Cas de la fonction booléenne “XOR”

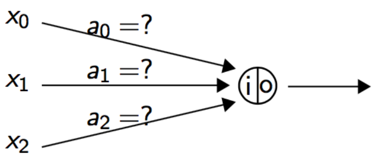
Signal post-synaptique et fonction d’Heaviside :

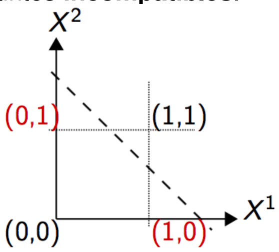
X^1	X^2	XOR
1	1	0
1	0	1
0	1	1
0	0	0

$$w_0 + w_1 X^1 + w_2 X^2 \lesseqgtr 0$$

$$\begin{cases} w_0 + w_1 + w_2 \leq 0 \\ w_0 + w_1 > 0 \\ w_0 + w_2 > 0 \\ w_0 \leq 0 \end{cases}$$

Ensemble de contraintes **incompatibles**.



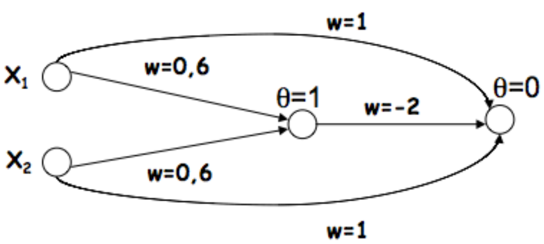


Et pourtant, une solution existe...

On peut apprendre le XOR avec un réseau à une couche cachée :

Et pourtant, une solution existe...

On peut apprendre le XOR avec un réseau à une couche cachée :



Plan

Introduction générale

Le perceptron simple

Le perceptron multicouches

Le perceptron multicouches

- Les modèles précédents définissent des modèles linéaires avec certaines limites.

Le perceptron multicouches

- Les modèles précédents définissent des modèles linéaires avec certaines limites.
- Le perceptron multicouche (“multilayer perceptron”) est une généralisation de ces modèles :

Le perceptron multicouches

- Les modèles précédents définissent des modèles linéaires avec certaines limites.
- Le perceptron multicouche (“multilayer perceptron”) est une généralisation de ces modèles :
  - en régression il permet de traiter les cas **non linéaires** de régression

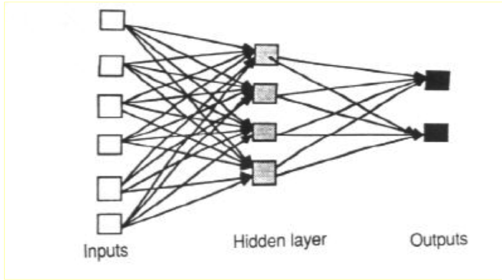
Le perceptron multicouches

- Les modèles précédents définissent des modèles linéaires avec certaines limites.
- Le perceptron multicouche (“multilayer perceptron”) est une généralisation de ces modèles :
  - en régression il permet de traiter les cas **non linéaires** de régression
  - en classification, il permet de déterminer des fonctions de décision textbfnon linéaire permettant de résoudre le problème “XOR” précédent par exemple

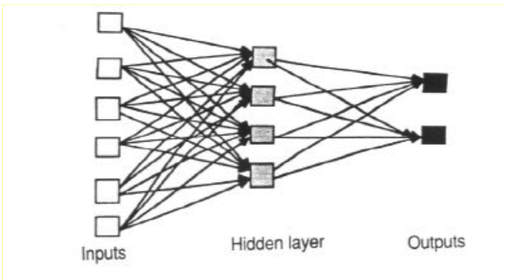
Le perceptron multicouches

- Les modèles précédents définissent des modèles linéaires avec certaines limites.
- Le perceptron multicouche (“multilayer perceptron”) est une généralisation de ces modèles :
  - en régression il permet de traiter les cas **non linéaires** de régression
  - en classification, il permet de déterminer des fonctions de décision textbfnon linéaire permettant de résoudre le problème “XOR” précédent par exemple
- Il consiste en l’ajout de **couches de neurones dites cachées** entre les données en entrée et les données en sortie.

