

Activation function

Overview

앞으로 우리가 배워야 할 것

기존에 배운 모델을 바탕으로, 딥러닝 모델을 구성하는 각 파트를 하나씩 개선해나간다
주로 **Optimizer**, **Weight Initialization**, **Activate Function** 등이 중요하다

```
learning_rate = 0.000001

w1 = np.random.uniform(low=-0.058, high=+0.058, size=(784, 1000))
w2 = np.random.uniform(low=-0.077, high=+0.077, size=(1000, 10))

num_epoch = 100

for epoch in range(num_epoch):
    # Forward propagation
    z1 = X_train.dot(w1)
    a1 = sigmoid(z1)
    z2 = a1.dot(w2)
    a2 = sigmoid(z2)

    # Backpropagation
    d2 = a2 - y_train_hot
    d1 = d2.dot(w2.T) * a1 * (1 - a1)

    w2 = w2 - learning_rate * a1.T.dot(d2)
    w1 = w1 - learning_rate * X_train.T.dot(d1)
```

앞으로 우리가 배워야 할 것

기존에 배운 모델을 바탕으로, 딥러닝 모델을 구성하는 각 파트를 하나씩 개선해나간다
주로 **Optimizer**, **Weight Initialization**, **Activate Function** 등이 중요하다

```
learning_rate = 0.000001
```

Weight Initialization

```
w1 = np.random.uniform(low=-0.058, high=+0.058, size=(784, 1000))  
w2 = np.random.uniform(low=-0.077, high=+0.077, size=(1000, 10))
```

```
num_epoch = 100
```

```
for epoch in range(num_epoch):
```

```
    # Forward propagation
```

```
    z1 = X_train.dot(w1)
```

```
    a1 = sigmoid(z1)
```

```
    z2 = a1.dot(w2)
```

```
    a2 = sigmoid(z2)
```

Activation Function

```
    # Backpropagation
```

```
    d2 = a2 - y_train_hot
```

```
    d1 = d2.dot(w2.T) * a1 * (1 - a1)
```

Optimizer

```
    w2 = w2 - learning_rate * a1.T.dot(d2)
```

```
    w1 = w1 - learning_rate * X_train.T.dot(d1)
```

앞으로 우리가 배워야 할 것

기존에 배운 모델을 바탕으로, 딥러닝 모델을 구성하는 각 파트를 하나씩 개선해나간다
주로 **Optimizer**, **Weight Initialization**, **Activate Function** 등이 중요하다

```
learning_rate = 0.000001

w1 = np.random.uniform(low=-0.058, high=+0.058, size=(784, 1000))
w2 = np.random.uniform(low=-0.077, high=+0.077, size=(1000, 10))

num_epoch = 100

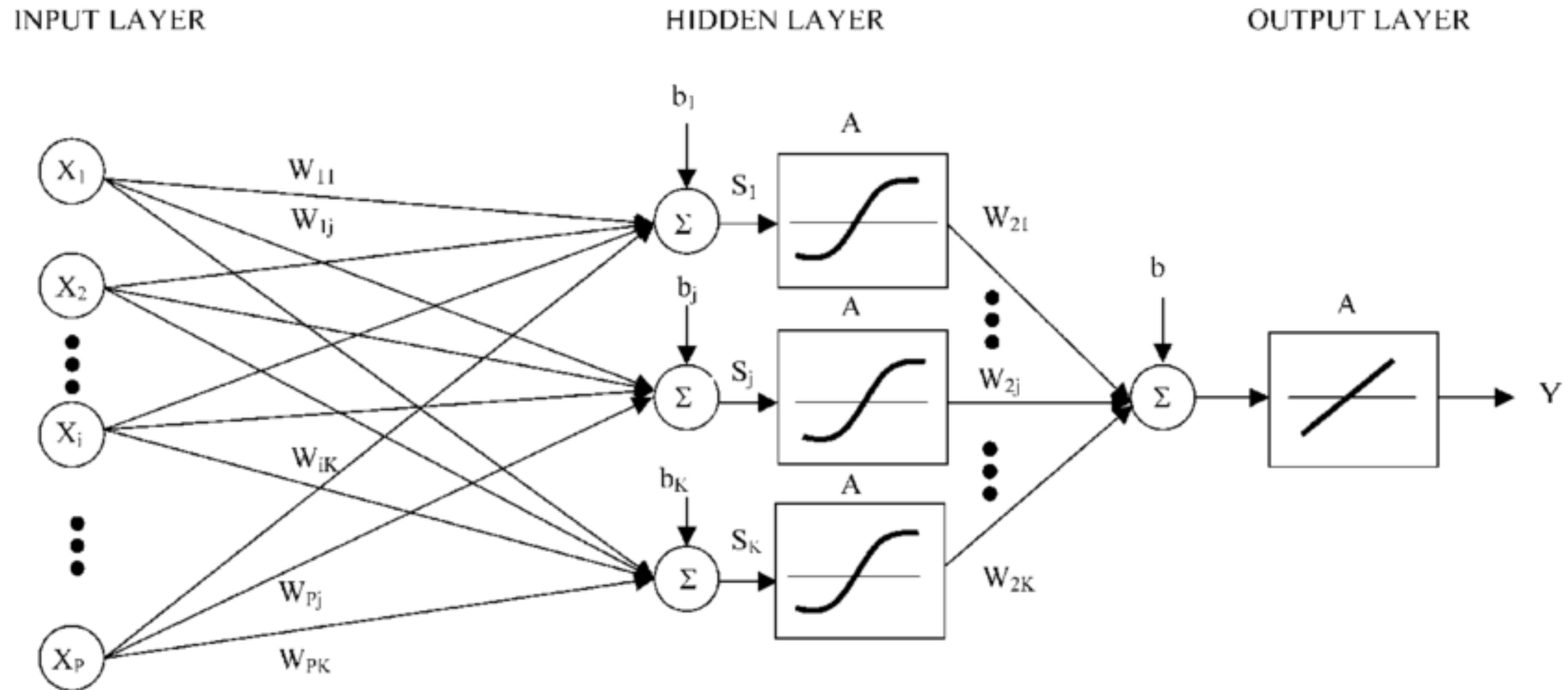
for epoch in range(num_epoch):
    # Forward propagation
    z1 = X_train.dot(w1)
    a1 = sigmoid(z1)
    z2 = a1.dot(w2)
    a2 = sigmoid(z2)

    # Backpropagation
    d2 = a2 - y_train_hot
    d1 = d2.dot(w2.T) * a1 * (1 - a1)

    w2 = w2 - learning_rate * a1.T.dot(d2)
    w1 = w1 - learning_rate * X_train.T.dot(d1)
```

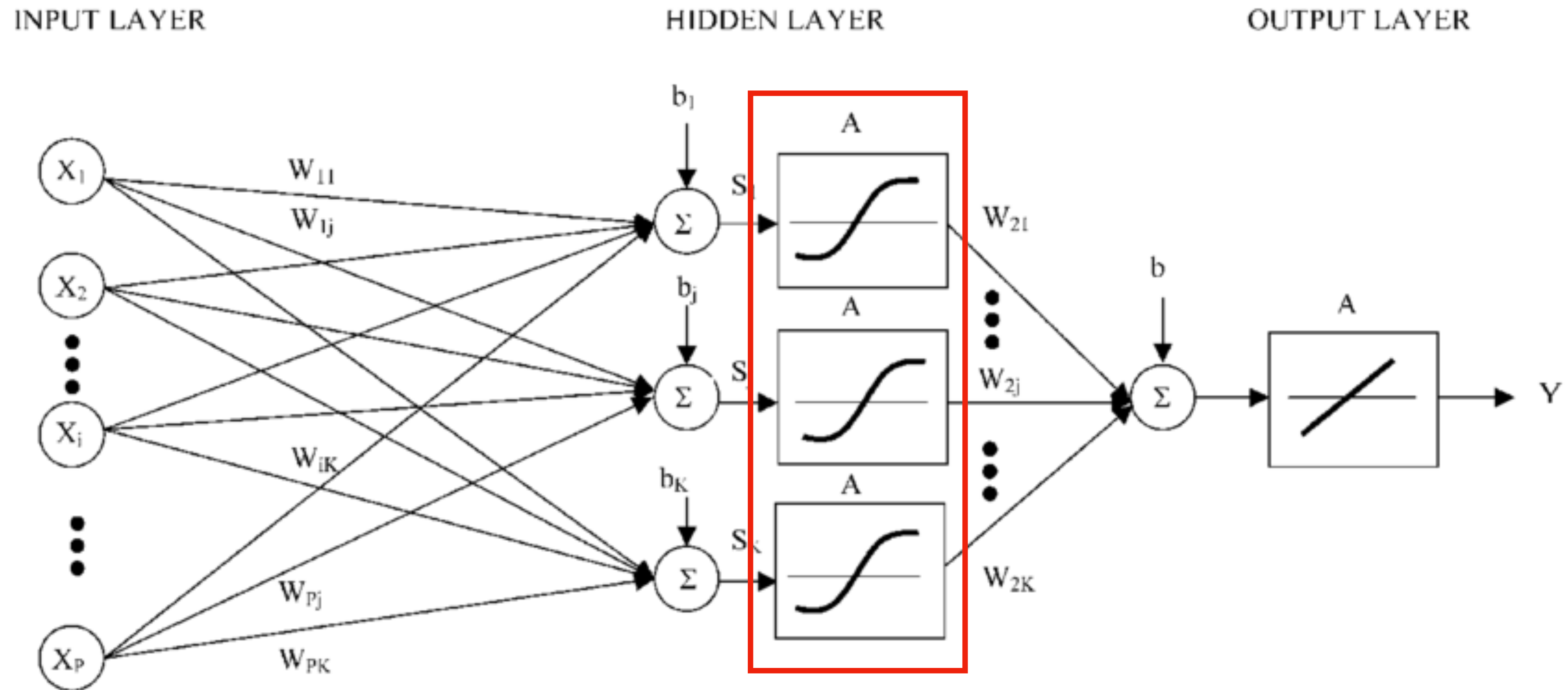
Activate function은 무엇인가?

Network를 구축할 때 Hidden Layer 사이에 집어넣는 것
보통 $WX+b$ 또는 Convolutional Layer의 convolve 연산 이후에 바로 activation을 한다



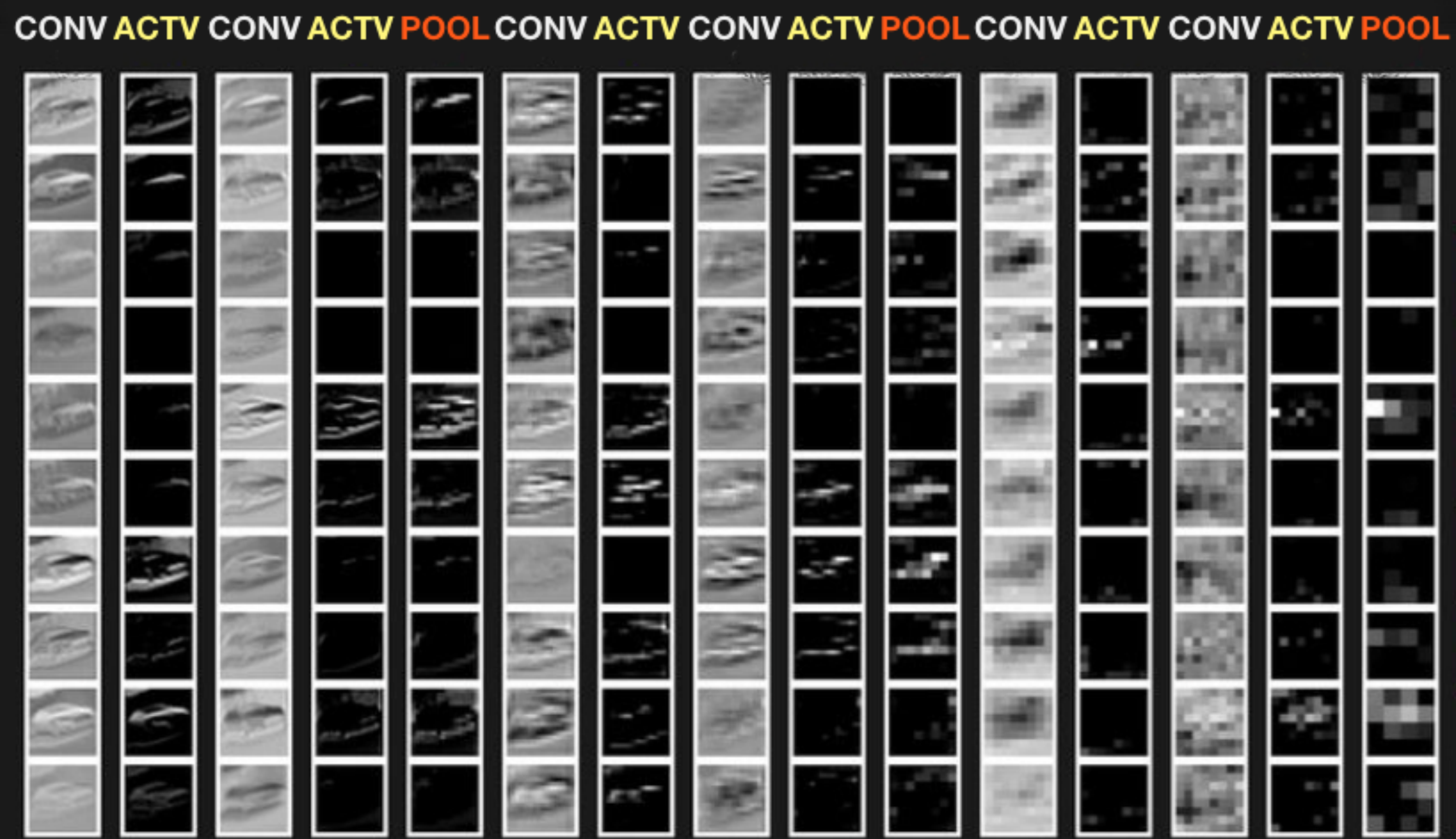
Activate function은 무엇인가?

Network를 구축할 때 Hidden Layer 사이에 집어넣는 것
보통 $WX+b$ 또는 Convolutional Layer의 convolve 연산 이후에 바로 activation을 한다



Activate function은 왜 필요한가?

Network를 구축할 때 Hidden Layer 사이에 집어넣는 것
보통 $WX+b$ 또는 Convolutional Layer의 convolve 연산 이후에 바로 activation을 한다

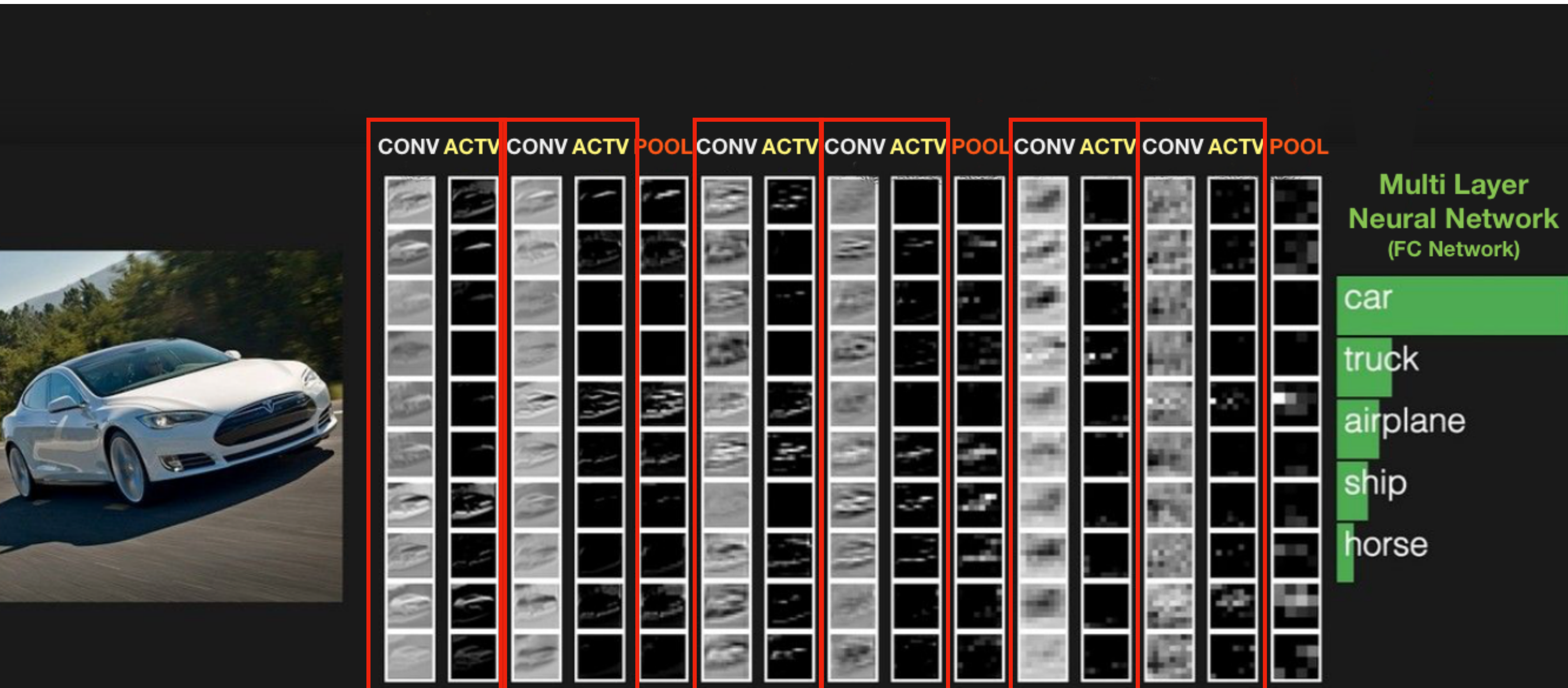


**Multi Layer
Neural Network
(FC Network)**

- car
- truck
- airplane
- ship
- horse

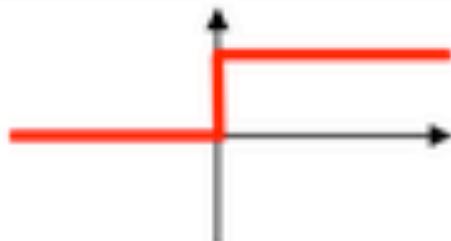
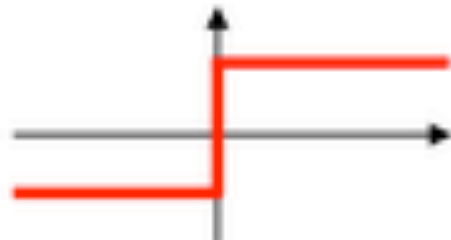

