Deep Feedforward Networks

Deep Neural Networks, Gradient Descent and Backpropagation

Debapriyo Majumdar

debapriyo@isical.ac.in

A supervised learning problem setup

Some property or *function* of x
we cannot model exactly

$$y = f^*(x)$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

x

Data point with different features / attributes / dimensions

Try to minimize the error on points for which the true values y are known (learn θ)

$$\hat{y} = f(x; \theta)$$

Prediction: compute some function that we can

Logistic regression: binary classification

Two possible values: $y \in \{0,1\}$

$$y = f^*(\boldsymbol{x})$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$\boldsymbol{x}$

Data point

Try to minimize the error on points for which the true values y are known (learn θ)

$$\hat{y} = f(\boldsymbol{x}; \theta)$$

Logistic regression: $f(\boldsymbol{x}) = \sigma(w^T \boldsymbol{x} + b) \in (0,1)$

The Sigmoid Function

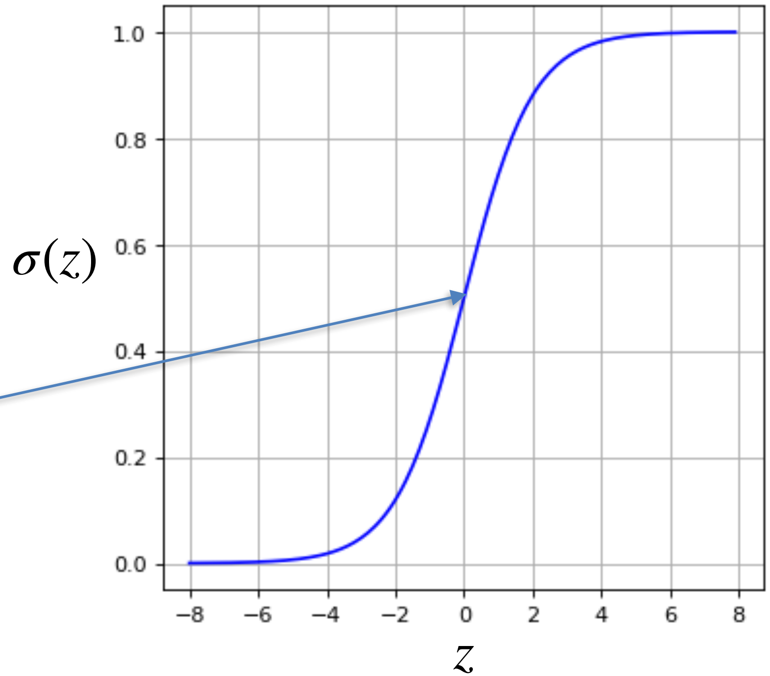
- The sigmoid function

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

- Properties

$$\sigma(0) = 0.5$$
$$\sigma(z) \to 1 \text{ as } z \to +\infty$$
$$\sigma(z) \to 0 \text{ as } z \to -\infty$$

- Can be used as a *probability*



Logistic regression: binary classification

Goal: compute a single value $\in \{0,1\}$

Weighted sum: weight w_i for feature x_i, and a bias b

Map it to $(0,1)$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \mapsto w_1x_1 + \cdots + w_nx_n + b = w^T\boldsymbol{x} + b \mapsto \sigma(w^T\boldsymbol{x} + b) = \hat{y}$$

