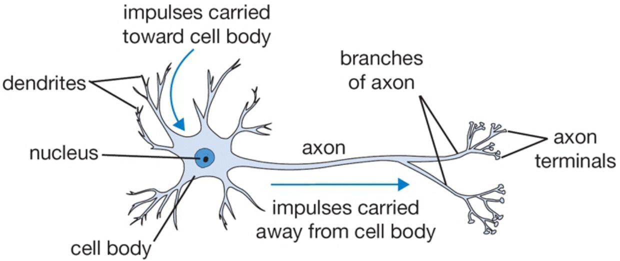
Introduction aux réseaux neuronaux

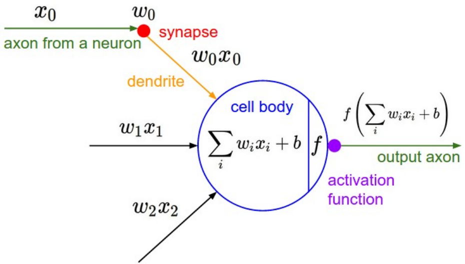
Olivier Ricou

2022

Un neurone



impulses carried toward cell body
dendrites
branches of axon
nucleus
axon
axon terminals
impulses carried away from cell body
cell body



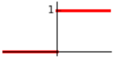
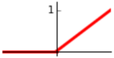
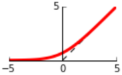
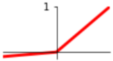
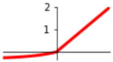
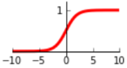
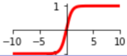
x_0
w_0
synapse
axon from a neuron
$w_0 x_0$
dendrite
cell body
$f\left(\sum_i w_i x_i + b\right)$
$w_1 x_1$
$\sum_i w_i x_i + b$
f
output axon
activation function
$w_2 x_2$

Les maths d'un neurone

1. $z = b + \sum_i w_i x_i$
2. $y = \sigma(z)$

avec

- les i entrées x_i
- b le biais
- w_i les poids
- σ la fct d'activation

Fonction	Definition	Courbe	Dérivée
Basic	$y = 1$ si $z \geq 0$ $y = 0$ sinon		Dirac
Rectifié (ou linéaire) ReLu	$y = \max(0, z)$		Heavyside
Softplus	$y = \ln(1 + e^z)$		logistique
Leaky ReLu	$y = 0.01\,z$ si $z < 0$ $y = z$ sinon		0.01 si z < 0 1 sinon
ELU Exp. Linear Unit	$y = \alpha(e^z - 1)$ si $z < 0$ $y = z$ sinon		$y + \alpha$ si z<0 1 sinon
Softmax	$y_k = \dfrac{e^{z_k}}{\sum_i e^{z_i}}$		$\dfrac{\partial y_k}{\partial z_k} = y_k\,(1 - y_k)$
Logistique (une sigmoïde)	$y = \dfrac{1}{1 + e^{-z}}$		$y\,(1 - y)$
Tangente hyperbolique	$y = \tanh(z)$		$1 - y^2$

Théorème d'approximation (A. Pinkus – 1999)

Soit une fonction d'activation $\sigma \in \mathcal{C}(\mathbb{R})$ avec σ **non polynomiale**, alors l'espace généré par une couche de neurones

$$\mathcal{M}_r(\sigma) := \left\{ \sum_{i=1}^{r} c_i \, \sigma(\mathbf{w}^i . \mathbf{x} + b_i), \mathbf{w}^i \in \mathbb{R}^d, c_i \text{ et } b_i \in \mathbb{R} \right\}$$

est dense dans $\mathcal{C}(\mathbb{R})$ au sens de la convergence uniforme sur tout compact,

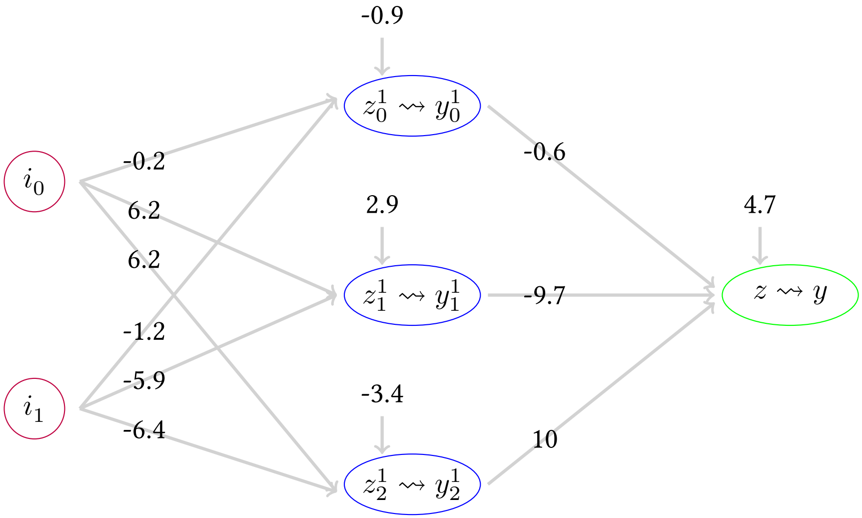
c.a.d. $\forall f \in \mathcal{C}(\mathbb{R}), \forall K$ compact de $\mathbb{R}$, $\forall \varepsilon > 0$, $\exists\, g \in \mathcal{M}_r(\sigma)$ t.q.

