MY BRIGHT FUTURE

DSU Dongseo University 동서대학교

Game Programming:

Artificial Neural Network

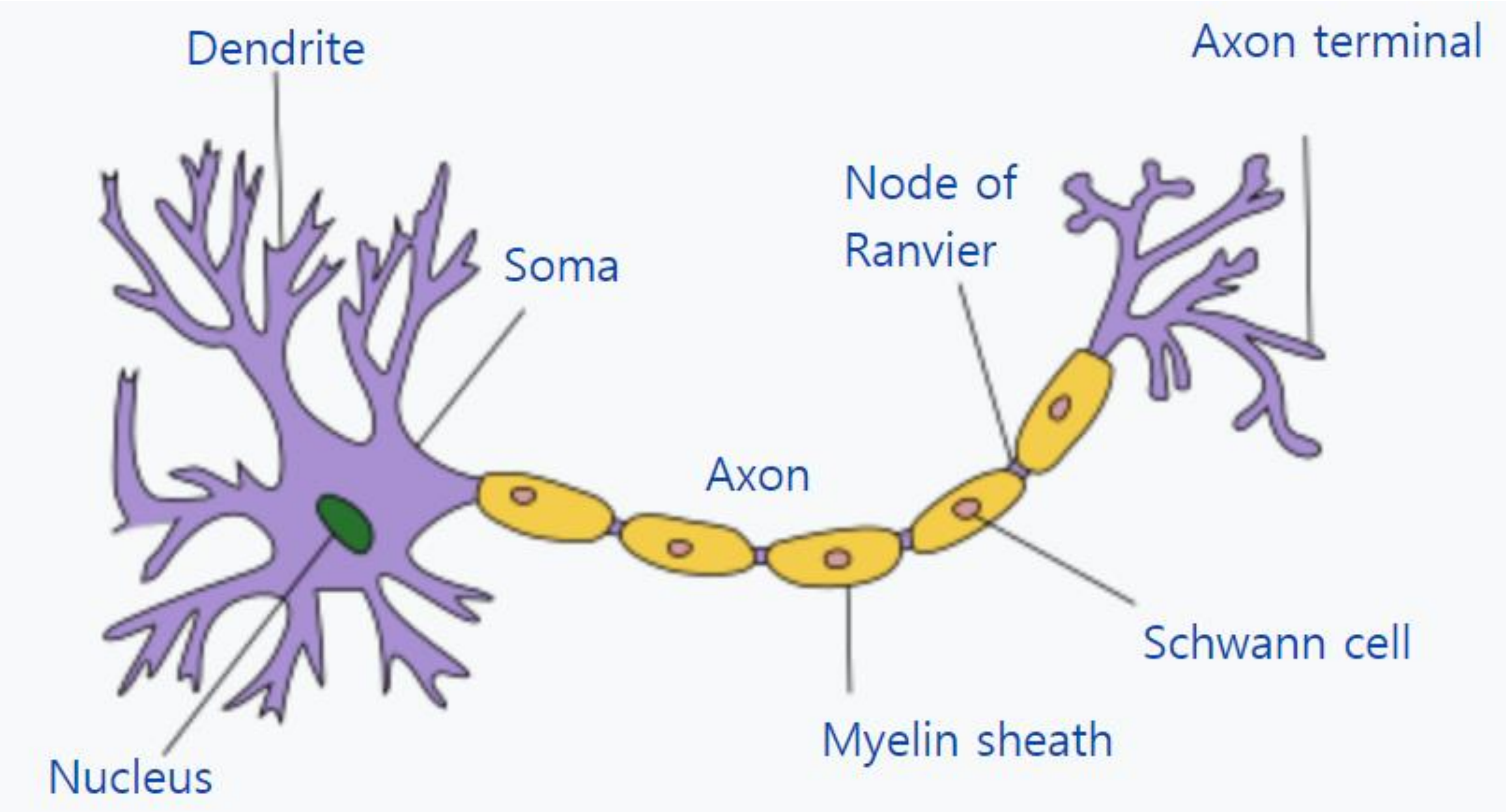
jintaeks@gmail.com

Division of Digital Contents, DongSeo University.

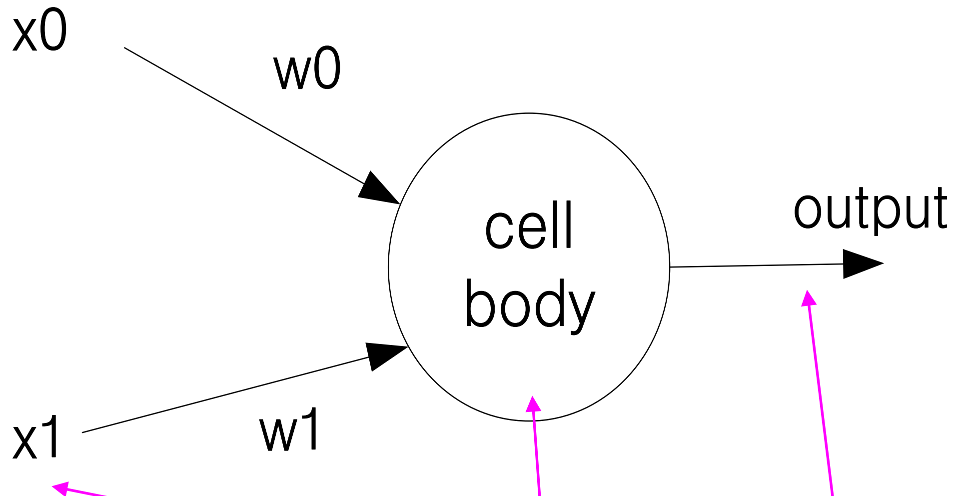
January 2019

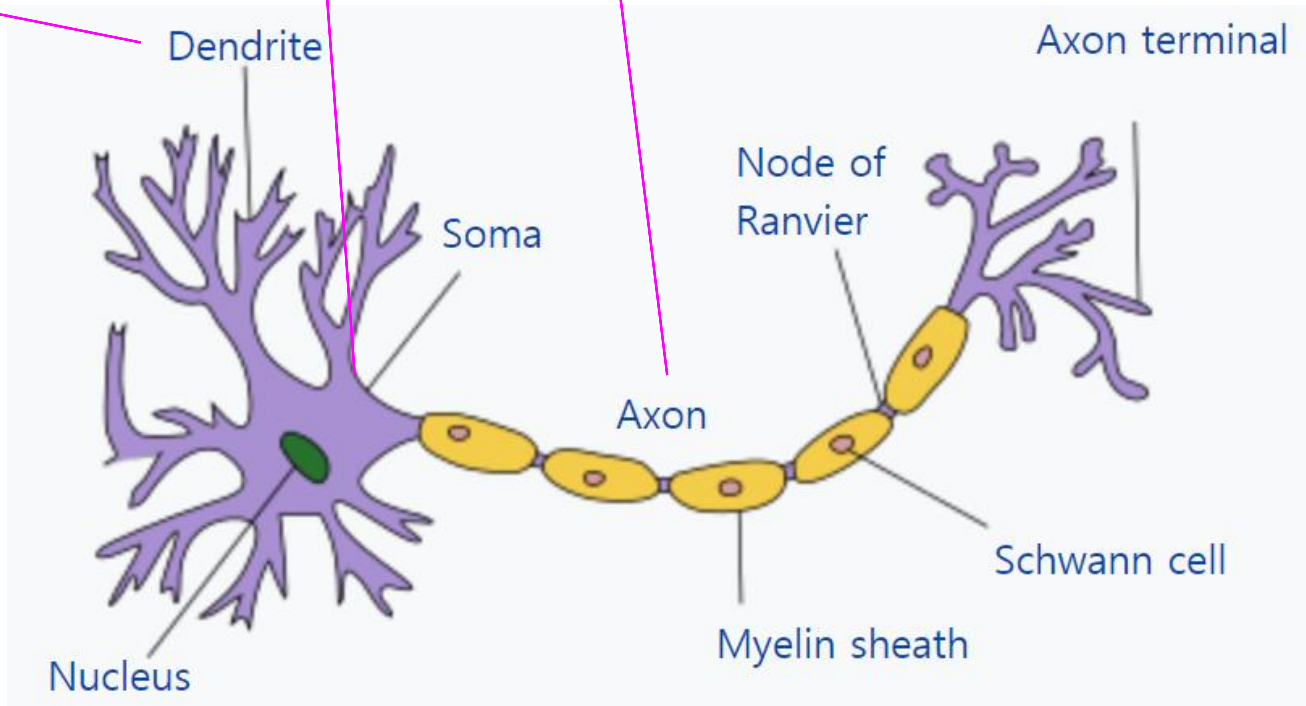
Neural Network

- ✓ Neuron
  - More than 100 billion neurons in a brain
- ✓ dendrites
- ✓ axon



Perceptron





Step1 : Receive inputs

Input 0: x0 = 12
Input 1: x1 = 4

Step 2: Weight inputs

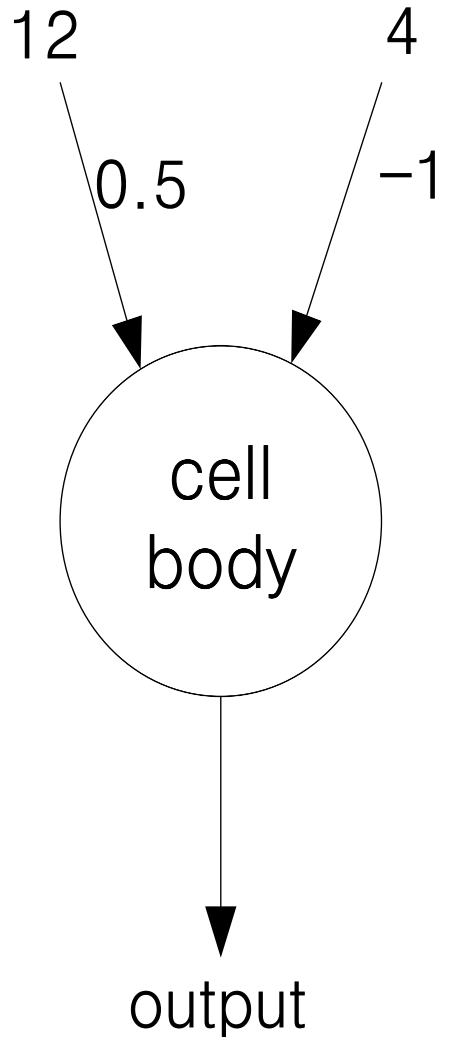
Weight 0: 0.5

Weight 1: -1

✓ We take each input and multiply it by its weight.

Input 0 * Weight 0 ⇒ 12 * 0.5 = 6

Input 1 * Weight 1 ⇒ 4 * -1 = -4



Step 3: Sum inputs

✓ The weighted inputs are then summed.

Sum = 6 + -4 = 2

Step 4: Generate output

Output = sign(sum) ⇒ sign(2) ⇒ +1

The Perceptron Algorithm:

1. For every input, multiply that input by its weight.
2. Sum all of the weighted inputs.
3. Compute the output of the perceptron based on that sum passed through an activation function (the sign of the sum).

```
float inputs[] = {12 , 4};
float weights[] = {0.5,-1};

float sum = 0;
for (int i = 0; i < inputs.length; i++) {
        sum += inputs[i]*weights[i];
}
```

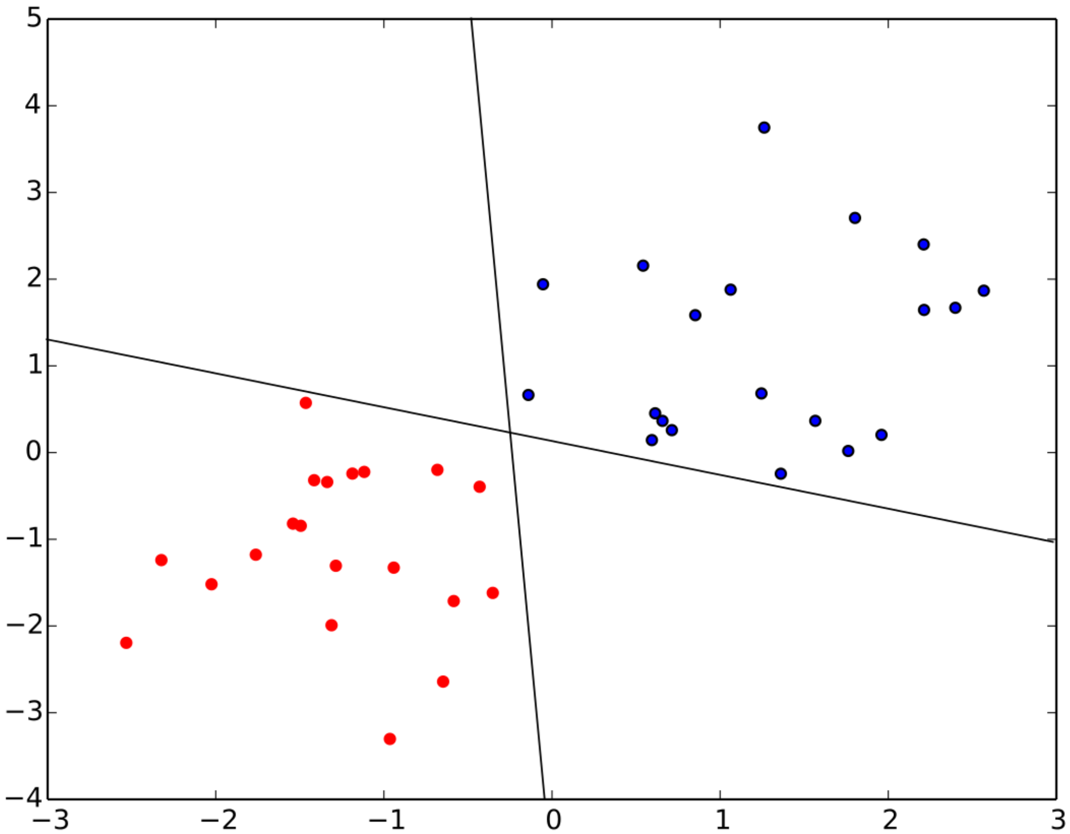
The Perceptron Algorithm:

1. For every input, multiply that input by its weight.
2. Sum all of the weighted inputs.
3. Compute the output of the perceptron based on that sum passed through an **activation function** (the sign of the sum).

