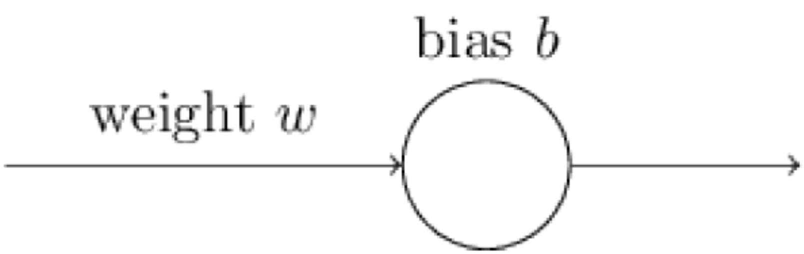
NEURAL NETWORK

Learning

- In general, learning refers to “learning from the errors”
- We kind of expect to the learning to process to be faster when the error margin is high and low when the error margin is low.
- How is it related to how humans learn?
- Does this make sense in NN context?

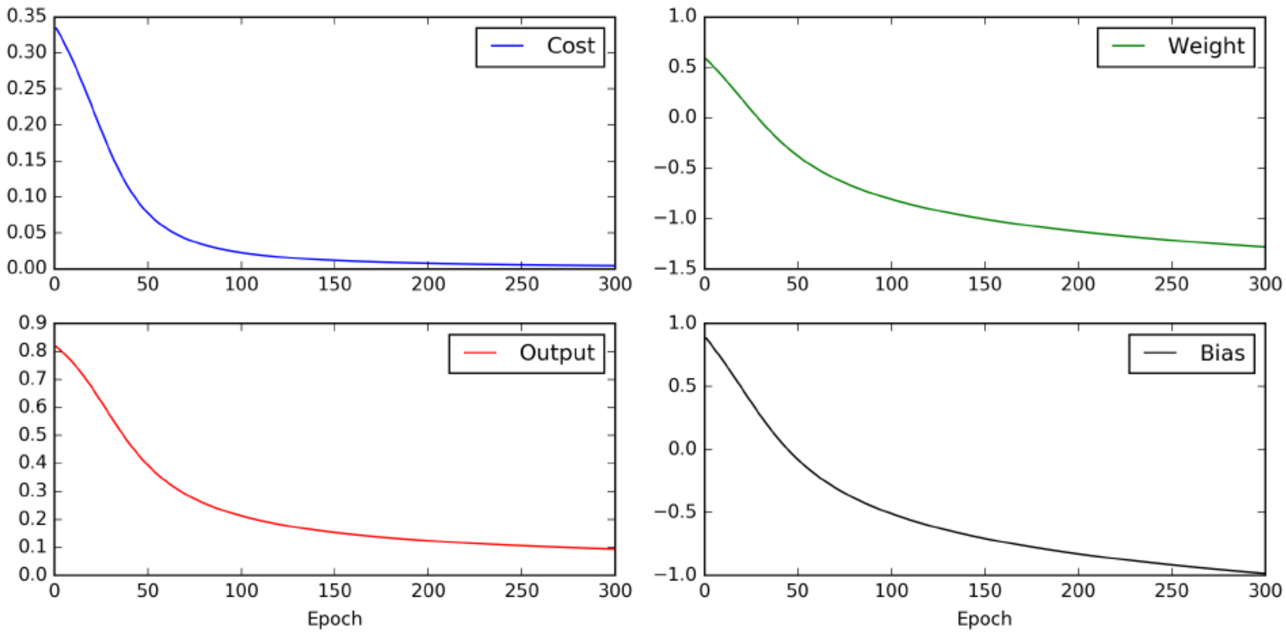
Learning

- Consider a very simple neuron, which ideally should take an input of 1 and give an output of 0.
- Say w=0.6 and b=0.9, and using sigmoid as activation the initial output = 0.82. So, the error margin is high.
- Assume a small learning rate = 0.15 (note in real implementation should be much smaller), what would be the convergence behavior of the output?