A real number $\in (-\infty, \infty)$

$\boldsymbol{x}$

Data point

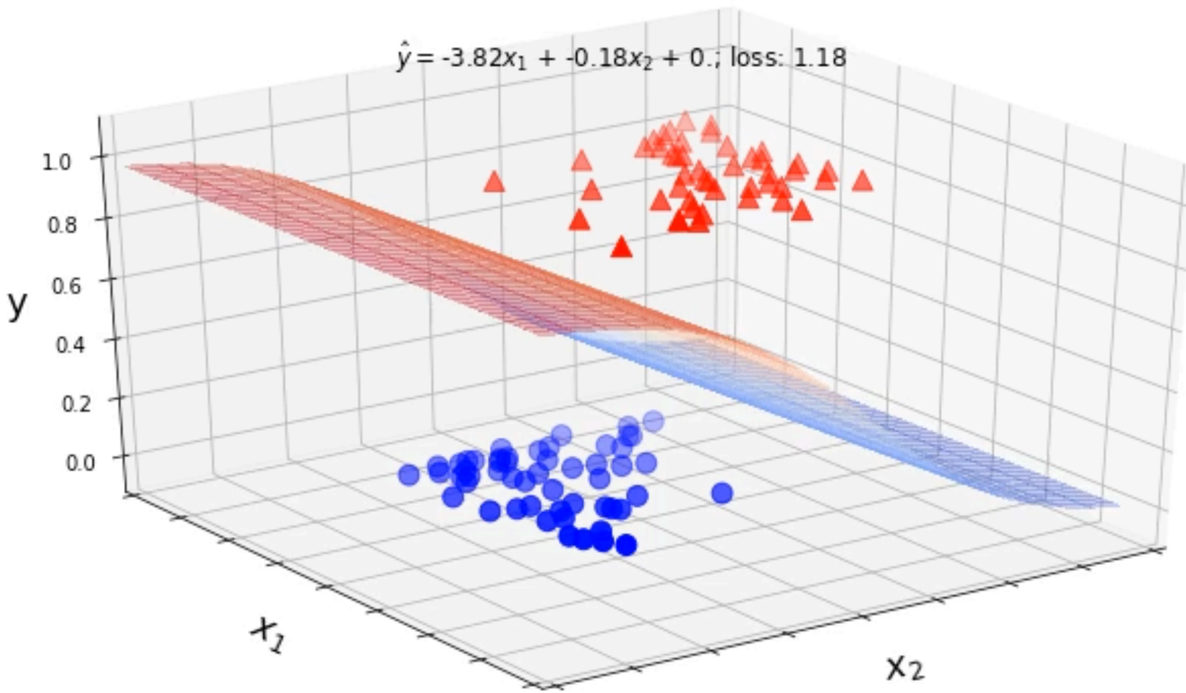
Logistic regression: $f(\boldsymbol{x}; w, b) = \sigma(w^T\boldsymbol{x} + b)$, with parameters w and b

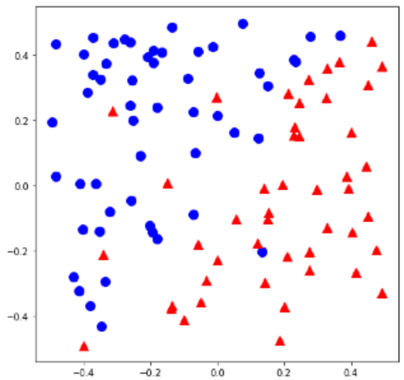
Logistic regression: visualization

Actual data points representing y as a function of $\boldsymbol{x}$ (x_1 and x_2)

