Neural Network

Advanced machine learning lecture

Sommaire du cours

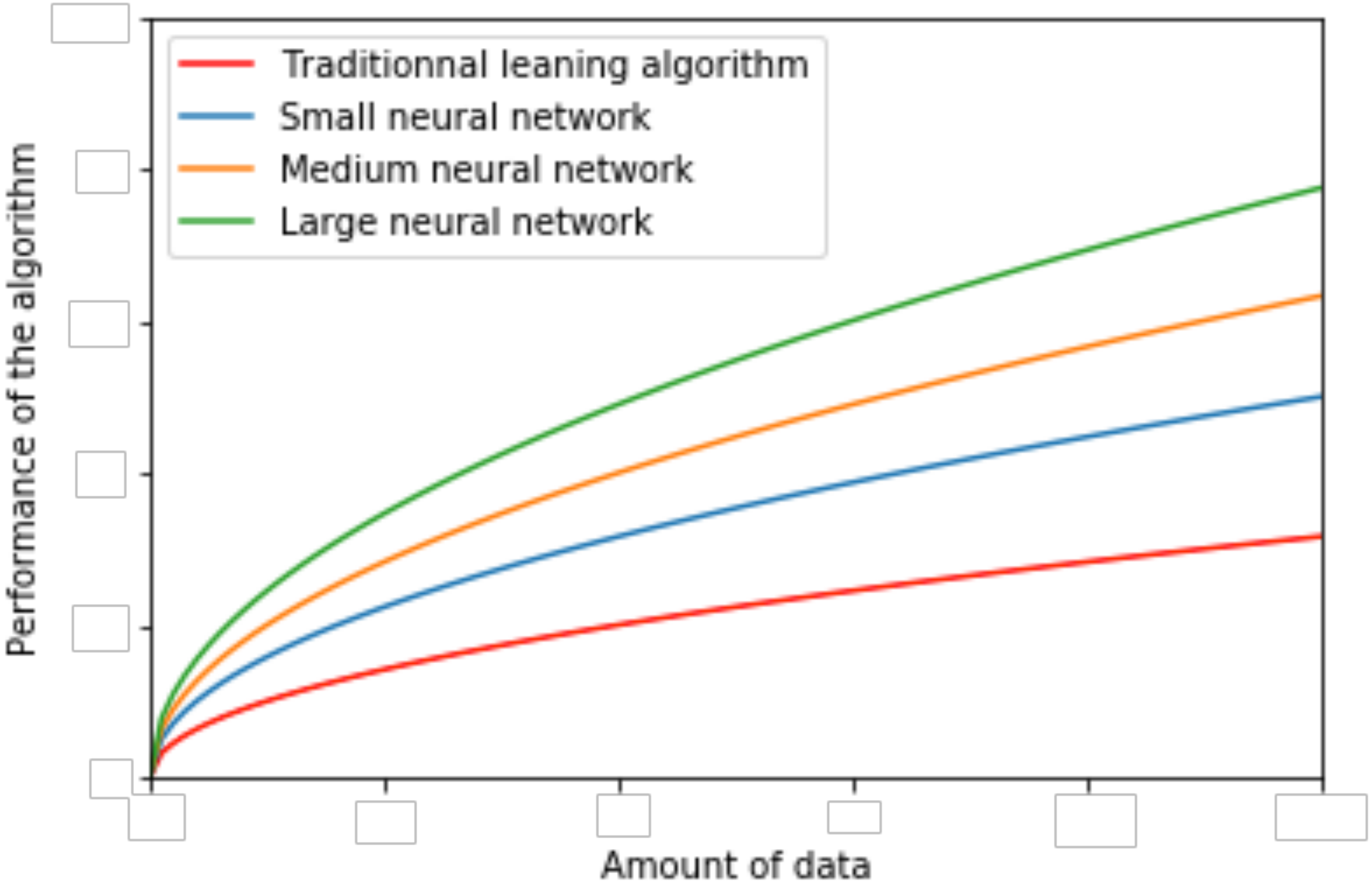
Summary

- I/ Introduction to deep learning
- II/ Neural networks Basics
- III/ Practical aspects of Deep Learning
- IV/ Optimization algorithms
- V/ Hyperparameters tuning, Batch Normalization

I/ Introduction to deep learning

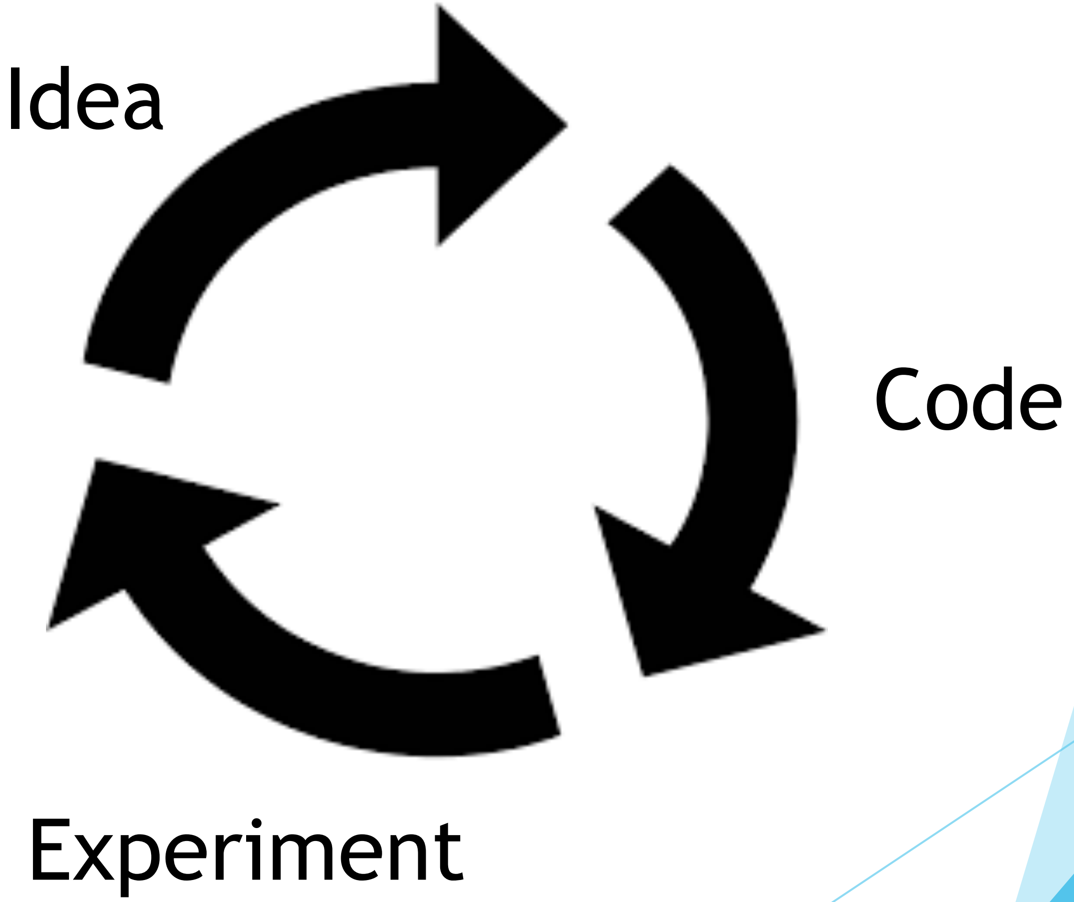
- **I/ Introduction to deep learning**
- II/ Neural networks Basics
- III/ Practical aspects of Deep Learning
- IV/ Optimization algorithms
- V/ Hyperparameters tuning, Batch Normalization

Scale drives deep learning progress

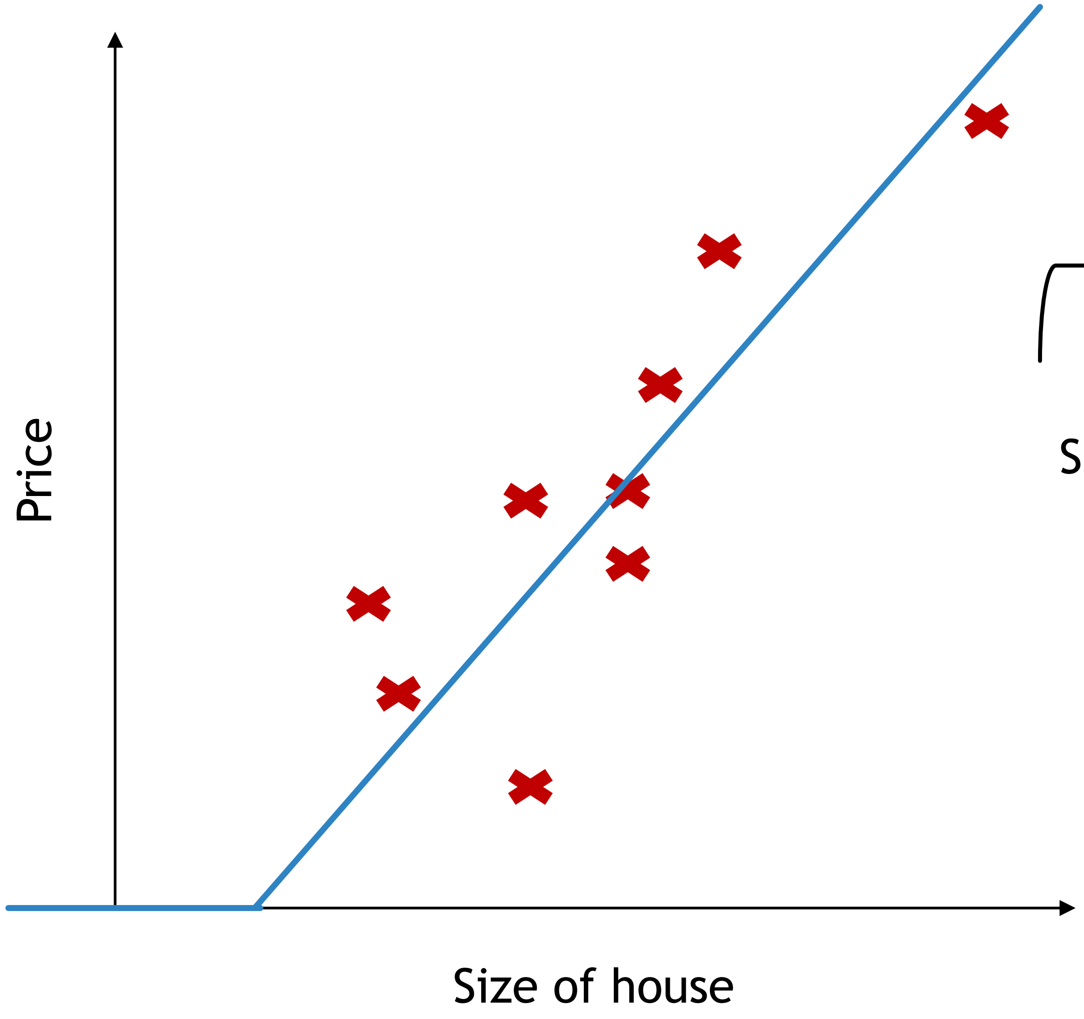


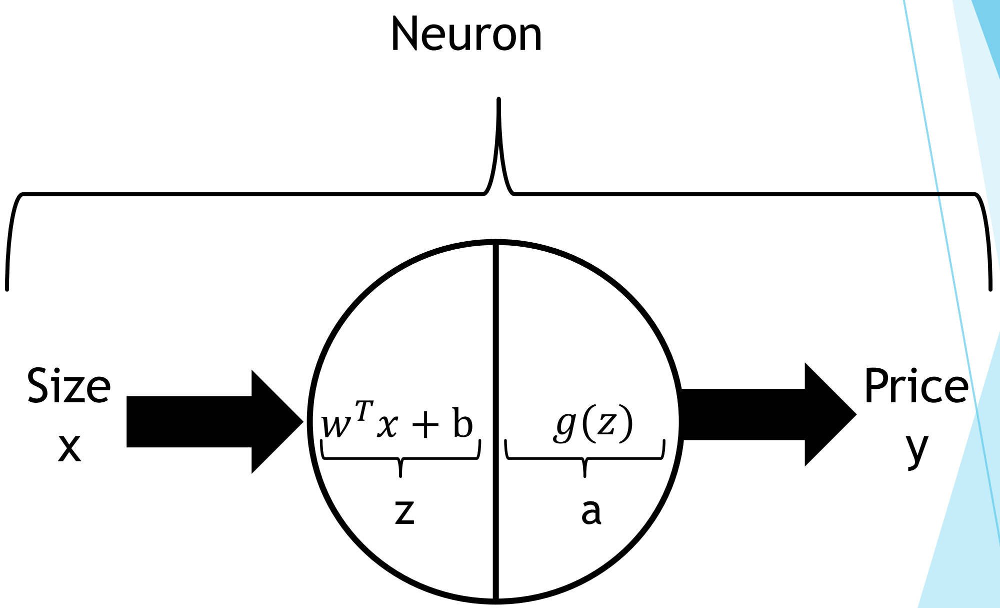
Scale drives deep learning progress

- Data
- Computation
- Algorithms



What is a Neural Network ?

