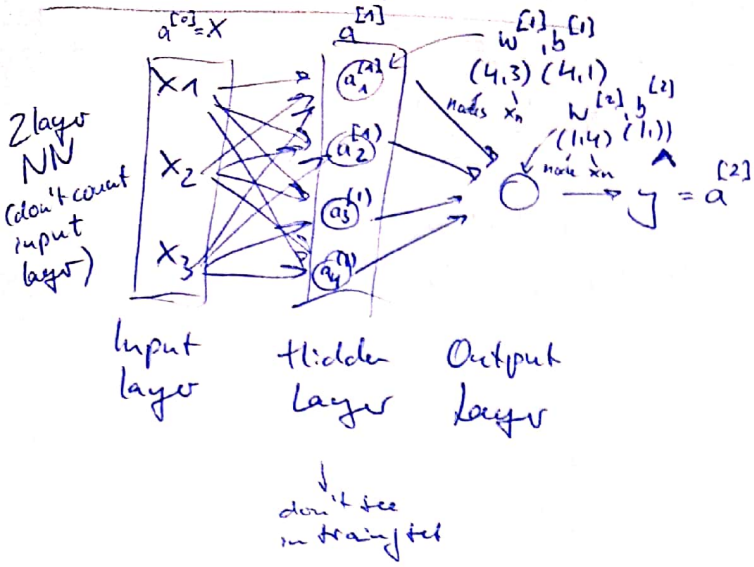
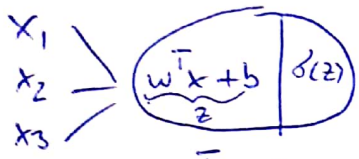
DL Spec | week 3 | course 1

Neural Network Representation

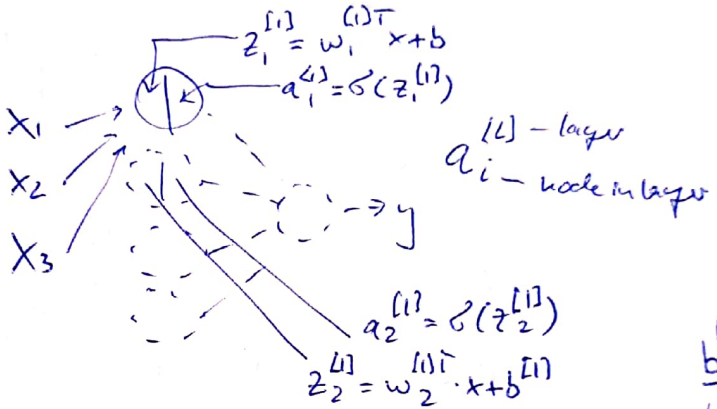


$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ \vdots \\ a_4^{[1]} \end{bmatrix} \quad a^{[1]} \in \mathbb{R}^4$$

Computing NN's output



$$z = w^T \cdot x + b$$
$$a = \sigma(z)$$



$a_i^{[l]}$ – l – layer, i – node in layer

calculate $z_1^{[1]} = \dots$, $z_2^{[1]} = \dots$, $z_3^{[1]} = \dots$, $z_4^{[1]} = \dots$ in for loop very unefficient!

Use Vectorization!

$$W^{[1]} = \begin{bmatrix} - w_1^{[1]T} - \\ - w_2^{[1]T} - \\ - w_3^{[1]T} - \\ - w_4^{[1]T} - \end{bmatrix} \in \mathbb{R}^4 \quad \text{shape } (4,3)$$

$$\cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}}_{b^{[1]}} = \underbrace{\begin{bmatrix} w_1^{[1]T} \cdot x + b_1^{[1]} \\ w_2^{[1]T} \cdot x + b_2^{[1]} \\ w_3^{[1]T} \cdot x + b_3^{[1]} \\ w_4^{[1]T} \cdot x + b_4^{[1]} \end{bmatrix}}_{(4,1)} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = z^{[1]}$$

$$\hookrightarrow a^{[1]} = \begin{Bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{Bmatrix} = \sigma(z^{[1]})$$

