Neural network

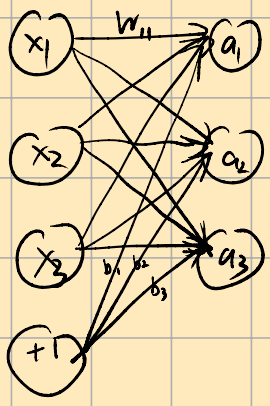
What's it?

equal to running several logistic regressions at the same time

matrix notation for a layer

we have:

$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$
$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$



In matrix notation:

$$z = Wx + b$$
$$a = f(z)$$

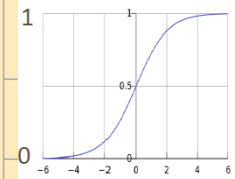
Activation f is applied elementwise (逐元素执行, "wise" here just like clockwise)

Activation

Non-linearities, old and new

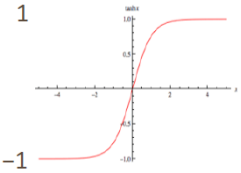
logistic ("sigmoid")

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



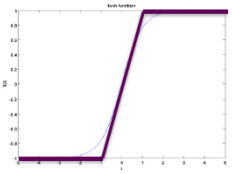
tanh

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



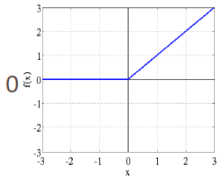
hard tanh

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

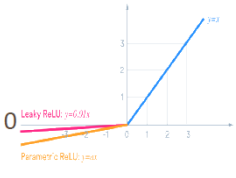


(Rectified Linear Unit) ReLU

$$\text{ReLU}(z) = \max(z, 0)$$



Leaky ReLU / Parametric ReLU



tanh is just a rescaled and shifted sigmoid (2 × as steep, [−1,1]):

$$\tanh(z) = 2\text{logistic}(2z) - 1$$

Logistic and tanh are still used (e.g., logistic to get a probability)

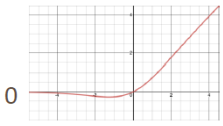
However, now, for deep networks, the first thing to try is ReLU: it trains quickly and performs well due to good gradient backflow.

ReLU has a negative "dead zone" that recent proposals mitigate

GELU is frequently used with Transformers (BERT, RoBERTa, etc.)

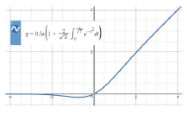
Swish arXiv:1710.05941

$$\text{swish}(x) = x \cdot \text{logistic}(x)$$



GELU arXiv:1606.08415

$$\text{GELU}(x) = x \cdot P(X \le x), X \sim N(0,1)$$
$$\approx x \cdot \text{logistic}(1.702x)$$



1. problem of sigmoid: allways non-negative => push towards bigger numbers
2. problem of both sigmoid and tanh: exponential computes are bit slow in computer
3. based on hard tanh, ReLU is developed
4. ReLu is sort of default in the norm
5. Swish / GELU's positive part, x is not exactly equal to y, but approximately do

Why should non-linearity?

if only linearity in net, matrix multiply (@ in Python) matrix is linear transition, and multiple matrices' combination give no representational power

$$M_1 @ M_2 @ M_3 @ M_4 \cdots @ M_n = M_{1 \sim n-1} @ M_n$$

So we don't need so much layers, net become back to logistic regression

① How to compute Gradient?

what we have is the equation of the neural network layer:

$$s = u^T h$$
$$h = f(z)$$
$$z = Wx + b$$

- input vector: $x \in R^d$
- weight matrix: $W \in R^{n \times d}$
- bias vector: $b \in R^n$

- result of the linear transformation: $z \in R^n$
- element-wise activation function: $f()$
- activated output: $h \in R^n$

- output layer weight vector: $u \in R^n$
- final scalar output: $s \in R$

- n: hidden layer dimension
- d: input dimension

What we want to do?

