Training Deep Neural Networks

Possible challenges:

- First, you would be faced with the tricky *vanishing gradients* problem (or the related *exploding gradients* problem) that affects deep neural networks and makes lower layers very hard to train.
- Second, you might not have enough training data for such a large network, or it might be too costly to label.
- Third, training may be extremely slow.
- Fourth, a model with millions of parameters would severely risk overfitting the training set, especially if there are not enough training instances, or they are too noisy.

Vanishing/Exploding Gradients Problems

- The backpropagation algorithm works by going from the output layer to the input layer, propagating the error gradient on the way. Once the algorithm has computed the gradient of the cost function with regard to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent step.
- Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the Gradient Descent update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is called the vanishing gradients problem.
- In some cases, the opposite can happen: the gradients can grow bigger and bigger, so many layers get insanely large weight updates and the algorithm diverges. This is the exploding gradients problem, which is mostly encountered in recurrent neural networks. More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

Weight Initialization for Deep Networks

- A partial solution to the Vanishing / Exploding gradients in NN is a better or more careful choice of the random initialization of weights.

Initialization	Activation functions
Glorot	None, Tanh, Logistic, Softmax
He	ReLU & variants
LeCun	SELU

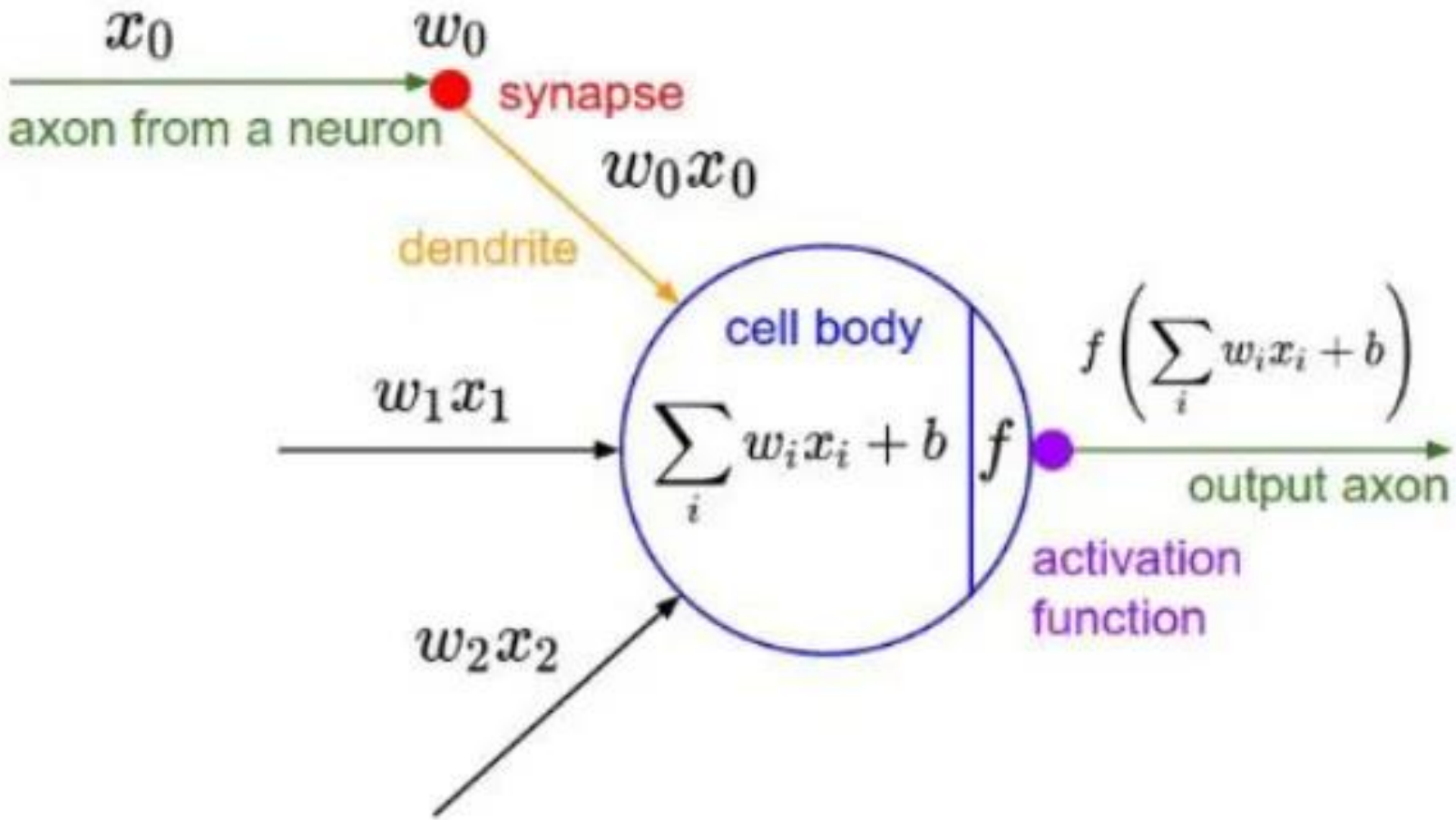
Initialization summary

- The weights $W^{[l]}$ should be initialized randomly to break symmetry
- It is however okay to initialize the biases $b^{[l]}$ to zeros. Symmetry is still broken so long as $W^{[l]}$ is initialized randomly
- Different initializations lead to different results
- Random initialization is used to break symmetry and make sure different hidden units can learn different things
- Don't intialize to values that are too large
- He initialization works well for networks with ReLU activations.

Nonsaturating Activation Functions

The vanishing/ exploding gradients problems are in part due to a poor choice of activation function. It turns out that the activation functions other than sigmoid behave much better in deep neural networks, in particular the ReLU activation function, mostly because it does not saturate for positive values (and also because it is quite fast to compute).

Activation fuction


x_0
w_0
synapse
axon from a neuron
$w_0 x_0$
dendrite
cell body
$w_1 x_1$
$\sum_i w_i x_i + b$
f
$f\left(\sum_i w_i x_i + b\right)$
output axon
activation function
$w_2 x_2$

What is an activation function?

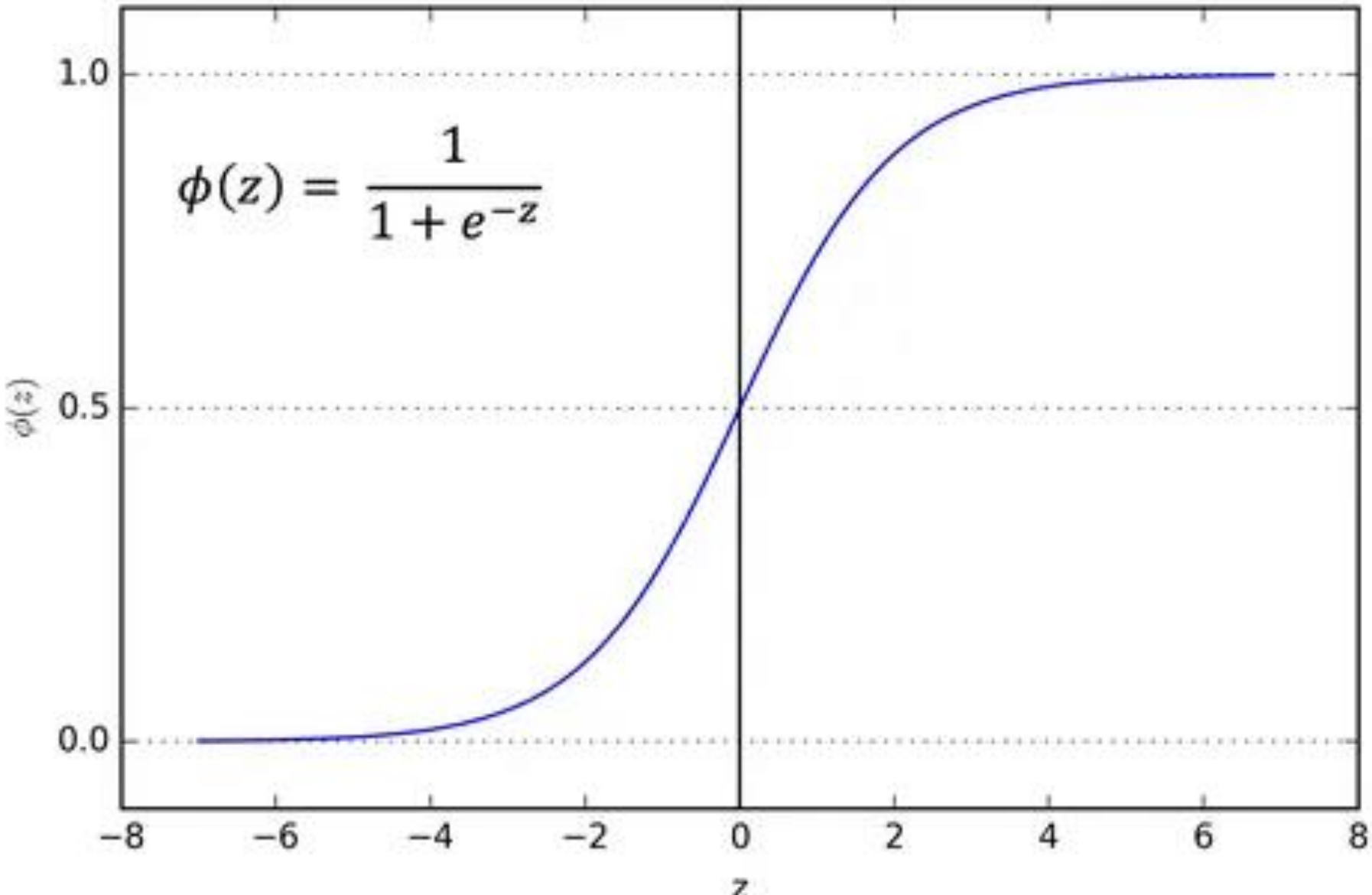
" An activation function is a non-linear function applied by the neuron to introduce non-linear properties in the network."