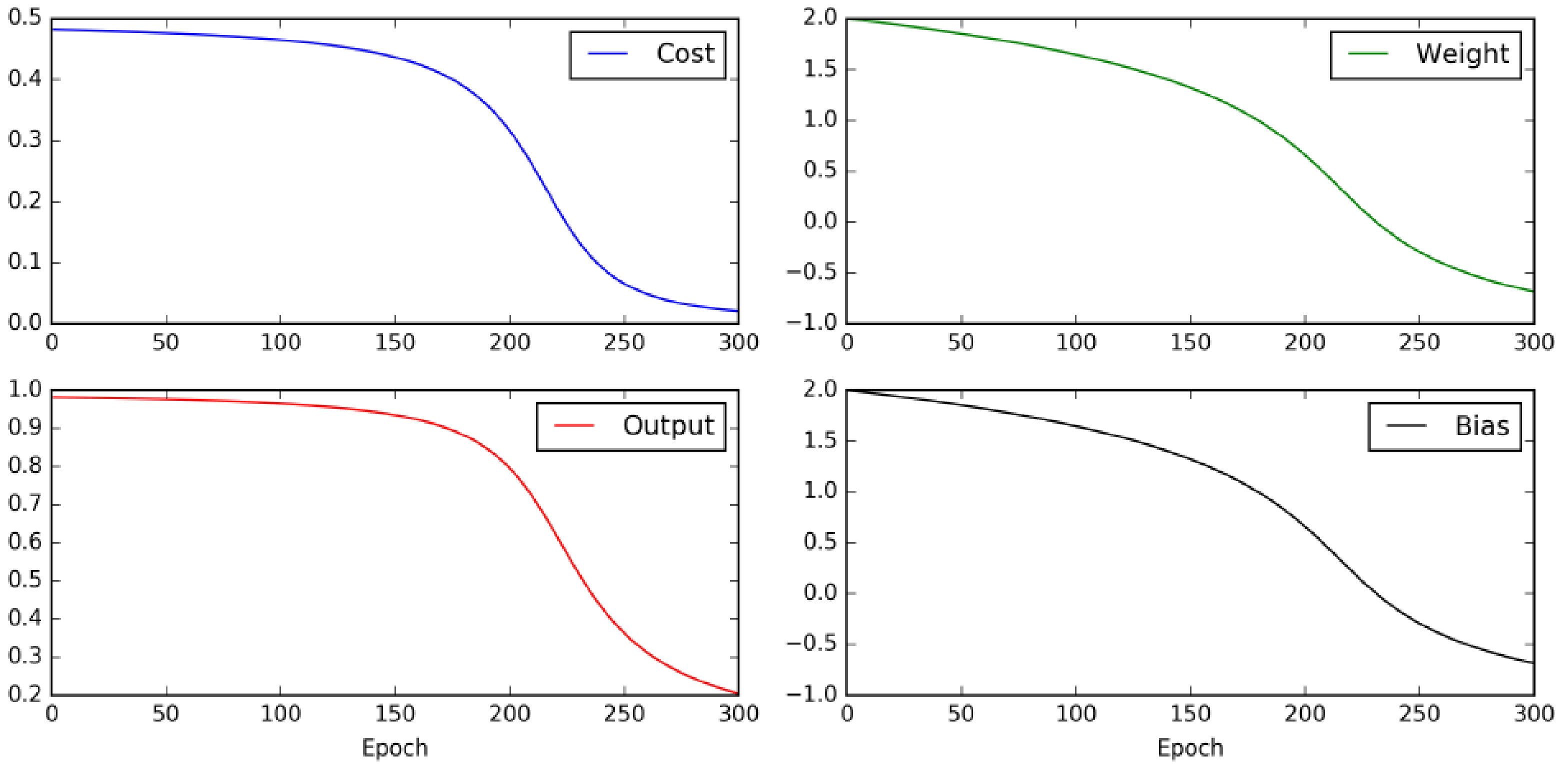
Learning

- This is what we observe



Learning

- What if w=2 and b=2, and so the output = 0.98. Now, the error margin is even higher.



Learning

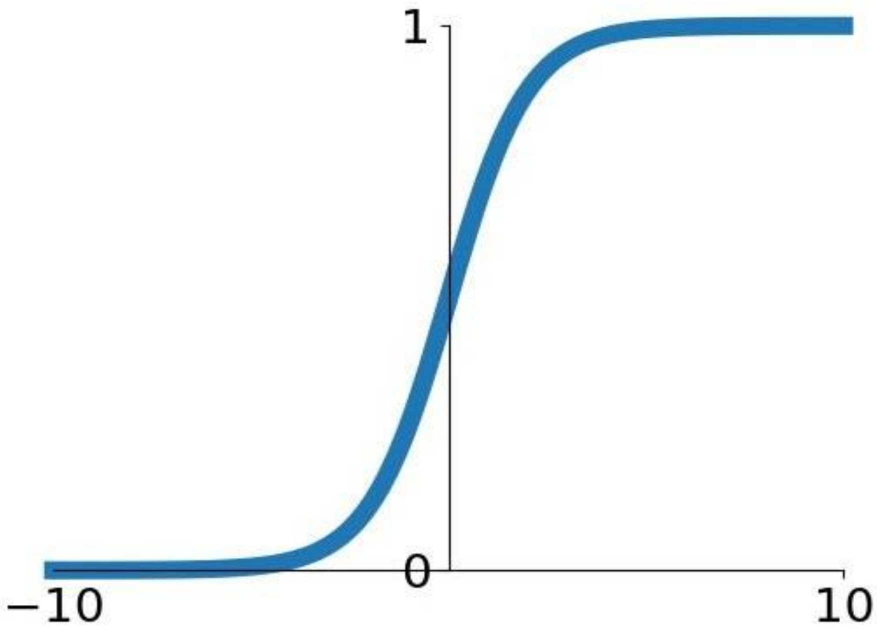
- Learning slow = Partial derivatives are small