● $y = 0$

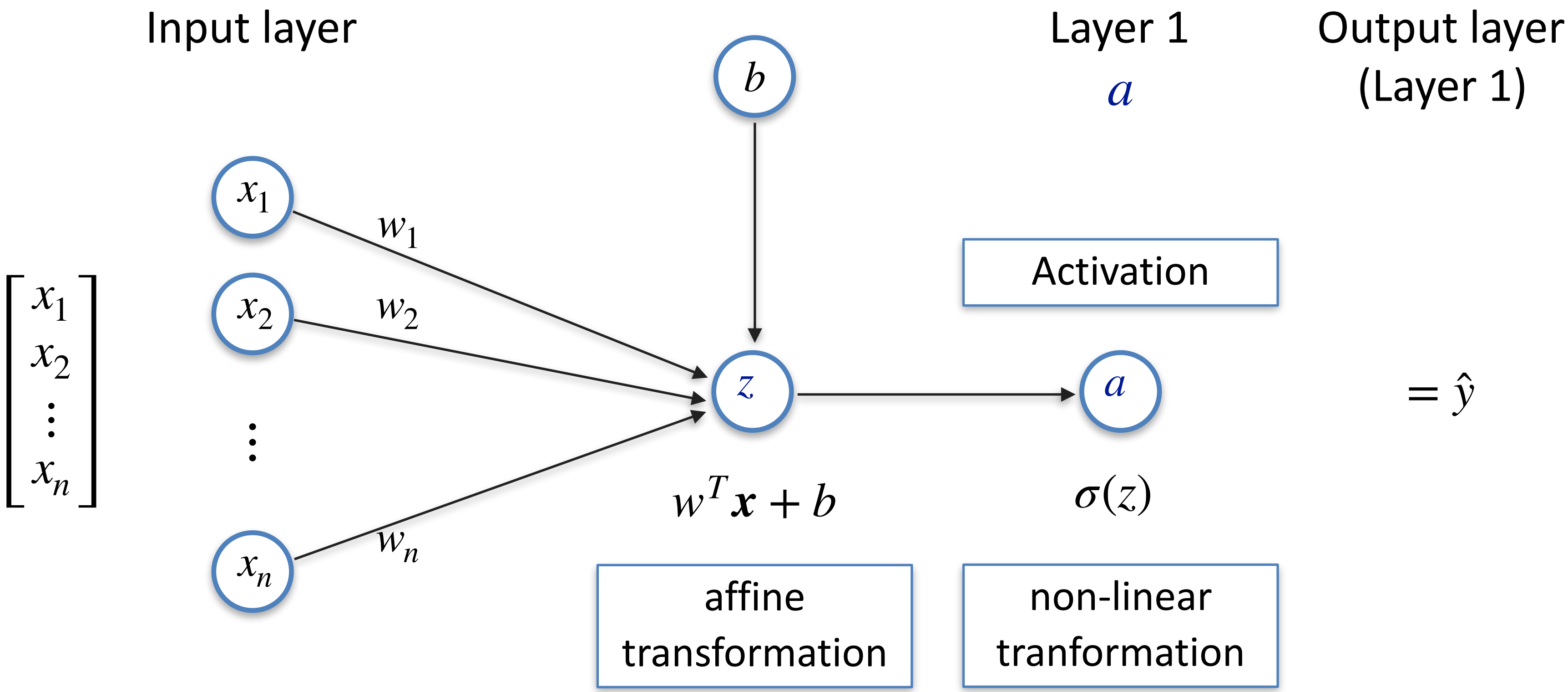
▲ $y = 1$





Logistic regression trying to learn the function as $\hat{y} = \sigma(w_1 x_1 + w_2 x_2 + b)$

Logistic regression as a basic neural network



Each layer: an affine transformation, followed by a typically non-linear activation

Loss function for Logistic Regression

- Goal: no loss for correct prediction
- High loss for incorrect prediction

	Loss
$\hat{y} = y = 1$	Zero
$\hat{y} = y = 0$	Zero
$\hat{y} = 1,\ y = 0$	High
$\hat{y} = 0,\ y = 1$	High

- Convex loss function:

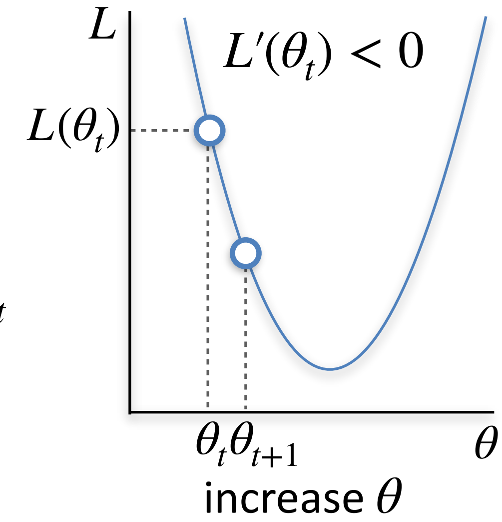
$$L(\hat{y}, y) = -(y \log \hat{y} \quad + \quad (1-y)\log(1-\hat{y}))$$

