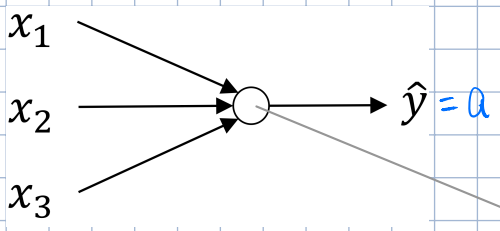
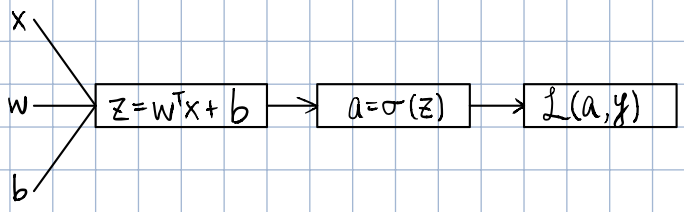
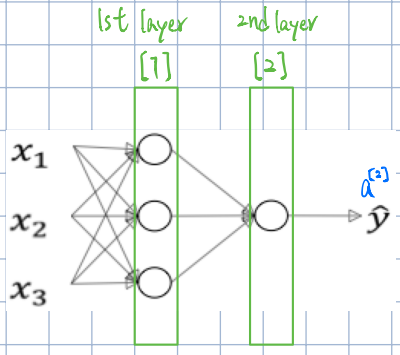
Neural Networks Overview:





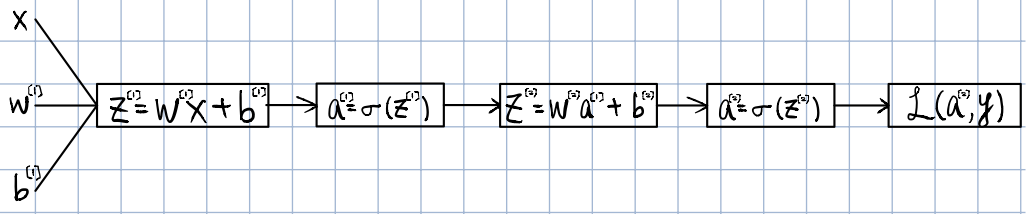
This node does two things: $z = w^T x + b$ and $a = \sigma(z)$



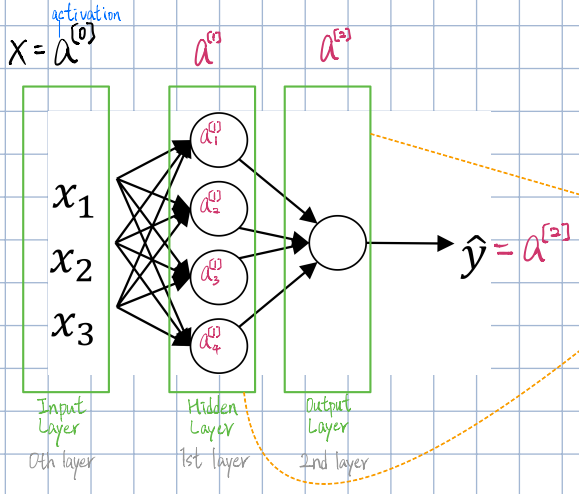
Actually every node in this graph can also do these two things.

$z = w^T x + b$

$a = \sigma(z)$



Neural Network Representation: One hidden Layer



Conventionally, it's called 2 layer neural network. The input layer is not counted.

The associated parameters for the hidden layer and output layer:

$w^{[2]}: 1 \times 4$, $b^{[2]}: 1 \times 1$

$w^{[1]}: 4 \times 3$, $b^{[1]}: 4 \times 1$

Standard notations for Deep Learning

This document has the purpose of discussing a new standard for deep learning mathematical notations.

1 Neural Networks Notations.

General comments:

· superscript (i) will denote the i^{th} training example while superscript [l] will denote the l^{th} layer

Sizes:

·m : number of examples in the dataset

·n_x : input size

·n_y : output size (or number of classes)

·$n_h^{[l]}$: number of hidden units of the l^{th} layer

In a for loop, it is possible to denote $n_x = n_h^{[0]}$ and $n_y = n_h^{[\text{number of layers } +1]}$.

·L : number of layers in the network.