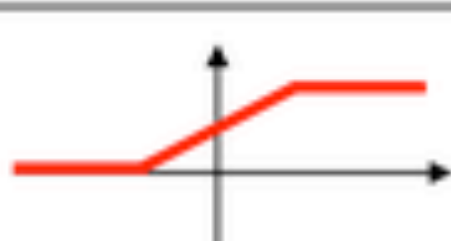
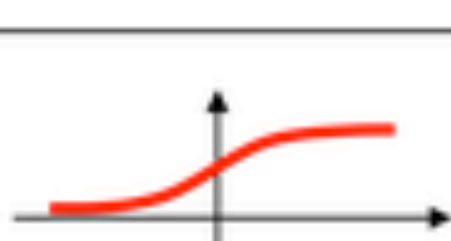

Activate function은 왜 필요한가?

Network를 구축할 때 Hidden Layer 사이에 집어넣는 것
보통 $WX+b$ 또는 Convolutional Layer의 convolve 연산 이후에 바로 activation을 한다



Activate function의 종류

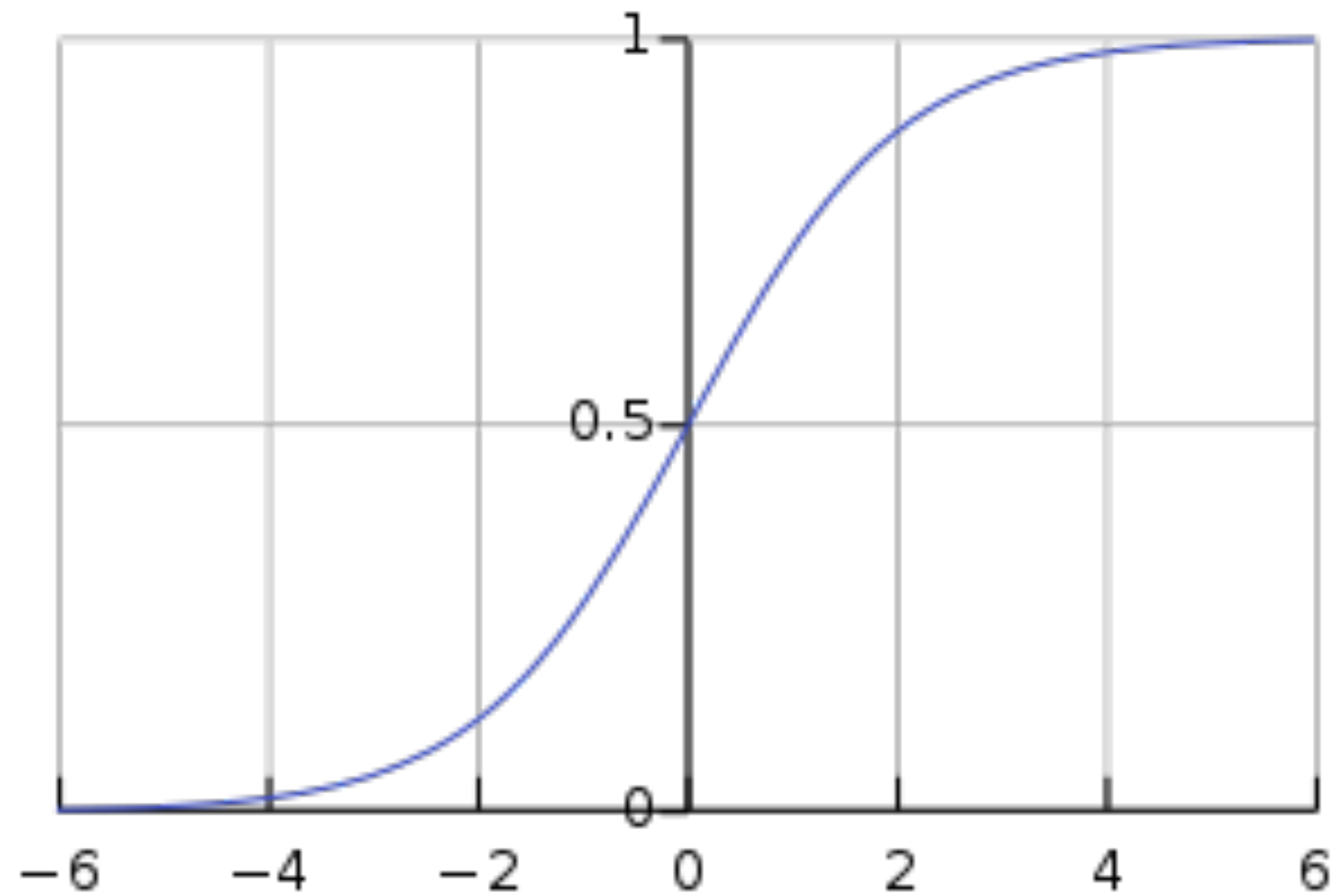
지금까지는 sigmoid만 사용하였지만, 실은 더 다양한 activation function이 있다.
(또한 앞으로 설명하겠지만, sigmoid는 그렇게 좋은 activation function이라고 할 수 없다)

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer NN	

Activate Function의 종류와 장단점

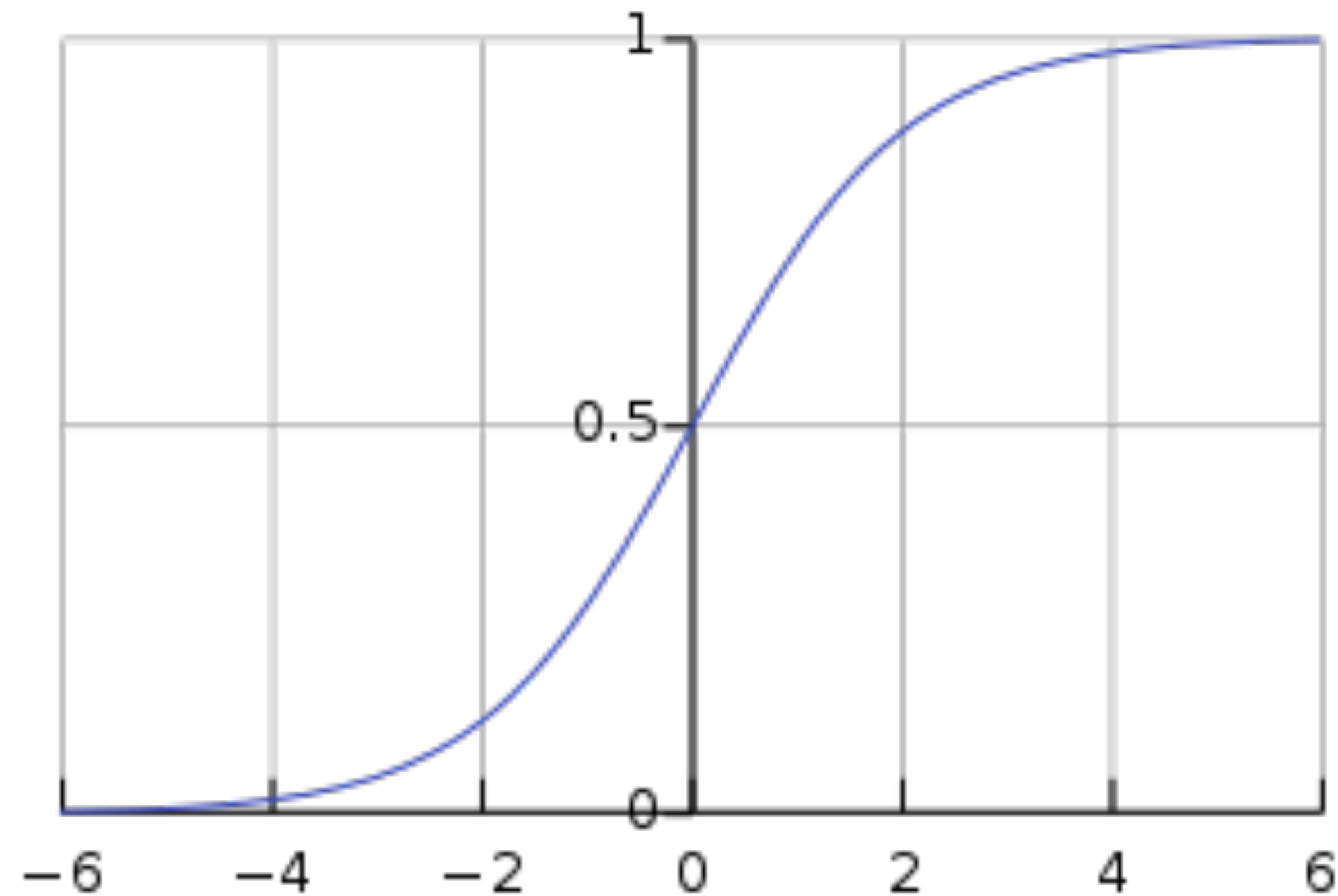
sigmoid

가장 유서깊은 역사를 자랑하는 **Activate Function**
input으로 어떠한 값이 들어가건, 그 결과를 0에서 1 사이로 squash 해준다



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

가장 유서깊은 역사를 자랑하는 **Activate Function**
input으로 어떠한 값이 들어가건, 그 결과를 0에서 1 사이로 **squash** 해준다



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

Python

```
def sigmoid(n):  
    return 1 / (1 + np.exp(-n))
```

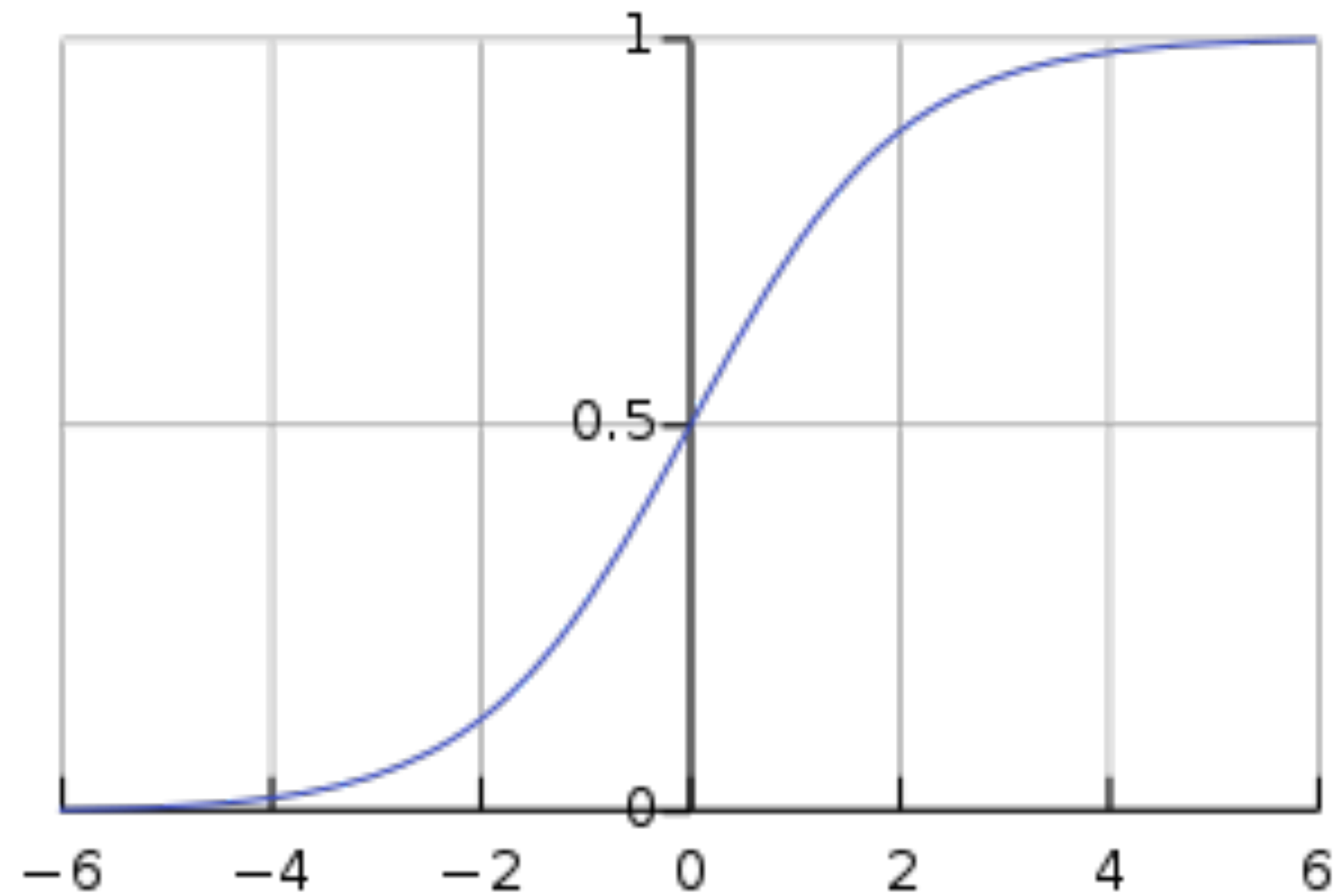
Keras

```
keras.layers.Conv2D(filters=64,  
kernel_size=(3, 3), strides=(1, 1),  
padding='same', activation='sigmoid')
```

```
keras.layers.Dense(units=1024,  
activation='sigmoid')
```

sigmoid

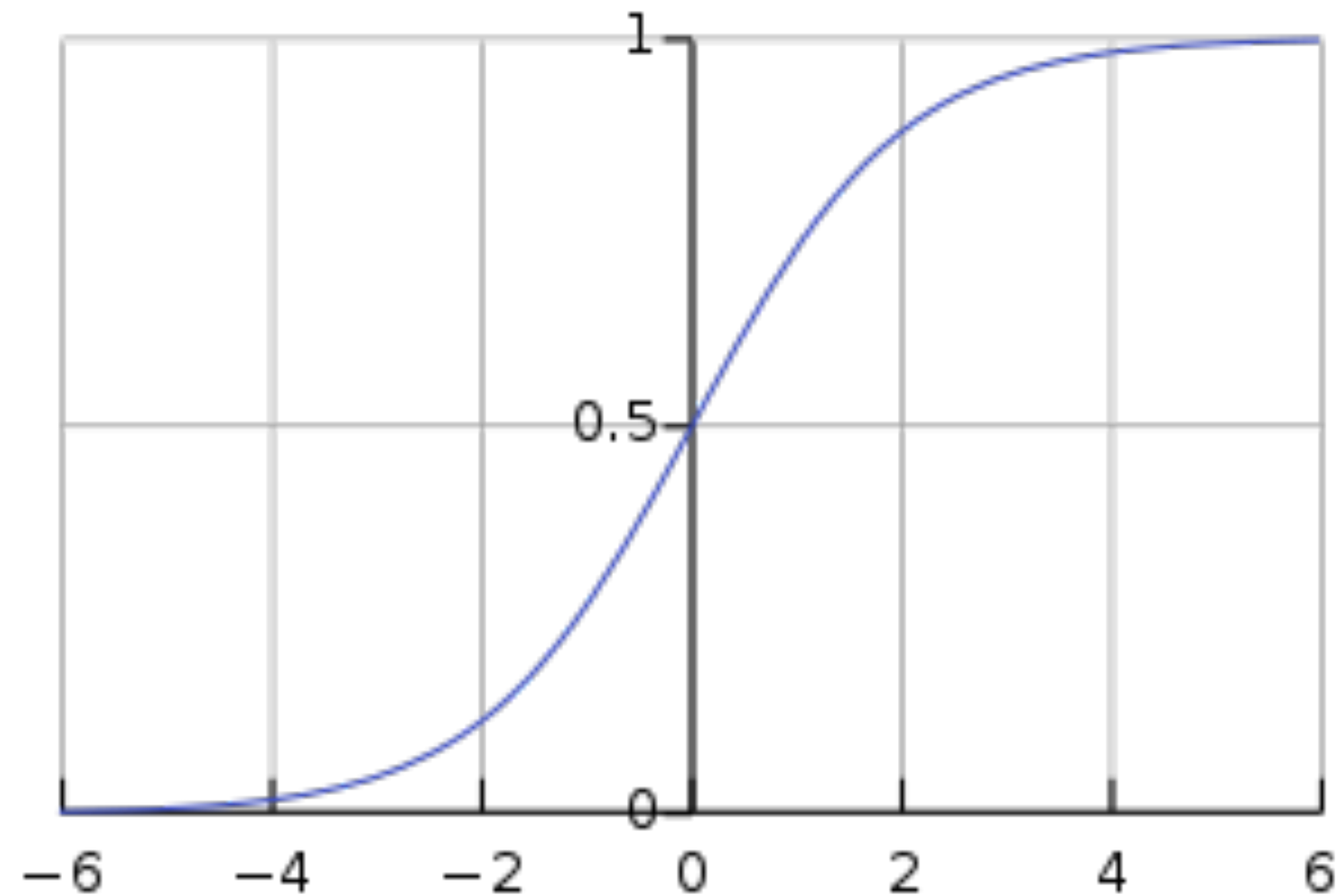
가장 유서깊은 역사를 자랑하는 **Activate Function**
input으로 어떠한 값이 들어가건, 그 결과를 0에서 1 사이로 **squash** 해준다



장점

- 없다. (..)

