Course Notes: Deep Learning for Visual Computing

Peter Wonka

August 30, 2021

Contents

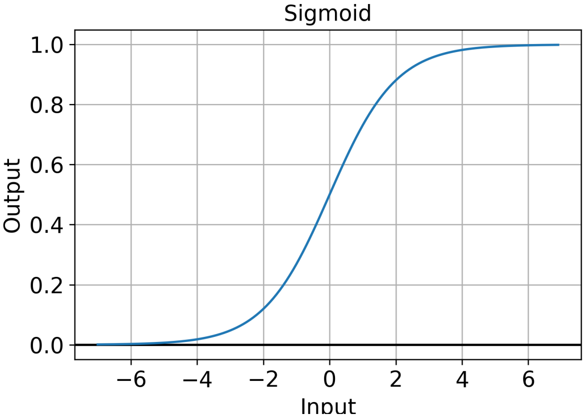
1 Non-linear Activation Functions

1.1 General Comments

- Typically activation functions have one input and one output
  - True for Sigmoid, Tanh, ReLU, LeakyReLU, ...
  - Exception: Maxout
- Typically activation functions do not have a learnable parameter that the network can train on
  - True for Sigmoid, Tanh, ReLU, LeakyReLU, ...
  - Exception: PReLU
  - It is important to distinguish between a hardcoded constant and a trainable parameter in the following
- Activation functions operate component wise on tensors

1.2 Sigmoid

$$\text{Sigmoid}(x) = \frac{1}{1 + \exp(-x)} \tag{1.1}$$



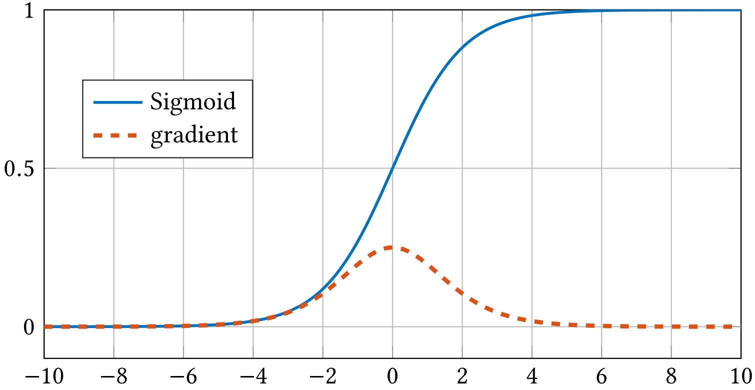
- No parameter to learn for the network
- Output is in the interval $[0, 1]$
  - Small negative numbers $\rightarrow 0$, large positive numbers $\rightarrow 1$
- Historical importance, semantic interpretation as the firing rate of a neuron:
  - from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1)
- Interpretation as probability

1.3 Properties of Activation Functions

- How large is the derivative (gradient) of the function?
  - Sigmoid can saturate and kill the gradients. Gradient is very small far from 0, e.g. at $+10$ and -10 it is almost 0
  - If the gradient is very small, no signal will flow through the neuron during back-propagation
  - If initializing the network weights leads to values in the saturated region, it can take a long time to change.
- Is the output zero-centered?
  - Sigmoid outputs are not zero-centered
  - Linear layers compute a sum of the form $\sum x_i w_i + b$, w_i are network parameters and x_i are the inputs to the layer
  - If all x_i are positive the partial derivatives with respect to the w_is will have the

