CM 4-5 Cognition Artificielle

Les modèles du connexionnisme

1. **Réseaux de Hopfield**

Le modèles de réseau récurrent de Hopfield (1982/1984)

* Réseau **récurrent**
* Totalement **interconnecté**
* Connexions **symétriques** : wij = wji
* Pas « d’auto-connexions » : wii = 0
* Activation **asynchrone**
  + On ne change qu’une cellule à la fois
  + A permis à Hopfield de prouver la convergence
* Les points peuvent être appris ou calculés

Inspiration : Hebb (biologiste) 🡪 « il faut renforcer les poids des cellules activées simultanément »

1. Calcul des poids d’un réseau Hopfield

Apprentissage selon la règle de Hebb => wij = wij + µyiyj

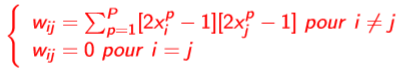
* (µ) : paramètre déterminant l’intensité de l’apprentissage
* (yi) : sortie de la cellule i = « état » de la cellule i

Calcul des poids :

* Pour P exemples binaires (-1, 1) X1, X2,... ,XP :



* Pour P exemples binaires plus classique, dans (0,1) :



1. Apprentissage des poids d’un réseau Hopfield

On obtient les mêmes poids que par le calcul, avec l’algorithme suivant :

1. Démarrer avec des connexions nulles
2. Présenter un exemple Xp : yi = xpi pour i = 1 → n
3. Modifier les connexions :

+1 si les cellules sont actives ou inactives simultanément

-1 sinon

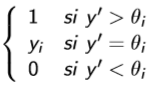
1. Boucler en 2, jusqu’à ce que les P exemples aient été présentés
2. Généralisation dans un réseau Hopfield

Algorithme de généralisation :

1. On présente l’exemple X : yi = xi pour i = 1 → n
2. Tant que le réseau n’a pas convergé, pour chaque cellule (dans un ordre aléatoire) :

On active la cellule : y’i = ∑nj = 1 yj wij

On détermine le nouvel état de la cellule :



Remarques :

* On a convergé si les états (yi) ne changent plus
* On prend généralement tous les θi = 0

Borne sur la capacité de stockage d’un réseau Hopdield

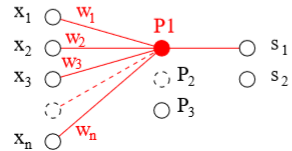
*Quel est le nombre de formes que l’on peut stocker et reconnaître avec une « fréquence raisonnable » avec un réseau de Hopfield de (n) cellules (avec l’algorithme d’apprentissage vu) ?*

Selon les études expérimentales de Hopfield : Pmax = 0,15n

Borne théorique en 1987 : Pmax = n / 2log2 n

1. **Les réseaux LVQ et les cartes de Kohonen**

Topologie de réseaux Learning Vector Quantization (LVQ)



Réseaux à 3 couches :

* L’entrée : X = (x1, x2, …, xn)
* Les prototypes Pi = (w1, w2, …, wn)
* Les sorties

Algorithme de généralisation :

1. On présente un exemple à apprendre / une forme à reconnaître
2. Tous les prototypes s’activent… et le plus fort gagne !
3. La sortie correspondant au prototype gagnant d’active 🡪 correspond en général à une classe

Algorithme d’apprentissage typique de LVQ

Pour chaque exemple X à apprendre :

Pour i tel que Pi = Pmeilleur :

Si C(Pi) = E alors ∀j wji ← wji + α (xj −wji)

Sinon ∀j wji ← wji −α (xj −wji)

Avec :

* E = étiquette de la forme X = (x1x2 … xn) en entrée
* C(Pi) = classe du prototype Pi
* α (α < 1) = paramètre de gain

Précision : α pondère les modifications et diminue lentement au cours de l’apprentissage

Variantes de l’apprentissage de LVQ

Il existe de nombreuses variantes, par exemple :

* Repousser tous les perdants
* Augmenter le θ du gagnant et éventuellement diminuer ceux du perdant
* …

Remarques sur le prototype gagnant :

* Il devient le représentant d’une classe
* Il est affiné d’exemple en exemple appris

Les cartes de Kohonen

Cartes de Kohonen = cartes « auto-organisatrices » = Self Organizing Maps (SOM)

Réseau à couches avec :

* **2** couches en non supervisé, **3** couches en supervisé
* La 2e couche forme une carte, généralement 2D ou 3D
* Connexions latérales sur la 2e couche, selon le voisinage des cellules sur la carte

Nombre de cellules :

* Couche 1 : selon la dimension de l’espace d’entrée
* Couche 2 : selon la taille de la carte

Algorithmique des cartes de Kohonen

Algorithme de généralisation : idem LVQ

Algorithme d’apprentissage :

