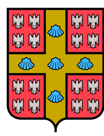


Étude théorique et expérimentale de différentes architectures neuronales dérivées de la théorie ART

Martin Busque

RT-LVSN-97-06 Août 1997

FACULTÉ DES SCIENCES ET DE GÉNIE
Département de génie électrique et de génie informatique

Laboratoire de Vision et Systèmes Numériques
Université Laval
Sainte-Foy (Québec), Canada
G1K 7P4
Tel: (418) 656-3554 / (418) 656-2979
Fax: (418) 656-3594
Email: lvsn@gel.ulaval.ca
WWW: http://www.gel.ulaval.ca/~vision

Table des matières

Introduction **1**

I Architectures non supervisées **3**

1 ART1 **5**

1.1 Algorithme d'apprentissage 5

1.1.1 Structure du réseau 6

1.1.2 Équations 7

1.1.3 Paramètres 10

1.1.4 Algorithme simplifié 12

1.2 Fonctionnement 15

1.2.1 Fichiers de données 16

1.2.2 Influence de la vigilance ρ 19

1.2.3 Frontières apprises 21

1.2.4 Ordre de présentation des données 23

1.2.5 Région d'attraction 26

1.2.6 Recouvrements 28

1.2.7 Influence de L 29
1.3 Implantation 30
1.3.1 Classe ART1 31
1.3.2 Programmes utilisant le ART1 35
1.3.3 Classe StackInterval 38

2 Fuzzy ART 41

2.1 Algorithme d'apprentissage 41
2.1.1 Structure du réseau 42
2.1.2 Équations 43
2.1.3 Paramètres 44
2.2 Fonctionnement 46
2.2.1 Comparaison avec le ART1 46
2.2.2 Influence du taux d'apprentissage β 48
2.2.3 Influence du paramètre de choix α 49
2.3 Implantation 52
2.3.1 Classe FuzzyART 52
2.3.2 Programmes utilisant le Fuzzy ART 57

3 Fuzzy Min-Max 61

3.1 Algorithme d'apprentissage 61
3.1.1 Structure du réseau 62
3.1.2 Équations 62
3.1.3 Paramètres 65

3.2 Fonctionnement 66
3.2.1 Élimination des recouvrements 67
3.2.2 Instabilité perpétuelle 70
3.2.3 Influence de la dimension maximale θ 72
3.2.4 Influence de la sensibilité γ 74
3.3 Implantation 76
3.3.1 Classe FuzzyMinMax 76
3.3.2 Classe HyperRectangle 79
3.3.3 Programmes utilisant le Fuzzy Min-Max 81

II Architectures supervisées 85

4 LAPART 87

4.1 Algorithme d'apprentissage 87
4.1.1 Structure du réseau 88
4.1.2 Principe de fonctionnement 89
4.1.3 Équations 89
4.1.4 Paramètres 91
4.2 Fonctionnement 92
4.2.1 Constatations initiales 92
4.2.2 Influence de ρ_A 94
4.2.3 Phénomène de capture 95
4.2.4 Mode classement 96
4.3 Implantation 97

4.3.1 Classe LAPART 97
4.3.2 Programme utilisant le LAPART 100

5 Fuzzy ARTMAP **103**

5.1 Algorithme d'apprentissage 103
5.1.1 Structure du réseau 104
5.1.2 Principe de fonctionnement 104
5.1.3 Équations 105
5.1.4 Paramètres 107
5.2 Fonctionnement 108
5.2.1 Influence de $\overline{\rho_a}$ 109
5.2.2 Apprentissage en séquence 112
5.2.3 Apprentissage d'une nouvelle classe 112
5.2.4 Distinction de deux spirales 115
5.3 Implantation 119
5.3.1 Classe FuzzyARTMAP 119
5.3.2 Programmes utilisant le Fuzzy ARTMAP 122

Conclusion **127**

Bibliographie **129**

A Codes sources — ART1 **131**

B Codes sources — Fuzzy ART **143**

C Codes sources — Fuzzy Min-Max 153

D Codes sources — LAPART 163

E Codes sources — Fuzzy ARTMAP 171

Liste des figures

1.1 *Schéma du réseau ART1.* 6

1.2 *Fichiers de données.* 17

1.3 *Classement des fichiers de données ($L = 1.1$ et $\rho = 0.5$).* 18

1.4 *Classement des données #2 ($L = 1.1$ et $\rho = 0.3$).* 20

1.5 *Classement des données #2 ($L = 1.1$ et $\rho = 0.75$).* 20

1.6 *Classement des données #3 ($L = 1.1$ et $\rho = 0.6$).* 20

1.7 *Classement des données #1 avec les frontières ($L = 1.1$ et $\rho = 0.5$).* 21

1.8 *Démonstration de l'apprentissage du ART1.* 22

1.9 *Classement des données uniformes ($L = 1.1$ et $\rho = 0.4$).* 24

1.10 *Classement des données uniformes avec permutation aléatoire ($L = 1.1$ et $\rho = 0.4$).* 24

1.11 *Classement des données #3 avec permutation aléatoire ($L = 1.1$ et $\rho = 0.5$).* 25

1.12 *Classement des données #5 avec permutation aléatoire ($L = 50$ et $\rho = 0.6$).* 26

1.13 *Région d'attraction autour d'une petite frontière.* 27

1.14 *Région d'attraction autour d'une frontière moyenne.* 27

1.15 *Région d'attraction autour d'une grande frontière.* 27

1.16 *Classement des données #2 avec les frontières ($L = 1.1$ et $\rho = 0.5$).* 28

1.17 *Classement des données #5 ($L = 500$ et $\rho = 0.2$).* 29

2.1 *Schéma du réseau Fuzzy ART.* 42

2.2 *Classement des données #1 ($\alpha = 0.01$, $\beta = 1$ et $\rho = 0.45$).* 47

2.3 *Frontière autour d'un point quand $\beta = 0.5$.* 48

2.4 *Frontière autour d'un point quand $\beta = 0.5$, après une seconde présentation du point.* 48

2.5 *Classement des données #2 avec permutation aléatoire ($\alpha = 0.01$, $\beta = 0.1$ et $\rho = 0.5$).* 49

2.6 *Classement des données #4 ($\alpha = 7$, $\beta = 1$ et $\rho = 0.5$).* 50

2.7 *Région d'attraction autour d'un point avec $\alpha = 5$.* 50

2.8 *Région d'attraction autour d'un point avec $\alpha = 50$.* 51

2.9 *Classement des données #1 après une passe ($\alpha = 1$, $\beta = 1$ et $\rho = 0.5$).* 51

3.1 *Exemple #1* 68

3.2 *Exemple #2* 69

3.3 *Exemple #3* 71

3.4 *Classement des données #5 avec permutation aléatoire ($\gamma = 4$ et $\theta = 0.4$).* 73

3.5 *Classement des données #3 avec permutation aléatoire ($\gamma = 4$ et $\theta = 0.5$).* 73

3.6 *Classement des données #2 avec permutation aléatoire ($\gamma = 4$ et $\theta = 0.7$).* 73

3.7 *Fonction d'appartenance autour d'un hyper-rectangle.* 75

4.1 *Schéma du réseau LAPART.* 88

4.2 *Classement des données #1* ($\rho_A = 0.5$). 93

4.3 *Classement des données #2* ($\rho_A = 0.5$). 93

4.4 *Classement des données #5* ($\rho_A = 0.5$). 93

4.5 *Classement des données #3* ($\rho_A = 0.5$). 94

4.6 *Classement des données #3* ($\rho_A = 0.6$). 95

4.7 *Classement de données après entraînement avec les données #4 avec* $\rho_A = 0.8$. 96

4.8 *Classement de données* ($\rho_A = 0.01$) *après entraînement avec les données #4 avec* $\rho_A = 0.8$. 97

5.1 *Schéma du réseau Fuzzy ARTMAP*. 104

5.2 *Classement de données après entraînement avec les données #3 avec* $\overline{\rho_a} = 0.5$. 109

5.3 *Classement de données après entraînement avec les données #3 avec* $\overline{\rho_a} = 0.5$ *et permutation aléatoire.* 110

5.4 *Classement de données après entraînement avec chacun des fichiers avec* $\overline{\rho_a} = 0.9$ *et permutation aléatoire.* 111

5.5 *Apprentissage des données #5 une classe à la fois avec* $\overline{\rho_a} = 0.9$. . 113

5.6 *Apprentissage des données #5 en excluant la classe 2 avec* $\overline{\rho_a} = 0.9$ *et permutation aléatoire.* 114

5.7 *Apprentissage des données #5 en excluant la classe 4 avec* $\overline{\rho_a} = 0.9$ *et permutation aléatoire.* 114

5.8 *Fichier de données "spirales"*. 115

5.9 *Classement de données après entraînement avec les spirales.* 117

5.10 *Rectangles créés après entraînement avec les spirales.* 118

5.11 *Classement de données après entraînement avec les spirales avec* $\overline{\rho_a} = 0$. 118

Introduction

Ce rapport porte sur le fruit d'un été de travail pour M. Marc Parizeau au Laboratoire de vision et systèmes numériques de l'Université Laval. Le but de ce travail était d'étudier différentes architectures neuronales dérivées de la théorie ART, Adaptive Resonance Theory, tant du point de vue théorique que pratique. Ces architectures ont comme utilité d'effectuer du classement de données.

L'objectif de ce rapport est d'illustrer le fonctionnement de chacun de ces réseaux de neurones à l'aide d'un cas simple afin d'avoir une idée de leurs capacités respectives, pour éventuellement en utiliser une ou plusieurs dans des applications plus complexes. Le cas choisi, en particulier pour sa simplicité à être visualisé, est le classement de points dans un plan à deux dimensions.

Les détails de chaque architecture ne sont pas répétés dans ce rapport. Nous supposons que le lecteur saura se référer aux articles mentionnés dans la bibliographie s'il recherche des informations plus détaillées.

Pour étudier le fonctionnement de chaque architecture, il faut au préalable expliquer suffisamment l'algorithme d'apprentissage de chacune, en clarifiant possiblement certains points qui sont plus ou moins évidents dans les articles originaux. Aussi, on retrouve la documentation des classes simulant ces réseaux, développées en C++, ainsi que celle des programmes qui ont été réalisés.

Certains réseaux de neurones sont non supervisés tandis que d'autres sont supervisés. Les architectures non supervisées divisent elles-mêmes les données en différentes catégories. Les architectures supervisées, elles, apprennent à classer des données en sachant à quelles catégories celles-ci appartiennent. Nous allons voir, dans la première partie du rapport, les architectures non supervisées qui ont été étudiées. Celles-ci sont le ART1, le Fuzzy ART et le Fuzzy Min-Max. La seconde partie traite des architectures supervisées, soit le LAPART et le Fuzzy ARTMAP.

Chacun des chapitres est structuré de la même façon. Il y a d'abord une

explication de l'algorithme d'apprentissage du réseau étudié, dont la structure, les équations et les paramètres. Ensuite, le fonctionnement du réseau sur des points dans un plan est illustré et expliqué. Finalement, il y a la documentation sur l'implantation du réseau concerné en C++.

Partie I

Architectures non supervisées

Chapitre 1

ART1

Le ART1 est le premier réseau de neurones de la famille ART (Adaptive Resonance Theory), introduite par Carpenter et Grossberg en 1987 [1]. Il a comme caractéristique importante la capacité d'apprendre de façon *continue* et *sans supervision*. Ceci signifie, d'une part, que le réseau effectue par lui-même une classification des données qui lui sont présentées, sans qu'on ait à l'entraîner avec un ensemble de données dont les classes sont connues. D'autre part, le ART1 apprend continuellement, sans oublier ce qu'il a appris précédemment. Le réseau n'accepte que des données binaires (0 ou 1), cependant il est en général assez simple de convertir d'autres types de données en ce format.

Pour débuter, nous verrons un résumé de la théorie concernant l'algorithme d'apprentissage du ART1. Ensuite, nous illustrerons son fonctionnement à l'aide d'une simulation de celui-ci en C++. Les détails de cette implantation seront vus à la dernière section du chapitre.

1.1 Algorithme d'apprentissage

L'algorithme d'apprentissage du ART1 est conçu de façon à lui permettre d'apprendre sans supervision et en continu, tout en étant capable de distinguer les données qui sont pertinentes de celles qui ne le sont pas.

La structure générale du réseau est d'abord présentée, suivie des équations gérant son fonctionnement. La section suivante analyse les différents paramètres du ART1 et pour terminer nous présentons un algorithme d'apprentissage sim-

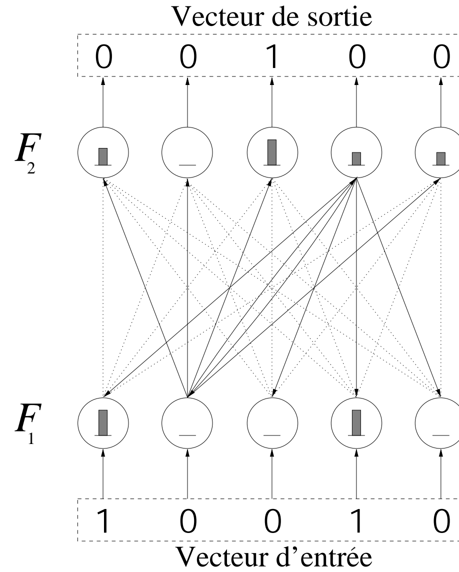


Figure 1.1: *Schéma du réseau ART1.*

plifié où des paramètres inutiles pour l'utilisation que nous faisons du ART1 sont éliminés.

1.1.1 Structure du réseau

Un réseau de type ART1 est formé de deux couches de neurones, tel qu'illustré à la figure 1.1. On nomme les couches F_1 et F_2, la première étant la couche d'entrée et l'autre la couche de sortie. À chaque neurone sont associées une activité et une sortie. Chacun des neurones est connecté à tous les neurones de l'autre couche. On nomme *bottom-up* les connexions de la couche F_1 vers la couche F_2, et *top-down* celles de F_2 vers F_1. Les connexions sont pondérées, c'est-à-dire qu'un poids est associé à chacune d'elles. L'ensemble des poids *bottom-up* et *top-down* servent de mémoire à long terme car ils sont conservés logtemps. Les activités des neurones, elles, forment la mémoire à court terme car elles changent pour chaque donnée.

La dimension de la couche F_1 correspond au nombre de composantes du vecteur d'entrée. La dimension de la couche F_2, elle, représente le nombre maximal de classes que le ART1 pourra former. Les composantes du vecteur d'entrée doivent être binaires (0 ou 1). Seulement un neurone de F_2, celui représentant la classe identifiée par le réseau, a une sortie égale à 1, les autres ayant une sortie nulle.

1.1.2 Équations

Voici maintenant un résumé de l'algorithme d'apprentissage du ART1 avec les équations correspondantes. La notation utilisée est celle de Freeman [5, 6]. Les neurones sur F_1 sont identifiés par v_i et ceux sur F_2 par v_j. L'indice i réfère toujours à la couche F_1 et j à la couche F_2. M et N représentent respectivement la dimension de F_1 et de F_2.

Il y a différents paramètres qui sont utilisés: ρ (qu'on nomme *vigilance*), L, A_1, B_1, C_1, et D_1. Plus de détails sur chacun de ces paramètres sont donnés à la section 1.1.3.

1. Initialiser les paramètres en respectant les contraintes suivantes:

$$0 < \rho \leq 1$$

$$L > 1$$

$$A_1 \geq 0$$

$$C_1 \geq 0$$

$$D_1 \geq 0$$

$$\max\{D_1, 1\} < B_1 < D_1 + 1$$

2. Initialiser les poids des connexions.

   - Poids *top-down* z_{ij} (reliant v_j à v_i):

$$1 \geq z_{ij}(0) > \frac{B_1 - 1}{D_1} \tag{1.1}$$

   Cette équation, l'inégalité d'apprentissage (*template learning inequality*), assure que les poids *top-down* initiaux ne soient pas trop petits et qu'ils empêchent ainsi les neurones inutilisés de F_2 d'apprendre tout vecteur d'entrée [1, section 17B][1].

   - Poids *bottom-up* z_{ji} (reliant v_i à v_j):

$$0 \leq z_{ji}(0) \leq \frac{L}{L - 1 + M} \tag{1.2}$$

[1] Attention à la notation différente de Carpenter et Grossberg, où z_{ji} représente le poids de la connexion allant de v_j vers v_i, contrairement à notre notation où z_{ji} représente le poids de la connexion allant de v_i vers v_j.

Cette équation, l'inégalité d'accès direct (*direct access inequality*), assure que les neurones ayant appris un certain vecteur d'entrée soient activés directement par celui-ci lorsqu'il se représentera [1, section 16].

3. Appliquer un vecteur d'entrée $\mathbf{I}$ et le propager vers F_1.
   - Calculer les activités x_{1i} pour F_1.

$$x_{1i} = \frac{I_i}{1 + A_1(I_i + B_1) + C_1} \tag{1.3}$$

   - Calculer les sorties s_i pour F_1.

$$s_i = \begin{cases} 1 & x_{1i} > 0 \\ 0 & x_{1i} = 0 \end{cases} \tag{1.4}$$

4. Propager les sorties de F_1 vers F_2.
   - Calculer les activités T_j pour F_2. Chaque activité est la somme des sorties de F_1 pondérées par les poids des connexions *bottom-up* appropriées.

$$T_j = \sum_{i=1}^{M} s_i z_{ji} \tag{1.5}$$

   Comme $s_i \in \{0, 1\}$ (éq. 1.4) et $z_{ji} \geq 0$ (éq. 1.11), $T_j \geq 0$.
   - Calculer les sorties u_j pour F_2. Seulement le neurone ayant l'activité la plus forte, le neurone gagnant v_J, a une sortie non nulle.

$$u_j = \begin{cases} 1 & T_j = \max_k \{T_k\} \forall k \\ 0 & \text{sinon} \end{cases} \tag{1.6}$$

5. Rétropropager les sorties de F_2 vers F_1.
   - Calculer les entrées V_i de F_1 provenant de F_2. Chaque entrée est la somme des sorties de F_2 pondérées par les poids des connexions *top-down* appropriées.

$$V_i = \sum_{j=1}^{N} u_j z_{ij} = z_{iJ} \tag{1.7}$$

   L'indice J est celui du neurone gagnant à l'étape 4. Comme $0 \leq z_{iJ} \leq 1$ (éq. 1.1 et 1.12), $0 \leq V_i \leq 1$.
   - Calculer les nouvelles activités pour F_1, en tenant compte à la fois du vecteur d'entrée $\mathbf{I}$ et des entrées provenant de F_2.

$$x_{1i} = \frac{I_i + D_1 V_i - B_1}{1 + A_1(I_i + D_1 V_i) + C_1} \tag{1.8}$$

- Calculer les sorties s_i pour F_1.

$$s_i = \begin{cases} 1 & x_{1i} > 0 \\ 0 & x_{1i} \leq 0 \end{cases} \tag{1.9}$$

On appelle le vecteur **S** *top-down template*.

6. Déterminer la correspondance entre le *top-down template* et le vecteur d'entrée.

- Calculer le degré de correspondance.

$$\frac{|\mathbf{S}|}{|\mathbf{I}|} = \frac{\sum_{i=1}^{M} s_i}{\sum_{i=1}^{M} I_i} \tag{1.10}$$

$|\ |$ est la norme L_1 du vecteur, c'est-à-dire la somme de ses composantes. Comme les vecteurs utilisés sont binaires, ceci correspond au nombre de composantes non nulles du vecteur.

- Comparer le degré de correspondance avec la vigilance ρ.
  - Si $|\mathbf{S}|/|\mathbf{I}| < \rho$, inhiber le neurone v_J pour éviter qu'à l'étape 4 il gagne à nouveau et retourner à l'étape 3 avec le même vecteur d'entrée.
  - Si $|\mathbf{S}|/|\mathbf{I}| \geq \rho$, continuer.

7. Mettre à jour les poids des connexions reliées au neurone gagnant v_J.

- Poids *bottom-up*:

$$z_{Ji} = \begin{cases} \frac{L}{L-1+|\mathbf{S}|} & \text{si } v_i \text{ est actif} \\ 0 & \text{si } v_i \text{ est inactif} \end{cases} \tag{1.11}$$

- Poids *top-down*:

$$z_{iJ} = \begin{cases} 1 & \text{si } v_i \text{ est actif} \\ 0 & \text{si } v_i \text{ est inactif} \end{cases} \tag{1.12}$$

On se trouve donc à mémoriser le *top-down template* (voir étape 5) dans les poids *top-down*.

8. Réactiver tous les neurones de F_2 et répéter l'algorithme à partir de l'étape 3 avec un nouveau vecteur d'entrée.

1.1.3 Paramètres

Comme nous l'avons vu à la section précédente, il y a un bon nombre de paramètres dans les équations qui régissent le ART1. Nous verrons l'utilité de chacun et leur effet sur le fonctionnement du réseau.

- A_1, B_1, C_1 et D_1

  D'une part, les paramètres B_1 et D_1 sont utilisés pour initialiser les poids *top-down*, mais la valeur initiale n'a pas d'effet sur l'apprentissage en autant qu'on respecte l'inégalité d'apprentissage (éq. 1.1).

  D'autre part, on utilise les quatre paramètres A_1, B_1, C_1 et D_1 pour calculer les activités sur la couche F_1. Comme les activités x_{1i} ne servent qu'à déterminer les sorties s_i, il y a deux cas à distinguer:

$$\begin{cases} x_{1i} > 0 & \Rightarrow s_i = 1 \\ x_{1i} \leq 0 & \Rightarrow s_i = 0 \end{cases}$$

  Il y a deux formules utilisées pour le calcul des activités.

  1. Lors de la propagation du vecteur d'entrée vers F_1 (éq. 1.3):

$$x_{1i} = \frac{I_i}{1 + A_1(I_i + B_1) + C_1}$$

     Comme A_1, B_1, C_1 et D_1 sont > 0 et $I_i \in \{0, 1\}$,

$$s_i = \begin{cases} 1 & \text{si } I_i = 1 \quad (x_{1i} > 0) \\ 0 & \text{si } I_i = 0 \quad (x_{1i} = 0) \end{cases}$$

     Les paramètres A_1, B_1, C_1 et D_1 n'influencent pas la valeur de la sortie dans ce cas-ci.

  2. Lors de la rétropropagation des sorties de F_2 vers F_1 (éq. 1.8):

$$x_{1i} = \frac{I_i + D_1 V_i - B_1}{1 + A_1(I_i + D_1 V_i) + C_1}$$

     Comme A_1, B_1, C_1 et D_1 sont > 0, $I_i \in \{0, 1\}$ et $0 \leq V_i \leq 1$, le dénominateur de l'équation est ≥ 1. La valeur de la sortie dépend donc seulement du numérateur.

     Quand $I_i = 0$, $x_{1i} \leq 0$ car $B_1 > D_1$ (rappelons que B_1 doit respecter $\max\{D_1, 1\} < B_1 < D_1 + 1$).

Quand $I_i = 1$, $x_{1i} > 0$ si $V_i > \frac{B_1-1}{D_1}$.
En résumé,

$$s_i = \begin{cases} 1 & \text{si } I_i = 1 \text{ et } V_i > \frac{B_1-1}{D_1} & (x_{1i} > 0) \\ 0 & \text{sinon} & (x_{1i} \leq 0) \end{cases}$$

Or, $V_i = z_{iJ}$ (éq. 1.7) et il y a 2 cas:

- Le poids z_{iJ} n'a pas été modifié depuis son initialisation. Dans ce cas, $s_i = 1$ ssi $I_i = 1$ car nous savons que $V_i > \frac{B_1-1}{D_1}$.
- Le poids $z_{iJ} \in \{0, 1\}$ (éq. 1.12). Puisque $\frac{B_1-1}{D_1} \in]0, 1[$ et $V_i \in \{0, 1\}$, $s_i = 1$ ssi $V_i = 1$. Notons ici que la sortie s_i est la conjonction (ET booléen) entre l'entrée I_i et V_i provenant de F_2 ($s_i = I_i \wedge V_i$). En effet, $s_i = 1$ ssi $I_i = 1$ et $V_i = 1$.

Les paramètres A_1, B_1, C_1 et D_1 n'influencent donc pas la valeur de la sortie dans ce cas-ci non plus.

- Vigilance ρ

  Le paramètre ρ sert de critère pour déterminer si la classe (neurone gagnant v_J) choisie par le réseau est acceptée ou non (voir l'étape 6 de l'algorithme à la section 1.1.2).

  Plus ρ est *petit*, moins le réseau cherchera une autre classe lorsque le degré de correspondance entre **S** et **I** est petit. Les classes créées seront donc plus *grossières*.

  Plus ρ est *grand*, plus le réseau cherchera une autre classe même si le degré de correspondance est relativement grand. Les classes créées seront donc plus *précises*.

  Bref, il faut essayer de trouver un ρ qui divisera les données en un nombre de classes suffisant, sans qu'il soit trop grand.

- L

  Le paramètre L est utilisé lors de l'initialisation et du calcul des poids *bottom-up*.

  Au niveau de l'initialisation, L ne peut avoir d'effet en autant qu'il ait une valeur respectant l'inégalité d'accès direct (éq. 1.2). Par contre, le paramètre L peut avoir de l'influence au niveau du calcul des poids *bottom-up* (éq. 1.11).

  En effet, plus L est grand, plus la valeur du rapport $\frac{L}{L-1+|\mathbf{S}|}$ se rapproche de 1. Ceci amènera le réseau à faire plus de classes même si ρ est petit [1, section 25]. En influençant la valeur des poids *bottom-up*, L peut aussi affecter l'ordre dans lequel les neurones de F_2 sont activés. Nous garderons L petit pour éviter que le réseau crée trop de classes.

En résumé, ce sont seulement les paramètres ρ et L qui ont de l'effet sur l'apprentissage. Il faut préciser, par contre, que les autres paramètres ne sont pas négligeables dans le cas d'un système fonctionnant en temps réel, par exemple une réalisation du ART1 en circuit intégré.

1.1.4 Algorithme simplifié

Nous venons de prouver que seulement 2 paramètres, ρ et L, ont un effet sur l'apprentissage que fait le réseau. Nous tenterons dans cette section de voir s'il est possible d'éliminer les autres paramètres de façon à simplifier l'algorithme du ART1 mais surtout pour éviter d'avoir à fixer leur valeur inutilement.

Le calcul des sorties de la couche F_1 à l'étape 3 peut se réduire à

$$\mathbf{S} = \mathbf{I}$$

En effet, comme nous l'avons vu à la section 1.1.3,

$$s_i = \begin{cases} 1 & \text{si } I_i = 1 \\ 0 & \text{si } I_i = 0 \end{cases}$$

Nous pouvons également simplifier le calcul des sorties de F_1 lors de la rétro-propagation de F_2 vers F_1 (étape 5). Nous avons remarqué, à la section 1.1.3 que $\mathbf{S} = \mathbf{I} \wedge \mathbf{V}$ sauf si un des poids z_{iJ} n'a pas été modifié depuis son initialisation.

Or, la condition d'initialisation des poids z_{ij} est (éq. 1.1)

$$1 \geq z_{ij}(0) > \frac{B_1 - 1}{D_1}$$

Il serait donc possible d'initialiser les poids *top-down* z_{ij} à 1, ainsi nous aurions $s_i = I_i \wedge V_i$ dans tous les cas. Le calcul des sorties de la couche F_1 à l'étape 5 peut donc se réduire à

$$s_i = \begin{cases} 1 & \text{si } I_i = 1 \text{ et } V_i = 1 \\ 0 & \text{sinon} \end{cases}$$

Ceci est équivalent à

$$\mathbf{S} = \mathbf{I} \wedge \mathbf{V}$$

Nous initialiserons les poids *bottom-up* z_{ji} à

$$z_{ji}(0) = \frac{L}{L - 1 + M}$$

Ceci respecte l'inégalité d'accès direct (éq. 1.2).

Par ailleurs, les activités des neurones de F_1 ne servent maintenant plus à rien, mais nous les mettrons simplement égales aux sorties.

Équations

Voici donc le résumé des équations pour l'algorithme simplifié.

1. Initialiser les paramètres en respectant les contraintes suivantes:

$$0 < \rho \leq 1$$

$$L > 1$$

   Des valeurs pour ces paramètres sont suggérées un peu plus loin.

2. Initialiser les poids des connexions.

   - Poids *top-down* z_{ij} (reliant v_j à v_i):

$$z_{ij}(0) = 1 \tag{1.13}$$

   - Poids *bottom-up* z_{ji} (reliant v_i à v_j):

$$z_{ji}(0) = \frac{L}{L - 1 + M} \tag{1.14}$$

3. Appliquer un vecteur d'entrée $\mathbf{I}$ et le propager vers F_1. Calculer les activités $\mathbf{x}_1$ et les sorties $\mathbf{S}$ pour F_1.

$$\mathbf{x}_1 = \mathbf{S} = \mathbf{I} \tag{1.15}$$

4. Propager les sorties de F_1 vers F_2.

   - Calculer les activités T_j pour F_2.

$$T_j = \sum_{i=1}^{M} s_i z_{ji} \tag{1.16}$$

   - Calculer les sorties u_j pour F_2.

$$u_j = \begin{cases} 1 & T_j = \max_k \{T_k\} \forall k \\ 0 & \text{sinon} \end{cases} \tag{1.17}$$

5. Rétropropager les sorties de F_2 vers F_1.

   - Calculer les entrées V_i de F_1 provenant de F_2.

   $$V_i = z_{iJ} \tag{1.18}$$

   L'indice J est celui du neurone gagnant à l'étape 4.

   - Calculer les nouvelles activités et sorties pour F_1.

   $$\mathbf{x}_1 = \mathbf{S} = \mathbf{I} \wedge \mathbf{V} \tag{1.19}$$

6. Déterminer la correspondance entre le *top-down template* et le vecteur d'entrée.

   - Calculer le degré de correspondance.

   $$\frac{|\mathbf{S}|}{|\mathbf{I}|} = \frac{\sum_{i=1}^{M} s_i}{\sum_{i=1}^{M} I_i} \tag{1.20}$$

   - Comparer le degré de correspondance avec la vigilance ρ.
     - Si $|\mathbf{S}|/|\mathbf{I}| < \rho$, inhiber le neurone v_J pour éviter qu'il gagne à nouveau et retourner à l'étape 3 avec le même vecteur d'entrée.
     - Si $|\mathbf{S}|/|\mathbf{I}| \geq \rho$, continuer.

7. Mettre à jour les poids des connexions reliées au neurone gagnant v_J.

   - Poids *bottom-up*:

   $$z_{Ji} = \begin{cases} \frac{L}{L-1+|\mathbf{S}|} & \text{si } v_i \text{ est actif} \\ 0 & \text{si } v_i \text{ est inactif} \end{cases} \tag{1.21}$$

   - Poids *top-down*:

   $$z_{iJ} = \begin{cases} 1 & \text{si } v_i \text{ est actif} \\ 0 & \text{si } v_i \text{ est inactif} \end{cases} \tag{1.22}$$

8. Réactiver tous les neurones de F_2 et répéter l'algorithme à partir de l'étape 3 avec un nouveau vecteur d'entrée.

Valeurs suggérées pour les paramètres

Voici les valeurs suggérées pour les paramètres, accompagnées des contraintes à respecter pour chacun d'eux.

Paramètre	*Contrainte*	*Valeur suggérée*
L	$L > 1$	1.1
ρ	$0 < \rho \leq 1$	0.5

Pour le paramètre L, une valeur proche de 1 semble donner de meilleurs résultats qu'une valeur plus élevée. Par ailleurs, le réseau classifie généralement bien les données quand $0.4 \leq \rho \leq 0.6$. Débuter avec une valeur dans cet intervalle, puis ajuster ρ pour effectuer une meilleure classification en faisant des essais sur les données à classer.

Nous avons vu dans cette section la structure du réseau de neurones ART1 et son algorithme d'apprentissage. Par la suite nous avons étudié les différents paramètres qui le contrôlent, ce qui nous a fait voir que seulement 2 d'entre eux étaient nécessaires. Nous avons donc élaboré un algorithme simplifié et c'est lui que nous utiliserons à la section suivante pour illustrer le fonctionnement du ART1.

1.2 Fonctionnement

Pour aider à la visualisation de son fonctionnement, le réseau ART1 a été appliqué au classement de points dans un espace à deux dimensions. Le tout a été programmé en C++, et plus d'informations sur cette implantation sont données à la section 1.3.

Comme le ART1 n'accepte pas des nombres réels en entrée, les coordonnées X et Y des points sont converties en format binaire, selon la notation *stack interval* [4, 8] (voir section 1.3.3). Cette notation a l'avantage suivant sur le codage binaire habituel: plus deux nombres sont rapprochés, plus ceux-ci ont un codage similaire. De plus, elle permet d'extraire les frontières apprises par le réseau.

Les vecteurs de 0 et de 1 représentant les coordonnées X et Y sont ensuite concaténés pour former le vecteur d'entrée du ART1.

Nous verrons tout d'abord les fichiers de données auxquels nous réfèrerons au fil des explications. Par la suite, nous étudierons le fonctionnement du réseau à différents niveaux: l'effet de la vigilance, les frontières apprises, l'ordre de présentation des données, les régions d'attraction, les recouvrements et l'influence du para-

mètre L.

1.2.1 Fichiers de données

Nous nous servirons de 5 fichiers de données différents. Ils contiennent chacun 100 points qui sont représentés par des coordonnées XY et qui appartiennent chacun à une classe. Par contre, nous n'utilisons pas les informations concernant la classe de chaque point, étant donné que le ART1 apprend sans supervision.

Nous voyons les ensembles de données à la figure 1.2. Chaque point est représenté par un chiffre identifiant sa classe, positionné aux coordonnées de celui-ci.

Nous utilisons le ART1 configuré de la façon suggérée à la section 1.1.4 avec 320 neurones sur F_1 (M) et 10 neurones sur F_2 (N). Ceci signifie que chaque coordonnée d'un point est codée sur 160 composantes binaires (bits) et que le réseau pourra créer jusqu'à 10 classes.

Nous fournissons les données de chaque fichier une fois au réseau, ce que nous appellerons une passe d'entraînement (ou lecture). Les données sont présentées dans l'ordre qu'elles ont dans le fichier. Nous verrons à la section 1.2.4 l'influence de l'ordre de présentation des données. Les classements effectués par le ART1 sont illustrés à la figure 1.3.

