CSE 587: Deep Learning for Natural Language Processing

Lecture 3. Neural Networks and Backpropagation

Rui Zhang

Spring 2023

Many Slides from CS224n

Outline - Key Concepts

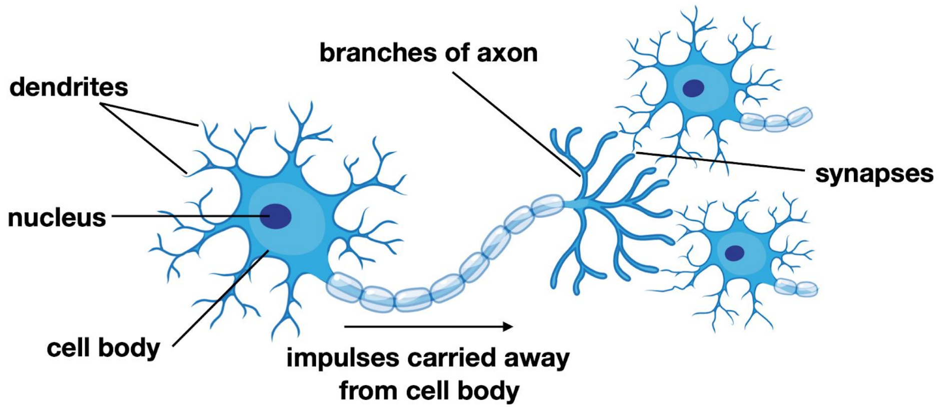
NLP

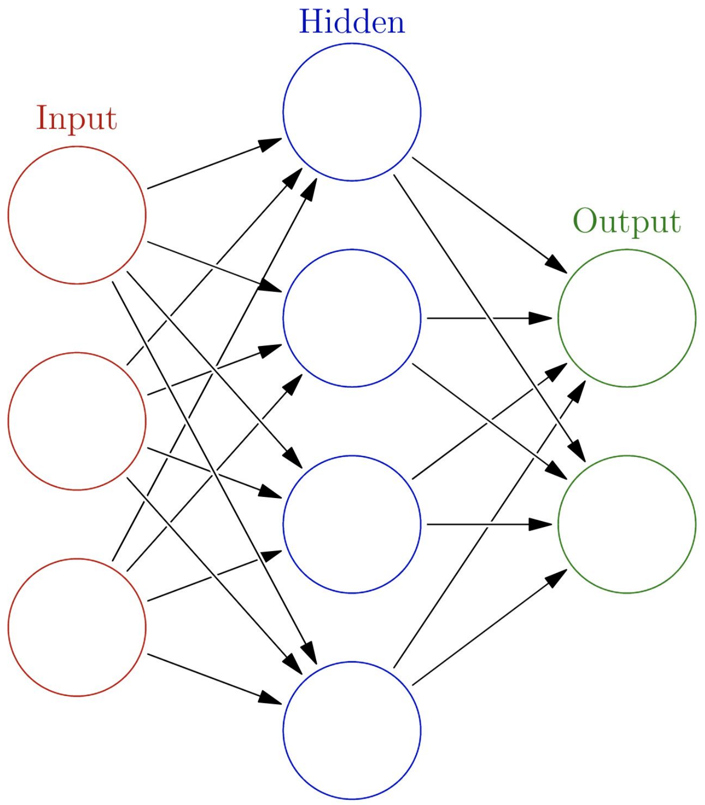
- N/A

ML

- Neural Networks
- Forward Propagation
- Gradient Descent
- Backpropagation
- Computational Graph
- Optimizers

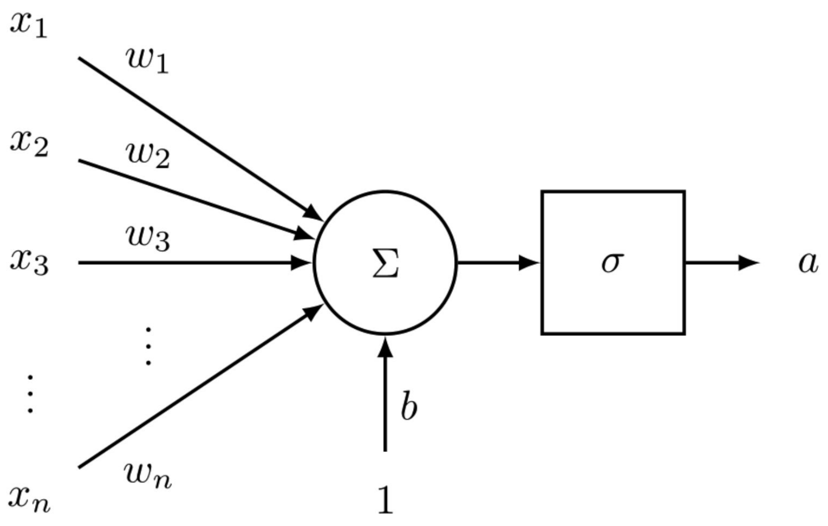
Neural Networks

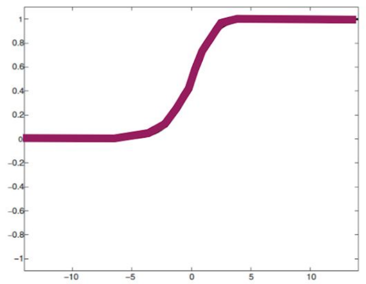




A Neuron

A neuron is the fundamental building block of neural networks.
e.g., a neuron with sigmoid function



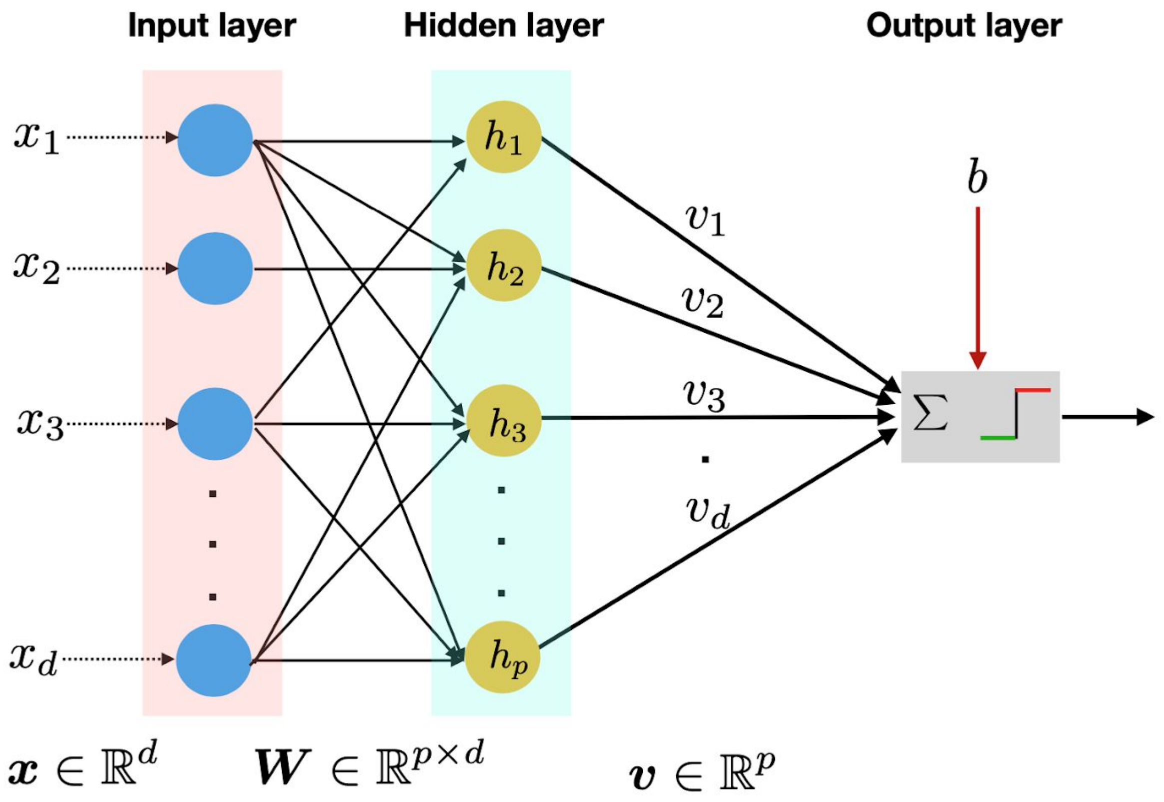


Sigmoid

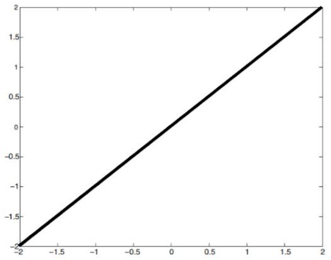
$$\sigma(z) = \frac{1}{1+\exp(-z)}$$

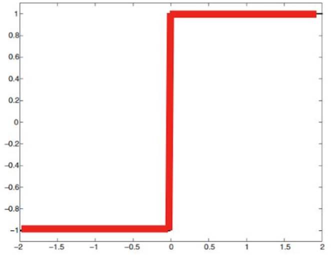
$$a = \frac{1}{1+\exp(-(w^T x + b))}$$

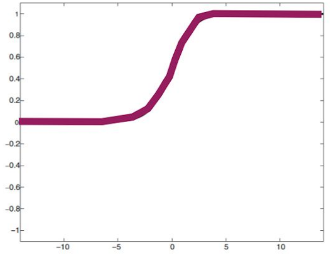
Two layer neural networks

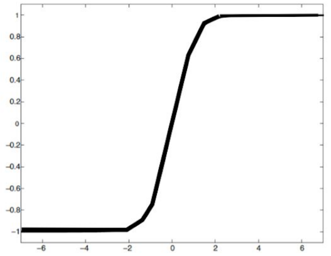

Input layer
Hidden layer
Output layer
x_1
x_2
x_3
x_d
h_1
h_2
h_3
h_p
v_1
v_2
v_3
v_d
b
Σ
$\boldsymbol{x} \in \mathbb{R}^d$
$\boldsymbol{W} \in \mathbb{R}^{p \times d}$
$\boldsymbol{v} \in \mathbb{R}^p$

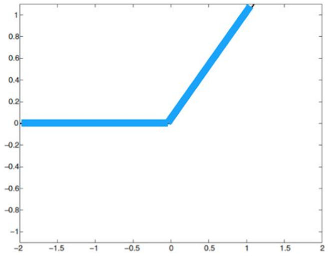
Activation Functions

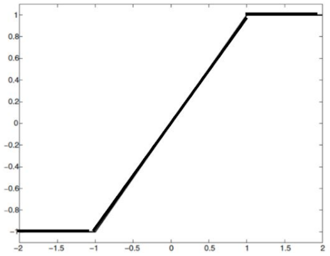

Identity

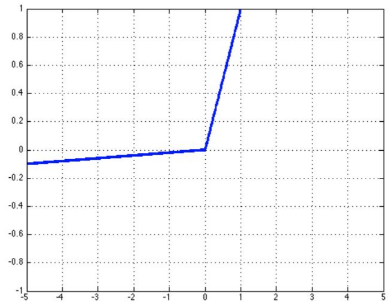

Sign


Sigmoid


Tanh


ReLU


Hard Tanh


Leaky ReLU

$$\text{leaky}(z) = \max(z, k \cdot z)$$

$$\text{where } 0 < k < 1$$

$$\sigma(z) = \text{sign}(z) \qquad \text{(Sign)}$$

$$\sigma(z) = \max\{0, z\} \qquad \text{(Rectified Linear Unit [ReLU])}$$

$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad \text{(Sigmoid)}$$

$$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

$$\text{hardtanh}(z) = \begin{cases} -1 & : z < -1 \\ z & : -1 \le z \le 1 \\ 1 & : z > 1 \end{cases}$$

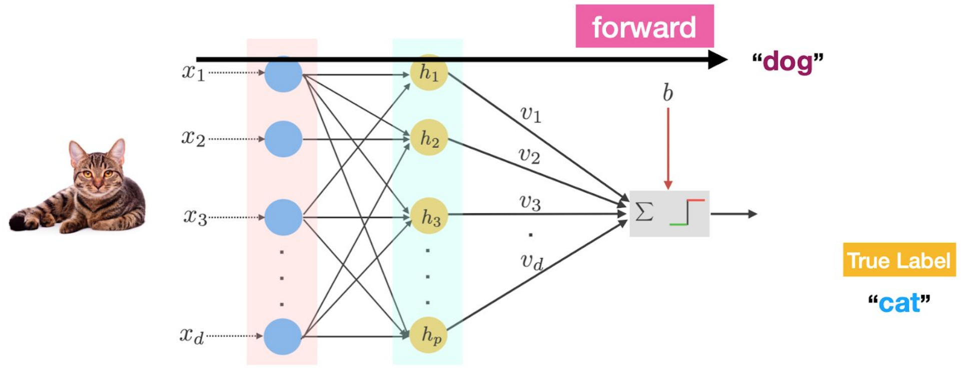
How to train two-layer Perceptron?

