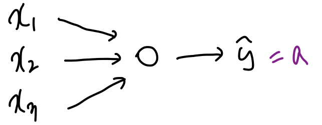
Neural Network Overview



≒

input x, w, b → compute z using compute a ($\hat{y}$)

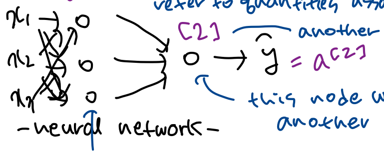
x
w → dw
b → db

✓ dz: $z = w^T x + b$ first calculation. → ✓ da: $a = \sigma(x)$ Second. → $L(a, y)$ finally we can get lost function. (L)

In logistic regression, do backward calculate in order to compute derivaties.

Can form neural networks by stacking together up all of sigmoid units.

square bracket 1 [1] to refer to quantities associated with this stack of node layer.



[2] another layer.

this node will correspond another z and another a

this stack of nodes will calculate a, z like that ✓

(not to be confused [ ] with ( )) — bracket 1 and 2 of this different layers. (layer 1, layer 2); ( ) individual training example.

($x^{(i)}$: ith training example.)

x, $w^{[1]}$, $b^{[1]}$ →

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

↓↑

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

↓↑ $w^{[2]}$ ($dw^{[2]}$), $b^{[2]}$ ($db^{[2]}$)

$z^{[2]} = w^{[2]} x + b^{[2]}$ $dz^{[2]}$

↓↑

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

↓↑

$L(a^{(2)}, y)$

Right to left backward calculation.

In logistic regression, we have z followed by a calculation in the neural network, we just do it multiple times and finally count a loss.

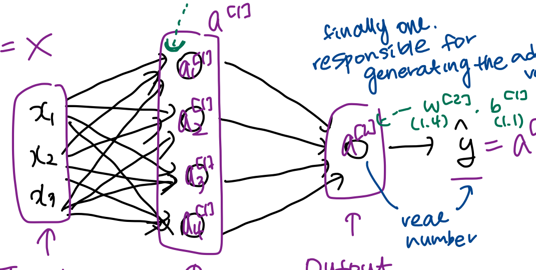
Neural Network Representation.

4 hidden unites. input features.
(4.3) (4.1)

associate with parameters $w^{[1]}$ and $b^{[1]}$

indicate what layer $a^{[0]} = X$

$a^{[0]}$ also means the activations of the info layer in superscript 0



finally one. responsible for generating the addict value $\hat{y}$.

we only had one output layer → don't use [ ]

"$\hat{y} = a$" in logistic regression.

≒

four dimensional vector ← in this case, we have 4 nodes unites hidden unites in this hidden layer.

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$

using neural network supervisor learning, the training set contains value of the inputs X and the target output Y.

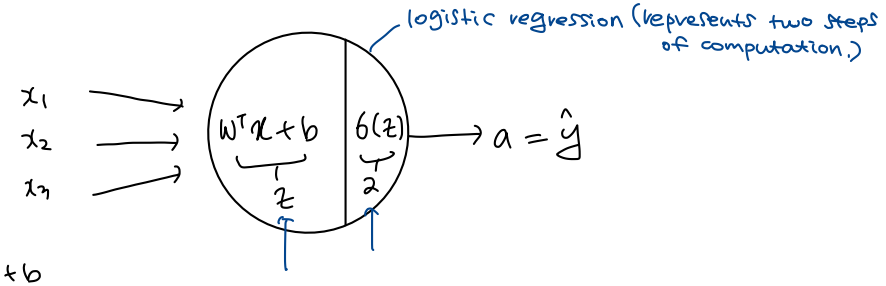
just the name hidden means: you can't see it in the training set.

⇒ You can see input X output Y but can't hidden layer.

< 2 Layer Neural Network > with 3 layers ≥ <

don't count the input layer.

