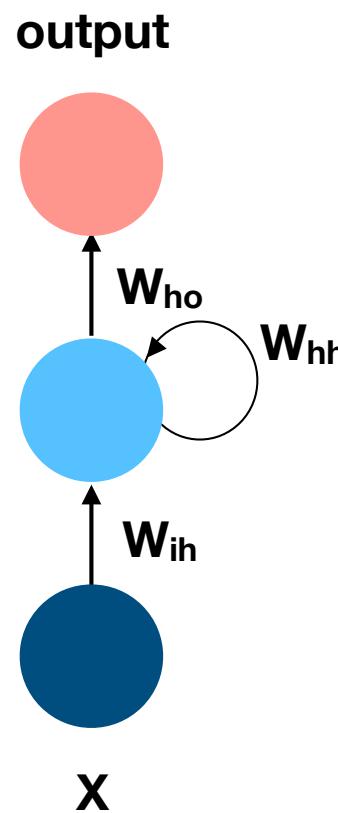
output₃



한 단어가 오면 그 다음 단어를 예측하도록 input과 target을 설정해줍니다.

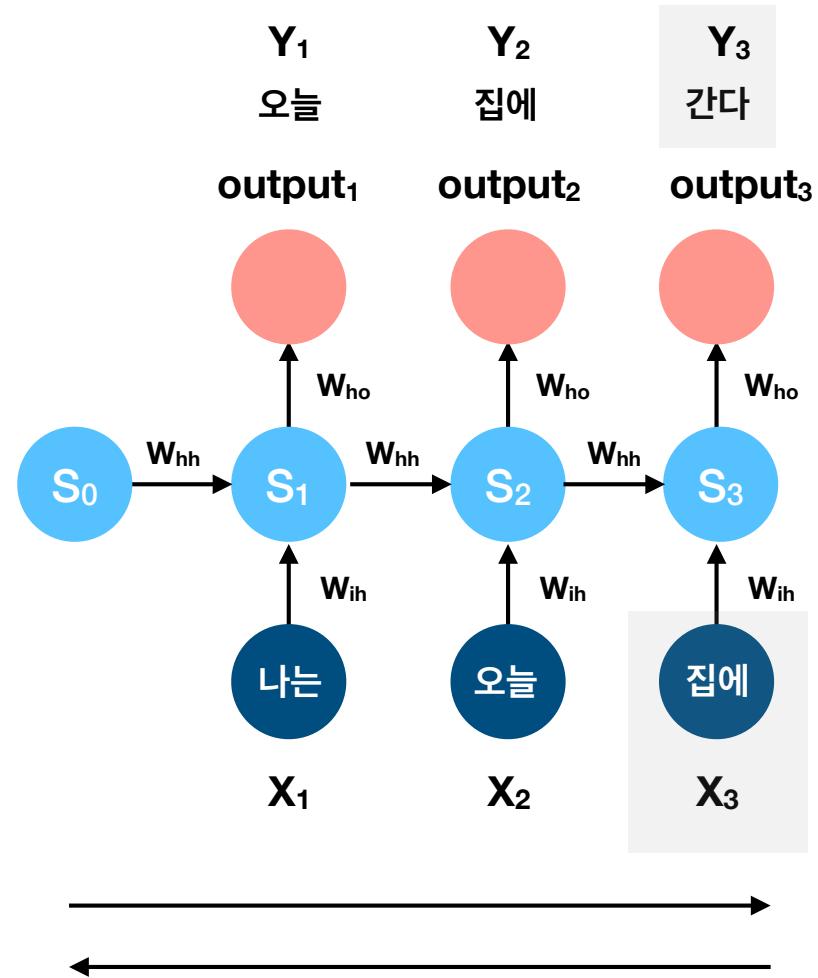
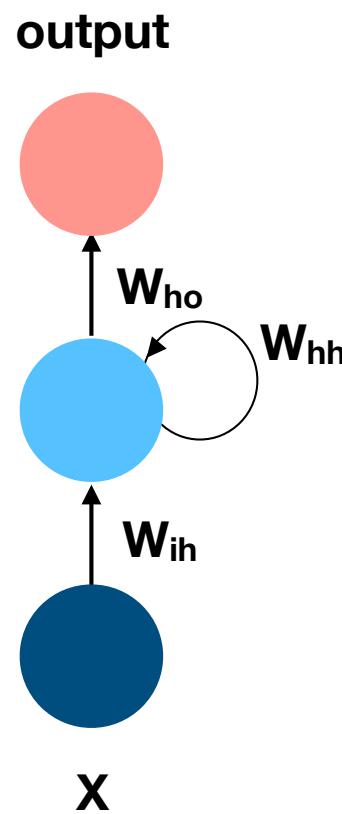
첫 time step에서는 '나는'이 input일 때 '오늘'이 target이 되도록

RNN



두 번째 time step에서는 '오늘'이 input일 때 '집에'가 target이 되도록

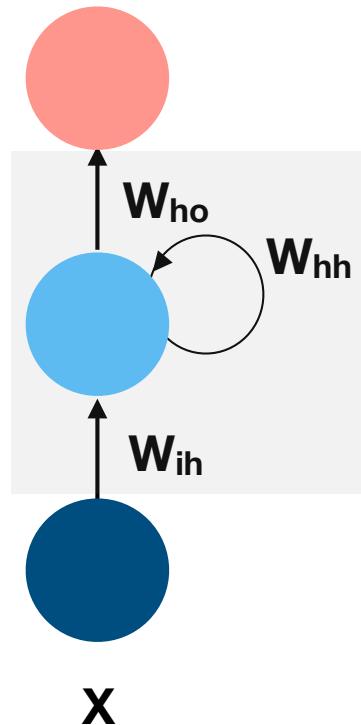
RNN



세 번째 time step에서는 '집에'가 input일 때 '간다'가 target이 되도록 RNN을 훈련해봅시다.

RNN

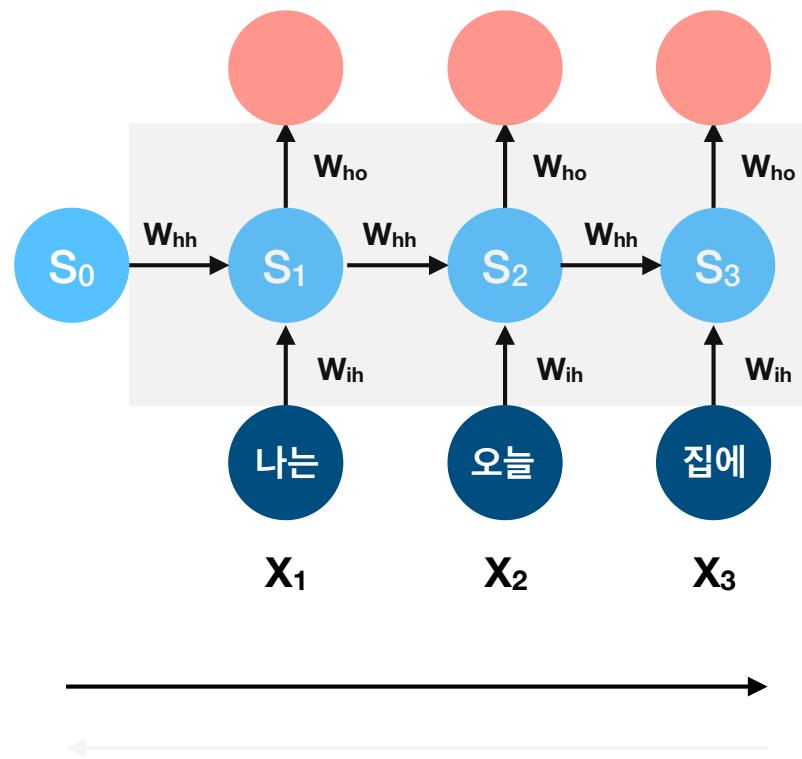
output



Y_1
오늘
 $output_1$

Y_2
집에
 $output_2$

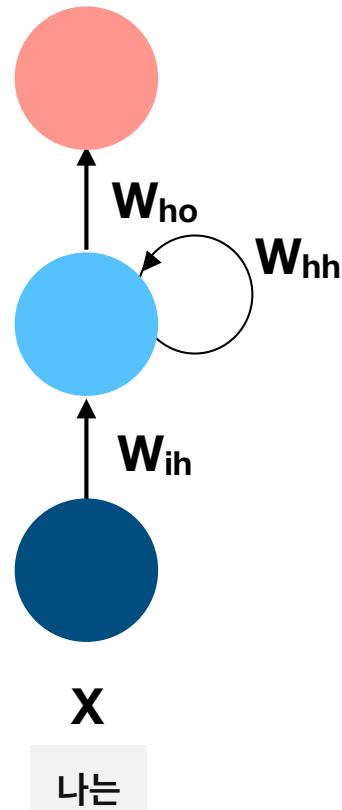
Y_3
간다
 $output_3$



각 time step에서의 W_{ih} , W_{hh} , W_{ho} 는 각각 동일한 weight입니다.

RNN

오늘
output



Y_1
오늘

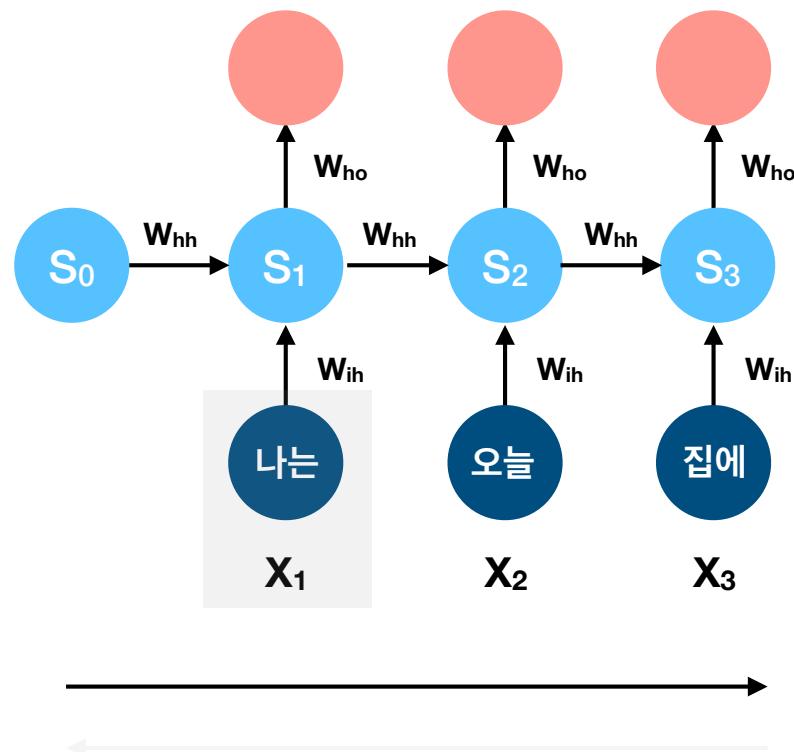
Y_2
집에

Y_3
간다

$output_1$

$output_2$

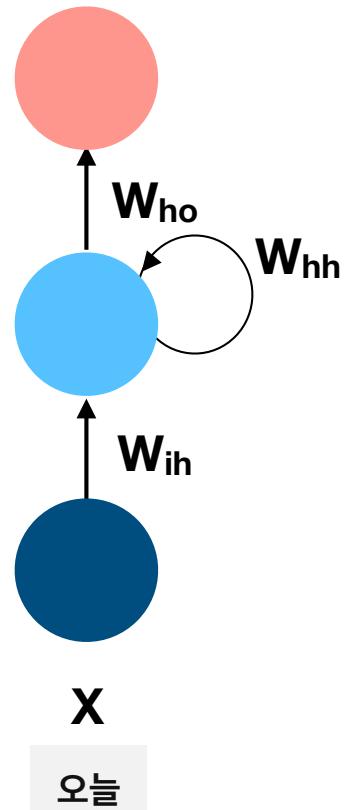
$output_3$



왼쪽 그림과 같은 ANN에 input X_1, X_2, X_3 가 순서대로 들어간다고 생각하면 됩니다.

RNN

집에
output

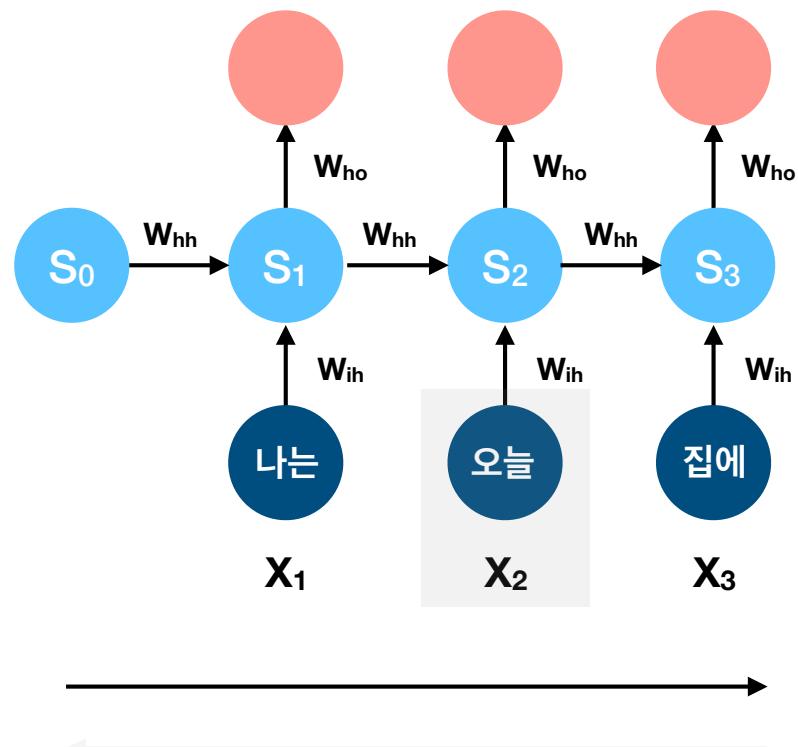


왼쪽 그림과 같은 ANN에 input X_1, X_2, X_3 가 순서대로 들어간다고 생각하면 됩니다.

Y_1
오늘
 $output_1$

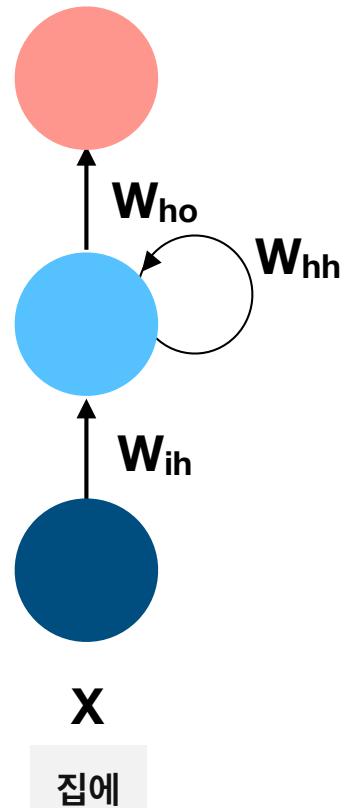
Y_2
집에
 $output_2$

Y_3
간다
 $output_3$

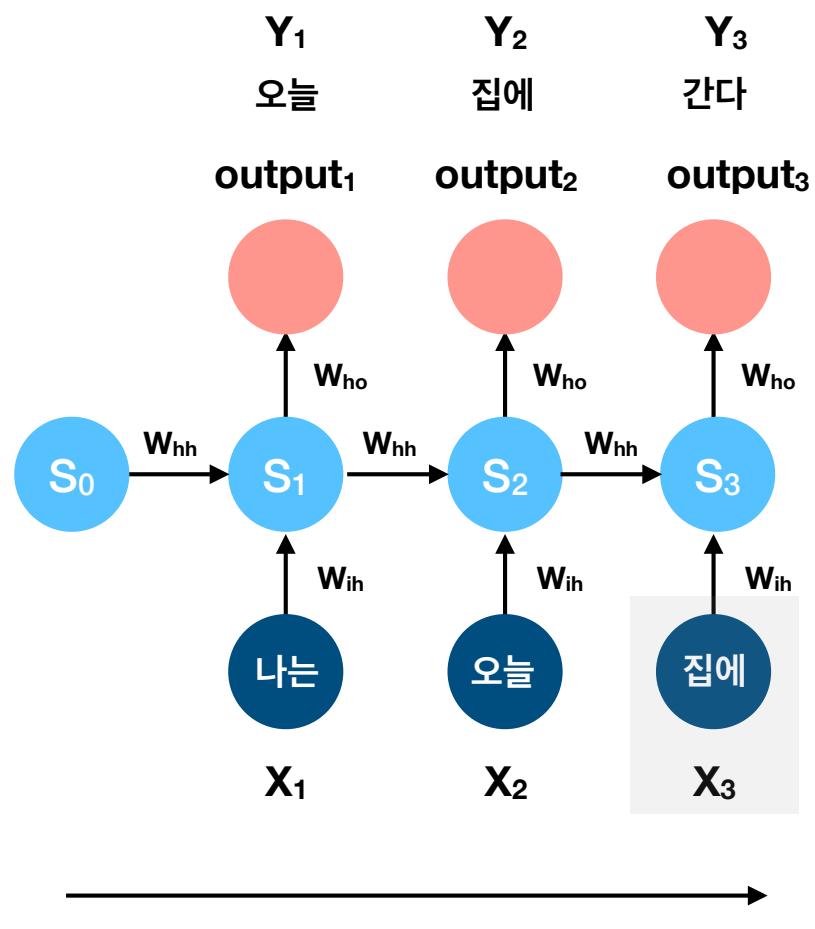


RNN

간다
output

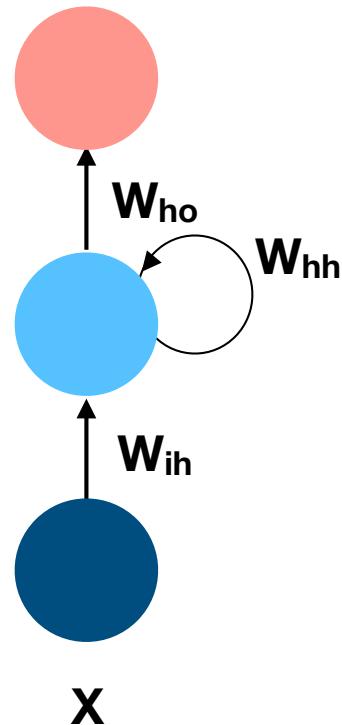


왼쪽 그림과 같은 ANN에 input X_1, X_2, X_3 가 순서대로 들어간다고 생각하면 됩니다.