Why we use activation functions ?

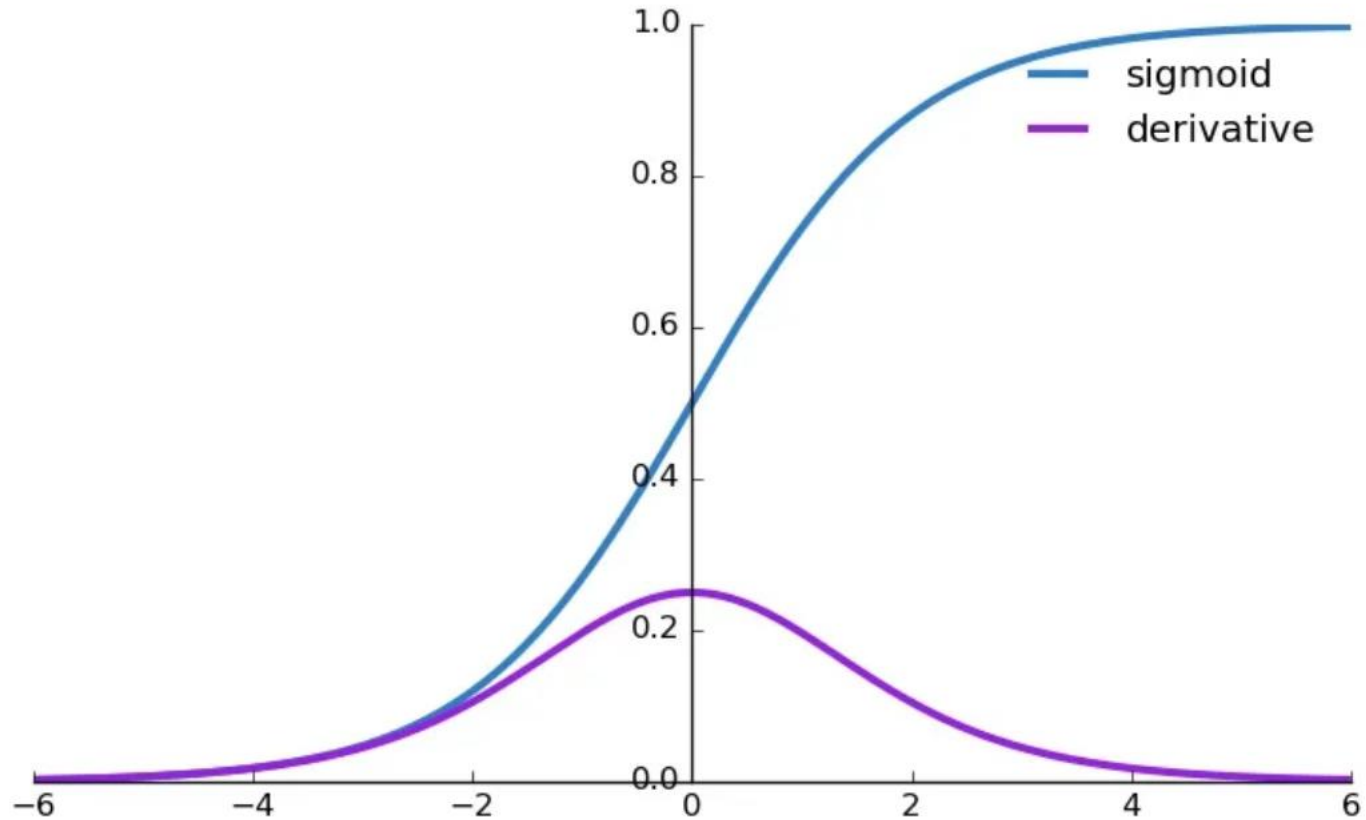
Why do you need non-linear activation functions?

- If we removed the activation function from our algorithm that can be called linear activation function.
- Linear activation function will output linear activations
    - Whatever hidden layers you add, the activation will be always linear like logistic regression (So its useless in a lot of complex problems)
- You might use linear activation function in one place - in the output layer if the output is real numbers (regression problem). But even in this case if the output value is non-negative you could use RELU instead.

