Réseaux de Neurones/Deep Learning

F. Panloup
LAREMA-Université d'Angers

—

Cours : Apprentissage Statistique en Grande Dimension

—

Réseaux de neurones artificiels : un peu d'histoire

- Réseaux de neurones dits artificiels ou formels initialement développés pour modéliser l'activité du cerveau.
- Fin des années 1950 : travaux des neurologues Warren McCulloch et Walter Pitts intitulé "What the frog's eye tells the frog's brain" (tentative de modélisation de l'activité d'un ensemble de neurones)
- Modélisation à l'aide une fonction de transfert qui prend en entrée une combinaison linéaire d'informations et qui renvoie un signal lorsque l'information d'entrée dépasse un certain seuil (Activation)
- Fonctionnement du cerveau modélisé par un empilement de structures simples.

Point de vue biologique

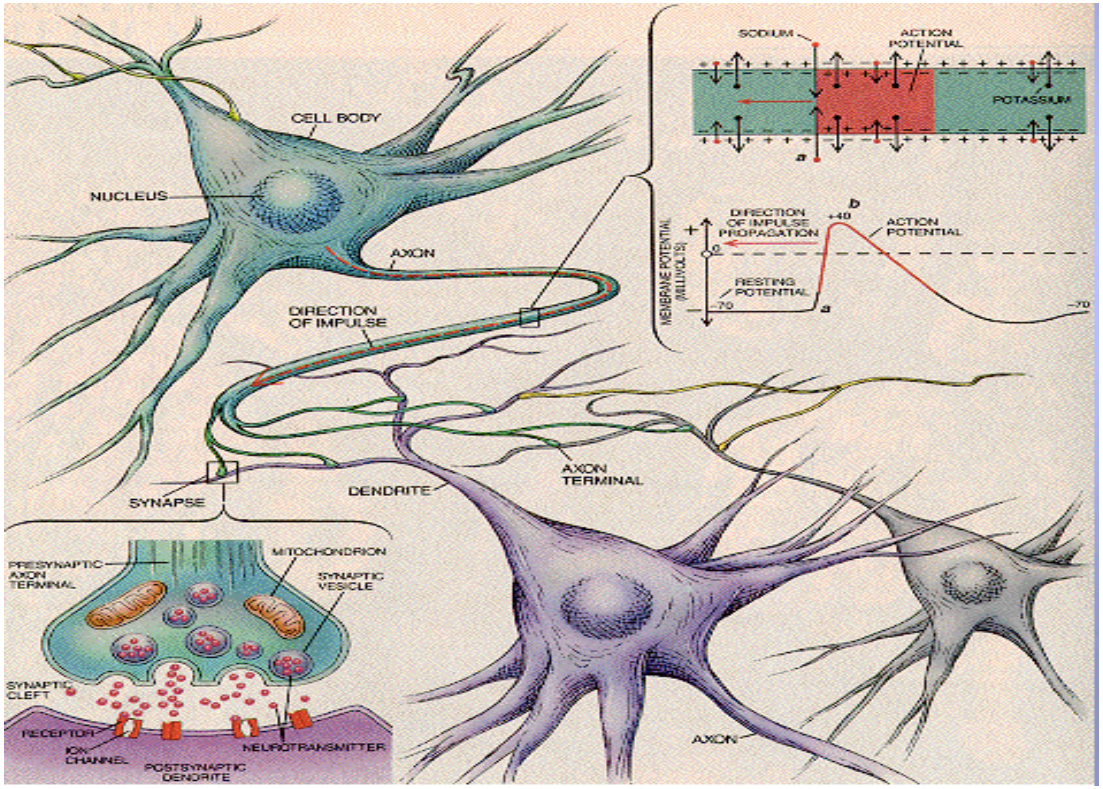


Figure 1: Neurones biologiques, axones et dendrites, figure issue de `http://www.lacim.uqam.ca/~chauve/Enseignement/BIF7002/Rapports/Simon-Beaulne/neurones.htm`

Point de vue formel

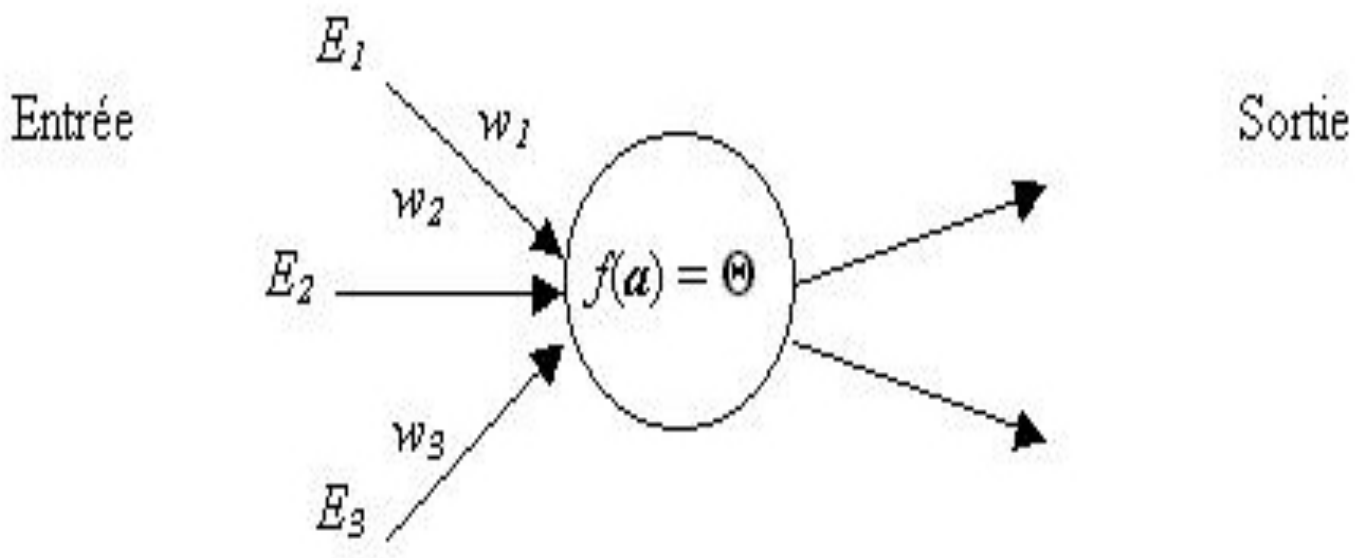


Figure 2: Neurone formel : figure issue de http://www.lacim.uqam.ca/~chauve/Enseignement/BIF7002/Rapports/Simon-Beaulne/neurones.htm

Des réseaux de neurones au Deep Learning

- L'empilement de ces briques (Couches) élémentaires peut modéliser un système complexe.
- Un tel système est appelé réseau de neurones.
- Deep Learning (ou Apprentissage profond) : réseau de neurones dont le nombre de couches et de neurones par couches est élevé.
- Remarque : La structure de ces couches (convolutionnelle...) joue aussi un rôle dans la terminologie.

Réseau de neurone : point de vue "apprentissage statistique"

- Ces différents empilements génèrent une famille de fonctions paramétriques $(f_\theta)_{\theta \in \mathbb{R}^p}$.
- Les réseaux de neurones peuvent alors être compris comme une famille de fonctions bien adaptée à la modélisation de phénomènes non linéaires.
- Comment mesurer mathématiquement cette capacité de modélisation ? Difficile.... Quelques résultats commencent à émerger...
- Par contre, numériquement, les performances sont remarquables...

Performances en traitement d'images

- Historiquement, les performances les plus remarquables des réseaux de neurones apparaissent d'abord en traitement d'images (puis se généralisent ensuite à d'autres problèmes : *Alphago*, reconnaissance de texte, reconnaissance vocale, robotique...).

- 2012 : modèle *Alexnet* utilisé dans [**?**] "écrasant" le Challenge *Image-Net Large Scale Visual Recognition Challenge* (INLSVRC) en obtenant moins de 16% d'erreurs contre 26% au minimum pour ses concurrents.

- S'explique par l'utilisation *d'architectures* de réseaux très adaptées à l'extraction d'information (Couches de convolution/pooling) notamment et une réelle utilisation de Deep Learning : 9 couches et près de 60 millions de paramètres à régler...

- et par des capacités informatiques (GPU) qui permettent réellement l'optimisation de ces paramètres.

Différents types de réseaux de neurones

Comme indiqué plus haut, différentes types de réseaux de neurones sont à distinguer :

1. Perceptron Multicouches (modèle "historique")
2. CNN : Convolutional Neural Networks (Réseaux de Neurones à Convolution adaptés pour le traitement d'images en particulier)
3. RNN : Recurrent Neural Networks (Réseaux de neurones récurrents, adaptés pour les données séquentielles telles que les données textuelles par exemple).
4. GAN : Generative Adversarial Neural Networks (Réseaux de Neurones Antagonistes Génératifs, utilisés pour la génération d'images et plus généralement pour la "création" artificielle)
5. ....

Librairies et Modules sous Python ou R

- Tensorflow/Keras : module "historique". https://keras.io/, https://keras.rstudio.com/
- Pytorch (développé par Facebook depuis 2016) : https://pytorch.org/
- Pour une comparaison entre Tensorflow et Pytorch, voir par exemple https://penseeartificielle.fr/meilleur-framework-machine-learning-2019/.
- Pytorch gagne du terrain dans le milieu de la recherche et donc à terme dans les entreprises probablement.

Réseaux de Neurones Artificiels : définitions

Définition

Un réseau de neurones artificiel est une famille de fonctions paramétriques que l'on notera

$$\{f_\theta : \mathbb{R}^p \mapsto \mathbb{R}, \theta \in \Theta\},$$

où $\Theta \subset \mathbb{R}^m$ *(m grand en général).*

Remarques : on fait ici l'hypothèse que l'entrée x est un vecteur de $\mathbb{R}^p$ et que la sortie $y = f_\theta(x)$ est un réel. Cette hypothèse, en particulier sur la sortie n'est pas adaptée à tous les contextes mais nous conserverons dans la suite ce formalisme pour simplifier.
A ce stade, il s'agit d'une définition très générale.
Cadre : Dans ce cours, Les réseaux de neurones seront envisagés pour des problèmes de régression ou de classification.

Neurone Artificiel

On s'intéresse à la structure d'un neurone dit artificiel ou formel qui a vocation à tenter de modéliser le fonctionnement d'un neurone biologique basé sur :

- des synapses : points de connexion entre neurones,
- des dentrites : entrées du neurones,
- des axones : sorties du neurone vers d'autres neurones,
- un noyau : activateur des sorties en fonction des stimulations en entrée.

Définition

Un neurone artificiel (d'indice j) est une fonction h de la variable d'entrée $x = (x_1, \dots, x_d)$ construite à l'aide de poids $w = (w_1, \dots, w_d)$, d'un biais $b \in \mathbb{R}$ et d'une fonction d'activation ϕ, définie par :

$$\forall x \in \mathbb{R}^d, \quad h(x) = \phi(\langle w, x \rangle + b)$$

où $\langle w, x \rangle = \sum_{i=1}^{d} w_i x_i$.

Représentation graphique

Remark

Lorsque $\phi = Id$, alors on retrouve la régression linéaire classique.

La figure 3 résume la définition précédente.

