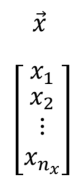
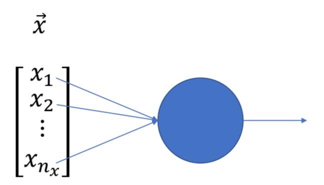
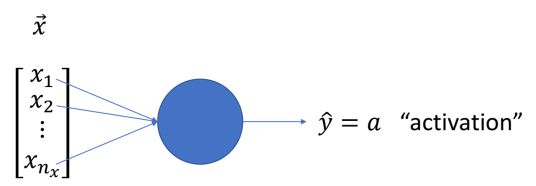
nn-overview.pdf
quinta-feira, 20 de julho de 2023 15:33

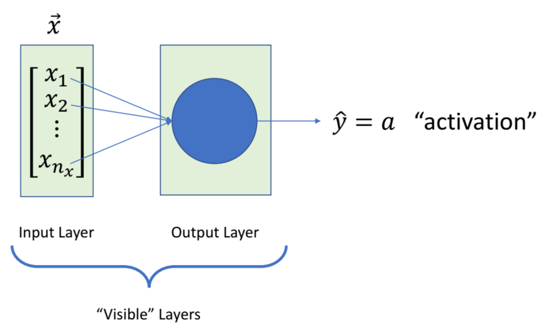
nn-overview...

Neural Networks

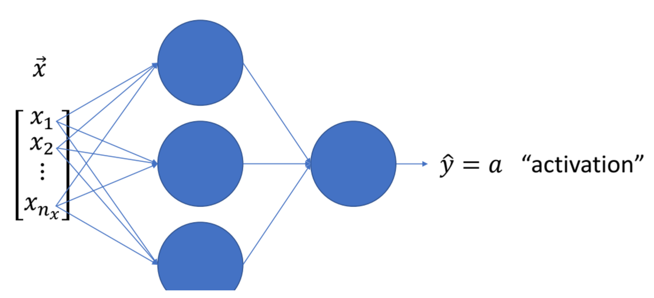








One hidden layer



* Tom Mitchele

Multilayer Networks and the Backpropagation Alg

- Usado para expandir a NN para decisões não lineares
- Função Sigmoid → Output são funções não lineares e diferenciáveis, para que possamos usar o gradiente descendente

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

→ Backpropagation

Algoritmo que usa o gradiente descendente para diminuir o squared error da rede neural.

Trabalha num mundo de rede neural de multiplos camadas, logo, precisamos atualizar o erro:

$$\boxed{E(\vec{w}) = \frac{1}{2} \sum \sum (t_{kd} - o_{kp})^2}$$

→ Se usamos o quadrado
→ Todos os possíveis outputs

→ Busca atualizar e encontrar os pesos ideais para todos da rede neural

→ Multilayer

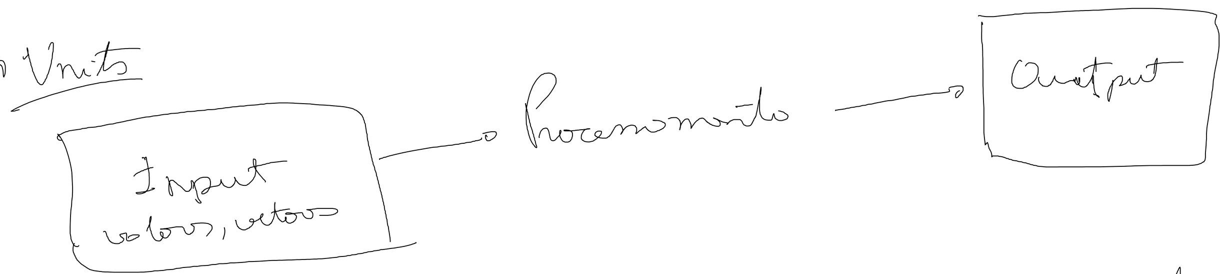
Revisão Geral

* Neural Network Cap7 Stanford

Conjunto de unidades de computação que recebem vetores e retornam valores para a proxima camada...