$$|f(\mathbf{x}) - g(\mathbf{x})| < \varepsilon$$

Ce théorème existe avec $\mathcal{C}^m(\mathbb{R})$.

⚠ Cela ne veut pas dire que c'est simple de converger. En fait c'est difficile !

Un premier réseau neuronal



Évaluer les couples d'entrée (1,1), (0,1), (1,0) et (0,0) avec σ une logistique.

Construction d'un réseau neuronal

Pour construire un réseau neuronal par apprentissage supervisé il faut :

- un grand jeu de données étiquetées par la sortie voulue
- définir l'architecture du réseau avec
  - le nombre de couches
  - les types de couches
  - le nombre de nœuds par couche
  - les fonctions d'activations
  - les connexions inter-couches
  - toutes astuces qui fonctionnent
- une fonction d'erreur pour guider la correction sur les poids
- une méthode pour faire converger le réseau (trouver les bons poids)

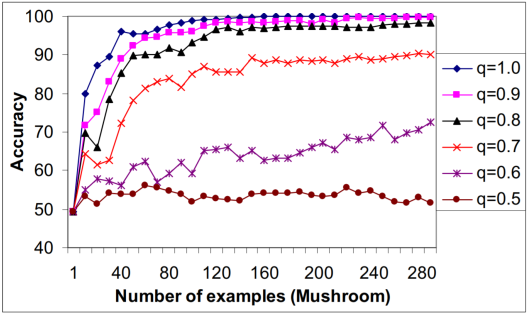
En cas de problème, on sacrifie un poulet.

Les données

Les données doivent être

- très nombreuses (assez pour définir toutes les inconnues du réseau)
- de bonne qualité (pour ne pas tromper le réseau)

On appronfondira avec des exemples et l'utilisation de Pandas pour nettoyer les données.



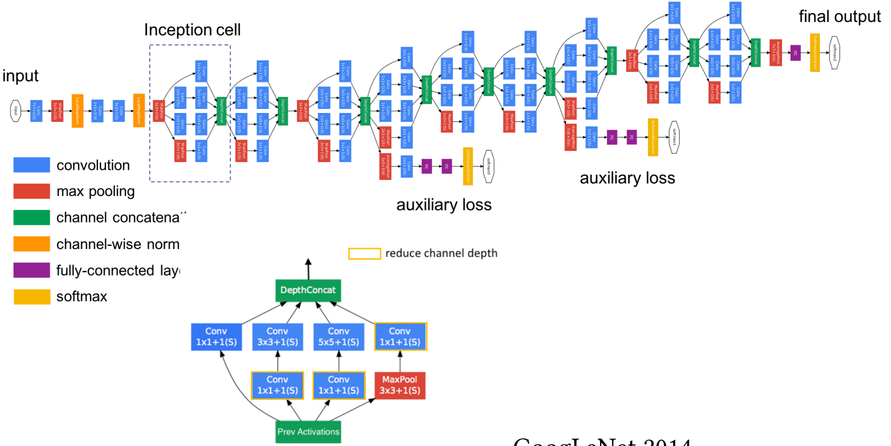
Est-ce un champignon ? Précision suivant la qualité des étiquettes.

source : Sheng et al. 2008

L’architecture du réseau

C’est la partie tactique et artistique.

L’étude des différents réseaux n’entre pas dans le cadre de ce cours d’introduction. On se limitera à quelques réseaux lors des TP.



GoogLeNet 2014

La fonction d'erreur

La fonction d'erreur indique de combien le réseau s'est trompé par rapport à la vérité terrain (y vs t). Elle doit

- être dérivable
- correspondre au problème traité

Cette fonction est aussi appelée fonction de coût (*cost function* ou *loss function* en anglais).

