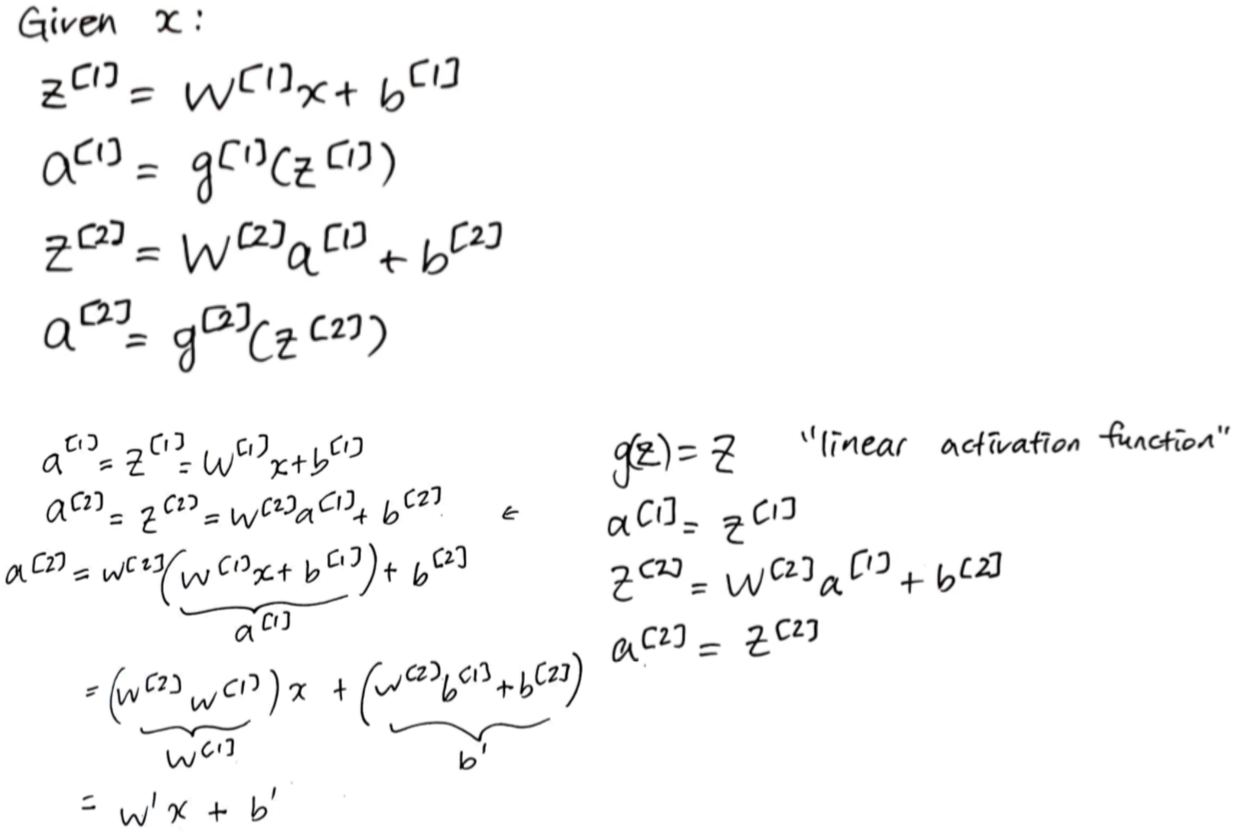
Neural Networks and Deep Learning

박달님 03.14.2021

Why need Non-Linear Activation Functions?

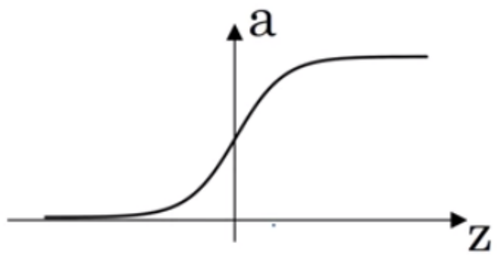
왜 비선형 함수가 필요한 것인가

- 신경망의 흥미로운 함수들을 산출하기 위해서는 비선형 activation function이 필요하다
- Forward propagation equations for the neural network ->
- 만약 선형 함수로 하게 되면 ->
- ‘Identity activation function’ -> just outputting linear function of input
- Activation 함수가 없는 경우에는 신경망이 몇개의 층으로 이루어졌다고 하더라도 단순히 선형 activation 함수를 산출하는 것
- linear function and sigmoid function - 표현적이지 않음. Hidden layer 쓸모 없음
  - 2개의 선형 함수의 구성요소 그 자체가 선형 함수이기 때문
  - 그러므로 비선형 특성을 기입하지 않는 이상은 더 흥미로운 함수를 산출하는 것이 아님
- 선형 함수를 쓰는 한가지의 경우: g(z) = Z machine learning on regression problem
  - 예시) 집 값 예측. Y는 0이나 1이 아니라 실수가 될 것
  - Hidden units가 activation function을 쓰면 안됨
    - ReLu나 tanh, Leaky ReLu etc. 집 값은 다 양수 이므로 여기에서는 ReLu 함수를 사용하면 됨
  - 주로 output layer에서 쓰인다. 하지만, 그 이외의 선형 activation 함수를 hidden layer에서 쓰는 경우는 아주 특별한 경우를 제외하고는 없다