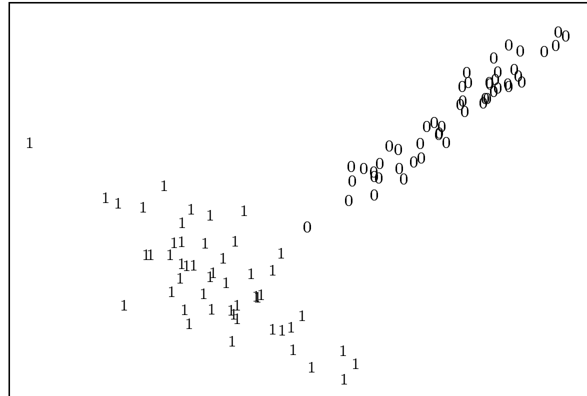
Le classement du fichier #1 est très bon, le réseau ayant visiblement regroupé ensemble les points qui étaient près les uns des autres. Il est à noter que la classe de chaque point n'est pas celle déterminée par le ART1 au moment où le point lui a été présenté, mais plutôt celle dans laquelle le point est classé *après* la lecture du fichier en entier. En effet, le réseau modifie son classement des données au fil de la lecture du fichier. Nous voyons donc sur la figure le classement de tous les points à un même moment[2].

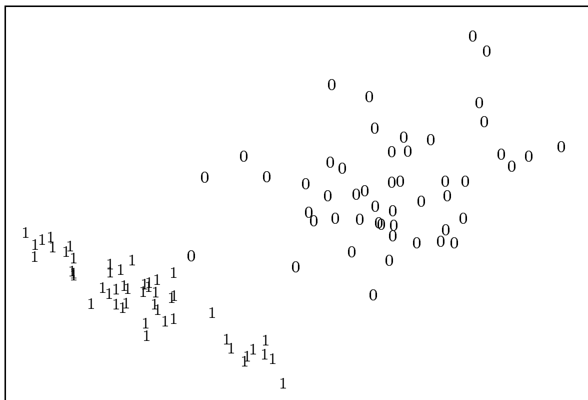
Lors des lectures subséquentes du fichier, le ART1 ne modifie pas son classement des données. Il se stabilise après une seule lecture, comme ce sera le cas pour les autres fichiers de données lorsque nous utilisons les paramètres par défaut.

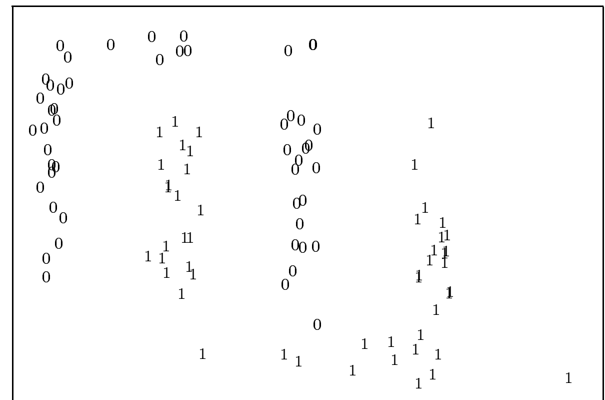
Dans le cas des données du deuxième fichier, le classement d'un point au bas vers la droite dans la classe #1 semble étrange, puisqu'il n'est pas du tout dans la même région que les autres points de cette classe. Ceci est dû à un phénomène

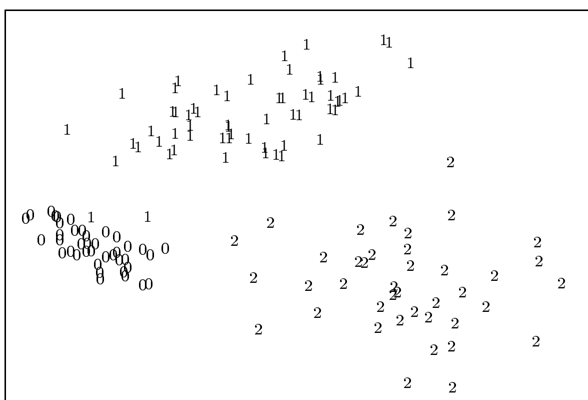
---

[2]Pour ce faire, nous présenté à nouveau toutes les données au réseau, en l'empêchant de modifier ses poids pour les apprendre.


#1


#2


#3


#4

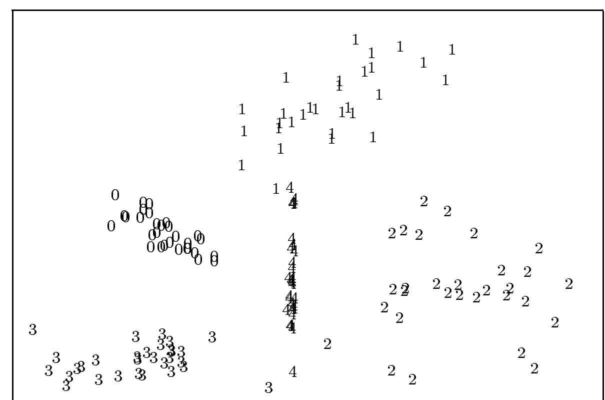
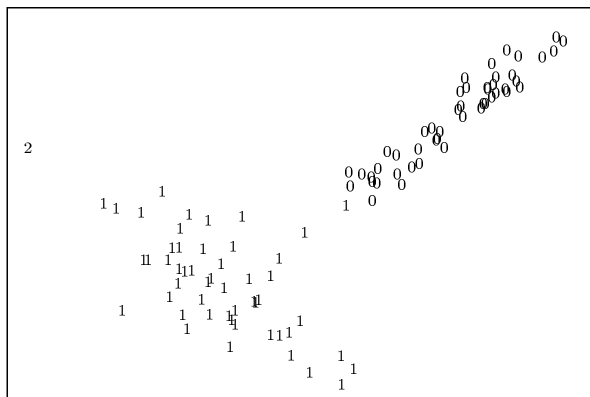
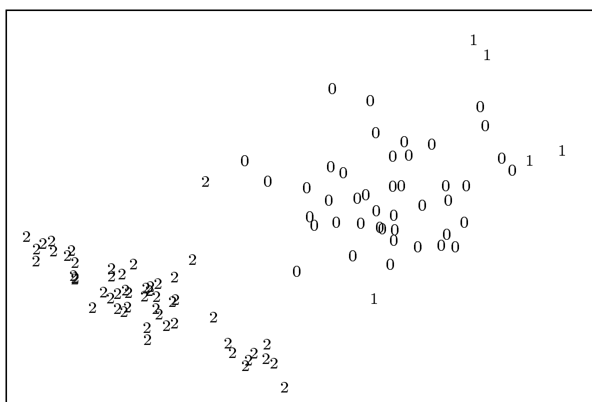
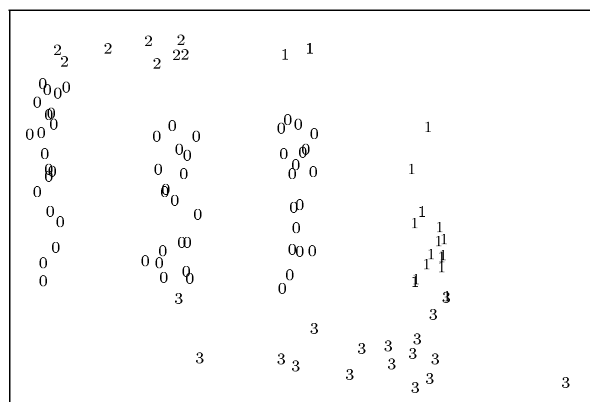
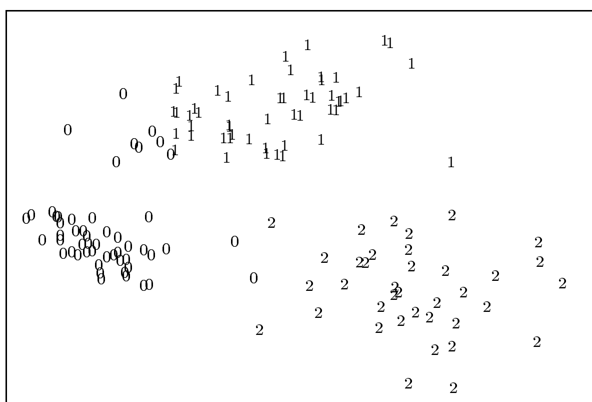

#5

Figure 1.2: *Fichiers de données.*


#1


#2


#3


#4

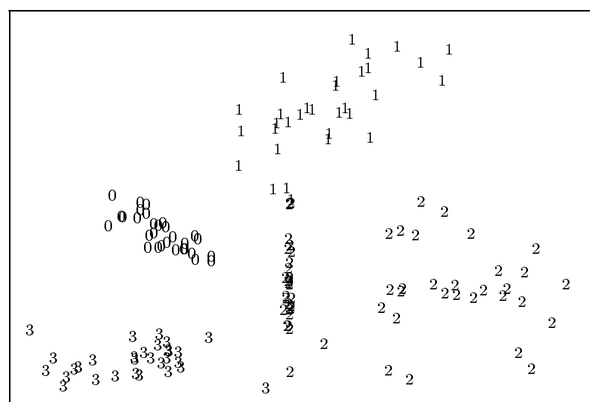

#5

Figure 1.3: *Classement des fichiers de données ($L = 1.1$ et $\rho = 0.5$).*

que nous expliquerons plus loin (voir section 1.2.6). Il est tout de même possible d'améliorer le classement en faisant varier la vigilance ρ, comme nous le verrons à la section 1.2.2.

Le fichier de données #3 contient des données séparées d'une façon que le ART1 ne pourra reproduire, les 2 classes étant mêlées l'une à l'autre. Puisqu'il ne sait pas à quelle classe appartiennent les points, il regroupe simplement ceux qui sont près les uns des autres.

Le quatrième fichier contient des données assez bien divisées en trois groupes différents. Le classement effectué par le réseau est assez bon, mais il peut certainement être amélioré en faisant varier les paramètres ou en changeant l'ordre de présentation des données.

Le fichier de données #5 nous montre que le ART1 n'est pas influencé par la densité des données. La classe #4 des données originales contient un grand nombre de points dans une faible surface, mais le réseau n'en fait pas une classe distincte.

1.2.2 Influence de la vigilance ρ

Nous venons de montrer le fonctionnement du ART1 sur différentes configurations de données, avec les paramètres suggérés à la section 1.1.4. Bien sûr, nous ne sommes pas assurés d'avoir le meilleur classement possible en utilisant ces paramètres. Voyons ce que changer la vigilance ρ a comme effet.

En fixant $\rho = 0.3$, les données #2 sont beaucoup mieux classées qu'avec $\rho = 0.5$. On voit effectivement à la figure 1.4 que le réseau a parfaitement bien identifié les deux groupes de points existants.

Par contre, en augmentant ρ à 0.75, le réseau forme un nombre plus grand de classes (fig. 1.5). Dans ce cas-ci, cette valeur ne donne pas de meilleurs résultats.

Pour les données #3, cependant, le classement est meilleur quand ρ passe de 0.5 (fig. 1.3) à 0.6 (fig. 1.6). La valeur optimale de ρ varie donc selon les données, c'est pourquoi il est important de faire des essais sur ce paramètre.

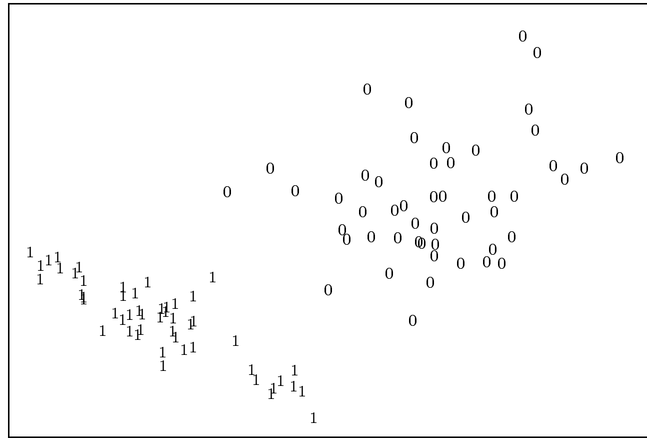


Figure 1.4: *Classement des données #2 ($L = 1.1$ et $\rho = 0.3$).*

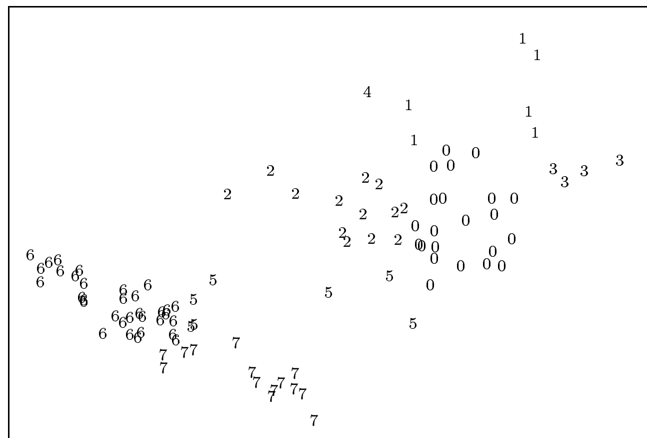


Figure 1.5: *Classement des données #2 ($L = 1.1$ et $\rho = 0.75$).*

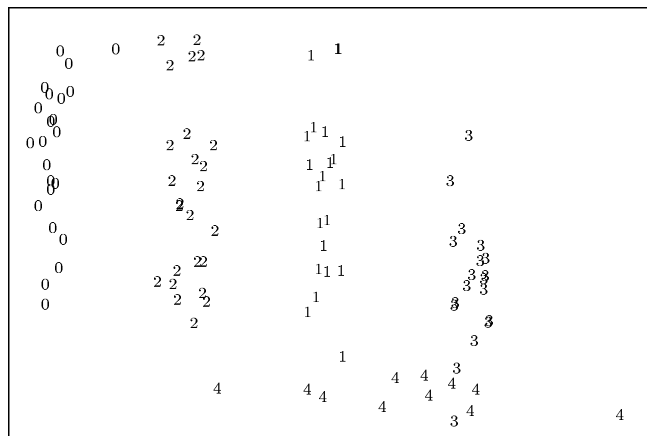


Figure 1.6: *Classement des données #3 ($L = 1.1$ et $\rho = 0.6$).*

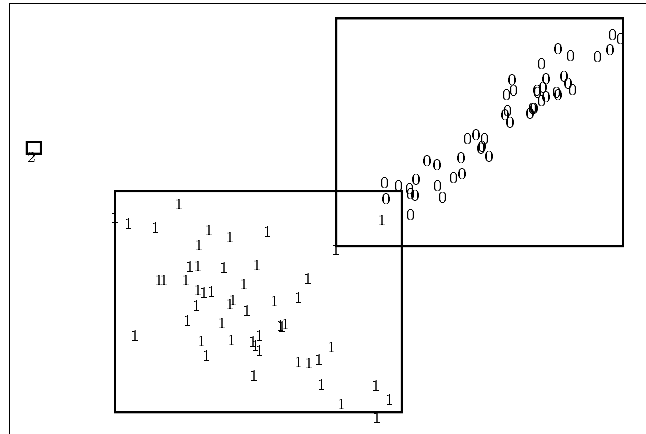


Figure 1.7: *Classement des données #1 avec les frontières ($L = 1.1$ et $\rho = 0.5$).*

1.2.3 Frontières apprises

Comme il a été mentionné au début de cette section, notre implantation du ART1 utilise la notation *stack interval* pour convertir des coordonnées réelles en vecteurs binaires pour le ART1. Cette notation est conçue de façon à permettre d'extraire les frontières que le réseau a apprises pour classer les données (plus de détails se retrouvent à la section 1.3.3).

Il est donc possible de visualiser quelles sont les frontières séparant chacune des classes formées par le réseau. Par exemple, pour le fichier de données #1, avec les paramètres par défaut, les frontières sont illustrées à la figure 1.7.

Ceci va nous permettre de voir de quelle façon le ART1 effectue son classement. Suivons son apprentissage étape par étape lorsqu'on lui donne une donnée à la fois. Quand on place le premier point, le réseau crée une boîte aussi petite que possible, selon le nombre de bits utilisés pour coder les coordonnées, qui inclut ce point (fig. 1.8a).

Lorsqu'on ajoute un point suffisamment près du premier, la boîte s'agrandit pour inclure le nouveau point (fig. 1.8b).

Si on place un autre point relativement loin des premiers, le réseau crée une nouvelle classe au lieu d'agrandir la classe existante (fig. 1.8c).

Chaque nouveau point est ajouté dans la classe la plus près, ou bien une nouvelle classe est créée si celui-ci est suffisamment éloigné des autres (fig. 1.8d à 1.8f).

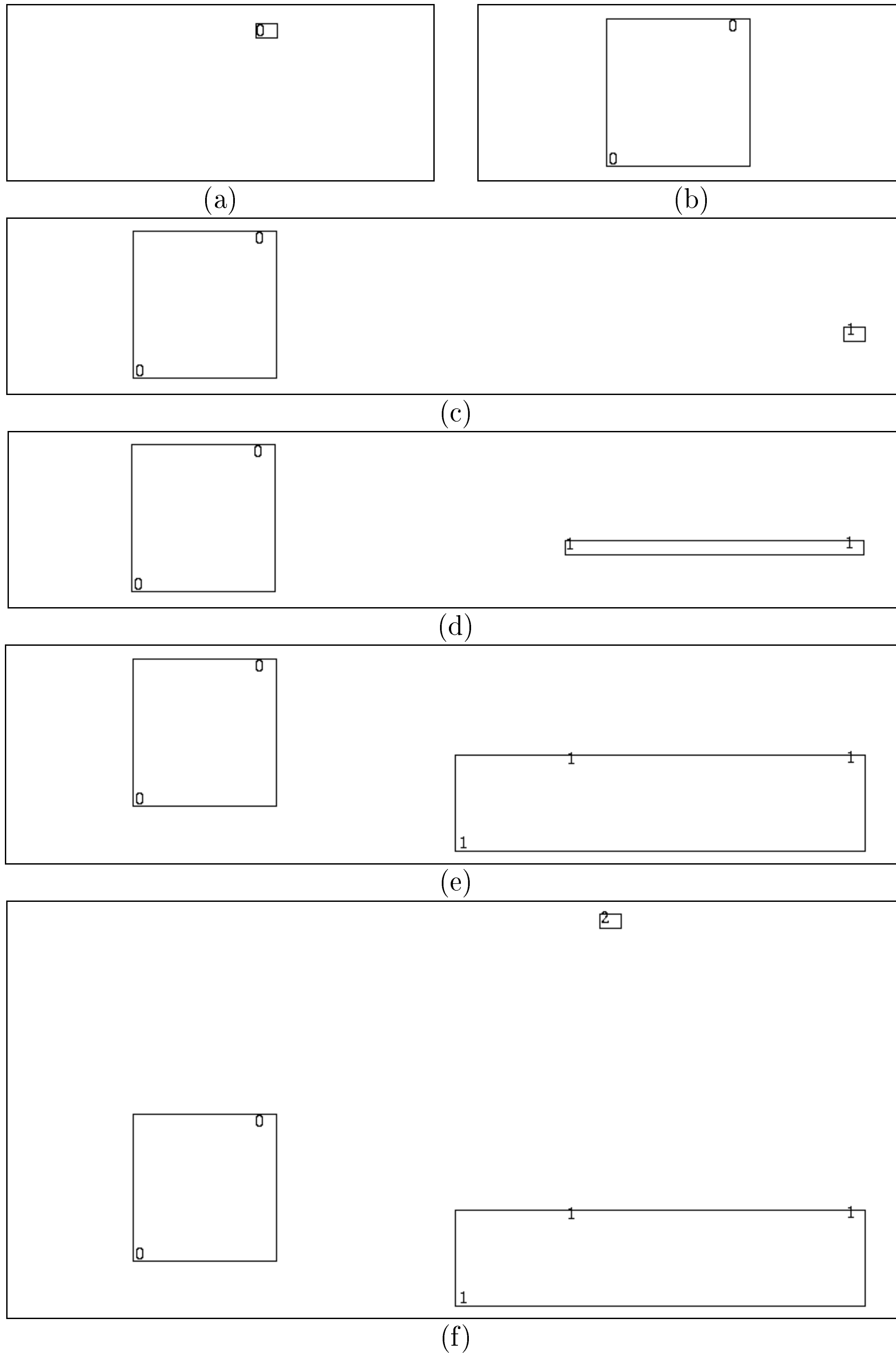


Figure 1.8: *Démonstration de l'apprentissage du ART1.*

La section 1.2.5 discute de la zone autour d'une frontière qui est considérée par le réseau comme suffisamment près pour agrandir la frontière (qu'on nomme la région d'attraction).

Il faut remarquer ici que les frontières que crée le ART1 ne rapetissent jamais. L'origine de ce phénomène est à l'étape 5 de l'algorithme d'apprentissage (section 1.1.4). Le *top-down template* **S**, qui représente une classe apprise par le réseau, est la conjonction entre le vecteur d'entrée **I** et le vecteur **V** provenant de F_2. Le vecteur **V** est en fait le *top-down template* correspondant au neurone gagnant à l'étape 4. Donc à mesure que le réseau apprend de nouveaux *top-down templates*, ceux-ci subissent une "érosion" graduelle, c'est-à-dire qu'ils contiennent de moins en moins de 1 à cause de leur conjonction avec les vecteurs d'entrée.

Avec la notation *stack interval*, ceci signifie que les extrémités des frontières sur chaque axe s'éloignent sans cesse et ne peuvent jamais se rapprocher (voir section 1.3.3). Ce phénomène a tendance à causer des recouvrements entre les frontières, dont nous discuterons à la section 1.2.6.

1.2.4 Ordre de présentation des données

La démonstration que nous avons faite nous permet aussi de voir que l'ordre de présentation des données est important. Par exemple, si le point ajouté à l'étape 5 avait été présenté à la place de celui à l'étape 2, il aurait été placé dans la classe 0 plutôt que dans la classe 1. Ceci aurait influencé le classement des points présentés subséquemment.

Les données de nos 5 fichiers sont énumérées dans un certain ordre. En général, les points qui sont groupés ensemble se suivent dans le fichier. Puisque l'ordre de présentation des données influence le travail du ART1, nous pourrions penser à permuter celles-ci pour voir de quelle façon se stabilisera le réseau dans cette situation.

Par exemple, prenons un fichier contenant une série de points répartis uniformément dans un plan. L'ensemble de ces points forme une grille de 20×20. En présentant les points au réseau dans cet ordre (avec $\rho = 0.4$ et $L = 1.1$), on obtient le classement de la figure 1.9.

Le ART1 a formé 5 rectangles qui divisent l'espace horizontalement. Ceci vient que les points sont ordonnés de bas en haut en allant de la gauche vers la droite. Au cours de la présentation des données, les rectangles s'agrandissent donc vers le

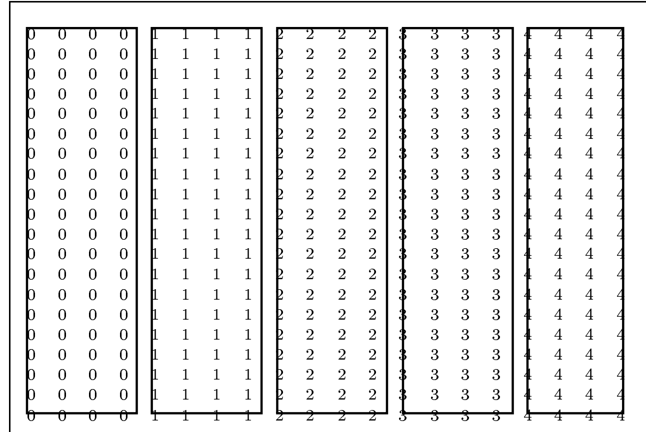


Figure 1.9: *Classement des données uniformes ($L = 1.1$ et $\rho = 0.4$).*

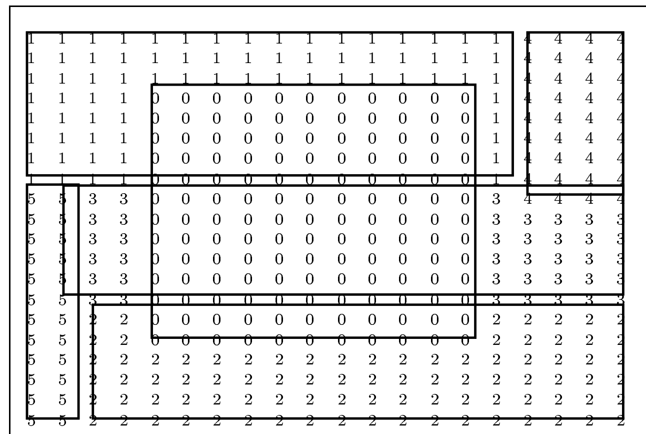


Figure 1.10: *Classement des données uniformes avec permutation aléatoire ($L = 1.1$ et $\rho = 0.4$).*

haut jusqu'à couvrir toute la hauteur, puis vers la gauche. De nouveaux rectangles sont créés à intervalles réguliers dépendant de la vigilance ρ.

Lorsqu'on permute aléatoirement les données avant de les présenter au ART1, le résultat, qu'on peut voir à la figure 1.10, est complètement différent. D'ailleurs, à chaque fois que nous recommençons et fournissons les données dans un nouvel ordre aléatoire, le résultat est différent.

L'effet de permuter les données aléatoirement ne semble pas particulièrement avantageux, si ce n'est que ceci ressemble davantage à ce que devrait en pratique recevoir un réseau ART1 lors de son utilisation.

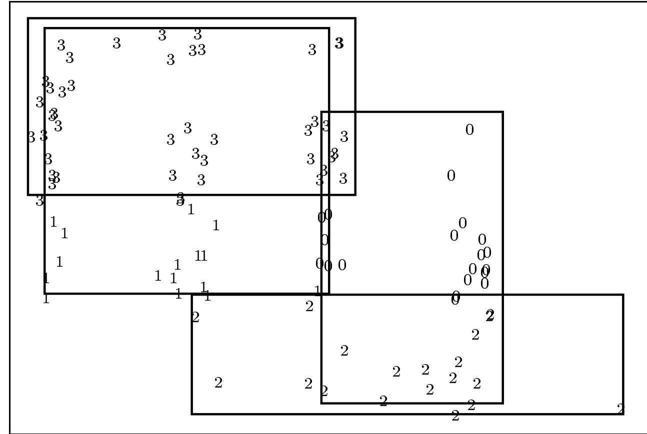


Figure 1.11: *Classement des données #3 avec permutation aléatoire ($L = 1.1$ et $\rho = 0.5$).*

Le fait de fournir l'un après l'autre au réseau des points qui sont quelque peu distancés fait tendre celui-ci à créer des frontières plus larges. Comme celles-ci ne peuvent pas se rapetisser, on augmente les chances qu'il y ait des recouvrements. En fait, il y a de fortes probabilités de créer de grandes boîtes pour les premières classes, qui seront ensuite recouvertes par les boîtes des autres classes. Une conséquence possible de ceci serait d'avoir des points appartenant à une même classe séparés par des points d'une ou plusieurs autres classes.

On voit qu'il y a plusieurs recoupements entre les rectangles et que la classe 3 est séparée en deux par la classe 0. En guise de comparaison avec nos résultats précédents, essayons de faire apprendre les données du fichier #3 au ART1 en ajoutant la permutation. Le résultat à la figure 1.11 est moins bon qu'auparavant (fig. 1.3). Bien sûr ce n'est qu'un ordre aléatoire parmi tant d'autres, mais en général on obtiendra des classements aussi peu intéressants et ceci est vrai pour tous les fichiers.

Par contre, il est possible d'obtenir de bons résultats en permutant les données particulièrement si on utilise un ρ assez élevé (> 0.5) et un L assez grand également. Nous voyons le résultat sur l'ensemble de données #5 à la figure 1.12. Ce classement est bon, malgré plusieurs recoupements entre les frontières des classes.

En pratique, la meilleure solution serait possiblement d'essayer de fournir au réseau, au début de son utilisation, des données qui sont similaires une à la suite de l'autre, pour que celui-ci crée de "bonnes" frontières qui se recoupent peu. Par la suite, les données fournies dans un ordre quelconque auront moins d'influence sur les frontières apprises.

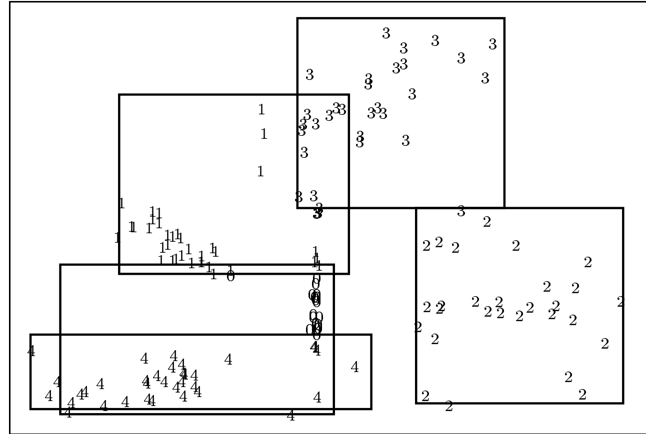


Figure 1.12: *Classement des données #5 avec permutation aléatoire ($L = 50$ et $\rho = 0.6$).*

1.2.5 Région d'attraction

Comme nous l'avons vu lors de la démonstration à la section 1.2.3, il y a une zone autour de la frontière d'une classe où le réseau est prêt à agrandir la frontière pour inclure un point qui y est placé. Nous appellerons cette zone la région d'attraction (*attractor region*) [4, 8].

À la figure 1.13, nous voyons l'allure de la région d'attraction autour d'une frontière seule dans un plan. Ceci a pu être fait en présentant au réseau une grille de 20×20 points couvrant le plan, tout en l'empêchant d'apprendre ces nouveaux points pour ne pas qu'il modifie ses frontières. Les points hors de la frontière qui sont identifiés par 0 sont donc ceux qui font partie de la région d'attraction et qui seraient inclus dans la classe 0 si le réseau avait à les apprendre. Ceux qui sont identifiés par 1 seraient par contre placés dans une nouvelle classe.

La région d'attraction autour d'une frontière diminue lorsque la surface de celle-ci augmente (ou le volume, dans le cas de plus de 2 dimensions). On constate ce phénomène aux figures 1.14 et 1.15. C'est pour cette raison que la frontière d'une classe a plus tendance à s'agrandir au début, lorsqu'elle est plus petite, que lorsqu'elle est devenue plus grande.

Ce phénomène s'explique par la notation *stack interval* (voir la section 1.3.3). Plus la frontière est grande, moins le *top-down template* $\mathbf{S}$ la représentant contient de 1, tandis que le vecteur d'entrée $\mathbf{I}$ en contient un nombre constant, ce qui fait diminuer le rapport $|\mathbf{S}|/|\mathbf{I}|$.

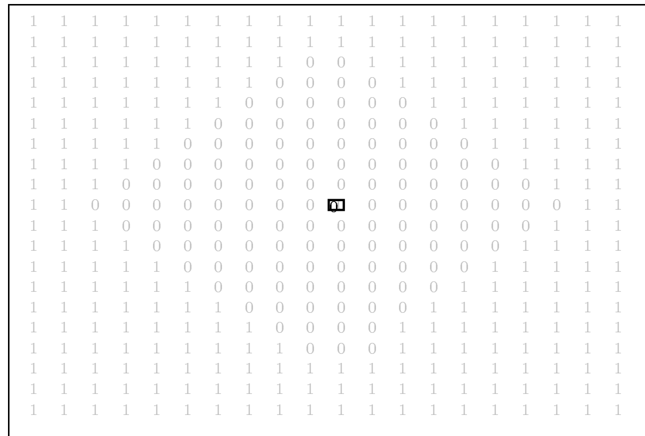


Figure 1.13: *Région d'attraction autour d'une petite frontière.*

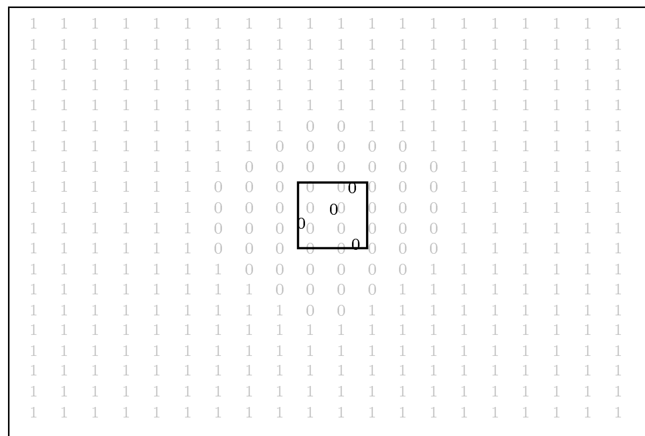


Figure 1.14: *Région d'attraction autour d'une frontière moyenne.*

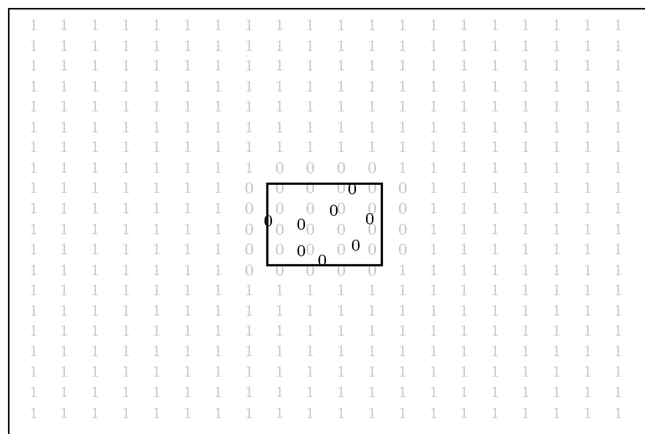


Figure 1.15: *Région d'attraction autour d'une grande frontière.*

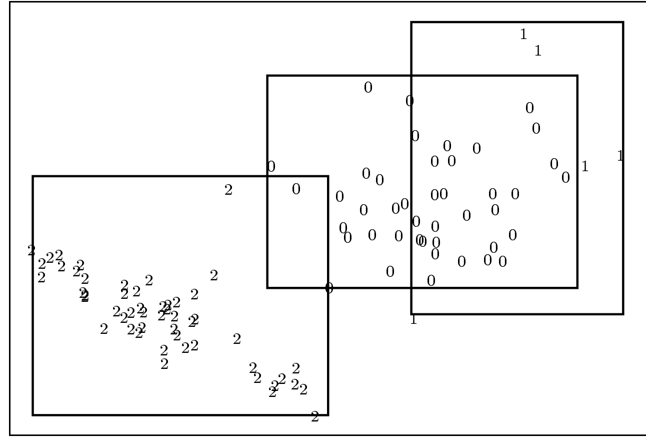


Figure 1.16: *Classement des données #2 avec les frontières ($L = 1.1$ et $\rho = 0.5$).*

1.2.6 Recouvrements

Les recouvrements entre les frontières des classes représentent une certaine incertitude de la part du ART1 quant au classement des points dans ces zones.