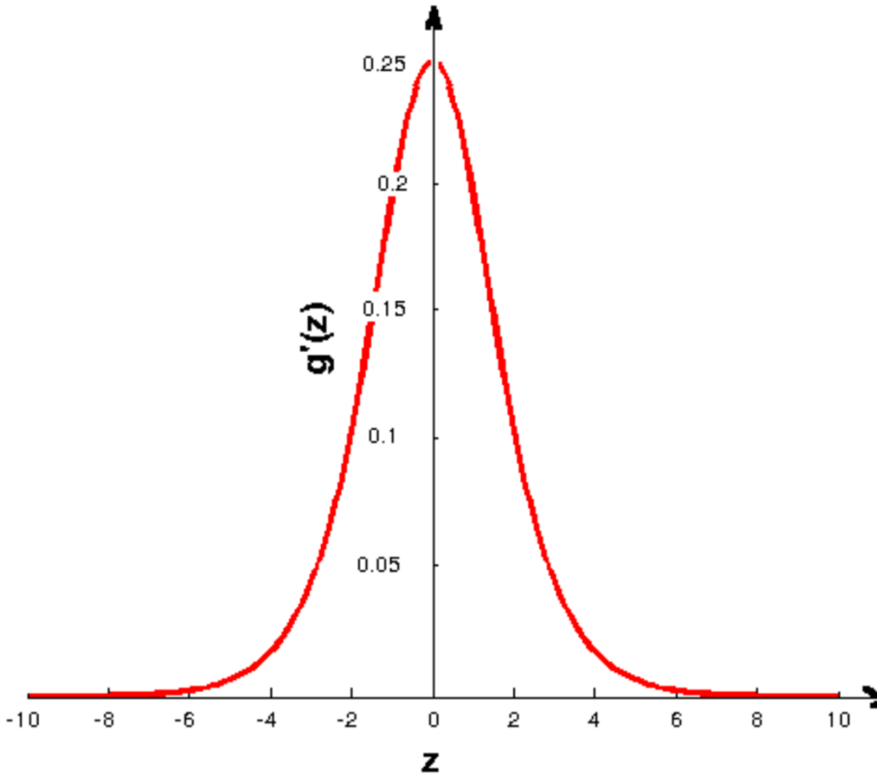
$$C = \frac{(y-a)^2}{2}$$

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z)$$

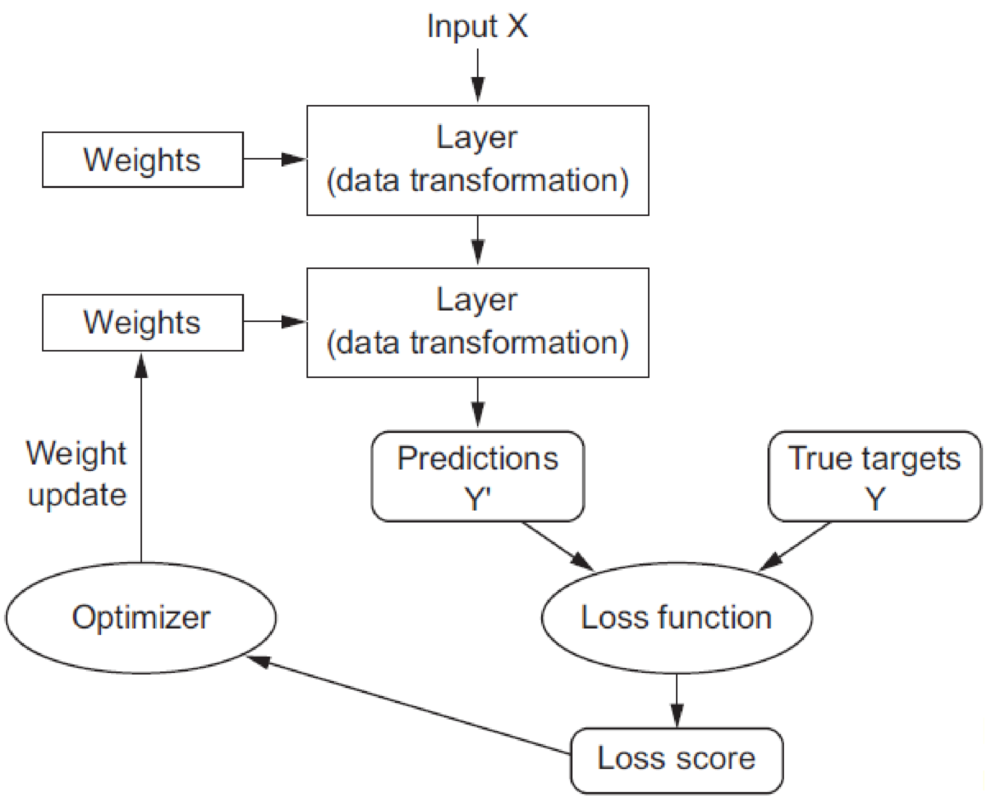
$$\frac{\partial C}{\partial b} = (a-y)\sigma'(z) = a\sigma'(z)$$

Sigmoid





Cost function



Cross-entropy cost function

- Lets design a new cost function

$$C = \frac{1}{n}\sum_{i}[y \ln a + (1 - y)\ln(1 - a)]$$

Cost Function for Classification

- Logic behind designing a cost function for classification is different.
- we ask “what does it mean for a guess to be wrong?”
  - This time rule of thumb is if you cannot guess correctly then we are **completely and utterly** wrong! It’s all or none. Not in between.
  - Since you can’t be more wrong than absolutely wrong, the penalty in this case is **enormous**.
  - Alternatively if the we guessed correctly, our cost function should not add any cost for each time this happens.

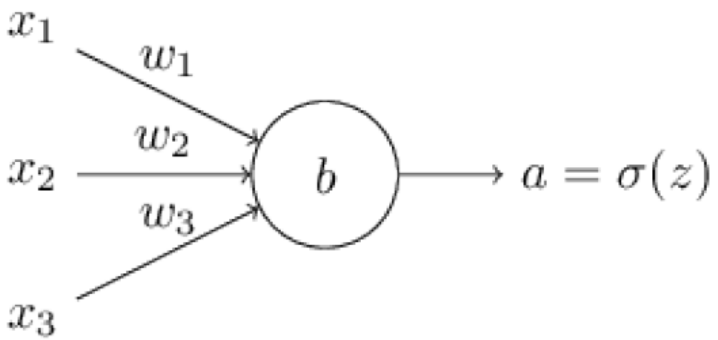
Cross-Entropy

- If our guess was right, but we weren’t completely confident
(y = 1, but a= 0.8), this should come with a small cost.
- if our guess was wrong but we weren’t completely confident
(y = 1, but a= 0.3), this should come with some significant cost, but not as much as if we were completely wrong.
- This is captured by the **log** function such that:

$$cost = \begin{cases} -\log(a) & if\ y = 0 \\ -\log(1-a) & if\ y = 1 \end{cases}$$

Cross-entropy cost function

- Think of a simple neuron



$$z = \sum_j w_j x_j + b \qquad a = \sigma(z)$$

Cross-entropy cost function

- If we take the derivative

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_i \left(\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)}\right)\frac{\partial \sigma}{\partial w_j}$$

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_i \left(\frac{\sigma(z)-y}{\sigma(z)(1-\sigma(z))}\right)\sigma'(z)x_j$$

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_i \left(\frac{\sigma'(z)}{\sigma(z)\left(1-\sigma(z)\right)}\right)(\sigma(z)-y)x_j$$

