Activation Functions in Deep Learning

- **Activation Functions** **introduce non-linearity** into the model.
    - **Non-linearity** allows neural networks to learn complex patterns and relationships in data.
- Without activation functions, the neural network would essentially become a linear model, no matter how many layers it had, and wouldn't be able to learn complicated patterns.

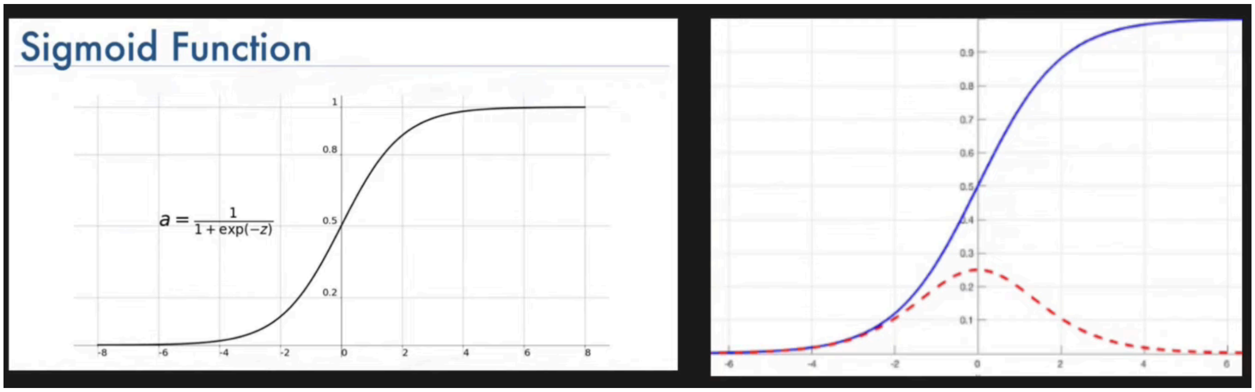
Characteristics of an ideal Activation Function

An Activation Function should be:

- Non-Linear
- Differentiable
- Computationally Inexpensive
- Zero-centered (Normalized)
    - Eg. tanh
- Non-Saturating
    - Saturating functions squiz the output in a range like 0 to 1
    - Saturating functions has vanishing gradient problem

🔥 Popular Activation Functions

Sigmoid (Logistic)



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

-

Range: `(0, 1)`

- **Use Case**:
  - Binary classification (output layer).
  - Historical use in hidden layers (now replaced by ReLU).

- Tends to squash large input values, leading to gradients that can vanish (vanishing gradient problem).
- Not commonly used in deep networks due to vanishing gradients.

> 💡 **We don’t use Sigmoid in hidden layers.**
>
> **It’s only used in the output layer.**

Pros:

✅ Smooth gradient.

✅ Outputs probabilities.

✅ Non-linear

✅ Differentiable

Cons:

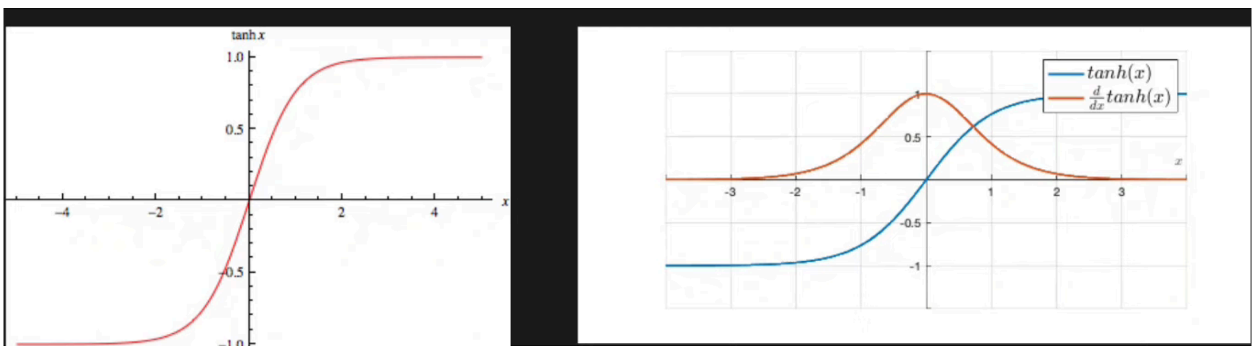
❌ Saturating Function

- ❌ Suffers from **vanishing gradients** (kills gradients for extreme inputs)

