deeplearning.ai

One hidden layer Neural Network

Neural Networks Overview

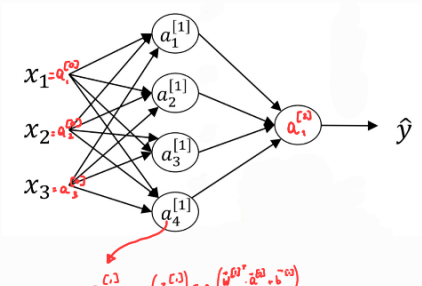
This lesson is about:

- GRADIENT DESCENT in NN via forward + backpropagation
- how to vectorize the algorithm (represent the network in terms of matrices and formulate forward and backpropagation as linear algebra operations) to optimize the computation
- what activation functions to choose (ReLu for hidden, sigmoid for out (bc))
- how to initialize the weights (randomly).

BRIEFLY, WHAT IS BACKPROPAGATION?

- GRADIENT DESCENT optimizes the NN's parameters based on the gradient of the cost fn wrt each parameter. The following applies similarly to other numerical optimiz methods that use the gradient of the cost function.
- BACKPROPAGATION is an efficient algo to compute the gradient of the cost fn wrt each parameter. Backpropagation is efficient because it first computes the loss gradient wrt "later" parameters (closer to the output) and reuses them to compute the gradient wrt "earlier" parameters.

INFORMALLY SKETCH BACKPROPAGATION FOR A 2-layer NN



- Each neuron implements a linear concatenation of the inputs followed by a nonlinear function:

$$a_k^{[i]} = g\left(z_k^{[i]}\right) = g\left(\vec{w}^{[i]T} \cdot \vec{x}^{[i-1]} + b^{[i]}\right)$$

("activation of the k-th neuron on the i-th layer).

- We operate a FORWARD PROPAGATION pass to get the prediction of the network $\hat{y} = \vec{a}^{[2]} = a_0^{[2]}$ for the given input sample(s), and the loss $L(\hat{y}, y)$. If we have many samples, we should instead compute the cost function $J = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$. However that is redundant because in the end we will need $\frac{\partial J}{\partial var}$ which is $\frac{1}{m}\sum_{i=1}^{m} \frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial var}$ (average of the gradient of the loss wrt variable). So, for each sample we compute the *bar* wrt variable of interest, we accumulate it, and we average its value after having cycled over all the samples.

- We operate a BACKPROPAGATION, that is we compute the gradient of the loss (averaged over the input samples because we actually need the gradient of the cost function) wrt the parameters of the network using the chain rule for efficient computation. For example, in this case we want $\frac{\partial L}{\partial W^{[2]}}, \frac{\partial L}{\partial b^{[2]}}, \frac{\partial L}{\partial W^{[1]}}, \frac{\partial L}{\partial b^{[1]}}$ and to get these we proceed backwards from the last layer to the first. We can do this because the loss function depends on the output of the network $a^{[2]}$ which corresponds to a concatenation of functions

$$L\left(a^{[2]}, y\right) = L\left(g\left(W^{[2]} a^{[1]} + b^{[1]}\right), y\right) = L\left(g\left(W^{[2]}\left(W^{[1]} a^{[0]} + b^{[1]}\right) + b^{[2]}\right), y\right).$$

- After iterating over all the samples and accumulating the gradient of the loss wrt the params of interest we compute its average (=gradient of the cost function) and use that to update the parameters according to the chosen optimization algorithm, eg, gradient descent.

$\vec{W}^{[2]} = \vec{0}$, $\vec{b}^{[2]} = \vec{0}$, $W^{[2]} = \vec{0}$, $\vec{b}^{[2]} = \vec{0}$

for $a^{[2]}, y$ in $\{x^{(i)}, y^{(i)}\}_{...}^{m}$:

FP
$$a^{[1]} = g\left(W^{[1]} \cdot a^{[0]} + b^{[1]}\right)$$
$$a^{[2]} = g\left(W^{[2]T} \cdot a^{[1]} + b^{[2]}\right)$$
$$L = L\left(a^{[2]}, y\right) \text{ (remember) } \cdot L\left(g\left(W^{[2]} g\left(W^{[1]} a^{[0]} + b^{[1]}\right) + b^{[2]}\right), y\right)$$

BP
$$da^{[2]} = \frac{\partial L}{\partial a^{[2]}}$$
$$dz^{[2]} = \frac{\partial L}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} = da^{[2]} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}}$$
$$dW^{[2]} \mathrel{+}= da^{[2]} \cdot dz^{[2]} \cdot \frac{\partial z^{[2]}}{\partial W^{[2]}} \quad \text{(chain rule)}$$
$$db^{[2]} \mathrel{+}= da^{[2]} \cdot dz^{[2]} \cdot \frac{\partial z^{[2]}}{\partial b^{[2]}}$$
$$da^{[1]} = dz^{[2]} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}}$$
$$dz^{[1]} = da^{[1]} \cdot da^{[1]} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}}$$
$$dW^{[1]} \mathrel{+}= dz^{[1]} \cdot da^{[1]} \cdot dz^{[1]} \cdot \frac{\partial z^{[1]}}{\partial W^{[1]}}$$
$$db^{[1]} \mathrel{+}= dz^{[1]} \cdot da^{[1]} \cdot dz^{[1]} \cdot \frac{\partial z^{[1]}}{\partial b^{[1]}}$$

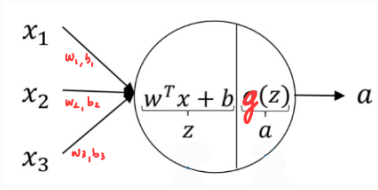
$dW^{[2]} /= m$; $db^{[2]} /= m$; $dW^{[1]} /= m$; $db^{[1]} /= m$; averaging the gradient of the loss over the m samples is equal to computing the gradient of the cost fn.

GD
$$W^{[2]} -= \alpha \cdot dW^{[2]}$$
$$b^{[2]} -= \alpha \cdot db^{[2]}$$
$$W^{[1]} -= \alpha \cdot dW^{[1]}$$
$$b^{[1]} -= \alpha \cdot db^{[1]}$$

VECTORIZED REPRESENTATION OF THE NETWORK, & EFFICIENT IMPLEMENTATION OF GRADIENT DESCENT FOR A 2-LAYER NETWORK WITH SIGMOID ACTIVATION NEURONS AND LOGISTIC REGRESSION LOSS

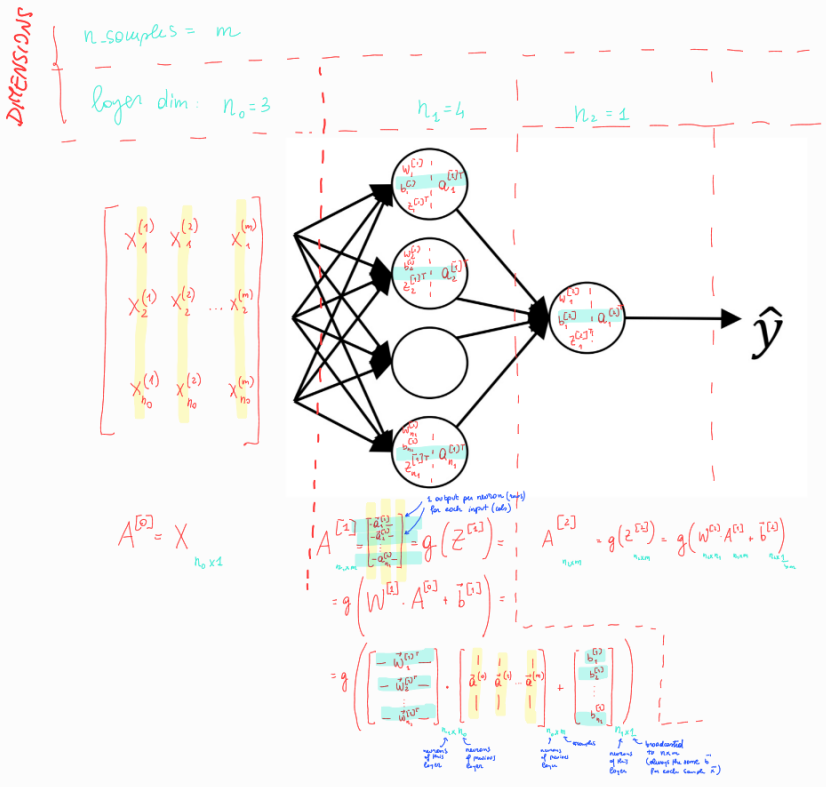
NOTATION AND VECTORIZED REPRESENTATION OF THE NETWORK

Single neuron



- Assume that each neuron applies a linear combination z of the inputs $\vec{x}$, and then applies a nonlinear function $g(z)$.
- The linear combination uses the parameters $\vec{w}$ and $\vec{b}$.
- The parameters are referred to as weights $\vec{w}$ and offsets $\vec{b}$ (one for each input / edge), the nonlinear function as "activation function", while the output of the neuron as "activation value" of the neuron.
- Functionally the neuron is $a = g(z) = g(w^T \cdot x + b)$.