가장 유서깊은 역사를 자랑하는 **Activate Function**
input으로 어떠한 값이 들어가건, 그 결과를 0에서 1 사이로 **squash** 해준다



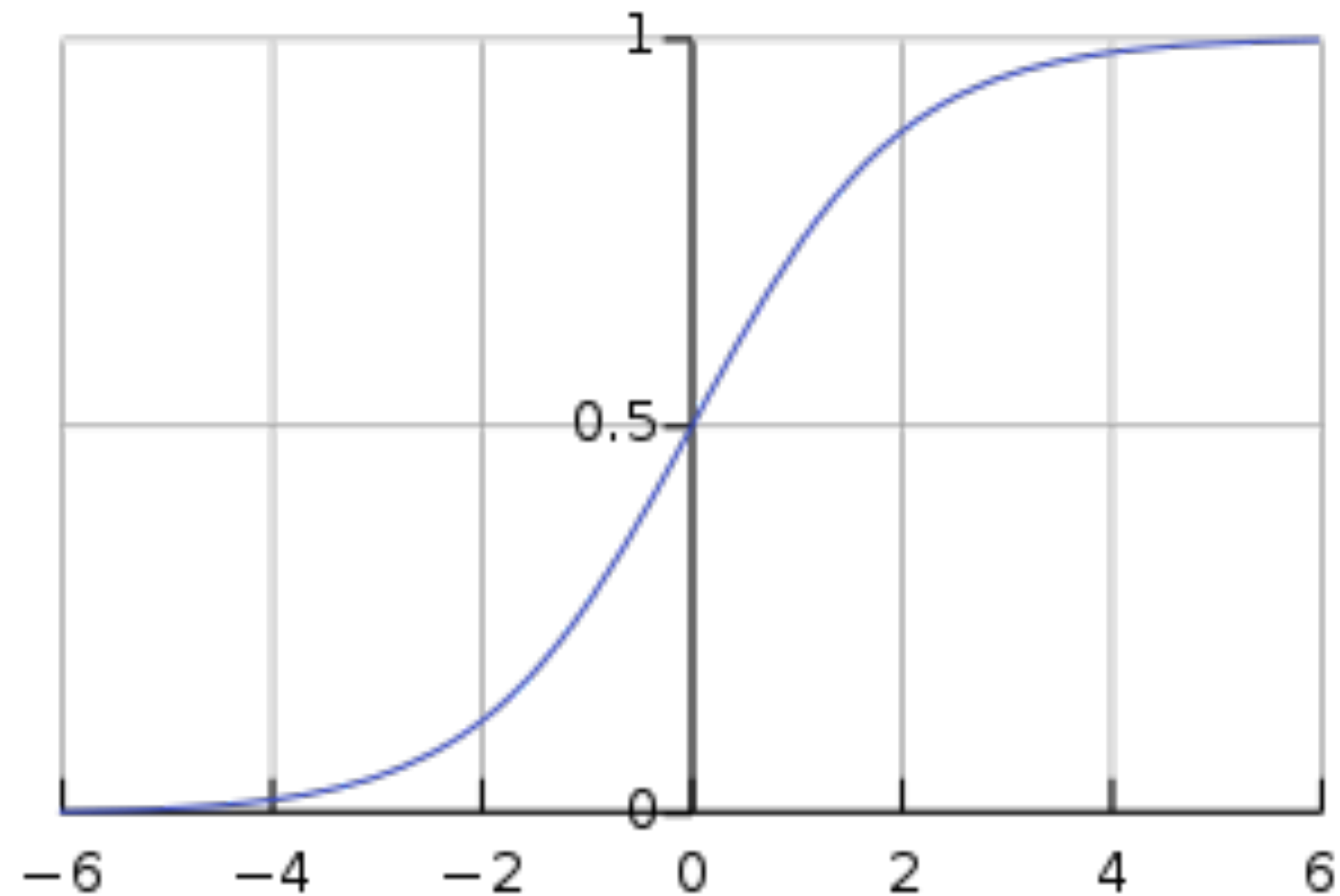
장점

- 없다. (..)

단점

- **Saturation** 현상이 발생한다. (=기울기가 평평하게 되며 gradient가 0이 된다) 만일 상위 레이어의 activate function이 saturated되어버리면, 하위 레이어의 업데이트량은 무조건 0이 된다.

가장 유서깊은 역사를 자랑하는 **Activate Function**
input으로 어떠한 값이 들어가건, 그 결과를 0에서 1 사이로 **squash** 해준다



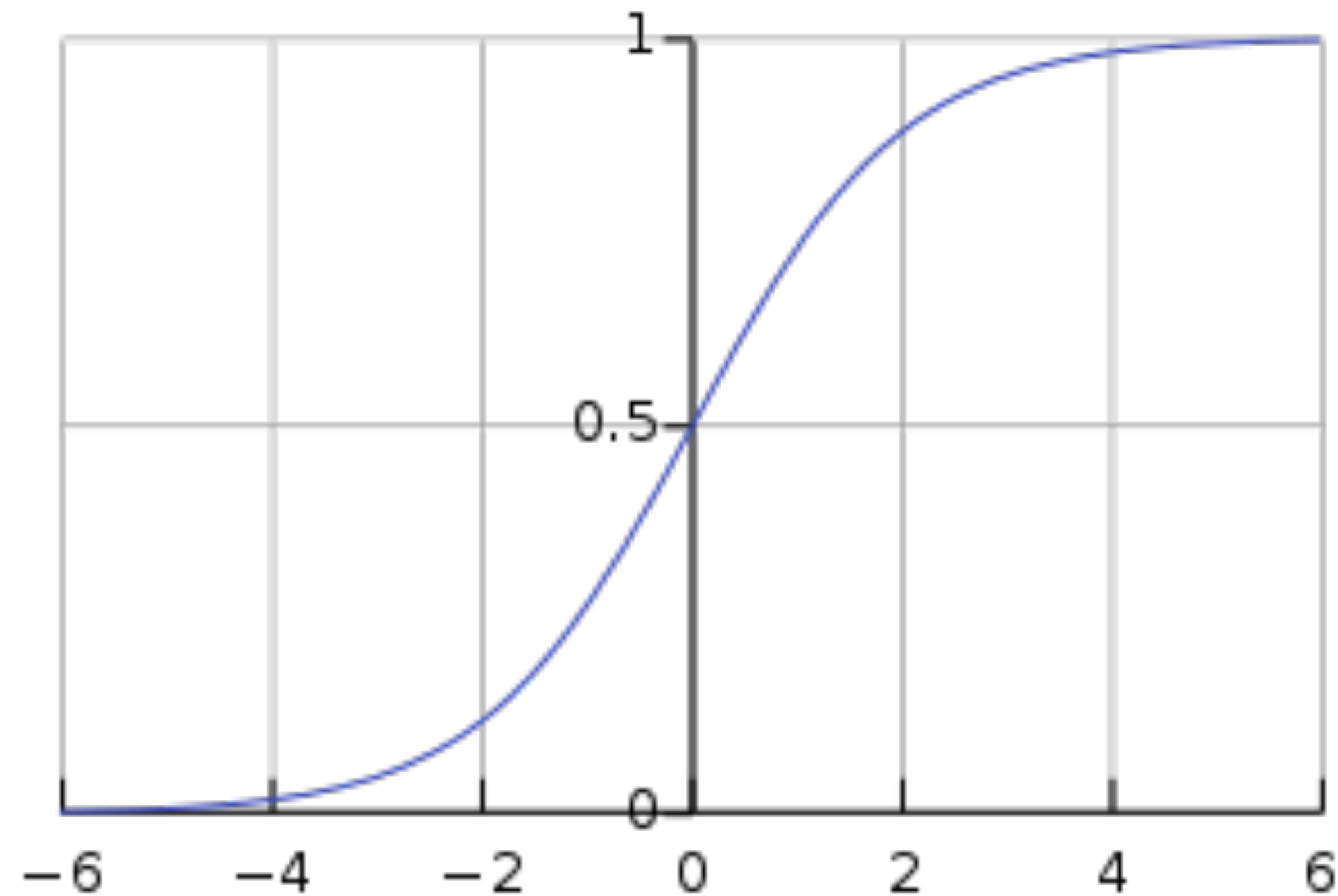
장점

- 없다. (..)

단점

- **Saturation** 현상이 발생한다. (=기울기가 평평하게 되며 gradient가 0이 된다) 만일 상위 레이어의 activate function이 saturated되어버리면, 하위 레이어의 업데이트량은 무조건 0이 된다.
- **결과값이 오직 0~1 사이로만 나온다**: weight를 업데이트 할 때 loss의 부호(+/-)를 바꿔주지 못해서 weight가 zigzag로 업데이트된다. (=업데이트가 굉장히 느려진다)

가장 유서깊은 역사를 자랑하는 **Activate Function**
input으로 어떠한 값이 들어가건, 그 결과를 0에서 1 사이로 **squash** 해준다



장점

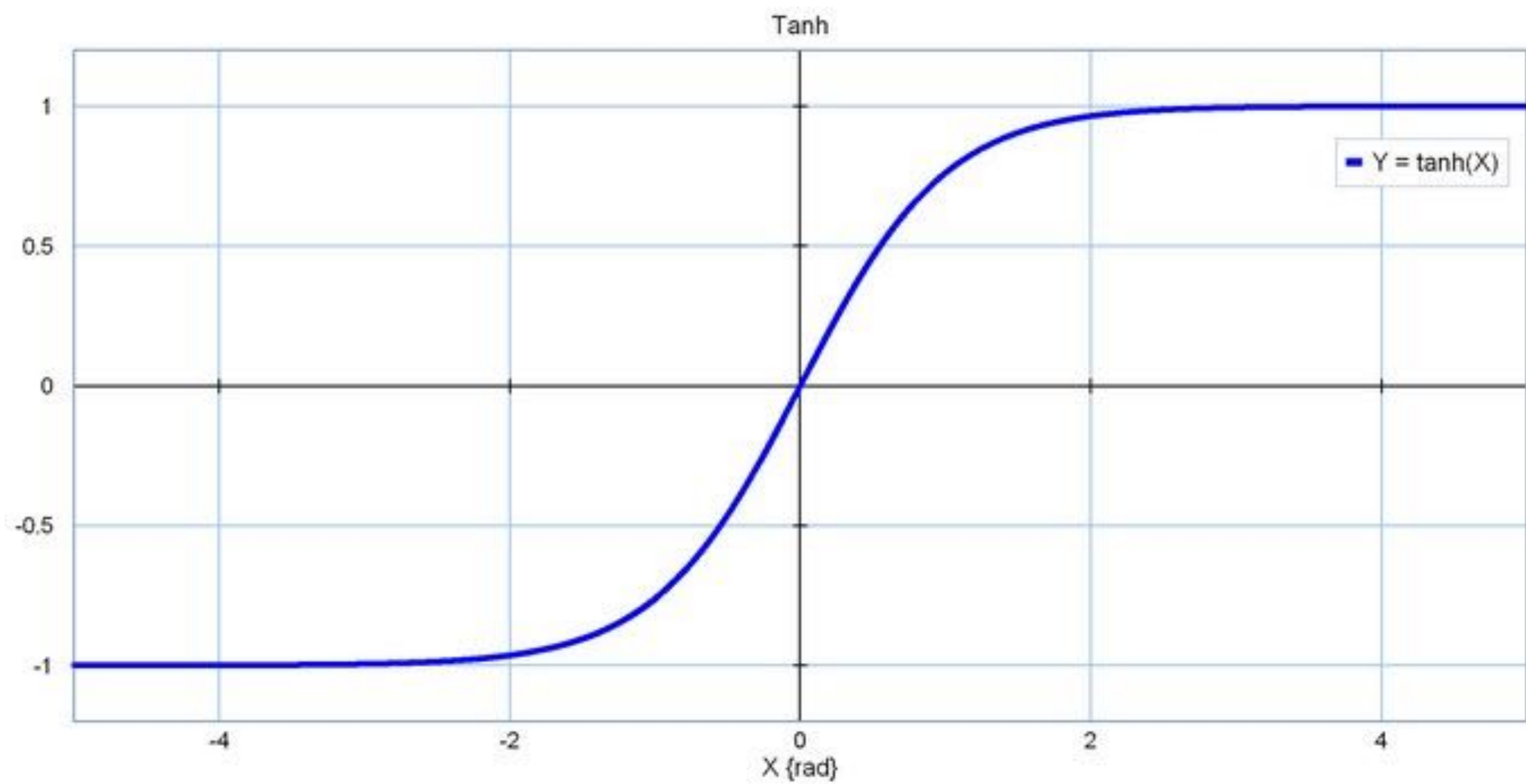
- 없다. (..)