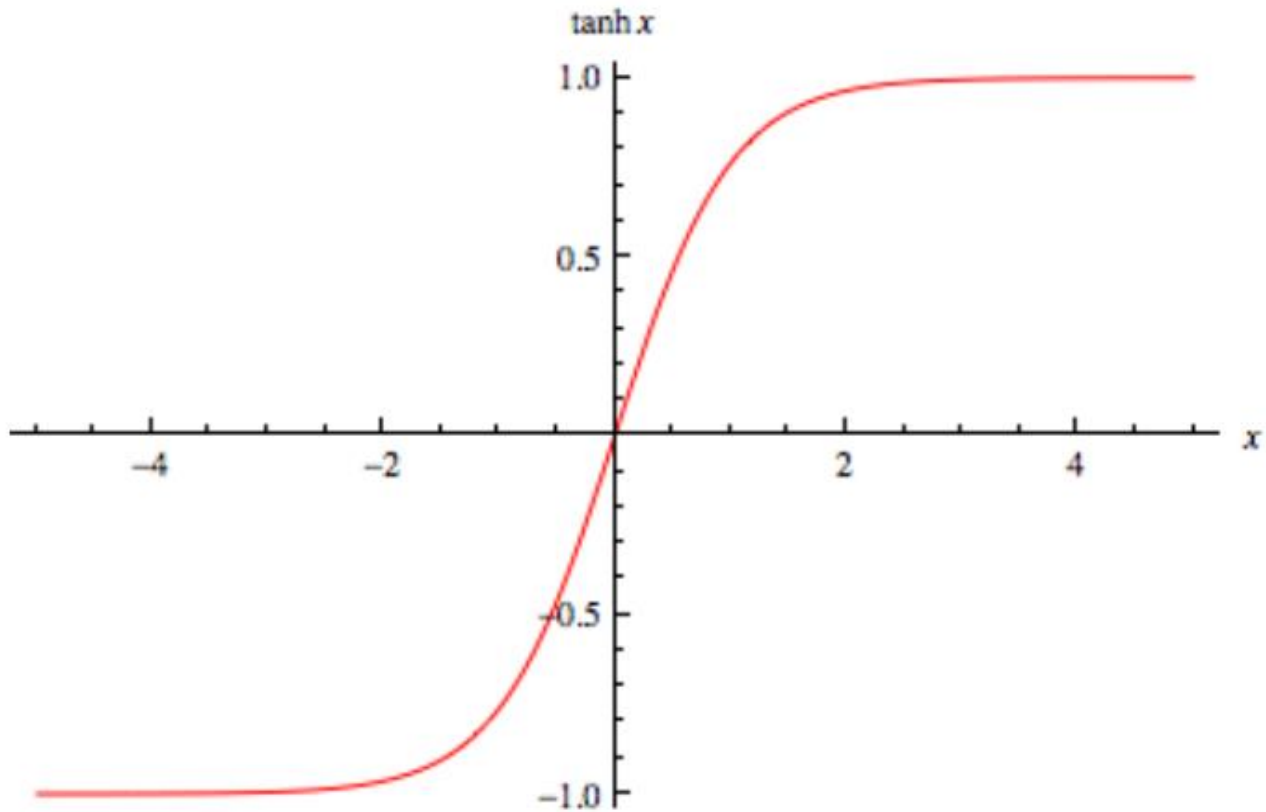
Sigmoid



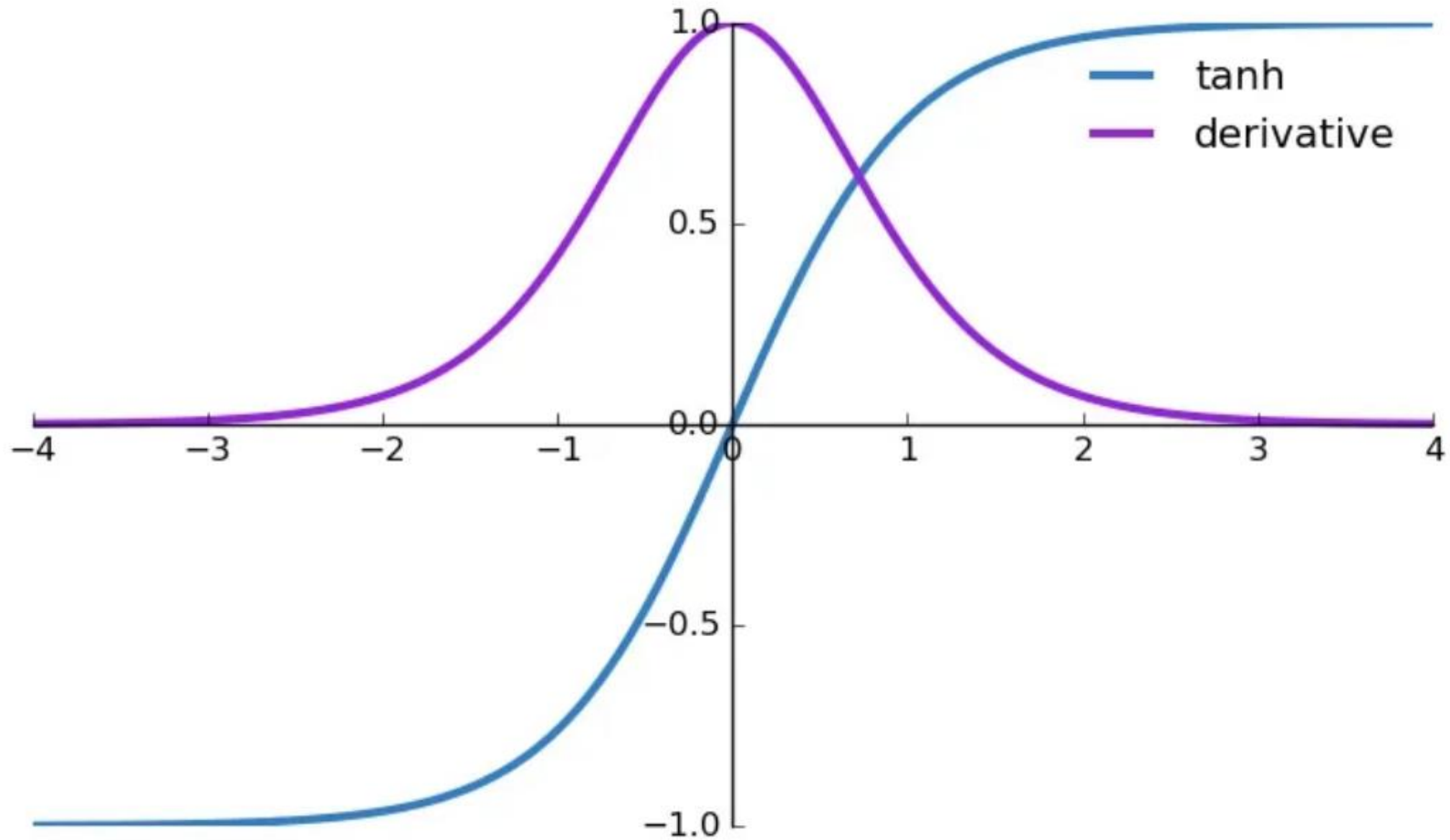
Graph of sigmoid derivative



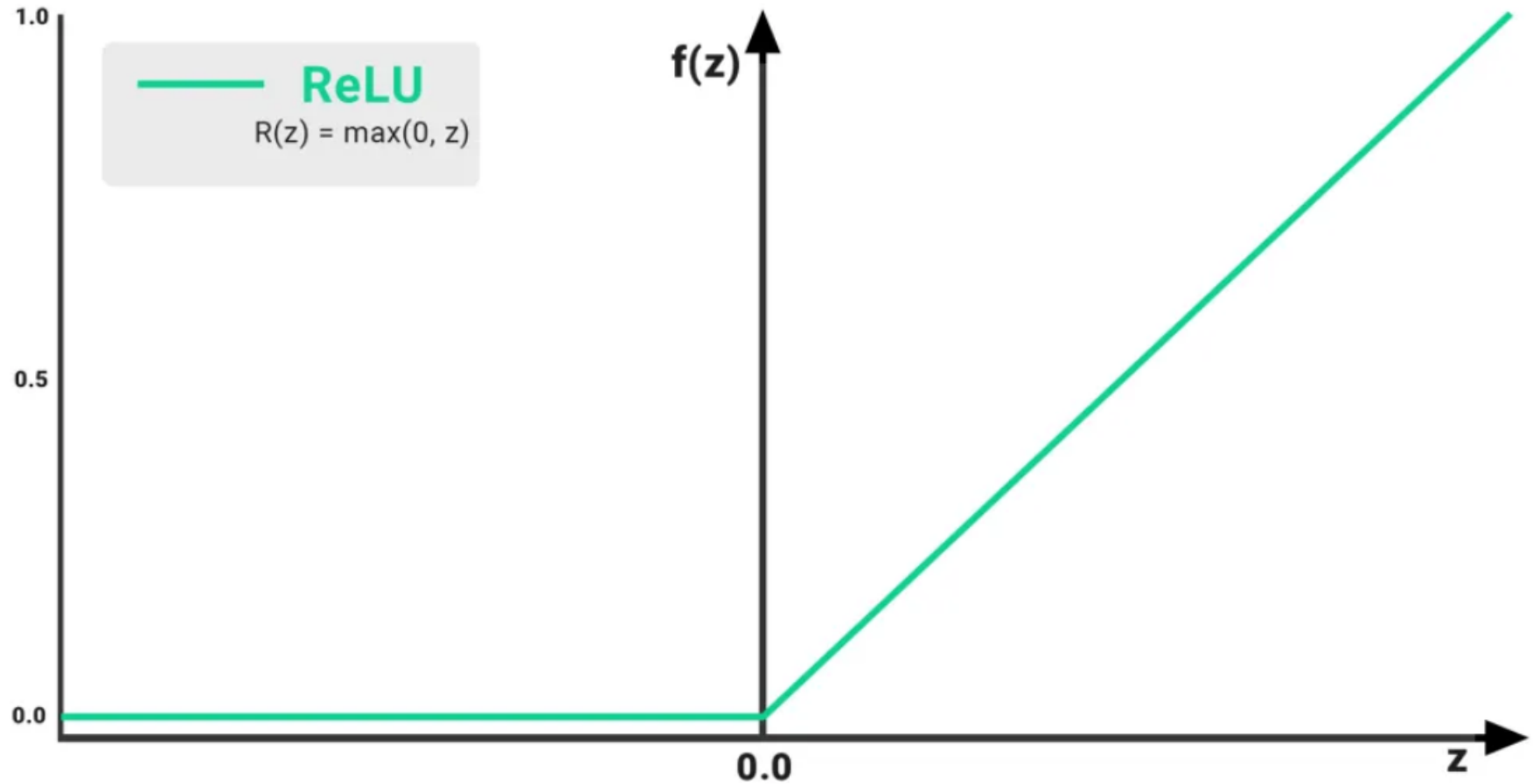
Graph of Tanh function



Graph of derivative of tanh function



Graph for ReLU function



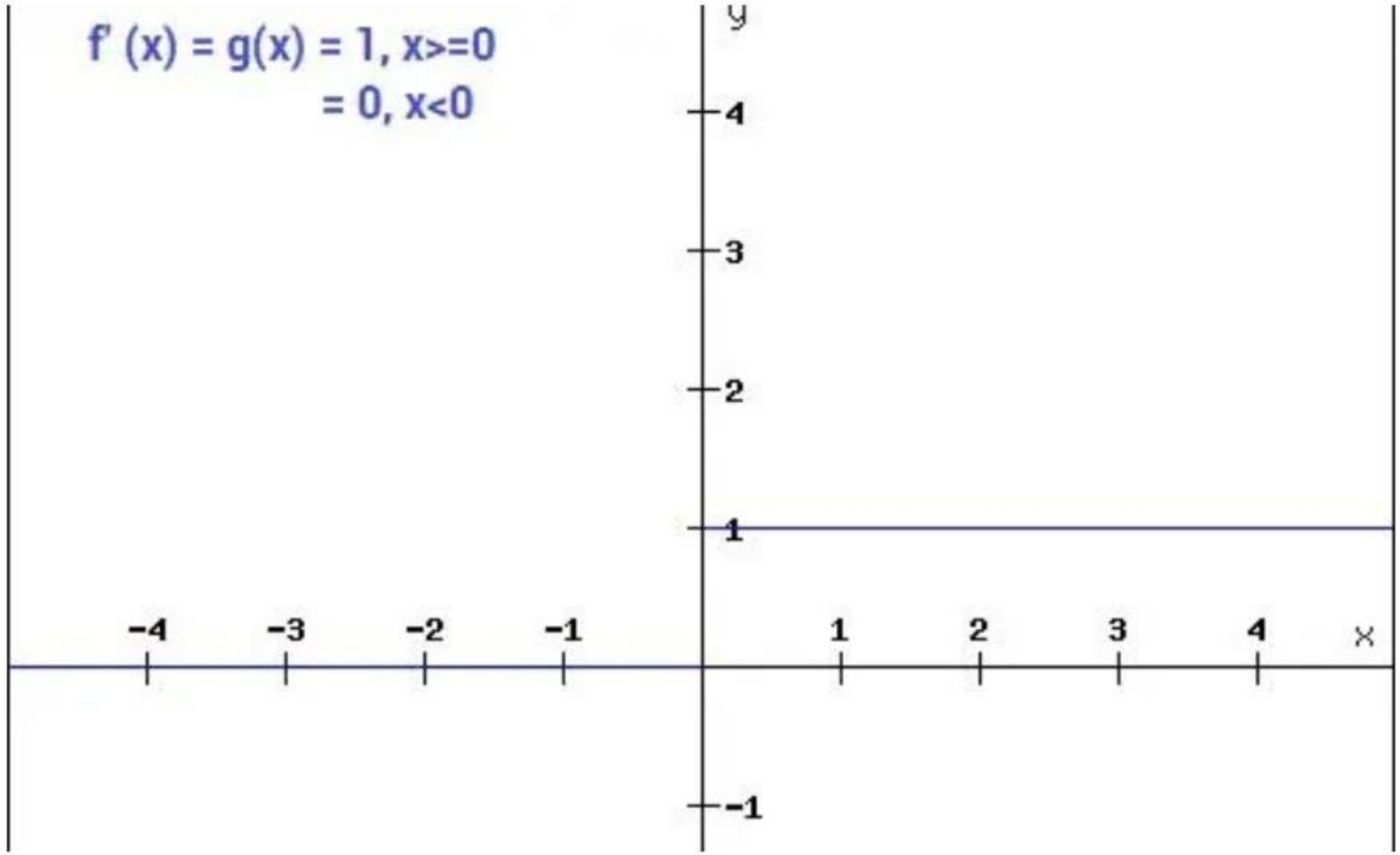
Equation of ReLU function

