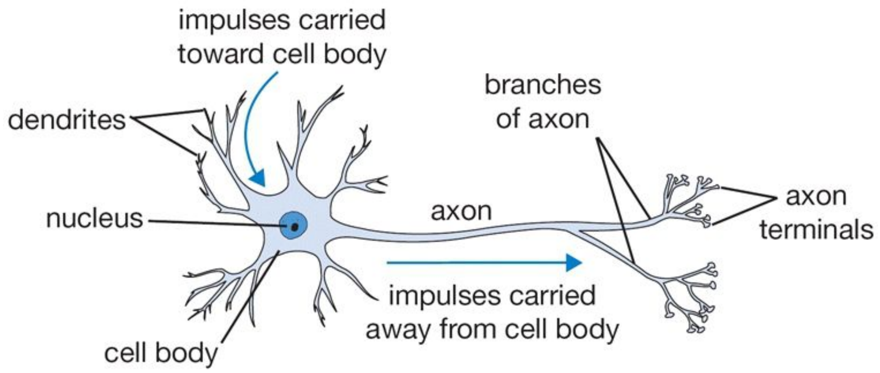
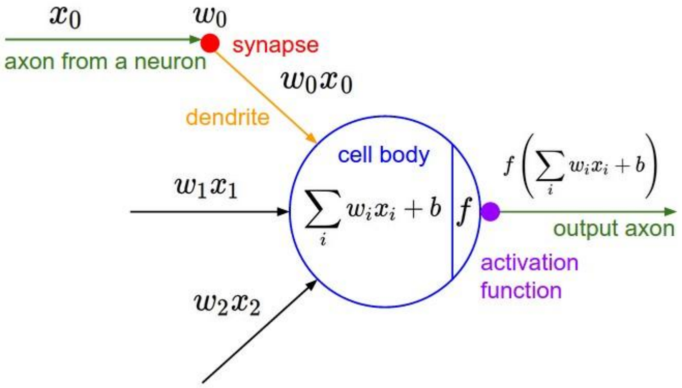


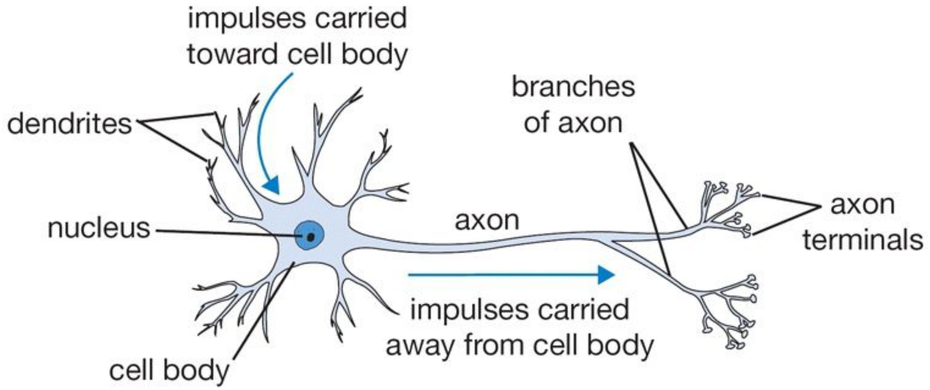


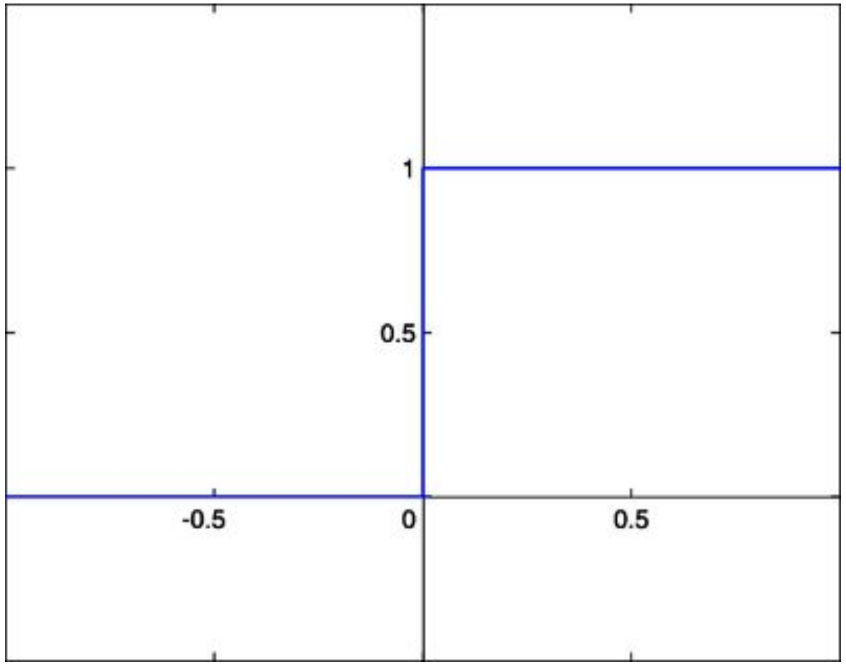
$$f\left(\sum_i w_i x_i + b\right)$$

Activation Functions

Non Linearity in Neural Networks

- Non-linearity is an essential component in the success of Neural Networks
- The most obvious activation function is a Heaviside function
- It has a biological meaning of a firing neuron
- What is the problem with it?



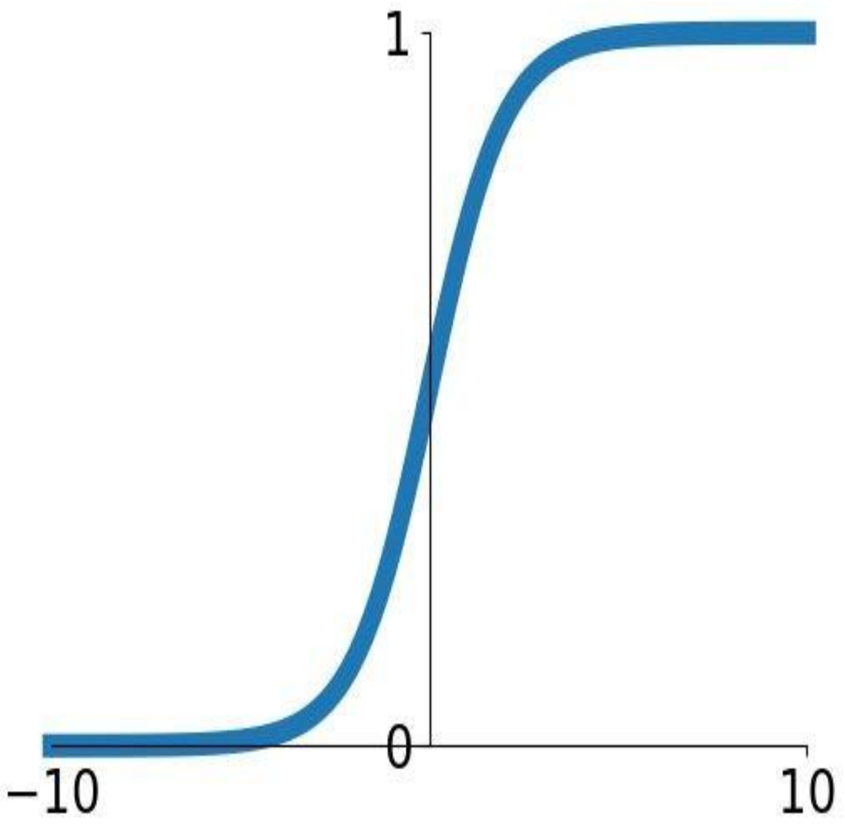


Non-Linear Activation Functions

Neural Networks require non-linear activation functions

- The first activation function was ***the Sigmoid***
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