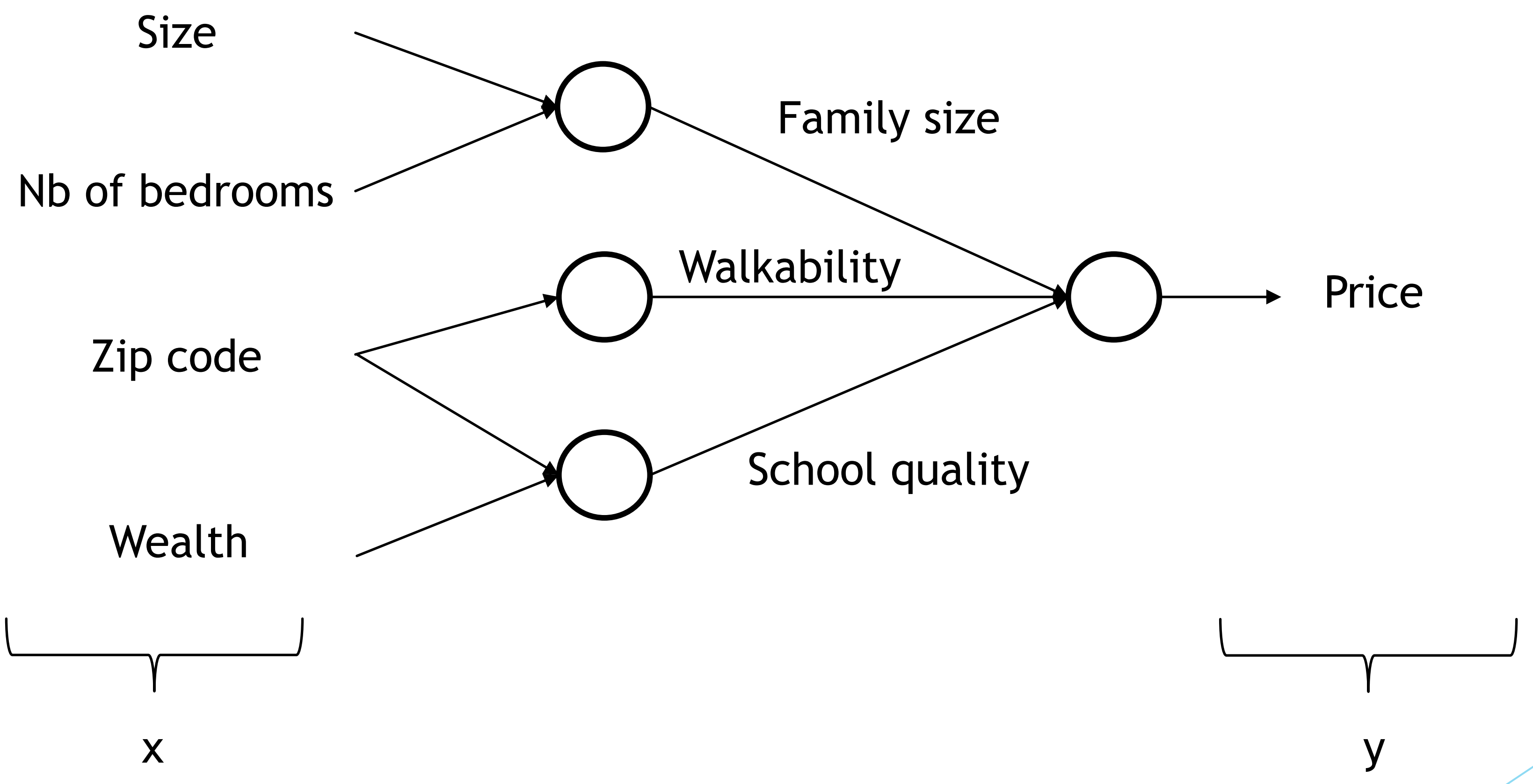




A neural network is constructed by stacking many neurons

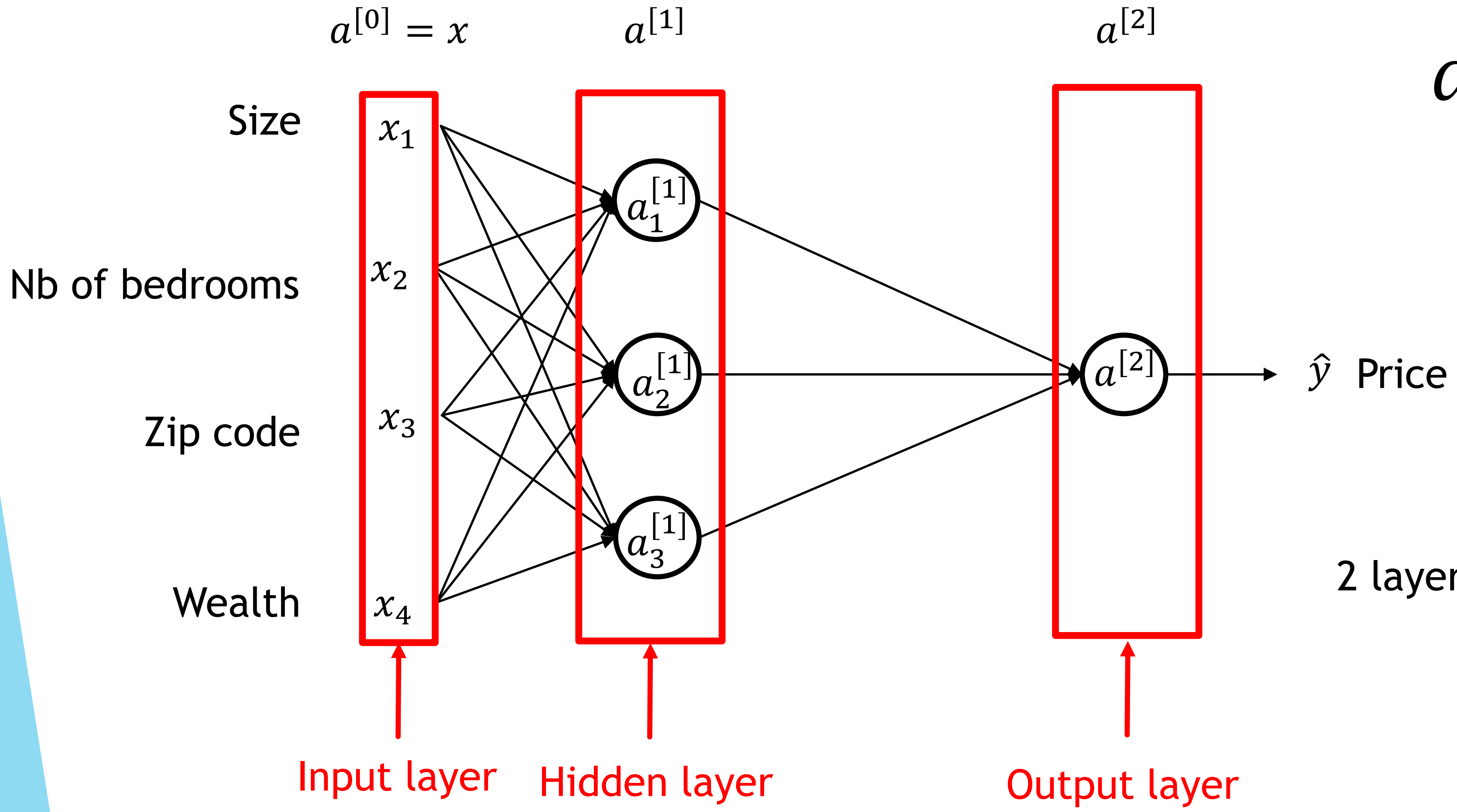
Activation function: $g(z) = \max(0, z)$

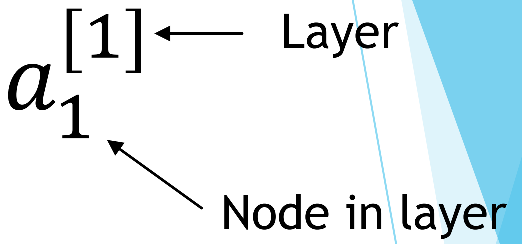
Your first neural network



II/ Neural network basics

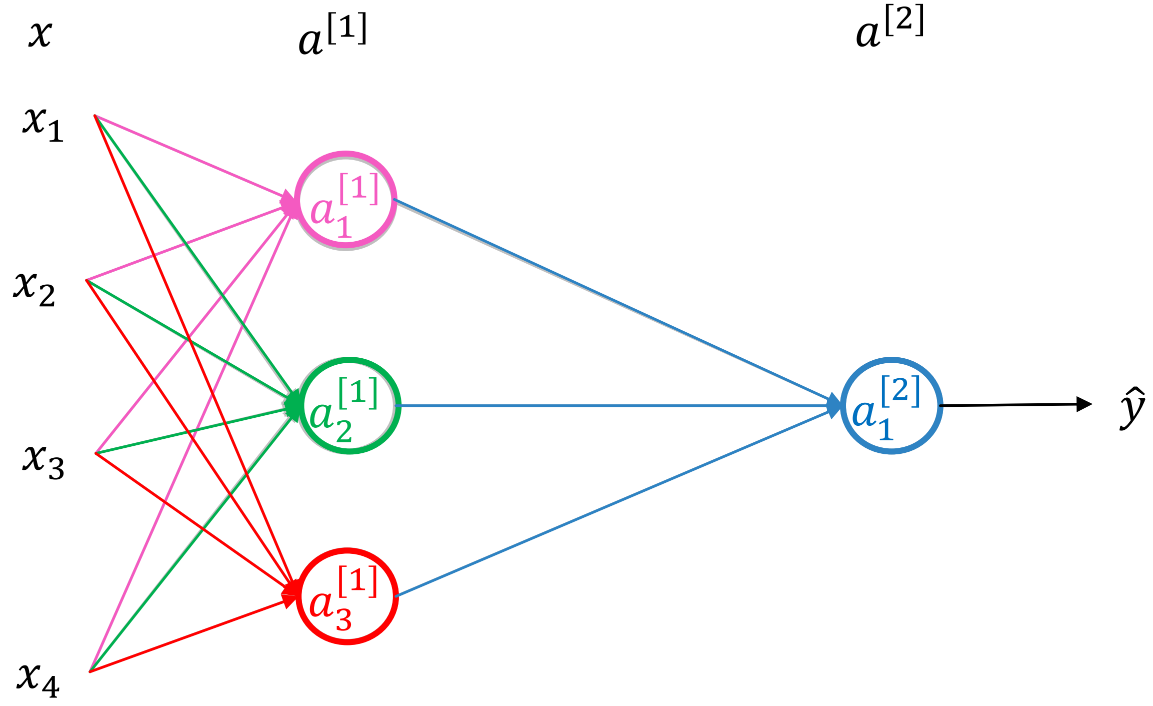
Neural network representation





2 layer Neural Network

Neural network representation



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

$$z_2^{[1]} = w_2^{[1]^T} x + b_2^{[1]} \quad ; \; a_2^{[1]} = g(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]^T} x + b_3^{[1]} \quad ; \; a_3^{[1]} = g(z_3^{[1]})$$

$$z_1^{[2]} = w_1^{[2]^T} a^{[1]} + b_1^{[2]}; \; a_1^{[2]} = g(z_1^{[2]})$$

Vectorization

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

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]} \quad ; \quad a_2^{[1]} = g(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]} \quad ; \quad a_3^{[1]} = g(z_3^{[1]})$$

$$\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{bmatrix} = \begin{bmatrix} w_{1,1}^{[1]} & w_{1,2}^{[1]} & w_{1,3}^{[1]} & w_{1,4}^{[1]} \\ w_{2,1}^{[1]} & w_{2,2}^{[1]} & w_{2,3}^{[1]} & w_{2,4}^{[1]} \\ w_{3,1}^{[1]} & w_{3,2}^{[1]} & w_{3,3}^{[1]} & w_{3,4}^{[1]} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix} \quad ; \quad \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{bmatrix} = g\left(\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{bmatrix}\right)$$

(3,1) (3,4) (4,1) (3,1)

$$z^{[1]} = W^{[1]T} \mathrm{x} + b^{[1]} \quad ; \quad a^{[1]} = g(z^{[1]})$$

(1,1) (1,3) (3,1) (1,1)

$$z^{[2]} = w^{[2]T} a^{[1]} + b^{[2]} \quad ; \quad a^{[2]} = g(z^{[2]})$$

Vectorizing across multiple examples

~~for i = 0 to m :~~

$$\overset{(3,1)}{a^{[1](i)}} = g(\overset{(3,4)}{w^{[1]^T}}\, \overset{(4,1)}{x^{(i)}} + \overset{(3,1)}{b})$$

$$\overset{(1,1)}{\hat{y}^{(i)}} = a^{[2](i)} = g(\overset{(1,3)}{w^{[2]^T}}\, \overset{(3,1)}{a^{[1](i)}} + \overset{(1,1)}{b})$$

$$\overset{(3,m)}{a^{[1]}} = g(\overset{(3,4)}{w^{[1]^T}}\, \overset{(4,m)}{X} + \overset{(3,1)}{b})$$

$$\overset{(1,m)}{\hat{y}} = a^{[2]} = g(\overset{(1,3)}{w^{[2]^T}}\, \overset{(3,m)}{a^{[1]}} + \overset{(1,1)}{b})$$

$x \in \mathbb{R}^{n_x}$ $\qquad x^{(i)} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$

$\hat{y} \in \mathbb{R}$

$X \in \mathbb{R}^{(n_x\,,\,m)}$ $\qquad X = \begin{bmatrix} x_1{}^{(1)} & x_1{}^{(2)} & x_1{}^{(3)} & \cdots & x_1{}^{(m)} \\ x_2{}^{(1)} & x_2{}^{(2)} & x_2{}^{(3)} & \cdots & x_2{}^{(m)} \\ x_3{}^{(1)} & x_3{}^{(2)} & x_3{}^{(3)} & \cdots & x_3{}^{(m)} \\ x_4{}^{(1)} & x_4{}^{(2)} & x_4{}^{(3)} & \cdots & x_4{}^{(m)} \end{bmatrix}$