$$RELU(x) = \begin{cases} 0 \; if \; x < 0 \\ x \; if \; x >= 0 \end{cases}$$

Equation of derivative of ReLU function

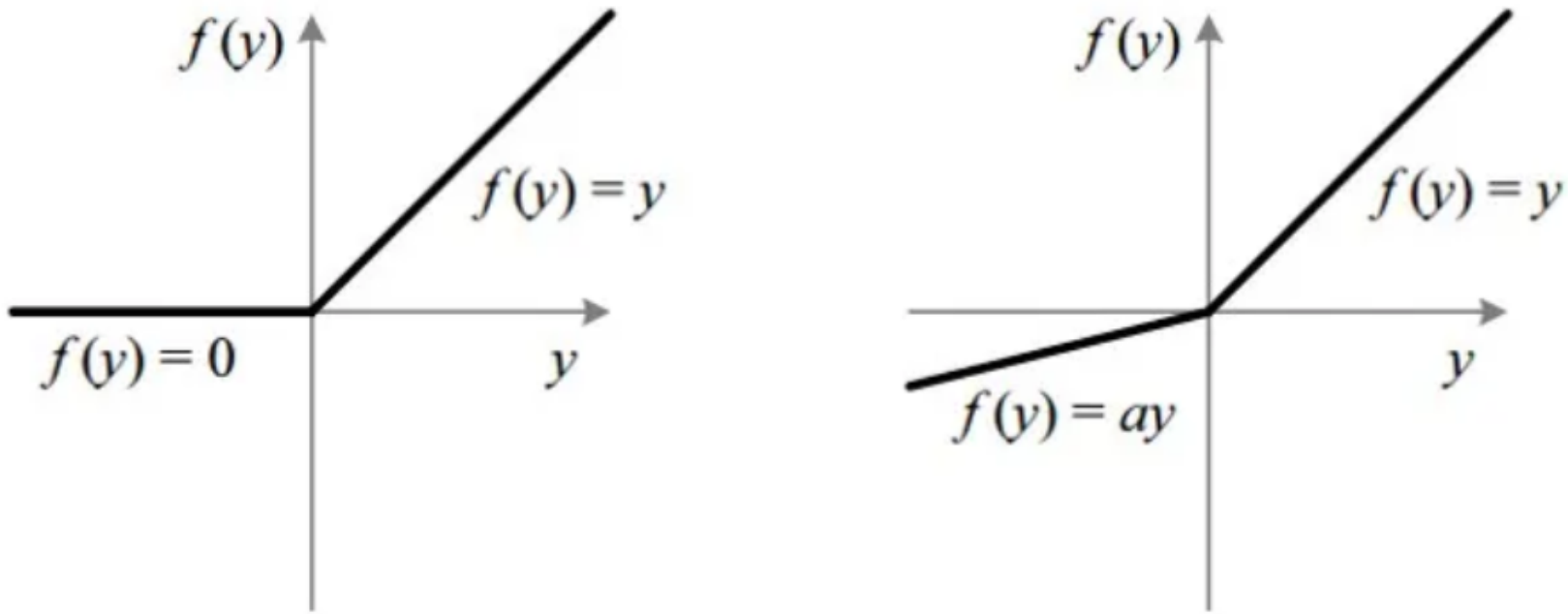
$$R'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$$

Graph for derivative of ReLU function



4. Leaky ReLU Activation Function-

An activation function **specifically designed to compensate** for the dying ReLU problem.