Le perceptron multicouches

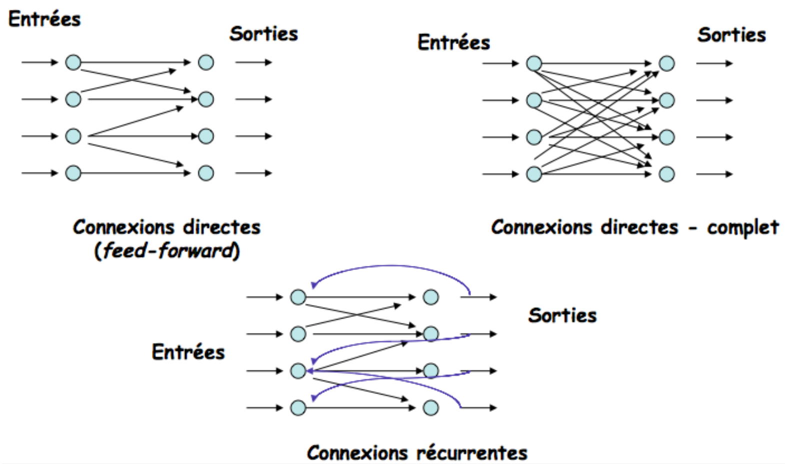


Le perceptron multicouches



Petite démo avec TensorFlow :
`http://playground.tensorflow.org`

Architecture des réseaux de neurones



Perceptron - Apprentissage des poids

Loi de Hebb

Lorsque deux neurones sont excités conjointement, il se crée ou renforce un lien les unissant : $w_i' = w_i + \eta(\hat{y}.x_i)$

Perceptron - Apprentissage des poids

Loi de Hebb

Lorsque deux neurones sont excités conjointement, il se crée ou renforce un lien les unissant : $w_i' = w_i + \eta(\hat{y}.x_i)$

Cette loi a inspiré le perceptron de Rosenblatt qui l'adapte pour prendre en compte l'erreur observée :

$$w_i' = w_i + \eta(y - \hat{y})x_i$$

Perceptron - Apprentissage des poids

Loi de Hebb

Lorsque deux neurones sont excités conjointement, il se crée ou renforce un lien les unissant : $w_i' = w_i + \eta(\hat{y}.x_i)$

Cette loi a inspiré le perceptron de Rosenblatt qui l'adapte pour prendre en compte l'erreur observée :

$$w_i' = w_i + \eta(y - \hat{y})x_i$$

Dans le perceptron de Rosenblatt, la règle passe par la fonction d'activation : $\hat{y} = h(\sum_{i=0}^{n} w_i x_i)$
Avec Adaline (1960), la règle de Widrow-Hoff utilise directement la somme pondérée des entrées : $\hat{y} = \sum_{i=0}^{n} w_i x_i$

Schéma général de l'algorithme pour un neurone

Pour chaque observation :

Schéma général de l'algorithme pour un neurone

Pour chaque observation :

- Phase de **propagation**
  - calculer l'activation du neurone
  - calculer la sortie de la fonction choisie

Schéma général de l'algorithme pour un neurone

Pour chaque observation :

- Phase de **propagation**
  - calculer l'activation du neurone
  - calculer la sortie de la fonction choisie
- Calcul de l'**erreur**

Schéma général de l'algorithme pour un neurone

Pour chaque observation :

- Phase de **propagation**
  - calculer l'activation du neurone
  - calculer la sortie de la fonction choisie
- Calcul de l'**erreur**
- **Mise à jour** des poids pour réduire l'erreur attendue

Schéma général de l'algorithme pour un neurone

Pour chaque observation :