2-layer network & many input samples



- assume m input samples with dimension n_0
- assume the dimension (= no. neurons) of the layer i to be n_i.
- with $a^{[i][j]}_k$ we refer to a variable that relates to the k^{th} neuron of the i-th layer for the j-th sample (i: 1, 2)
- intuition: the matrices that represent layers activations have 1 row for each neuron, and 1 column for each input sample.
- intuition: the parameter matrices (W, b) relate to edges) have 1 row for each neuron, and 1 column for each neuron of the previous layer.
- Logistic regression loss $L(a, y) = -\left[y \log(a) + (1-y) \log(1-a) \right]$
- Sigmoid activation function $g(z) = \frac{1}{1+e^{-z}}$
- Derivatives in the new vectorized case:
  - $\frac{\partial g(z)}{\partial z} = g(z) \cdot (1 - g(z))$; $\frac{\partial \log(a)}{\partial a} = \frac{1}{a}$
  - $\frac{\partial L}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a}$
  - $\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} = \left[-\frac{y}{a} + \frac{1-y}{1-a} \right] \cdot \left[a(1-a) \right] = a - y$
  - $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial W} = (a^{[2]} - y) \cdot a^{[1]T}$
  - $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial b} = a^{[2]} - y$

- Vectorized procedure:
  REPEAT UNTIL CONVERGENCE:
  Given $X = A^{[0]}$ $(n_0 \times m)$ and Y $(1 \times m)$,
  init $W^{[1]} = \vec{0}$ $(n_1 \times n_0)$, $\vec{b}^{[1]} = \vec{0}$ $(n_1 \times 1)$, $W^{[2]} = \vec{0}$ $(n_2 \times n_1)$, $b^{[2]} = \vec{0}$ $(n_2 \times 1)$

F.P.

$$A^{[1]} = \sigma\left(\underbrace{W^{[1]} A^{[0]}}_{n_1 \times m} + \underbrace{\vec{b}^{[1]}}_{\text{broad.}\ n_1 \times 1 \to n_1 \times m} \right)$$
$$A^{[2]} = \sigma\left(\underbrace{W^{[2]} A^{[1]}}_{n_2 \times m} + \underbrace{b^{[2]}}_{\text{broad.}\ n_2 \times 1 \to n_2 \times m} \right)$$

B.P.

- we may skip the explicit computation of the loss $\left(L(A^{[2]}, Y) = Y \log A^{[2]} + (1-Y)\log(1-A^{[2]})\right)$ because we already know its gradient wrt $z^{[2]}$:

$$dz^{[2]} = A^{[2]} - Y \quad (n_2 \times m)$$
each col is the grad computed by one sample
$$db^{[2]} = dz^{[2]} \cdot \frac{\partial z^{[2]}}{\partial b^{[2]}} = dz^{[2]} \quad (n_2 \times m)$$ but we want the average grad over all samples so
$$= \text{np.mean}\left(dz^{[2]}, \text{axis}=1, \text{keepdim}=\text{True}\right) \quad (n_2 \times 1)$$

$$dW^{[2]} = \frac{\partial \mathcal{L}}{\partial W^{[2]}} = dz^{[2]} \cdot A^{[2]T} \quad (n_2 \times n_1)$$
again, we want the average gradient, but in this case the cols of $dz^{[2]}$ have already been summed up by the dot product, so to obtain the average it is enough to divide by m
$$= \frac{1}{m} dz^{[2]} \cdot A^{[2]T}$$

(?)
$$dz^{[1]} = dz^{[2]} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} = \text{for some reason the } W^{[2]} \text{ goes to the left of } dz^{[2]} \ldots = W^{[2]T} \cdot dz^{[2]} \circledast g'(z^{[1]})$$
$(n_1 \times n_2)\ (n_2 \times m)\ (n_1 \times m)$
$$dW^{[1]} = dz^{[1]} \cdot A^{[1]T}$$
we want to average again
$$= \frac{1}{m} dz^{[1]} \cdot A^{[0]T}$$
$(n_1 \times m)\ (m \times n_0)\ (n_1 \times n_0)$
$$db^{[1]} = \text{np.mean}\left(dz^{[1]}, \text{axis}=1, \text{keepdim}=\text{True}\right)$$

G.D.

$$W^{[2]} -= \alpha\, dW^{[2]}\ ; \quad b^{[2]} -= \alpha\, db^{[2]}\ ; \quad W^{[1]} -= \alpha\, dW^{[1]}\ ; \quad b^{[1]} -= \alpha\, db^{[1]}$$

GENERAL FORM OF FP & BP (for the general layer l), provided in the last slide of week 4.

Forward and backward propagation

FP

$$
\begin{aligned}
Z^{[1]} &= W^{[1]}X + b^{[1]} \\
A^{[1]} &= g^{[1]}(Z^{[1]}) \\
Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\
A^{[2]} &= g^{[2]}(Z^{[2]}) \\
&\vdots \\
A^{[L]} &= g^{[L]}(Z^{[L]}) = \hat{Y}
\end{aligned}
$$

BP

$$
\begin{aligned}
dZ^{[L]} &= A^{[L]} - Y \\
dW^{[L]} &= \frac{1}{m} dZ^{[L]} A^{[L]T} \\
db^{[L]} &= \frac{1}{m} np.\text{sum}(\text{d}Z^{[L]}, axis = 1, keepdims = True) \\
dZ^{[L-1]} &= dW^{[L]T} dZ^{[L]} g'^{[L]}(Z^{[L-1]}) \\
&\vdots \\
dZ^{[1]} &= dW^{[L]T} dZ^{[2]} g'^{[1]}(Z^{[1]}) \\
dW^{[1]} &= \frac{1}{m} dZ^{[1]} A^{[1]T} \\
db^{[1]} &= \frac{1}{m} np.\text{sum}(\text{d}Z^{[1]}, axis = 1, keepdims = True)
\end{aligned}
$$

* Some formulas in the BP are not explained, they basically say to trust them or try verify it by yourself.

WHICH ACTIVATIONS TO USE:

- In general use ReLU (or maybe try leaky ReLU) because they it is less affected by the vanishing gradient problem (at least for large inputs). The derivative of the ReLU although analytically inexact, works well in practice.

$$
\frac{d\, max(0, x)}{dx} := \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}
$$

- Never use linear activation: the resulting NN would be linear and boil down to simple linear regression/classification.

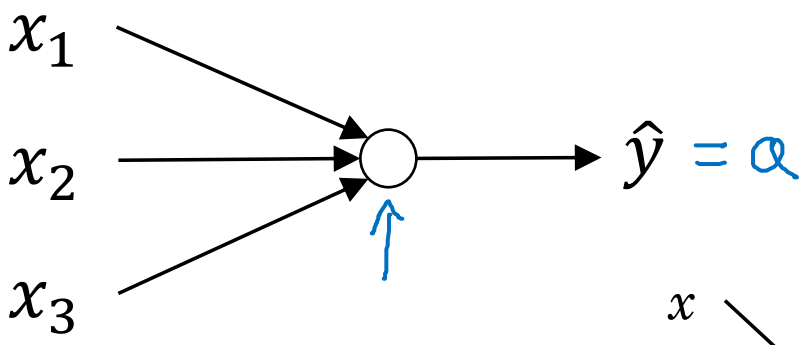
- A sigmoid activation can be used in the last layer if binary classification is desired.

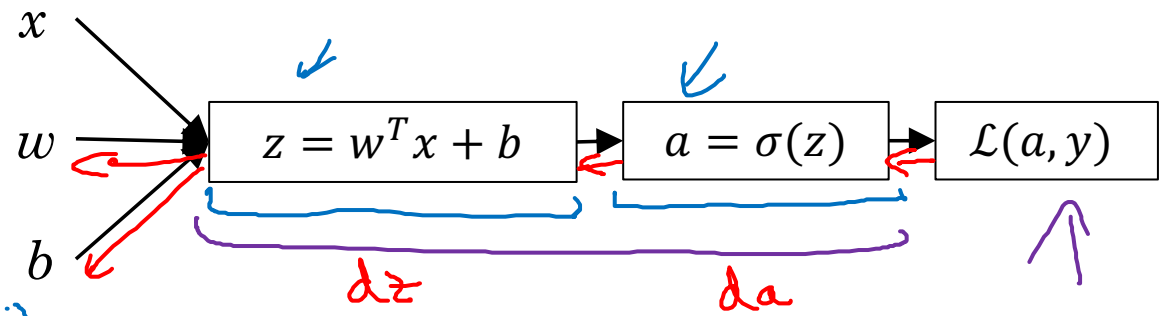
HOW TO INITIALIZE THE WEIGHTS:

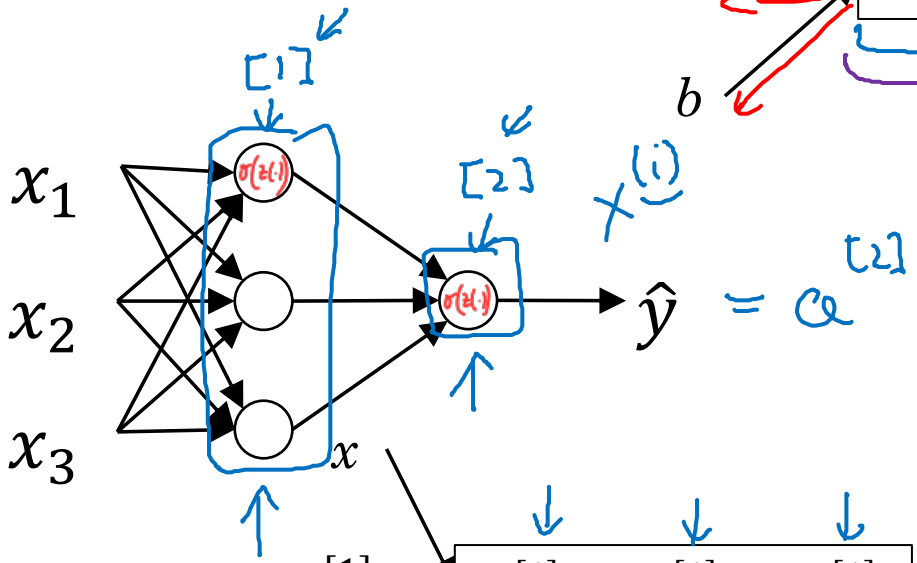
- To small random number, especially if a sigmoid activation is used anywhere in the network

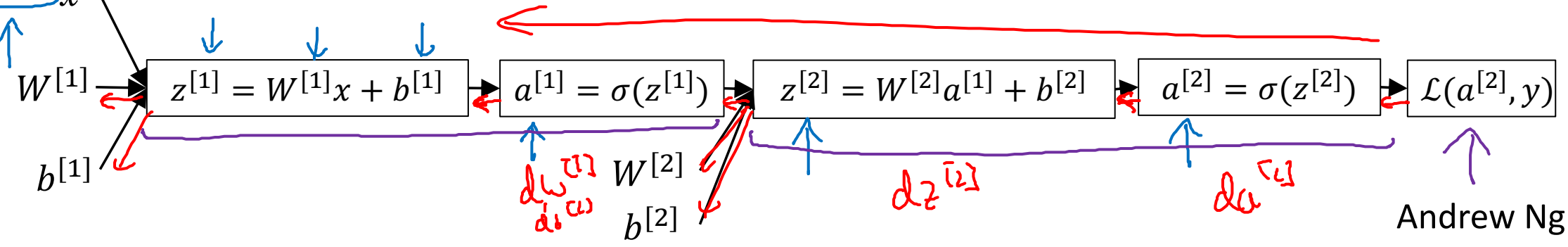
What is a Neural Network?