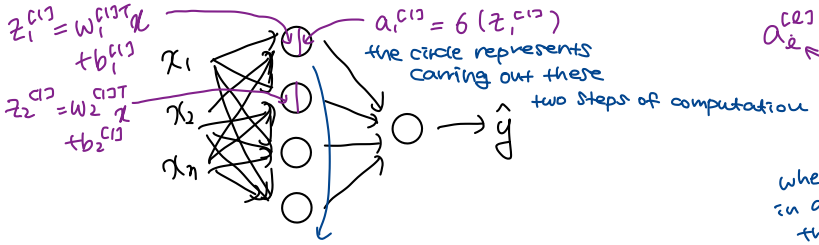
Computing Neural Network Output



$z = w^T x + b$

$a = \sigma(z)$

In neural network, just do this a more time.



$a_i^{[1]} = \sigma(z_i^{[1]})$ the circle represents carring out these two steps of computation.

$a_i^{[2]}$ ← layer number

↑ node in layer

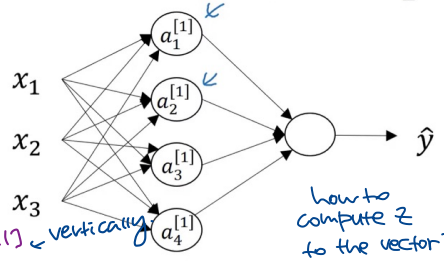
also represents similar computation.

when different nodes in a layer, we'll stack them vertically.

for loop → inefficient.

⇒ vectorize !!

Neural Network Representation



$z^{[1]}$ ← vertically

How to compute z to the vector?

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

$a^{[1]}$

$$= \begin{bmatrix} - & w_1^{[1]T} & - \\ - & w_2^{[1]T} & - \\ - & w_3^{[1]T} & - \\ - & w_4^{[1]T} & - \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

(4, 3) → $W^{[1]}$

Stacking 4 w vectors together → end up with matrix.

If regression units. each has corresponding parameter vector w and by stacking:

$b^{[1]}$ correspond with each of these rows.

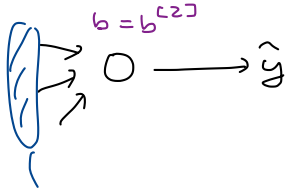
$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ \vdots \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

↑ sigmoid function that takes 4 elements of zs and applies the sigmoid function element wise.

In the output layer, there are parameter $w^{[2]}$, $b^{[2]}$ (1, 4) (1, 1)

$w^T = w^{[2]}$

$b = b^{[2]}$



ignoring all fronts, then this is (this last output unit) is a lot like logistic regression

Given input x:

$$z^{[1]}_{(4,1)} = w^{[1]}_{(4,3)} x_{(3,1)} + b^{[1]}_{(4,1)}$$

$a^{[0]}$

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

remember $x = a^{[0]}$, $\hat{y} = a^{[2]}$

first & hidden layer.

representation of next layer is similar.

$$z^{[2]}_{(1,1)} = w^{[2]}_{(1,4)} x_{(4,1)} + b^{[2]}_{(1,1)}$$

real number

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

Just a real number.

output layer.

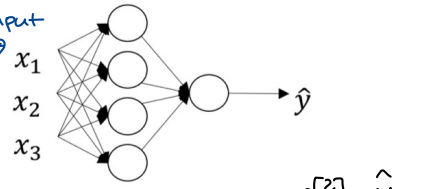
Vectorized implementation of computing

for computing this neural network, you just need these 4 lines of code !).

Vectorizing Across Multiple Examples.

How to vectorize multiple examples? Outcome would be quite similar with Logistic Regression. Stacking up training examples in the different columns of matrices.

given an input vector $x \Rightarrow$



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

use them to generate $a^{[2]} = \hat{y}$ for single train example.

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

for these 4 eq on top and adding the superscript round bracket i to all the variables that depend on the training example.

for i=1 to m,

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

Vectorize to remove this for loop..