1. On présente un exemple
2. On cherche le prototype le plus activé
3. On approche le prototype gagnant et ses voisins sur la carte

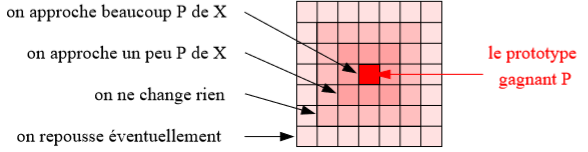
Remarques :

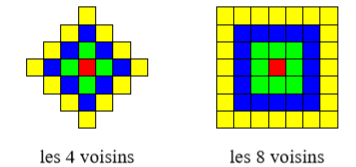
* En général on laisse les prototypes perdants inchangés
* C’est de l’apprentissage non supervisé mais... on peut ajouter une 3e couche pour l’´étiquetage → supervisé

Inﬂuences de la distance des voisins

Modiﬁcations des poids :

* La modiﬁcations des poids d´dépend de la distance des voisins
* La modiﬁcations peut être négative en limite de voisinage par exemple en utilisant une fonction type « chapeau mexicain »



Forme et taille du voisinage

On peut déﬁnir plusieurs types de voisinages :

* Les 4 voisins (et leurs voisins...)
* Les 8 voisins (et leurs voisins...)

Il y a d’autres voisinages possibles, surtout en 3D...

Auto-organisation de la carte !

Il est fréquent de faire varier la taille du voisinage au cours de l’apprentissage, par exemple :

* Au début, toute la carte
* À chaque « passe » de la base d’exemples on diminue le voisinage d’une cellule
* À la ﬁn, on ne modiﬁe que le prototype gagnant

On constate une auto-organisation de la carte :

* Les prototypes représentants d’une même classe sont adjacents → facilite l’´étiquetage en non supervisé
* La disposition des classes sur la carte rend compte de propriétés communes

1. **La perception multicouche et la rétropropagation du gradient**

Comment apprendre avec un perceptron multi-couche

Rappel :

* Une couche = on sait faire (perceptron)
* Plusieurs couches = credit assigment problem

Idée : retro-propager l’erreur de la sortie vers l’entrée, en modiﬁant les poids des connexions en fonction de leur contribution à l’erreur

Rappel : apprentissage = modiﬁer les poids

⇒ il faut expliciter l’erreur globale en fonction de tous les poids puis dériver cette fonction par rapport aux poids dont on veut connaitre la contribution à l’erreur globale

⇒ il faut utiliser des fonctions d’activation dérivables !

La fonction sigmoïde

On veut toujours un « eﬀet seuil » mais avec une fonction dérivable

On peut par exemple utiliser la fonction sigmoïde : f (a) =1 / (1 + e−a)

Couche d’entrée : X = (x0,x1,x2,...,xn) ∈ IRn+1 (x0 = 1)

Couche cachée : Z = (z0,z1,x2,...,zJ) ∈ IRJ+1 (z0 = 1)

Couche de sortie : Y = (y1,y2,...,yL) ∈ IRL

Sortie désirée : D = (d1,d2,...,dL) ∈ IRL

Sortie désirée connue = l’apprentissage est dit « supervisé »

Apprentissage d’un perceptron multi-couche

1. On présente un exemple X
2. On active la couche cachée ≡ on calcul Z
3. On active la couche de sortie ≡ on calcul Y
4. On détermine l’erreur, entre Y et D
5. On modiﬁe les poids entre la couche cachée et la couche de sortie, en fonction de leur contribution à l’erreur
6. On modiﬁe les poids entre la couche d’entrée et la couche cachée, en fonction de leur contribution à l’erreur

Choix de la fonction d’activation des couches cachées

Pour la couche cachée, par exemple :

* Sigmoïde :

fh(net) = 1 / (1 + e−λnet)

* Tangente hyperbolique :

fh(net) = tanh(βnet)

Avec :

* Net = somme pondérée des entrées de la cellule, en l’occurrence : net = ∑ni =0 xiwji
* λ et β = paramètres de l’apprentissage (proches de 1)

Choix de la fonction d’activation de la couche de sortie

Pour la couche de sortie → d´dépend de l’application :

* Approximation, prévision...

⇒ sortie réelle et proportionnelle, par exemple : fo (net) = λ net

* classiﬁcation, décision...

⇒ sortie binaire, par exemple : fo (net) = 1 / (1 + e−λnet)

Remarque : il faut que la sortie d´désirée (D) soit dans l’intervalle des valeurs que peut prendre fo ⇒ le cas échéant il faut normaliser

Fonction d’erreur

On veut apprendre m paires d’exemples : {Xk,Dk} k = 1,..., m ≡ on veut adapter les J(n + 1) + L(J + 1) poids du réseau aﬁn de minimiser l’erreur sur l’ensemble des exemples

⇒ on a besoin d’une fonction d’erreur !