- Phase de **propagation**
  - calculer l'activation du neurone
  - calculer la sortie de la fonction choisie
- Calcul de l'**erreur**
- **Mise à jour** des poids pour réduire l'erreur attendue
  Dans le cas d'une fonction simple à seuil (*heaviside*), les règles sont simples :
  - $w_{t+1} = w_t + x$ si x est positif et $w_t.x \leq 0$
  - $w_{t+1} = w_t - x$ si x est négatif et $w_t.x > 0$

Illustration de l'apprentissage

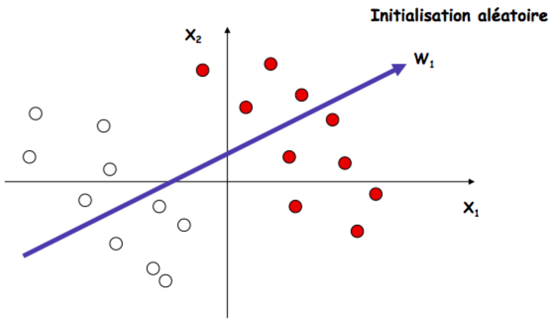


Illustration de l'apprentissage

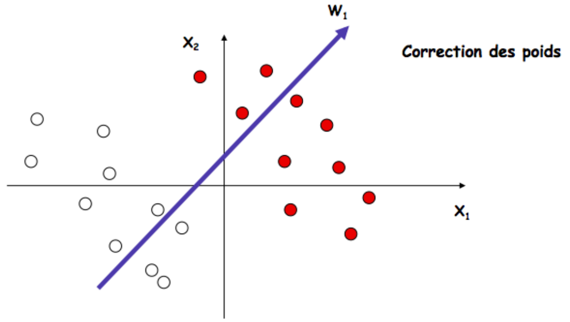


Illustration de l'apprentissage

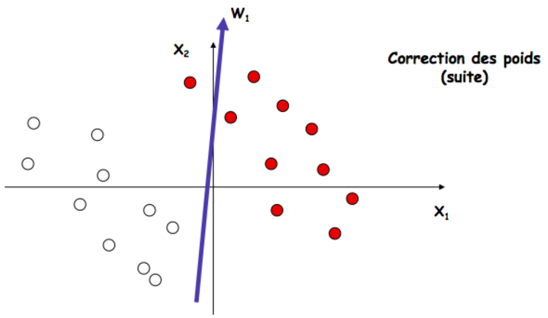
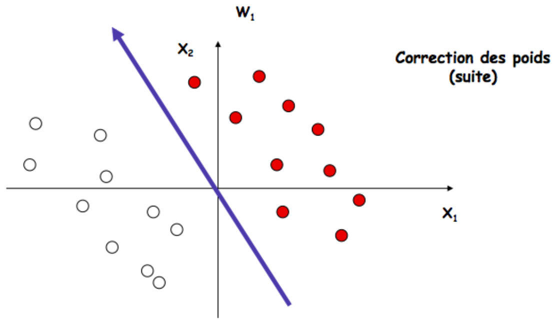


Illustration de l'apprentissage



Interprétation géométrique

Le problème revient revient à trouver un vecteur $\vec{w}$ dans l'espace des objets tel que $\vec{w}^T.\vec{z} \geq 0$, avec $\vec{z} = \vec{x}$ si $\vec{x}$ est de la classe positive et $\vec{z} = -\vec{x}$ sinon.

Interprétation géométrique

Le problème revient revient à trouver un vecteur $\vec{w}$ dans l'espace des objets tel que $\vec{w}^T.\vec{z} \geq 0$, avec $\vec{z} = \vec{x}$ si $\vec{x}$ est de la classe positive et $\vec{z} = -\vec{x}$ sinon.

Donc, la loi d'apprentissage de Widrow-Hoff revient simplement à faire pivoter le vecteur $\vec{w}$ de manière que les projections des $\vec{z}$ soient positives.