- Notation :
  - Superscript $[i]$ refers to the i-th layer
  - superscript (i) refers to the i-th training sample
- They use sigmoid neurons : each neuron implements $\sigma(W^T X + B)$









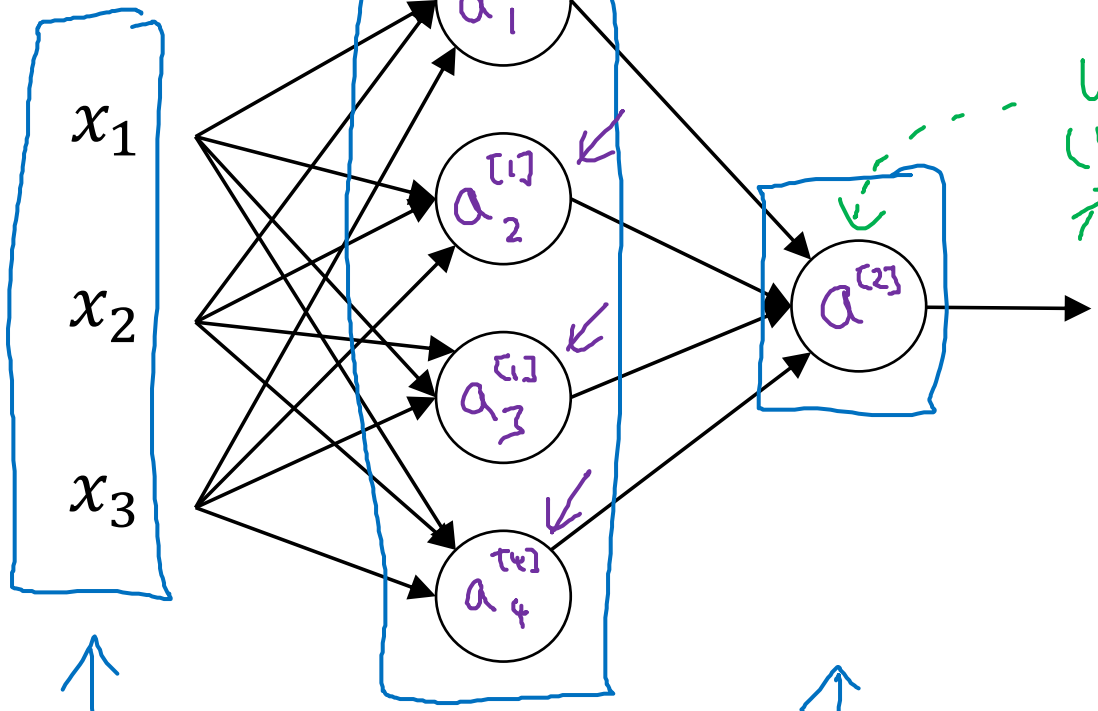
Andrew Ng

deeplearning.ai

One hidden layer Neural Network

Neural Network Representation

Neural Network Representation



$a^{[0]} = X$

$a^{[1]}$ — $w^{[1]}$, $b^{[1]}$ — (4,3), (4,1)

$w^{[2]}$, $b^{[2]}$ — (1,4), (1,1)

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

"y = a"

→ Input layer

→ Hidden layer

→ Output layer

"you don't see it in the training set"

NOTATION:
- in/hid/out layer
- activation of a neuron
- "2 layer NN"
- the parameters of each layer can be grouped as $W^{[i]}$ (#neurons in i × #inputs) and $b^{[i]}$ (#neurons in i × 1)

"2 layer NN"

because we don't count the input layer.

(sometimes the input layer is referred to with the superscript [0]).

$$a^{[1]} = \begin{bmatrix} a^{[1]}_1 \\ a^{[1]}_2 \\ \vdots \\ a^{[1]}_4 \end{bmatrix}$$