Deep Learning vem da ideia de várias camadas

→ Units



Uma unit se baseia na soma ponderada de inputs junto com um bias de modo que $\boxed{z = w \cdot x + b}$ isso irá retornar apenas um valor

Após isso, aplicamos uma função de ativação não linear, para evitar que tudo fique linear. Três são famosas: Sigmoid; tanh; ReLU

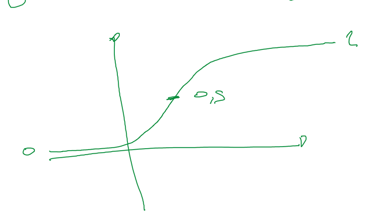
↳ É importante que essas funções sejam deriváveis

↳ Sigmoid irá mapear os resultados entre [0,1]

↳ tanh irá mapear entre -1 e 1

↳ Relu é o max (z, 0), ou seja, z se positivo, zero se negativo

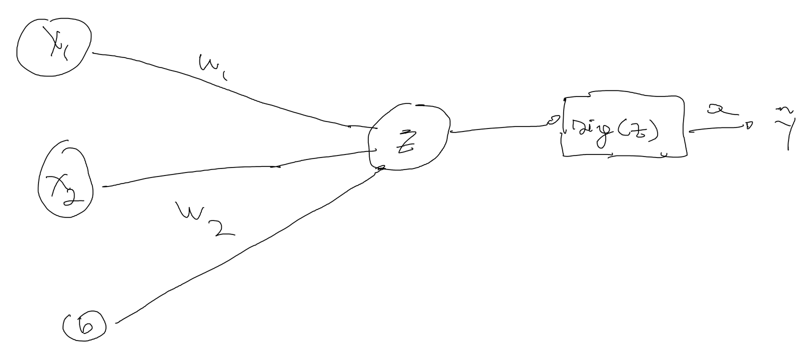
Sigmoid $= \frac{1}{1 + e^{-z}}$

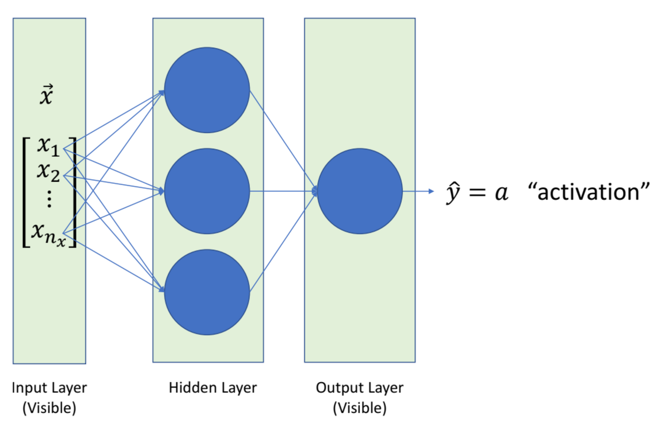


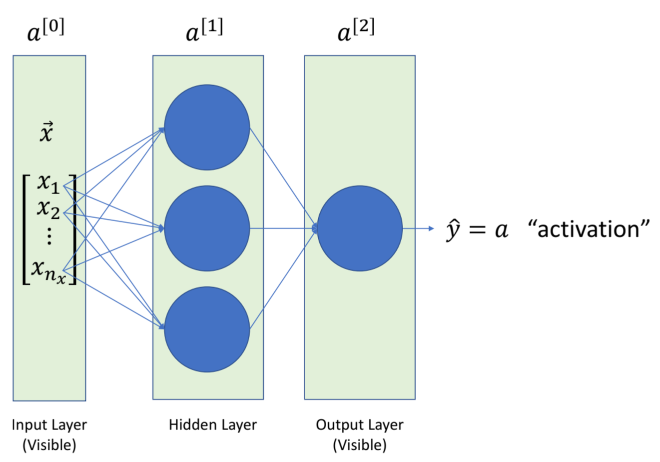
→ Cada função de ativação tem caracteristicas próprias que podem ajudar em certos contextos. Por isso ter camadas com diferentes funções de ativação é interessante

