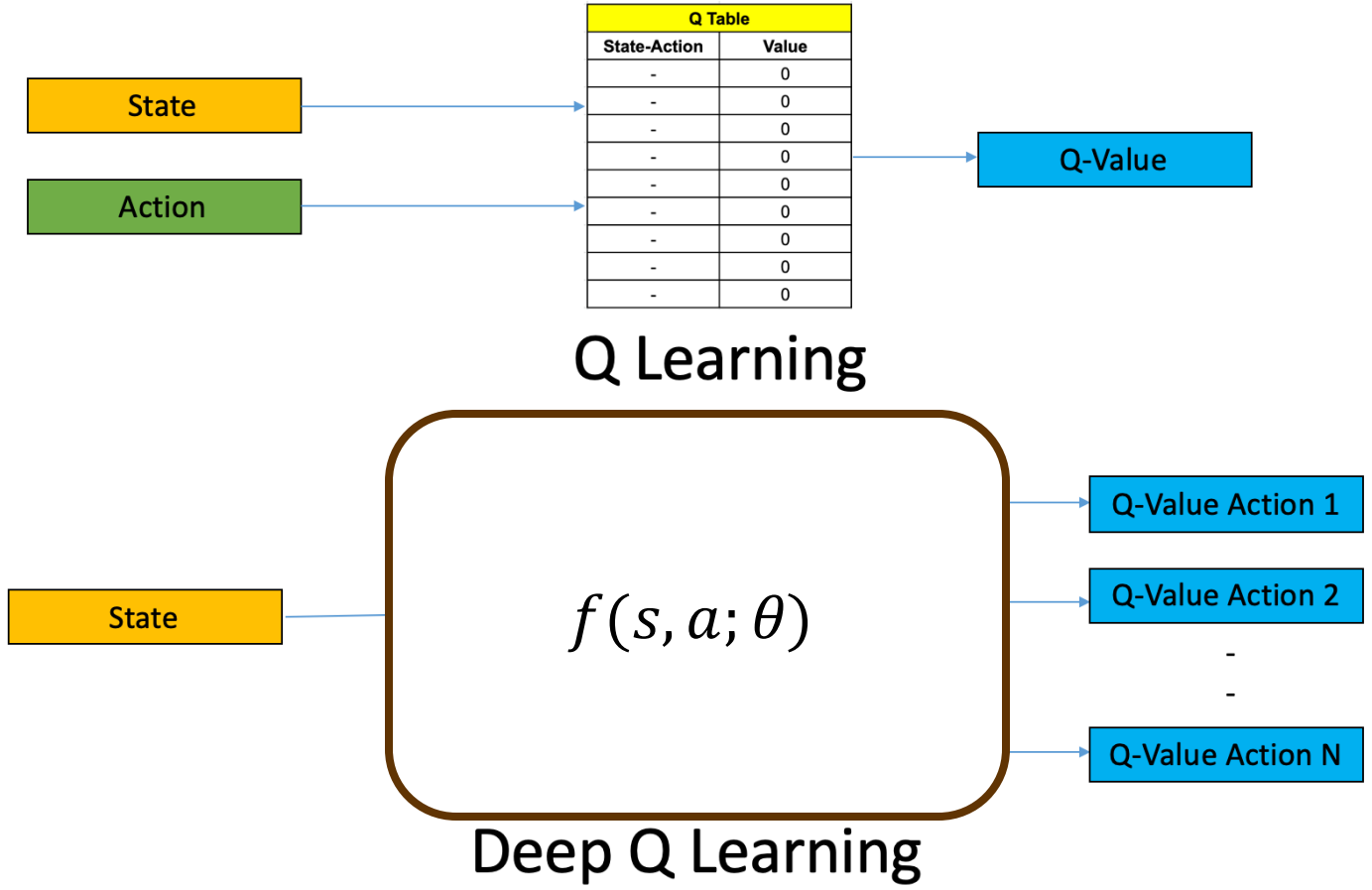
지능 시스템
Intelligent Systems

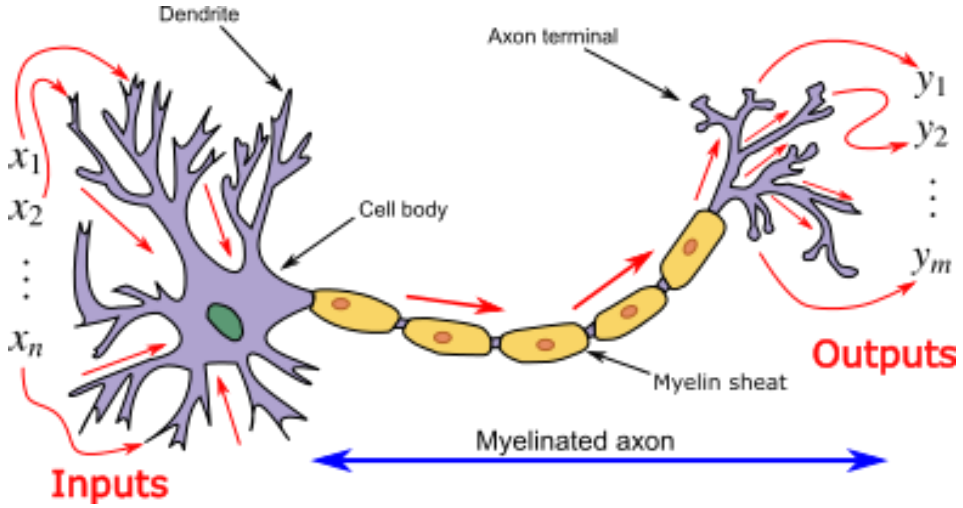
Lecture 5 – Revision of Neural Network

From Last Lecture



Perceptron

The idea of Perceptron was introduced by “Frank Rosenblatt(1957)”.
Old but still the key idea of Machine Learning and Deep Learning algorithms.
The Idea of Perceptron took its motivation from our neuro-cells



Perceptron

The circles containing data is called "**node**".