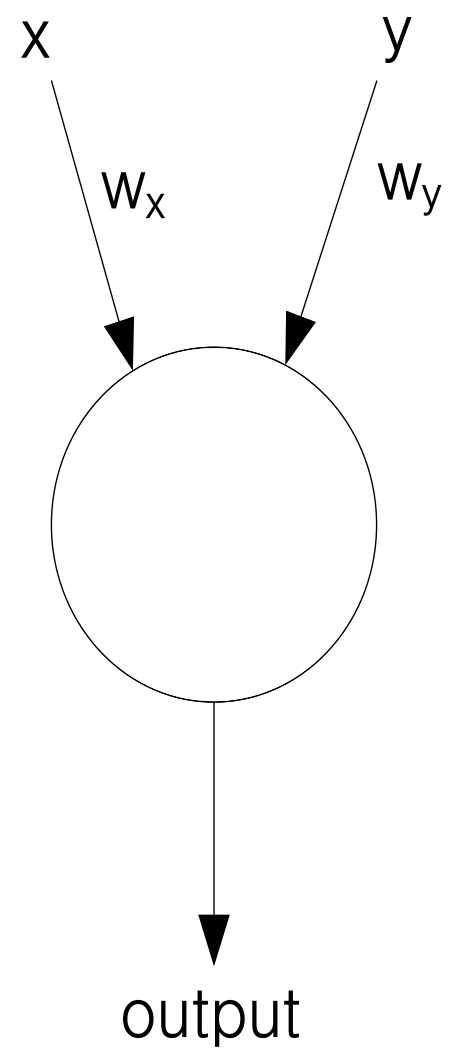
```
float output = activate(sum);

int activate(float sum) { // Return a 1 if positive, -1 if negative.
    if (sum > 0) return 1;
    else return -1;
}
```

Simple Pattern Recognition using a Perceptron

- ✓ Consider a line in two-dimensional space.
- ✓ Points in that space can be classified as living on either one side of the line or the other.



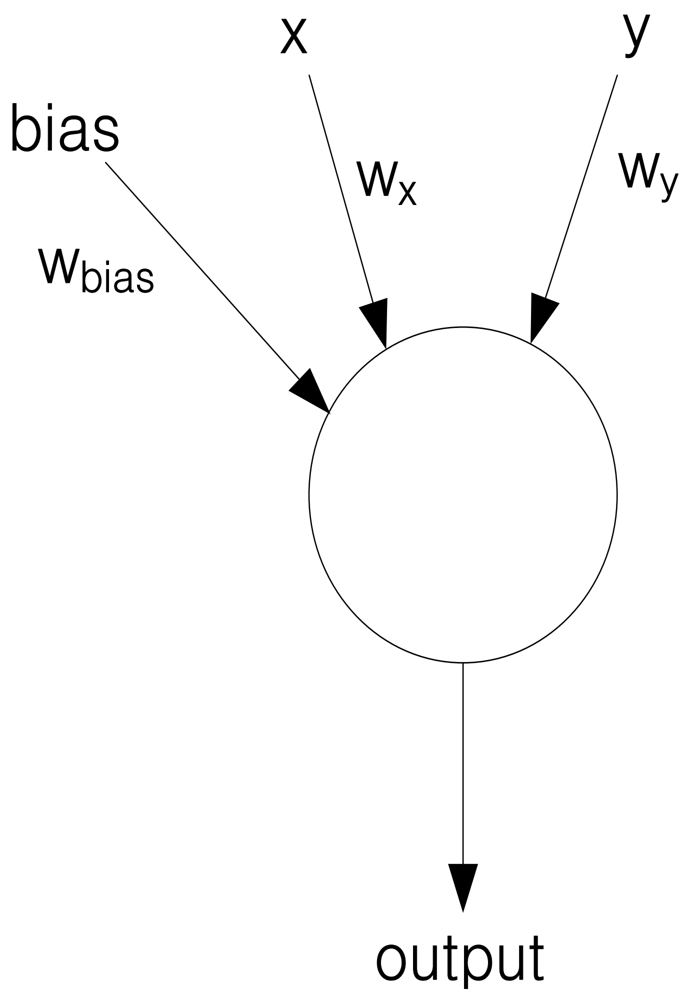
- ✓ Let’s say a perceptron has 2 inputs (the x- and y-coordinates of a point).
- ✓ Using a sign activation function, the output will either be -1 or +1.
- ✓ In the previous diagram, we can see how each point is either below the line (-1) or above (+1).



Bias

✓ Let’s consider the point (0,0).

0 * weight for x = 0
0 * weight for y = 0
1 * weight for bias = weight for bias



Coding the Perceptron

```
class Perceptron
{
private:
    //The Perceptron stores its weights and learning constants.
    std::vector<float>  spWeights;
    int mWeightsSize = 0;
    float c = 0.01; // learning constant
```

Initialize

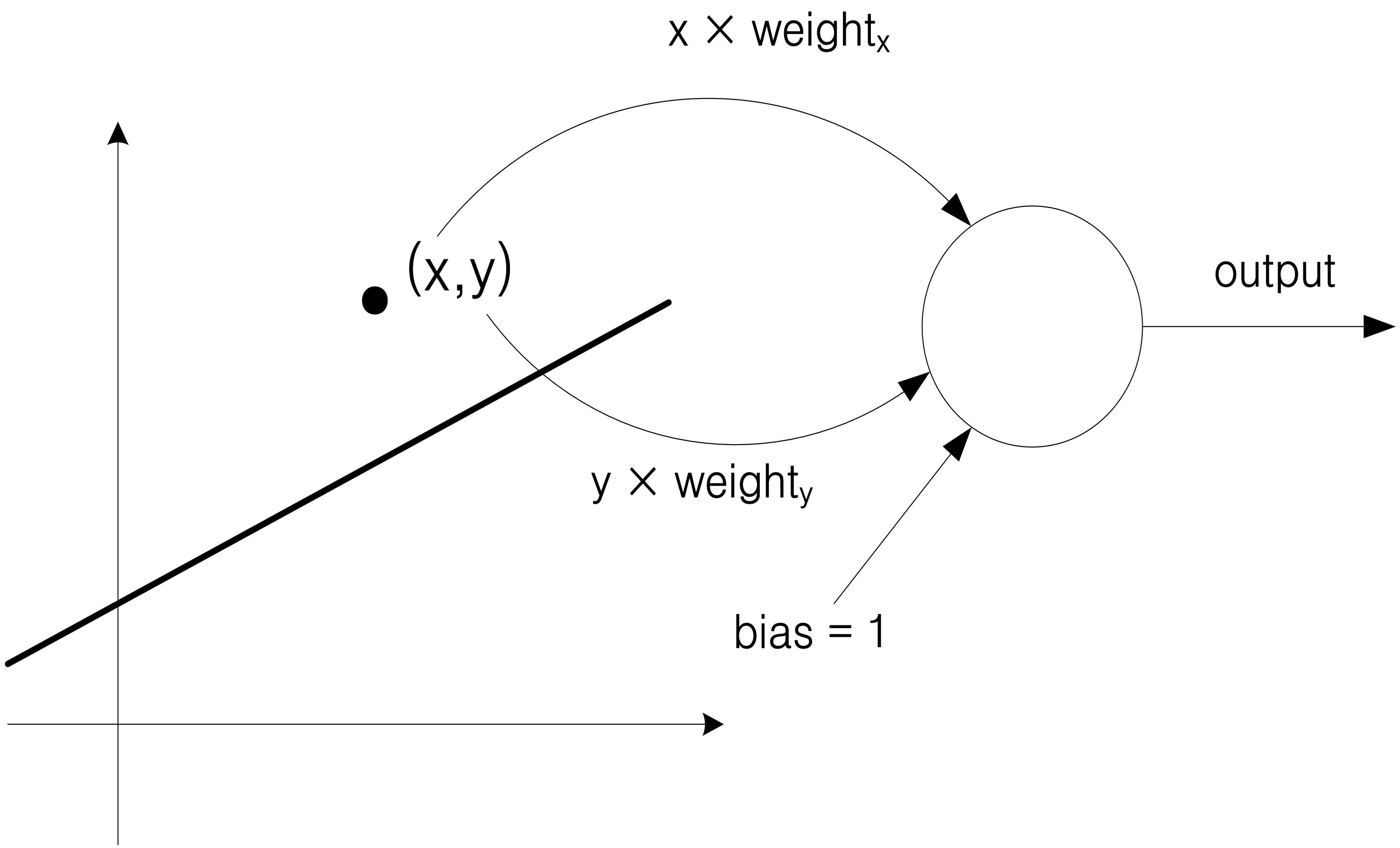
```
Perceptron(int n)
{
    mWeightsSize = n;
    spWeights = new float[ n ];
    //Weights start off random.
    for (int i = 0; i < mWeightsSize; i++) {
        spWeights[ i ] = random( -1, 1 );
    }
}
```

Feed forward

```
int feedforward(float inputs[])
{
    float sum = 0;
    for (int i = 0; i < mWeightsSize; i++) {
        sum += inputs[ i ] * spWeights[ i ];
    }
    return activate(sum);
}

//Output is a +1 or -1.
int activate(float sum)
{
    if (sum > 0) return 1;
    else return -1;
}
```

Use the Perceptron

```
Perceptron p = new Perceptron(3);
float point[] = {50,-12,1}; // The input is 3 values: x,y and bias.
int result = p.feedforward(point);
```



Supervised Learning

① Provide the perceptron with inputs for which **there is a known answer**.

② Ask the perceptron to guess an answer.

③ Compute the error.

④ Adjust all the weights according to the error.

⑤ Return to Step 1 and repeat.

Perceptron Learning Rule

- ✓ Hebb rule(1949)
  - 뉴런 A, B가 반복적이고 지속적으로 점화(firing)하여 어느 한쪽 또는 양쪽 모두에 어떤 변화를 야기한다면 상호간의 점화의 효율은 점점 커지게 된다.

- ✓ Delta Rule(1957)
  - 어떤 뉴런의 활성이 다른 뉴런의 오류에 공헌했다면, 그 사이의 연결가중치를 그것에 비례하여 조절해 주어야 한다.

- ✓ Generalized Rule(1986)
  - (Delta Rule) + 그리고 그 과정은 그 아래의 뉴런들에게까지 계속된다.

- ✓ Competitve Rule(1981)
  - 한 마디의 활성화는 동일원 속에 있는 다른 마디들을 억제시키며, 직접적이든 간접적이든 그 마디와 연계되어 있는 마디들을 활성화 시킨다.

The Perceptron's error

ERROR = DESIRED - GUESS

Desired	Guess	Error
-1	-1	0
-1	1	-2
1	-1	2
1	1	0

✓ The error is the determining factor in how the perceptron's weights should be adjusted

- ✓ For any given weight, what we are looking to calculate is the change in weight, often called *Δweight*.

  NEW WEIGHT = WEIGHT + ΔWEIGHT

- ✓ Δweight is calculated as the error multiplied by the input.

  ΔWEIGHT = ERROR × INPUT

- ✓ Therefore:

  NEW WEIGHT = WEIGHT + ERROR × INPUT

Learning Rate

NEW WEIGHT = WEIGHT + ERROR * INPUT * LEARNING CONSTANT

- ✓ With a small learning constant, the weights will be adjusted slowly, requiring more training time but allowing the network to make very small adjustments that could improve the network's overall accuracy.