Interprétation géométrique

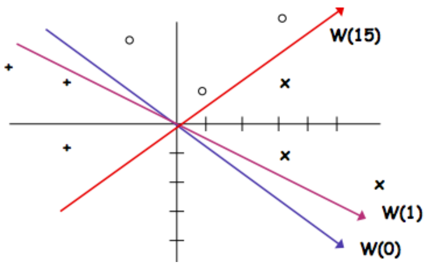
Le problème revient revient à trouver un vecteur $\vec{w}$ dans l'espace des objets tel que $\vec{w}^T.\vec{z} \geq 0$, avec $\vec{z} = \vec{x}$ si $\vec{x}$ est de la classe positive et $\vec{z} = -\vec{x}$ sinon.

Donc, la loi d'apprentissage de Widrow-Hoff revient simplement à faire pivoter le vecteur $\vec{w}$ de manière que les projections des $\vec{z}$ soient positives.

Rappel : la projection d'un vecteur $\vec{u}$ sur un vecteur $\vec{v}$ s'obtient par $proj_{\vec{v}}\vec{u} = \frac{\vec{u}^T.\vec{v}}{||\vec{v}||}\vec{v}$ Donc chaque fois qu'un $\vec{z}$ ne se projette pas sur la partie positive de $\vec{w}$, on fait pivoter $\vec{w}$ vers $\vec{z}$ de manière proportionnelle à l'écart entre $\vec{w}$ et $\vec{z}$:

$$w_{t+1} = w_t + \eta(y - \hat{y})x = w_t + \Delta w$$

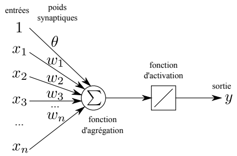
Illustration de l'apprentissage



Les O sont les objets positifs, les X les objets négatifs et les + la transformation de ces derniers.

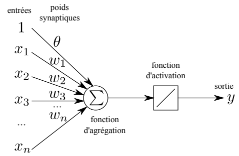
Perceptron - Apprentissage des poids

Sortie du neurone: $\hat{y} = \sum_{i=0}^{n} w_i x_i$

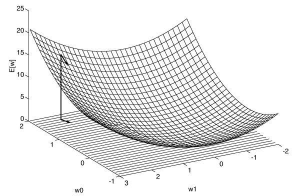


Perceptron - Apprentissage des poids

Sortie du neurone: $\hat{y} = \sum_{i=0}^{n} w_i x_i$

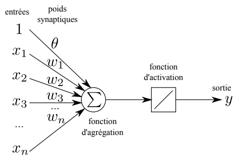


Apprentissage: On veut minimiser l'erreur quadratique $E = \frac{1}{2}\sum_{\mathbf{x}}(y-\hat{y})^2$

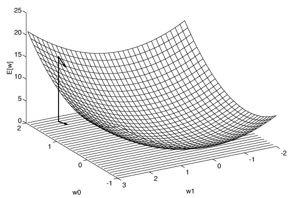


Perceptron - Apprentissage des poids

Sortie du neurone: $\hat{y} = \sum_{i=0}^{n} w_i x_i$



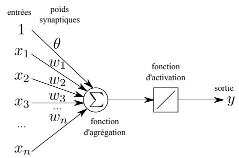
Apprentissage: On veut minimiser l'erreur quadratique $E = \frac{1}{2}\sum_{\mathbf{x}}(y-\hat{y})^2$



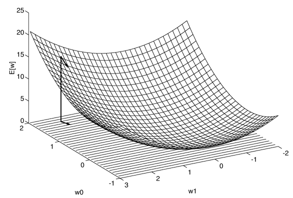
$$
\begin{aligned}
\Delta w_i &= -\eta \frac{\partial E}{\partial w_i} \\
&= -\frac{\eta}{2}\sum_{\mathbf{x}} \frac{\partial (y-\hat{y})^2}{\partial w_i} \\
&= -\frac{\eta}{2}\sum_{\mathbf{x}} -2\frac{\partial \hat{y}}{\partial w_i}(y-\hat{y}) = \sum_{\mathbf{x}} \eta x_i (y-\hat{y})
\end{aligned}
$$