≡ une mesure de la distance entre D et Y

⇒ très dépendant de l’application

Exemple, erreur quadratique :

d(W) = 1 / 2 ∑li=1 (dl −yl)2

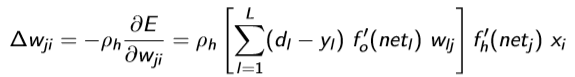
Avec W représentant l’ensemble des poids du réseau

Modification des poids

* Couche de sortie, on connait la sortie désirée :



* Couches cachées :



1. **Utilisation de la bibliothèque FANN**

Fast Artiﬁcial Neural Network (FANN) library is a free open source neural network library, which implements multilayer artiﬁcial neural networks in C with support for both fully connected and sparsely connected networks. FANN includes a framework for easy handling of training data sets. FANN is easy to use, versatile, well documented, and fast. Bindings to more than 20 programming languages are available.

Quelques caractéristiques de FANN

* Multilayer Artiﬁcial Neural Network Library in C
* Backpropagation training
* Easy to use (create, train and run with just three functions)
* Cross-platform (linux and unix, windows)
* Open source, but can still be used in commercial applications
* Graphical Interfaces B
* indings to a large number of programming languages
* Widely used (approximately 100 downloads a day)

La licence LGPL de FANN

Rappel sur la GPL : GNU General Public License

L’objectif est de garantir à l’utilisateur les droits suivants :

1. Liberté d’exécuter le logiciel, pour n’importe quel usage ;
2. Liberté d’étudier le programme et de l’adapter à ses besoins, ce qui passe par l’accès aux codes sources ;
3. Liberté de redistribuer des copies ;
4. Obligation de faire bénéﬁcier à tous des modiﬁcations.

La LesserGPL (ou licence GPL « faible ») de FANN

La licence libre LGPL autorise à lier le programme `a du code non LGPL, sans pour autant révoquer la licence. Cette Licence LGPL permet donc de s’aﬀranchir du caractère héréditaire de la GPL. C’est donc la clause de copyleft que n’a pas la LGPL.

* Permet à un logiciel propriétaire d’utiliser une librairie libre, sans devenir un logiciel libre ;

Toute modiﬁcation de code source de la bibliothèque LGPL devra être également publiée sous la licence LGPL ;

* Passage sous GPL par simple mise à jour des notiﬁcations.

FANN à partir de Python pour apprendre un XOR

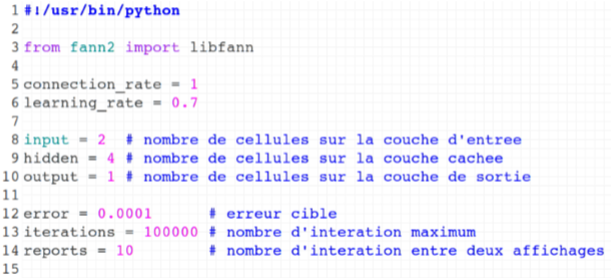
On va apprendre la fonction XOR avec un MLP, il nous faut une base d’apprentissage : ﬁchier texte « xor.data » → avec la première ligne comprenant :

* Le nombre d’exemples (4)
* Le nombre d’entrées (2)
* Le nombre de sortie(s) (1)

Puis les exemples : les entrées et à la ligne la sortie

Remarque : on a pris les entrées et sorties dans [-1, 1] mais on aurait pu les prendre dans [0, 1]

Paramètre du réseau



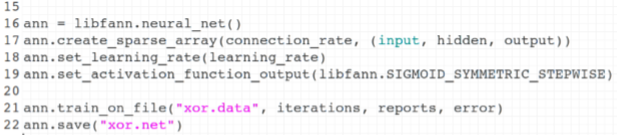
Création du réseau et apprentissage

On crée le MLP (lignes 16 et 17),

on ﬁxe le taux d’apprentissage (18),

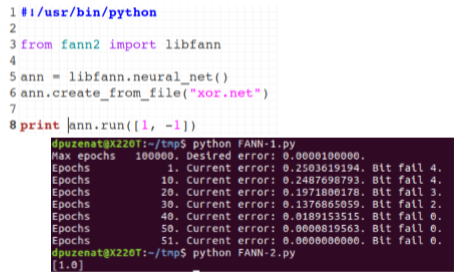
on choisit une fonction sigmoïde dans [-1,1] pour la cellule de la couche de sortie (19),

puis on apprend (21) et on sauve l’architecture et les poids (22) :



Mise en œuvre de réseau et résultat

Dans un autre programme on peut charger le réseau et l’utiliser :



Remarques : le résultat (valeur réelle 1) aurait pu être récupérée dans une variable

1. **Machine à vecteurs de support**

En anglais Support Vector Machine ou SVP.