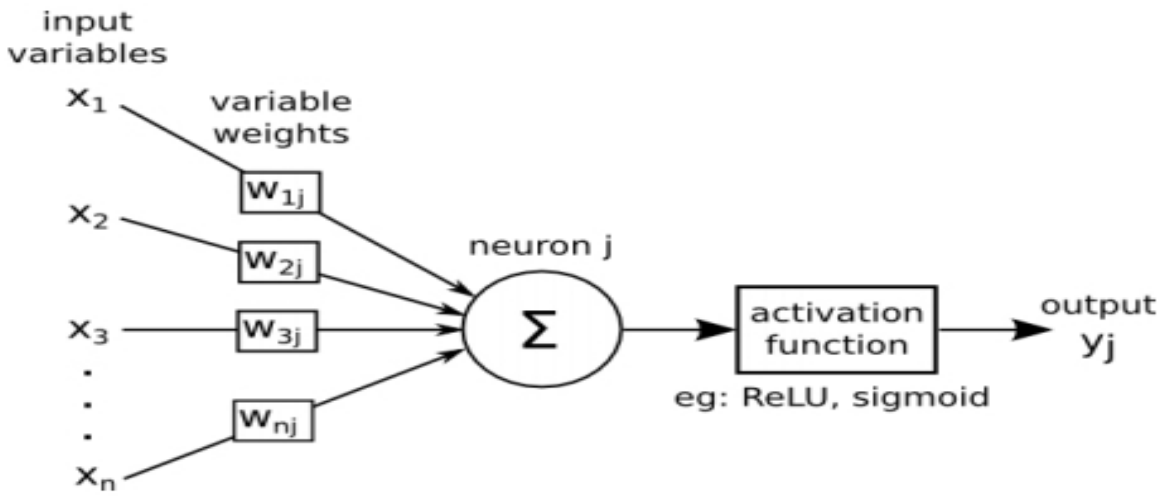


Figure 1: source: andrewjames turner.co.uk

Figure 3: Structure d'un neurone

Fonctions d'activation

Ces fonctions ont vocation à reproduire le fonctionnement d'un neurone. La fonction la plus naturelle est donc

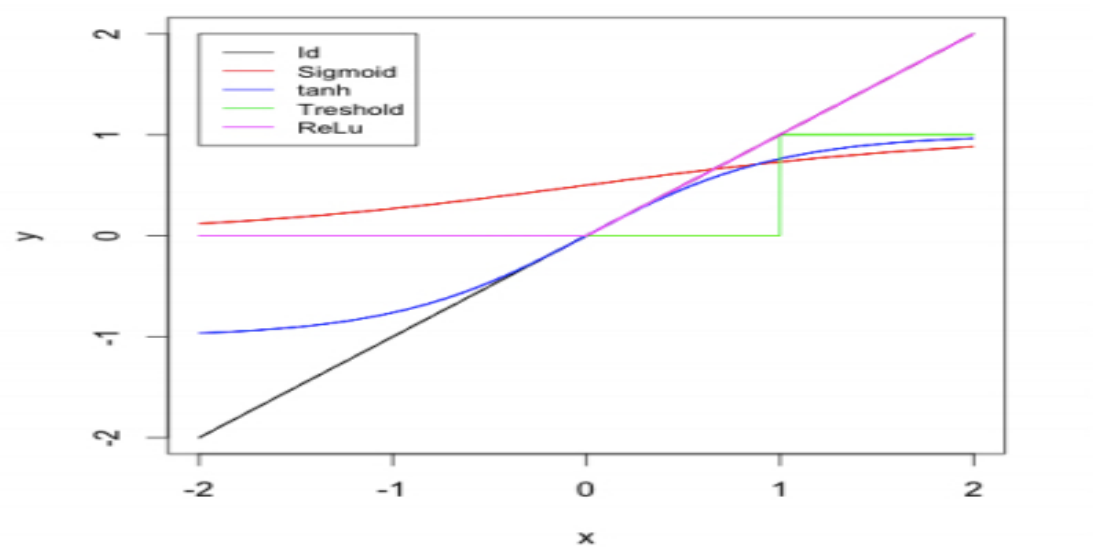
- La fonction seuil (threshold), définie par : $\phi(x) = \mathbf{1}_{x \geq 0}$. Néanmoins, en pratique, on utilise plutôt les fonctions suivantes :

- La fonction ReLU (Rectified Linear Unit), définie par : $\phi(x) = \max(0, x)$ qui est un peu moins "binaire" que la fonction seuil.

- La fonction Sigmoïde (réciproque de *logit*), définie par : $\phi(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$ (qui croît de 0 et 1)

- La fonction Tangente Hyperbolique : $\phi(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ qui croît de -1 à 1.

Fonctions d'activation



- Historiquement, la sigmoïde très utilisée en tant que fonction régulière.
- C'est aussi son défaut : fonction trop plate qui rend la recherche de paramètres optimaux difficile. On lui préfère aujourd'hui la fonction *ReLU* (bien que non différentiable en 0). On peut ajouter un léger biais à la fonction ReLU

$$\phi(x) = \max(x, 0) + \alpha \min(x, 0)$$

Fonctions d'activation

- Pour la dernière couche, on utilise une fonction d'activation adaptée au problème, *i.e.* qui renvoie une réponse de type régression ou classification selon les cas (en app. supervisé). Par exemple,
  - En classification binaire : la fonction Sigmoïde.
  - En classification multiclasses : la fonction Softmax à valeurs dans l'ensemble des probabilités sur $\{1, \dots, K\}$, définie pour $x \in \mathbb{R}^K$ par :

$$\mathrm{softmax}(x) = \left(\frac{e^{x_k}}{\sum_{i=1}^{K} e^{x_i}} \right)_{k \in \{1,\dots,K\}}.$$

- Sigmoïde est un cas particulier de softmax (quitte à multiplier par e^{-x_1}).

- Ces fonctions génèrent un *score* : une approximation de $\mathbb{P}(Y = k | X = x)$.

Softmax

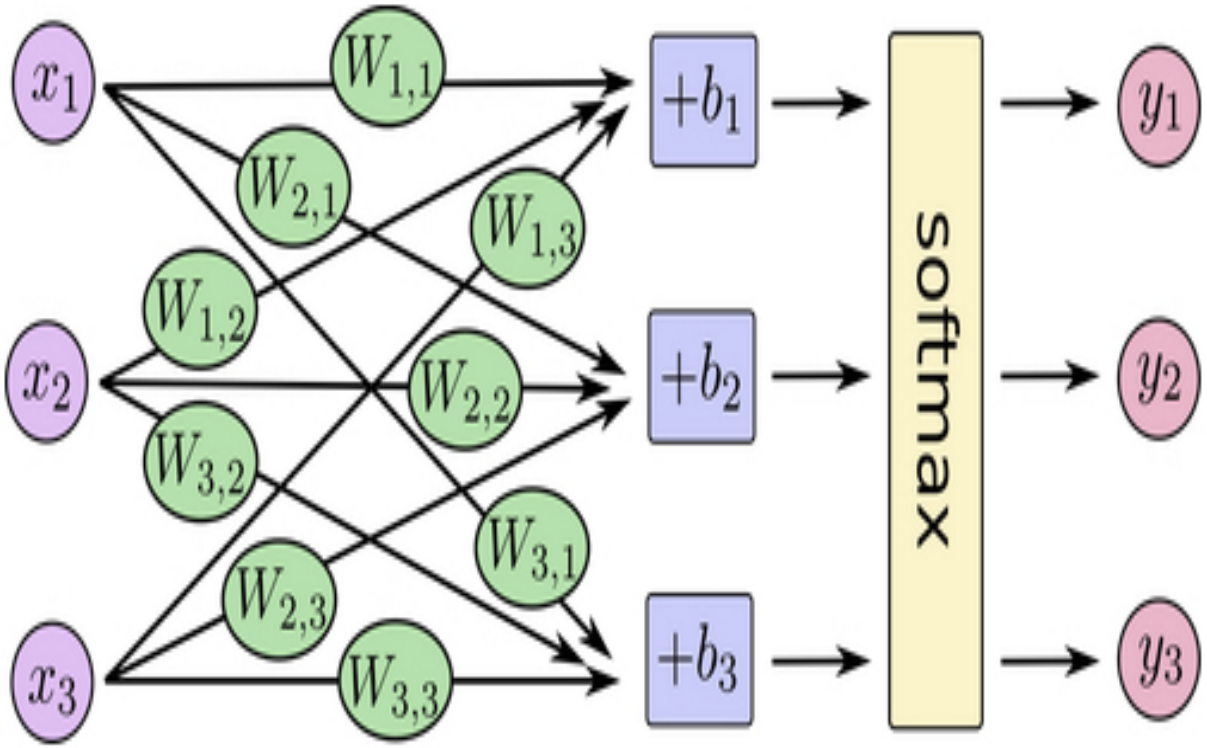


Figure 4: Dernière couche avec application de softmax

Le perceptron multicouches

Definition

Le perceptron est une structure composée d'une ou plusieurs couches cachées de neurones où la sortie d'un neurone devient l'entrée du suivant.

Remark

On peut aussi envisager des modifications où la sortie d'un neurone est l'entrée d'un neurone de la même couche ou d'un neurone inférieur.

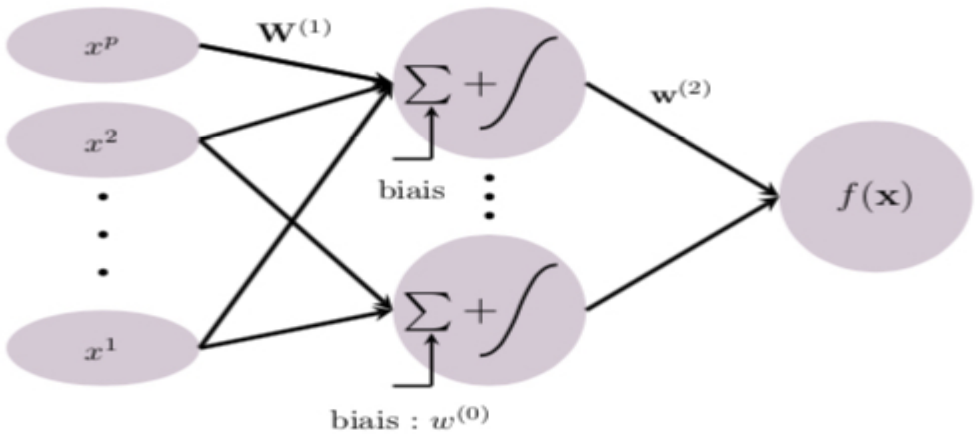


Figure 5: Exemple de perceptron à une couche cachée (et deux neurones dans la couche cachée)

Structure du perceptron multicouches

La structure est la suivante :

- Entrée : vecteur $x = (x_1, \ldots, x_p)$.
- Notons k l'indice d'un neurone de la première couche cachée (intermédiaire). Celui-ci peut être défini par :

$$h^{(k,1)}(x) = \phi_1(\langle W^{(k,1)}, x \rangle + b_{k,1}).$$

- Une fonction d'activation par couche cachée (ϕ_m pour la couche m).
- poids $W^{(k,1)}$ et biais $b_{k,1}$ dépendent de chaque neurone afin que ces derniers se "déclenchent" selon des règles différentes.
- La sortie de la couche 1 devient l'entrée de la couche suivante (entrée de dimension le nombre de neurones de la couche précédente).
- Même construction pour la couche ℓ avec une fonction d'activation ϕ_ℓ. . . . Entités qui peuvent être vues comme des filtres.
- La dernière couche, elle, à pour rôle de délivrer une réponse $\hat{f}(x)$.