$\hat{y} \in \mathbb{R}^m$

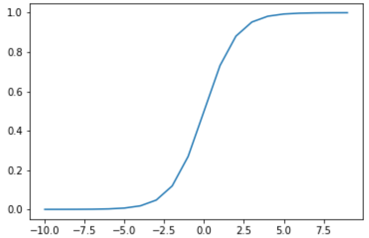
Activation function

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

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

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

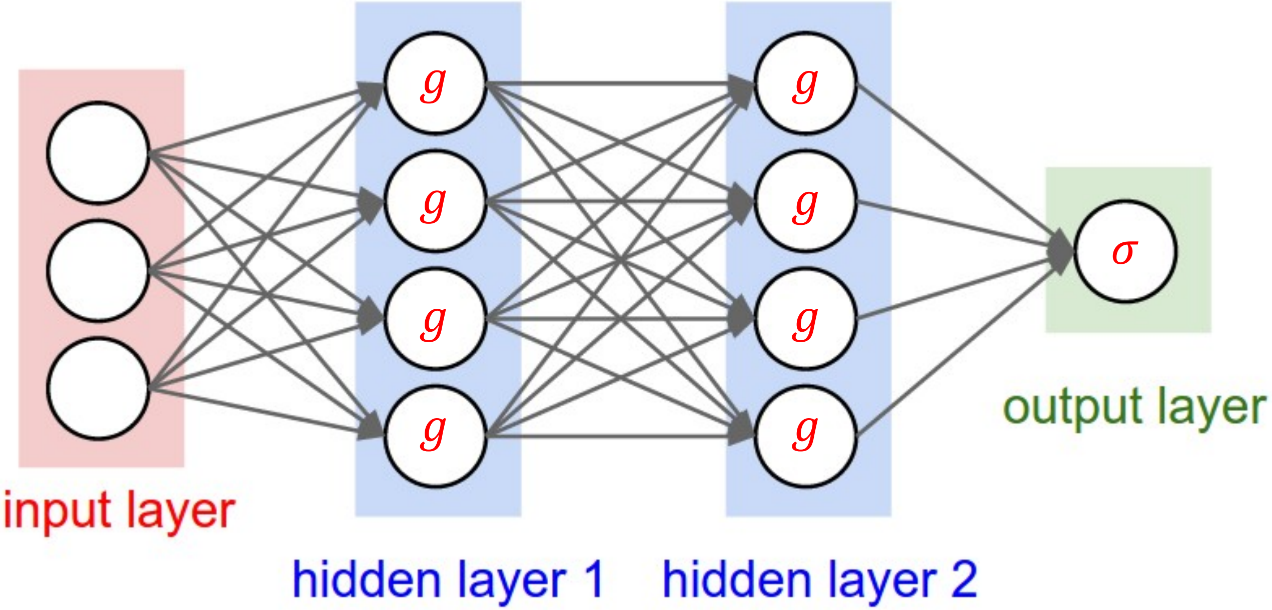
$a^{[2]} = \cancel{\sigma(z^{[2]})} \quad g(z^{[2]})$



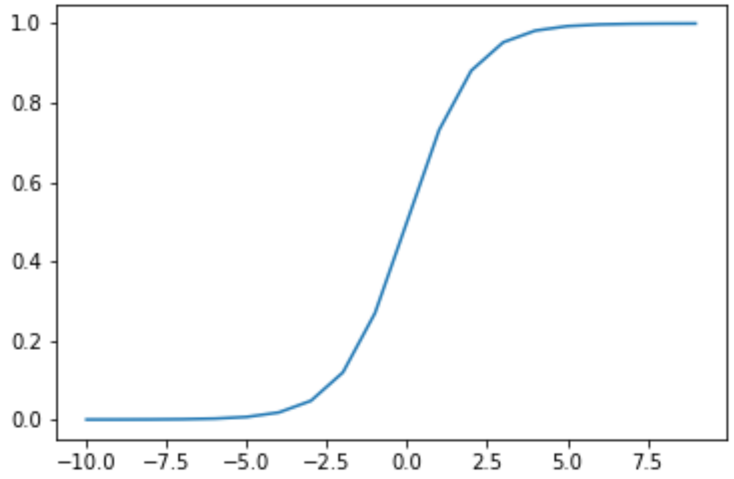
Sigmoïd saturate for large values

Do not use Sigmoïd in activation function in neural network

Expect for the output layer in case of binary classification



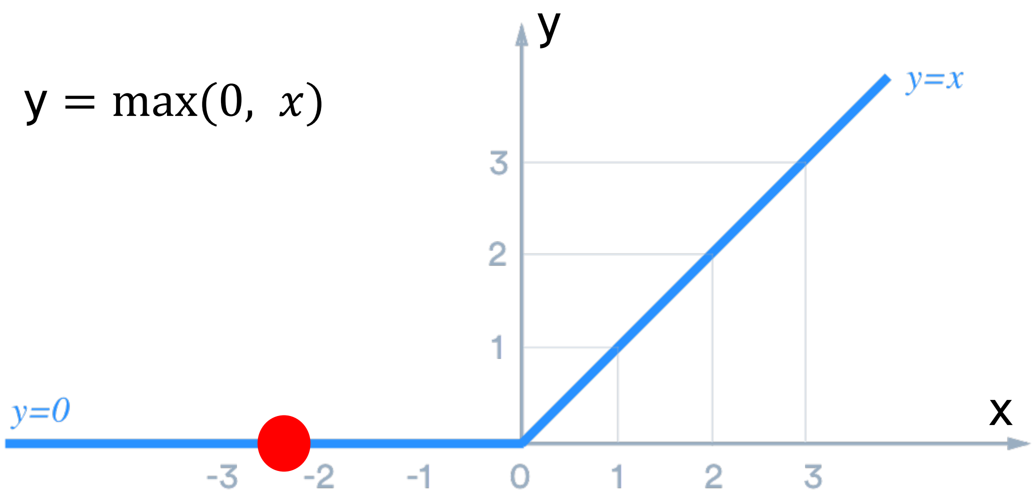
Sigmoid activation



- Sigmoid can saturate and lead to vanishing gradients
- Not zero centered
- e^X is computationally expensive

Rectified Linear Unit (ReLU)

$y = \max(0,\ x)$



Pros:

- ReLU does not saturate for positive values
- ReLU is quite fast to compute

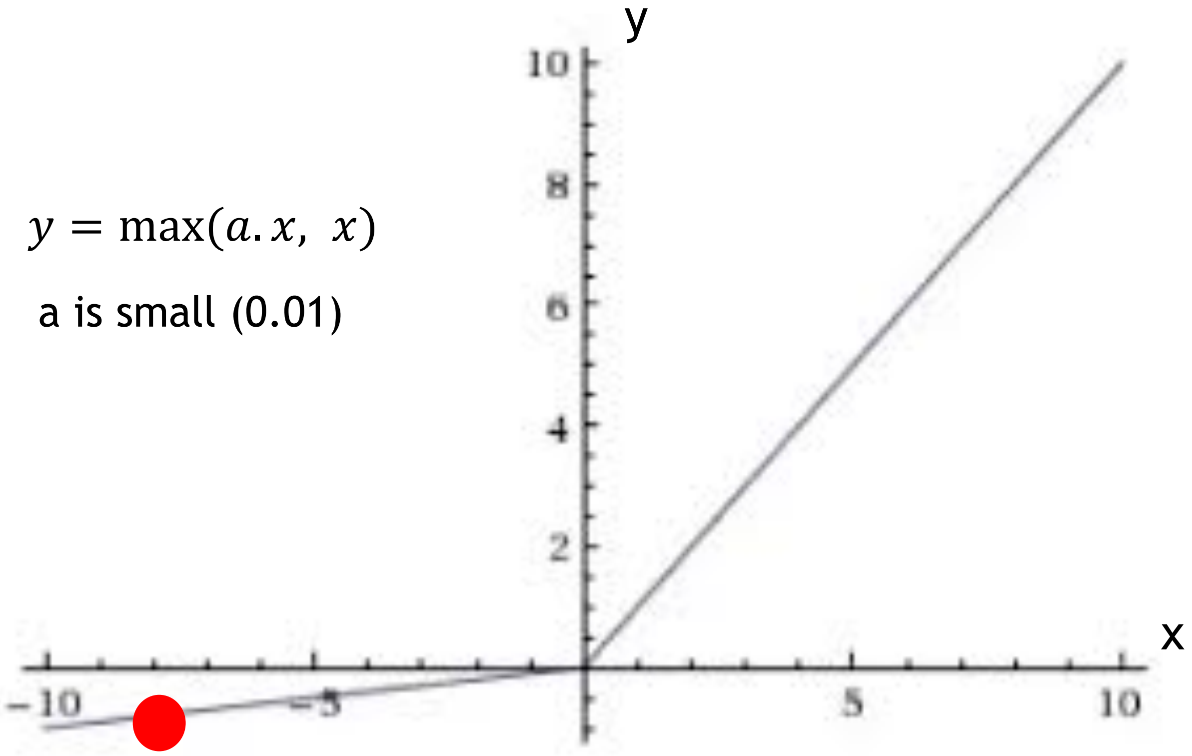
Cons:

- ReLU suffers from a problem kown as '*dying ReLU*'

Leaky ReLU

$$y = \max(a.x,\ x)$$

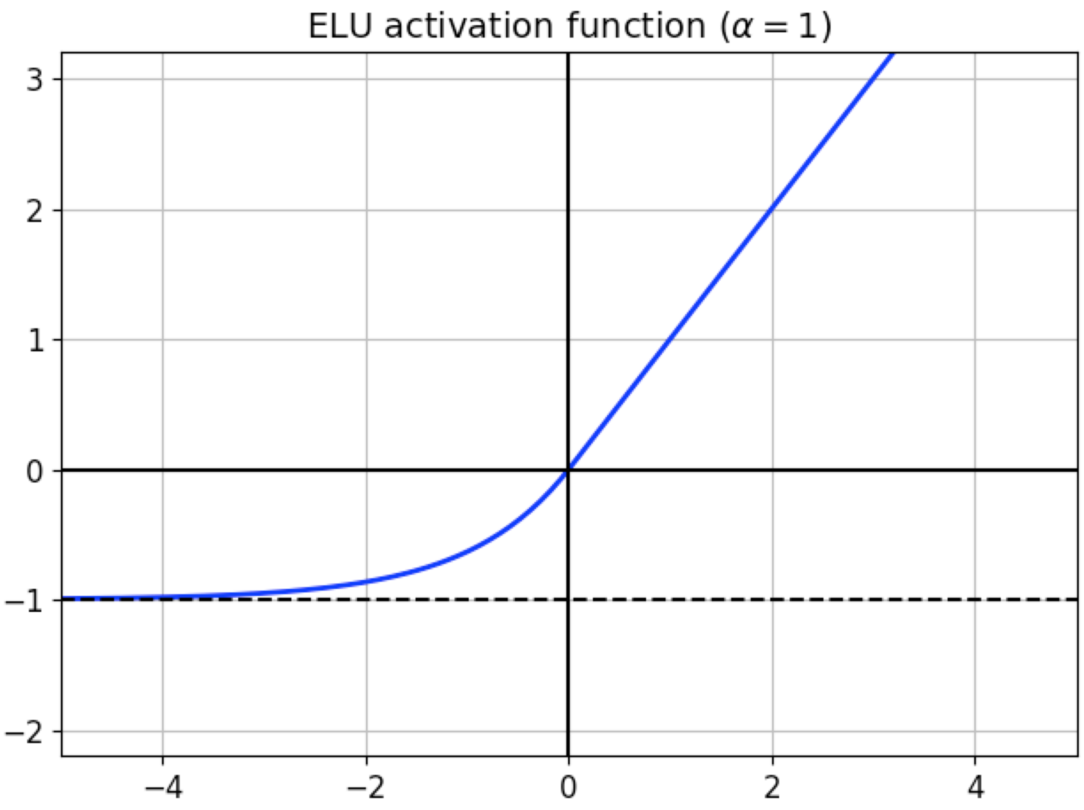
a is small (0.01)



Variants:

- Randomized leaky Relu (RReLU), if you neural network is overfitting.
- Parametric leaky Relu (PReLU), if you have a huge training set.

Exponential Linear Unit (ELU)



$$ELU_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & if\ z < 0 \\ z & if\ z \geq 0 \end{cases}$$

Pros:

- It takes on negative values which allows the unit to have an average output closer to 0

- It has non zero gradient for z < 0, which avoiding dying units issue.

- It is smooth everywhere, including around z=0, which help speed up gradient descent.