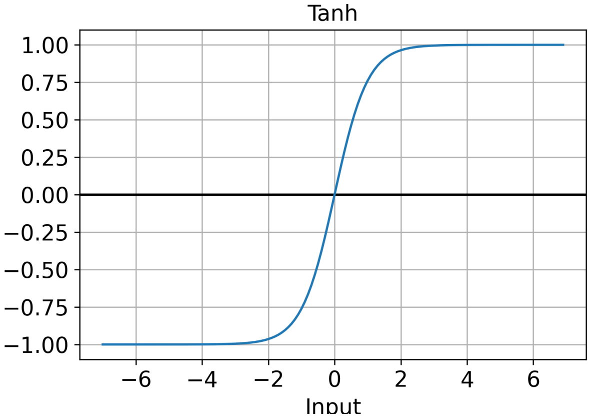
same sign

  - Less problematic for mini-batches where gradients are averaged
- Is the function smooth?
  - Sigmoid is smooth
- How expensive is the function to compute?
  - exponential function in Sigmoid is expensive
  - Bigger concern on mobile devices and CPUs than GPUs
- Is the function monotone?
- Is the derivative monotone?
- Does the function approximate the identity near 0?


1
0.5
0
Sigmoid
gradient
−10
−8
−6
−4
−2
0
2
4
6
8
10

1.4 Tanh

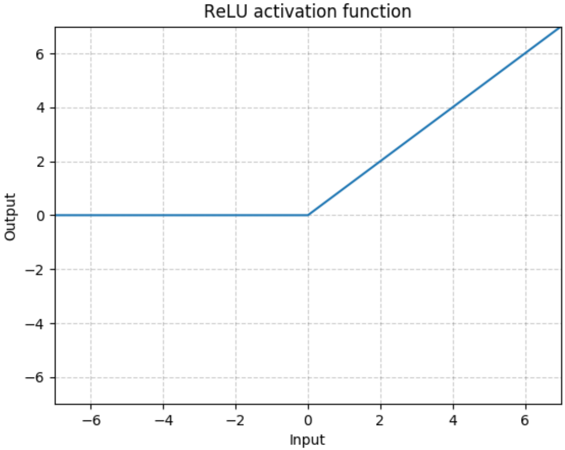
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.2}$$



- Output is in the interval $[-1, 1]$
- Saturates like the sigmoid function
- Output is zero centered
- Tanh is simply a scaled and shifted sigmoid function:
  - $\text{Tanh}(x) = 2\text{Sigmoid}(2x) - 1$

1.5 ReLU

$$\mathsf{ReLU}(x) = \max(0, x) \tag{1.3}$$



- Output is in the interval $[0, +\infty]$
- Empirically much better convergence than tanh / sigmoid

  - e.g. a factor of 6 in Krizhevsky et al.
- Very simple to implement
- ReLU units can die (i.e. never activate again).
  - If the ReLU unit does not activate for any input, gradient is always 0
  - Happens more often with large learning rates?

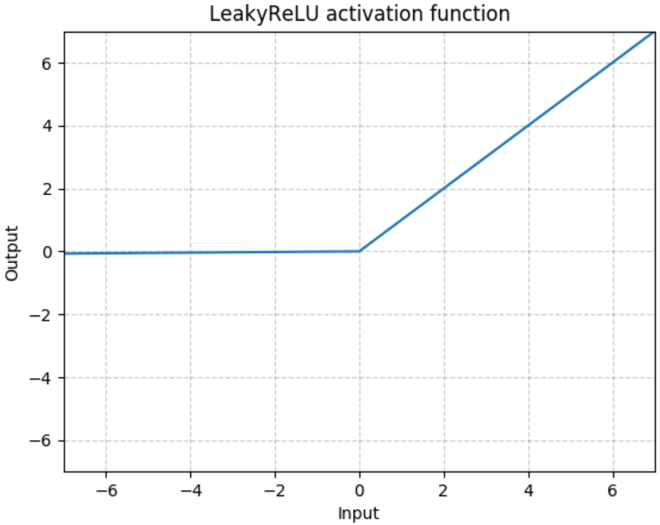
1.6 Leaky ReLU

$$\text{LeakyRELU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \text{negative\_slope} \times x, & \text{otherwise} \end{cases}$$

- Alternative formulation:

$$\text{LeakyReLU}(x) = \max(0, x) + \text{negative\_slope} * \min(0, x) \tag{1.4}$$