Forward propagation:

Backward propagation:

How to train two-layer Perceptron?

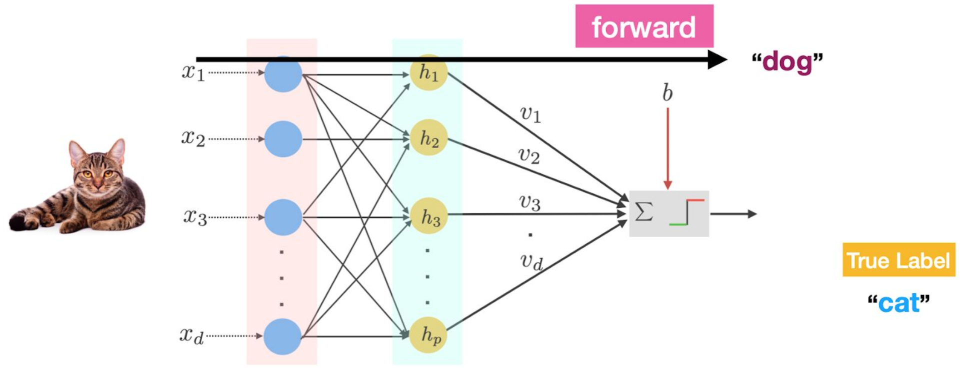
Forward propagation: In this phase, the inputs for a training instance are fed into the neural network. This results in a forward cascade of computations across the layers, using the current set of weights. The output is the prediction of current weights.

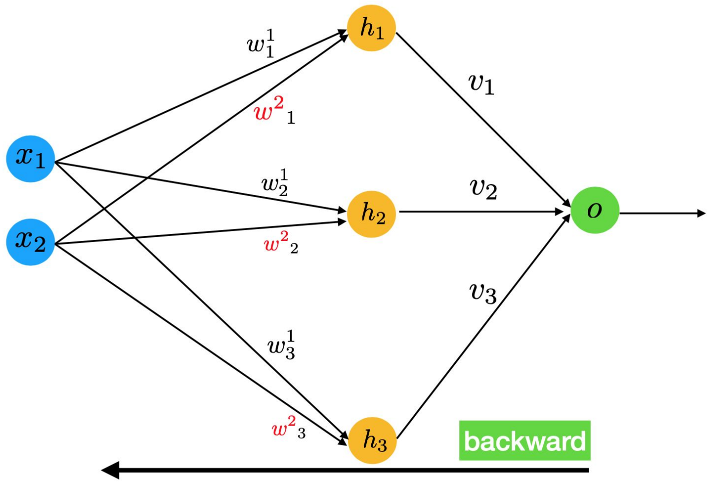


Backward propagation:

How to train two-layer Perceptron?

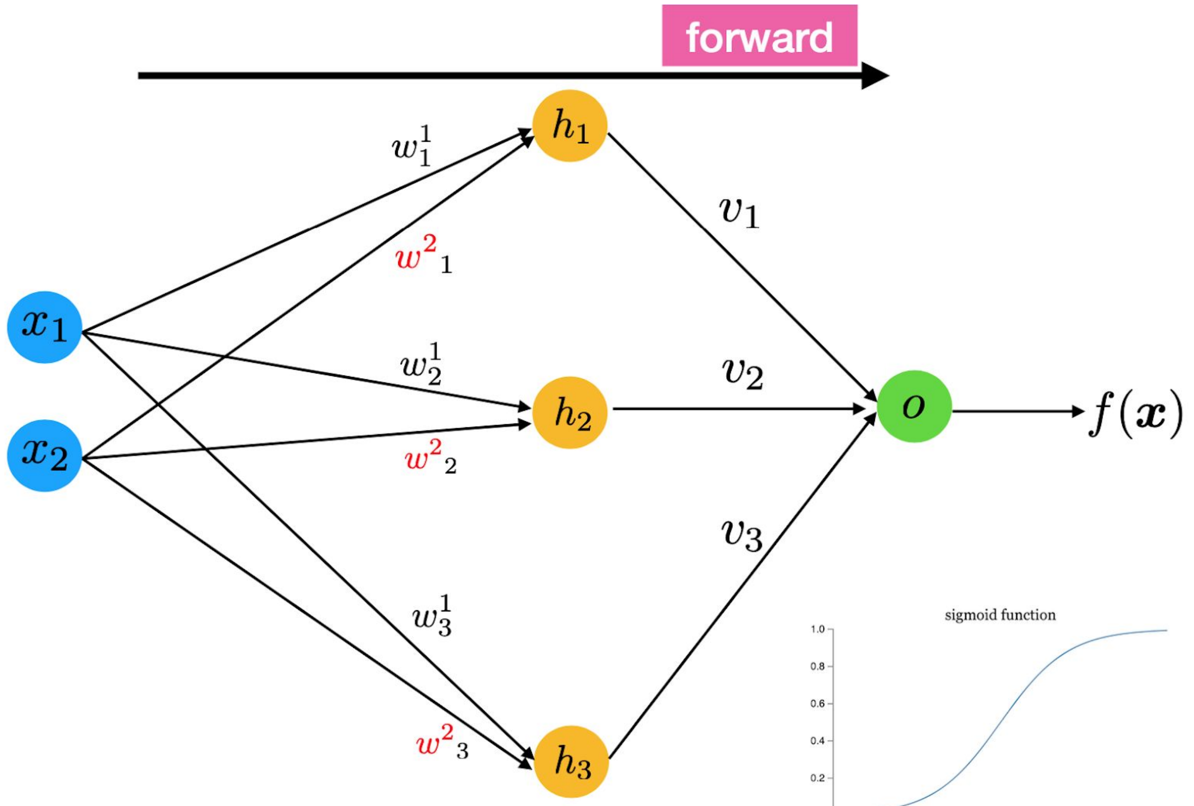
Forward propagation: In this phase, the inputs for a training instance are fed into the neural network. This results in a forward cascade of computations across the layers, using the current set of weights. The output is the prediction of current weights.

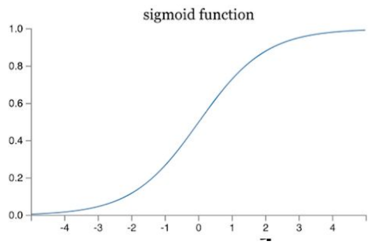




Backward propagation: The main goal of the backward phase is to **learn the gradient of the loss function** with respect to the different weights by **using the chain rule** of differential calculus. The weights are just adjusted based on error.