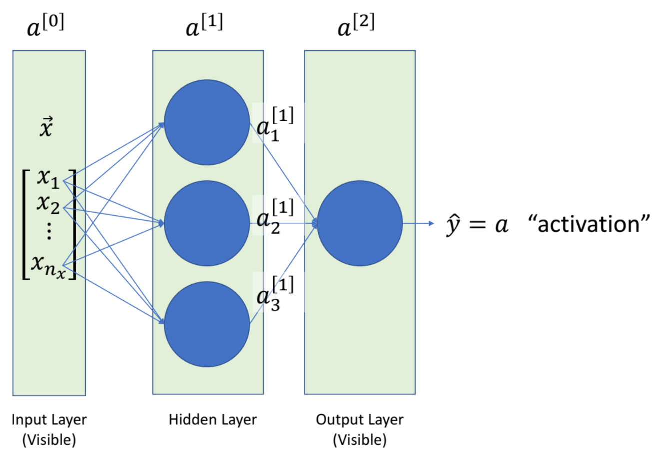
Algumas são facilmente diferenciaveis, etc

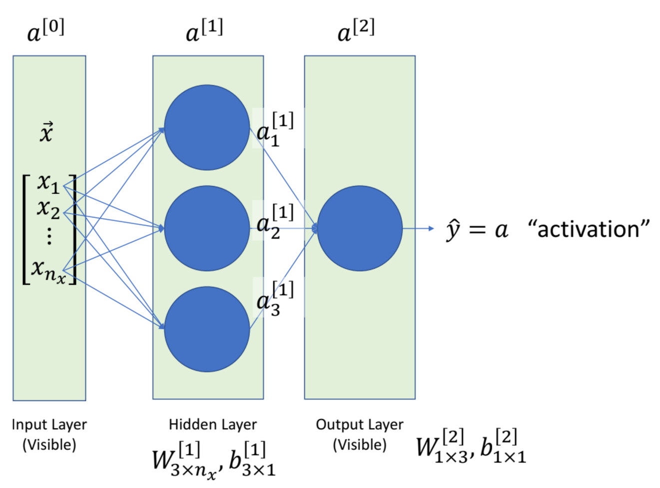
→ Computational Graph



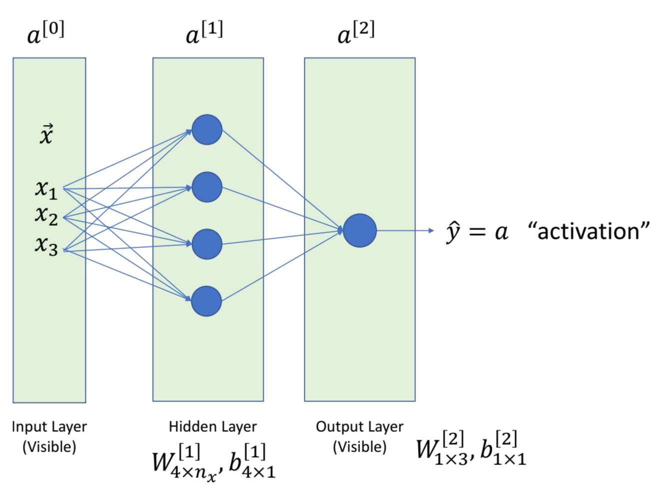


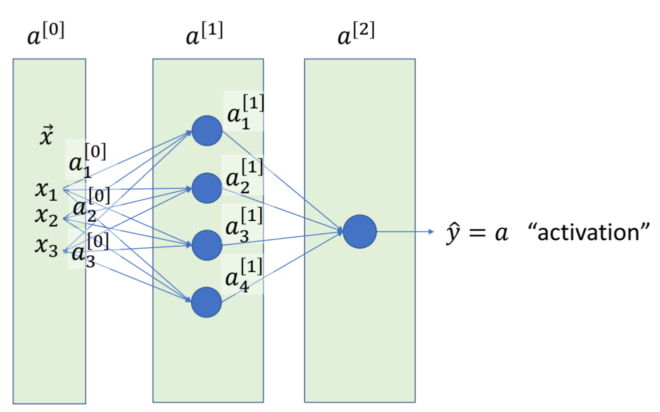






Computing an NN's output





→ Feedforward Propagation NN

Multilayer Perceptron

↳ Basicamente existem três partes

- Input
- Hidden layers = Podem ser diversas
- Output layers

→ Cada hidden layer irá fazer a soma ponderada das entradas e aplicar uma função de ativação não-linear

↳ Cada unit de cada layer tem como input todo o resto da rede

→ Cada layer tem como entrada um vetor w_x. A matriz W é a composição de todos os W_x

↳ por isso a entrada é a matriz W, porque são todos os W_x

→ Bias se torna um vetor.