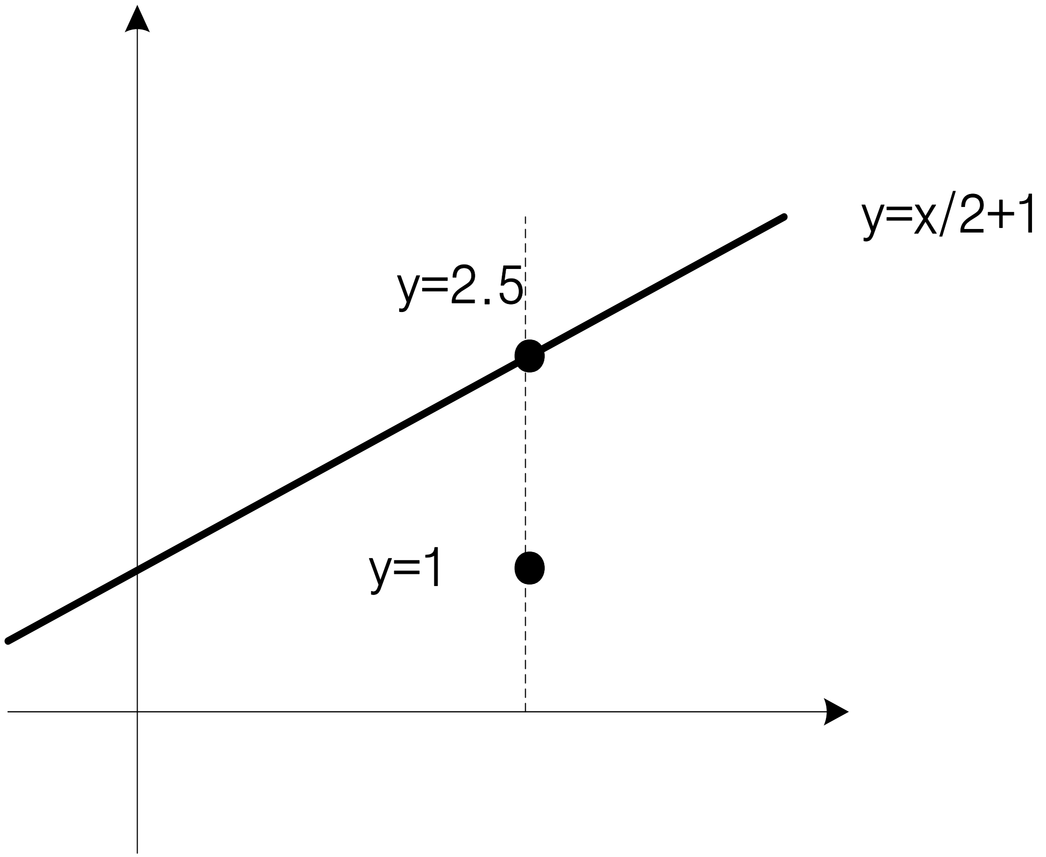
```
 float c = 0.01; // learning constant
//Train the network against known data.
void train(float inputs[], int desired) {
    int guess = feedforward(inputs);
    float error = desired - guess;
    for (int i = 0; i < mWeightsSize; i++) {
        spWeights[ i ] += c * error * inputs[ i ];
    }
}
```

Full source of class Perceptron

Trainer

```
class Trainer
{
public:
    //A "Trainer" object stores the inputs and the correct answer.
    float mInputs[ 3 ];
    int mAnswer;

    void SetData( float x, float y, int a )
    {
        mInputs[ 0 ] = x;
        mInputs[ 1 ] = y;
        //Note that the Trainer has the bias input built into its array.
        mInputs[ 2 ] = 1;
        mAnswer = a;
    }
```

```
//The formula for a line
float f( float x )
{
    return x/2 + 1;
}
```

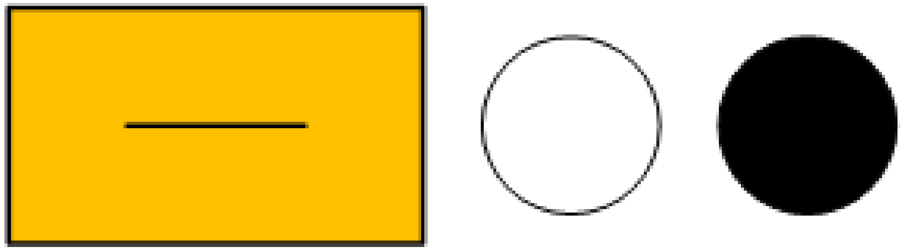


```
void Setup()
{
    srand( time( 0 ) );
    //size( 640, 360 );
    spPerceptron.reset( new Perceptron( 3 ) );

    // Make 2,000 training points.
    for( int i = 0; i < gTrainerSize; i++ ) {
        float x = random( -gWidth / 2, gWidth / 2 );
        float y = random( -gHeight / 2, gHeight / 2 );
        //Is the correct answer 1 or - 1 ?
        int answer = 1;
        if( y < f( x ) ) answer = -1;
        gTraining[ i ].SetData( x, y, answer );
    }
}
```

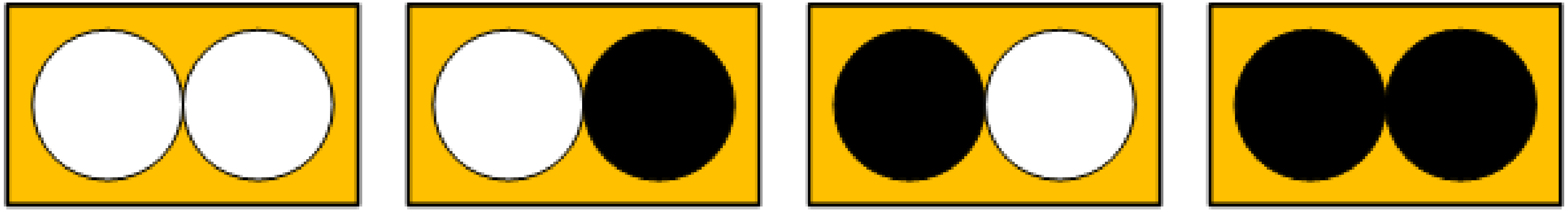
```
void Training()
{
    for( int i = 0; i < gTrainerSize; i++ ) {
        spPerceptron->train( gTraining[ i ].mInputs, gTraining[ i
].mAnswer );
    }
}

void main()
{
    Setup();
    Training();
}
```

Full source of “NeuralNetwork Perceptron”

Second Example: Computer Go



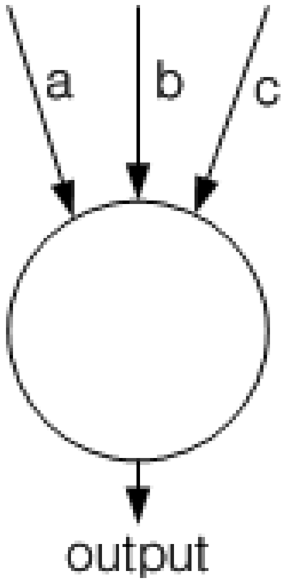
요즘 알파고와 이세돌의 바둑대국이 화제거리입니다. **머신 러닝machine learning** 혹은 **신경망neural network**이라는 용어들이 나오는데, 가장 간단한 바둑판 1×2를 이용해서 설명해 보겠습니다^^



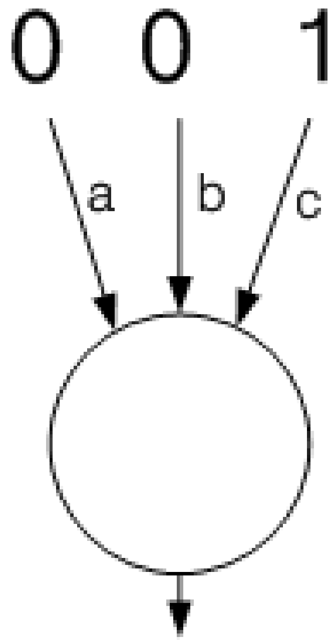
+1	+1	+1	−1

위의 그림은 모든 바둑판 상황입니다. 흰돌이 우세하면 +1, 검은 돌이 우세하면 -1이라고 가정을 합니다. 세번째 경우가 +1인 것은 임의로 정한 것입니다. 흰돌은 숫자 0, 검은돌을 숫자 1로 나타냅니다. 그리고 내부적인 오류를 막기 위해 가장 마지막에 항상 1을 추가합니다. 그러면 바둑판은 다음과 같은 0과 1의 조합으로 나타낼 수 있습니다.

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



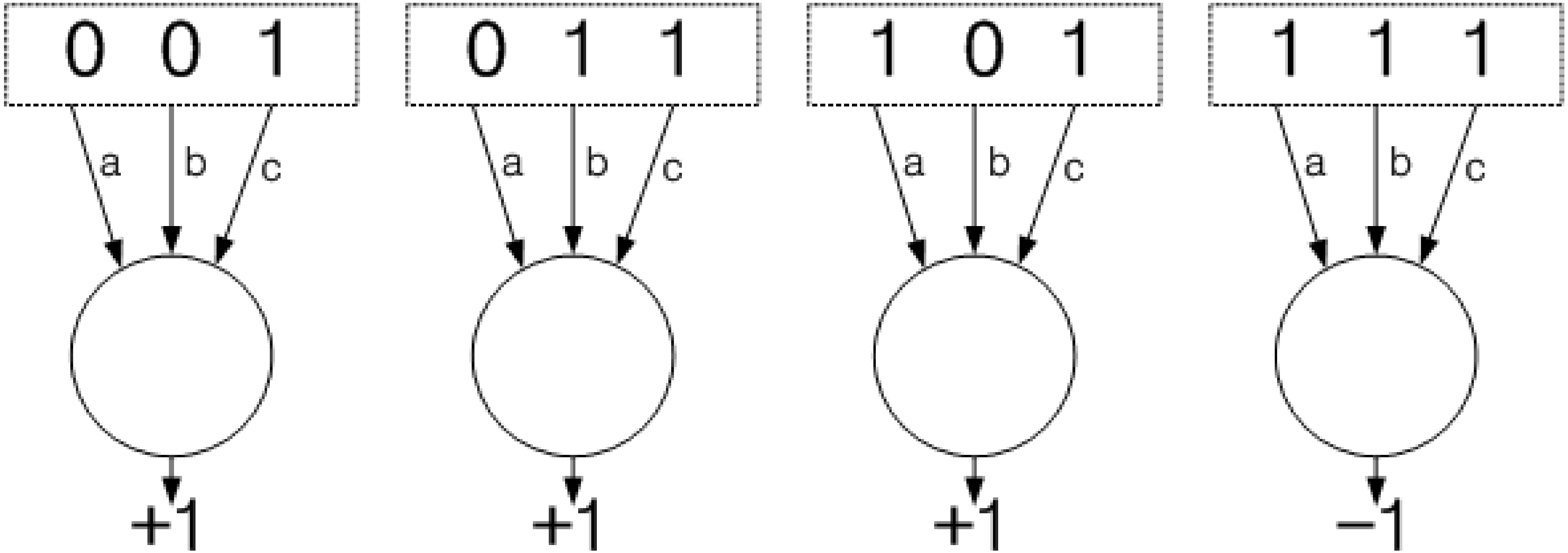
이제 문제는 입력 001, 011, 101에 대해서는 출력 +1, 입력 111에 대해서는 출력 -1을 가지도록 무언가를 만들어주는 문제가 됩니다.
이 문제를 해결하기 위해 왼쪽 그림과 같은 간단한 망network를 디자인 할 수 있습니다.



예를 들면, 001에 대해서 생각해 보면, 식을 다음과 같이 구성합니다.

$0 \times a + 0 \times b + 1 \times c = +1$

즉, 위의 식을 만족하도록 a,b와 c의 **가중치weight 값**을 정하는 것입니다. 하지만 최종 값은 모든 입력에 대해서 결과를 만족하도록 a,b와 c를 정해야 합니다.

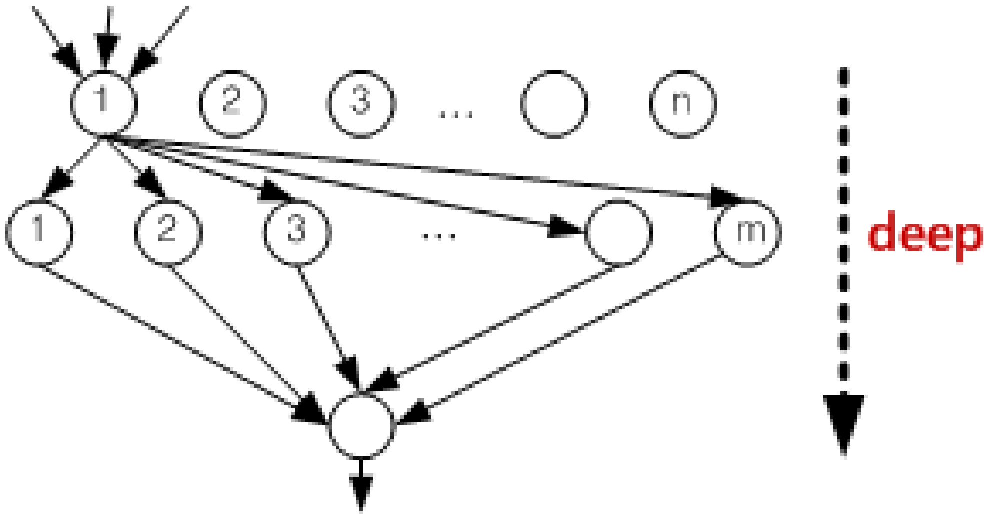


a,b와c를 어떻게 정할 수 있을까요? 위와 같은 구성에서 동그라미에 해당하는 것이 **뇌세포neuron** 하나에 해당합니다. 그리고 이 뇌세포의 출력은 다른 뇌세포의 입력으로 전달됩니다.
그리고 a,b와 c를 구하는 과정을 **기계 학습machine learning**이라고 합니다.

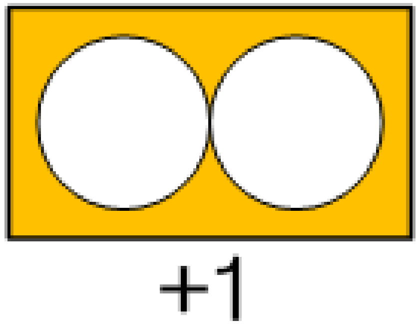
a,b와c를 구하는 과정을 살펴보기에 앞서 위와 같은 구성을 좀 더 **깊게deep** 구성할 수 있겠지요?
그러면 다음 그림과 같은 모양이 됩니다.