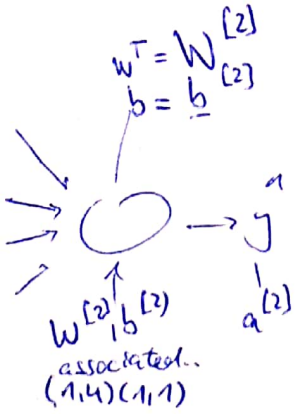
Given input x:

$$z^{[1]} = W^{[1]} x + b^{[1]}$$
(4,1) (4,3) (3,1) (4,1)

$$a^{[1]} = \sigma(z^{[1]})$$
(4,1) (4,1)

$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$
(1,1) (1,4) (4,1) (1,1)

$$a^{[2]} = \sigma(z^{[2]})$$
(1,1) (1,1)



↳ 4 lines of code to compute output

Vectorizing across multiple examples

$x \longrightarrow a^{[2]} = \hat{y}$

$x^{(1)} \longrightarrow a^{[2](1)} = \hat{y}^{(1)}$

$x^{(2)} \longrightarrow a^{[2](2)} = \hat{y}^{(2)}$

$\vdots$

$x^{(m)} \longrightarrow a^{[2](m)} = \hat{y}^{(m)}$

$a^{[2](i)}$: [2] = layer 2, (i) = i-th training example

to compute for multiple examples:

for i = 1 to m $\left(\sum_{i=1}\right)$

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

→ want to vectorize:

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad (n_x, m)$$

(rows: n_x features; columns: m examples)

Vectorized implementation

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

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

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

(columns: m examples; rows: corresponds to node in NN (hidden units))

Justification for vectorized implementation:

$W^{[1]} = \begin{bmatrix} \equiv \end{bmatrix}$ $\quad W^{[1]} x^{(1)} = \begin{bmatrix} \vdots \end{bmatrix}$ $\quad W^{[1]} x^{(2)} = \begin{bmatrix} \vdots \end{bmatrix}$ $\quad W^{[1]} x^{(3)} = \begin{bmatrix} \vdots \end{bmatrix}$

$$W^{[1]} \cdot \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = Z^{[1]}$$

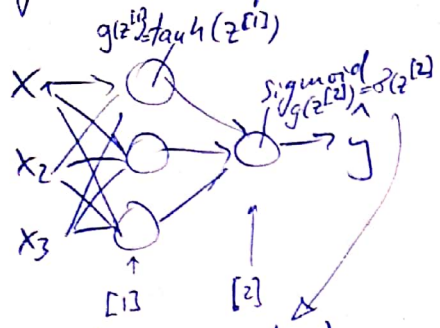
(the matrix $\begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} \end{bmatrix}$ is X)

$W^{[1]} x^{(i)} = z^{[1](i)}$ $\quad +b^{[1]} \quad +b^{[1]} \quad +b^{[1]}$ → done correctly with Python broadcasting

DL Spec Week 3 course 1

Activation functions

When building NN architecture you need to chose Activation functions for hidden units and output unit



Almost always use tanh

exception: binary classification
↳ output unit should give $\hat{y} \in \{0, 1\}$ rather than $-1 < y < 1$
→ Sigmoid!

So far:

Given x:

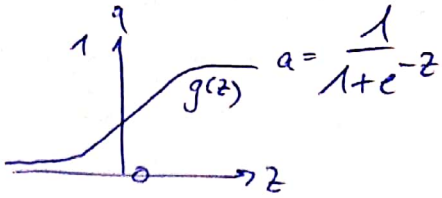
$z^{[1]} = W^{[1]} x + b^{[1]}$

$a^{[1]} = g(z^{[1]}) = g(z^{[1]})$

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

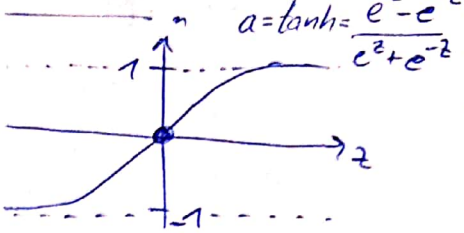
$a^{[2]} = g(z^{[2]}) = g(z^{[2]})$

Sigmoid



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

Tanh (shifted version of Sigmoid)



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

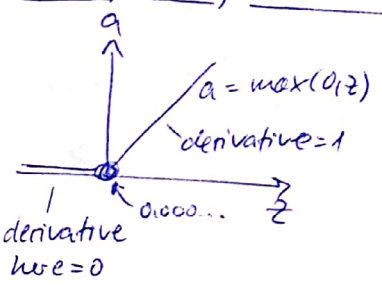
(almost always work better than Sigmoid)
values between -1 and 1
the mean of the activation is close to 0.
When learning an algorithm you want to center the data