Il y a recouvrement entre deux classes lorsqu'une donnée qui activait un certain neurone sur F_2 en active maintenant un nouveau, suite à une modification des poids *bottom-up*. Il peut également y avoir recouvrement si une donnée génère la même activité sur 2 neurones ou plus. Les recouvrements peuvent causer des classements étranges, tels qu'une même classe séparée en deux par une autre classe.

Par exemple, c'est cela qui explique le classement des données du fichier #2 à la section 1.2.1, où la classe #1 semble contenir un point qui ne va pas avec les autres. Nous voyons les frontières que le réseau a formées à la figure 1.16. Dans la zone où se recouvrent les classes 0 et 1, les activités sur les neurones correspondants sont égales. Autrement dit, les points de cette zone n'appartiennent pas plus à une classe qu'à l'autre. S'ils sont identifiés comme étant dans la classe 0, c'est parce que dans notre algorithme nous prenons le premier neurone parmi ceux dont les activités sont égales. Il faudrait éventuellement songer à une autre façon de procéder dans une telle situation.

Notons ici que la classe gagnante dans une zone de recouvrement est celle dont le rectangle est le plus petit. En effet, par la notation *stack interval* (voir la section 1.3.3), plus un rectangle est petit, plus le vecteur correspondant contient de 1. Le rectangle associé à chaque neurone de F_2 est emmagasiné dans les poids des connexions top-down de ce neurone. Comme l'activité est la somme des composantes

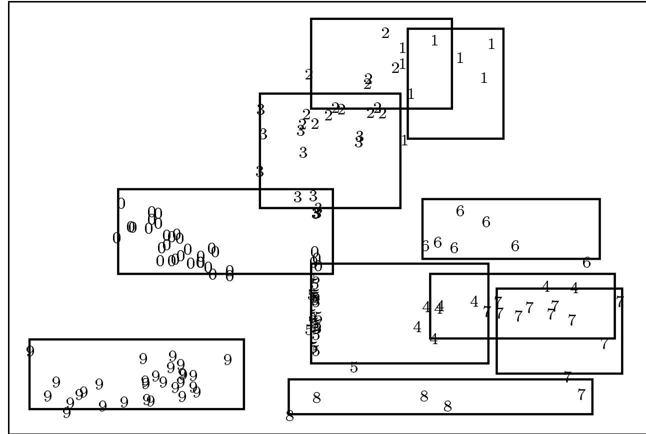


Figure 1.17: *Classement des données #5 ($L = 500$ et $\rho = 0.2$).*

du vecteur d'entrée pondérées par les poids des connexions *top-down* (éq. 1.16), le neurone gagnant, dont l'activité est la plus forte, est nécessairement celui associé au rectangle le plus petit.

1.2.7 Influence de L

Le paramètre L est celui dont l'effet est le plus difficile à saisir. Il ne faut cependant pas le négliger car il peut avoir beaucoup d'influence sur le réseau ART1.

Premièrement, notons que plus L est grand, plus le ART1 a tendance à créer de classes distinctes. Pour le fichier de données #2, par exemple, le réseau donne toujours le même résultat après une seule lecture sans permutation quand $L < 39.5$ (pour $\rho = 0.5$). Pour des valeurs plus grandes, le classement commencera à varier et il créera de plus en plus de classes.

Le même phénomène se produit avec n'importe quel ensemble de données. À la figure 1.17, on voit les données #5 classées avec $L = 500$ et $\rho = 0.2$. Par rapport au classement obtenu avec $L = 1.1$ et $\rho = 0.5$ (fig. 1.3), il y a beaucoup plus de classes même si ρ est plus petit.

Aussi, quand le paramètre L est très grand, l'effet de la vigilance ρ devient négligeable. Par exemple, pour les données #5 avec $L = 500$, le classement demeure celui de la figure 1.17 pour $\rho < 0.73$.

Lorsque L grandit, par ailleurs, le réseau atteint moins rapidement un état stable. Dans tous nos exemples jusqu'à présent, avec $L = 1.1$, le ART1 avait

un classement stable dès la première lecture complète du fichier. En augmentant la valeur de L, le réseau peut prendre plusieurs lectures avant de stabiliser son classement. Cela n'est pas nécessairement négatif, car cela signifie que les frontières ont tendance à mieux s'ajuster avant de rester stables.

Pour cette raison, il est possible qu'en choisissant L plus grand que 1.1, la valeur suggérée, le réseau fonctionne mieux sur un certain ensemble de données. Par contre, lorsque L est près de 1, le réseau crée moins de classes et il recherche plus en profondeur parmi les neurones utilisés avant d'en sélectionner un nouveau. Car le paramètre L influence également l'ordre d'inhibition des neurones lors de la présentation d'un vecteur d'entrée [1].

Cette section nous a permis de mieux comprendre le fonctionnement du ART1 en analysant le classement qu'il effectue avec des points dans un plan. Nous avons d'abord vu les résultats qu'a donnés le réseau avec les paramètres par défaut, puis nous avons étudié l'influence de la vigilance. Par la suite nous avons vu l'allure des frontières que crée un ART1 et nous avons fait des remarques sur l'effet de l'ordre de présentation des données. Une visualisation de la région d'attraction qui existe autour d'une frontière a suivi, ainsi qu'une discussion sur les recouvrements entre les frontières. Pour finir, nous avons regardé l'influence du paramètre L sur le réseau. Nous sommes maintenant prêts à passer à la réalisation concrète du ART1 en C++.

1.3 Implantation

Cette section porte sur l'implantation en C++ du réseau ART1. Nous verrons d'abord la classe `ART1` elle-même, puis les différents programmes qui l'utilisent, dont ceux qui ont servi à illustrer le fonctionnement du réseau à la section 1.2. Le tout est suivi des détails concernant la notation *stack interval* utilisée pour convertir des nombres réels en vecteurs binaires.

Le code source de cette implantation est disponible sur le réseau du département de génie électrique et de génie informatique de l'Université Laval, dans le répertoire `~busque00/vision/ART`. Les fichiers `.hpp` sont dans le sous-répertoire `include`, les fichiers `.cpp` dans `src`, les exécutables et les fichiers de données sont quant à eux dans le sous-répertoire `bin`.

1.3.1 Classe ART1

La classe `ART1` permet de gérer un réseau de neurones de type ART1 dont la structure est décrite à la section 1.1.1. Elle utilise l'algorithme simplifié de la section 1.1.4. La classe ART1 originale qui utilise tous les paramètres est conservée dans les fichiers `ART1orig.hpp` et `ART1orig.cpp`.

Le ART1 comprend deux matrices de poids (*top-down* et *bottom-up*), un vecteur d'entrée, et pour chacune des 2 couches, un vecteur d'activité et un vecteur de sortie. Il y a également un vecteur qui mémorise les inhibitions des neurones sur la couche F_2.[3]

Code source : `ART1.hpp` et `ART1.cpp`.

Constructeur

Le constructeur de la classe permet de créer la structure du réseau. Il possède deux paramètres :

```
ART1(int _dimF1=5, int _dimF2=6);
```

Le premier paramètre est la dimension de la couche F_1 et le second est la dimension de la couche F_2.

Les paramètres du réseau sont initialisés aux valeurs suivantes: $L = 1.1$ et $\rho = 0.5$. Ils peuvent être modifiés directement avec les fonctions `L` et `Vigilance`. On peut aussi les modifier en les lisant à partir d'un fichier à l'aide de la fonction `LireParam`.

Les poids *top-down* et *bottom-up* sont initialisés aux valeurs mentionnées à la section 1.1.4, page 12.

Opérateurs

La classe `ART1` possède les opérateurs suivants :

- `friend ostream &operator<<(ostream &_os, const ART1 &_art1);`

[3]Les classes `Vecteur` et `Matrice` utilisées sont celles créées par Marc Parizeau et sont disponibles à `www.gel.ulaval.ca/~parizeau/C++/classes/doc/Classes/Classes.html`.

Insère le réseau *_art1* dans le flot de sortie *_os* (voir `Ecrire`).

- `friend istream &operator>>(istream &_is, ART1 &_art1);`

  Extrait le réseau *_art1* dans le flot d'entrée *_is* (voir `Lire`).

Fonctions d'interface

La classe `ART1` possède les fonctions d'interface suivantes :

- `Vecteur ActivitesF1();`

  Retourne les activités sur la couche F_1.

- `Vecteur ActivitesF2();`

  Retourne les activités sur la couche F_2.

- `void Agrandissement(int _m=1);`

  Le réseau a un mode d'agrandissement automatique, actif par défaut, qui permet à la couche F_2 de s'agrandir si tous ses neurones sont utilisés. La fonction `Agrandissement` contrôle ce mode. Le paramètre *_m* doit être 0 ou 1, 0 signifiant que l'agrandissement automatique est désactivé.

- `void Apprentissage(int _m=1);`

  Permet de désactiver (ou d'activer) l'apprentissage du réseau, c'est-à-dire de l'empêcher (ou de lui permettre) de modifier les poids de ses connexions. Normalement l'apprentissage est actif, mais il peut être utile, dans certains cas, de le désactiver. Le paramètre *_m* doit être 0 ou 1, 0 signifiant que l'apprentissage est désactivé.

- `int Correspondance();`

  Vérifie s'il y a correspondance entre le vecteur d'entrée et le *top-down template*. Si le degré de correspondance est supérieur ou égal à la vigilance, la fonction retourne 1, sinon elle retourne 0.

- `void Dim(int _dimF1, int _dimF2);`

  Fixe les dimensions des couches F_1 et F_2. Les poids sont réinitialisés à leur valeur par défaut.

- `int DimF1();`

  Retourne la dimension de la couche F_1.

- `int DimF2();`

  Retourne la dimension de la couche F_2.

- `void DimF2(int _n);`

  Change le nombre de neurones sur la couche F_2. Cette fonction est normalement utilisée pour agrandir la couche quand tous les neurones sont utilisés. Les nouveaux poids dans les matrices agrandies sont initialisés à leur valeur par défaut. Ceux qui existent déjà demeurent intacts.

- `void Ecrire(ostream &_os=cout);`

  Écrit le réseau, y compris les paramètres, dans le flot de sortie *_os*. Par défaut, *cout* est employé.

- `void EnleverInhibitions();`

  Remet tous les neurones de la couche F_2 dans l'état non inhibé.

- `void Entrer(const Vecteur &_e);`

  Place le vecteur *_e* en entrée du réseau, sans le propager.

- `int GagnantF2();`

  Retourne l'indice du neurone gagnant sur la couche F_2.

- `void Inhiber(int _j);`

  Inhibe le neurone d'indice *_j* sur la couche F_2.

- `double L();`

  Retourne la valeur du paramètre L.

- `void L(double _l);`

  Fixe à *_l* la valeur du paramètre L. Les poids sont également réinitialisés (pour respecter l'équation 1.2).

- `void Lire(istream &_is=cin);`

  Lit le réseau à partir du flot d'entrée *_is*. Par défaut, *cin* est employé.

- `void LireParam(istream &_is=cin);`

  Lit les paramètres du réseau à partir du flot d'entrée *_is*. Par défaut, *cin* est employé. Le fichier peut contenir n'importe quel paramètre, inscrit sous la forme `nom: valeur`, où `nom` peut être `l` ou `rho` (par exemple `rho: 0.6`). Si un des paramètres n'est pas spécifié, il est initialisé à sa valeur par défaut. Les poids sont aussi réinitialisés à leur valeur par défaut.

- `void MettreAJour();`

  Met à jour les poids des connexions reliées au neurone gagnant sur la couche F_2. Cette fonction fait partie du processus de propagation et n'a généralement pas à être appelée individuellement.

- `void Propager(const Vecteur &_entree);`

  Place le vecteur *_entree* en entrée du réseau (il n'est pas nécessaire d'appeler la fonction `Entrer`) et effectue l'algorithme de propagation jusqu'à ce qu'il y ait correspondance. Lorsque le tout sera complété, il est possible de récupérer l'indice du neurone gagnant par la fonction `GagnantF2`.

- `void PropagerVersF1();`

  Propage le vecteur d'entrée vers la couche F_1. Cette fonction fait partie du processus de propagation et n'a généralement pas à être appelée individuellement.

- `void PropagerVersF2();`

  Propage les sorties de la couche F_1 vers la couche F_2. Cette fonction fait partie du processus de propagation et n'a généralement pas à être appelée individuellement.

- `void RetropropagerVersF1(int _gF2);`

  Rétropropage les sorties de la couche F_2 vers la couche F_1. Le neurone d'indice *_gF2* est fixé comme étant le neurone gagnant. Celui-ci est normalement déterminé lors de la propagation vers F_2 mais il peut être choisi "manuellement".

- `Vecteur SortiesF1();`

  Retourne les sorties de la couche F_1.

- `Vecteur SortiesF2();`

  Retourne les sorties de la couche F_2.

- `Vecteur Template(int _j);`

  Retourne le *top-down template* de la classe *_j*.

- `void Trace(int _mode=1);`

  Permet d'activer ou de désactiver le mode trace, qui fait afficher à l'écran des informations lors de la propagation d'un vecteur à travers le réseau. Par défaut le mode trace est inactif. Le paramètre *_m* doit être 0 ou 1, 0 signifiant que le mode trace est désactivé.

- `double Vigilance();`

  Retourne la valeur de la vigilance ρ.

- `void Vigilance(double _r);`

  Fixe à $_r$ la valeur de la vigilance ρ.

Exemple d'utilisation

Le programme suivant crée un réseau ART1 avec 25 entrées et 10 sorties, avec une vigilance de 0.8 ($L = 1.1$, la valeur par défaut). On crée un vecteur qui contient seulement des 1 et on le propage dans le réseau. L'indice du neurone gagnant sera affiché à l'écran.

```
#include "ART1.hpp"

int main() {
  ART1 res(25,10);
  res.Vigilance(0.8);
  Vecteur v(25);
  v.Initialiser(1);
  res.Propager(v);
  cout << "Le neurone " << res.GagnantF2() << " gagne!" << endl;
}
```

1.3.2 Programmes utilisant le ART1

Trois programmes utilisant la classe `ART1` ont été réalisés.

art	Programme de base pour le réseau ART1.
art_2D	Permet d'utiliser le ART1 sur des données dans un plan XY.
art_souris	Permet d'utiliser le ART1 pour classer des points cliqués par la souris.

Programme art

Le programme `art` est le programme de base pour le réseau ART1. Il lit dans un fichier une `Table1D` de `Vecteur`. Ces vecteurs d'entrées sont fournis un à la suite de l'autre à un réseau ART1.

Ce programme accepte plusieurs paramètres sur la ligne de commande :

-d *nom*	Nom du fichier contenant les données, `defaut.dat` par défaut.
-P	Permuter les données au lieu de les lire dans l'ordre.
-L *valeur*	Valeur du paramètre L, 1.1 par défaut.
-R *valeur*	Valeur de la vigilance ρ, 0.5 par défaut.
-p *nom*	Nom du fichier dans lequel lire les paramètres, une alternative à fournir ceux-ci sur la ligne de commande.
-n *valeur*	Nombre de sorties du réseau (dimension N de la couche F_2), 10 par défaut. Le nombre d'entrées (dimension M de la couche F_1) est fixé automatiquement en fonction du fichier de données.
-l *nom*	Nom du fichier dans lequel lire le réseau.
-e *nom*	Nom du fichier dans lequel écrire le réseau.
-r *nom*	Nom du fichier dans lequel lire et écrire le réseau.

Code source : `art.cpp`.

Programme art_2D

Le programme `art_2D` permet d'utiliser le ART1 sur des données dans un plan XY. Il lit dans un fichier une série de points enregistrés sous forme de table de table de vecteurs (`<Table1D<Table1D<Vecteur>>>`), chaque sous-table contenant les vecteurs qui appartiennent à la même classe. Ces points sont convertis en *stack interval* (voir section 1.3.3) et sont fournis un à la suite de l'autre à un réseau ART1.

Lorsque le fichier de données s'affiche dans une fenêtre, cliquer sur celle-ci pour que le ART1 démarre son apprentissage. Lorsque la lecture des données est complétée, le classement effectué par le réseau s'affiche dans une autre fenêtre. À ce moment, il y a trois options :

- Cliquer sur le bouton de gauche de la souris fait lire à nouveau les données au réseau.
- Cliquer sur le bouton du centre de la souris fait afficher les frontières apprises par le réseau.
- Cliquer sur le bouton de droite de la souris fait afficher une grille de points montrant comment le réseau classerait ces points dans son état actuel.

Ce programme accepte plusieurs paramètres sur la ligne de commande :

-d *nom*	Nom du fichier contenant les données, `data1.tp2` par défaut.
-P	Permuter les données (à chaque lecture) au lieu de les lire dans l'ordre.
-L *valeur*	Valeur du paramètre L, 1.1 par défaut.
-R *valeur*	Valeur de la vigilance ρ, 0.5 par défaut.
-p *nom*	Nom du fichier dans lequel lire les paramètres, une alternative à fournir ceux-ci sur la ligne de commande.
-m *valeur*	Nombre d'entrées du réseau (dimension M de la couche F_1), 320 par défaut.
-n *valeur*	Nombre de sorties du réseau (dimension N de la couche F_2), 10 par défaut.

Code source : `art_2D.cpp`.

Programme art_souris

Le programme `art_souris` permet d'utiliser le ART1 pour classer des points cliqués à l'écran par la souris. Les points sont convertis en *stack interval* (voir section 1.3.3) avant d'être fournis au réseau.

Lorsque la fenêtre s'affiche, il y a trois options :

- Cliquer sur le bouton de gauche de la souris fait lire un nouveau point au réseau.
- Cliquer sur le bouton du centre de la souris fait afficher les frontières apprises par le réseau.

- Cliquer sur le bouton de droite de la souris fait afficher une grille de points montrant comment le réseau classerait ces points dans son état actuel.

Ce programme accepte quelques paramètres sur la ligne de commande :

-L *valeur*	Valeur du paramètre L, 1.1 par défaut.
-R *valeur*	Valeur de la vigilance ρ, 0.5 par défaut.
-p *nom*	Nom du fichier dans lequel lire les paramètres, une alternative à fournir ceux-ci sur la ligne de commande.

Code source : `art_souris.cpp`.

1.3.3 Classe StackInterval

La classe `StackInterval` permet de convertir des nombres réels en notation *stack interval* [7, 8]. Cette notation représente un nombre par un vecteur binaire, avec la propriété que plus deux nombres sont rapprochés, plus leur notation est semblable. Ceci n'est pas le cas avec la notation binaire habituelle, ou deux nombres rapprochés n'ont pas nécessairement une notation semblable, par exemple 15 (01111) et 16 (10000). Le vecteur binaire pourra servir d'entrée pour un réseau ART1.

Pour convertir un nombre x en *stack interval*, il faut d'abord déterminer la longueur $2d$ du vecteur désiré, ainsi qu'une borne inférieure l et supérieure u pour les différents nombres qu'on aura à coder. Alors il y aura m composantes égales à 1 au début du vecteur, suivis de $(d - m)$ composantes égales à 0, où

$$m = \left\lceil \frac{x - l}{u - l} \cdot d \right\rceil$$

($\lceil r \rceil$ représente le plus petit entier supérieur ou égal à r)

On se trouve donc à graduer la position du nombre entre les bornes inférieure et supérieure, avec une résolution proportionnelle à la longueur du vecteur. Mais ceci n'est que la première moitié du vecteur.

La seconde moitié du vecteur est quant à elle le complément de la première, composante par composante. C'est ce qui va nous permettre éventuellement d'extraire les frontières apprises par le réseau.

En effet, le vecteur est composé d'un certain nombre de 1, suivis de d composantes égales à 0, puis de 1 pour le reste. Il y a donc toujours autant de 0 que de 1 dans le vecteur. Plus le nombre à convertir est petit, moins les 1 au début du vecteur sont nombreux. Par contre, plus le nombre est grand, moins il y a de 1 à la fin du vecteur. Rappelons-nous maintenant ceci: lorsque le ART1 apprend un vecteur qui lui est présenté, il crée un *top-down template* en effectuant une conjonction entre ce vecteur et l'ancien *top-down template* (voir section 1.1.4). Ce *template* subit ainsi une "érosion" graduelle qui élimine des 1 pour laisser de plus en plus de 0. Étant donné la notation *stack interval*, les 1 restants seront aux extrémités du *template*. Les 1 au début indiquent le plus petit nombre appris, tandis que les 1 à la fin indiquent le plus grand nombre appris. C'est ainsi que l'on peut extraire les frontières créées par le réseau ART1.

Par exemple, supposons un ART1 à 10 entrées qui représentent un nombre entre 0 et 5. Le *top-down template* de chaque neurone de la couche F_2 est initialisé à 1111111111. Si un neurone apprend le nombre 2, son *template* deviendra 1100000111, la représentation de ce nombre. Si le même neurone apprend le nombre 4, 1111000001, le *template* devient 1100000001, l'intersection des deux vecteurs. On peut extraire le plus petit nombre appris, 2, grâce aux deux 1 au début du vecteur, et le plus grand, 4, grâce au 1 à la fin.

Code source : `StackInterval.hpp`.

Constructeur

Le constructeur de la classe permet de créer le vecteur représentant un nombre en notation *stack interval* :

```
StackInterval(double _x, int _dd, double _l, double _u);
```

Les paramètres sont le nombre *_x* à convertir, la longueur totale *_dd* du vecteur (*_dd*$= 2d$) ainsi que les bornes inférieure *_l* et supérieure *_u*.

La classe `StackInterval` est dérivée de la classe `Vecteur` (voir section 1.3.1).

Opérateurs

La classe `StackInterval` ne surdéfinit aucun opérateur mais hérite cependant de tous les opérateurs de `Vecteur`.

Fonction d'interface

- `Vecteur Limites();`

  Retourne les limites représentées par le *stack interval*. La limite inférieure dépend du nombre de 1 au début du vecteur, tandis que la limite supérieure dépend du nombre de 1 à la fin du vecteur. La fonction retourne un vecteur à deux dimensions dont la première contient la limite inférieure et la seconde, la limite supérieure. Cette fonction est particulièrement utile pour extraire les frontières apprises par le ART1. Notons que si l'objet `StackInterval` n'a pas été modifié depuis sa construction, les limites inférieure et supérieure sont égales.

Exemple d'utilisation

L'exemple suivant convertit les nombres 15 et 16 en *stack interval*. La longueur des vecteurs est 20 et les bornes inférieure et supérieure sont 0 et 19.

```
#include "StackInterval.hpp"

int main() {
  StackInterval si1(15,20,0,19);
  StackInterval si2(16,20,0,19);
  cout << "15 : " << ~si1 << endl;
  cout << "16 : " << ~si2 << endl;
}
```

Le résultat est le suivant:

```
15 : (1,20)
       1  1  1  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  1  1

16 : (1,20)
       1  1  1  1  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  1
```

C'est ce qui complète la section sur l'implantation du ART1 en C++. Nous avons étudié la classe `ART1` elle-même, puis les différents programmes qui l'utilisent et qui ont pu servir pour faire les démonstrations à la section 1.2. Nous avons enfin vu comment la classe `StackInterval` permet de convertir des nombres réels en vecteurs binaires pouvant être utilisés par un réseau ART1.

Chapitre 2

Fuzzy ART

Le Fuzzy ART est un réseau de neurones introduit par Carpenter, Grossberg et Rosen en 1991 [3]. Il est une version modifiée du ART1, qui lui permet notamment d'accepter des entrées analogiques (nombres réels entre 0 et 1). Ce réseau demeure capable de recevoir des entrées binaires (0 ou 1) et de se comporter de la même façon qu'un ART1. Le Fuzzy ART a les mêmes caractéristiques que le ART1, soit la capacité d'apprendre de façon *continue* et *sans supervision*.

Nous verrons, pour commencer, un résumé de l'algorithme d'apprentissage du Fuzzy ART. Ensuite, comme nous l'avons fait pour le ART1, nous illustrerons son fonctionnement à l'aide d'une implantation de celui-ci en C++. Nous verrons finalement les détails de cette implantation à la dernière section du chapitre.

2.1 Algorithme d'apprentissage

L'algorithme d'apprentissage du Fuzzy ART est similaire à l'algorithme simplifié du ART1 (section 1.1.4). Comme nous allons le voir, les deux réseaux ont une structure presque identique. Le Fuzzy ART utilise cependant 3 paramètres au lieu de 2 dans le cas du ART1.

La structure générale du réseau sera d'abord présentée, suivie des équations gérant son fonctionnement. Finalement, nous analyserons les différents paramètres du Fuzzy ART.

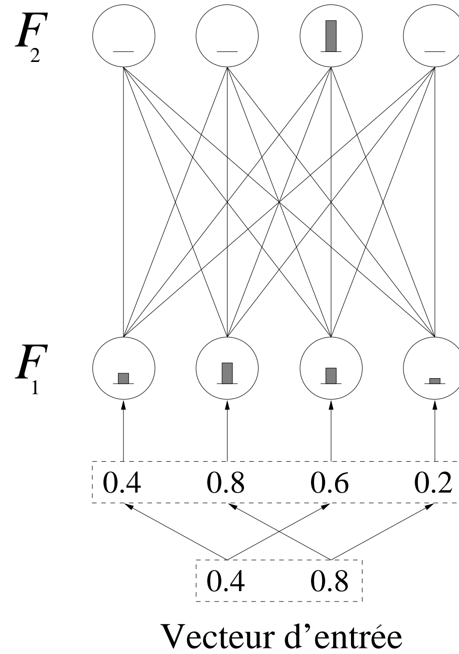


Figure 2.1: *Schéma du réseau Fuzzy ART.*

2.1.1 Structure du réseau

Le Fuzzy ART a presque la même structure que le ART1 (voir la section 1.1.1). Les sorties des neurones ont été éliminées, et ceux-ci n'ont seulement qu'une activité, qui tient aussi le rôle de sortie. L'autre différence est qu'il y a seulement une série de connexions entre les deux couches de neurones, contrairement à deux dans le ART1 (*top-down* et *bottom-up*).

Le réseau effectue un codage complémentaire (*complement coding*) sur les entrées qu'il reçoit. Ce codage est une forme de normalisation qui permet d'éviter la prolifération inutile des catégories. L'opération consiste à prendre le vecteur d'entrée et à le concaténer avec son complément (voir l'étape 3 de l'algorithme). C'est le vecteur résultant qui est présenté à la couche F_1. Celle-ci a donc le double de la dimension d'un vecteur d'entrée. Un schéma du réseau est présenté à la figure 2.1.

Il est possible de désactiver ce codage complémentaire. À ce moment, la couche F_1 du réseau aura la même dimension que le vecteur d'entrée. Par contre, il y a danger, tel que mentionné, de prolifération des catégories, comme démontré dans l'article de Carpenter, Grossberg et Rosen [3].

2.1.2 Équations

Voici un résumé de l'algorithme d'apprentissage du Fuzzy ART avec les équations correspondantes. Les neurones sur F_1 sont identifiés par v_i et ceux sur F_2 par v_j. L'indice i réfère toujours à la couche F_1 et j à la couche F_2. M et N représentent respectivement la dimension de F_1 et de F_2.

Il y a trois paramètres qui sont utilisés: le paramètre de sélection α, le taux d'apprentissage β (similaire au L du ART1) et la vigilance ρ. Plus de détails sur chacun de ces paramètres sont donnés à la section 2.1.3.

1. Initialiser les paramètres en respectant les contraintes suivantes:

$$\alpha > 0$$
$$\beta \in [0, 1]$$
$$\rho \in [0, 1]$$

2. Initialiser les poids des connexions.

$$w_{ji}(0) = 1 \tag{2.1}$$

3. Appliquer un vecteur d'entrée $\mathbf{a}$ et le propager vers F_1.

   - Effectuer le codage complémentaire du vecteur d'entrée[1]).

   $$\mathbf{I} = (\mathbf{a}, \mathbf{a}^c) \equiv (a_1, \ldots, a_M, a_1^c, \ldots, a_M^c) \tag{2.2}$$

   où $a_i^c \equiv 1 - a_i$.

   - Calculer les activités $\mathbf{x}$ (qui sont aussi les sorties) pour F_1.

   $$\mathbf{x} = \mathbf{I} \tag{2.3}$$

4. Propager les sorties de F_1 vers F_2.

   - Calculer les fonctions de sélection T_j pour F_2.

   $$T_j = \frac{|\mathbf{x} \wedge \mathbf{w}_j|}{\alpha + |\mathbf{w}_j|} \tag{2.4}$$

   Ici $|\ |$ est la norme L_1 du vecteur, c'est-à-dire la somme de ses composantes ($|\mathbf{p}| \equiv \sum_{i=1}^{M} p_i$) et l'opérateur $\wedge$ correspond au ET de la logique floue ($(\mathbf{p} \wedge \mathbf{q})_i \equiv \min(p_i, q_i)$). Le vecteur $\mathbf{w}_j$ est le vecteur des poids des connexions reliées au neurone j de F_2.

[1] Si on n'utilise pas le codage complémentaire, prendre directement $\mathbf{I} = \mathbf{a}$.

- Calculer les activités y_j pour F_2.

$$y_j = \begin{cases} 1 & T_j = \max_k \{T_k\} \forall k \\ 0 & \text{sinon} \end{cases} \tag{2.5}$$

L'indice J est celui du neurone gagnant à l'étape 4.

5. Rétropropager les sorties de F_2 vers F_1. Calculer les nouvelles activités pour F_1.

$$\mathbf{x} = \mathbf{I} \wedge \mathbf{w}_J \tag{2.6}$$

6. Déterminer la correspondance entre le vecteur d'activité $\mathbf{x}$ et le vecteur $\mathbf{I}$.

   - Calculer le degré de correspondance.

$$\frac{|\mathbf{x}|}{|\mathbf{I}|} = \frac{\sum_{i=1}^{M} x_i}{\sum_{i=1}^{M} I_i} \tag{2.7}$$

   - Comparer le degré de correspondance avec la vigilance ρ.
     – Si $|\mathbf{x}|/|\mathbf{I}| < \rho$, inhiber le neurone v_J pour éviter qu'il gagne à nouveau et retourner à l'étape 3 avec le même vecteur d'entrée.
     – Si $|\mathbf{x}|/|\mathbf{I}| \geq \rho$, continuer.

7. Mettre à jour les poids des connexions reliées au neurone gagnant v_J.

$$\mathbf{w}_J = \beta\mathbf{x} + (1 - \beta)\mathbf{w}_J \tag{2.8}$$

   Si $\beta = 1$ le réseau est en *fast learning*.

8. Réactiver tous les neurones de F_2 et répéter l'algorithme à partir de l'étape 3 avec un nouveau vecteur d'entrée.

2.1.3 Paramètres

Voyons maintenant l'utilité de chaque paramètre et leur effet sur le fonctionnement du réseau.

- Paramètre de sélection α

  Le paramètre de sélection α est utilisé dans le calcul des fonctions de sélection (éq. 2.4). Il équivaut à $L - 1$ du ART1 [4]. Lorsque $\alpha \approx 0$, le recodage (neurone gagnant différent pour un même vecteur d'entrée) est minimisé lors de l'apprentissage. Aussi, quand α augmente, le réseau a tendance à créer plus de classes.

- Taux d'apprentissage β

  Le taux d'apprentissage β est utilisé lors de la mise à jour des poids des connexions (éq. 2.8). Il influence la vitesse à laquelle les poids du réseau sont modifiés lors de l'apprentissage. Plus β est petit, plus les poids varient lentement. À la limite, si $\beta = 0$, l'équation devient $\mathbf{w}_J = \mathbf{w}_J$ et les poids ne varieront jamais. Par contre, si $\beta = 1$, le réseau est en *fast learning*, comme le ART1, car $\mathbf{w}_J = \mathbf{x}$.

- Vigilance ρ

  La vigilance ρ a la même utilité que dans le ART1 (voir la section 1.1.3, page 10). Elle sert de critère pour déterminer si la classe (neurone gagnant v_J) choisie par le réseau est acceptée ou non (voir l'étape 6 de l'algorithme).

  Plus ρ est petit, plus les classes créées sont grossières, tandis que plus ρ est grand, plus les classes créées sont précises.

Valeurs suggérées pour les paramètres

Voici les valeurs suggérées pour les paramètres, accompagnées des contraintes à respecter pour chacun d'eux.

Paramètre	*Contrainte*	*Valeur suggérée*
α	$\alpha > 0$	0.01
β	$\beta \in [0,1]$	1
ρ	$\rho \in [0,1]$	0.5

Le paramètre α doit être très petit pour minimiser le recodage et éviter un trop grand nombre de classes. Carpenter et Grossberg [2] utilisent $\alpha \geq 0.001$.

Quant au taux d'apprentissage, utiliser $\beta = 1$ pour que l'apprentissage soit instantané (*fast learning*). Par contre, il peut être avantageux de diminuer β pour que les poids varient plus lentement.

La vigilance ρ, enfin, a exactement le même effet que pour le ART1 (voir la section 1.1.3).

Nous avons vu dans cette section la structure du réseau de neurones Fuzzy ART et son algorithme d'apprentissage. Nous avons aussi étudié les différents paramètres qui le contrôlent. Nous sommes maintenant prêts à illustrer son fonctionnement, ce que nous ferons à la section suivante.

2.2 Fonctionnement

Comme nous l'avons fait pour le ART1, nous appliquerons le réseau Fuzzy ART au classement de points dans un espace à deux dimensions, ce qui aide à mieux visualiser son fonctionnement. Ceci a été fait à l'aide de programmes en C++, sur lesquels plus d'informations sont données à la section 2.3.

Le Fuzzy ART a deux entrées, représentant chacune des coordonnées X et Y. Bien sûr, les coordonnées ont d'abord été normalisées entre 0 et 1. Le codage complémentaire qu'effectue le réseau nous permet d'extraire les frontières apprises par celui-ci.

En effet, quand un vecteur d'entrée est appris, les poids des connexions du neurone gagnant sont modifiés de façon à tendre vers le vecteur d'activité de la couche F_1 (éq. 2.8). Or, les activités sont le ET flou entre le vecteur d'entrée en codage complémentaire $\mathbf{I}$ et les anciens poids du neurone (éq. 2.6). Les valeurs des poids diminuent donc toujours au cours de l'apprentissage. Comme les deux premières composantes du vecteur $\mathbf{I}$ sont les coordonnées du point, le réseau mémorisera dans les deux premiers poids d'un neurone de F_2 le point minimum appris par celui-ci. Le point maximum sera emmagasiné dans les deux autres poids, grâce au codage complémentaire. Il s'agit simplement de soustraire leur valeur de 1 pour obtenir les coordonnées du point maximum appris par ce neurone.

C'est de cette façon qu'on obtient les rectangles représentant les frontières apprises par le Fuzzy ART. Notons ici qu'il est possible, si $\beta < 1$ et que le réseau n'est pas encore stabilisé, que les extrémités d'une frontière soient inversées dans les poids, i.e. qu'on retrouve le point minimum dans les 2 derniers poids et le point maximum dans les deux premiers.