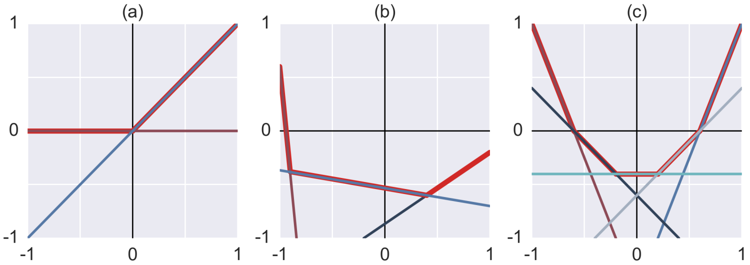
- Output is in the interval $[-\infty, +\infty]$
- negative_slope is a hardcoded parameter, not learnable
  - default parameter in PyTorch is 0.01
- Idea: LeakyReLU cannot die, because there is a gradient for positive and negative inputs
  - While the idea is very intuitive, a clear benefit has not been established experimentally


LeakyReLU activation function
Output
Input
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

1.7 Maxout

$$\text{Maxout}(x_1, \ldots, x_k) = \max(x_1, \ldots, x_k) \tag{1.5}$$

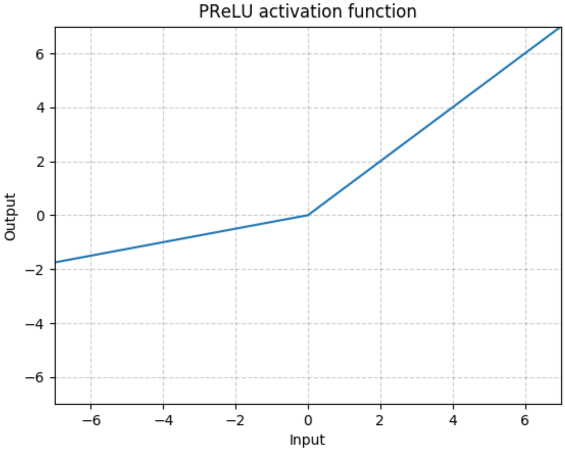
- Literature: Goodfellow et al., Maxout Networks
- The idea of maxout is to compute multiple possible outputs for a neuron and then choose the maximum value.
- Requires k-times the number of weights, because instead of one value per neuron we have to compute k values.
- Comparison
  - MaxPooling computes the maximum among neighboring pixels in the same channel
  - Maxout combines k candidate values for the same pixel. Basically you compute k channels instead of one and combine them with max.



- Maxout can build piecewise linear convex functions (if applied to linear input functions)

1.8 PReLU

$$\text{PReLU}(x) = \max(0, x) + a * \min(0, x) \tag{1.6}$$

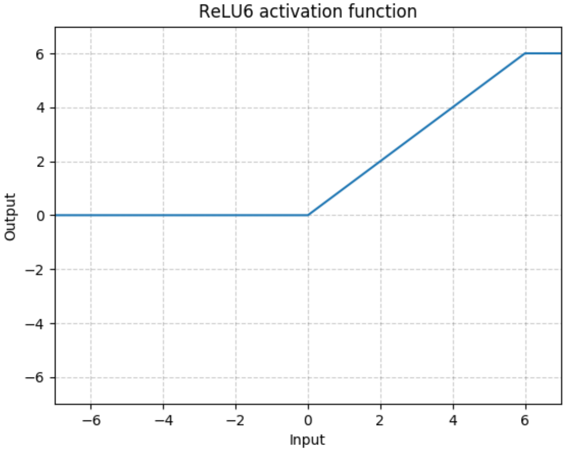


- Name: **Parametric ReLU**
- Extends the LeakyReLU further by making a a learnable parameter

  - PyTorch recommends not to use weight decay on the parameter a
  - Initial value in PyTorch: $a = 0.25$
- Literature: He et al., Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