Intérêt :

* Capacité à travailler avec des données de grandes dimensions
* Faible nombre d’hyper paramètres
* Garanties théoriques
* Bons résultats en pratique

Origine : développé dans les années 1990 à partir des travaux de Vladimir Vapnik

Machine `a vecteurs de support – principe

On part d’un neurone type perceptron mais on recherche l’hyperplan séparateur avec « a marge maximale »

Problème : limité aux problèmes linéairement séparables...

Solution : transformer l’espace de représentation des données d’entrées en un espace de plus grande dimension dans lequel existera une s´séparatrice linéaire

Astuce : on va utiliser des fonctions noyaux particulières

Les fonctions noyau utilisées ne n´nécessitent pas la connaissance explicite de la transformation `a appliquer pour le changement d’espace ⇒ m´méthode peu conteuse en calculs!

Machine à vecteurs de support – extensions du modèle

On peut faire du multi-classes avec plusieurs classiﬁeurs SVM : méthodes one-versus-all (≡ one-versus-the-rest) et one-versus-one

1995 : introduction des « marges souples »

⇒ permettent de relâcher les contraintes sur les exemples

⇒ évite le sur-apprentissage et permet de gérer des exemples mal étiquetés mais introduit un hyper - paramètre « C » (C = compromis nombre d’erreurs / largeur de la marge)

1996 : méthode pour utiliser des SVM en régression (sortie réelle) ⇒ on peut tout faire ⇒ devient très populaire

1. **Réseaux neuronaux convolutifs et apprentissage profond**

Réseaux neuronaux convolutifs

Réseau de neurones convolutifs = « réseau de neurones à convolution » = « CNN » = « ConvNet » réseau acyclique,

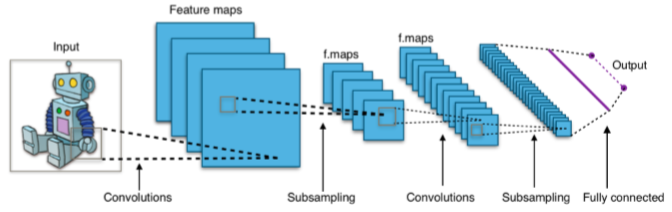
* ie feed-forward
* Connexion entre les neurones inspirée du cortex visuel

Cortex visuel : Les neurones sont arrangés de sorte à ce qu’ils correspondent à des régions qui se chevauchent lors du pavage du champ visuel

🡪 Empilage multicouche de perceptrons dont le but est de prétraiter de petites quantités d’informations

Applications : reconnaissance d’image et vidéo, traitement du langage naturel, ...

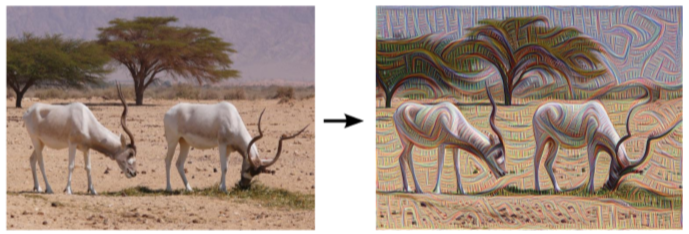
Exemple d’architecture d’un réseau convolutif



Attention : n´nécessite davantage d’hyperparamètres qu’un MLP standard (nombre de ﬁltres, leur forme, etc.).

Exemple d’image intermédiaire créée par un CNN

Exemple d’image intermédiaire, ou « carte d’activation », créée par un réseau neuronal convolutif :



Avantage des CNN par rapport aux MLP

Les avantages des CNN :

* Un seul poids pour tous les signaux entrant dans tous les neurones d’un même noyau de convolution

⇒ invariance du traitement par translation

* Relativement peu de pré-traitement

⇒ moins d’intervention humaine (programmation)

⇒ adaptation des ﬁltres par apprentissage (non supervisé)

D’un point de vu CPU : moins de calculs (un seul poids par noyau) et utilisation fréquente de GPU

Apprentissage profond ou « Deep-learning »

Plusieurs architectures possibles : « deep neural networks » « convolutional deep neural networks » « deep belief network » etc.

→ excellents résultats pour diﬀérentes problématiques

Particularités : diﬀérentes couches non-linéaires pour l’extraction et la transformation des caractéristiques chaque couche prend en entrée la sortie de la précédente les algorithmes peuvent être supervisés ou non supervisé

→ apprentissage à plusieurs niveaux de d´détails ou de représentations des données