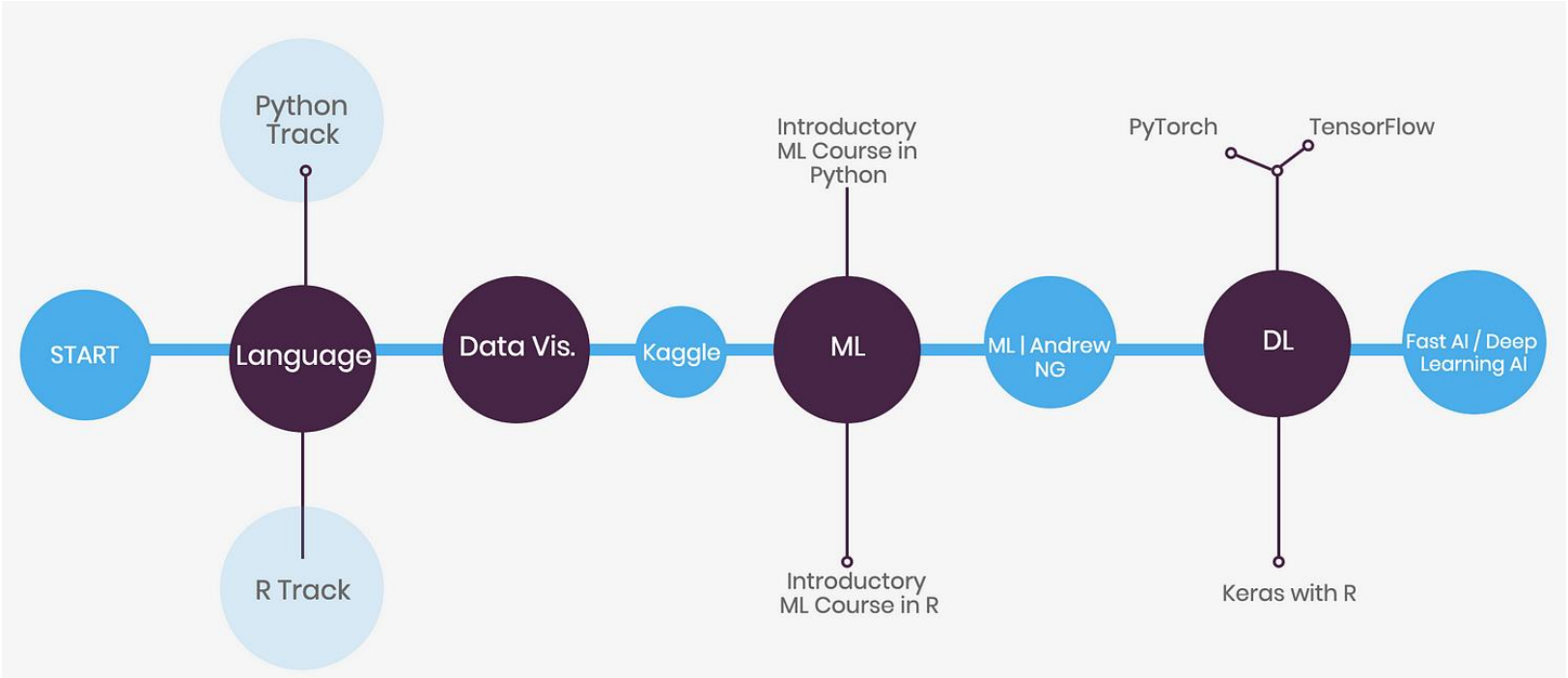
Library for Deep Learning

- PyTorch
  - Facebook's AI Research Lab (FAIR)
- TensorFlow
  - Google Brain Team



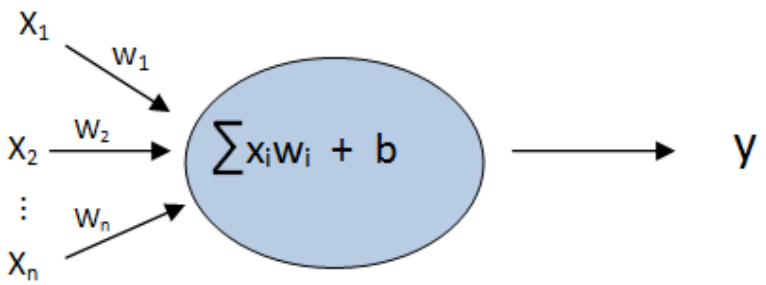
https://medium.com/ml-cheat-sheet/machine-learning-curriculum-from-rookie-to-mastery-636bdbf74856

Activation Functions

- determines if a neuron should be activated or not activated
  - use some simple mathematical operations to determine if the neuron's input to the network is relevant or not relevant in the prediction process
- introduce non-linearity to an artificial neural network
- generate output from a collection of input values fed to a layer

	Human brain	Neural networks
Input	Receive from outside world	Provide as images, sounds, numbers, etc.
Process	Neuron	Artificial neuron
Activate	Neuron tail	Algorithm

- universal approximation: a neural network can approximate any continuous function that maps inputs (X) to outputs (y)
  - The ability to represent any function is what makes the neural networks so powerful and widely-used
  - To be able to approximate any function, we need non-linearity: activation functions
  - Without activation functions, neural networks can be considered as a collection of linear models