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

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

$\vdots$

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

$a^{[2](i)}$: example i, layer 2

have to compute

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

for i = 1 to m:

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

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

we went from lowercase vector xs to this capital case X matrix by stacking up the lower case x's in different columns.

hidden units ↕ training examples

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

horizontally we're going to index across training examples - corresponds to different training examples.

activation of the first hidden unit on the first training ex.

activation in the second hidden unit on the first training ex...

lowercase ⇓ capital. → $A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$ hidden units

⇒ similarly $Z^{[2]}$ & $A^{[2]}$

vertically, this vertical index corresponds to different notes in neural network.

Explanation for Vectorized Implementation.

<Justification for vectorized implementation>

(for simplification)

$z^{[1](1)} = W^{[1]}x^{(1)} + b^{[1]}$, $z^{[1](2)} = W^{[1]}x^{(2)} + b^{[1]}$, $z^{[1](3)} = W^{[1]}x^{(3)} + b^{[1]}$

$W^{[1]} = \begin{bmatrix} \text{---} \\ \text{---} \\ \text{---} \end{bmatrix}$ $W^{[1]}x^{(1)} = \begin{bmatrix} | \\ | \\ | \end{bmatrix}$ $W^{[1]}x^{(2)} = \begin{bmatrix} | \\ | \\ | \end{bmatrix}$ $W^{[1]}x^{(3)} = \begin{bmatrix} | \\ | \\ | \end{bmatrix}$

Column vector. ↳ would be another column vector ($z^{[1](2)}$)

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

X more stack when training sample is more.

featuring examples - you have more examples and dug in more columns

$+b^{[1]}$ $+b^{[1]}$ $+b^{[1]}$

with python broadcasting, add $b^{[1]}$ individually.

$W^{[1]}x^{(i)} = z^{[1](i)}$ $= Z^{[1]}$

Stack them up in different columns → corresponding result is that you end up with the z also stack in columns.

can convince yourself if you want. Stuff with python broadcast if you add back in these values of b the values are still correct.

$Z^{[1]} = W^{[1]}X + b^{[1]}$: correct vectorization in the first step of the 4 steps in the previous slide.

input columns → output also stacked up in columns.

for i = 1 to m:

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

한 번에 하나의 훈련 샘플에 대해 정방향 전파를 한다면 1개부터 m개까지 모든 실행해야 함 ⇒ 훈련 샘플을 열로 쌓아서 여기 있는 값들 모두 열을 쌓음

help vectorize $W^{[1]}A^{[0]} + b^{[1]}$

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

equal to $A^{[0]}$ input feature vector

$X = a^{[0]}$

$x^{(i)} = a^{[0](i)}$

Symmetric ⇒ doing the same thing / same computation over and over

Activation Functions

what activation functions to use in hidden layers and output unit.

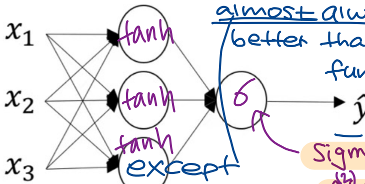
Sigmoid – Sometimes, other function is better.

Activation functions

indicate $g^{(1)} \neq g^{(2)}$

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

almost always work better than sigmoid function cause ±1 → mean of the activation closer to 0.



sigmoid except output layer $g^{[2]}(z^{[2]}) = \sigma(z^{[2]})$

$y \in \{0, 1\}$

Given x: $0 \le \hat{y} \le 1$

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

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

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

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

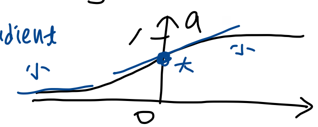
① → ② → use sigmoid function ⇓ called "activation function"

g could be a nonlinear function that may not be sigmoid function.

Andrew Ng

<Sigmoid function>

range: 0 ~ 1.

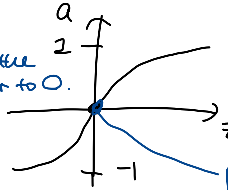


$a = \frac{1}{1+e^{-z}}$

✓ <tanh function>

almost work better than sigmoid function

hyperbolic tangent function.



