Les cartes cognitives floues (Fuzzy Cognitive Maps) : introduction progressive aux applications avancées en Python

Franck Jeannot

Montréal, Canada, AB844B, 28 Décembre 2025

Résumé

Les cartes cognitives floues (**fuzzy cognitive maps, FCM)** constituent une méthodologie puissante pour la modélisation de systèmes complexes caractérisés par l'incertitude et les relations causales. Ce document présente une progression logique et pédagogique des concepts fondamentaux vers des techniques avancées. Nous débutons par une introduction accessible aux concepts de base, puis nous progressons graduellement vers les mécanismes d'inférence, les algorithmes d'apprentissage automatique, et les implémentations pratiques en Python. L'approche adoptée privilégie la compréhension intuitive avant l'introduction de formalismes mathématiques avancés. Des exemples concrets, des diagrammes pédagogiques et des implémentations complètes en Python illustrent chaque concept.

Keywords: Cartes Cognitives Floues, Fuzzy Cognitive Maps, FCM, Logique Floue, Fuzzy Logic, Modélisation de Systèmes Complexes, Graphes Causaux, Réseaux de Neurones Récurrents, Systèmes Dynamiques, Apprentissage Automatique, Nonlinear Hebbian Learning, NHL, Active Hebbian Learning, Algorithme Génétique, RCGA, Analyse de Scénarios, Simulation Monte Carlo, Analyse d'Incertitude, Analyse What-If, Cybersécurité, Priorisation de Vulnérabilités, Aide à la Décision, Évaluation des Risques, Python, FCMpy

Table des matières

1. Introduction : Pourquoi les Cartes Cognitives Floues ?

Dans notre monde de plus en plus complexe, nous sommes confrontés à des systèmes où de nombreux facteurs interagissent de manière non-linéaire et souvent imprévisible. Comment modéliser l'impact du changement climatique sur l'économie ? Comment comprendre les interactions entre les différents facteurs affectant la santé d'un patient ? Comment anticiper l'évolution d'un écosystème urbain ? **Quels sont les usages possibles en cybersécurité ?**

Les Cartes Cognitives Floues (FCM) [1] offrent une réponse élégante à ces questions (Kosko, 1986) [1]. Elles combinent la capacité de la logique floue [2] à gérer l'incertitude avec la puissance des réseaux de neurones [3] pour apprendre et s'adapter (Papageorgiou, 2014) [2].

1.1. Un exemple simple : la gestion d'une entreprise

Imaginons que vous dirigez une petite entreprise. Vous savez intuitivement que :

- Plus vous investissez dans la recherche, plus vos produits sont innovants
- De meilleurs produits conduisent à plus de ventes
- Plus de ventes génèrent plus de profits
- Plus de profits permettent plus d'investissements

Ces relations forment un réseau de cause à effet. Mais comment quantifier ces relations ? Comment simuler l'impact d'une décision d'investissement ? C'est exactement ce que permettent les FCM.

1.2. Objectifs de ce document

Ce document adopte une approche progressive pour vous guider à travers l'univers des FCM :

1. **Fondamentaux** (Sections 2-3) : Comprendre les concepts de base sans formules complexes
2. **Mécanismes** (Sections 4-5) : Explorer comment les FCM fonctionnent et évoluent
3. **Implémentation** (Sections 6-7) : Apprendre à construire et simuler des FCM en Python

1. FCM : Fuzzy Cognitive Maps, introduites par Bart Kosko en 1986 [1]
2. Logique floue : système de raisonnement permettant de traiter l'incertitude et l'imprécision
3. Réseaux de neurones : modèles computationnels inspirés du cerveau humain

4. **Avancé** (Sections 8-9) : Maîtriser les algorithmes d'apprentissage et l'analyse d'incertitude
5. **Applications** (Section 10) : Découvrir les cas d'usage dans différents domaines

2. Les Concepts Fondamentaux des FCM

2.1. Qu'est-ce qu'une carte cognitive ?

Une carte cognitive (Axelrod, 1976) [3] est simplement une représentation visuelle[4] de la façon dont nous comprenons un système. Elle se compose de :

— **Concepts** (ou nœuds) : les éléments clés du système
— **Relations** (ou arêtes) : les liens de causalité entre ces éléments

2.2. L'apport de la « Flou-itude »

Le terme « floue » (fuzzy) fait référence à la logique floue, qui reconnaît que dans le monde réel, les choses ne sont pas simplement vraies ou fausses, présentes ou absentes. Par exemple, une personne n'est pas simplement « malade » ou « en bonne santé » — elle peut être « légèrement malade », « assez en forme », etc. (Zadeh, 1965) [4].