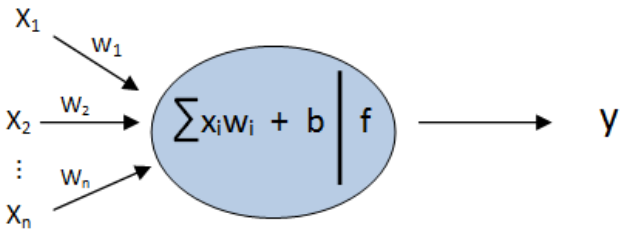
X = input

W = weight

b = bias term

y = output

$$y = \sum(weight * input) + bias$$



X = input

W = weight

b = bias term

y = output

f = activation function

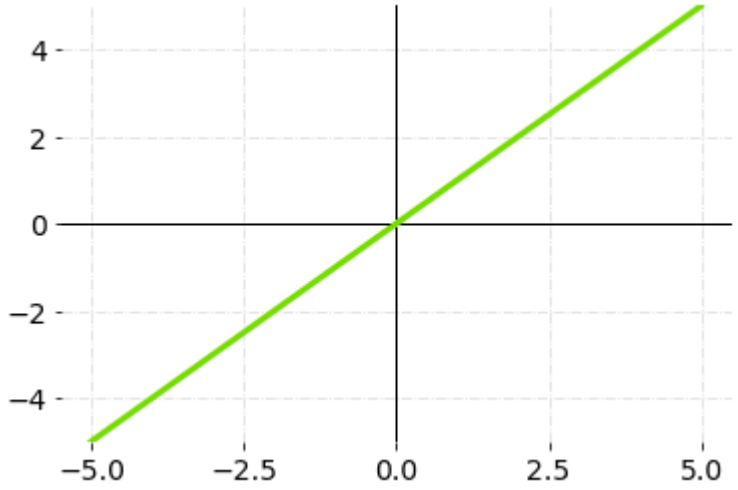
$$y = f(\sum(weight * input) + bias)$$

https://www.youtube.com/watch?v=joA6fEAbAQc

Activation Functions

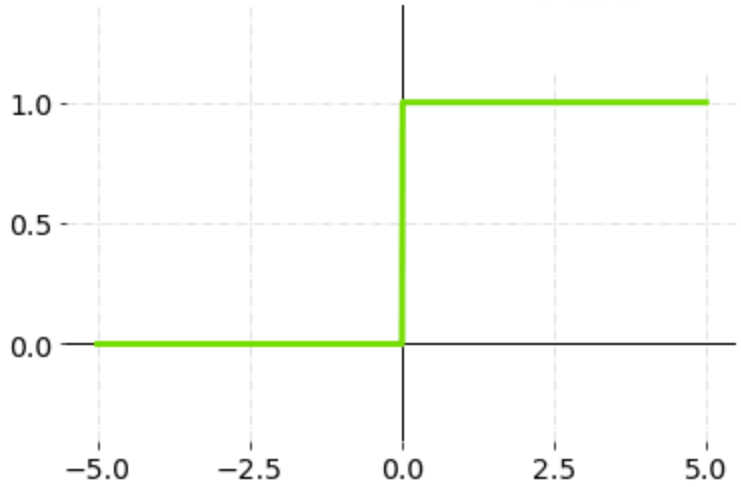
- Linear activation function
  - proportional to the input, (-∞, ∞)
  - simply adds up the weighted total of the inputs and returns the result
  - Pros and Cons
    - not a binary activation because it only delivers a range of activations
    - Constant derivative: unrelated to x

$$f(x) = x$$



- Binary step function
  - A **threshold value** determines whether a neuron should be activated or not activated
  - Pros and Cons
    - cannot provide multi-value outputs
    - gradient is zero

$$f(x) = \begin{cases} 0 & for\ x < 0 \\ 1 & for\ x \geqslant 0 \end{cases}$$



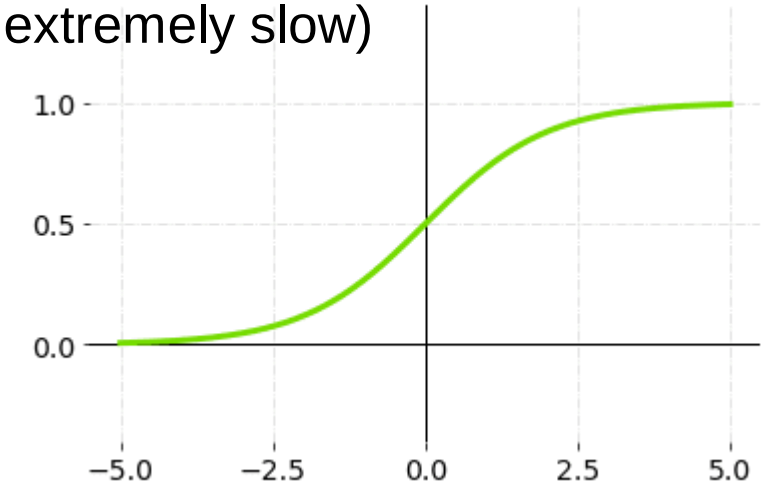
Non-linear Activation Functions: Sigmoid