Derivatives of Activation Functions - Sigmoid

Back Propagation에서 필요한 activation function의 derivative



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

Sigmoid function derivative: $\frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right)$

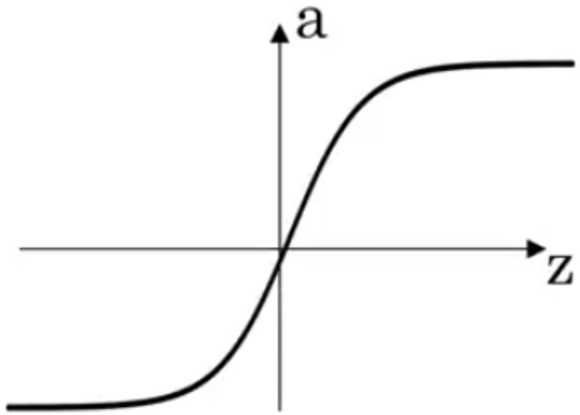
$= g(z)(1-g(z))$

$= a(1\text{-}a)$

- If z is very large -> ex) z = 10, g(z) = 1
- g(z)(1-g(z))
- d/dz g(z) = g’(z)= 1(1-1) = 0
- If z is very small -> ex) z = -10 g(z) = 0
- g’(z) = 0(1-0) = 0
- If z = 0 -> g(z) = ½
- g’(z) = ½(1-½) = ¼
- In neural network: a = $g(z) = \frac{1}{1+e^{-z}}$

Derivatives of Activation Functions

Tanh activation function



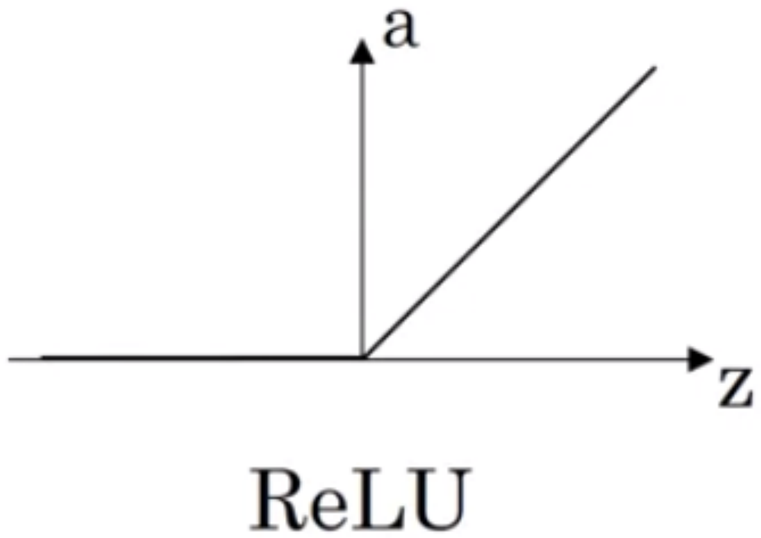
$$g(z) = \tanh(z) = \frac{(e^z - e^{-z})}{(e^z + e^{-z})}$$

- If z = 10, tanh(z) = 1, g'(z) = 0
- If z = -10, tanh(z) = -1, g'(z) = 0
- If z = 0, tanh(z) = 0, g'(z) = 1

$$g'(z) = \frac{d}{dz} g(z) = 1 - (tanh(z))^2$$

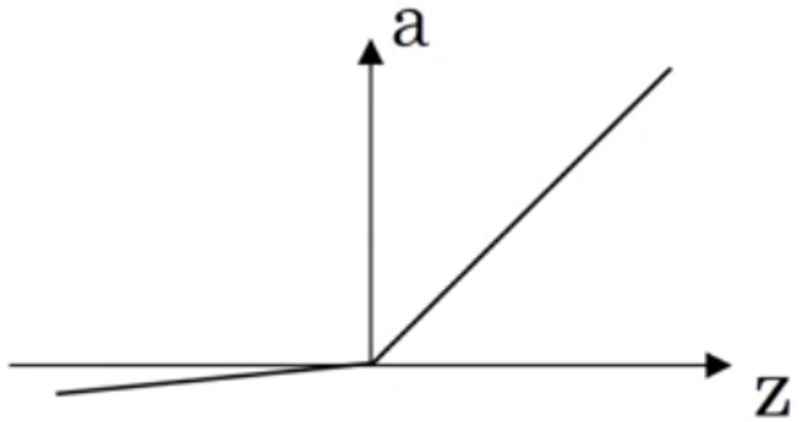
Derivatives of Activation Functions

ReLU and Leaky ReLU



ReLU

- g(z) = max(0, z)
- g’(z) = 0 if z < 0
- g’(z) = 1 if z >= 0
- ~~g’(z) = undefined if z = 0~~