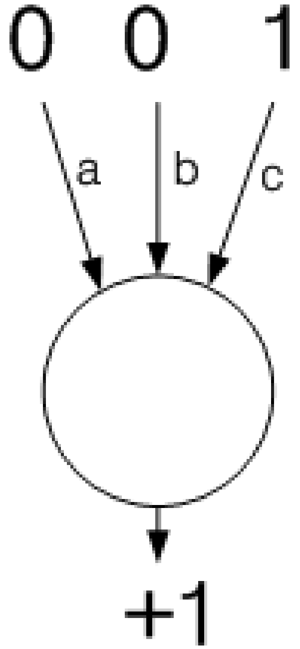
a,b와c를 어떻게 정할 수 있을까요? 위와 같은 구성에서 동그라미에 해당하는 것이 **뇌세포neuron** 하나에 해당합니다. 그리고 이 뇌세포의 출력은 다른 뇌세포의 입력으로 전달됩니다. 그리고 a,b와 c를 구하는 과정을 **기계 학습machine learning**이라고 합니다.

a,b와c를 구하는 과정을 살펴보기에 앞서 위와 같은 구성을 좀 더 **깊게deep** 구성할 수 있겠지요? 그러면 다음 그림과 같은 모양이 됩니다.



왼쪽 그림에서는 모든 뉴런neuron간의 연결을 그리지 않았습니다. (1)번 neuron처럼 다른 모든 neuron도 아래 **층layer**과 연결되어 있습니다.
이런 구성에서도 시간이 걸리기는 하겠지만, neuron간의 연결에 할당된 **가중치weight**를 구할 수 있습니다.
이것이 알파고AlphaGo의 **딥러닝deep learning** 구현 원리입니다(실제는 더 복잡합니다).





입력 001에 대해서 a,b와c의 값을 **처음에는 랜덤random하게 0.3, -0.2와 0.4라고 정해** 보겠습니다. 기사를 읽다가 **몬테카를로**라는 말이 나오면, 난수를 사용하는 방법을 말합니다. 몬테카를로에 도박장이 많았던 것 같습니다^^ 그러면 가중치의 합은 다음과 같습니다.

0×0.3+0×(-0.2)+1×0.4=0.4
0 + 0 + 0.4 = 0.4

결과가 0보다 크면 +1, 0보다 작으면 -1로 **결정decision**하도록 하겠습니다. 이 함수를 **결정함수**라고 합니다.

그러면 결과 0.4는 0보다 크므로, **뉴런의 출력은 +1**입니다.

0	0	1
0.3	-0.2	0.4
0	0	0.4
		0.4
	1	1
0.3	-0.2	0.4

위의 과정을 왼쪽 표와 같이 나타냅니다.
a,b와c의 값을 처음에는 랜덤random하게 0.3, -0.2와 0.4라고 정했습니다. 가중치의 합은 다음과 같습니다.

$0 \times 0.3 + 0 \times (-0.2) + 1 \times 0.4 = 0.4$
$0 + 0 + 0.4 = 0.4$

결과가 0보다 크면 +1, 0보다 작으면 -1로 결정decision하도록 하겠습니다.

그러면 결과 0.4는 0보다 크므로, **뉴런의 출력은 +1**입니다.

예상되는 결과와 실제값의 차이를 **오차error**라고 합니다. 이 예에서 **오차는 0**입니다.

이제 가중치를 update하기 위해서 오차를 고려해서 가중치 a를 다음과 같이 구합니다.

$a = a + input \times error$

$0.3 = 0.3 + 0 \times 0$

a = 0.3

그러면 나머지 b와 c도 같은 방법으로 구합니다.

$b = b + input \times error$

$-0.2 = -0.2 + 0 \times 0$

b = -0.2

$c = c + input \times error$

$0.4 = 0.4 + 1 \times 0$

c = 0.4

첫번째 학습에서는 가중치가 변하지 않았습니다.

0	0	1
0.3	-0.2	0.4
0	0	0.4
		0.4
	1	1
0.3	-0.2	0.4

이제 새롭게 구한 가중치 a,b와c즉 0.3, -0.2와 0.4를 **다음 학습 011**을 위해 사용합니다.

0	0	1
0.3	-0.2	0.4
0	0	0.4
		0.4
	1	1
0.3	-0.2	0.4

0	1	1
0.3	-0.2	0.4
0	-0.2	0.4
		0.2
	1	1
0.3	-0.2	0.4

최종 결과! 두둥!!

1	1	1
-3.7	-2.2	4.4
-3.7	-2.2	4.4
		-1.5
	-1	-1
-3.7	-2.2	4.4

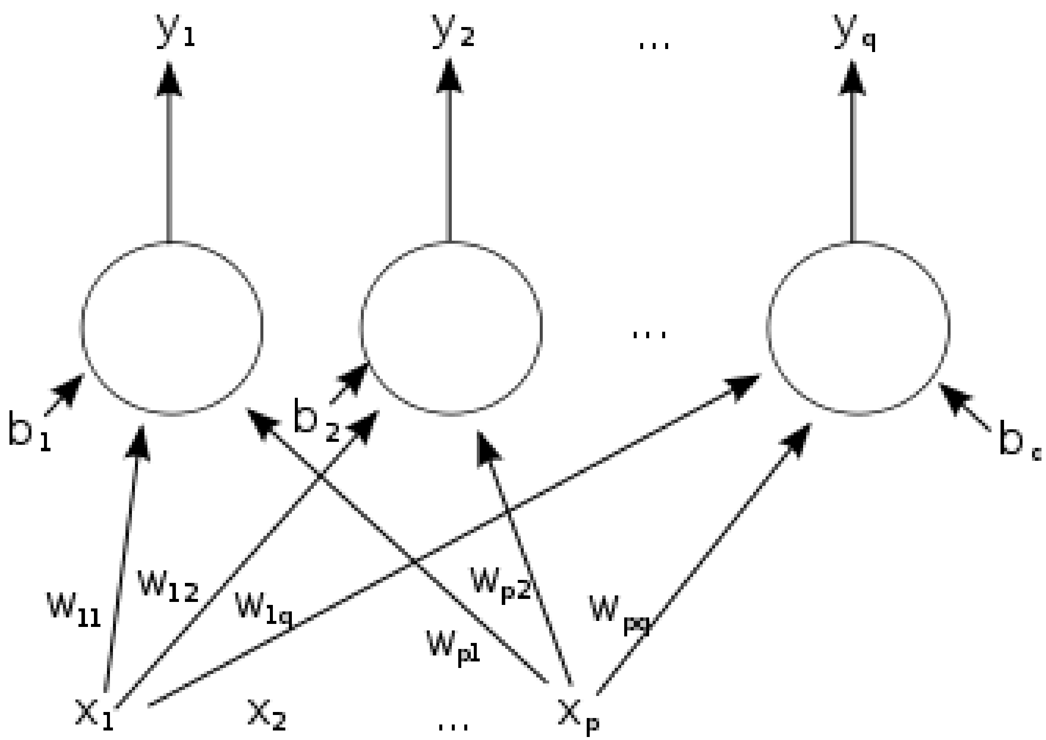
이 과정을 계속 반복하다보면, 언젠가는 **가중치가 변하지 않는 때**가 오겠지요? 그 순간이 기계학습을 마친 순간입니다. 지금까지 설명은 신경망중에서 가장 간단한 **Perceptron**에 대한 설명이었습니다~

연습Practice

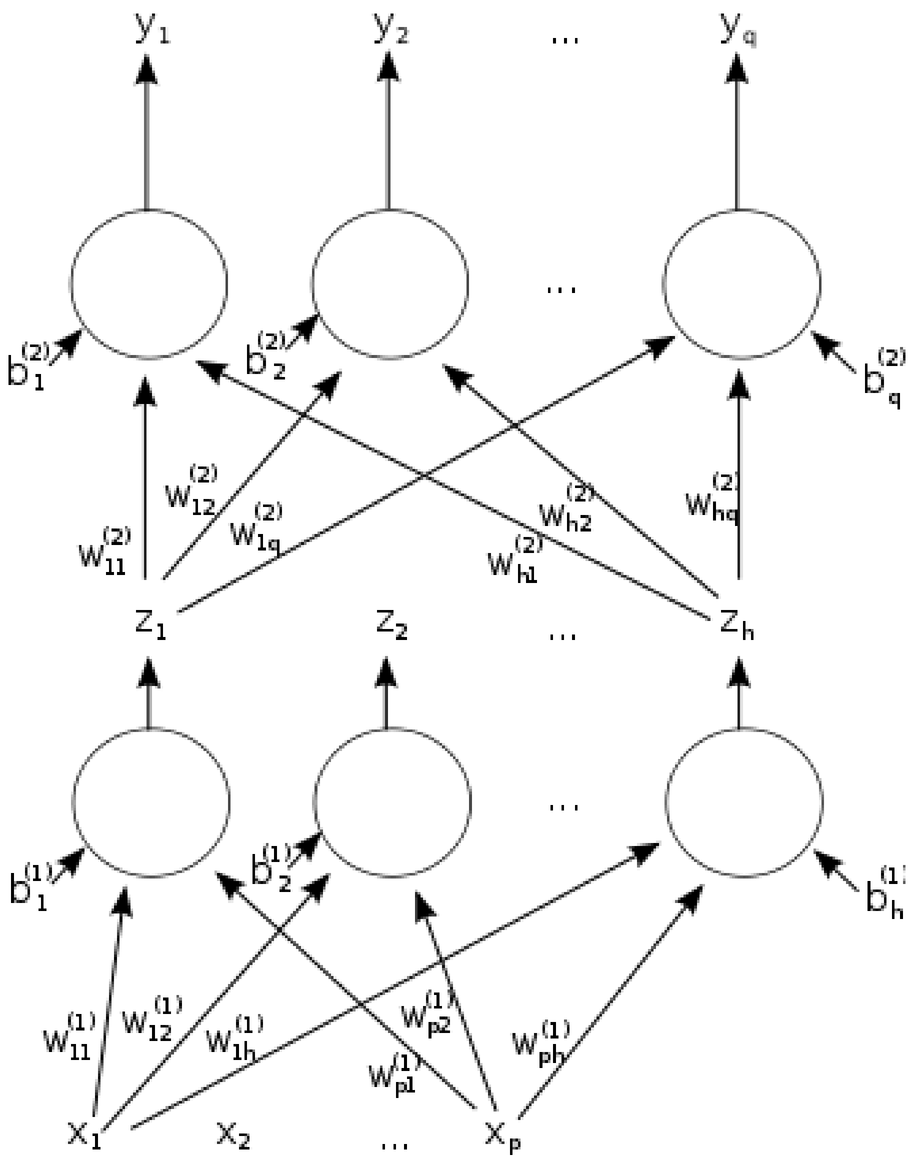
- ✓ Calculate weights of the Perceptron for Go by using paper and pencil.