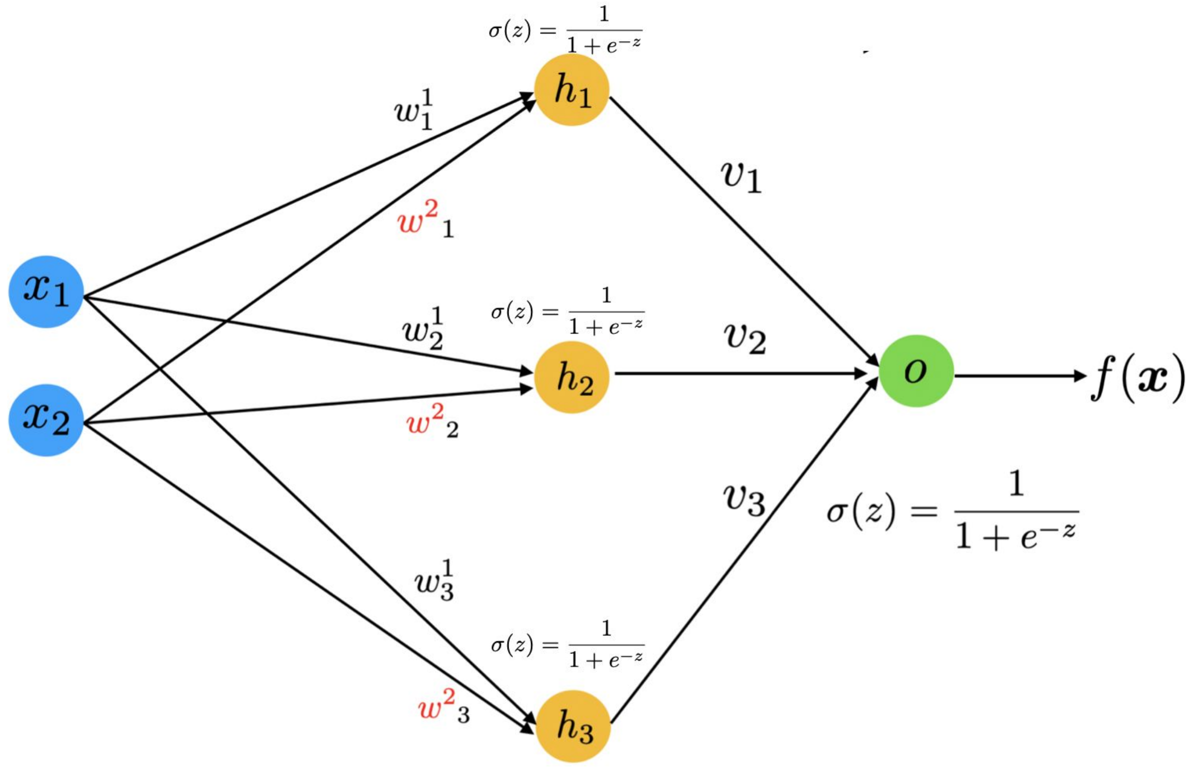
Forward propagation





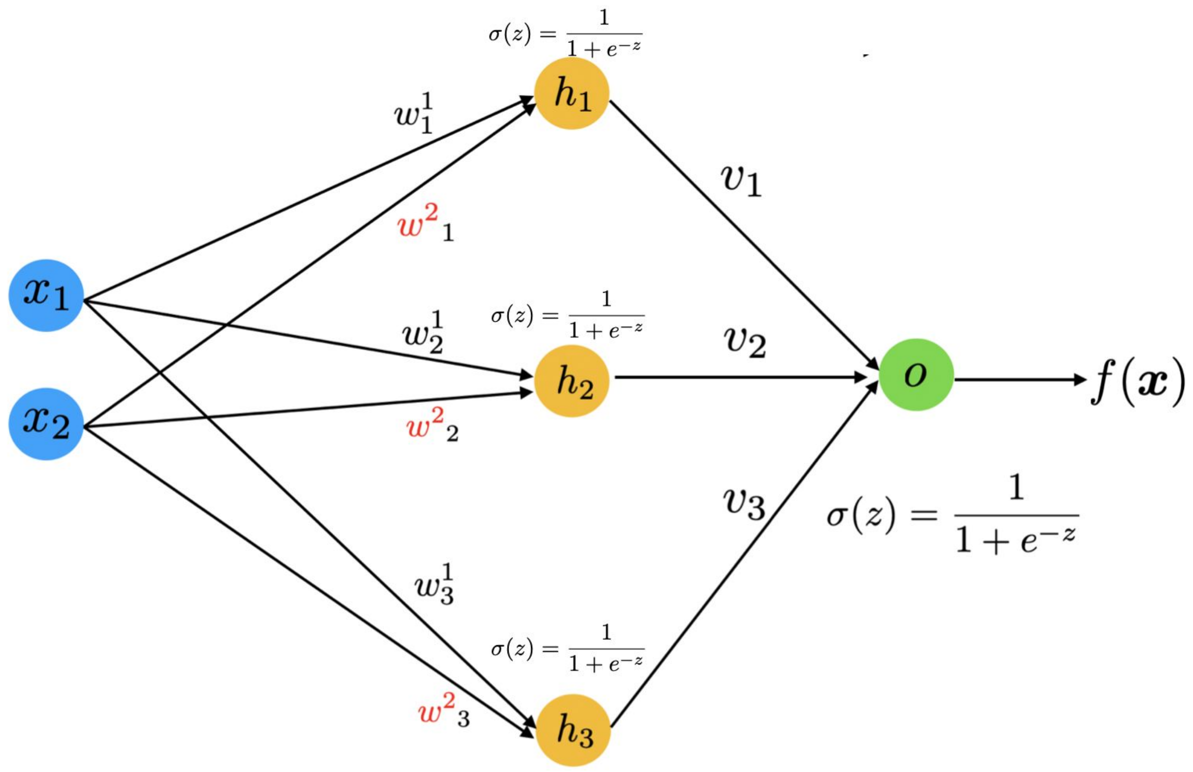
We use the **sigmoid** activation function which introduces better non-linearity:

Forward propagation



$\sigma(z) = \frac{1}{1+e^{-z}}$
w_1^1
$w^2{}_1$
h_1
v_1
x_1
w_2^1
$\sigma(z) = \frac{1}{1+e^{-z}}$
v_2
x_2
h_2
o
$f(\boldsymbol{x})$
$w^2{}_2$
v_3
$\sigma(z) = \frac{1}{1+e^{-z}}$
w_3^1
$\sigma(z) = \frac{1}{1+e^{-z}}$
$w^2{}_3$
h_3

Forward propagation



$\sigma(z) = \frac{1}{1+e^{-z}}$
h_1
w_1^1
$w^2{}_1$
v_1
x_1
w_2^1
$\sigma(z) = \frac{1}{1+e^{-z}}$
v_2
h_2
o
$f(\boldsymbol{x})$
x_2
$w^2{}_2$
v_3
$\sigma(z) = \frac{1}{1+e^{-z}}$
w_3^1
$\sigma(z) = \frac{1}{1+e^{-z}}$
$w^2{}_3$
h_3

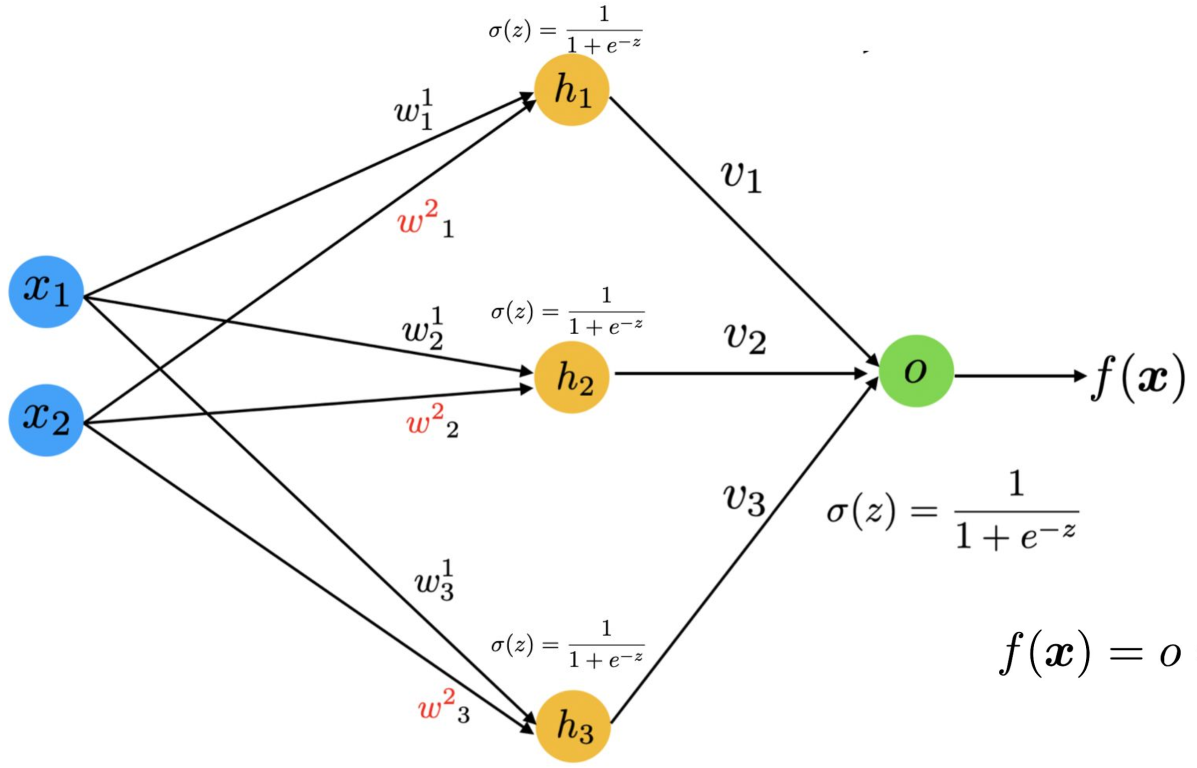
$$h_1 = \sigma(w_1^1 x_1 + w_1^2 x_2)$$

$$h_2 = \sigma(w_2^1 x_1 + w_2^2 x_2)$$

$$h_3 = \sigma(w_3^1 x_1 + w_3^2 x_2)$$

$$o = \sigma(v_1 h_1 + v_2 h_2 + v_3 h_3)$$

Forward propagation


$\sigma(z) = \frac{1}{1+e^{-z}}$
h_1
w_1^1
${w^2}_1$
v_1
x_1
w_2^1
$\sigma(z) = \frac{1}{1+e^{-z}}$
v_2
h_2
o
$f(\boldsymbol{x})$
x_2
${w^2}_2$
v_3
$\sigma(z) = \frac{1}{1+e^{-z}}$
w_3^1
$\sigma(z) = \frac{1}{1+e^{-z}}$
${w^2}_3$
h_3