Cross-entropy cost function

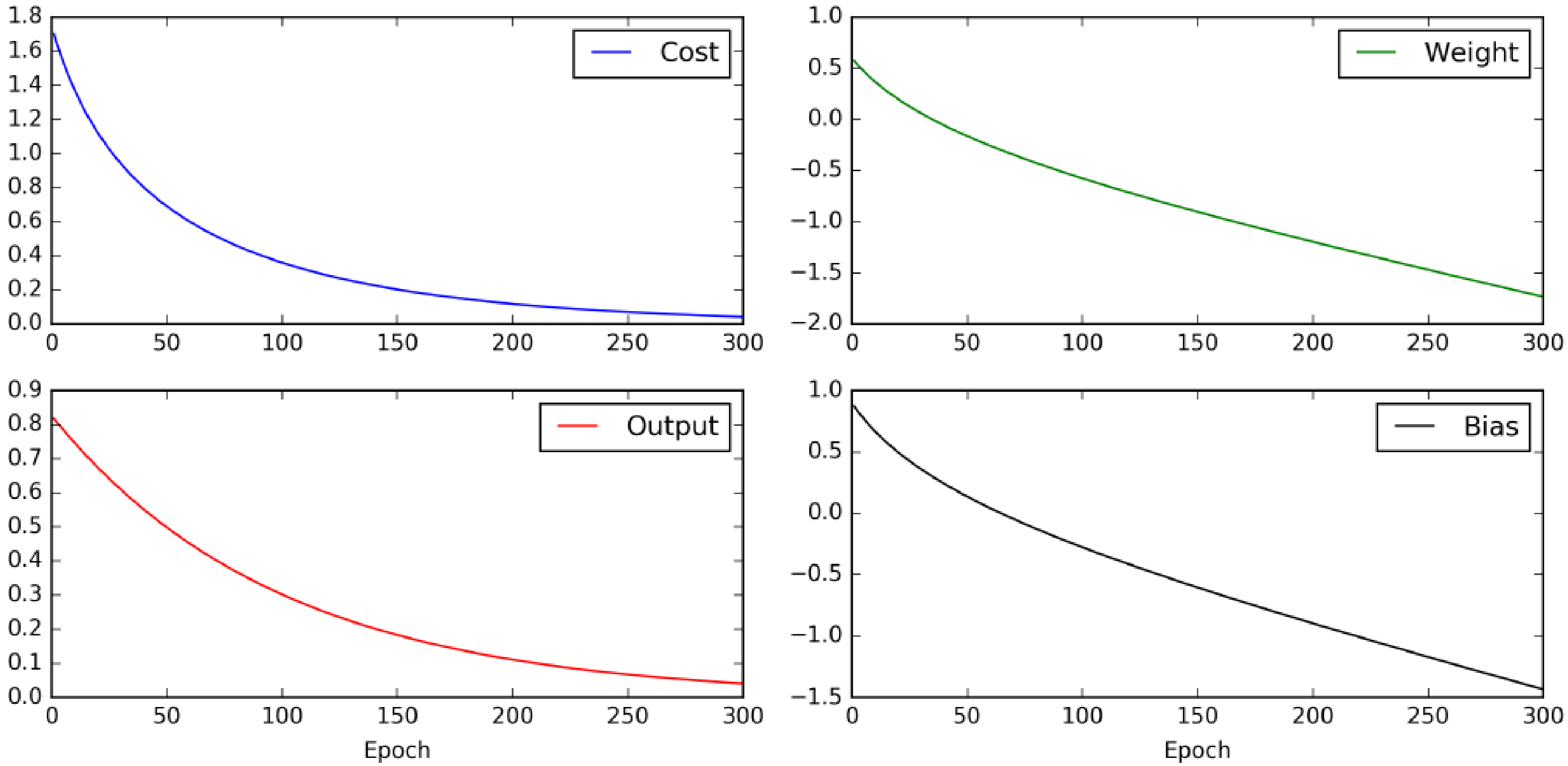
- As we have

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

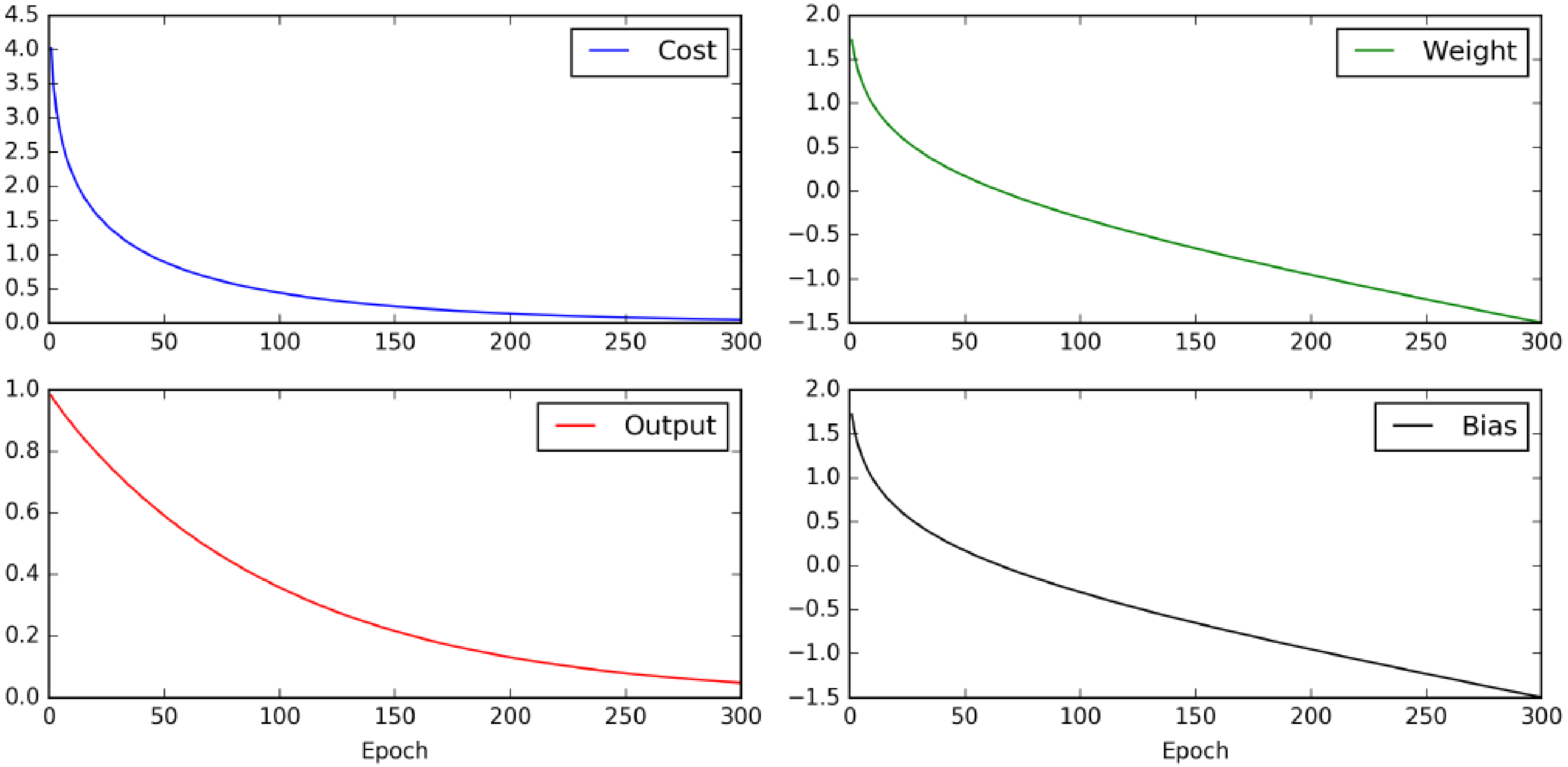
$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_i \left(\frac{\sigma'(z)}{\sigma(z)\big(1 - \sigma(z)\big)}\right)(\sigma(z) - y)x_j$$

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_i (\sigma(z) - y)x_j \qquad \frac{\partial C}{\partial b} = \frac{1}{n}\sum_i (\sigma(z) - y)$$

Cross-entropy cost function



Cross-entropy cost function



Cross-entropy cost function

- Generalization for the output layer