Case: $y = 0$.
Then, the first term is 0.
If $\hat{y} \sim 0$, then $L \sim \log 1 = 0$.
But if $\hat{y} \sim 1$, $L \sim -\log 0$
(very high)

Case: $y = 1$.
The second term is 0.
If $\hat{y} \sim 1$, then $L \sim \log 1 = 0$.
But if $\hat{y} \sim 0$, $L \sim -\log 0$
(very high)

Learning parameter(s) θ : usually by gradient descent

- Training data: $x^{(1)}, \cdots, x^{(m)}$ and known answers $y^{(1)}, \cdots, y^{(m)}$
- We decide on our function $f(x; \theta)$, then try to learn the *best* parameter θ
- We decide on a *suitable* error / loss function, that depends on θ : $L(\theta)$
- Randomly initialize $\theta = \theta_0$ (better approaches later)
  - $L(\theta_0)$ can be improved (decreased)
- Iteration (t-th) until L is very small (or some other condition):
  - Compute the *gradient* (*derivative*) $L'(\theta_t)$ of L w.r.t. θ at $\theta = \theta_t$
  - If $L'(\theta_t) < 0$, then L is *decreasing* at $\theta = \theta_t$
  - If $L'(\theta_t) > 0$, then L is *increasing* at $\theta = \theta_t$
  - Want to keep decreasing L : go towards the opposite of the gradient
  - Update $\theta_{t+1} = \theta_t - \epsilon L'(\theta_t)$ for some small ϵ (learning rate)



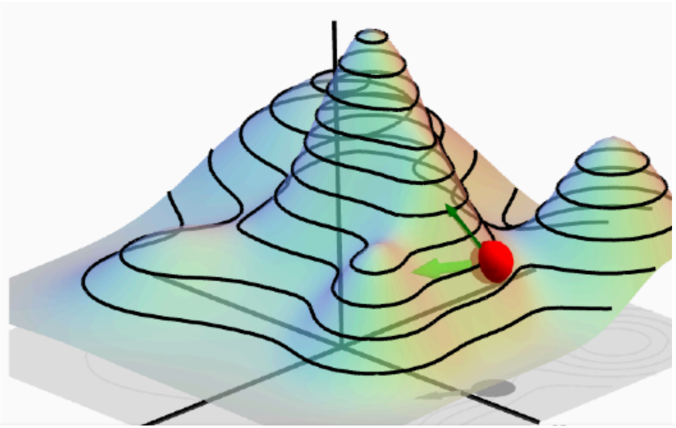
Gradient for multiple parameters

- Suppose $\theta = (\theta_1, \ldots, \theta_k) \in \mathbb{R}^k$
- Then the loss function can be thought of as $L : \mathbb{R}^k \to \mathbb{R}$
- The partial derivative $\dfrac{\partial L}{\partial \theta_i}$ measures how L changes as only θ_i changes
- Gradient $\nabla_\theta L(\theta)$ of L w.r.t. θ is the vector

$$\nabla_\theta L(\theta) = \begin{bmatrix} \dfrac{\partial L}{\partial \theta_1} \\ \vdots \\ \dfrac{\partial L}{\partial \theta_k} \end{bmatrix}$$