→ A partir disso, você consegue sempre calcular de todos

→ A hidden layer vai ter geralmente esse output: $h = f(W_x + b)$, sendo "f" a função de ativação

↳ Basicamente vai aplicar p/ todo mundo $W_1X_1 + b_1$, $W_2X_n + b_2$ e etc

→ Pseudo-código para uma feedforward

for i in 1...n:

$z^{[i]} = W^{[i]} a^{[i-1]} + b^{[i]}$ ↳ bias e weight da atual

(entrada da camada anterior)

$a^{[i]} = g^{[i]}(z^{[i]})$ ↳ função de ativação

(Resultado da camada atual)

$\underbrace{\hat{y}}_{} = a^{[n]}$

↳ Predições da rede neural!

* Treinando Redes Neurais

O objetivo é ajustar os pesos "W" e o bias "b" para obtermos um $\hat{y}$ mais próximo do valor real "y"

Etapas:
- Loss Function: Calcula o erro entre $\hat{y}$ e y
- Gradient Descent: Encontra os parametros que minimizam a loss function
- Back-propagation: Achamos as derivadas parciais p/ o gradiente

* Andrew Ng

→ Vectorization

Forma de tirar os for loops do código, ajudando a rodar em grandes datasets

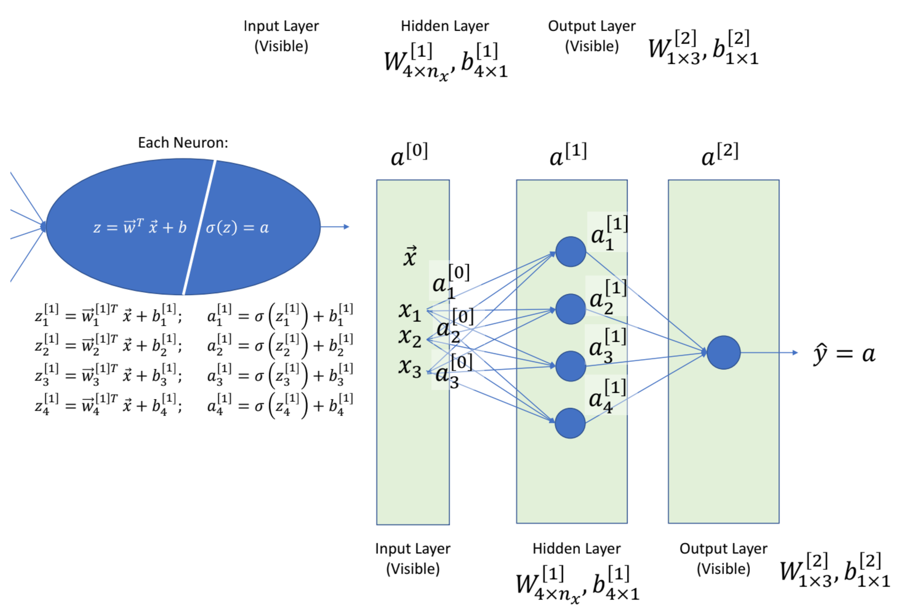
A ideia é transformar as coisas em operações de matriz, que são realizadas mais rapidamente até em gpu's. Todas as operações tem que ser feitas em numpy como realmente vectors/arrays/matrizes