$$C_i = \sum_j \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right]$$

Cost functions

Problem type	Loss function
Binary classification	`binary_crossentropy`
Multiclass, single-label classification	`categorical_crossentropy`
Multiclass, multilabel classification	`binary_crossentropy`
Regression to arbitrary values	`mse`
Regression to values between 0 and 1	`mse` or `binary_crossentropy`

Last Layer Activation Function

- Sigmoid

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

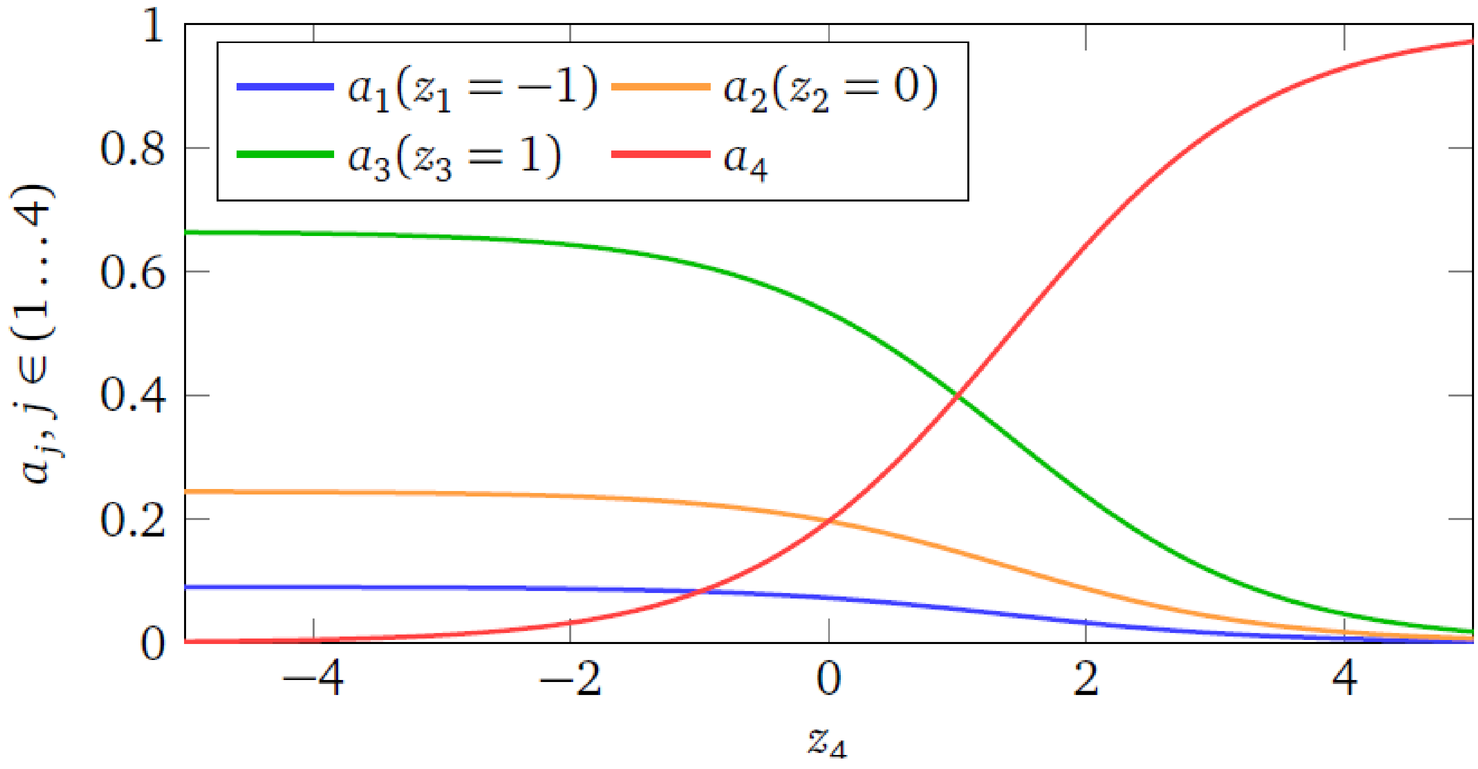
- Softmax

$$\mathrm{softmax}(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for j = 1,...,K}$$