단점

- **Saturation** 현상이 발생한다. (=기울기가 평평하게 되며 gradient가 0이 된다) 만일 상위 레이어의 activate function이 saturated되어버리면, 하위 레이어의 업데이트량은 무조건 0이 된다.
- **결과값이 오직 0~1 사이로만 나온다**: weight를 업데이트 할 때 loss의 부호(+/-)를 바꿔주지 못해서 weight가 zigzag로 업데이트된다. (=업데이트가 굉장히 느려진다)
- exp 연산이 굉장히 느리다.

hyperbolic tangent(tanh)

sigmoid의 단점을 개선하기 위해 만든 Activate Function
sigmoid와는 달리 결과를 -1에서 1 사이로 squash 해준다

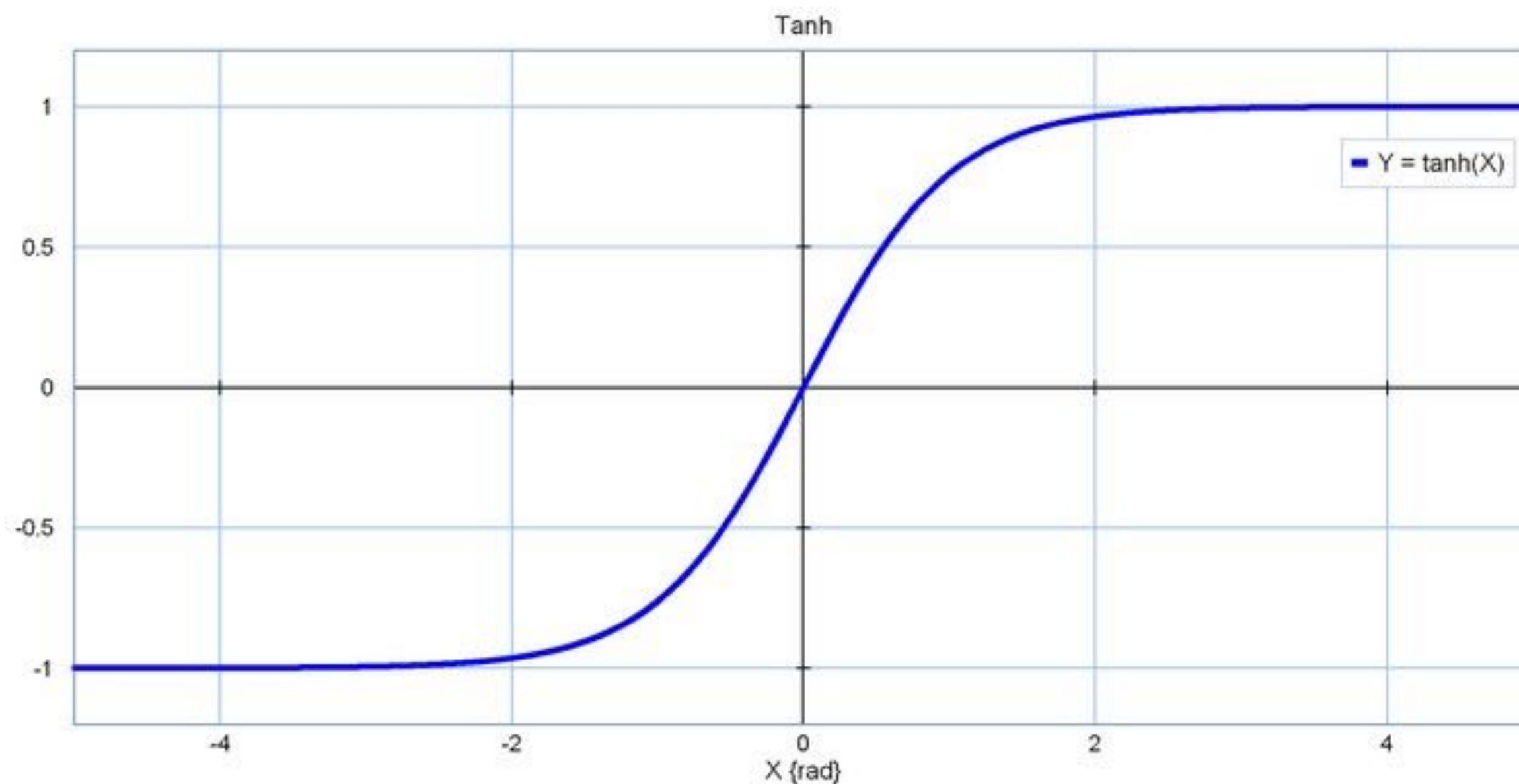


$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$

hyperbolic tangent(tanh)

sigmoid의 단점을 개선하기 위해 만든 Activate Function
sigmoid와는 달리 결과를 -1에서 1 사이로 squash 해준다

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$



```
def tanh(n):  
    numerator = np.exp(n) - np.exp(-n)  
    denominator = np.exp(n) + np.exp(-n)  
  
    return numerator / denominator
```

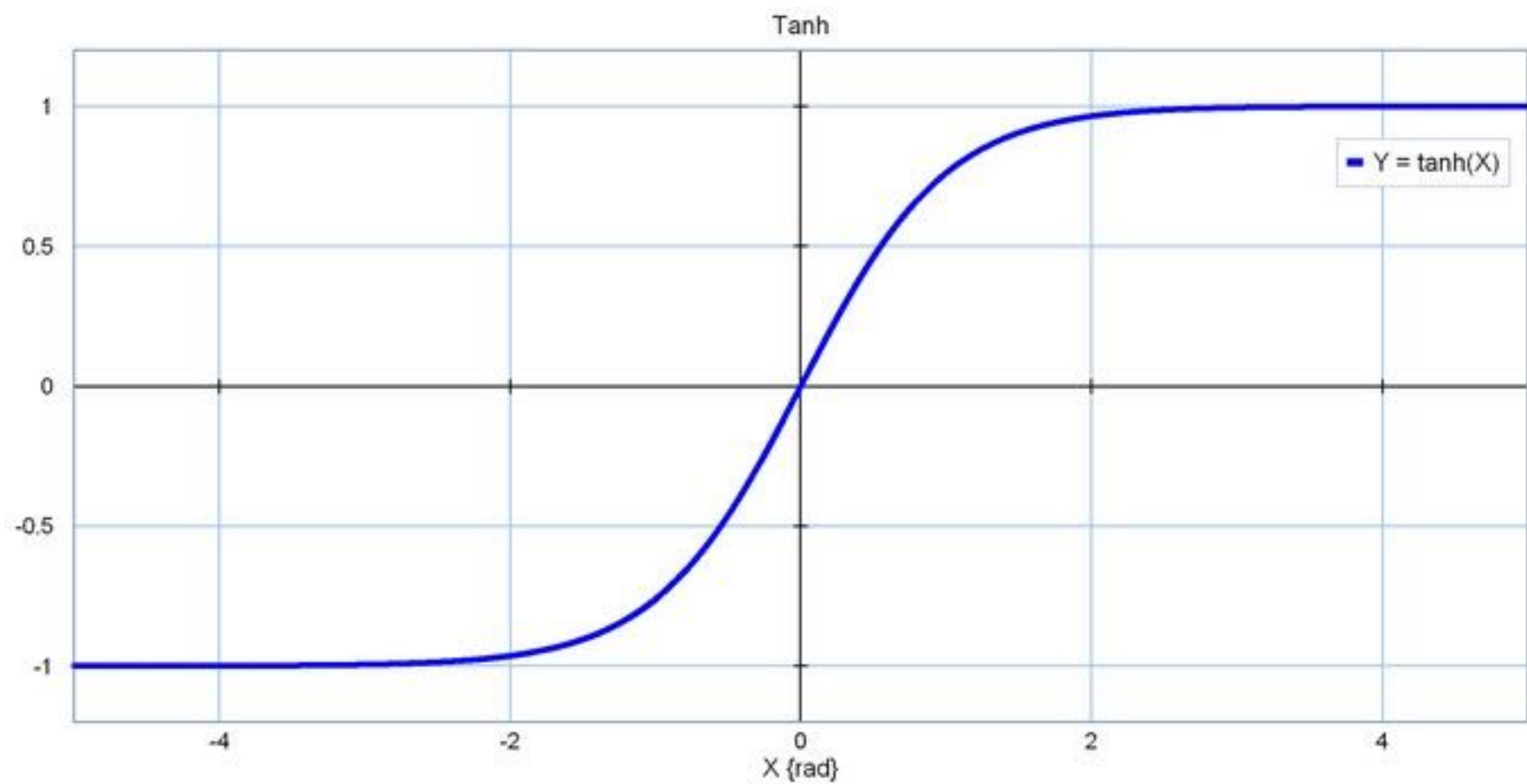
Keras

```
keras.layers.Conv2D(filters=64,  
    kernel_size=(3, 3), strides=(1, 1),  
    padding='same', activation='tanh')
```

```
keras.layers.Dense(units=1024,  
    activation='tanh')
```

hyperbolic tangent(tanh)

sigmoid의 단점을 개선하기 위해 만든 Activate Function
sigmoid와는 달리 결과를 -1에서 1 사이로 squash 해준다

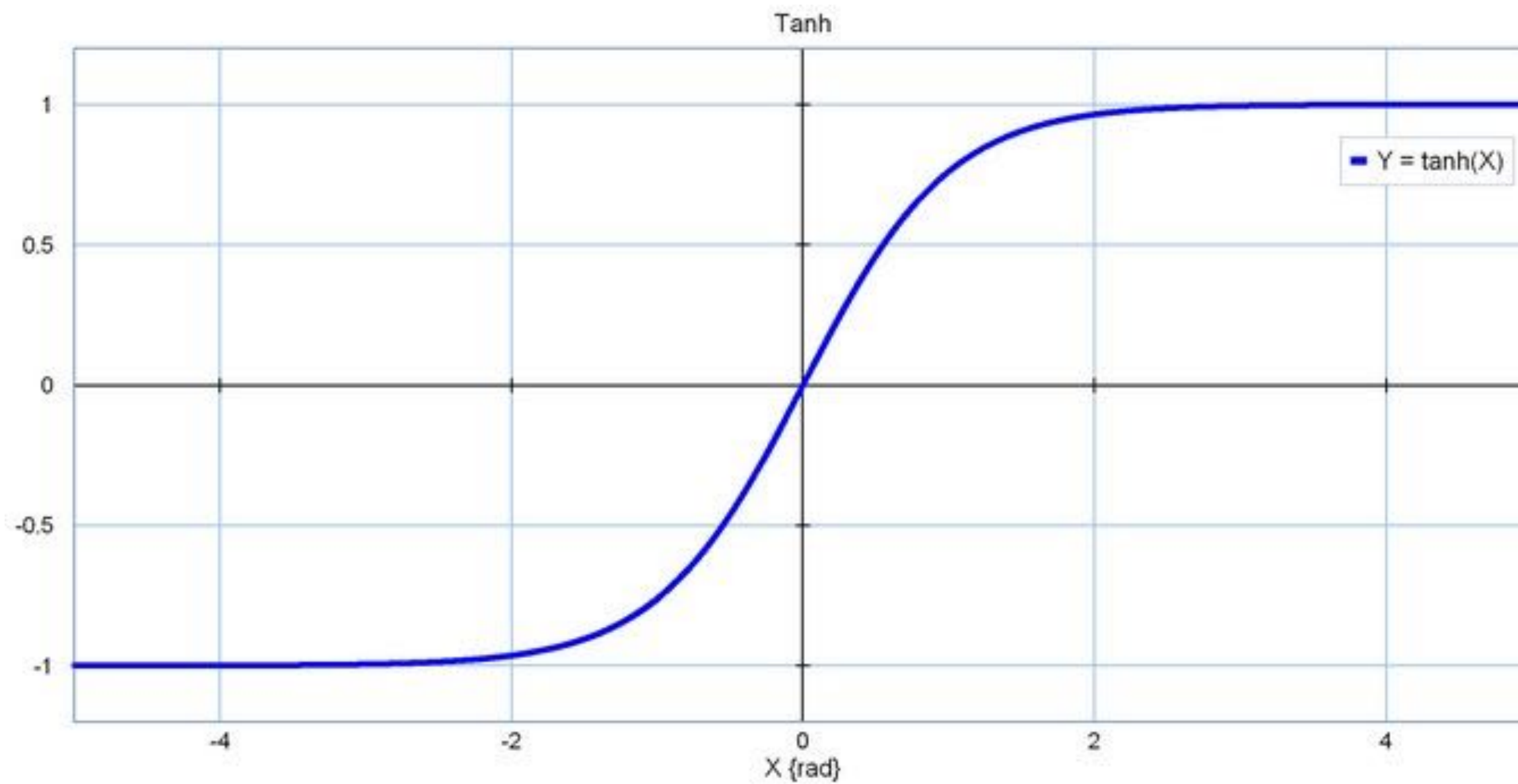


장점

- 결과값이 -1 ~ 1 사이로만 나온다: sigmoid와는 달리 weight가 zigzag로 업데이트하지 않는다.

hyperbolic tangent(tanh)

sigmoid의 단점을 개선하기 위해 만든 Activate Function
sigmoid와는 달리 결과를 -1에서 1 사이로 squash 해준다



장점

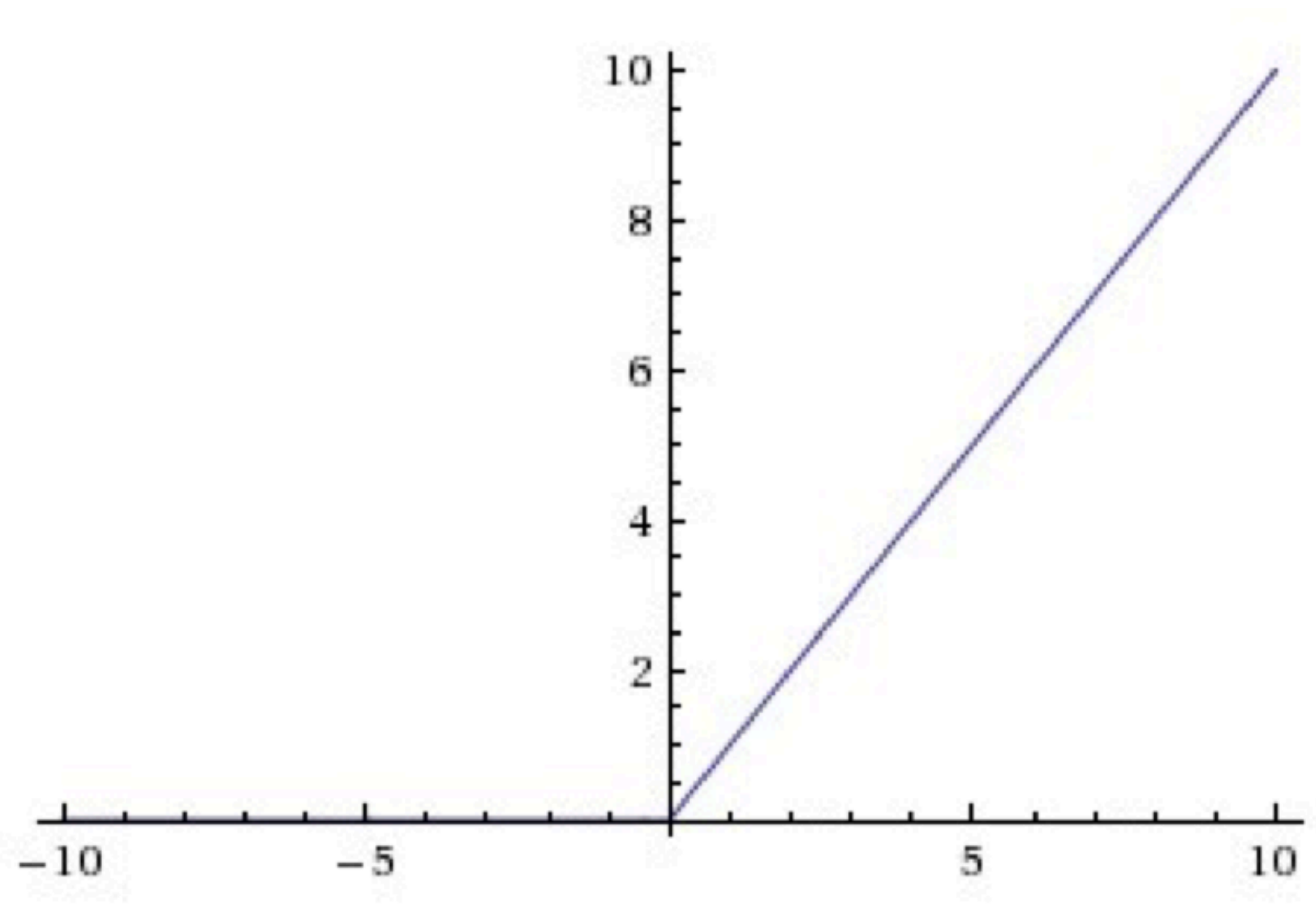
- 결과값이 -1 ~ 1 사이로만 나온다: sigmoid와는 달리 weight가 zigzag로 업데이트하지 않는다.

단점

- 변함없이 **Saturation** 현상이 발생한다.
- sigmoid보다 더 많은 exp 연산을 한다.