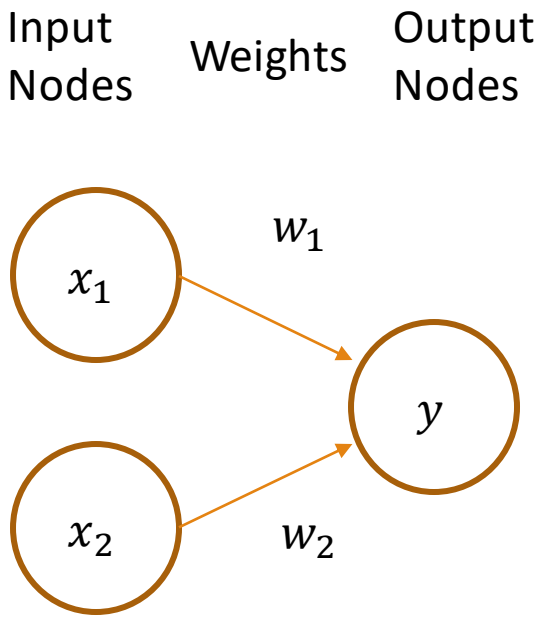
The Output node, containing y, may be expressed in 2 forms

1: Linear function of the input and weights:

$$y = w_1 x_1 + w_2 x_2$$

2: Binary Classification via a threshold value, θ, via activation function:

$$y = \begin{cases} 0, & w_1 x_1 + w_2 x_2 \leq \theta \\ 1, & w_1 x_1 + w_2 x_2 > \theta \end{cases}$$

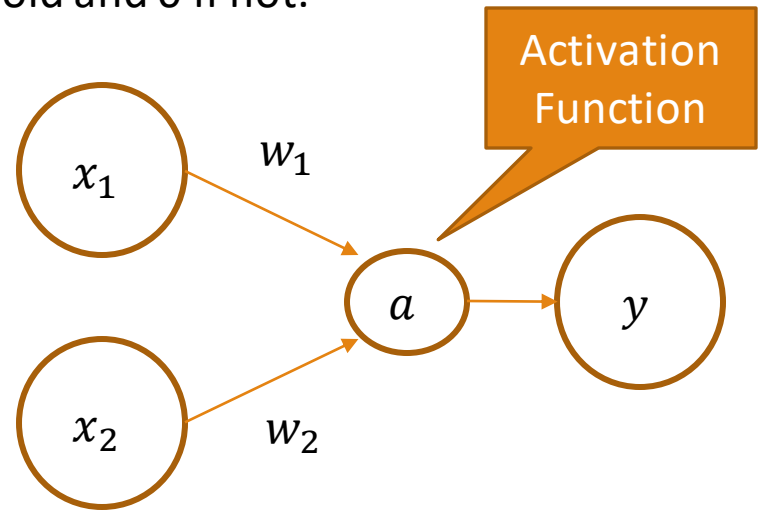


고키, p68~71

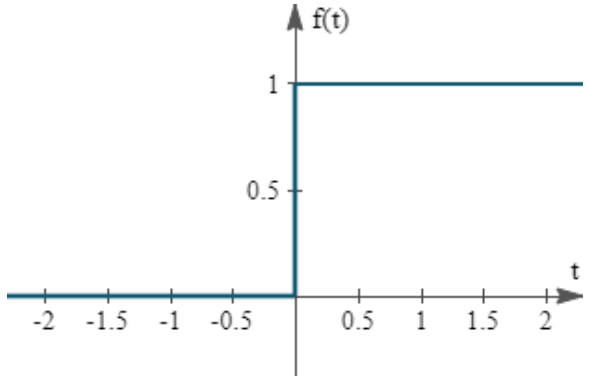
Activation Function

Activation function: A function that activates the next node from given information by previous nodes and weights.

An example: Step Function → Returns 1 if input is above certain threshold and 0 if not.



For simplicity, from now on, we will skip drawing the activation node in the diagram.

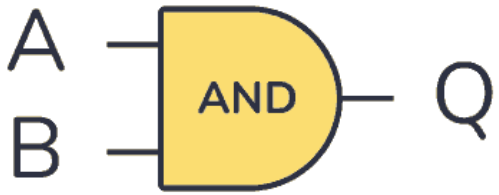


$$y = h(w_1x_1 + w_2x_2)\ ,$$
where

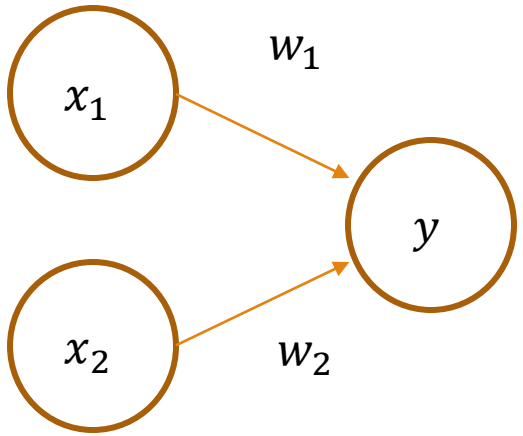
$$h(x) = \begin{cases} 0, & x \le \theta \\ 1, & x > \theta \end{cases}$$

$$y = \begin{cases} 0, & w_1x_1 + w_2x_2 \le \theta \\ 1, & w_1x_1 + w_2x_2 > \theta \end{cases}$$

Simple Logic Gate



A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1



x_1	x_1	y
0	0	0
1	0	0
0	1	0
1	1	1

$$y = \begin{cases} 0, & w_1 x_1 + w_2 x_2 \leq \theta \\ 1, & w_1 x_1 + w_2 x_2 > \theta \end{cases}$$

Various combinations of (w_1, w_2, θ) exist:

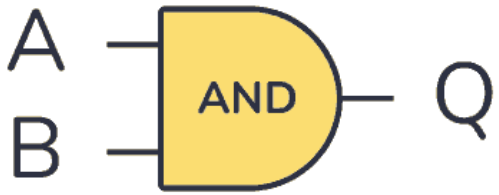
(0.5, 0.5, 0.7)

(0.5, 0.5, 0.8)

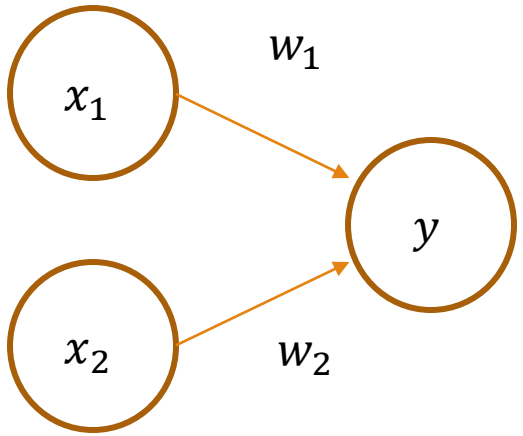
(1.0, 1.0, 1.0)

Try Out!!