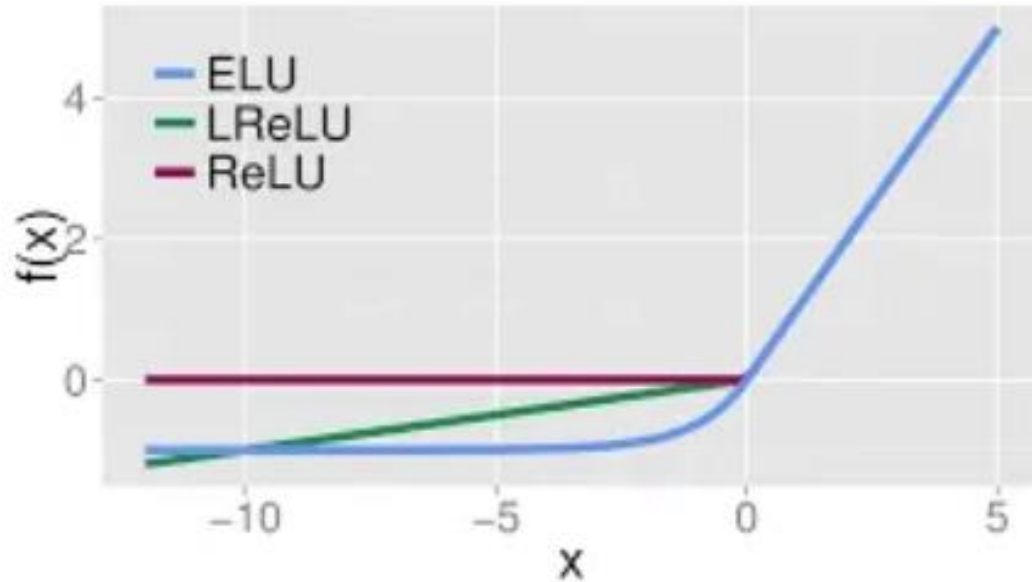
ReLU vs Leaky ReLU

Why Leaky ReLU is **better** than ReLU?

$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ a_i y_i, & \text{if } y_i \leq 0 \end{cases}.$$

1. The leaky ReLU **adjusts the problem of zero gradients** for negative value, by giving a very **small linear component of x** to negative inputs**(0.01x).**
2. The leak helps to increase the range of the ReLU function. Usually, the value of **a** is **0.01** or so.
3. Range of the Leaky ReLU is **(-infinity to infinity).**

ELU (Exponential Linear Units) function-

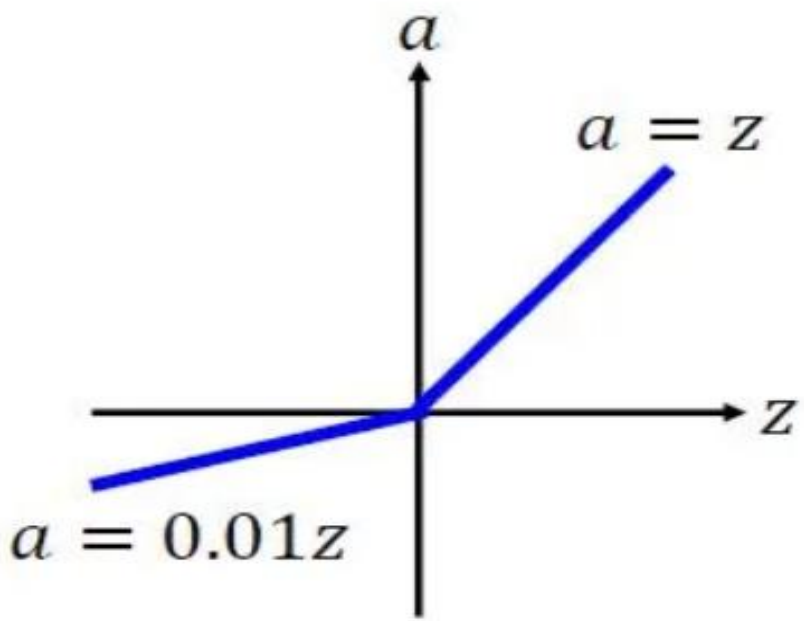


ELU vs Leaky ReLU vs ReLU

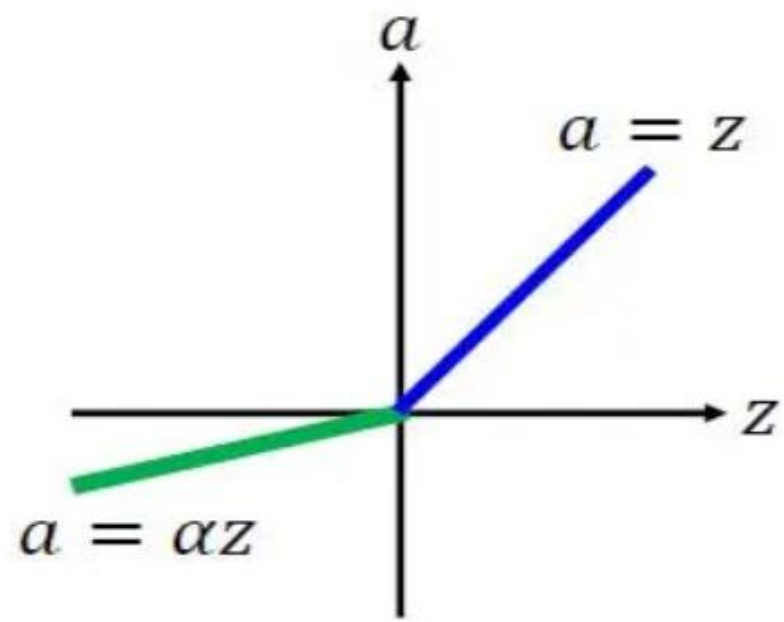
$$g(x) = \mathrm{ELU}(x) = \begin{cases} x, & x > 0 \\ \alpha(e^x - 1), & x \leqslant 0 \end{cases}$$

ReLU - variant

Leaky ReLU



Parametric ReLU



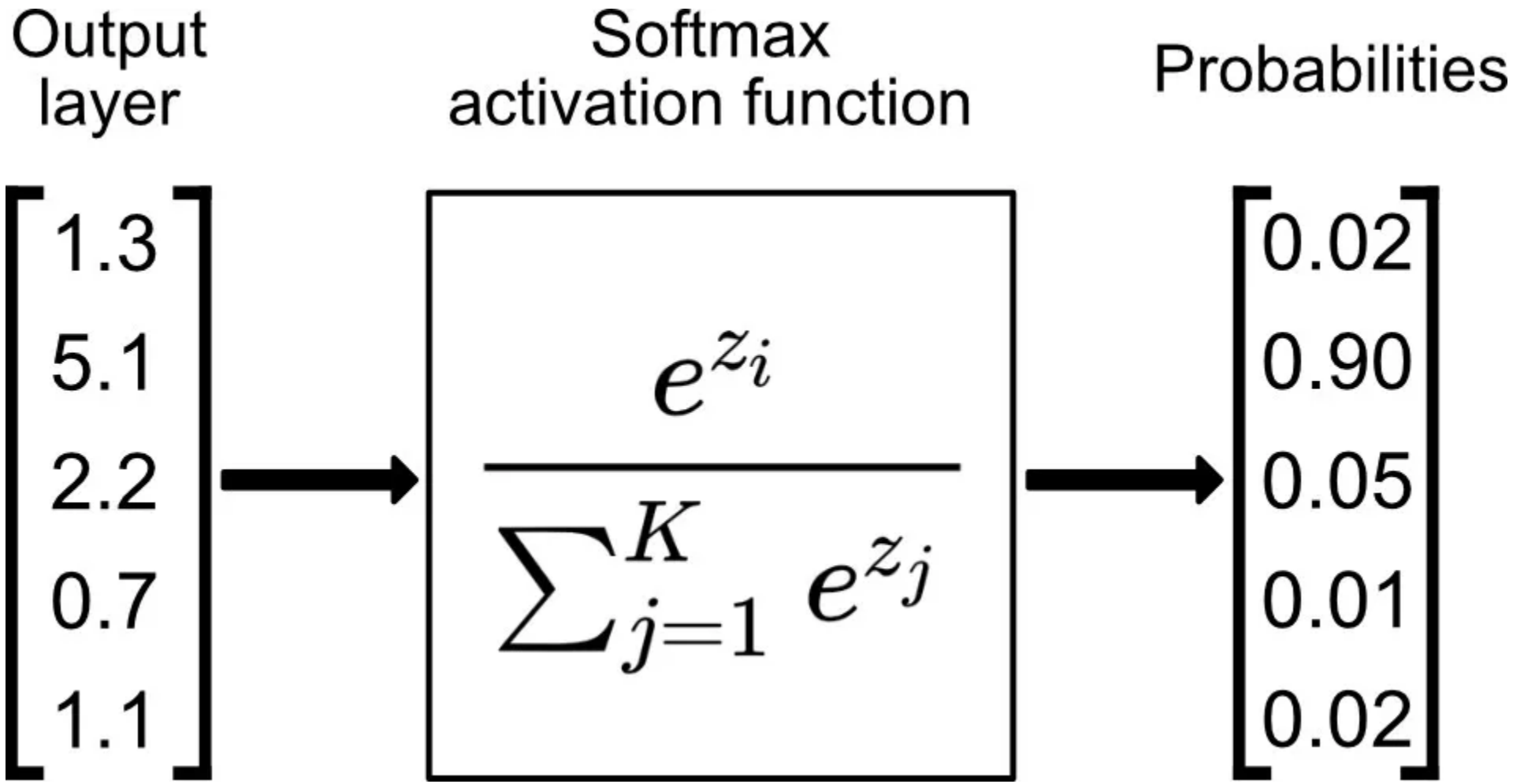
α also learned by gradient descent

$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ a_i y_i, & \text{if } y_i \leq 0 \end{cases}$$