Exemples

- L'erreur quadratique $E = (y-t)^2$
- $E = \log(\cosh(y-t))$ quadratique puis linéaire lorsque l'écart croît
- L'entropie croisée pour des probabilités (valeurs entre 0 et 1)

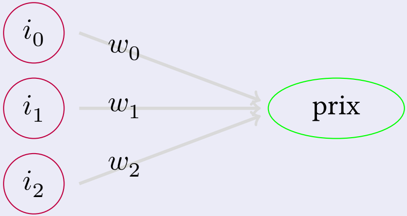
$$E = -\sum_k t_k \log y_k + (1-t_k)\log(1-y_k)$$

Une méthode pour trouver les bons poids

Comment l'erreur nous guide pour trouver les poids ?

Exemple

Vous êtes le directeur et tous les jours vous invitez votre équipe à déjeuner. Il y a le choix entre le plat A, B ou C. Vous payez chaque jour l'addition.



Avec les données [(5,3,2), 114], [(6,2,2), 108], [(3,4,5), 147] qui correspondent aux quantités de chaque plat et au prix global, déduire le prix de chaque plat par une méthode d'apprentissage. *Que proposez-vous ?*

Utilisons l'erreur pour corriger les poids

L'algorithme consiste à trouver les w_i qui minimisent l'erreur :

1. On initialise les poids à une valeur probable (disons 10 pour tous).
2. On corrige les poids au prorata de leur part dans l'erreur $E = y - t$:
   $w_j = w_j - \eta\, d_j$ avec $d_j = \dfrac{E \times i_j}{\sum_k i_k}$ et η petit pour éviter de sur-corriger.

Déroulons l'algorithme avec $\eta = 1/10$:

[(5,3,2), 114] Notre prix estimé est de 100.

- $d_0 = (y - t) \times i_0/10 = -7.0$ donc $w_0 = 10 + 0.70 = 10.7$
- $d_1 = (y - t) \times i_1/10 = -4.2$ donc $w_1 = 10 + 0.42 = 10.42$
- $d_2 = (y - t) \times i_2/10 = -2.8$ donc $w_2 = 10 + 0.28 = 10.28$

[(6,2,2), 108] Notre prix estimé est de 105.6 et on obtient $w_0 = 10.84$, $w_1 = 10.46$ et $w_2 = 10.33$

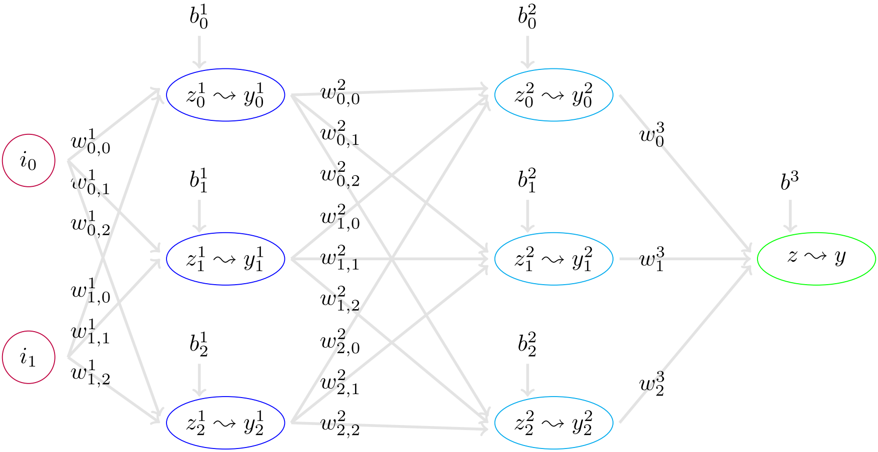
[(3,4,5), 147] Notre prix estimé est de 126.04 et on obtient $w_0 = 11.37$, $w_1 = 11.16$ et $w_2 = 11.20$

On peut rejouer les données jusqu'à converger

- la convergence peut être longue avec un petit η
- cela peut diverger avec un trop grand η

$$\boxed{\frac{E \times i_j}{\sum_k i_k} = \alpha \frac{\partial (y-t)^2}{\partial w_j}}$$

Rétropropagation du gradient



Calculons l'inflence du poids $w_{2,2}^2$ sur l'erreur quadratique E : $\dfrac{\partial E}{\partial w_{2,2}^2}$

Que vaut le gradient de E : ∇E ou $\dfrac{\partial E}{\partial \mathbf{w}}$? Pourquoi ce titre ?

GradiantTape pour calculer des dérivées partielles

Si vous désirez vérifier vos calculs, vous pouvez utiliser GradiantTape de Tensorflow. Le code suivant

```
x = tf.Variable(3.)

with tf.GradientTape() as tape:
    y = x**2
    z = y**2
    dz_dx = tape.gradient(z,x)
    print(dz_dx.numpy())
```

affiche 108.

Vérifions : $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} = 2\,y\;2\,x = 2\,x^2\;2\,x = 4\,x^3$

ce qui est donne bien 108 pour $x = 3$.