Dans les FCM, cette « flou-itude » se manifeste de deux manières :

1. Les **valeurs des concepts** peuvent varier continuellement (par exemple, de 0 à 1)
2. Les **forces des relations** peuvent être graduelles (par exemple, une influence « forte », « moyenne » ou « faible »)

2.3. Structure de base d'une FCM

Commençons par visualiser la structure d'une FCM simple :

2.4. Les trois types de relations causales

Dans une FCM, chaque relation entre deux concepts peut être de trois types (Stylios, 2004)[5] :

1. **Causalité positive** (poids $w_{ij} > 0$) : Si le concept C_i augmente, alors C_j augmente également
2. **Causalité négative** (poids $w_{ij} < 0$) : Si le concept C_i augmente, alors C_j diminue
3. **Absence de relation** (poids $w_{ij} = 0$) : Les concepts C_i et C_j sont indépendants

Illustrons cela avec un exemple concret :

4. Carte cognitive : représentation graphique des connaissances d'un individu ou d'un groupe sur un système

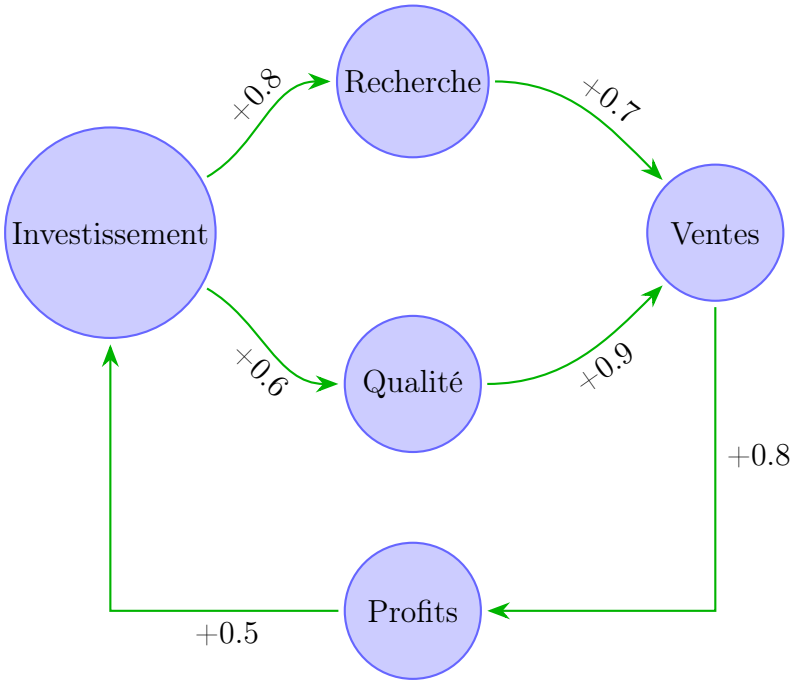


FIGURE 1: Exemple de FCM simple pour un système d'entreprise. Les flèches vertes indiquent des influences positives, avec des poids entre 0 et 1.

3. Comment fonctionne une FCM ? Les bases de la simulation

3.1. L'état d'une FCM

À tout moment, chaque concept de la FCM possède une **valeur d'activation**[5] $A_i(t)$, généralement comprise entre 0 et 1 (ou parfois entre -1 et 1). Cette valeur représente « à quel point » ce concept est actif ou présent dans le système au temps t.