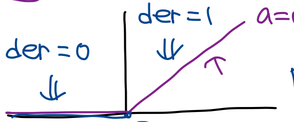
$a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

pass (0,0)

range: -1 ~ +1

<Relu function>

Rectified Linear Unit.



$a = \max(0, z)$

pretend 0 or 1 → work fine

↖ z = 0.00000000...

미분불가능! 상관없어

you might center data and have your data has 0 mean using a tanh instead of a sigmoid function

- Centering your data → the mean of the data is close to the zero
  → actually makes learning for the next layer a little bit easier.

- one exception we use sigmoid function: binary classification.

→ tanh for hidden layer, sigmoid for output layer.

& Relu for all the Rectified Linear Unit.

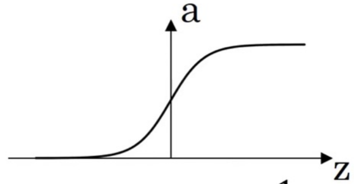
⇒ Not sure what to use in hidden unit, just use "Relu function."

함수의 기울기가 0에 가까워지면 학습이 느려짐
Relu는 기울기가 0에 가까워지는 것을 막으므로 다른 활성화 함수보다 빠르다.

Leaky Relu — work better, but not used many practice.

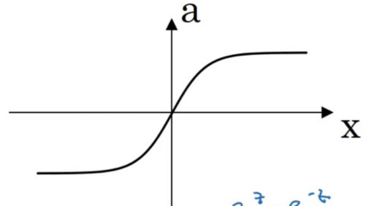
neural network would be much faster when we use Relu than using sigmoid or tanh. (∵ most part are not 0)

Pros and cons of activation functions



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

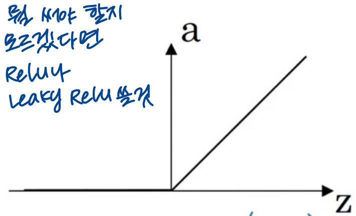
never use except for "Output Layer" (binary classification)



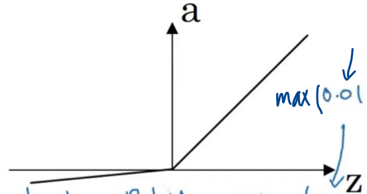
tanh: $a = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

pretty much better than sigmoid

뭘 써야 할지 모르겠다면 Relu나 Leaky Relu 쓸것



ReLU default. $a = \max(0, z)$



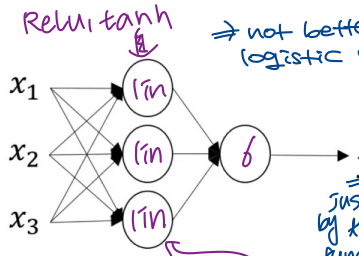
Leaky ReLU $a = \max(0.01z, z)$

Andrew Ng

when you don't know which to use → try it all and see!!

Why Non-Linear Activation functions

Activation functions



⇒ not better than logistic regression.

just compute by the linear function about the input x.

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

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

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

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

$= W'x + b'$ ($W' = W^{[2]}W^{[1]}$, $b' = W^{[2]}b^{[1]} + b^{[2]}$)

<prop Equations>

Given x:

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

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

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

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

linear hidden units are more useless.

∵ composition of 2 linear functions → itself a linear function.

⇒ ∵ not using non-linearity function, you can't compute more ♡ even as you go deeper.

$g(z) = z$

"Linear Equation functions"

≒ identity activation functions.

Andrew Ng

if you use a linear activation function or alternatively don't have an activation function, then no matter how many layers your neural network has, all those doing is just computing a linear activation function. ≒ not have any hidden layer

$g(z) = z$ is useful when you do machine learning regression program.

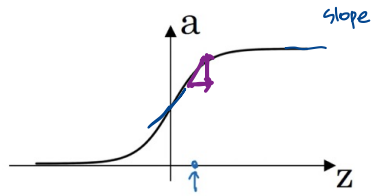
↗ only used at out layer.