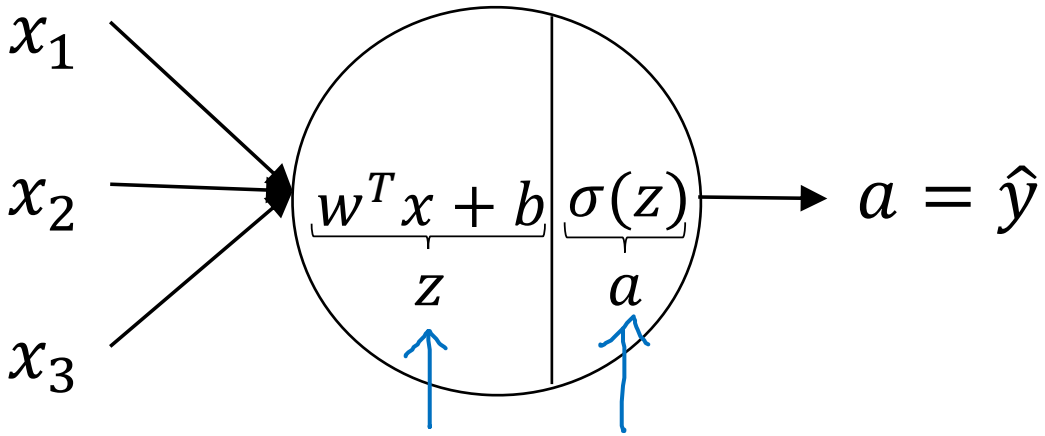
Andrew Ng

deeplearning.ai

One hidden layer Neural Network

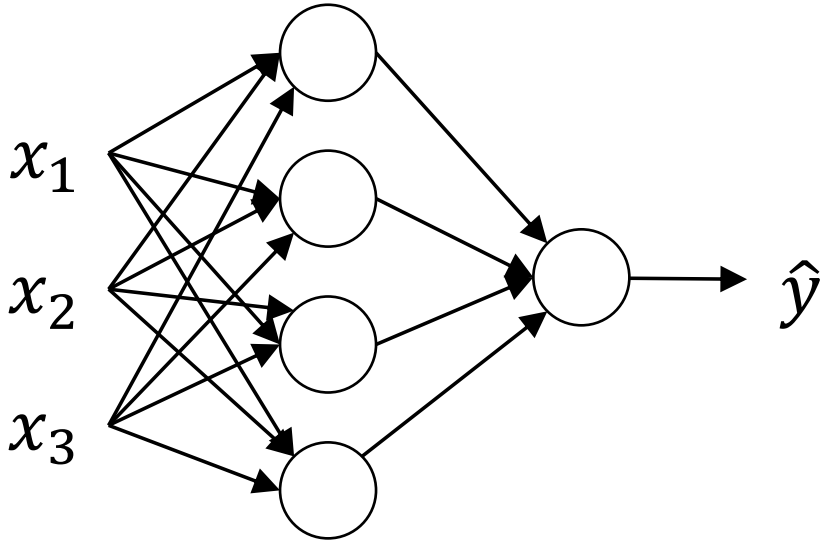
Computing a Neural Network's Output

Neural Network Representation

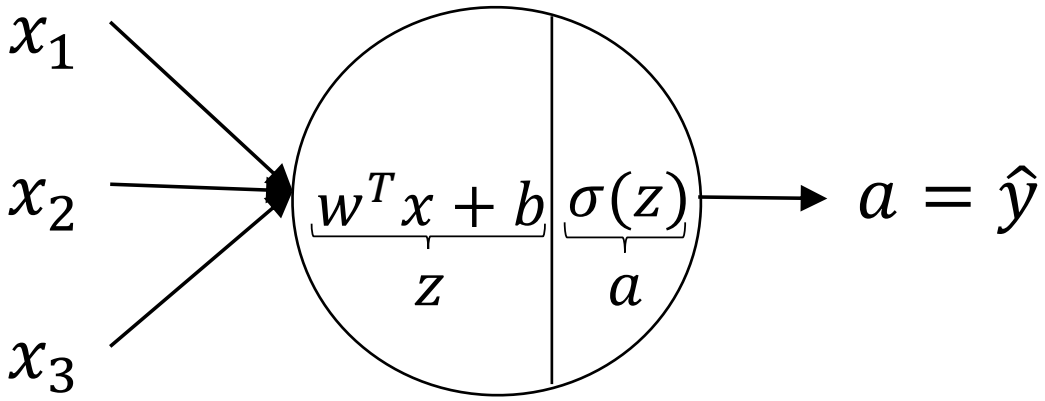


$$z = w^T x + b$$

$$a = \sigma(z)$$

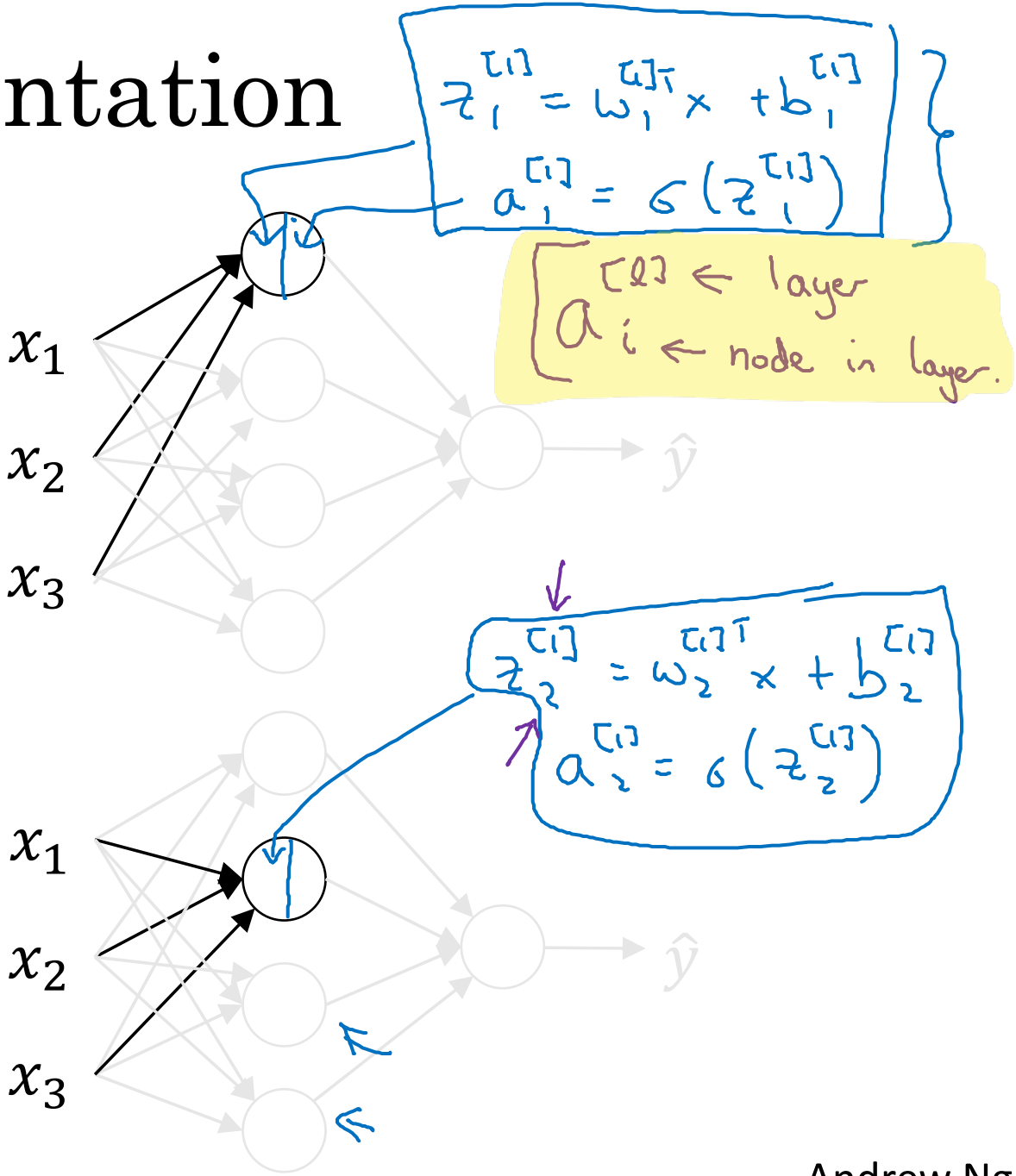


Neural Network Representation



$$z = w^T x + b$$

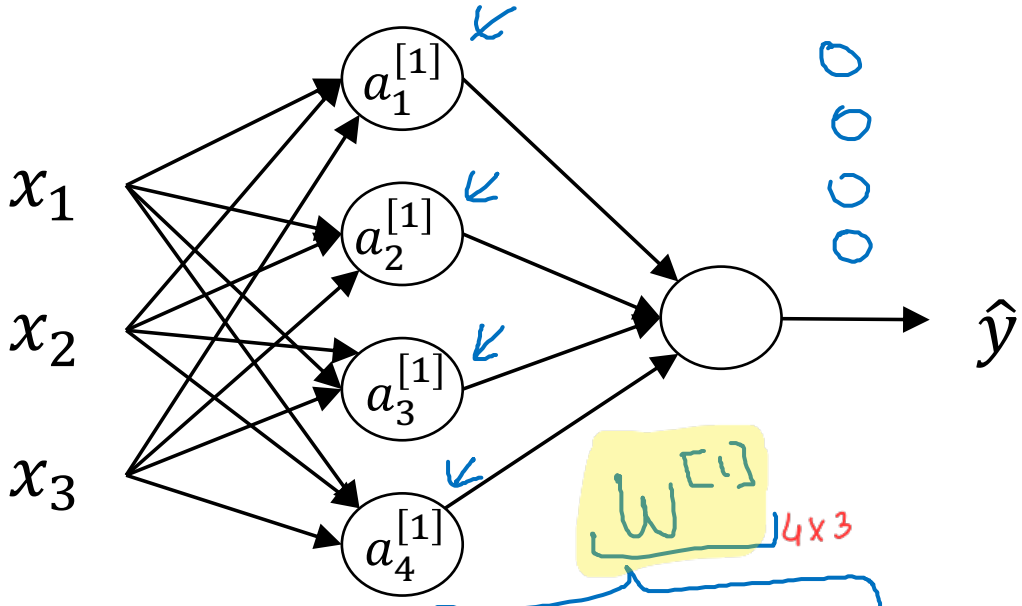
$$a = \sigma(z)$$



Andrew Ng

- how to vectorize the operations of the neurons of one layer.

Neural Network Representation



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

$(w_1^{[1]})^T x$ $a^{[1]}$

$W^{[1]}$ (4x3)

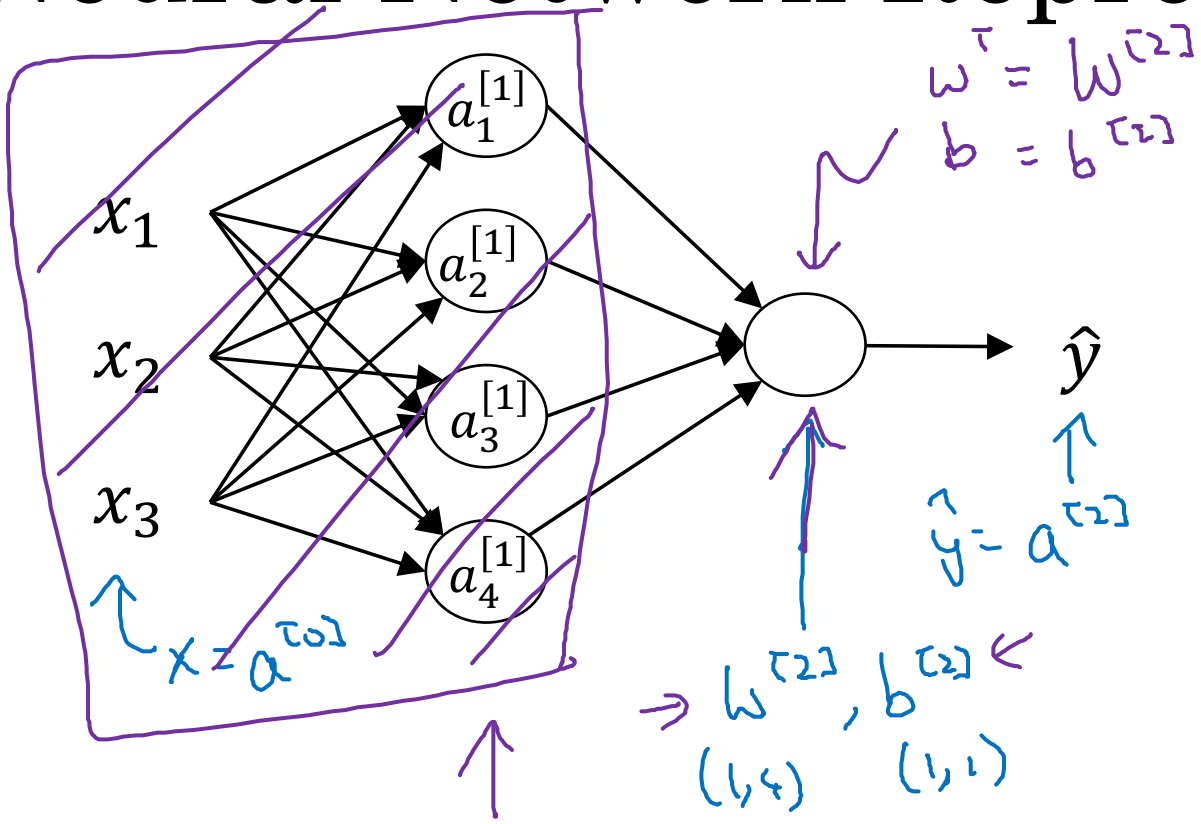
$$z^{[1]} = \begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_3^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x +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

$W^{[1]}$ (4,3) $b^{[1]}$ (4,1)

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

Andrew Ng

Neural Network Representation learning



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

$b = b^{[2]}$

$z = w^T x + b$

$\hat{y} = a = \sigma(z)$

Given input x:

$$z^{[1]}_{(4,1)} = W^{[1]}_{(4,3)} a^{[0]}_{(3,1)} + b^{[1]}_{(4,1)}$$

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

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

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

vectorized representation of the NN on the left.

deeplearning.ai

One hidden layer Neural Network

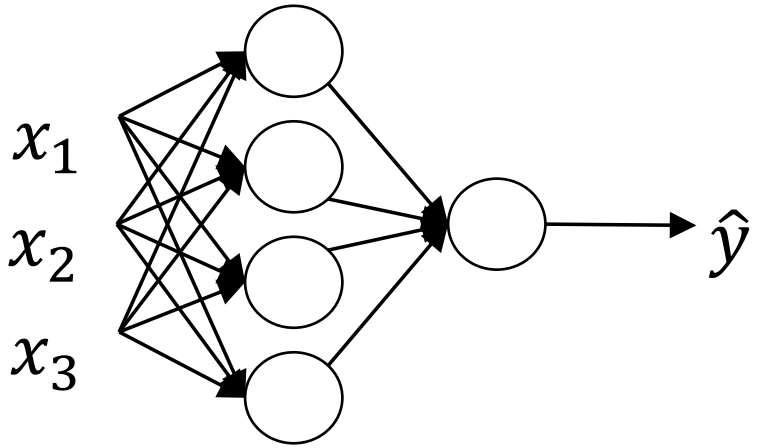
Vectorizing across multiple examples

$X = A^{[0]}$ are $n_x \times m$ (n_x: input dim, m: no. samples)

$Z^{[i]}, A^{[i]}$ are no_neurons$^{[i]} \times m$

$W^{[i]} = \begin{bmatrix} - w^{[i]T} - \end{bmatrix}$ is no_neurons$^{[i]} \times n_x$, $b^{[i]} = \begin{bmatrix} b^{[i]} \\ \vdots \end{bmatrix}$ is no_neurons$^{[i]} \times 1$