Nous comparerons d'abord le fonctionnement du Fuzzy ART avec celui du ART1, puis nous verrons l'influence des deux paramètres du Fuzzy ART qui n'existent pas dans le ART1, le taux d'apprentissage β et le paramètre de sélection α.

2.2.1 Comparaison avec le ART1

Le Fuzzy ART se comporte de façon similaire au ART1 lorsque le taux d'apprentissage $\beta = 1$ (*fast learning*). La discussion effectuée à la section 1.2 est donc aussi valide à peu de choses près, notamment au niveau de la vigilance, des frontières apprises, de l'ordre de présentation des données, de la région d'attraction et des recouvrements. Comme nous ne fournissons pas les données sous la même forme

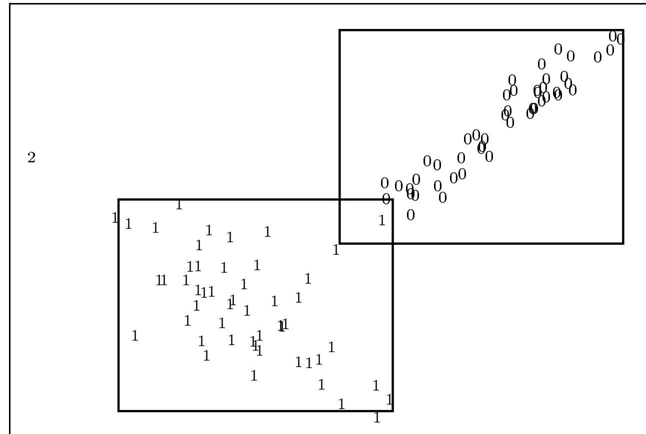


Figure 2.2: *Classement des données #1* ($\alpha = 0.01$, $\beta = 1$ *et* $\rho = 0.45$).

pour les deux réseaux, il faut réaliser que les classements de nos fichiers de données seront quelque peu différents avec le ART1 et le Fuzzy ART. Il n'en demeure pas moins que les principes de base demeurent valides.

Le ART1 crée toutefois des frontières dont la précision dépend du nombre de composantes du vecteur d'entrée codé en *stack interval*, tandis que le Fuzzy ART permet une résolution pratiquement infinie. Donc, si celui-ci crée une classe qui contient un seul point, la frontière ne sera pas un carré encadrant celle-ci, mais bien le point lui-même. On peut remarquer la petite différence entre les frontières du Fuzzy ART sur les données #1, à la figure 2.2, et celles du ART1, à la figure 1.7 (p. 21). Notons ici qu'il a fallu prendre $\rho = 0.45$ pour obtenir ce classement avec le Fuzzy ART.

Remarquons aussi que lorsqu'il y a un recouvrement entre deux frontières, la classe choisie est celle dont le rectangle est le plus petit, comme pour le ART1 (voir la section 1.2.6). La raison de cela est différente, toutefois. À cause du codage complémentaire, plus un rectangle est petit, plus ses composantes sont grandes. La fonction de sélection, par l'équation 2.4 (p.43), sera donc la plus grande pour le rectangle le plus petit.

L'avantage majeur du Fuzzy ART sur le ART1 est, bien sûr, d'accepter des entrées continues entre 0 et 1. Il est possible de coder des nombres réels en *stack interval* pour pouvoir les fournir au ART1, cependant ceci implique des vecteurs de dimension beaucoup plus grande. Il y a donc un énorme gain en temps de traitement lorsqu'on utilise le Fuzzy ART, en plus d'une résolution très précise.

En fait, le Fuzzy ART ne semble avoir aucun inconvénient face au ART1, puis-

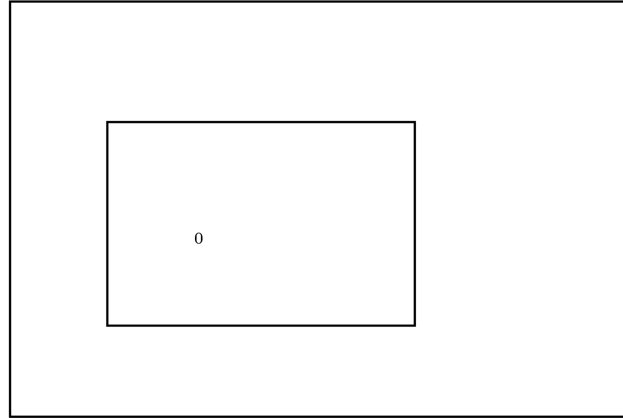


Figure 2.3: *Frontière autour d'un point quand* $\beta = 0.5$.

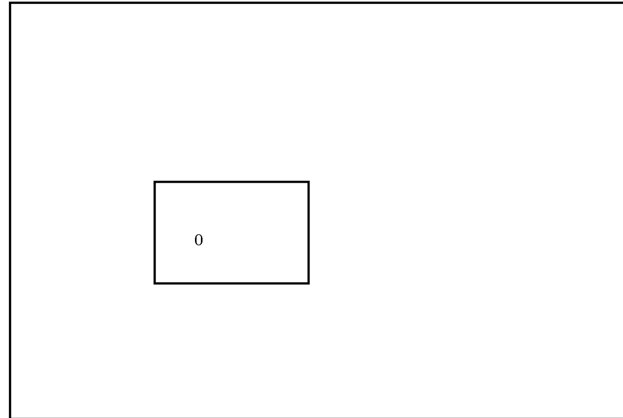


Figure 2.4: *Frontière autour d'un point quand* $\beta = 0.5$*, après une seconde présentation du point.*

qu'en plus d'accepter des entrées analogiques, il se comporte exactement comme le ART1 lorsqu'il reçoit des entrées binaires (si le codage complémentaire est désactivé, car le ART1 n'effectue pas ce codage[2]).

2.2.2 Influence du taux d'apprentissage β

Il est possible de contrôler le taux d'apprentissage du Fuzzy ART grâce au paramètre β. Concrètement, l'influence du paramètre β est la suivante: plus β est petit, plus les poids varient lentement.

[2]Cependant la notation *stack interval* était une forme de codage complémentaire

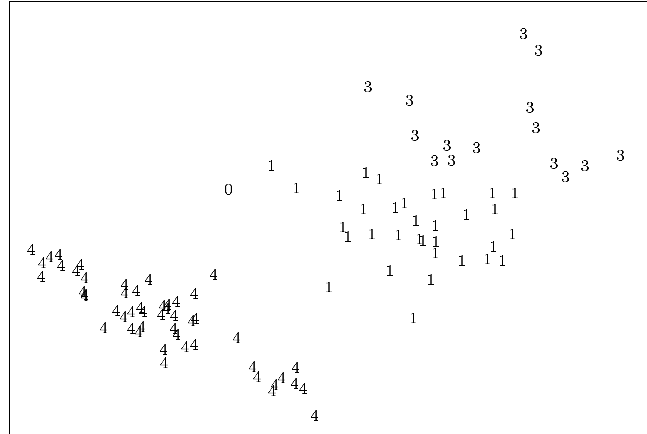


Figure 2.5: *Classement des données #2 avec permutation aléatoire* ($\alpha = 0.01$, $\beta = 0.1$ *et* $\rho = 0.5$*).*

Par exemple, lorsque $\beta = 0.5$ et qu'on présente un premier point à un réseau Fuzzy ART, la frontière de la première classe (0) sera à mi-chemin entre le point et le carré ayant comme coins opposés les points (0,0) et (1,1), étant donné que les poids sont initialisés à 1 (fig. 2.3).

Lorsqu'on représente le même point au réseau, la frontière se rapproche encore de celui-ci (fig. 2.4). À long terme, la frontière finira donc par coincider avec le point, et ceci est toujours vrai, peu importe le taux d'apprentissage.

Comme les frontières s'ajustent plus lentement lorsque le taux d'apprentissage est plus petit que 1, le classement effectué par le réseau en sera modifié. Il n'est toutefois pas évident de dire s'il est amélioré ou non. Cela dépend des cas. Par contre, le réseau classerait probablement mieux des données bruitées lorsque $\beta < 1$.

Il pourrait sembler intéressant de présenter les données dans un ordre différent à chaque passe d'entraînement en utilisant $\beta < 1$. Cependant les résultats obtenus sur nos fichiers ne sont pas exceptionnels non plus, même s'il peut y avoir de meilleurs cas comme celui illustré à la figure 2.5.

2.2.3 Influence du paramètre de choix α

Le phénomène constaté lorsqu'on augmente la valeur du paramètre de choix α est l'augmentation du nombre de classes crées par le réseau. Un peu comme le paramètre L du ART1 (section 1.2.7, p.29), plus le paramètre est grand, plus il

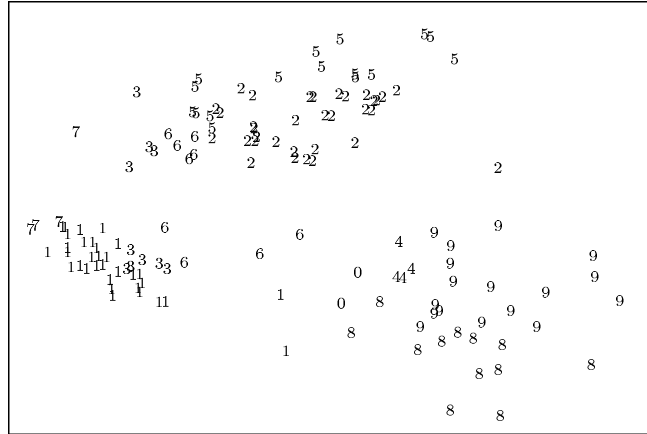


Figure 2.6: *Classement des données #4 ($\alpha = 7$, $\beta = 1$ et $\rho = 0.5$).*

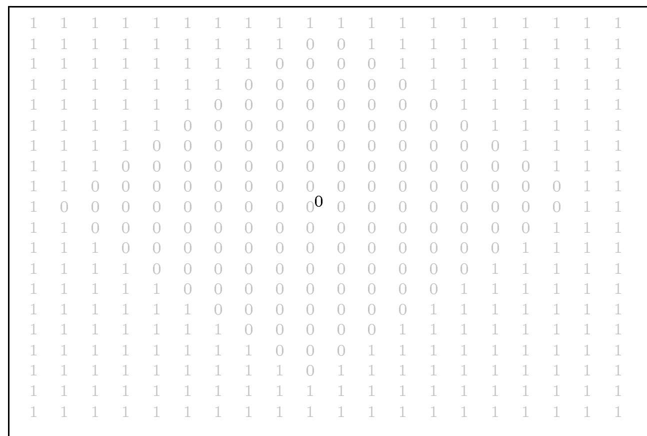


Figure 2.7: *Région d'attraction autour d'un point avec $\alpha = 5$.*

y a prolifération du nombre de classes. Par exemple, nous voyons le classement des données #4 avec $\alpha = 7$ à la figure 2.6. Un total de 10 classes ont été créées, contre 3 avec la valeur par défaut ($\alpha = 0.01$).

Si le réseau crée plus de classes quand α est grand, c'est que la région d'attraction (voir la section 1.2.5, p.26) autour des frontières des classes diminue. Nous voyons la différence entre $\alpha = 5$ et $\alpha = 50$ aux figures 2.7 et 2.8.

Aussi, il peut arriver, à la fin d'une passe d'entraînement, que des points à l'intérieur d'une frontière activent un autre neurone que celui qui est associé à celle-ci. Le neurone activé est toujours d'indice plus élevé que le neurone associé à la frontière. C'est qu'il s'est créé une nouvelle classe dont le neurone a une fonction de sélection plus élevée, pour ces points, que celle du neurone qui gagnait auparavant

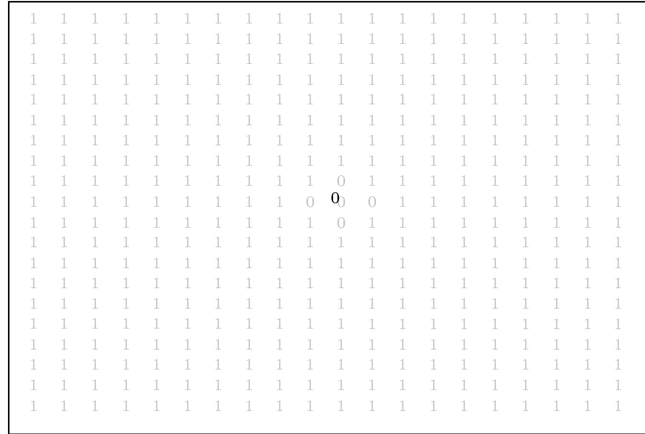


Figure 2.8: *Région d'attraction autour d'un point avec $\alpha = 50$.*

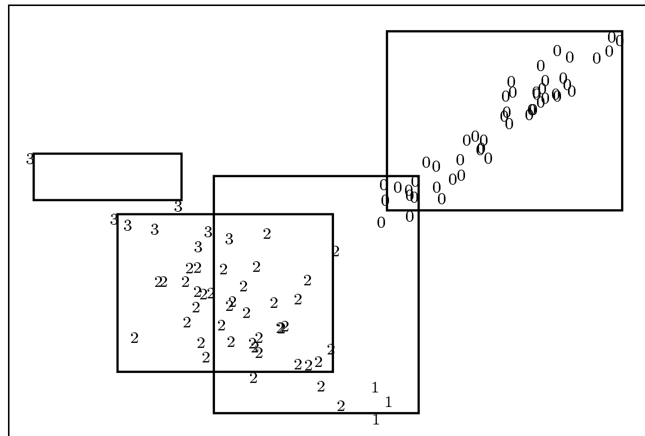


Figure 2.9: *Classement des données #1 après une passe ($\alpha = 1$, $\beta = 1$ et $\rho = 0.5$).*

(étape 4 de l'algorithme, section 2.1.2). Nous constatons ce phénomène après une passe d'entraînement avec les données #1 quand $\alpha = 1$ (fig. 2.9). Quelques points dans la frontière de la classe 1 sont classés dans la classe 2 et certains points de la classe 2 sont placés dans la classe 3. Les frontières s'agrandiront pour inclure ces points qui sont à l'extérieur dès la passe suivante.

Par ailleurs, quand le paramètre de sélection devient relativement grand, la vigilance perd de son effet. Quand $\alpha = 30$, par exemple, avec les données #5, le classement (après stabilisation) est le même pour $\rho \leq 0.5$. Il semble donc plus avantageux de garder la valeur du paramètre de choix α assez petite.

Nous comprenons maintenant mieux le fonctionnement du Fuzzy ART, après avoir analysé le classement qu'il effectue avec des points dans le plan. Nous avons

vu que celui-ci se comporte comme le ART1 sous plusieurs aspects. Nous avons ensuite étudié plus précisément le Fuzzy ART en discutant de l'influence des paramètres β et α.

2.3 Implantation

Cette section porte sur l'implantation en C++ du réseau Fuzzy ART. Nous verrons d'abord la classe `FuzzyART` elle-même, puis les différents programmes qui l'utilisent, soit ceux qui ont servi à illustrer le fonctionnement du réseau à la section 2.2.

Le code source de cette implantation est disponible sur le réseau du département de génie électrique et de génie informatique de l'Université Laval, dans les mêmes répertoires que le ART1 (voir la section 1.3, p.30).

2.3.1 Classe FuzzyART

La classe `FuzzyART` permet de gérer un réseau de neurones de type Fuzzy ART dont la structure est décrite à la section 2.1.1.

Le Fuzzy ART comprend une matrice de poids, un vecteur d'entrée, ainsi qu'un vecteur d'activité pour chacune des 2 couches. Il y a également un vecteur qui mémorise les inhibitions des neurones sur la couche F_2.[3]

Code source : `FuzzyART.hpp` et `FuzzyART.cpp`.

Constructeur

Le constructeur de la classe permet de créer la structure du réseau. Il possède deux paramètres :

```
FuzzyART(int _dimF1=4, int _dimF2=6);
```

Le premier paramètre est la dimension de la couche F_1 et le second est la dimension de la couche F_2. Si le codage complémentaire est utilisé, ce qui est

[3] Les classes `Vecteur` et `Matrice` utilisées sont celles créées par Marc Parizeau et sont disponibles à `www.gel.ulaval.ca/~parizeau/C++/classes/doc/Classes/Classes.html`.

normalement le cas, la couche F_1 doit être 2 fois plus grande que les vecteurs d'entrée.

Les paramètres du réseau sont initialisés aux valeurs suivantes: $\alpha = 0.01$, $\beta = 1$ et $\rho = 0.5$. Ils peuvent être modifiés directement avec les fonctions `Alpha`, `Beta` et `Vigilance`. On peut aussi les modifier en les lisant à partir d'un fichier à l'aide de la fonction `LireParam`. Tous les poids des connexions sont initialisés à 1.

Opérateurs

La classe `FuzzyART` possède les opérateurs suivants :

- `friend ostream &operator<<(ostream &_os, const FuzzyART &_art);`

  Insère le réseau *_art* dans le flot de sortie *_os* (voir `Ecrire`).

- `friend istream &operator>>(istream &_is, FuzzyART &_art);`

  Extrait le réseau *_art* dans le flot d'entrée *_is* (voir `Lire`).

Fonctions d'interface

La classe `FuzzyART` possède les fonctions d'interface suivantes :

- `Vecteur ActivitesF1();`

  Retourne les activités sur la couche F_1.

- `Vecteur ActivitesF2();`

  Retourne les activités sur la couche F_2.

- `void Agrandissement(int _m=1);`

  Le réseau a un mode d'agrandissement automatique, actif par défaut, qui permet à la couche F_2 de s'agrandir si tous ses neurones sont utilisés. La fonction `Agrandissement` contrôle ce mode. Le paramètre *_m* doit être 0 ou 1, 0 signifiant que l'agrandissement automatique est désactivé.

- `double Alpha();`

  Retourne la valeur du paramètre de sélection α.

- `void Alpha(double _a);`

  Fixe à $_a$ la valeur du paramètre de sélection α.

- `void Apprentissage(int _m=1);`

  Permet de désactiver (ou d'activer) l'apprentissage du réseau, c'est-à-dire de l'empêcher (ou de lui permettre) de modifier les poids de ses connexions. Normalement l'apprentissage est actif, mais il peut être utile, dans certains cas, de le désactiver. Le paramètre $_m$ doit être 0 ou 1, 0 signifiant que l'apprentissage est désactivé.

- `double Beta();`

  Retourne la valeur du taux d'apprentissage β.

- `void Beta(double _b);`

  Fixe à $_b$ la valeur du taux d'apprentissage β.

- `void CodageComplementaire(int _m);`

  Permet de désactiver (ou d'activer) le codage complémentaire des entrées. Par défaut, celui-ci est actif. Il peut être utile de le désactiver, par exemple pour que le Fuzzy ART se comporte comme un ART1 sur des données binaires. Le paramètre $_m$ doit être 0 ou 1, 0 signifiant que le codage complémentaire est désactivé.

- `int Correspondance();`

  Vérifie s'il y a correspondance entre le vecteur d'entrée en codage complémentaire et le vecteur d'activité de F_1. Si le degré de correspondance est supérieur ou égal à la vigilance, la fonction retourne 1, sinon elle retourne 0.

- `void Dim(int _dimF1, int _dimF2);`

  Fixe les dimensions des couches F_1 et F_2. Les poids sont réinitialisés à leur valeur par défaut.

- `int DimF1();`

  Retourne la dimension de la couche F_1.

- `int DimF2();`

  Retourne la dimension de la couche F_2.

- `void DimF2(int _n);`

  Change le nombre de neurones sur la couche F_2. Cette fonction est normalement utilisée pour agrandir la couche quand tous les neurones sont utilisés. Les nouveaux poids dans les matrices agrandies sont initialisés à leur valeur par défaut. Ceux qui existent déjà demeurent intacts.

- `void Ecrire(ostream &_os=cout);`

  Écrit le réseau, y compris les paramètres, dans le flot de sortie *_os*. Par défaut, *cout* est employé.

- `void EnleverInhibitions();`

  Remet tous les neurones de la couche F_2 dans l'état non inhibé.

- `Vecteur Entrees();`

  Retourne le vecteur d'entrée de la couche F_1 (le vecteur d'entrée en codage complémentaire).

- `void Entrer(const Vecteur &_e);`

  Place le vecteur *_e* en entrée du réseau, sans le propager. Effectue s'il y a liet le codage complémentaire du vecteur.

- `int GagnantF2();`

  Retourne l'indice du neurone gagnant sur la couche F_2.

- `void Inhiber(int _j);`

  Inhibe le neurone d'indice *_j* sur la couche F_2.

- `void Lire(istream &_is=cin);`

  Lit le réseau à partir du flot d'entrée *_is*. Par défaut, *cin* est employé.

- `void LireParam(istream &_is=cin);`

  Lit les paramètres du réseau à partir du flot d'entrée *_is*. Par défaut, *cin* est employé. Le fichier peut contenir n'importe quel paramètre, inscrit sous la forme `nom: valeur`, où `nom` peut être `alpha`, `beta` ou `rho` (par exemple `rho: 0.6`). Si un des paramètres n'est pas spécifié, il est initialisé à sa valeur par défaut.

- `void MettreAJour();`

  Met à jour les poids des connexions reliées au neurone gagnant sur la couche F_2. Cette fonction fait partie du processus de propagation et n'a généralement pas à être appelée individuellement.

- `void Propager(const Vecteur &_entree);`

  Place le vecteur *_entree* en entrée du réseau (il n'est pas nécessaire d'appeler la fonction `Entrer`) et effectue l'algorithme de propagation jusqu'à ce qu'il y ait correspondance. Lorsque le tout sera complété, il est possible de récupérer l'indice du neurone gagnant par la fonction `GagnantF2`.

- `void PropagerVersF1();`

  Propage le vecteur d'entrée vers la couche F_1. Cette fonction fait partie du processus de propagation et n'a généralement pas à être appelée individuellement.

- `void PropagerVersF2();`

  Propage les sorties de la couche F_1 vers la couche F_2. Cette fonction fait partie du processus de propagation et n'a généralement pas à être appelée individuellement.

- `void RetropropagerVersF1(int _gF2);`

  Rétropropage les sorties de la couche F_2 vers la couche F_1. Le neurone d'indice *_gF2* est fixé comme étant le neurone gagnant. Celui-ci est normalement déterminé lors de la propagation vers F_2 mais il peut être choisi "manuellement".

- `Vecteur Template(int _j);`

  Retourne le *top-down template* de la classe *_j*, c'est-à-dire le vecteur $\mathbf{w}_{_j}$. Cette fonction est très utile pour extraire ce qu'a appris le réseau.

- `void Trace(int _mode=1);`

  Permet d'activer ou de désactiver le mode trace, qui fait afficher à l'écran des informations lors de la propagation d'un vecteur à travers le réseau. Par défaut le mode trace est inactif. Le paramètre *_m* doit être 0 ou 1, 0 signifiant que le mode trace est désactivé.

- `double Vigilance();`

  Retourne la valeur de la vigilance ρ.

- `void Vigilance(double _r);`

  Fixe à *_r* la valeur de la vigilance ρ.

Exemple d'utilisation

Le programme suivant crée un réseau Fuzzy ART avec 4 entrées et 10 sorties, avec une vigilance de 0.8 et un taux d'apprentissage de 0.5 ($\alpha = 0.01$, la valeur par défaut). On crée un vecteur et on le propage dans le réseau. La vecteur a 2 dimensions car il subit un codage complémentaire lors de sa présentation au Fuzzy ART. L'indice du neurone gagnant sera affiché à l'écran.

```
#include "FuzzyART.hpp"

int main() {
  FuzzyART res(4,10);
  res.Vigilance(0.8);
  res.Beta(0.5);
  Vecteur v(2);
  v(0)=0.4; v(1)=0.8;
  res.Propager(v);
  cout << "Le neurone " << res.GagnantF2() << " gagne!" << endl;
}
```

2.3.2 Programmes utilisant le Fuzzy ART

Deux programmes utilisant la classe `FuzzyART` ont été réalisés.

fuzzyart_2D	Permet d'utiliser le Fuzzy ART sur des données dans un plan XY.
fuzzyart_souris	Permet d'utiliser le Fuzzy ART pour classer des points cliqués par la souris.

Programme fuzzyart_2D

Le programme `fuzzyart_2D` permet d'utiliser le Fuzzy ART sur des données dans un plan XY. Il lit dans un fichier une série de points enregistrés sous forme de table de table de vecteurs (`<Table1D<Table1D<Vecteur>>>`), chaque sous-table contenant les vecteurs qui appartiennent à la même classe. Ces points sont normalisés dans l'intervalle [0,1] et sont fournis un à la suite de l'autre à un réseau Fuzzy ART.

Lorsque le fichier de données s'affiche dans une fenêtre, cliquer sur celle-ci pour que le Fuzzy ART démarre son apprentissage. Lorsque la lecture des données est complétée, le classement effectué par le réseau s'affiche dans une autre fenêtre. À ce moment, il y a trois options :

- Cliquer sur le bouton de gauche de la souris fait lire à nouveau les données au réseau.
- Cliquer sur le bouton du centre de la souris fait afficher les frontières apprises par le réseau.
- Cliquer sur le bouton de droite de la souris fait afficher une grille de points montrant comment le réseau classerait ces points dans son état actuel.

Ce programme accepte plusieurs paramètres sur la ligne de commande :

-d *nom*	Nom du fichier contenant les données, `data1.tp2` par défaut.
-P	Permuter les données (à chaque lecture) au lieu de les lire dans l'ordre.
-a *valeur*	Valeur du paramètre de sélection α, 0.01 par défaut.
-b *valeur*	Valeur du taux d'apprentissage β, 1 par défaut.
-R *valeur*	Valeur de la vigilance ρ, 0.5 par défaut.
-p *nom*	Nom du fichier dans lequel lire les paramètres, une alternative à fournir ceux-ci sur la ligne de commande.
-n *valeur*	Nombre de sorties du réseau (dimension N de la couche F_2), 10 par défaut.

Code source : `fuzzyart_2D.cpp`.

Programme fuzzyart_souris

Le programme `fuzzyart_souris` permet d'utiliser le Fuzzy ART pour classer des points cliqués à l'écran par la souris.

Lorsque la fenêtre s'affiche, il y a quatre options :

- Cliquer sur le bouton de gauche de la souris fait lire un nouveau point au réseau.

- Cliquer sur le bouton du centre de la souris fait afficher les frontières apprises par le réseau.
- Cliquer sur le bouton de droite de la souris fait afficher une grille de points montrant comment le réseau classerait ces points dans son état actuel.
- Cliquer sur le bouton du centre et le bouton de gauche représente tous les points cliqués au réseau et réaffiche les frontières. Cette option est particulièrement utile lorsque le taux d'apprentissage $\beta < 1$.

Ce programme accepte quelques paramètres sur la ligne de commande :

-a *valeur*	Valeur du paramètre de sélection α, 0.01 par défaut.
-b *valeur*	Valeur du taux d'apprentissage β, 1 par défaut.
-R *valeur*	Valeur de la vigilance ρ, 0.5 par défaut.
-p *nom*	Nom du fichier dans lequel lire les paramètres, une alternative à fournir ceux-ci sur la ligne de commande.

Code source : `fuzzyart_souris.cpp`.

C'est ce qui complète la section sur l'implantation du Fuzzy ART en C++. Nous avons étudié la classe `FuzzyART` elle-même, puis les différents programmes qui l'utilisent et qui ont pu servir à faire les démonstrations à la section 2.2.

Chapitre 3

Fuzzy Min-Max

Le réseau Fuzzy Min-Max a comme utilité de classer des données *sans supervision* et de façon *continue*, comme les autres réseaux de neurones que nous avons vus jusqu'à présent. Il accepte des entrées analogiques, ce qui le rend similaire au Fuzzy ART. De plus, il utilise des hyper-rectangles comme frontières pour séparer les classes, ce qui le rapproche encore davantage des architectures ART. Par contre, le Fuzzy Min-Max cherche à éviter les recouvrements entre les frontières des différentes classes qu'il forme. Il a été introduit par Patrick K. Simpson en 1993 [11].

Nous suivrons la même structure qu'aux autres chapitres, c'est-à-dire que nous ferons d'abord un résumé de l'algorithme d'apprentissage du réseau, puis nous regarderons son fonctionnement sur des points dans un plan, pour terminer avec l'implantation de celui-ci en C++.

3.1 Algorithme d'apprentissage

L'algorithme d'apprentissage du Fuzzy Min-Max consiste à placer des hyper-rectangles[1] dans l'espace des données qui encadrent les différentes classes formées. Il y a deux paramètres à fixer, soit la sensibilité et la grandeur maximale des hyper-rectangles.

Dans l'ordre, nous verrons la structure générale du réseau, les équations gérant son fonctionnement, puis nous analyserons les paramètres du Fuzzy Min-Max.

[1] Un hyper-rectangle est un rectangle qui possède un nombre quelconque de dimensions.

3.1.1 Structure du réseau

Le réseau Fuzzy Min-Max peut être structuré sous la forme d'un réseau de neurones [11, pp. 38–40]. Par contre, nous ne suivrons pas cette approche, car l'algorithme peut se réaliser plus simplement. De plus, l'avantage du réseau neuronal est au niveau de son fonctionnement en parallèle, ce qui n'aurait pas été reproduit en l'implantant en C++.

Donc, bien que le Fuzzy Min-Max soit un réseau de neurones, nous l'implanterons sous une forme différente. Il sera un ensemble d'hyper-rectangles qui peuvent s'agrandir et se contracter pour créer des frontières séparant des données en un certain nombre de classes.

Chaque hyper-rectangle, défini par ses points minimum et maximum, représente un ensemble flou (d'où le nom du réseau). Tout ce qui est à l'intérieur appartient à une classe bien précise, tandis que ce qui est à l'extrérieur appartient à cette classe à un degré plus ou moins élevé. Les hyper-rectangles ne peuvent jamais se recouvrir, car une donnée ne peut appartenir à 100% à deux classes ou plus à la fois.

Cette idée d'utiliser des hyper-rectangles apporte un avantage sur le Fuzzy ART. Il n'est pas nécessaire de prévoir à l'avance le nombre maximal de classes qui seront formées en fixant la dimension de la couche F_2. Le nombre d'hyper-rectangles augmente automatiquement selon les besoins.

3.1.2 Équations

Voici un résumé de l'algorithme d'apprentissage du Fuzzy Min-Max avec les équations correspondantes. On suppose que les vecteurs d'entrée ont n composantes. On identifie les j hyper-rectangles (à n dimensions) du réseau Fuzzy Min-Max par B_j. Les points minimum et maximum qui les définissent sont identifiés respectivement par V_j et W_j.

Il y a deux paramètres qui sont utilisés: la sensibilité γ et la dimension maximale des hyper-rectangles θ. Plus de détails sur chacun de ces paramètres sont donnés à la section 3.1.3.

1. Initialisation du réseau

- Initialiser les paramètres en respectant les contraintes suivantes:

$$\gamma > 0$$

$$0 \leq \theta \leq 1$$

- Initialiser les points minimum V_0 et maximum W_0 d'un premier hyper-rectangle B_0.

$$V_0 = \underline{1} \tag{3.1}$$

$$W_0 = \underline{0} \tag{3.2}$$

Les vecteurs $\underline{1}$ et $\underline{0}$ sont des vecteurs à n dimensions contenant seulement des 1 et des 0, respectivement. Cette initialisation assure que la première donnée apprise par le rectangle soit le point correspondant à cette donnée.

2. Présentation d'une entrée

- Appliquer un vecteur A_h de n composantes à l'entrée du réseau.

3. Expansion d'un hyper-rectangle

- Rechercher l'hyper-rectangle B_j pour lequel A_h a le plus grand degré d'appartenance b_j.

$$b_j(A_h, V_j, W_j) = \frac{1}{n}\sum_{i=1}^{n}[1 - f(a_{hi} - w_{ji}, \gamma) - f(v_{ji} - a_{hi}, \gamma)] \tag{3.3}$$

La fonction $f()$ est la fonction d'activation.

$$f(x, \gamma) = \begin{cases} 1 & \text{si } x\gamma > 1 \\ x\gamma & \text{si } 0 \leq x\gamma \leq 1 \\ 0 & \text{si } x\gamma < 0 \end{cases} \tag{3.4}$$

- Vérifier si cet hyper-rectangle peut être étendu pour inclure A_h. L'expansion est possible si la condition suivante est respectée:

$$\frac{1}{n}\sum_{i=1}^{n}(\max(w_{ji}, a_{hi}) - \min(v_{ji}, a_{hi})) \leq \theta \tag{3.5}$$

  - Si la condition n'est pas respectée, inhiber l'hyper-rectangle B_j pour éviter qu'il soit sélectionné à nouveau et retourner au début de l'étape 3.
  - Si la condition est respectée, continuer.

- Modifier les points minimum V_j et maximum W_j de l'hyper-rectangle B_j pour y inclure le vecteur A_h.

$$v_{ji} = \min(v_{ji}, a_{hi}) \ \forall i = 1, 2, \ldots, n \tag{3.6}$$

$$w_{ji} = \max(w_{ji}, a_{hi}) \ \forall i = 1, 2, \ldots, n \tag{3.7}$$

4. Élimination des recouvrements Effectuer les étapes suivantes pour chacun des autres hyper-rectangles B_k.

   - Vérifier si l'hyper-rectangle B_j expansé respecte un des cas suivants de recouvrement pour chacune des n dimensions.
     - Cas 1: Le maximum de B_j recouvre le minimum de B_k.
       $$v_{ji} \leq v_{ki} < w_{ji} \leq w_{ki}$$
     - Cas 2: Le minimum de B_j recouvre le maximum de B_k.
       $$v_{ki} \leq v_{ji} < w_{ki} \leq w_{ji}$$
     - Cas 3: L'hyper-rectangle B_k est à l'intérieur de B_j.
       $$v_{ji} \leq v_{ki} \leq w_{ki} \leq w_{ji}$$
     - Cas 4: L'hyper-rectangle B_j est à l'intérieur de B_k.
       $$v_{ki} \leq v_{ji} \leq w_{ji} \leq w_{ki}$$
   - Si un de ces cas est respecté pour chacune des dimensions, il y a recouvrement entre B_j et B_k et on l'élimine dimension par dimension, en suivant les formules suivantes, selon le cas correspondant.
     - Cas 1:
       $$v_{ki} = w_{ji} = \frac{v_{ki} + w_{ji}}{2} \tag{3.8}$$
     - Cas 2:
       $$v_{ji} = w_{ki} = \frac{v_{ji} + w_{ki}}{2} \tag{3.9}$$
     - Cas 3: La contraction la plus petite nécessaire est effectuée.
       Si $w_{ki} - v_{ji} < w_{ji} - v_{ki}$, faire
       $$v_{ji} = w_{ki} \tag{3.10}$$
       Sinon,
       $$w_{ji} = v_{ki} \tag{3.11}$$
     - Cas 4: Utiliser la même procédure que dans le cas 3.