❌ Computationally expensive

❌ Non-zero centered

Tanh (Hyperbolic Tangent)



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

Derivative:

$$f'(x) = (1 - \tanh^2(x))$$

- **Range:** `(-1, 1)`

- **Use Case**:
  - Hidden layers (better than sigmoid for gradients).
  - Often used in recurrent neural networks (**RNNs**).

Pros:

✅ Zero-centered (helps optimization).

✅ Non-Linear

✅ Differentiable

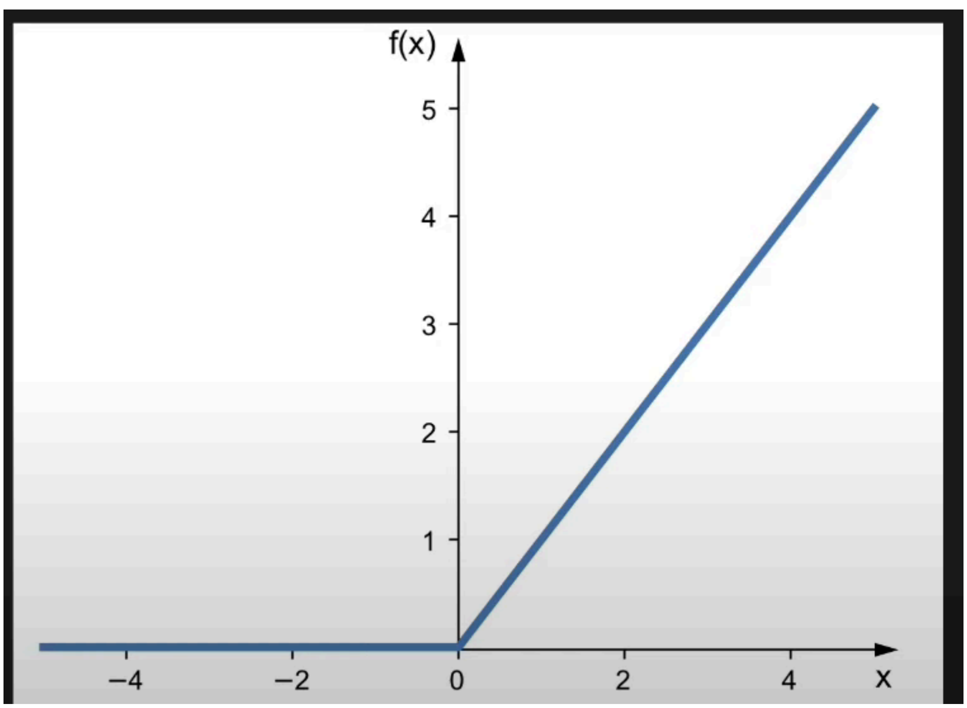
Cons:

❌ Saturating Function

❌ Still suffers from **vanishing gradients** for large inputs.

❌ Computationally expensive

ReLU (Rectified Linear Unit)



$$\mathrm{ReLU}(x) = \max(0, x)$$

💡 **Widely used in Hidden Layers**

- **Range**: `[0, ∞)`
- **Use Case**:
  - Default for **hidden layers** in most networks.
- Simple and efficient, widely used in many deep learning models.
- Helps mitigate the vanishing gradient problem.
- However, can lead to "dead neurons" where certain neurons always output 0 and never contribute to learning.
- Often used in hidden layers of convolutional neural networks (**CNNs**) and fully connected layers.