Compute the gradients of the scalar output s with respect to the parameters b and W, namely $\frac{\partial s}{\partial b}$ and $\frac{\partial s}{\partial W}$

How to compute?

use what we learned from calculus

the Jacobian Matrix

Jacobian matrix

$$\vec{y} = f(\vec{x}) = [f_1(x_1, x_2, \cdots, x_n), \cdots, f_m(x_1, x_2, \cdots, x_n)] \quad \text{if } \vec{y} \in R^n \quad \vec{x} \in R^m$$

$$J = \frac{\partial \vec{y}}{\partial \vec{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix} \in R^{n \times m}$$

✓ Compute $\frac{\partial s}{\partial b}$

apply the Chain Rule

$$\frac{\partial s}{\partial b} = \frac{\partial s}{\partial h} \cdot \frac{\partial h}{\partial z} \cdot \frac{\partial z}{\partial b}$$

compute Individual Jabobians

1. $s = u^T h \Rightarrow \frac{\partial s}{\partial h} = u^T$

$$J = \left[\frac{\partial s}{\partial h_1}, \cdots, \frac{\partial s}{\partial h_n}\right] = [u_1 \; u_2 \; \cdots \; u_n] = u^T$$

2. $h = f(z)$

$$\frac{\partial h}{\partial z} = \begin{bmatrix} \frac{\partial h_1}{\partial z_1} & \cdots & \frac{\partial h_1}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_n}{\partial z_1} & \cdots & \frac{\partial h_n}{\partial z_n} \end{bmatrix}$$

for $\frac{\partial h_i}{\partial z_j}$, when $i = j$ $\quad \frac{\partial h_i}{\partial z_j} = \frac{\partial h_i}{\partial z_i} = f'(z_i)$

when $i \neq j$ $\quad \frac{\partial h_i}{\partial z_j} = 0$

because h_i only depend on z_i

$$\therefore \frac{\partial h}{\partial z} = \begin{bmatrix} f'(z_1) & & 0 \\ & \ddots & \\ 0 & & f'(z_n) \end{bmatrix} = \text{diag}(f'(z))$$

3. $\vec{z} = W\vec{x} + \vec{b} \Rightarrow \frac{\partial z}{\partial b} = I$ (identity matrix)

$z_1 = (Wx)_1 + b_1,\ z_2 = (Wx)_2 + b_2,\ \cdots,\ z_n = (Wx)_n + b_n$

$$J = \begin{bmatrix} \frac{\partial z_1}{\partial b_1} & \cdots & \frac{\partial z_1}{\partial b_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_n}{\partial b_1} & \cdots & \frac{\partial z_n}{\partial b_n} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} = I_n$$

Eventually, combine 1, 2, 3

$$\frac{\partial s}{\partial b} = u^T \cdot \text{diag}(f'(z)) \cdot I_n = u^T \cdot \text{diag}(f'(z))$$

$$= [u_1 f'(z_1), u_2 f'(z_2), \ldots, u_n f'(z_n)]$$

But this is a row vector.

Following the shape convention (gradient has the same shape as the parameter), we transpose it to get a column vector:

$$\frac{\partial s}{\partial b} = (u \circ f'(z))$$

where $\circ$ denotes the element-wise product (Hadamard product)

✓ compute $\frac{\partial s}{\partial W}$

we find that the $\frac{\partial s}{\partial h} \cdot \frac{\partial h}{\partial z}$ part is identical to before

$$\frac{\partial s}{\partial h} \cdot \frac{\partial h}{\partial z} = u^T \cdot \text{diag}(f'(z)) = \delta^T \qquad (\delta \in \mathbb{R}^n)$$

where $\delta^T = u^T \circ f'(z)$ (delta) is the error signal / upstream gradient in a row vector form.