RNN

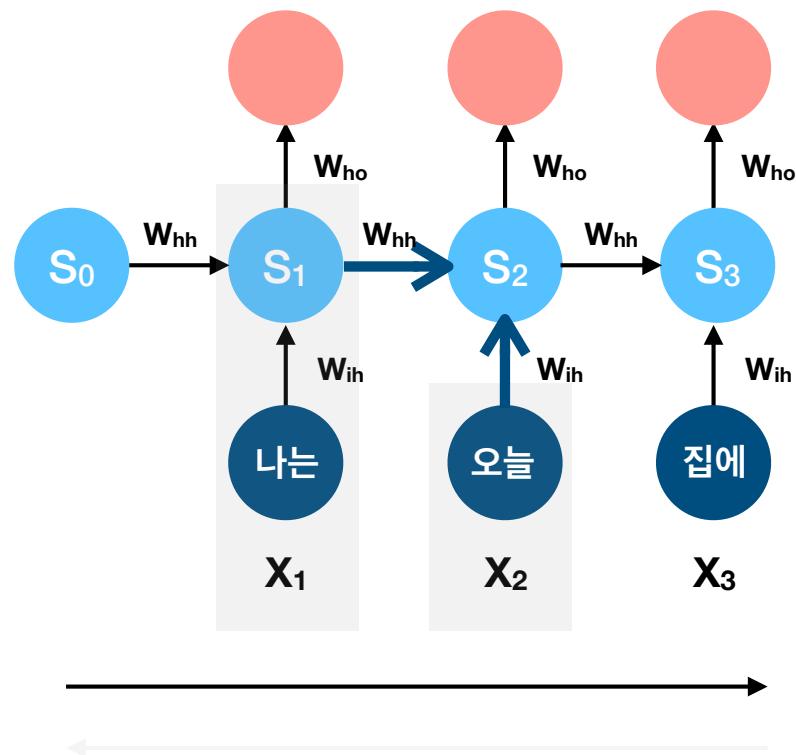
output



Y_1
오늘
 $output_1$

Y_2
집에
 $output_2$

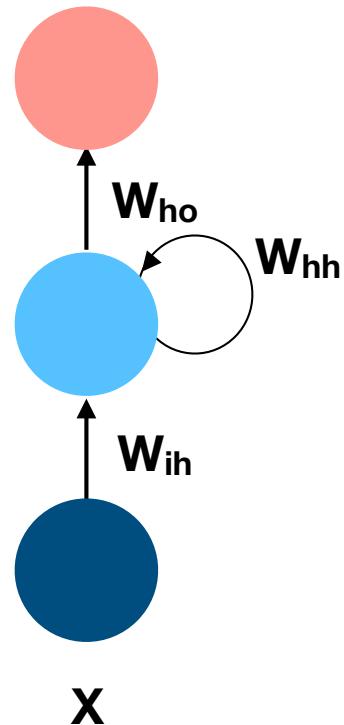
Y_3
간다
 $output_3$



하지만 ANN과는 다르게 hidden layer는 현재 time step의 input 뿐만 아니라
이전 time step의 hidden layer의 영향도 받습니다. ($S(t) = S(t-1) * W_{hh} + X(t) * W_{ih}$)

RNN

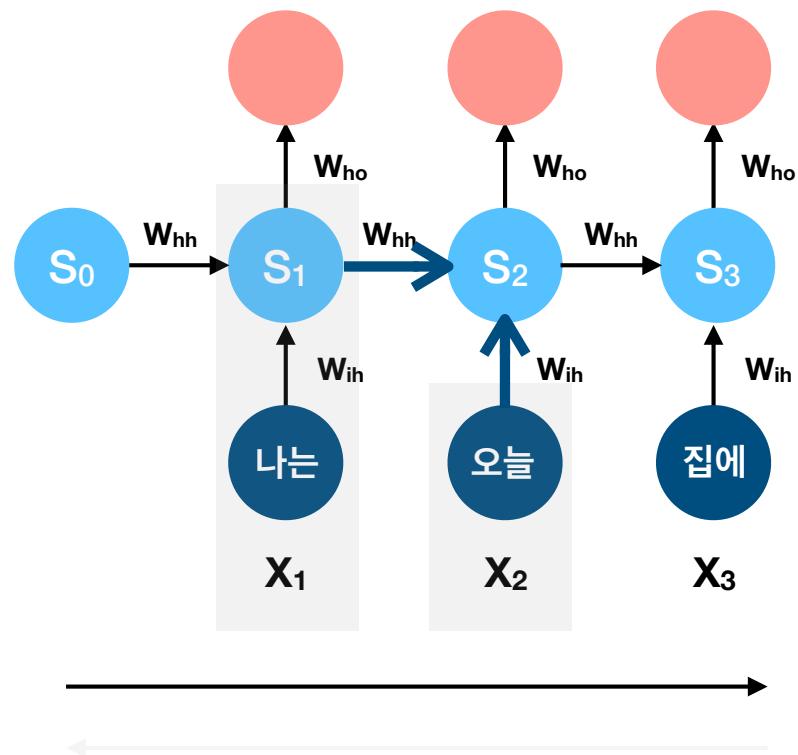
output



Y_1
오늘
 $output_1$

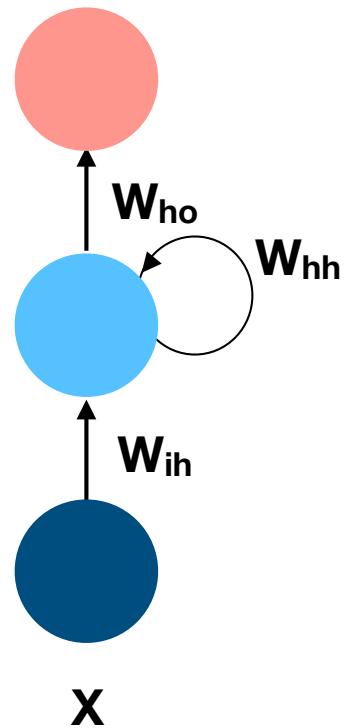
Y_2
집에
 $output_2$

Y_3
간다
 $output_3$



즉 '오늘'이란 단어가 두 번째 input (X_2)으로 들어갔을 때 이전 input (X_1)인 '나는'이 영향을 줍니다.

output

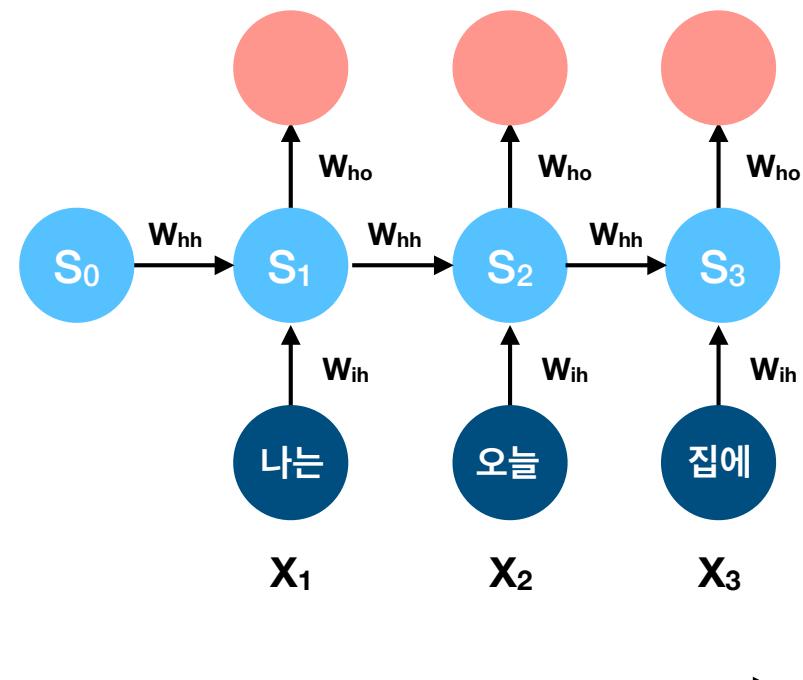


RNN $L_1 + L_2 + L_3 = \text{total loss}$

Y_1
오늘
 output_1

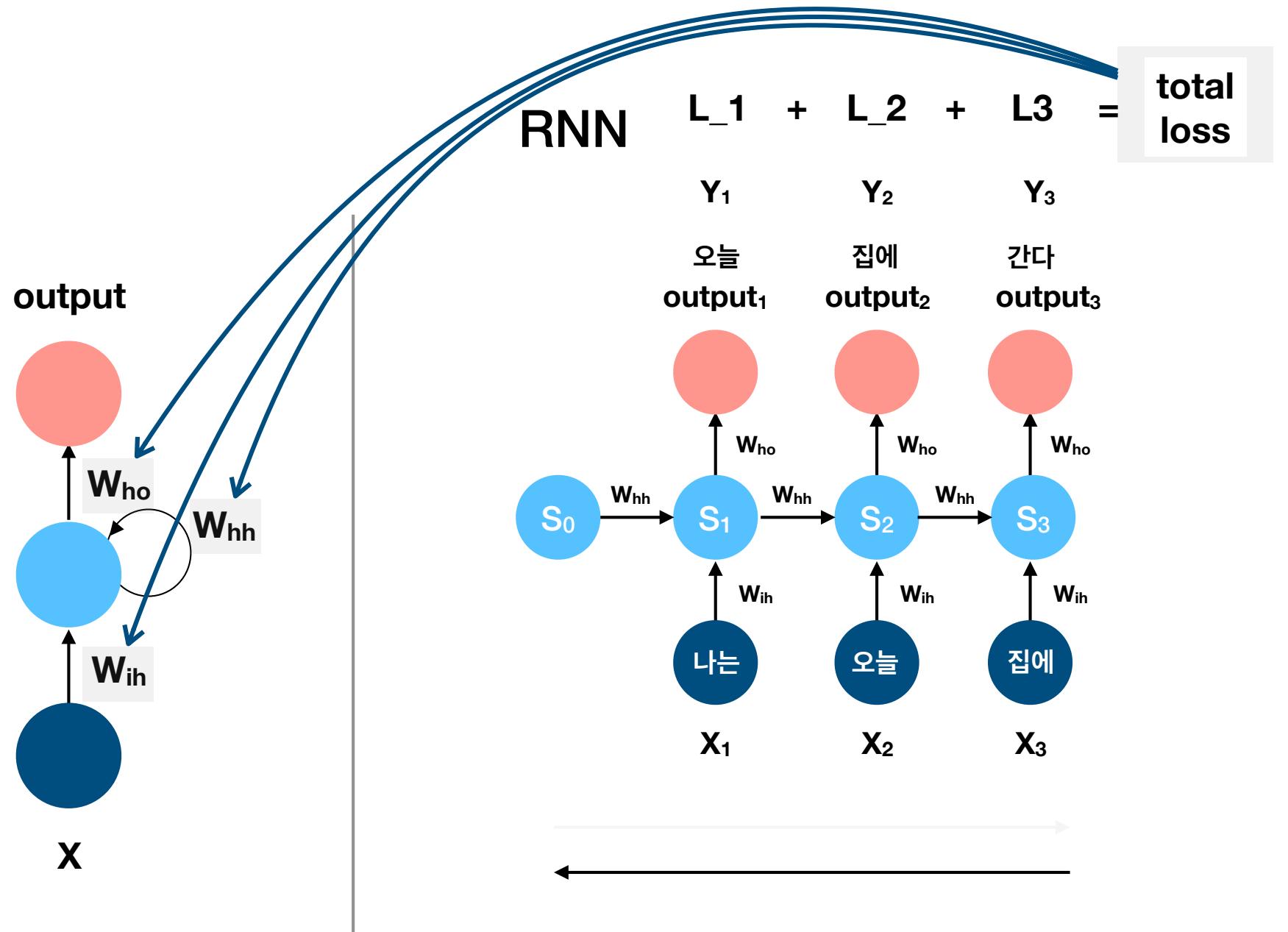
Y_2
집에
 output_2

Y_3
간다
 output_3



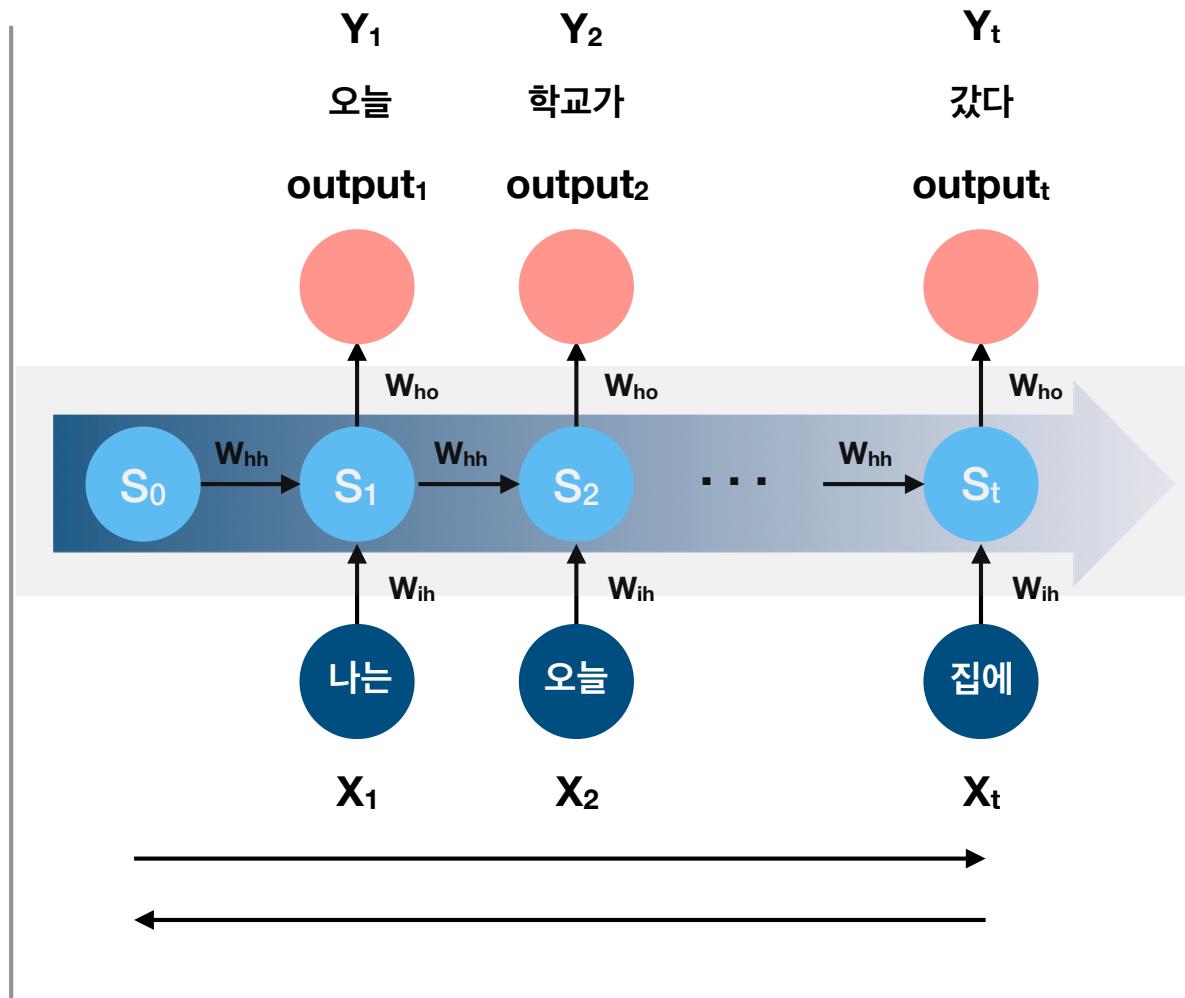
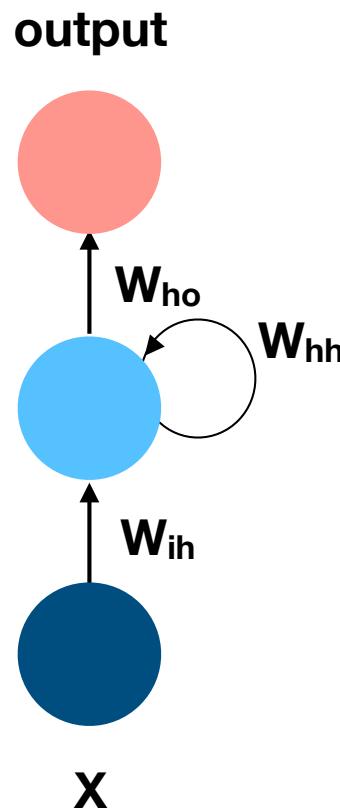
output과 정답 target (Y)을 비교하여 loss를 구합니다.

그러면 각각의 time step에서 loss가 나오게 될 것이고 이를 모두 더하여 전체 loss라고 정의합니다.



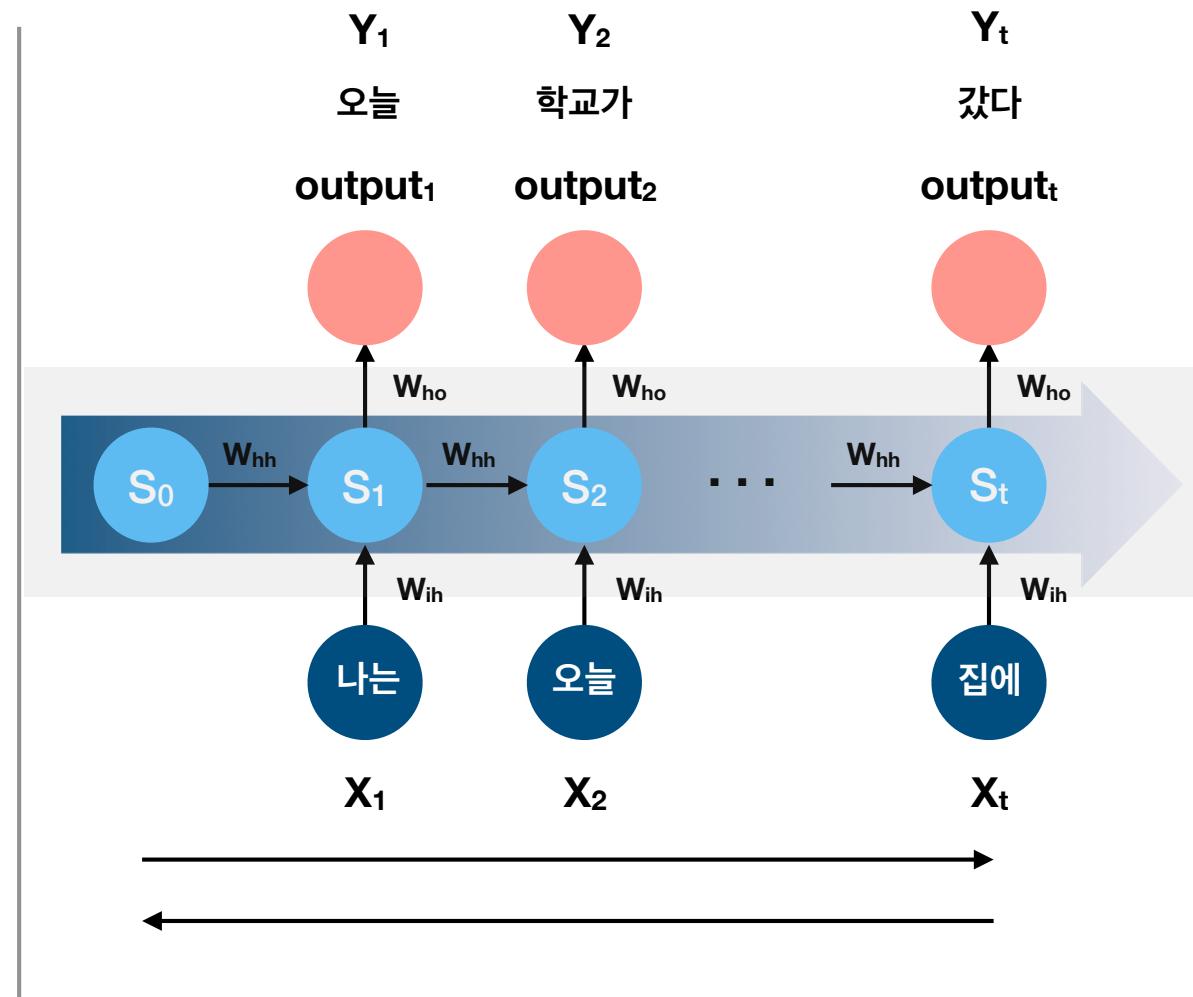
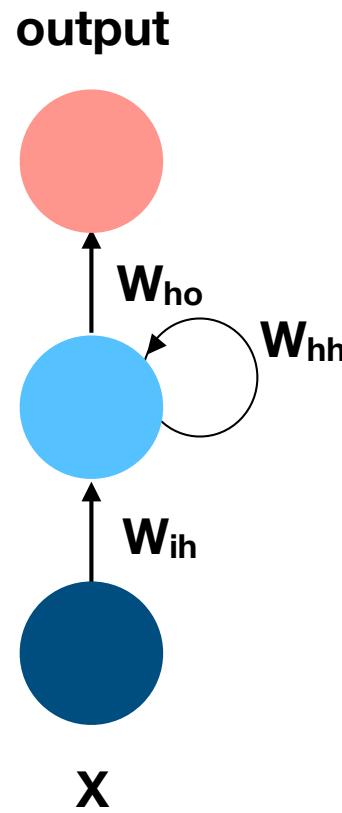
이 전체 loss를 최소화하는 방향으로, weight를 update 합니다.