Neural Network

✓ Single Layer

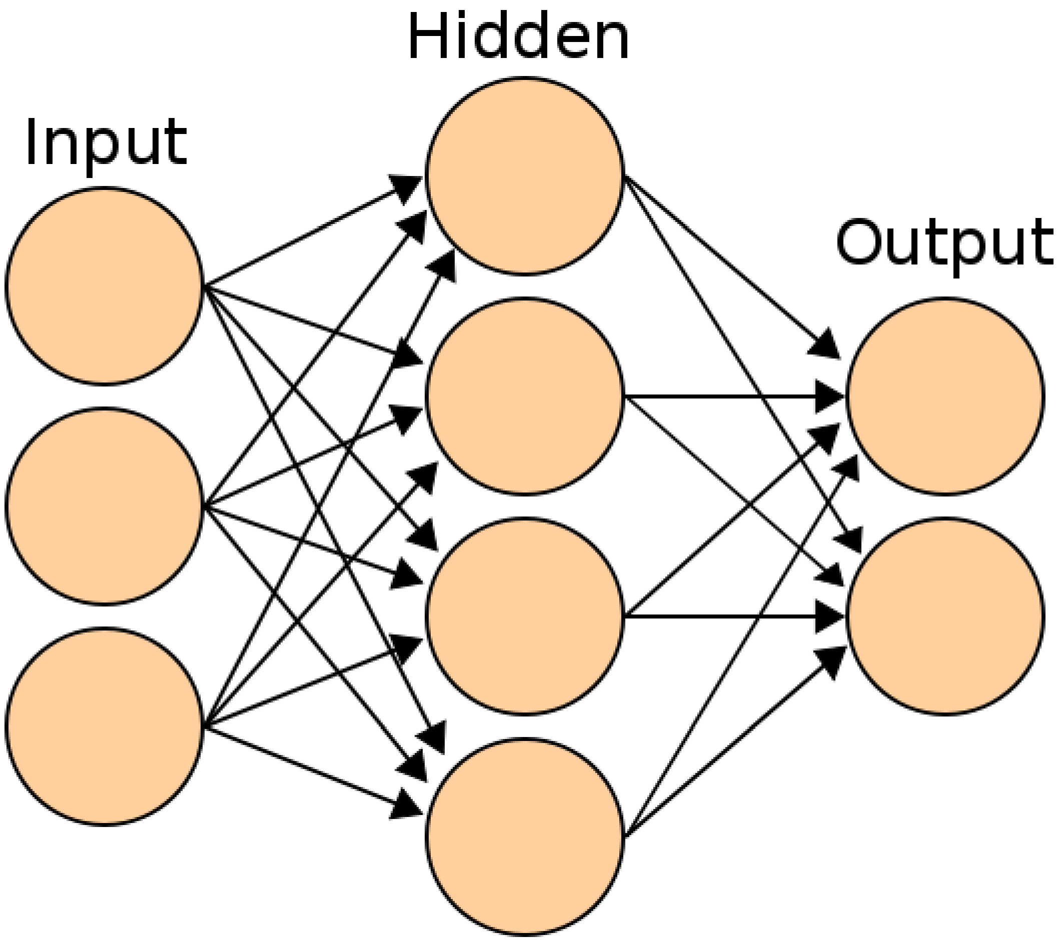


✓ Two Layers

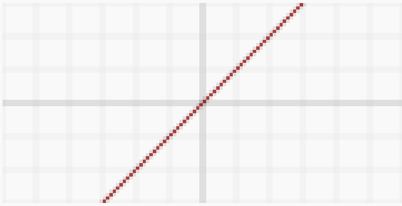
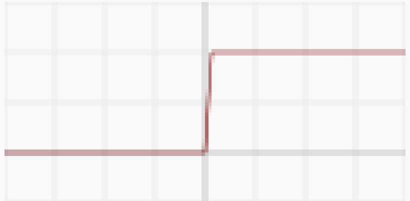
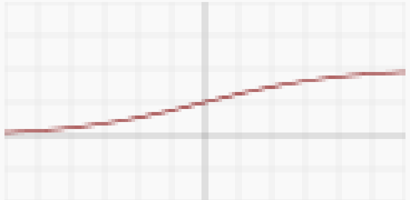
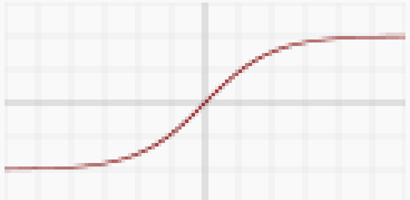


y1
y2
...
yq
b1(2)
b2(2)
bq(2)
w11(2)
w12(2)
w1q(2)
wh2(2)
wh1(2)
whq(2)
z1
z2
...
zh
b1(1)
b2(1)
bh(1)
w11(1)
w12(1)
w1h(1)
wp2(1)
wp1(1)
wph(1)
x1
x2
...
xp

Neural Network



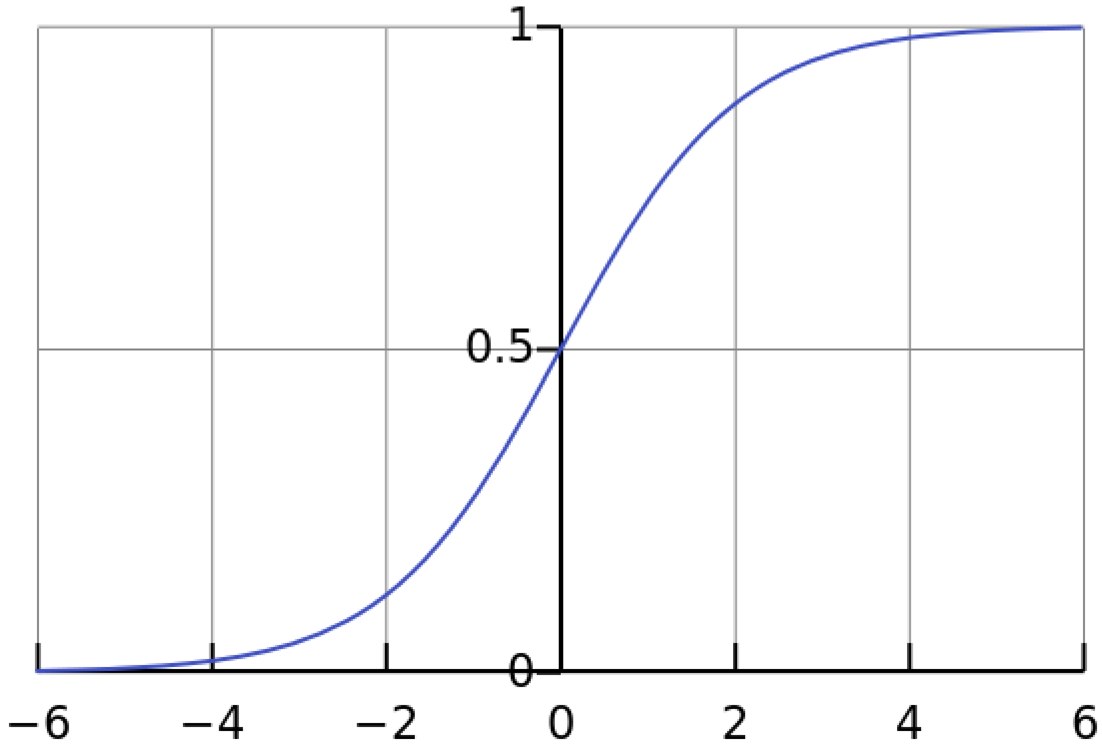
Activation Functions

Name	Plot	Equation
Identity		$f(x) = x$
Binary step		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \dfrac{1}{1 + e^{-x}}$
TanH		$f(x) = \tanh(x) = \dfrac{2}{1 + e^{-2x}} - 1$

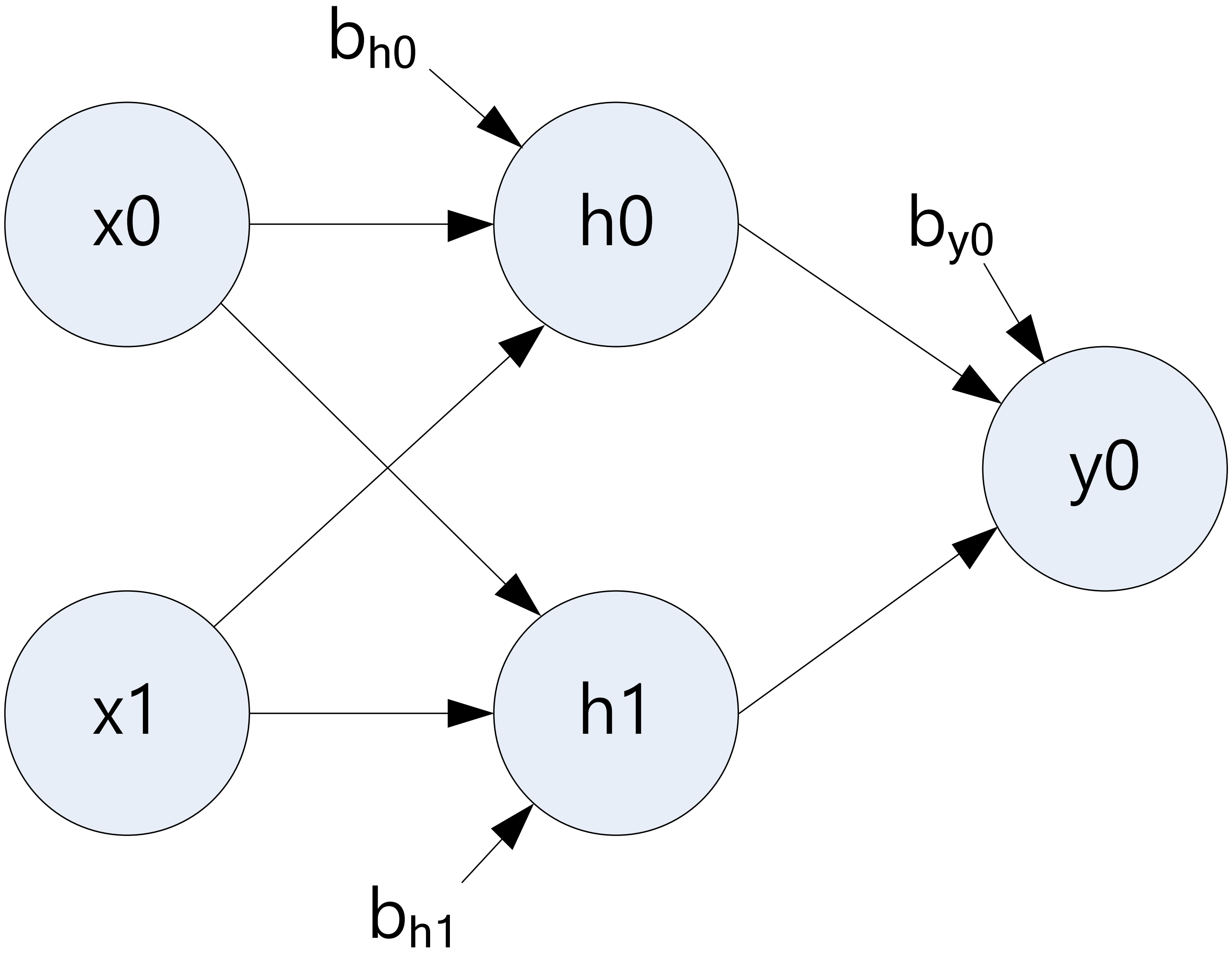
Sigmoid Function

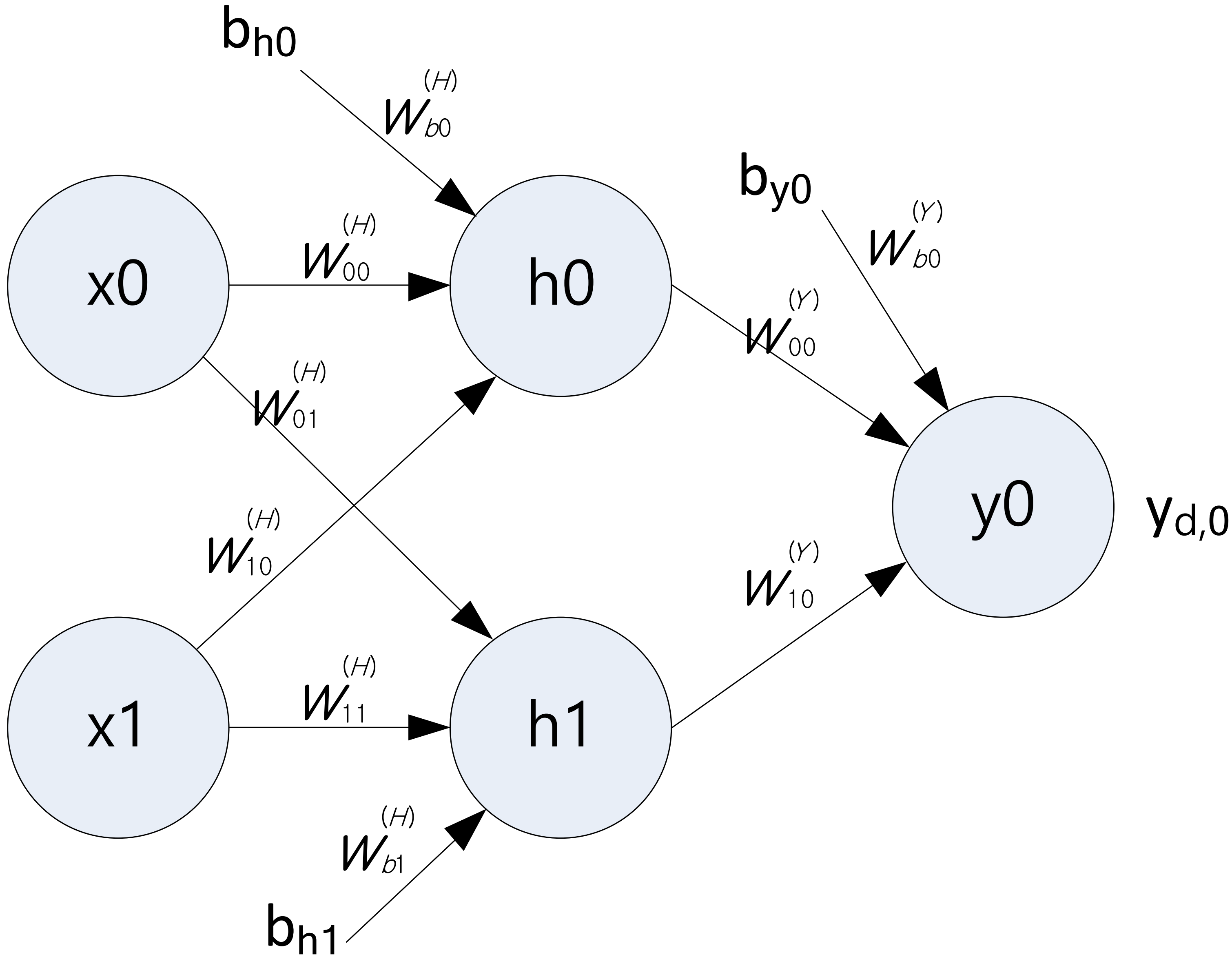
- ✓ A **sigmoid function** is a mathematical function having an "S" shape (**sigmoid curve**).
- ✓ A wide variety of sigmoid functions have been used as the activation function of artificial neurons.