Leaky ReLU

- g(z) = max(0.01z, z)
- g’(z) = 0.01 if z < 0
- g’(z) = 1 if z > 0

Gradient Descent for Neural Networks

Equations needed to implement back-propagation/gradient descent

- Parameters: $w^{[1]}$, $b^{[1]}$, $w^{[2]}$, $b^{[2]}$
- Nx = $n^{0]}$, $n^{[1]}$, $n^{[2]}$=1
- Cost Function: J($w^{[1]}$, $b^{[1]}$, $w^{[2]}$, $b^{[2]}$) = $\frac{1}{m} \sum_{i}^{n} L(\hat{y}, y)$
- Gradient descent:

Repeat {
Compute predictions ($\hat{y}^{(i)}$, i=1,…,m)
$Dw^{[1]} = dJ/dw^{[1]}$, $db^{[1]} = dJ/db^{[1]}$, …
$w^{[1]} = w^{[1]} - \alpha dw^{[1]}$
$b^{[1]} := b^{[1]} - \alpha db^{[2]}$
$b^{[2]} := b^{[2]} - \alpha db^{[2}$
}

Formulas for computing derivatives

Forward Propagation

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

Backward Propagation

$$dZ^{[2]} = A^{[2]} - Y \quad \leftarrow \text{output layer} \qquad Y = [y^{(1)}\, y^{(2)}, \ldots, y^{(m)}]$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} \underline{np.sum}(dZ^{[2]}, \underline{axis=1}, \underline{keepdims=True})$$

- np.sum: summing horizontally
- axis=1: summing horizontally
- keepdims=True: prevents producing $(n,)$ instead insures producing $(n^{[2]}, 1)$

$$dZ^{[1]} = \underbrace{W^{[2]T} dZ^{[2]}}_{\text{element-wise product}} * \underbrace{g^{[1]\prime}(Z^{[1]})}_{\text{derivative of activation function used in hidden layer}}$$

both $(n^{[1]}, m)$ matrix

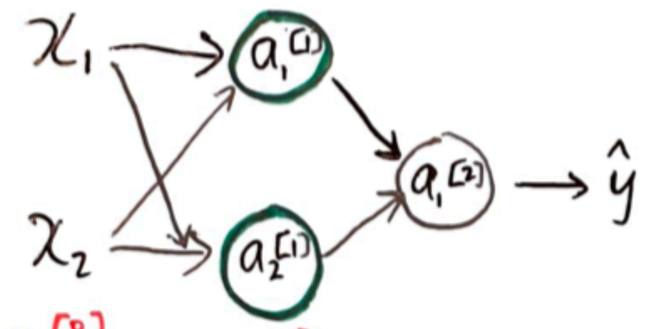
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$
$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis=1, keepdims=True)$$

$(n^{[1]}, 1)$

Random Initialization

- Neural network를 훈련시킬때 weights를 랜덤하게 initialize 해주는게 중요함
- Logistic regression은 weights를 0으로 initialize 해줌
- Neural network는 parameter weight를 0으로 initialize 해주고 gradient descent를 적용하면 작동하지 않을 것
  - 그렇게 하면 어떻게 되는가?

Random Initialization



$n^{[0]}=2 \quad n^{[1]}=2$

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

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

어떤 example 이던 둘의 값이 같다.
둘이 같은 function을 compute 하기 때문에.

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

Hidden units initialize the same way.
outgoing weights are also identical.

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

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

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

몇번을 iterate 하던 상관 없이 두개의 hidden units는 같은 함수를 계산한다.
그러므로 두개의 hidden units는 output unit에 똑같은 영향을 준다.
다른 hidden units들을 이용해서 다른 함수들을 계산하려고 필요한건데 이 경우에는 Hidden unit이 필요가 없어진다.
Solution: parameter를 random하게 initialize한다.

Symmetric - 0 으로 initialize 하면 둘의 hidden units $a_1^{[1]}$, $a_2^{[1]}$가 완전히 같다.

Random Initialization

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

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

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

$b^{[2]} = 0$

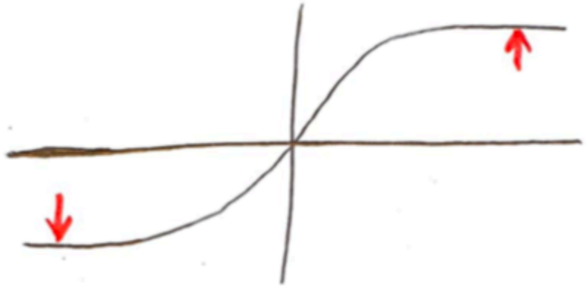
prefer to initialize with small random values.

b does not have symmetry breaking problem

As long as w is initialized randomly, start off with different hidden units computing different things

∴ No longer have symmetry breaking problem

Sigmoid Function



if w is big

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

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

z will be big

slope of gradient function is very small.