Centering
↳ mean close to 0, rather than 0.5 (Sigmoid) makes learning for next layer easier

One downsides for Sigmoid and tanh:

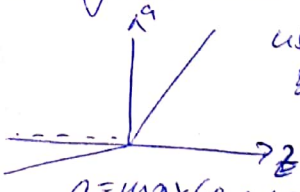
- If z is either very large or very small the gradient is really small! Slope of function is close to zero → slows down Gradient Descent

ReLU - Rectified Linear Unit



→ disadvantage that derivative is 0 when z is negative

Leaky ReLU



usually works better but not often used in practice

works better →

$a = \max(0.01 \cdot z, z)$
↑ can also be used as parameter

common advantage: for a large space of z the derivative is very different from 0
↳ NN will learn much faster

Pros and Cons of activation functions

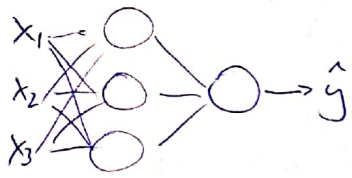
Sigmoid: never use this except for output layer when doing binary classification

tanh: superior to sigmoid

new default: ReLU

or Leaky ReLU

Why do we need non-linear activation functions?



↳ Neural Network would just compute a linear response of the input no matter how many hidden layers we have

A linear hidden layer is useless!

Given x:

$z^{[1]} = W^{[1]} x + b^{[1]}$

$a^{[1]} = g^{[1]}(z^{[1]}) \rightarrow z^{[1]}$ → $g(z) = z$ would be called "linear activation function" or "identity function"

$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$

$a^{[2]} = g^{[2]}(z^{[2]}) \rightarrow z^{[2]}$

Exception:

ML in Regression Problem

e.g. Housing Prices

where $y \in \mathbb{R}$

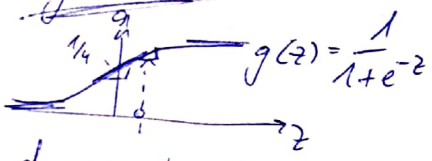
$0 ..... $ 100'000 .....

→ output $\hat{y}$ will be $\mathbb{R}$

output can be linear function

Derivatives of activation functions

Sigmoid



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

$\frac{d}{dz} g(z)$ = slope of $g(z)$ at z

$g'(z) = \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right)$

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

with $a = g(z)$

↳ $g'(z) = a(1-a)$

Sanity Check:

when z is large

→ $\frac{dg(z)}{dz} \approx 1(1-1) = 0$ ✓

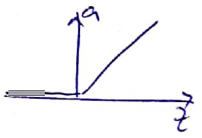
when z is small

$\frac{dg(z)}{dz} \approx 0 \cdot (1-0) \approx 0$ ✓

when $z = 0$

$\frac{dg(z)}{dz} = \frac{1}{2}\left(1 - \frac{1}{2}\right) = \frac{1}{4}$

ReLU

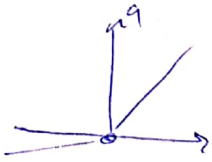


$g(z) = \max(0, z)$

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

(undefined if $z = 0$)

Leaky ReLU

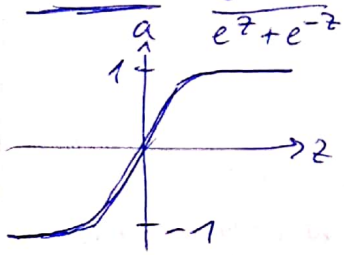


$g(z) = \max(0.01 \cdot z, z)$

$g'(z) = \begin{cases} 0{,}01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$

tanh

$= \frac{e^z - e^{-z}}{e^z + e^{-z}}$



$\frac{dg(z)}{dz} = 1 - (\tanh(z))^2$

Sanity Check

when z is large

$\tanh(z) \approx 1$ ✓ $g'(z) \approx 0$ ✓

when z is small

$\tanh(z) \approx -1$ $g'(z) \approx 0$ ✓

when $z = 0$

$\tanh(z) = 0$

$g'(z) = 1$

$a = g(z)$

↳ $g'(z) = 1 - a^2$

DL spec \week3\ course II

Gradient Descent for Neural Networks

Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$ $\quad n_x = n^{[0]}, n^{[1]}, n^{[2]} = 1$