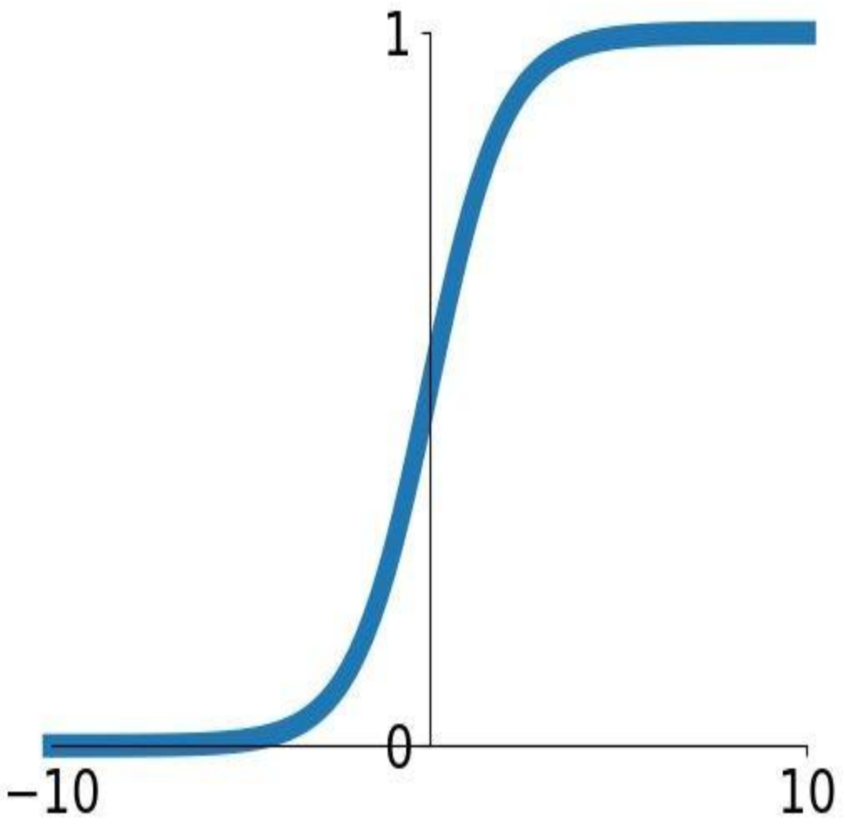
$$\sigma(x) = 1/(1 + e^{-x})$$



Activation Function - Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron
- How does the gradient of a sigmoid look?