Objects:

·$X \in \mathbb{R}^{n_x \times m}$ is the input matrix

·$x^{(i)} \in \mathbb{R}^{n_x}$ is the i^{th}example represented as a column vector

·$Y \in \mathbb{R}^{n_y \times m}$ is the label matrix

·$y^{(i)} \in \mathbb{R}^{n_y}$ is the output label for the i^{th} example

·$W^{[l]} \in \mathbb{R}^{\text{number of units in next layer} \times \text{number of units in the previous layer}}$ is the weight matrix,superscript [l] indicates the layer

·$b^{[l]} \in \mathbb{R}^{\text{number of units in next layer}}$ is the bias vector in the l^{th} layer

·$\hat{y} \in \mathbb{R}^{n_y}$ is the predicted output vector. It can also be denoted $a^{[L]}$ where L is the number of layers in the network.

Common forward propagation equation examples:

$a = g^{[l]}(W_x x^{(i)} + b_1) = g^{[l]}(z_1)$ where $g^{[l]}$ denotes the l^{th} layer activation function

$\hat{y}^{(i)} = softmax(W_h h + b_2)$

· General Activation Formula: $a_j^{[l]} = g^{[l]}(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}) = g^{[l]}(z_j^{[l]})$

· $J(x, W, b, y)$ or $J(\hat{y}, y)$ denote the cost function.

Examples of cost function:

· $J_{CE}(\hat{y}, y) = -\sum_{i=0}^{m} y^{(i)} \log \hat{y}^{(i)}$

· $J_1(\hat{y}, y) = \sum_{i=0}^{m} \mid y^{(i)} - \hat{y}^{(i)} \mid$

2 Deep Learning representations

For representations:

· nodes represent inputs, activations or outputs

· edges represent weights or biases

Here are several examples of Standard deep learning representations

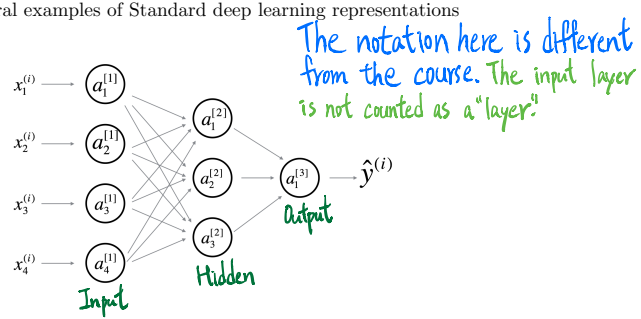


Figure 1: Comprehensive Network: representation commonly used for Neural Networks. For better aesthetic, we omitted the details on the parameters ($w_{ij}^{[l]}$ and $b_i^{[l]}$ etc...) that should appear on the edges

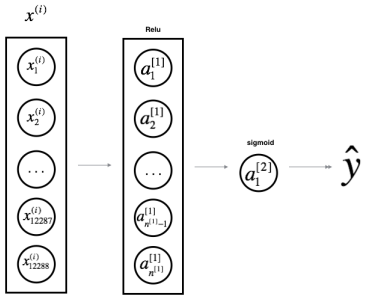
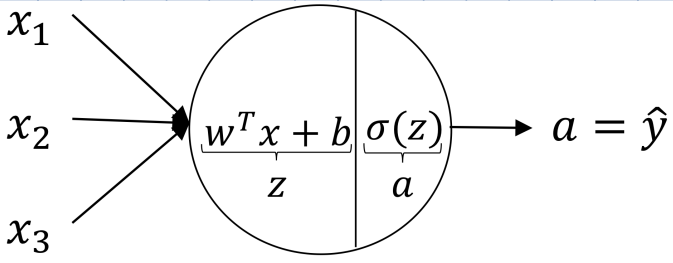


Figure 2: Simplified Network: a simpler representation of a two layer neural network, both are equivalent.

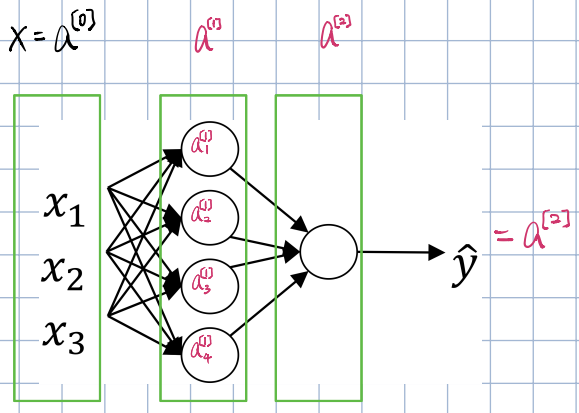
Computing a Neural Network's Output:



One node has two steps.

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

$$a = \sigma(z)$$



$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$

$$\vdots$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

Thus, $z = W^T x + b$, W^T: 4×3

X: 3×1

b: 4×1

$a^{[1]}$: 4×1

Vectorize across Multiple Samples.