Structure du perceptron multicouches

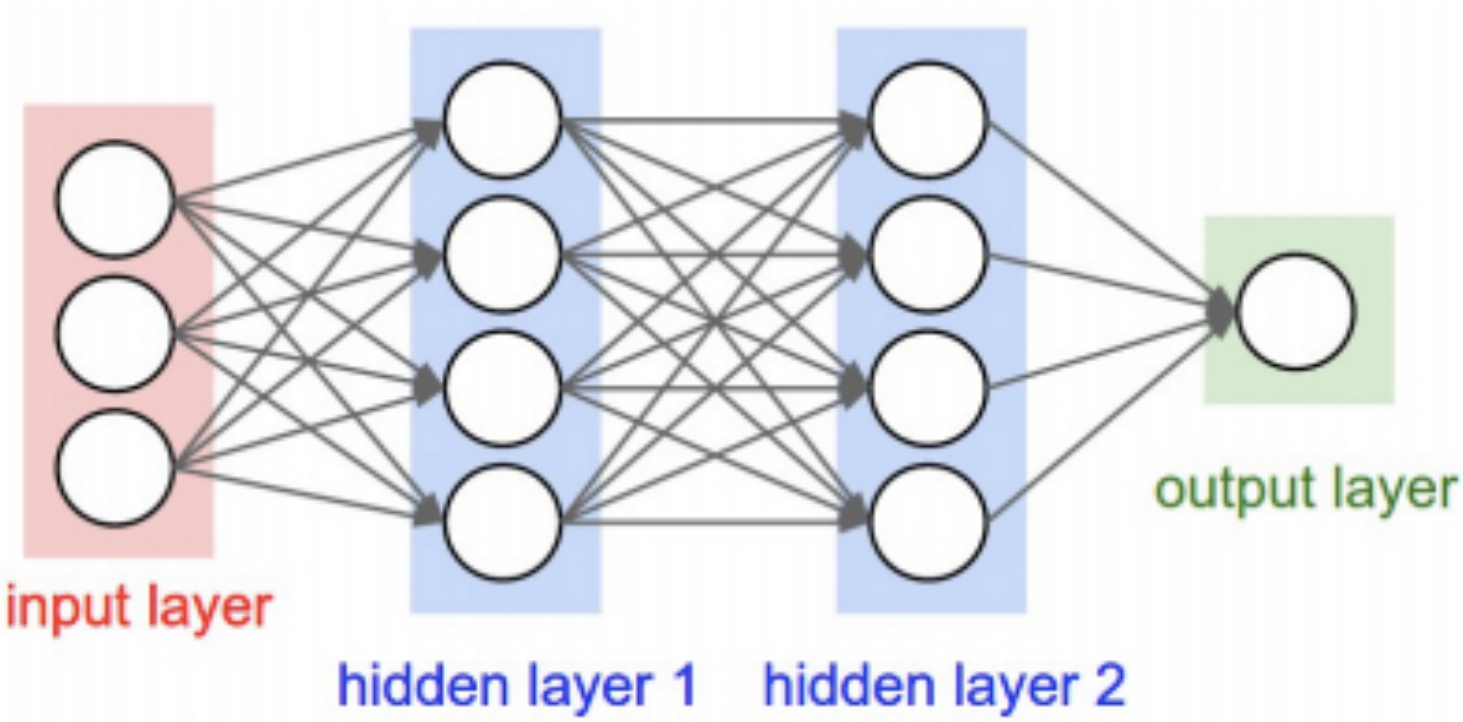


Figure 6: Perceptron à deux couches

Construction itérative du réseau pour des paramètres fixés :

- $h^{(0)}(x) = x \in \mathbb{R}^d$, nb de couches cachées L, nb de neurones par couche d_ℓ ($d_0 = d$).
- Pour $\ell = 1, \dots, L$,

$$a^{(\ell)}(x) := b^{(\ell)} + W^{(\ell)} h^{(\ell-1)}(x) \quad \text{où}$$
$$b^{(\ell)} \in \mathbb{R}^{d_\ell}, \quad W^{(\ell)} \in \mathcal{M}(d_\ell, d_{\ell-1}),$$
$$h^{(\ell)}(x) := \phi_\ell(a^{(\ell)}(x)) := (\phi_\ell(a_1^{(\ell)}(x)), \dots, \phi_\ell(a_{d_\ell}^{(\ell)}(x)),$$
où ϕ_ℓ est une fonction d'activation.

- Pour $\ell = L+1$ (dernière couche), même principe mais on note Ψ (par exemple, softmax en multiclasses) la fonction de sortie :

$$a^{(L+1)}(x) := b^{(L+1)} + W^{(L+1)} h^{(L)}(x) \quad \text{où}$$
$$h^{(L+1)}(x) := \Psi(a^{(L+1)}(x)).$$

Paramètres

Pour une suite de fonctions d'activation fixée, on a donc une classe de fonctions f_θ, $\theta \in \Theta$ où Θ est défini par

$$\Theta = \{(W^{(\ell)}, b^{(\ell)}), \ell = 1, \dots, L+1\}.$$

- Espace de dimension (rapidement) grande.
- hyperparamètres L; d_1,..., d_L
- En pratique, on peut dans un premier temps fixer les hyperparamètres pour la modélisation puis tenter de comparer différents modèles (*i.e.* avec des hyperparamètres ou des fonctions d'activation différentes).
- Choix potentiellement adaptatif des paramètres.

Exercice : Considérons un réseau à une couche cachée avec 3 neurones et une sortie quantitative et une entrée de dimension 2. Précisez les paramètres à estimer ainsi que la dimension de l'espace Θ.

Théorèmes d'approximation universelle

Question : Quelles fonctions peut on approcher par des réseaux de neurones ?

- Question fondamentale pour l'apprentissage.
- Comme indiqué plus haut, peu de résultats mathématiques.
- Résultats les plus courants : théorèmes d'approximation universelle qui expriment la densité des réseaux de neurones dans des espaces de fonctions généraux.
- Très récemment, résultats qui montrent ce que peut approcher un réseau à 2 couches vs un réseau à 1 couche. Résultats plus "pertinents" (non présentés ici) : voir par exemple
  https://www.math.univ-toulouse.fr/%7Efmalgouy/enseignement/downloadMva_deep_L/theorie_deep_learning.pdf

Théorème d'approximation universelle

On énonce ici un résultat de Hornik (1991).

Théorème

Soit $\phi : \mathbb{R} \mapsto \mathbb{R}$ une fonction bornée continue croissante et non-constante. Soit K un compact de $\mathbb{R}^d$. Alors, l'ensemble $\mathcal{F}$ des fonctions $F : K \to \mathbb{R}$ de la forme

$$F(x) = \sum_{i=1}^{N} v_i \phi(\langle w_i, x \rangle + b_i), \quad N \in \mathbb{N}*$$

est dense dans $\mathcal{C}(K, \mathbb{R})$ (ensemble des fonctions continues de K dans $\mathbb{R}$). Autrement dit, pour tout $f : K \mapsto \mathbb{R}$ continue, pour tout $\varepsilon > 0$, il existe $N \in \mathbb{N}*$, il existe des réels v_i, w_i, b_i, $i = 1, \ldots, N$ tels que la fonction F associée satisfasse :

$$\forall x \in K, \quad |F(x) - f(x)| \leq \varepsilon.$$

- Fonctions F : perceptrons à une couche cachée avec $\psi = Id$ et coefficients de biais dans la couche finale nuls.

$$\mathcal{F} = \mathrm{Vect}(\phi(\langle w, x \rangle + b), \quad w \in \mathbb{R}^d, b \in \mathbb{R}).$$

Théorème d'approximation universelle

Remarques

- Résultat non quantitatif.
- Pinkus (1999) montre que ce résultat est vrai si et seulement ϕ n'est pas une fonction polynomiale.
- La condition de Pinkus est clairement nécessaire (si ϕ est de degré r, alors les fonctions de F seront aussi de degré r).
- L'argument principal est le théorème de Stone-Weierstrass qui garantit qu'une classe de fonctions sur un intervalle compact, stable par addition, produit et multiplication par un scalaire (algèbre) est dense dans l'ensemble des fonctions continues dès qu'elle *sépare les points* et qu'elle contient les fonctions constantes.
- Exemple : dans le cas de l'intervalle $[0, 1]$ et de la fonction seuil $\phi(x) = 1_{x \geq 0}$, on peut remarquer que F contient toutes les fonctions étagées, classe de fonctions qui a bien la propriété ci-dessus.

Rétropropagation/Optimisation

- **Question :** Comment trouver les paramètres optimaux ?
- **Réponse (très) approximative** : En faisant une descente de gradient (stochastique).
- Qu'est-ce qu'une descente gradient stochastique ? Quelles sont ses extensions usuelles ?
- Comment calcule-t-on le gradient ? Par *Rétropropagation*.
- On détaille ces questions dans les slides à venir.

De l'optimisation à l'optimisation stochastique

- Comme dans tout problème d'apprentissage supervisé, on a à minimiser une erreur définie formellement comme suit :

$$L(\theta) = \mathbb{E}_{(X,Y)}[\ell(Y, f_\theta(X))]$$

avec

- $f_\theta(x)$ sortie du réseau de neurones et ℓ une fonction de perte (éventuellement pénalisée pour limiter le surapprentissage).

- En quantitatif, on peut prendre classiquement $\ell(y, y') = (y - y')^2$. Dans le cadre classification, on utilise souvent ici l'entropie croisée :

$$\ell(Y, q_\theta(X)) = -\sum_{k=1}^{K} 1_{Y=k} \log(q_\theta^k(x))$$

où $q_\theta^k(x)$ désigne l'estimation de $\mathbb{P}(Y = k|X = x)$ (classiquement produite par la fonction softmax).

Entropie croisée : détails

Pour deux distributions p et q (sur un ensemble discret pour simplifier), l'entropie croisée $H(p,q)$ s'écrit

$$H(p,q) = -\sum_k p(k)\log(q(k)) = H(p) + KL(p||q)$$

où $H(p)$ est l'entropie de p ($H(p) = -\sum_k p(k)\log(p(k)) = \mathbb{E}_{Z\sim p}[-\log p(Z)]$) et $KL(p||q)$ est la divergence de Kullback-Leibler définie par

$$KL(p||q) = \sum_k p(k)\log\left(\frac{p(k)}{q(k)}\right) = \mathbb{E}_{Z\sim p}[\log\left(\frac{p(Z)}{q(Z)}\right)].$$

Si pour un x fixé, on pose $p(k) = \mathbb{P}(Y = k|X = x)$ et $q_\theta(k) = q_\theta^k(x)$, alors

$$\begin{aligned} H(p,q_\theta) &= -\sum_k \mathbb{P}(Y = k|X = x)\log(q_\theta^k(x)) = \mathbb{E}[\ell(Y, q_\theta(X))|X = x] \\ &= KL(p,q_\theta) + C \end{aligned}$$

où C désigne une constante vis-à-vis θ.