Rectified Linear Unit(ReLU)

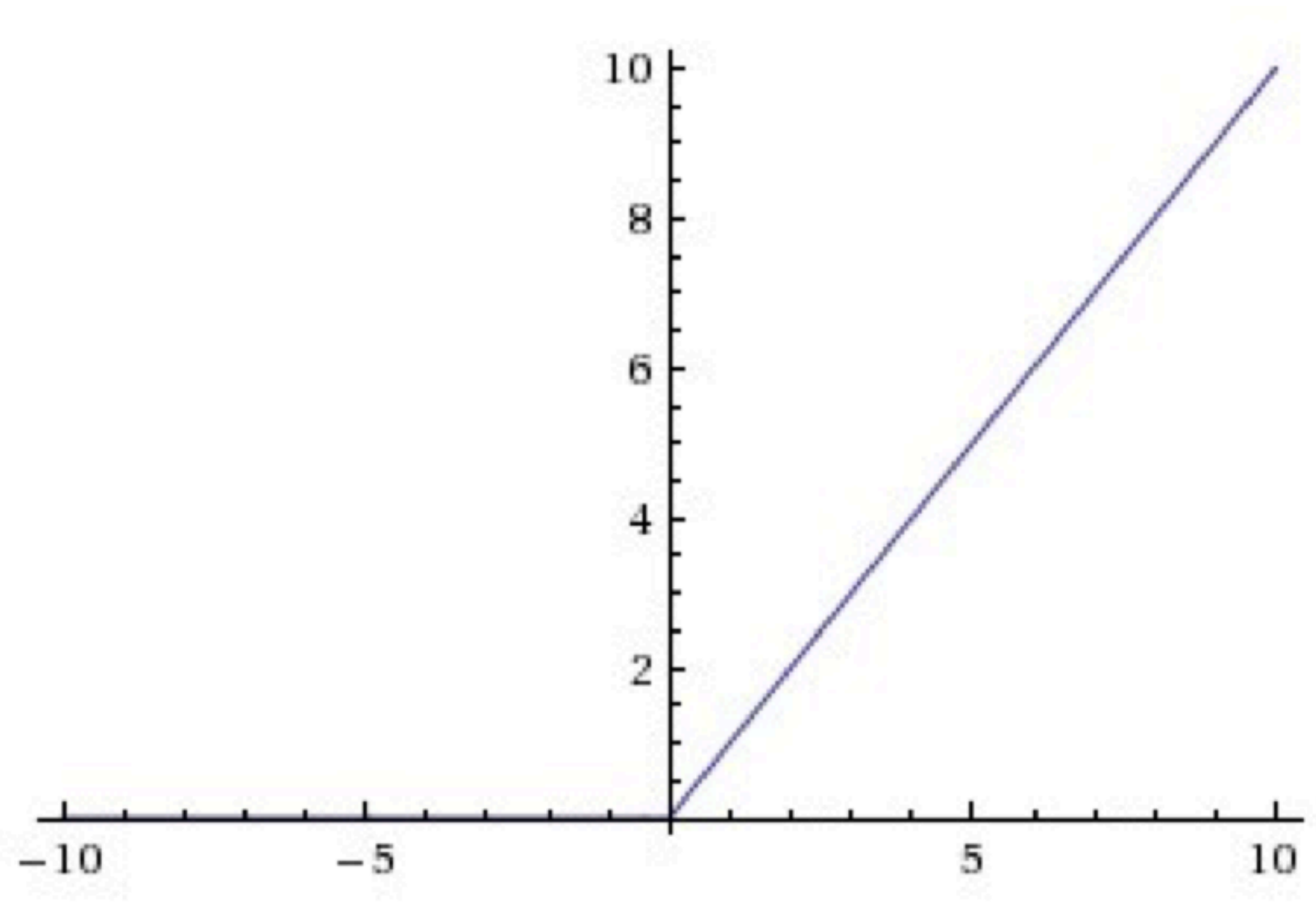
가장 대중적으로 많이 쓰이는 **Activate Function**
x가 양수면 그대로 놔두고, 음수면 0으로 만든다



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

Rectified Linear Unit(ReLU)

가장 대중적으로 많이 쓰이는 Activate Function
x가 양수면 그대로 놔두고, 음수면 0으로 만든다



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

Python

```
def relu(n):  
    return n * (n > 0)
```

Keras

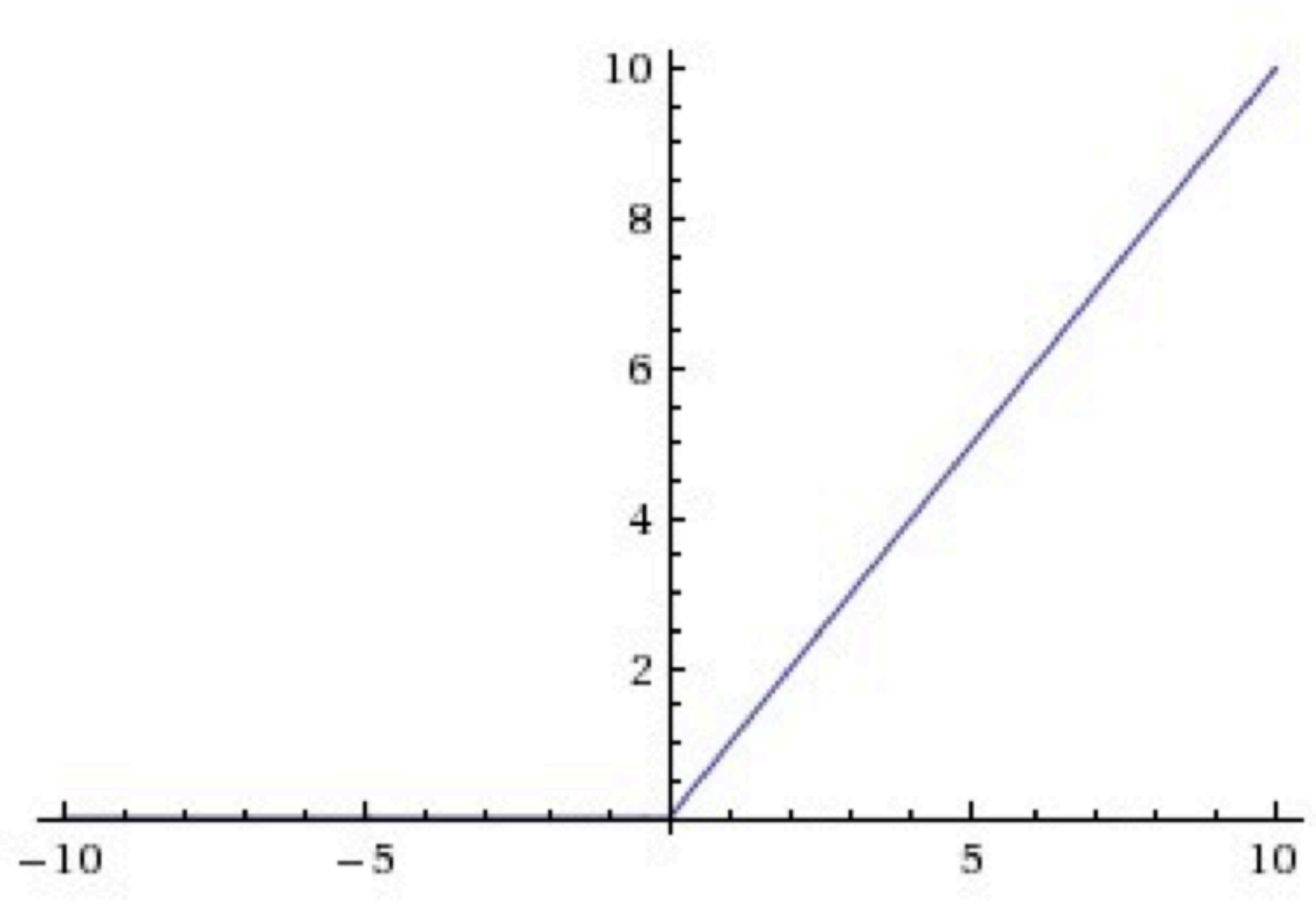
```
keras.layers.Conv2D(filters=64,  
    kernel_size=(3, 3), strides=(1, 1),  
    padding='same', activation='relu')
```

```
keras.layers.Dense(units=1024,  
    activation='relu')
```

ImageNet Classification with Deep Convolutional Neural Networks
Krizhevsky et al, 2012. <https://goo.gl/hqAxZD>

Rectified Linear Unit(ReLU)

가장 대중적으로 많이 쓰이는 **Activate Function**
 x 가 양수면 그대로 놔두고, 음수면 0으로 만든다

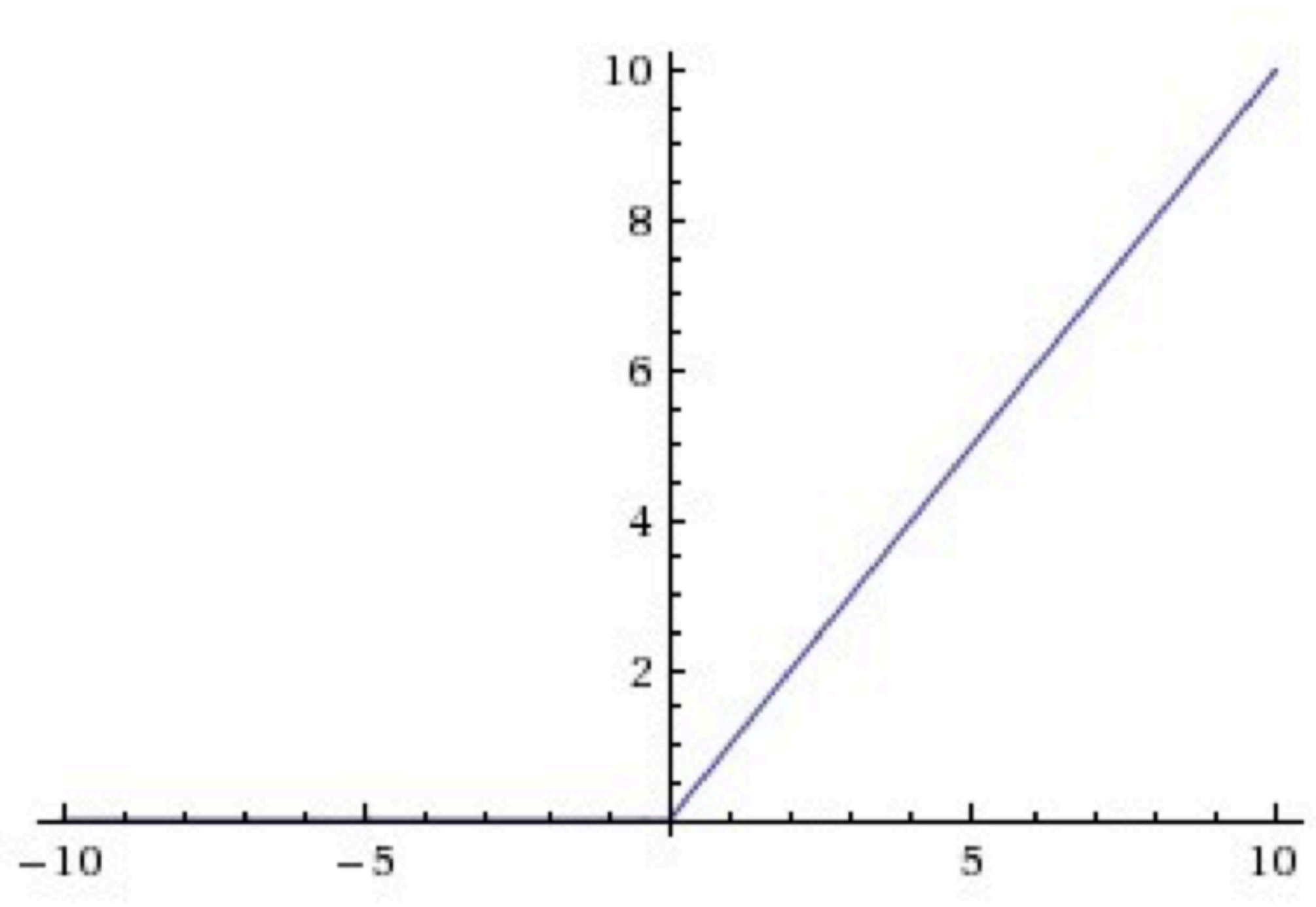


장점

- x 값이 플러스라면 **saturate** 하지 않는다.
- 공식이 굉장히 간결하다. (= 속도가 매우 빠르다)
- 실제 써보면 sigmoid/tanh보다 loss가 수렴하는 속도가 매우 빠르다. (6배 정도?)

Rectified Linear Unit(ReLU)

가장 대중적으로 많이 쓰이는 **Activate Function**
 x 가 양수면 그대로 놔두고, 음수면 0으로 만든다



장점

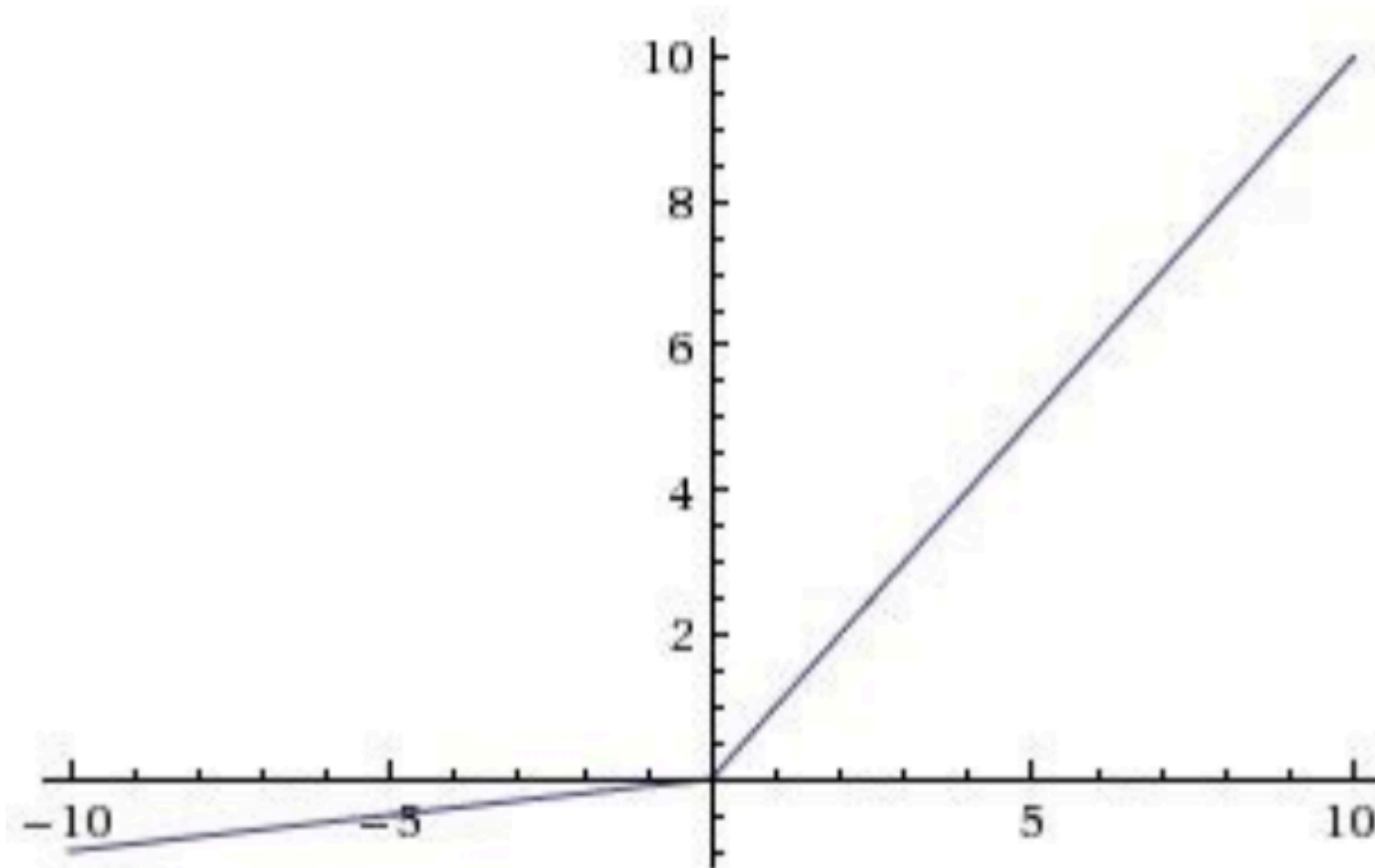
- x 값이 플러스라면 **saturate** 하지 않는다.
- 공식이 굉장히 간결하다. (= 속도가 매우 빠르다)
- 실제 써보면 sigmoid/tanh보다 loss가 수렴하는 속도가 매우 빠르다. (6배 정도?)

단점

- 결과값이 언제나 플러스이다. (zigzag 할 수도 있다)

Leaky ReLU(LReLU) & Parametric ReLU(PReLU)

ReLU가 음수에서 saturate되는 현상을 막기 위해
음수일 경우에도 아주 작은 값을 준다.