Let $\delta = u \circ f'(z)$ be its column vector equivalent

4. compute $\frac{\partial z}{\partial W}$

for $z = Wx + b$, it can be written element-wise as

$$z_i = \sum_j^d W_{ij} x_j + b_i$$

we can find the partial derivative of the the scalar with W_{ij}

$$\frac{\partial S}{\partial W_{ij}} = \sum_{k=1}^{n} \frac{\partial S}{\partial z_k} \cdot \frac{\partial z_k}{\partial W_{ij}}$$

Since z_k only depend on the k-th row of the W

$$\frac{\partial z_k}{\partial W_{ij}} = \begin{cases} x_j & \text{if } k=i \\ 0 & \text{if } k \neq i \end{cases}$$

for example $z_3 = \sum_{j=1}^{d} W_{3j} \cdot x_j + b_3$

$$= (W_{31} \cdot x_1 + W_{32} \cdot x_2 + \cdots W_{3n} \cdot x_n) + b_3$$

$$\frac{\partial z_3}{\partial W_{31}} = x_1$$

Therefore:

$$\frac{\partial S}{\partial W_{ij}} = \frac{\partial S}{\partial z_i} \cdot x_i = \delta_i \cdot x_j$$

where $\delta_i = \frac{\partial S}{\partial z_i}$ is the i-th component of the error signal δ

Finally, Construct the Full Gradient Matrix

$$\frac{\partial S}{\partial W} = \begin{bmatrix} \delta_1 x_1 & \delta_1 x_2 & \cdots & \delta_1 x_d \\ \delta_2 x_1 & \delta_2 x_2 & \cdots & \delta_2 x_d \\ \vdots & \vdots & \ddots & \vdots \\ \delta_n x_1 & \delta_n x_2 & \cdots & \delta_n x_d \end{bmatrix} = \delta x^T$$

$$= (u \circ f'(x)) x^T$$

② Backpropagation

core
- (one), Backpropagation is the efficient application of the chain rule on a computational graph.
- (two) Backpropagation store intermediate results, so never recompute the same stuff again

why use it?

to efficiently calculate the gradient at a time computational

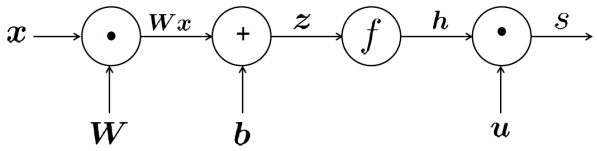
complexity of O(n)

Forward propagation ↻

Computation Graphs and Backpropagation

- Software represents our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations
  - Edges pass along result of the operation

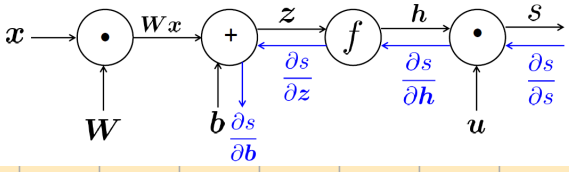
$$s = u^T h$$
$$h = f(z)$$
$$z = Wx + b$$
$$x \quad \text{(input)}$$



Backpropagation

- Then go backwards along edges
  - Pass along **gradients**

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



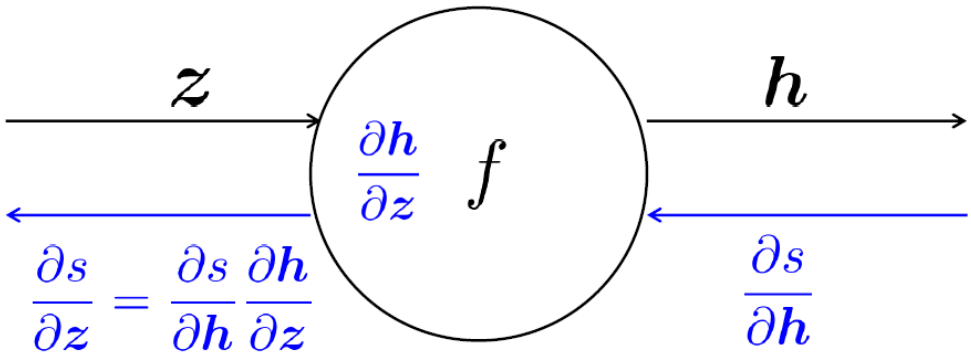
64

Backpropagation: Single Node

- Each node has a **local gradient**
  - The gradient of its output with respect to its input