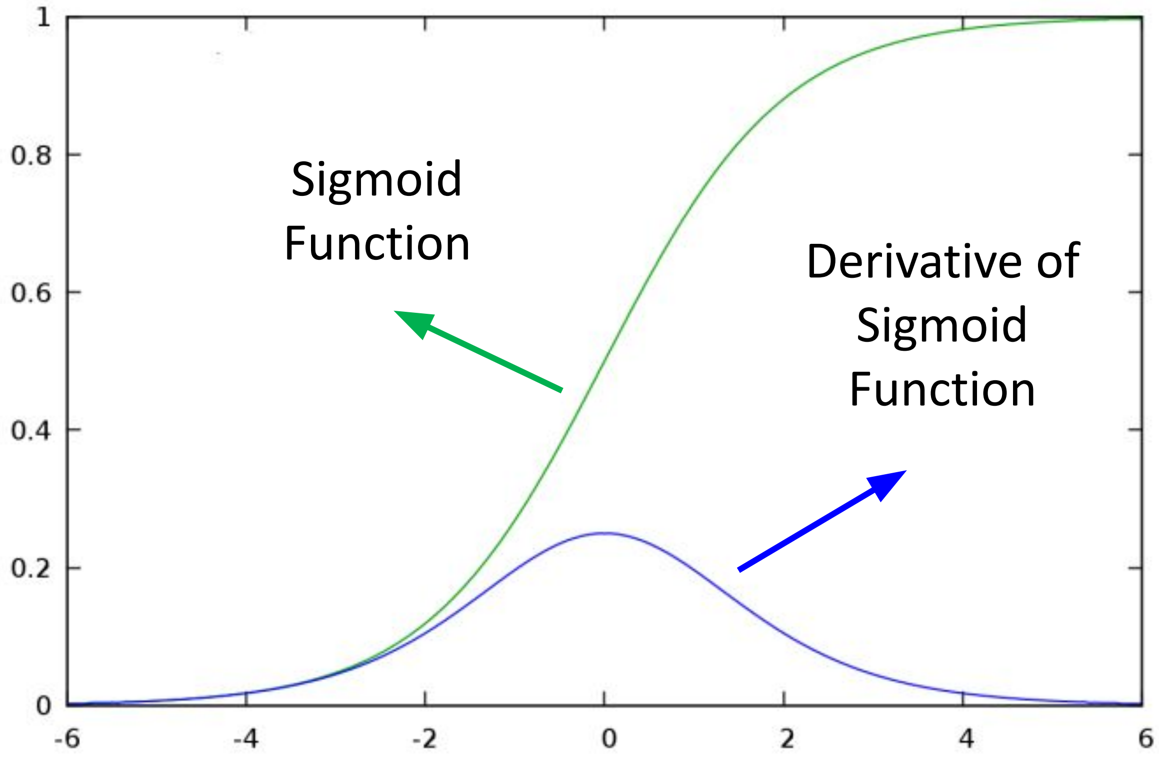
$$\sigma(x) = 1/(1+e^{-x})$$



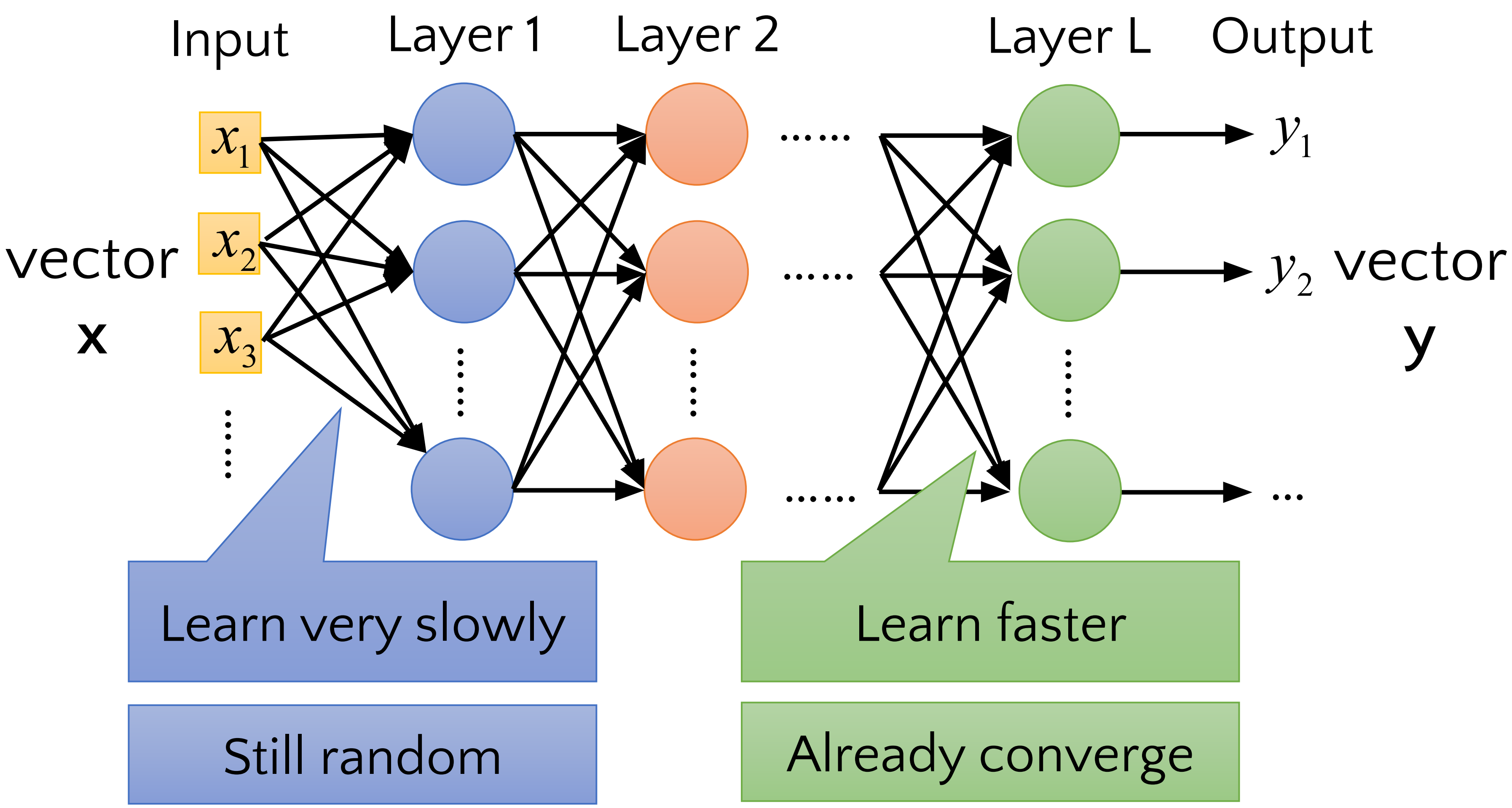
Problem of Sigmoid

- Saturated neurons "kill" the gradients
- Derivative of Sigmoid Function is always smaller than 1
- Error signal is getting smaller and smaller

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$$



Vanishing Gradient Problem



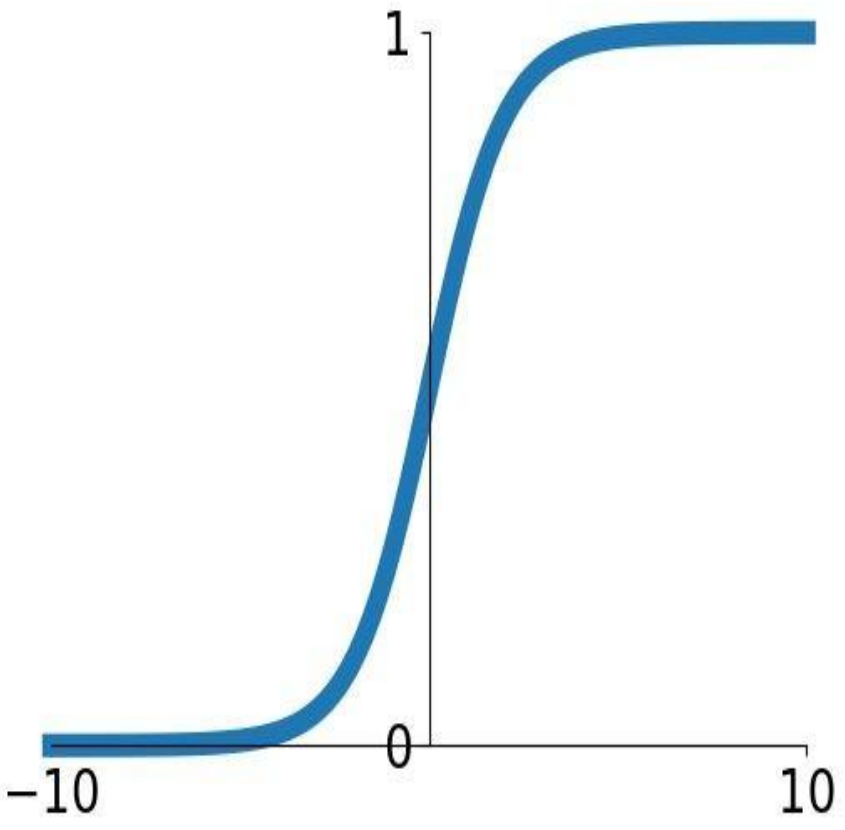
The weights might converge based on random weights…

Slide credit: Lior Rokach

Activation Function - Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron
- How does the gradient of a sigmoid look?
- exp() is slightly computationally expensive