It might be okay to use linear activation when $y \in \mathbb{R}$ and $\hat{y} \in \mathbb{R}$

But the hidden unit, don't use linear function and use Relu, tahn, leaky ReLu or other non-linear function.

Derivatives of Activation functions.

Sigmoid activation function



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

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

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

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

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

$= a(1-a)$

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

advantage: if you already know a, you can know $g'(z)$ (slope of $g(z)$) easily.

i) $z = 10$. $g(z) \approx 1$

$\frac{d}{dz}g(z) \approx 1(1-1) \approx 0$

ii) $z = -10$. $g(z) \approx 0$

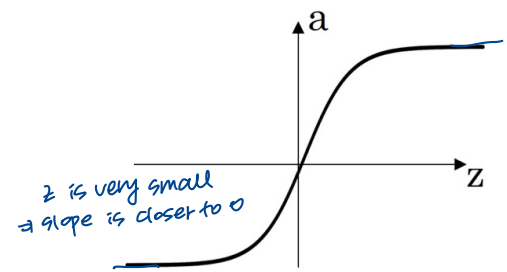
$\frac{d}{dz}g(z) \approx 0 \cdot (1-0) \approx 0$

iii) $z = 0$. $g(z) = \frac{1}{2}$

$\frac{d}{dz}g(z) = \frac{1}{2}(1-\frac{1}{2}) = \frac{1}{4}$

Andrew Ng

Tanh activation function



$g(z) = \tanh(z)$

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

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

$= 1 - (\tanh(z))^2$

$a = g(z)$, $g'(z) = 1 - a^2$ → quickly calculate

i) $z = 10$ $\tanh(z) \approx 1$

$g'(z) \approx 0$

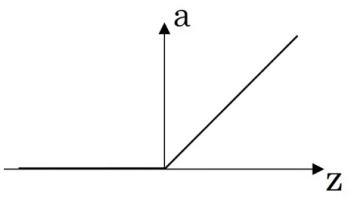
ii) $z = -10$ $\tanh(z) \approx -1$

$g'(z) \approx -1$

iii) $z = 0$ $\tanh(z) = 0$

$g'(z) = 1$

ReLU and Leaky ReLU



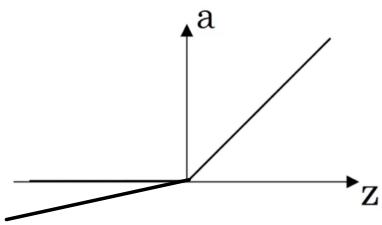
ReLU

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

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

~~undefined if $z = 0$~~

doesn't matter to think $z=0$ ..... this value to 0 or 1. (either.. 어차피 상관없다)



Leaky ReLU

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

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

no matter to set either 0 or 1.

Andrew Ng

Gradient Descent for Neural Networks

parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$

$(n^{[1]}, n^{[0]})\ (n^{[2]}, 1)\ (n^{[2]}, n^{[1]})\ (n^{[1]}, 1)$

$n_x = n^{[0]}$ (input), $n^{[1]}$ (hidden), $n^{[2]} = 1$ (output) — example.

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

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

Gradient descent:

initialize the parameters randomly rather than to all 0 ≠ not to 0 !!

Repeat {

Compute Predicts $(\hat{y}^{(i)},\ i = 1 \cdots m)$

$dW^{[1]} = \frac{dJ}{dW^{[1]}}$, $db^{[1]} = \frac{dJ}{db^{[1]}}$, ... same with $W^{[2]}$ and $b^{[2]}$

$W^{[1]} = W^{[1]} - \alpha \cdot dW^{[1]}$

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

⋮ same with $W^{[2]}, b^{[2]}$

repeat this until parameters converges.

}

$Y = [y^{(1)}\ y^{(2)} \cdots y^{(m)}]$

< Formulas For Computing Derivatives >

Forward Propagation:

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

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

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

$A^{[2]} = g^{[2]}(Z^{[1]}) = \sigma(Z^{[2]})$

binary classification in here..