RNN



가장 기본적인 RNN은 훈련할 time step의 수가 많아졌을 때,
뒤쪽의 time step이 앞쪽 time step의 정보를 제대로 반영하지 못하는 한계가 있습니다.

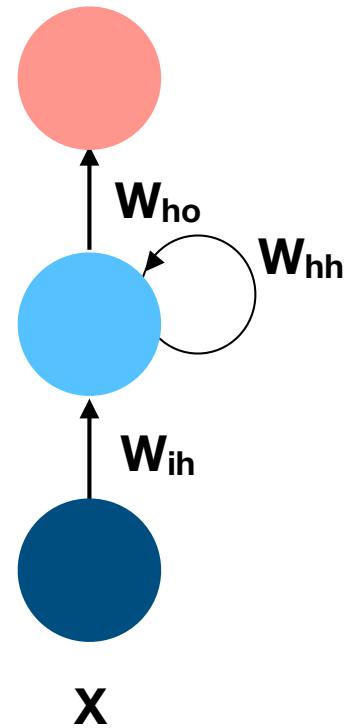
RNN



예를 들어, '나는 오늘 학교가 끝나고 서점에 들러 책을 산 후 집에 갔다'라는 11개의 time step으로 이루어진 문장을 훈련할 때 마지막 11번째 time step의 hidden node S_{11} 은 첫 번째 time step의 input인 '나는'에 대한 정보를 거의 갖고 있지 않습니다.

RNN

output

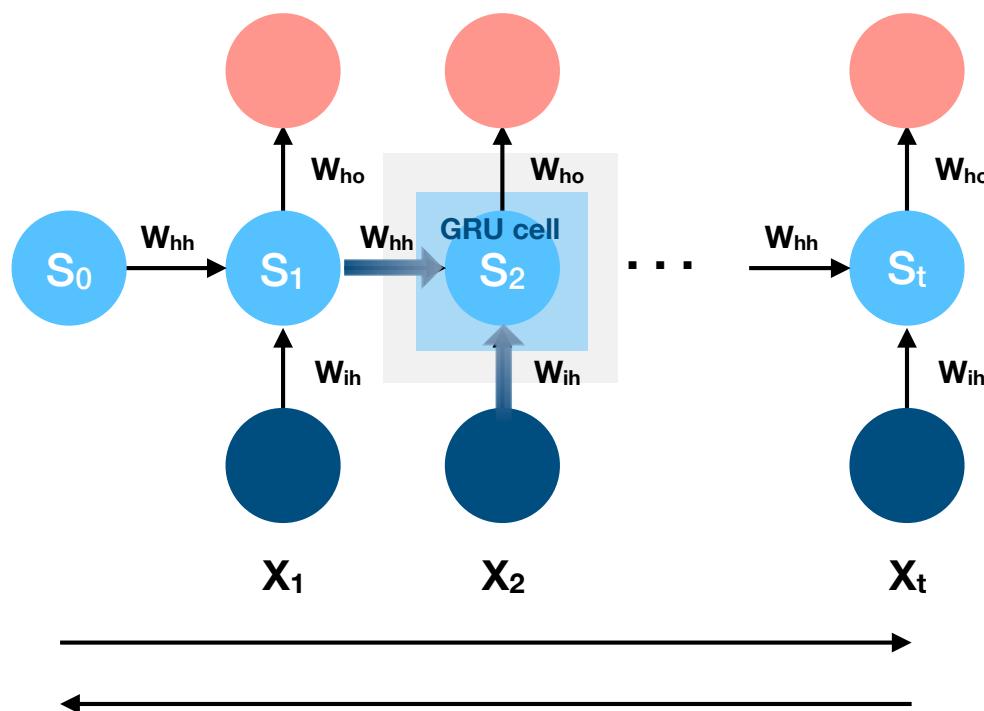


Y_1
오늘
 Y_2
학교가
 Y_t
갔다

output₁

output₂

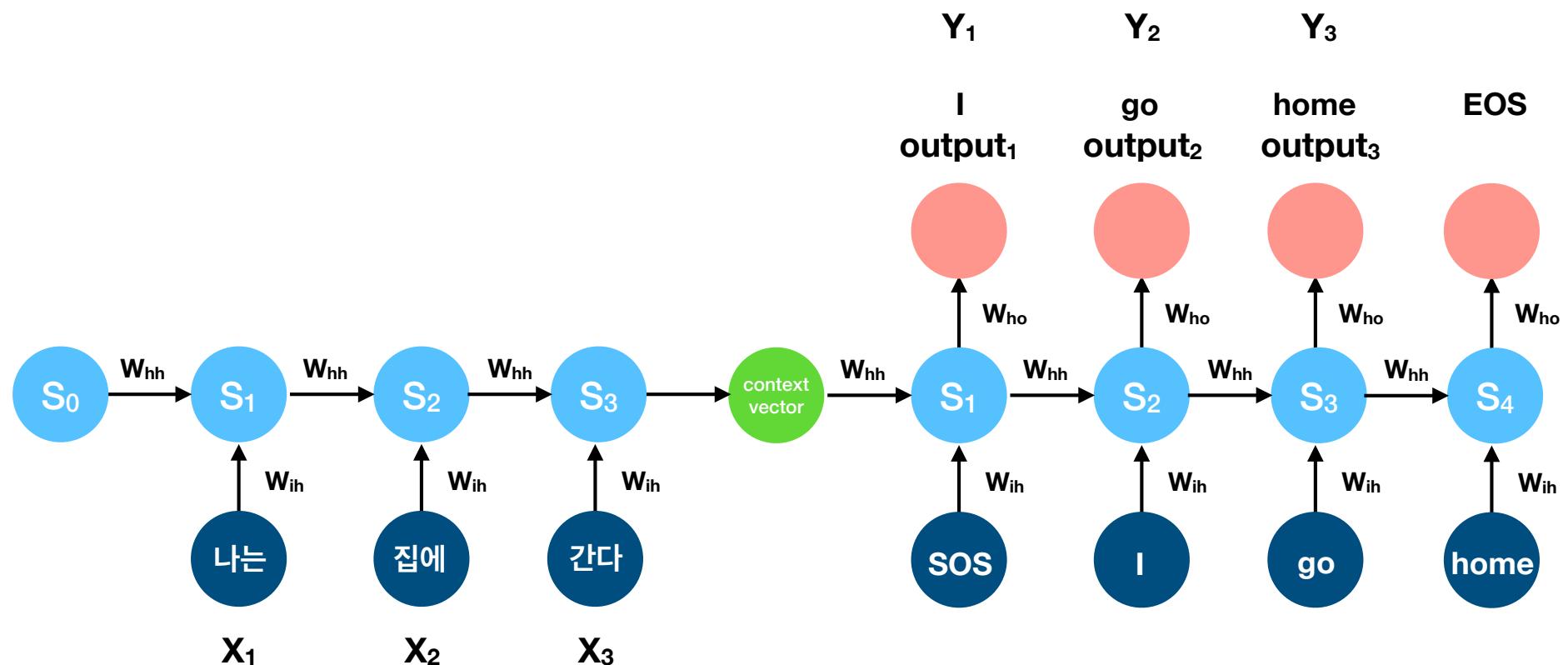
output_t



따라서 hidden layer가 이전 time step의 정보를 보다 효율적으로 기억할 수 있도록 하는 방법이 고안되었습니다.