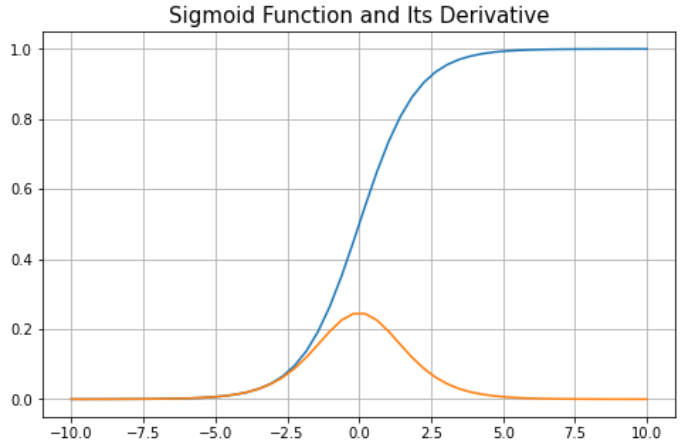
  - simple to use, all the desirable qualities: nonlinearity, continuous differentiation, monotonicity, and a set output range, mainly used in binary classification problems

- Pros and Cons
  - non-linear in nature, smooth gradient, good for a classifier type problem
  - output is always going to be in the range (0,1) → define a range for our activations
  - problem of “Vanishing gradients”
  - Its output isn’t zero centered → gradient updates go too far in different directions
  - output value is between zero and one → optimization harder (network either refuses to learn more or is extremely slow)

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

$$\frac{df}{dx} = \frac{e^{-x}}{\left(1+e^{-x}\right)^2} = \frac{1+e^{-x}-1}{1+e^{-x}}\left(\frac{1}{1+e^{-x}}\right) = (1-f)\,f$$

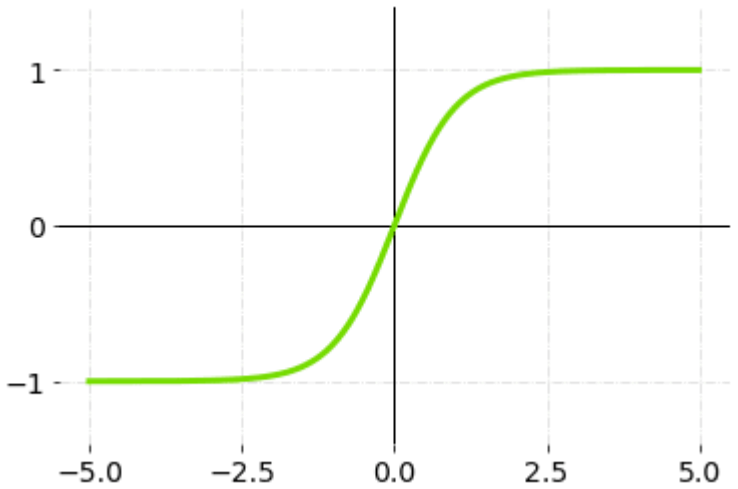




Non-linear Activation Functions: Tanh (Hyperbolic Tangent)

  - real-valued number to the range [-1, 1]
  - non-linear, its output is zero-centered (different from Sigmoid)
  - negative inputs will be mapped strongly to the negative and zero inputs will be mapped to almost zero in the graph of TanH
- Pros and Cons
  - vanishing gradient problem, but the gradient is stronger for TanH than sigmoid (derivatives are steeper)

$$f(x) = \frac{\left(e^{x} - e^{-x}\right)}{\left(e^{x} + e^{-x}\right)}$$

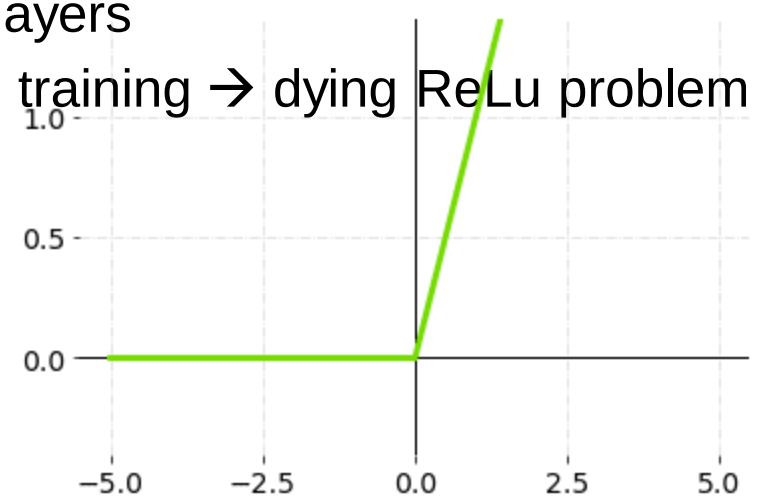


Non-linear Activation Functions: ReLU (Rectified Linear Unit)

- one of the most commonly used in the applications
- maximum value of the gradient is one → solve the problem of vanishing gradient:
- slope is never zero → solved the problem of saturating neuron

- Pros and Cons
  - only a certain number of neurons are activated → far more computationally efficient
  - accelerates the convergence of gradient descent towards the global minimum of the loss function due to its linear, non-saturating property
  - It should only be used within hidden layers
  - Some gradients can be fragile during training → dying ReLu problem