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

[downstream gradient] = [upstream gradient] x [local gradient]

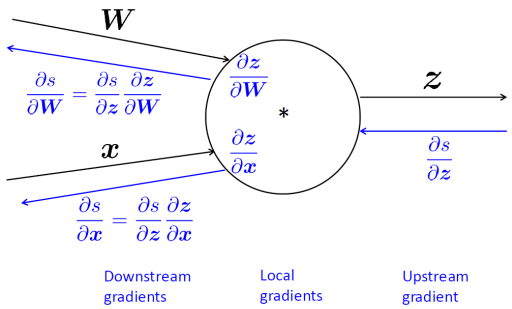


68

Backpropagation: Single Node

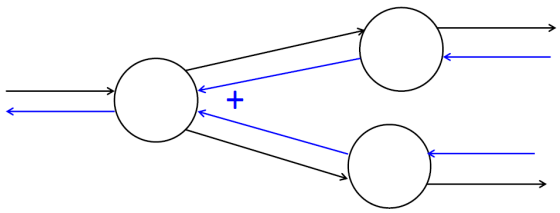
- Multiple inputs → multiple local gradients

$$z = Wx$$



70

Gradients sum at outward branches

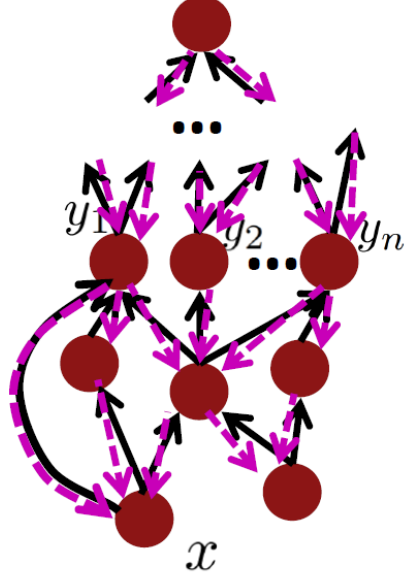


$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial y}$$

Back-Prop in General Computation Graph

Single scalar output z

1. Fprop: visit nodes in topological sort order
   - Compute value of node given predecessors



Inputs

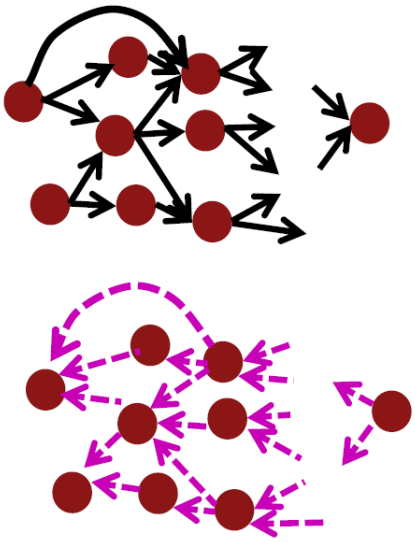
2. Bprop:
   - initialize output gradient = 1
   - visit nodes in reverse order:
   Compute gradient wrt each node using gradient wrt successors
   $\{y_1, y_2, \ldots y_n\}$ = successors of x

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

Done correctly, big O() complexity of fprop and bprop is **the same**

In general, our nets have regular layer-structure and so we can use matrices and Jacobians…

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
- Modern DL frameworks (Tensorflow, PyTorch, etc.) do backpropagation for you but mainly leave layer/node writer to hand-calculate the local derivative

Manual Gradient checking: Numeric Gradient

- For small *h* (≈ 1e-4), $f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$
- Easy to implement correctly
- But approximate and **very** slow:
  - You have to recompute *f* for **every parameter** of our model
- Useful for checking your implementation
  - In the old days, we hand-wrote everything, doing this everywhere was the key test
  - Now much less needed; you can use it to check layers are correctly implemented