Back Propagation:

$dZ^{[2]} = A^{[2]} - Y$ — true value, $(m, 1)$ vector

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

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

keepdims = True: Sum across one dimension of a matrix → this case, sum horizontally

잘못된 1차원 배열 출력방지

$(n,)$ True → 출력값 $(n, 1)$ 바뀜

$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$

$(n^{[1]}, m)$ element-wise product $(n^{[1]}, m)$

derivative of activation function used in hidden layer.

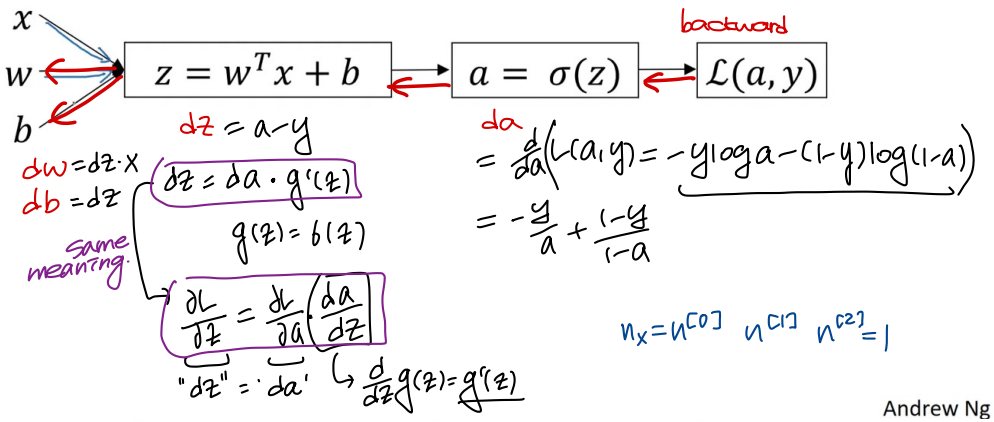
$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$

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

Backpropagation Intuition

Computing gradients

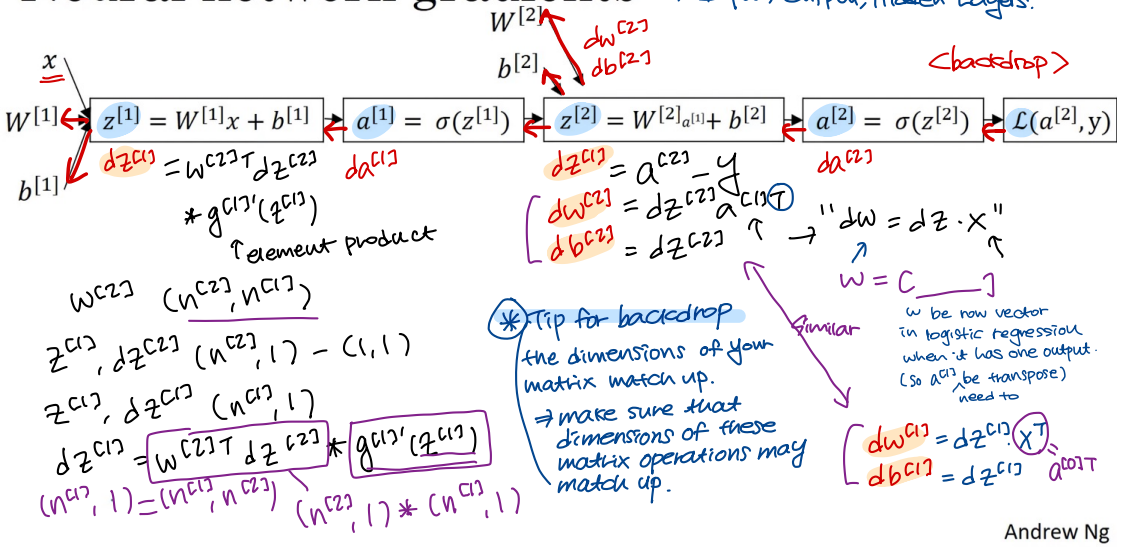
Logistic regression



Andrew Ng

Neural network gradients

Neural Network with 2 Layers
⇒ Input, Output, Hidden Layers.