Perceptron - Apprentissage des poids

Sortie du neurone: $\hat{y} = \sum_{i=0}^{n} w_i x_i$



Apprentissage: On veut minimiser l'erreur quadratique $E = \frac{1}{2}\sum_{\mathbf{x}}(y-\hat{y})^2$ par descente de gradient stochastique:
$\Delta \mathbf{w} = \eta \mathbf{x}(y-\hat{y})$



Optimisation de E par descente du gradient stochastique :
$\Delta \mathbf{w} = \eta \mathbf{x}(y - \hat{y})$

Propriétés

- Classifieur linéaire
- Convergence garantie si η est faible (même si les données ne sont pas linéairement séparables)
- Convergence efficace (car la fonction à optimiser est quadratique)
- Convergence souvent plus rapide en mode en ligne mais pas garantie comme en mode *batch*

Pseudo-code de l'algorithme

Require: E un ensemble de données : $E = \{(x_1, y_1), (x_2, y_2) \ldots (x_n, y_n)\}$
  W les paramètres du réseau de neurones initialisés
  h la fonction d'activation
  η le pas d'apprentissage
  repeat
    for all $e_i = (x_i, y_i)$ dans E **do**
      for all j dans $\{1, 2 \ldots k\}$ **do**
        $in = w_j.x_i$
        $err = y_i - h(in)$
        $w_j = w_j + \eta.err.h'(in).x_i$
      end for
    end for
  until un certain critère de convergence

Cross-entropy vs. Mean Squared Error

$$H(p, q) = -\sum_x p(x) \log q(x)$$

Modèle num. 1 :

	$\hat{y}$			y			correct ?
e_1	0.3	0.3	0.4	0	0	1	oui
e_2	0.3	0.4	0.3	0	1	0	oui
e_3	0.1	0.2	0.7	1	0	0	non

Cross-entropy vs. Mean Squared Error

$$H(p, q) = -\sum_x p(x) \log q(x)$$

Modèle num. 1 :

	$\hat{y}$			y			correct ?
e_1	0.3	0.3	0.4	0	0	1	oui
e_2	0.3	0.4	0.3	0	1	0	oui
e_3	0.1	0.2	0.7	1	0	0	non

Modèle num. 2 :

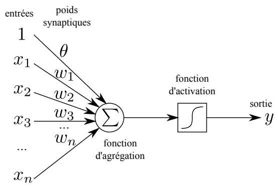
	$\hat{y}$			y			correct ?
e_1	0.1	0.2	0.7	0	0	1	oui
e_2	0.1	0.7	0.2	0	1	0	oui
e_3	0.3	0.4	0.3	1	0	0	non

Perceptron multi-couches - Werbos & Rumelhard (1984-1986)

Sortie du neurone:

$$s = \sum_{i=0}^{n} w_i x_i$$

$$\hat{y} = \frac{1}{1+e^{-s}}$$

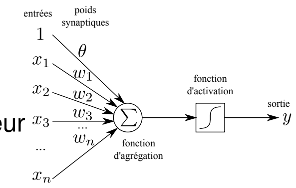


Perceptron multi-couches - Werbos & Rumelhard (1984-1986)

Sortie du neurone:

$$s = \sum_{i=0}^{n} w_i x_i$$

$$\hat{y} = \frac{1}{1+e^{-s}}$$

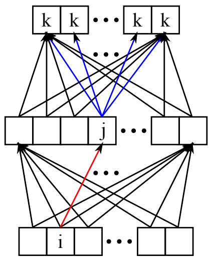


Apprentissage: On veut minimiser l'erreur quadratique $E = \frac{1}{2}\sum_{\mathbf{x}}(y-\hat{y})^2$ par descente de gradient stochastique:
$\Delta\mathbf{w} = \eta\mathbf{x}(y-\hat{y})\hat{y}(1-\hat{y})$