Directional Derivatives

- Slope of the function L in the direction of some unit vector u is $\frac{\partial}{\partial\alpha} L(\theta + \alpha\boldsymbol{u})$ evaluated at $\alpha = 0$
- Using chain rule, we get $\frac{\partial}{\partial\alpha} L(\theta + \alpha\boldsymbol{u})\Big|_{\alpha=0} = \frac{\partial}{\partial(\theta + \alpha\boldsymbol{u})} L(\theta + \alpha\boldsymbol{u})\Big|_{\alpha=0} \cdot \frac{\partial}{\partial\alpha}(\theta + \alpha\boldsymbol{u})\Big|_{\alpha=0}$



$$= \boldsymbol{u}^T \nabla_\theta L(\theta)$$

$$= \|\boldsymbol{u}\|_2 \|\nabla_\theta L(\theta)\|_2 \cos\phi_{\boldsymbol{u},\nabla L}$$

$$= \|\nabla_\theta L(\theta)\|_2 \cos\phi_{\boldsymbol{u},\nabla L}$$

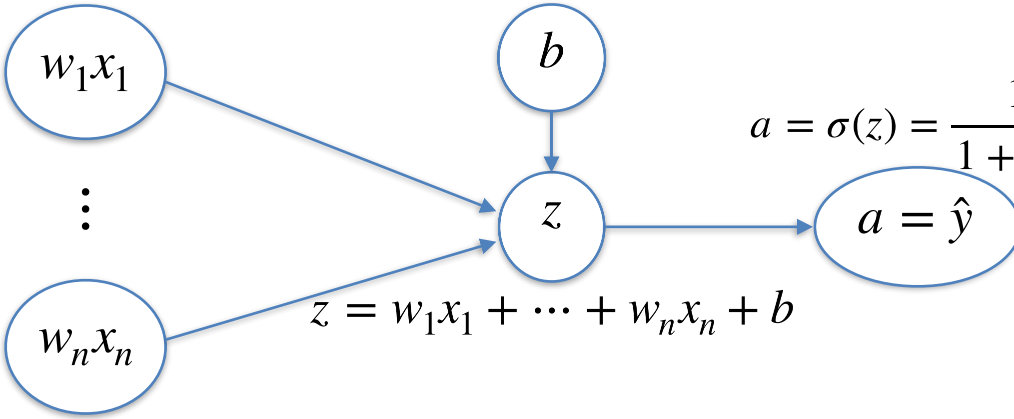
Angle between the vectors

Our goal is to minimize the directional derivative (so that L decreases the fastest)
That happens when $\cos\phi = -1$ ($\boldsymbol{u}$ is in the opposite direction of the gradient $\nabla_\theta L(\theta)$)

Reference for the picture: https://mathinsight.org/directional_derivative_gradient_introduction

Gradient descent for Logistic Regression

Forward propagation



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

$$z = w_1x_1 + \cdots + w_nx_n + b$$

Compute loss

$L(a, y)$

$$L(a, y) = -\left(y \log a + (1-y)\log(1-a)\right)$$

← Backward propagation

$$\frac{\partial L}{\partial w_i} = x_i \frac{dL}{dz} \qquad \frac{\partial L}{\partial b} = \frac{dL}{dz} \qquad \frac{dL}{dz} = \frac{dL}{da}\frac{da}{dz} = a - y \qquad \frac{dL}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

Parameters: $\theta = (w_1, \cdots, w_n, b)$

Gradient descent: compute partial derivatives of L w.r.t. the parameters and keep updating the parameters

Gradient descent iteration:

$$w_i := w_i - \alpha \frac{\partial L}{\partial w_i}; b := b - \alpha \frac{\partial L}{\partial b}$$