Pros:

✅ Non-Linear

✅ Non-saturating in positive region

✅ Computationally cheap.

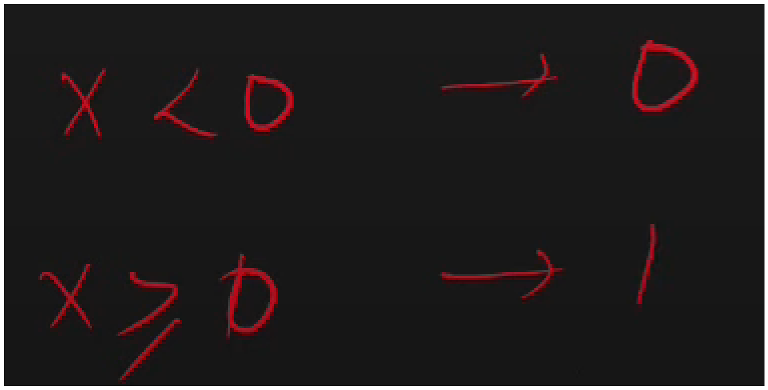
✅ Faster conversion

✅ Avoids vanishing gradients (for positive inputs).

Cons:

❌ **"Dying ReLU" problem** (neurons can get stuck at 0).

❌ Not completely differentiable



❌ Not zero centered

- We use **batch normalization** to solve this problem.
- It normalizes the output of hidden layers

Dying ReLU problem

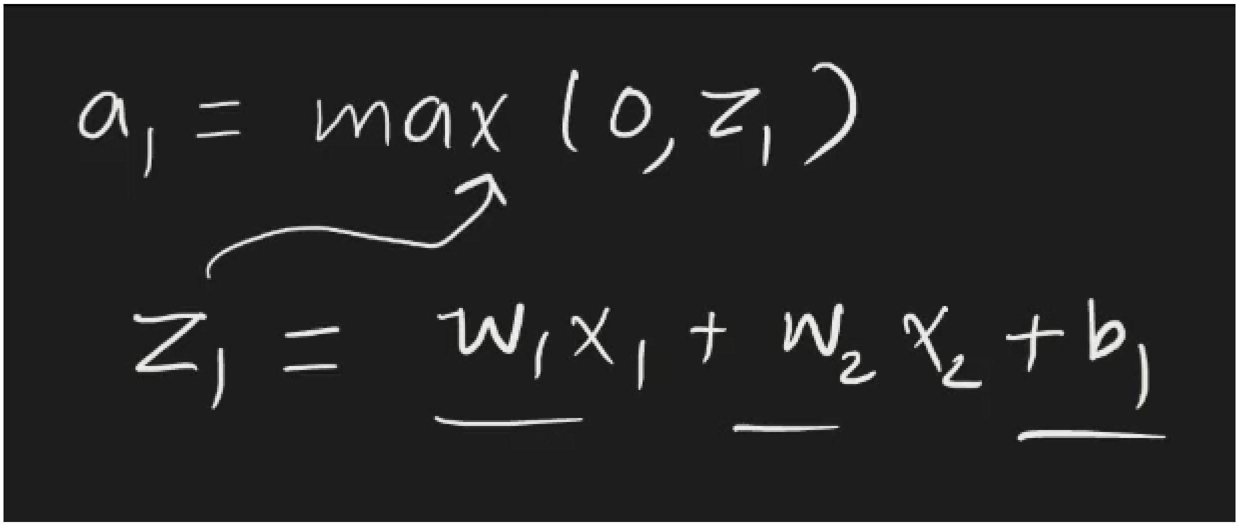
- On applying Relu, output of some neurons become **zero** i.e. there's **no learning**.
    - These neurons are called → **Dead Neurons**
    - They're forever dead.
- If 50% neurons are dead, it won't be able to capture the data representation.
- Worst condition is 100% neurons becoming dead.

Why does the Dying ReLU problem happen?