Entropie croisée : détails

- Sur le dernier calcul, on voit que si p est fixe et q dépend de θ, alors

$$\text{Minimiser } H(p, q_\theta) \iff \text{Minimiser } KL(p, q_\theta).$$

- Ceci explique l'utilisation de l'entropie croisée comme une version de KL égale à une constante près (si p est fixe).
- Or, KL s'interprète classiquement (et un peu abusivement) comme une distance entre probabilités (ça n'est pas une vraie distance).
- Ainsi, cette minimisation est à comprendre comme une minimisation d'une approximation de la distance entre $\mathbb{P}(Y = k|X)$ et $q_\theta^k(X)$ (intégré ou non en $X \dots$).

Optimisation : rappels

Pour une telle fonction $L : \mathbb{R}^d \to \mathbb{R}$ donnée, notre objectif est donc de trouver $\theta^* = \mathrm{Argmin} L$ (à supposer qu'il soit bien défini).

Remark

En réalité, il y a généralement beaucoup de minima locaux en deep learning.

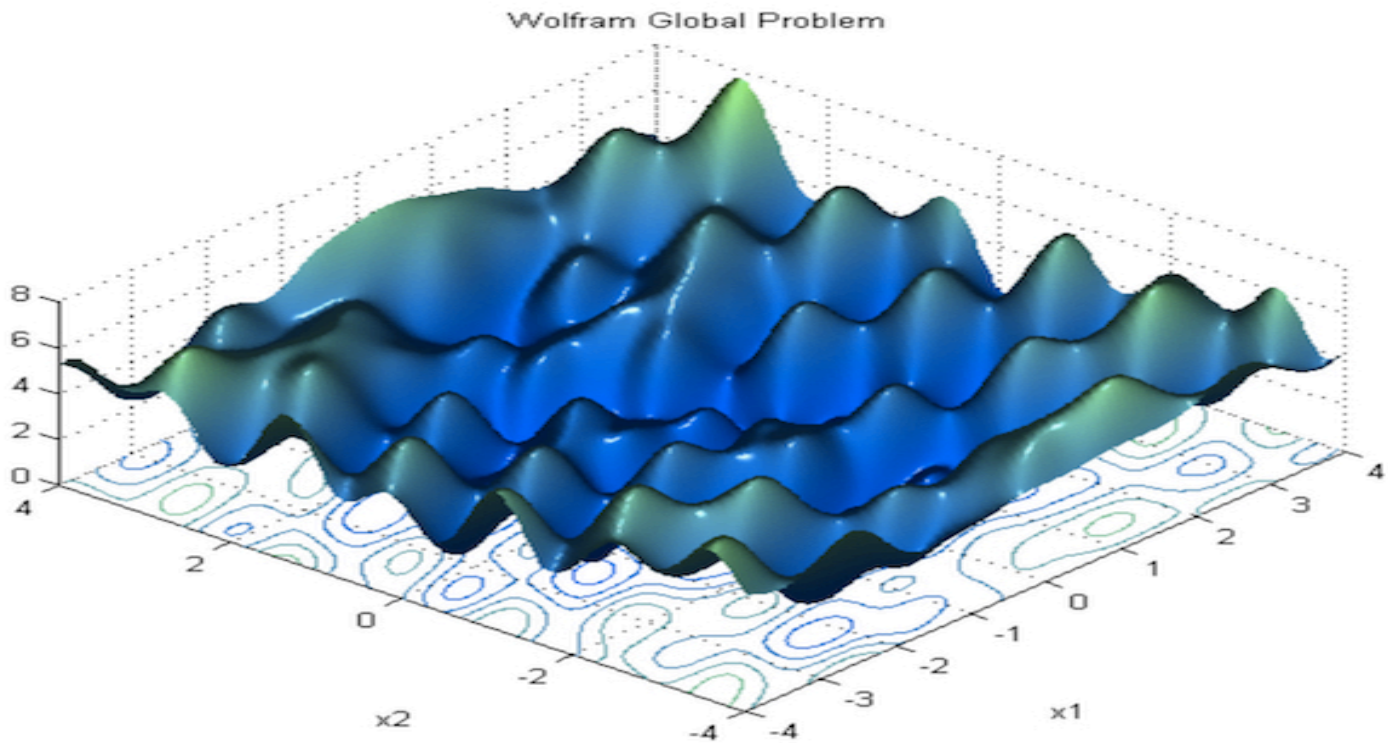


Figure 7: Potentiel non convexe !, Figure issue de `https://`

Optimisation/descente de gradient

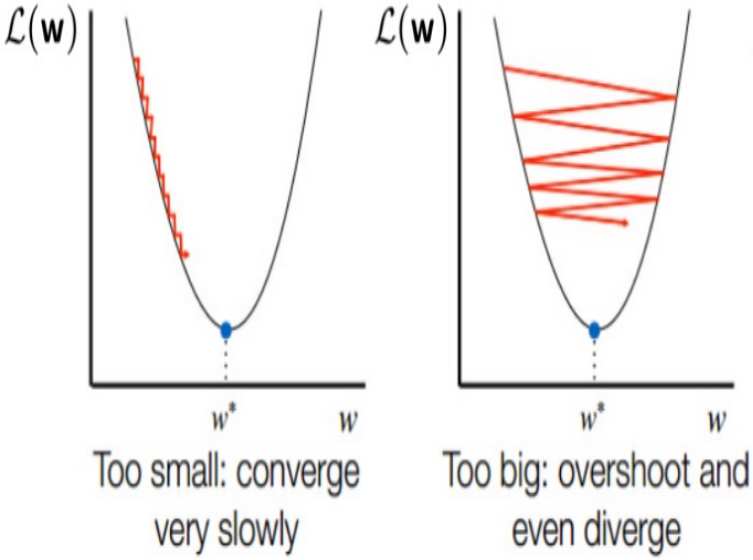
- Le principe de base pour rechercher un minimum local est la descente de gradient :

$$\theta_{n+1} = \theta_n - \gamma_{n+1} \nabla L(\theta_n)$$

.

- Il existe des raffinements qui permettent d'approcher les minima globaux (recuit simulé) mais c'est absolument hors d'atteinte en grande dimension.
- On va donc se contenter de "lancer" des descentes de gradient ou plutôt des descentes de gradient stochastique pour tenter d'approcher un minimum local (ou éventuellement plusieurs si on lance plusieurs trajectoires).

Descente de gradient/Figures



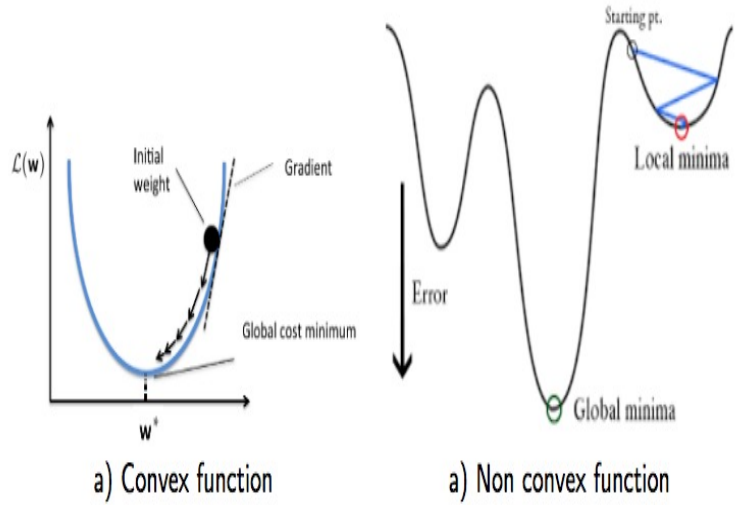


Figure 8: issues de
`http://cedric.cnam.fr/vertigo/Cours/ml2/docs/coursDeep1.pdf`

Descente de gradient stochastique

- Ce type de méthode s'applique pour une fonction L s'écrivant
$$V(\theta) = \mathbb{E}[v(\theta, Z)].$$
- Ici, on peut l'écrire sous cette forme en introduisant I une variable aléatoire de loi uniforme sur $\{1, \dots, N\}$:
$$L(\theta) = \frac{1}{N} \sum_{k=1}^{N} \ell(y_k, f_\theta(x_k)) = \mathbb{E}_I[\ell(y_I, f_\theta(x_I))].$$
- Pour une fonction V, l'idée est de remplacer l'algorithme de descente classique par
$$\theta_{n+1} = \theta_n - \gamma \partial_\theta v(\theta_n, Z_{n+1})$$
où $(Z_n)_{n \geq 0}$ est une suite de v.a. i.i.d.
- Dans le cas qui nous intéresse, cela donne en considérant une suite $i.i.d.$ (I_n)
$$\theta_{n+1} = \theta_n - \gamma \partial_\theta \ell(y_{I_{n+1}}, f_{\theta_n}(x_{I_{n+1}})).$$
Exercice: Ecrire l'algorithme dans le cas où $\ell(Y, f_\theta(X)) = (Y - \langle \theta, X \rangle)^2$.
-

Descente de gradient stochastique (suite)

- Intuition : Comme $\mathbb{E}[\nabla v(\theta_n, Z_{n+1})|\theta_n] = \nabla V(\theta_n)$

$$\theta_{n+1} = \theta_n - \gamma \nabla V(\theta_n, Z_{n+1}) + \gamma \varepsilon_{n+1}$$

  où (ε_n) est une suite de variables aléatoires centrées (bruit centré). Voir TD pour plus de détails.

- En pratique, cet algorithme peut s'avérer instable. On peut lui préférer une version mini-batch (on tire M indices au sort et on fait une moyenne sur ces indices).
- Avantage de la version stochastique : ne demande le calcul que d'un ou M calculs de gradient à chaque étape.

- Pratique sur Tensorflow/Pytorch : versions adaptatives (Adagrad/RMSprop/Adam, voir TD).

- On parle d'EPOCH lorsque l'algorithme est passé sur "tout" l'échantillon.

Calcul du gradient : "Rétropropagation"

- Difficulté numérique majeure : Calcul du gradient, possible mais très coûteux.
- Principe : Rétropropagation.
- Idée : Le réseau de neurones étant basé sur une composition de fonctions, on peut utiliser la "chain rule", *i.e.* la formule de dérivation des fonctions composées.
- Cela donne un algorithme "backward" de calcul d'où le nom "backpropagation".