Cons:

- It is slower to compute than ReLU

Activations functions tips

- In general for performance:

  ELU > Leaky ReLU > ReLU

- If you care about runtime:

  Leaky ReLU > ELU

- Default:

  ReLU

Gradient Descent for neural networks

Parameters: $\overset{(n^{[1]}, n^{[0]})}{w^{[1]}}$, $\overset{(n^{[1]}, 1)}{b^{[1]}}$, $\overset{(n^{[2]}, n^{[1]})}{w^{[2]}}$, $\overset{(n^{[2]}, 1)}{b^{[2]}}$ $\quad n_x = n^{[0]}, \quad n^{[1]}, \quad n^{[2]} = 1$

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

Gradient descent:

Repeat {

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

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

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

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

$w^{[2]} := w^{[2]} - \alpha\, dw^{[2]}$

$b^{[2]} := b^{[2]} - \alpha\, db^{[2]}$}

Forward propagation

$$\begin{tikzcd}
 & & w^{[2]}, \; b^{[2]} \arrow[d] & & \\
\mathrm{x}, \; w^{[1]}, \; b^{[1]} \arrow[r] & \boxed{z^{[1]} = {w^{[1]}}^{T} X + b^{[1]}} \arrow[r] & \boxed{a^{[1]} = g(z^{[1]})} \arrow[r] & \boxed{z^{[2]} = {w^{[2]}}^{T} a^{[1]} + b^{[2]}} \arrow[r] & \boxed{a^{[2]} = g(z^{[2]})} \arrow[r] & \boxed{\mathcal{L}(a^{[2]}, y)}
\end{tikzcd}$$

Back propagation

x

$w^{[1]}$, $b^{[1]}$ → $z^{[1]} = w^{[1]^T} X + b^{[1]}$ → $a^{[1]} = g(z^{[1]})$ → $z^{[2]} = w^{[2]^T} a^{[1]} + b^{[2]}$ → $a^{[2]} = g(z^{[2]})$ → $\mathcal{L}(a^{[2]}, y)$

$w^{[2]}$, $b^{[2]}$

$\mathrm{d}z^{[1]} = W^{[2]} dz^{[2]} * g^{[1]'}(z^{[1]})$

$\mathrm{d}w^{[1]} = dz^{[1]} . x^T$

$a_0{}^T$

$\mathrm{d}b^{[1]} = \mathrm{d}z^{[1]}$

$\mathrm{d}a^{[1]} = \cdots$

$\mathrm{d}z^{[2]} = da \,.\, g'(z) = a - y$

$\mathrm{d}w^{[2]} = dz^{[2]} a^{[1]^T}$

$\mathrm{d}b^{[2]} = dz^{[2]}$

$$\mathrm{da} = \mathcal{L}(a^{[2]}, y) = -y\, log(a) - (1-y)\log(1-a) = -\frac{y}{a} + \frac{1-y}{1-a}$$

Formulas for computing derivatives

Forward propagation:

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

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

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

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

Back propagation:

$$dz^{[2]} = a^{[2]} - Y \qquad Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}]$$

$$dw^{[2]} = \frac{1}{m} dz^{[2]} a^{[1]^T}$$

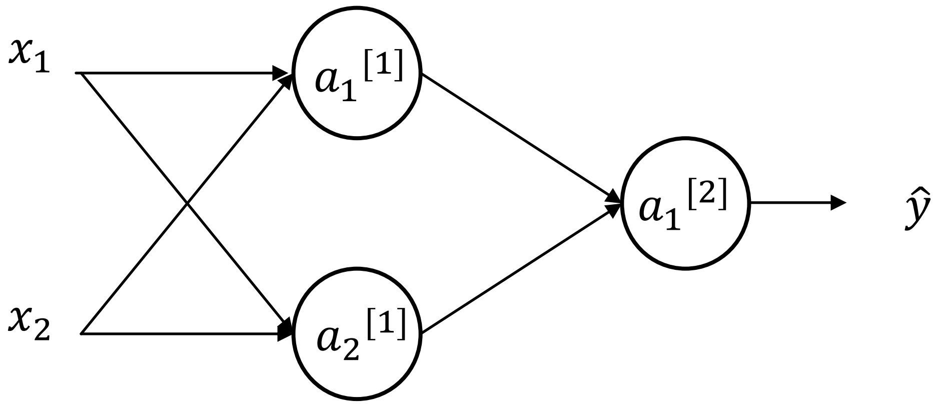
$$db^{[2]} = \frac{1}{m} sum(dz^{[2]})$$

$$dz^{[1]} = w^{[2]^T} dz^{[2]} * g^{[1]'}(z^{[1]})$$

$$dw^{[1]} = \frac{1}{m} dz^{[1]} X^T$$

$$db^{[1]} = \frac{1}{m} sum(dz^{[1]})$$

Initialize weights to zero ?

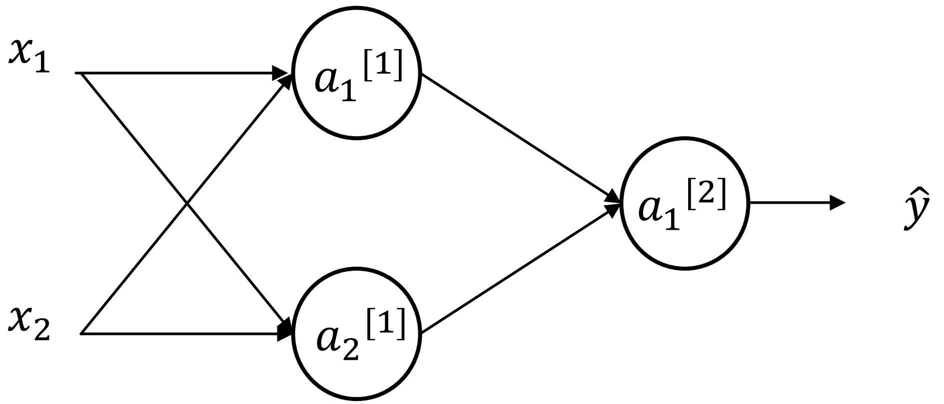


$$w^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

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

$$a_1^{[1]} = a_2^{[1]} \longrightarrow dz_1^{[1]} = dz_2^{[1]} \longrightarrow dw^{[1]} = \begin{bmatrix} u & v \\ u & v \end{bmatrix} \longrightarrow w^{[1]} := w^{[1]} - \alpha dw$$

Initialize weight randomly



$$w^{[1]} = np.random.rand\big((2{,}2)\big)\,.\overbrace{0.01}^{\text{Small number}}$$

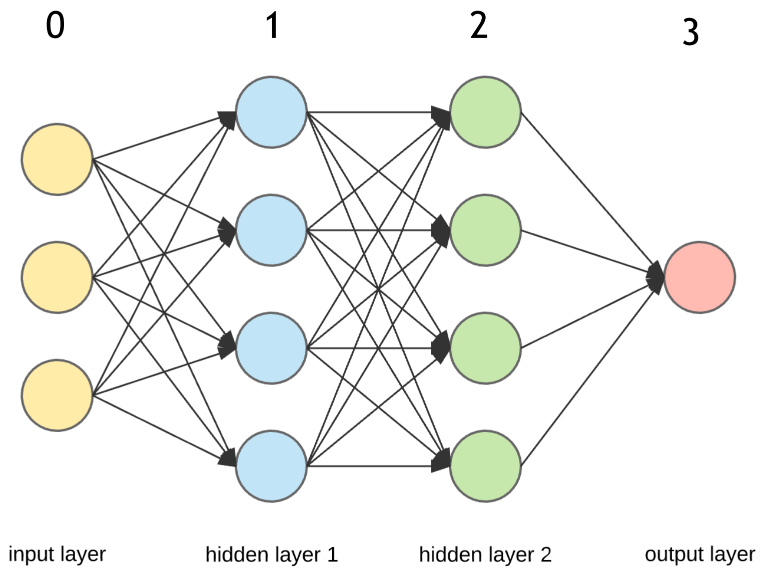
$$b^{[1]} = np.zero((2{,}1))$$

$$w^{[1]} = np.random.rand\big((1{,}2)\big)\,.\overbrace{0.01}^{\text{Small number}}$$

$$b^{[1]} = 0$$

Getting Matrix Dimensions Right

$L = 3$



$n^{[0]} = n_x = 3 \quad n^{[1]} = 4 \quad n^{[2]} = 4 \quad n^{[3]} = 1$

$$w^{[l]}: (n^{[l]}, n^{[l-1]})$$

$$b^{[l]}: (n^{[l]}, 1)$$

x

$$\underset{(n^{[1]},m)}{Z^{[1]}} = \underset{(n^{[1]},n^{[0]})}{{w^{[1]}}^T} \; \underset{(n^{[0]},m)}{A^{[0]}} + \underset{(n^{[1]},1)}{b^{[1]}}$$

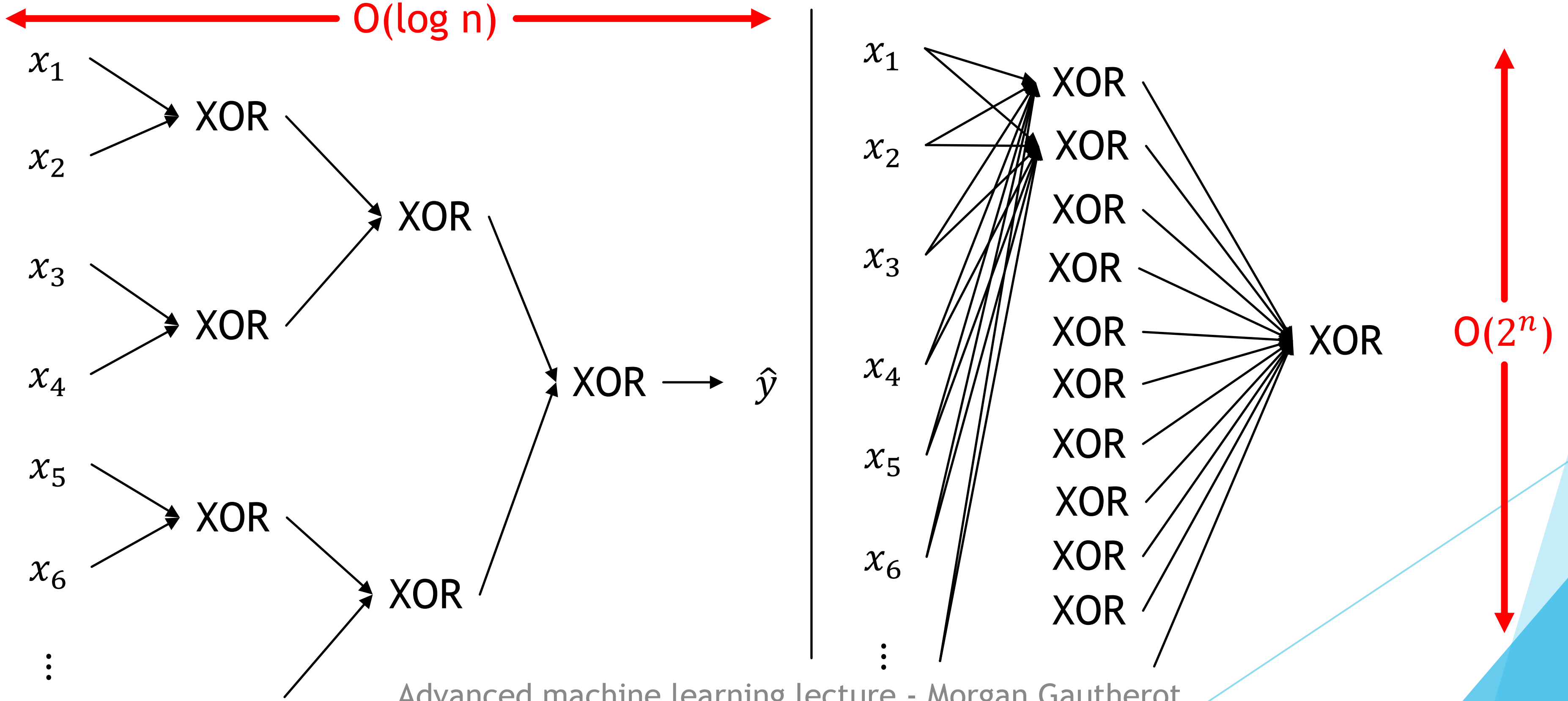
$$\underset{(n^{[2]},m)}{Z^{[2]}} = \underset{(n^{[2]},n^{[1]})}{{w^{[2]}}^T} \; \underset{(n^{[1]},m)}{A^{[1]}} + \underset{(n^{[2]},1)}{b^{[2]}}$$

$$\underset{(n^{[3]},m)}{Z^{[3]}} = \underset{(n^{[3]},n^{[2]})}{{w^{[3]}}^T} \; \underset{(n^{[2]},m)}{A^{[2]}} + \underset{(n^{[3]},1)}{b^{[3]}}$$

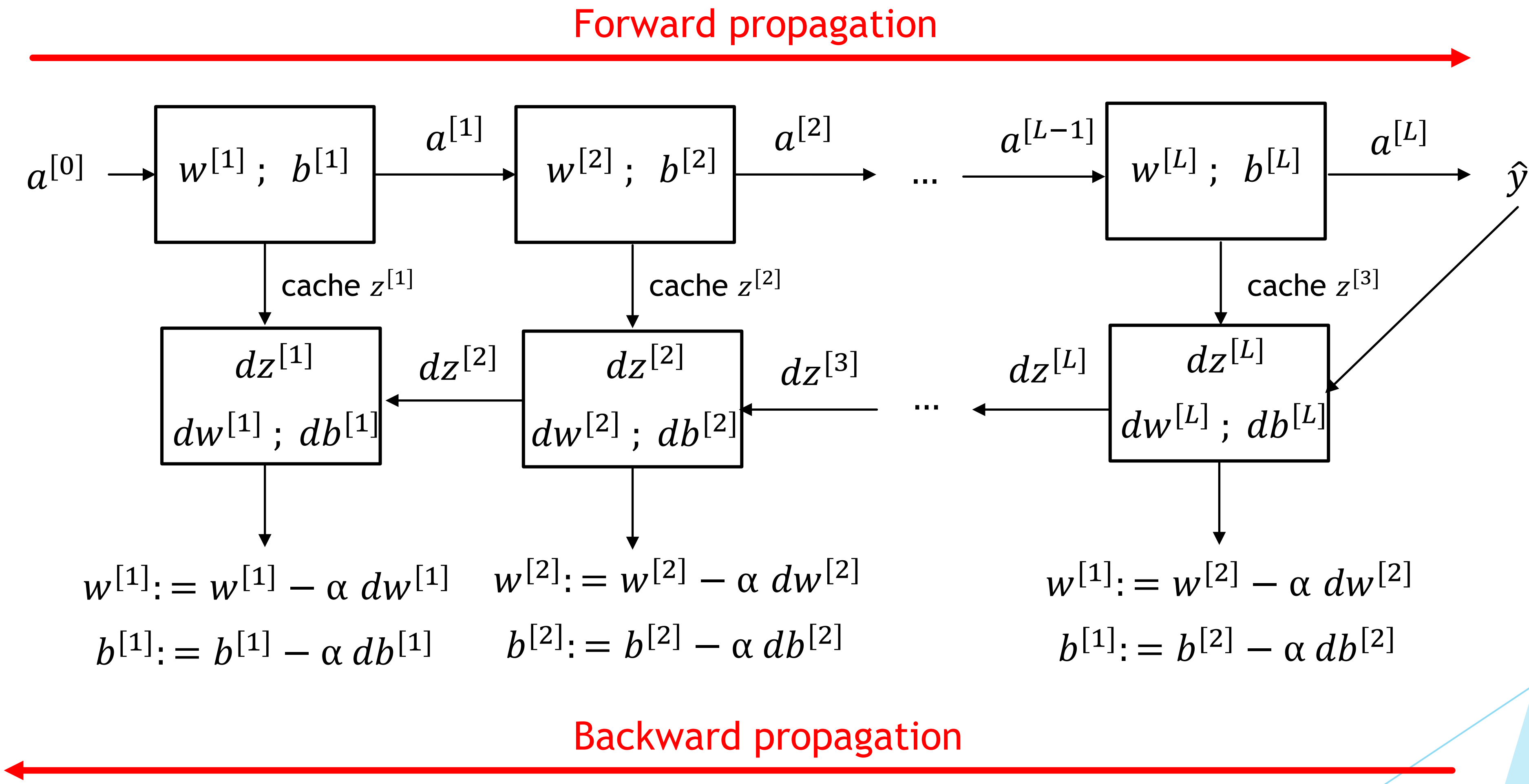
Why deep representations ?