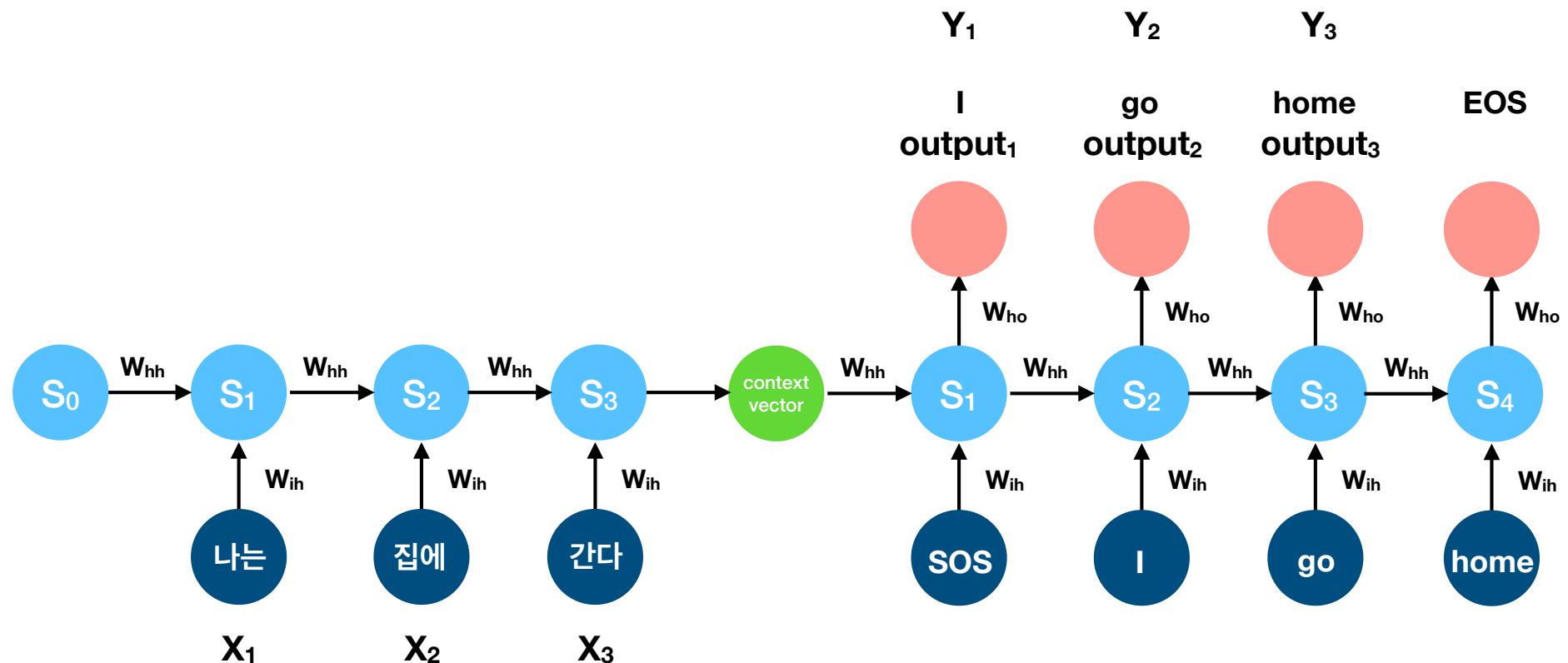
SEQ2SEQ

SEQ2SEQ

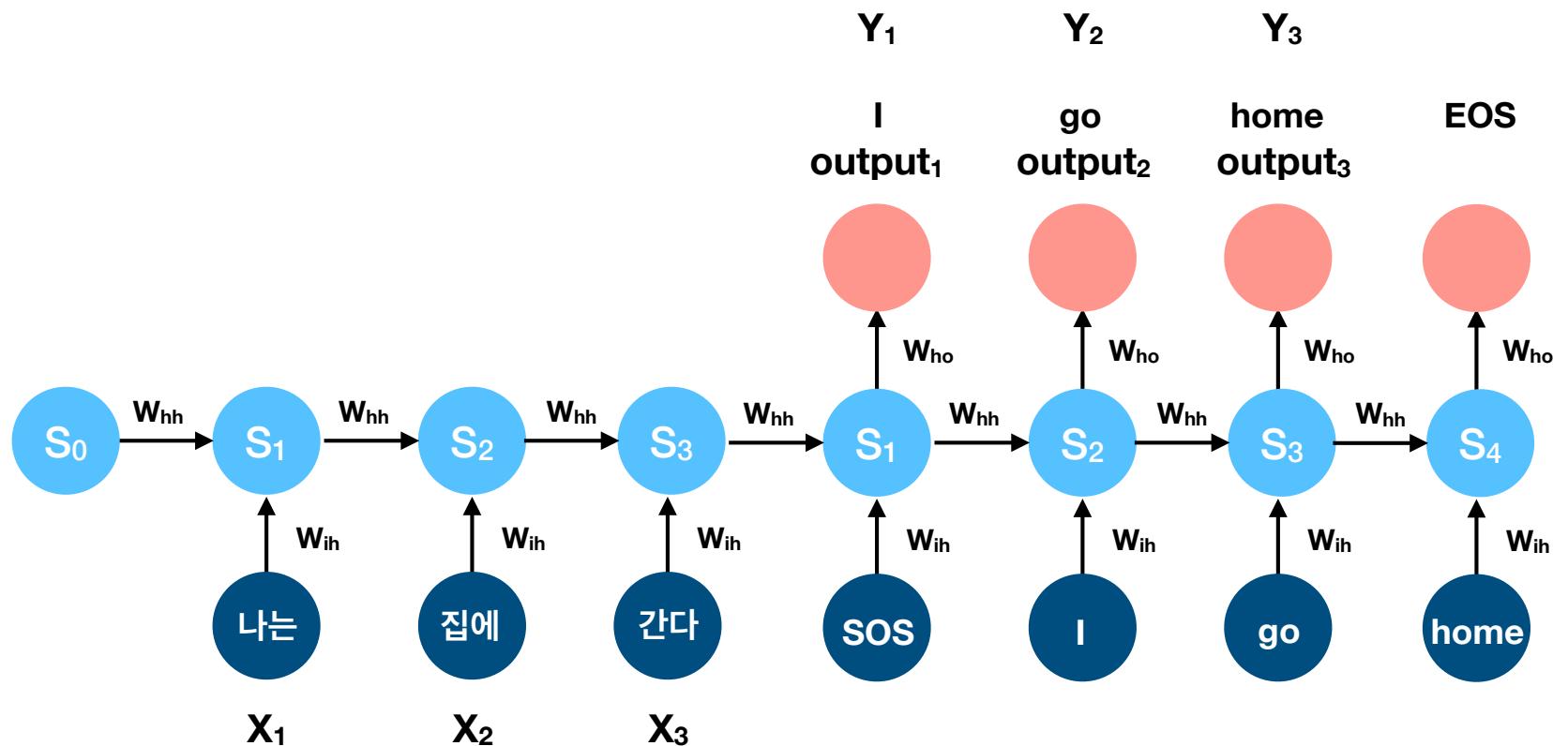


Transformer

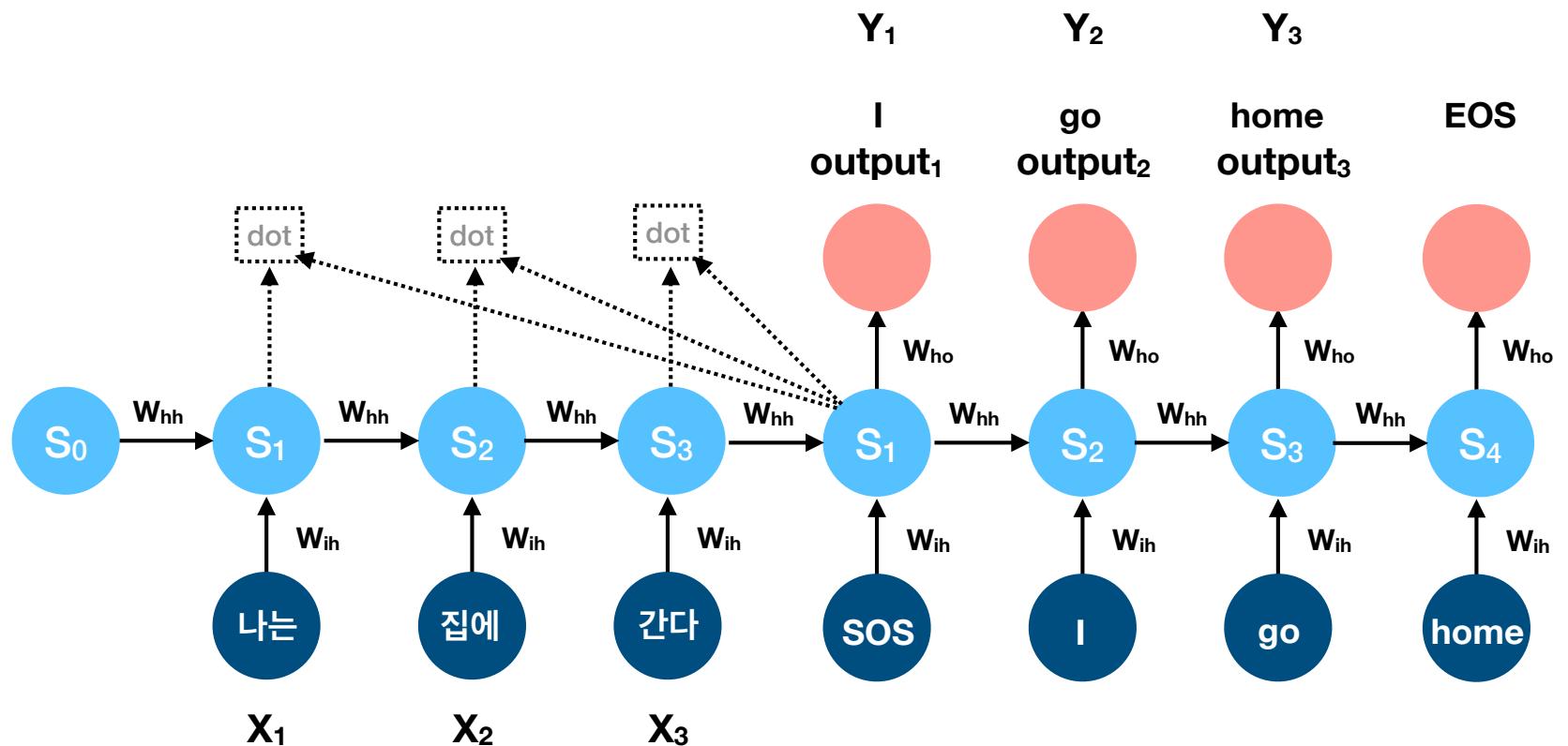
SEQ2SEQ



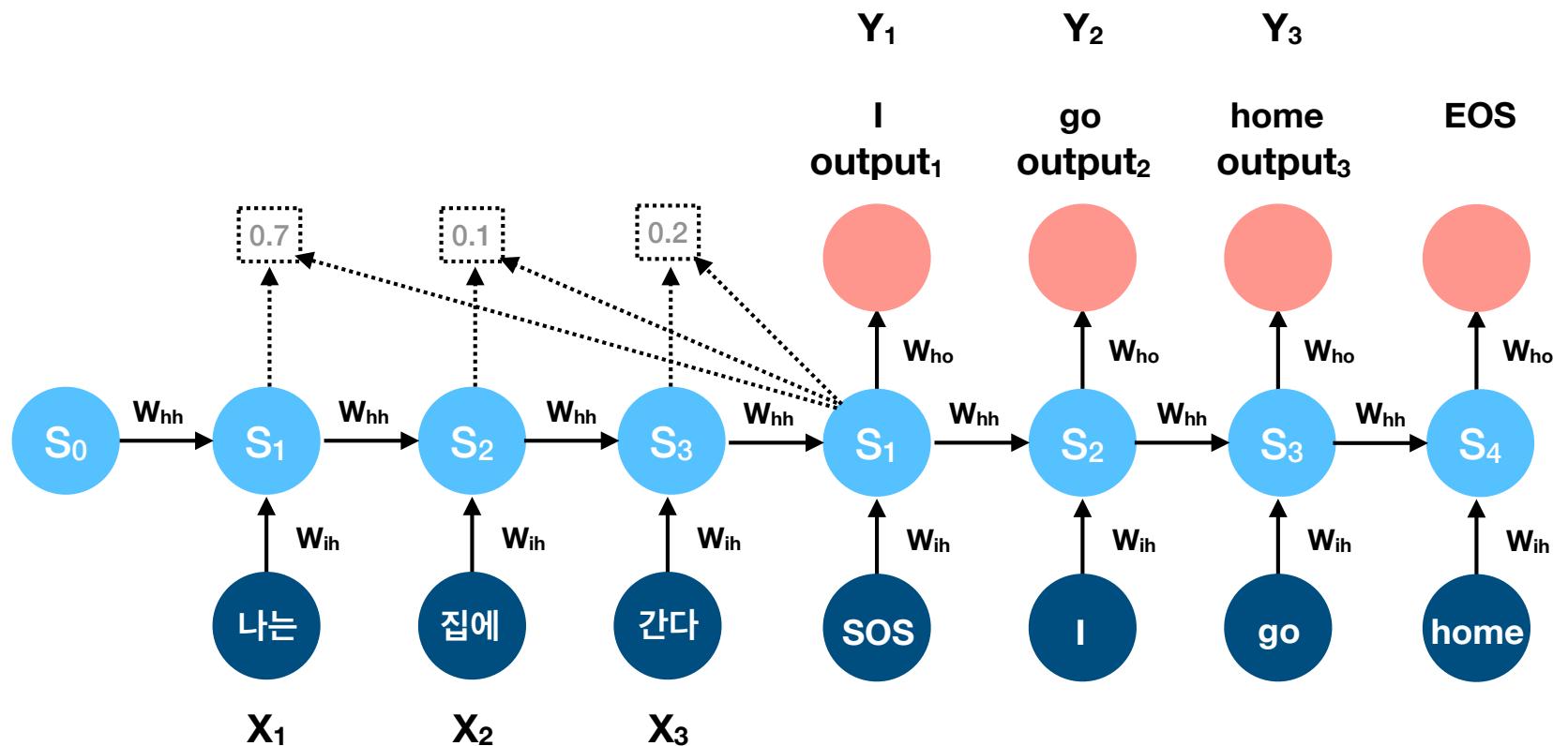
Transformer



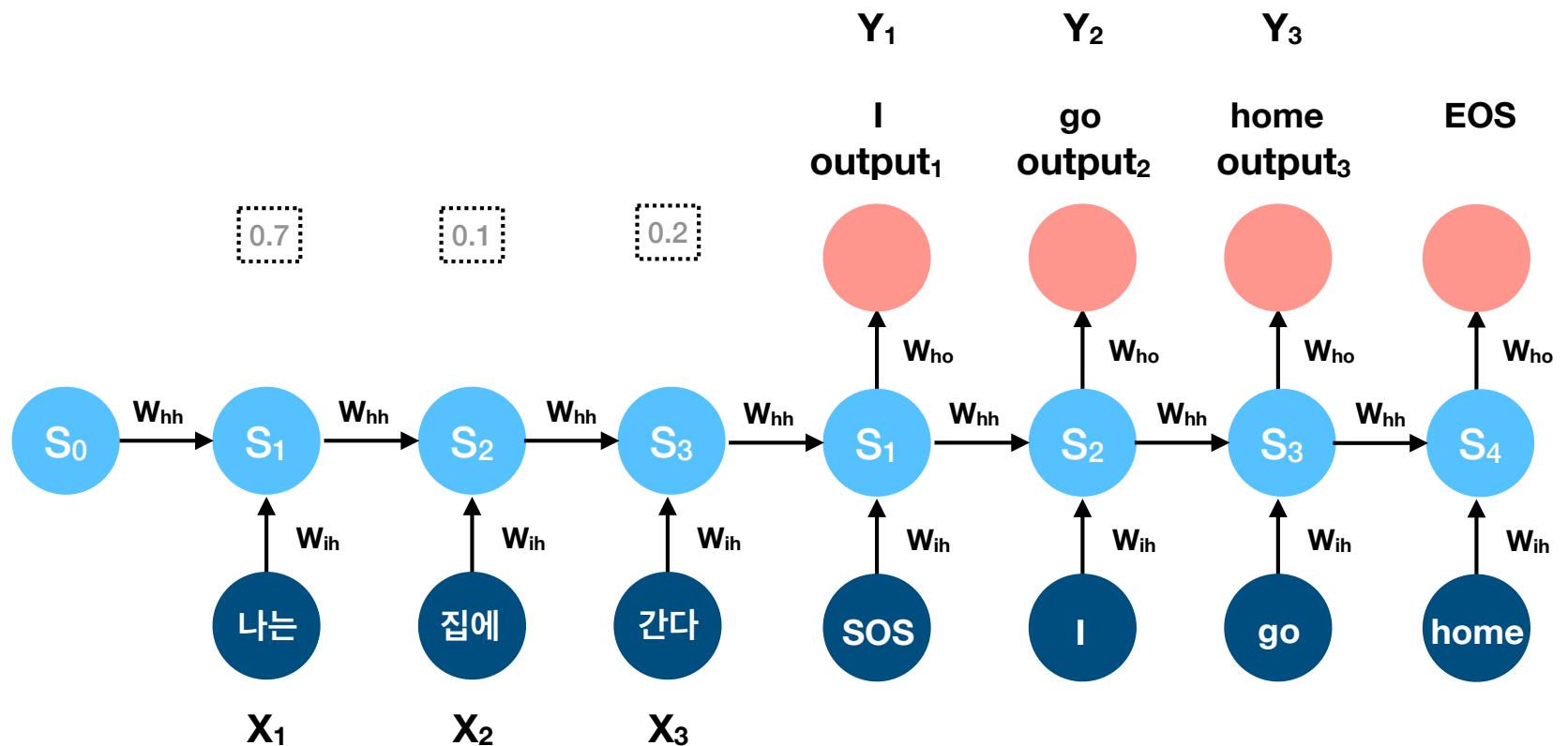
Transformer



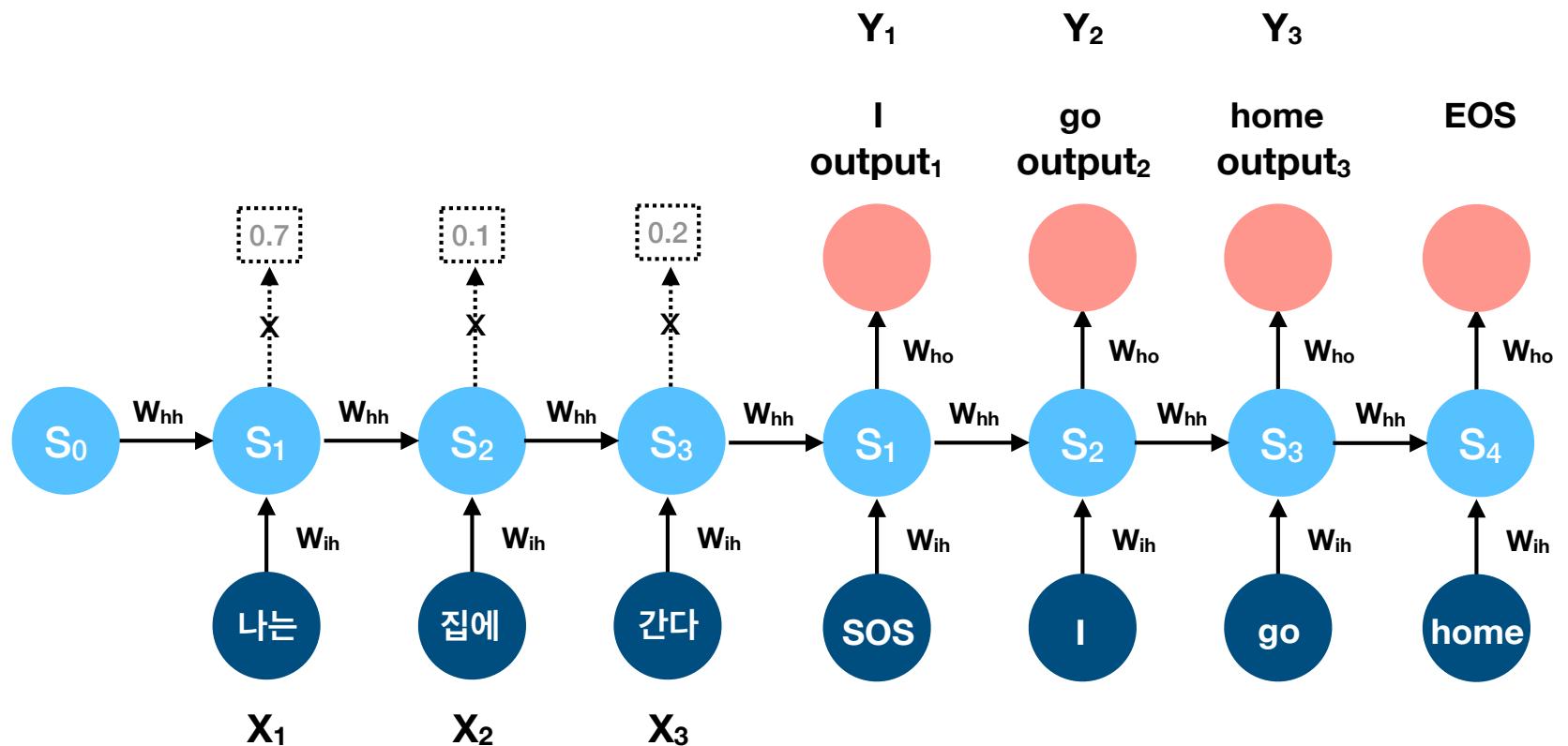
Transformer



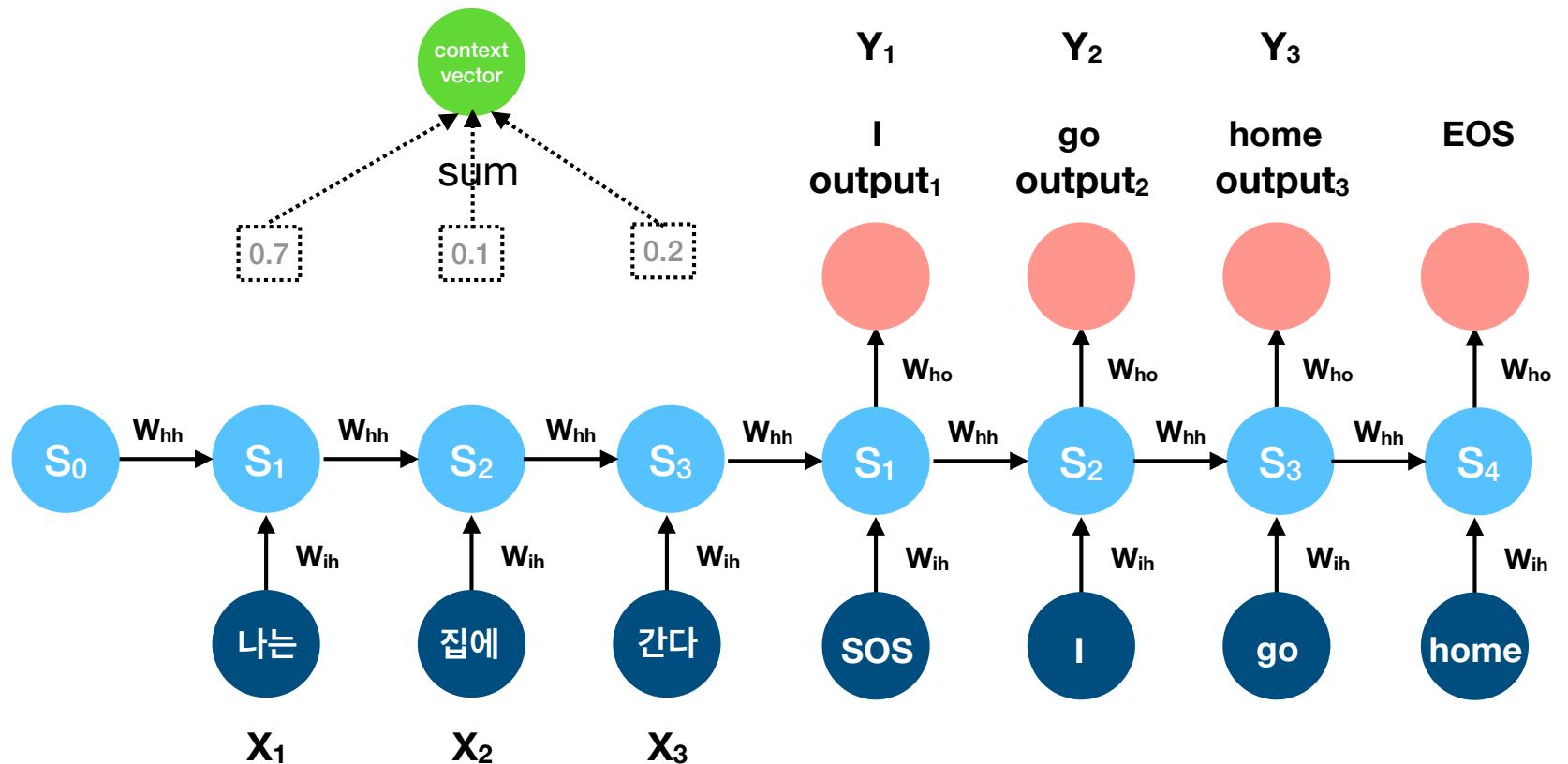
Transformer



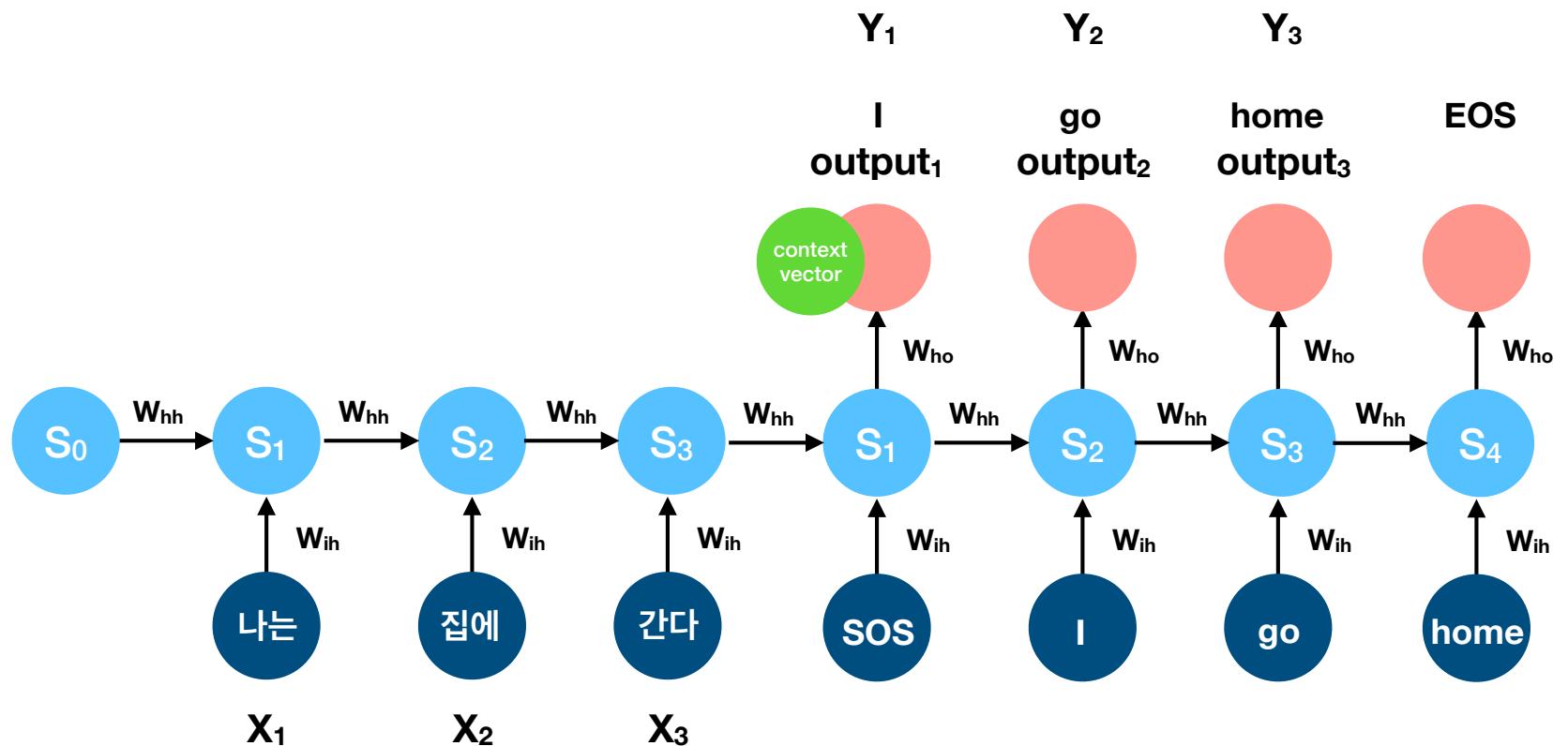
Transformer



Transformer

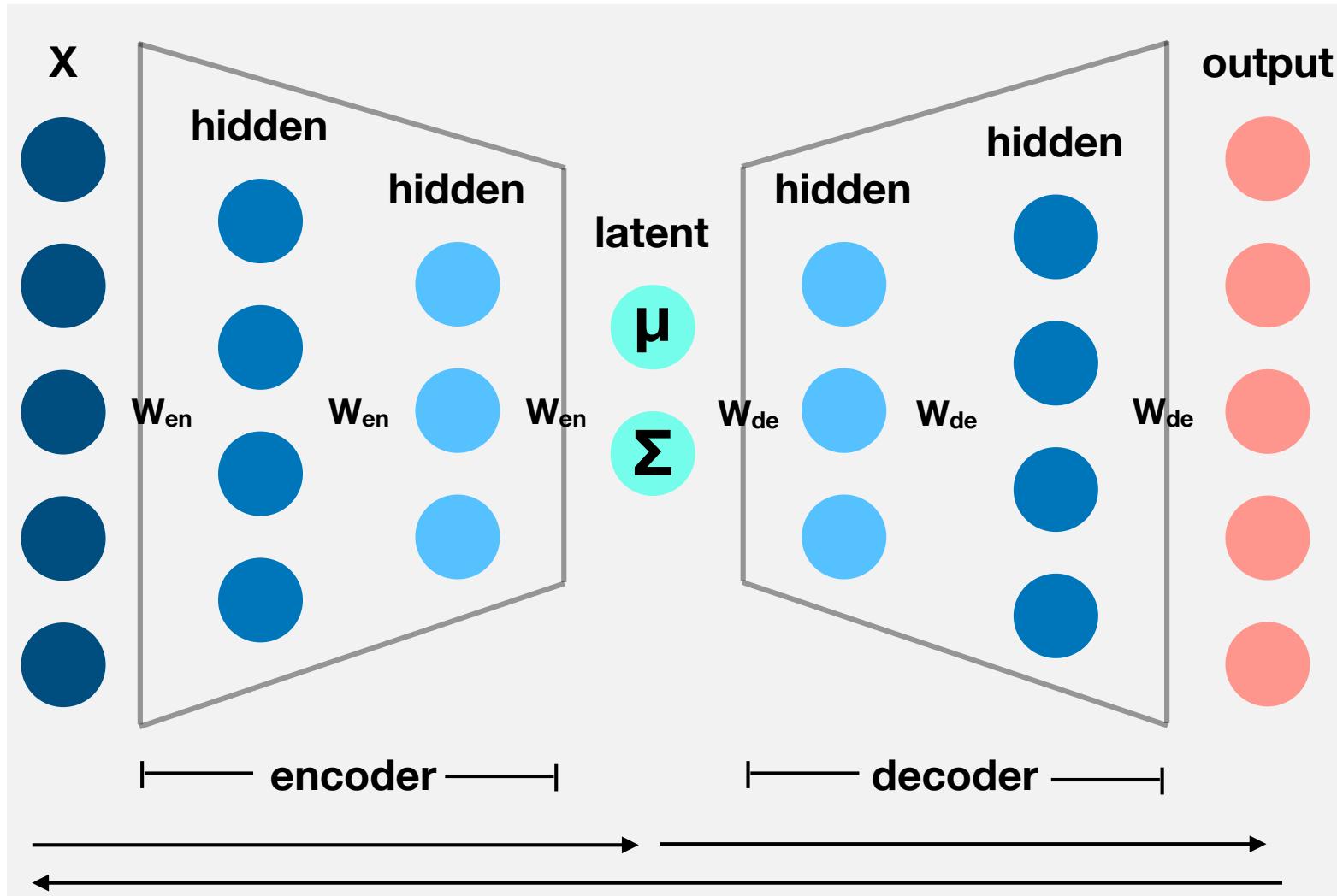