Andrew Ng

<Summary of gradient descent>

$dz^{[2]} = a^{[2]} - y$
$dw^{[2]} = dz^{[2]} a^{[1]T}$
$db^{[2]} = dz^{[2]}$
$dz^{[1]} = w^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$
$dw^{[1]} = dz^{[1]} x^T$
$db^{[1]} = dz^{[1]}$

— Vectorizal Implemetation —

$z^{[1]} = w^{[1]} x + b^{[1]}$
$a^{[1]} = g^{[1]}(z^{[1]})$

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

capital $\begin{cases} Z^{[1]} = w^{[1]} X + b^{[1]} \\ A^{[1]} = g^{[1]}(z^{[1]}) \end{cases}$

$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$
$db^{[2]} = \frac{1}{m}$ np.sum($dZ^{[2]}$, axis=1, keepdims=True)
$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]})$
$(n^{[1]}, m)$ $(n^{[1]}, m)$ $(n^{[1]}, m)$
elemental product.
$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
$db^{[1]} = \frac{1}{m}$ np.sum($dZ^{[1]}$, axis=1, keepdims=True)

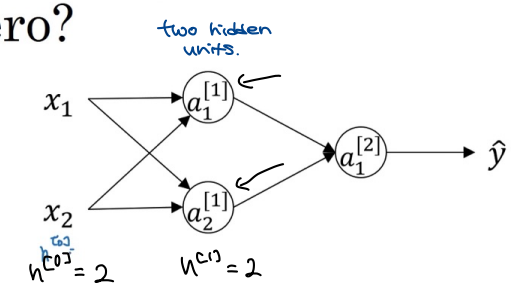
$J = (\cdots) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$

Random Initialization

It's important to initialize number randomly.

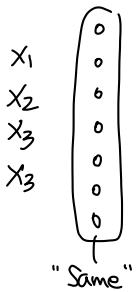
In Logistic Regression, It is okay to initialize 0. But for Neural Network, To initialize the numbers to 0 and use gradient descent ⇒ Don't work.

What happens if you initialize weights to zero?



Symmetric

⇒ compute exactly same function.



if you initialize to 2, all these hidden units are symmetric

⇒ how long you use gradient descent, all units would always compute same function. (not helpful)

if) $W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ $b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ ← okay

↑ also $W^{[1]} = 0$.. can make problem.

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

$W^{[2]} = [0 \ 0]$

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

∴ hidden unit $a_1^{[1]}$ and $a_2^{[1]}$ be completely same. (identical)

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

$W^{[1]} = \begin{bmatrix} — \\ — \end{bmatrix}$ first row equal to second row.

∴ Start with W initialized with zero,

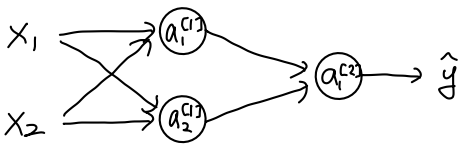
It starts with two hidden units compute same function ⇒ same input to output unit

⇒ how long neural network iterate, both hidden unit are still compute exactly same function.

⇒ = one hidden unit ↑ has

Andrew Ng

<Random Initialization>



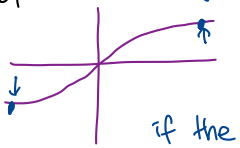
⇒ $W^{[1]} = np.random.randn((2,2)) * 0.01$ ← any very small number

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

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

$b^{[2]} = 0$

∵ Symmetry breaking problem, it's okay to initialize b to 0



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

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

if the weight is too large, activation value → slow gradient. slope is very small, gradient descent is very slow ⇓ learning is slow.