$$h_1 = \sigma(w_1^1 x_1 + w_1^2 x_2)$$

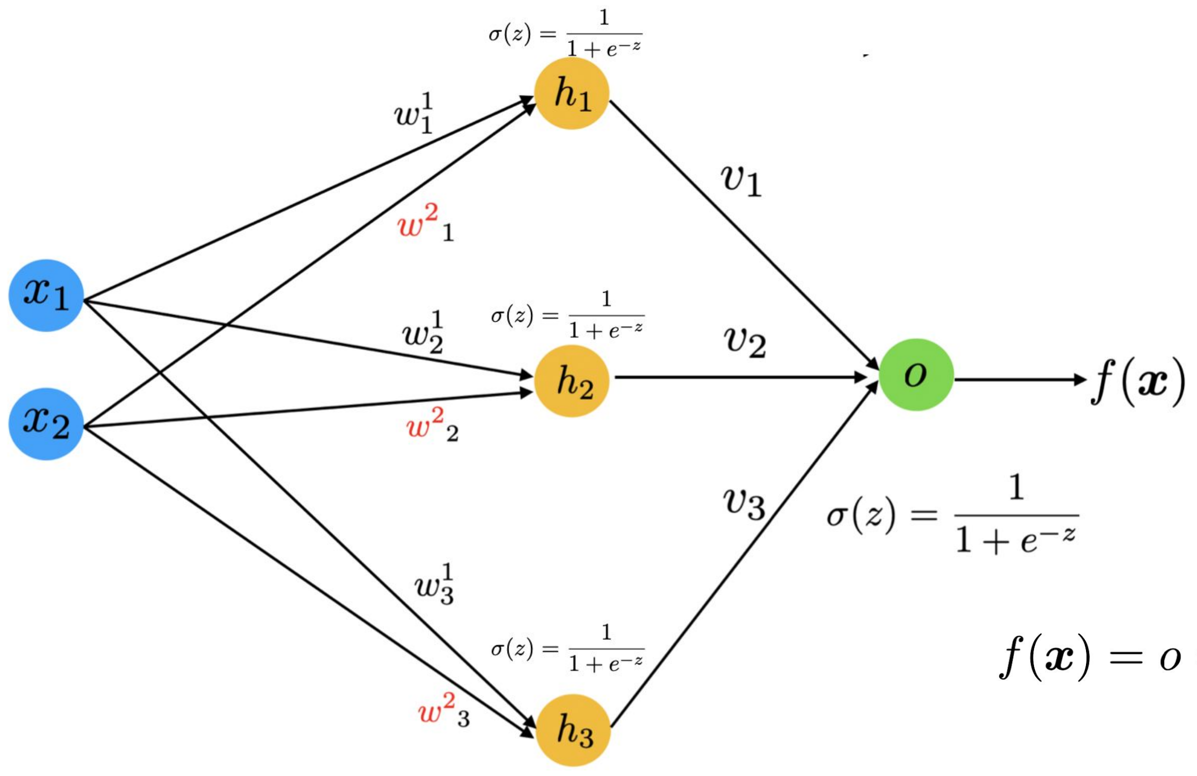
$$h_2 = \sigma(w_2^1 x_1 + w_2^2 x_2)$$

$$h_3 = \sigma(w_3^1 x_1 + w_3^2 x_2)$$

$$o = \sigma(v_1 h_1 + v_2 h_2 + v_3 h_3)$$

$$f(\boldsymbol{x}) = o = \quad \sigma(v_1 h_1 + v_2 h_2 + v_3 h_3)$$

Forward propagation - Compact Form



$$h_1 = \sigma(w_1^1 x_1 + w_1^2 x_2)$$

$$h_2 = \sigma(w_2^1 x_1 + w_2^2 x_2)$$

$$h_3 = \sigma(w_3^1 x_1 + w_3^2 x_2)$$

$$o = \sigma(v_1 h_1 + v_2 h_2 + v_3 h_3)$$

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

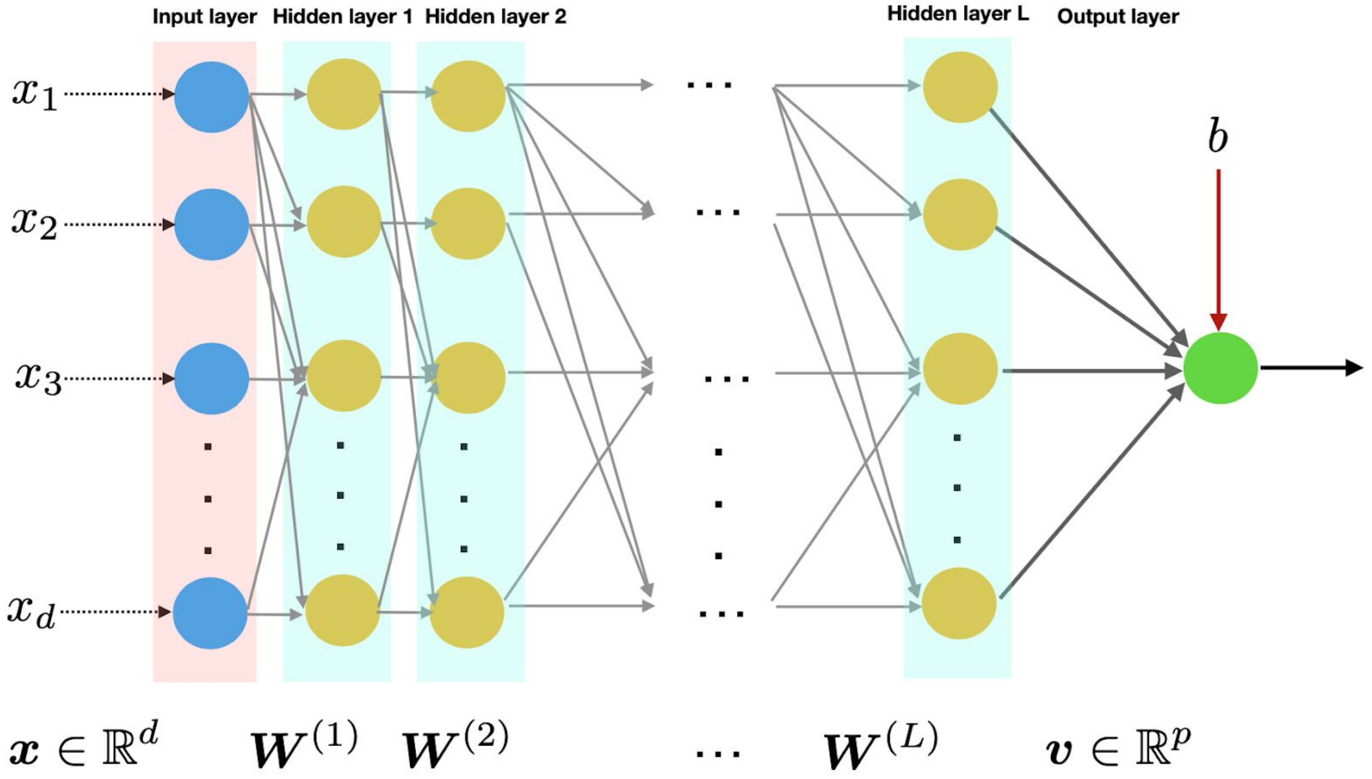
$$f(\boldsymbol{x}) = o = \sigma(v_1 h_1 + v_2 h_2 + v_3 h_3)$$

$$= \sigma\left(\boldsymbol{v}^\top \boldsymbol{\sigma}\left(\boldsymbol{W}^\top \boldsymbol{x}\right)\right)$$

$\boldsymbol{W} \in \mathbb{R}^{3\times 2}$ $\qquad$ $\boldsymbol{v} \in \mathbb{R}^3$

Beyond two layers: multilayer neural networks

Why not add more and more layers to learn more complex decision boundaries?



Discriminative Model

The discriminative model is parameterized by θ

$$P(y|X;\theta)$$

Discriminative Model Objective Function

The discriminative model is parameterized by θ

$$P(y|X;\theta)$$

We often use **negative log likelihood** over training data as our **objective function** or **loss function.** It is a function of θ

$$\mathcal{L}(\theta) = -\sum_{(X,y)\in\mathcal{D}_{\text{train}}} \log P(y|X;\theta)$$

Discriminative Model Objective Function

The discriminative model is parameterized by θ

$$P(y|X;\theta)$$

We often use **negative log likelihood** over training data as our **objective function** or **loss function.** It is a function of θ

$$\mathcal{L}(\theta) = -\sum_{(X,y)\in\mathcal{D}_{\text{train}}} \log P(y|X;\theta)$$

We then optimize the parameters to minimize the loss. Better model has lower loss

$$\hat{\theta} = \arg\min_{\theta} \mathcal{L}(\theta)$$

Optimize Objective Function by Gradient Descent

Calculate the gradient of the loss function with respect to the parameter

$$\frac{\partial \mathcal{L}(\theta)}{\partial \theta}$$

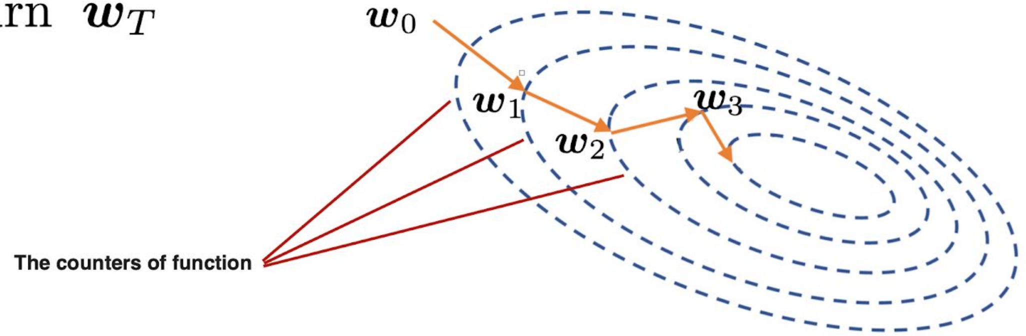
Update θ by moving a small step in the gradient direction to decrease the loss

$$\theta_{\text{new}} \leftarrow \theta_{\text{old}} - \eta \frac{\partial \mathcal{L}(\theta)}{\partial \theta}$$

η is the **learning rate**

Gradient Descent

```
$\boldsymbol{w}_0 = \mathbf{0}$
for $t = 1, 2, \ldots, T$
    $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta_t \nabla f(\boldsymbol{w}_t)$
end for
return $\boldsymbol{w}_T$
```



Gradient Descent

The key operation in gradient descent is to calculate the gradient.

Gradient of univariate scalar-valued functions

We are all familiar with basic calculus, and functions of the form $f : \mathbb{R} \to \mathbb{R}$ and their derivatives:

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}.$$

The derivative of a function of a real variable measures the sensitivity to change of the function value (output value) with respect to a change in its argument (input value).

- $f(x) = x^r$, then $f'(x) = rx^{r-1}$,
- $\frac{d}{dx} e^x = e^x$.
- Sums rule: $(\alpha f + \beta g)' = \alpha f' + \beta g'$ for all functions f and g and all real numbers α and β
- Product rule: $(fg)' = f'g + fg'$ for all functions f and g.
- Chain rule: If $f(x) = h(g(x))$, then

$$f'(x) = h'(g(x)) \cdot g'(x).$$

Gradient of multivariate scalar-valued functions

Definition

Let $f : \mathcal{C} \subseteq \mathbb{R}^d \to \mathbb{R}$ be a differentiable function. Then, the gradient of f at $\boldsymbol{x} \in \mathcal{C}$ is the vector in $\mathbb{R}^d$ denoted by $\nabla f(\boldsymbol{x})$ and defined by

$$f(\boldsymbol{x}) = f(x_1, x_2, ..., x_n) \qquad \nabla f(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\boldsymbol{x}) \\ \vdots \\ \frac{\partial f}{\partial x_d}(\boldsymbol{x}) \end{bmatrix}$$

Gradient of multivariate vector-valued functions: Jacobian Matrix

Given a function with ***m* outputs** and *n* inputs

$$\boldsymbol{f}(\boldsymbol{x}) = [f_1(x_1, x_2, ..., x_n), ..., f_m(x_1, x_2, ..., x_n)]$$

It's Jacobian is an ***m* x *n* matrix** of partial derivatives

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

$$\boxed{\left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right)_{ij} = \frac{\partial f_i}{\partial x_j}}$$

Example Jacobian: Elementwise activation Function

$$\boldsymbol{h} = f(\boldsymbol{z}), \text{what is } \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}? \qquad \boldsymbol{h}, \boldsymbol{z} \in \mathbb{R}^n$$

$$h_i = f(z_i)$$

$$\left(\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}\right)_{ij} = \frac{\partial h_i}{\partial z_j} = \frac{\partial}{\partial z_j} f(z_i) \qquad \text{definition of Jacobian}$$

$$= \begin{cases} f'(z_i) & \text{if } i = j \\ 0 & \text{if otherwise} \end{cases} \qquad \text{regular 1-variable derivative}$$

$$\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} = \begin{pmatrix} f'(z_1) & & 0 \\ & \ddots & \\ 0 & & f'(z_n) \end{pmatrix} = \operatorname{diag}(\boldsymbol{f}'(\boldsymbol{z}))$$

Other Jacobians

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}) = \boldsymbol{W}$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}) = \boldsymbol{I} \;\; \text{(Identity matrix)}$$

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T\boldsymbol{h}) = \boldsymbol{h}^{\boldsymbol{T}}$$

Left as exercise. Will be used later.

Chain Rule

Chain Rule tells us how to calculate gradients of composite functions.

For composition of one-variable functions: **multiply derivatives**

$$z = 3y$$

$$y = x^2$$

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = (3)(2x) = 6x$$

For multiple variables at once: **multiply Jacobians**

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \ldots$$

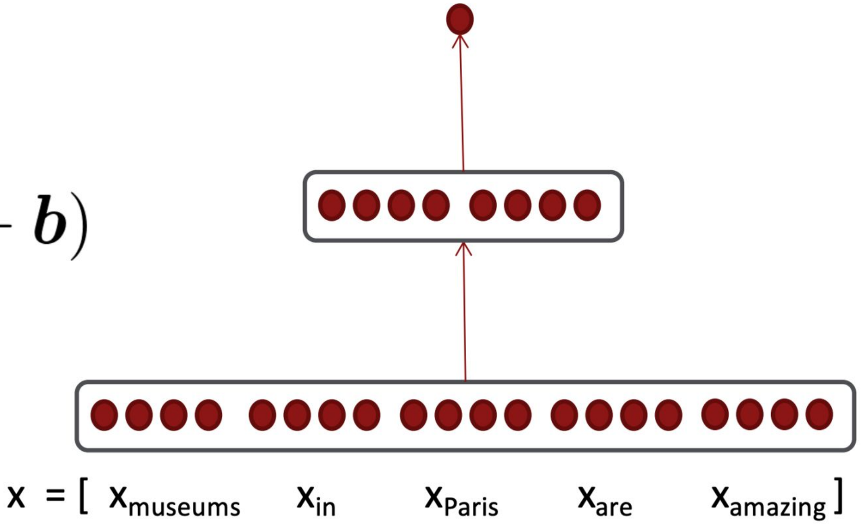
Example

Calculate $\frac{\partial s}{\partial \boldsymbol{b}}$ for the following feed-forward neural network.