∴ learning rate will be very small.

ex) binary classification. tanh/sigmoid.

When training a neural network with just one hidden layer, 0.01 fine.

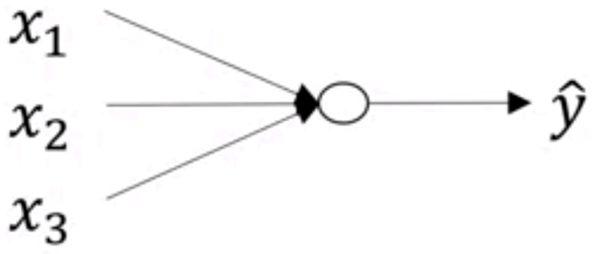
When training a deep neural network, pick a different #.

Relatively small #.

Deep L-layer Neural Network

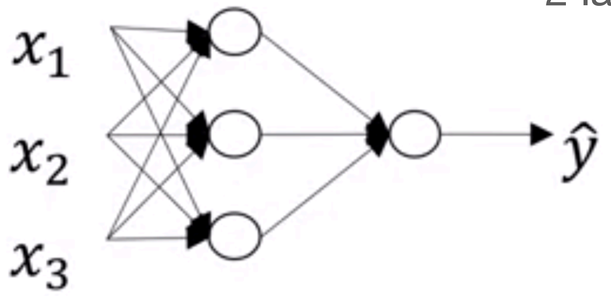
What is a deep neural network?

"shallow"