Vectorizing across multiple examples



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

1st sample $x \rightarrow a^{[2]} = \hat{y}$

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

3rd sample $x^{(2)} \rightarrow a^{[2](2)} = \hat{y}^{(2)}$

$\vdots$

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

$a^{[2](i)}$ ← example i, layer 2

for $i = 1$ to m,

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

Vectorizing across multiple examples

```
for i = 1 to m:
```

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

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

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

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

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

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

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

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

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

(n_x, m)

training examples

hidden units

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

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

hidden units

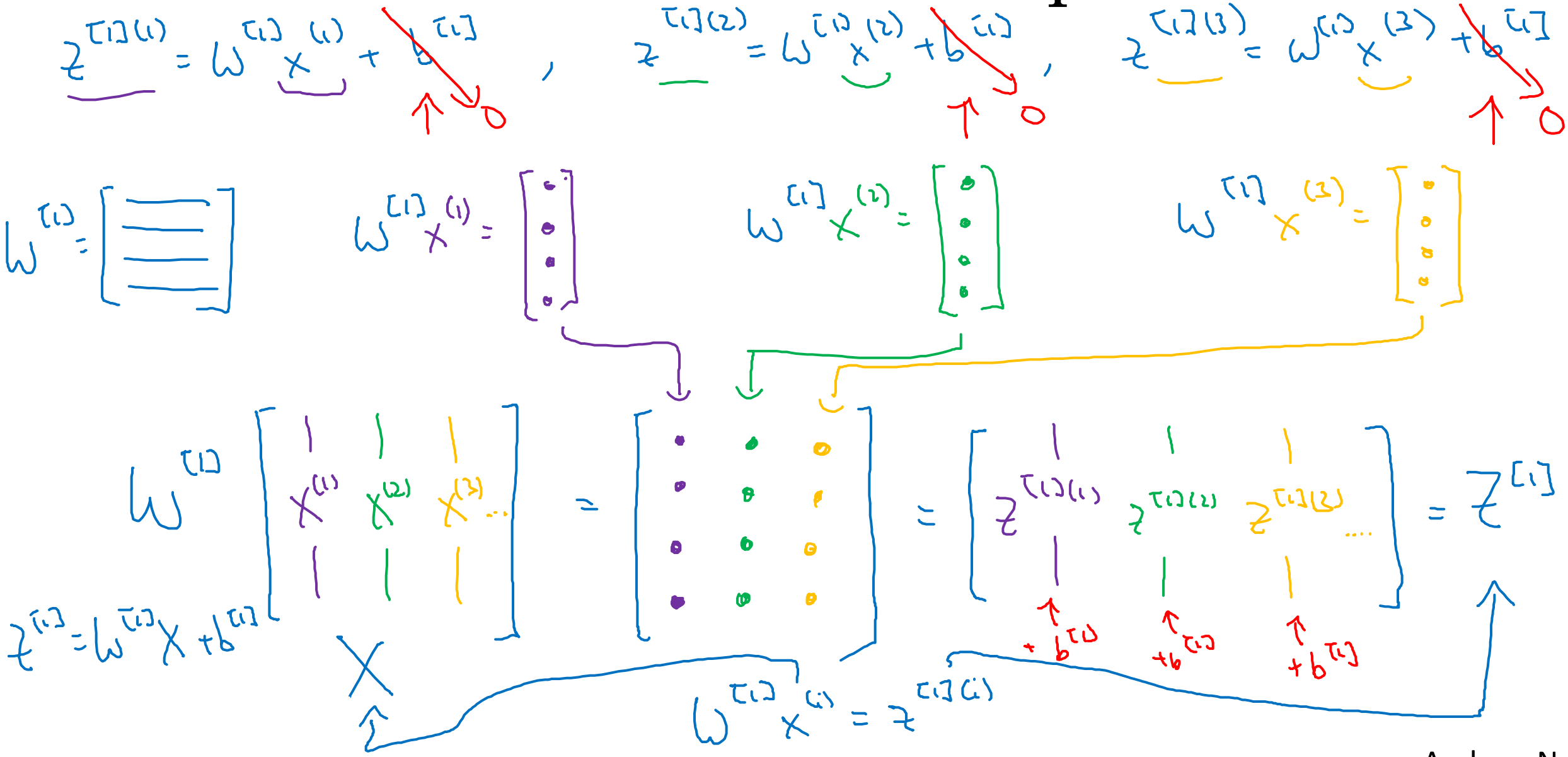
Andrew Ng

deeplearning.ai

One hidden layer Neural Network

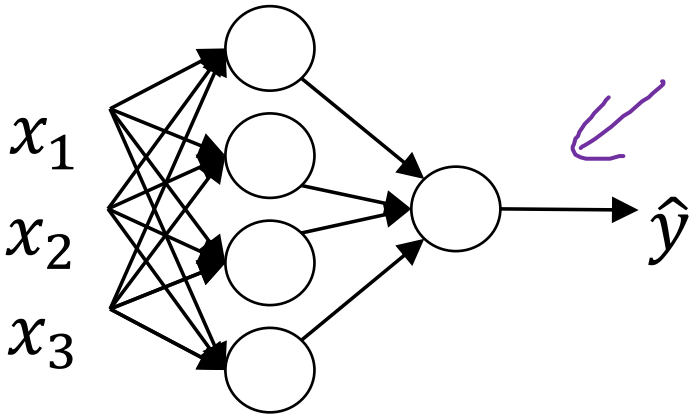
Explanation for vectorized implementation

Justification for vectorized implementation



Andrew Ng

Recap of vectorizing across multiple examples



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

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

```
for i = 1 to m
```

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

$A^{[0]} = X = a^{[0]}$ $\quad x^{(i)} = a^{[0](i)}$

$$Z^{[1]} = W^{[1]}X + b^{[1]} \quad \leftarrow \quad W^{[1]}A^{[0]} + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