1.9 ReLU6

$$\text{ReLU6}(x) = \min(\max(0, x), 6) \tag{1.7}$$

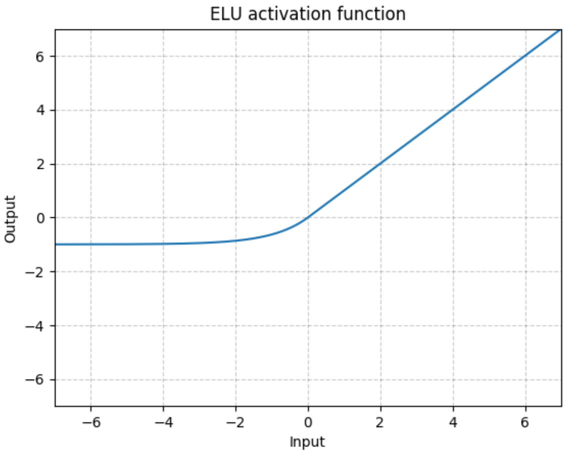


- ReLU, but large values are clamped to 6
- Why is there such a wierd activation function in PyTorch?

- A: Krizhevsky, Convolutional Deep Belief Networks on CIFAR-10
- useful for mobile computing and fixed point arithmetic?

1.10 ELU

$$\mathsf{ELU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x) - 1)) \tag{1.8}$$



- Name: **Exponential Linear Unit**
- Default in PyTorch: $\alpha = 1$

- α is not learnable

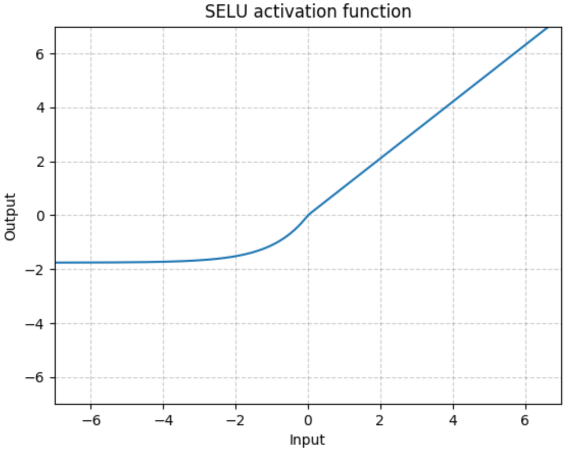
1.11 RRelu

$$\mathrm{RReLU}(x) = \max(0, x) + a * \min(0, x) \tag{1.9}$$

- α is randomly sampled in an interval $[lower, upper]$
  - α, $lower$, $upper$ are hardcoded parameters and not learnable
- Literature: Xu et al., Empirical Evaluation of Rectified Activations in Convolutional Network
  - The authors claim some success with RReLU
  - Unclear if it's really a good idea

1.12 SELU

$$\mathsf{SELU}(x) = \mathsf{scale} * (\max(0, x) + \min(0, \alpha * (\exp(x) - 1))) \tag{1.10}$$

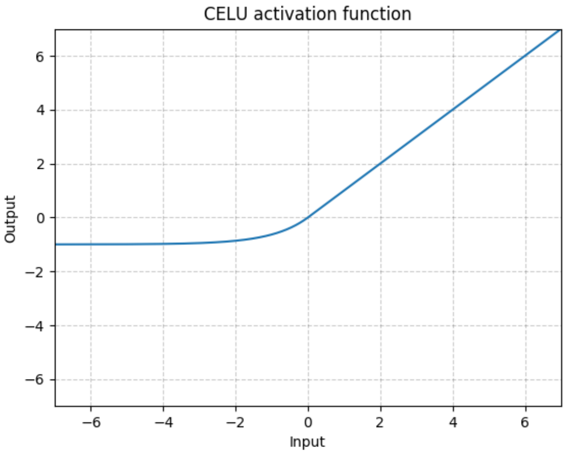


- Proposed Idea: build a self-normalizing network to avoid batch-normalization
- $\alpha = 1.6732632423543772848170429916717$