Simple Logic Gate



A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1



x_1	x_1	y
0	0	0
1	0	0
0	1	0
1	1	1

Instead of threshold, we introduce a bias;

$$b = -\theta$$

$$y = \begin{cases} 0, & b + w_1 x_1 + w_2 x_2 \leq 0 \\ 1, & b + w_1 x_1 + w_2 x_2 > 0 \end{cases}$$

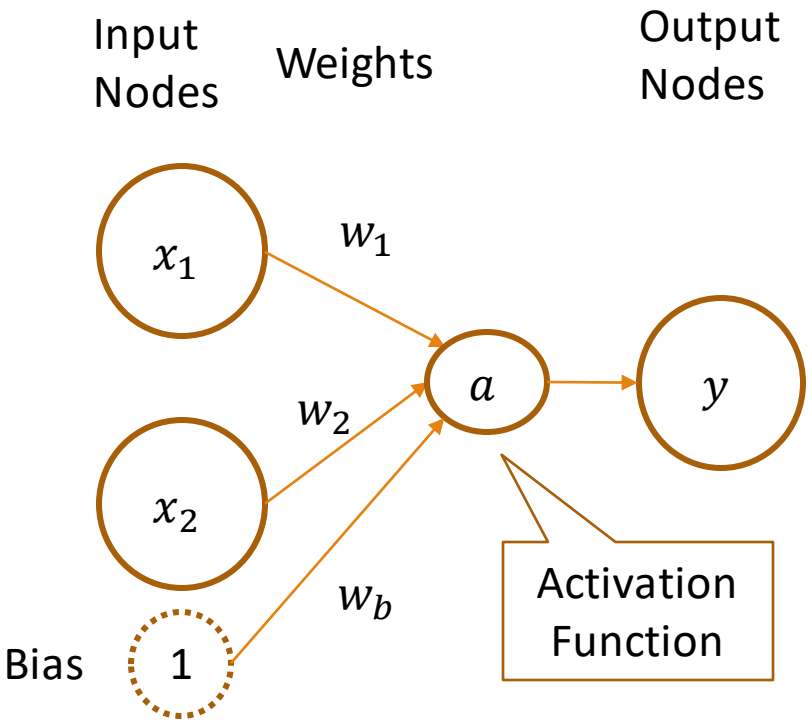
Various combinations of (w_1, w_2, b) exist:

(0.5, 0.5, -0.7)
(0.5, 0.5, -0.8)
(1.0, 1.0, -1.0)

Try Out!!

Perceptron with bias

A Perceptron without hidden layers can be used to compute a simple logic gate and classification but more complex non-linear classifications, such as XOR gate representation, were impossible to be represented.



For Step Function activation:

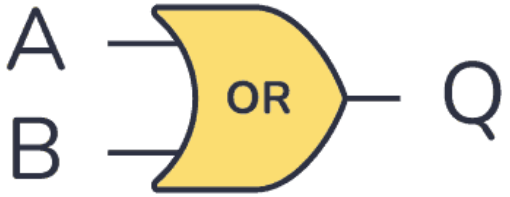
$$y = \begin{cases} 0, & w_b + w_1 x_1 + w_2 x_2 \leq 0 \\ 1, & w_b + w_1 x_1 + w_2 x_2 > 0 \end{cases}$$

Note we have also added a weight for the bias, w_b, and we usually set the bias to 1.

For simplicity, we omit representing the bias and activation function in the diagram from now on.

고키, p49~51

NAND and OR Gates



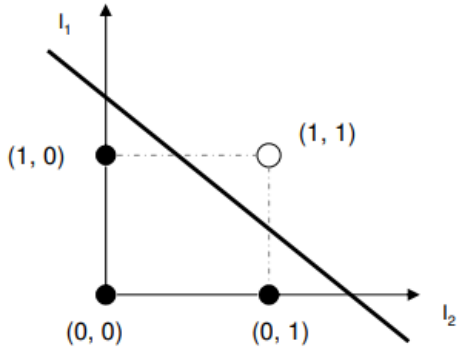
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

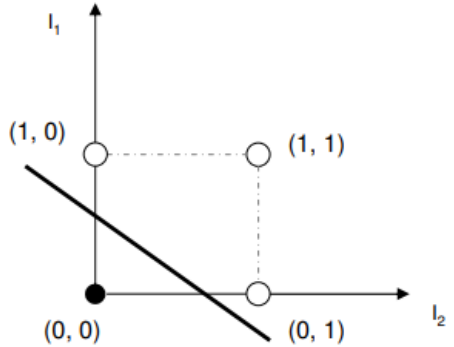
XOR Gate

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1

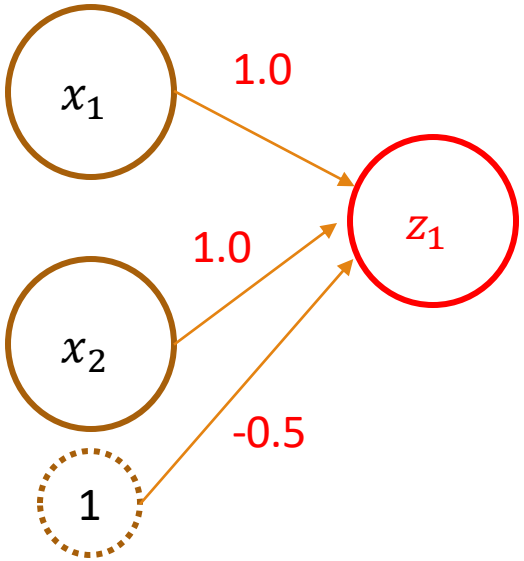


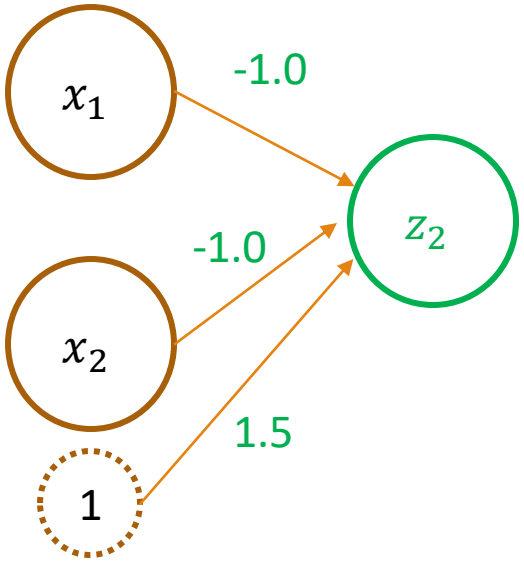
OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1

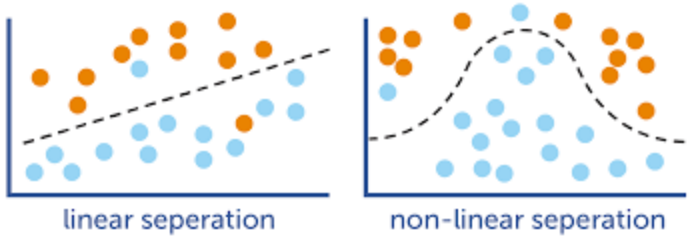


XOR Gate

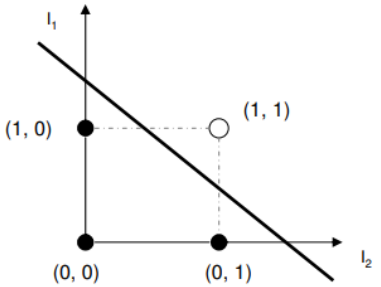
A single-layer perceptron is capable of only classifying data linearly.