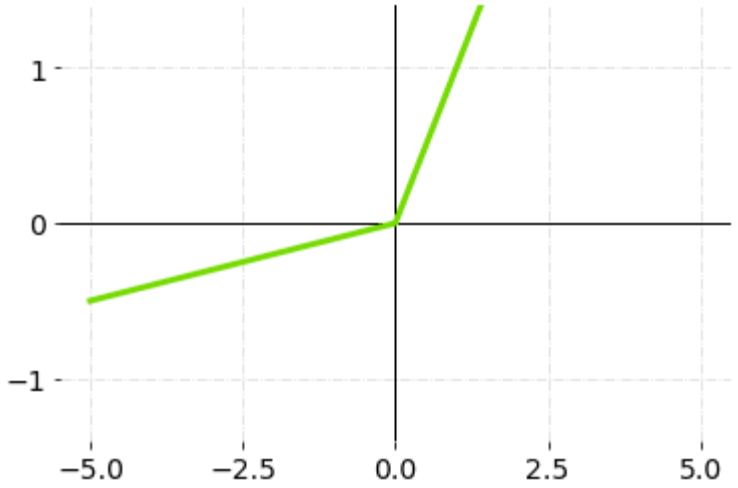
$$f(x) = max\,(0, x)$$



Non-linear Activation Functions: Leaky ReLU

  - upgraded version of the ReLU to solve the dying ReLU problem
  - consistency of the benefit across tasks is presently ambiguous
- Pros and Cons
  - ReLU + enable back propagation, even for negative input values
  - Making minor modification of negative input values, the gradient of the left side of the graph comes out to be a real (non-zero) value. As a result, there would be no more dead neurons in that area
  - predictions may not be steady for negative input values

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

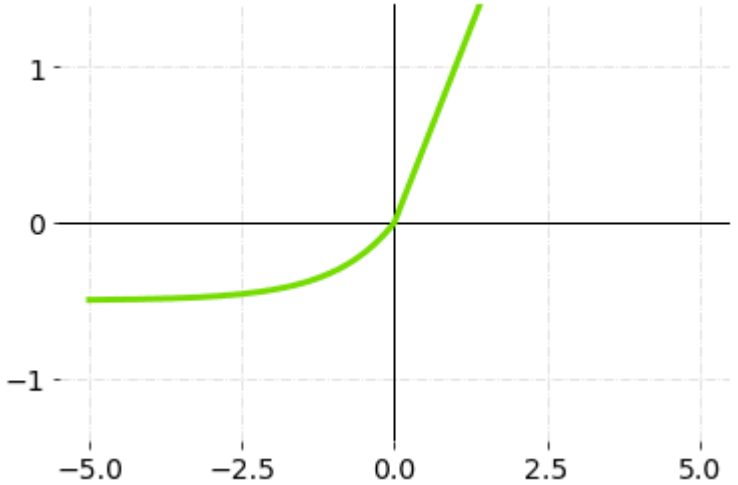


Non-linear Activation Functions: ELU (Exponential Linear Units)

  - one of the variations of ReLU which also solves the dead ReLU problem
  - very similar to ReLU except negative inputs, slightly more computationally expensive than leaky ReLU

- Pros and Cons
  - strong alternative to ReLU: can produce negative outputs
  - Exponential operations → increase the computational time
  - No learning about the ‘a’ value takes place, and exploding gradient problem

$$\begin{cases} x & for\ x \geqslant 0 \\ \alpha\left(e^{x}-1\right) & for\ x < 0 \end{cases}$$

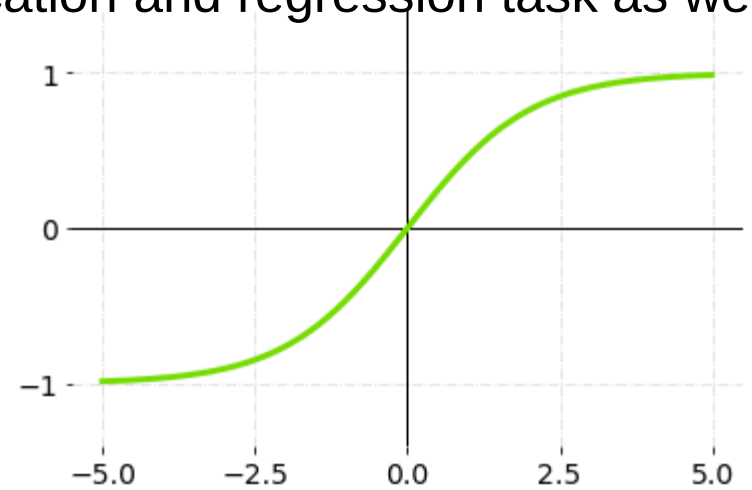


Non-linear Activation Functions: Softmax

- A combination of many sigmoids, determines relative probability, returns the probability of each class/labels, most commonly used in multi-class classification
- gives the probability of the current class with respect to others → consider the possibility of other classes

- Pros and Cons
  - It mimics the one encoded label better than the absolute values
  - We would lose information if we used absolute (modulus) values, but the exponential takes care of this on its own
  - should be used for multi-label classification and regression task as well