- scale $= 1.0507009873554804934193349852946$
- Literature: Klambauer et al., Self-Normalizing Neural Networks

1.13 CELU

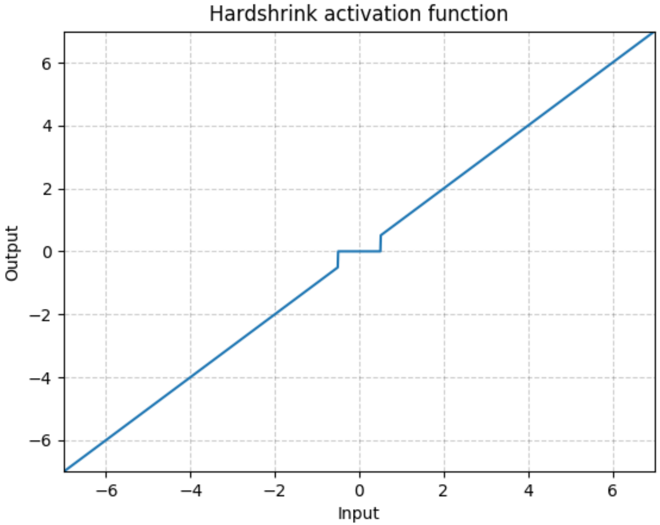
$$\text{CELU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x/\alpha) - 1)) \tag{1.11}$$



- Parameter: α, default $\alpha = 1$, not learnable
- Literature: Barron, Continuously Differentiable Exponential Linear Units

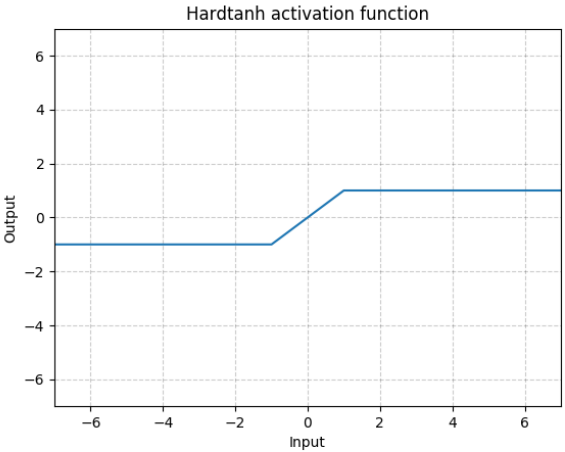
1.14 Hardshrink

$$\text{HardShrink}(x) = \begin{cases} x, & \text{if } x > \lambda \\ x, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$


Hardshrink activation function
Output
Input
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

1.15 Hardtanh

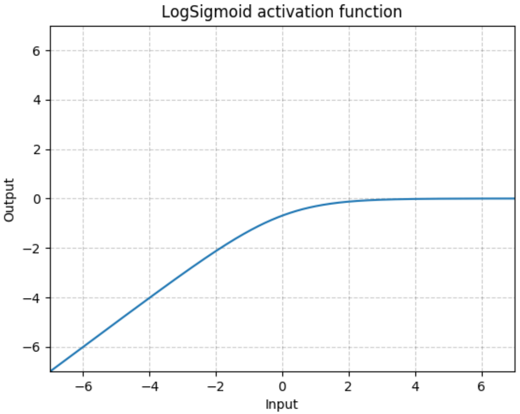
$$\text{HardTanh}(x) = \begin{cases} 1 & \text{if } x > 1 \\ -1 & \text{if } x < -1 \\ x & \text{otherwise} \end{cases}$$