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} = -\eta\frac{\partial E}{\partial s}\frac{\partial s}{\partial w_i} = \eta\sum_{\mathbf{x}}\frac{\partial \hat{y}}{\partial s}(y-\hat{y})\frac{\partial s}{\partial w_i} = \sum_{\mathbf{x}}\eta\hat{y}(1-\hat{y})(y-\hat{y})x_i$$

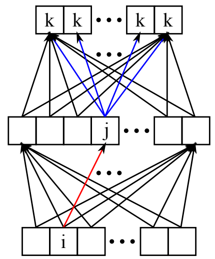
Perceptron multi-couches - Werbos & Rumelhard (1984-1986)

Réseau: connectivité complète entre la couche d'entrée, la(les) couche(s) cachée(s) et la couche de sortie



Perceptron multi-couches - Werbos & Rumelhard (1984-1986)

Réseau: connectivité complète entre la couche d'entrée, la(les) couche(s) cachée(s) et la couche de sortie



Apprentissage: On veut minimiser l'erreur quadratique $E = \frac{1}{2}\sum_{\mathbf{x}}(y-\hat{y})^2$ par descente de gradient stochastique, on suppose connus les $\delta_k = -\frac{\partial E}{\partial s_k}$ de la couche supérieure

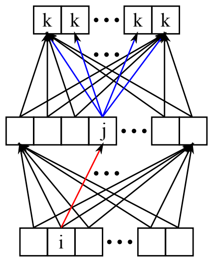
$$\begin{aligned}
-\frac{\partial E}{\partial s_j} &= -\sum_k \frac{\partial E}{\partial s_k}\frac{\partial s_k}{\partial y_j}\frac{\partial y_j}{\partial s_j} \\
&= -\sum_k -\delta_k w_{jk}\hat{y}_j(1-\hat{y}_j) \\
&= \hat{y}_j(1-\hat{y}_j)\sum \delta_k w_{jk}
\end{aligned}$$

Perceptron multi-couches - Werbos & Rumelhard (1984-1986)

Réseau: connectivité complète entre la couche d'entrée, la(les) couche(s) cachée(s) et la couche de sortie

Apprentissage: On veut minimiser l'erreur quadratique $E = \frac{1}{2}\sum_{\mathbf{x}}(y - \hat{y})^2$ par descente de gradient stochastique, on suppose connus les $\delta_k = -\frac{\partial E}{\partial s_k}$ de la couche supérieure

$$
\begin{aligned}
-\frac{\partial E}{\partial s_j} &= -\sum_k \frac{\partial E}{\partial s_k}\frac{\partial s_k}{\partial y_j}\frac{\partial y_j}{\partial s_j} \\
&= -\sum_k -\delta_k w_{jk}\hat{y}_j(1-\hat{y}_j) \\
&= \hat{y}_j(1-\hat{y}_j)\sum_k \delta_k w_{jk}
\end{aligned}
$$

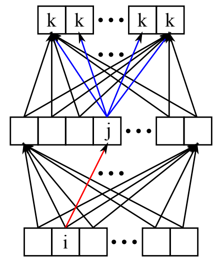


Perceptron multi-couches - Werbos & Rumelhard (1984-1986)

Réseau: connectivité complète entre la couche d'entrée, la(les) couche(s) cachée(s) et la couche de sortie

Apprentissage: Pour chaque entrée reçue:

1. Calculer la sortie $\hat{y}$ du réseau par propagation (couche par couche) de l'activité
2. Calculer l'erreur de la couche de sortie: $\delta_k = \hat{y}_k(1-\hat{y}_k)(y_k - \hat{y}_k)$
3. Rétropropager l'erreur à travers chaque couche j du réseau $\delta_j = \hat{y}_j(1-\hat{y}_j)\sum_k \delta_k w_{jk}$
4. Modifier chaque poids $\Delta w_{ij} = \eta\delta_j x_i$

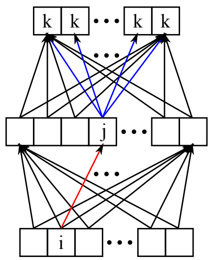


Perceptron multi-couches - Werbos & Rumelhard (1984-1986)