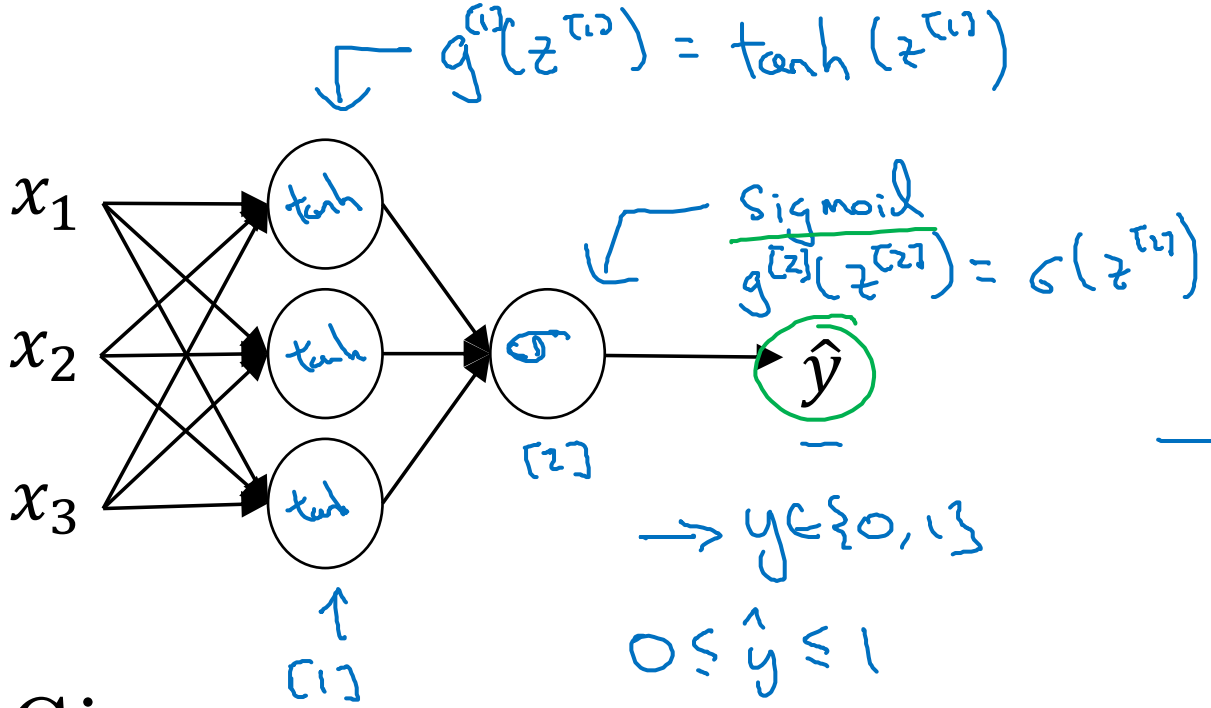
Andrew Ng

deeplearning.ai

One hidden layer Neural Network

Activation functions

Activation functions



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

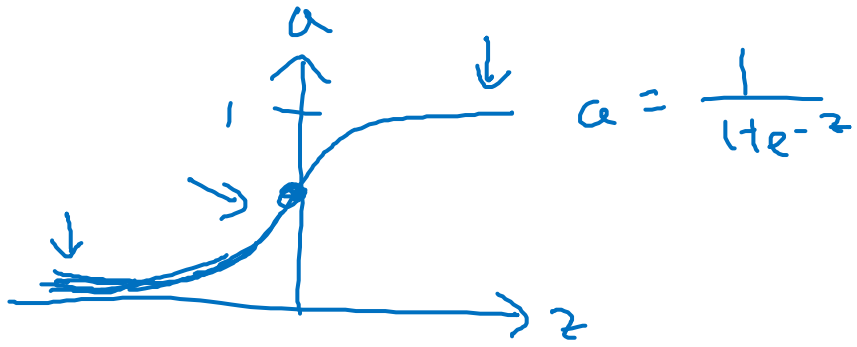
Sigmoid

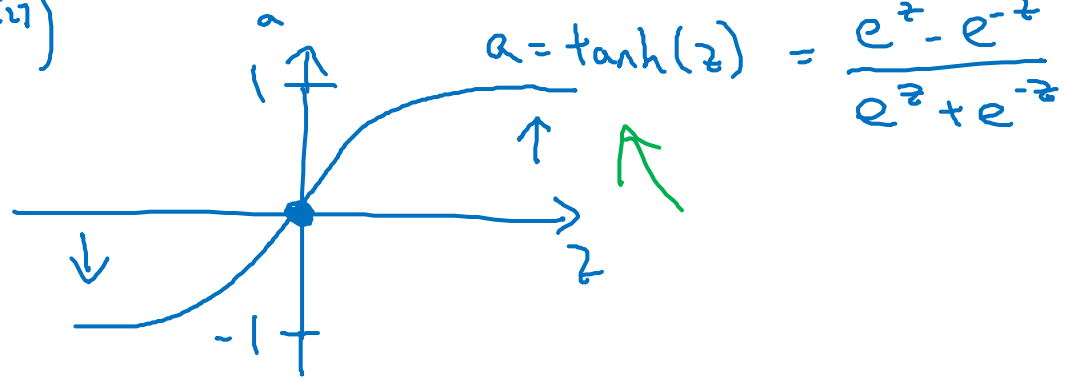
$g^{[2]}(z^{[2]}) = \sigma(z^{[2]})$

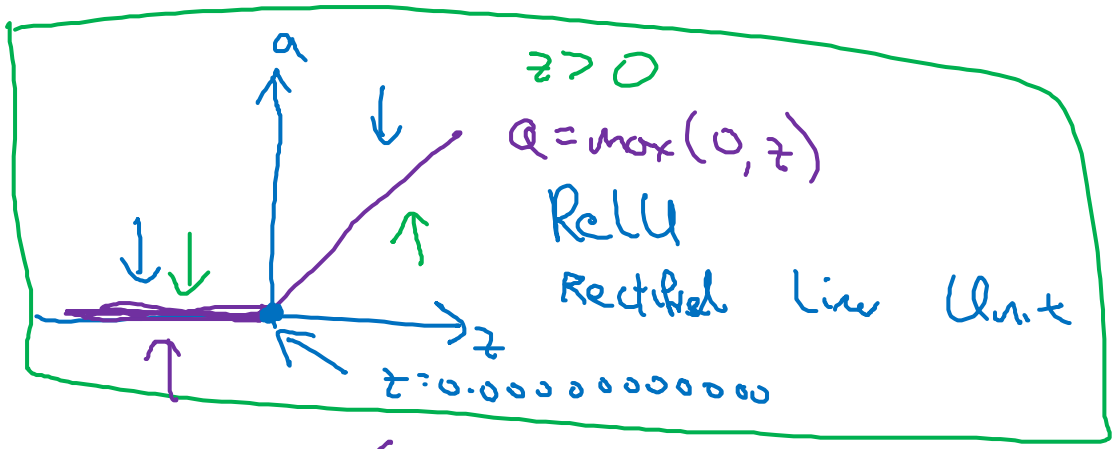
$\rightarrow y \in \{0, 1\}$

$0 \le \hat{y} \le 1$

$-1 \quad 1$







Given x:

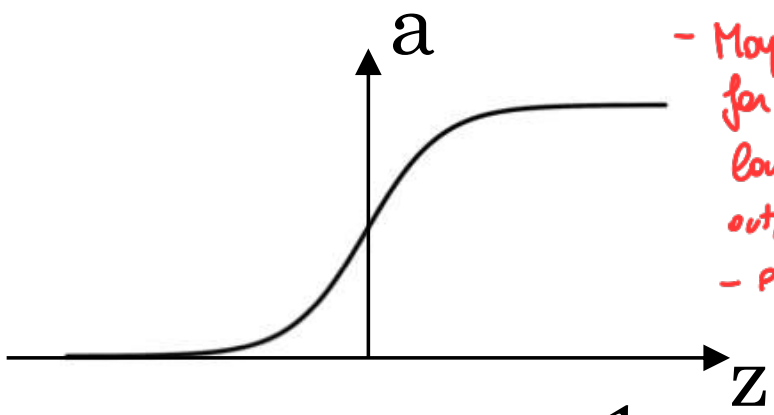
$$z^{[1]} = W^{[1]}x + b^{[1]}$$

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

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

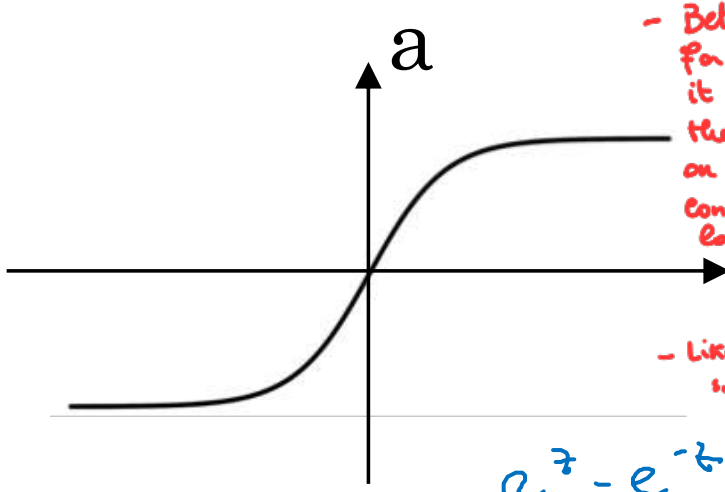
$$a^{[2]} = \sigma(z^{[2]}) \quad g^{[2]}(z^{[2]})$$

Pros and cons of activation functions



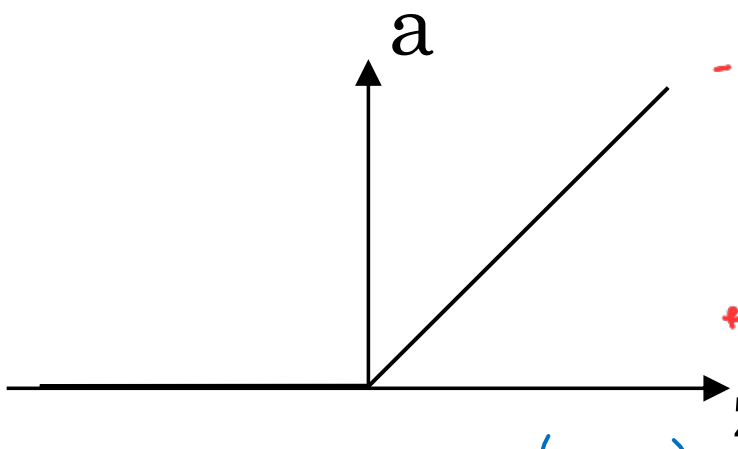
sigmoid: $a = \frac{1}{1+e^{-z}}$

- May be useful for the output layer, if the output is in $\{0,1\}$.
- PROBLEM: "VANISHING GRADIENT" It causes the update of the neuron's parameters to get stuck for large or small inputs (for which the gradient, and therefore the update, is close to zero).



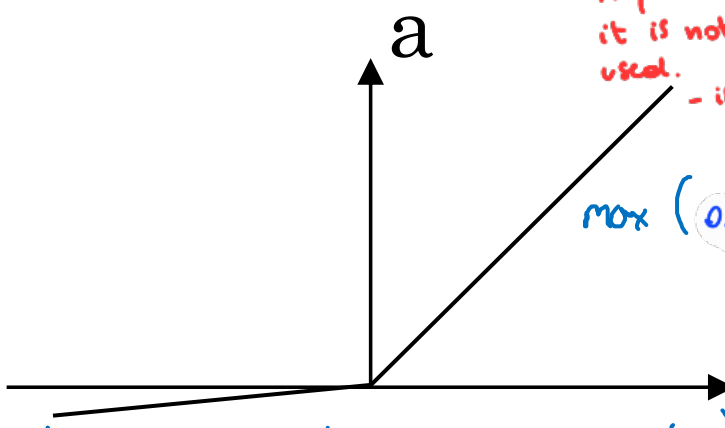
- Better than sigmoid for hidden layers because it produces activations that are close to zero on average, which simplifies convergence of following layers. (not explained)
- Like the sigmoid it suffers from the vanishing gradient problem.

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



- It is the default activation function and works better than σ and tanh because it reduces the vanishing gradient problem (the gradient is nonzero for a larger range of inputs).
- produces a sparser activation than sigmoids.
- less computationally demanding than sigmoid.

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



- May be better than ReLu but it is not very commonly used.
- it helps in case too many neurons die with ReLu.

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

Andrew Ng

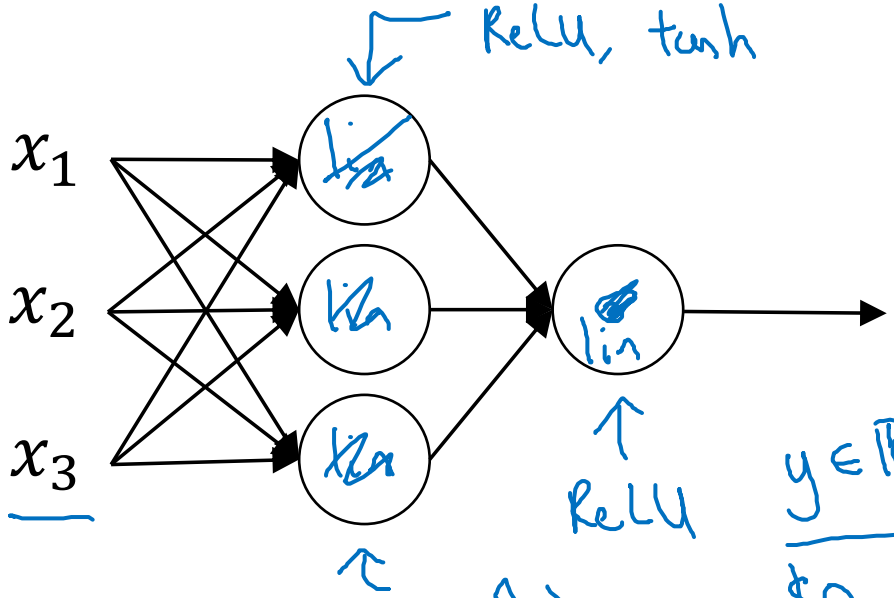
Because otherwise the NN computes a linear function overall...
(the composition of linear functions is a linear function).

deeplearning.ai

One hidden layer Neural Network

Why do you need non-linear activation functions?