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\boldsymbol{x} \quad \text{(input)}$$


x = [x_{museums} x_{in} x_{Paris} x_{are} x_{amazing}]

Example

Calculate $\frac{\partial s}{\partial \boldsymbol{b}}$

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad \text{(input)}$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \quad \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \quad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

Example

Calculate $\frac{\partial s}{\partial \boldsymbol{b}}$

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}) = \boldsymbol{W}$$
$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}) = \boldsymbol{I} \quad \text{(Identity matrix)}$$
$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T\boldsymbol{h}) = \boldsymbol{h}^{\boldsymbol{T}}$$

$$s = \boldsymbol{u}^T\boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$

$$\begin{aligned}\frac{\partial s}{\partial \boldsymbol{b}} &= \frac{\partial s}{\partial \boldsymbol{h}} \quad \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \quad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}} \\ &\quad\;\; \downarrow \qquad\;\; \downarrow \qquad\;\; \downarrow \\ &= \boldsymbol{u}^{\boldsymbol{T}} \operatorname{diag}(f'(\boldsymbol{z}))\boldsymbol{I} \\ &= \boldsymbol{u}^{\boldsymbol{T}} \circ f'(\boldsymbol{z})\end{aligned}$$

Example

Suppose we now want to compute $\frac{\partial s}{\partial \boldsymbol{W}}$

Using the chain rule again:

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$

$$\frac{\partial s}{\partial \boldsymbol{W}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$$
$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

The same! Let's avoid duplicated computation …

Example

Suppose we now want to compute $\frac{\partial s}{\partial \boldsymbol{W}}$

Using the chain rule again:

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$

$$\frac{\partial s}{\partial \boldsymbol{W}} = \boldsymbol{\delta} \boxed{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}}$$
$$\frac{\partial s}{\partial \boldsymbol{b}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}} = \boldsymbol{\delta}$$
$$\boldsymbol{\delta} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} = \boldsymbol{u}^T \circ f'(\boldsymbol{z})$$

Example

How to compute $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$, where $\boldsymbol{z} \in \mathbb{R}^n$ and $\boldsymbol{W} \in \mathbb{R}^{n \times m}$

n outputs, nm inputs, by definition, the Jacobian $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$ is $n \times nm$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \in \mathbb{R}^m$$

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} = \begin{bmatrix} \boldsymbol{x}^\top & & & & \\ & \boldsymbol{x}^\top & & & \\ & & \ddots & & \\ & & & \boldsymbol{x}^\top & \\ & & & & \boldsymbol{x}^\top \end{bmatrix}$$

Example

Now we can calculate $\frac{\partial s}{\partial \boldsymbol{W}}$ of shape $1 \times nm$

$$\frac{\partial \boldsymbol{s}}{\partial \boldsymbol{W}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} = \begin{bmatrix} \delta_1 & \delta_2 & \dots & \delta_n \end{bmatrix} \begin{bmatrix} \boldsymbol{x}^\top & & & & \\ & \boldsymbol{x}^\top & & & \\ & & \ddots & & \\ & & & \boldsymbol{x}^\top & \\ & & & & \boldsymbol{x}^\top \end{bmatrix} = \begin{bmatrix} \delta_1 \boldsymbol{x}^\top & \delta_2 \boldsymbol{x}^\top & \delta_3 \boldsymbol{x}^\top & \dots & \delta_n \boldsymbol{x}^\top \end{bmatrix}$$

Shape Convention

The shape of the parameter is $\boldsymbol{W} \in \mathbb{R}^{n \times m}$

By definition of Jacobian, 1 output, nm inputs, so the shape of $\frac{\partial s}{\partial \boldsymbol{W}}$ is $1 \times nm$

To make it more convenient for make gradient update, we use the
shape convention: the shape of the gradient is the shape of the parameters!

So $\frac{\partial s}{\partial \boldsymbol{W}}$ is *n* by *m*:
$$\begin{bmatrix} \frac{\partial s}{\partial W_{11}} & \cdots & \frac{\partial s}{\partial W_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial s}{\partial W_{n1}} & \cdots & \frac{\partial s}{\partial W_{nm}} \end{bmatrix}$$

Example

For shape convention, we reshape $\frac{\partial s}{\partial \boldsymbol{W}}$, so it has the same shape as $\boldsymbol{W} \in \mathbb{R}^{n \times m}$

$$\underset{1 \times nm}{\begin{bmatrix} \delta_1 \boldsymbol{x}^\top & \delta_2 \boldsymbol{x}^\top & \delta_3 \boldsymbol{x}^\top & \dots & \delta_n \boldsymbol{x}^\top \end{bmatrix}} \Longrightarrow \begin{bmatrix} \delta_1 \boldsymbol{x}^\top \\ \delta_2 \boldsymbol{x}^\top \\ \delta_3 \boldsymbol{x}^\top \\ \dots \\ \delta_n \boldsymbol{x}^\top \end{bmatrix} \quad n \times m$$

Example

For shape convention, we reshape $\frac{\partial s}{\partial \boldsymbol{W}}$, so it has the same shape as $\boldsymbol{W} \in \mathbb{R}^{n \times m}$

$$\underset{1 \times nm}{\begin{bmatrix} \delta_1 \boldsymbol{x}^\top & \delta_2 \boldsymbol{x}^\top & \delta_3 \boldsymbol{x}^\top & \dots & \delta_n \boldsymbol{x}^\top \end{bmatrix}} \Longrightarrow \begin{bmatrix} \delta_1 \boldsymbol{x}^\top \\ \delta_2 \boldsymbol{x}^\top \\ \delta_3 \boldsymbol{x}^\top \\ \dots \\ \delta_n \boldsymbol{x}^\top \end{bmatrix} \quad n \times m$$

A more direct way: compute the **outer product** of $\boldsymbol{\delta}$ and $\boldsymbol{x}$

and this directly makes the shape $n \times m$

$$\underset{[n \times m]}{\frac{\partial s}{\partial \boldsymbol{W}}} = \underset{[n \times 1]}{\boldsymbol{\delta}^T} \underset{[1 \times m]}{\boldsymbol{x}^T}$$

$$= \begin{bmatrix} \delta_1 \\ \vdots \\ \delta_n \end{bmatrix} [x_1, \dots, x_m] = \begin{bmatrix} \delta_1 x_1 & \dots & \delta_1 x_m \\ \vdots & \ddots & \vdots \\ \delta_n x_1 & \dots & \delta_n x_m \end{bmatrix}$$

Backpropagation

We use **gradient descent** to optimize the parameters in neural networks.

The key question: How can we efficiently calculate the gradients in neural networks (or any computational graph)?

The key intuition: View neural networks (or any computational graph) as compositions of functions and use **chain rules** to calculate the gradient.

This is **Backpropagation = Gradient Descent + Chain Rule**

Computation Graph

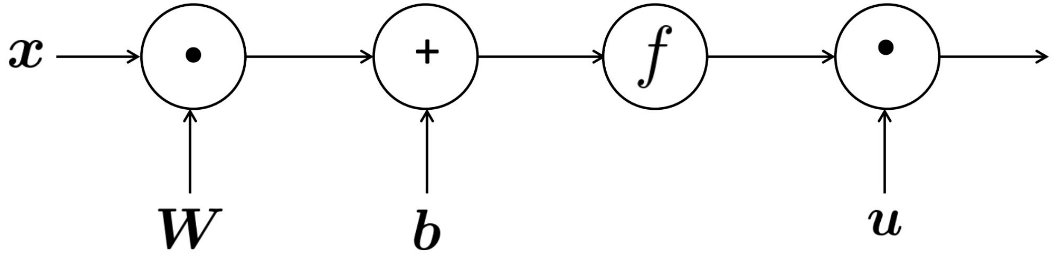
In software library, neural networks are implemented as Computation Graph (CG).

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

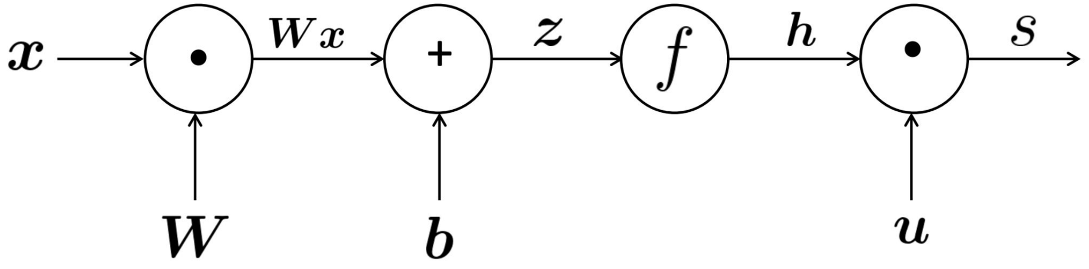
$$\boldsymbol{x} \quad \text{(input)}$$



Forward Propagation in CG

In software library, neural networks are implemented as Computation Graph (CG).

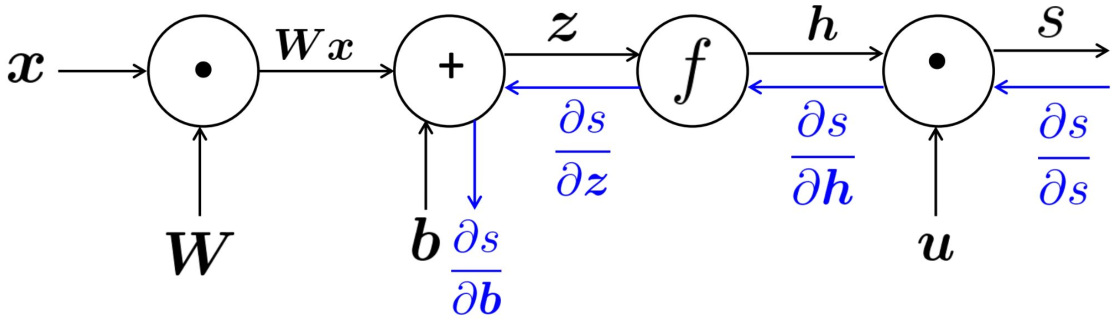
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



Backpropagation in CG

In software library, neural networks are implemented as Computation Graph (CG).

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



$\boldsymbol{x}$
$\boldsymbol{W}\boldsymbol{x}$
$\boldsymbol{z}$
$\boldsymbol{h}$
s
f
+
$\boldsymbol{W}$
$\boldsymbol{b}$
$\boldsymbol{u}$
$\frac{\partial s}{\partial \boldsymbol{b}}$
$\frac{\partial s}{\partial \boldsymbol{z}}$
$\frac{\partial s}{\partial \boldsymbol{h}}$
$\frac{\partial s}{\partial s}$

Backpropagation avoids duplicated computation

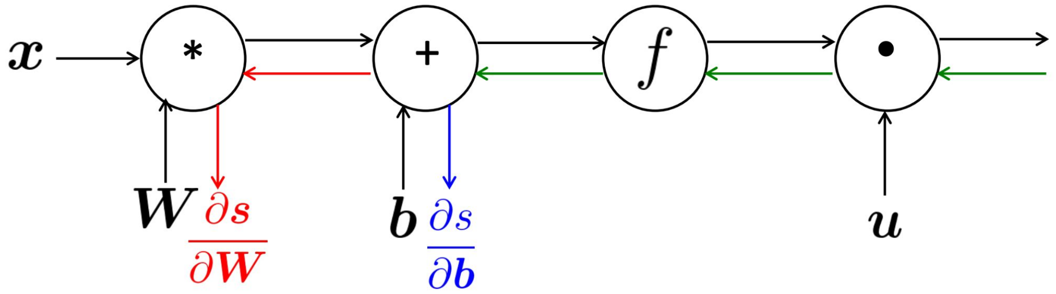
Backpropagation follows the computation graph in reverse order, so you avoid dupdated computation.