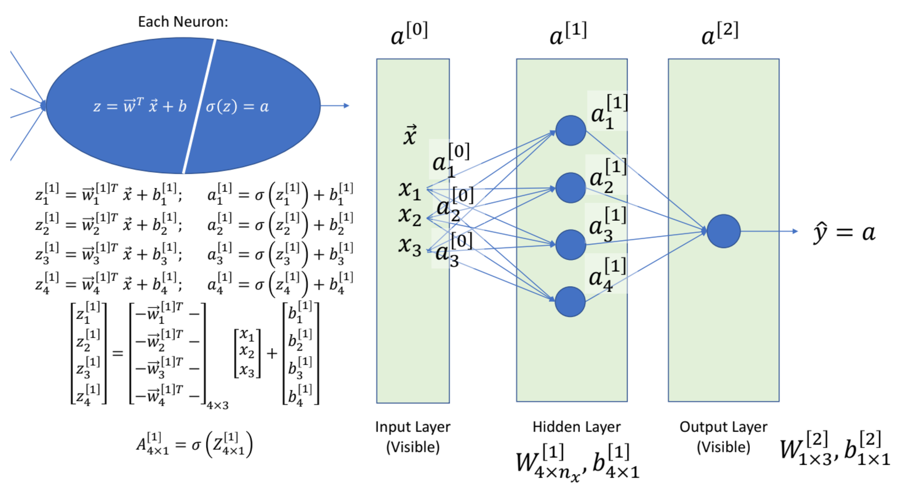
é feito em GPU, a CPU também tem

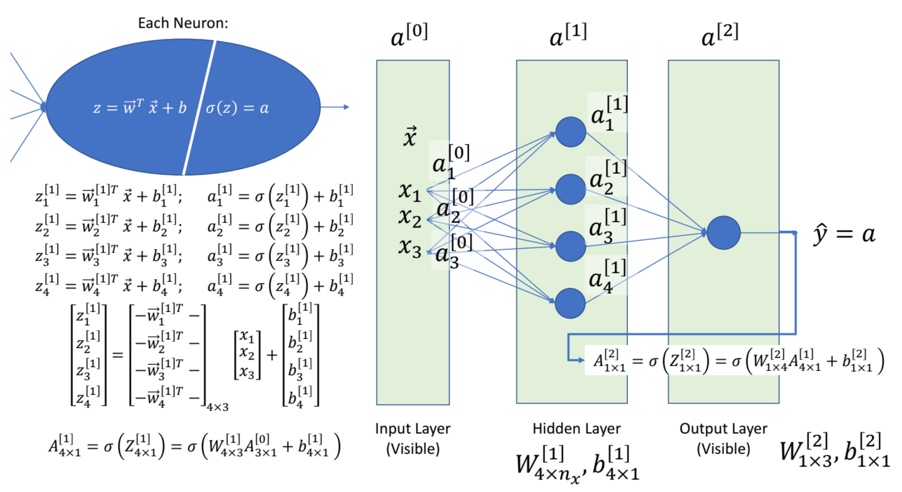
↳ CPU x GPU ⇝ Nem tudo x ...
alguns algoritmos de paralelismo que o numpy usa

Já é o que fazemos no numpy, evitando de usar for loops e sim realizando operações vetorizadas mais performáticas