Last Layer Activation Function

Raw output values	[3.2, -5.7, 0.6]
Applying **sigmoid** to raw output values	sigmoid calculation for the first raw output value: $\sigma(3.2)=\frac{e^{3.2}}{1+e^{3.2}}=0.96$ result of sigmoid calculation for all three output values: [0.96, 0.0033, 0.65] Sum: $0.96+0.0033+0.65=1.61\neq 1$
Applying **softmax** to raw output values	softmax calculation for the first raw output value: $\text{softmax}(3.2)=\frac{e^{3.2}}{e^{3.2}+e^{-5.7}+e^{0.6}}=0.93$ result of softmax calculation for all three output values: [0.93, 0.00013, 0.069] Sum: $0.93+0.069+0.00013=1$

Last Layer Activation Function



Cost and Last Layer Activation Functions

Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	None	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`

Log-likelihood

- Cross-Entropy

$$C_{\mathrm{CE}} = -\frac{1}{n}\sum_{x}\sum_{k=1}^{K}(y_k \ln a_k^L + (1-y_k)\ln(1-a_k^L))$$

- Log-likelihood

$$C_{\mathrm{LL}} = -\frac{1}{n}\sum_{x} y^T \ln(a^L) = -\frac{1}{n}\sum_{x}\sum_{k=1}^{K} y_k \ln(a_k^L)$$

Log-likelihood

$$a^L = [0.55, 0.02, 0.01, 0.03, 0.01, 0.05, 0.17, 0.01, 0.06, 0.09], \text{ and}$$
$$y = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

$$C_{\text{CE}} = 1.0725$$
$$C_{\text{LL}} = 0.5978$$

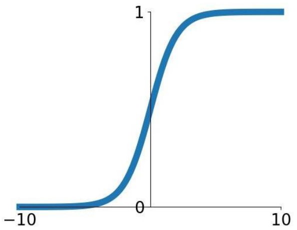
$$a^L = [0.55, 0.002, 0.001, 0.003, 0.001, 0.04, 0.37, 0.001, 0.012, 0.02]$$

$$C_{\text{CE}} = 1.1410$$
$$C_{\text{LL}} = 0.5978$$

Activation functions

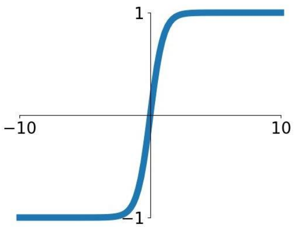
Sigmoid

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



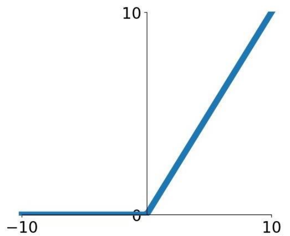
tanh

$$\tanh(x)$$



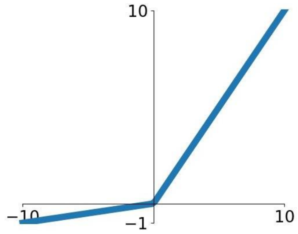
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

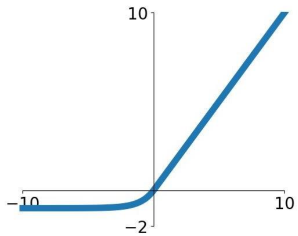


Maxout

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

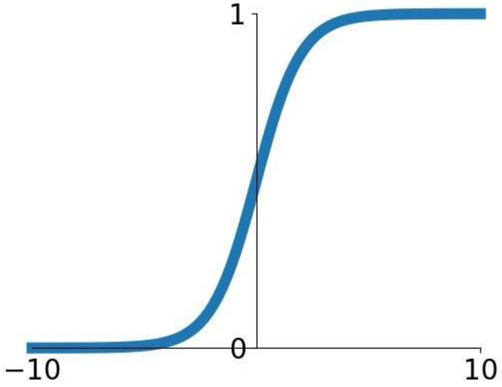
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activation Functions

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

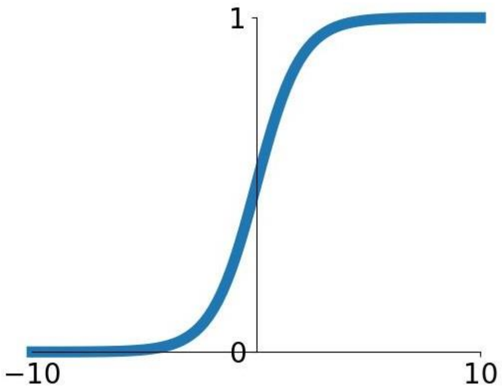


Sigmoid

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

Activation Functions

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



Sigmoid

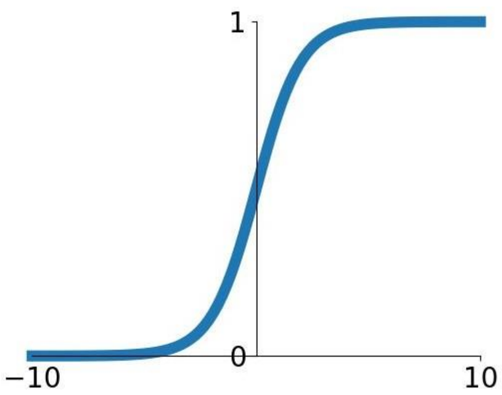
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