Exemple intuitif :

— $A_{\text{Qualité}}(0) = 0.3$: La qualité du produit est faible au départ
— $A_{\text{Investissement}}(0) = 0.8$: L'entreprise investit fortement
— $A_{\text{Ventes}}(0) = 0.5$: Les ventes sont moyennes

3.2. Évolution dans le temps : Le principe de base

La puissance des FCM réside dans leur capacité à simuler comment le système évolue. Le principe est simple :

5. Valeur d'activation : mesure numérique indiquant l'intensité ou la présence d'un concept

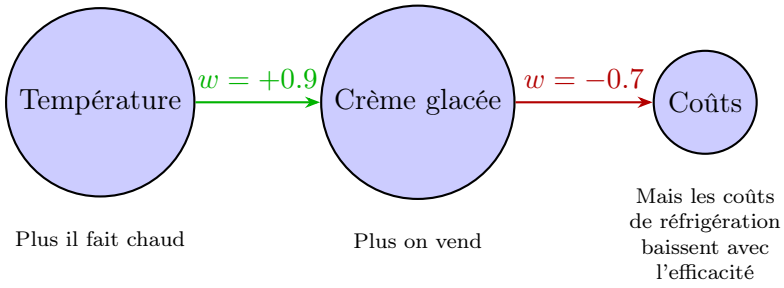


FIGURE 2: Exemple de causalités positive et négative dans un système simple

La nouvelle valeur d'un concept dépend :

1. Des valeurs actuelles des concepts qui l'influencent
2. De la force de ces influences (les poids)

3.3. Visualisation de la propagation

Imaginons une situation simple avec trois concepts :

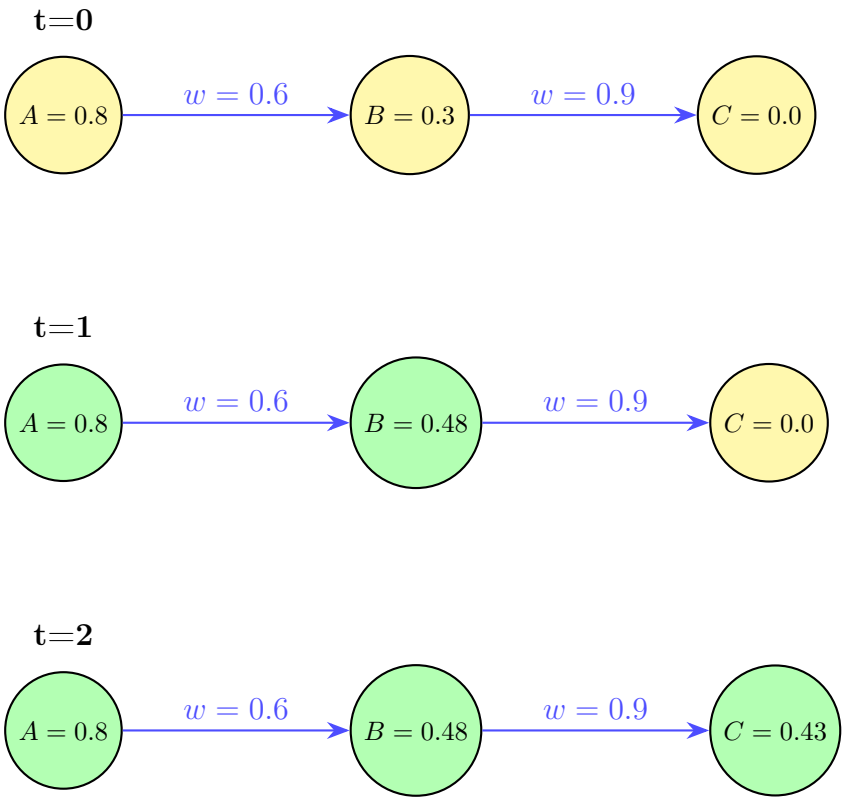


FIGURE 3: Propagation des activations dans le temps. B reçoit l'influence de A, puis C reçoit l'influence de B.