We look at the formula of PReLU. The parameter **α** is generally a number between 0 and 1, and it is generally relatively small, such as a few zeros.

- if $\mathbf{a_i=0}$, f becomes ReLU
- if $\mathbf{a_i>0}$, f becomes leaky ReLU
- if $\mathbf{a_i}$ **is a learnable parameter**, f becomes PReLU

Softmax

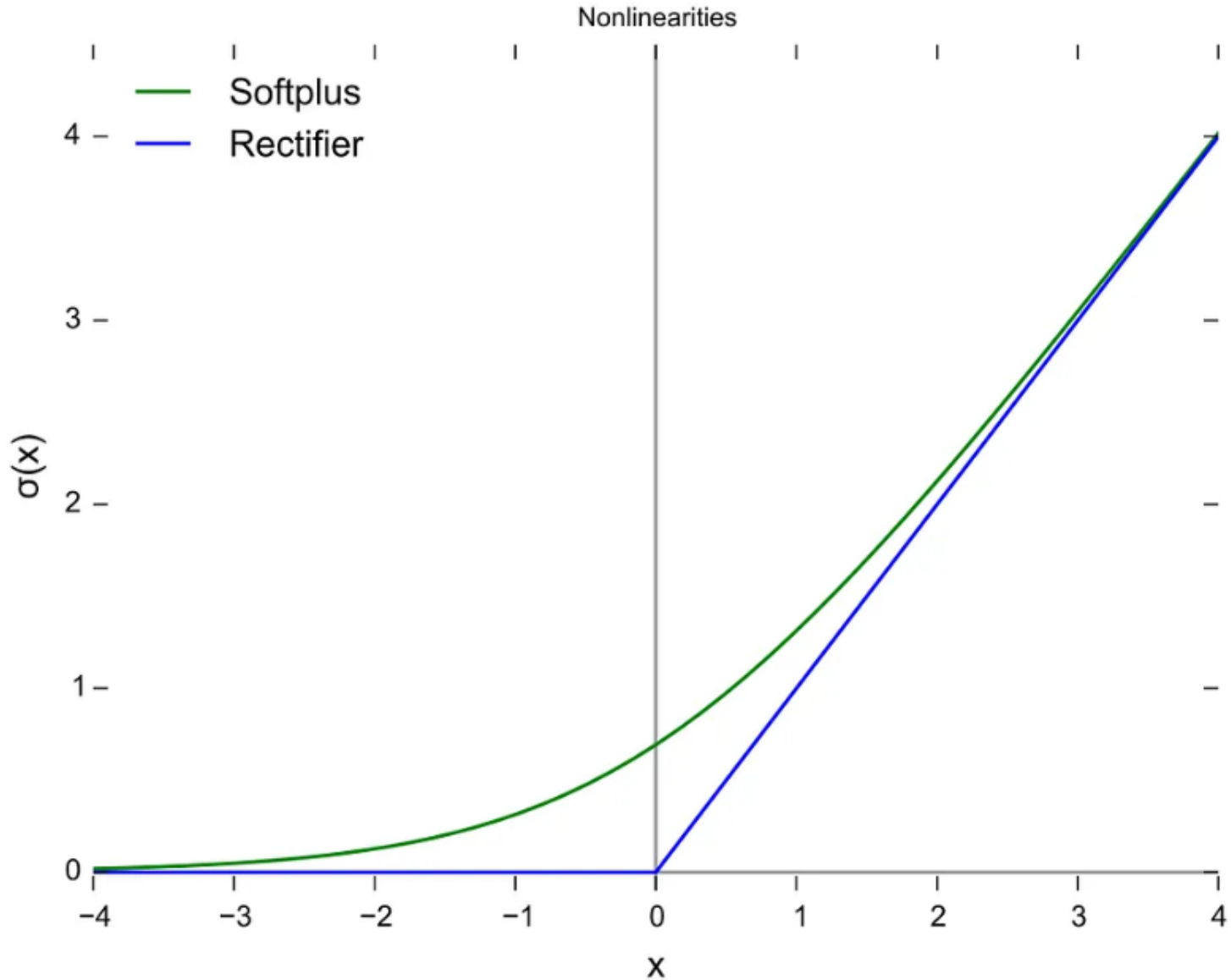

Output layer
Softmax activation function
Probabilities
1.3
5.1
2.2
0.7
1.1
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
0.02
0.90
0.05
0.01
0.02

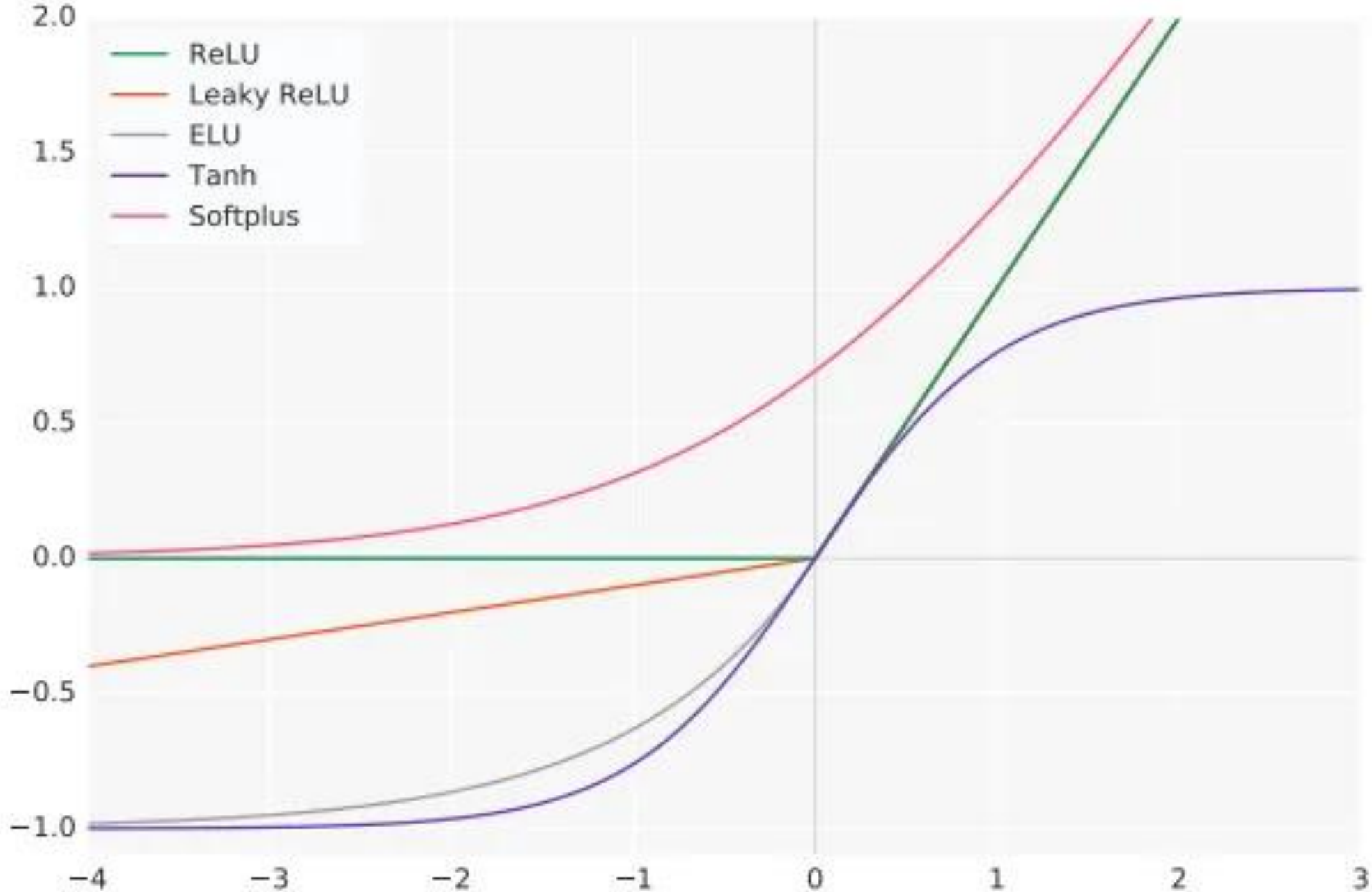
Softplus function: $f(x) = \ln(1+\exp x)$