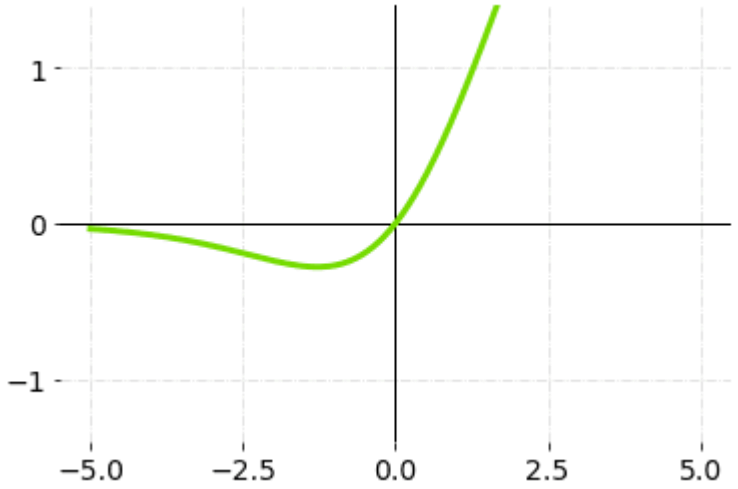
$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$



Non-linear Activation Functions: Swish

  - allows for the propagation of a few numbers of negative weights
  - crucial property that determines the success of non-monotonic smooth functions
  - self-gated activation function created by Google researchers
- Pros and Cons
  - smooth activation function: it does not suddenly change direction like ReLU does near x equal to zero
  - large negative numbers are wiped out, resulting in a win-win situation
  - non-monotonous function enhances the term of input data and weight to be learnt
  - Slightly more computationally expensive

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



Important Considerations

- Vanishing gradient is a common problem encountered during neural network training
  - sigmoid activation function: small output range (0 to 1), huge change in the input → small modification in the output, derivative also becomes small
  - These are only used for shallow networks with only a few layers, For a multi-layer network, the gradient may become too small for expected training
- Exploding gradients are situations in which massive incorrect gradients build during training, resulting in huge updates to neural network model weights
  - An unstable network might form, and training cannot be completed
  - weights' values can potentially grow to the point where they overflow, resulting in loss in NaN values

Final Takeaways

- All hidden layers generally use the same activation functions. ReLU activation function should only be used in the hidden layer for better results.
- Sigmoid and TanH activation functions should not be utilized in hidden layers due to the vanishing gradient, since they make the model more susceptible to problems during training.
- Swish function is used in artificial neural networks having a depth more than 40 layers.
- Regression problems should use linear activation functions
- Binary classification problems should use the sigmoid activation function
- Multiclass classification problems should use the softmax activation function

- Neural network architecture and their usable activation functions,
  - Convolutional Neural Network (CNN): ReLU activation function
  - Recurrent Neural Network (RNN): TanH or sigmoid activation functions

Computing Gradients

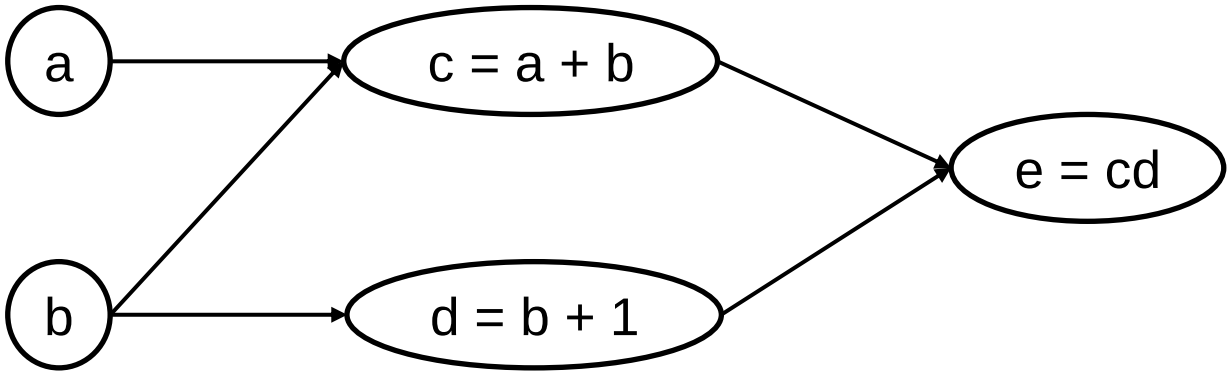
$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

$$F(x) = F_3\left(F_2\left(F_1(x)\right)\right) \rightarrow \frac{dF}{dx} = \frac{dF_3}{dx}\left(F_2\left(F_1(x)\right)\right)\frac{dF_2}{dx}\left(F_1(x)\right)\frac{dF_1}{dx}(x)$$

$$\text{derivatives} \begin{cases} \text{sum rule: } \frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1 \\ \text{product rule: } \frac{\partial}{\partial u}(uv) = \frac{\partial u}{\partial u}v + u\frac{\partial v}{\partial u} = v \end{cases}$$