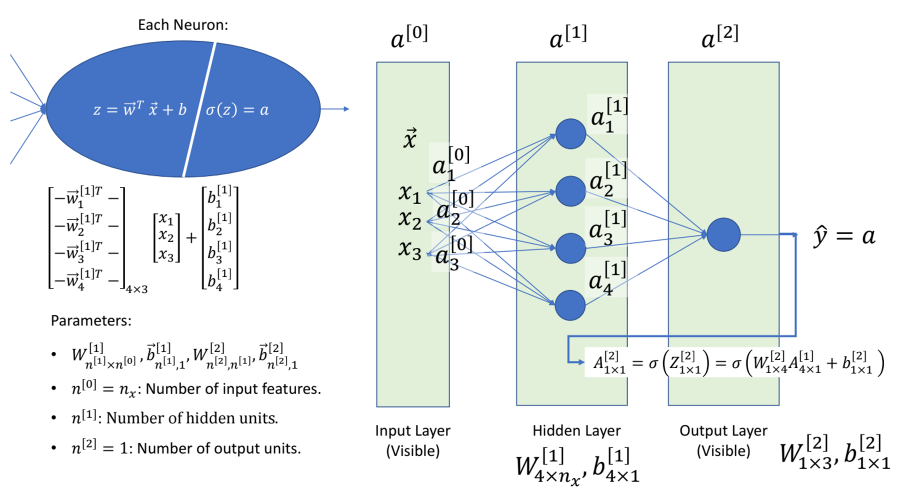
* NN Computing

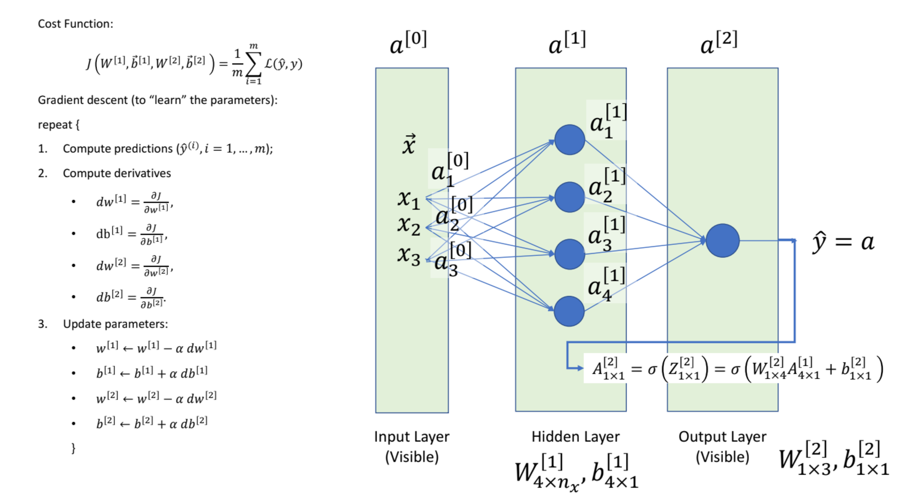


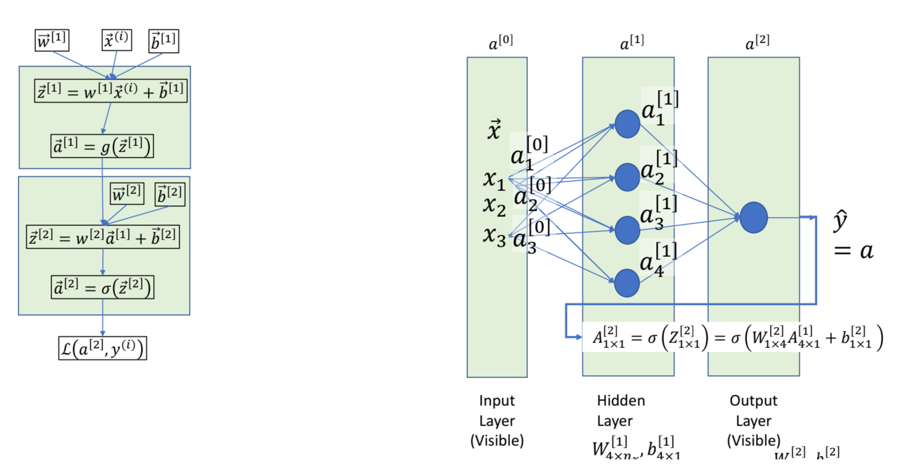




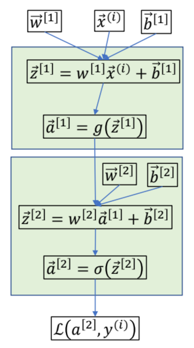
Gradient descent for NNs
(one hidden layer)