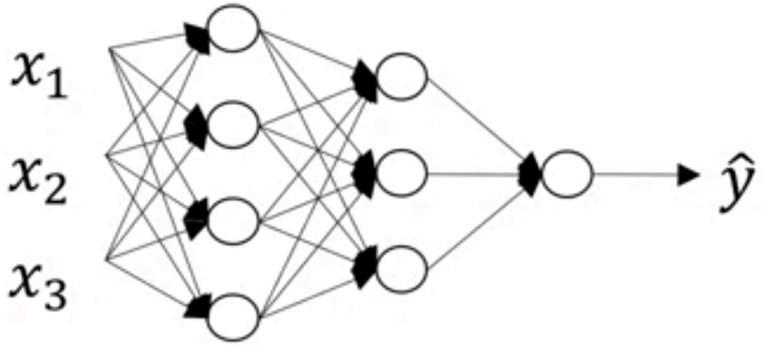


logistic regression

"2 layer" NN

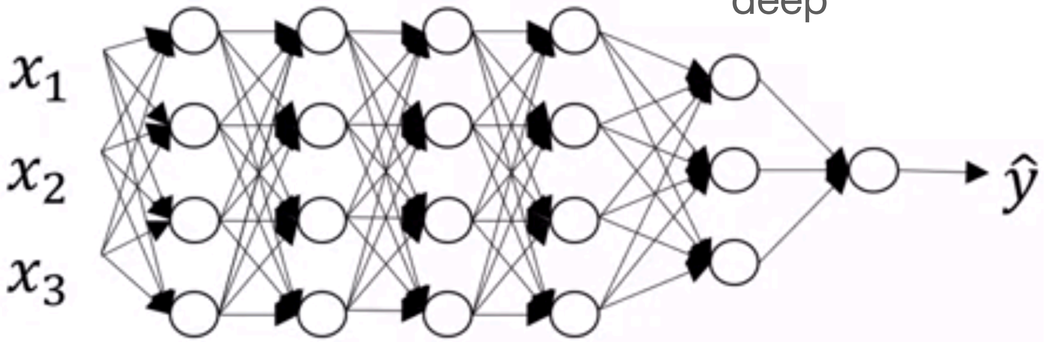


1 hidden layer



2 hidden layers

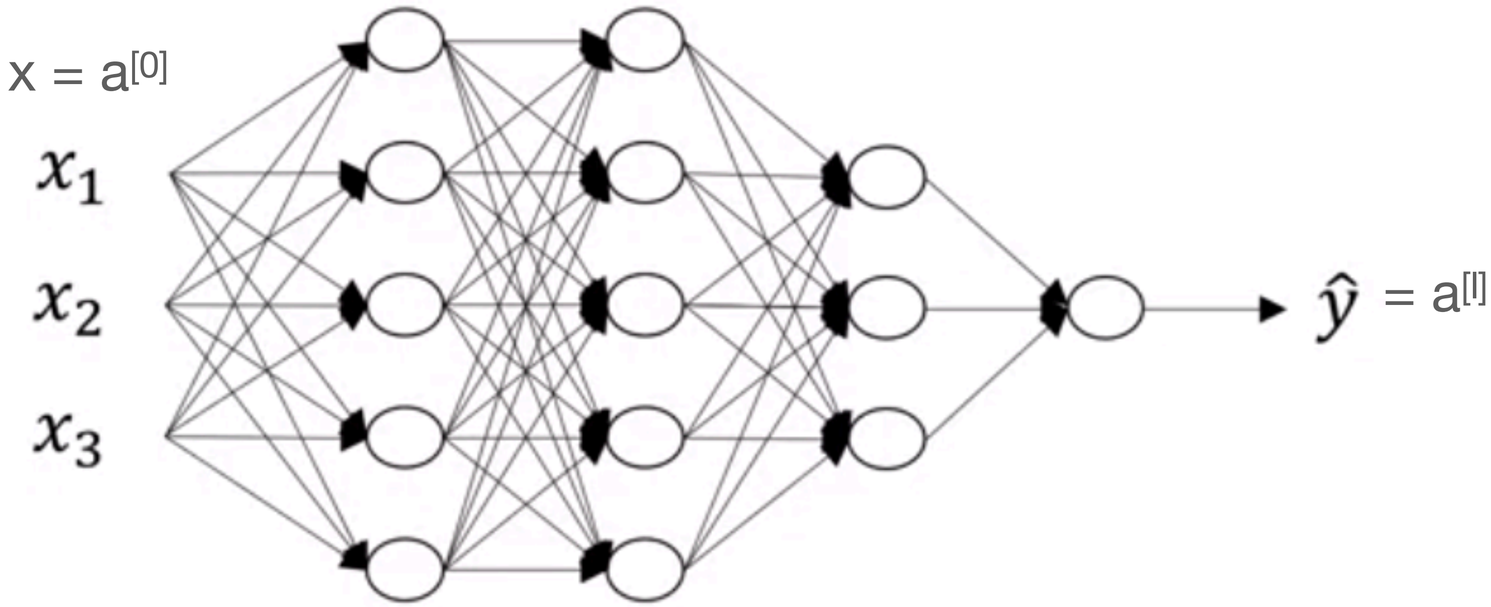
"deep"



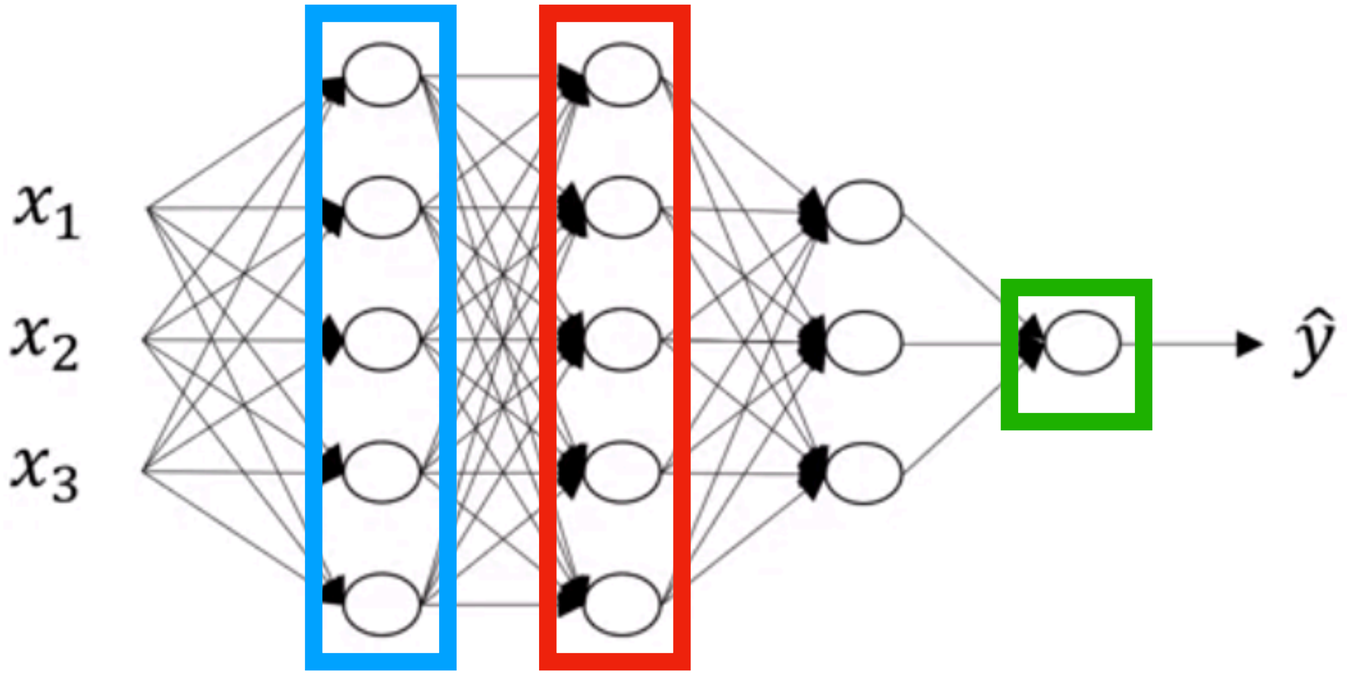
5 hidden layers

Deep neural network notation

L = 4 (# of layers)
$n^{[l]}$ = # units in layer l
$a^{[l]}$ = activations in layer l
$a^{[l]} = g^{[l]}(z^{[l]})$
$w^{[l]}$ = weights for $z^{[l]}$
$b^{[l]}$ = used to compute $z^{[l]}$
$n^{[1]} = 5$
$n^{[2]} = 5$
$n^{[3]} = 3$
$n^{[4]} = n^{[L]} = 1$
$n^{[0]} = n_x = 3$