GradiantTape pour calculer automatiquement ∇E

Pour un réseau à 2 entrées enregistré dans la fonction `model`, on a :

```
x = tf.constant([[1.,0.]])
with tf.GradientTape() as tape:
    y = model(x)
    loss = tf.reduce_mean((1 - y)**2)
    grad = tape.gradient(loss,
                         model.trainable_variables)
```

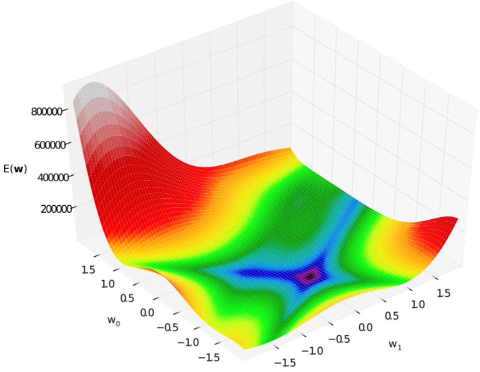
`grad` a l'ensemble des dérivées partielles de l'erreur par rapport aux poids du réseau ce qui correspond à ∇E.

On peut applique ce résultat à la main pour mettre à jour les poids ou le donner à un solveur (cf la suite).

L'ensemble des calculs pour un réseau est présenté sur :
`www.lrde.epita.fr/~ricou/iren/gradient_tape.html`

La méthode du gradient

Le but est de trouver le vecteur **w** qui minimise notre erreur E.



L'algorithme de descente du gradient est, avec un $\mathbf{w}_0$ choisi, :

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla E(\mathbf{w}_t) \quad (1)$$

jusqu'a ce que l'erreur soit inférieure à un seuil choisi.

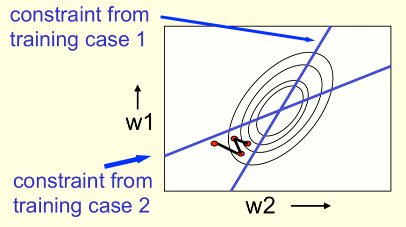
Représentation graphique

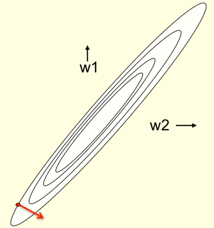
Commençons par un cas simple : l'erreur est une fonction convexe.

Dans le cas où on modifie les poids après chaque donnée, on risque fort de zigzaguer.

→ travailler par paquet de données.

Notion de *batch*

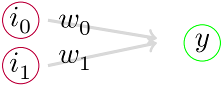




Lorsque l'ellipse est allongée, son gradient est quasiment orthogonal à son axe long ce qui n'est pas du tout la bonne direction vers le minimum.

→ la convergence sera longue

Travail sur les données – Jouer sur l'échelle



$$y_{w_0,w_1}(i_0, i_1) = w_0\, i_0 + w_1\, i_1$$
$$E_{w_0,w_1}(i_0, i_1) = (y - t)^2$$

Soit ces deux jeux de données :

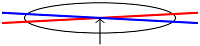
i_0	i_1	t
0.1	10	2
0.1	-10	0

i_0	i_1	t
1	1	2
1	-1	0

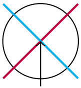
Les fonctions d'erreur correspondantes ont les formes suivantes :

$0.1\, w_0 + 10\, w_1 = 2$

$0.1\, w_0 - 10\, w_1 = 0$



(10, 0.1)



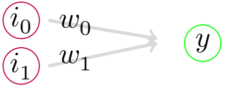
(1, 1)

$w_0 - w_1 = 0$

$w_0 + w_1 = 2$

$\longrightarrow$ **normaliser** les données pour éviter des fonctions d'erreur écrasées.

Travail sur les données – Translation



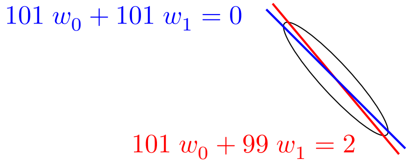
$$y_{w_0,w_1}(i_0, i_1) = w_0\, i_0 + w_1\, i_1$$
$$E_{w_0,w_1}(i_0, i_1) = (y - t)^2$$

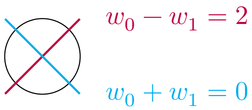
Soit ces deux jeux de données :

i_0	i_1	t
101	101	0
101	99	2

i_0	i_1	t
1	1	0
1	-1	2

Les fonctions d'erreur correspondantes ont les formes suivantes :

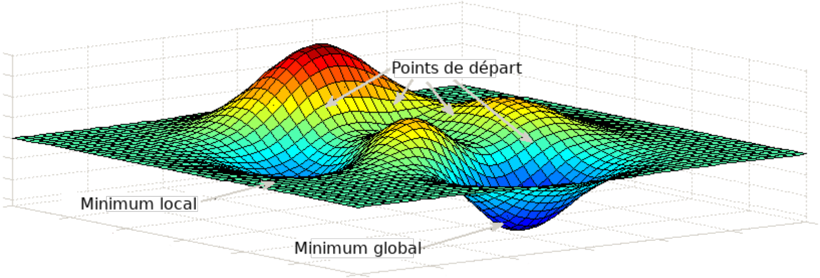




$\longrightarrow$ **centrer** les données pour éviter des fonctions d'erreur écrasées.