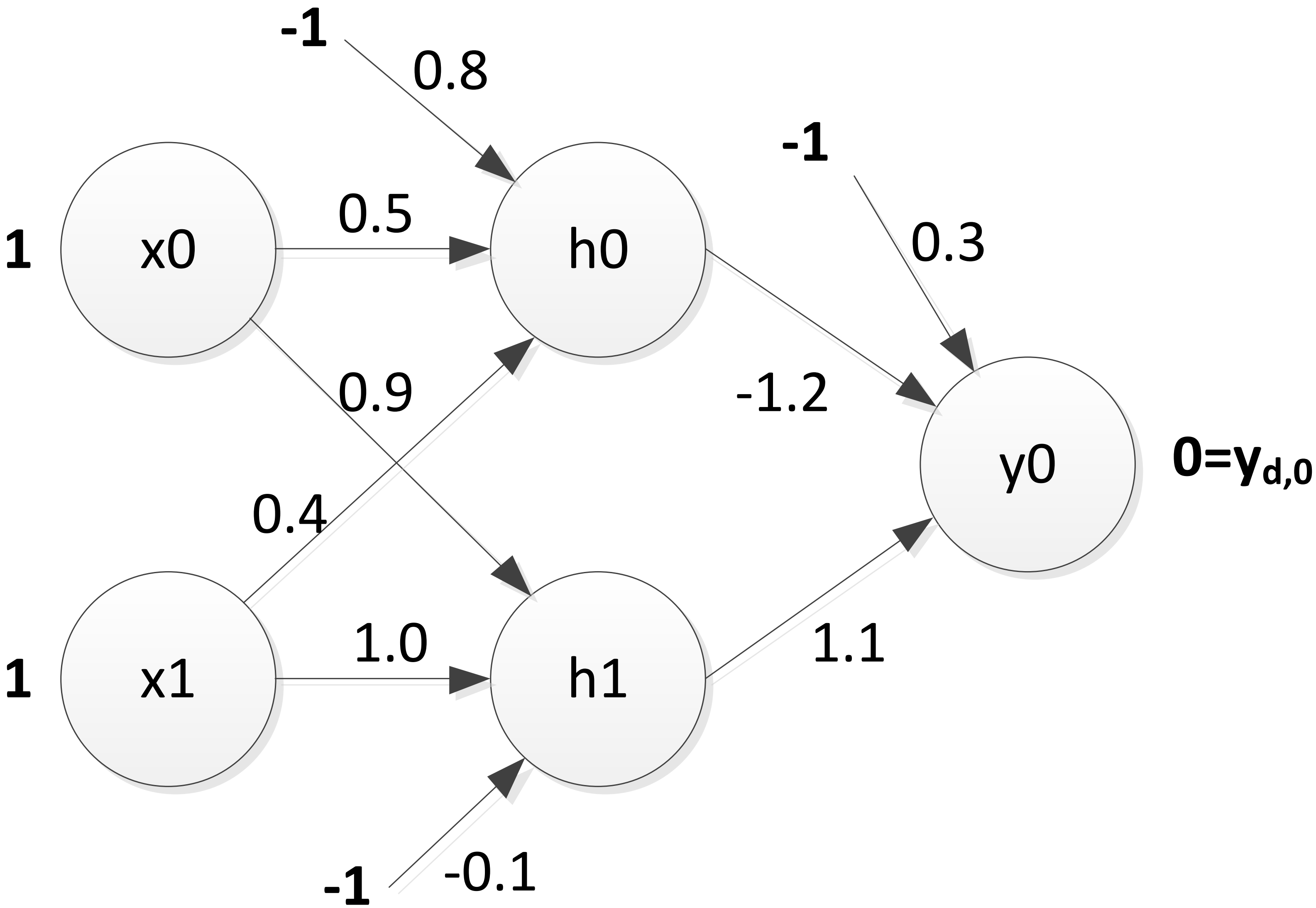
$$S(t) = \frac{1}{1+e^{-t}}.$$



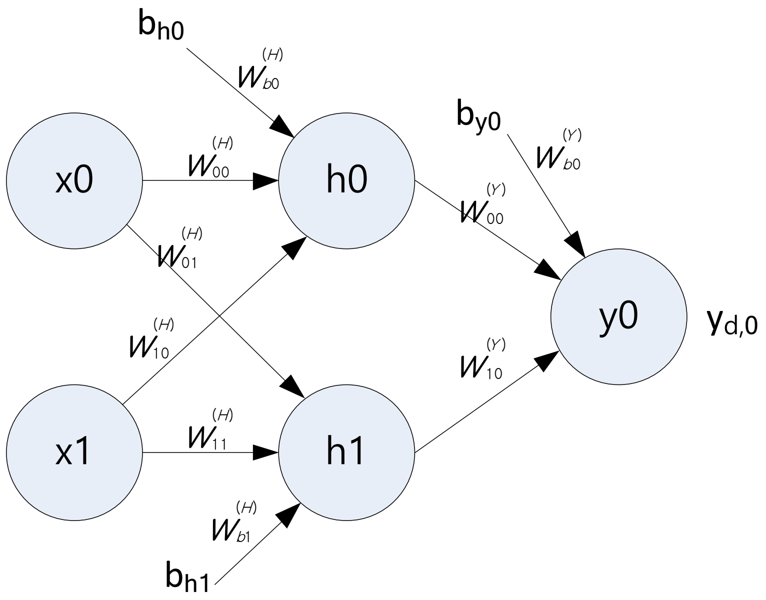
Backpropagation Example

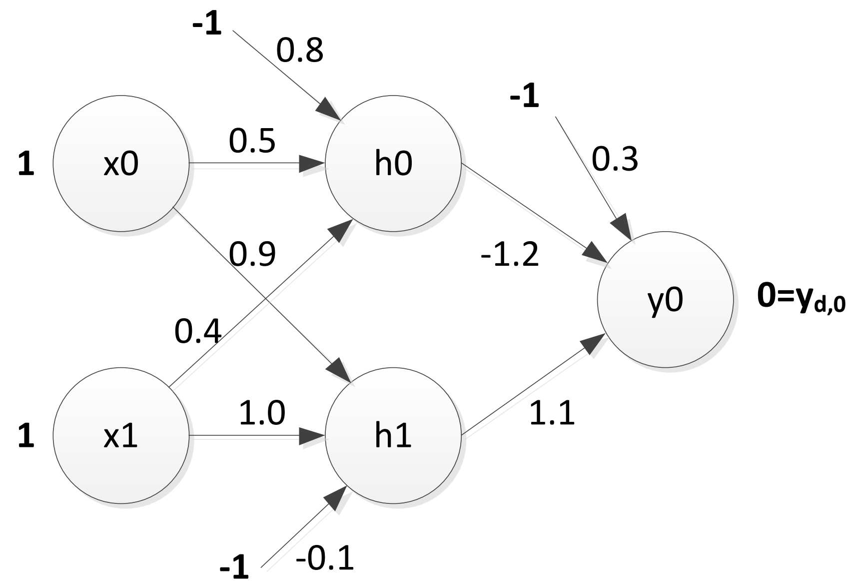



b_{h0}
$W_{b0}^{(H)}$
x0
$W_{00}^{(H)}$
h0
b_{y0}
$W_{b0}^{(Y)}$
$W_{00}^{(Y)}$
$W_{01}^{(H)}$
y0
$y_{d,0}$
$W_{10}^{(H)}$
$W_{10}^{(Y)}$
x1
$W_{11}^{(H)}$
h1
$W_{b1}^{(H)}$
b_{h1}

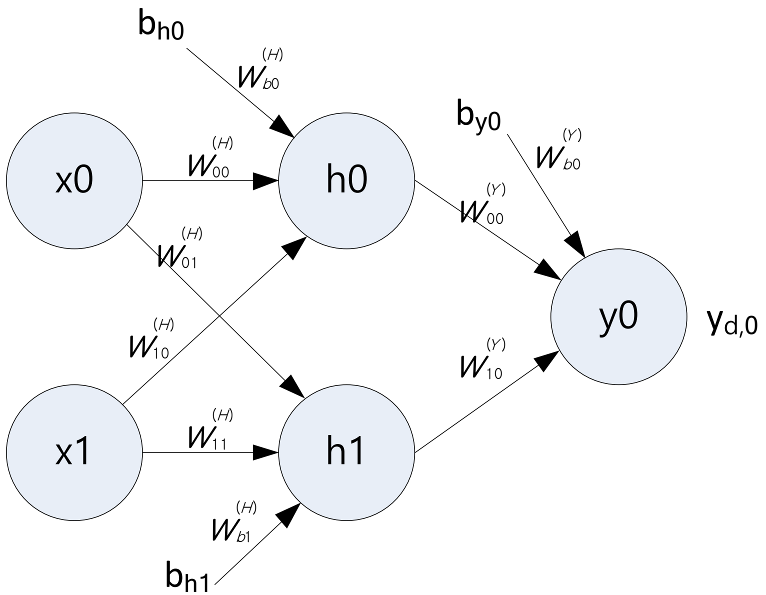

-1
0.8
-1
1
x0
0.5
h0
0.3
0.9
-1.2
y0
0=y_{d,0}
0.4
1.0
1.1
1
x1
h1
-1
-0.1

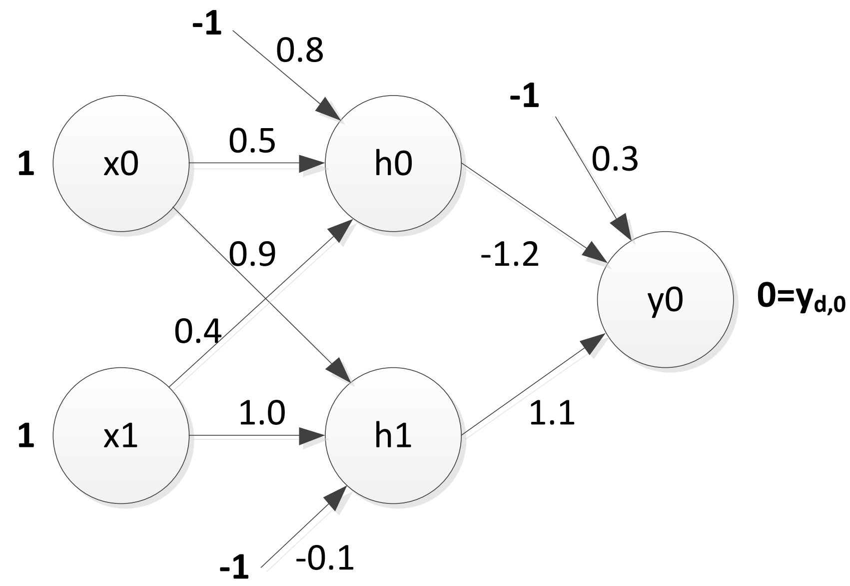
Feed Forward



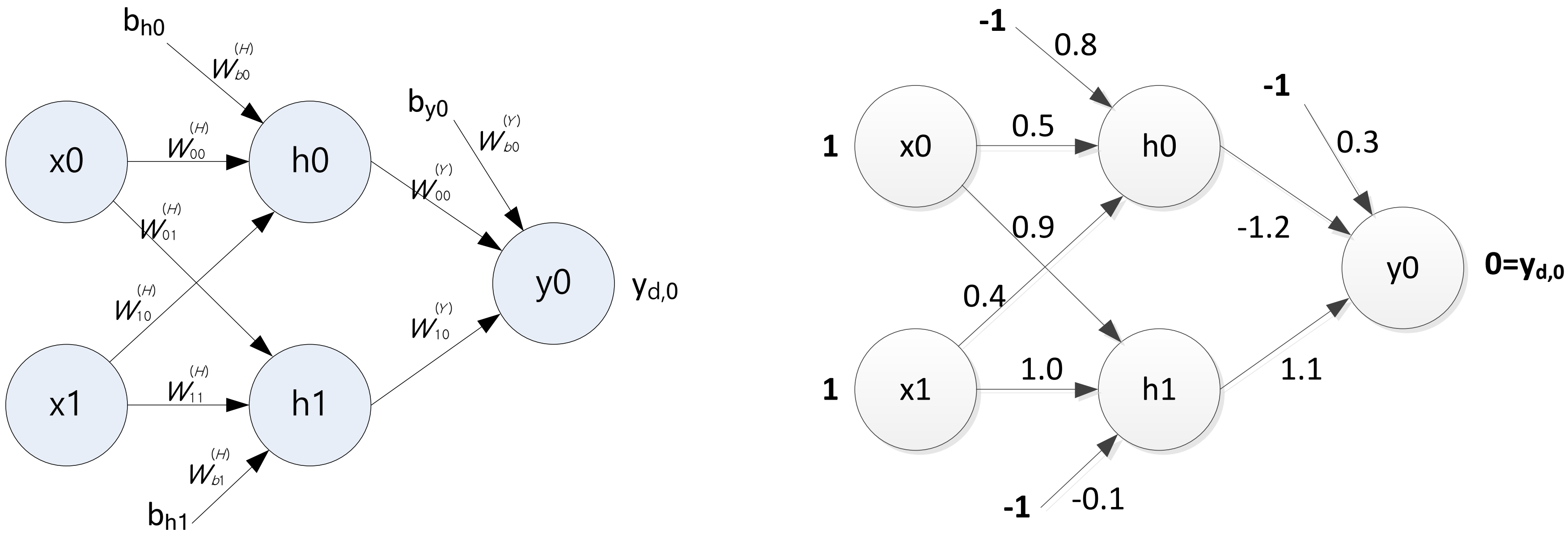


$$
\begin{aligned}
h0 &= sigmoid(x_0 w_{00}^{(H)} + x_1 w_{10}^{(H)} + b_{h0} w_{b0}^{(H)}) \\
&= 1/(1+e^{-(x_0 w_{00}^{(H)} + x_1 w_{10}^{(H)} + b_{h0} w_{b0}^{(H)})}) \\
&= 1/(1+e^{-(1\times 0.5 + 1\times 0.4 + (-1)\times 0.8)}) \\
&= 0.5250
\end{aligned}
$$