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad \text{(input)}$$



$\boldsymbol{x}$
*
+
f
•
$\boldsymbol{W}$
$\frac{\partial s}{\partial \boldsymbol{W}}$
$\boldsymbol{b}$
$\frac{\partial s}{\partial \boldsymbol{b}}$
$\boldsymbol{u}$

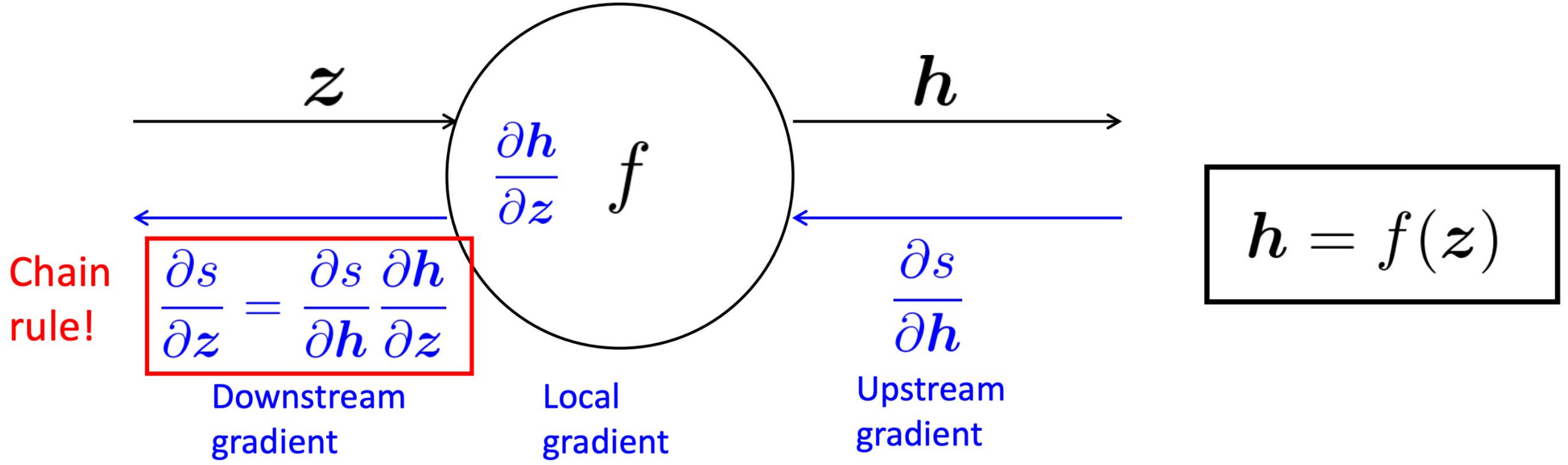
Backpropagation: Single Node

Node receives an **upstream gradient**.

Each node has a **local gradient.**

Goal is to pass on the correct **downstream gradient**.

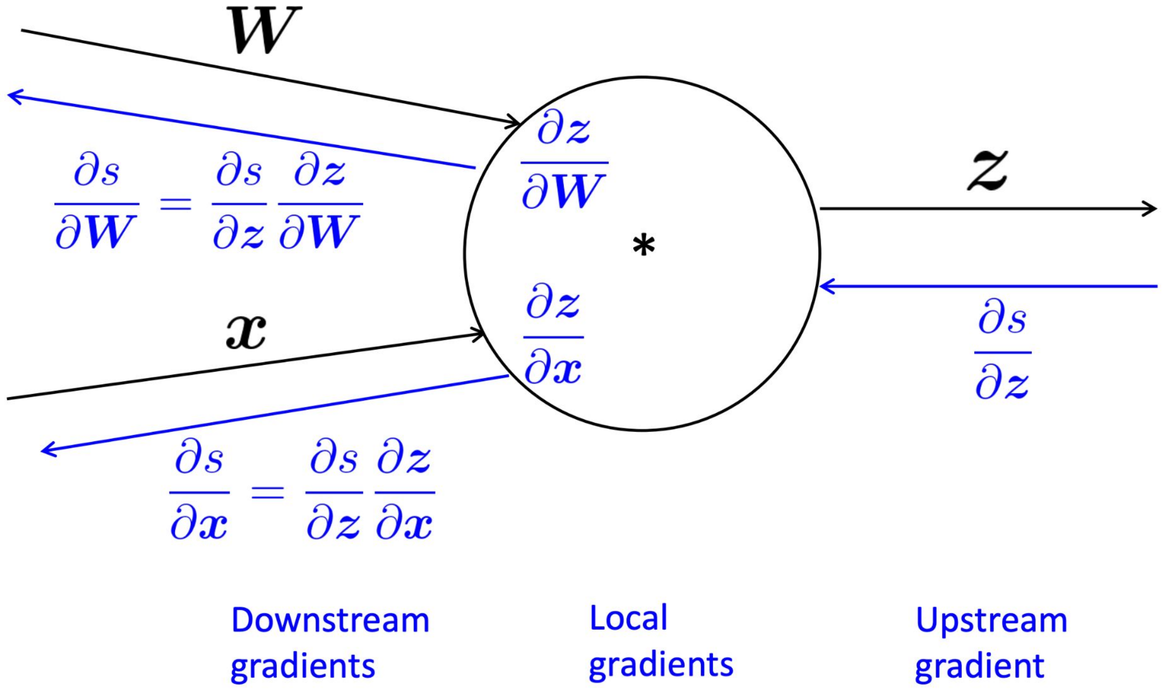
By Chain Rule: **downstream gradient = upstream gradient x local gradient.**



$\boldsymbol{z}$
$\boldsymbol{h}$
$\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}$
f
Chain rule!
$\boxed{\frac{\partial s}{\partial \boldsymbol{z}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}}$
$\frac{\partial s}{\partial \boldsymbol{h}}$
$\boxed{\boldsymbol{h} = f(\boldsymbol{z})}$
Downstream gradient
Local gradient
Upstream gradient

Backpropagation: Multiple Inputs

With multiple inputs, each input has a local gradient.

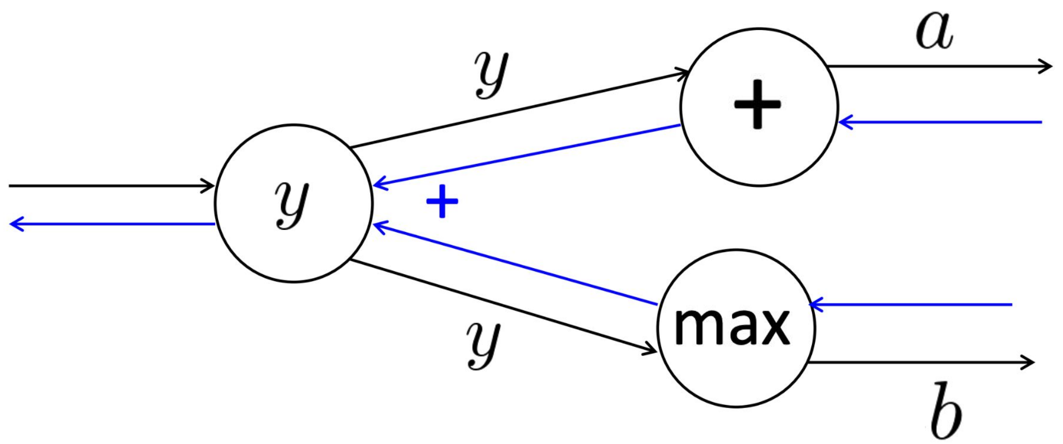


$\boldsymbol{W}$
$\frac{\partial s}{\partial \boldsymbol{W}} = \frac{\partial s}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$
$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$
$*$
$\boldsymbol{z}$
$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$
$\boldsymbol{x}$
$\frac{\partial s}{\partial \boldsymbol{z}}$
$\frac{\partial s}{\partial \boldsymbol{x}} = \frac{\partial s}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$
Downstream gradients
Local gradients
Upstream gradient

$$\boxed{\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}}$$

Backpropagation: Multiple Outputs

If a node has multiple outgoing edges during forward propagation, we sum the gradients during the backpropagation.



$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial y}$$

Example

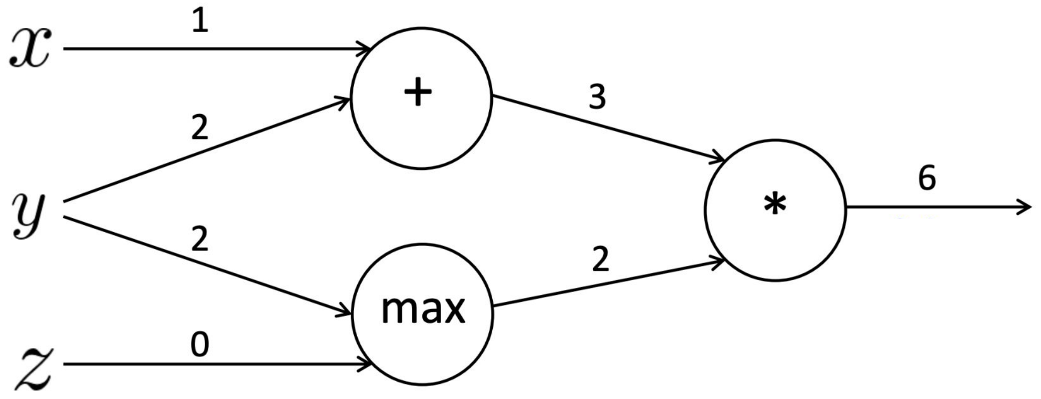
$$\boxed{\begin{array}{l} f(x,y,z) = (x+y)\max(y,z) \\ x=1, y=2, z=0 \end{array}}$$

Forward prop steps

$$a = x + y$$

$$b = \max(y, z)$$

$$f = ab$$



Example

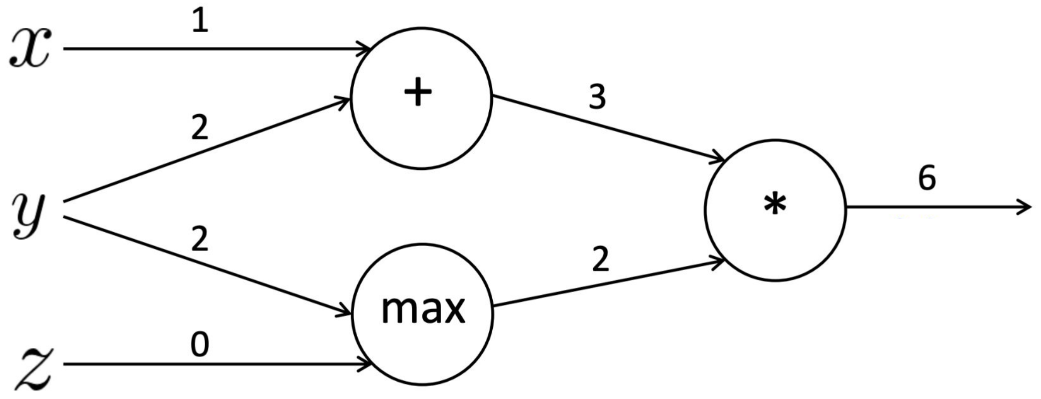
$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