5. Réactiver tous les hyper-rectangles et répéter l'algorithme à partir de l'étape 2 avec un nouveau vecteur d'entrée.

3.1.3 Paramètres

Voyons maintenant l'utilité de chaque paramètre et leur effet sur le fonctionnement du réseau.

- Sensibilité γ

  La sensibilité γ est utilisée dans la fonction d'activation (éq. 3.4), qui elle sert dans le calcul de l'appartenance d'un vecteur à un hyper-rectangle (éq. 3.3).

  Pour une dimension i, l'appartenance a une valeur maximale de 1 quand la composante a_{hi} du vecteur d'entrée A_h est entre le minimum v_{ji} et le maximum w_{ji} de l'hyper-rectangle dans cette dimension ($v_{ji} < a_{hi} < w_{ji}$). En effet, $a_{hi} - w_{ji} < 0$ et $v_{ji} - a_{hi} < 0$, donc le terme à l'intérieur de la sommation de l'équation 3.3, par l'équation 3.4, est égal à 1.

  Quand la composante a_{hi} est à l'extérieur de l'hyper-rectangle, l'appartenance diminue avec la distance, jusqu'à ce qu'elle devienne nulle, à une distance de $1/\gamma$ et plus de l'extrémum le plus près. En effet, si $a_{hi} < v_{ji} \leq w_{ji}$, par l'équation 3.4,

  $$1 - f(a_{hi} - w_{ji}, \gamma) - f(v_{ji} - a_{hi}, \gamma) = 1 - 0 - f(v_{ji} - a_{hi}, \gamma)$$

  Or,

  $$\begin{aligned} 0 \leq f(v_{ji} - a_{hi}, \gamma) \leq 1 &\iff 0 \leq (v_{ji} - a_{hi})\gamma \leq 1 \\ &\iff v_{ji} - \tfrac{1}{\gamma} \leq a_{hi} \leq v_{ji} \end{aligned}$$

  De même, si $v_{ji} \leq w_{ji} < a_{hi}$,

  $$1 - f(a_{hi} - w_{ji}, \gamma) - f(v_{ji} - a_{hi}, \gamma) = 1 - f(a_{hi} - w_{ji} - 0, \gamma)$$

  et

  $$\begin{aligned} 0 \leq f(a_{hi} - w_{ji}, \gamma) \leq 1 &\iff 0 \leq (a_{hi} - w_{ji})\gamma \leq 1 \\ &\iff w_{ji} \leq a_{hi} \leq w_{ji} + \tfrac{1}{\gamma} \end{aligned}$$

  L'appartenance d'un vecteur à un hyper-rectangle, dans une dimension précise, est donc non nulle à moins de $1/\gamma$ de distance de ce rectangle et elle a une valeur maximale de 1 à l'intérieur de celui-ci. Nous voyons maintenant que la sensibilité γ détermine à quelle vitesse la fonction d'appartenance diminue à l'extérieur d'un hyper-rectangle. Autrement dit, plus γ est petit, plus l'ensemble est flou et plus γ est grand, plus l'ensemble est non flou (*crisp*).

- Dimension maximale θ

  Le paramètre θ est la dimension maximale que peut avoir un hyper-rectangle. La dimension d'un rectangle est ici définie comme étant la moyenne des longueurs des côtés. Lorsqu'on a trouvé le rectangle où le vecteur d'entrée a la plus grande appartenance, on agrandit celui-ci seulement si sa nouvelle dimension est inférieure ou égale à θ (éq. 3.5).

  Donc, plus θ est petit, plus les classes créées sont précises, tandis que plus θ est grand, plus les classes créées sont grossières. Le rôle de ce paramètre est similaire à la vigilance ρ dans le ART1 et le Fuzzy ART.

Valeurs suggérées pour les paramètres

Voici les valeurs suggérées pour les paramètres, accompagnées des contraintes à respecter pour chacun d'eux.

Paramètre	*Contrainte*	*Valeur suggérée*
γ	$\gamma > 0$	4
θ	$\theta \in [0, 1]$	0.4

Il faut garder la sensibilité γ relativement petite pour avoir une certaine frontière floue autour de chaque hyper-rectangle. Si γ est suffisamment petite, un vecteur peut appartenir à plusieurs hyper-rectangles à différents degrés. Une sensibilité trop grande fait que l'appartenance d'un vecteur à un hyper-rectangle est nulle dès que le vecteur est à l'extérieur de celui-ci. Simpson suggère $\gamma = 4$ [11, p. 36].

Pour la dimension maximale des hyper-rectangles θ, faire varier ce paramètre pour trouver quelle valeur donne les meilleurs résultats. En général, une valeur autour de 0.4 donne de bons classements.

Nous avons vu la structure du réseau Fuzzy Min-Max et son algorithme d'apprentissage. Nous avons également étudié les paramètres qui le contrôlent. À la section suivante, nous illustrerons son fonctionnement sur des points dans un plan, comme nous l'avons fait pour les autres réseaux de neurones.

3.2 Fonctionnement

Voyons quelle performance donne le réseau Fuzzy Min-Max en classant des points dans un plan à deux dimensions. Le réseau a été programmé en C++, et plus

d'informations sur cette implantation sont données à la section 3.3.

Le réseau Fuzzy Min-Max a deux dimensions, une pour chacune des coordonnées X et Y. Les coordonnées sont d'abord normalisées entre 0 et 1. Nous pouvons extraire les frontières apprises par le réseau en récupérant les points minimum et maximum de chaque hyper-rectangle.

Nous verrons d'abord de quelle façon la Fuzzy Min-Max empêche les recouvrements entre les hyper-rectangles, puis nous discuterons du phénomène d'instabilité perpétuelle. Nous verrons ensuite l'effet des deux paramètres du réseau, la dimension maximale θ et la sensibilité γ.

3.2.1 Élimination des recouvrements

La caractéristique distinctive du Fuzzy Min-Max est d'éliminer les recouvrements entre les frontières des classes quand il y en a. Regardons avec deux exemples comment cela se fait.

À la figure 3.1a, nous présentons à un réseau, qui a déjà appris 6 points, un nouveau point qui cause un recouvrement des frontières. Le réseau place effectivement le nouveau point (le plus à droite sur la figure) dans la classe 1, ce qui amène cette classe à recouvrir une partie de la classe 0. Le réseau élimine immédiatement le recouvrement, ce qui donne les frontières que l'on voit à la figure 3.1b. Le recouvrement était un cas 2 sur X et un cas 1 sur Y (voir la section 3.1.2, étape 4).

Prenons un autre exemple avec un type de recouvrement différent. À la figure 3.2a, nous ajoutons un point, au haut, que le réseau place dans la classe 1. En agrandissant la frontière pour inclure ce point, il est aux prises avec un cas 4 de recouvrement sur X et un cas 3 sur Y. En effectuant la contraction la plus petite nécessaire pour chaque hyper-rectangle, nous obtenons les frontières de la figure 3.2b.

La contraction des hyper-rectangles lorsqu'il y a un recouvrement cause des changements importants dans le classement effectué par le réseau. Dans cet exemple, nous remarquons qu'après contraction des frontières, le réseau place le dernier point ajouté dans la classe 0 plutôt que dans la classe 1. Ceci signifie que lorsqu'il y a un grand nombre de points et que plusieurs recouvrements se produisent, le classement peut varier énormément après chaque passe d'entraînement.

La figure 3.2c présente le classement final des points de l'exemple #2, après

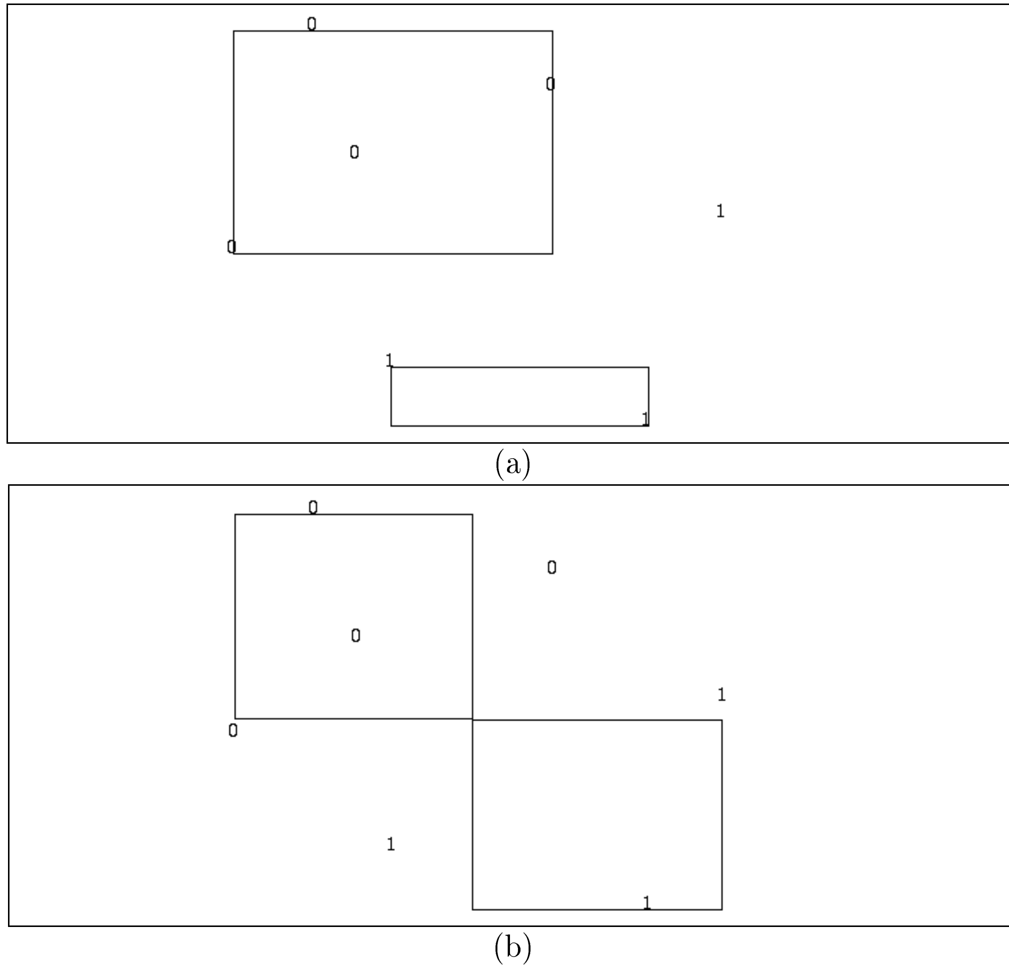


Figure 3.1: *Exemple #1 — (a) Ajout d'un point qui cause un recouvrement. (b) Frontières après l'élimination du recouvrement.*

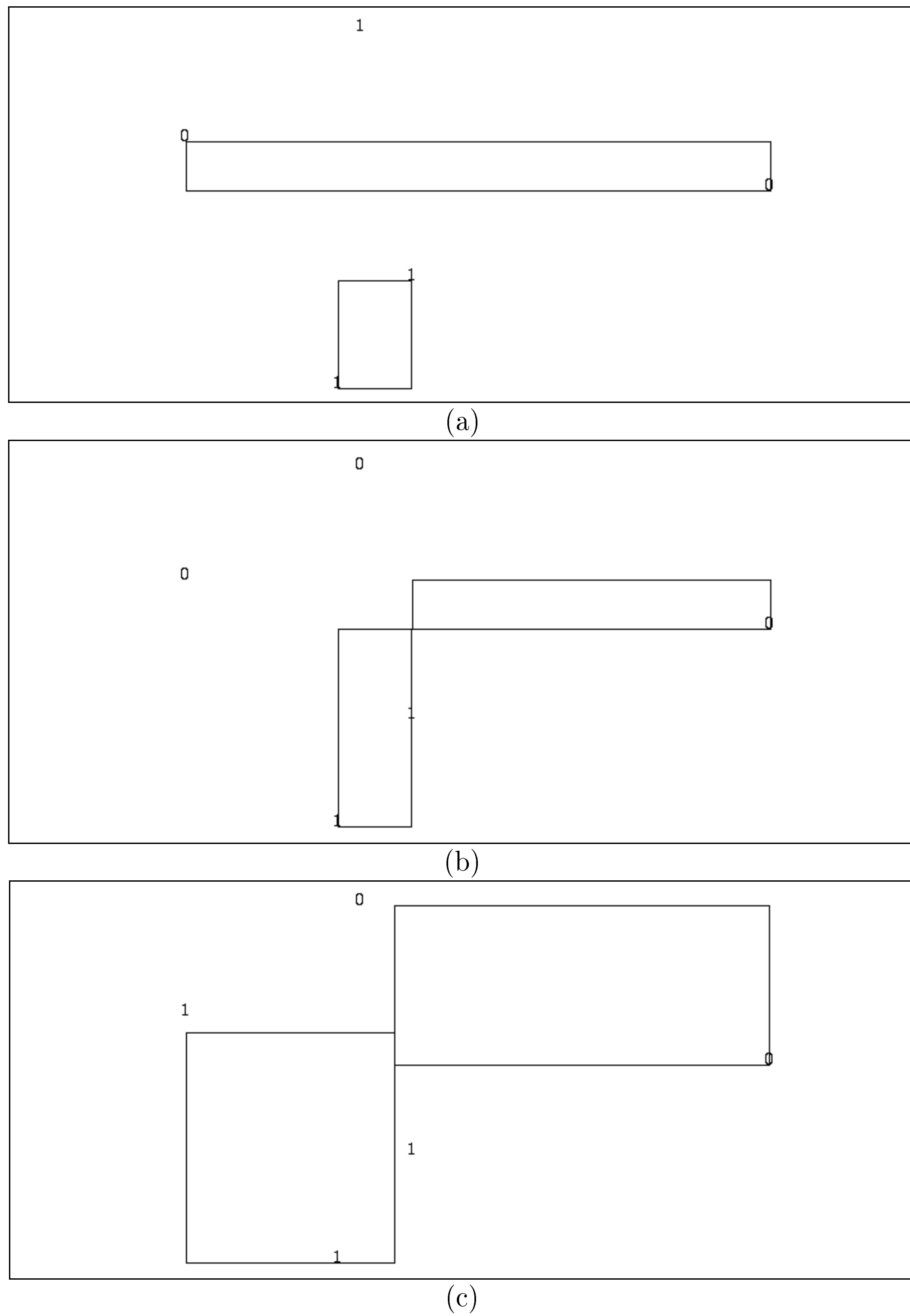


Figure 3.2: *Exemple #2 — (a) Ajout d'un point qui cause un recouvrement. (b) Frontières après l'élimination du recouvrement. (c) Classement final.*

quelques passes d'entraînement supplémentaires avec les mêmes données dans le même ordre. Notons que 2 points sur 5 ont changé de classe depuis la figure 3.2a, sans qu'on présente aucun nouveau point au réseau. Remarquons aussi que les hyper-rectangles ne contiennent pas toutes les données. Le réseau est incapable d'aller chercher certains points. Nous discuterons de ce phénomène dans la section qui suit.

3.2.2 Instabilité perpétuelle

Simpson définit la stabilité du réseau comme un stade où les hyper-rectangles ne changent plus lors de la présentation successive de données dans le même ordre [11, p. 38]. Or, il y a un problème dans le réseau Fuzzy Min-Max dont il ne fait pas mention: il peut se produire des cas où le réseau ne parviendra jamais à se stabiliser. Nous nommerons cette situation “instabilité perpétuelle”.

Nous avions une illustration de ceci à la figure 3.2c de la section précédente. Le réseau, après chaque passe d'entraînement, demeure dans cet état, mais ne parvient pas à inclure tous les points dans les hyper-rectangles. Ceci peut donner l'impression qu'il est stabilisé, mais ce n'est pas le cas, car les frontières sont modifiées au cours d'une passe d'entraînement mais reviennent toujours au même état à la fin. Le réseau est en instabilité perpétuelle.

Nous comprendrons mieux ce phénomène avec un autre exemple, où nous suivrons l'évolution des frontières au cours d'une passe d'entraînement. La figure 3.3a montre l'état des frontières après une passe, état dans lequel elles reviennent perpétuellement.

Sur les figures suivantes, nous voyons à chaque étape le point qui vient d'être présenté au réseau ainsi que les nouvelles frontières qui ont été formées. À la figure 3.3b, l'hyper-rectangle de la classe 1 s'est agrandi pour aller chercher le point, mais ceci a causé un recouvrement avec celui de la classe 0, ce qui donne le résultat que l'on voit après les contractions. Par la suite, le réseau est en mesure d'aller chercher les deux points suivants, tel que nous le voyons aux figures 3.3c et 3.3d. Le quatrième point est placé dans la classe 0, ce qui cause un recouvrement puis une contraction des frontières, pour donner la situation illustrée à la figure 3.3e. Finalement, la frontière de la classe 1 s'agrandit pour inclure le dernier point (fig. 3.3f) et les frontières se retrouvent dans le même état qu'au départ. Si nous faisions une autre passe d'entraînement, la même séquence se répèterait à nouveau.

C'est donc le principe d'élimination des recouvrements lui-même qui cause les

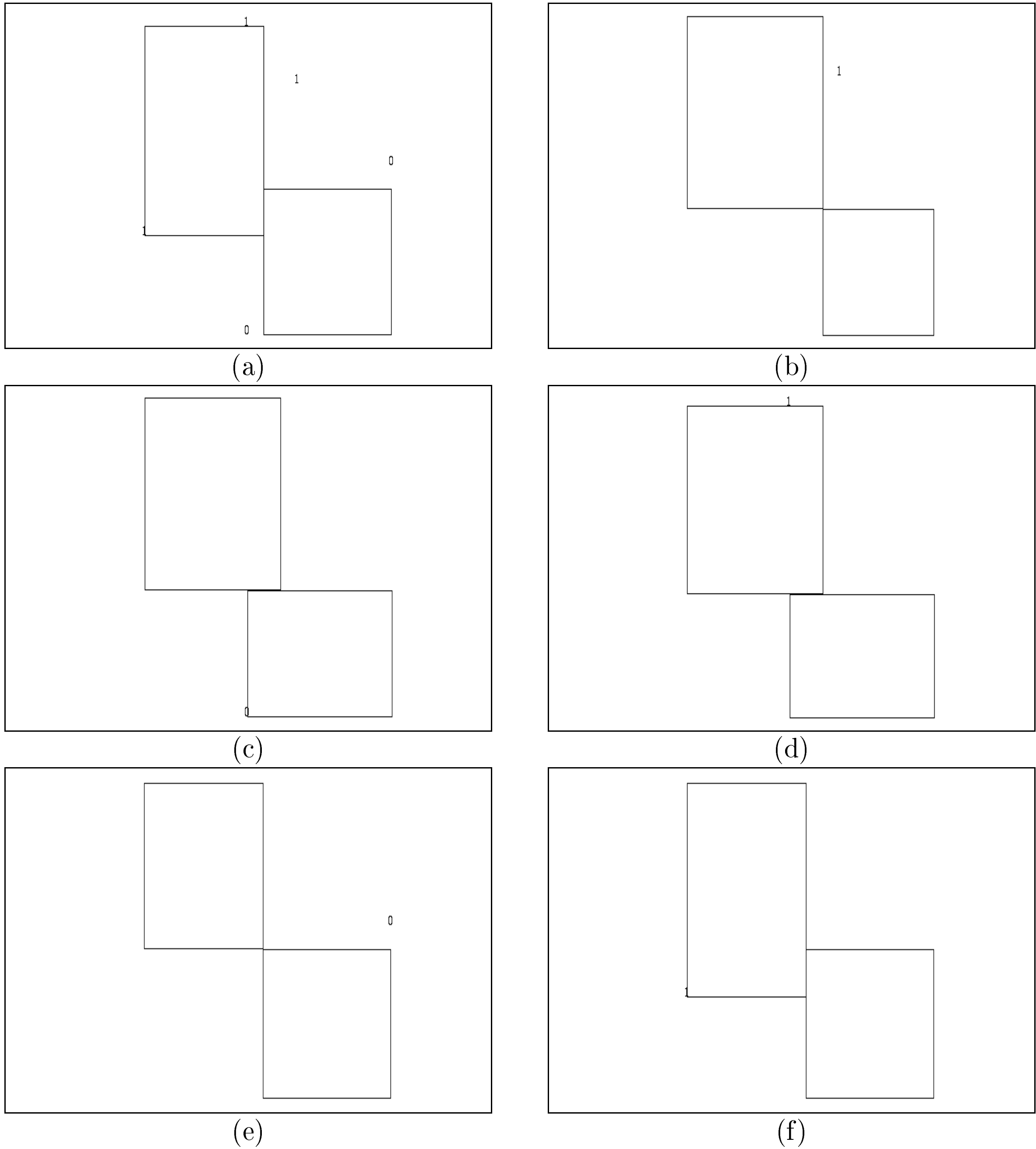


Figure 3.3: *Exemple #3 — Situation d'instabilité perpétuelle.*

situations d'instabilité perpétuelle. Le Fuzzy Min-Max cherche à inclure tous les points dans les hyper-rectangles tout en évitant de faire des recouvements. Or, il peut y avoir une séquence critique de données ou la contraction des rectangles après un ou des recouvrements ramène constamment les frontières dans le même état.

Bien sûr, il est possible de diminuer la probabilité d'avoir une séquence critique de données causant une instabilité perpétuelle en permutant les données à chaque passe d'entraînement. Toutefois on ne peut éliminer complètement cette probabilité.

Il faut par contre noter qu'il n'est pas catastrophique que certains points ne soient à l'intérieur d'aucun hyper-rectangle. Si les points à l'intérieur de l'un d'entre eux appartiennent complètement à la classe qui correspond, ceux qui sont à l'extérieur appartiennent à différents degrés à plusieurs classes. Il est donc possible de les catégoriser car ils ont une appartenance floue à plusieurs ensembles. À la limite, on pourra dire qu'un point fait partie de la classe pour lequel il a le plus grand degré d'appartenance. Seulement, le fait que certains points restent à l'extérieur montre qu'on risque de ne jamais obtenir la stabilité, de la façon dont elle est définie par Simpson.

3.2.3 Influence de la dimension maximale θ

L'effet de la dimension maximale des hyper-rectangles θ est assez prévisible. Selon les données, le classement pourra s'améliorer si les hyper-rectangles sont plus ou moins grands.

En général, la valeur par défaut ($\theta = 0.4$) fonctionne bien. Nous voyons à la figure 3.4 un classement obtenu sur les données #5 (voir la section 1.2.1). Nous avons permuté les données à chaque passe d'entraînement pour éviter le plus possible d'être en instabilité perpétuelle (section 3.2.2). Nous ferons également des permutations pour les deux exemples qui suivent.

D'autres ensembles de données peuvent être mieux classés si la dimension maximale des hyper-rectangles est modifiée. Par exemple, nous avons souvent de meilleurs résultats sur le fichier #3 avec $\theta = 0.5$ (fig. 3.5) qu'avec $\theta = 0.4$. Le fichier de données #2, lui, nous donne des classements intéressants quand $\theta = 0.7$, par exemple celui de la figure 3.6.

Bref, un peu comme la vigilance ρ dans le ART1 et le Fuzzy ART, la valeur la

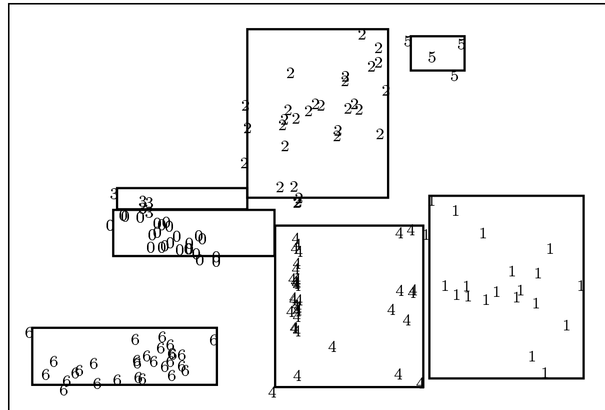


Figure 3.4: *Classement des données #5 avec permutation aléatoire ($\gamma = 4$ et $\theta = 0.4$).*

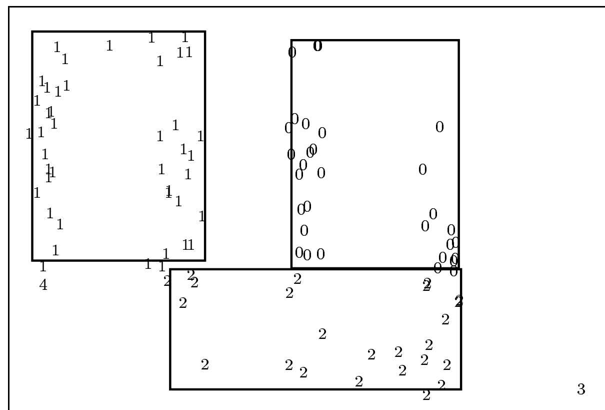


Figure 3.5: *Classement des données #3 avec permutation aléatoire ($\gamma = 4$ et $\theta = 0.5$).*

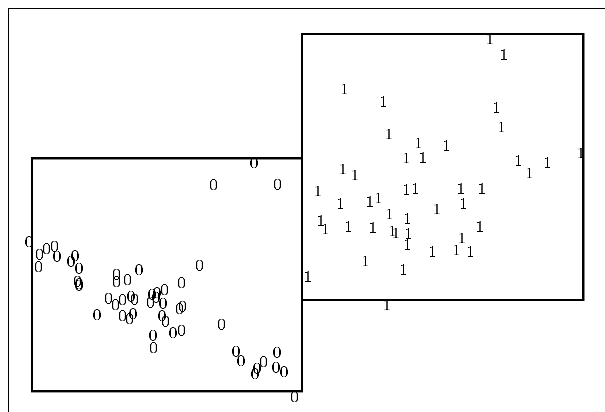


Figure 3.6: *Classement des données #2 avec permutation aléatoire ($\gamma = 4$ et $\theta = 0.7$).*

plus efficace de la dimension des hyper-rectangles θ varie selon les données. C'est en jouant avec ce paramètre que l'on peut essayer d'améliorer la performance du Fuzzy Min-Max, car la sensibilité, comme nous allons le voir, a un effet moins important.

3.2.4 Influence de la sensibilité γ

La sensibilité γ détermine à quel point les frontières sont floues. Tel que mentionné à la section 3.1.3, l'appartenance d'un point à un hyper-rectangle est non nulle à moins de $1/\gamma$ de distance de ce rectangle, pour une dimension précise, et elle a une valeur maximale de 1 à l'intérieur de celui-ci.

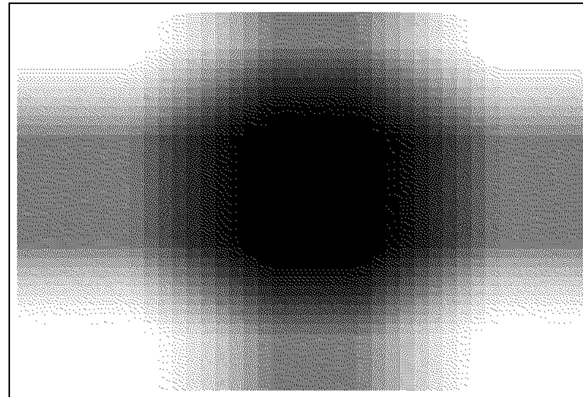
Nous voyons à la figure 3.7a l'allure de la fonction d'appartenance autour d'un hyper-rectangle avec $\gamma = 4$. Le rectangle est au centre de la figure, en noir. Le noir représente une appartenance de 1, le blanc 0 et le gris les valeurs intermédiaires. La figure nous montre le carré allant du point (0,0) au point (1,1) avec une résolution de 0.25. Notons qu'à partir d'un certain point, l'appartenance est constante lorsqu'on s'éloigne perpendiculairement à un côté du rectangle. Ceci est ennuyant, mais est normal étant donné la façon dont l'appartenance est calculée (éq. 3.3), qui est en fait la moyenne des appartenances sur chaque dimension.

Si nous augmentons la sensibilité, l'appartenance diminue plus vite autour de l'hyper-rectangle, mais il y a toujours les bandes où elle est constante vis-à-vis les côtés du rectangle. La figure 3.7b présente le même hyper-rectangle avec une sensibilité de 10.

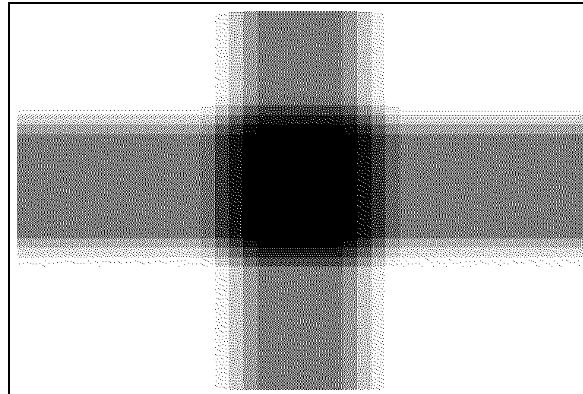
À la limite, quand γ devient très grand, on a la fonction d'appartenance illustrée à la figure 3.7c. L'appartenance est de 1 à l'intérieur de l'hyper-rectangle, 0.5 vis-à-vis un des côtés et 0 ailleurs. La frontière n'est donc pas vraiment non floue (*crisp*) comme le prétend Simpson. Il faudrait plutôt que l'on ait un rectangle noir entouré complètement de blanc. Pour ce faire nous aurions probablement avantage à prendre le minimum des appartenances au lieu de leur moyenne.

La sensibilité n'influence pas du tout le classement qu'effectue le réseau Fuzzy Min-Max. Il y a une différence seulement si on cherche à classer des points qui sont à l'extérieur de tous les hyper-rectangles.

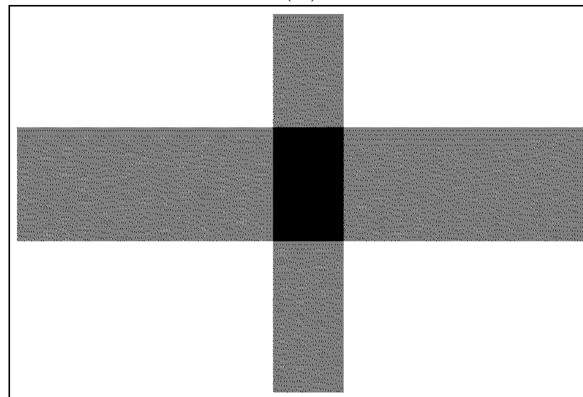
Cette section nous a permis de mieux comprendre le fonctionnement du Fuzzy Min-Max, en l'étudiant dans un espace à deux dimensions. Nous avons d'abord vu comment il procède pour éliminer les recouvrements entre les classes. Ensuite



(a)



(b)



(c)

Figure 3.7: *Fonction d'appartenance autour d'un hyper-rectangle. (a)* $\gamma = 4$. *(b)* $\gamma = 10$. *(c)* $\gamma = 500$.

nous avons constaté que le réseau peut parfois être en instabilité perpétuelle. Le tout a été complété par une analyse des deux paramètres θ et γ. Nous sommes maintenant prêts à passer à la réalisation concrète du Fuzzy Min-Max en C++.

3.3 Implantation

Nous traiterons maintenant de l'implantation en C++ du réseau Fuzzy Min-Max. Nous verrons d'abord la classe `FuzzyMinMax` elle-même, puis la classe `HyperRectangle` qu'elle utilise. Nous parcourrons ensuite les différents programmes qui utilisent le Fuzzy Min-Max et qui ont servi à illustrer son fonctionnement à la section 3.2.

Le code source de cette implantation est disponible sur le réseau du département de génie électrique et de génie informatique de l'Université Laval, dans les mêmes répertoires que le ART1 (voir la section 1.3, p.30).

3.3.1 Classe FuzzyMinMax

La classe `FuzzyMinMax` permet de gérer un réseau de type Fuzzy Min-Max dont la structure est décrite à la section 3.1.1.

Le Fuzzy Min-Max contient essentiellement une liste d'hyper-rectangles. Ceux-ci sont implémentés par la classe `HyperRectangle`, décrite à la section 3.3.2. Le réseau reçoit des vecteurs en entrée.[2]

Code source : `FuzzyMinMax.hpp` et `FuzzyMinMax.cpp`.

Constructeur

Le constructeur de la classe permet de créer la structure du réseau. Il possède un seul paramètre :

```
FuzzyMinMax(int _dims=2);
```

[2]Les classes `Liste` et `Vecteur` utilisées sont celles créées par Marc Parizeau et sont disponibles à `www.gel.ulaval.ca/~parizeau/C++/classes/doc/Classes/Classes.html`.

Le paramètre est le nombre de dimensions des vecteurs d'entrée et des hyper-rectangles. Par exemple, quand nous avons appliqué le réseau sur des points dans un plan, les vecteurs avaient 2 dimensions.

Les paramètres du réseau sont initialisés aux valeurs suivantes: $\gamma = 4$ et $\theta = 0.4$. Ils peuvent être modifiés directement avec les fonctions `Sensibilite` et `DimMax`. On peut aussi les modifier en les lisant à partir d'un fichier à l'aide de la fonction `LireParam`.

Opérateurs

La classe `FuzzyMinMax` possède les opérateurs suivants :

- `friend ostream &operator<<(ostream &_os, FuzzyMinMax &_fmm);`

  Insère le réseau *_fmm* dans le flot de sortie *_os* (voir `Ecrire`).

- `friend istream &operator>>(istream &_is, FuzzyMinMax &_fmm);`

  Extrait le réseau *_fmm* dans le flot d'entrée *_is* (voir `Lire`).

Fonctions d'interface

La classe `FuzzyMinMax` possède les fonctions d'interface suivantes :

- `double Appartenance(const Vecteur &_v, int _c);`

  Retourne l'appartenance du vecteur *_v* à l'hyper-rectangle d'indice *_c*.

- `void Apprentissage(int _m=1);`

  Permet de désactiver (ou d'activer) l'apprentissage du réseau, c'est-à-dire de l'empêcher (ou de lui permettre) de modifier les hyper-rectangles. Normalement l'apprentissage est actif, mais il peut être utile, dans certains cas, de le désactiver. Le paramètre *_m* doit être 0 ou 1, 0 signifiant que l'apprentissage est désactivé.

- `int Classer(const Vecteur &_v);`

  Effectue l'algorithme d'apprentissage avec le vecteur d'entrée *_v* et retourne l'indice de la classe où il a été placé.

- `double DimMax();`

  Retourne la valeur de la dimension maximale des hyper-rectangles θ.

- `void DimMax(double _d);`

  Fixe à *_d* la valeur de la dimension maximale des hyper-rectangles θ.

- `void Ecrire(ostream &_os=cout);`

  Écrit le réseau, y compris les paramètres, dans le flot de sortie *_os*. Par défaut, *cout* est employé.

- `void Lire(istream &_is=cin);`

  Lit le réseau à partir du flot d'entrée *_is*. Par défaut, *cin* est employé.

