weight corresponds to the j^{th} node in the i^{th} layer:

$$w_j^{[i]} = [\, w_{j_1}^{[i]},\ w_{j_2}^{[i]},\ \cdots,\ w_{j_{n'}}^{[i]} \,]$$

dimension of the previous layer

weights for the ith layer:

ith layer

$$W^{[i]} = \begin{bmatrix} — w_1^{[i]} — \\ — w_2^{[i]} — \\ \vdots \\ — w_n^{[i]} — \end{bmatrix} \quad (n \times n')$$

dimension of the current layer

$$\therefore \quad z^{[i]} = W^{[i]} x + b^{[i]}$$

$$z^{[i]} = \begin{bmatrix} z_1^{[i]} \\ z_2^{[i]} \\ \vdots \\ z_n^{[i]} \end{bmatrix} \qquad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n'} \end{bmatrix}$$

(single n'-dimensional point)

Multiple Training Examples

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} = A^{[0]}$$

$$Z^{[i]} = W^{[i]} X + b^{[i]}$$

$$Z^{[i]} = \begin{bmatrix} | & | & \\ z^{[i](1)} & z^{[i](2)} & \cdots \\ | & | & \end{bmatrix}$$

$(n \times n')$ $(n' \times m)$ $(n \times 1)$ $(n \times m)$

$$A^{[i]} = \sigma(z^{[i]}) = \begin{bmatrix} | & | & \\ a^{[i](1)} & a^{[i](2)} & \cdots \\ | & | & \end{bmatrix}$$

$(n \times m)$

$$a^{[i](1)} = \begin{bmatrix} a_1^{[i]} \\ a_2^{[i]} \\ \vdots \\ a_n^{[i]} \end{bmatrix}$$ => n hidden states at the i^{th} layer

Activation Functions

great for binary output

sigmoid : $a = \dfrac{1}{1+e^{-z}}$, $[0,1]$

better
(because it centers the data with a zero mean)

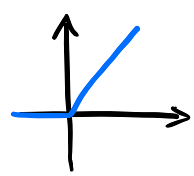
hyperbolic tangent : $a = \tanh(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$, $[-1,1]$

however, the gradient of the above two functions becomes very small when $z \to \infty$ or $-\infty$.

slow down gradient descent

ReLU : $a = \max(0, z)$

most commonly used

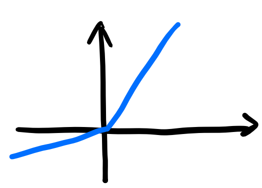


derivative at $z=0$ is not defined, but no decreasing effect in gradient, thus faster training.

Leaky ReLU :

default

$a = \max(0.01z, z)$



Why do we need activation functions?

the NN will become a linear model

if we are performing regression,
we might use linear activation function at the output layer.

Derivatives of Activation Functions

sigmoid: $g(z) = \frac{1}{1+e^{-z}}$

$$g'(z) = g(z)\left(1-g(z)\right)$$

$$\begin{cases} \text{when } z \to \infty \text{ or } -\infty, & g'(z) \to 0 \\ \text{when } z = 0, & g'(z) = \frac{1}{4} \end{cases}$$

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

$$g'(z) = 1 - g(z)^2$$

$$\begin{cases} \text{when } z \to \infty \text{ or } -\infty, & g'(z) \to 0 \\ \text{when } z = 0, & g'(z) = 1 \end{cases}$$

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

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \\ \text{undefined} & z = 0 \end{cases}$$

Leaky ReLU: $g(z) = \max(0.01z, z)$

$$g'(z) = \begin{cases} 0.01 & \text{if } z<0 \\ 1 & \text{if } z \geq 0 \\ \text{undefined} & z=0 \end{cases}$$

Gradient Descent

consider a 2-layer NN

- parameters : $W^{[1]}$ $(n^{[1]}, n^{[0]})$
  $b^{[1]}$ $(n^{[1]}, 1)$
  $W^{[2]}$ $(n^{[2]}, n^{[1]})$
  $b^{[2]}$ $(n^{[2]}, 1)$
- cost function : $J = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y)$

  $a^{[2]}$

Gradient Descent :

Initialize the parameters randomly (not zeros)

repeat { compute predictions $\hat{y}^{[1]}, \hat{y}^{[2]} \ldots, \hat{y}^{[m]}$

for i in 2 : number of layers

$dw^{[i]}, db^{[i]}$

$w^{[i]} = w^{[i]} - \alpha\, dw^{[i]}$

$b^{[i]} = b^{[i]} - \alpha\, db^{[i]}$ }

Forward Propagation :

$Z^{[1]} = W^{[1]} X + b^{[1]}$

$A^{[1]} = g(Z^{[1]})$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = g(Z^{[2]})$$

Back Propagation:

$$dZ^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} \cdot A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} \text{np.sum}(dZ^{[2]}, \text{axis}=1, \text{keepdims}=\text{True})$$

summing horizontally

$$dZ^{[1]} = \underbrace{W^{[2]T} \cdot dZ^{[2]}}_{n^{[1]} \times m} * \underbrace{g^{[1]\prime}(Z^{[1]})}_{n^{[1]} \times m}$$

element-wise

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

$$db^{[1]} = \frac{1}{m} \text{np.sum}(dZ^{[1]}, \text{axis}=1, \text{keepdims}=\text{True})$$

Random Initialization

if the weights are initialized to zeros, the hidden states become symmetric and the update in gradient descent becomes identical for all weights.

(making no difference)

$$W^{[i]} = \text{np.random.randn}((n^{[i]}, n^{[i-1]})) \times 0.01$$

$$b^{[i]} = \text{np.zeros}((n^{[i]}, 1))$$

we can
use zeros for b

we want
small initial
values.

if weights are large
we will have very small gradients
(when we use sigmoid and tanh)
which will slow down the training.