- 3 problems:

1. Saturated neurons “kill” the gradients

Activation Functions

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



Sigmoid

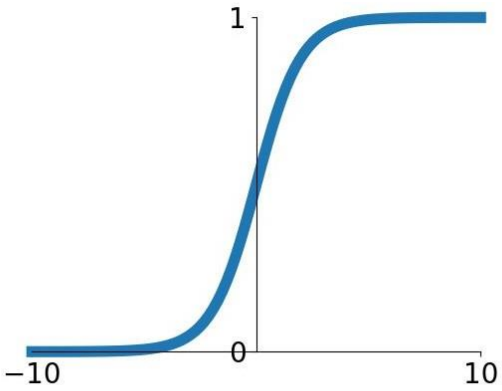
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

- 3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered

Activation Functions

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



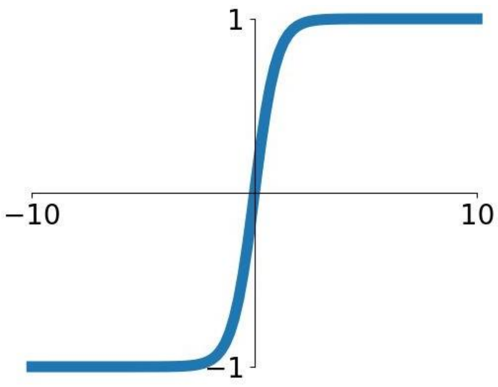
Sigmoid

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

- 3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

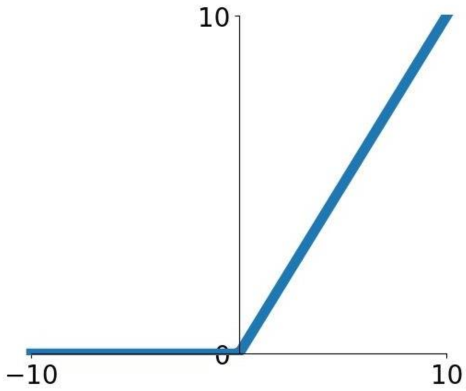
Activation Functions



tanh(x)

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

Activation Functions



ReLU
(Rectified Linear Unit)

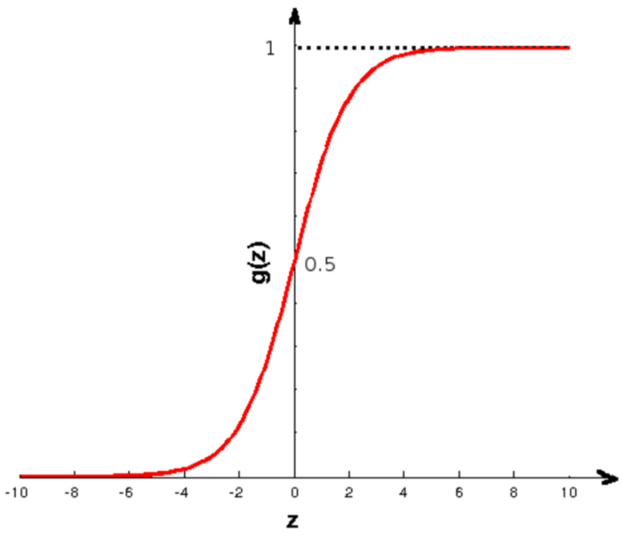
- Computes f(x) = max(0,x)
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- Dead neurons

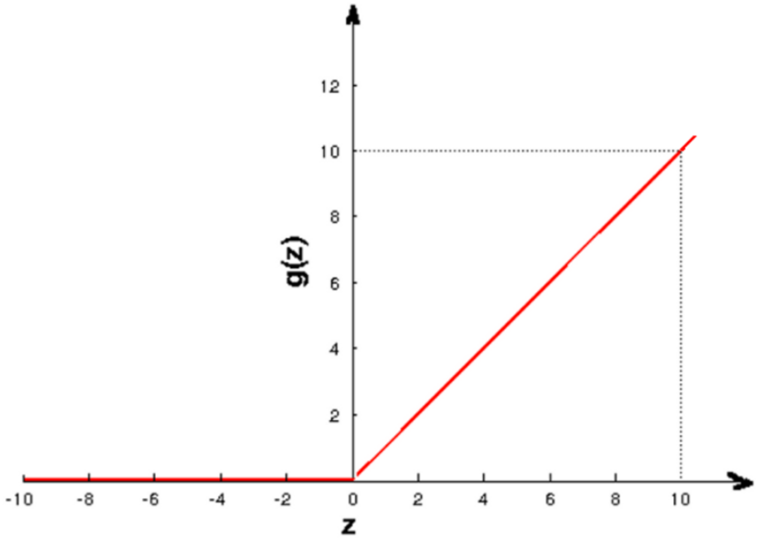
Rectified Linear Unit (ReLU)