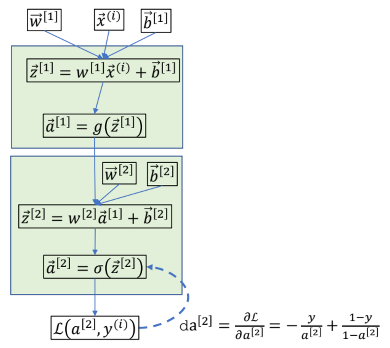


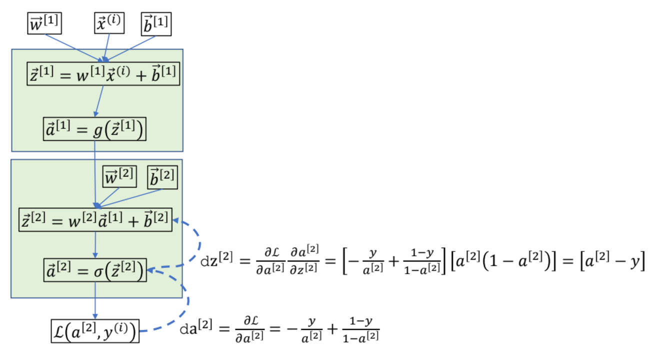


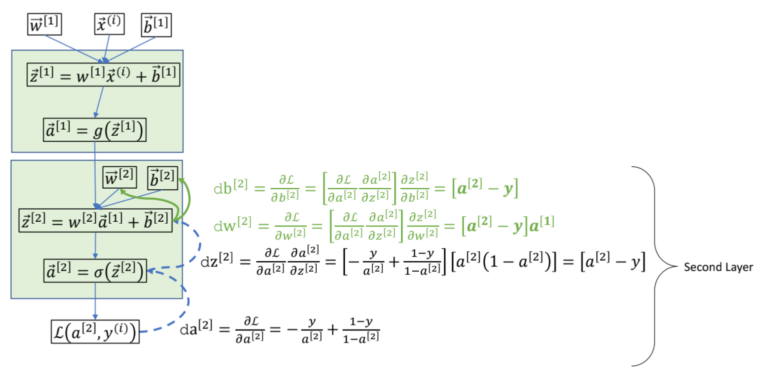


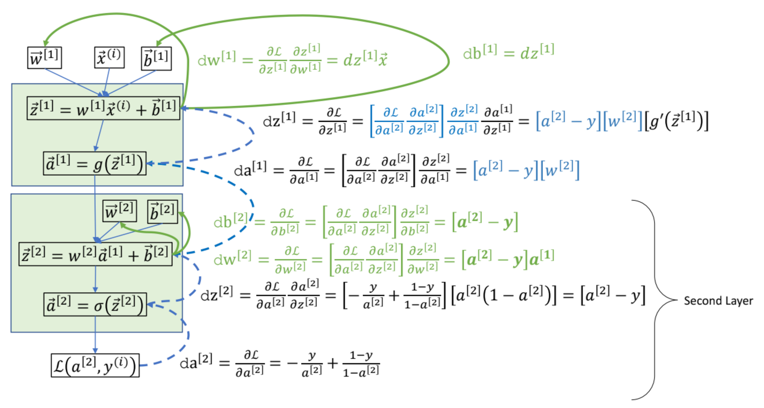
$w_{1\times3}, b_{1\times1}$











Six key equations

1. $\mathrm{dz}^{[2]} = \frac{\partial \mathcal{L}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}} = \left[-\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}\right]\left[a^{[2]}\left(1-a^{[2]}\right)\right] = \left[\boldsymbol{a}^{[2]} - \boldsymbol{y}\right]$
2. $\mathrm{dw}^{[2]} = \frac{\partial \mathcal{L}}{\partial w^{[2]}} = \left[\frac{\partial \mathcal{L}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\right]\frac{\partial z^{[2]}}{\partial w^{[2]}} = \left[\boldsymbol{a}^{[2]} - \boldsymbol{y}\right]\boldsymbol{a}^{[1]}$
3. $\mathrm{db}^{[2]} = \frac{\partial \mathcal{L}}{\partial b^{[2]}} = \left[\frac{\partial \mathcal{L}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\right]\frac{\partial z^{[2]}}{\partial b^{[2]}} = \left[\boldsymbol{a}^{[2]} - \boldsymbol{y}\right]$
4. $\mathrm{dz}^{[1]} = \frac{\partial \mathcal{L}}{\partial z^{[1]}} = \left[\frac{\partial \mathcal{L}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\right]\frac{\partial z^{[2]}}{\partial a^{[1]}}\frac{\partial a^{[1]}}{\partial z^{[1]}} = \left[\boldsymbol{a}^{[2]} - \boldsymbol{y}\right]\left[\boldsymbol{w}^{[2]}\right]\left[\boldsymbol{g}'\left(\vec{\boldsymbol{z}}^{[1]}\right)\right]$
5. $\mathrm{dw}^{[1]} = \frac{\partial \mathcal{L}}{\partial z^{[1]}}\frac{\partial z^{[1]}}{\partial w^{[1]}} = dz^{[1]}\vec{x} = \left[\boldsymbol{a}^{[2]} - \boldsymbol{y}\right]\left[\boldsymbol{w}^{[2]}\right]\left[\boldsymbol{g}'\left(\vec{\boldsymbol{z}}^{[1]}\right)\right]\vec{\boldsymbol{x}}$
6. $\mathrm{db}^{[1]} = dz^{[1]} = \left[\boldsymbol{a}^{[2]} - \boldsymbol{y}\right]\left[\boldsymbol{w}^{[2]}\right]\left[\boldsymbol{g}'\left(\vec{\boldsymbol{z}}^{[1]}\right)\right]$