Transformer



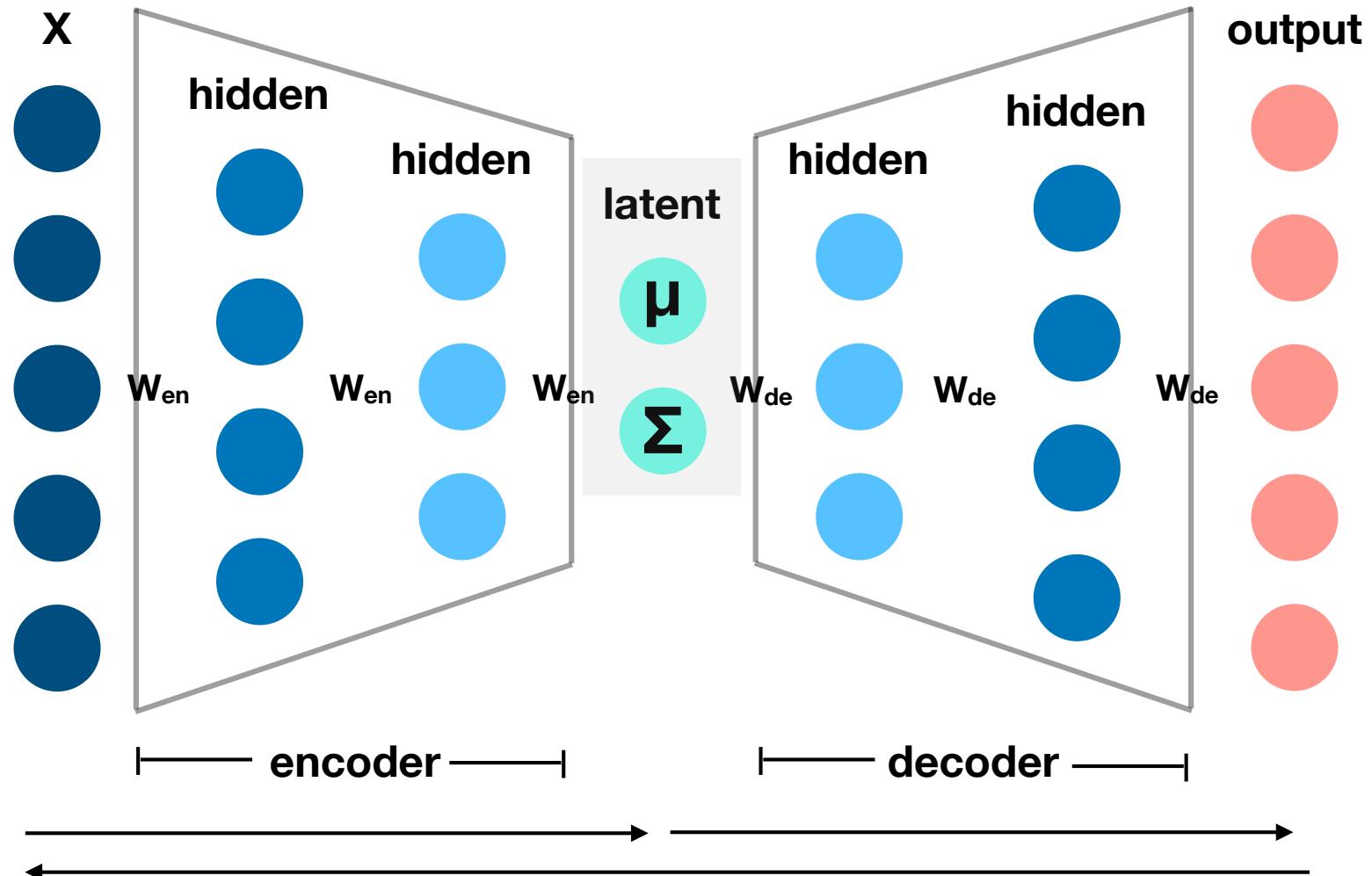
VAE

Variational Autoencoder



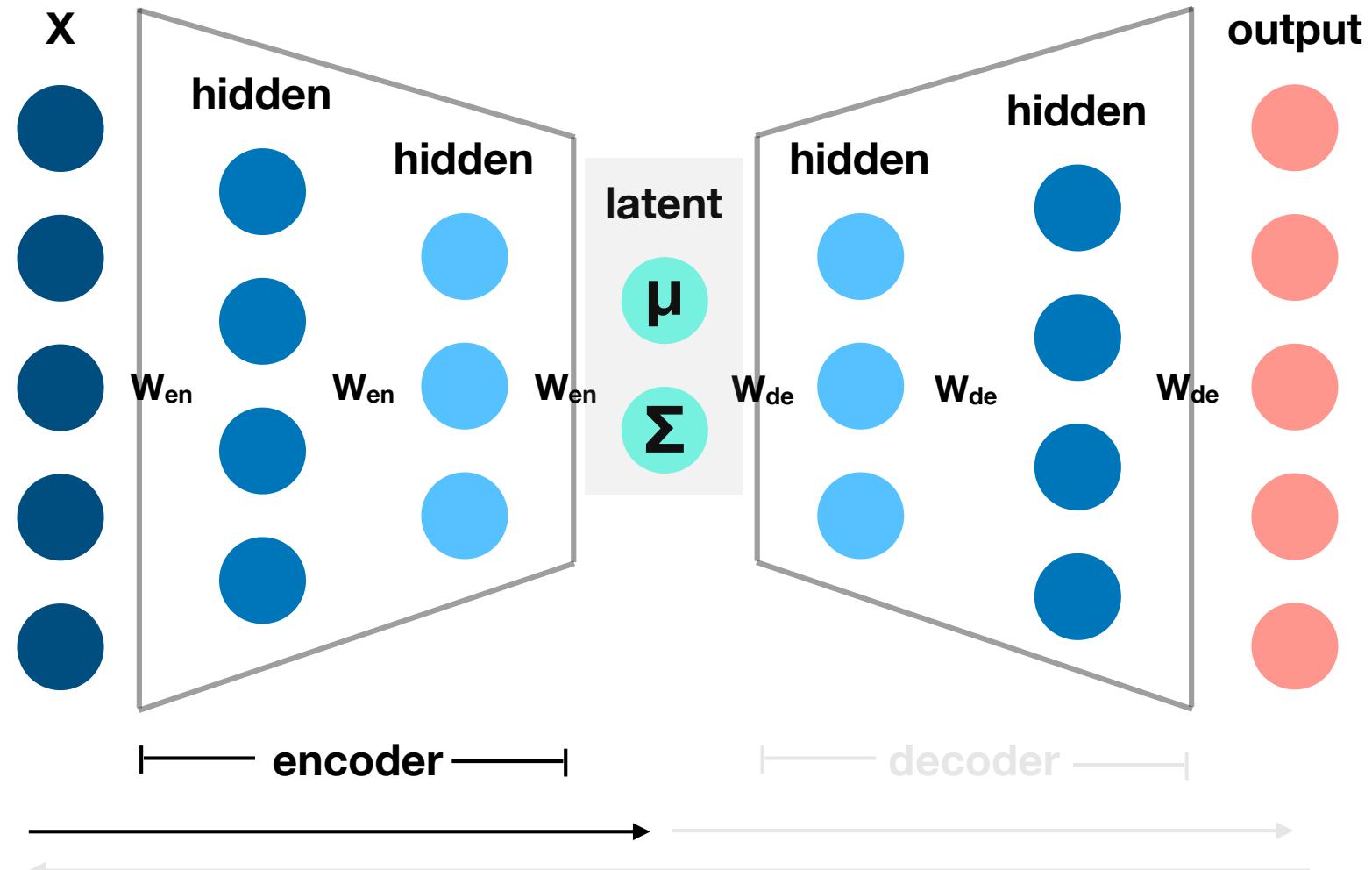
VAE는 기본적으로 autoencoder와 같이 encoder와 decoder로 이루어져 있습니다.

Variational Autoencoder



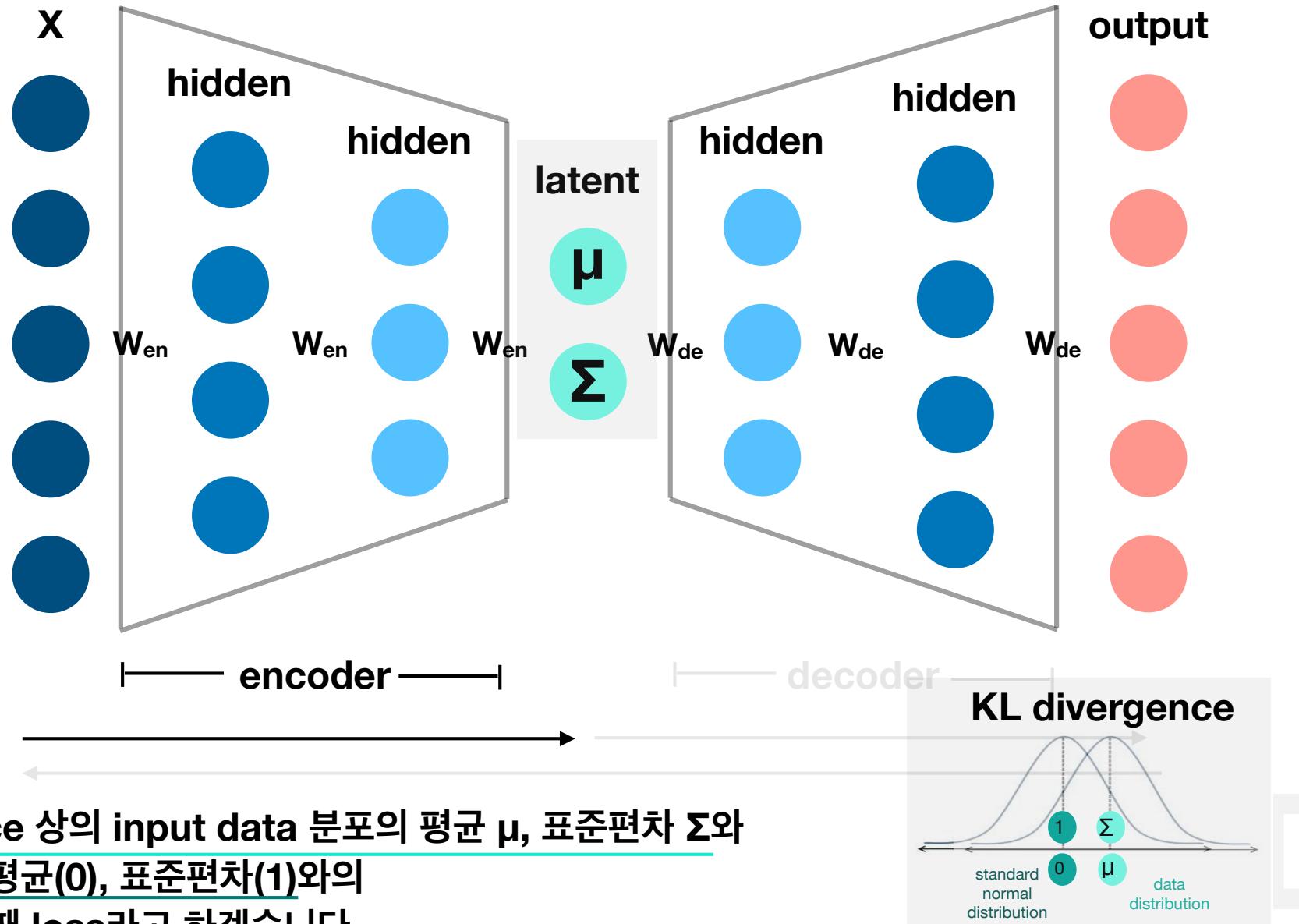
하지만 autoencoder와는 달리, 가운데에 layer 한 개를 추가해줍니다.
이를 latent layer라고 합니다.

Variational Autoencoder

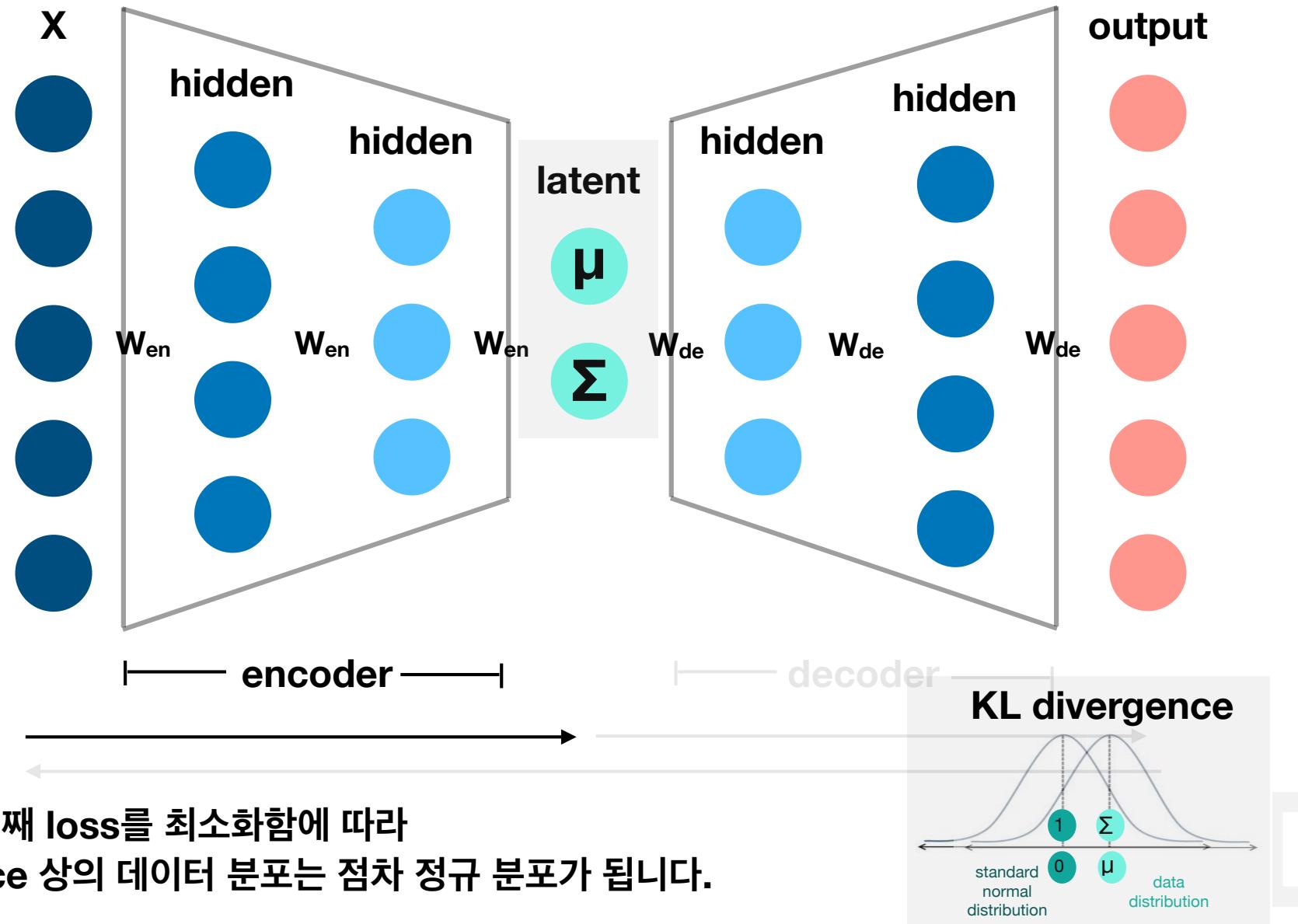


encoder를 거친 이 latent layer의 값이
input 데이터 분포의 평균(μ)과 표준편차(Σ)라고 가정해봅시다.