3.4. L'équation de base (version simple)

Sans entrer dans les détails mathématiques complexes, voici l'idée générale :

Nouvelle valeur de C_i = Fonction(Somme des influences reçues)

Plus formellement, mais toujours simplement :

$$A_i(t+1) = f\left(\sum_j w_{ji} \times A_j(t)\right)$$

où :
— $A_i(t+1)$: nouvelle valeur du concept i
— $A_j(t)$: valeur actuelle des concepts j qui influencent i
— w_{ji} : poids de l'influence du concept j vers le concept i
— $f(\cdot)$: une fonction qui «normalise» le résultat

4. Les fonctions de transfert : garder le contrôle

4.1. Pourquoi avons-nous besoin de fonctions de transfert ?

Imaginez que vous additionnez plusieurs influences. Le résultat pourrait devenir très grand (par exemple, 5.3, 12.8, etc.). Mais nous voulons que nos valeurs de concepts restent dans un intervalle raisonnable, typiquement $[0, 1]$ ou $[-1, 1]$.

C'est le rôle des **fonctions de transfert** [6] (aussi appelées fonctions d'activation) : elles « compriment » les résultats pour qu'ils restent dans les limites souhaitées (Papageorgiou, 2003) [6].

4.2. Les principales fonctions de transfert

4.2.1. La fonction bivalente (la plus simple)

C'est la fonction « tout ou rien » :

$$f(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases}$$

Interprétation : Si un concept reçoit des influences positives nettes, il devient complètement actif (1). Sinon, il est inactif (0).

4.2.2. La fonction trivalente

Un peu plus nuancée, elle permet trois états :

$$f(x) = \begin{cases} 1 & \text{si } x > 0.5 \\ 0 & \text{si } -0.5 \leq x \leq 0.5 \\ -1 & \text{si } x < -0.5 \end{cases}$$

4.2.3. La fonction sigmoïde (la plus populaire)

La fonction sigmoïde est une courbe en « S » qui offre une transition douce :

$$f(x) = \frac{1}{1 + e^{-\lambda x}}$$

où λ contrôle la « raideur » de la courbe.

6. Fonction de transfert : fonction mathématique qui limite les valeurs dans un intervalle donné

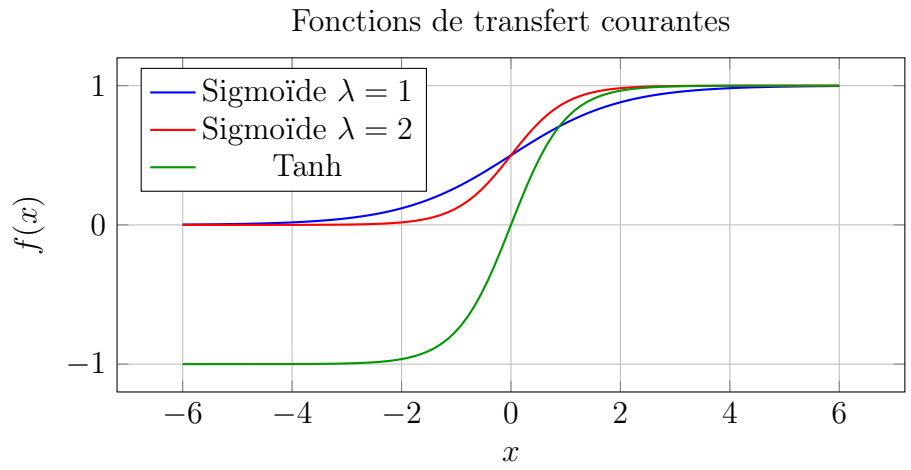


FIGURE 4: Comparaison des fonctions de transfert continues

4.2.4. La fonction tangente hyperbolique (tanh)

Pour des valeurs dans $[-1, 1]$:

$$f(x) = \tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Cette fonction est utile quand vous voulez permettre des influences négatives.