- minimum and maximum value can be given as parameter

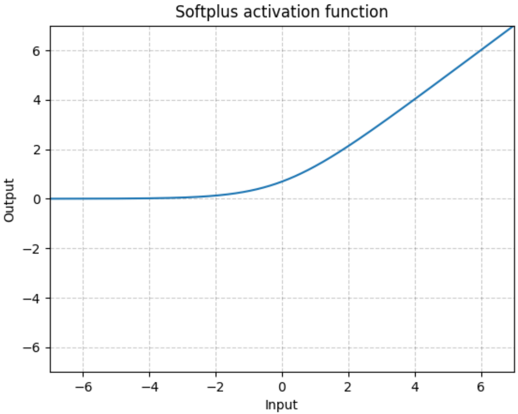
1.16 LogSigmoid

$$\text{LogSigmoid}(x) = \log\left(\frac{1}{1+\exp(-x)}\right) \tag{1.12}$$



1.17 Softplus

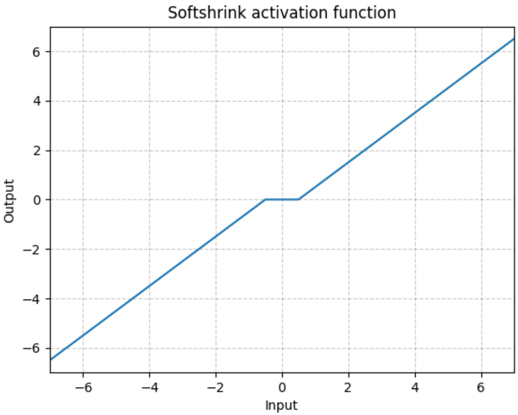
$$\text{Softplus}(x) = \frac{1}{\beta} * \log(1 + \exp(\beta * x)) \tag{1.13}$$



- smooth approximation of the ReLU
- parameters:
  - β, default $\beta = 1$
  - treshold, for inputs above the treshold the function reverts to a linear function for numerical stability
- Output is always positive

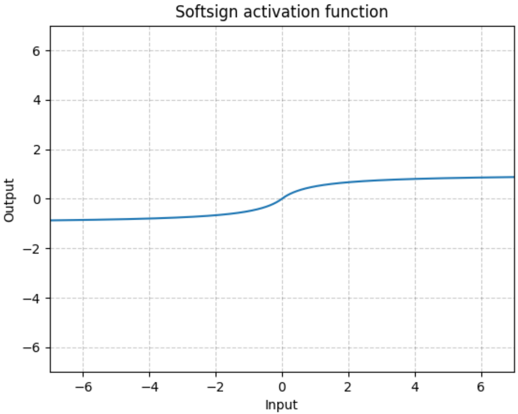
1.18 Softshrink

$$\text{SoftShrinkage}(x) = \begin{cases} x - \lambda, & \text{if } x > \lambda \\ x + \lambda, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$


Softshrink activation function
Output
Input
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

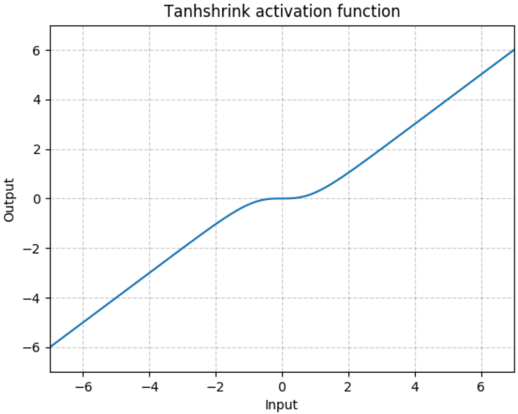
1.19 Softsign

$$\text{SoftSign}(x) = \frac{x}{1+|x|} \tag{1.14}$$



1.20 Tanhshrink

$$\text{Tanhshrink}(x) = x - \text{Tanh}(x) \tag{1.15}$$



1.21 Treshold

$$\begin{cases} x, & \text{if } x > \text{threshold} \\ \text{value}, & \text{otherwise} \end{cases}$$

1.22 Softmin

$$\text{Softmin}(x_i) = \frac{\exp(-x_i)}{\sum_j \exp(-x_j)}$$

1.23 Softmax

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

- Output values will be in the range $[0, 1]$
- There is also a 2D version Softmax2d (softmax per pixel)