Rétropropagation : un exemple

- Considèrons un réseau de neurones à 1 couche cachée :

$$f_\theta(x) = \psi\left(\beta + \sum_{i=1}^{N} v_i \phi(\langle w_i, x\rangle + b_i)\right).$$

- Dérivation pas à pas : on commence par les paramètres les plus "récents", v_i et b_i : notons

$$z = a^{(2)}(x) = \beta + \sum_{i=1}^{N} v_i \phi(\langle w_i, x\rangle + b_i).$$

On a

$$\partial_\beta f_\theta(x) = \frac{\partial \psi}{\partial z}(a_2(x))\partial_\beta(a^{(2)}(x)) = \psi'(a^{(2)}(x)).$$

De même,

$$\partial_{v_i} f_\theta(x) = \frac{\partial \psi}{\partial z}(a_2(x))\partial_{v_i}(a^{(2)}(x)) = \psi'(a^{(2)}(x))\phi(\langle w_i, x\rangle + b_i).$$

Rétropropagation : un exemple

- puis dérivées par rapport aux paramètres plus anciens : notons $a_i^{(1)}(x) = \langle w_i, x \rangle + b_i$. et commençons par les b_i :

$$\partial_{b_i} f_\theta(x) = \frac{\partial \psi}{\partial z} \times \frac{\partial z}{\partial a_i^{(1)}} \times \partial_{b_i}(a_i^{(1)}(x)) = \frac{\partial \psi}{\partial z} v_i \phi'(a_i^{(1)}(x)) 1.$$

Regardons enfin les dérivées par rapport à w_i^j, *i.e.* le poids associé à la connexion entre l'entrée x_j et le neurone i. On a

$$\partial_{w_i^j} f_\theta(x) = \frac{\partial \psi}{\partial z} \times \frac{\partial z}{\partial a_i^{(1)}} \times \partial_{w_i^j}(a_i^{(1)}(x)) = \psi'(a^{(2)}(x)) v_i \phi'(a_i^{(1)}(x)) x_j.$$

Commentaires

- Pour passer de l'étape 1 à l'étape 2, on a eu recours à $\frac{\partial f_\theta}{\partial a^{(2)}}$ qui a déjà été calculé.
- Formellement (formules matricielles),

$$\frac{\partial f_\theta}{\partial w_i} = \frac{\partial f_\theta}{\partial a^{(2)}} \times \frac{\partial a^{(2)}}{\partial a^{(1)}} \times \frac{\partial a^{(1)}}{\partial w_i}.$$

- En itérant cette relation, on pourra récursivement calculer toutes les dérivées partielles pour finalement pouvoir itérer la descente de gradient (stochastique).

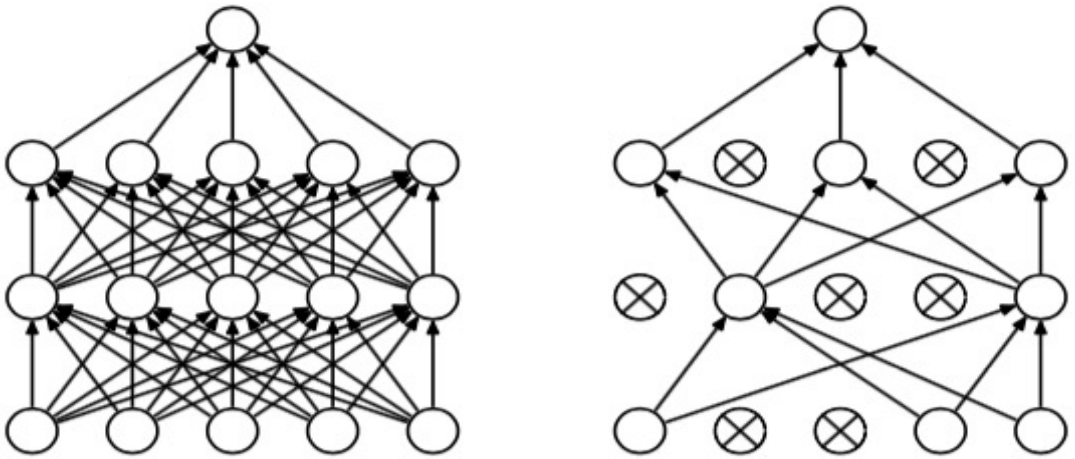
Dropout

Pour éviter les problèmes de surapprentissage dû à la grande flexibilité,

- Augmentation du nombre données via simulation ou transformations déterministes (translations, rotations, convolution avec un bruit dans le cas des images par exemple).

- Régularisation par Dropout : Technique de régularisation qui consiste à annuler la réponse d'un neurone avec une certaine probabilité p (typiquement, 1 chance sur 2). Les neurones ainsi ignorés ne contribuent pas au modèle. A chaque étape de l'apprentissage, c'est donc un modèle légèrement différent qui est utilisé. Lors de la phase de test, une solution consiste à utiliser tous les neurones mais à réduire les poids d'un facteur p ce qui peut s'interpréter comme une approximation de l'effet moyen d'un neurone activé avec probabilité $1 - p$ (penser à l'application pour la fonction ReLU). On peut aussi conserver le dropout dans la phase de test quitte à calculer plusieurs évaluations d'un même individu (voir les paramètres par défaut des fonctions implémentées pour plus de détails).

Dropout



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Figure 9: Exemple de dropout issu de `JMLRdropout.pdf`

Autres raffinements

- Raffinements ou tentatives d'amélioration numérique nombreuses.
- Exemple 1 : *Dropconnect* qui consiste à annihiler des connexions aléatoirement (mettre les poids à 0 avec une certaine probabilité)
- Exemple 2 : Pénalisation de type Lasso/Ridge afin de "shrinker" ou "seuiller" les paramètres à estimer pour limiter le surapprentissage
- Voir `https://www.deeplearningbook.org/contents/regularization.html` pour des détails précis sur les techniques de régularisation.

Réseaux de neurones à convolution (CNN)

- Réseaux à architecture spécifique adaptée au traitement d'image.
- Idée : appliquer en amont d'un perceptron un certain nombre de filtres afin de dégager des caractéristiques de l'image qui faciliteront sa reconnaissance.
- Prend en entrée une ou plusieurs matrices $p \times p$ (selon que l'image est N/B ou RGB).

Architecture d'un CNN

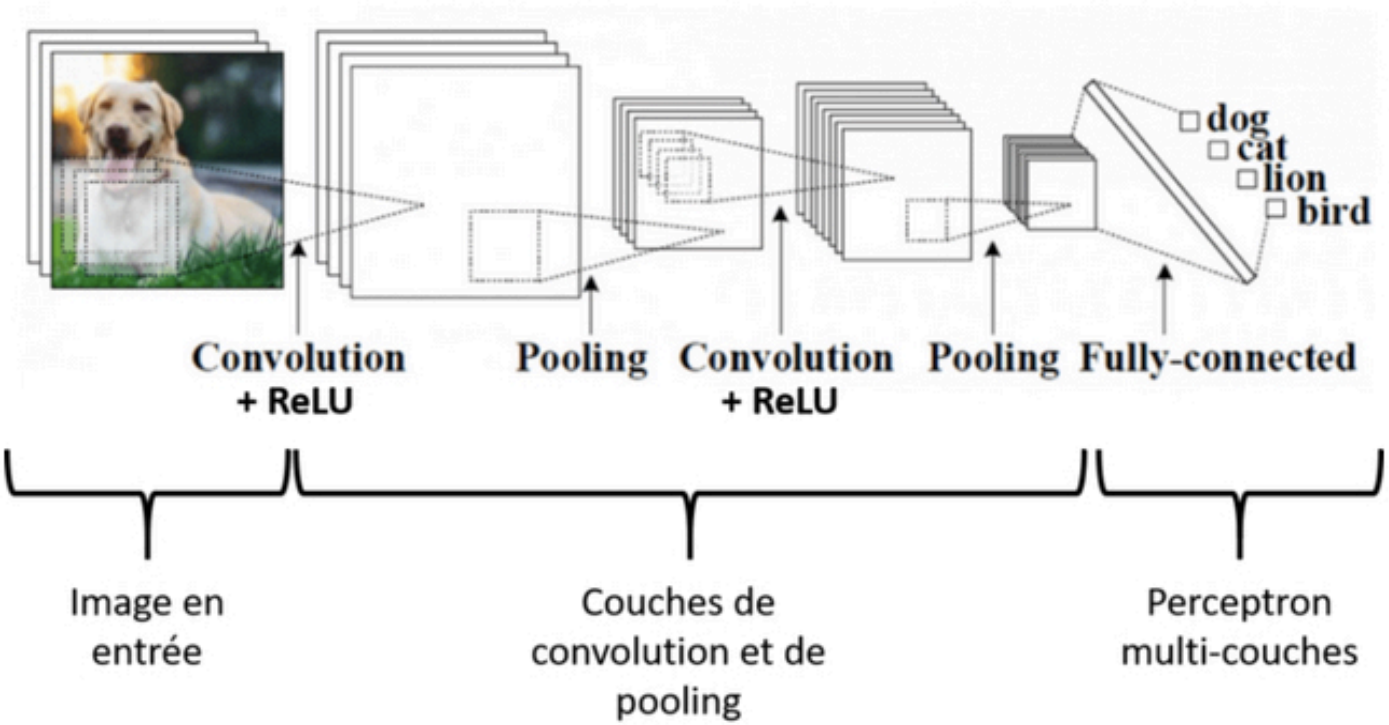


Figure 10: Architecture d'un CNN

Convolution

- Convolution : étape de filtrage de l'image basée sur une convolution au sens mathématique du terme, *i.e.* une moyennisation de la forme suivante (dans le cas discret)

$$f * g(x) = \sum_t f(t)g(x+t).$$

(Convolution de g par f).

- **Exemple :** sur l'espace des fonctions de $\mathbb{Z}$ dans $\mathbb{Z}$ avec $f(z) = \frac{1}{3}1_{\{z\in -1,0,1\}}$. Dans ce cas,

$$f * g(x) = \frac{1}{3}(g(x-1) + g(x) + g(x+1)).$$

f est appelée noyau de convolution qui s'applique une fonction g quelconque. On voit que sur l'exemple ci-dessus, la convolution a pour effet de faire une moyenne des valeurs de la fonction g au voisinage de x.

Convolution

La convolution va avoir pour rôle de transformer les petites zones de l'image. Formellement, si l'on considère une image I comme un élément de $\mathbb{Z}^2$ (quitte à mettre des zéros en dehors du carré $\{0, \ldots, p\} \times \{0, \ldots, p\}$,) alors :

Definition

Si K désigne une application de $\mathbb{Z}^2$ dans $\mathbb{R}$, la convolution de l'image I par (le noyau/filtre) K est définie par : pour tous i, j,

$$K * I(i,j) = \sum_{n,m} K(n,m) I(i+n, j+m).$$

En pratique, la fonction K est généralement non nulle sur quelques pixels seulement. En d'autres termes, les convolutions sont juste un ensemble fini de poids.