Example

$$\boxed{\begin{array}{l} f(x,y,z) = (x+y)\max(y,z) \\ x=1, y=2, z=0 \end{array}}$$

Forward prop steps

$$a = x + y$$

$$b = \max(y,z)$$

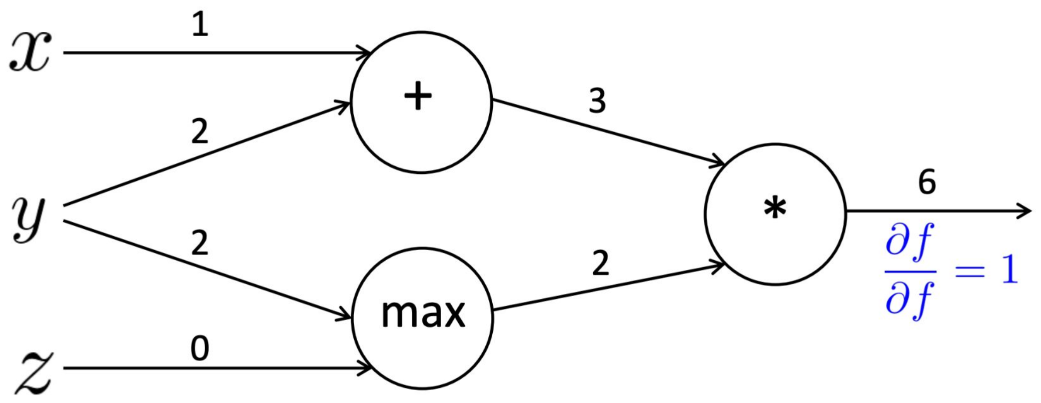
$$f = ab$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$

$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$

$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



Example

$$\boxed{\begin{array}{l} f(x,y,z) = (x+y)\max(y,z) \\ x=1, y=2, z=0 \end{array}}$$

Forward prop steps

$$a = x + y$$

$$b = \max(y, z)$$

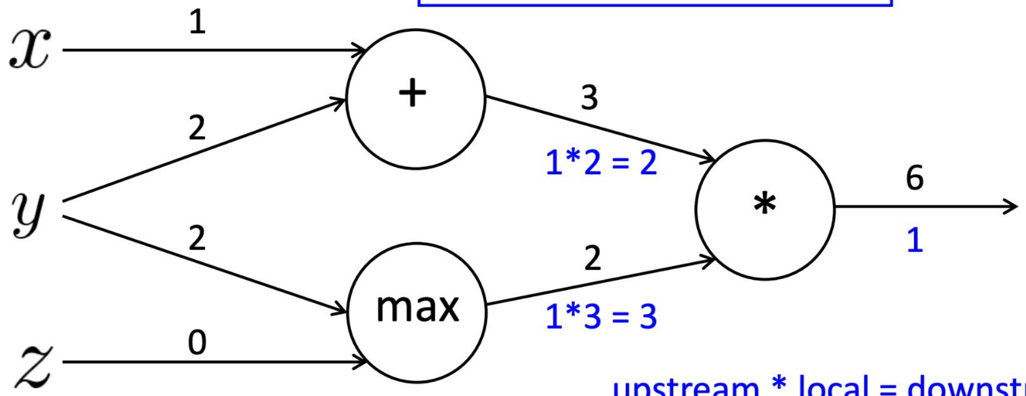
$$f = ab$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$

$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$

$$\boxed{\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3}$$



upstream * local = downstream

Example

$$\boxed{\begin{array}{l} f(x,y,z) = (x+y)\max(y,z) \\ x=1, y=2, z=0 \end{array}}$$

Forward prop steps

$$a = x + y$$

$$b = \max(y, z)$$

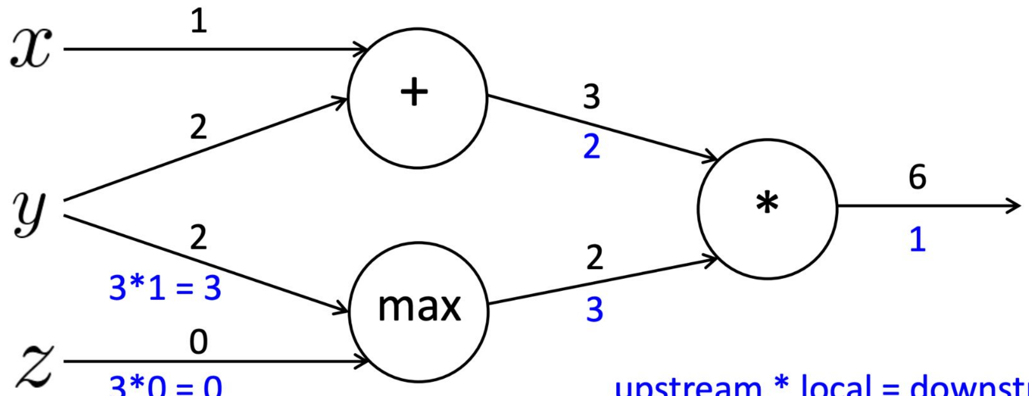
$$f = ab$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$

$$\boxed{\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0}$$

$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



upstream * local = downstream

Example

$$\boxed{\begin{array}{l} f(x, y, z) = (x + y)\max(y, z) \\ x = 1, y = 2, z = 0 \end{array}}$$

Forward prop steps

$$a = x + y$$

$$b = \max(y, z)$$

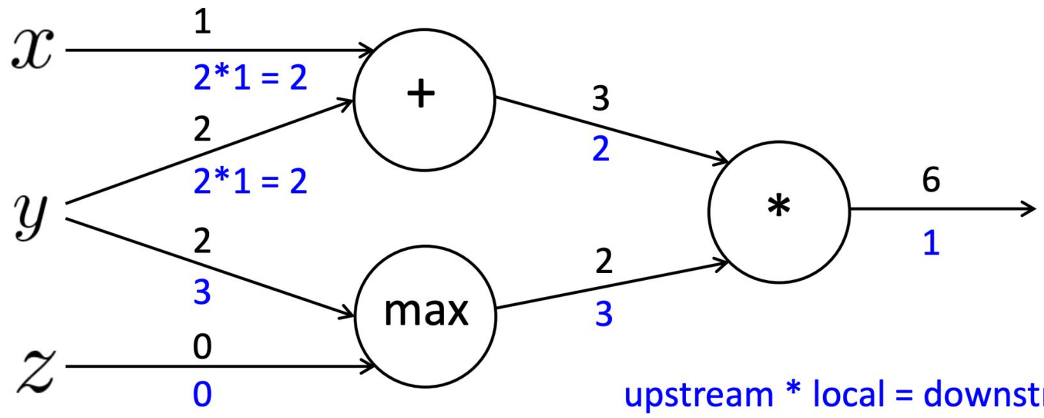
$$f = ab$$

Local gradients

$$\boxed{\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1}$$

$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$

$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



upstream * local = downstream

Example

$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

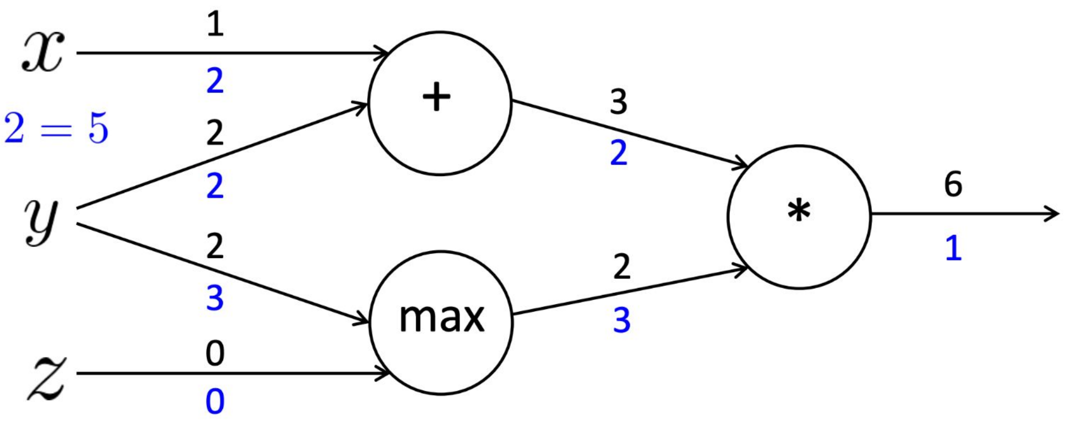
Forward prop steps

$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$

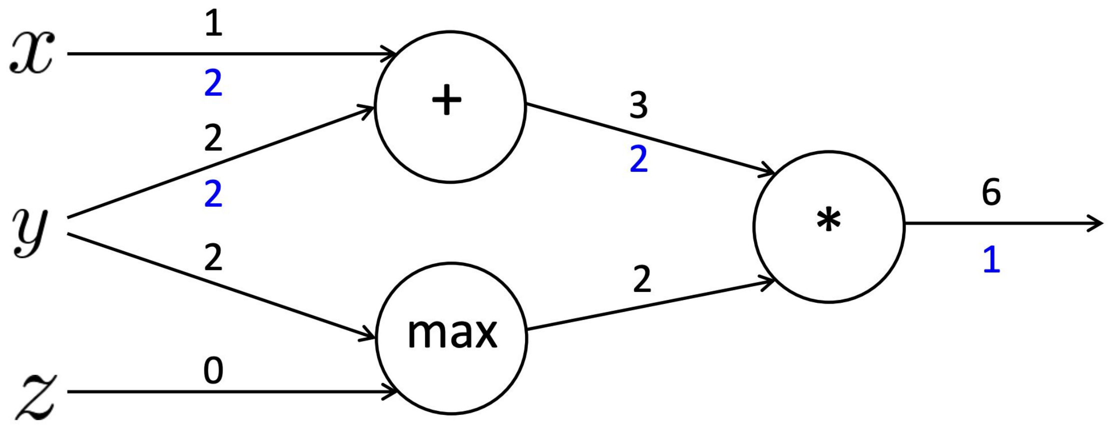
$$\frac{\partial f}{\partial x} = 2$$
$$\frac{\partial f}{\partial y} = 3 + 2 = 5$$
$$\frac{\partial f}{\partial z} = 0$$



Node Intuitions

+ “distributes” the upstream gradient to each summand

$$f(x,y,z) = (x+y)\max(y,z)$$
$$x=1, y=2, z=0$$

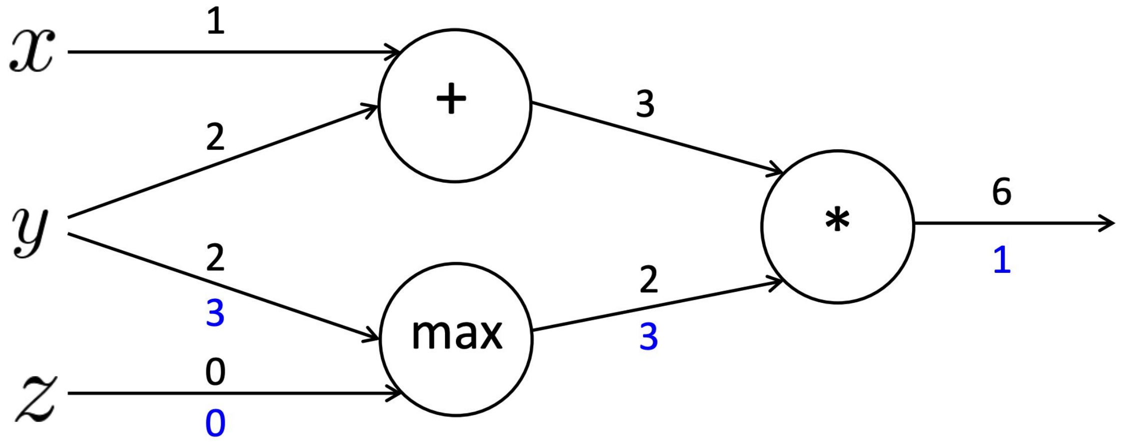


Node Intuitions

+ “distributes” the upstream gradient to each summand

max “routes” the upstream gradient

$$\boxed{\begin{aligned} &f(x,y,z) = (x+y)\max(y,z) \\ &x=1, y=2, z=0 \end{aligned}}$$