4.3. Choix de la fonction : quel impact ?

Le choix de la fonction de transfert influence significativement le comportement de votre FCM :

TABLE 1: Comparaison des fonctions de transfert

Fonction	**Quand l'utiliser**
Bivalente	Systèmes avec décisions binaires claires (oui/non, actif/inactif)
Trivalente	Systèmes permettant un état neutre ou d'équilibre
Sigmoïde	La plupart des cas réels avec transitions graduelles ; intervalle $[0, 1]$
Tanh	Quand les concepts peuvent avoir des valeurs négatives significatives ; intervalle $[-1, 1]$

5. Règles d'inférence : les différentes façons de calculer

Maintenant que nous comprenons les fonctions de transfert, explorons les différentes **règles d'inférence**[7] — c'est-à-dire les différentes manières de calculer comment les concepts évoluent (Kosko, 1986) [1], (Stylios, 2004) [5].

5.1. Règle de Kosko Originale

C'est la règle la plus simple, proposée par Bart Kosko lui-même :

$$A_i^{(k+1)} = f\left(\sum_{j \neq i} w_{ji} A_j^{(k)}\right)$$

Caractéristique clé : Un concept n'est PAS influencé par sa propre valeur précédente. Seules les influences externes comptent.

Exemple : Si le concept « Ventes » est influencé par « Qualité » et « Marketing », sa nouvelle valeur dépendra uniquement de ces deux facteurs, pas de sa valeur précédente.

5.2. Règle de Kosko Modifiée

Cette variante inclut l'auto-influence :

$$A_i^{(k+1)} = f\left(A_i^{(k)} + \sum_{j \neq i} w_{ji} A_j^{(k)}\right)$$

Pourquoi est-ce important ? Certains concepts ont une « inertie » — par exemple, la réputation d'une entreprise ne change pas instantanément, même avec de nouvelles approches (Stylios, 2004) [5].

5.3. Règle rescalée

Papageorgiou (2011) [7] utilise la *règle rescalée* («rescaled inference rule»). Cette règle aide à éviter la saturation (quand toutes les valeurs tendent vers 0 ou 1) :

$$A_i^{(k+1)} = f\left((2A_i^{(k)} - 1) + \sum_{j \neq i} w_{ji}(2A_j^{(k)} - 1)\right)$$

où :

— $A_i(k)$ est l'activation du concept i à l'instant k ;[8]

7. Règle d'inférence : méthode de calcul des nouvelles valeurs des concepts
8. Définition standard des activations de concepts dans les FCM.

— w_{ji} est le poids de l'influence du concept j vers i ; [9]
— $f(\cdot)$ est une fonction de seuillage (souvent sigmoïde ou bornante) qui renvoie une valeur dans l'intervalle $[0, 1]$. [10]

Cette règle est dite « rescalée » car les activations définies dans $[0, 1]$ sont d'abord transformées en $[-1, 1]$ via le terme $2A - 1$, ce qui modifie la dynamique interne tout en conservant une sortie finale bornée après application de f. [11].

Astuce : Elle transforme temporairement l'intervalle $[0, 1]$ en $[-1, 1]$ pour les calculs, ce qui donne plus de flexibilité .

5.4. Visualisation Comparative

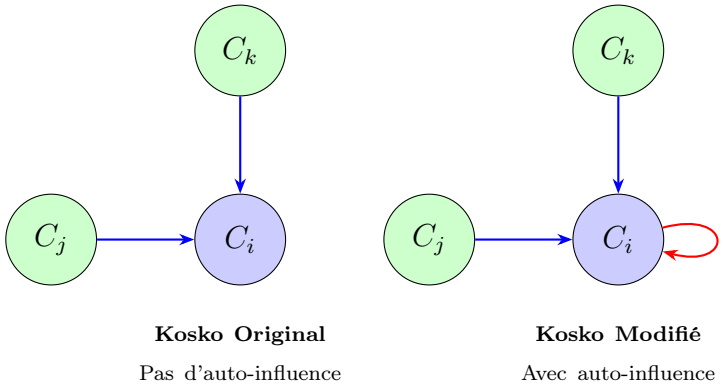


FIGURE 5: Différence entre Kosko Original et Kosko Modifié

6. Convergence et Dynamique des FCM

6.1. Qu'est-ce que la convergence ?

Quand vous simulez une FCM en itérant le processus de calcul, le système peut aboutir à différents états finaux (Kosko, 1988) [8] :