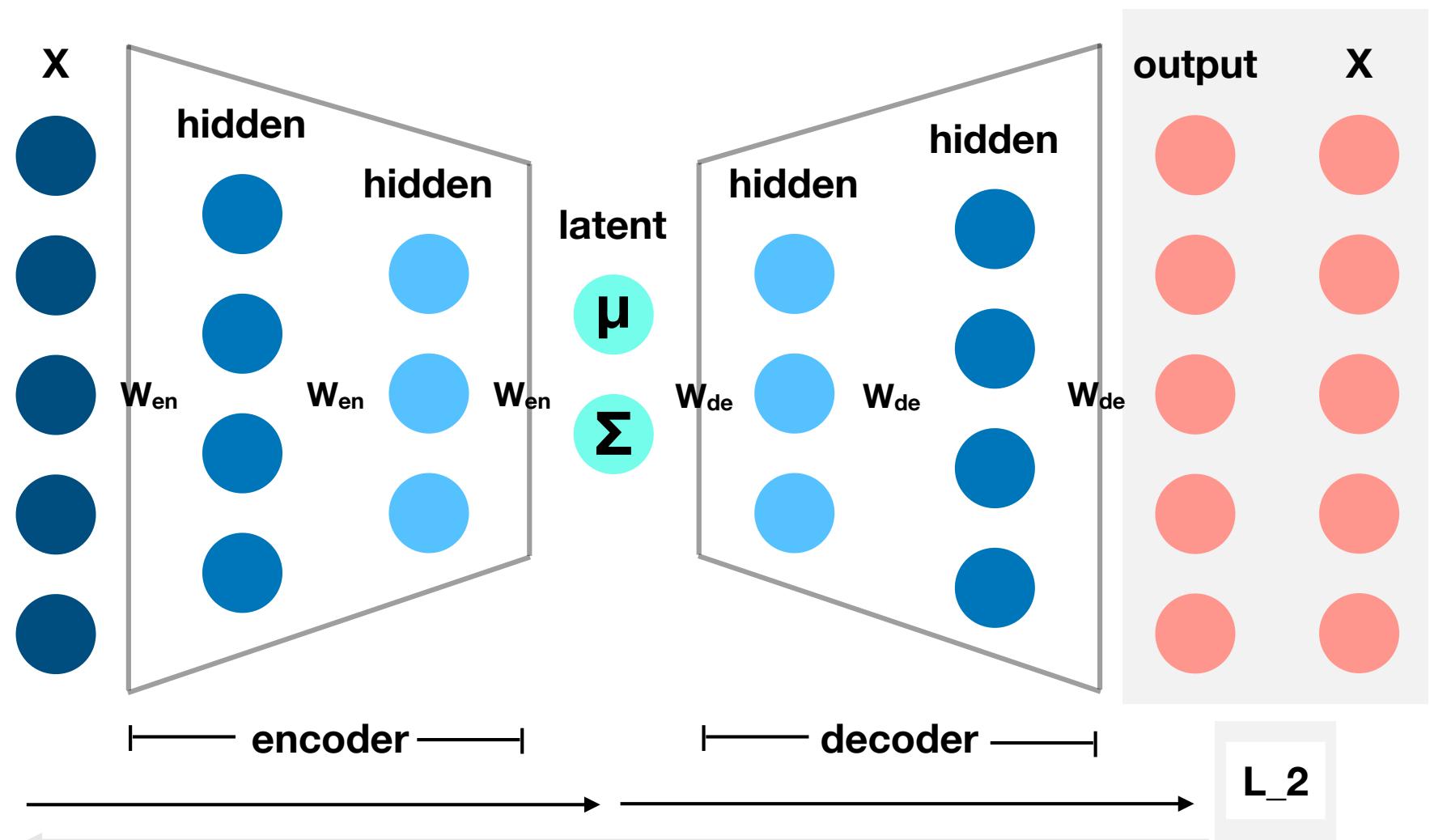
Variational Autoencoder



Variational Autoencoder



Variational Autoencoder

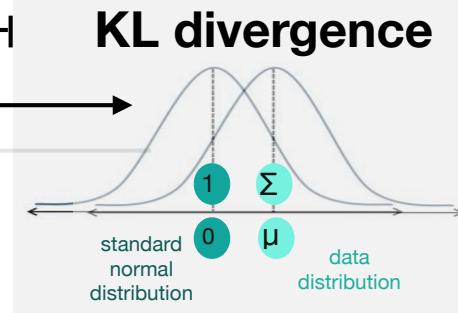
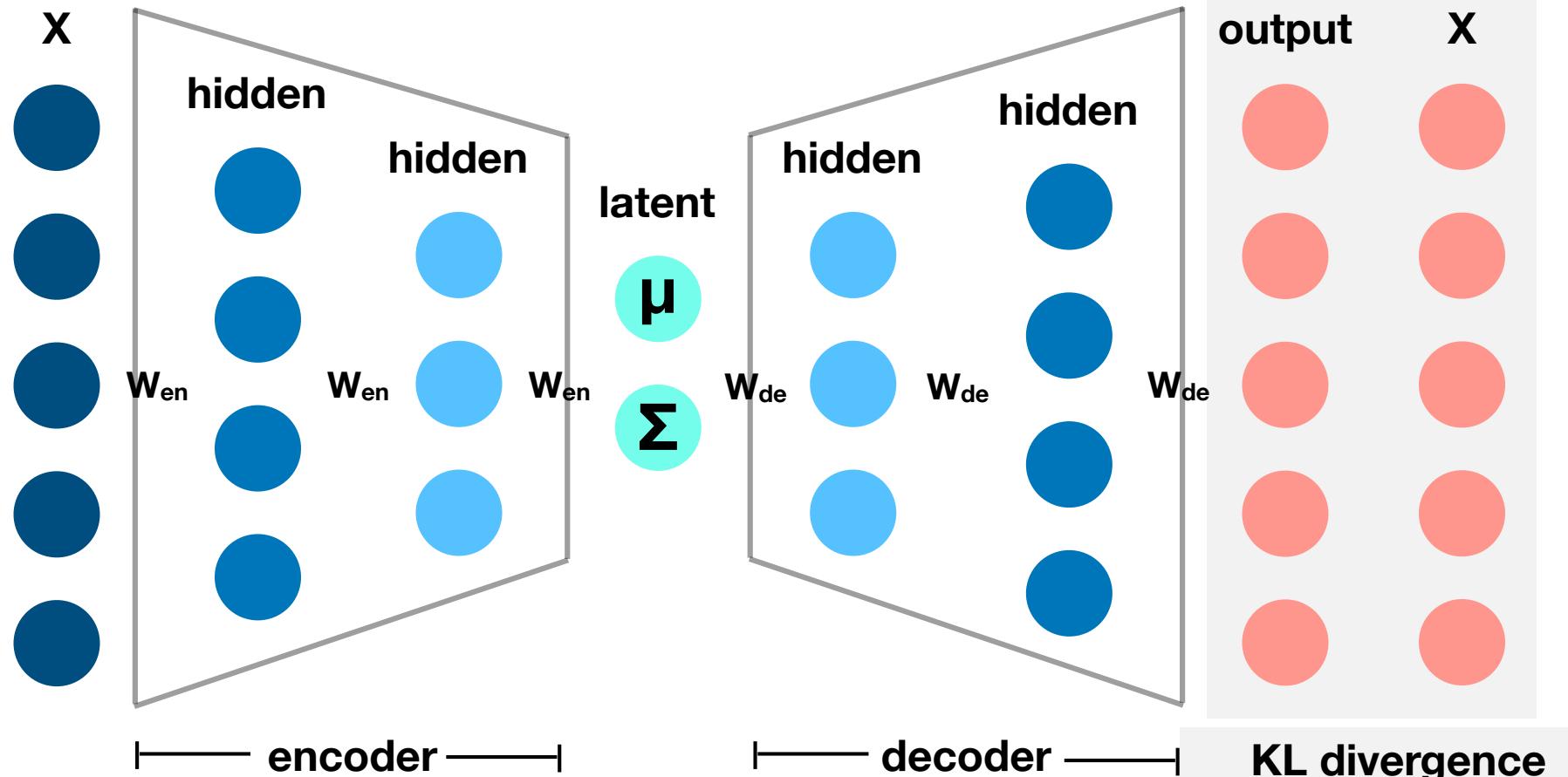


target은 X 자기 자신과 동일합니다.

output과 정답 target (X)을 비교하여 두 번째 loss를 구합니다.

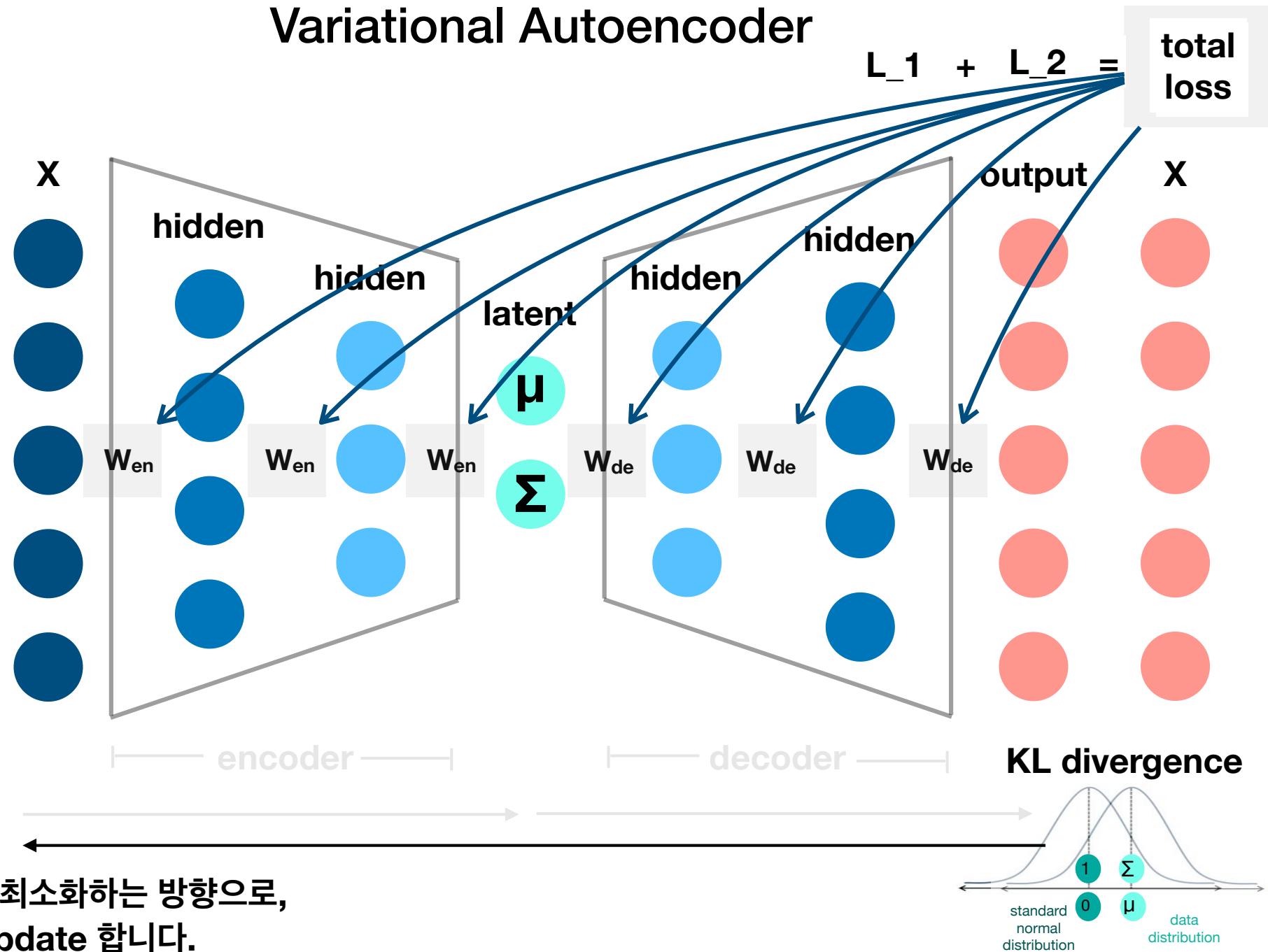
Variational Autoencoder

$$L_1 + L_2 = \text{total loss}$$

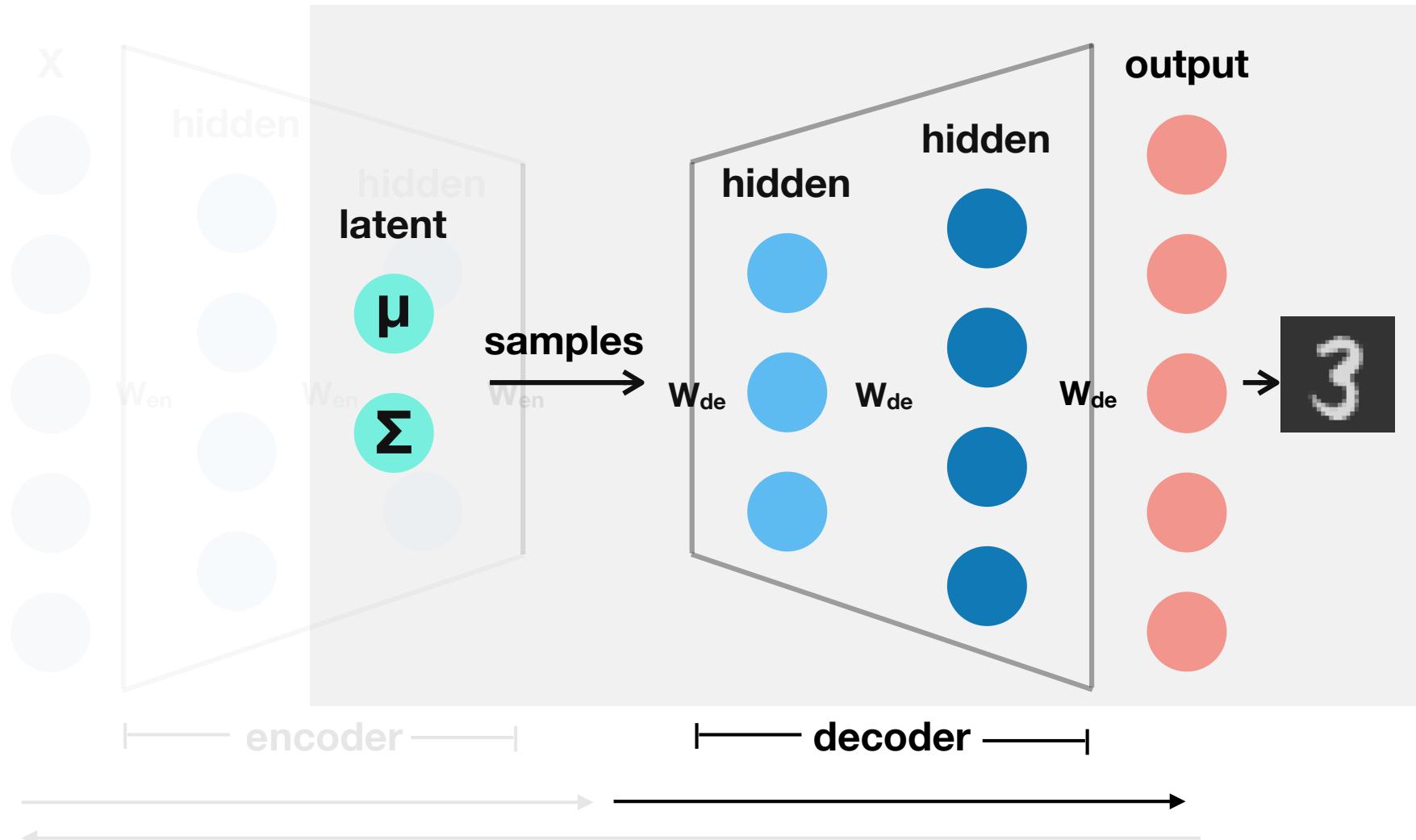


이제 첫 번째 loss와 두 번째 loss를 더한 전체 loss를 정의합니다.

Variational Autoencoder



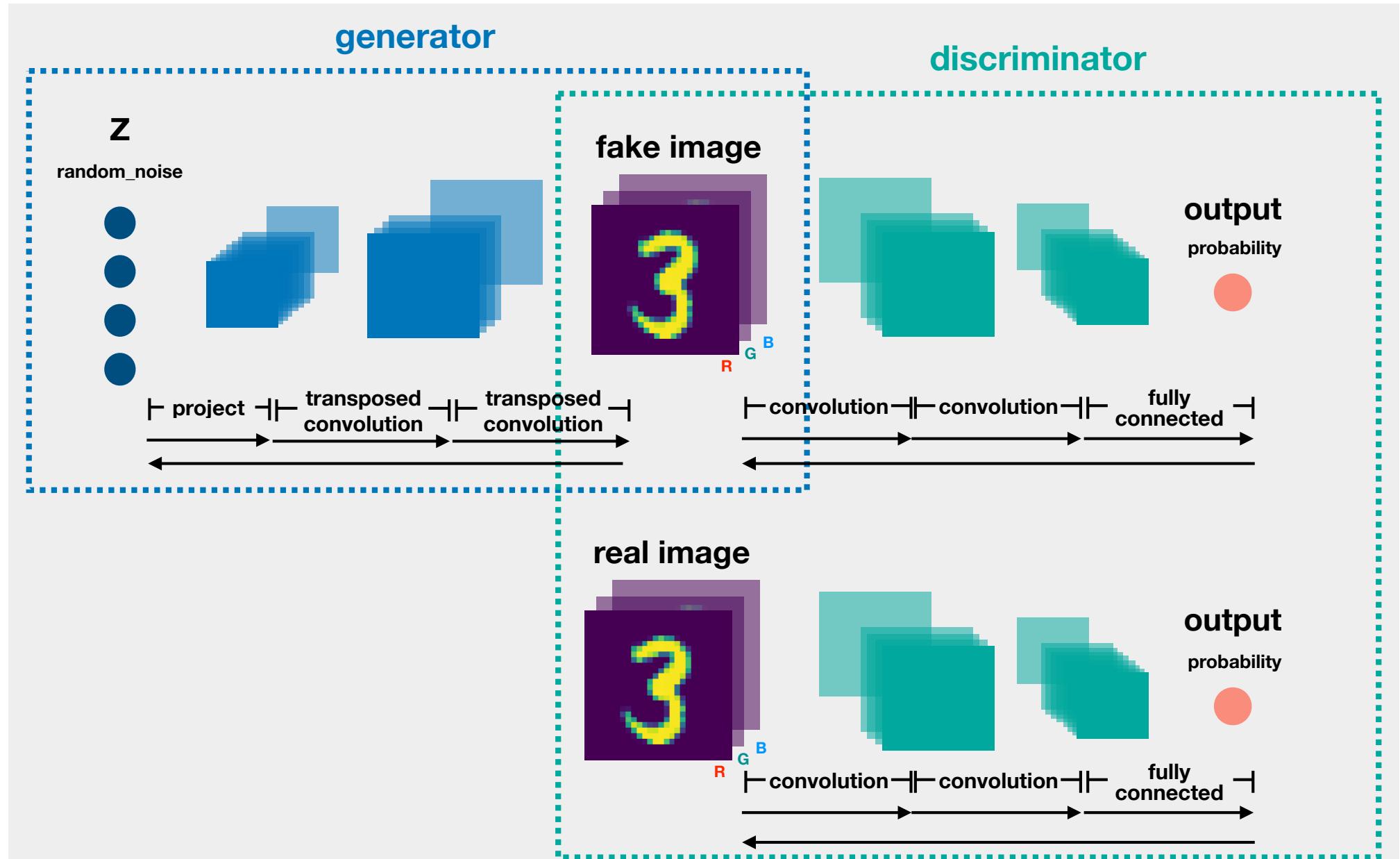
Variational Autoencoder



첫 번째 loss를 최소화함에 따라 latent layer 상의 데이터 분포는 점차 정규 분포가 되어가므로, 훈련 후에 latent layer의 정규 분포에서 random한 값을 뽑아 decoder에 넣으면 진짜 같은 이미지가 만들어집니다.