- A better activation function:

$$g(z) = \max(0, z)$$



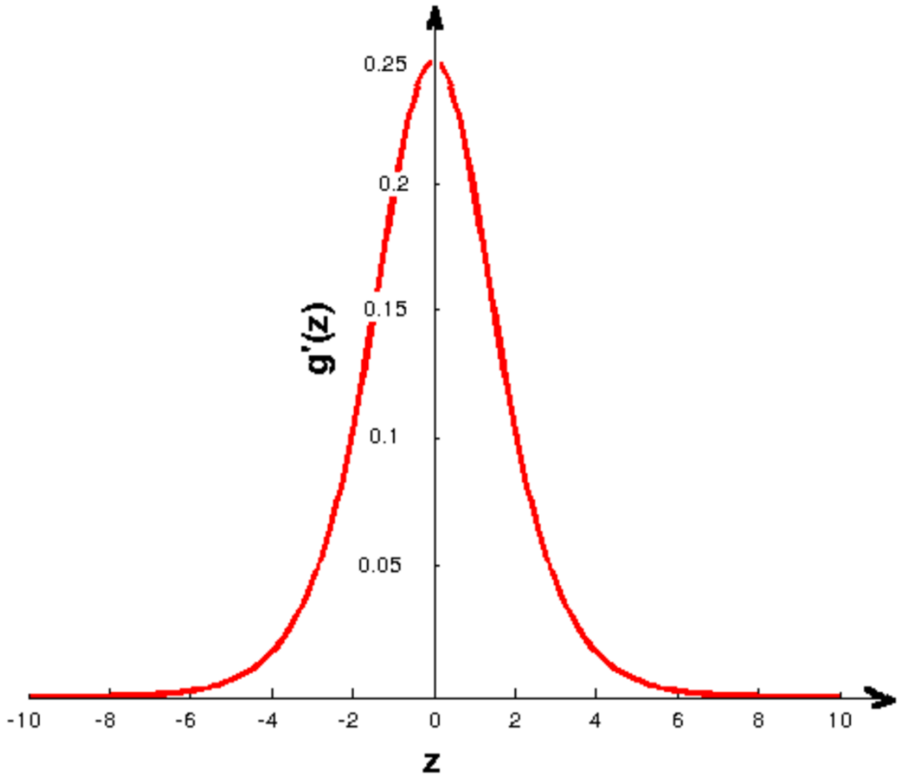
The sigmoid activation function



The rectified linear activation function

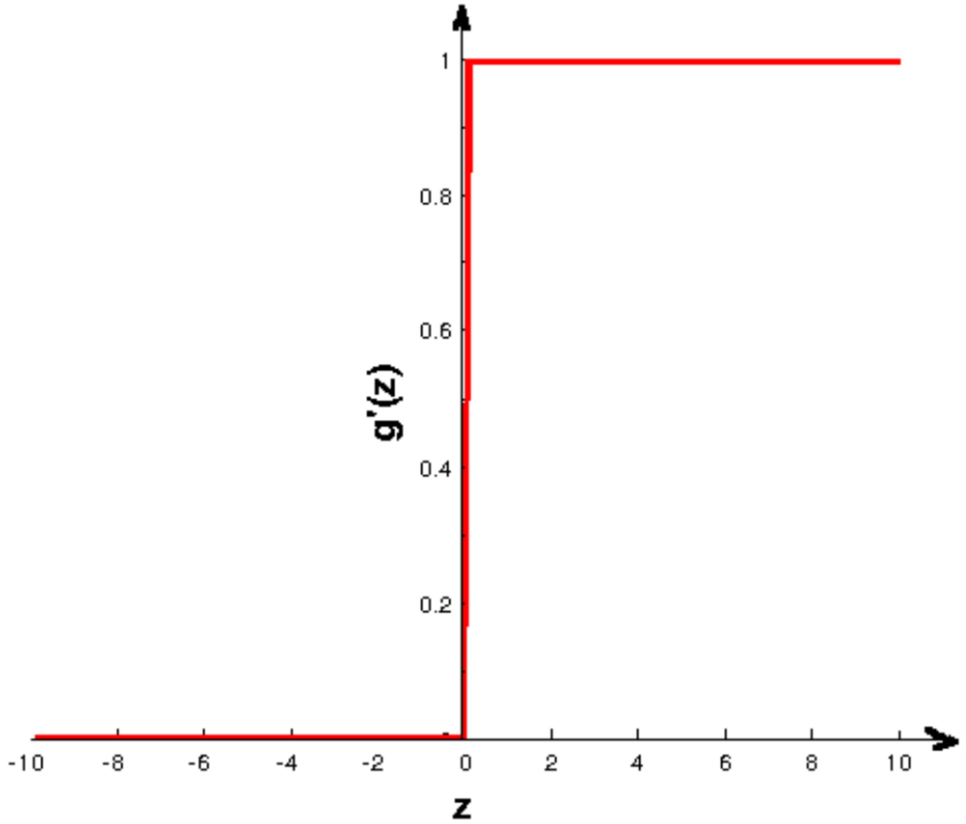
ReLU

- Why is ReLU better?
  Consider derivative of sigmoid.

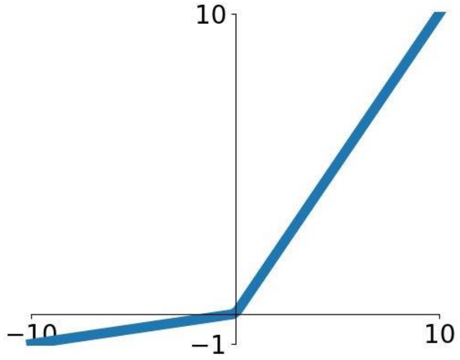


ReLU

- Derivative of ReLU



Activation Functions



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

Leaky ReLU

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

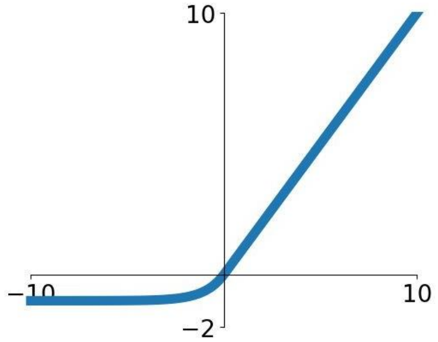
Parametric Rectifier (PReLU)

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

backprop into \alpha (parameter)

Activation Functions

Exponential Linear Units (ELU)



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

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

- Computation requires exp()

Maxout "Neuron"

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

Problem: doubles the number of parameters/neuron :(

In practice:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don’t expect much
- Don’t use sigmoid