1. **Point fixe (équilibre)** : Les valeurs se stabilisent et ne changent plus
2. **Cycle limite** : Les valeurs oscillent entre quelques états
3. **Chaos** : Comportement imprévisible et erratique

9. Poids causaux entre concepts dans un FCM.
10. Par exemple, une fonction sigmoïde comme décrite dans la littérature FCM.
11. Interprétation courante du terme $(2A - 1)$ comme biais dans la règle rescalée

6.2. Point fixe : l'état désiré

Pour la plupart des applications, on souhaite atteindre un point fixe, car cela représente un état d'équilibre stable du système.

Définition mathématique simple : Un point fixe est atteint quand :

$$|A_i^{(k+1)} - A_i^{(k)}| < \epsilon \quad \text{pour tout } i$$

où ϵ est un seuil très petit (par exemple, 0.001).

6.3. Visualisation de la Convergence

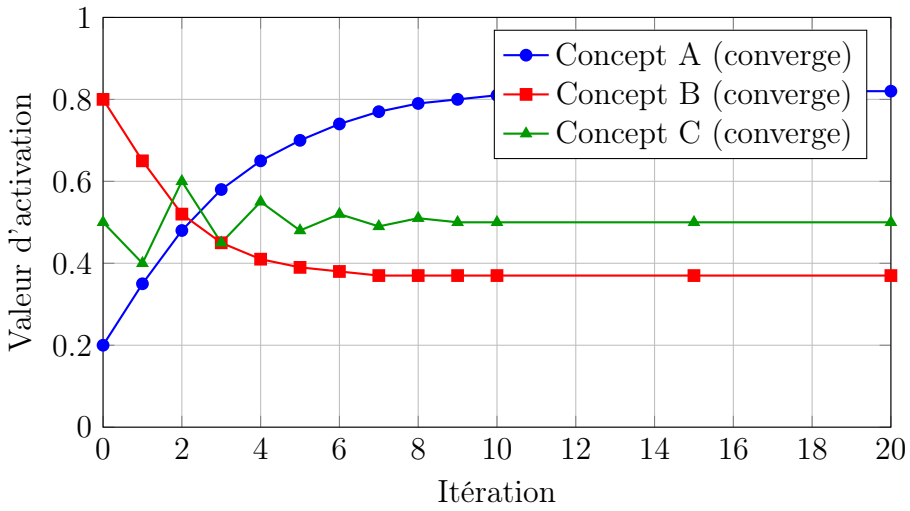


FIGURE 6: Exemple de convergence vers un point fixe

6.4. Facteurs influençant la convergence

Plusieurs paramètres affectent si (et comment) une FCM convergera :

— **Les poids** : Des poids très élevés peuvent causer une instabilité
— **La fonction de transfert** : La sigmoïde favorise généralement la convergence
— **Le paramètre λ** : Un λ trop élevé dans la sigmoïde peut causer des oscillations
— **La structure du réseau** : Les boucles de rétroaction peuvent créer des cycles

7. Construction d'une FCM : De l'expertise à l'implémentation

7.1. Méthodologie de construction

La construction d'une FCM suit généralement ces étapes (Gray, 2013) [9] :

1. **Identification des concepts** : Quels sont les facteurs clés du système ?
2. **Détermination des relations** : Comment ces facteurs s'influencent-ils mutuellement ?
3. **Quantification des poids** : Quelle est la force de chaque influence ?
4. **Validation** : Le modèle reflète-t-il bien la réalité ?

7.2. De l'expertise qualitative aux valeurs numériques

Souvent, les experts s'expriment en termes qualitatifs (« forte influence », « faible impact »). Il faut convertir ces évaluations linguistiques en valeurs numériques.