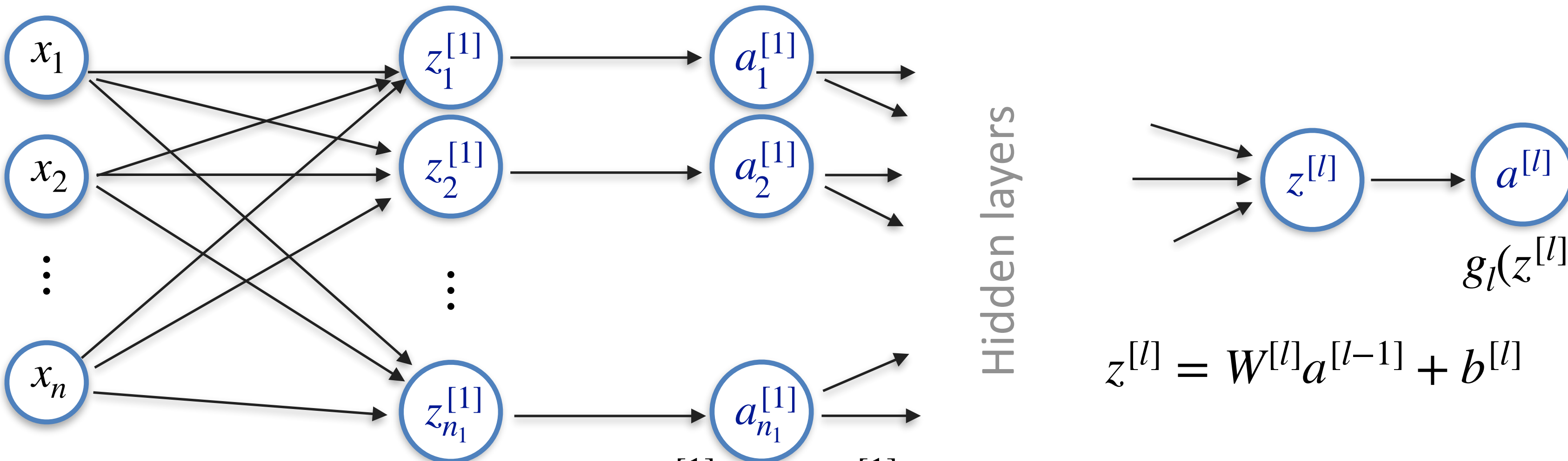
Deep feedforward networks (l - layers)

Input layer 0 $\boldsymbol{a}^{[0]} = \boldsymbol{x}$

Layer 1 $\boldsymbol{z}^{[1]}$

Layer 1 $\boldsymbol{a}^{[1]}$

Output layer l $a^{[l]} = \hat{y}$



$g_l(z^{[l]})$

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$

$$\boldsymbol{z}^{[1]} = W^{[1]}\boldsymbol{a}^{[0]} + \boldsymbol{b}^{[1]}$$

affine transformation

$$a_i^{[1]} = g_1(z_i^{[1]})$$

non-linear activation g_1 (element-wise)

Each layer: an affine transformation, followed by a non-linear activation

Deep feedforward networks (l - layers)

Layer k

Input layer 0

$a^{[0]} = \boldsymbol{x}$

$$\boldsymbol{z}^{[k]} = W^{[k]}\boldsymbol{a}^{[k-1]} + \boldsymbol{b}^{[k]}$$

$$a_i^{[k]} = g_k(z_i^{[k]})$$

Output layer l

$a^{[l]} = \hat{y}$

$n_0 \times 1$

$W^{[k]} : n_k \times n_{k-1}$ $\quad$ $\boldsymbol{z}^{[k]} : n_k \times 1$

$\boldsymbol{b}^{[k]} : n_k \times 1$ $\quad$ $\boldsymbol{a}^{[k]} : n_k \times 1$

1×1

Each layer: an affine transformation, followed by an element-wise non-linear activation

Jacobian: vector / tensor in, vector / tensor out

- Suppose $h : \mathbb{R}^p \rightarrow \mathbb{R}^q$, given as $h(\boldsymbol{x}) = \boldsymbol{y}$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors
- Jacobian of h is defined by the $p \times q$ matrix

$$J_h = \nabla_h \quad = \begin{bmatrix} \dfrac{\partial \boldsymbol{y}}{\partial x_1} & \cdots & \dfrac{\partial \boldsymbol{y}}{\partial x_p} \end{bmatrix} \quad = \begin{bmatrix} \dfrac{\partial y_1}{\partial x_1} & \cdots & \dfrac{\partial y_q}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial y_1}{\partial x_p} & \cdots & \dfrac{\partial y_q}{\partial x_p} \end{bmatrix} \quad = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$$