Réseau: connectivité complète entre la couche d'entrée, la(les) couche(s) cachée(s) et la couche de sortie

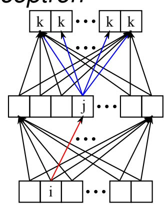
Apprentissage: Pour chaque entrée reçue:

1. Calculer la sortie $\hat{y}$ du réseau par propagation (couche par couche) de l'activité
2. Calculer l'erreur de la couche de sortie: $\delta_k = \hat{y}_k(1-\hat{y}_k)(y_k - \hat{y}_k)$
3. Rétropropager l'erreur à travers chaque couche j du réseau $\delta_j = \hat{y}_j(1-\hat{y}_j)\sum_k \delta_k w_{jk}$
4. Modifier chaque poids $\Delta w_{ij} = \eta\delta_j x_i$



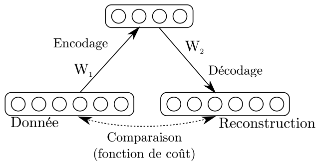
Bref aperçu de l'apprentissage profond

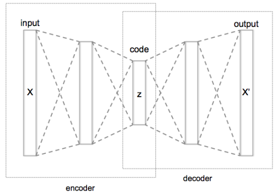
- *Perceptron / Multi-layer perceptron*



Bref aperçu de l'apprentissage profond

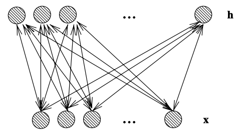
- *Perceptron / Multi-layer perceptron*
- *Auto-encoder / Stacked auto-encoder*



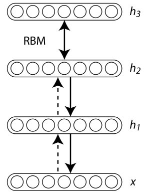


Bref aperçu de l'apprentissage profond

- *Perceptron / Multi-layer perceptron*
- *Auto-encoder / Stacked auto-encoder*
- *Restricted Boltzmann machine / Deep belief network*

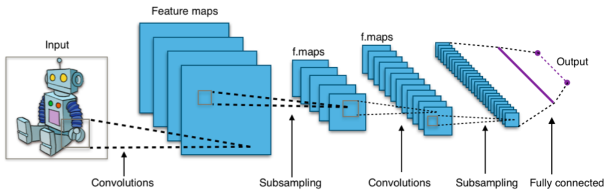


$$P(\mathbf{h}^{\ell-1}, \mathbf{h}^{\ell}) \propto e^{\mathbf{b}'\mathbf{h}^{\ell-1} + \mathbf{c}'\mathbf{h}^{\ell} + \mathbf{h}^{\ell'} W \mathbf{h}^{\ell-1}}$$



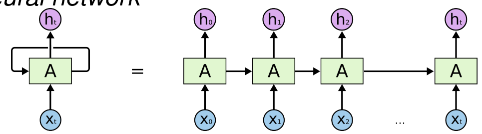
Bref aperçu de l'apprentissage profond

- *Perceptron / Multi-layer perceptron*
- *Auto-encoder / Stacked auto-encoder*
- *Restricted Boltzmann machine / Deep belief network*
- *Convolutional neural network*



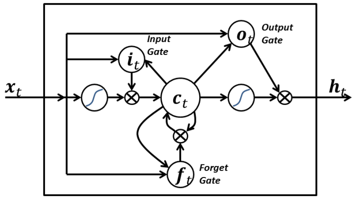
Bref aperçu de l'apprentissage profond

- *Perceptron / Multi-layer perceptron*
- *Auto-encoder / Stacked auto-encoder*
- *Restricted Boltzmann machine / Deep belief network*
- *Convolutional neural network*
- *Recurrent neural network*



Bref aperçu de l'apprentissage profond

- *Perceptron / Multi-layer perceptron*
- *Auto-encoder / Stacked auto-encoder*
- *Restricted Boltzmann machine / Deep belief network*
- *Convolutional neural network*
- *Recurrent neural network*
- *Long Short Term Memory*