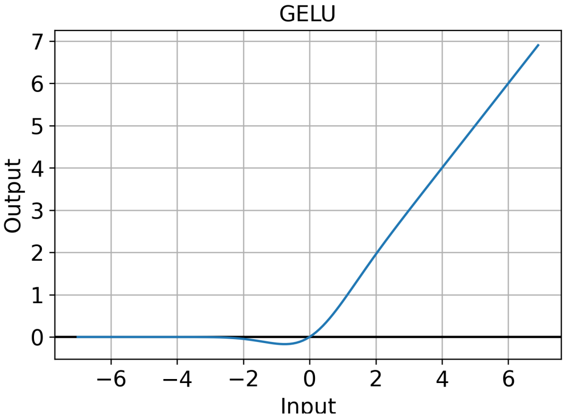
1.24 LogSoftmax

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

- Often the output of softmax is further processed by a log function
- Computing log and softmax together has a more efficient and more stable implementation

1.25 GELU

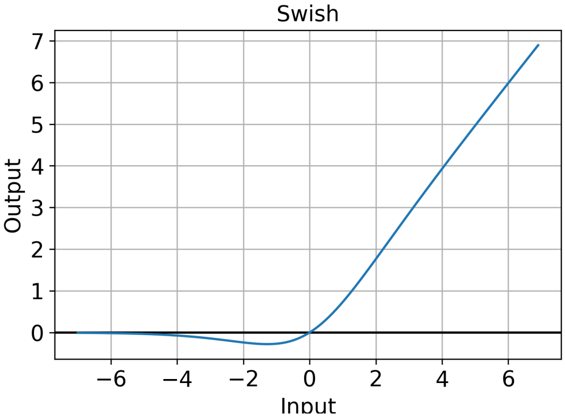
$$\text{GELU}(x) = x * \Phi(x) \tag{1.16}$$



- $\Phi(x) is the Cumulative Distribution Function for the Gaussian Distribution$

1.26 Swish

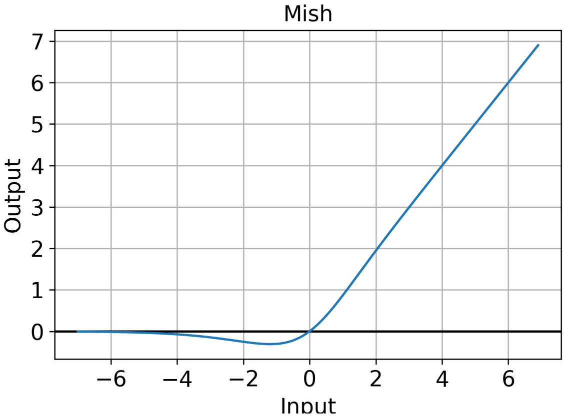
$$\text{swish}(x) = x\text{sigmoid}(x) \tag{1.17}$$



- Literature: Swish: a Self-Gated Activation Function
- Non-monotonic function
- Authors claim this property is desirable and brings an advantage
- Smooth
- **Self-gating**

1.27 Mish

$$\text{Mish}(x) = x \tanh(\text{softplus}(x)) = x \tanh(ln(1 + e^x)) \tag{1.18}$$



- Literature: Mish: A Self Regularized Non-Monotonic Neural Activation Function
- Endorsed by FastAI: blogpost

1.28 Problem of Non-differentiable Functions

- Not all activation functions are differentiable at all points
- Solution:
  - Use concepts such as subgradient
  - Use left or right derivative
- For example, ReLU does not have a defined gradient at 0
  - Left derivative $= 0$
  - Right derivative $= 1$
  - Return one of those values in a software implementation