Convolution

Voici un exemple de convolution par noyau :

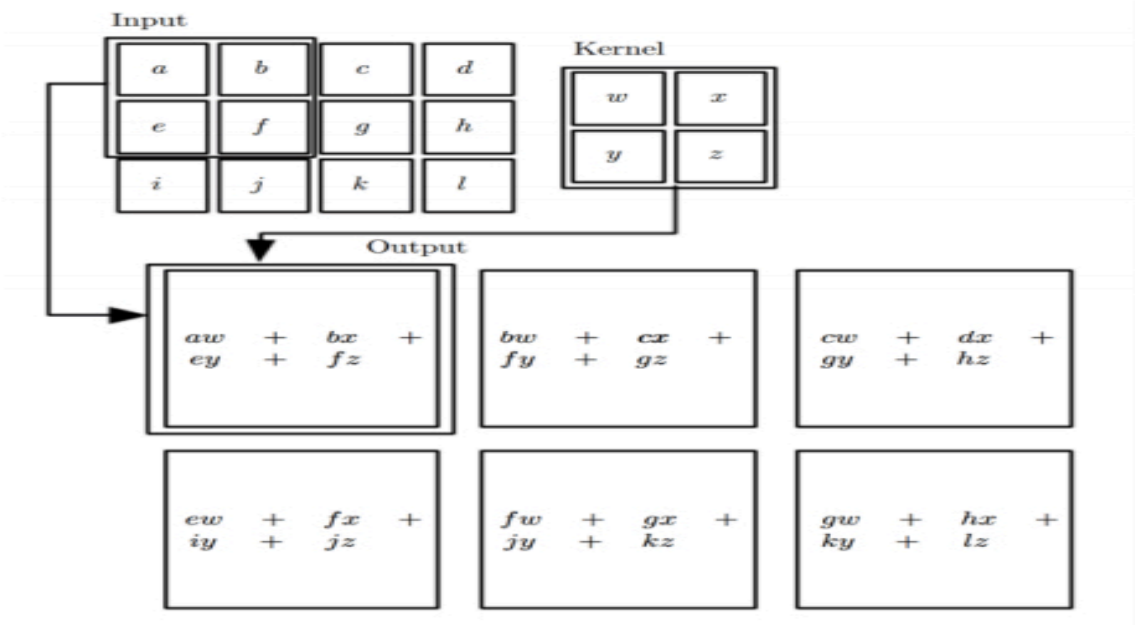


Figure 8: 2D convolution. Source :https://i2.wp.com/khshim.files.wordpress.com/2016/10/2d-convolution-example.png

Figure 11: Exemple de transformation

Chaque manière de convoler l'image a vocation a avoir un effet particulier. Certaines convolutions augmentent le contraste, d'autres les contours, d'autres le flou. . . .

Différents types convolution

Voici ci-dessous quelques exemples issus de la page
https://docs.gimp.org/2.6/fr/plug-in-convmatrix.html :