Forward Propagation in a Deep Network



$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$
$a^{[1]} = g^{[1]}(z^{[1]})$
$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$
$a^{[2]} = g^{[2]}(z^{[2]})$
$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$
$a^{[4]} = g^{[4]}(z^{[4]})$

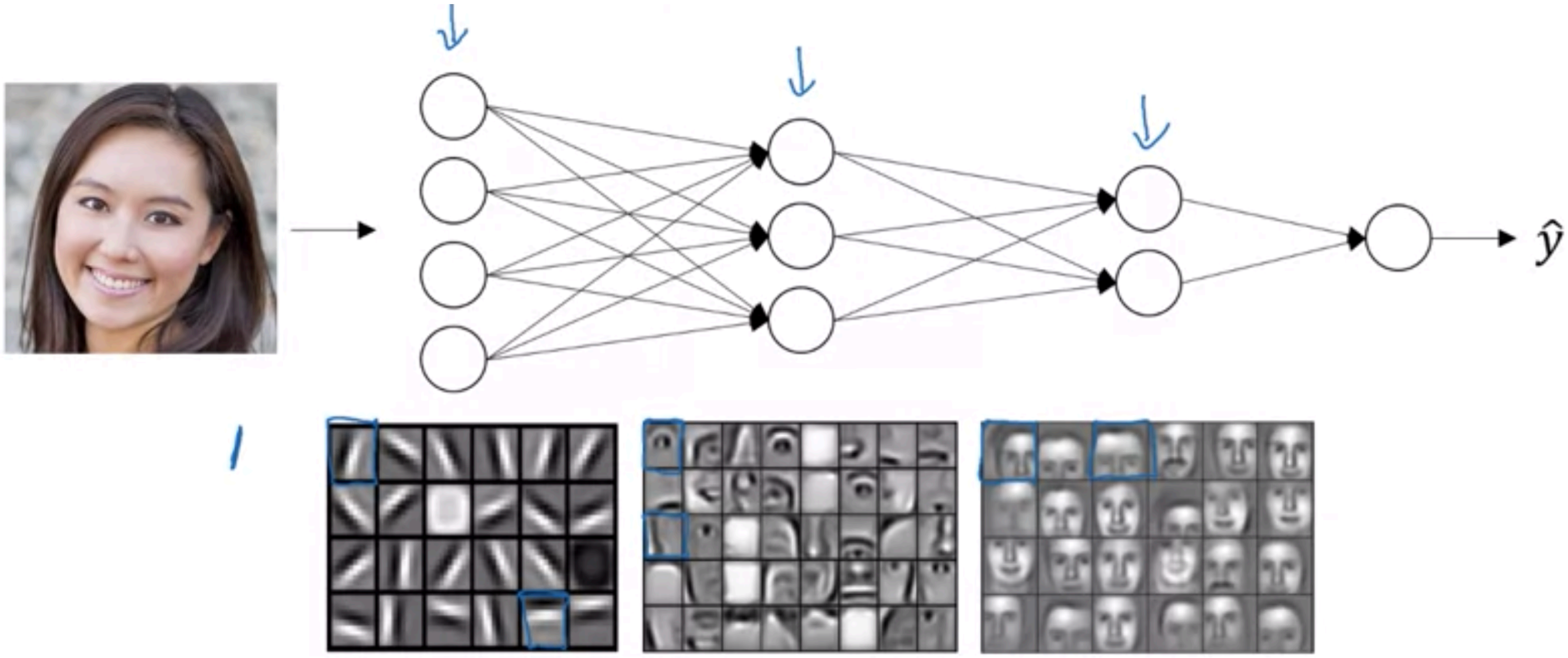
Vectorized:
$Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]}$
$A^{[1]} = g^{[1]}(z^{[1]})$
$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$
$A^{[2]} = g^{[2]}(Z^{[2]})$
$\hat{y} = g(Z^{[4]}) = A^{[4]}$