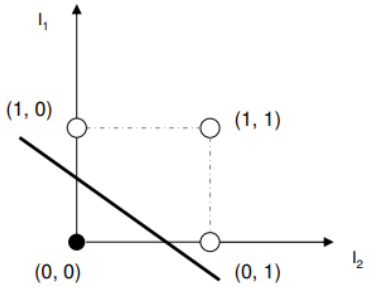




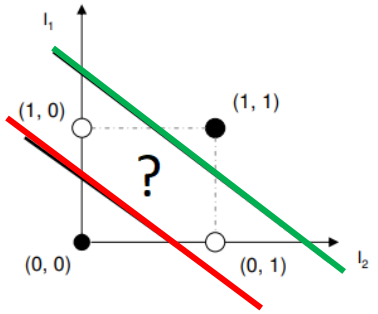
AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1



OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1

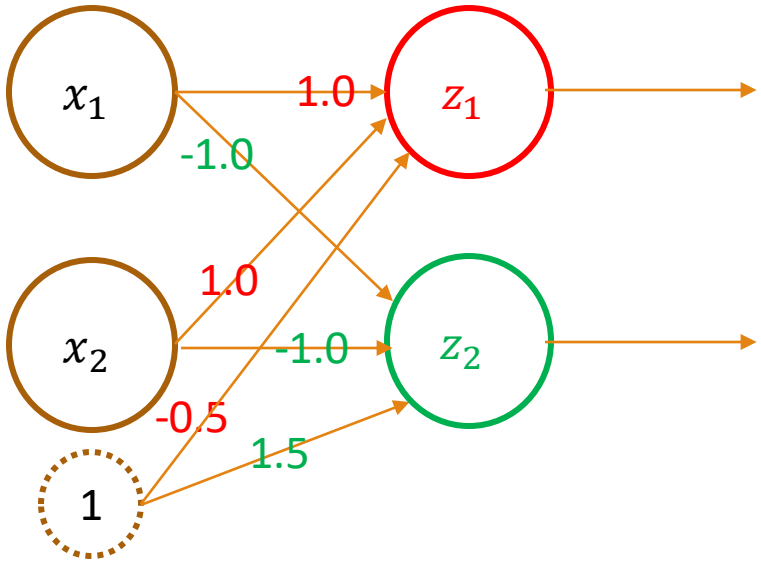


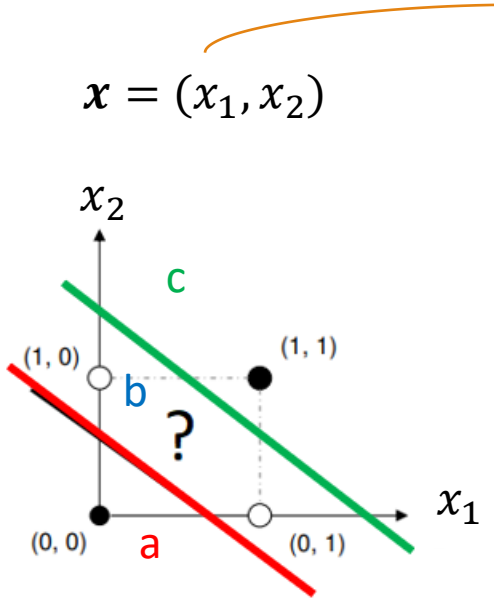
XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0



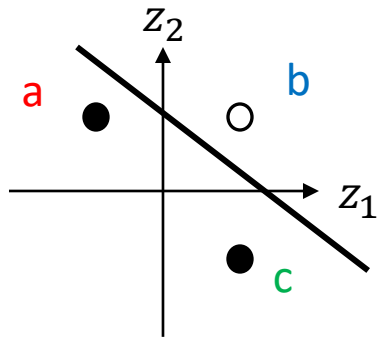
XOR Gate

By adding an additional layer, now the transformed space is linearly separable.



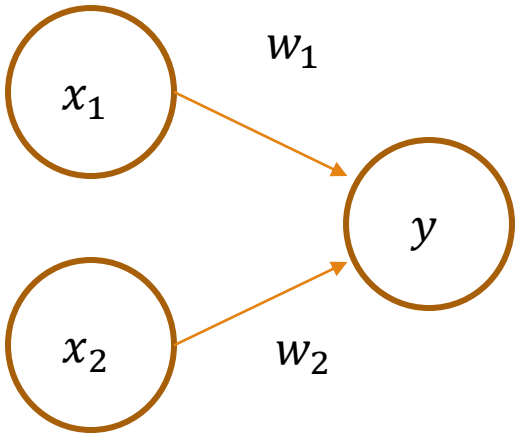


$\boldsymbol{x} = (x_1, x_2)$ $\quad$ $\boldsymbol{z} = (z_1, z_2)$



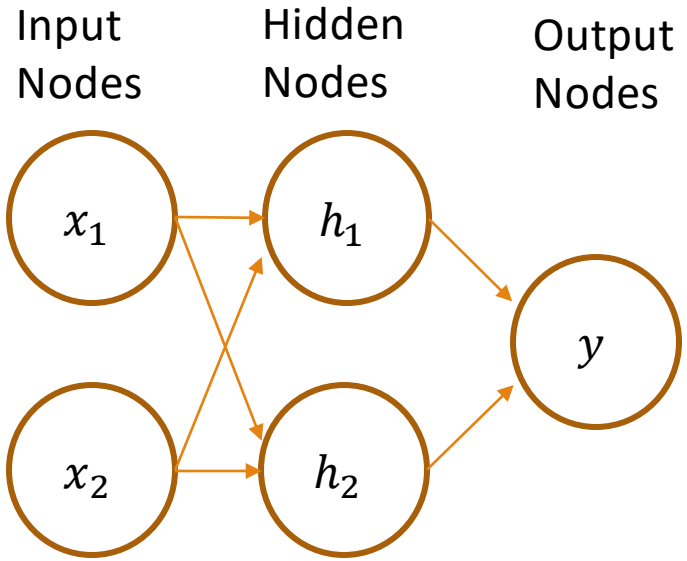
고키, p57~60

Multi-layer Perceptron



A perceptron can only classify data linearly. Just like linear regression.