- `void LireParam(istream &_is=cin);`

  Lit les paramètres du réseau à partir du flot d'entrée *_is*. Par défaut, *cin* est employé. Le fichier peut contenir n'importe quel paramètre, inscrit sous la forme `nom: valeur`, où `nom` peut être `gamma` ou `theta` (par exemple `theta: 0.6`). Si un des paramètres n'est pas spécifié, il est initialisé à sa valeur par défaut.

- `int NbRectangles();`

  Retourne le nombre d'hyper-rectangles utilisés par le réseau. Le Fuzzy Min-Max contient toujours au moins un hyper-rectangle.

- `HyperRectangle Rectangle(int _i);`

  Retourne l'hyper-rectangle d'indice *_i*. Le premier hyper-rectangle a comme indice 0.

- `double Sensibilite();`

  Retourne la valeur de la sensibilité γ.

- `void Sensibilite(double _s);`

  Fixe à *_s* la valeur de la sensibilité γ.

Exemple d'utilisation

Le programme suivant crée un réseau Fuzzy Min-Max à 2 dimensions, avec comme dimension maximale des hyper-rectangles 0.5 ($\gamma = 4$, la valeur par défaut). On crée un vecteur et on le fournit au réseau. L'indice de la classe où il a été placé sera affiché à l'écran.

```
#include "FuzzyMinMax.hpp"

int main() {
  FuzzyMinMax res(2);
  res.DimMax(0.5);
  Vecteur v(2);
  v(0)=0.4; v(1)=0.8;
  cout <<"Ce vecteur est dans la classe "<< res.Classer(v) <<endl;
}
```

3.3.2 Classe HyperRectangle

La classe `HyperRectangle` permet de gérer un hyper-rectangle, servant dans le réseau Fuzzy Min-Max à former les frontières des classes apprises.

Un hyper-rectangle est défini par ses points minimum et maximum (voir la section 3.1.1). Les points sont des vecteurs de doubles, d'autant de composantes que le rectangle a de dimensions. Un hyper-rectangle a une sensibilité γ et une dimension maximale θ.

Code source : `FuzzyMinMax.hpp` et `FuzzyMinMax.cpp`.

Constructeur

Le constructeur de la classe initialise un hyper-rectangle ayant comme minimum le vecteur $\underline{1}$ et comme maximum le vecteur $\underline{0}$. Il possède trois paramètres :

```
HyperRectangle(double _dimmax=0.4, double _gamma=4, int _dims=2);
```

Les deux premiers paramètres sont la dimension maximale et la sensibilité. L'autre est le nombre de dimensions de l'hyper-rectangle.

Opérateurs

La classe `HyperRectangle` possède les opérateurs suivants :

- `friend ostream &operator<<(ostream &_os, HyperRectangle &_hr);`

Insère l'hyper-rectangle *_hr* dans le flot de sortie *_os* (voir `Ecrire`).

- `friend istream &operator>>(istream &_is, HyperRectangle &_hr);`
  Extrait l'hyper-rectangle *_hr* dans le flot d'entrée *_is* (voir `Lire`).

Fonctions d'interface

La classe `HyperRectangle` possède les fonctions d'interface suivantes :

- `void Agrandir(const Vecteur &_v);`
  Agrandit l'hyper-rectangle pour y inclure le vecteur *_v*.
- `double Appartenance(const Vecteur &_v);`
  Retourne l'appartenance du vecteur *_v* à l'hyper-rectangle.
- `double DimMax();`
  Retourne la valeur de la dimension maximale θ.
- `void DimMax(double _d);`
  Fixe à *_d* la valeur de la dimension maximale θ.
- `int Dims();`
  Retourne le nombre de dimensions de l'hyper-rectangle.
- `void Ecrire(ostream &_os=cout);`
  Écrit l'hyper-rectangle dans le flot de sortie *_os*. Par défaut, *cout* est employé.
- `int Expansible(const Vecteur &_v);`
  Vérifie si l'hyper-rectangle peut s'agrandir pour inclure le vecteur *_v*, étant donné la dimension maximale θ.
- `void Lire(istream &_is=cin);`
  Lit l'hyper-rectangle à partir du flot d'entrée *_is*. Par défaut, *cin* est employé.
- `Vecteur& Max();`
  Retourne le point maximum de l'hyper-rectangle.
- `Vecteur& Min();`
  Retourne le point minimum de l'hyper-rectangle.

- `double Sensibilite();`

  Retourne la valeur de la sensibilité γ.

- `void Sensibilite(double _s);`

  Fixe à $_s$ la valeur de la sensibilité γ.

3.3.3 Programmes utilisant le Fuzzy Min-Max

Deux programmes utilisant la classe `FuzzyMinMax` ont été réalisés.

minmax_2D	Permet d'utiliser le Fuzzy Min-Max sur des données dans un plan XY.
minmax_souris	Permet d'utiliser le Fuzzy Min-Max pour classer des points cliqués par la souris.

Programme minmax_2D

Le programme `minmax_2D` permet d'utiliser le Fuzzy Min-Max sur des données dans un plan XY. Il lit dans un fichier une série de points enregistrés sous forme de table de table de vecteurs (<Table1D<Table1D<Vecteur>>>), chaque sous-table contenant les vecteurs qui appartiennent à la même classe. Ces points sont normalisés dans l'intervalle [0,1] et sont fournis un à la suite de l'autre à un réseau Fuzzy Min-Max.

Lorsque le fichier de données s'affiche dans une fenêtre, cliquer sur celle-ci pour que le Fuzzy Min-Max démarre son apprentissage. Lorsque la lecture des données est complétée, le classement effectué par le réseau s'affiche dans une autre fenêtre. À ce moment, il y a trois options :

- Cliquer sur le bouton de gauche de la souris fait lire à nouveau les données au réseau.
- Cliquer sur le bouton du centre de la souris fait afficher les frontières apprises par le réseau.
- Cliquer sur le bouton de droite de la souris fait afficher une grille de points montrant comment le réseau classerait ces points dans son état actuel.

Ce programme accepte plusieurs paramètres sur la ligne de commande :

-d *nom*	Nom du fichier contenant les données, `data1.tp2` par défaut.
-P	Permuter les données (à chaque lecture) au lieu de les lire dans l'ordre.
-D *valeur*	Valeur de la dimension maximale des hyper-rectangles, 0.4 par défaut.
-g *valeur*	Valeur de la sensibilité, 4 par défaut.
-p *nom*	Nom du fichier dans lequel lire les paramètres, une alternative à fournir ceux-ci sur la ligne de commande.
-n *valeur*	Nombre maximal d'hyper-rectangles, 10 par défaut.

Code source : `minmax_2D.cpp`.

Programme minmax_souris

Le programme `minmax_souris` permet d'utiliser le Fuzzy Min-Max pour classer des points cliqués par la souris.

Lorsque la fenêtre s'affiche, il y a quatre options :

- Cliquer sur le bouton de gauche de la souris fait lire un nouveau point au réseau.
- Cliquer sur le bouton du centre de la souris fait afficher les frontières apprises par le réseau.
- Cliquer sur le bouton de droite de la souris fait afficher une grille de points montrant comment le réseau classerait ces points dans son état actuel.
- Cliquer sur le bouton du centre et le bouton de gauche représente tous les points cliqués au réseau et réaffiche les frontières.

Ce programme accepte quelques paramètres sur la ligne de commande :

-D *valeur*	Valeur de la dimension maximale des hyper-rectangles, 0.4 par défaut.
-g *valeur*	Valeur de la sensibilité, 4 par défaut.

-p *nom* Nom du fichier dans lequel lire les paramètres, une alternative à fournir ceux-ci sur la ligne de commande.

Code source : `minmax_souris.cpp`.

Ceci complète la section sur l'implantation du Fuzzy Min-Max en C++. Nous avons étudié les classes `FuzzyMinMax` et `HyperRectangle`, puis les programmes qui les utilisent et qui ont permis de faire les démonstrations à la section 3.2.

Partie II

Architectures supervisées

Chapitre 4

LAPART

Le LAPART (LAteral Priming Adaptive Resonance Theory) [8, 9] est un réseau de neurones composé de deux ART1 (chapitre 1). Il fut introduit en 1993 par Healy, Caudell et Smith. Ce réseau apprend des inférences entre les vecteurs qui lui sont présentés par paires. Après un entraînement suffisant, il peut déduire le vecteur qui devrait être associé à un vecteur qui lui est présenté.

Ce réseau peut donc être utilisé pour faire de l'apprentissage supervisé. Il doit être entraîné avec des paires de vecteurs avant de pouvoir être utilisé pour effectuer du classement. Comme le réseau est formé de deux ART1, il n'accepte que des données binaires (0 ou 1).

Nous verrons d'abord l'essentiel de la théorie concernant l'algorithme d'apprentissage du LAPART, puis nous étudierons son fonctionnement à l'aide d'une simulation en C++. Les détails de cette implantation seront vus à la dernière section du chapitre.

4.1 Algorithme d'apprentissage

Le LAPART apprend à l'aide de paires de vecteurs, qui représentent un antécédent et un conséquent. Il peut être utilisé pour faire de l'apprentissage supervisé. Dans ce cas, le premier vecteur représente une donnée et le second identifie la classe à laquelle elle appartient.

La procédure que le réseau utilise est en quelque sorte une vérification d'hy-

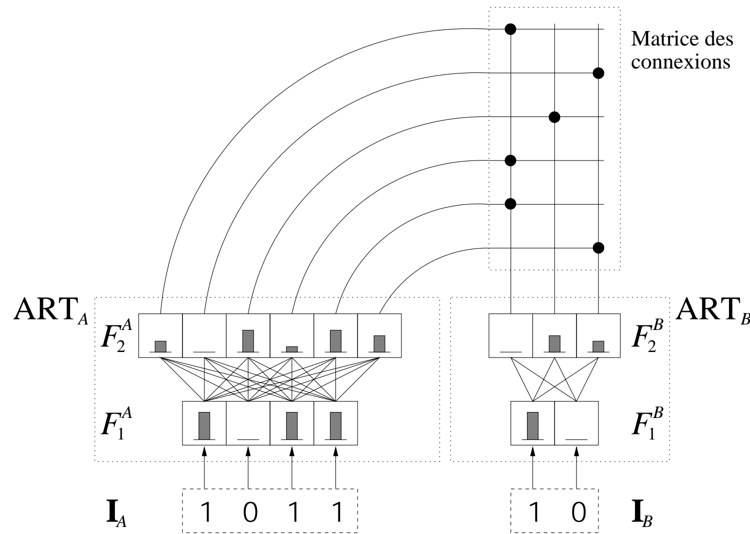


Figure 4.1: *Schéma du réseau LAPART.*

pothèses. Lorsqu'il reçoit le premier vecteur, il déduit quelle devrait être sa classe. Le second vecteur lui permet de confirmer son hypothèse ou de la réfuter, auquel cas le processus recommence avec une hypothèse différente.

La structure du LAPART est présentée, suivie de son principe de fonctionnement, d'un résumé de l'algorithme avec les équations nécessaires et d'une discussion sur les paramètres.

4.1.1 Structure du réseau

Le LAPART est formé de deux réseaux ART1, dénommés ART$_A$ et ART$_B$, reliés par une série de connexions $F_{2,i}^A \rightarrow F_{2,j}^B$ entre les neurones des couches F_2 de ART$_A$ (F_2^A) et de ART$_B$ (F_2^B). Les poids des connexions peuvent être 0 ou 1, selon l'existence ou non d'une connexion entre deux neurones particuliers. Au départ elles sont toutes nulles, et c'est au cours de l'apprentissage que certaines sont mises à 1. Ces connexions sont illustrées symboliquement à la figure 4.1 par de petits cercles noirs dans la matrice des connexions [4].

Les composantes des vecteurs d'entrée doivent être binaires (0 ou 1), mais nous pourrons utiliser la notation *stack interval* pour convertir des données réelles en ce format (voir la section 1.3.3).

4.1.2 Principe de fonctionnement

Nous utilisons le LAPART pour faire de l'apprentissage supervisé, donc le réseau ART_A reçoit un vecteur représentant une donnée, tandis que le ART_B reçoit un vecteur identifiant la classe à laquelle elle appartient. Chaque neurone de la couche F_2^B représente une classe formée par le réseau.

Lorsque le réseau apprend une paire de vecteurs, le premier est d'abord propagé dans le ART_A. S'il n'y a pas de connexion du neurone gagnant vers un neurone de ART_B, le second vecteur est propagé dans ART_B et le neurone gagnant est relié à son homologue du ART_A. S'il y a déjà une connexion, ou bien le réseau l'accepte, ou bien il choisit un nouveau neurone gagnant pour le ART_A.

Lorsque le LAPART a été entraîné avec suffisamment de paires de vecteurs, il peut être utilisé pour classer des données. Pour ce faire, chaque donnée est présentée au ART_A et le réseau retourne comme classe l'indice du neurone sur F_2^B auquel le neurone gagnant sur F_2^A est connecté (chaque neurone de F_2^A est connecté à au plus un neurone de F_2^B). S'il n'existe pas de connexion, le LAPART ne peut associer la donnée à une classe.

C'est, en résumé, le principe de fonctionnement du LAPART. Nous allons voir à l'instant l'algorithme précis que suit ce réseau.

4.1.3 Équations

Voici l'algorithme d'apprentissage du LAPART avec les équations correspondantes.

Les variables de ART_A et ART_B sont identifiées respectivement par l'exposant ou l'indice A et B.

1. Initialiser les poids des connexions w_{ij}^{AB} entre les réseaux ART_A et ART_B.

$$w_{ij}^{AB} = 0 \tag{4.1}$$

   La variable w_{ij}^{AB} représente le poids de la connexion entre le neurone $F_{2,i}^A$ de ART_A et le neurone $F_{2,j}^B$ de ART_B.

2. Appliquer un vecteur d'entrée $\mathbf{I}_A$ au réseau ART_A et un vecteur d'entrée $\mathbf{I}_B$ au réseau ART_B. Propager le vecteur $\mathbf{I}_A$ selon l'algorithme de la section 1.1.4.

3. Vérifier s'il y a une connexion du neurone gagnant $F_{2,I}^{A}$ vers un neurone de la couche F_2^B ($\exists j \mid w_{Ij}^{AB} = 1$).

    - Si oui, poursuivre l'algorithme à l'étape 4.
    - Si non,
        - Propager le vecteur d'entrée $\mathbf{I}_B$ selon l'algorithme de la section 1.1.4.
        - Établir une connexion entre $F_{2,I}^{A}$ et le neurone gagnant de ART$_B$, $F_{2,J}^{B}$.
        $$w_{IJ}^{AB} = 1 \tag{4.2}$$
        - Retourner à l'étape 2 avec de nouveaux vecteurs d'entrée.

4. Effectuer une rétropropagation des sorties dans le ART$_B$ en considérant comme neurone gagnant $F_{2,J}^{B}$ le neurone connecté à $F_{2,I}^{A}$. Les nouvelles sorties $\mathbf{S}_B$ de F_1^B sont
$$\mathbf{S}_B = \mathbf{T}^{B,J} \tag{4.3}$$
Nous notons $\mathbf{T}^{B,j}$ le *top-down template* du neurone $F_{2,j}^{B}$. Selon la notation que nous utilisions pour le ART1 (chapitre 1), $T_i^{B,j} \Leftrightarrow z_{ij}^{B}$.

5. Déterminer s'il y a correspondance entre le vecteur d'entrée $\mathbf{I}_B$ et le *top-down template*, comme à l'étape 6 de l'algorithme du ART1 à la section 1.1.4.

    - Si non ($|\mathbf{S}_B|/|\mathbf{I}_B| < \rho^B$), inhiber les neurones $F_{2,I}^{A}$ et $F_{2,J}^{B}$ pour éviter qu'ils gagnent à nouveau et retourner à l'étape 2 avec les mêmes vecteurs d'entrée.
    - Si oui ($|\mathbf{S}_B|/|\mathbf{I}_B| \geq \rho^B$), continuer.

6. Mettre à jour les poids des deux réseaux ART1.
$$\mathbf{T}^{A,I} = \mathbf{I}_A \wedge \mathbf{T}^{A,I} \tag{4.4}$$
$$\mathbf{T}^{B,J} = \mathbf{I}_B \wedge \mathbf{T}^{B,J} \tag{4.5}$$

7. Retourner à l'étape 2 avec de nouveaux vecteurs d'entrée.

Lorsque le LAPART est utilisé pour classer des données, changer l'étape 3 pour la suivante:

3. Vérifier s'il y a une connexion du neurone gagnant $F_{2,I}^{A}$ vers un neurone de la couche F_2^B ($\exists j \mid w_{Ij}^{AB} = 1$).

   - Si oui, retourner l'indice (j) du neurone de F_2^B.
   - Si non, il est impossible d'identifier la classe.

4.1.4 Paramètres

Le LAPART a comme paramètres ceux des réseaux ART_A et ART_B, soit 4 paramètres au total (L_A, ρ_A, L_B et ρ_B). Voici les valeurs suggérées pour les paramètres, accompagnées des contraintes à respecter pour chacun d'eux.

Paramètre	*Contrainte*	*Valeur suggérée*
L_A	$L_A > 1$	1.1
ρ_A	$0 < \rho_A \leq 1$	0.5
L_B	$L_B > 1$	1.1
ρ_B	$0 < \rho_B \leq 1$	1

Dans le cas de ART_A, les mêmes suggestions que nous avons fait pour le ART1 à la section 1.1.4 s'appliquent. Ce réseau doit agir de la même façon qu'un ART1 seul, c'est-à-dire créer des classes de façon non supervisée. Une vigilance plus élevée, par contre, pourrait être avantageuse comme nous le verrons à la section 4.2.2.

Par contre, il est suggéré pour le ART_B d'utiliser une vigilance de 1. En général, le rôle de ce réseau est d'associer la classe qui correspond à chaque neurone de la couche F_2 de ART_A. Il est essentiel que chaque classe soit distinguée parfaitement des autres. En d'autres mots, il ne faut pas que des vecteurs d'entrée représentant des classes différentes activent le même neurone de F_2^B. C'est pour cette raison que nous suggérons $\rho_B = 1$. Toutefois si le LAPART est utilisé à d'autres fins que l'apprentissage supervisé, comme par exemple l'apprentissage de séquences de patrons [9], ρ_B devrait être choisi de la même façon que ρ_A. Quand au paramètre L_B, la valeur par défaut conviendra.

Nous avons couvert, dans cette section, la structure du LAPART, son algorithme d'apprentissage et ses paramètres. Nous illustrerons maintenant le fonctionnement de ce réseau sur des points dans un plan.

4.2 Fonctionnement

Nous allons appliquer le LAPART au classement de points dans un espace à deux dimensions. Plus d'informations sur l'implantation en C++, grâce à laquelle cette démonstration est possible, sont données à la section 4.3.

Comme pour le ART1, il faut utiliser la notation *stack interval* (section 1.3.3 pour convertir les coordonnées des points en format binaire. Les vecteurs binaires représentant les coordonnées X et Y sont ensuite concaténés pour former le vecteur d'entrée de ART_A.

La classe de chaque point est encodée dans le vecteur d'entrée de ART_B. Le vecteur contient seulement des 0 sauf une composante de valeur 1 à l'indice correspondant au numéro de la classe.

Nous utiliserons les mêmes fichiers de données que pour le ART1, qui sont décrits à la section 1.2.1.

Tout d'abord nous ferons quelques constatations sur le fonctionnement du LAPART avec les paramètres par défaut. Par la suite, nous observerons l'influence de ρ_A ainsi qu'un certain phénomène de capture, puis nous discuterons de l'utilisation du réseau pour effectuer du classsement.

4.2.1 Constatations initiales

Pour commencer, essayons d'entraîner le LAPART à reconnaître le fichier de données #1, avec les paramètres par défaut. Nous voyons, à la figure 4.2 que le réseau a créé une frontière pour la classe 0 et deux frontières pour la classe 1. Chaque frontière est associée à un neurone de F_2^A et les frontières du même niveau de gris sont associées à la même classe (i.e. neurone de F_2^B).

Notons qu'il y a 2 données de la classe 0 qui sont incorrectement placées dans la classe 1 (comparer avec le fichier original à la figure 1.2). Le LAPART est souvent incapable d'apprendre parfaitement les données d'entraînement. Nous y reviendrons à la section 4.2.3.

Le fichier de données #2 est classé correctement par le LAPART à 100% dès la première passe d'entraînement, tel qu'illustré à la figure 4.3. Même chose pour le fichier #5, à la figure 4.4.

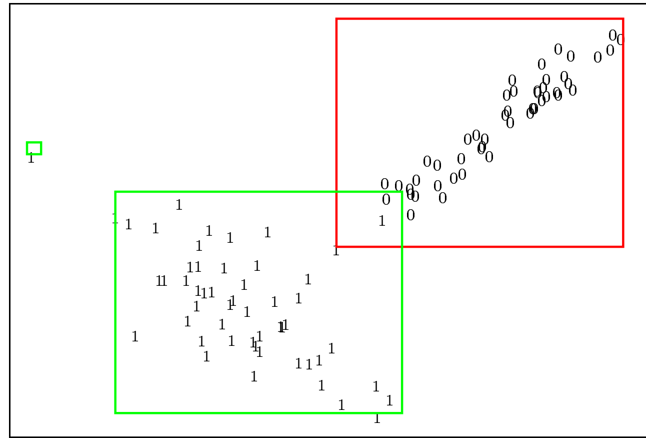


Figure 4.2: *Classement des données #1 ($\rho_A = 0.5$).*

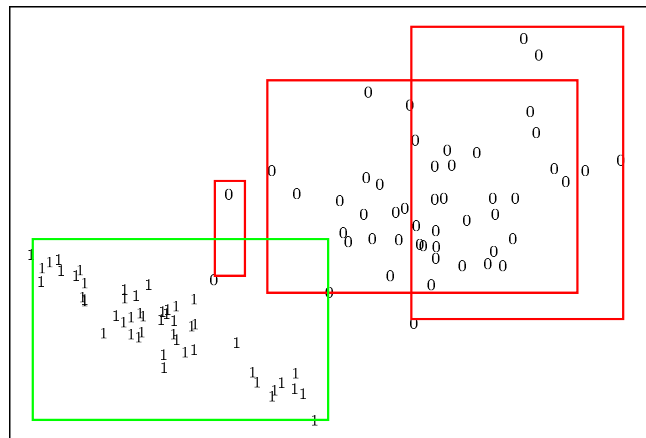


Figure 4.3: *Classement des données #2 ($\rho_A = 0.5$).*

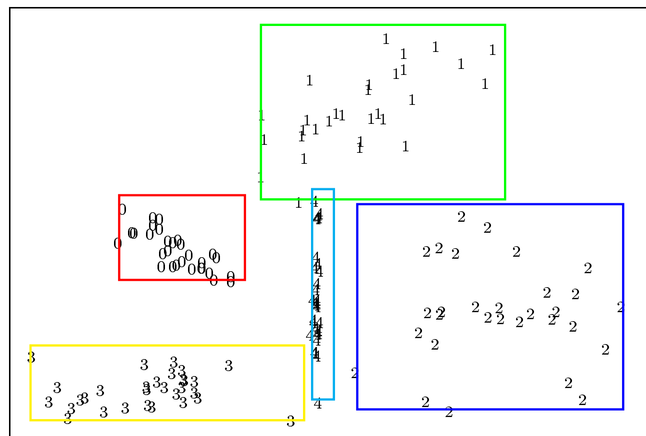


Figure 4.4: *Classement des données #5 ($\rho_A = 0.5$).*

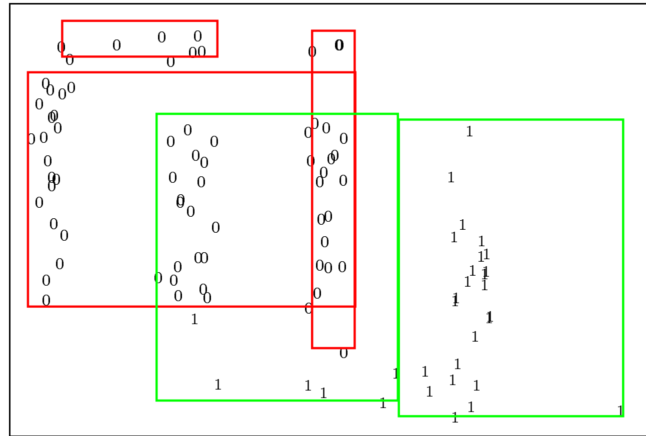


Figure 4.5: *Classement des données #3 ($\rho_A = 0.5$).*

4.2.2 Influence de ρ_A

Tous les fichiers ne se classent pas aussi bien avec les paramètres par défaut. Par exemple, le fichier de données #3 (fig. 4.5) n'est bien classé qu'à 81%. Par contre nous pouvons obtenir de meilleures performances en augmentant la vigilance du réseau ART_A, ρ_A.

En effet, en augmentant ρ_A, les rectangles formés par ART_A sont plus petits et il y a moins de chances de recouvrements. Et les erreurs de classement sont justement causées par des recouvrements entre les frontières (plus de détails là-dessus à la section 4.2.3).

Par défaut, la vigilance ρ_A est égale à 0.5. Seulement en l'augmentant à 0.6, le pourcentage de classement des données #3 passe à 100% (fig. 4.6).

Plus ρ_A est élevé, cependant, plus il y a de neurones qui sont utilisés dans le réseau ART_A. La taille du réseau augmente donc, ce qui demande plus de temps de traitement. À la limite, si $\rho_A = 1$, chaque point serait appris individuellement, ce qui serait inutile en plus d'être un manque de généralité.

Bref, il y a avantage à utiliser un ρ_A assez élevé pour effectuer de meilleurs classements, cependant il faut faire attention de ne pas trop l'augmenter pour éviter les problèmes de sur-apprentissage.

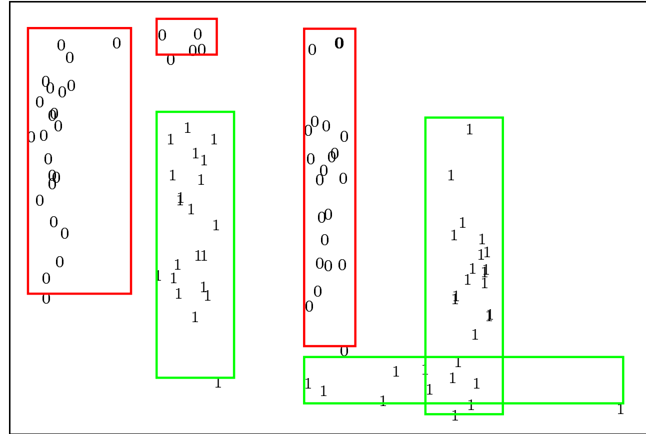


Figure 4.6: *Classement des données #3 ($\rho_A = 0.6$).*

4.2.3 Phénomène de capture

Voyons maintenant pourquoi le LAPART est incapable d'apprendre parfaitement les données d'entraînement.

Les erreurs de classement sont dues à des recouvrements entre les frontières de classes différentes. Par exemple, c'est dans l'espace où il y a un recouvrement entre les classes 0 et 1 que les données des fichiers #1 et #3 sont mal classées (figures 4.2 et 4.5). Un exemple simple nous permettra de comprendre ce qui se produit.

Supposons qu'une donnée est placée dans la classe 0 par le réseau alors qu'elle appartient à la classe 1, et qu'elle se situe dans une zone où les classes 0 et 1 se recouvrent. Nous avons remarqué que le LAPART est incapable de corriger le classement de la donnée même si nous effectuons d'autres passes d'entraînement. L'explication vient du fait qu'en présentant à nouveau cette donnée, le neurone gagnant sur F_2^A est inhibé car il n'y a pas correspondance dans le ART$_B$ (voir l'étape 5 de l'algorithme), celui-ci étant associé à la classe 0. Or, le prochain neurone gagnant sur F_2^A, à cause du recouvrement, sera un neurone relié à celui représentant la classe 1 sur F_2^B et il y aura correspondance dans le ART$_B$. À ce moment, on met à jour les poids des réseaux pour apprendre la donnée, mais il n'y a aucun changement car celle-ci est déjà à l'intérieur de la frontière de la classe 1. Par la suite, le neurone associé à la classe 0 sera encore et toujours activé avant celui de la classe 1.

Bref, on pourrait décrire cette situation comme un cas où une ou plusieurs données sont capturées par la mauvaise classe à cause d'un recouvrement entre

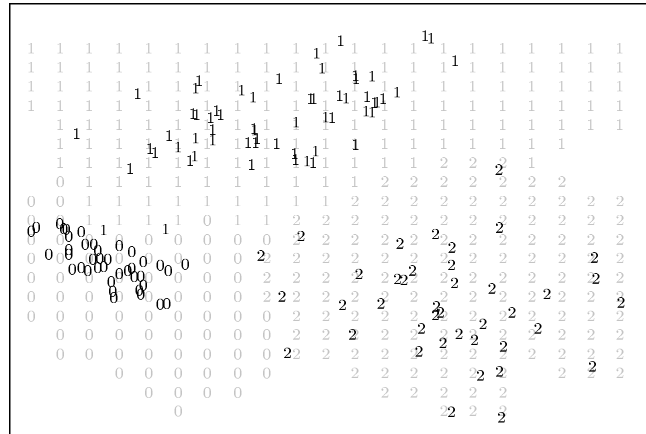


Figure 4.7: *Classement de données après entraînement avec les données #4 avec* $\rho_A = 0.8$.

les frontières. C'est ce phénomène de capture qui peut faire que le LAPART est incapable d'apprendre parfaitement les données d'entraînement.

4.2.4 Mode classement

Lorsque le LAPART a été entraîné, nous souhaitons l'utiliser purement pour classer des données. Nous visualiserons son comportement en lui faisant classer une série de données qui formeront une grille de points couvrant le plan.

Prenons le fichier #4, qui est bien classé à 100% avec $\rho_A = 0.8$. Nous voyons à la figure 4.7 que le réseau sépare relativement bien les données. Par contre, à partir d'une certaine distance, le réseau n'est plus capable de classer les données car elles sont trop éloignées des frontières (au bas, notamment).

Nous sommes confrontés à ce problème seulement parce que la vigilance de ART_A est trop élevée. Comme l'entraînement et le classement sont deux modes distincts d'utilisation du LAPART, nous pourrions diminuer ρ_A près de 0 (0.01, par exemple) lorsque le réseau est utilisé en mode classement.

La figure 4.8 présente le nouveau classement qui est clairement meilleur que le premier. En plus qu'il n'y a pas de points non classés, les séparations entre les classes sont mieux centrées entre les groupes de points. À l'avenir nous fixerons toujours $\rho_A = 0.01$ lorsque le LAPART sera utilisé pour classer des données.

Nous comprenons maintenant mieux le fonctionnement du LAPART après

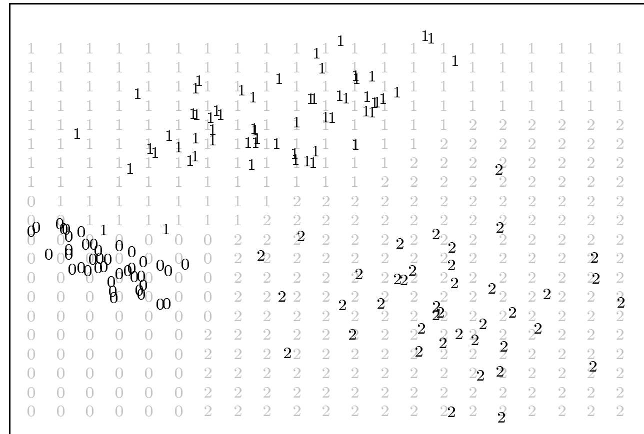


Figure 4.8: *Classement de données ($\rho_A = 0.01$) après entraînement avec les données #4 avec $\rho_A = 0.8$.*

avoir analysé le classement qu'il effectue avec des points dans un plan. Nous avons vu les résultats que donne le réseau avec les paramètres par défaut, puis nous avons étudié l'influence de la vigilance de ART$_A$. Par la suite nous avons discuté du phénomène de capture et du mode classement. Nous sommes maintenant prêts à passer à la réalisation concrète du LAPART en C++.

4.3 Implantation

Cette section porte sur l'implantation en C++ du réseau LAPART. Nous verrons d'abord la classe `LAPART` elle-même, puis le programme qui l'utilise et qui a servi à illustrer le fonctionnement du réseau à la section 4.2.

Le code source de cette implantation est disponible sur le réseau du département de génie électrique et de génie informatique de l'Université Laval, dans les mêmes répertoires que le ART1 (voir la section 1.3, p.30).

4.3.1 Classe LAPART

La classe `LAPART` permet de gérer un réseau de neurones de type LAPART dont la structure est décrite à la section 4.1.1.

Le LAPART est formé de deux réseaux ART1 (voir la classe `ART1` à la section

1.3.1), `artA` et `artB`, et d'une matrice[1] de poids des connexions entre ces deux réseaux. Les ART1 sont des variables publiques et il est possible d'accéder à leurs fonctions, par exemple pour changer leurs paramètres.

Code source : `LAPART.hpp` et `LAPART.cpp`.

Constructeur

Le constructeur de la classe permet de créer la structure du réseau. Il possède quatre paramètres :

```
LAPART(int _d1A=5, int _d2A=6, int _d1B=5, int _d2B=6);
```

Les paramètres *_d1A* et *_d2A* sont les dimensions des couches de ART_A tandis que *_d1B* et *_d2B* sont les dimensions des couches de ART_B.

Les paramètres relatifs à chacun des ART1 sont initialisés à leur valeur par défaut (voir la section 1.3.1), sauf ρ_B qui est initialisé à 1. Les poids des connexions entre les deux réseaux sont initialisés à 0.

Opérateurs

La classe `LAPART` possède les opérateurs suivants :

- `friend ostream &operator<<(ostream &_os, const LAPART &_lap);`

  Insère le réseau *_lap* dans le flot de sortie *_os* (voir `Ecrire`).

- `friend istream &operator>>(istream &_is, LAPART &_lap);`

  Extrait le réseau *_lap* dans le flot d'entrée *_is* (voir `Lire`).

Fonctions d'interface

La classe `LAPART` possède les fonctions d'interface suivantes :

[1]La classe `Matrice` utilisée est celle créée par Marc Parizeau et est disponible à `www.gel.ulaval.ca/~parizeau/C++/classes/doc/Classes/Classes.html`.

- `void Agrandissement(int _m=1);`

  Le réseau a un mode d'agrandissement automatique, actif par défaut, qui permet aux couches F_2 de s'agrandir si tous les neurones de l'une d'entre elles sont utilisés. La fonction `Agrandissement` contrôle ce mode. Le paramètre *_m* doit être 0 ou 1, 0 signifiant que l'agrandissement automatique est désactivé.