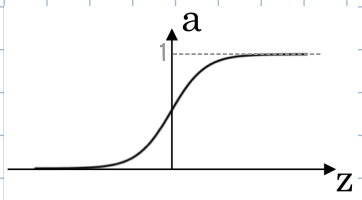
Do Not use for-loop.
Use vectorization.

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

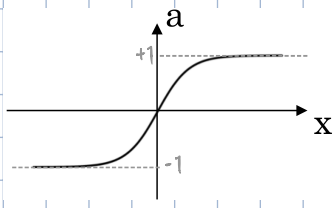
$$\vdots$$

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

Activation functions:



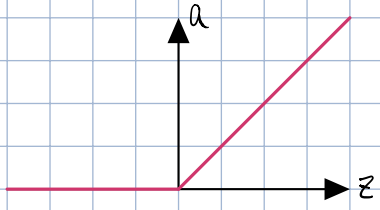
Sigmoid func: $a = \frac{1}{1+e^{-z}}$



Hyperbolic tangent: $a = \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

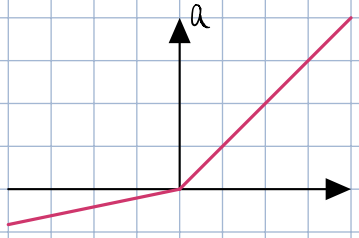
The main drawback of the sigmoid function and hyperbolic tanget is when z is very large or small, then the gradient/slope of sig() and tanh() becomes being close to 0, which slows down gradient descent. Thus, we need ReLU.

Actually, in practice, sigmoid function is almost never used. Except for the output layer of binary classification. Explain Why.



ReLU
Rectified Linear Unit

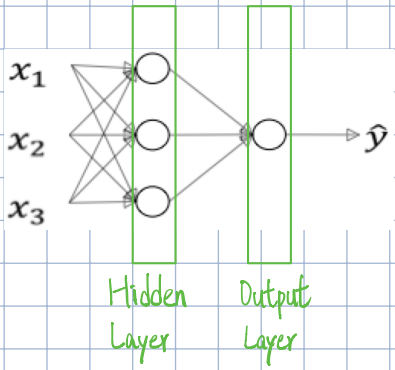
$a = \max(0, z)$



Leaky ReLU

$a = \max(0.00|z, z)$

Why Non-Linear Activation functions:



Given the network and X, we have

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

$a^{[1]} = g^{[1]}(z^{[1]})$, $g^{[1]}$ is the activiation func. of the hidden layer.

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

$a^{[2]} = g^{[2]}(z^{[2]})$, $g^{[2]}$ is the activiation func. of the output layer.

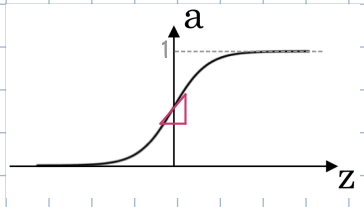
Now, we select a linear function, an identity function, as the activiation function.
Thus, $g(z) = z$, $a^{[1]} = z^{[1]}$, $a^{[2]} = z^{[2]}$
Then we can calculate $\hat{y}$.

$$\hat{y} = a^{[2]} = z^{[2]} = W^{[2]}(W^{[1]}X + b^{[1]}) + b^{[2]}$$
$$= (W^{[2]}W^{[1]})X + (b^{[2]}b^{[1]})$$
$$= W'X + b'$$

This shows that the neural network is generating a linear function of an input, which is equal to no hidden layers at all.

Now, linear activiation function may be used in one case, which is the output layer of a regression problem. E.g., we want to predict houses' price $y \in \mathbb{R}$, $0 < y < 10^6$\$. Or the disparity map of a stereo pair image, $-10 < y < 10$. But all hidden layers should still use non-linear activiation function.

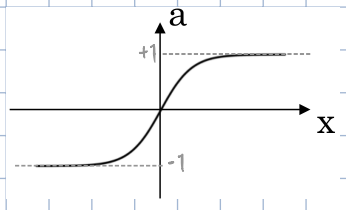
Derivatives of Activation functions:



Sigmoid func:

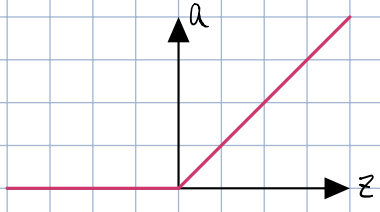
$$g(z)=\frac{1}{1+e^{-z}},\ \frac{d}{dz}g(z)=\frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right)=g(z)(1-g(z))$$