Leaky ReLU(LReLU)

$$f(x) = \max(0.01x, x)$$

Parametric ReLU(PReLU)

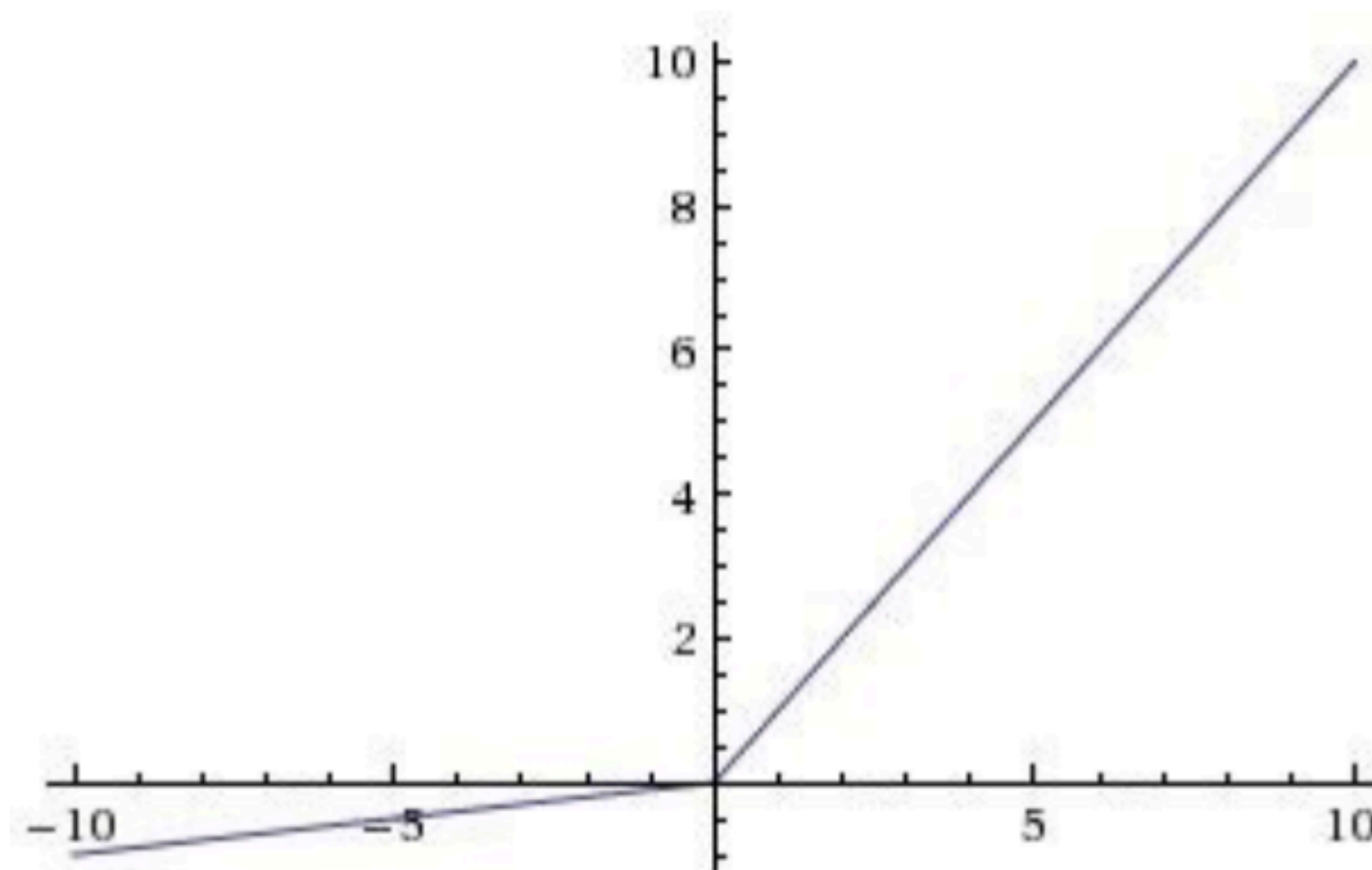
$$f(x) = \max(\alpha x, x)$$

Rectifier Nonlinearities Improve Neural Network Acoustic Models
Mass et al, 2013. <https://goo.gl/rsMJhx>

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification
He et al, 2015. <https://goo.gl/FmhxUv>

Leaky ReLU(LReLU) & Parametric ReLU(PReLU)

ReLU가 음수에서 saturate되는 현상을 막기 위해
음수일 경우에도 아주 작은 값을 준다.



Leaky ReLU(LReLU)

$$f(x) = \max(0.01x, x)$$

언제나 고정된 값이다

Parametric ReLU(PReLU)

$$f(x) = \max(\alpha x, x)$$

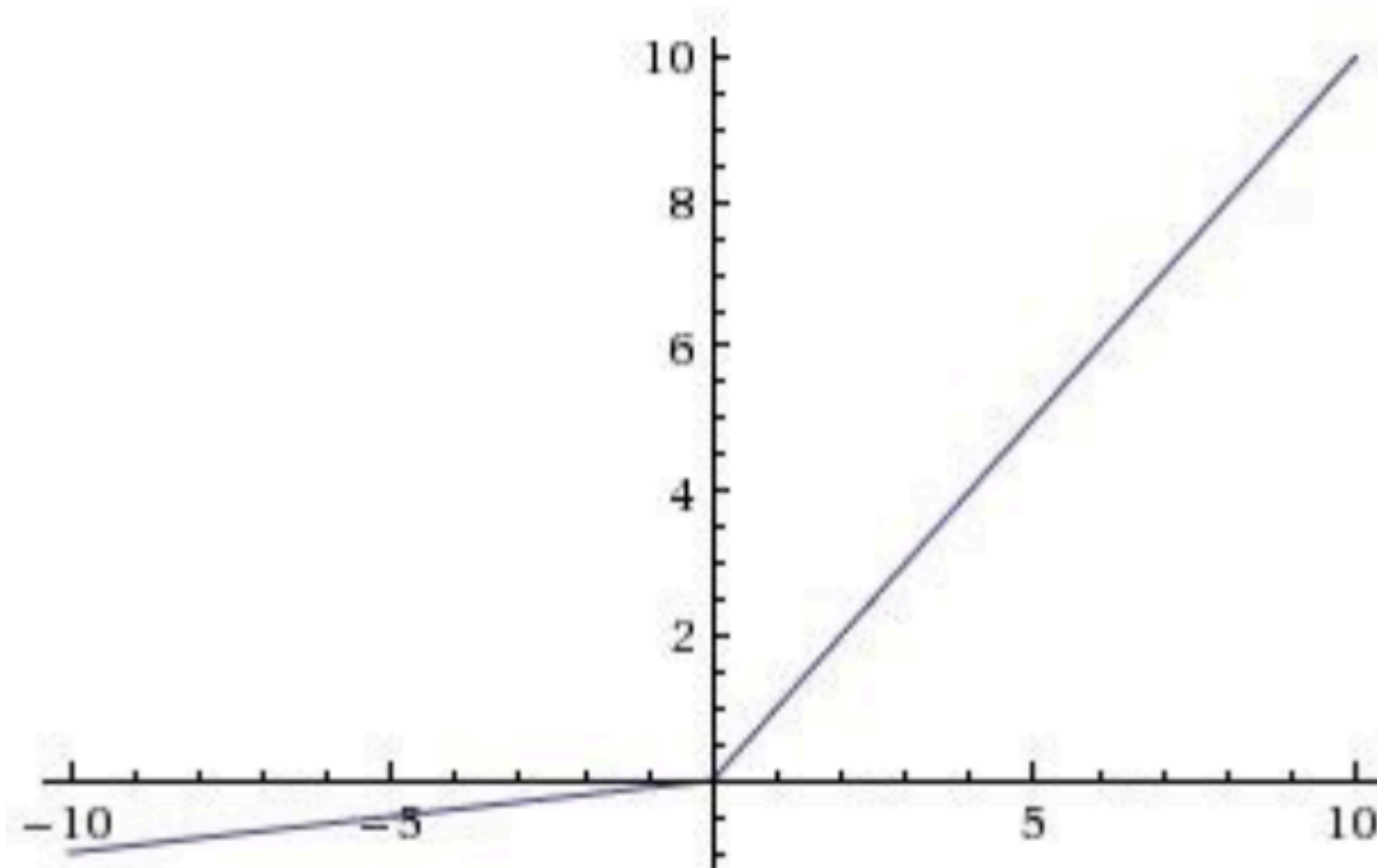
backpropagation을 할 때
weight처럼 업데이트 해준다

Rectifier Nonlinearities Improve Neural Network Acoustic Models
Mass et al, 2013. <https://goo.gl/rsMJhx>

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification
He et al, 2015. <https://goo.gl/FmhxUv>

Leaky ReLU(LReLU) & Parametric ReLU(PReLU)

ReLU가 음수에서 saturate되는 현상을 막기 위해
음수일 경우에도 아주 작은 값을 준다.



장점

- saturate 하지 않는다.
- 연산이 빠르다. (exp가 없으므로)
- 변함없이 sigmoid/tanh보다 loss가 수렴하는 속도가 매우 빠르다. (6배 정도?)

단점

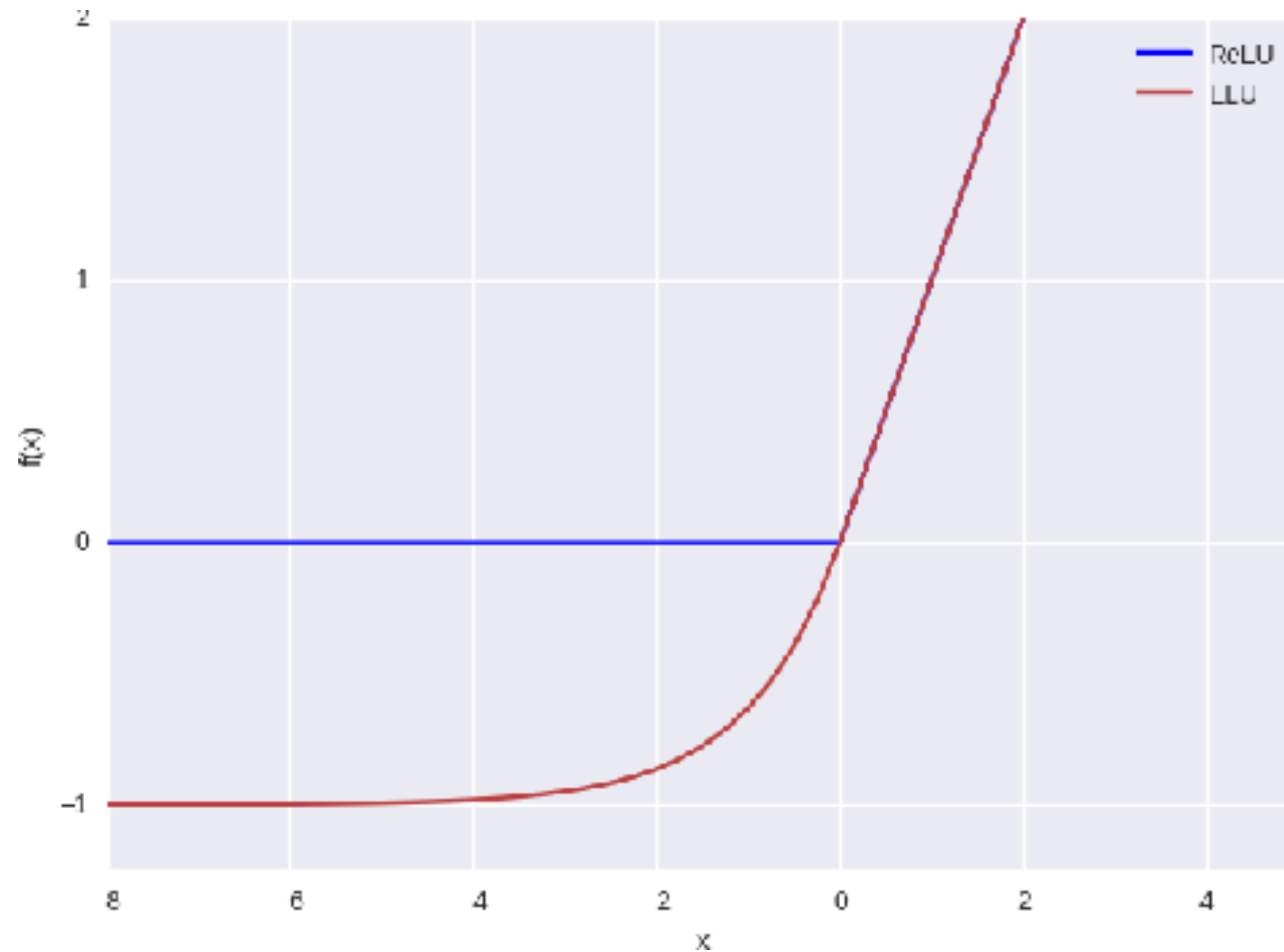
- 지금까지는 딱히... (그렇다고 언제나 100% 높은 성능을 보장하는 건 아니다)

Rectifier Nonlinearities Improve Neural Network Acoustic Models
Mass et al, 2013. <https://goo.gl/rsMJhx>

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification
He et al, 2015. <https://goo.gl/FmhxUv>

Exponential Linear Unit (ELU)

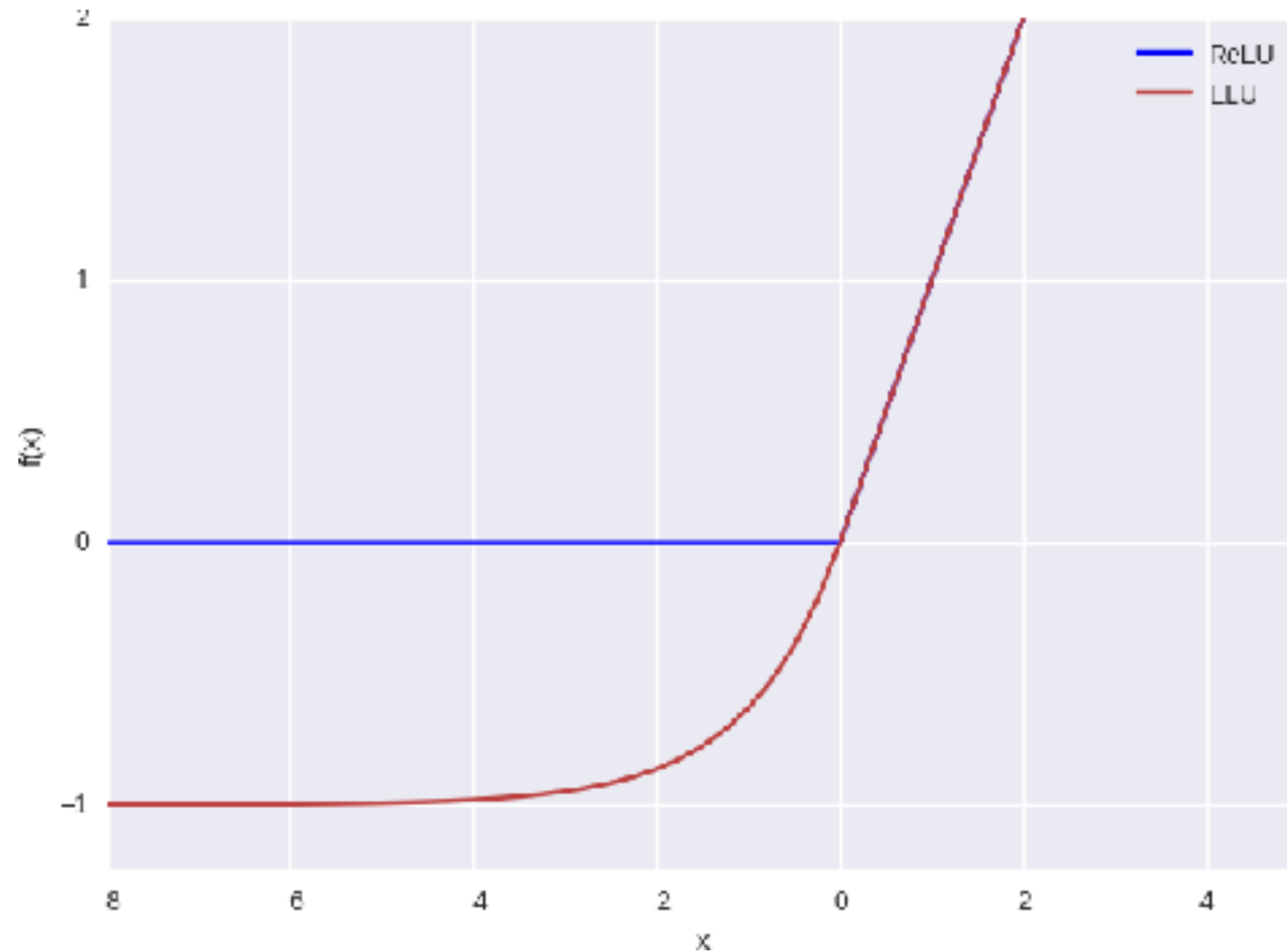
ReLU의 gradient가 smooth하지 않은 현상을 해결하기 위해
exp 연산을 사용한다.



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

Exponential Linear Unit (ELU)

ReLU의 gradient가 smooth하지 않은 현상을 해결하기 위해 exp 연산을 사용한다.