Dimensions: $(n^{[1]}, n^{[0]})$ | $(n^{[2]}, 1)$ | $(n^{[1]}, 1)$ | $(n^{[2]}, n^{[1]})$

n_x = input features; $n^{[1]}$ = hidden; $n^{[2]}$ = output

Cost function: (binary classification)

$$J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$$

$\hat{y} = a^{[2]}$

to train parameters: Gradient Descent

- Initialize parameters randomly!
- Compute predictions $(\hat{y}^{(1)}, \dots \hat{y}^{(m)})$
- derivatives $dW^{[1]} = \frac{\partial J}{\partial W^{[1]}}$, $db^{[1]} = \frac{\partial J}{\partial b^{[1]}}$, ...
- GD update =

$$W^{[1]} = W^{[1]} - \alpha\, dW^{[1]}$$
$$b^{[1]} = b^{[1]} - \alpha\, db^{[1]}$$

⋮

Forward propagation

$$z^{[1]} = W^{[1]} X + b^{[1]}$$
$$A^{[1]} = g^{[1]}(z^{[1]})$$
$$z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(z^{[2]}) = \sigma(z^{[2]})$$

can be Sigmoid here...

↳ obtain loss $\mathcal{L}(A^{[2]}, y)$

Back propagation:

$$dz^{[2]} = A^{[2]} - Y \qquad Y = [y^{(1)} \dots y^{(m)}]$$
$$dW^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} \text{np.sum}(dz^{[2]}, \text{axis}=1, \text{keepdims}=\text{True})$$

sum up horizontally; prevents Python from "rank-1-array"; can also be done with reshape!

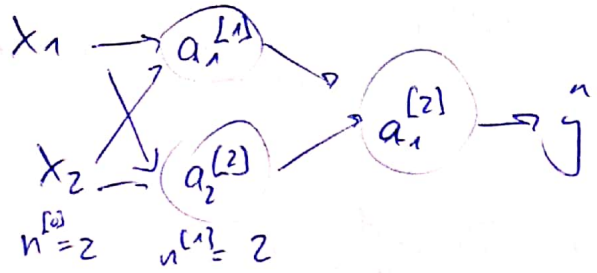
$$dz^{[1]} = \underbrace{W^{[2]T} dz^{[2]}}_{(n^{[1]}, m)} * \underbrace{g'^{[1]}(z^{[1]})}_{(n^{[1]}, m)}$$

element wise product of 2 matrices; derivative

$$dW^{[1]} = \frac{1}{m} dz^{[1]} X^T$$
$$db^{[1]} = \frac{1}{m} \text{np.sum}(dz^{[1]}, \text{axis}=1, \text{keepdims}=\text{True})$$
$(n^{[1]}, 1)$ ← ! avoid "rank-1-arrays!"

Random initialization

→ initialize parametrs (w, b) randomly!



$$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$(2,2)$ ↑ problem! ↑ Ok!

→ $a_1^{[1]}$ will be equal to $a_2^{[1]}$!

$a_1^{[1]} = a_2^{[1]}$ 😐, so $dz_1^{[1]} = dz_2^{[1]}$ 😐

→ hidden units in 1st layer are "symmetric" → compute exactly the same function!

after iterations still compute the same function

$$dW = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$$

when perform weight update: $W^{[1]} = W^{[1]} - \alpha\, dW$

→ $W^{[1]} = \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \end{bmatrix}$ } 1st row equal to 2nd row

Solution: Initialize W randomly!

$W^{[1]} = np.random.randn((2,2)) * 0.01$

initialize to small values

$b^{[1]} = np.zeros((2,1))$

$W^{[2]} = np.random.randn((1,2)) * 0.01$

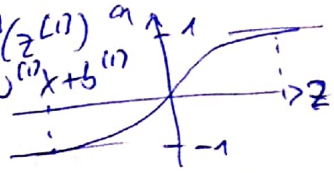
$b^{[2]} = 0$

Why use 0.01 and not 1000?

→ for activations z will be large!

$a^{[1]} = g^{[1]}(z^{[1]})$

$z^{[1]} = W^{[1]}x + b^{[1]}$



↳ we will end up at flat parts of tanh or sigmoid and have low slope and therefore slow convergence for Gradient Descent