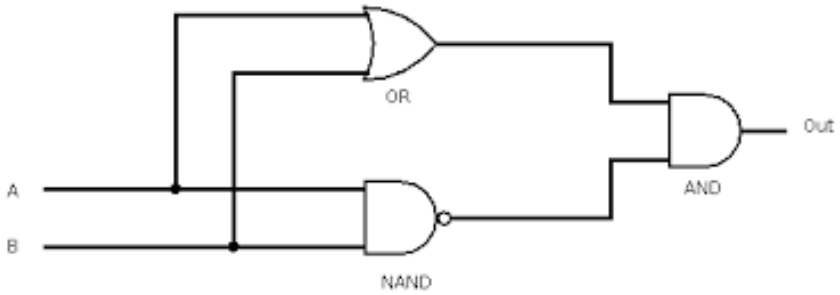
By adding an additional layer between the input and output layer, it can now classify data non-linearly.
This is called Multi-layer Perceptron

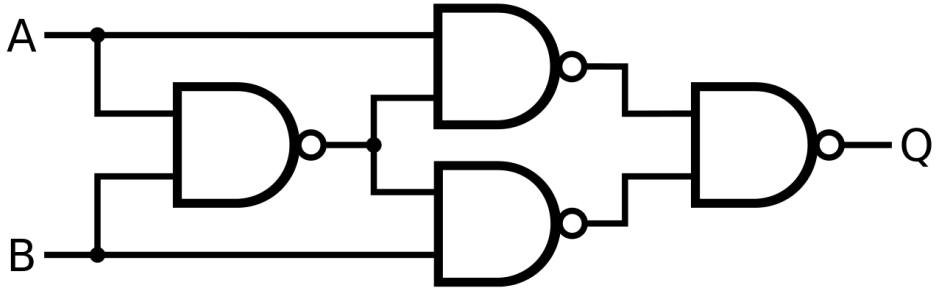


고키, p57~60

Stacking Logic Gates

Given AND, OR and NAND logic gates can be designed by perceptrons, the logic circuit diagram below can be built and have the same effect as XOR gate.

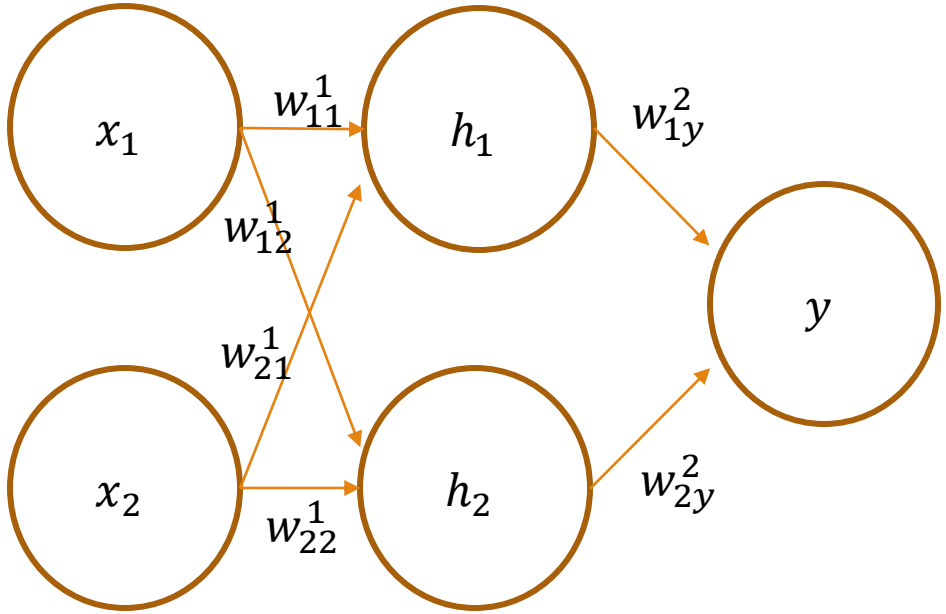




This is identical to having multiple layers in a perceptron.

Multi-layer Perceptron

Multi-layer Perceptron enables non-linear classifications due to a stacking of non-linear activation functions.
However, we inevitably have more weight parameters to determine.
We call this structure a **Fully-Connected Neural Network**
Notice that the notations have changed.



$$h_1 = \begin{cases} 0, & w_{b1}^1 + w_{11}^1 x_1 + w_{21}^1 x_2 \leq 0 \\ 1, & w_{b1}^1 + w_{11}^1 x_1 + w_{21}^1 x_2 > 0 \end{cases}$$

$$h_2 = \begin{cases} 0, & w_{b2}^1 + w_{12}^1 x_1 + w_{22}^1 x_2 \leq 0 \\ 1, & w_{b2}^1 + w_{12}^1 x_1 + w_{22}^1 x_2 > 0 \end{cases}$$

$$y = \begin{cases} 0, & w_{by}^2 + w_{1y}^2 h_1 + w_{2y}^2 h_2 \leq 0 \\ 1, & w_{by}^2 + w_{1y}^2 h_1 + w_{2y}^2 h_2 > 0 \end{cases}$$

고키, p75

Multi-layer Perceptron

How does the Multi-layer Perceptron classify data non-linearly?

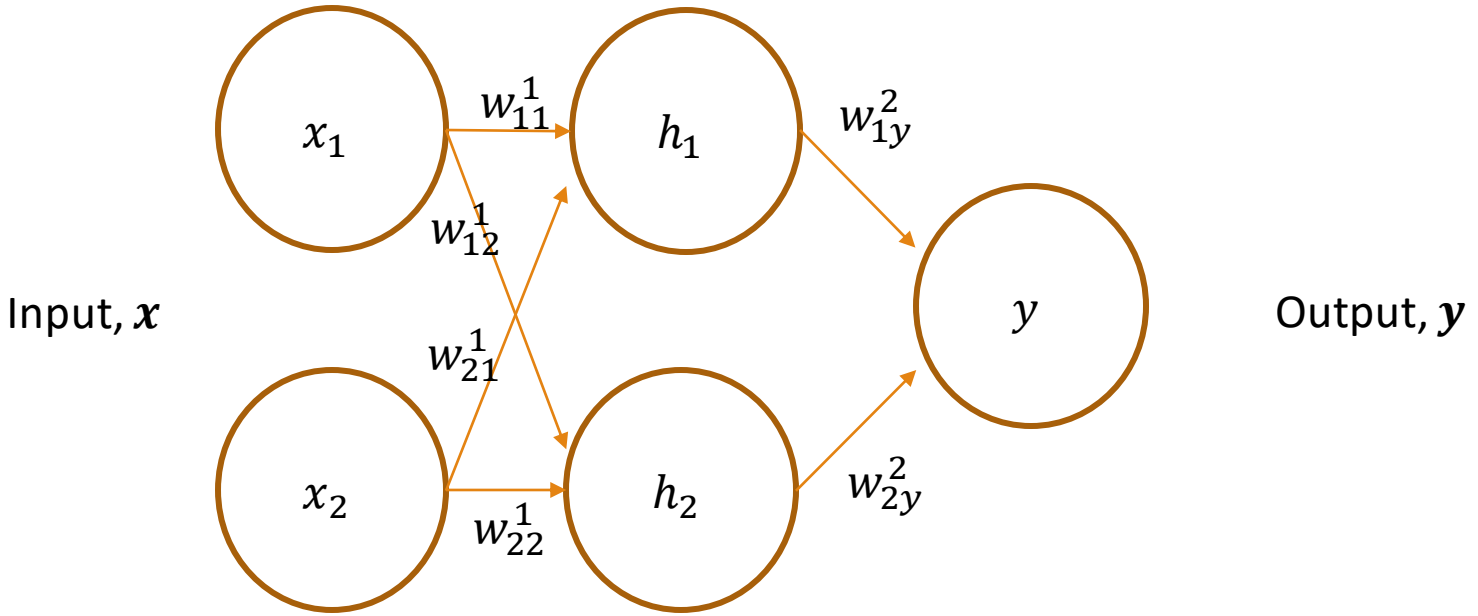
The activation function must be non-linear, and what we used, the Step Function is Non-Linear Function!