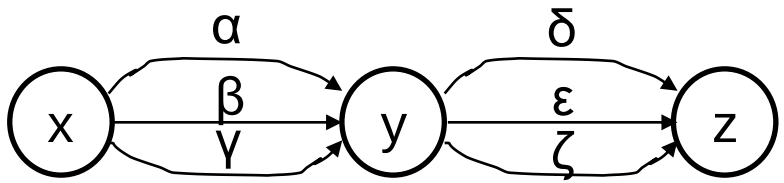
[computational graph]

$$e = (a+b)(b+1) \rightarrow \begin{cases} c = a+b \\ d = b+1 \\ e = cd \end{cases}$$



$$\begin{cases} \frac{\partial e}{\partial a} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial a} = \\ \frac{\partial e}{\partial b} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = \end{cases}$$

Multivariate Chain Rule



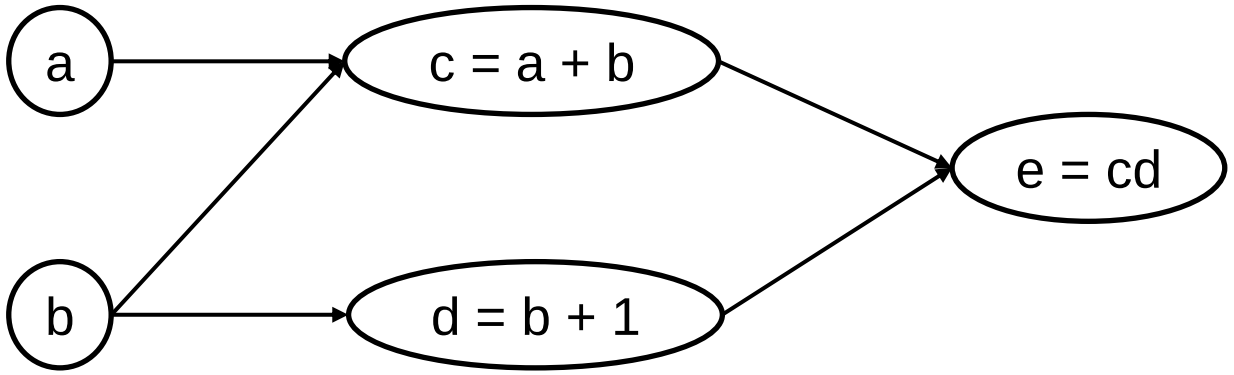
$$\frac{\partial x}{\partial x} = 1 \qquad \frac{\partial y}{\partial x} = (\alpha + \beta + \gamma) \qquad \frac{\partial z}{\partial x} = (\alpha + \beta + \gamma)(\delta + \varepsilon + \zeta) \xrightarrow[\text{forward mode}]{} \left(\frac{\partial ?}{\partial x}\right)$$

$$\frac{\partial z}{\partial x} = (\alpha + \beta + \gamma)(\delta + \varepsilon + \zeta) \qquad \frac{\partial z}{\partial y} = (\delta + \varepsilon + \zeta) \qquad \frac{\partial z}{\partial z} = 1 \xleftarrow[\text{reverse mode}]{} \left(\frac{\partial z}{\partial ?}\right)$$

differentiation $\begin{cases} \text{forward-mode: how one input affects every mode} \\ \text{backward-mode: how every mode affects one input} \end{cases}$

depends on how many $\begin{cases} \text{inputs} \\ \text{functions} \end{cases}$

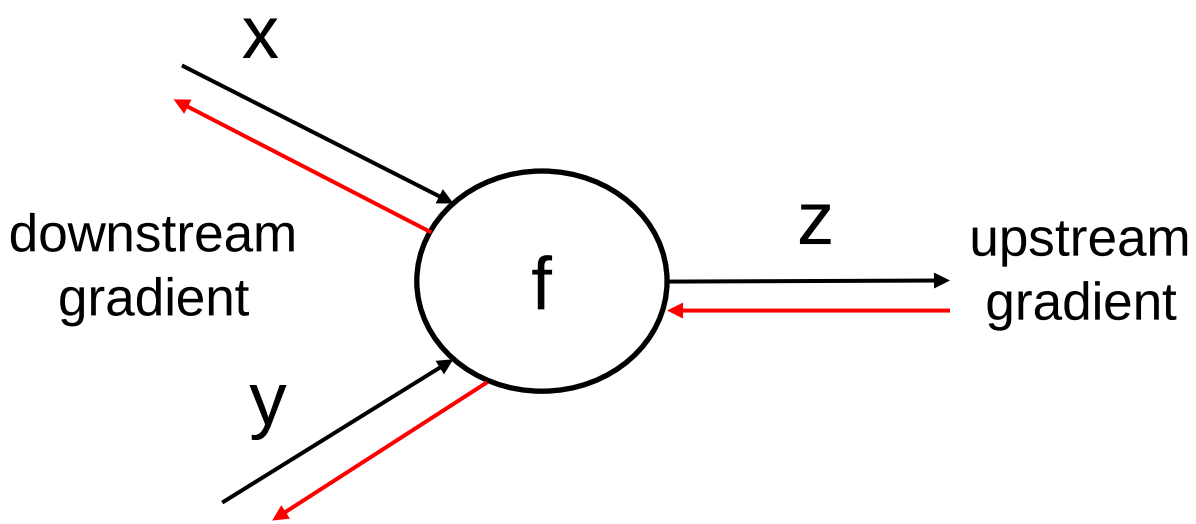
Deep Learning: one loss function with many inputs (weights) $\rightarrow$ backward!