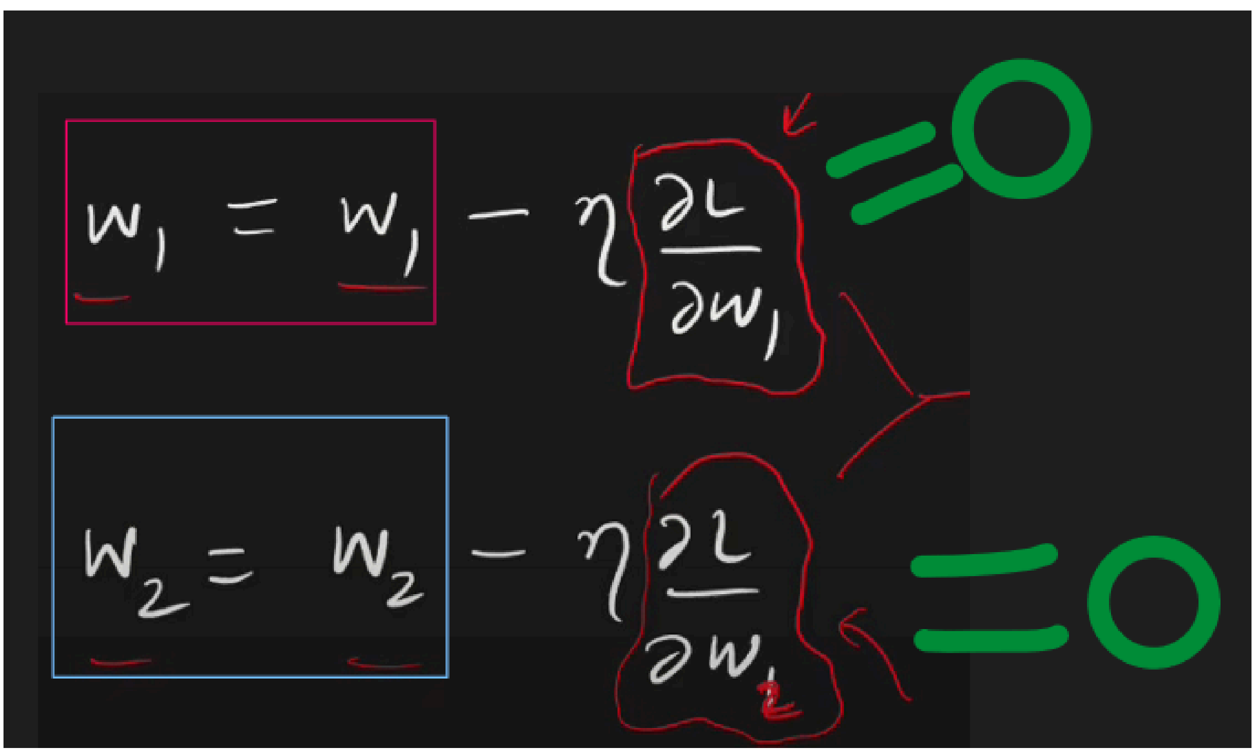
The ReLU activation function is defined as:

$$ReLU(x) = max(0, x)$$

- If the input to ReLU is **positive**, the output is the same value.
- If the input to ReLU is **negative**, the output is zero.



- If Z is negative, a_1 will be 0
- If a_1 becomes 0, its derivative wrt z_1 will also = 0
- And id the derivative= 0, w_1 & w_2 will never be updated.