To approximate a function Deeper neural network architecture required less parameters to train than a shallower one.

$$y = x_1 XOR\ x_2\ XOR\ x_3\ XOR\ x_4\ XOR\ x_5\ XOR\ x_6\ XOR\ \ldots$$



Summary of deep learning



III/ Practical aspects of Deep Learning

- I/ Introduction to deep learning
- II/ Neural network basics
- III/ Practical aspects of Deep Learning
- IV/ Optimization algorithms
- V/ Hyperparameters tuning, Batch Normalization

Deep learning a highly iterative process

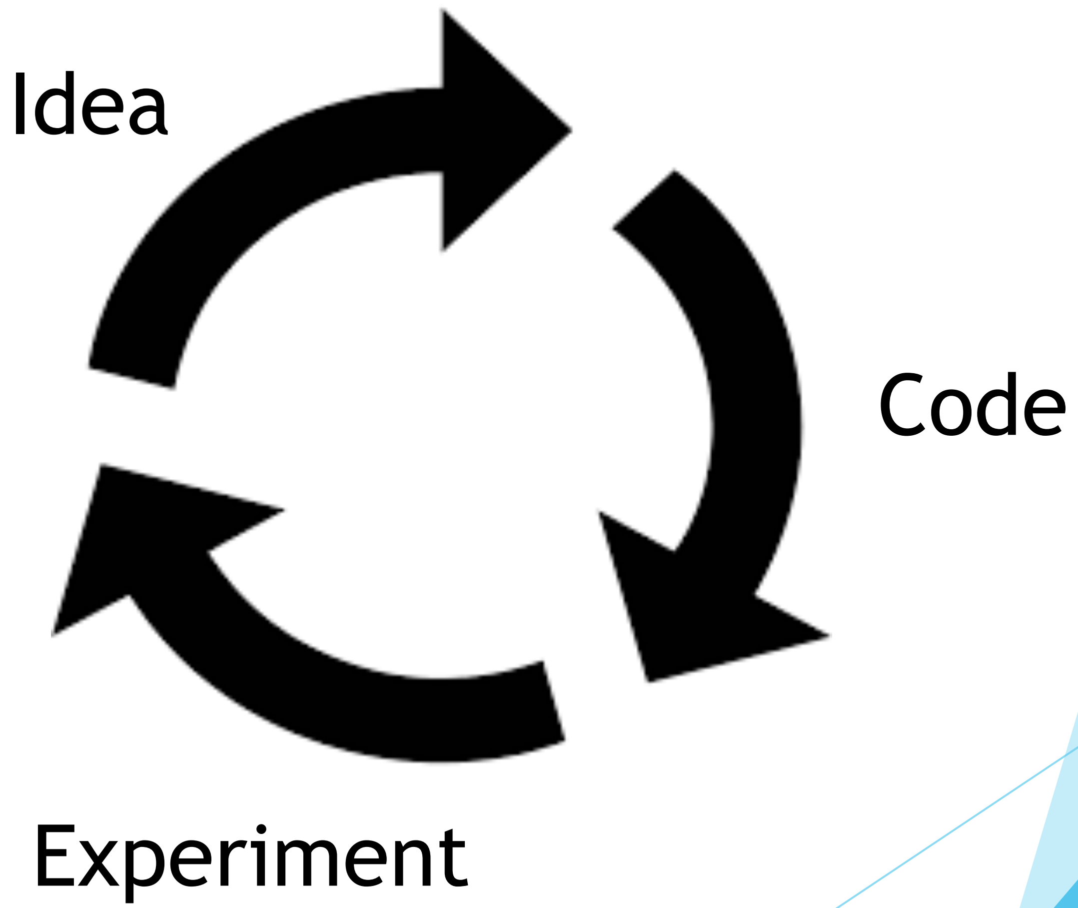
layers

hidden units

Learning rates

Activation functions

…



Train / dev / test sets

Bias & Variance

High bias ?

(training set performance)

→ Bigger network
Train longer
(NN architecture search)

↓

High variance ?

(test set performance)

→ More data
Regularization
(NN architecture search)

Regularization for logistic regression

$\min\limits_{w,b} J(w,b)$ $\quad w \in \mathbb{R}^{n_x} \quad b \in \mathbb{R}$ $\quad \lambda =$regularization parameter

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

L_2 regularization

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{j=1}^{n_x}{w_j}^2$$

We use only L_2

L_1 regularization

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{j=1}^{n_x}|w_j|$$

w will be sparse

Regularization for Neural Network

$$J\left(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\sum_{l=1}^{L} \left\|W^{[l]}\right\|^2$$

$$\left\|W^{[l]}\right\|^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (w_{ij}{}^{[l]})^2$$

$$w: (n^{[l-1]}, n^{[l]})$$

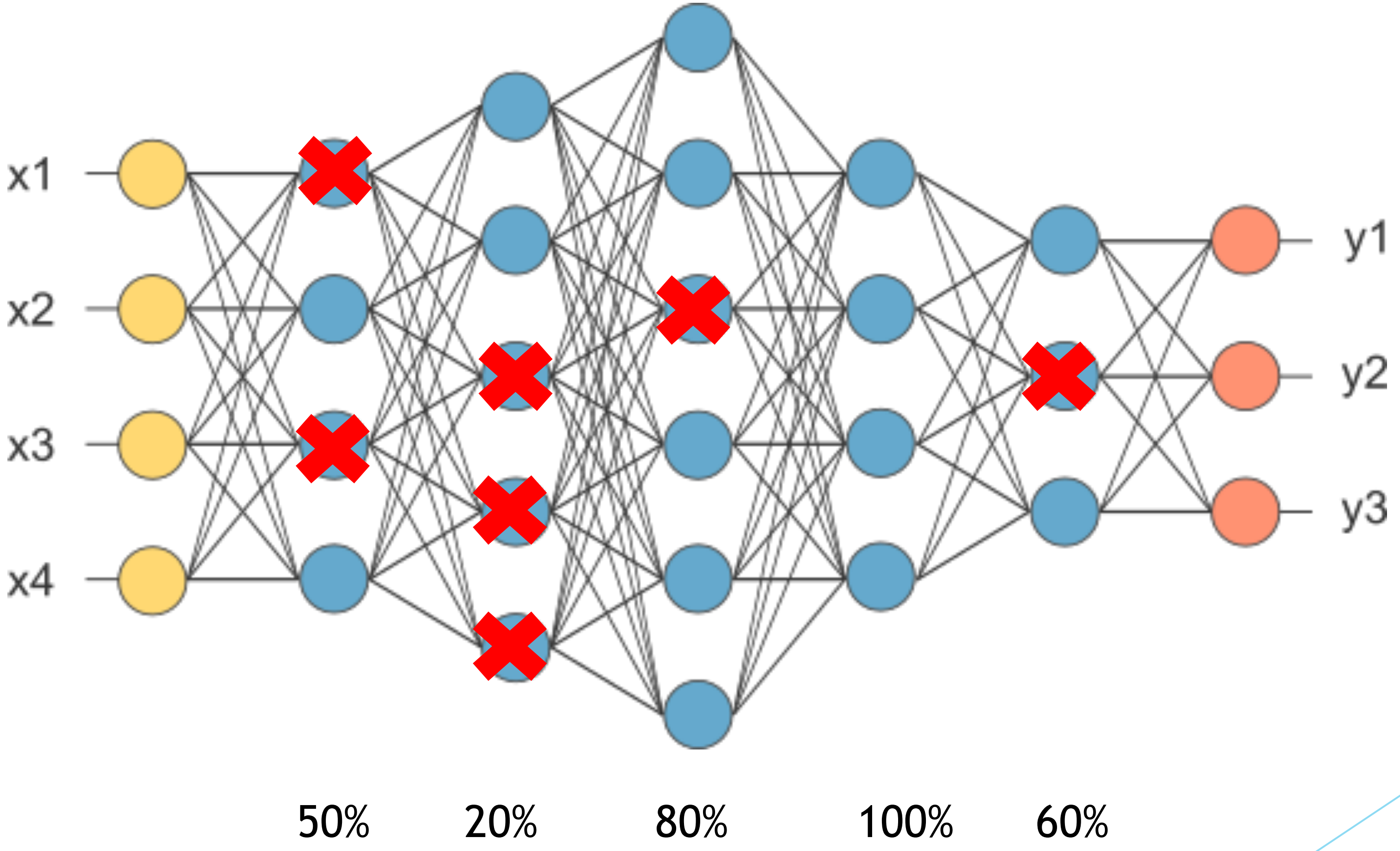
Gradient descent with regularization

$$dw^{[l]} = (from\ backprop) + \frac{\lambda}{m} w^{[l]}$$

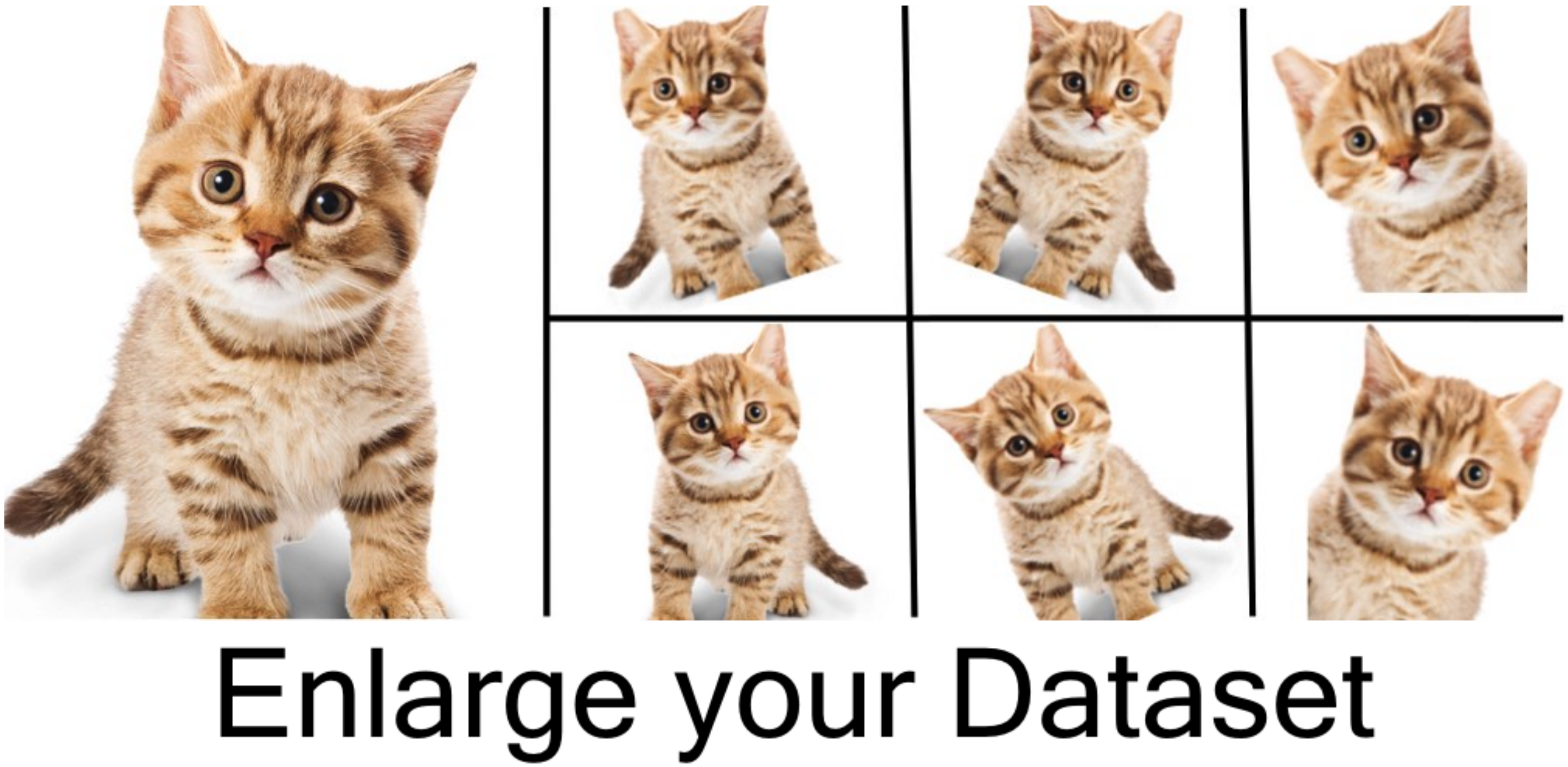
$$w^{[l]} := w^{[l]} - \alpha\, dw^{[l]}$$