$$\frac{\partial e}{\partial b} = ?$$

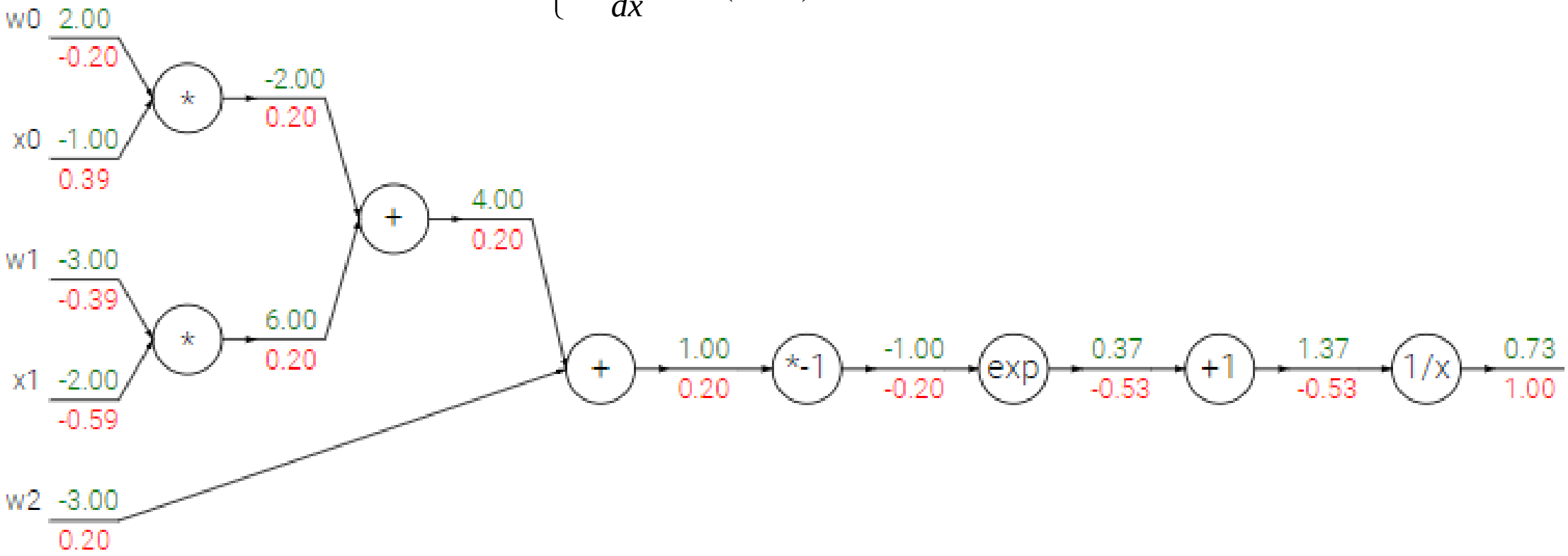
Patterns in Gradient Flow

- Add gate: gradient distributor
- Mul gate: swap multiplier
- Copy gate: gradient adder
- Max gate: gradient router



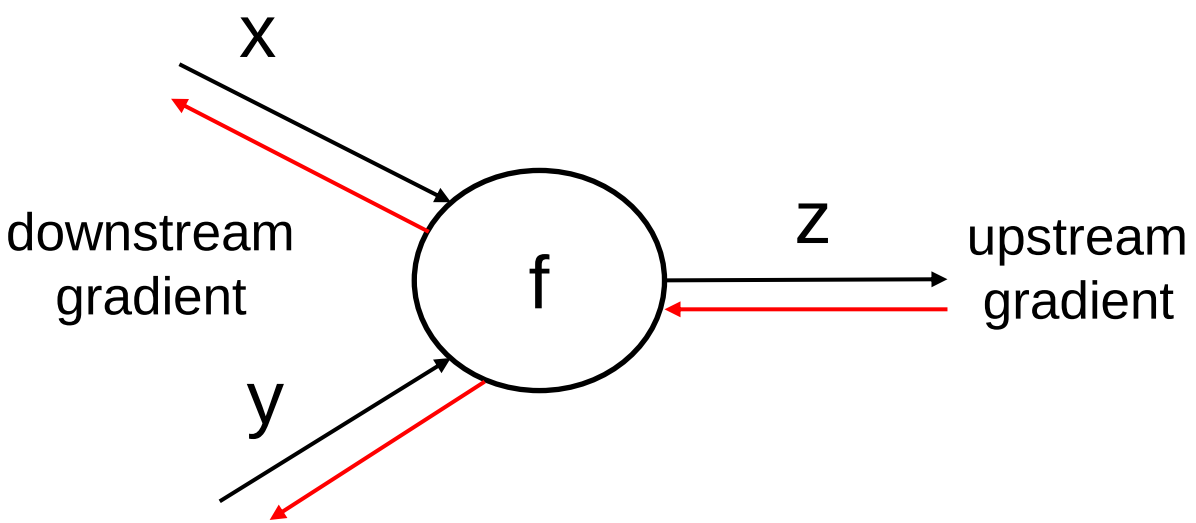
Example

$$\begin{cases} f(w,x) = \dfrac{1}{1+e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \\ \dfrac{df(w,x)}{dx} = (1-f)\,f \end{cases}$$