Stacking Non-Linear functions allow the system to behave non-linearly.

Imagine have a linear activation function; $h(x) = cx$, where c is a constant.
Then, stacking up 3 layers will result (roughly); $\mathrm{y}(x) = h\Big(h\big(h(x)\big)\Big) = c * c * c * x$
Which can be done by just one layer with an activation function of; $h(x) = c^3 x$

Forward Propagation

Forward Propagation: Input → Output



Forward Propagation

Let Input vector, $\boldsymbol{x} = \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix}$ ← bias

Then,

$$h_1 = a\left(w_{b1}^1 + w_{11}^1 x_1 + w_{21}^1 x_2\right)$$
$$h_2 = a\left(w_{b2}^1 + w_{12}^1 x_1 + w_{22}^1 x_2\right)$$

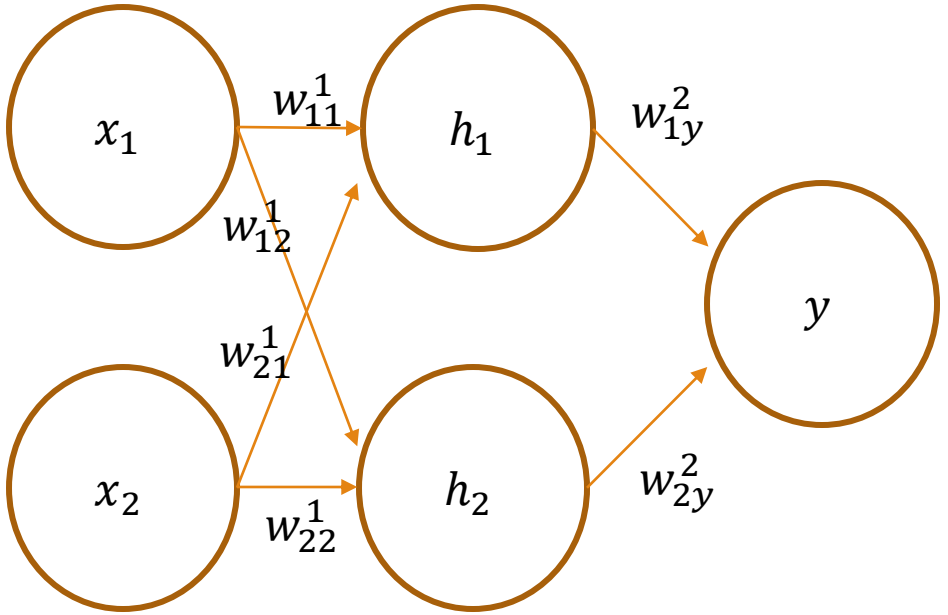
And we can combine these expressions,

$$\boldsymbol{h} = a(\boldsymbol{w}^{(1)}\boldsymbol{x})$$

Where

$$\boldsymbol{w}^{(1)} = \begin{pmatrix} w_{b1}^1 & w_{11}^1 & w_{21}^1 \\ w_{b2}^1 & w_{12}^1 & w_{22}^1 \end{pmatrix}$$

Where (1) denotes that the weights are between input and the first hidden layer. (Not power)



We denote the activation function as $a()$

Forward Propagation

Let Input vector, $\boldsymbol{x} = \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix}$ ← bias

Then,

$$\boldsymbol{h} = a(\boldsymbol{w}^{(1)}\boldsymbol{x})$$

Where

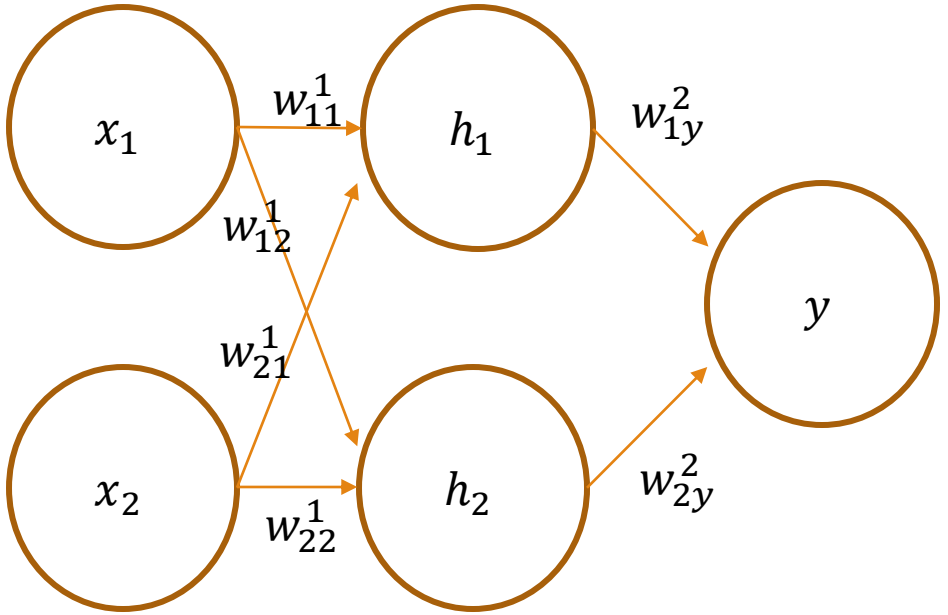
$$\boldsymbol{w}^{(1)} = \begin{pmatrix} w_{b1}^1 & w_{11}^1 & w_{21}^1 \\ w_{b2}^1 & w_{12}^1 & w_{22}^1 \end{pmatrix}$$