Drop out regularization

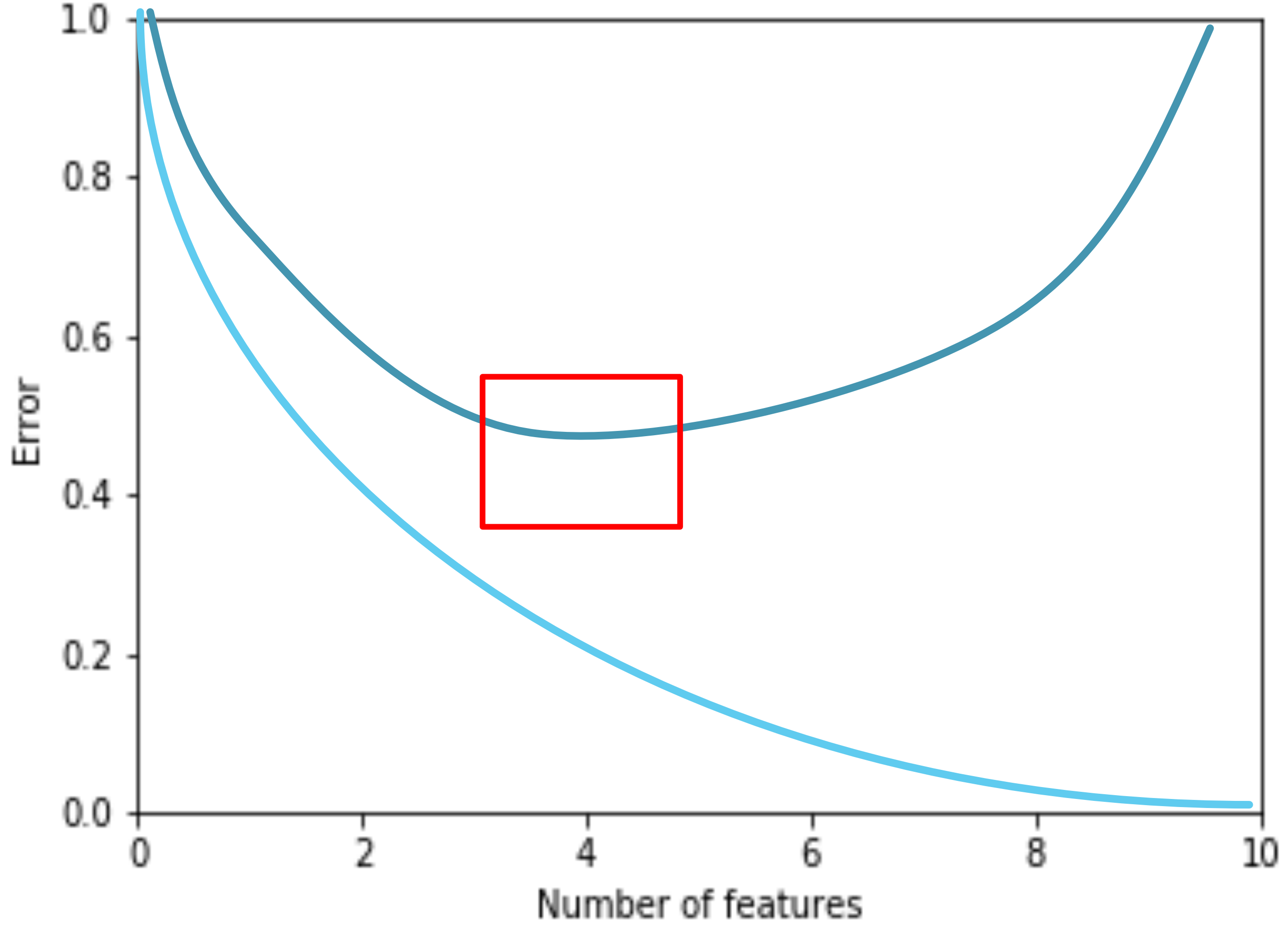
Using drop-out you can't rely on any one feature, so have to spread out weights



Data augmentation



Early stopping



Normalizing inputs

$$x' = \frac{x - \bar{x}}{\sigma}$$

Train $\bar{x}$ and σ with your train set and save them to apply them to the dev and test set.

Vanishing and exploding gradient

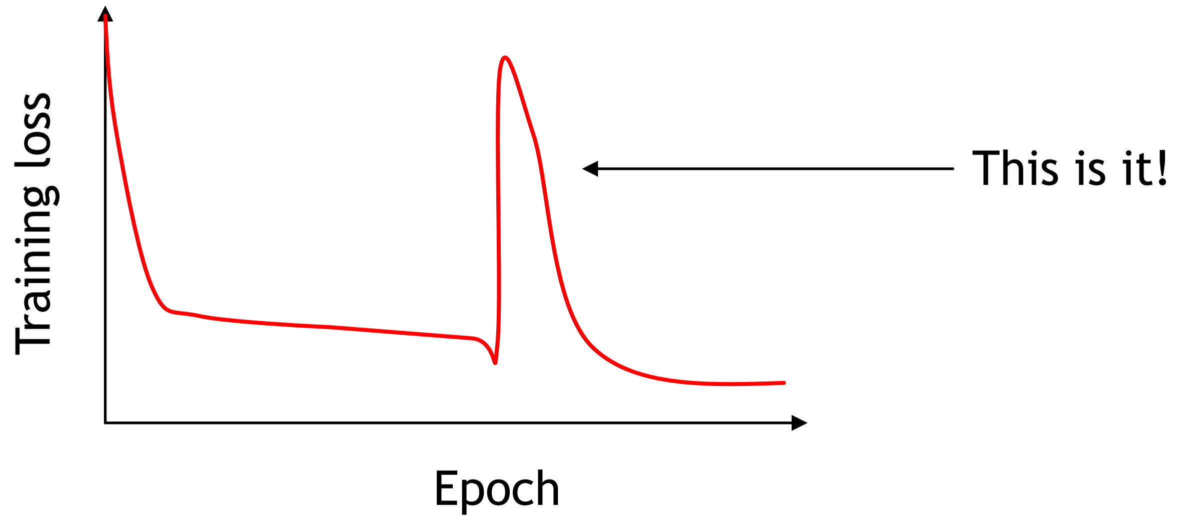
When you construct a very deep neural network your gradient in the last hidden layer can be very large or be approximate to zero.

It’s a huge problem because with this problem your model can not learning properly.

Exploding gradients: detection

Exploding gradients are easy to detect

Unstable learning curve



The gradients can be to large and contain NaNs
and you end up with NaNs in the weights

Gradient clipping

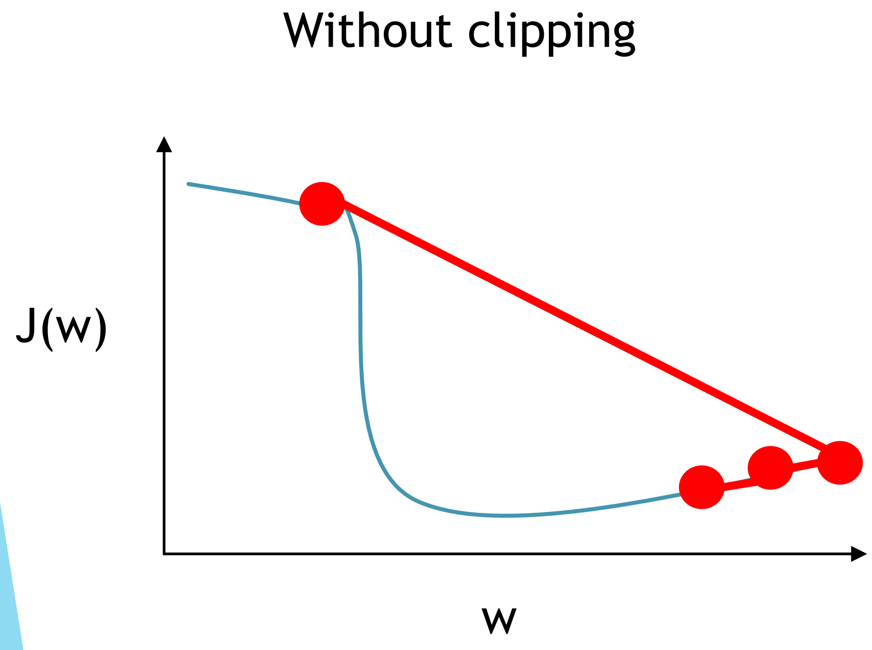
Gradient $g = \frac{\partial L}{\partial \theta}$, θ- all the network parameters

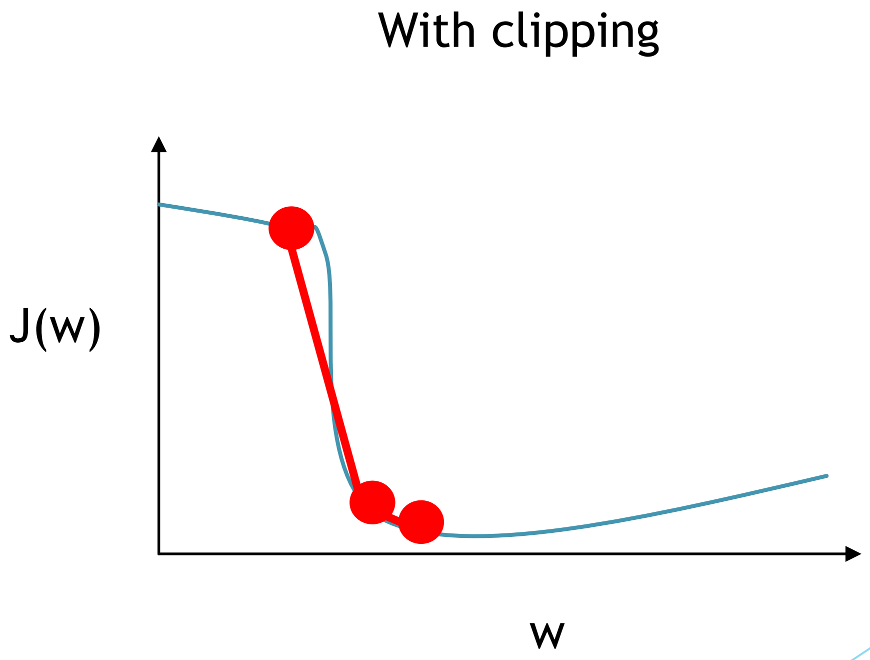
$$if\ \|g\| > threshold$$

$$g \leftarrow \frac{threshold}{\|g\|} g$$

Choose the highest threshold which helps to overcome the exploding gradient problem

Gradient clipping





Weight Initialization for relu activation

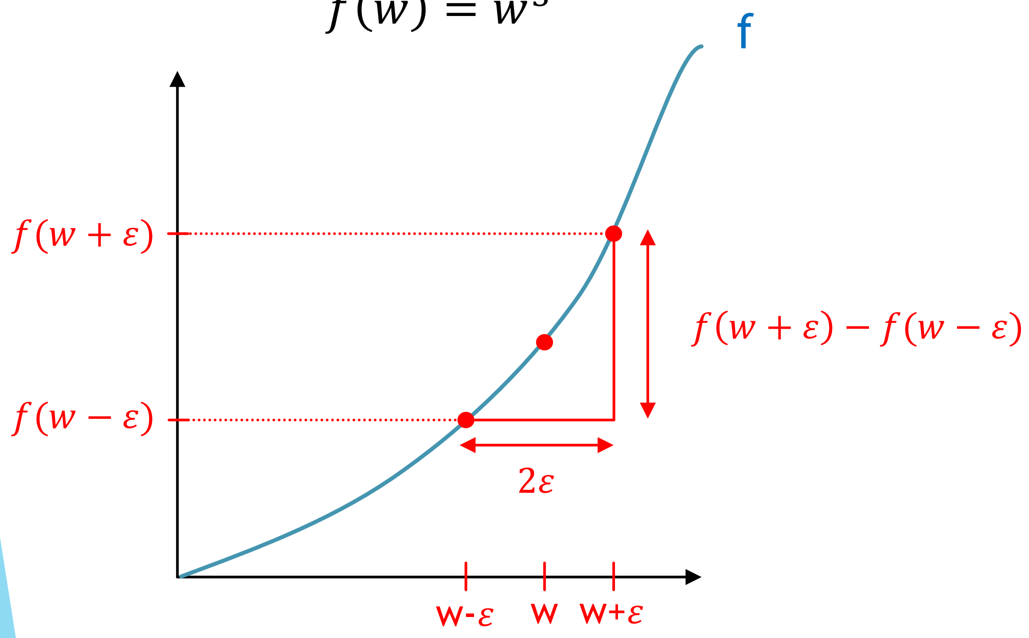
To avoid exploding gradient, for large n, we want w_i small.

$$Var(w_i) = \frac{2}{n}$$

$$w^{[L]} = np.random.rand\left(w^{[l]}.shape\right) * np.sqrt(\frac{2}{n^{[L-1]}})$$

Gradient checking

$$f(w) = w^3$$



$\varepsilon = 0.01$

$$f'(w) = \lim_{\varepsilon \to 0} \frac{f(w+\varepsilon) - f(w-\varepsilon)}{2\varepsilon}$$

$$\frac{f(w+\varepsilon) - f(w-\varepsilon)}{2\varepsilon} \approx g(w)$$

$$\frac{(1.01)^3 - (0.99)^3}{2(0.01)} = 3.0001$$

$$g(1) = 3w^2 = 3$$

approx error: 0.0001

Gradient checking

$\mathrm{J}(\theta) = J(\theta_1, \theta_2, \theta_3, \dots)$

For each i:

$$\mathrm{d}\theta approx\ [i] = \frac{J(\theta_1,\theta_2,\theta_3,\dots,\theta_i+\varepsilon,\dots) - J(\theta_1,\theta_2,\theta_3,\dots,\theta_i-\varepsilon,\dots)}{2\varepsilon}$$

$$\mathrm{d}\theta approx\ [i] \approx \frac{\partial J}{\partial \theta_i}$$

Check $\dfrac{\|\mathrm{d}\theta approx - \mathrm{d}\theta\|_2}{\|\mathrm{d}\theta approx\|_2 + \|\mathrm{d}\theta\|_2} \approx 10^{-7}$

If check $\geq 10^{-5}$ your implementation of gradient can have some problems.