- `void Apprentissage(int _mode=1);`

  Permet de désactiver (ou d'activer) l'apprentissage du réseau, c'est-à-dire de l'empêcher (ou de lui permettre) de modifier les poids de ses connexions. Normalement l'apprentissage est actif, mais il peut être utile, dans certains cas, de le désactiver. Le paramètre *_m* doit être 0 ou 1, 0 signifiant que l'apprentissage est désactivé.

- `int Connexion(int _na, int _nb);`

  Vérifie s'il y a une connexion entre le neurone *_na* de ART_A et le neurone *_nb* de ART_B. Il y a une connexion si le poids est égal à 1 et il n'y en a pas s'il est égal à 0.

- `void Ecrire(ostream &_os=cout);`

  Écrit le réseau, y compris les paramètres, dans le flot de sortie *_os*. Par défaut, *cout* est employé.

- `void Lire(istream &_is=cin);`

  Lit le réseau à partir du flot d'entrée *_is*. Par défaut, *cin* est employé.

- `void LireParam(istream &_is=cin);`

  Lit les paramètres du réseau à partir du flot d'entrée *_is*. Par défaut, *cin* est employé. Le fichier peut contenir n'importe quel paramètre, inscrit sous la forme `nom: valeur`, où `nom` peut être `lA`, `rhoA`, `lB` ou `rhoB` (par exemple `rhoB: 1`). Si un des paramètres n'est pas spécifié, il est initialisé à sa valeur par défaut.

- `int PropagerA(const Vecteur &_entree);`

  Propage le vecteur *_entree* dans le réseau ART_A et retourne l'indice du neurone de F_2^B représentant la classe du vecteur. Si le réseau est incapable de classer le vecteur, la fonction retourne -1. La vigilance ρ_A est diminuée à 0.01 avant d'effectuer la propagation.

- `void PropagerAB(const Vecteur &_eA, const Vecteur &_eB);`

Place les vecteurs $_eA$ et $_eB$ en entrée des réseaux ART$_A$ et ART$_B$, respectivement, et effectue l'algorithme d'appentissage décrit à la section 4.1.3.

- `int RetournerConnexion(int _i);`

  Retourne l'indice du neurone de F_2^B connecté avec le neurone $_i$ de la couche F_2^A. Si aucun neurone de F_2^B n'est connecté à celui-ci, la fonction retourne -1.

Exemple d'utilisation

Le programme suivant crée un réseau LAPART avec 4 entrées et 10 sorties pour le ART$_A$, et 1 entrée et 2 sorties pour le ART$_B$. La vigilance de ART$_A$ est fixée à 0.6 (les autres paramètres ont leur valeur par défaut). On crée deux paires de vecteurs et on les propage dans le réseau. Ensuite on lui fait classer un nouveau vecteur.

```
#include "LAPART.hpp"

int main() {
  LAPART res(4,10,1,2);
  res.artA.Vigilance(0.6);
  Vecteur va(4), vb(1);
  va.Initialiser(0); va(0)=1;     // va: (1,0,0,0)
  vb(0)=0;                        // vb: (0)
  res.PropagerAB(va,vb);
  va.Initialiser(1);              // va: (1,1,1,1)
  vb(0)=1;                        // vb: (1)
  res.PropagerAB(va,vb);
  va.Initialiser(0);              // va: (0,0,0,0)
  int classe = res.PropagerA(va);
  cout << "La classe du vecteur est " << classe << endl;
}
```

4.3.2 Programme utilisant le LAPART

Un seul programme utilisant la classe `LAPART` a été réalisé. Il se nomme `lapart_2D`.

Programme lapart_2D

Le programme `lapart_2D` permet d'utiliser le LAPART sur des données dans un plan XY. Il lit dans un fichier une série de points enregistrés sous forme de table de table de vecteurs (`<Table1D<Table1D<Vecteur>>>`), chaque sous-table contenant les vecteurs qui appartiennent à la même classe. Ces points sont convertis en *stack interval* (voir section 1.3.3, p.38) et sont fournis un à la suite de l'autre à un réseau LAPART.

Lorsque le fichier de données s'affiche dans une fenêtre, cliquer sur celle-ci pour que le LAPART démarre son apprentissage. Lorsque la lecture des données est complétée, le classement effectué par le réseau s'affiche dans une autre fenêtre. À ce moment, il y a trois options :

- Cliquer sur le bouton de gauche de la souris fait lire à nouveau les données au réseau.
- Cliquer sur le bouton du centre de la souris fait afficher les frontières apprises par le réseau.
- Cliquer sur le bouton de droite de la souris fait afficher une grille de points montrant comment le réseau classerait ces points dans son état actuel.

Ce programme accepte plusieurs paramètres sur la ligne de commande :

-d *nom*	Nom du fichier contenant les données, `data1.tp2` par défaut.
-P	Permuter les données (à chaque lecture) au lieu de les lire dans l'ordre.
-p *nom*	Nom du fichier dans lequel lire les paramètres, une alternative à fournir ceux-ci sur la ligne de commande.
-m *valeur*	Nombre d'entrées de ART$_A$ (dimension de la couche F_1^A), 320 par défaut.
-n *valeur*	Nombre de sorties de ART$_A$ (dimension de la couche F_2^A), 30 par défaut.

Code source : `lapart_2D.cpp`.

Chapitre 5

Fuzzy ARTMAP

Le Fuzzy ARTMAP est un réseau de neurones composé de deux Fuzzy ART (chapitre 2). Il fut introduit en 1992 par Carpenter et al. [2]. Sa structure se rapproche de celle du LAPART (chapitre 4), avec des Fuzzy ART au lieu des ART1 (chapitre 1). Ce réseau effectue de l'apprentissage supervisé, c'est-à-dire qu'il apprend à séparer des données appartenant à des classes connues. Il reçoit des paires de vecteurs, un qui représente une donnée et l'autre la classe à laquelle celle-ci appartient.

Nous verrons, pour commencer, un résumé de l'algorithme d'apprentissage du Fuzzy ARTMAP. Ensuite, comme nous l'avons fait pour les autres réseaux, nous illustrerons son fonctionnement à l'aide d'une implantation de celui-ci en C++. Nous verrons finalement les détails de cette implantation à la dernière section du chapitre.

5.1 Algorithme d'apprentissage

Le Fuzzy ARTMAP est un réseau très efficace pour effectuer de l'apprentissage supervisé. Il est en mesure d'apprendre parfaitement des données d'entraînement en très peu de passes.

Cette section porte sur la structure du réseau, les équations gérant son fonctionnement ainsi que les paramètres qui le contrôlent.

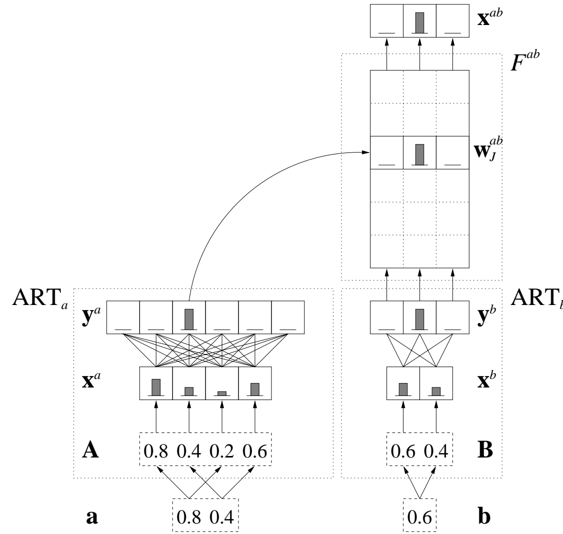


Figure 5.1: *Schéma du réseau Fuzzy ARTMAP.*

5.1.1 Structure du réseau

Le Fuzzy ARTMAP est composé de deux réseaux Fuzzy ART, dénommés ART_a et ART_b, reliés par une série de connexions entre les neurones des couches F_2 de ART_a et ART_b. Les connexions sont pondérées, un poids w_{ij} entre 0 et 1 étant associé à chacune d'elles. L'ensemble de ces connexions est nommé *map field*, F^{ab}. Le *map field* a ses paramètres (β_{ab} et ρ_{ab}) et un vecteur de sortie $\mathbf{x}^{ab}$.

Les vecteurs d'entrée **a** et **b** des réseaux ART_a et ART_b subissent un codage complémentaire (voir la section 2.1.1, p.42). On identifie les vecteurs résultants, de la même dimension que la couche F_1 de leur réseau respectif, par **A** et **B**. La structure du Fuzzy ARTMAP est illustrée à la figure 5.1. Les poids des connexions du *map field* F^{ab} sont représentés par des carrés contenant une barre verticale plus ou moins haute selon la grandeur du poids.

La vigilance du réseau ART_a varie au cours de l'apprentissage. On identifie la vigilance initiale de ART_a (la vigilance *de base*) par $\overline{\rho_a}$.

5.1.2 Principe de fonctionnement

Nous utilisons le Fuzzy ARTMAP pour faire de l'apprentissage supervisé, donc le réseau ART_a reçoit un vecteur représentant une donnée, tandis que le ART_b

reçoit un vecteur identifiant la classe à laquelle elle appartient. Chaque neurone de la couche F_2^b représente une classe formée par le réseau.

Lorsque le réseau apprend une paire de vecteurs, ceux-ci sont d'abord propagés dans leur Fuzzy ART respectif. Le réseau compare alors les poids du *map field* reliant le neurone gagnant de ART$_a$ à la couche F_2^b et les activités des neurones sur F_2^b. S'ils sont suffisamment semblables, le vecteur de poids du neurone gagnant de ART$_a$ tend à égaler le vecteur d'activités des neurones sur F_2^b (d'autant plus rapidement que β_{ab} est grand). Sinon, la vigilance de ART$_a$ est augmentée juste assez pour que ART$_a$ choisisse un nouveau neurone gagnant et le même vecteur est repropagé dans ART$_a$.

Lorsque le Fuzzy ARTMAP a été entraîné avec suffisamment de paires de vecteurs, il peut être utilisé pour classer des données. Pour ce faire, chaque donnée est présentée au ART$_a$ et le réseau met le vecteur de sortie $\mathbf{x}^{ab}$ égal au vecteur de poids du *map field* reliant ce neurone à la couche F_2^b. Le réseau retourne comme classe l'indice de la composante du vecteur $\mathbf{x}^{ab}$ ayant l'activité la plus forte.

C'est, en résumé, le principe de fonctionnement du Fuzzy ARTMAP. Nous allons voir à l'instant l'algorithme précis que suit ce réseau.

5.1.3 Équations

Voici l'algorithme d'apprentissage du Fuzzy ARTMAP avec les équations correspondantes. Les variables de ART$_a$ et ART$_b$ sont identifiées respectivement par l'exposant ou l'indice a et b.

1. Initialiser les paramètres en respectant les contraintes suivantes:

$$\beta_{ab} \in [0, 1]$$

$$\rho_{ab} \in]0, 1]$$

2. Initialiser les poids w_{jk}^{ab} des connexions du *map field* F^{ab}.

$$w_{jk}^{ab} = 1 \tag{5.1}$$

   La variable w_{jk}^{ab} représente le poids de la connexion entre le j^{e} neurone de F_2^a et le k^{e} neurone de F_2^b.

La suite de l'algorithme diffère si le réseau est entraîné ou s'il est utilisé pour classer :

- Si on propage un vecteur dans ART$_a$ et un vecteur dans ART$_b$ (entraînement) :

3. Appliquer un vecteur d'entrée $\mathbf{a}$ au ART$_a$ et un vecteur d'entrée $\mathbf{b}$ au ART$_b$ et les propager selon l'algorithme de la section 2.1.2, p.43 (en s'assurant de faire le codage complémentaire).
4. Calculer les sorties $\mathbf{x}^{ab}$ du *map field.*

$$\mathbf{x}^{ab} = \mathbf{y}^b \wedge \mathbf{w}_J^{ab} \tag{5.2}$$

Le vecteur $\mathbf{y}^b$ est le vecteur d'activités de la couche F_2^b. Le vecteur $\mathbf{w}_J^{ab}$ correspond aux poids des connexions du J^{e} neurone de F_2^a. L'indice J est celui du neurone gagnant sur F_2^a.

5. Déterminer s'il y a correspondance entre le vecteur $\mathbf{x}^{ab}$ et le vecteur $\mathbf{y}^b$.
   - Si $|\mathbf{x}^{ab}|/|\mathbf{y}^b| < \rho_{ab}$, augmenter ρ_a juste assez pour que ART$_a$ choisisse un nouveau neurone gagnant ($\rho_a > |\mathbf{A} \wedge \mathbf{w}_J^a|/|\mathbf{A}|$) et retourner à l'étape 3.
   - Si $|\mathbf{x}^{ab}|/|\mathbf{y}^b| \geq \rho_{ab}$, continuer.
6. Mettre à jour les poids du *map field.*

$$\mathbf{w}_J^{ab} = \beta_{ab}\mathbf{x}^{ab} + (1 - \beta_{ab})\mathbf{w}_J^{ab} \tag{5.3}$$

- Si on propage un vecteur dans ART$_a$ seulement (classement) :

3. Appliquer un vecteur d'entrée $\mathbf{a}$ au réseau et le propager selon l'algorithme de la section 2.1.2, p.43 (en s'assurant de faire le codage complémentaire). La vigilance ρ_a est temporairement mise à zéro[1].
4. Calculer les sorties $\mathbf{x}^{ab}$ du *map field.*

$$\mathbf{x}^{ab} = \mathbf{w}_J^{ab} \tag{5.4}$$

L'indice J est celui du neurone gagnant sur F_2^a.

- Si on propage un vecteur dans ART$_b$ seulement (peu utile) :

3. Appliquer un vecteur d'entrée $\mathbf{b}$ au réseau et le propager selon l'algorithme de la section 2.1.2, p.43 (en s'assurant de faire le codage complémentaire).
4. Calculer les sorties $\mathbf{x}^{ab}$ du *map field.*

$$\mathbf{x}^{ab} = \mathbf{y}^b \tag{5.5}$$

[1]Nous faisons cela pour les mêmes raisons que nous avons expliquées à la section 4.2.4 (p.96) pour le LAPART.

5.1.4 Paramètres

Voyons l'utilité des paramètres du *map field* et leur effet sur le fonctionnement du réseau. L'effet des paramètres de l'un ou l'autre des Fuzzy ART est décrit à la section 2.1.3, p.44.

- Taux d'apprentissage β_{ab}

  Le taux d'apprentissage β_{ab} est utilisé lors de la mise à jour des poids des connexions du *map field* (éq. 5.3). Il influence la vitesse à laquelle les poids sont modifiés lors de l'apprentissage. Plus β_{ab} diminue, plus les poids varient lentement.

- Vigilance ρ_{ab}

  La vigilance sert de critère pour déterminer s'il y a correspondance entre le vecteur de poids du neurone gagnant de ART$_a$ et le vecteur d'activités de F_2^b (voir l'étape 5). En d'autres mots, ρ_{ab} sert à déterminer si la classe présumée par le ART$_a$ correspond à la classe réelle.

  Si le réseau est en *fast learning* ($\beta_{ab} = 1$), ρ_{ab} n'a pas d'effet car le degré de correspondance sera toujours égal soit à 0, soit à 1.

Valeurs suggérées pour les paramètres

Voici maintenant les valeurs suggérées pour les paramètres, accompagnées des contraintes à respecter pour chacun d'eux.

Paramètre	*Contrainte*	*Valeur suggérée*
β_{ab}	$\beta_{ab} \in [0, 1]$	1
ρ_{ab}	$\rho_{ab} \in]0, 1]$	1
α_a	$\alpha_a > 0$	0.01
β_a	$\beta_a \in [0, 1]$	1
ρ_a	$\rho_a \in [0, 1]$	0.5
α_b	$\alpha_b > 0$	0.01
β_b	$\beta_b \in [0, 1]$	1
ρ_b	$\rho_b \in [0, 1]$	1

Nous suggérons un taux d'apprentissage du *map field* $\beta_{ab} = 1$ pour que le réseau fasse de l'apprentissage rapide. À ce moment, la valeur de ρ_{ab} n'a pas d'importance mais nous la fixerons à 1.

Dans le cas de ART_a, les mêmes suggestions que nous avons fait pour le Fuzzy ART à la section 2.1.3 (p.44) s'appliquent. Ce réseau doit agir de la même façon qu'un Fuzzy ART seul, c'est-à-dire créer des classes de façon non supervisée. Une vigilance plus élevée, par contre, pourrait être avantageuse comme nous le verrons à la section 5.2.1.

Pour le ART_b, il est suggéré d'utiliser une vigilance de 1. Le rôle de ce réseau est d'associer la classe qui correspond à chaque neurone de la couche F_2^a. Il est essentiel que chaque classe soit distinguée parfaitement des autres. En d'autres mots, il ne faut pas que des vecteurs d'entrée représentant des classes différentes activent le même neurone de F_2^b. C'est pour cette raison que nous suggérons $\rho_b = 1$. Quand aux paramètres α_b et β_b, la valeur par défaut conviendra.

Nous avons couvert, dans cette section, la structure du Fuzzy ARTMAP, son algorithme d'apprentissage et ses paramètres. Nous illustrerons maintenant le fonctionnement de ce réseau sur des points dans un plan.

5.2 Fonctionnement

Nous allons appliquer le Fuzzy ARTMAP au classement de points dans un espace à deux dimensions. Plus d'informations sur l'implantation en C++, grâce à laquelle cette démonstration est possible, sont données à la section 5.3.

Le ART_a a deux entrées, représentant chacune des coordonnées X et Y. Pour respecter les contraintes du Fuzzy ART, les coordonnées ont d'abord été normalisées entre 0 et 1.

La classe de chaque point est encodée dans le vecteur d'entrée de ART_b. Ce vecteur a une seule composante dont la valeur est l'indice de la classe divisé par le nombre de classes moins 1. Les indices des classes vont de 0 au nombre de classes moins 1. Par exemple, s'il y a 2 classes, les vecteurs d'entrée de ART_b pour ces classes seront 0 et 1. S'il y a 3 classes, les vecteurs seront 0, 0.5 et 1.

Nous utiliserons les mêmes fichiers de données que pour le ART1, qui sont décrits à la section 1.2.1, p.16.

Nous allons d'abord voir l'influence du paramètre $\overline{\rho_a}$ sur le classement effectué par le réseau. Ensuite nous étudierons la capacité du Fuzzy ARTMAP à apprendre des classes en séquence, et celle d'en apprendre une nouvelle séparément des autres. Finalement, nous regarderons ce que peut donner le réseau sur deux spirales l'une

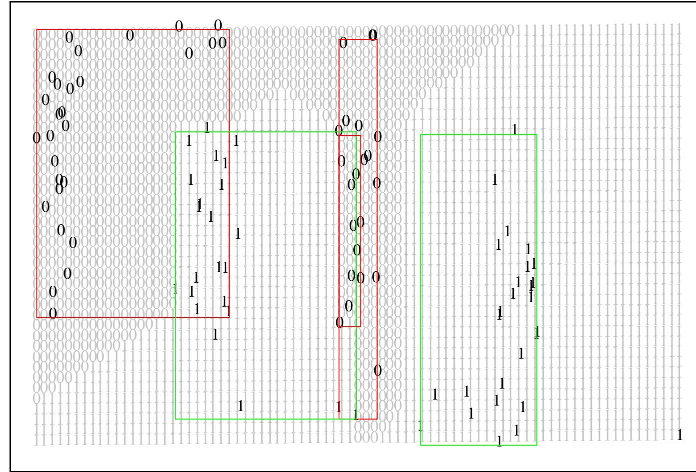


Figure 5.2: *Classement de données après entraînement avec les données #3 avec* $\overline{\rho_a} = 0.5$.

dans l'autre.

5.2.1 Influence de $\overline{\rho_a}$

La méthode de classement utilisée par le Fuzzy ARTMAP est similaire à celle du LAPART. Chaque neurone du réseau ART_a représente un rectangle cernant un ensemble de données. Le réseau ART_b associe la classe correpondant aux rectangles créés par le ART_a. Chaque classe est donc représentée par plusieurs rectangles.

La vigilance de ART_a, ρ_a, influence la précision des rectangles délimitant les classes. Ce paramètre varie au cours de l'apprentissage. La vigilance initiale que nous fixons est appelée vigilance *de base* et est identifiée par $\overline{\rho_a}$. Plus $\overline{\rho_a}$ est grand, plus les rectangles sont petits et vice-versa.

Prenons par exemple les données #3 avec $\overline{\rho_a} = 0.5$. Il faut deux passes pour que le Fuzzy ARTMAP classe les données à 100%. Après l'avoir entraîné, nous lui faisons classer une série de données formant une grille de points couvrant le plan. Nous voyons à la figure 5.2 que les deux classes sont bien divisées, et ce avec seulement 6 rectangles, soit 3 pour chaque classe.

Nous avons présenté les données dans l'ordre du fichier, soit une classe à la fois. Si nous entraînons le réseau avec les données dans un ordre aléatoire, le résultat est la plupart du temps moins intéressant. La figure 5.3 en montre un exemple, où il a fallu 2 passes d'entraînement pour un classement parfait des données #3. La

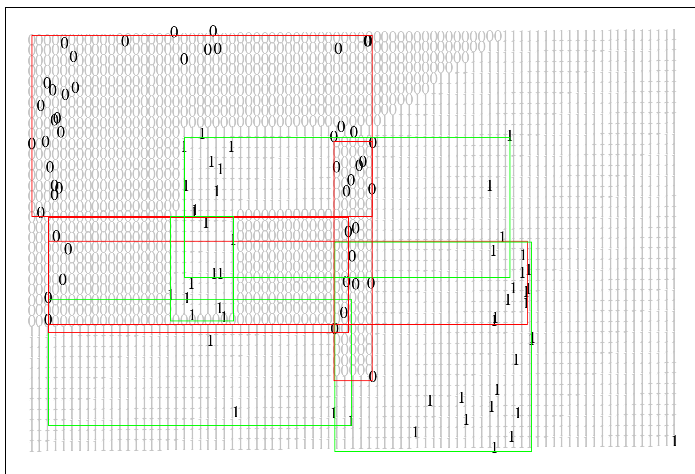


Figure 5.3: *Classement de données après entraînement avec les données #3 avec* $\overline{\rho_a} = 0.5$ *et permutation aléatoire.*

grille de points nous fait voir que les deux classes sont plus ou moins bien divisées par le Fuzzy ARTMAP.

Pour réussir à obtenir de très bons résultats en entraînant le réseau avec des données dans un ordre quelconque, il faut augmenter $\overline{\rho_a}$. Comme nous l'avions mentionné à la section 4.2.2 (p. 94) pour le LAPART, plus la vigilance de ART_a est élevée, plus il y a de neurones utilisés dans ce réseau. Il faut donc faire attention de ne pas trop augmenter la vigilance $\overline{\rho_a}$ si on veut avoir une certaine rapidité de traitement de la part du Fuzzy ARTMAP. Mais comme il est structuré de façon à être efficace et à converger rapidement, il faudra se méfier davantage du sur-apprentissage[2] que de la lenteur quand $\overline{\rho_a}$ sera grand.

Observons le travail du Fuzzy ARTMAP entraîné avec $\overline{\rho_a} = 0.9$ avec les 5 fichiers de données à la figure 5.4. Les données ont été présentées dans un ordre aléatoire à chaque passe d'entraînement jusqu'à ce que le pourcentage de bon classement soit de 100%. Ceci a été réalisé en une seule passe, sauf dans le cas du fichier #2 qui en a demandé 2. Selon l'ordre de présentation, les autres fichiers peuvent parfois nécessiter 2 passes d'entraînement, mais rarement plus. La grille de points en gris pâle nous montre que le réseau a dans tous les cas formé de belles frontières pour diviser les classes.

Il semble donc avantageux d'utiliser une vigilance de base $\overline{\rho_a}$ assez élevée, aux environs de 0.9.

[2]Le sur-apprentissage est la spécialisation du réseau à reconnaître les données d'entraînement, ce qui n'améliore généralement pas le classement qu'il effectue.

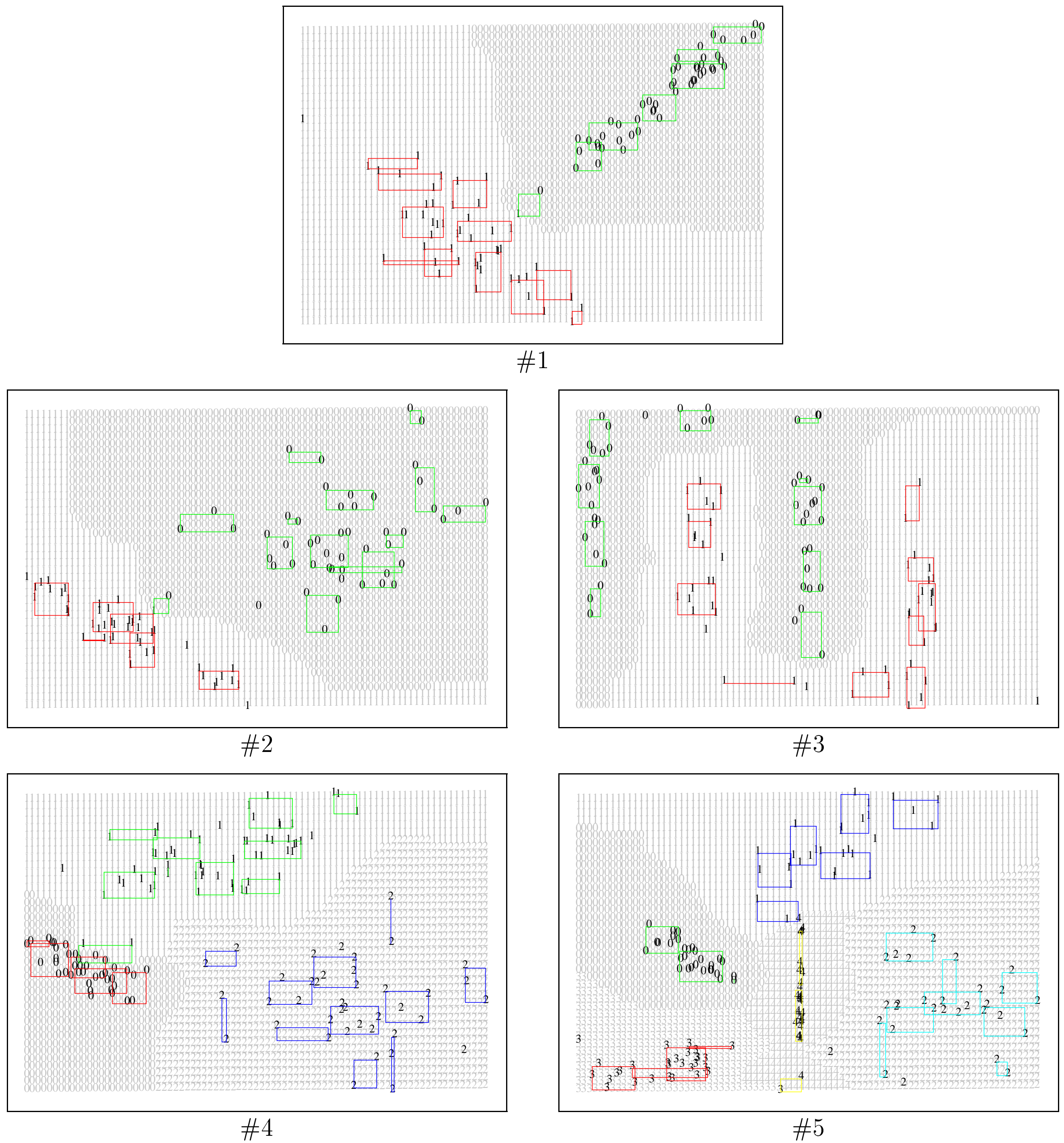


Figure 5.4: *Classement de données après entraînement avec chacun des fichiers avec $\overline{\rho_a} = 0.9$ et permutation aléatoire.*

5.2.2 Apprentissage en séquence

Une caractéristique recherchée chez les réseaux de neurones qui font de l'apprentissage est la capacité d'apprendre de nouvelles données sans avoir à être réentraîné avec les autres. Le Fuzzy ARTMAP a cette capacité, si les paramètres sont choisis correctement.

Nous allons entraîner un réseau Fuzzy ARTMAP avec les données #5 en lui présentant une classe à la fois. Nous avons choisi ce fichier car il est probablement le plus difficile à classer parmi ceux que nous avons. Nous utiliserons $\overline{\rho_a} = 0.9$. Nous pouvons suivre l'apprentissage de chacune des classes à la figure 5.5.

Les données de chaque nouvelle classe sont présentées autant de fois que nécessaire pour qu'elles soient apprises correctement à 100%. Chacune des classes de 0 à 3 est apprise à 100% dès la première passe, pendant que les données présentées auparavant demeurent classées parfaitement. La classe 4 est apprise à 80% après la première passe et en nécessite une deuxième pour être apprise parfaitement. Lorsque cette dernière passe a été effectuée, nous constatons que le réseau classe encore à la perfection le reste des données.

Il est donc possible de faire apprendre de nouvelles classes au Fuzzy ARTMAP sans nuire aux classes déjà existantes. Il faut bien sûr supposer que les classes sont suffisamment bien séparées et que la vigilance de base est assez élevée.

5.2.3 Apprentissage d'une nouvelle classe

Dans la même optique que la section précédente, nous allons vérifier si le Fuzzy ARTMAP est en mesure d'apprendre une nouvelle classe, mais cette fois après avoir appris plusieurs classes simultanément, et non en séquence.

Nous utiliserons à nouveau les données #5, desquelles nous exclurons une des 5 classes. Le réseau sera entraîné avec le reste des données dans un ordre aléatoire, jusqu'à un classement parfait. Ensuite nous ferons apprendre au réseau la classe exclue et nous regarderons si le classement des autres données a été affecté.

Nous avons illustré aux figures 5.6 et 5.7 deux exemples du classement avant et après avoir appris la classe exclue, soit les classes 2 et 4 respectivement. Dans le premier cas, seules 3 données de la classe 4 sont erronées, pour un pourcentage de bon classement de 90% pour cette classe. Dans le second cas, le réseau apprend la classe exclue sans que le classement du reste des données ne soit modifié.