- Generalized Jacobian: when $h : \mathbb{R}^{p_1 \times \cdots \times p_n} \rightarrow \mathbb{R}^{q_1 \times \cdots q_m}$, given as $h(\boldsymbol{x}) = \boldsymbol{y}$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are *tensors*
- Then the Jacobian is an $(p_1 \times \cdots \times p_n) \times (q_1 \times \cdots \times q_m)$ dimensional tensor, each element being a partial derivative

A good reading material: http://cs231n.stanford.edu/handouts/derivatives.pdf

Backpropagation step: (k-th) layer

Layer k

Input layer 0

$\boldsymbol{a}^{[0]} = \boldsymbol{x}$

$n_0 \times 1$

$$\boldsymbol{z}^{[k]} = W^{[k]}\boldsymbol{a}^{[k-1]} + \boldsymbol{b}^{[k]}$$

$$a_i^{[k]} = g_k(z_i^{[k]})$$

$W^{[k]} : n_k \times n_{k-1}$ $\quad$ $\boldsymbol{z}^{[k]} : n_k \times 1$

$\boldsymbol{b}^{[k]} : n_k \times 1$ $\quad$ $\boldsymbol{a}^{[k]} : n_k \times 1$

Output layer l

$a^{[l]} = \hat{y}$

1×1

$$\frac{\partial L}{\partial \boldsymbol{a}^{[k-1]}} = \frac{\partial L}{\partial \boldsymbol{z}^{[k]}} \cdot \frac{\partial \boldsymbol{z}^{[k]}}{\partial \boldsymbol{a}^{[k]}}$$

$$= (W^{[k]})^T \frac{\partial L}{\partial \boldsymbol{z}^{[k]}}$$

$$\frac{\partial L}{\partial z_i^{[k]}} = g^{[k]\prime}(z_i^{[k]})\frac{\partial L}{\partial a_i^{[k]}}$$

element-wise

$$\frac{\partial L}{\partial W^{[k]}} = \frac{\partial L}{\partial \boldsymbol{z}^{[k]}} \cdot \frac{\partial \boldsymbol{z}^{[k]}}{\partial W^{[k]}} = \frac{\partial L}{\partial \boldsymbol{z}^{[k]}}(\boldsymbol{a}^{[k-1]})^T$$

$$\frac{\partial L}{\partial \boldsymbol{b}^{[k]}} = \frac{\partial L}{\partial \boldsymbol{z}^{[k]}} \cdot \frac{\partial \boldsymbol{z}^{[k]}}{\partial \boldsymbol{b}^{[k]}} = \frac{\partial L}{\partial \boldsymbol{z}^{[k]}}$$

Input

$$\frac{\partial L}{\partial \boldsymbol{a}^{[k]}}$$

Why Non-Linearity?

- Suppose we have a *deep* network, but with only affine transformation (or, activations are identity functions)
- Consider $z^{[2]} = W^{[2]} z^{[1]} + b^{[2]}$
- We have: $a^{[2]} = W^{[2]} a^{[1]} + b^{[2]} = W^{[2]}(W^{[1]} a^{[0]} + b^{[1]}) + b^{[2]}$

$$= W^{[2]} W^{[1]} a^{[0]} + (W^{[2]} b^{[1]} + b^{[2]})$$

Equivalent to a single affine transformation

Recommended for visualization and intuition

[Tensorflow Playground]

References

- Andrew Ng's lectures on *Neural networks and deep learning*: https://www.coursera.org/learn/neural-networks-deep-learning
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. www.deeplearningbook.org
- Justin Johnson. *Derivatives, Backpropagation and Vectorization.* Stanford University (CS231n) Handout, 2017. http://cs231n.stanford.edu/handouts/derivatives.pdf
- Math Insight. *An Introduction to the directional derivative and the gradient.* *https://mathinsight.org/directional_derivative_gradient_introduction*