Gradient checking

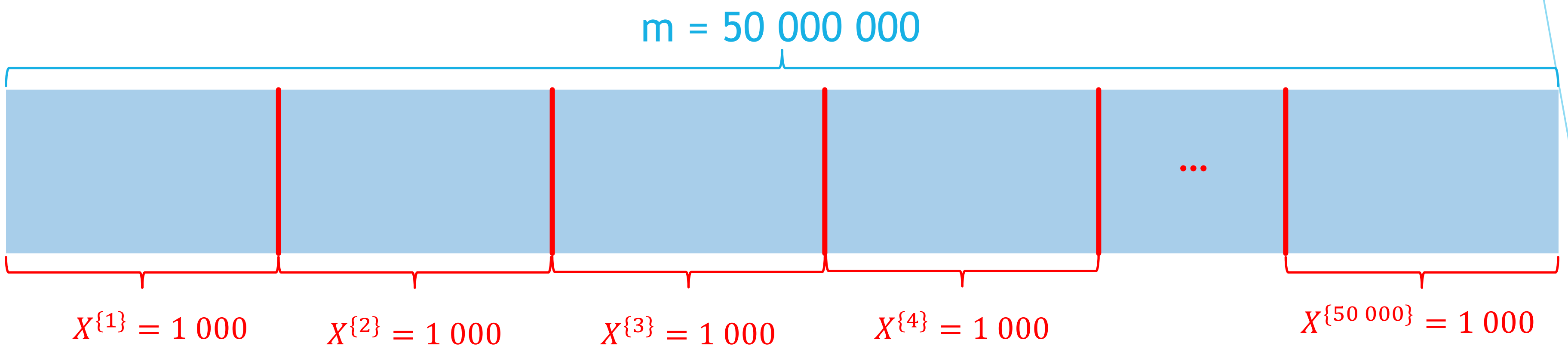
- Don’t use in training – only to debug.
- If algorithm fails grad check, look at components to try to identify bug.
- Remember regularization
- Doesn’t work with dropout
- Run at random initialization

III/ Practical aspects of Deep Learning

- I/ Introduction to deep learning
- II/ Neural network basics
- III/ Practical aspects of Deep Learning
- IV/ Optimization algorithms
- V/ Hyperparameters tuning, Batch Normalization

Batch vs mini-batch gradient descent

X =
(n_x, m)



For iter = 1, …, 1 000

  For I = 1, …, 50 000

    Forward prop on $X^{\{i\}}$

    Compute cost for $X^{\{i\}}$

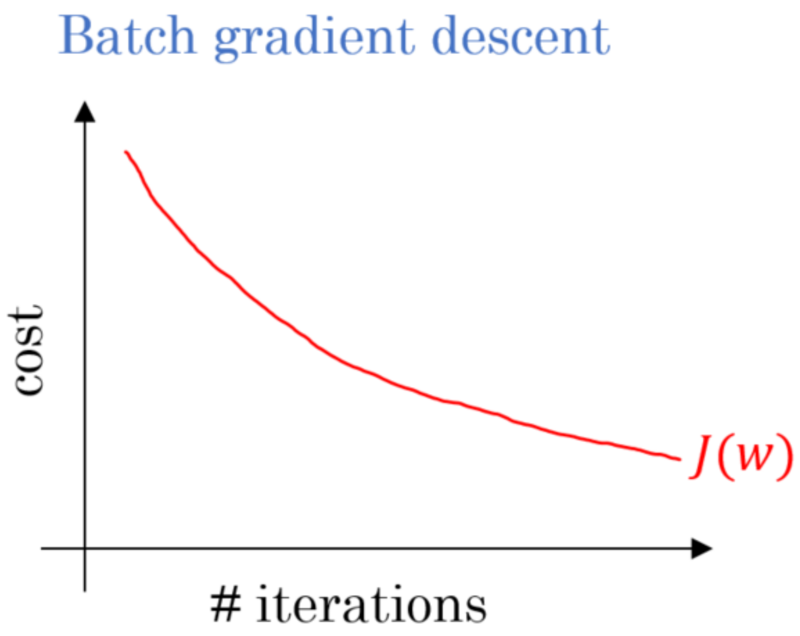
    Backward prop on $X^{\{i\}}$

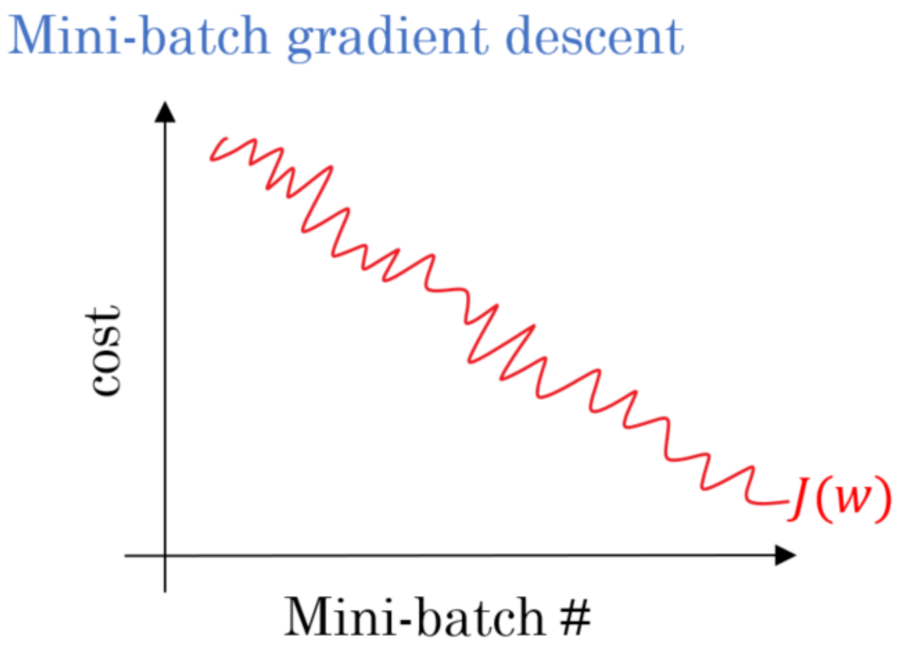
    Update W and b

1 epoch

1 iteration

Batch vs mini-batch gradient descent





Batch too large:

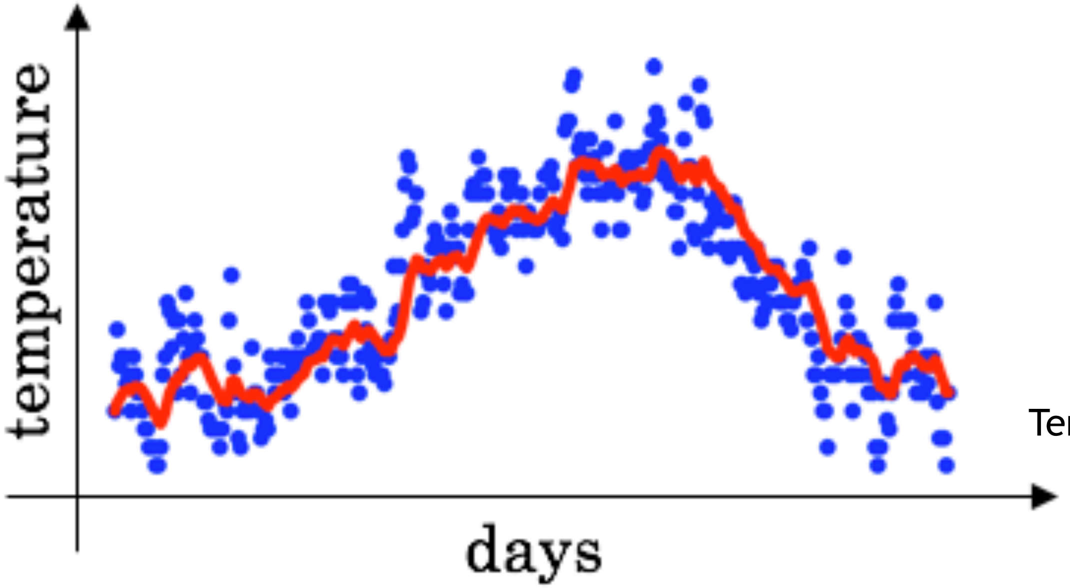
Too long per iteration

Batch too small:

Loose speeding from vectorization

Use a power of 2 and make sure that your mini batch match your GPU memory

Exponentially weighted Averages



Tradeoff between past and training

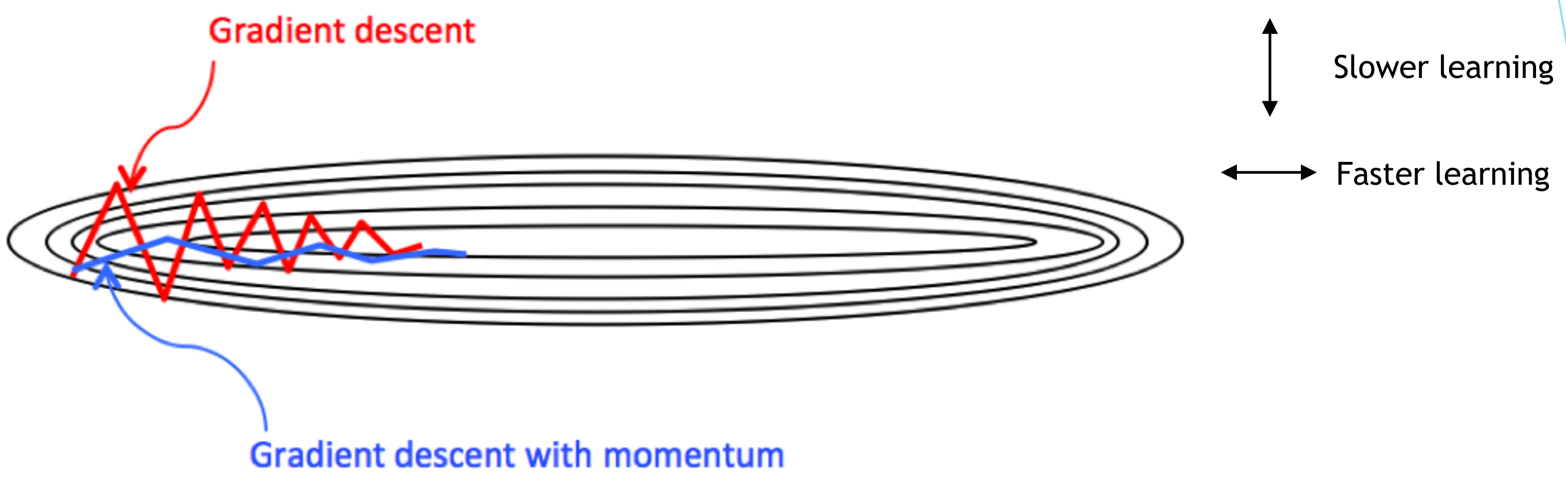
Temperature for time t-1

$$V_t = \beta\, V_{t-1} + (1 - \beta)\, \theta_t$$

Temperature for time t

Weight for time t

Gradient descent with momemtum



On iteration t:

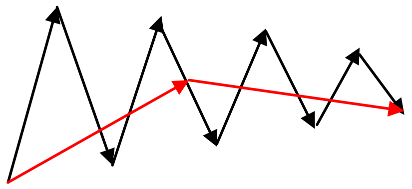
Compute dw, db on current mini-batch

$V_{dw} := \beta\, V_{dw} + (1-\beta)\, dw$

$V_{db} := \beta\, V_{db} + (1-\beta)\, db$

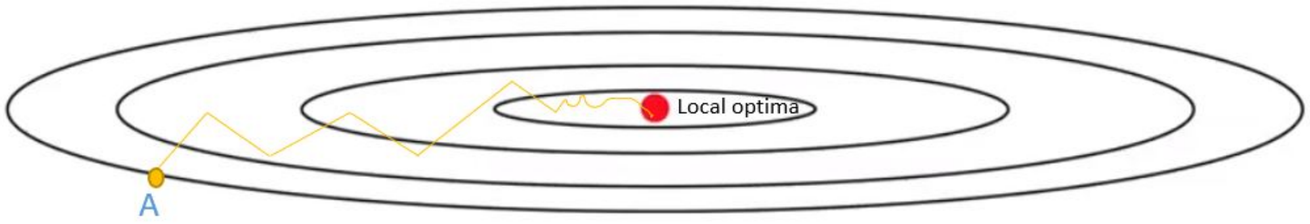
$\mathrm{W} := W - \alpha\, V_{dw}$

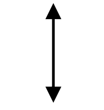
$\mathrm{b} := b - \alpha\, V_{db}$



$\beta = 0.9$ in most of the case
but you can change it
It can be an hyperparameter.