For m samples

$$\begin{bmatrix} | & | & & | \\ \vec{z}^{[l](1)} & \vec{z}^{[l](2)} & \cdots & \vec{z}^{[l](m)} \\ | & | & & | \end{bmatrix}_{n^{[l]}\times m} = \begin{bmatrix} - & \vec{w}_1^{[l]T} & - \\ - & \vec{w}_2^{[l]T} & - \\ & \vdots & \\ - & \vec{w}_{n^{[l]}}^{[l]T} & - \end{bmatrix}_{n^{[l]}\times n^{[l]-1}} \begin{bmatrix} | & | & & | \\ \vec{a}^{[l-1](1)} & \vec{a}^{[l-1](2)} & \cdots & \vec{a}^{[l-1](m)} \\ | & | & & | \end{bmatrix}_{n^{[l-1]}\times m} + \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n^{[l]}}^{[l]} \end{bmatrix}_{n^{[l]}\times m}$$

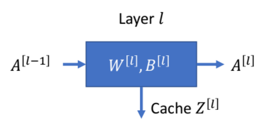
$$Z^{[l]} = W^{[l]T}X + B$$

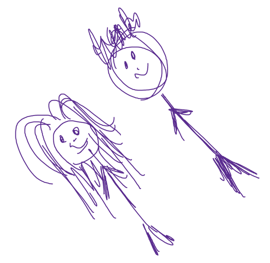
$$A^{[l]} = g^{[l]}(Z) = g^{[l]}\left(W^{[l]T}X + B\right)$$

Deep Neural Networks

DL Building Blocks

Forward:
- Input:
  - $A^{[l-1]}$.
- Compute:
  - $Z^{[l]} = W^{[l]}A^{[l-1]} + B$.
  - $A^{[l]} = g^{[l]}(Z^{[l]})$.
- Output:
  - $A^{[l]}$.
  - Cache $Z^{[l]}$.





$dz = \frac{dA}{dL}$

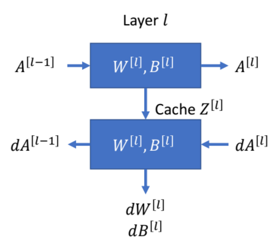
$dw = \frac{dL}{dz} = a(1-a)x$

$dB = dz$

DL Building Blocks

Forward:
- Input:
  - $A^{[l-1]}$.
- Compute:
  - $Z^{[l]} = W^{[l]}A^{[l-1]} + B$.
  - $A^{[l]} = g^{[l]}(Z^{[l]})$.
- Output:
  - $A^{[l]}$.
  - Cache $Z^{[l]}$.

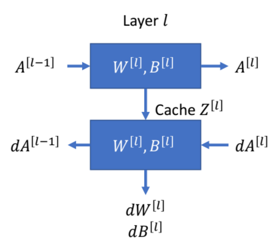
Backward:
- Input:
  - $dA^{[l]}$.
  - Cache $Z^{[l]}$.
- Output:
  - $dA^{[l-1]}$.
  - $dW^{[l]}$.
  - $dB^{[l]}$.



DL Building Blocks

Forward:
- Input:
  - $A^{[l-1]}$.
- Compute:
  - $Z^{[l]} = W^{[l]}A^{[l-1]} + B$.
  - $A^{[l]} = g^{[l]}(Z^{[l]})$.
- Output:
  - $A^{[l]}$.
  - Cache $Z^{[l]}$.

Backward:
- Input:
  - $dA^{[l]}$.
  - Cache $Z^{[l]}$.
- Output:
  - $dA^{[l-1]}$.
  - $dW^{[l]}$.
  - $dB^{[l]}$.



$$W^{[l]} \leftarrow W^{[l]} - \alpha dW^{[l]}$$
$$B^{[l]} \leftarrow B^{[l]} - \alpha dB^{[l]}$$

Training adjustment

- Parameters:
  - $W^{[l]}, B^{[l]}$.
- Hyperparameters:
  - Learning rate α.
  - Number of iterations.
  - Number of hidden units $n^{[1]}, n^{[2]}, \ldots$.
  - Choice of activation function.

Thank you