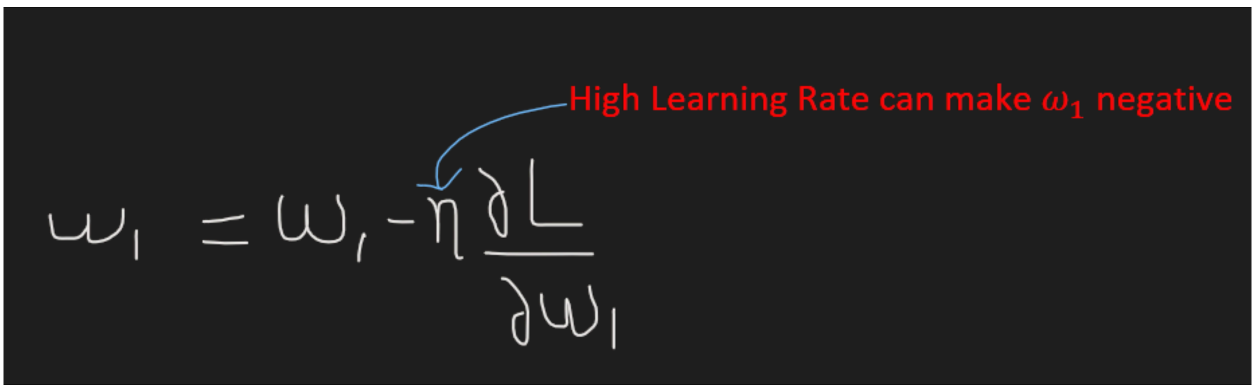
When a large number of neurons in a network output zero (due to negative inputs), their gradients during backpropagation also become zero.

As a result:

1. The weights of these neurons are not updated.
2. This leads to these neurons "dying" permanently, meaning they do not contribute to the learning process anymore.

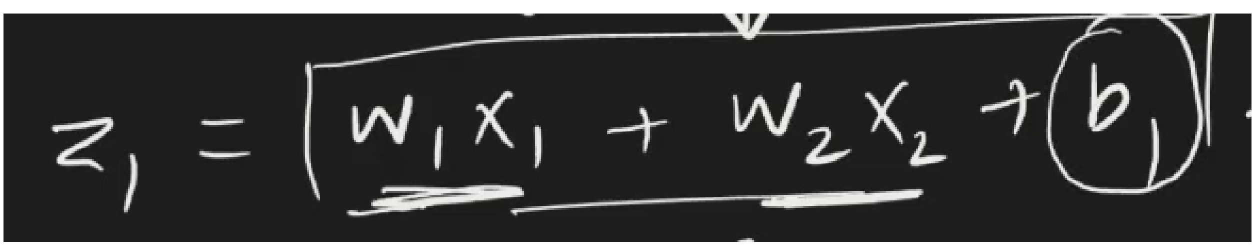
Causes of the Dying ReLU problem:

1. **High learning rates**: If the learning rate is too high, the weight updates during training might overshoot the optimal values, pushing the neuron inputs into regions where the ReLU activation function outputs zero.



- This can make z_1 negative in next cycle.

2. **High negative bias**



- If b becomes negative, z_1 will could become negative.

3. **Data distribution**: If the input data is not properly normalized or centered, it could lead to negative inputs for a large number of neurons, resulting in dead neurons.