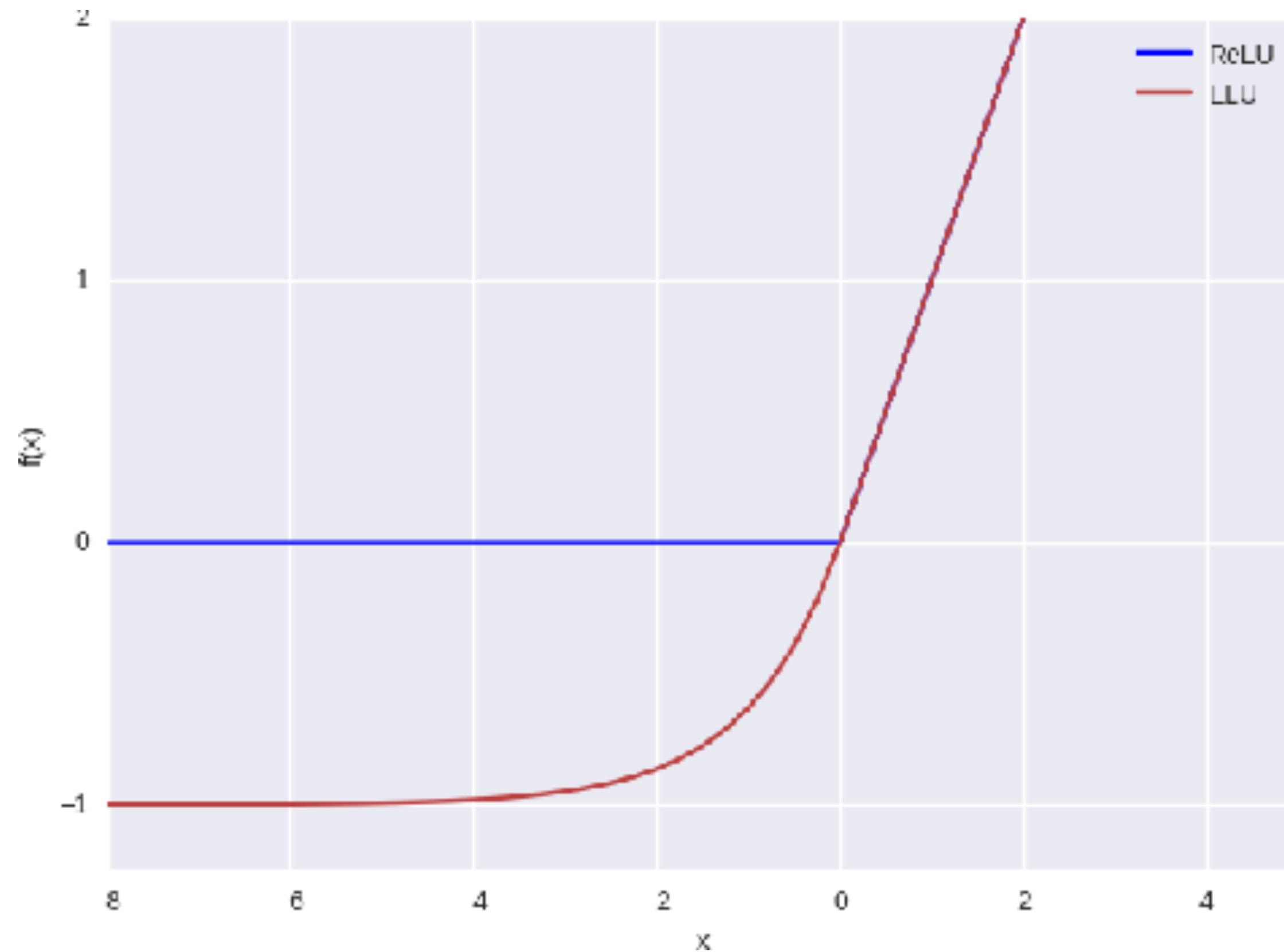


장점

- Gradient가 smooth하다.
- 결과값이 마이너스도 나온다. (=덜 zigzag 한다)
- sigmoid/tanh보다 loss가 수렴하는 속도가 매우 빠르다.

Exponential Linear Unit (ELU)

ReLU의 gradient가 smooth하지 않은 현상을 해결하기 위해 exp 연산을 사용한다.



장점

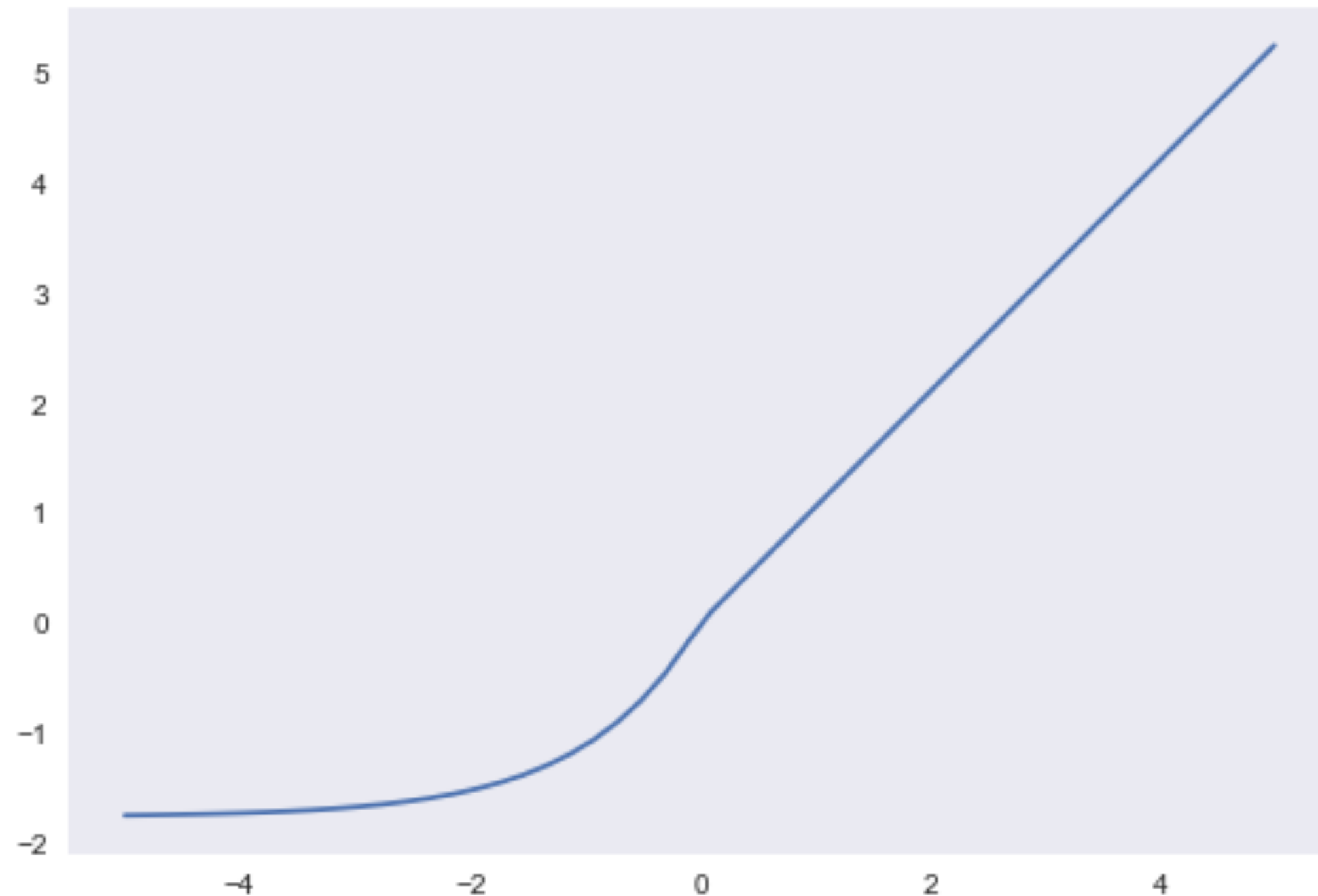
- Gradient가 smooth하다.
- 결과값이 마이너스도 나온다. (=덜 zigzag 한다)
- sigmoid/tanh보다 loss가 수렴하는 속도가 매우 빠르다.

단점

- exp연산이 필요하다.

Scaled Exponential Linear Unit (SELU)

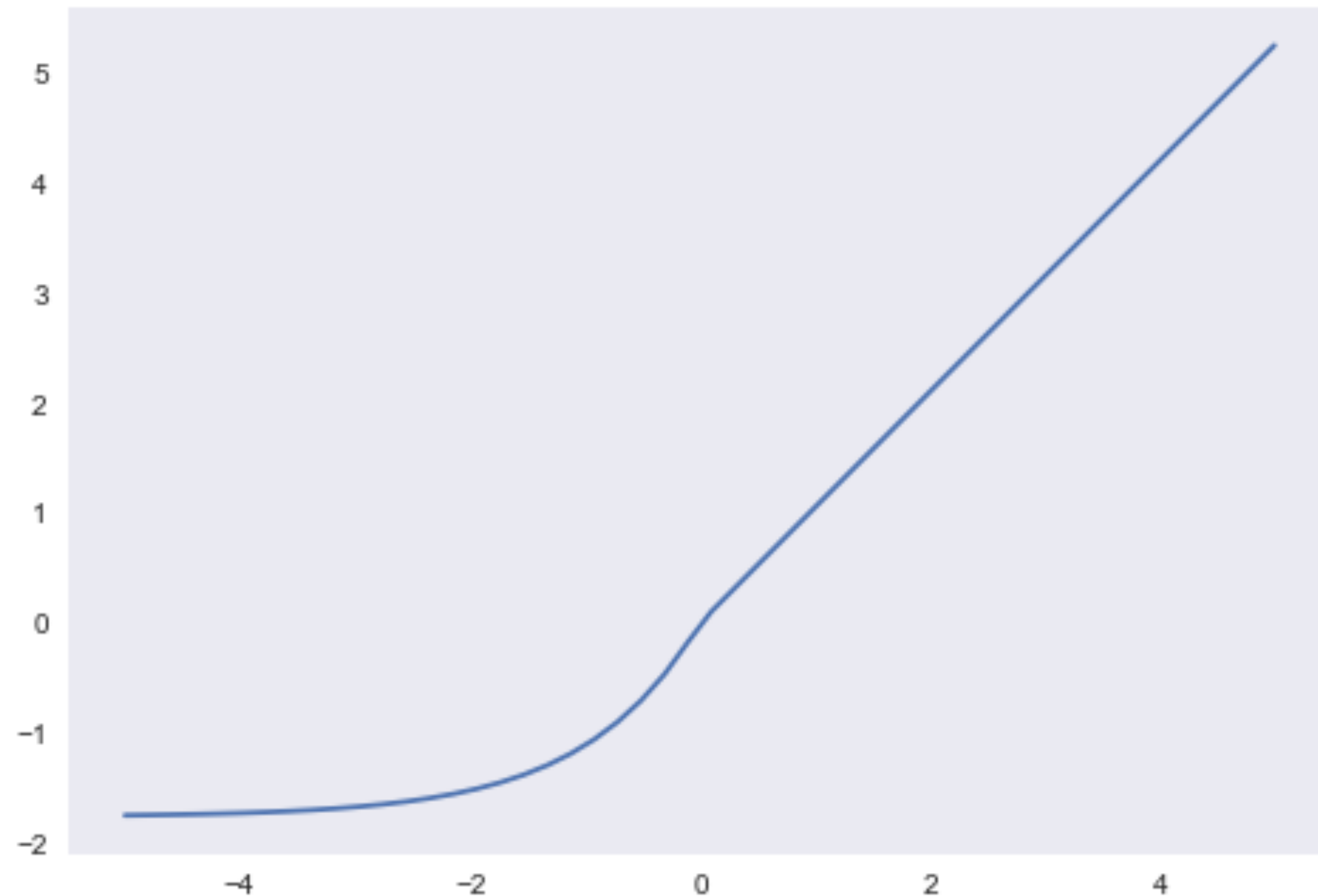
ELU에서 두 개의 하이퍼패러미터(λ , α)를 추가.
이 두 개가 output을 자동으로 normalized 함으로써 레이어를 더 깊게 쌓을 수 있게 한다.



$$\text{selu}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leq 0 \end{cases}$$

Scaled Exponential Linear Unit (SELU)

ELU에서 두 개의 하이퍼패러미터(λ , α)를 추가.
이 두 개가 output을 자동으로 normalized 함으로써 레이어를 더 깊게 쌓을 수 있게 한다.

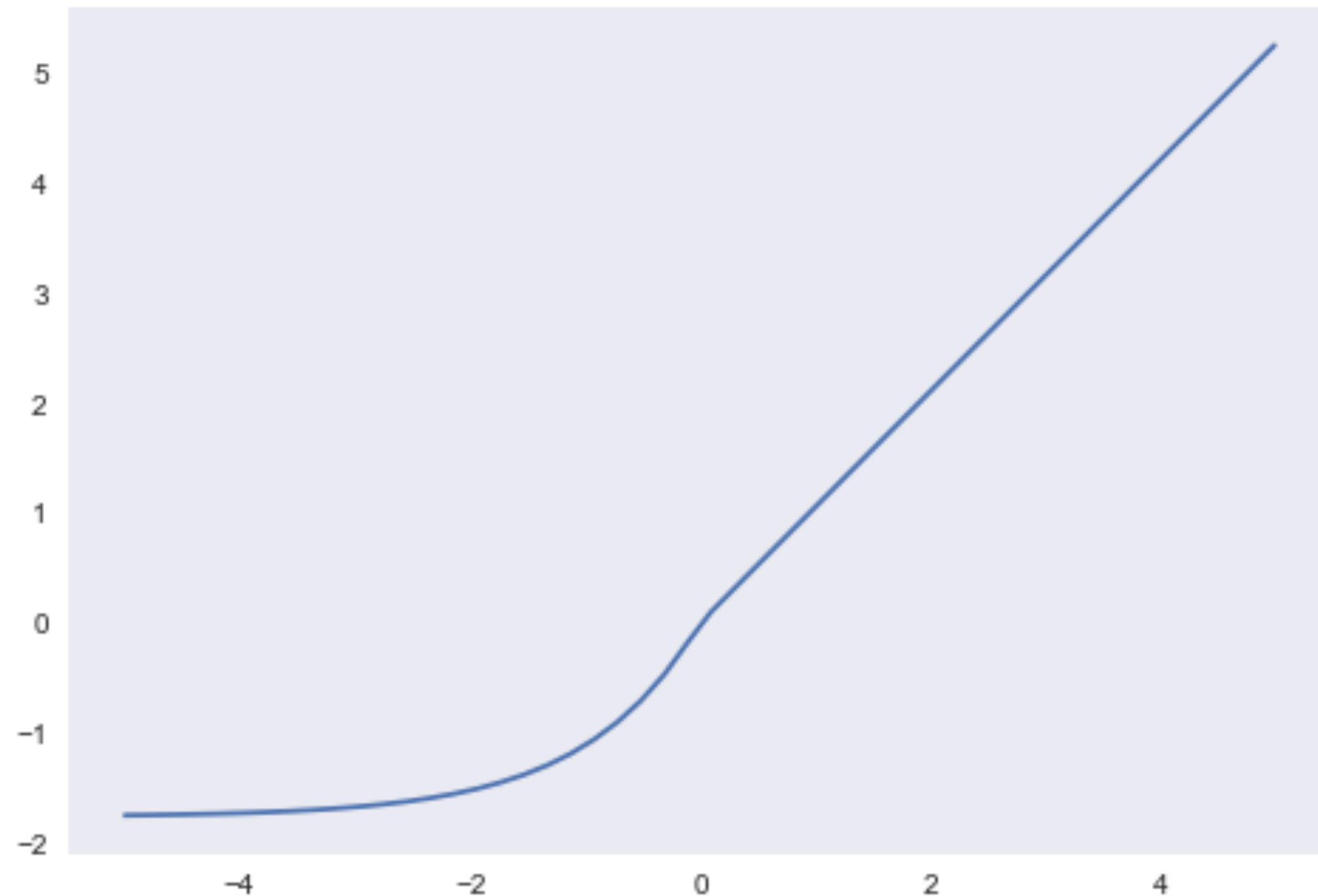


장점

- Gradient가 smooth하다.
- 결과값이 마이너스도 나온다. (=덜 zigzag 한다)
- sigmoid/tanh보다 loss가 수렴하는 속도가 매우 빠르다.
- **output이 스스로 normalized 되면서 saturated 되는 현상을 방지한다.**

Scaled Exponential Linear Unit (SELU)

ELU에서 두 개의 하이퍼패러미터(λ , α)를 추가.
이 두 개가 output을 자동으로 normalized 함으로써 레이어를 더 깊게 쌓을 수 있게 한다.