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

$Z^{[l]} = w^{[l]} A^{[l-1]} + b^{[1]}$
$A^{[l]} = g^{[l]}(Z^{[l]})$

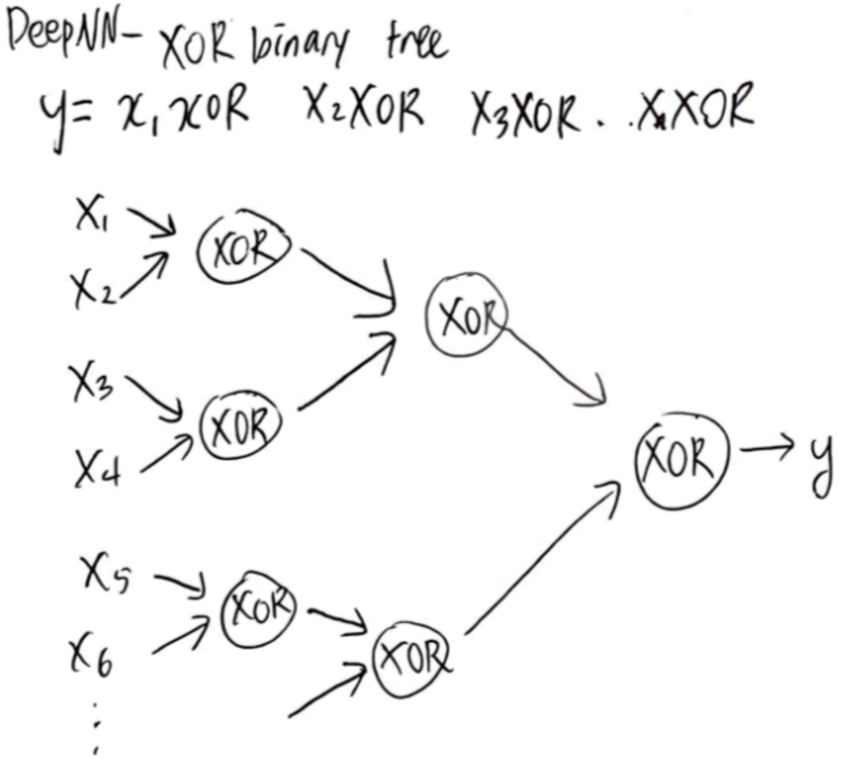
Why Deep Representations?

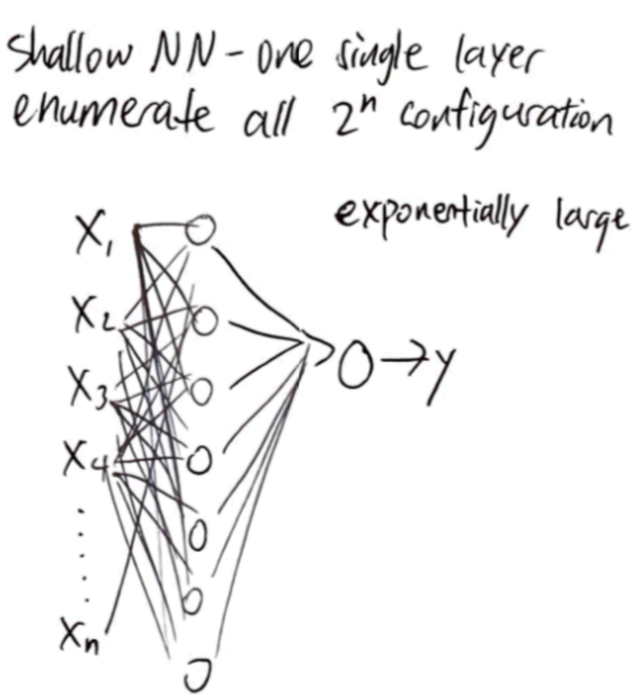
What is the deep network computing?



Circuit theory and deep learning

Informally: There are functions you can compute with a "small" L-layer deep neural network that shallower networks require exponentially more hidden units to compute.





Building Blocks of Deep Neural Networks

One step forward and backward propagation

Layer L: $w^{[L]}, b^{[L]}$

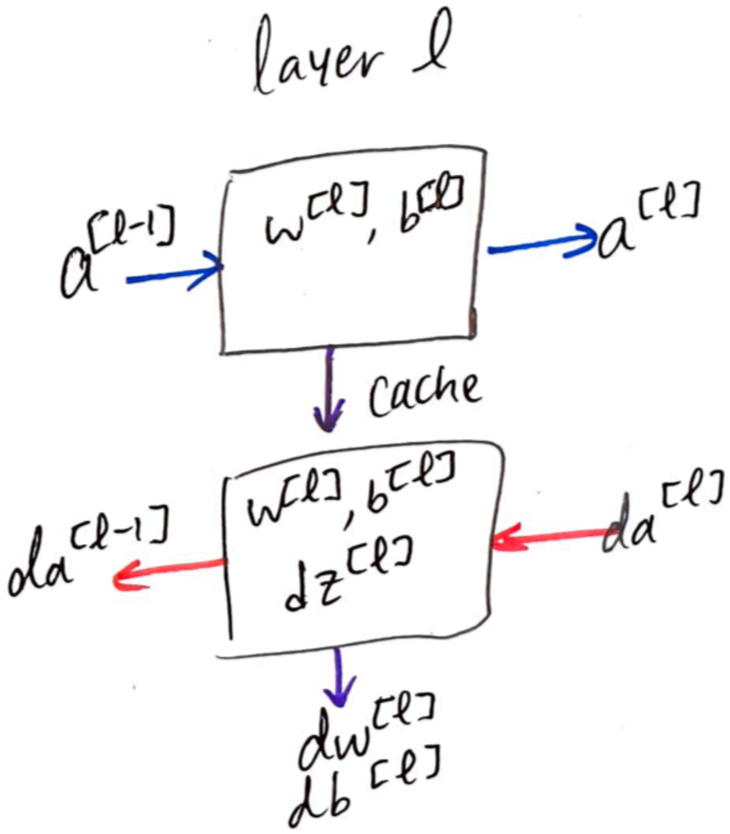
Forward: Input $a^{[l-1]}$, output $a^{[l]}$