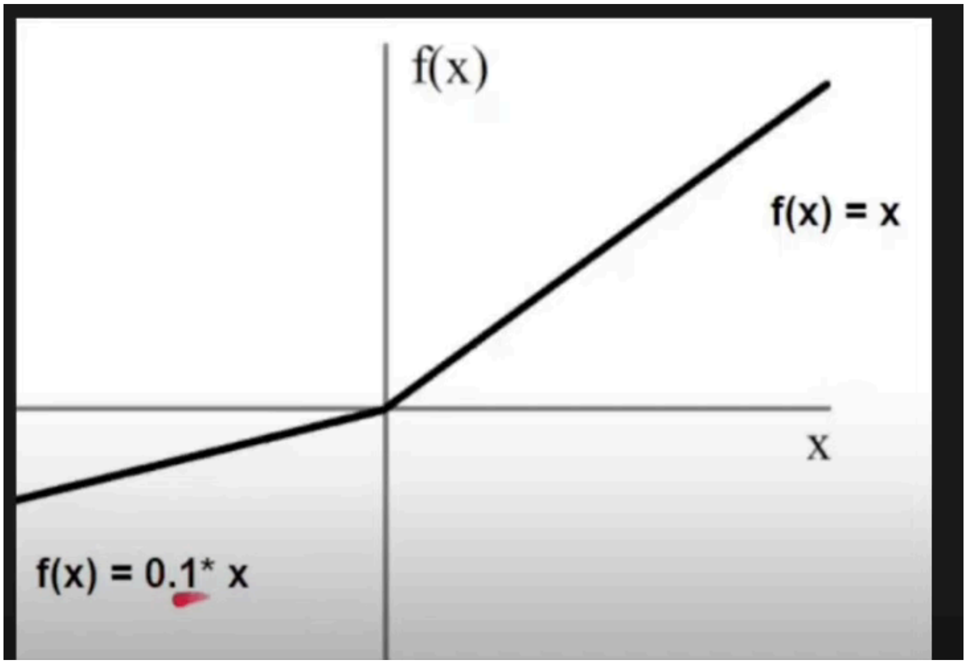
Solutions to the Dying ReLU problem

- Set a low learning rate
- Set bias with a positive value like ***0.01***
- Don’t use Relu

Relu Variants

1. Linear variants → Applying linear transformation on Relu
    a. **Leaky ReLU**
    b. **Parametric ReLU (PReLU)**
2. Non-linear variants
    a. ELU (Exponential Linear Unit)
    b. SELU (Scaled Exponential Linear Unit)

Leaky ReLU



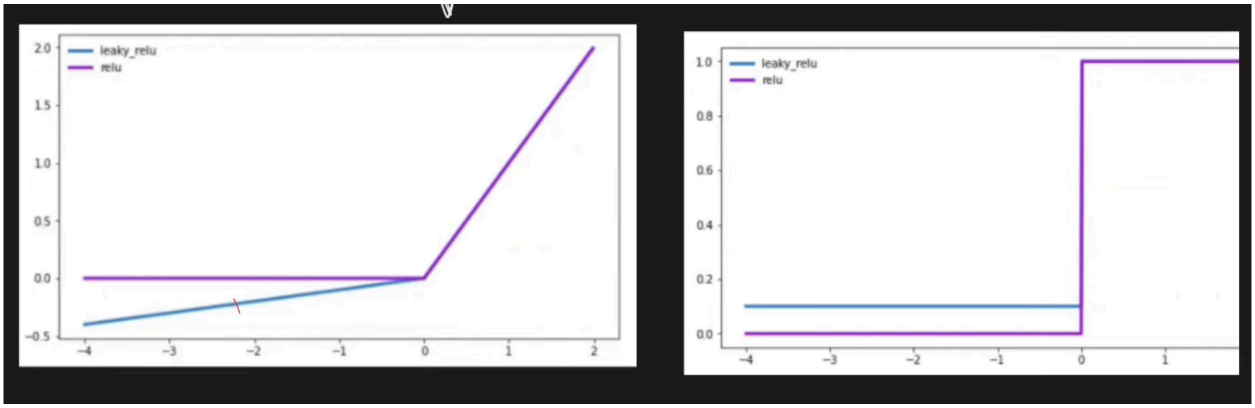
$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$$

$$\text{Leaky ReLU}(x) = \max(\alpha x, x) \quad \text{(where } \alpha \text{ is a small constant, usually 0.01)}$$

(where `α` is a small slope, e.g., `0.01`)

$z \geq 0 \rightarrow z$

$z < 0 \rightarrow \frac{1}{100} z$ (fraction of z)



Characteristics:

- A variant of ReLU that allows small negative values when the input is less than zero, thus preventing the "dying ReLU" problem.
- Still computationally efficient, but introduces a small negative slope when the input is negative.

- **Range**: `(-∞, ∞)`
- **Use Case**:
  - Fixes "dying ReLU" problem.

Pros:

✅ No dead neurons.

✅ Non-saturated

✅ easily computed

✅ close to zero centered

Cons:

❌ Results may be less consistent than ReLU.