The **derivative** of softplus is -

$f'(x)=\exp(x) / (1+\exp x)$

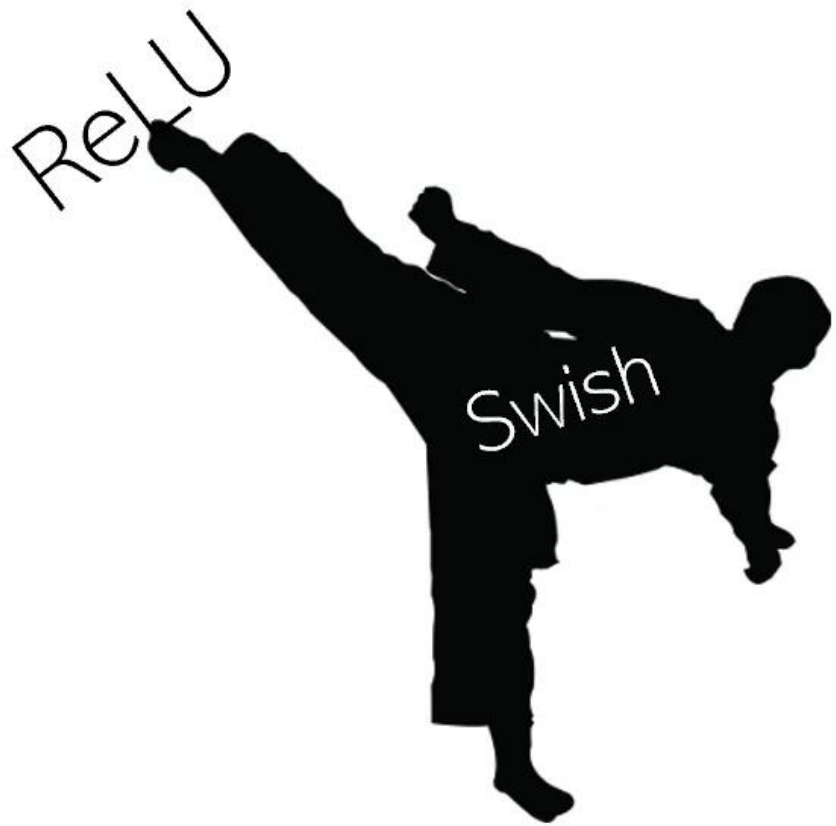
$= 1/ (1 +\exp(-x))$

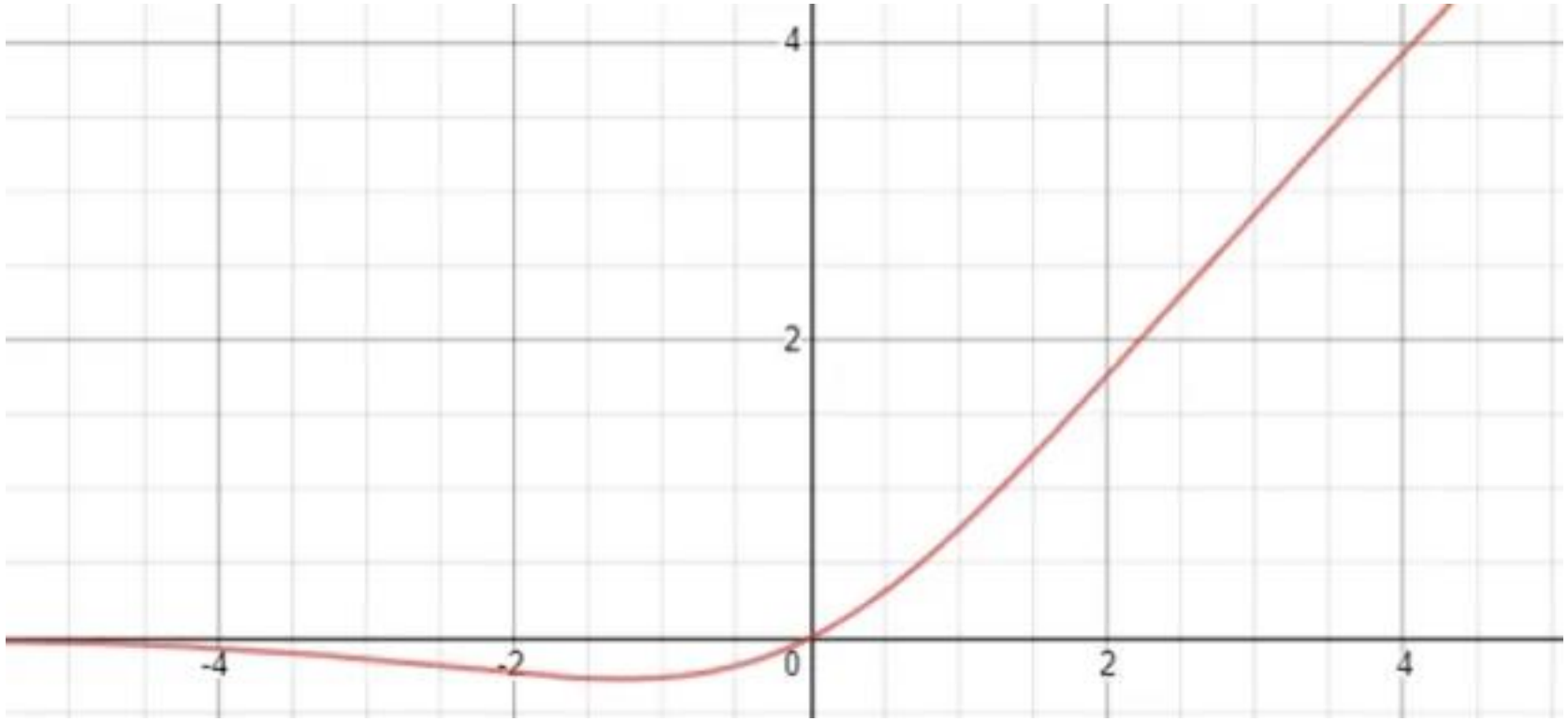



ReLU
Leaky ReLU
ELU
Tanh
Softplus
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−4
−3
−2
−1
0
1
2
3