Then,

$$y = a(\boldsymbol{w}^{(2)}\boldsymbol{h})$$

Where

$$\boldsymbol{w}^{(2)} = \begin{pmatrix} w_{by}^2 & w_{1y}^2 & w_{2y}^2 \end{pmatrix}$$



We denote the activation function as $a()$

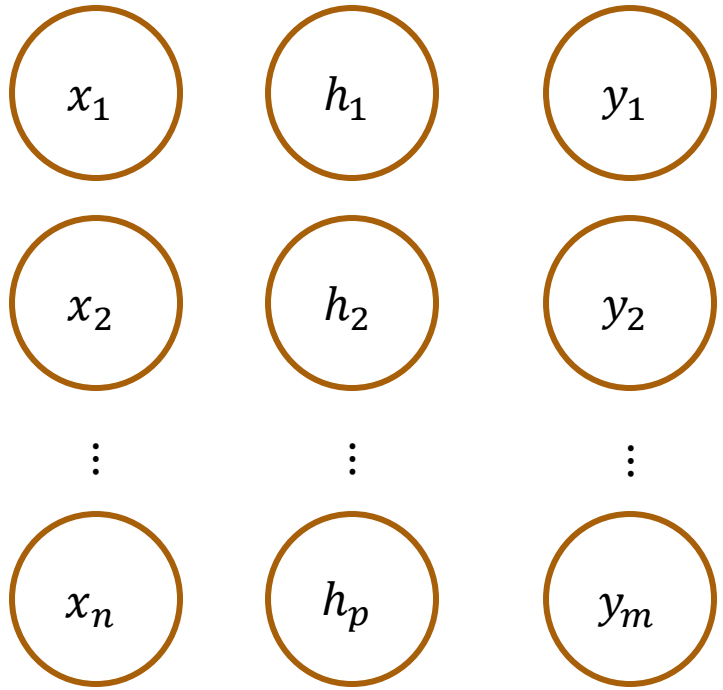
Forward Propagation

Let Input vector, $\boldsymbol{x} = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$ and $\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$

$$\boldsymbol{w^{(1)}} = \begin{pmatrix} w_{b1}^1 & w_{11}^1 & w_{21}^1 & \cdots & w_{n1}^1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{bp}^1 & w_{1p}^1 & w_{2p}^1 & \cdots & w_{np}^1 \end{pmatrix} \qquad p \times (n+1)$$

$$\boldsymbol{w^{(2)}} = \begin{pmatrix} w_{b1}^2 & w_{11}^2 & w_{21}^2 & \cdots & w_{p1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{bm}^2 & w_{1m}^2 & w_{2m}^2 & \cdots & w_{pm}^2 \end{pmatrix} \qquad m \times (p+1)$$

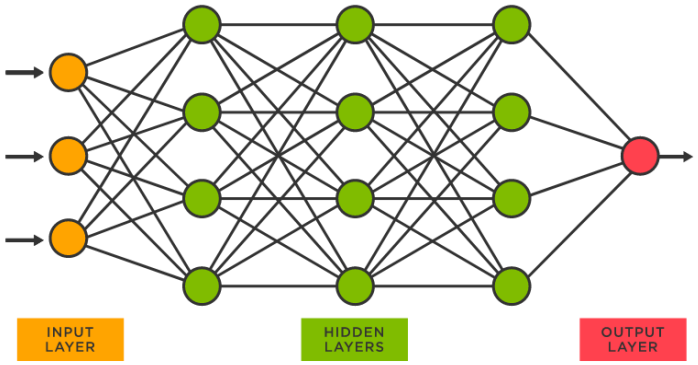
$$y = a_2(\boldsymbol{w^{(2)}} \underbrace{a_1(\boldsymbol{w^{(1)}}\boldsymbol{x})}_{\boldsymbol{h}})$$



QUIZ!

Consider a fully connected neural network with 3 hidden layers of 4 nodes each. The input is state information about the blackjack card game, my current sum, a dealer's showing card, and whether I have a usable ace. The network is used to output Q values of all possible actions, that are "Hit" and "Stick" in this game.

Q: State the size of the weight matrix between each layer and the total number of weight parameters to be deduced/learned.



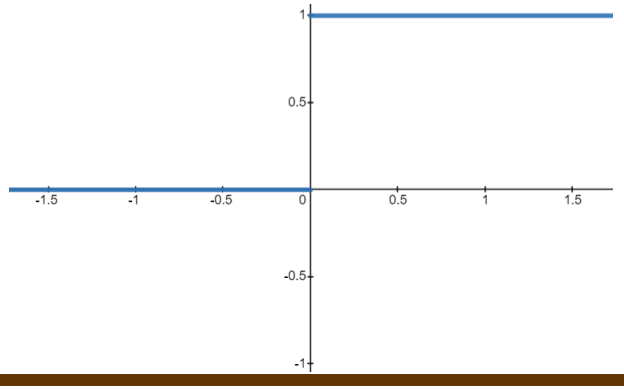
Q?

Activation Functions

Step Function:

$$a(x) = \begin{cases} 1, & x \geq 0 \\ \beta, & x < 0 \end{cases}$$

Where $\beta = 0$ or $\beta = -1$

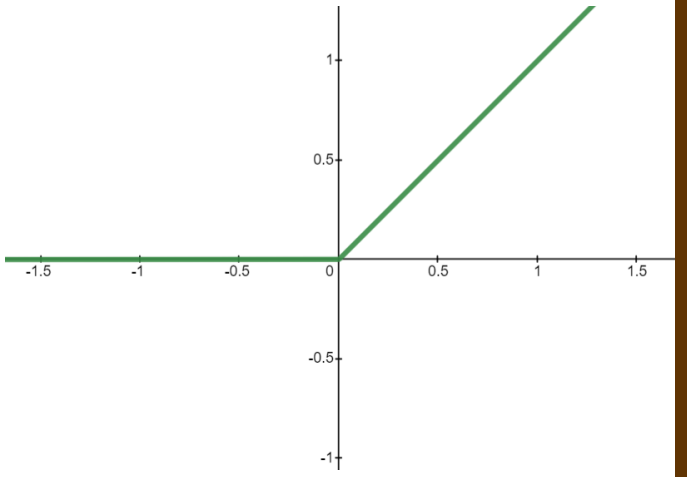


Step function:

- Enables Classification.
- Gradient is always 0 for any input.

ReLU(Rectified Linear Unit):

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



ReLU:

- Enables Classification.
- Gradient is always 0 for negative input.
- Gradient is always 1 for positive input.