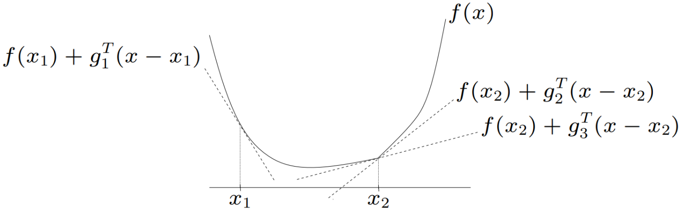
1.29 Subgradient Review

Subgradient of a function

g is a **subgradient** of f (not necessarily convex) at x if

$$f(y) \geq f(x) + g^T(y - x) \quad \text{for all } y$$

($\Longleftrightarrow$ $(g, -1)$ supports $\mathbf{epi}\, f$ at $(x, f(x))$)



g_2, g_3 are subgradients at x_2; g_1 is a subgradient at x_1

1.30 Recommendation on what to use

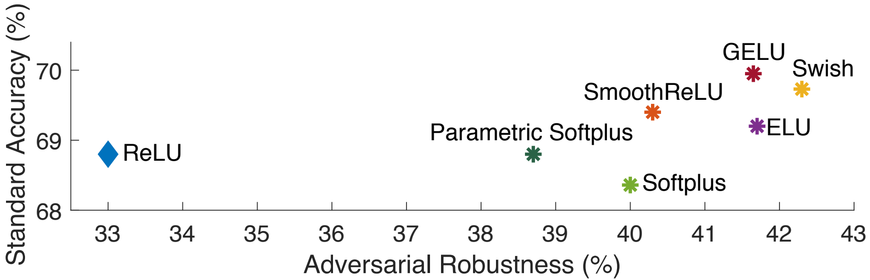
- Try ReLU first
- Use whatever the best other networks in your field are doing
- It is rare to mix different types of activation functions in a network

1.31 Comparing Activation Functions

- Many comparisons, no clear winner overall
- Activation function interacts with all other network components: optimizer, learning rate, network depth, type of network, data, initialization, ...

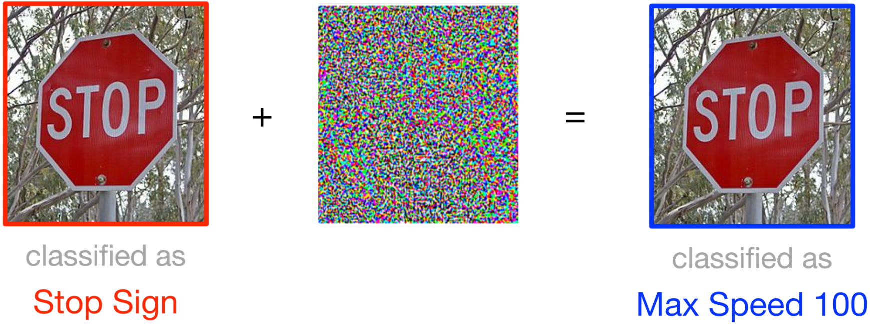
1.32 Comparing Activation Functions

- Literature: Xie et al, Smooth Adversarial Training
- Authors claim: smooth activation functions improve adversarial training. Compared to ReLU, all smooth activation functions significantly boost robustness, while keeping accuracy almost the same.



1.33 What are Adversarial Examples?

- **Adversarial Examples:** Samples that an attacker has designed to cause the neural network to make a mistake. E.g. add designed noise.



1.34 What is Adversarial Training?

Adversarial training trains networks with adversarial examples on-the-fly to optimize the following framework:

$$\arg\min_{\theta} \mathbb{E}_{(x,y)\sim\mathbb{D}}\Big[\max_{\epsilon\in\mathbb{S}} L(\theta, x+\epsilon, y)\Big], \tag{1.19}$$

where $\mathbb{D}$ is the underlying data distribution, $L(\cdot,\cdot,\cdot)$ is the loss function, θ is the network parameter, x is a training sample with the ground-truth label y, ϵ is the added adversarial perturbation, and $\mathbb{S}$ is the allowed perturbation range. As shown in Equation (1.19), adversarial training consists of two computation steps: an **inner maximization step**, which computes adversarial examples, and an **outer minimization step**, which computes parameter updates.