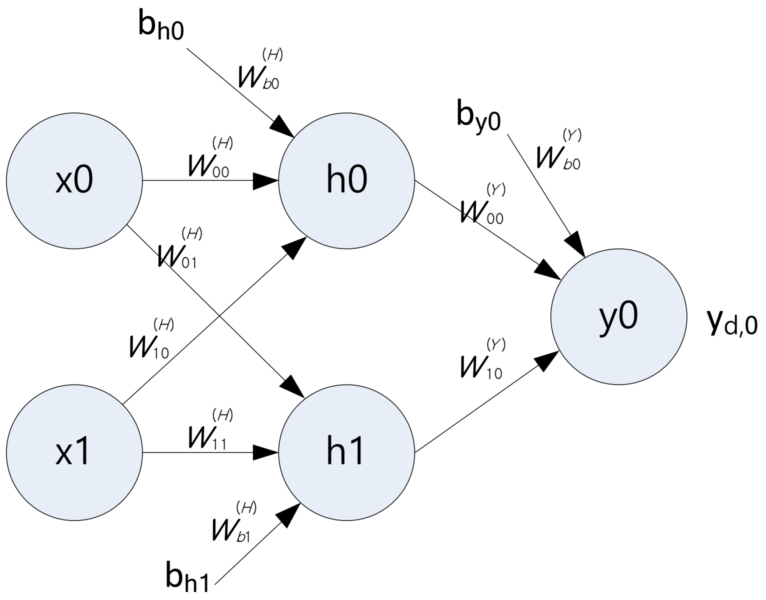
$$
\begin{aligned}
h1 &= sigmoid(x_0 w_{01}^{(H)} + x_1 w_{11}^{(H)} + b_{h1} w_{b1}^{(H)}) \\
&= 1/(1+e^{-(x_0 w_{01}^{(H)} + x_1 w_{11}^{(H)} + b_{h1} w_{b1}^{(H)})}) \\
&= 1/(1+e^{-(1\times 0.9 + 1\times 1.0 + (-1)\times(-0.1))}) \\
&= 0.8808
\end{aligned}
$$

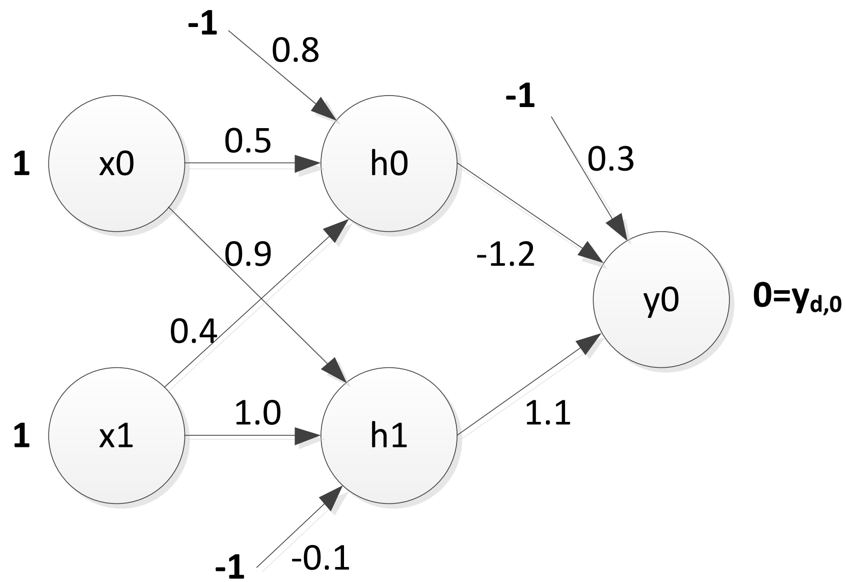


$$y0 = sigmoid(h_0 w_{00}^{(Y)} + h_1 w_{10}^{(Y)} + b_{y0} w_{b0}^{(Y)})$$
$$= 1/(1 + e^{-(0.5250 \times (-1.2) + 0.8808 \times 1.1 + (-1) \times 0.3)})$$
$$= 0.5097$$

$$e = y_{d,0} - y_0 = 0 - 0.5097 = -0.5097$$

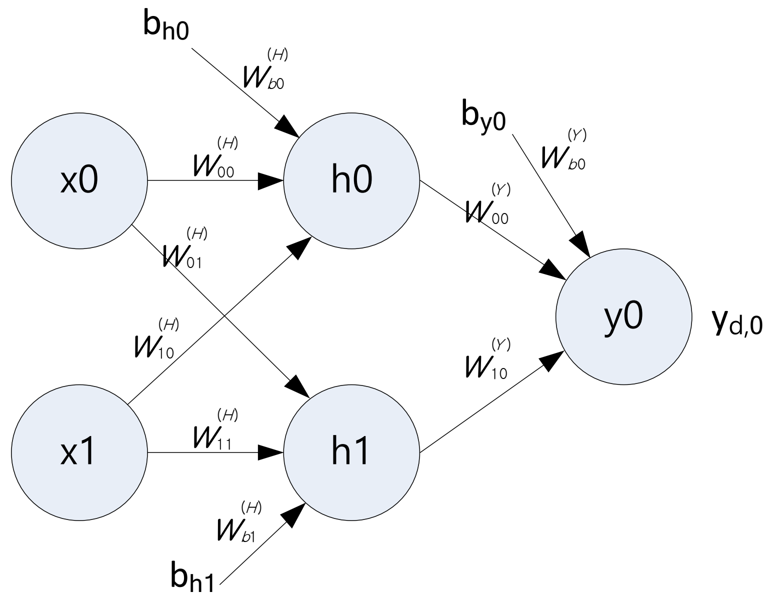
Back Propagation

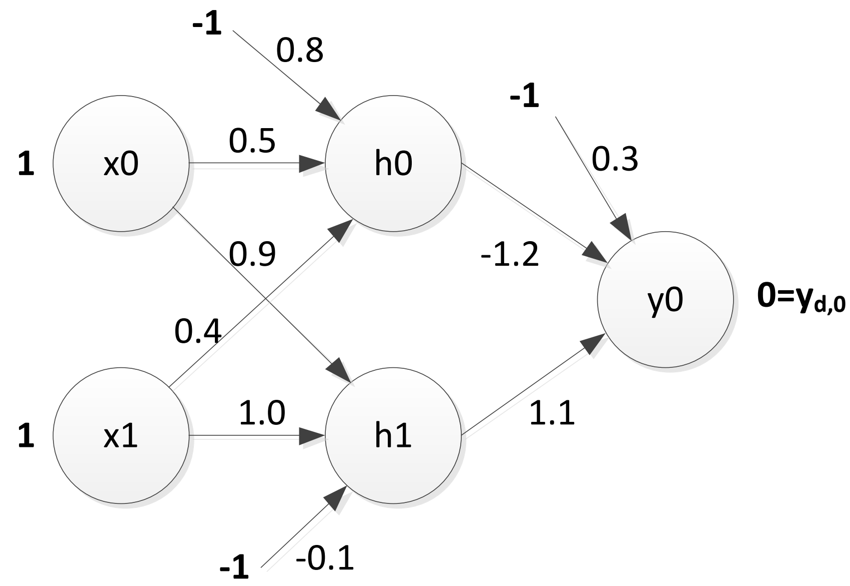




error gradient

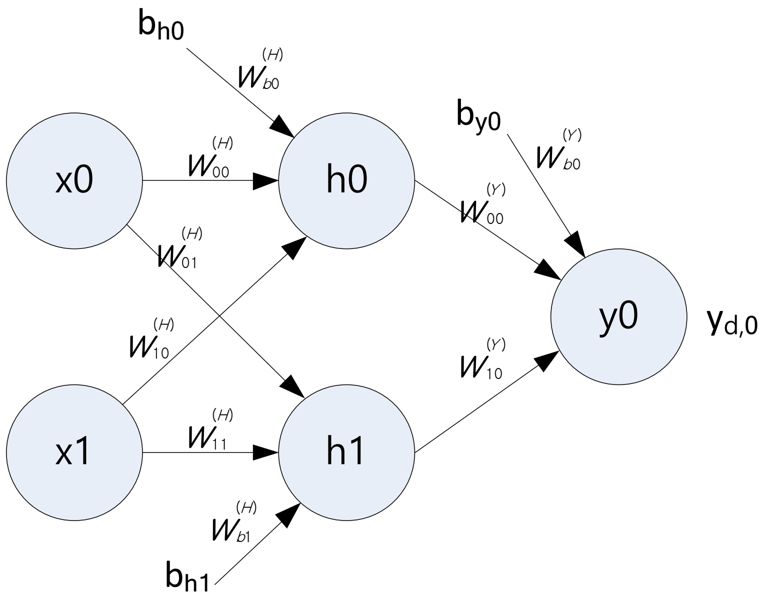
$$\delta^Y_0 = y_0(1-y_0)e$$
$$= 0.5097 \times (1-0.5097) \times (-0.5097)$$
$$= -0.1274$$

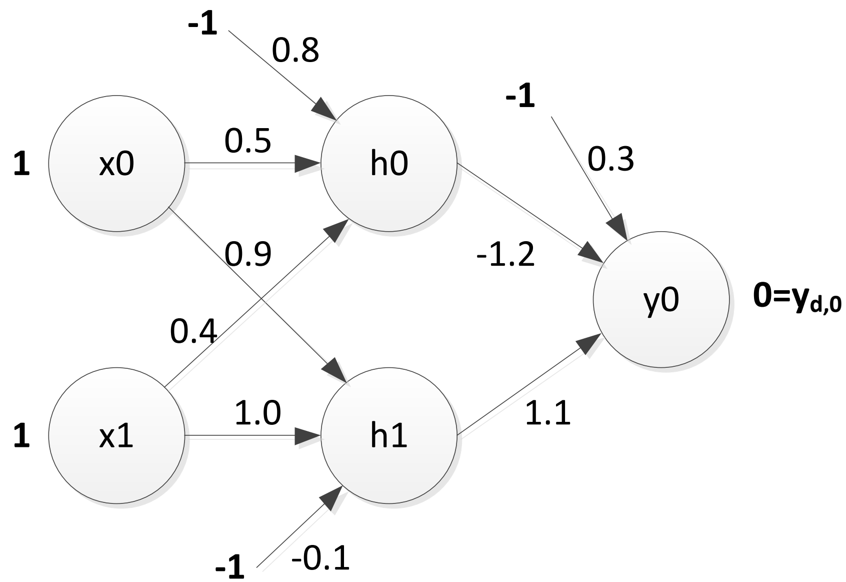




learning ratio $\alpha = 0.1$

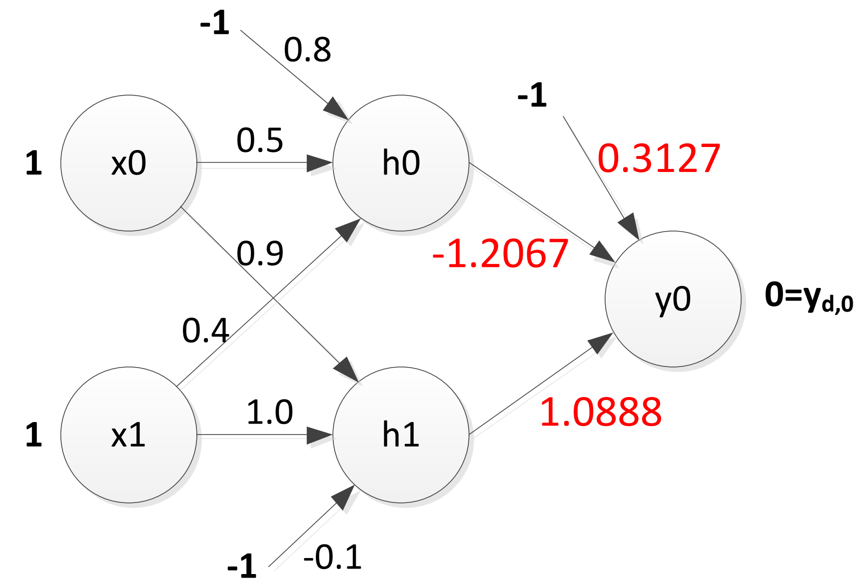
$$\begin{aligned}\Delta w^{Y}_{00} &= \alpha \times h_0 \times \delta_0 \\ &= 0.1 \times 0.5250 \times (-0.1274) \\ &= -0.0067\end{aligned}$$





$$\Delta w_{10}^{Y} = \alpha \times h_1 \times \delta_0 = 0.1 \times 0.8808 \times (-0.1274) = -0.0112$$

$$\Delta w_{b0}^{Y} = \alpha \times b_{y0} \times \delta_0 = 0.1 \times (-1) \times (-0.1274) = -0.0127$$