Les minimums locaux

La fonction d'erreur n'est pas souvent convexe. Il faut s'attendre à avoir des minimums locaux.



Le point de convergence dépend du point de départ d'où le risque de finir dans un minimum local.

Comment sortir d'un minimum local pour rejoindre un minimum global ?

Les solveurs

Pour contrer ces différents problèmes (et d'autres) on a construit différents solveurs.

- Moment et Nesterov (introduisent une inertie)
- RMSprop (varie le taux d'apprentissage en fct des poids)
- Adagrad (développement limité à l'ordre 1)
- Adadelta (comme Adagrad mais avec une fenêtre glissante)
- Adam (moment à l'ordre 2)
- ...

Itération par paquet de données ou 1 par 1 $\rightarrow$ gradient stochastique.

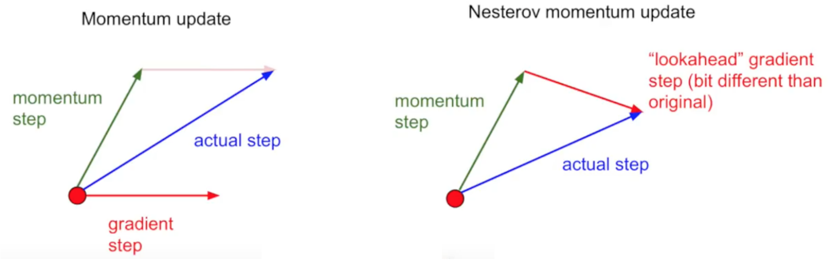
Moment et Nesterov

L'idée du moment est de donner une inertie α à la méthode :

1. $\mathbf{v} \leftarrow \alpha\, \mathbf{v} + \beta\, \nabla E(\mathbf{w})$
2. $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{v}$

Nesterov propose de travailler sur les données mise à jour :

1. $\mathbf{v} \leftarrow \alpha\, \mathbf{v} + \beta\, \nabla E(\mathbf{w} + \alpha\, \mathbf{v})$
2. $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{v}$



Cela peut aider à sortir des trous et aténuer les zigzags.

RMSprop

Le coefficient d'apprentissage, η, influence fortement la convergence.

On peut choisir autant de η_i qu'il y a de paramètre : $\eta_i = \varepsilon \mu_i \dfrac{\partial E}{\partial w_i}$
avec

- $\mu_i = \mu_i + 0.05$ si $\dfrac{\partial E}{\partial w_i}(t) \dfrac{\partial E}{\partial w_i}(t-1) > 0$
- $\mu_i = \mu_i \times 0.95$ sinon

Malheureusement cela marche mal avec les mini-batches
(9 corrections d'un poids de 0.1 suivies d'une de -0.9 devrait faire du surplace, mais pas avec cette méthode).

Aussi on préfère utiliser une inertie temporelle et l'algorithme est :

1. $\mathbf{g} = \nabla E(\mathbf{w})$
2. $\mu \leftarrow \alpha\, \mu + (1-\alpha)\, \mathbf{g}^2$ avec α = 0.9 par exemple
3. $\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{\sqrt{\mu + \varepsilon}}\, \mathbf{g}$ avec ε pour éviter des pas trop grands si $\mu \to 0$

Adagrad

On cherche le $\mathbf{w}$ qui minimise E donc tel que $\nabla E(\mathbf{w}) = 0$.

Au pas de temps t on est au point $\mathbf{w}_t$ donc on cherche $\boldsymbol{\delta}\mathbf{w}$ tel que $\nabla E(\mathbf{w}_t + \boldsymbol{\delta}\mathbf{w}) = 0$ donc, avec un développement limité, tel que

$$\nabla E(\mathbf{w}_t) + \boldsymbol{\delta}\mathbf{w}\, \nabla^2 E(\mathbf{w}_t) + o(||\boldsymbol{\delta}\mathbf{w}||) = 0$$

avec $\nabla^2 E$ la matrice hessienne de E. Ainsi l'algorithme itératif est :

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \boldsymbol{\delta}\mathbf{w} = \mathbf{w}_t - \left(\nabla^2 E_t(\mathbf{w}_k)\right)^{-1} \nabla E_t(\mathbf{w}_t)$$