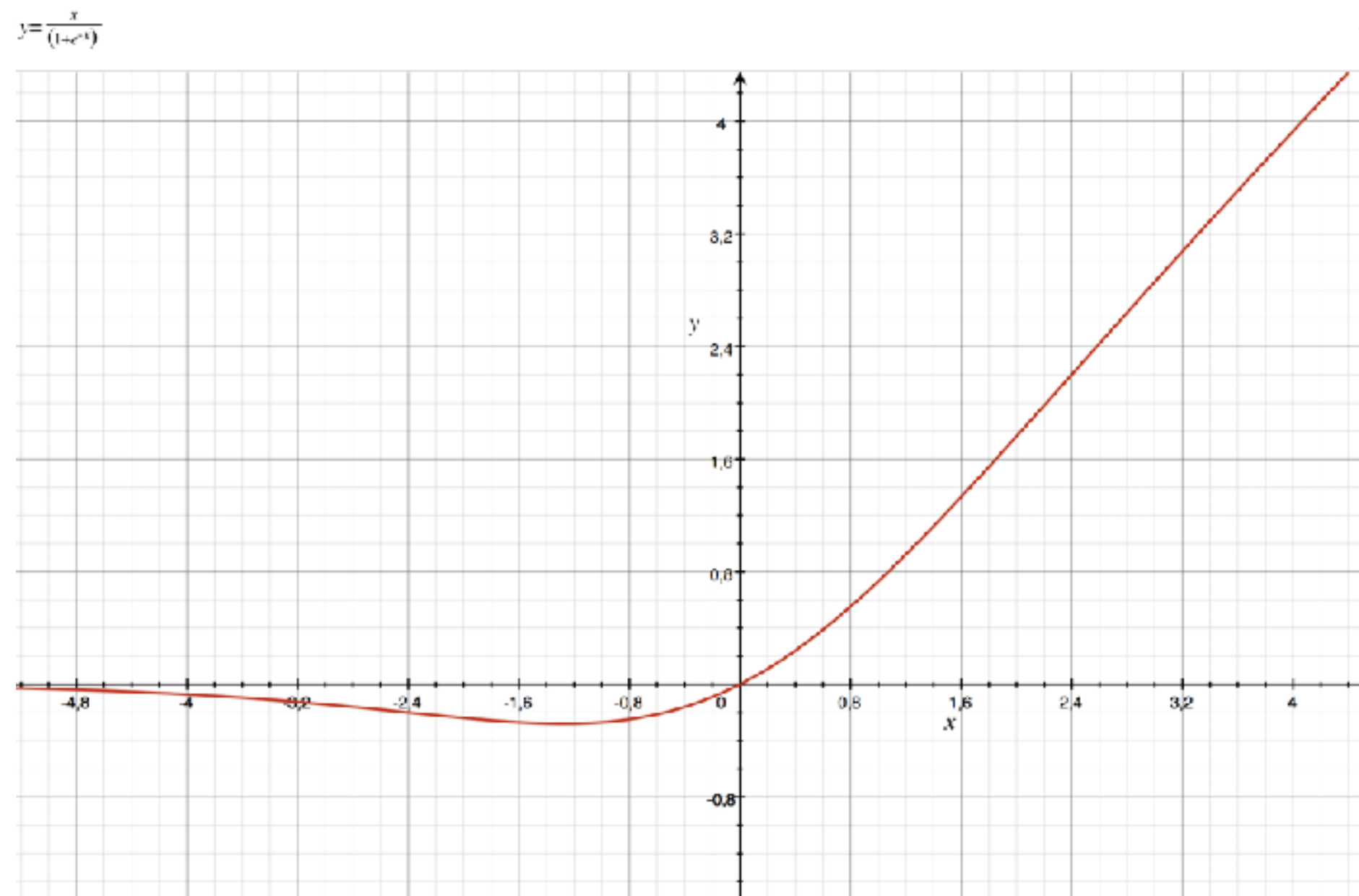
장점

- Gradient가 smooth하다.
- 결과값이 마이너스도 나온다. (=덜 zigzag 한다)
- sigmoid/tanh보다 loss가 수렴하는 속도가 매우 빠르다.
- **output이 스스로 normalized 되면서 saturated 되는 현상을 방지한다.**

단점

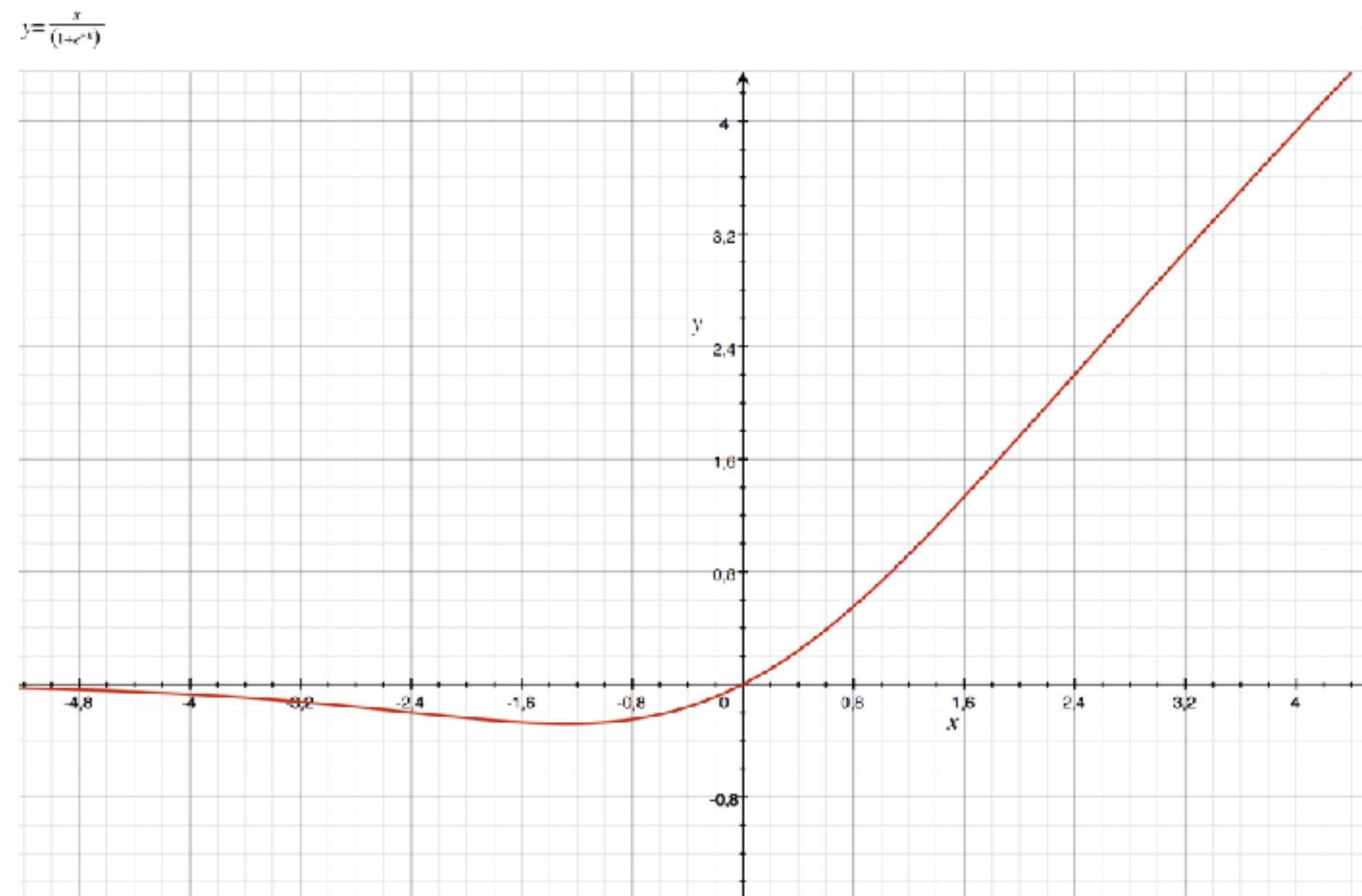
- exp연산이 필요하다.

가장 최근에 나온 activate function.
ReLU와 유사하나 조금 더 smooth하고 0과 근접한 마이너스 부분이 가장 낮은게 특징.



$$f(x) = x * \text{sigmoid}(x)$$

가장 최근에 나온 activate function.
ReLU와 유사하나 조금 더 smooth하고 0과 근접한 마이너스 부분이 가장 낮은게 특징.



최근에 나온 activate fucntion이기 때문에,
벤치마킹 결과가 상대적으로 부족하다.

- 1) 하나의 layer에서 하나가 아닌 두 개의 weight를 보유한다.
- 2) 두 개의 $WX + b$ 를 한다.
- 3) 둘 중에 높은 값을 고른다.

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

장점

- 두 개의 선형 연산이기 때문에 saturate 하지 않는다.
- 언제나 꽤 좋은 성능을 보장한다. (=경진대회 용으로 좋다)

- 1) 하나의 layer에서 하나가 아닌 두 개의 weight를 보유한다.
- 2) 두 개의 $WX + b$ 를 한다.
- 3) 둘 중에 높은 값을 고른다.

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

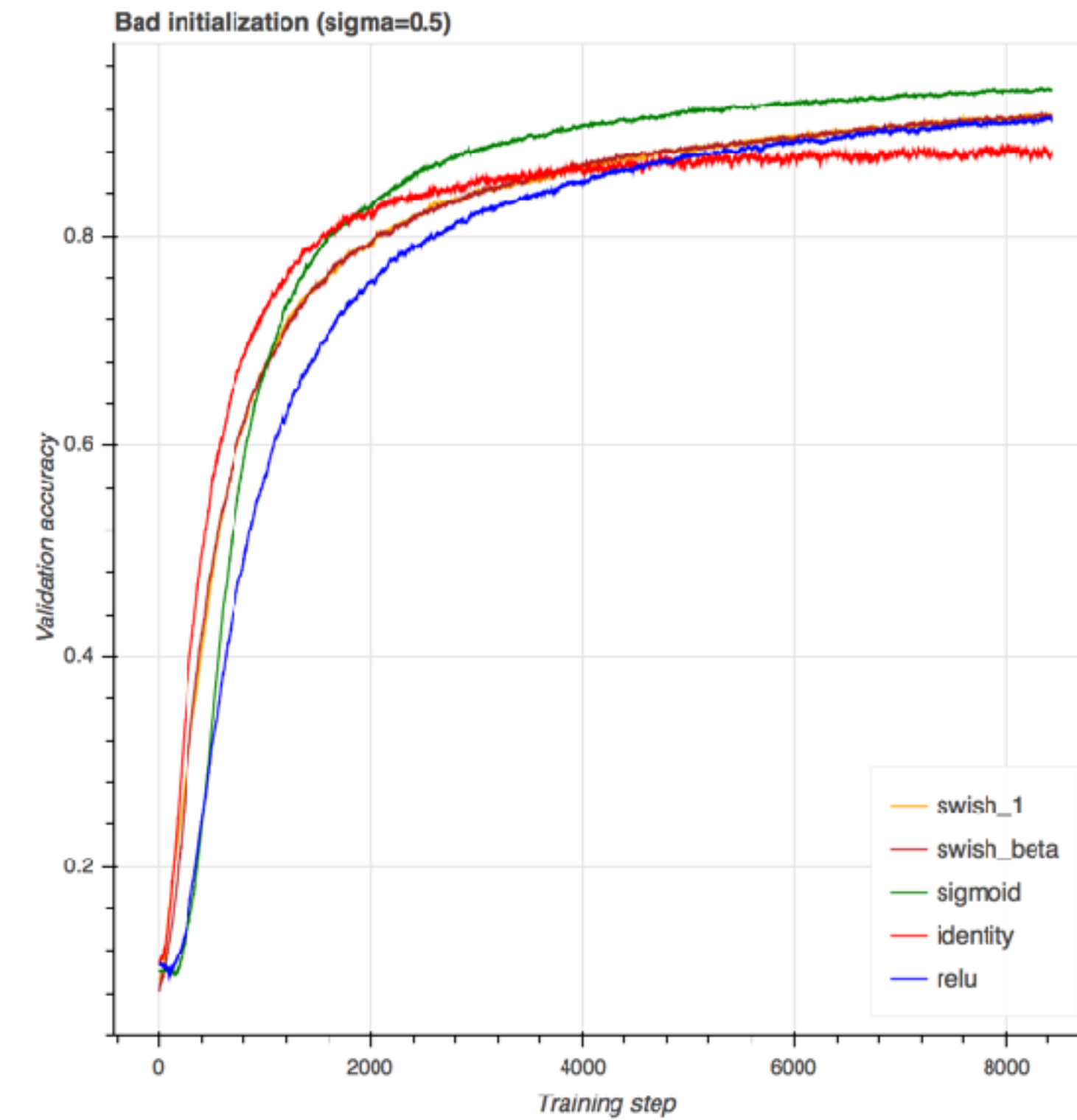
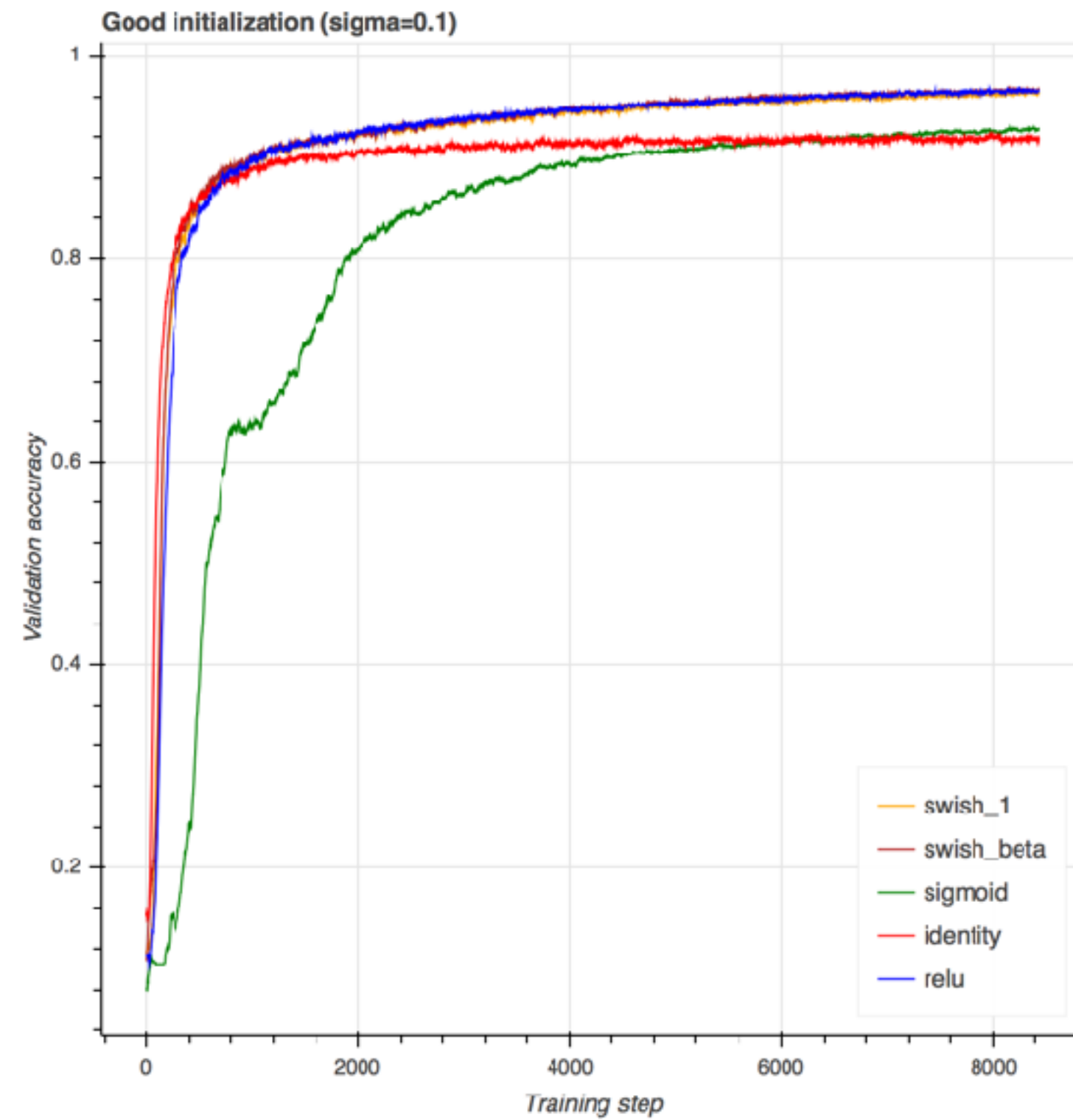
장점

- 두 개의 선형 연산이기 때문에 saturate 하지 않는다.
- 언제나 꽤 좋은 성능을 보장한다. (=경진대회 용으로 좋다)

단점

- weight가 두 배! 연산도 두 배!

다양한 activate function들이 등장하였지만 아직 벤치마킹 결과가 부족하다
그러므로 특정 activate function이 좋다고 주장하더라도 무조건 신뢰하면 안 된다.



- Activate Function은 딥러닝 모델에 non-linearity를 주기 위해 사용된다.
- Hidden Layer에 Activate Function을 넣지 않으면 아무리 레이어를 깊게 쌓아도 사실상 레이어가 하나인 것 처럼 인식한다.
- 가장 대중적으로 쓰이는 건 **ReLU**이다. 일단 언제나 ReLU를 기본으로 사용할 것.
- ReLU로 모델을 완성하였으면, **LReLU**, **PReLU**, **ELU**, **SELU**, 또는 **Swish** 등을 시도해 볼 것. 성능이 오를 가능성이 있다. (100%는 아니다)
- **Maxout**도 사용해볼만한 가치가 있다. 다만 weight가 두 배로 늘어난다는 점은 염두해 두어야 한다.
- tanh를 시도해 볼만 하지만 아마도 성능은 오르지 않을 것이다.
- **sigmoid**는 사용하지 않는다.

Q & A