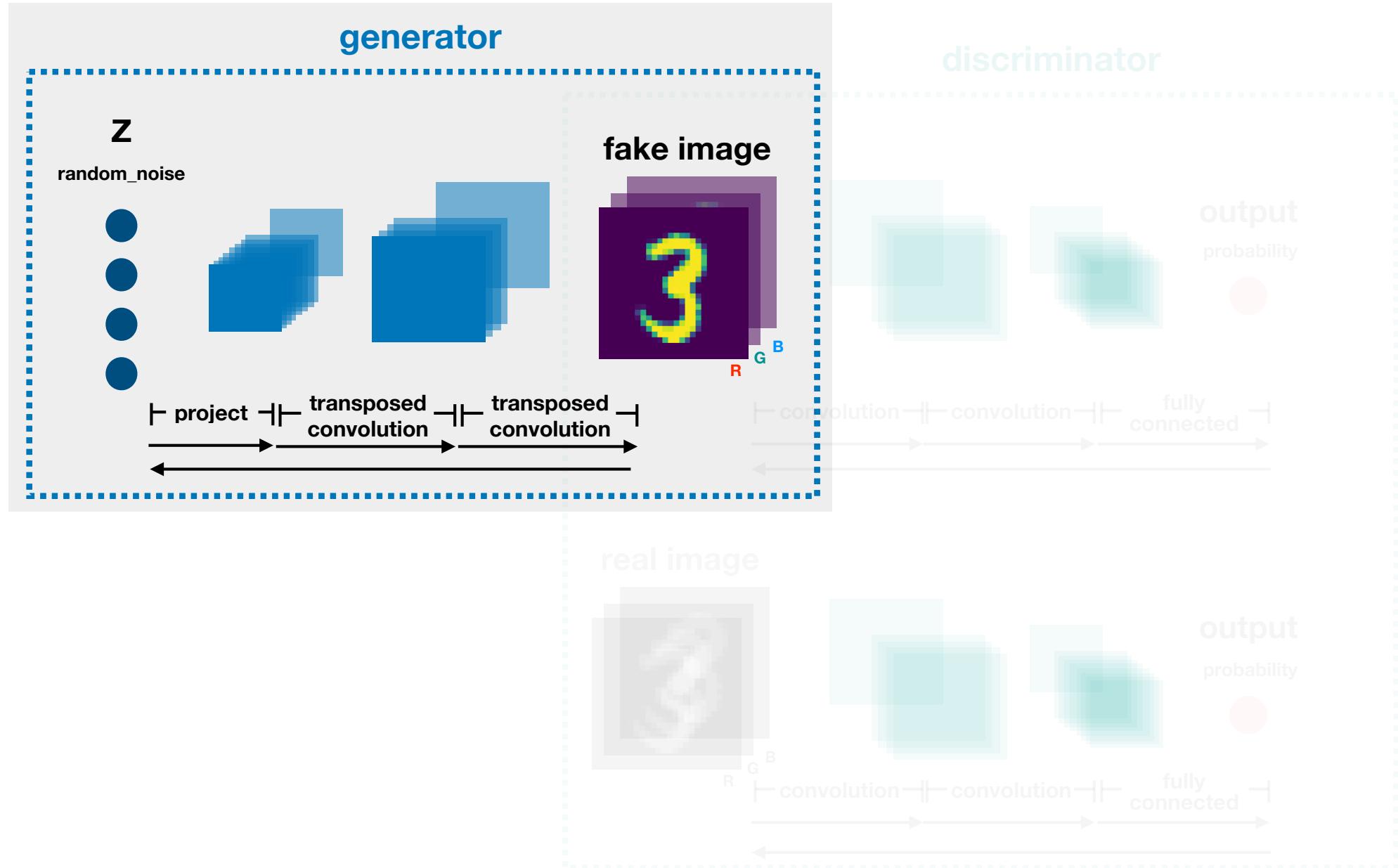
GAN

DCGAN



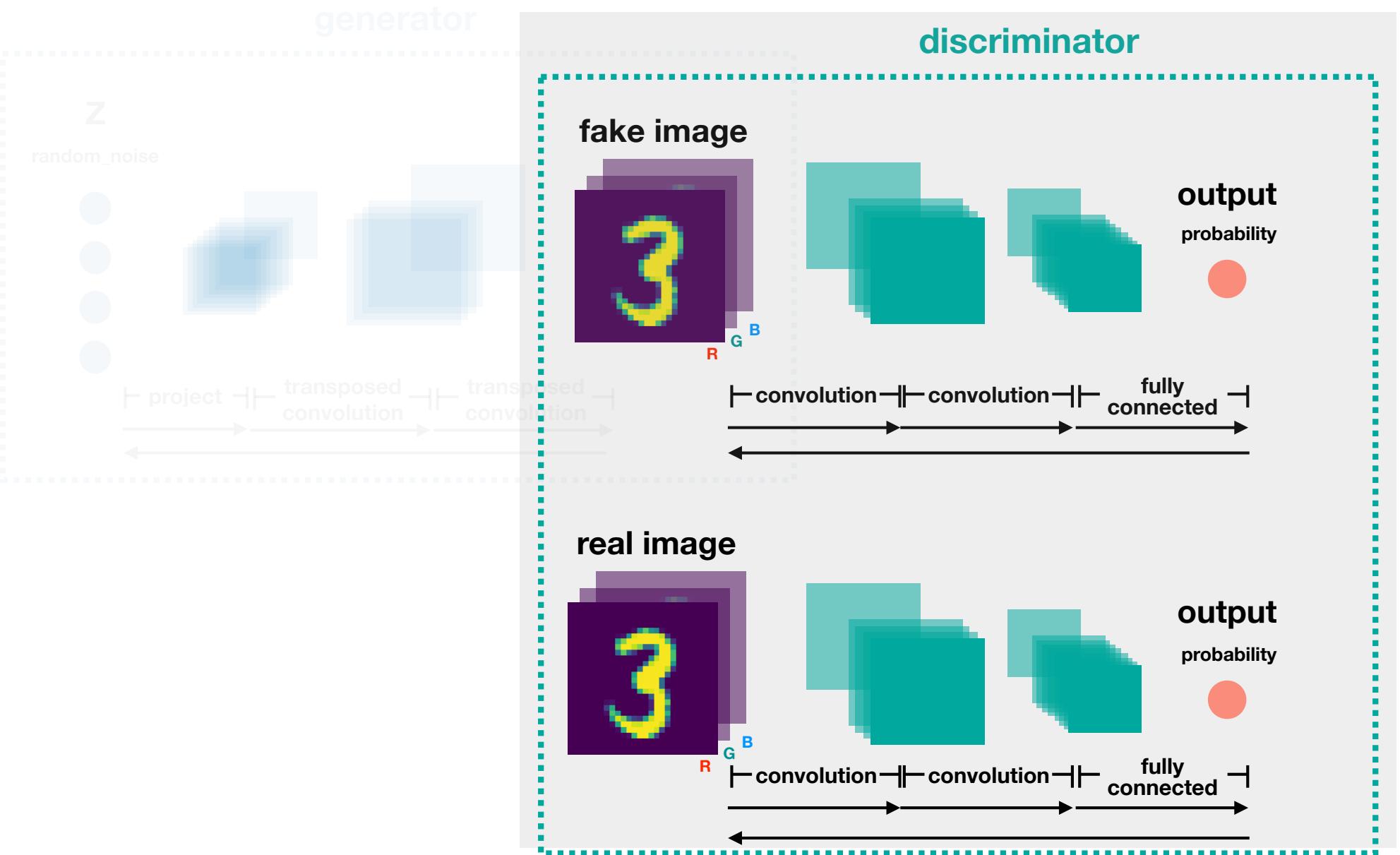
GAN은 두 부분으로 나뉩니다.

DCGAN



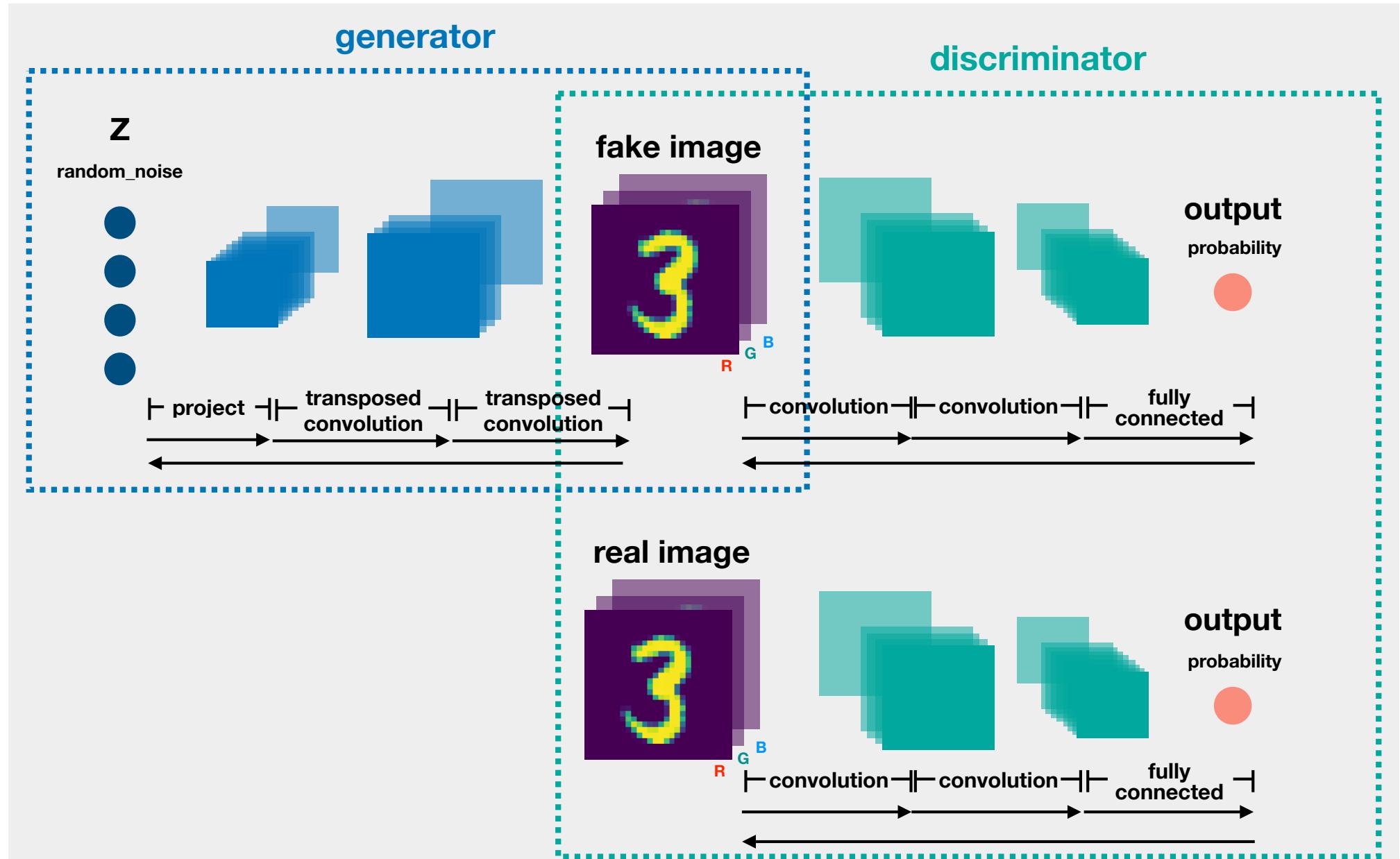
generator는 임의의 숫자 vector로부터 진짜 같은 이미지를 만듭니다.

DCGAN



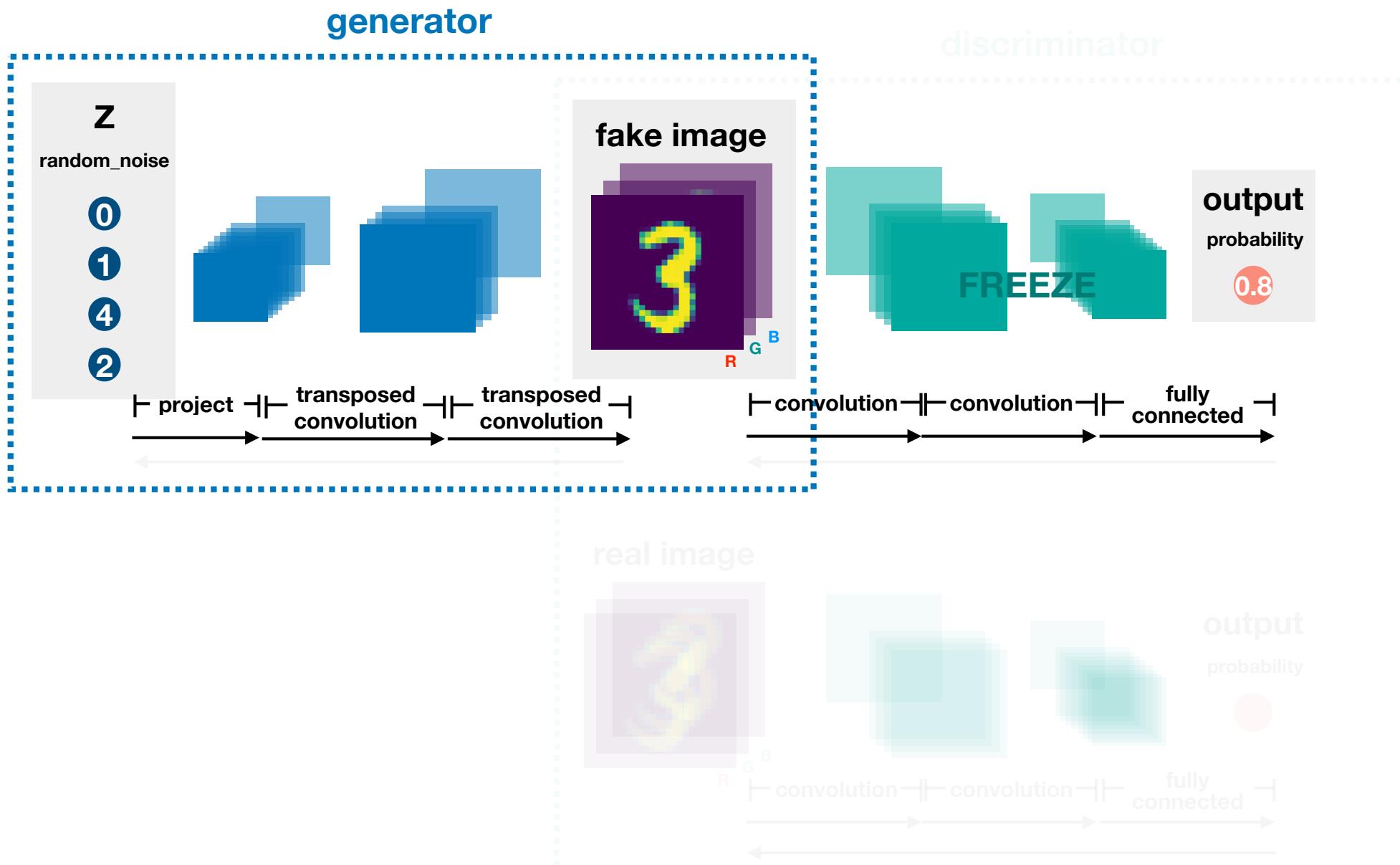
discriminator는 이미지를 input으로 받아서 그것이 진짜 이미지로 인식될 확률을 계산합니다.
generator가 만들어낸 가짜 이미지라면 0, 실제 데이터에서 온 진짜 이미지라면 1이라고 판단합니다.

DCGAN



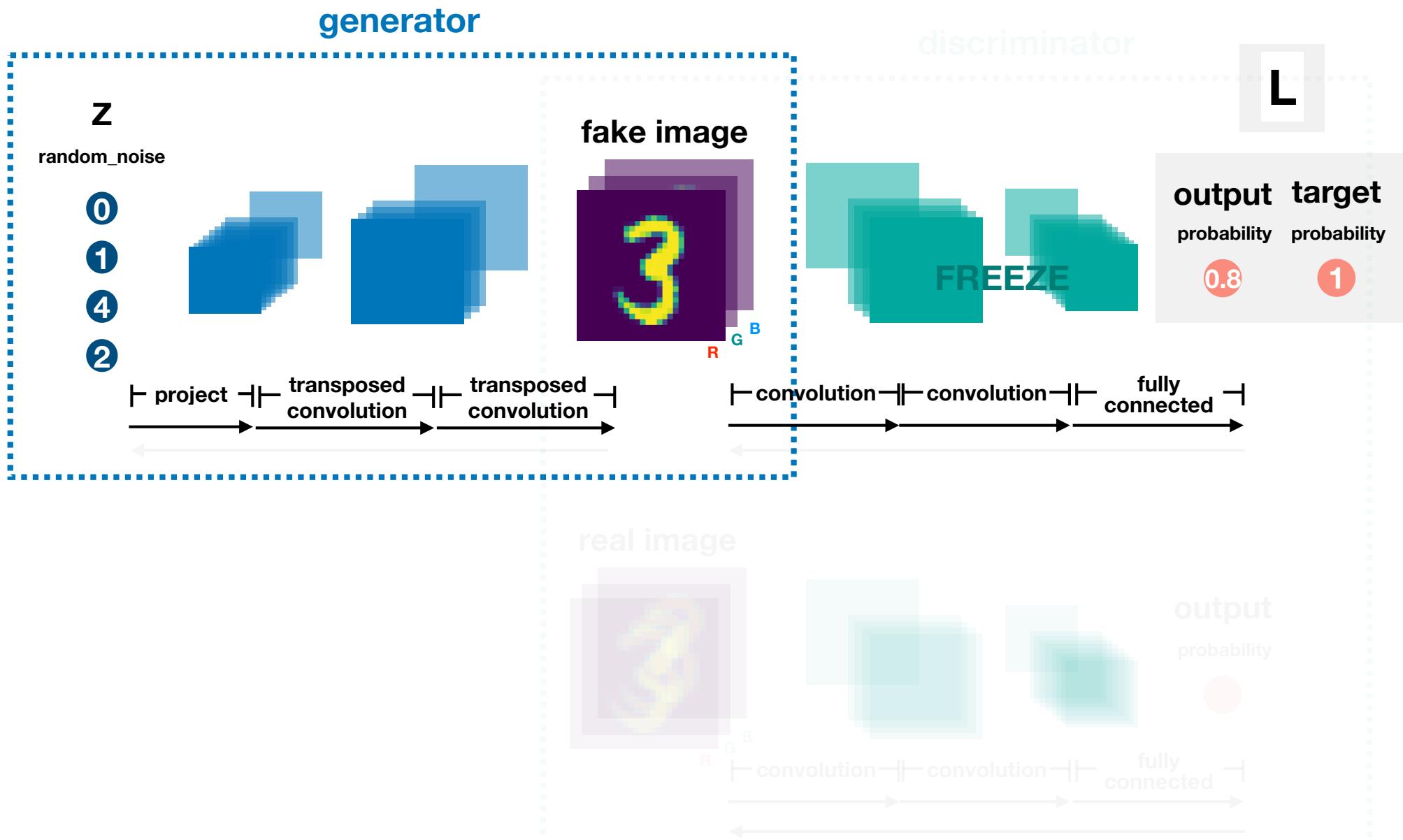
GAN의 훈련과정을 살펴봅시다.

DCGAN



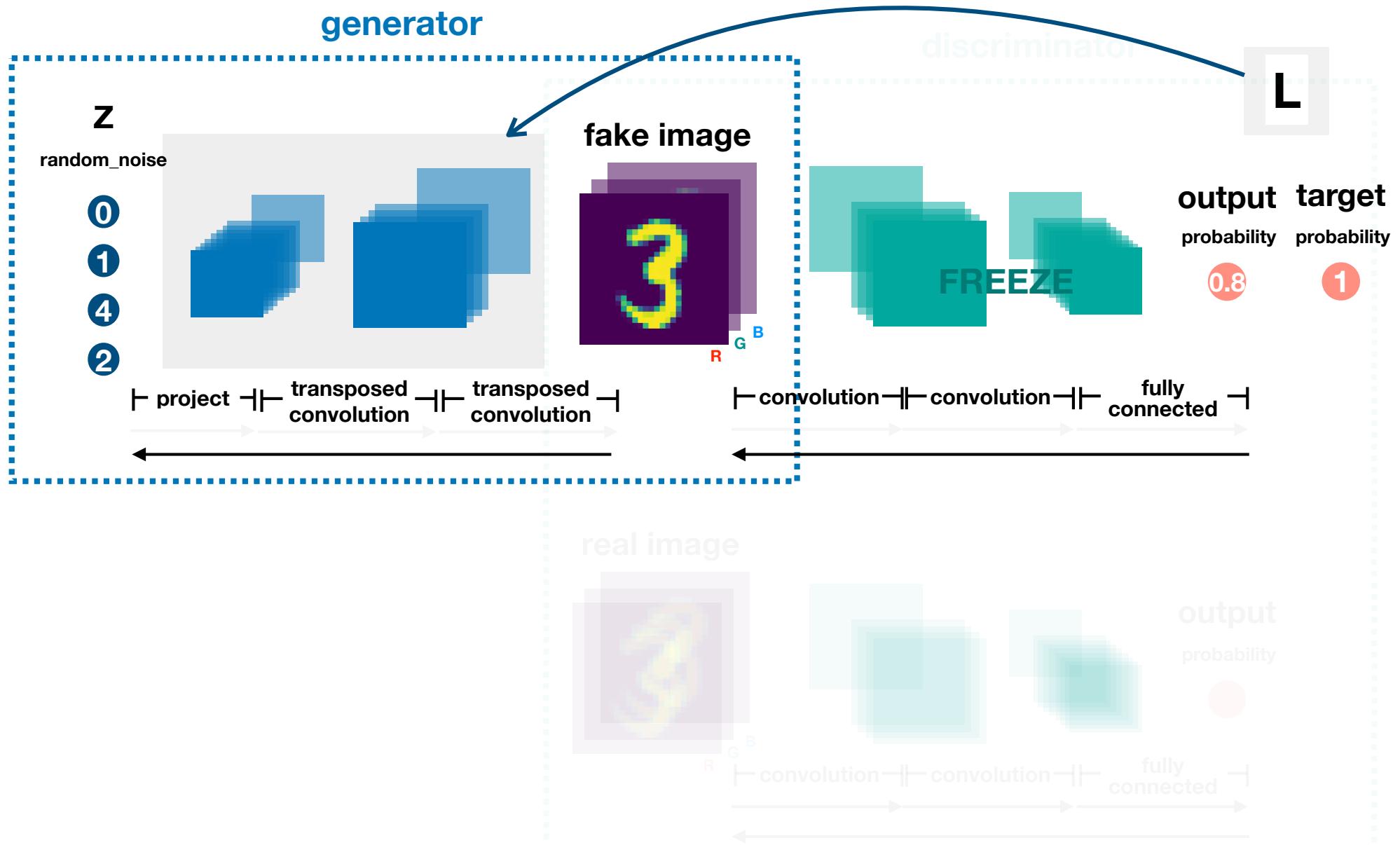
generator에 Z 를 넣어 가짜 이미지를 만들어내고,
discriminator를 통해 가짜 이미지가 진짜로 인식될 확률을 구합니다.