Calculer l'inverse de la matrice hessienne est bien trop cher ! Aussi on va chercher quelque chose qui lui ressemble, V_t pour Adagrad :

1. $\mathbf{g}_t = \nabla E_t(\mathbf{w}_t)$
2. $\mathbf{V}_t = \left[\mathrm{diag}\left(\sum_{i=1}^t \mathbf{g}_i^2\right)\right]^{1/2}$
3. $\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \mathbf{V}_t^{-1}\, \mathbf{g}_t$

XOR & Co. complet en Numpy

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, t):
    learning_rate = 0.1
    w1 = np.random.rand(2, 3)
    w2 = np.random.rand(3, 1)

    for epoch in range(10000):
        layer1 = sigmoid(x @ w1)
        output = sigmoid(layer1 @ w2)
        delta2 = -2 * (t-output) * output * (1-output)
        delta1 = (delta2 @ w2.T) * layer1 * (1-layer1)
        w2 -= learning_rate * layer1.T @ delta2
        w1 -= learning_rate * x.T @ delta1

    return output.flatten()
```

XOR & Co. complet en Numpy (2)

À l'usage :

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

print("X␣=␣",X[0], X[1], X[2], X[3])
print("OR", neural_network(X, np.array([[0, 1, 1, 1]]).T))
print("NOR", neural_network(X, np.array([[1, 0, 0, 0]]).T))
print("AND", neural_network(X, np.array([[0, 0, 0, 1]]).T))
print("NAND", neural_network(X, np.array([[1, 1, 1, 0]]).T)
print("XOR", neural_network(X, np.array([[0, 1, 1, 0]]).T))
```

Code original : https://gist.github.com/svpino/e54ff030c424cefaffeec1bd690042cc.

Exemples de convergence

Regardons à quelle vitesse convergent différentes méthodes suivant la forme de la fonction d'erreur.

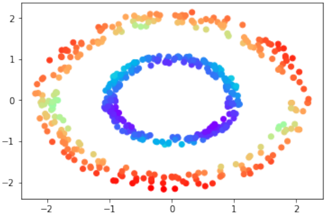
- Point selle
- Fonction de Beale
- Presque point selle

source : Sebastien Ruder http ://ruder.io/optimizing-gradient-descent/

Trois types de réseaux neuronaux

Regardons quelques exemples de réseaux neuronaux :

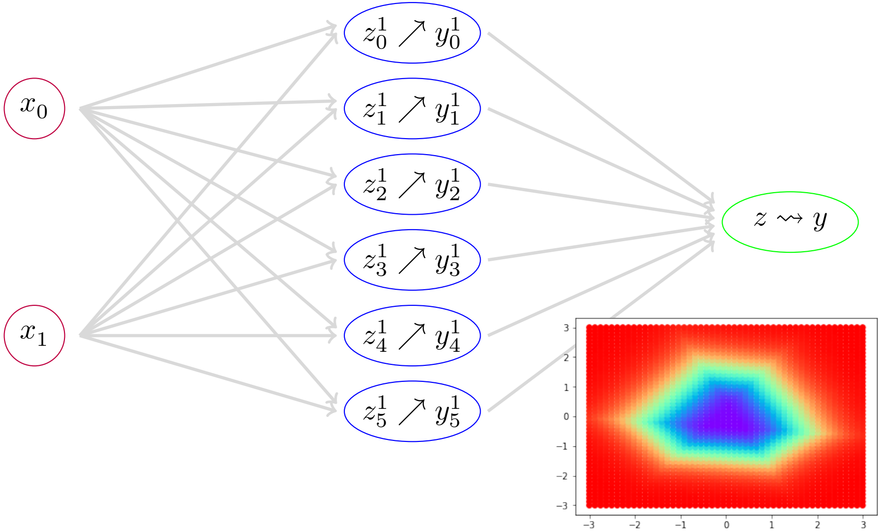
- un réseau simple pour séparer des données
- un réseau récursif pour faire des additions
- un réseau de convolution pour comprendre une image



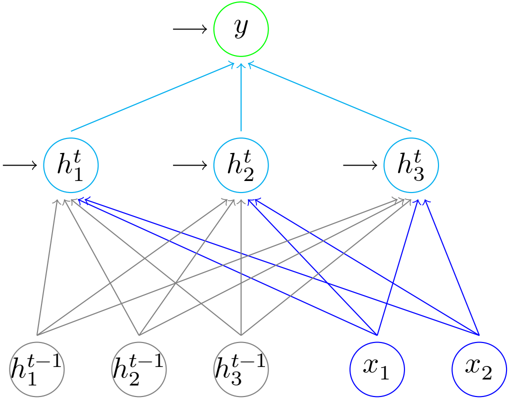
Une idée pour séparer les données sur deux cercles ?