Activation function



Given x:

$$\rightarrow \begin{cases} z^{[1]} = W^{[1]}x + b^{[1]} \\ a^{[1]} = g^{[1]}(z^{[1]}) \end{cases} z^{[1]}$$

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

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

$g(z) = z$

"linear activation function"

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

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

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

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

$$= W'x + b'$$

$g(z) = z$

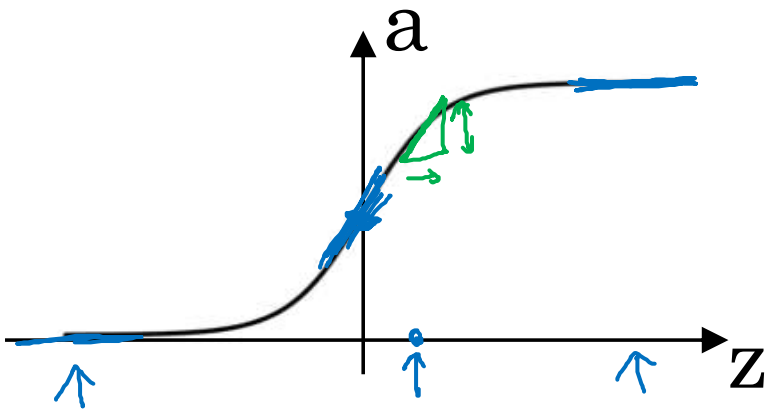
Andrew Ng

deeplearning.ai

One hidden layer Neural Network

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 $g(x)$ at z

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

$= g(z)\,(1 - g(z))$ ←

$= a\,(1-a)$

being $a = g(z)$

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

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

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

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

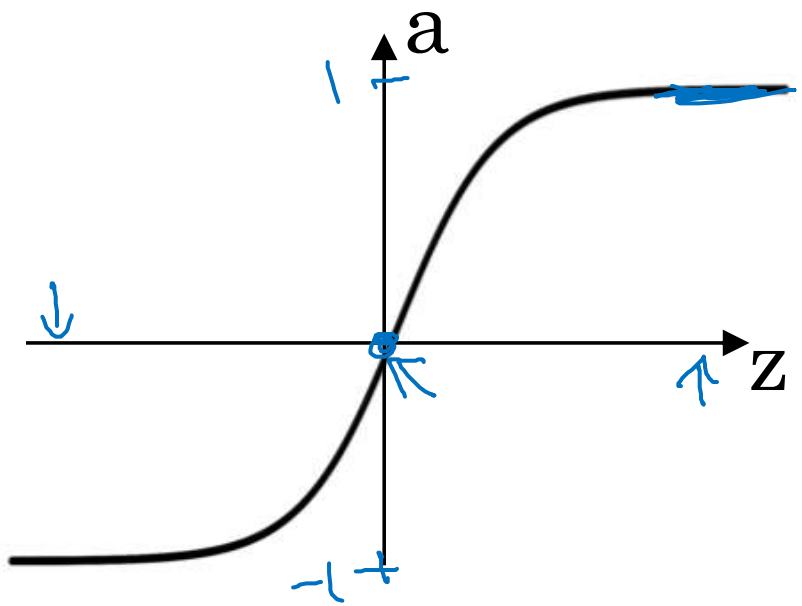
$\frac{d}{dz} g(z) \approx 0 \cdot (1-0) \approx 0$

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

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

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

$= 1 - (\tanh(z))^2$ ←

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

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

$g'(z) \approx 0$

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

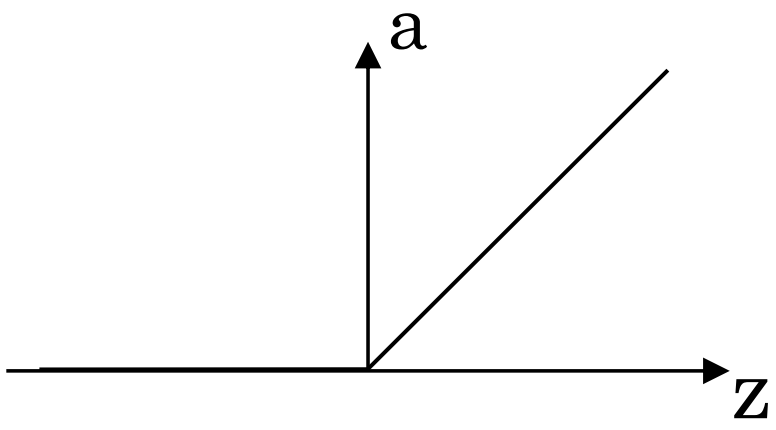
$g'(z) \approx 0$

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

$g'(z) = 1$

Andrew Ng

ReLU and Leaky ReLU



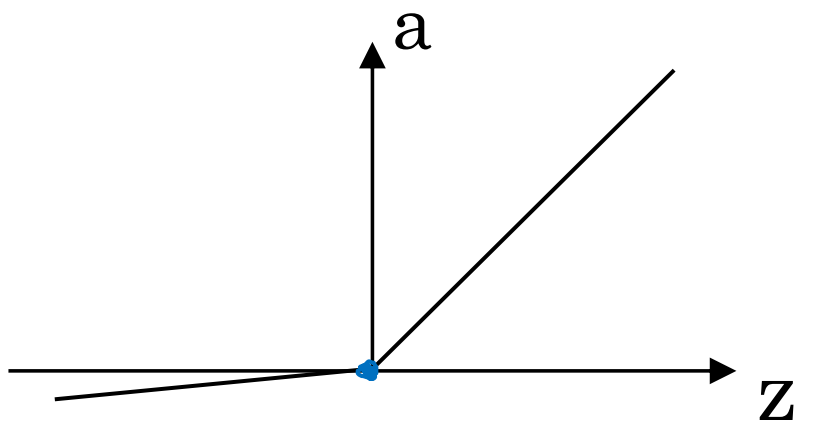
ReLU

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

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

$z = 0.0000\ldots0$

mathematically the derivative is not defined in $z=0$. In practice, however, the chance to have z exactly $=0$ is so low that we can approximate this with the $\geq$ sign



Leaky ReLU

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

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

deeplearning.ai

One hidden layer Neural Network

Gradient descent for neural networks

Gradient descent for neural networks

Parameters: $W^{[1]}$ $(n^{[1]}, n^{[0]})$, $b^{[1]}$ $(n^{[1]}, 1)$, $W^{[2]}$ $(n^{[2]}, n^{[1]})$, $b^{[2]}$ $(n^{[2]}, 1)$

Activations dimensions: $n_x = n^{[0]}$, $n^{[1]}$, $n^{[2]} = 1$

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

Gradient descent: (VECTORIZED FORM)

→ Repeat until convergence {

→ Compute predicts $(\hat{y}^{(i)}, i = 1, \dots, m)$

$dW^{[1]} = \frac{\partial J}{\partial W^{[1]}}$, $db^{[1]} = \frac{\partial J}{\partial b^{[1]}}$, $dW^{[2]}$, $db^{[2]}$] compute derivatives

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

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

$W^{[2]} := \dots$ $b^{[2]} := \dots$] update the parameters

}

Andrew Ng

Formulas for computing derivatives

Forward propagation:

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

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

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

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

Back propagation:

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

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

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

$(n^{[2]}, m)$

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

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

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

$(n^{[1]}, m)$ $\quad$ $(n^{[1]}, m)$

$*$ element-wise product

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

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

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

reshape

Andrew Ng

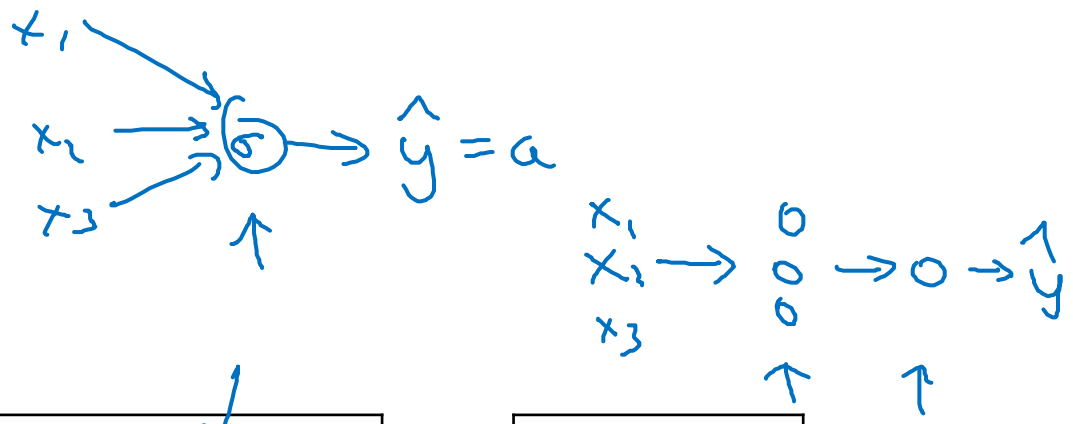
deeplearning.ai

One hidden layer Neural Network

Backpropagation intuition (Optional)