$z^{[l]}: w^{[l]}a^{[l-1]} + b^{[l]}$

$a^{[l]}: g^{[l]}(z^{[l]})$

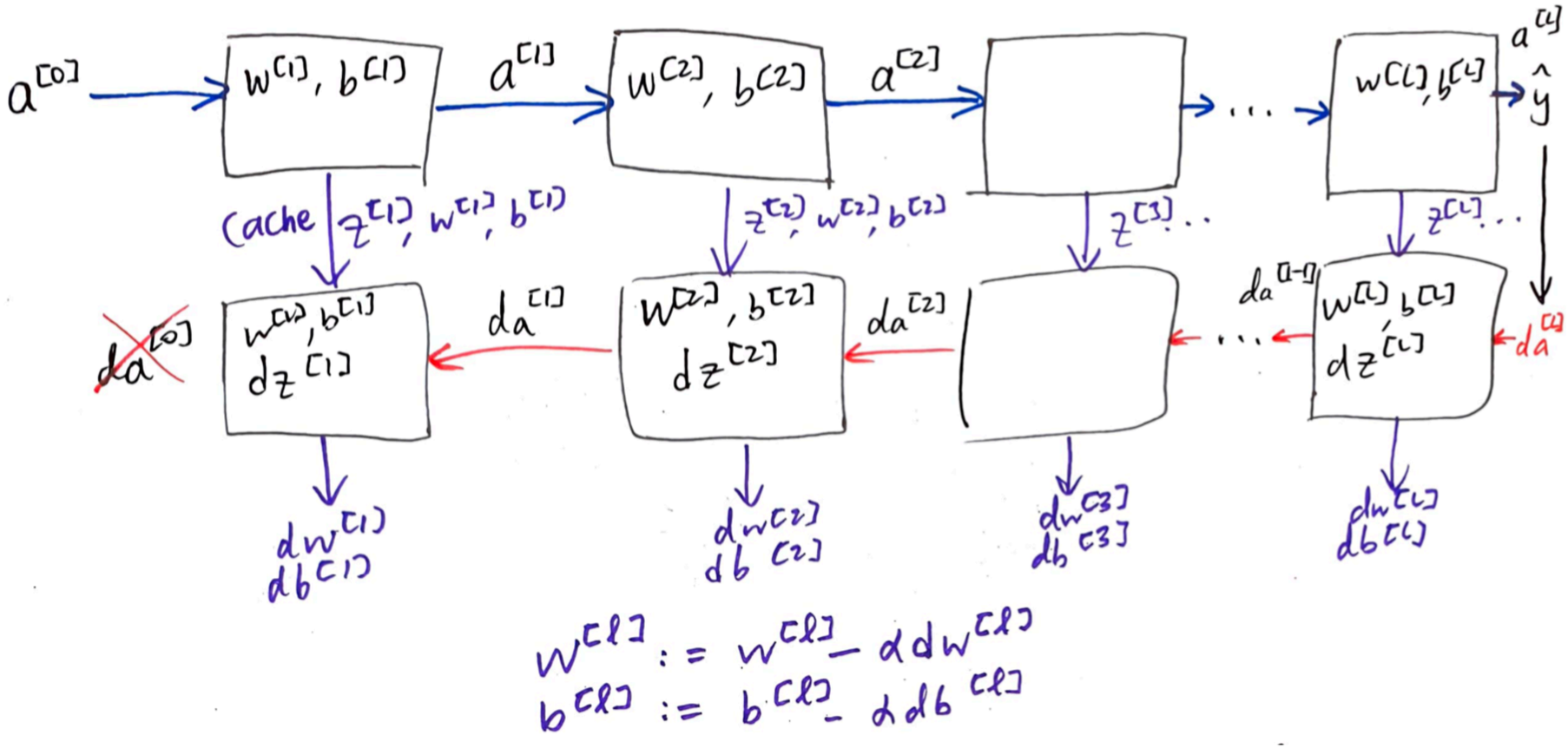
Backward: Input $da^{[l]}$, output $da^{[l-1]}$, $dw^{[l]}$, $db^{[l]}$

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



Building Blocks of Deep Neural Networks

One Iteration forward and backward propagation



Parameters vs. Hyperparameters

- parameters: $W^{[l]}$ and $b^{[l]}$ → trained from data
- **hyperparameters**:
  - **alpha (learning_rate), number of iterations, L, $n^{[l]}$ size of each layer, $g^{[l]}$ at each layer…**
  - **momentum, mini-batch, regularization...**

→ finally decides what params will be.

empirical: try out different hyperparameters