Séparation

Puisque Relu définit un demi-plan, on va utiliser 6 Relu ($\nearrow$) pour faire un cercle grossier et une sigmoïde ($\rightsquigarrow$) pour séparer les 2 cercles :



Un réseau récursif pour faire une addition

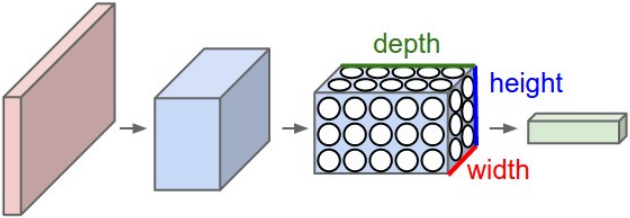


On veut calculer
0101011 + 1001110
pour cela on entre les valeurs au réseau en partant des unités comme pour faire une addition à la main :
1 , 0 puis 1 , 1 puis 0 , 1...

Les cellules grises sont la mémoire des opérations précédentes.

Des CNN pour voir

Les *Convolution Neural Network* sont la grande réussite du *deep learning*. Le principe est de travailler sur des images pour en extraire les caractéristiques



En entrée nous avons une image $N \times N \times 3$ (en RGB) dont nous diminuons la surface à chaque couche du réseau pour augmenter sa profondeur.

À la fin on peut voir l'image comme un vecteur de caractéristiques.

Ensuite (pas sur le dessin) on peut utiliser un réseau neuronal classique pour classer l'image.

Quelques convolutions

Continue 1D :

$$(f * g)(x) = \int_{-\infty}^{+\infty} f(x-t)\ g(t)\,\mathrm{d}t = \int_{-\infty}^{+\infty} f(t)\ g(x-t)\,\mathrm{d}t$$

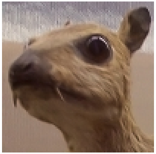
Discrète 2D :

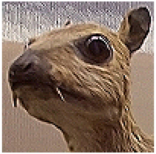
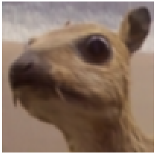
$$(f * \omega)(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(a+dx, b+dy)\ f(x+dx, y+dy)$$

f est l'image. ω est le noyau, son support est $[-a,a] \times [-b,b]$.

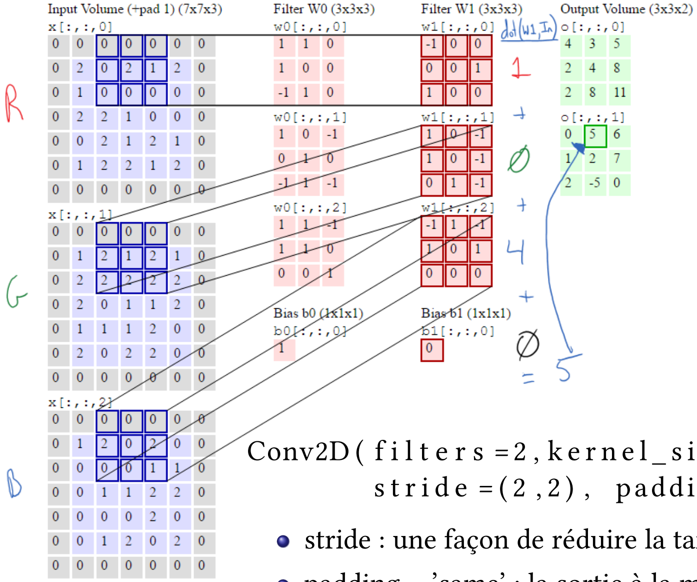
Exemple de noyaux ω (WP Noyau_(traitement_d'image)) :

$$\begin{bmatrix} 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \qquad \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \qquad \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$



Les filtres de convolution



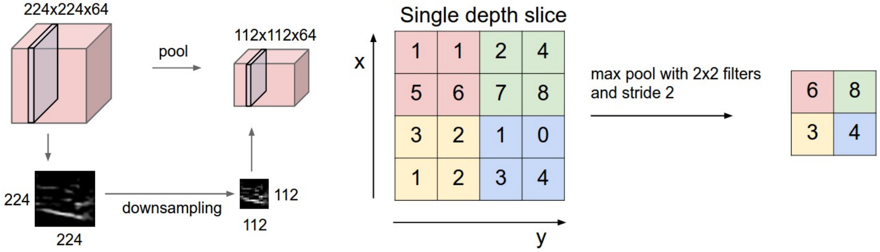

```
Conv2D( filters=2, kernel_size=(3,3),
        stride=(2,2), padding='same')
```

- stride : une façon de réduire la taille d'une image.
- padding = 'same' : la sortie à la même taille.