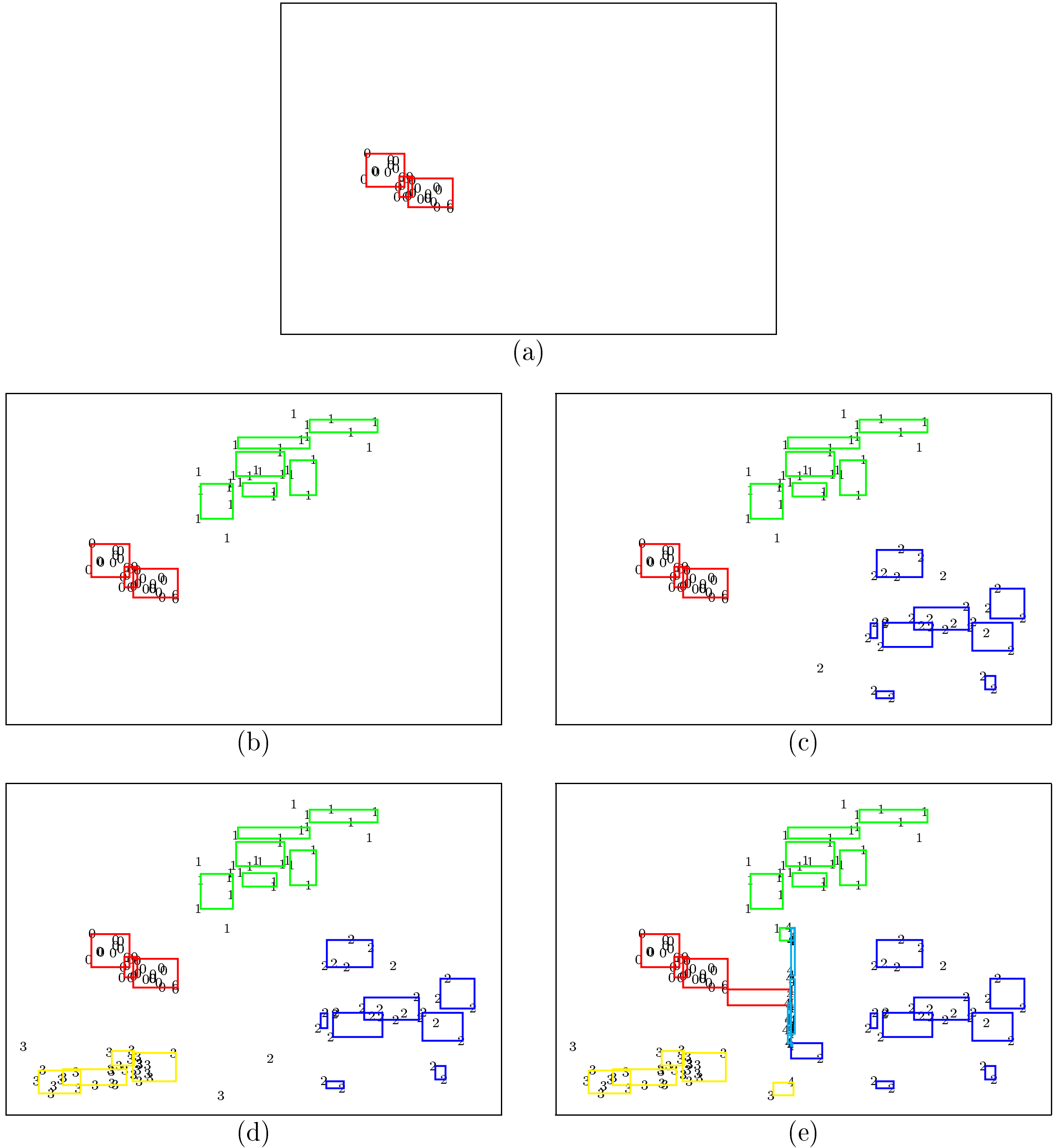


Figure 5.5: *Apprentissage des données #5 une classe à la fois avec* $\overline{\rho_a} = 0.9$. *(a) Classe 0. (b) Classe 1. (c) Classe 2. (d) Classe 3. (e) Classe 4.*

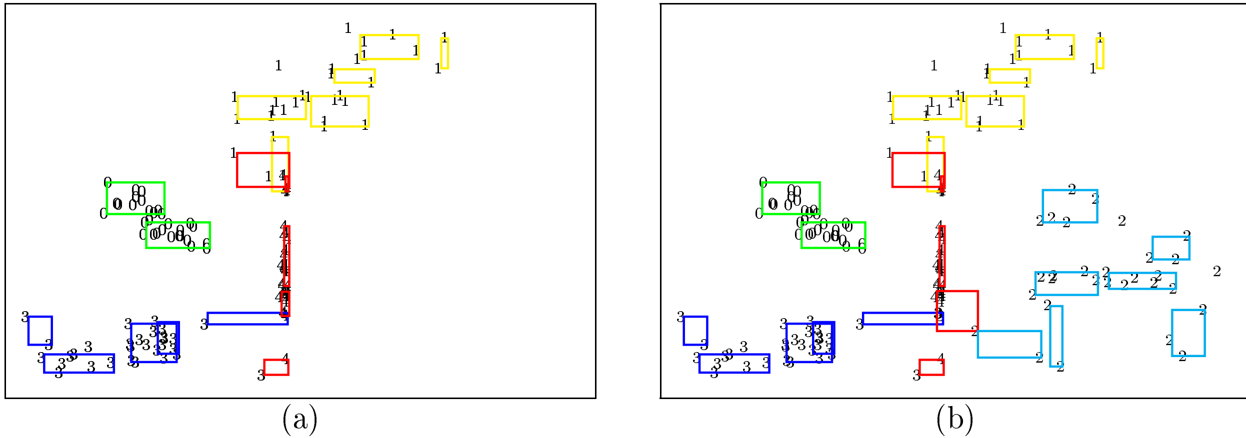


Figure 5.6: *Apprentissage des données #5 en excluant la classe 2 avec* $\overline{\rho_a} = 0.9$ *et permutation aléatoire. (a) Avant l'apprentissage de la classe 2. (b) Après l'apprentissage de la classe 2.*

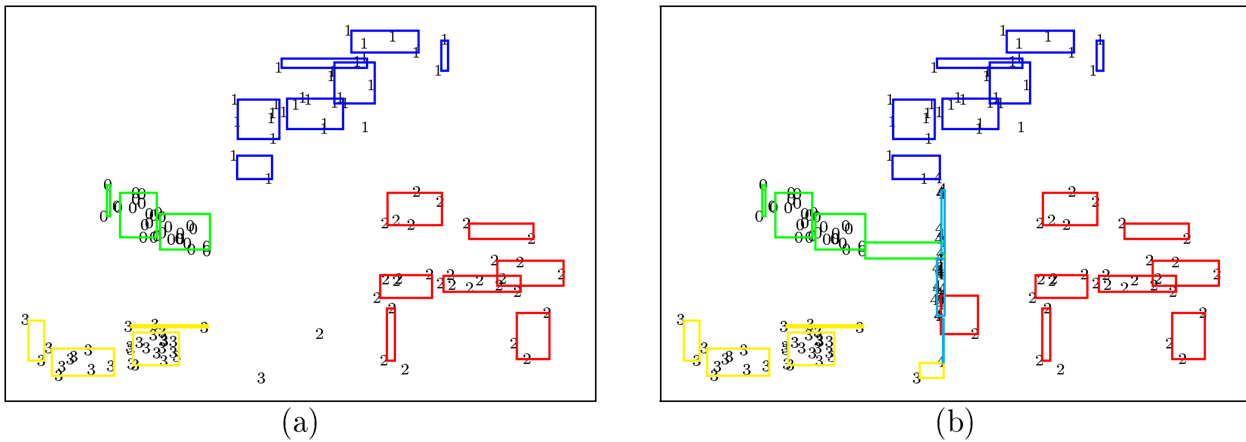


Figure 5.7: *Apprentissage des données #5 en excluant la classe 4 avec* $\overline{\rho_a} = 0.9$ *et permutation aléatoire. (a) Avant l'apprentissage de la classe 4. (b) Après l'apprentissage de la classe 4.*

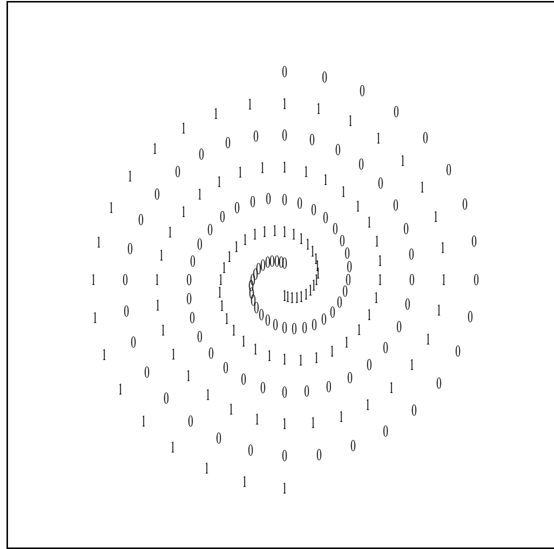


Figure 5.8: *Fichier de données "spirales".*

Nous sommes donc à présent convaincus que le Fuzzy ARTMAP est capable d'apprendre de nouvelles classes sans qu'un réentraînement ne soit nécessaire, car l'impact est minimal sur le reste des données. Cette flexibilité pourra s'avérer très utile en pratique. Par exemple, si le réseau est utilisé pour reconnaître des chiffres manuscrits, il pourra apprendre aisément un nouveau style d'écriture tout en demeurant capable de reconnaître ceux qu'il connaît déjà. Cette propriété rend le Fuzzy ARTMAP bien plus attrayant que d'autres réseaux comme le perceptron dont l'entraînement doit être recommencé à partir de zéro dans une telle situation.

5.2.4 Distinction de deux spirales

Dans l'article de Carpenter et al. [2, section VII], on teste le Fuzzy ARTMAP en tentant de lui faire distinguer deux spirales l'une à l'intérieur de l'autre [10]. Ces spirales sont illustrées à la figure 5.8. Nous allons regarder ce que nous obtenons et le comparer avec les résultats de l'article.

Pour entraîner le réseau, nous utiliserons deux spirales de 97 points chacune. Nous l'aiderons, comme l'ont fait les auteurs, en lui présentant un point de chaque spirale en alternance, en allant de l'extérieur vers le centre. Nous allons tester le réseau entraîné sur deux spirales plus denses, contenant chacune 385 points. Si nous choisissons une vigilance de base $\overline{\rho_a}$ assez élevée, le réseau est bel et bien capable d'apprendre à distinguer ces deux spirales. Le classement des données de

test est parfait avec $\overline{\rho_a} = 1$ (fig. 5.9a). Le hic est que le réseau a appris chacune des données d'entraînement individuellement, ce qui fait un total de 194 rectangles.

En réduisant $\overline{\rho_a}$ à 0.95, le réseau forme 78 rectangles, pour le classement que nous voyons à la figure 5.9b. Le taux de classement des données de test est alors de 96.9%. Ce pourcentage diminue très rapidement, car lorsque $\overline{\rho_a} = 0.9$ (fig. 5.9c), il devient égal à 77.8%. La figure 5.10 montre les rectangles formés par le Fuzzy ARTMAP pour ces deux cas.

À ce niveau, nos résultats ne concordent pas avec ceux des auteurs du Fuzzy ARTMAP. Bien que le nombre de rectangles qu'ils ont formés soit le même pour les trois cas, leur pourcentage de classement des données de test est de 99% avec $\overline{\rho_a} = 0.95$ et de 96.4% avec $\overline{\rho_a} = 0.9$. La raison pour laquelle nos résultats sont de beaucoup inférieurs aux leurs n'est pas encore expliquée.

Il y a aussi des différences quand nous prenons $\overline{\rho_a} = 0$. Le classement que nous obtenons après chacune des deux premières passes est illustré à la figure 5.11. Cette fois, non seulement le pourcentage de classement des données est différent, mais aussi le nombre de rectangles formés. Le tableau suivant résume la situation :

Nombre de passes	*Nombre de rectangles*		*% de classement des données d'entraînement*	
	Notre résultat	Carpenter et al.	Notre résultat	Carpenter et al.
1	10	13	86.1%	90.7%
2	22	16	94.3%	94.3%
5	38	25	92.3%	100%

Malgré tout, nous pouvons constater que le Fuzzy ARTMAP est capable de faire du bon travail dans des cas aussi complexes que la distinction de deux spirales l'une dans l'autre. Il faut cependant une vigilance de base $\overline{\rho_a}$ très élevée et les données ne doivent pas être présentées dans un ordre quelconque.

Nous savons maintenant mieux ce que peut faire le Fuzzy ARTMAP après avoir analysé le classement qu'il effectue avec des points dans un plan. Nous avons étudié l'effet de la vigilance de ART_a, puis nous avons discuté de l'apprentissage des classes en séquence et de l'apprentissage d'une classe séparément. Enfin, nous avons analysé le travail du réseau pour distinguer deux spirales. Nous sommes maintenant prêts à passer à la réalisation concrète du Fuzzy ARTMAP en C++.

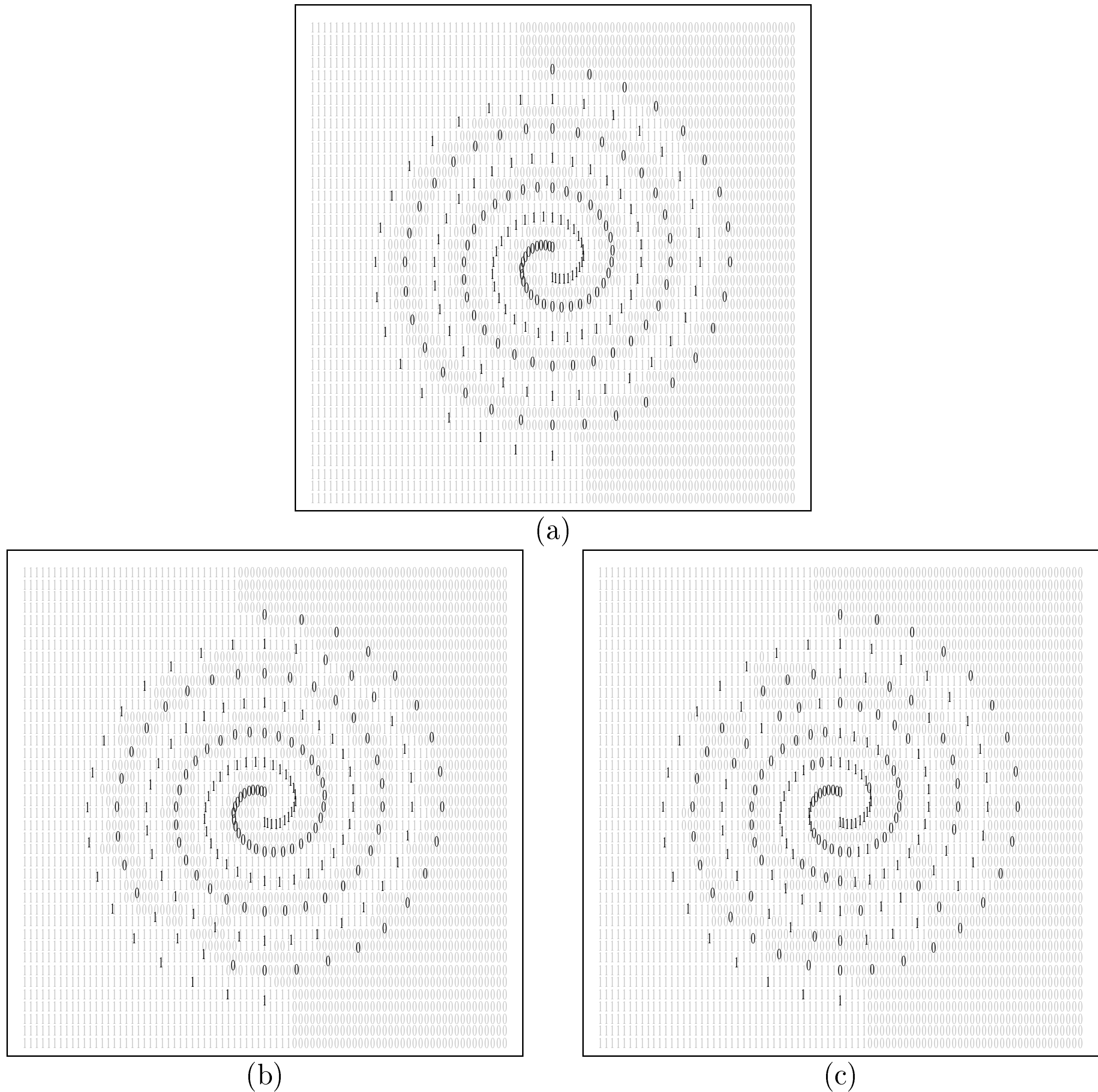
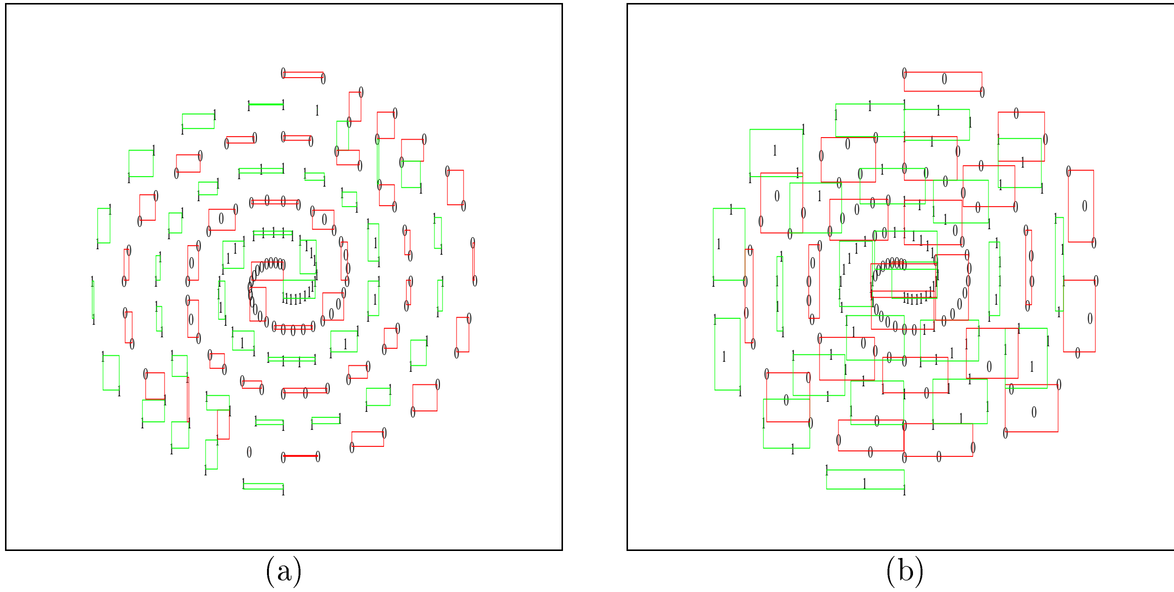
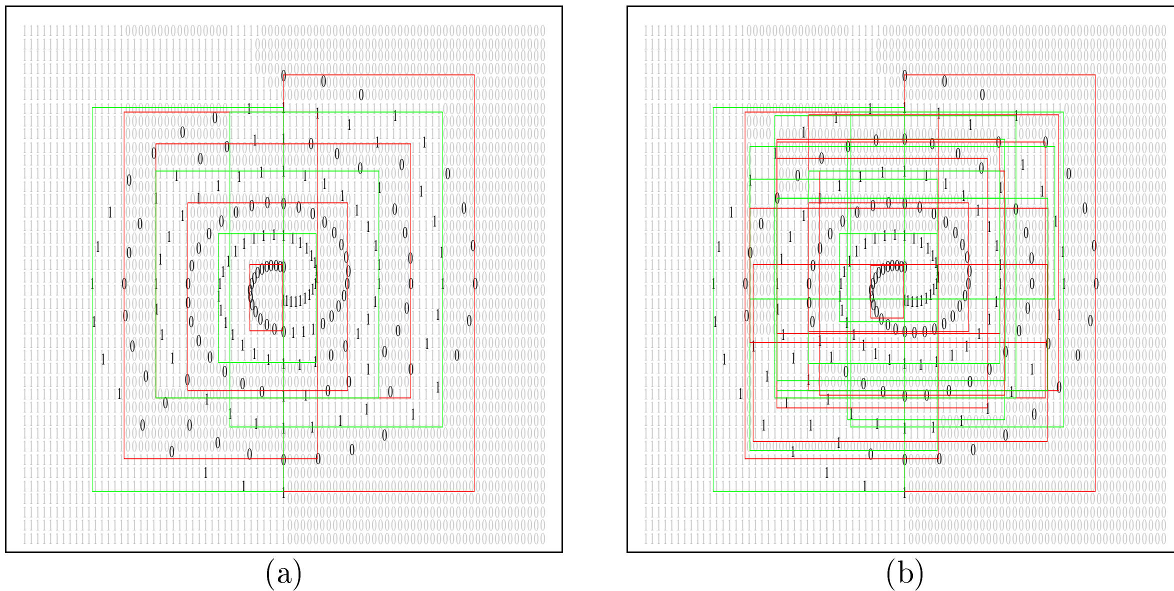


Figure 5.9: *Classement de données après entraînement avec les spirales. (a)* $\overline{\rho_a} = 1$. *(b)* $\overline{\rho_a} = 0.95$. *(c)* $\overline{\rho_a} = 0.9$.



(a) (b)

Figure 5.10: *Rectangles créés après entraînement avec les spirales. (a)* $\overline{\rho_a} = 0.95$. *(b)* $\overline{\rho_a} = 0.9$.



(a) (b)

Figure 5.11: *Classement de données après entraînement avec les spirales avec* $\overline{\rho_a} =$ 0. *(a) Après une passe d'entraînement. (b) Après deux passes d'entraînement.*

5.3 Implantation

Cette section porte sur l'implantation en C++ du réseau Fuzzy ARTMAP. Nous verrons d'abord la classe `FuzzyARTMAP` elle-même, puis les différents programmes qui l'utilisent et qui ont servi à illustrer le fonctionnement du réseau à la section 5.2.

Le code source de cette implantation est disponible sur le réseau du département de génie électrique et de génie informatique de l'Université Laval, dans les mêmes répertoires que le ART1 (voir la section 1.3, p.30).

5.3.1 Classe FuzzyARTMAP

La classe `FuzzyARTMAP` permet de gérer un réseau de neurones de type Fuzzy ARTMAP dont la structure est décrite à la section 5.1.1.

Le Fuzzy ARTMAP est composé de 2 Fuzzy ART (voir la classe `FuzzyART` à la section 2.3.1, p.52), `artA` et `artB`, d'une matrice de poids des connexions entre ces deux réseaux et d'un vecteur de sortie.[3]

Code source : `FuzzyARTMAP.hpp` et `FuzzyARTMAP.cpp`.

Constructeur

Le constructeur de la classe permet de créer la structure du réseau. Il possède quatre paramètres :

```
FuzzyARTMAP(int _d1A=6, int _d2A=6, int _d1B=6, int _d2B=6);
```

Les paramètres *_d1A* et *_d2A* sont les dimensions des couches de ART_a tandis que *_d1B* et *_d2B* sont les dimensions des couches de ART_b.

Les paramètres du réseau sont initialisés aux valeurs suivantes: $\beta_{ab} = 1$ et $\rho_{ab} = 1$. Les paramètres relatifs à chacun des FuzzyART sont initialisés à leur valeur par défaut (voir la section 2.3.1, p.52), sauf ρ_b qui est initialisé à 1. Les poids des connexions entre les deux réseaux sont initialisés à 1.

[3]Les classes `Vecteur` et `Matrice` utilisées sont celles créées par Marc Parizeau et sont disponibles à `www.gel.ulaval.ca/~parizeau/C++/classes/doc/Classes/Classes.html`.

Opérateurs

La classe `FuzzyARTMAP` possède les opérateurs suivants :

- `friend ostream &operator<<(ostream &_os, const FuzzyARTMAP &_art);`

  Insère le réseau *_art* dans le flot de sortie *_os* (voir `Ecrire`).

- `friend istream &operator>>(istream &_is, FuzzyARTMAP &_art);`

  Extrait le réseau *_art* dans le flot d'entrée *_is* (voir `Lire`).

Fonctions d'interface

La classe `FuzzyARTMAP` possède les fonctions d'interface suivantes :

- `Vecteur Activites();`

  Retourne les activités du vecteur de sortie $\mathbf{x}^{ab}$.

- `void Agrandissement(int _m=1);`

  Le réseau a un mode d'agrandissement automatique, actif par défaut, qui permet aux couches F_2 de s'agrandir si tous les neurones de l'une d'entre elles sont utilisés. La fonction `Agrandissement` contrôle ce mode. Le paramètre *_m* doit être 0 ou 1, 0 signifiant que l'agrandissement automatique est désactivé.

- `void Apprentissage(int _mode=1);`

  Permet de désactiver (ou d'activer) l'apprentissage du réseau, c'est-à-dire de l'empêcher (ou de lui permettre) de modifier les poids de ses connexions. Normalement l'apprentissage est actif, mais il peut être utile, dans certains cas, de le désactiver. Le paramètre *_m* doit être 0 ou 1, 0 signifiant que l'apprentissage est désactivé.

- `double Beta();`

  Retourne la valeur du taux d'apprentissage du *map field*, β_{ab}.

- `void Beta(double _b);`

  Fixe à *_b* la valeur du taux d'apprentissage du *map field*, β_{ab}.

- `void Betas(double _b);`

  Fixe à *_b* les valeurs des taux d'apprentissages β_a, β_b et β_{ab}.

- `void Ecrire(ostream &_os=cout);`

  Écrit le réseau, y compris les paramètres, dans le flot de sortie *_os*. Par défaut, *cout* est employé.

- `void Lire(istream &_is=cin);`

  Lit le réseau à partir du flot d'entrée *_is*. Par défaut, *cin* est employé.

- `void LireParam(istream &_is=cin);`

  Lit les paramètres du réseau à partir du flot d'entrée *_is*. Par défaut, *cin* est employé. Le fichier peut contenir n'importe quel paramètre, inscrit sous la forme `nom: valeur`, où `nom` peut être `alphaA`, `betaA`, `rhoA`, `alphaB`, `betaB`, `rhoB`, `betaAB` ou `rhoAB` (par exemple `rhoB: 1`). Si un des paramètres n'est pas spécifié, il est initialisé à sa valeur par défaut.

- `Vecteur PoidsAB(int _j);`

  Retourne le vecteur $\mathbf{w}^{ab}_{_j}$ de poids des connexions du *map field* reliées au neurone *_j* de F_2^a.

- `int PropagerA(const Vecteur &_entree);`

  Propage le vecteur *_entree* dans le réseau ART_a et retourne l'indice du neurone de F_2^B représentant la classe du vecteur. La vigilance ρ_a est diminuée à 0 avant d'effectuer la propagation.

- `void PropagerB(const Vecteur &_entree);`

  Propage le vecteur *_entree* dans le réseau ART_b.

- `void PropagerAB(const Vecteur &_eA, const Vecteur &_eB);`

  Place les vecteurs *_eA* et *_eB* en entrée des réseaux ART_a et ART_b, respectivement, et effectue l'algorithme d'appentissage décrit à la section 5.1.3.

- `double Vigilance();`

  Retourne la valeur de la vigilance du *map field*, ρ_{ab}.

- `void Vigilance(double _r);`

  Fixe à *_r* la valeur de la vigilance du *map field*, ρ_{ab}.

- `void Vigilances(double _r);`

  Fixe à *_r* les valeurs des vigilances ρ_a, ρ_b et ρ_{ab}.

Exemple d'utilisation

Le programme suivant crée un réseau Fuzzy ARTMAP avec 4 entrées et 10 sorties pour le ART_a, et 2 entrées et 2 sorties pour le ART_b. La vigilance de ART_a est fixée à 0.6 (les autres paramètres ont leur valeur par défaut). On crée deux paires de vecteurs et on les propage dans le réseau. Les vecteurs ont 2 et 1 dimensions car ils subissent un codage complémentaire lors de leur présentation au Fuzzy ARTMAP. Ensuite on lui fait classer un nouveau vecteur.

```
#include "FuzzyARTMAP.hpp"

int main() {
  FuzzyARTMAP res(4,10,2,2);
  res.artA.Vigilance(0.6);
  Vecteur va(2), vb(1);
  va(0)=0.3; va(1)=0.8;
  vb(0)=0;
  res.PropagerAB(va,vb);
  va(0)=0.9; va(1)=0.2;
  vb(0)=1;
  res.PropagerAB(va,vb);
  va(0)=0; va(1)=1;
  int classe = res.PropagerA(va);
  cout << "La classe du vecteur est " << classe << endl;
}
```

5.3.2 Programmes utilisant le Fuzzy ARTMAP

Trois programmes utilisant la classe `FuzzyARTMAP` ont été réalisés.

fuzzyartmap_2D	Permet d'utiliser le Fuzzy ARTMAP sur des données dans un plan XY.
fuzzyartmap_seq	Permet d'utiliser le Fuzzy ARTMAP pour apprendre des classes en séquence, une à la fois. Les données sont dans un plan XY.
fuzzyartmap_excl	Permet d'utiliser le Fuzzy ARTMAP pour apprendre en excluant une classe, puis apprendre celle-ci. Les données sont dans un plan XY.

Programme fuzzyartmap_2D

Le programme `fuzzyartmap_2D` permet d'utiliser le Fuzzy ARTMAP sur des données dans un plan XY. Il lit dans un fichier une série de points enregistrés sous forme de table de table de vecteurs (`<Table1D<Table1D<Vecteur>>>`), chaque sous-table contenant les vecteurs qui appartiennent à la même classe. Ces points sont normalisés dans l'intervalle [0,1] et sont fournis un à la suite de l'autre à un réseau Fuzzy ARTMAP.

Lorsque le fichier de données s'affiche dans une fenêtre, cliquer sur celle-ci pour que le Fuzzy ARTMAP démarre son apprentissage. Lorsque la lecture des données est complétée, le classement effectué par le réseau s'affiche dans une autre fenêtre. À ce moment, il y a trois options :

- Cliquer sur le bouton de gauche de la souris fait lire à nouveau les données au réseau.
- Cliquer sur le bouton du centre de la souris fait afficher les frontières apprises par le réseau.
- Cliquer sur le bouton de droite de la souris fait afficher une grille de points montrant comment le réseau classerait ces points dans son état actuel.

Ce programme accepte plusieurs paramètres sur la ligne de commande:

-d *nom*	Nom du fichier contenant les données, `data1.tp2` par défaut.
-P	Permuter les données (à chaque lecture) au lieu de les lire dans l'ordre.
-p *nom*	Nom du fichier dans lequel lire les paramètres, une alternative à fournir ceux-ci sur la ligne de commande.
-n *valeur*	Nombre de sorties de ART$_a$ (dimension de la couche F_2^a), 20 par défaut.
-s	Propagation spéciale pour la spirale (pas de normalisation et alternance entre les classes).

Code source : `fuzzyartmap_2D.cpp`.

Programme fuzzyartmap_seq

Le programme `fuzzyartmap_seq` permet d'utiliser le Fuzzy ARTMAP pour apprendre des classes en séquence, une à la fois. Il lit dans un fichier une série de points enregistrés sous le même format que le programme `fuzzyartmap_2D`. Ce programme aide à visualiser la capacité du Fuzzy ARTMAP à apprendre des classes séparément sans être réentraîné avec les anciennes données (voir la section 5.2.2).

Lorsque le fichier de données s'affiche dans une fenêtre, cliquer sur celle-ci pour que le Fuzzy ARTMAP démarre son apprentissage. Lorsque la lecture des données est complétée, le classement effectué par le réseau s'affiche dans une autre fenêtre. À ce moment, il y a trois options :

- Cliquer sur le bouton de gauche de la souris fait lire la prochaine classe au réseau.
- Cliquer sur le bouton du centre de la souris fait afficher les frontières apprises par le réseau.
- Cliquer sur le bouton de droite de la souris fait afficher une grille de points montrant comment le réseau classerait ces points dans son état actuel.

Ce programme accepte plusieurs paramètres sur la ligne de commande:

-d *nom*	Nom du fichier contenant les données, `data1.tp2` par défaut.
-P	Permuter les données de chaque classe au lieu de les lire dans l'ordre.
-p *nom*	Nom du fichier dans lequel lire les paramètres, une alternative à fournir ceux-ci sur la ligne de commande.
-n *valeur*	Nombre de sorties de ART_a (dimension de la couche F_2^a), 20 par défaut.

Code source : `fuzzyartmap_seq.cpp`.

Programme fuzzyartmap_excl

Le programme `fuzzyartmap_excl` permet d'utiliser le Fuzzy ARTMAP pour apprendre en excluant une classe, puis apprendre celle-ci. Il lit dans un fichier une

série de points enregistrés sous le même format que le programme `fuzzyartmap_2D`. Ce programme aide à visualiser la capacité du Fuzzy ARTMAP à apprendre une nouvelle classe sans être réentraîné avec les anciennes données (voir la section 5.2.3).

Lorsque le fichier de données s'affiche dans une fenêtre, cliquer sur celle-ci pour que le Fuzzy ARTMAP démarre son apprentissage. Lorsque la lecture des données est complétée, le classement effectué par le réseau s'affiche dans une autre fenêtre. Cliquer sur la fenêtre pour faire apprendre la classe exclue au réseau. Lorsque le nouveau classement s'affiche, il y a trois options :

- Cliquer sur le bouton de gauche de la souris fait lire à nouveau les données au réseau.
- Cliquer sur le bouton du centre de la souris fait afficher les frontières apprises par le réseau.
- Cliquer sur le bouton de droite de la souris fait afficher une grille de points montrant comment le réseau classerait ces points dans son état actuel.

Ce programme accepte plusieurs paramètres sur la ligne de commande :

-e *valeur*	Indice de la classe à exclure, 0 par défaut.
-d *nom*	Nom du fichier contenant les données, `data1.tp2` par défaut.
-P	Permuter les données au lieu de les lire dans l'ordre.
-p *nom*	Nom du fichier dans lequel lire les paramètres, une alternative à fournir ceux-ci sur la ligne de commande.
-n *valeur*	Nombre de sorties de ART_a (dimension de la couche F_2^a), 20 par défaut.

Code source : `fuzzyartmap_excl.cpp`.

Conclusion

Nous avons étudié cinq différentes architectures neuronales capables de faire du classement de données, supervisé ou non. Ce sont le ART1, le Fuzzy ART, le Fuzzy Min-Max, le LAPART et le Fuzzy ARTMAP. Nous avons visualisé leur fonctionnement grâce à un cas simple qui est le classement de points dans un plan. Les implantations que nous avons réalisées en C++, en plus d'avoir servi à faire les démonstrations, permettront à ceux qui le désirent d'utiliser ces architectures pour des tâches plus avancées.

Le ART1, introduit en 1987, est un bon réseau pour faire de l'apprentissage non supervisé. Cependant il doit recevoir des données binaires. Le Fuzzy ART (1991) est en mesure d'effectuer un travail semblable, et il a l'avantage d'accepter des données analogiques. Il peut donc faire tout ce dont le ART1 est capable et plus, ce qui rend en quelque sorte le ART1 désuet.

L'autre architecture non supervisée que nous avons vue est le Fuzzy Min-Max. Elle se distingue des autres par le fait qu'elle évite les recouvrements entre les différentes classes qu'elle forme. Comme le Fuzzy ART, elle accepte des données analogiques. Elle pourrait probablement être améliorée grâce à une légère modification de la fonction d'appartenance.

Du côté supervisé, le LAPART est un réseau assez intéressant. Par contre, il est souvent incapable d'apprendre parfaitement les données qui lui sont fournies pour l'entraîner. Il faut lui fournir des données binaires.

Le Fuzzy ARTMAP est probablement plus attrayant comme réseau supervisé. Il accepte des entrées analogiques et normalement il est capable d'apprendre correctement toutes ses données d'entraînement.

Il reste à espérer que le survol de ces architectures neuronales aura été profitable et qu'elles seront mises à profit sur des problèmes complexes, car les possibilités qu'elles offrent sont nombreuses.

Bibliographie

[1] Gail A. Carpenter et Stephen Grossberg. A Massively Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine. *Computer Vision, Graphics and Image Processing*, 37:54–115, 1987.

[2] Gail A. Carpenter, Stephen Grossberg, Natalya Markuzon, John H. Reynolds, et David B. Rosen. Fuzzy ARTMAP: A Neural Network Architecture for Incremental Supervised Learning of Analog Multidimensional Maps. *IEEE Transactions on Neural Networks*, 3:698–713, 1992.

[3] Gail A. Carpenter, Stephen Grossberg, et David B. Rosen. Fuzzy ART: Fast Stable Learning and Categorization of Analog Patterns by an Adaptive Resonance System. *Neural Networks*, 4:759–771, 1991.

[4] Thomas P. Caudell et Michael J. Healy. Studies of Inference Rule Creation using LAPART. In *Proceedings of the 5th IEEE International Conference on Fuzzy Systems*, pages 1–6, New Orleans, LA, 1996.

[5] James A. Freeman. *Simulating Neural Networks with Mathematica.* Addison-Wesley, 1994.

[6] James A. Freeman et David M. Skapura. *Neural Networks: Algorithms, Applications, and Programming Techniques.* Addison-Wesley, 1991.

[7] Michael J. Healy et Thomas P. Caudell. Discrete Stack Interval Representations and Fuzzy ART1. In *Proceedings of the World Congress on Neural Networks*, pages II–82–II–91, Portland, 1993.

[8] Michael J. Healy et Thomas P. Caudell. Acquiring Rule Sets as a Product of Learning in a Logical Neural Architecture. *IEEE Transactions on Neural Networks*, 8:461–474, 1997.

[9] Michael J. Healy, Thomas P. Caudell, et Scott D. G. Smith. A Neural Architecture for Pattern Sequence Verification Through Inferencing. *IEEE Transactions on Neural Networks*, 4:9–20, 1993.

[10] Kevin J. Lang et Michael J. Witbrock. Learning to Tell Two Spirals Apart. In *Proceedings of the 1988 Connectionnist Models Summer School*, pages 52–59, 1989.

[11] Patrick K. Simpson. Fuzzy Min-Max Neural Networks — Part 2: Clustering. *IEEE Transactions on Fuzzy Systems*, 1:32–45, 1993.

Annexe A

Codes sources — ART1

- ART1.hpp
- ART1.cpp
- StackInterval.cpp
- art.cpp
- art_2D.cpp
- art_souris.cpp

Annexe B

Codes sources — Fuzzy ART

- FuzzyART.hpp
- FuzzyART.cpp
- fuzzyart_2D.cpp
- fuzzyart_souris.cpp

Annexe C

Codes sources — Fuzzy Min-Max

- FuzzyMinMax.hpp
- FuzzyMinMax.cpp
- minmax_2D.cpp
- minmax_souris.cpp

Annexe D

Codes sources — LAPART

- LAPART.hpp
- LAPART.cpp
- lapart_2D.cpp

Annexe E

Codes sources — Fuzzy ARTMAP

- FuzzyARTMAP.hpp
- FuzzyARTMAP.cpp
- fuzzyartmap_2D.cpp
- fuzzyartmap_excl.cpp
- fuzzyartmap_seq.cpp