Parametric ReLU (PReLU)

$$\mathrm{PReLU}(x) = \max(\alpha x, x)$$

- Like LeakyReLU, but **α is learned** during training.
- **Use Case**:
  - When you want the model to learn the slope.

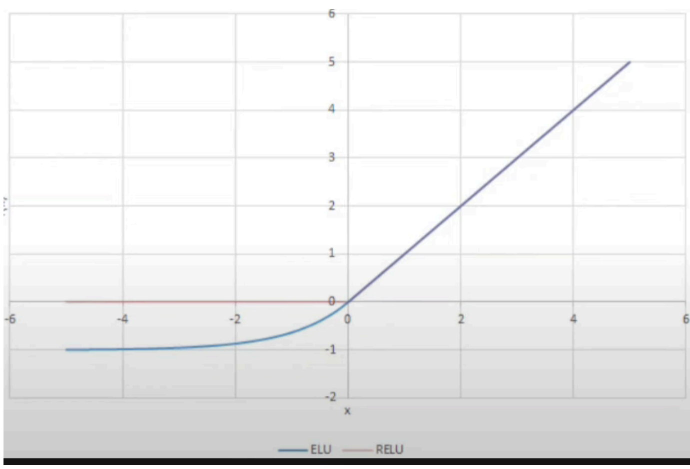
Pros:

✅ More flexible than Leaky ReLU because α is learned.

Cons:

❌ Adds more parameters to the model, which could lead to overfitting if not carefully regularized.

ELU (Exponential Linear Unit)



$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases}$$

$$\mathrm{ELU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \mathrm{ELU}(x) + \alpha & \text{if } x \leq 0 \end{cases}$$

$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases}$$

Range: (-α, ∞)

Characteristics:

- Smooth and differentiable.
- The negative part is an exponential function, which helps with the vanishing gradient problem.
- Often used when trying to **achieve better training performance than ReLU or Leaky ReLU.**

Pros:

✅ Close to zero-centered

✅ Better generalized

✅ No dying Relu problem

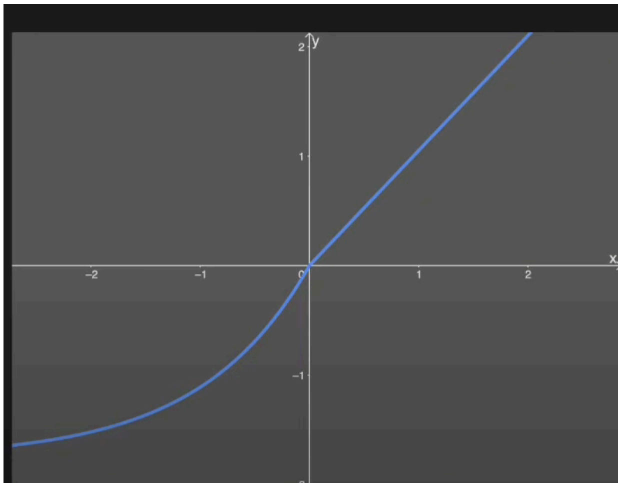
✅ Always continuous & differentiable

Cons:

❌ Computationally expensive

SELU (Scaled Exponential Linear Unit)

- **Not widely used.**



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

$$a \approx 1.6732632423543772848170429916717$$

$$\lambda \approx 1.0507009873554804934193349852946$$

$$\mathrm{SELU}'(x) = \lambda \begin{cases} 1 & \text{if } x > 0 \\ \alpha e^{x} & \text{if } x \leq 0 \end{cases}$$

Formula: Similar to ELU, but with a scaling factor:

$$\mathrm{SELU}(x) = \lambda \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases}$$

💡 α & λ are fixed parameters.

- **Range**: (-∞, ∞)
- **Characteristics**:
    - Introduces **self-normalizing properties**, helping the network **converge faster**.
    - Often used in networks where the goal is to **minimize the vanishing and exploding gradient problems.**

Pros:

✅ Self-normalizing

Cons:

❌ Not enough research