Diminuer la surface

L'exemple de convolution précédent saute un pas ce qui réduit la surface. Mais pas de saut et plus de filtres → le nombre de données explose.

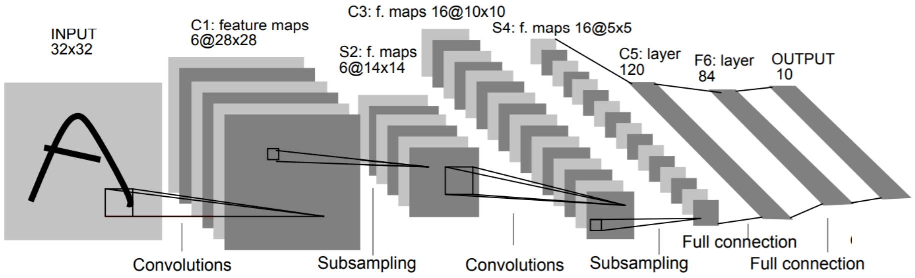
Aussi on réduit la surface de l'image à fur à et mesure qu'on augmente sa profondeur (*pooling*).



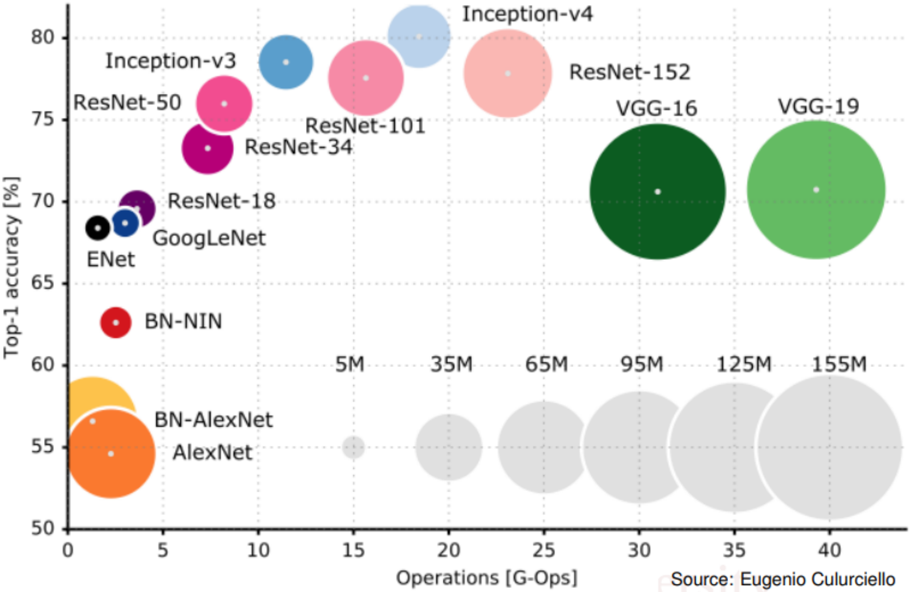
Le choix du maximum est le plus utilisé. On pourrait faire une moyenne mais cela risque d'atténuer l'image.

Le Net 5

Voici le premier CNN qui a bien fonctionné pour lire les codes postaux sur les enveloppes de la poste. Il a été créé par Yann Le Cun dans les années 90.

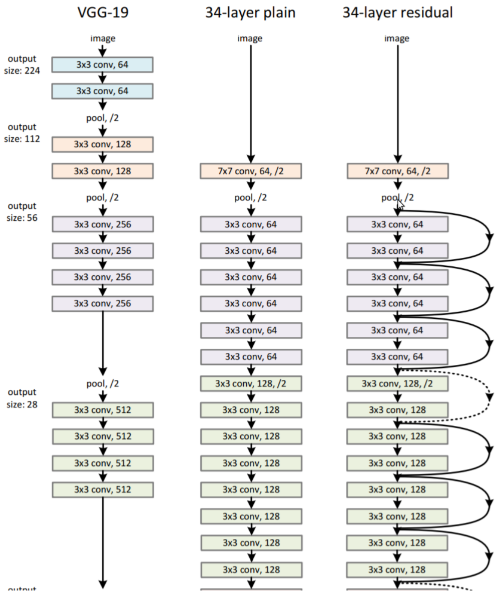


Évolution des CNN



Source: Eugenio Culurciello

De plus en plus compliqué



On ajoute des trucs pour améliorer les résultats (ou pour converger).

Ici l'idée est reprendre des données antérieures pour ne pas trop oublier. Le saut correspond à l'opération :

$$\mathbf{y} = F(\mathbf{x}, \mathbf{w}) + \mathbf{x}$$