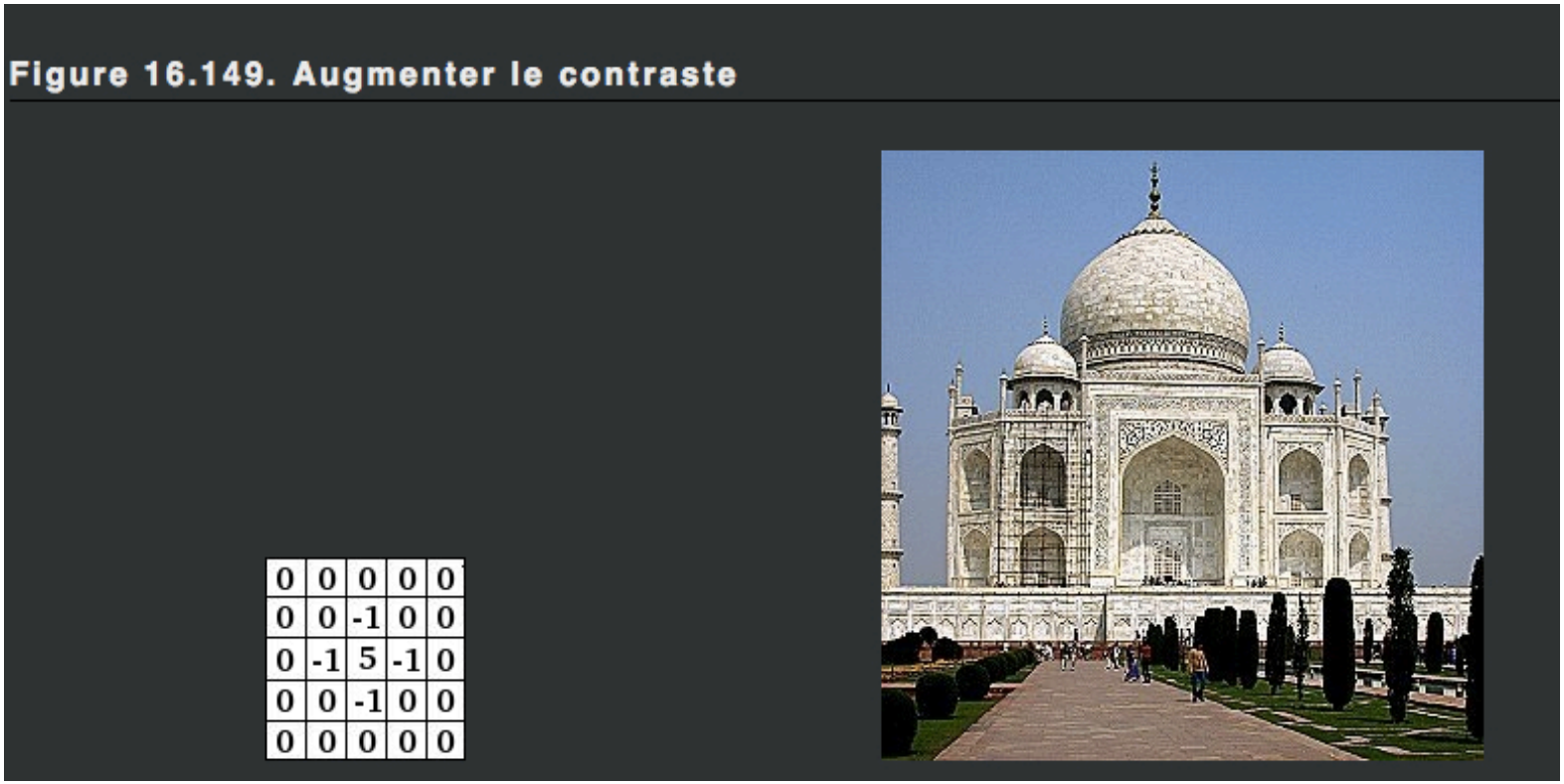


Figure 12: Augmentation des contrastes

Différents types convolution

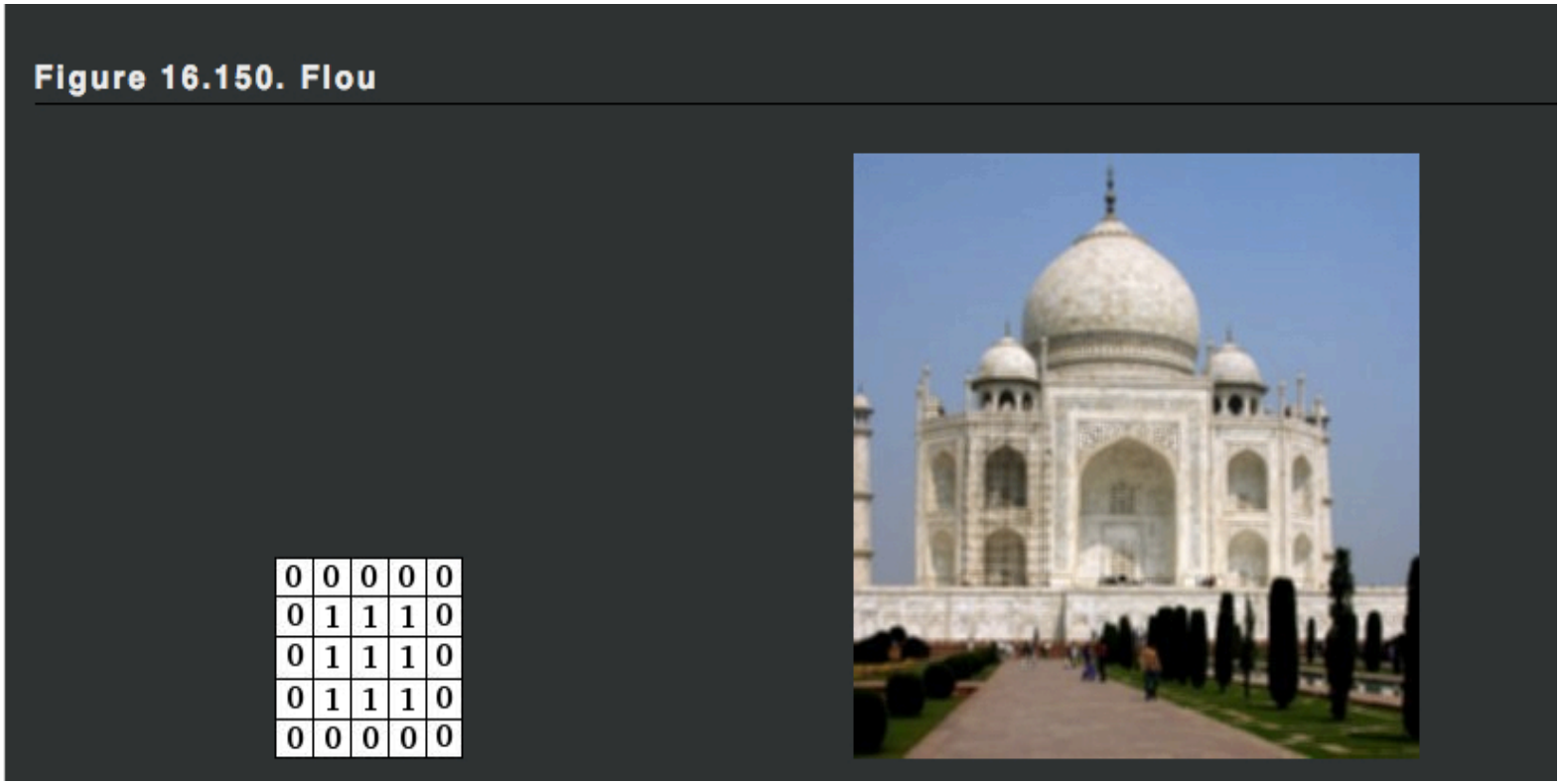


Figure 13: Convolution et flou

Différents types convolution

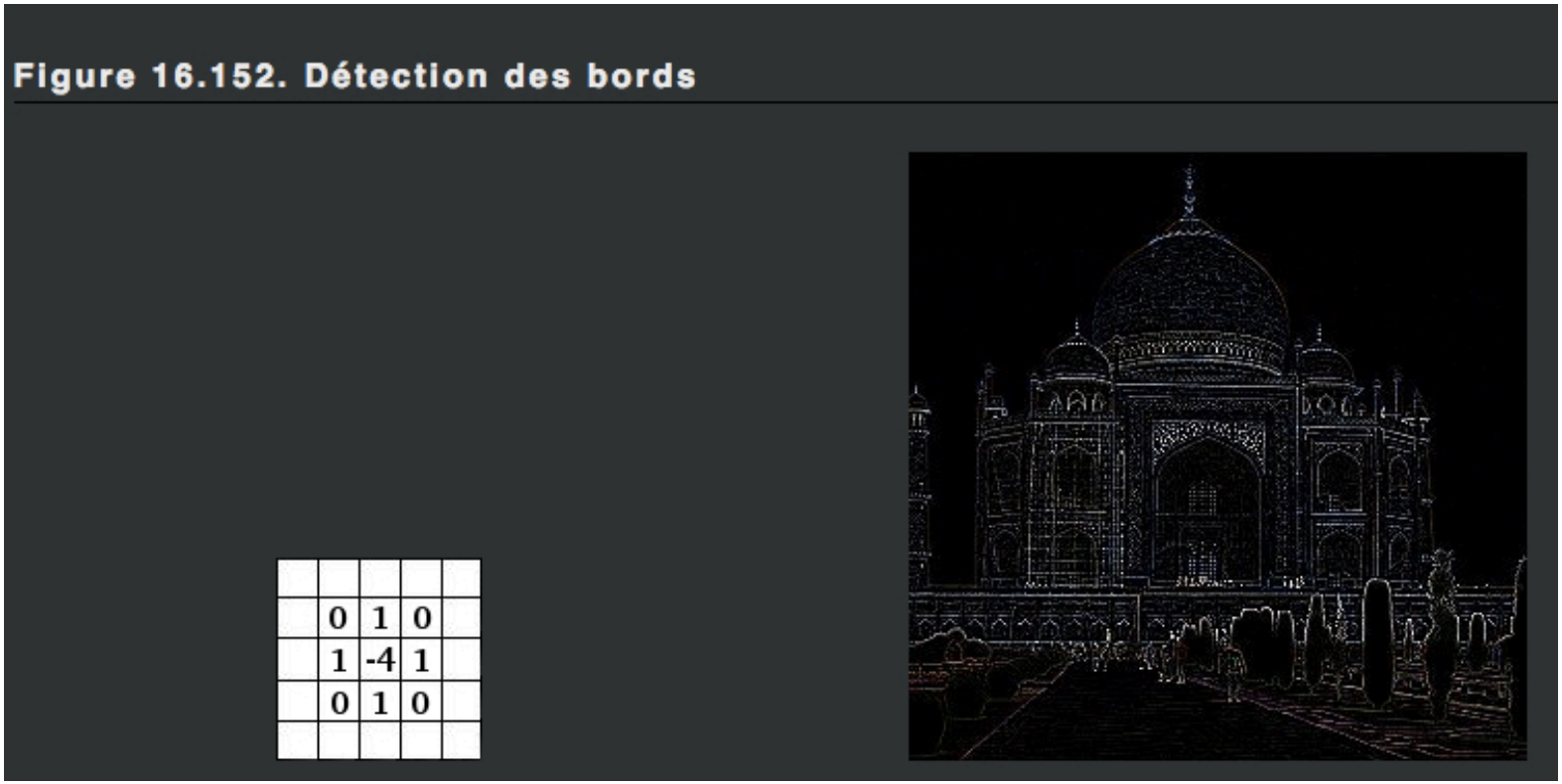


Figure 14: Convolution et bordures

Pooling

Le pooling constitue le deuxième type de filtrage. Il a vocation à réduire la dimension en remplaçant un ensemble de pixels par un seul pixel. On a ainsi :

- Le *max-pooling* qui sélectionne la valeur maximale sur un ensemble de pixels. En voici un exemple ci-dessous (Figure 15) :

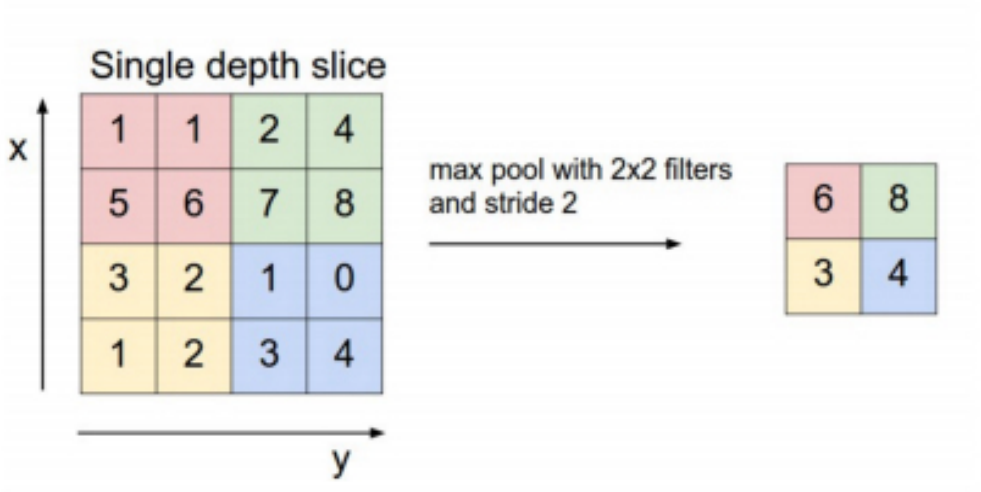


Figure 11: Maxpooling and effect on the dimension - Source : http://www.wildml.com/wp-content/uploads/2015/11/Screen-Shot-2015-11-05-at-2.18.38-PM.png

Figure 15: Max-pooling

Pooling

- Le *mean-pooling* : remplace un ensemble de pixels par leur moyenne.
- Les étapes de pooling sont aussi appelées “subsampling”.

Fully Connected Network

A l'issue de cette suite de filtrages, on reconstitue un objet contenant les images ainsi construites à la dernière couche de convolution/pooling puis on lui applique un perceptron multi-couches. L'exemple de Alexnet est représenté dans la figure 16 https://www.learnopencv.com/understanding-alexnet/

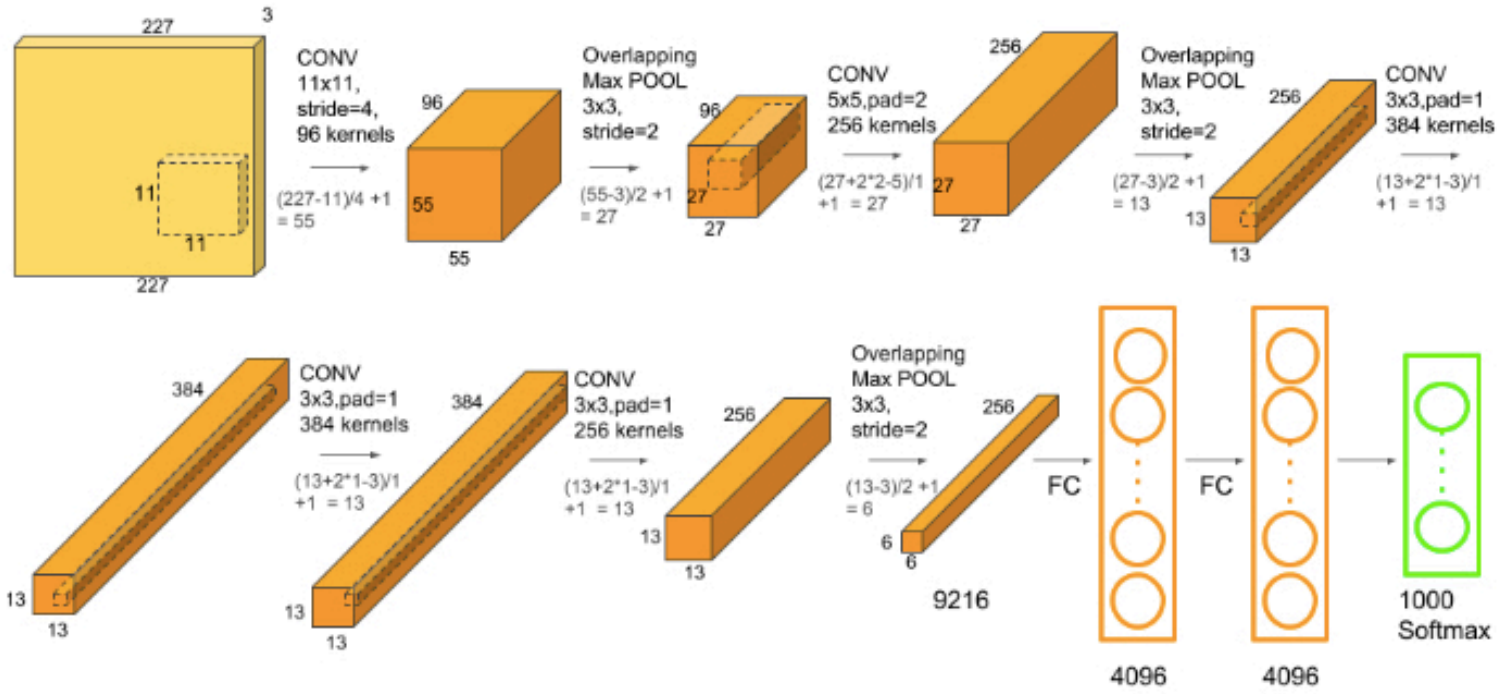


Figure 16: AlexNet Neural Network. Figure issue de https://www.learnopencv.com/understanding-alexnet/

Exemple de code Python

```
model_conv = km.Sequential()
model_conv.add(kl.Conv2D(32, (3, 3), input_shape=(img_width, img_height, 3), data_format="channels_last"))
model_conv.add(kl.Activation('relu'))
model_conv.add(kl.MaxPooling2D(pool_size=(2, 2)))

model_conv.add(kl.Conv2D(32, (3, 3)))
model_conv.add(kl.Activation('relu'))
model_conv.add(kl.MaxPooling2D(pool_size=(2, 2)))

model_conv.add(kl.Conv2D(64, (3, 3)))
model_conv.add(kl.Activation('relu'))
model_conv.add(kl.MaxPooling2D(pool_size=(2, 2)))

model_conv.add(kl.Flatten())  # this converts our 3D feature maps to 1D feature vectors
model_conv.add(kl.Dense(64))
model_conv.add(kl.Activation('relu'))
model_conv.add(kl.Dropout(0.5))
model_conv.add(kl.Dense(1))
model_conv.add(kl.Activation('sigmoid'))
```

Réseaux de neurones récurrents

- Les réseaux de neurones récurrents sont conçus pour prendre en charge des séries temporelles.
- Les exemples d'application les plus "parlants" sont l'analyse, la génération ou la traduction de textes, ou encore la reconnaissance vocale.
- Leur application dans d'autres domaines peut néanmoins être envisagée (on trouvera par exemple plusieurs références sur leur utilisation en prévision de la consommation d'électricité).
- Voici quelques slides d'introduction sur le sujet.