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

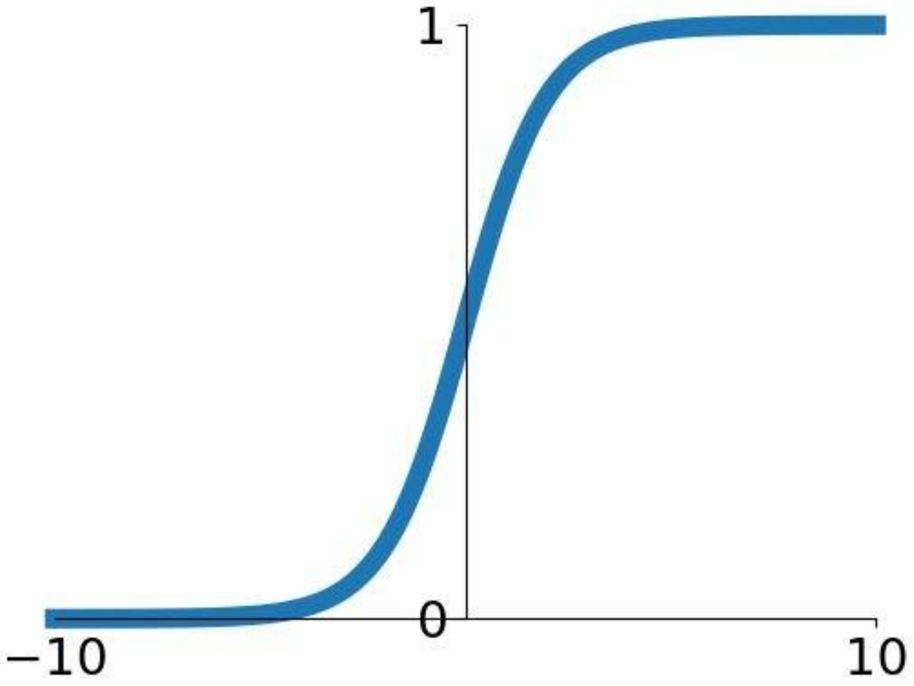


Activation Functions - Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

- Saturated neurons "kill" the gradients
- exp() is slightly computationally expensive
- Sigmoid outputs are not zero-centered

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

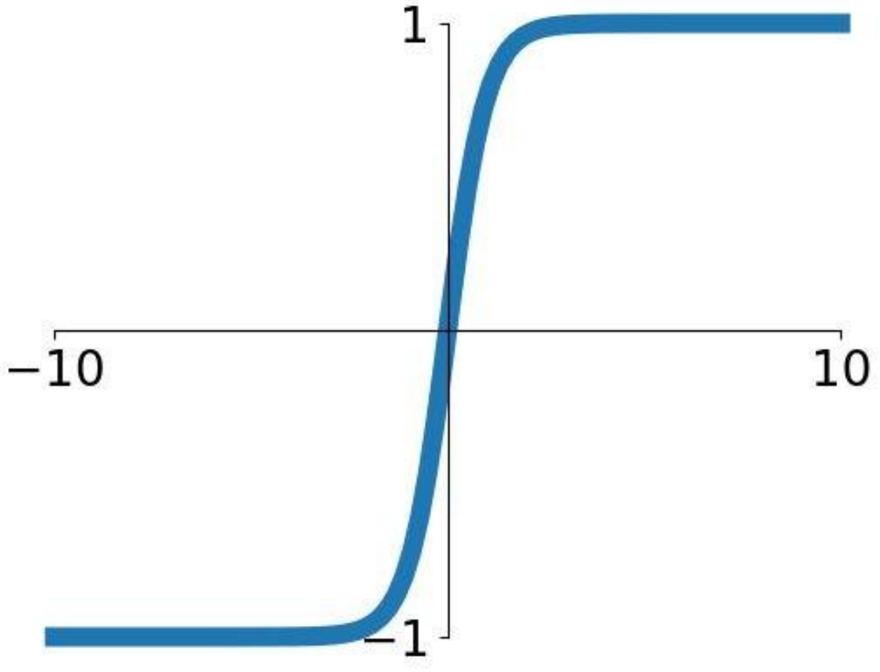


Activation Functions - Tanh

Hyperbolic Tangent

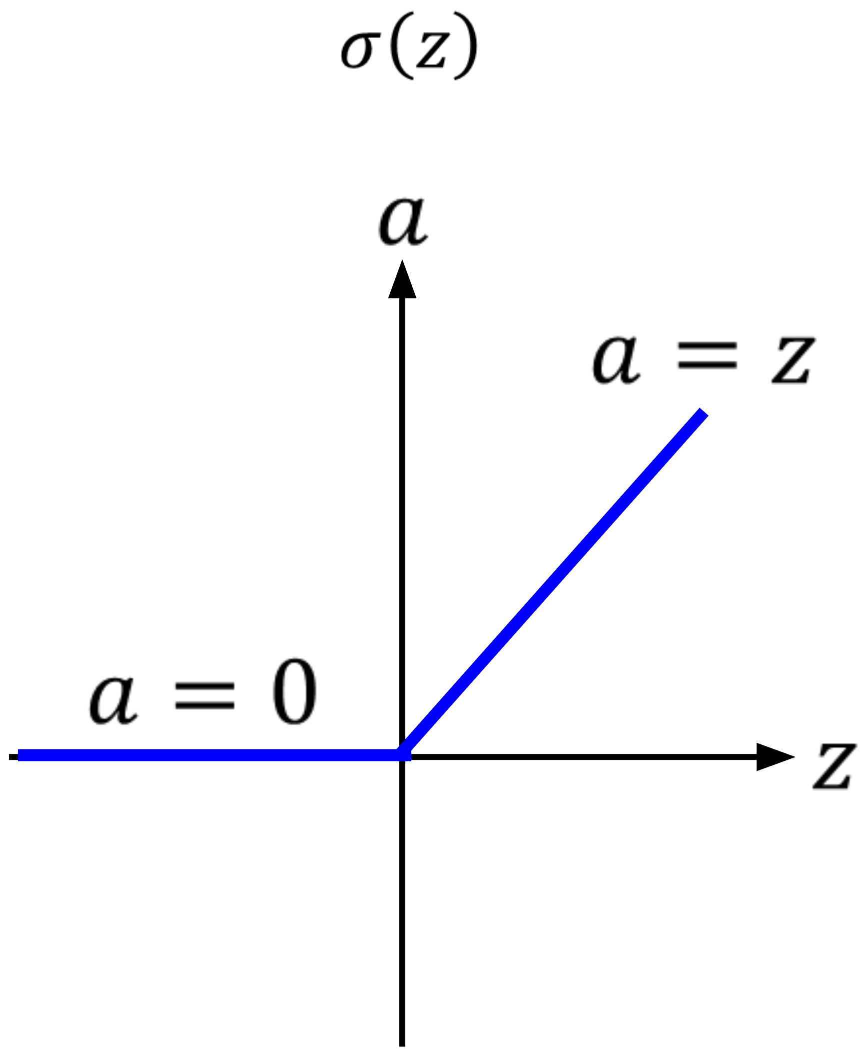
- Squashes numbers to range [–1,1]
- zero centered (nice)
- still kills gradients when saturated :(