Tips and tricks

Quelques "règles du pouce" (*rules of thumb*) :

- Préférer des fonctions symétriques autour de l'origine (simoïde ou tanh) car elles fournissent une entrée centrée en 0 pour la couche suivante ; on a observé que tanh a de meilleurs propriétés de convergence
- Cependant, la fonction reLU semble très employée ces derniers temps (bonnes propriétés héritées de la linéarité)
- Les poids initiaux doivent être petits et proches de 0 afin d'avoir des variations linéaires au démarrage
  - pour tanh : *uniforme*$[-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}]$ où f_{in} et f_{out} sont le nombre de connexions entrantes et sortantes respectivement
  - pour la sigmoïde : *uniforme*$[-\frac{4*\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{4*\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}]$
- Taux d'apprentissage η, nombre de neurones, régularisation...

cf. http://deeplearning.net/tutorial/deeplearning.pdf

Régularisation

Comme pour les autres modèles d'apprentissage automatique, les réseaux de neurones sont menacés par le sur-apprentissage (*overfitting*).

Régularisation

Comme pour les autres modèles d'apprentissage automatique, les réseaux de neurones sont menacés par le sur-apprentissage (*overfitting*). Une solution est de *régulariser* la fonction objectif :

$$f_{obj} = f_{err} + \lambda.\Omega(\theta)$$

où $\lambda \in [0, \infty)$ est un hyper-paramètre à fixer.

Régularisation

Comme pour les autres modèles d'apprentissage automatique, les réseaux de neurones sont menacés par le sur-apprentissage (*overfitting*). Une solution est de *régulariser* la fonction objectif :

$$f_{obj} = f_{err} + \lambda.\Omega(\theta)$$

où $\lambda \in [0, \infty)$ est un hyper-paramètre à fixer. Des valeurs typiques à essayer pour λ sont 10^{-2}, 10^{-3}, etc.

Régularisation

Comme pour les autres modèles d'apprentissage automatique, les réseaux de neurones sont menacés par le sur-apprentissage (*overfitting*). Une solution est de *régulariser* la fonction objectif :

$$f_{obj} = f_{err} + \lambda.\Omega(\theta)$$

où $\lambda \in [0, \infty)$ est un hyper-paramètre à fixer. Des valeurs typiques à essayer pour λ sont 10^{-2}, 10^{-3}, etc.

Ω est souvent une norme, telle que la norme $L1$ (cf. LASSO), $L2$ (cf. *ridge regression*) ou les deux à la fois (cf. *elastic net*).

Régularisation

Comme pour les autres modèles d'apprentissage automatique, les réseaux de neurones sont menacés par le sur-apprentissage (*overfitting*). Une solution est de *régulariser* la fonction objectif :

$$f_{obj} = f_{err} + \lambda.\Omega(\theta)$$

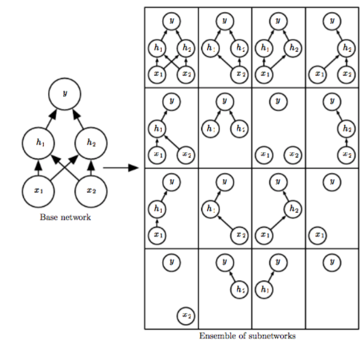
où $\lambda \in [0, \infty)$ est un hyper-paramètre à fixer. Des valeurs typiques à essayer pour λ sont 10^{-2}, 10^{-3}, etc.

Ω est souvent une norme, telle que la norme $L1$ (cf. LASSO), $L2$ (cf. *ridge regression*) ou les deux à la fois (cf. *elastic net*).

Une autre manière de régulariser consiste à arrêter l'apprentissage à temps (*early stopping*).

Drop-out

Forme de régularisation basée sur l'estimation d'un *ensemble* de réseaux calculés à partir du réseau initial :



tiré de Goodfellow et al. (2015)