Computing gradients

Logistic regression



x, w, b → $z = w^T x + b$ → $a = \sigma(z)$ → $\mathcal{L}(a, y)$

$dz = a - y$

$dw = dz \cdot x$

$db = dz$

$da = \frac{d}{da}\mathcal{L}(a,y) = -y\log a - (1-y)\log(1-a)$

$= -\frac{y}{a} + \frac{1-y}{1-a}$

$dz = da \cdot g'(z)$

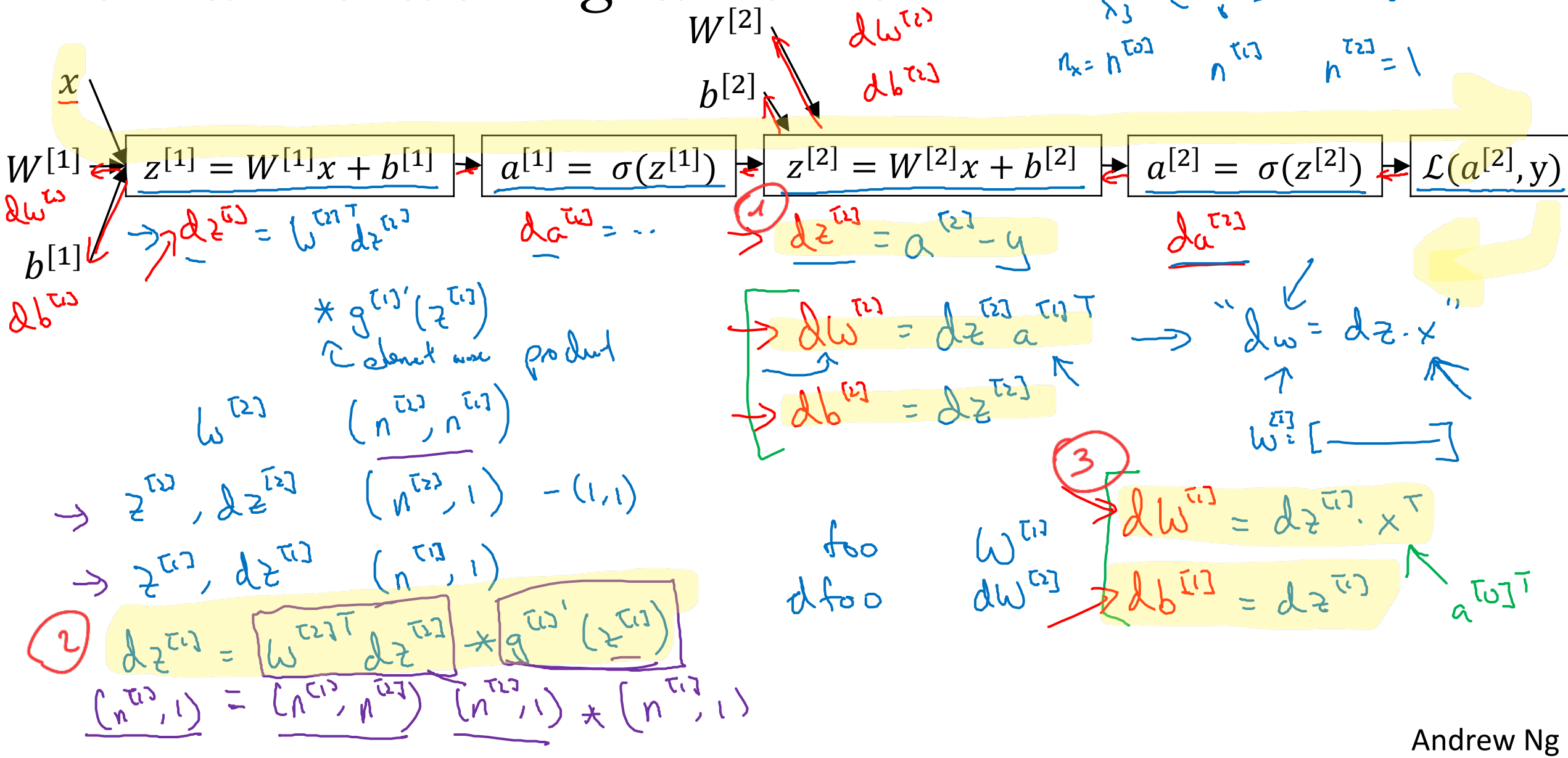
$g(z) = \sigma(z)$

$\frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{da} \cdot \frac{da}{dz}$

"$dz = da$"

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

Neural network gradients



Andrew Ng

Summary of gradient descent

$$dz^{[2]} = a^{[2]} - y$$

$$dW^{[2]} = dz^{[2]} a^{[1]T}$$

$$db^{[2]} = dz^{[2]}$$

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

$$dW^{[1]} = dz^{[1]} x^{T}$$

$$db^{[1]} = dz^{[1]}$$

Vectorized Implementation:

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

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

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

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

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

Andrew Ng

Summary of gradient descent

Vectorized implementation

$$dz^{[2]} = a^{[2]} - y$$

$$dW^{[2]} = dz^{[2]} a^{[1]T}$$

$$db^{[2]} = dz^{[2]}$$

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

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

$$dW^{[1]} = dz^{[1]} x^T$$

$$db^{[1]} = dz^{[1]}$$

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

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

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

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

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

elementwise product

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

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

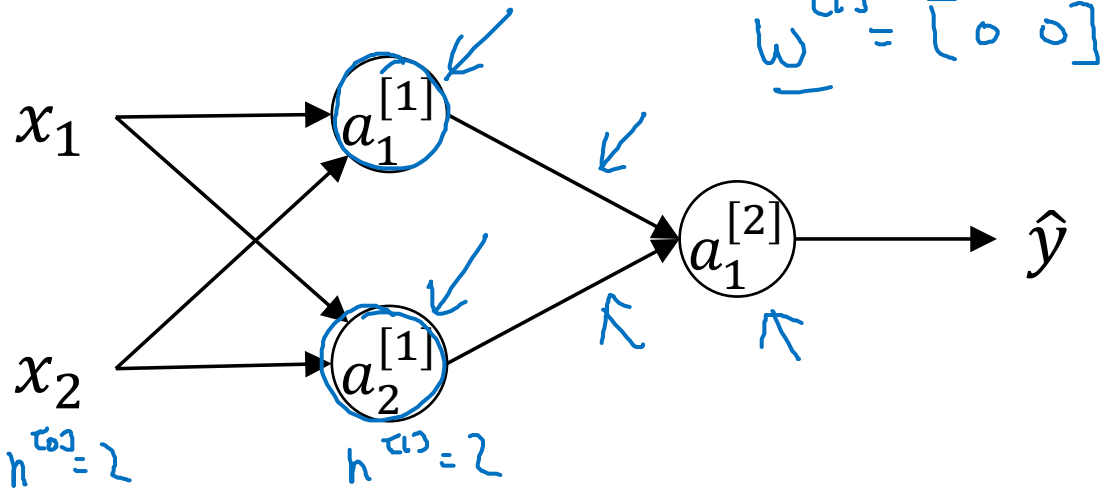
$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$$

deeplearning.ai

One hidden layer Neural Network

Random Initialization

What happens if you initialize weights to zero?



$w^{[1]} = [0 \ 0]$

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

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

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

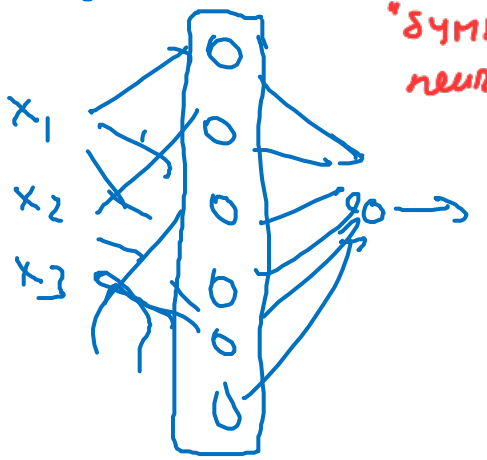
$$dw = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$$

$$w^{[1]} = w^{[1]} - \alpha\, dw$$

- Every neuron in the same layer would have the same activation
  => they would therefore be updated in the same way
  => they would therefore always be identical.

Symmetric

- One talks about "SYMMETRY" of the neurons activations

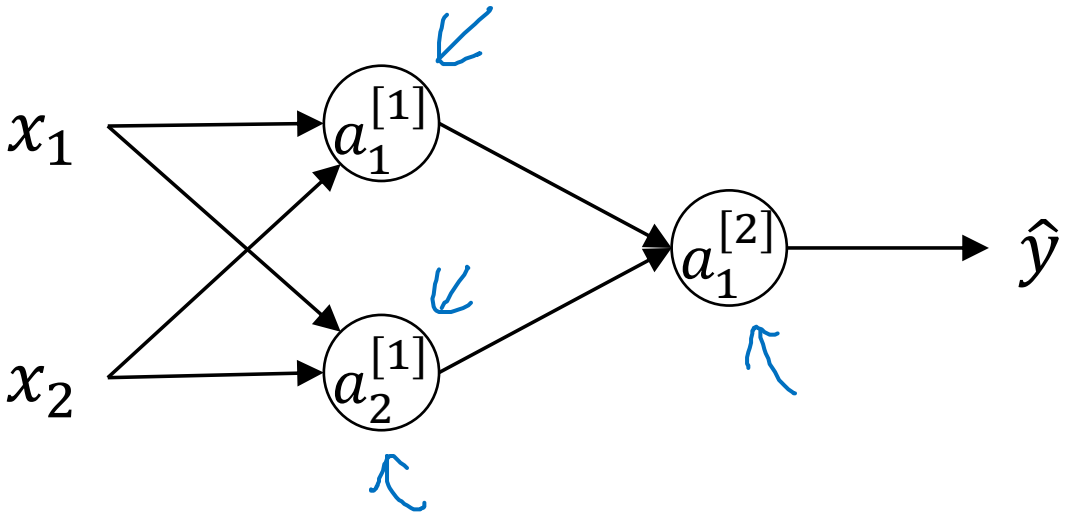


$$w^{[1]} = \begin{bmatrix} \cdots & \cdots \\ \cdots & \cdots \end{bmatrix}$$

Andrew Ng

Random initialization

- Random initialization breaks the symmetry of the neurons activations.
- It's often good to scale the random initializations to make them small enough to avoid vanishing gradients (if sigmoid activations are used). This is especially important in shallow networks.



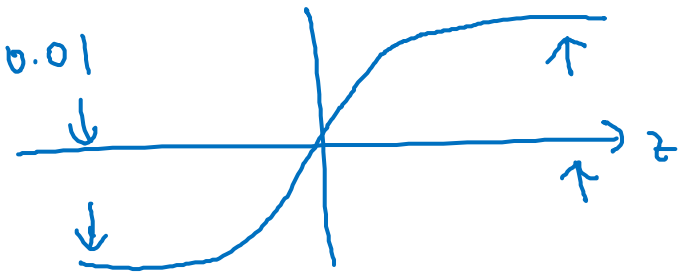
$\rightarrow W^{[1]} = np.random.randn((2,2)) * 0.01$

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

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

$b^{[2]} = 0$

100?



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

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

Andrew Ng