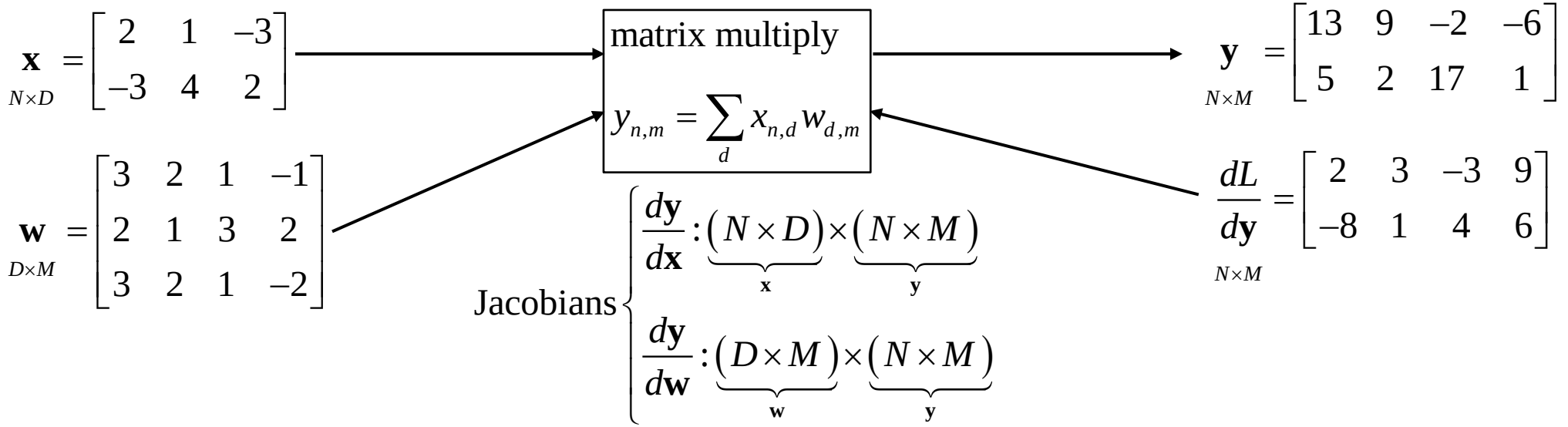


Vector Derivatives

$$
\begin{cases}
\text{Scalar to Scalar } \left(x \in \mathbb{R}, y \in \mathbb{R}\right) \rightarrow \dfrac{\partial y}{\partial x} \in \mathbb{R} \\
\text{Vector to Scalar } \left(\mathbf{x} \in \mathbb{R}^N, y \in \mathbb{R}\right) \rightarrow \dfrac{\partial y}{\partial \mathbf{x}} \in \mathbb{R}^N, \left(\dfrac{\partial y}{\partial \mathbf{x}}\right)_n = \dfrac{\partial y}{\partial x_n} : \text{ gradient} \\
\text{Vector to Vector } \left(\mathbf{x} \in \mathbb{R}^N, \mathbf{y} \in \mathbb{R}^M\right) \rightarrow \dfrac{\partial \mathbf{y}}{\partial \mathbf{x}} \in \mathbb{R}^{N \times M}, \left(\dfrac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)_{n,m} = \dfrac{\partial y_m}{\partial x_n} : \text{ Jacobian}
\end{cases}
$$



Back Propagation with Matrices



What part of $\mathbf{y}$ are affected by one element of $\mathbf{x}$? $x_{n,d}$ affects the whole row y_m

How much does $x_{n,d}$ affect $y_{n,m}$? $w_{d,m}$

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} w_{d,m} \rightarrow \begin{cases} \underset{N\times D}{\frac{\partial L}{\partial \mathbf{x}}} = \underset{N\times M}{\left(\frac{\partial L}{\partial \mathbf{y}}\right)} \underset{M\times D}{\mathbf{w}^T} \\ \underset{D\times M}{\frac{\partial L}{\partial \mathbf{w}}} = \underset{D\times N}{\mathbf{x}^T} \underbrace{\left(\frac{\partial L}{\partial \mathbf{y}}\right)}_{N\times M} \end{cases}$$

$$\underset{m\times n}{\mathbf{A}}\left(\underset{n\times p}{\mathbf{B}}\ \underset{p\times q}{\mathbf{C}}\right)=\left(\mathbf{AB}\right)\mathbf{C}$$

$$\mathbf{BC}: npq \rightarrow \underset{m\times n}{\mathbf{A}}\ \underset{n\times q}{\left(\mathbf{BC}\right)}: mnq \rightarrow \text{cost} = npq + mnq = nq\left(m+p\right)$$

$$\mathbf{AB}: mnp \rightarrow \underset{m\times p}{\left(\mathbf{AB}\right)}\ \underset{p\times q}{\mathbf{C}}: mpq \rightarrow \text{cost} = mnp + mpq = mp\left(n+q\right)$$

specialize to $\mathbf{C} = \text{column vector}\ \left(q=1\right): \mathbf{A}\left(\mathbf{BC}\right)$ vs. $\left(\mathbf{AB}\right)\mathbf{C}$

$$\underbrace{\text{vector}}_{n\left(m+p\right)} \qquad \underbrace{\text{big number}}_{mp\left(n+1\right)}$$