$\tanh(x) = \sinh(x)/\cosh(x) = (e^{x} - e^{-x})/(e^{x} + e^{-x})$



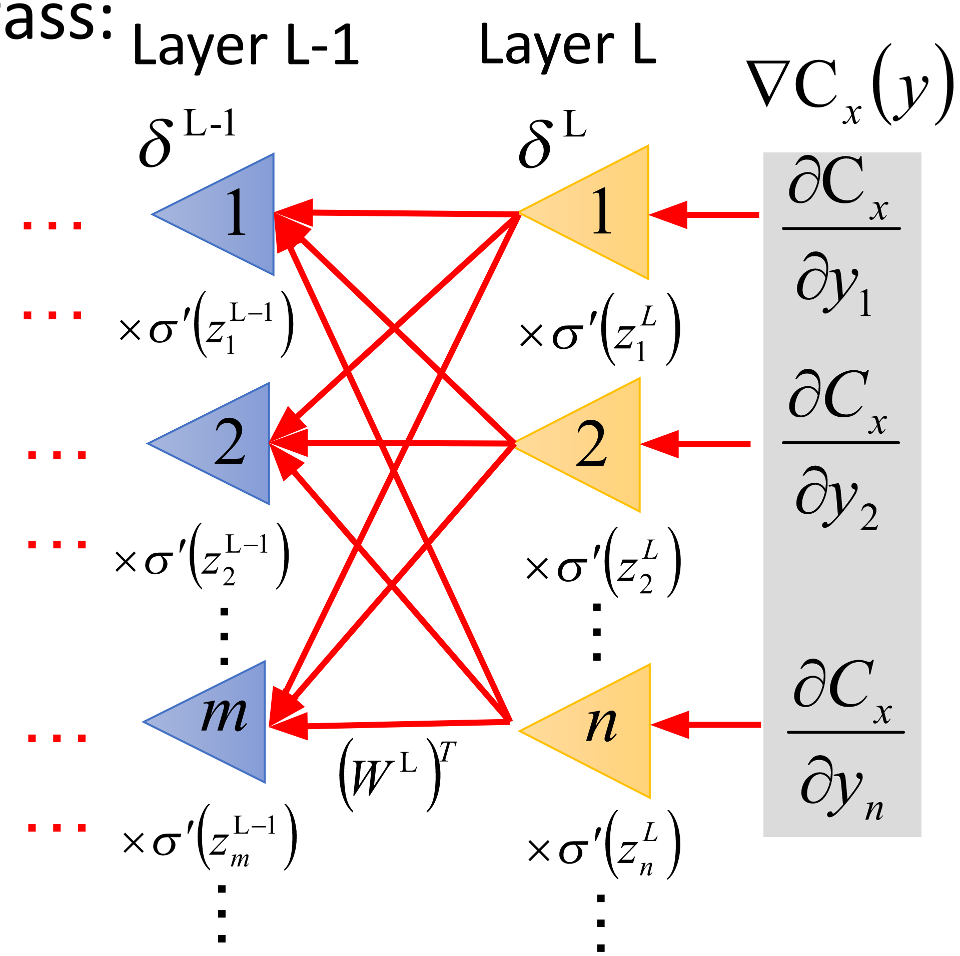
ReLU

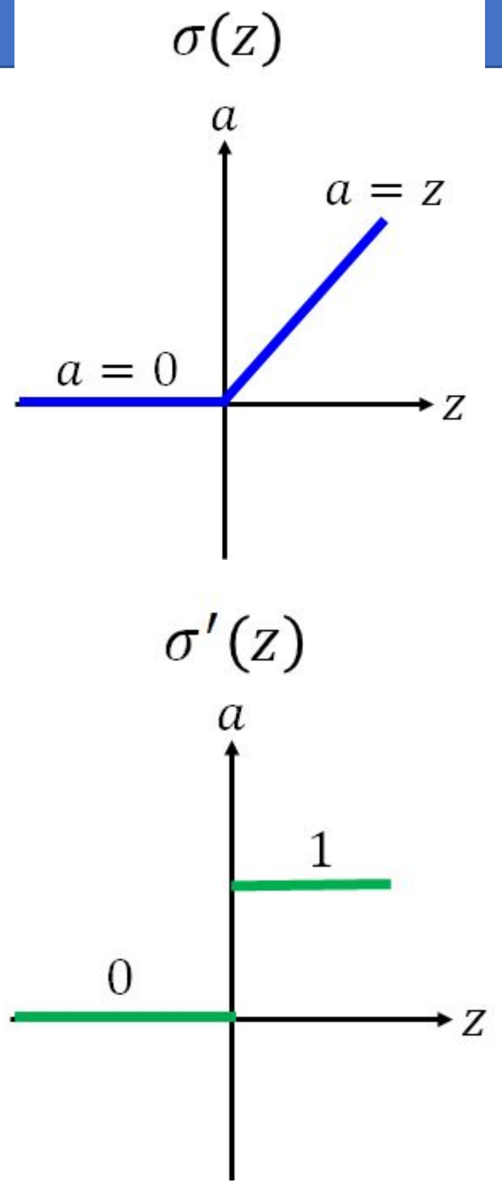
- Does not saturate
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice
- Output is not zero-centered



ReLU

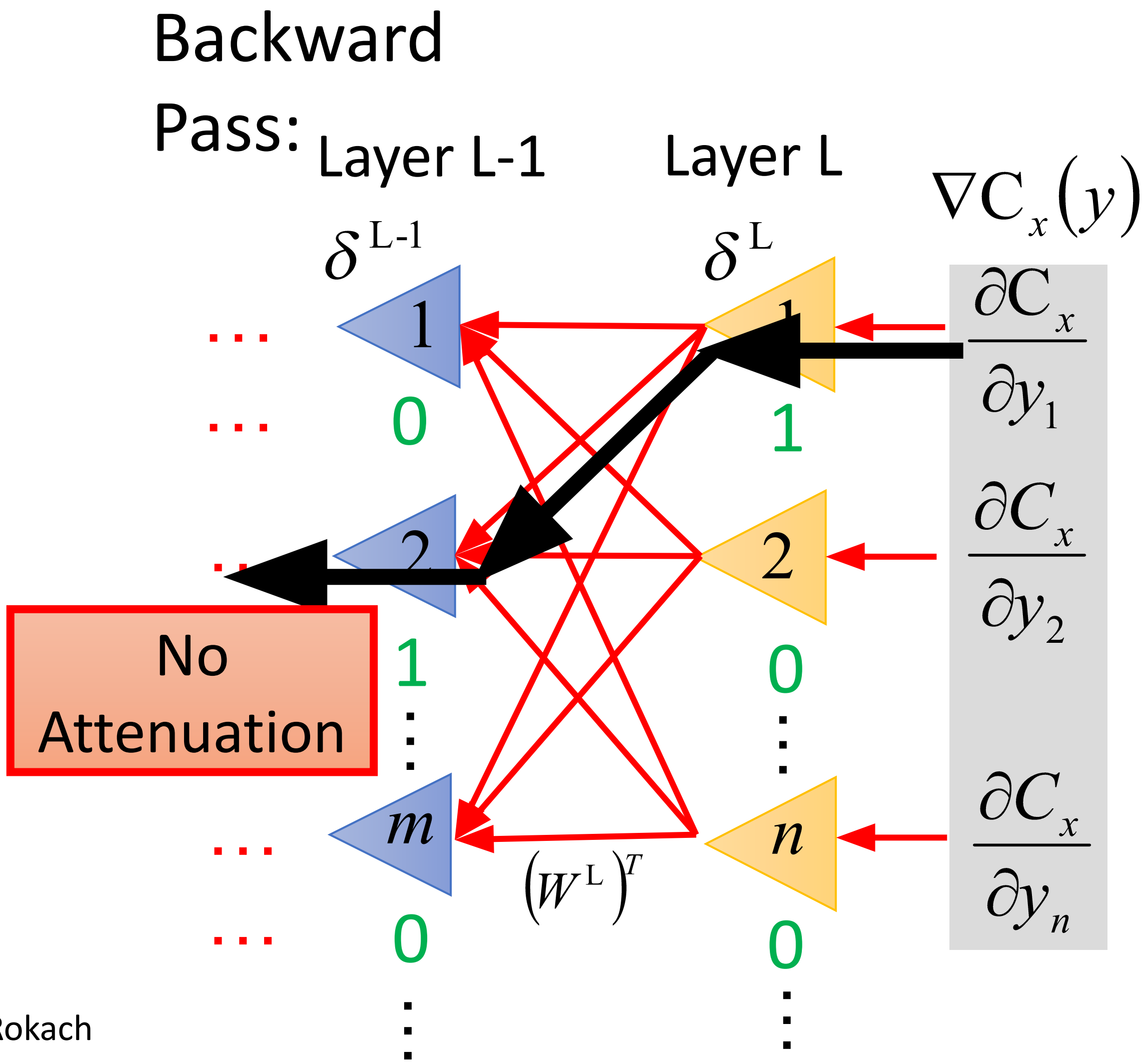
Backward Pass:

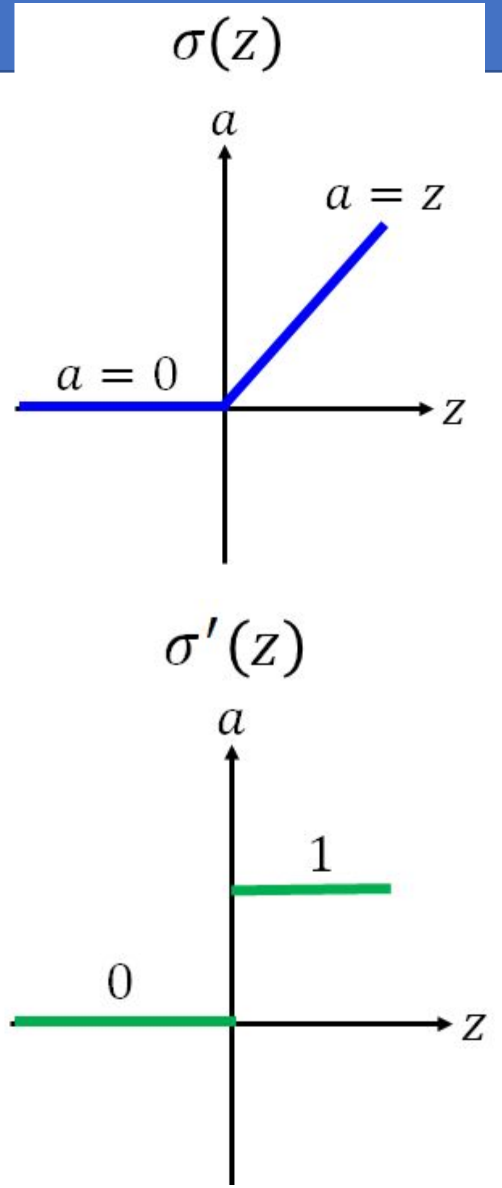




Slide credit: Lior Rokach

ReLU



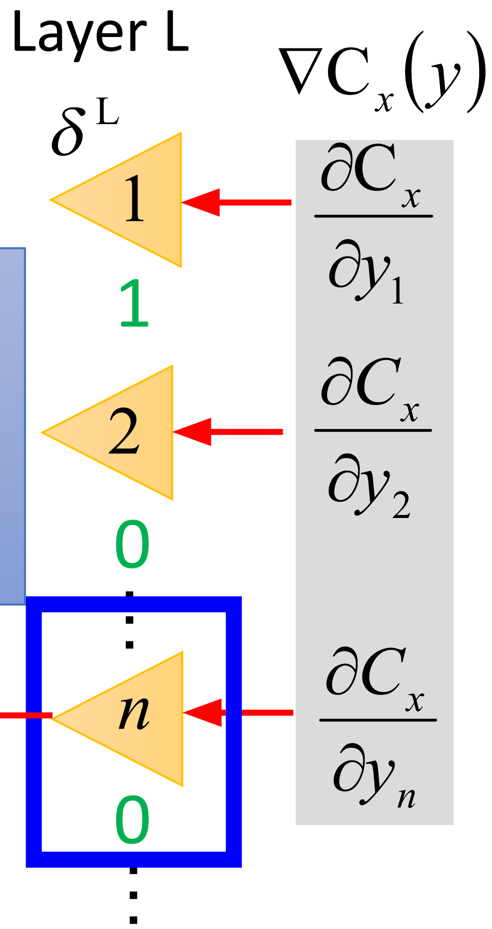


Slide credit: Lior Rokach

ReLU

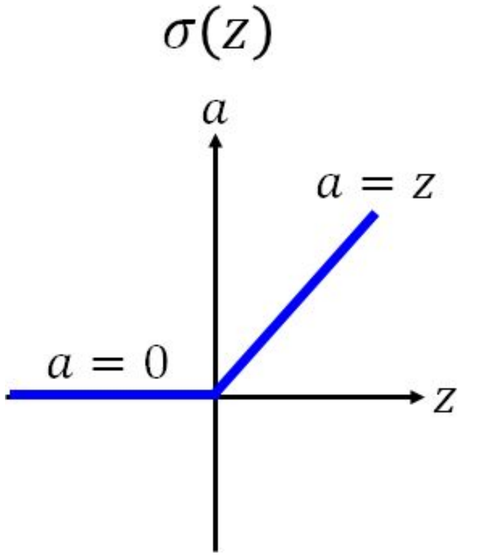
Backward Pass:

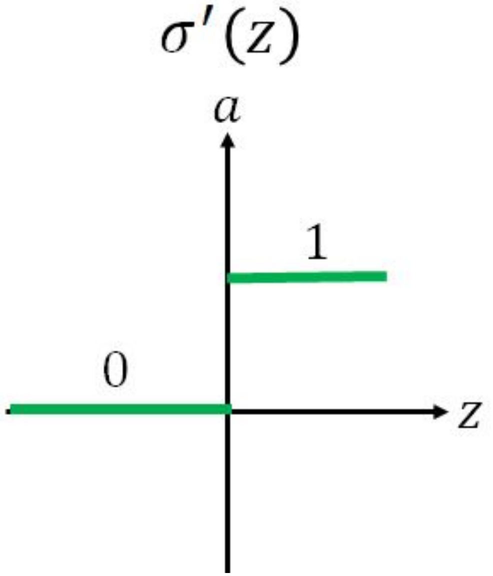
$$\frac{\partial C_x}{\partial w_{nj}^L} = \frac{\partial z_n^L}{\partial w_{nj}^L} \frac{\partial C_x}{\partial z_n^L}$$



All the weights connected to this neuron will have zero gradient.

$$\delta_n^L = \frac{\partial C_x}{\partial z_n^L} = 0$$



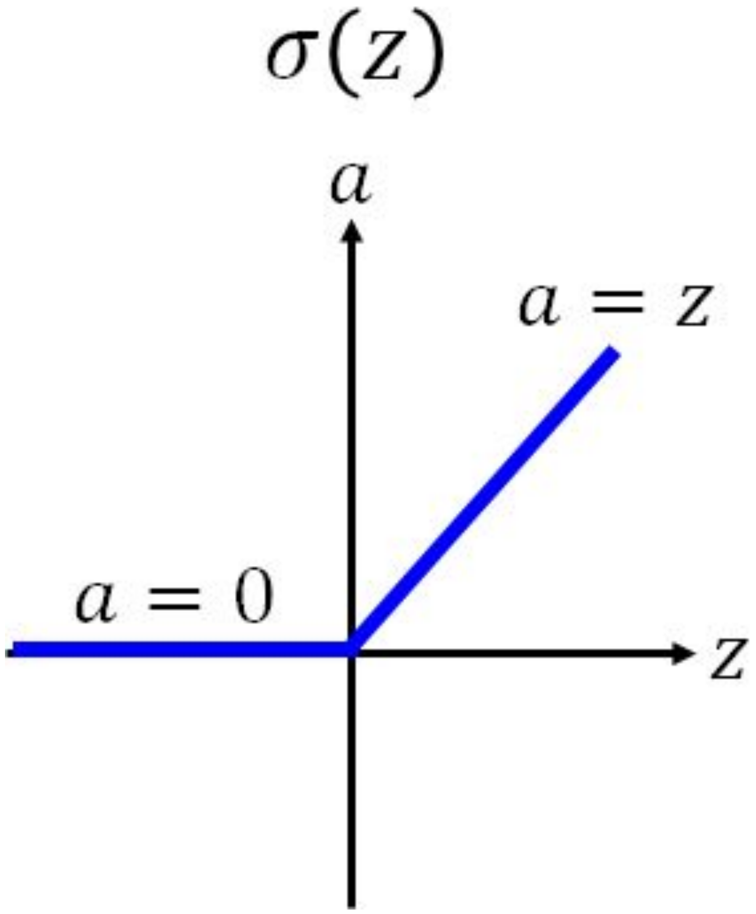


Slide credit: Lior Rokach

ReLU

- If the activation is >0, then the gradient is 1
- There is no attenuation!

- If the activation is 0, then the gradient is also 0
- Due to the chain rule and the multiplication by 0, all the weights connected to that neuron will have 0 gradient and will not change



Dying ReLU Neurons

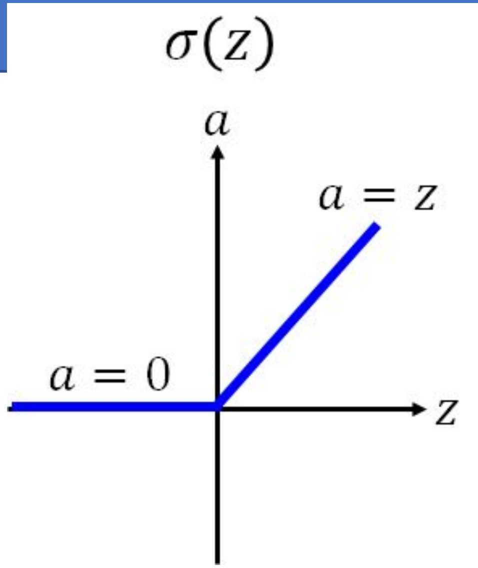
- ReLU neurons can become inactive for some input (produce 0)
- The gradient in this case is 0
- If the neuron is inactive for all inputs in the training set, it will never change state
- The neuron becomes stuck in a perpetually inactive state and “dies”
- In some cases, large numbers of neurons in a network can become stuck in dead states, effectively decreasing the model capacity
- This problem typically arises when the learning rate is set too high

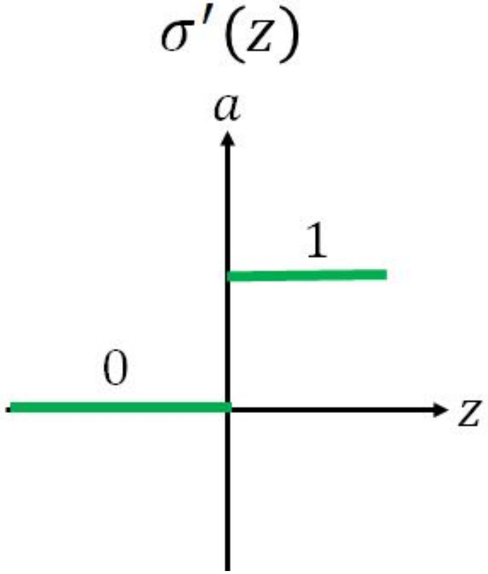
Discussion

- Does a dead neuron during training always stay dead during inference?
- How can we identify this problem?
- Can lowering the learning rate help?
- Can adding more data help?
- How can we solve the problem?

ReLU

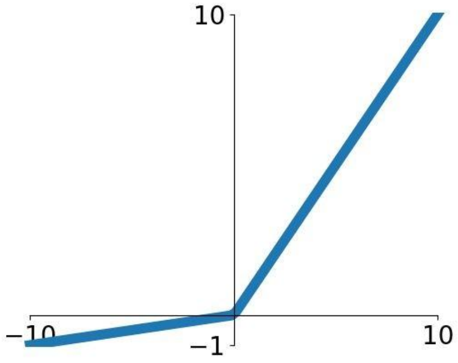
- Gradient not vanishing
- Efficient computation
- Sparse activation
- Non-zero centered
- Can potentially blow up as it is unbounded





Activation Functions

[Mass et al., 2013] [He et al., 2015]



- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)