RMSProp





On iteration t:

Compute dw, db on current mini-batch

$$S_{dw} := \beta\, S_{dw} + (1-\beta)\, dw^2$$

$$S_{db} := \beta\, S_{db} + (1-\beta)\, db^2$$

$$\mathrm{W} := W - \alpha \frac{dw}{\sqrt{S_{dw}}}$$

$$\mathrm{b} := b - \alpha \frac{db}{\sqrt{S_{db}}}$$

Adam optimization

$V_{dw} = 0,\ S_{dw} = 0, V_{db} = 0, S_{db} = 0$

On iteration t:

Compute dw, db on current mini-batch

$$\left.\begin{aligned} V_{dw} &:= \beta_1\ V_{dw} + (1-\beta_1)\ dw \\ V_{db} &:= \beta_1\ V_{db} + (1-\beta_1)\ db \end{aligned}\right\} \quad \text{Momentum } \beta_1$$

$$\left.\begin{aligned} S_{dw} &:= \beta_2\ S_{dw} + (1-\beta_2)\ dw^2 \\ S_{db} &:= \beta_2\ S_{db} + (1-\beta_2)\ db^2 \end{aligned}\right\} \quad \text{RMSProp } \beta_2$$

$$\mathrm{W} := W - \alpha \frac{V_{dw}}{\sqrt{S_{dw}}}$$

$$\mathrm{b} := b - \alpha \frac{b}{\sqrt{S_{db}}}$$

You need top choose :

- α
- β_1
- β_2

Learning rate decay

alpha can change to permit your gradient descent to reach easily the minimum.

$$\alpha = \frac{1}{1 + decay\ rate\ * epoch\ num}\alpha_0$$

$$\alpha = rate^{epoch\ num}\alpha_0$$

$$\alpha \frac{rate}{\sqrt{epoch\ num}}\alpha_0$$

Epoch: one pass for all your mini-batch

III/ Practical aspects of Deep Learning

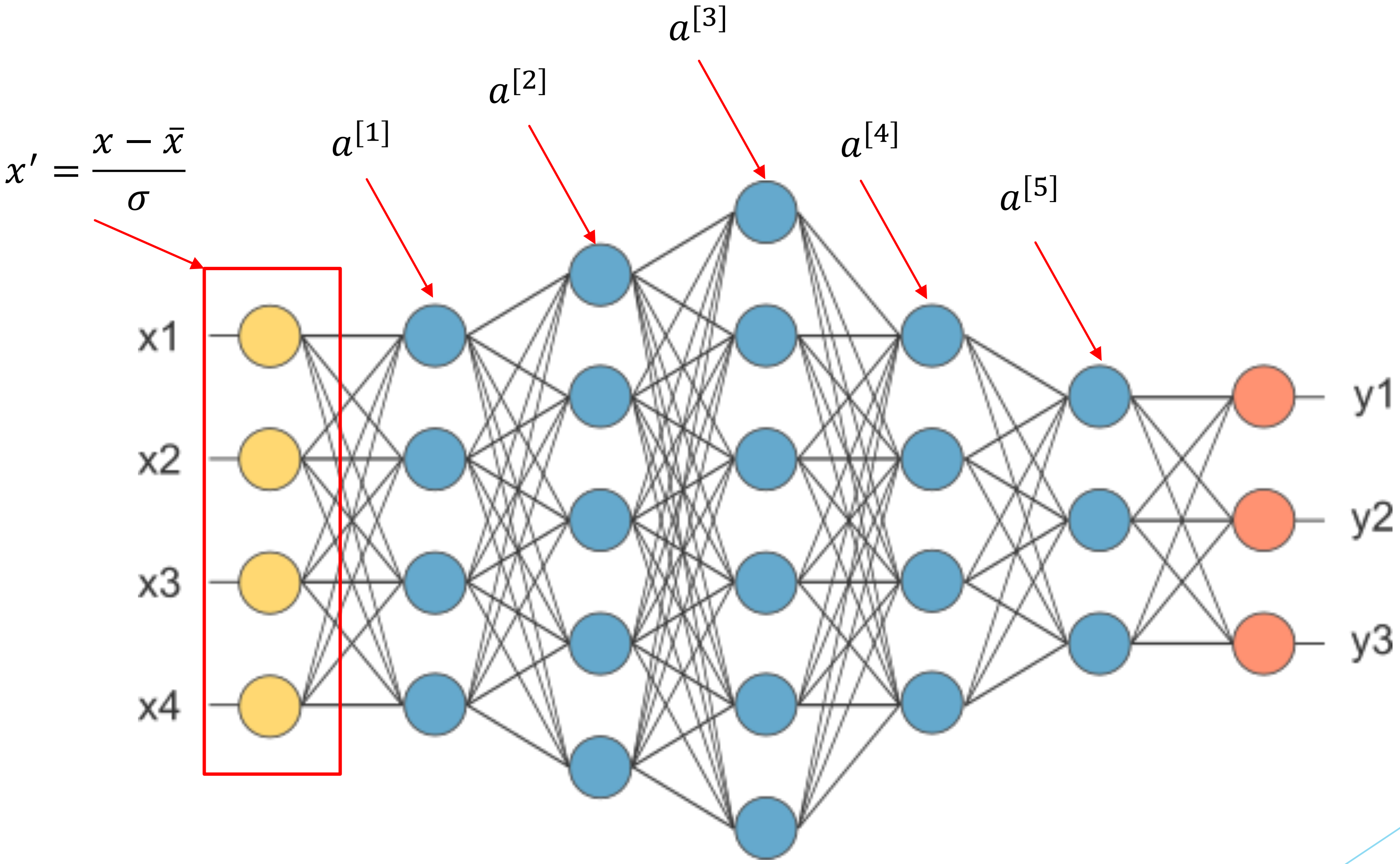
- I/ Introduction to deep learning
- II/ Neural network basics
- III/ Practical aspects of Deep Learning
- IV/ Optimization algorithms
- V/ Hyperparameters tuning, Batch Normalization

Tuning process

- $\boxed{\alpha}$
- β or β_1, β_2
- $\boxed{\text{\# layers}}$
- $\boxed{\text{\# hidden units}}$
- $\boxed{\text{Learning rate decay}}$
- $\boxed{\text{Mini-batch Size}}$

$\beta = 0.9$ or $\beta_1 = 0.9, \beta_2 = 0.999$

Batch normalization



Batch normalization

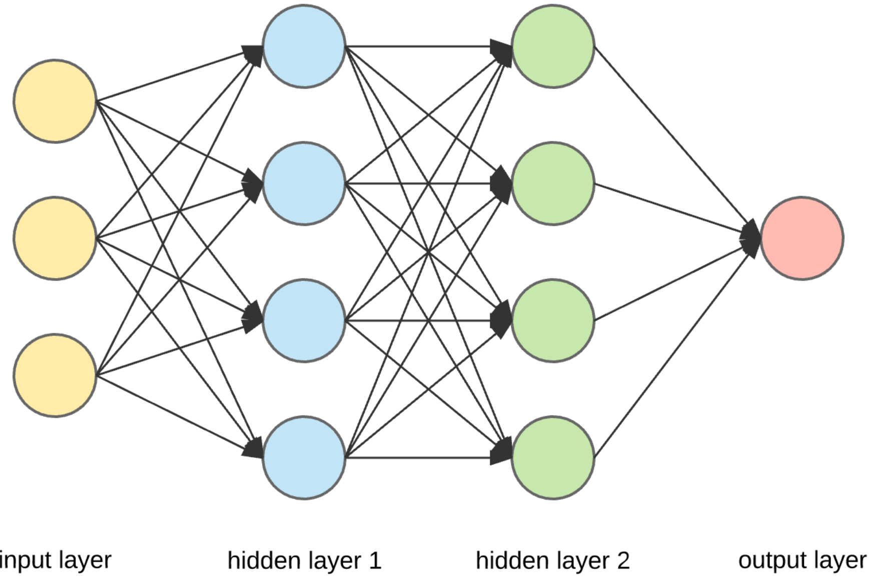
For each l:

$$\mu^l = \frac{1}{m}\sum_i {z_i}^l$$

$$\sigma^{l^2} = \frac{1}{m}\sum_i ({z_i}^l - \mu^l)^2$$

$$Z_{norm}{}^l = \frac{z^l - \mu^l}{\sqrt{\sigma^{l^2} - \varepsilon}}$$

$$\tilde{Z}^l = \gamma \, Z_{norm}{}^l + \beta$$



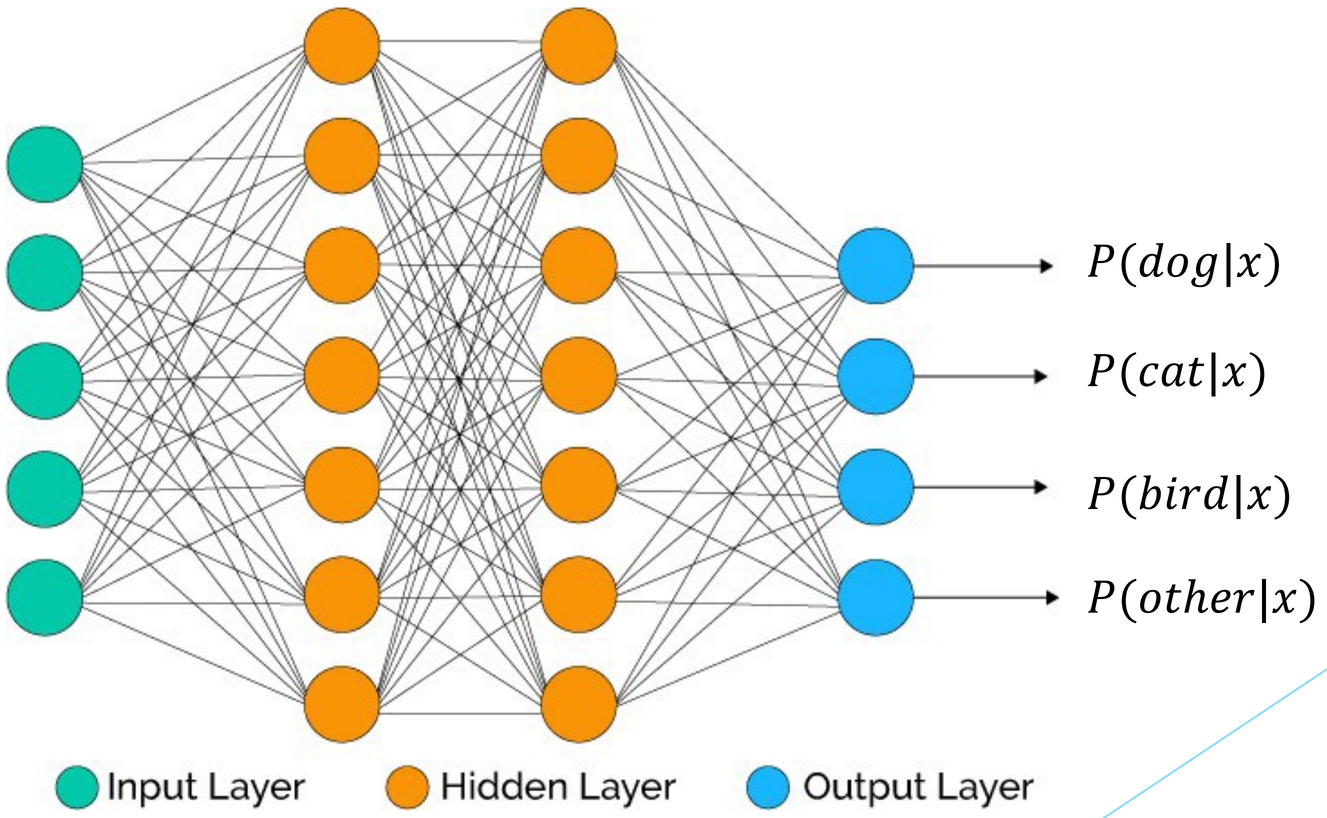
$$X \xrightarrow{(w^{[1]}, b^{[1]})} z^{[1]} \xrightarrow{(\gamma^{[1]}, \beta^{[1]})} \tilde{z}^{[1]} \xrightarrow{g(\tilde{z}^{[1]})} a^{[1]}$$

$$a^{[1]} \xrightarrow{(w^{[2]}, b^{[2]})} z^{[2]} \xrightarrow{(\gamma^{[2]}, \beta^{[2]})} \tilde{z}^{[2]} \xrightarrow{g(\tilde{z}^{[2]})} a^{[2]}$$

$$a^{[2]} \xrightarrow{(w^{[3]}, b^{[3]})} z^{[3]} \xrightarrow{(\gamma^{[3]}, \beta^{[3]})} \tilde{z}^{[3]} \xrightarrow{\sigma(\tilde{z}^{[3]})} a^{[3]} \longrightarrow \hat{y}$$

Softmax

Imagine you want an algorithm able to predict four classes, if the pictures is a dog, a cat, a bird or other



softmax

Activation function:

$$t_i = e^{(z_i{}^{[L]})}$$

$$a_i{}^{[L]} = \frac{t_i}{\sum_{j=1}^{n^{[L]}} t_j} = \frac{e^{(z_i{}^{[L]})}}{\sum_{j=1}^{n^{[L]}} e^{(z_j{}^{[L]})}}$$

$$\sum_{i=1}^{n^{[L]}} a_i{}^{[L]} = 1$$

Softmax loss function

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{n^{[L]}} y_j \log \hat{y}_j$$