Finding optimal weights – Loss function

As discussed so far, the key of deep learning and using neural networks lies on being able to find optimal/near-optimal weight parameters, $\boldsymbol{w}$.

Consider a function expression of neural network:

$$\boldsymbol{y} = f(\boldsymbol{x}, \boldsymbol{w})$$

If wrong weights are used, the outputs will be far from our expectations and correct answers.
Therefore, we need a measure of how good the weights are!
We call this a **Loss function**.

Loss Function: a function that represents differences between network outputs and given answers.

Eg; $L(\boldsymbol{x}, \boldsymbol{w}) = (y - t)^2 = (f(\boldsymbol{x}, \boldsymbol{w}) - t)^2$, where t is the given answer label

Finding optimal weights – Loss function

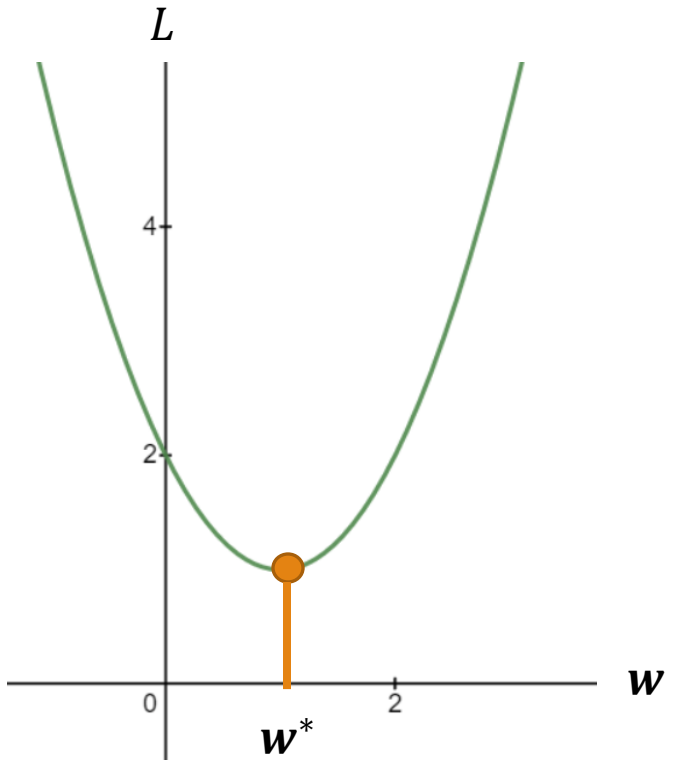
Consider: $L(\boldsymbol{x}, \boldsymbol{w}) = (y - t)^2 = (f(\boldsymbol{x}, \boldsymbol{w}) - t)^2$

Our task is finding weights, $\boldsymbol{w}$, that minimizes the loss, L.

From intuition, we can simply find the optimal weights is finding the turning points of the loss function by differentiating it.

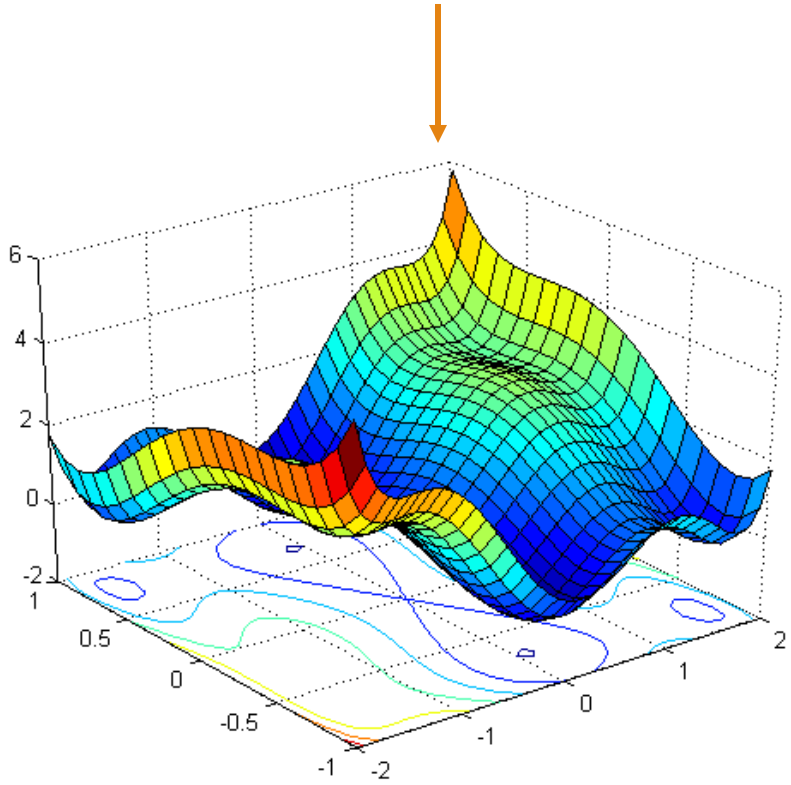
$$\boldsymbol{w}^* = \boldsymbol{argmin}_{\boldsymbol{w}}(L(\boldsymbol{x}, \boldsymbol{w}))$$

$$0 = \frac{\partial L}{\partial \boldsymbol{w}}|_{\boldsymbol{w}=\boldsymbol{w}^*}$$



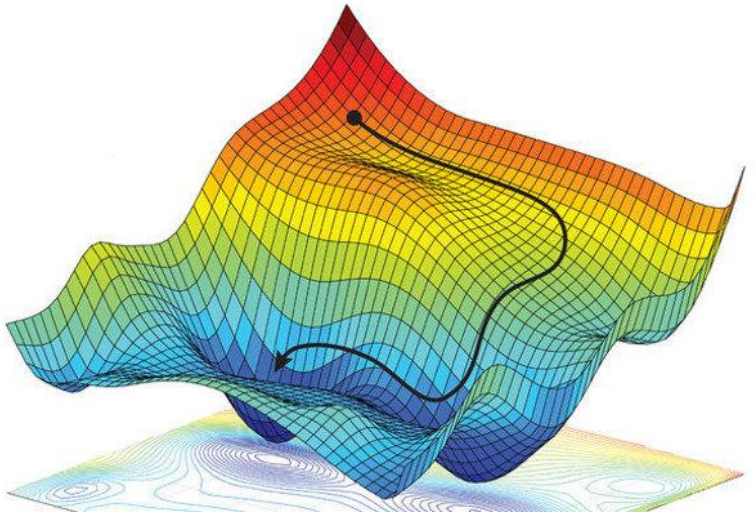
Finding optimal weights – Loss function

Example loss function with two weights



Actual loss functions with billions of weight parameters are computationally impossible to compute the full derivative.

So, without the derivative, how can we find the optimal weights?



Next Lecture

Learning Algorithms!