The other notation: $g'(z)=\frac{d}{dz}g(z)$, $g(z)=a$, Thus, $g'(z)=a(1-a)$



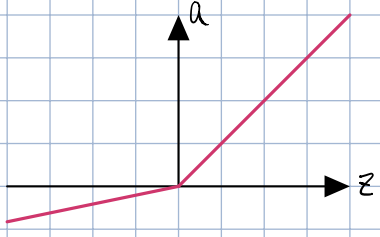
Hyperbolic Tangent: $g(z)=\tanh(z)=\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$

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



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

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



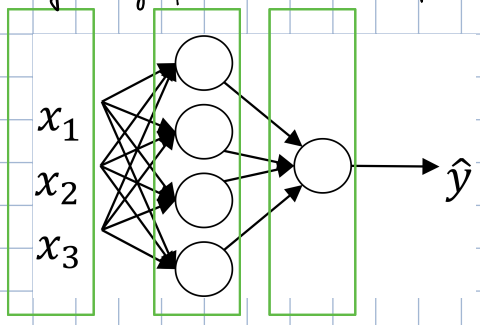
Leaky ReLU: $g(z)=\max(0.001z,z)$

$a=\max(0.001z,z)$

$$g'(z)=\begin{cases}0.001, & \text{if } z<0\\ 1, & \text{if } z>0\\ \text{undefined}, & \text{if } z=0\end{cases}$$

Gradient Descent for Neural Networks:

Using the graph below as an example:



n_x: Number of nodes in each layer. $n_x: n^{[0]}=3, n^{[1]}=4, n^{[2]}=1$.

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

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

Gradient Descent:

Repeat {

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

$dw^{[1]} = \frac{dJ}{dw^{[1]}}$, $db^{[1]} = \frac{dJ}{db^{[1]}}$, $dw^{[2]} = \frac{dJ}{dw^{[2]}}$, $db^{[2]} = \frac{dJ}{db^{[2]}}$

$w^{[1]} = w^{[1]} - \alpha dw^{[1]}$, $b^{[1]} = b^{[1]} - \alpha db^{[1]}$, $w^{[2]} = w^{[2]} - \alpha dw^{[2]}$, $b^{[2]} = b^{[2]} - \alpha db^{[2]}$

}

Now, the question is how to find out dw and db ? Follow the formula below: Note that the capital letter denotes the vectorised components: Eg. $Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}]$

Forward Propagation:

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

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

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

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

sigmoid function

Backward Propagation:

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

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

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

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

$n^{[1]} \times m$ $\quad$ $n^{[1]} \times m$

$*$: Element-wise product

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

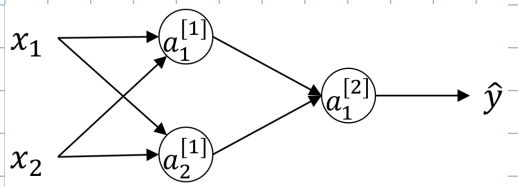
$db^{[1]} = \frac{1}{m} \text{sum}(dZ^{[1]}, \text{axis}=1)$

$n^{[1]} \times 1$

Please explain/derive the formula above.
Check C1W3L10

Random Initialization of Weights

What happens if I initialize weights to zero?



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

$w^{[1]}=\begin{bmatrix}0 & 0\\ 0 & 0\end{bmatrix}, \quad b^{[1]}=\begin{bmatrix}0\\ 0\end{bmatrix}, \quad w^{[2]}=[0 \ 0]$

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

$dz_1^{[1]}=dz_2^{[1]}, \quad dw=\begin{bmatrix}u & v\\ u & v\end{bmatrix},$

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

W will end up with the same content for every row.

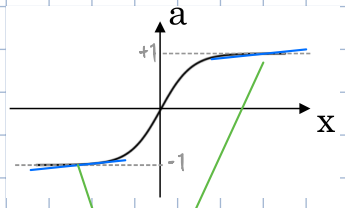
Here is the right way to initialize weights.

$w^{[1]}=\text{random}(2,2)\times 0.01$
$b^{[1]}=\text{random}(2,1)$
⋮

Why initialize weights a small value?

Recall: $z^{[1]}=w^{[1]}X+b^{[1]}$

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



The gradient is small or close to zero.

If weights $w^{[1]}$ are large value, $z^{[1]}$ will be a either very large value or a very small value. Either way, it will result to a small gradient which will make the gradient descent (learning) really slow.

The best strategy to initialize weights is probably Xaveir Initializer.