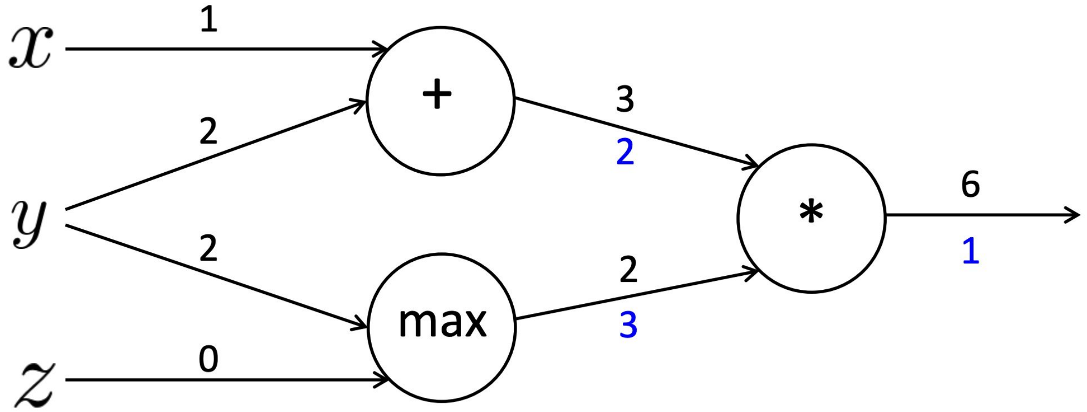
Node Intuitions

+ “distributes” the upstream gradient to each summand

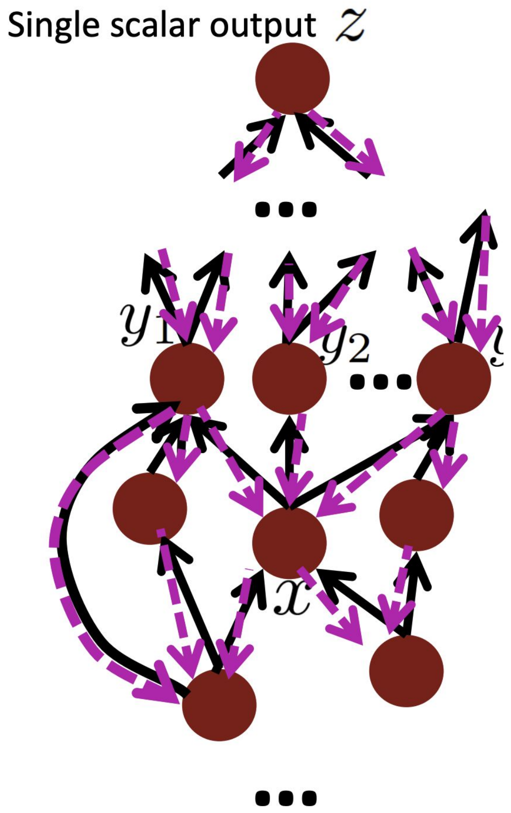
max “routes” the upstream gradient

* “switches” the upstream gradient

$$\boxed{\begin{array}{l} f(x,y,z) = (x+y)\max(y,z) \\ x=1, y=2, z=0 \end{array}}$$



Backpropagation in General Computation Graph



- F-prop: visit nodes in topological sort order - Compute value of node given predecessors.
- B-prop:
  - initialize output gradient = 1
  - visit nodes in reverse order: Compute gradient wrt each node using gradient wrt successors

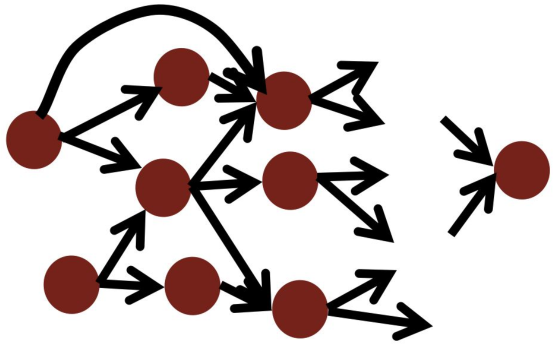
$\{y_1,\ y_2,\ \ldots\ y_n\}$ = successors of x

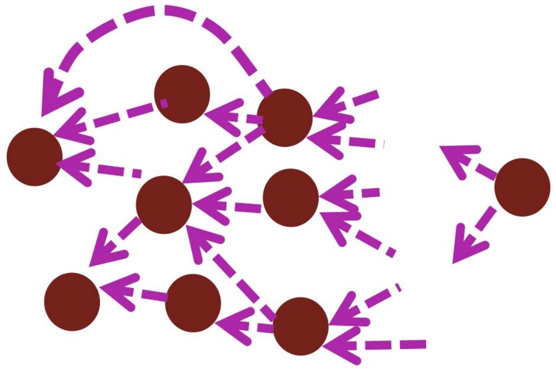
$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

AutoDiff: Automatic Differentiation

The gradient computation can be automatically inferred from the symbolic expression of the fprop.

Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output.





Check Your Implementation with Numeric Gradient

Following the definition of gradient:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Use a very small h to compute this value numerically.

Useful for checking your implementation.

Optimizers

Once we calculate the gradient with respect to the objective function, we optimize objective function by gradient descent.

The very basic one is standard gradient descent. But there are many other choices of optimizers:

- Batch Gradient Descent
- Stochastic Gradient Descent (SGD)
- Mini-batch Gradient Descent
- SGD with Momentum
- Nesterov accelerated gradient
- Adagrad
- Adadelta
- RMSProp
- **Adam; AdamW (most commonly used)**
- Adafactor

Batch Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

New Parameters Old Parameters Learning Rate Gradient

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Mini-batch Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

$\uparrow$

Gradient for a batch of training examples; batch size n

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Stochastic Gradient Descent (SGD)

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

$\uparrow$

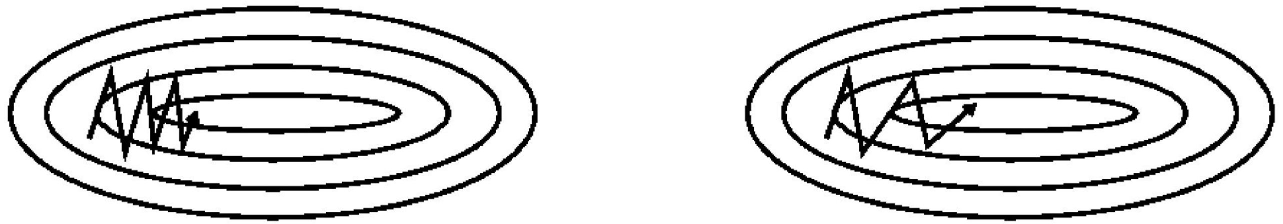
Gradient for each training example

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

SGD with Momentum

Momentum parameter, usually 0.9 or a similar value

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$



(a) SGD without momentum (b) SGD with momentum

Figure 2: Source: Genevieve B. Orr

Nesterov Accelerated Gradient (NAG)

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters

Nesterov Accelerated Gradient (NAG)

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

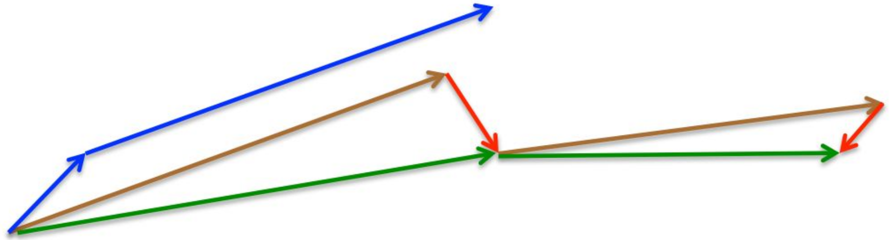
calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters

Momentum

- first computes the current gradient (small blue vector)
- then takes a big jump in the direction of the updated accumulated gradient (big blue vector)

NAG

- first makes a big jump in the direction of the previous accumulated gradient (brown vector)
- measures the gradient and then makes a correction (red vector)
- results in the complete NAG update (green vector).



Adagrad

Adagrad adapts the learning rate to the parameters: **larger updates for infrequent parameters, smaller updates for frequent parameters.**

It uses a different learning rate for every parameter θi at every time step t,

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i})$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\boxed{\sqrt{G_{t,ii} + \epsilon}}} \cdot g_{t,i}$$

$G_t \in \mathbb{R}^{d \times d}$ A diagonal matrix where each diagonal element is the sum of the squares of the gradients w.r.t. θi up to time step t

ϵ smoothing term that avoids division by zero

Adagrad

Adagrad adapts the learning rate to the parameters: **larger updates for infrequent parameters, smaller updates for frequent parameters.**

It uses a different learning rate for every parameter θi at every time step t,

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i})$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\boxed{\sqrt{G_{t,ii} + \epsilon}}} \cdot g_{t,i}$$

$G_t \in \mathbb{R}^{d \times d}$ A diagonal matrix where each diagonal element is the sum of the squares of the gradients w.r.t. θi up to time step t

ϵ smoothing term that avoids division by zero

Problem: learning rate continuously decreases, and training can stall -- fixed by using **rolling average** in AdaDelta, RMSProp, Adam.

Adaptive Moment Estimation (Adam)

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \longleftarrow \text{like momentum}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \longleftarrow \text{rolling average!}$$

Adaptive Moment Estimation (Adam)

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad \longleftarrow \quad \text{like momentum}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \quad \longleftarrow \quad \text{rolling average!}$$

As m_t and v_t are initialized as vectors of 0’s, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β1 and β2 are close to 1). They counteract these biases by computing bias-corrected first and second moment estimates,

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Then update the parameters

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Training Tricks

Shuffling the Training Data Every Epoch: avoid providing the training examples in a meaningful order to our model as this may bias the optimization algorithm.

Regularization: L2 regularization adds L2 norm of parameters to the loss function.

Dropout: randomly zero-out nodes in the hidden layer with probability p at training time only

Learning Rate Decay: gradually reduce learning rate as training continues

Early stopping

- You should thus always monitor error on a validation set during training and stop (with some patience) if your validation error does not improve enough.
- Use **Patience**

Parameter Initialization

Very Important for neural networks to achieve good performance.

Uniform Initialization: Initialize weights in some range, such as [-0.1, 0.1] for example

Xavier Initialization: n(l) is the number of input units to W (fan-in) and n(l+1) is the number of output units from W (fan-out).

$$W \sim U\left[-\sqrt{\frac{6}{n^{(l)} + n^{(l+1)}}}, \sqrt{\frac{6}{n^{(l)} + n^{(l+1)}}} \right]$$