RNN pour différents types de problèmes

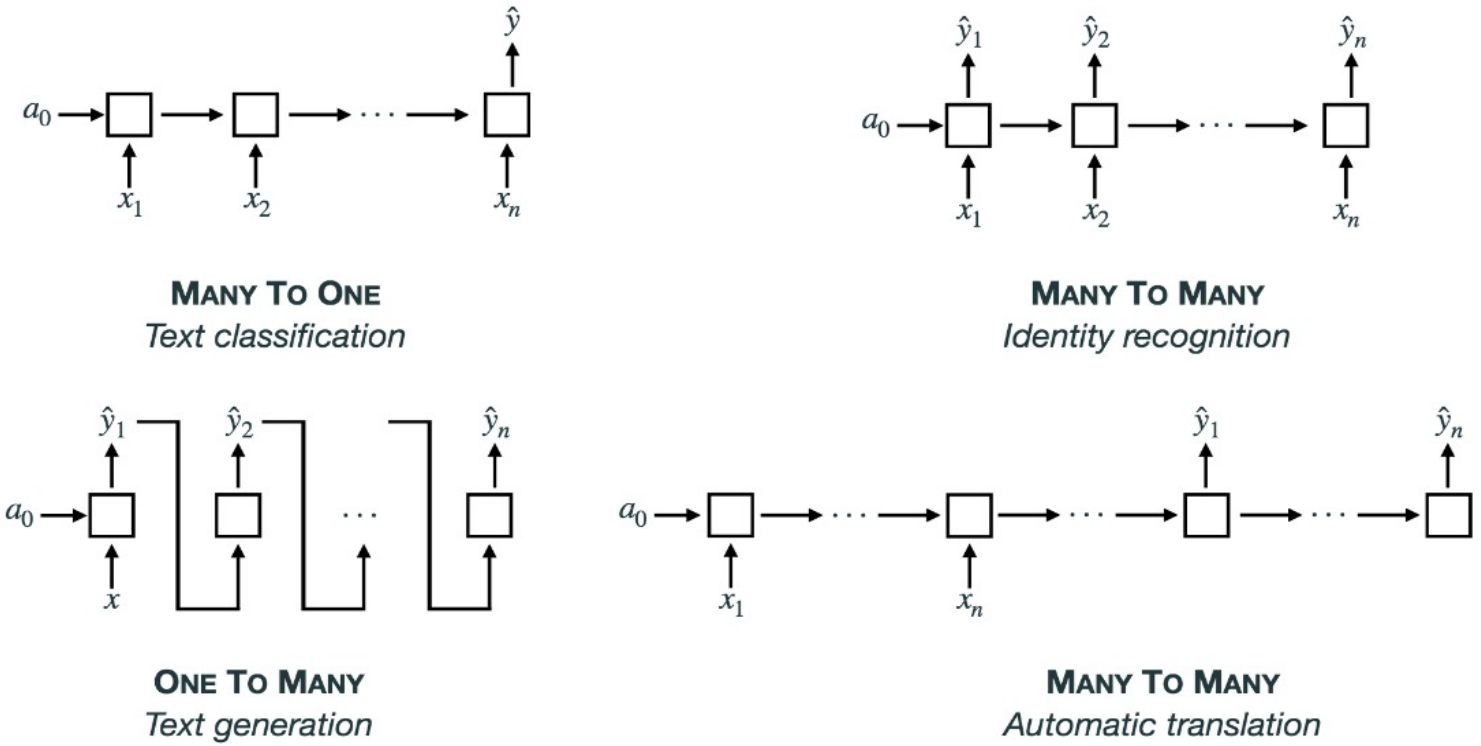
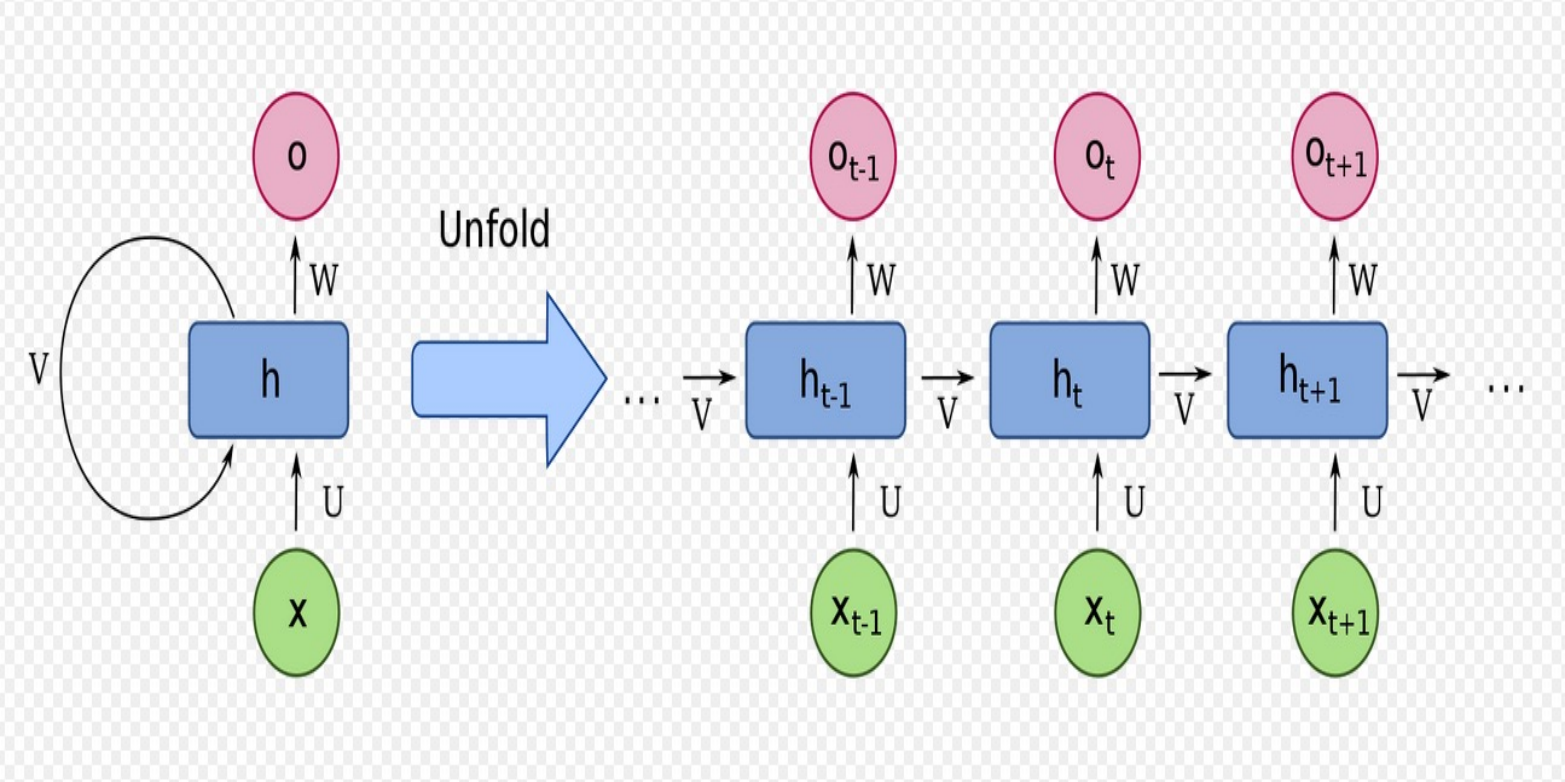


Figure 17: Image issue de `https://github.com/wikistat/`

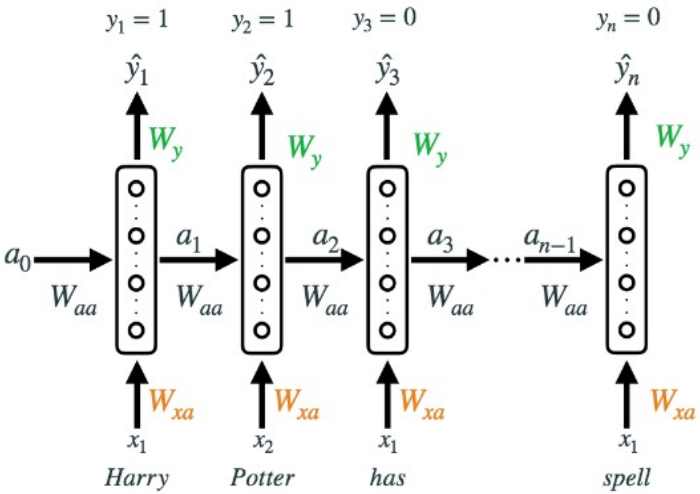
Point commun : les “flèches horizontales” qui traduisent la dépendance/mémoire.

Quelle architecture envisager ?



Idée : fabriquer un perceptron à chaque instant t et autoriser une dépendance en le “hidden layer” précédent.

En plus précis



Activation Function

$$a_t = g_a(a_{t-1}W_{aa} + x_t W_{xa} + ba)$$
$$= g_a([a_{t-1}, x_t] \cdot [W_{aa} + W_{xa}]^T + ba)$$
$$= g_a([a_{t-1}, x_t]W_y + b_y)$$

$$\hat{y} = g_y(a_t \cdot W_{xa} + by)$$

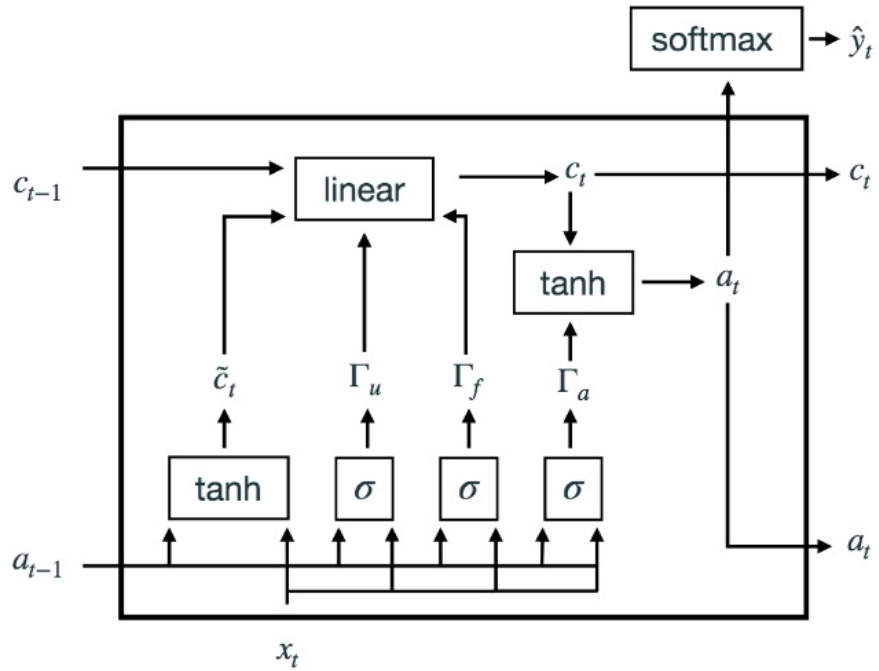
- $x_t, (1 \times V)$: OHE representation.
- V : Dictionary's size.
- $y_t(1 \times 1)$: 0 or 1.
- $a_t : 1 \times H$.
- H : Number of neutrons.
- N : Number of timestep (Not layer!).
- g_a : Activation Function (**tanh/Relu**).
- g_y Activation Function (**sigmoid**).
- $W_{aa}, (H \times H, \forall t \in [1,...,n])$
- $W_{xa}, (V \times H, \forall t \in [1,...,n])$
- $W_y, (H \times 1, \forall t \in [1,...,n])$
- $W_a, (H + V \times H, \forall t \in [1,...,n])$

9

Figure 18: Image issue de `https://github.com/wikistat/`

LSTM

La structure naïve rend le "gradient trop plat" et par conséquent rend difficile voire impossible l'optimisation. Pour pallier ce problème, les LSTM (Long-Short-Time-Memory) proposent une structure locale permettant de filtrer la mémoire afin de limiter l'interdépendance entre les couches...



- c_t : memory cell.
- $c_t \neq a_t$.
- $\tilde{c} = tanh([a_{t-1}, x_t]W_a + b_a)$.
- $\Gamma_u = \sigma([a_{t-1}, x_t]W_u + b_u)$.
- $\Gamma_f = \sigma([a_{t-1}, x_t]W_u + b_f)$.
- $c_t = \Gamma_u * \tilde{c}_t + \Gamma_f * c_{t-1}$.
- $\Gamma_o = \sigma([a_{t-1}, x_t]W_r + b_o)$.
- $a_t = \Gamma_o * tanh(c_t)$.
- $\hat{y}_t = softmax(a_t W_y + b_y)$.

Figure 19: Image issue de https://github.com/wikistat/