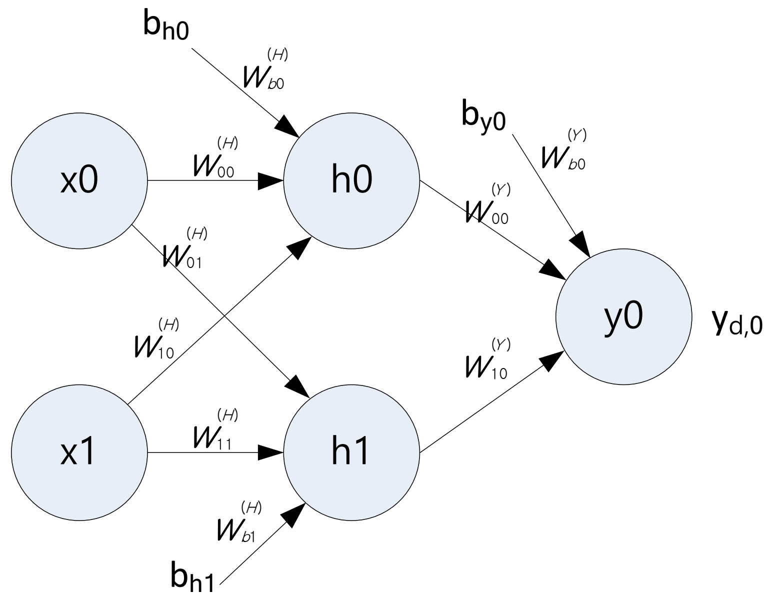
Update Output Weights

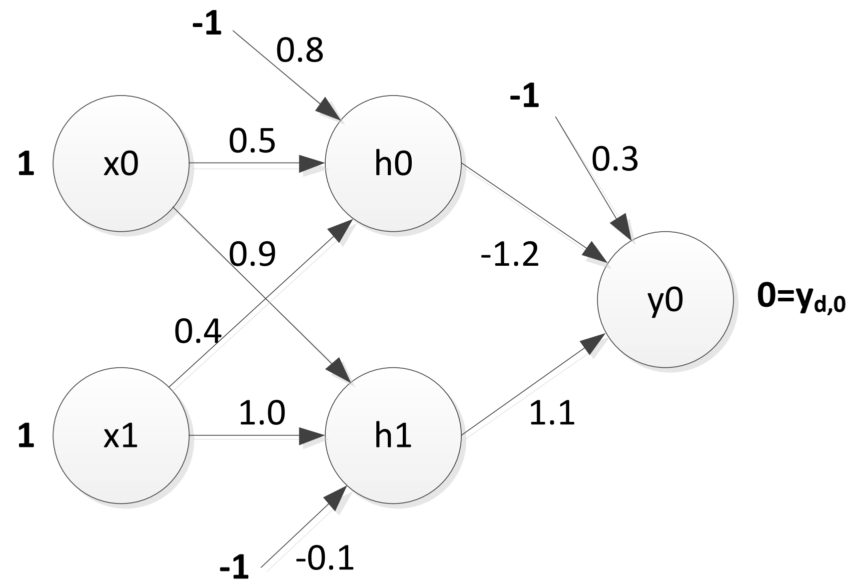
$$w_{00}^{Y} = w_{00}^{Y} + \Delta w_{00}^{Y} = -1.2 + (-0.0067) = -1.2067$$

$$w_{10}^{Y} = w_{10}^{Y} + \Delta w_{10}^{Y} = -1.1 + (-0.0112) = 1.0888$$

$$w_{b0}^{Y} = w_{b0}^{Y} + \Delta w_{b0}^{Y} = 0.3 + (0.0127) = 0.3127$$

Hidden Layer

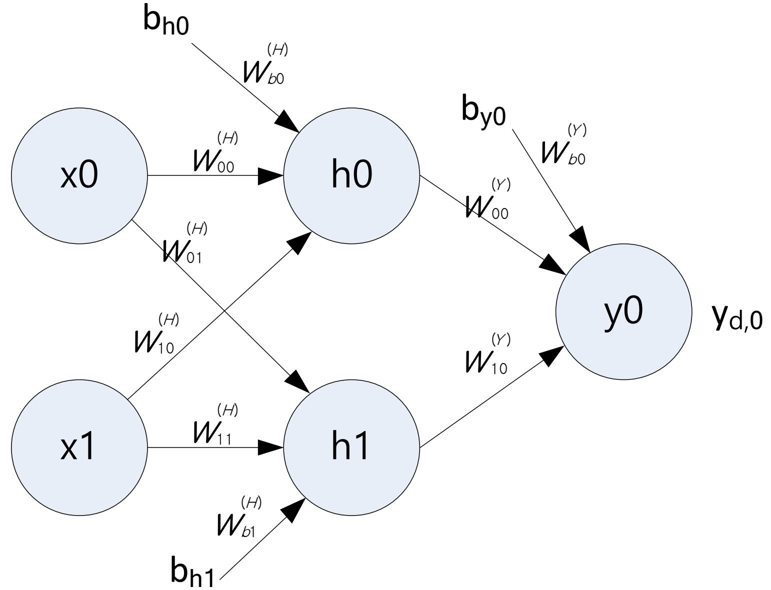


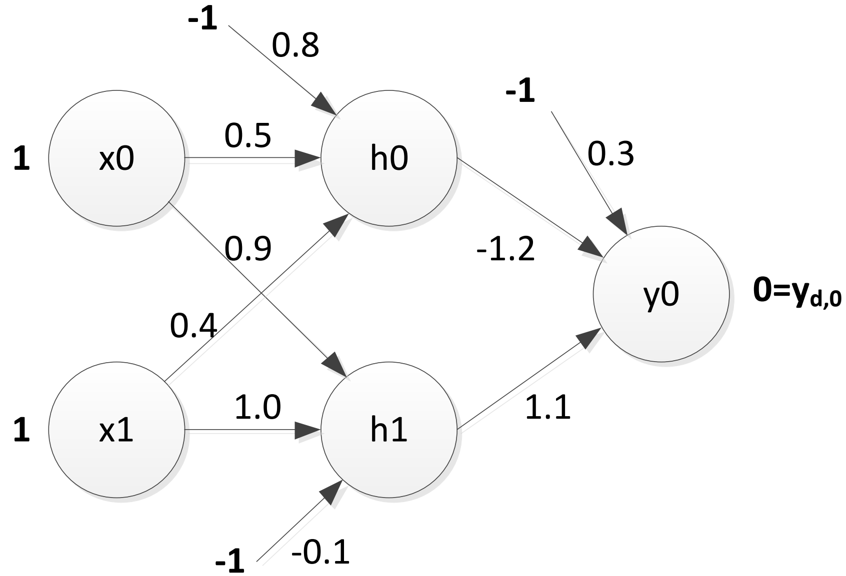


$$
\begin{aligned}
\delta_0^H &= h_0(1-h_0)\times\delta_0^Y\times w_{00}^Y \\
&= 0.5250\times(1-0.5250)\times(-0.1274)\times(-1.2) \\
&= -0.0381
\end{aligned}
$$

$$
\begin{aligned}
\delta_1^H &= h_1(1-h_1)\times\delta_0^Y\times w_{10}^Y \\
&= 0.8808\times(1-0.8808)\times(-0.1274)\times(1.1) \\
&= -0.0147
\end{aligned}
$$

Hidden Layer





$$\Delta w_{00}^{H} = \alpha \times x_0 \times \delta_0^H = 0.1 \times 1 \times 0.0381 = 0.0038$$

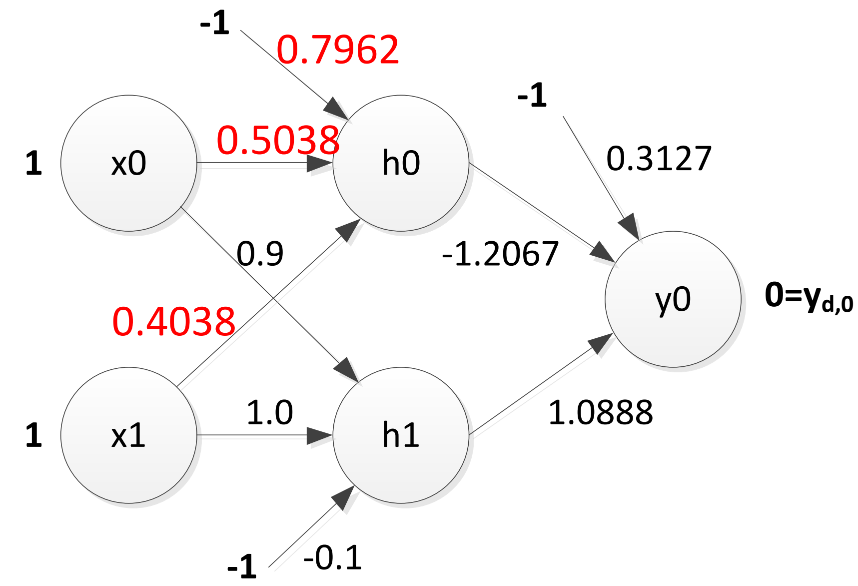
$$\Delta w_{10}^{H} = \alpha \times x_1 \times \delta_0^H = 0.1 \times 1 \times 0.0381 = 0.0038$$

$$\Delta w_{b0}^{H} = \alpha \times b_{h0} \times \delta_0^H = 0.1 \times (-1) \times 0.0381 = -0.0038$$

$$\Delta w_{01}^{H} = \alpha \times x_0 \times \delta_1^H = 0.1 \times 1 \times (-0.0147) = -0.0015$$

$$\Delta w_{11}^{H} = \alpha \times x_1 \times \delta_1^H = 0.1 \times 1 \times (-0.0147) = -0.0015$$

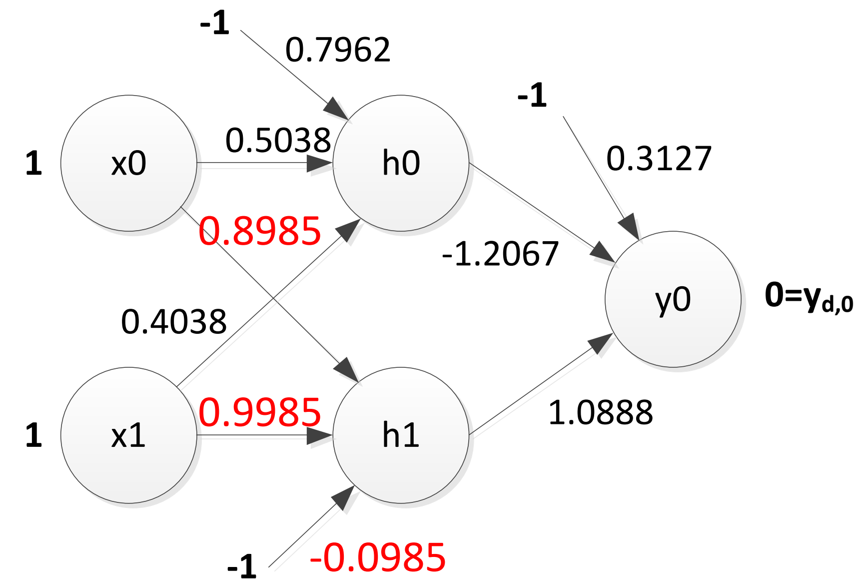
$$\Delta w_{b1}^{H} = \alpha \times b_{h1} \times \delta_1^H = 0.1 \times (-1) \times (-0.0147) = 0.0015$$



$$w_{00}^{H} = w_{00}^{H} + \Delta w_{00}^{H} = 0.5 + (0.0038) = 0.5038$$

$$w_{10}^{H} = w_{10}^{H} + \Delta w_{10}^{H} = 0.4 + (0.0038) = 0.4038$$

$$w_{b0}^{H} = w_{b0}^{H} + \Delta w_{b0}^{H} = 0.8 + (-0.0038) = 0.7962$$



$$w_{01}^{H} = w_{01}^{H} + \Delta w_{01}^{H} = 0.9 + (-0.0015) = 0.8985$$

$$w_{11}^{H} = w_{11}^{H} + \Delta w_{11}^{H} = 1.0 + (-0.0015) = 0.9985$$

$$w_{b1}^{H} = w_{b1}^{H} + \Delta w_{b1}^{H} = (-0.1) + 0.0015 = -0.0985$$

Sum of Square Errors

$$SSE = \sum_{i=1}^{n} (x_i - \overline{x})^2$$

$$D_i = \sum_{i=0}^{n} (y_i - y_{d,i})^2$$

$$SSE = \sum_{i=0}^{n} (y_i - \overline{y})^2 + \sum_{i=0}^{n} (y_{d,i} - \overline{y})^2$$

$$= D_i/2 = \left\{ \sum_{i=0}^{n} (y_i - y_{d,i})^2 \right\} /2$$

Practice

✓ 8×8이미지에 적힌 10개의 숫자를 판단하는 신경망 프로그램을 작성하세요.

References

- ✓ http://natureofcode.com/book/chapter-10-neural-networks/
- ✓ http://rimstar.org/science_electronics_projects/backpropagation_neural_network_software_3_layer.htm
- ✓ https://en.wikipedia.org/wiki/Sigmoid_function

MY **BRIGHT** FUTURE

동서대학교

DSU Dongseo University 동서대학교

DSU Dongseo University 동서대학교