Recommendations

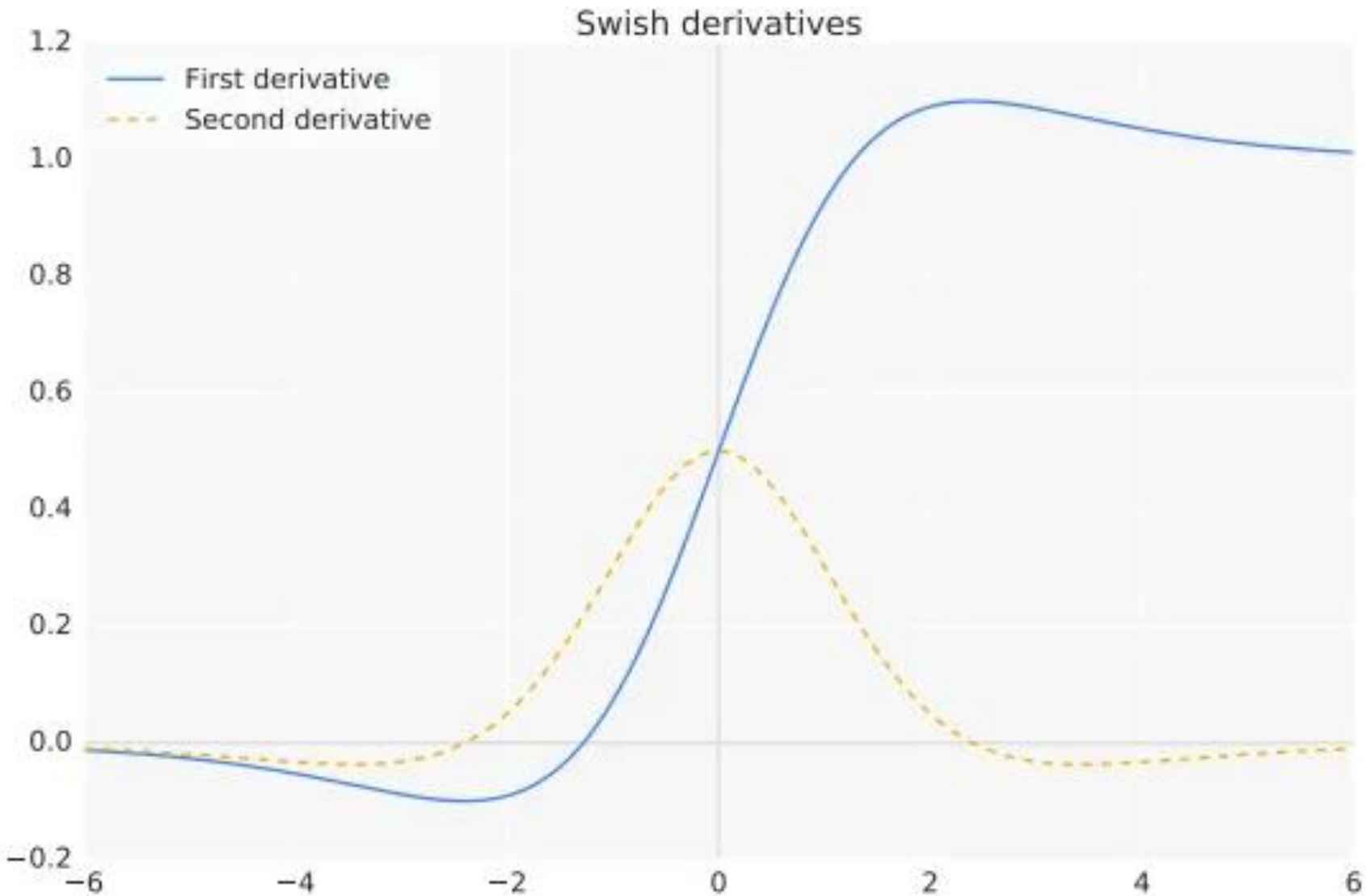
So which activation function should you use for the hidden layers of your deep neural networks? Although your mileage will vary, in general SELU > ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic. If the network's architecture prevents it from self-normalizing, then ELU may perform better than SELU (since SELU is not smooth at $z = 0$). If you care a lot about runtime latency, then you may prefer leaky ReLU. If you don't want to tweak yet another hyperparameter, you may just use the default α values used by Keras (e.g., 0.3 for the leaky ReLU). If you have spare time and computing power, you can use cross-validation to evaluate other activation functions, in particular RReLU if your network is over-fitting, or PReLU if you have a huge training set.


ReLU
Swish



the Swish activation function is...

$$f(x) = x * (1 + \exp(-x))^{-1}$$


Swish derivatives
First derivative
Second derivative
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−6
−4
−2
0
2
4
6

Testing on Various Datasets

The authors tested Swish against the following baseline activation functions with the following results:

- CIFAR-10 and CIFAR-100 datasets — Swish consistently matches or outperforms ReLU on every model for both CIFAR-10 and CIFAR-100.
- ImageNet — Swish outperforms ReLU by 0.6%, with a 0.9% boosts on Mobile NASNet-A and a 2.2% boost on MobileNet over ReLU.
- WMT 2014 English to German — Swish outperforms ReLU on all of four test datasets.

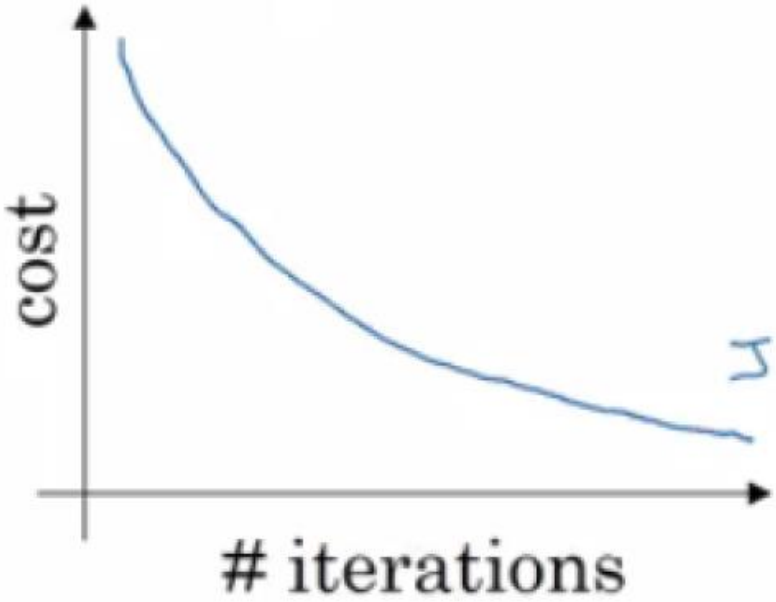
Dropout

What you should remember about dropout:

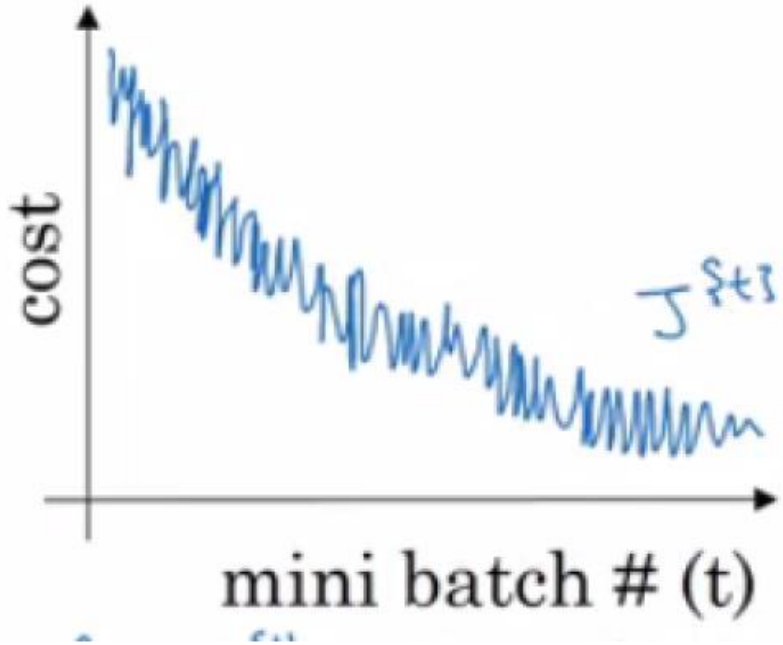
- Dropout is a regularization technique.
- You only use dropout during training. Don't use dropout (randomly eliminate nodes) during test time.
- Apply dropout both during forward and backward propagation.
- During training time, divide each dropout layer by keep_prob to keep the same expected value for the activations. For example, if `keep_prob` is 0.5, then we will on average shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half are contributing to the solution. Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value. You can check that this works even when keep_prob is other values than 0.5.

Training with mini batch gradient descent

Batch gradient descent



Mini-batch gradient descent



Faster Optimizers

- Training a very large deep neural network can be painfully slow. So far we have seen a few ways to speed up training (and reach a better solution): applying a good initialization strategy for the connection weights, and using a good activation function. Another huge speed boost comes from using a faster optimizer than the regular Gradient Descent optimizer.
- The most popular ones: Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and finally Adam and Nadam optimization.