DCGAN



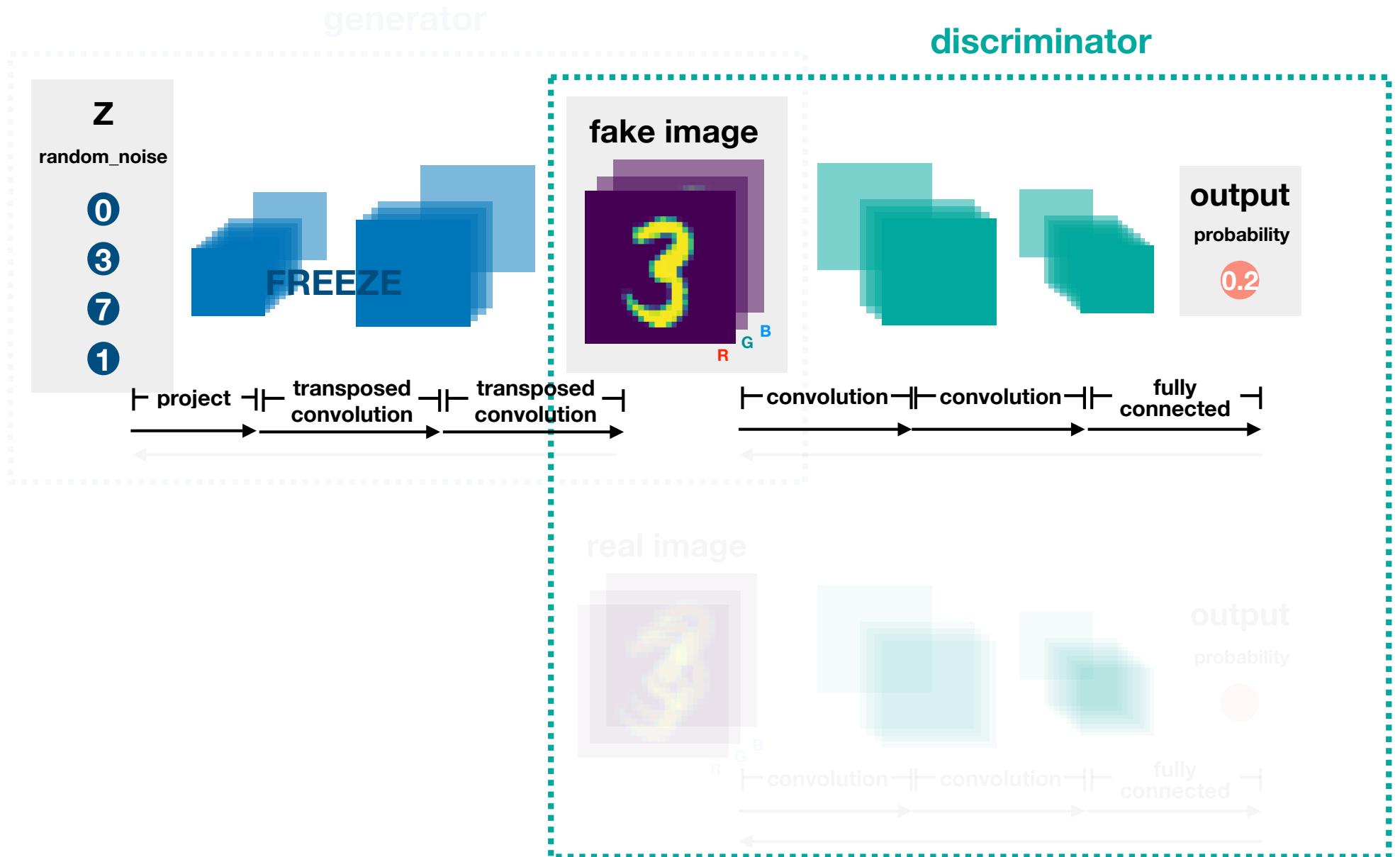
generator의 목적이 진짜 같은 이미지를 만드는 것이므로 target은 1입니다.
output과 정답 target (1)을 비교하여 loss를 구합니다.

DCGAN



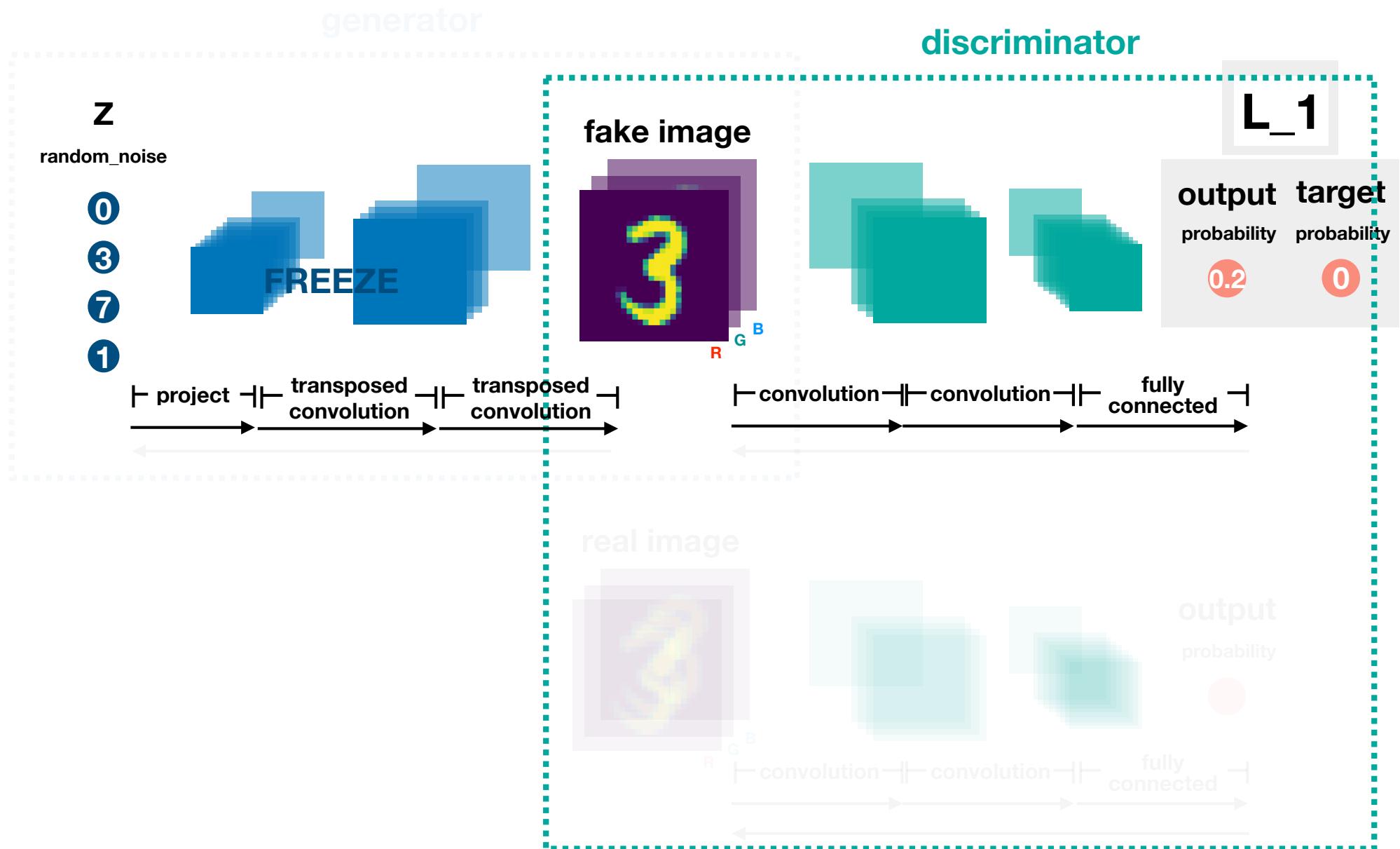
loss를 최소화하는 방향으로, weight을 update 합니다.

DCGAN



이번엔 discriminator 차례입니다. generator에 Z를 넣어 가짜 이미지를 만들어내고, discriminator를 통해 가짜 이미지가 진짜로 인식될 확률을 구합니다.

DCGAN



discriminator의 목적이 진짜 이미지를 구별하는 것이므로 target은 0입니다.
output과 정답 target (0)을 비교하여 첫 번째 loss를 구합니다.

DCGAN



이번에는 **discriminator**에 실제 이미지를 넣고 이미지가 진짜일 확률을 구합니다.

DCGAN



discriminator의 목적이 진짜 이미지를 구별하는 것이므로 target은 1입니다.
output과 정답 target (1)을 비교하여 두 번째 loss를 구합니다.

DCGAN

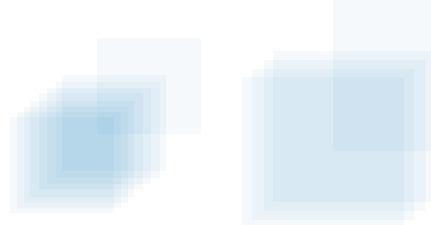
=

total
loss

generator

discriminator

Z
random_noise



project transposed convolution transposed convolution

fake image



convolution convolution fully connected

output target
probability probability

0.2

0

L_1

real image



convolution convolution fully connected

output target
probability probability

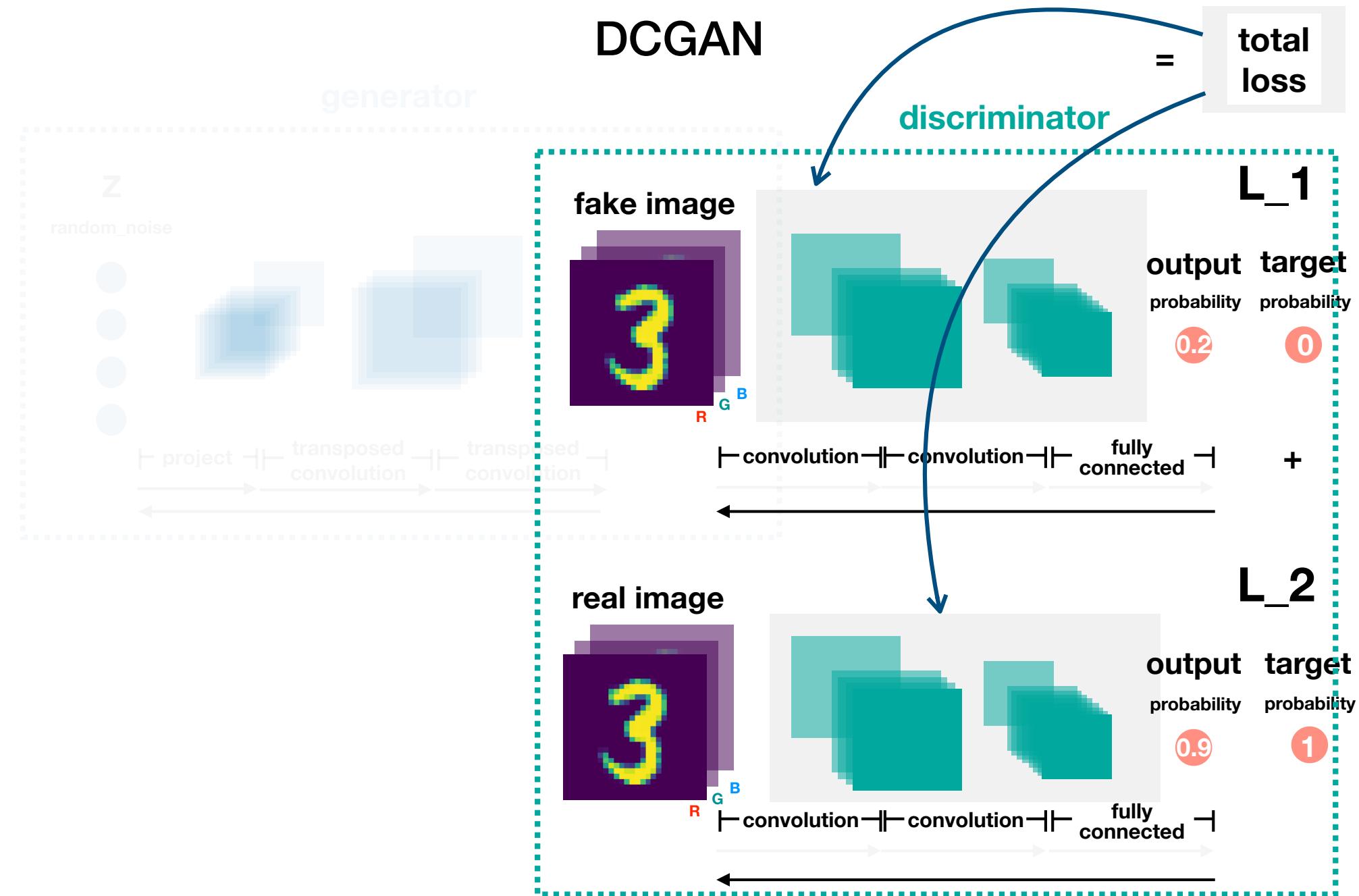
0.9

1

L_2

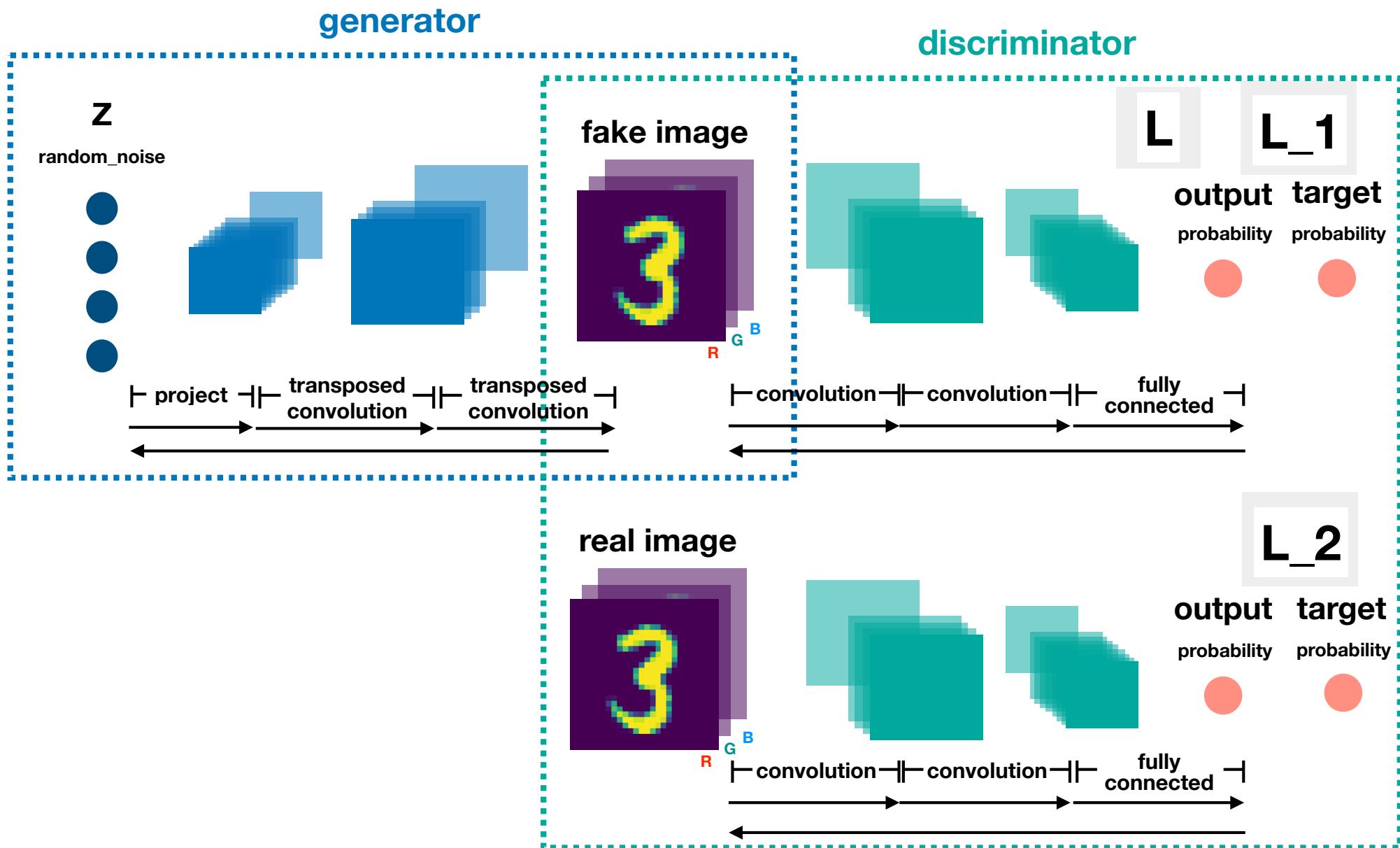
이제 첫 번째 loss와 두 번째 loss를 더한 전체 loss를 정의합니다.

DCGAN



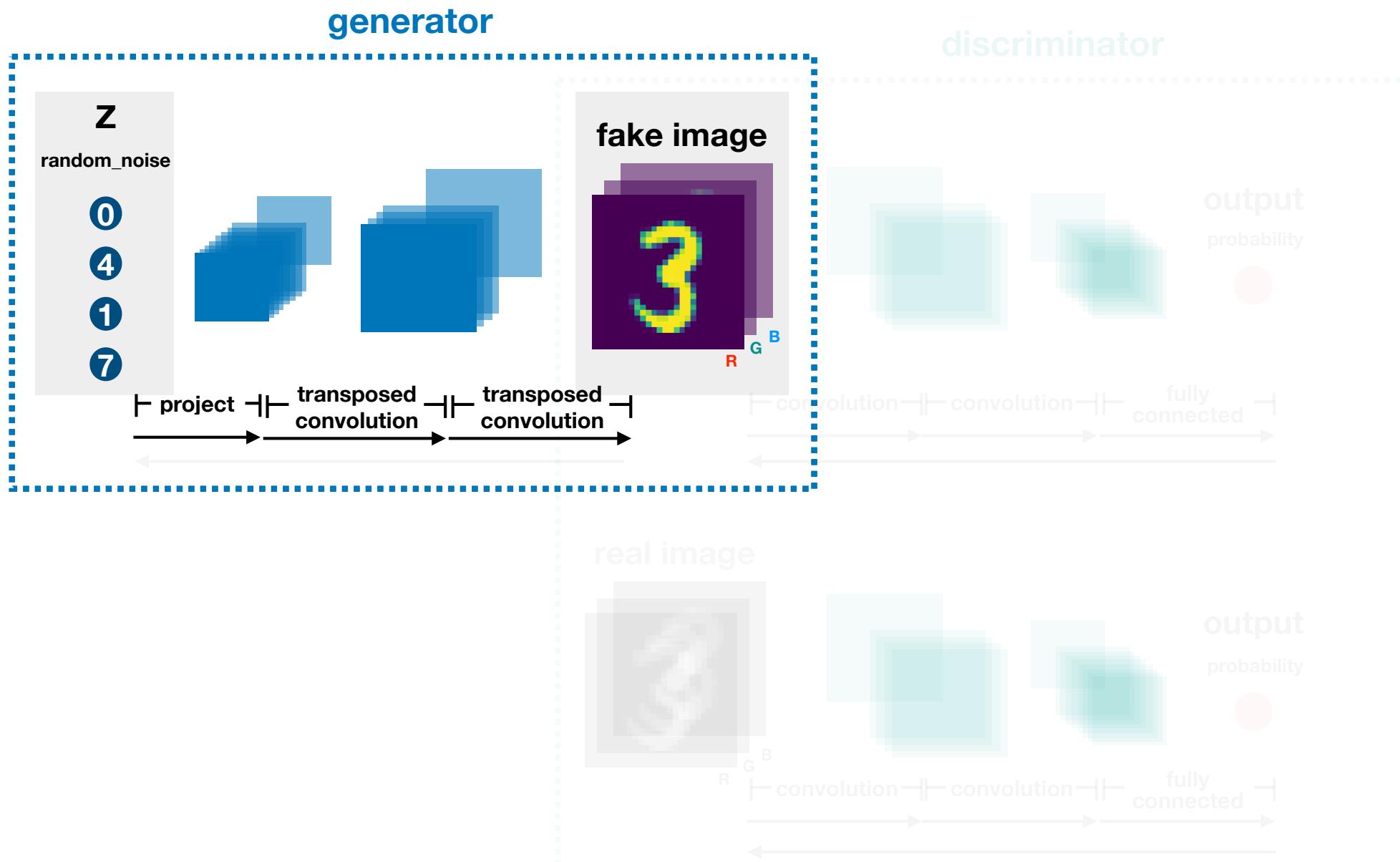
전체 loss를 최소화하는 방향으로, weight를 update 합니다.

DCGAN



generator loss와 discriminator loss는 하나가 작아지면 다른 하나가 커지는 trade-off 관계입니다.
두 loss가 모두 적당히 작아지는 타협점은 모델에 따라 다를 수 있습니다.

DCGAN



훈련이 끝난 뒤 random한 숫자 vector를 generator에 넣어주면 진짜 같은 이미지가 만들어집니다.