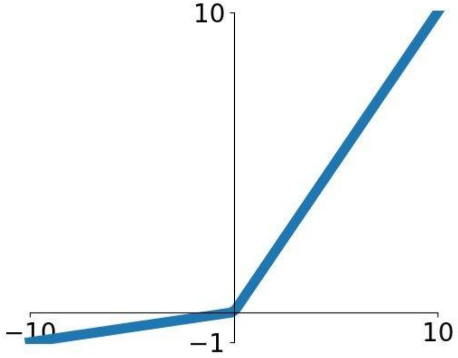
Leaky ReLU

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

Activation Functions

[Mass et al., 2013]
[He et al., 2015]

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)



Leaky ReLU

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

Parametric Rectifier (PReLU)

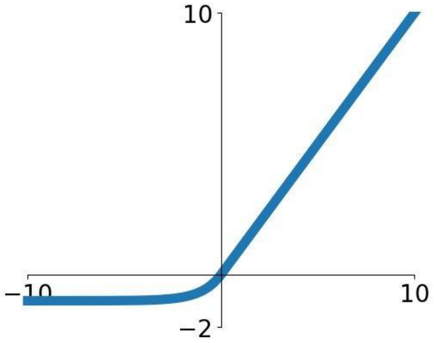
$$f(x) = \max(\alpha x, x)$$

Backprop into α
(parameter)

Activation Functions

[Clevert et al., 2015]

Exponential Linear Units (ELU)



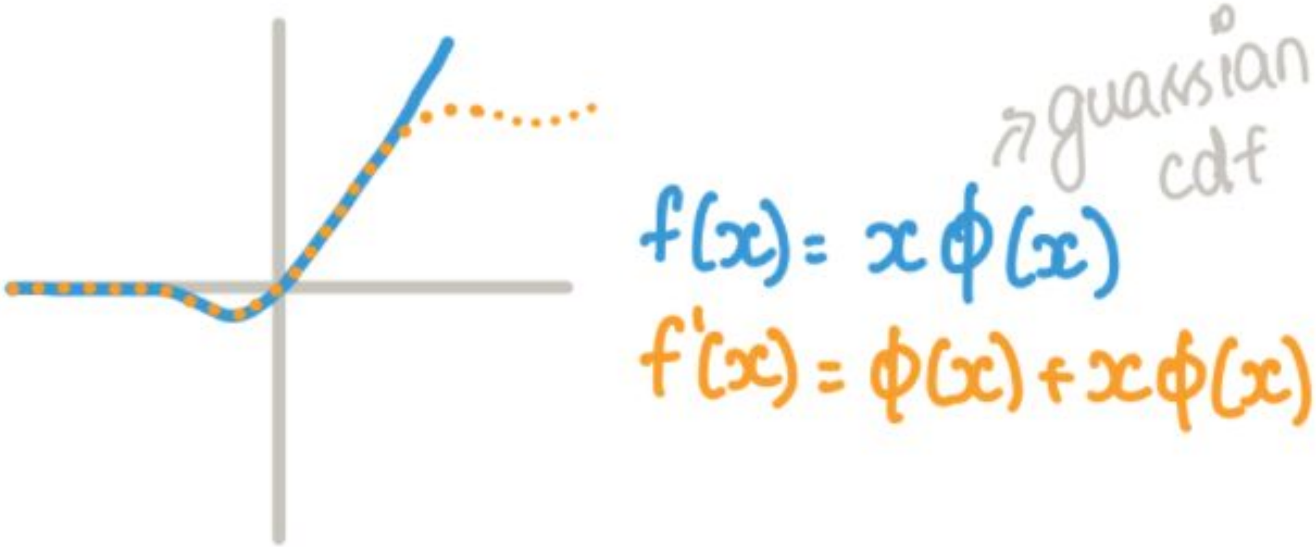
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

- Computation requires exp()

Gaussian Error Linear Unit

- Introduced in 2018
- Used in GPT-3, BERT and other SOTA NLP models
- It combines dropout (zeroing out neurons randomly for a sparse network), zone out (maintain previous value), and ReLU
- It weights inputs by percentile rather than gates, leading to a smoother version of ReLU
- The derivative is highly curved



Activation Functions – Best Practices

- Use ReLU.
  Check for dead neurons
  Be careful with your learning rates
- Try out Leaky ReLU / ELU / GELU
- Avoid sigmoid
- When fine tuning, use the activation function used by the original trainers of the pre-trained network

In PyTorch

```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

In PyTorch

Non-linear Activations (weighted sum, nonlinearity)

`nn.ELU`	Applies the element-wise function:
`nn.Hardshrink`	Applies the hard shrinkage function element-wise:
`nn.Hardsigmoid`	Applies the element-wise function:
`nn.Hardtanh`	Applies the HardTanh function element-wise
`nn.Hardswish`	Applies the hardswish function, element-wise, as described in the paper:
`nn.LeakyReLU`	Applies the element-wise function:
`nn.LogSigmoid`	Applies the element-wise function:
`nn.MultiheadAttention`	Allows the model to jointly attend to information from different representation subspaces.
`nn.PReLU`	Applies the element-wise function:
`nn.ReLU`	Applies the rectified linear unit function element-wise:
`nn.ReLU6`	Applies the element-wise function:
`nn.RReLU`	Applies the randomized leaky rectified liner unit function, element-wise, as described in the paper:
`nn.SELU`	Applied element-wise, as:
`nn.CELU`	Applies the element-wise function:
`nn.GELU`	Applies the Gaussian Error Linear Units function:
`nn.Sigmoid`	Applies the element-wise function:
`nn.SiLU`	Applies the silu function, element-wise.
`nn.Softplus`	Applies the element-wise function:
`nn.Softshrink`	Applies the soft shrinkage function elementwise:
`nn.Softsign`	Applies the element-wise function:
`nn.Tanh`	Applies the element-wise function:
`nn.Threshold`	Thresholds each element of the input Tensor.

Non-linear Activations (other)

`nn.Softmin`	Applies the Softmin function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range $[0, 1]$ and sum to 1.
`nn.Softmax`	Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum to 1.
`nn.Softmax2d`	Applies SoftMax over features to each spatial location.
`nn.LogSoftmax`	Applies the $\log(\text{Softmax}(x))$ function to an n-dimensional input Tensor.
`nn.AdaptiveLogSoftmaxWithLoss`	Efficient softmax approximation as described in Efficient softmax approximation for GPUs by Edouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou.

Suggested Reading

- https://cs231n.github.io/neural-networks-1/#intro
- https://machinelearningknowledge.ai/activation-functions-neural-network/

  or

  https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/
- https://medium.com/@snaily16/what-why-and-which-activation-functions-b2bf748c0441

Recommended Resources

- https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/
- https://www.youtube.com/watch?v=-7scQpJT7uo
- https://atcold.github.io/pytorch-Deep-Learning/en/week11/11-1/
- https://arxiv.org/abs/2005.00817

Home Experiments

- Change activation function to one of these: https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity
- Implement a custom non-linear activation function - based on this guide – https://towardsdatascience.com/extending-pytorch-with-custom-activation-functions-2d8b065ef2fa