Échelle typique :

TABLE 2: Conversion des termes linguistiques en poids

Terme linguistique	Valeur numérique	Type
Très forte influence positive	+0.9 à +1.0	Positif
Forte influence positive	+0.7 à +0.9	Positif
Influence positive moyenne	+0.4 à +0.7	Positif
Faible influence positive	+0.1 à +0.4	Positif
Pas d'influence	0	Neutre
Faible influence négative	-0.1 à -0.4	Négatif
Influence négative moyenne	-0.4 à -0.7	Négatif
Forte influence négative	-0.7 à -0.9	Négatif
Très forte influence négative	-0.9 à -1.0	Négatif

7.3. Exemple complet : système de santé

Construisons une FCM pour modéliser les facteurs affectant la santé d'un patient :

Étape 1 : Identifier les concepts

— C1 : Exercice physique
— C2 : Alimentation saine
— C3 : Stress
— C4 : Qualité du sommeil

— C5 : État de santé général

Étape 2 : Déterminer les relations

— L'exercice améliore la santé : C1 → C5 (positif)
— L'alimentation saine améliore la santé : C2 → C5 (positif)
— Le stress dégrade la santé : C3 → C5 (négatif)
— Le stress perturbe le sommeil : C3 → C4 (négatif)
— Un bon sommeil améliore la santé : C4 → C5 (positif)
— L'exercice réduit le stress : C1 → C3 (négatif)

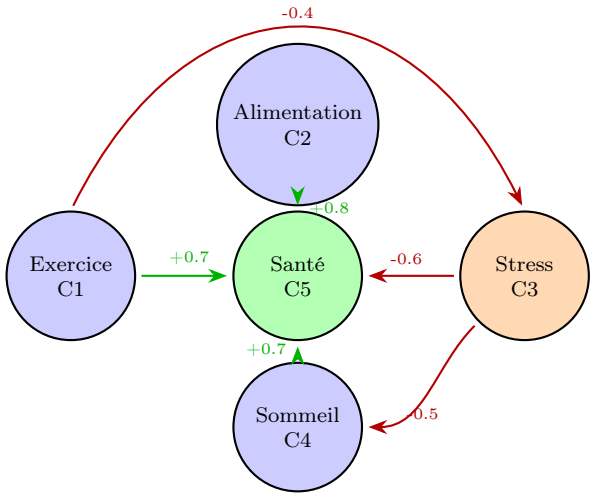


FIGURE 7: FCM pour un système de santé personnel

8. Implémentation en Python : premiers pas

Passons maintenant à l'implémentation pratique. Nous allons construire progressivement notre bibliothèque FCM en Python.

Voir script complet https://github.com/blue101010/fuzycognitivemap/blob/main/montecarlo.py

8.1. Structure de base : la classe FCM

Commençons par créer une classe simple pour représenter une FCM :

```python
import numpy as np

class SimpleFCM:
    """
    Classe de base pour une Carte Cognitive Floue (FCM)
    """
    def __init__(self, n_concepts):
        """
        Initialise une FCM avec n_concepts concepts

        Parametres:
        -----------
        n_concepts : int
            Nombre de concepts dans la FCM
        """
        self.n_concepts = n_concepts
        # Matrice des poids (n x n)
        self.weights = np.zeros((n_concepts, n_concepts))
        # Vecteur d'etat initial (valeurs des concepts)
        self.state = np.zeros(n_concepts)
        # Historique des etats
        self.history = []

    def set_weight(self, from_concept, to_concept, weight):
        """
        Definit le poids de la relation entre deux concepts

        Parametres:
        -----------
        from_concept : int
            Indice du concept source
        to_concept : int
            Indice du concept cible
        weight : float
            Poids de la relation (entre -1 et 1)
        """
        if not -1 <= weight <= 1:
            raise ValueError("Le poids doit etre entre -1 et
    1")
```

```
        self.weights[to_concept, from_concept] = weight

    def set_initial_state(self, state):
        """
        Definit l'etat initial des concepts

        Parametres:
        -----------
        state : array-like
            Vecteur des valeurs initiales
        """
        self.state = np.array(state)
        self.history = [self.state.copy()]

    def sigmoid(self, x, lambda_param=1):
        """
        Fonction de transfert sigmoide

        Parametres:
        -----------
        x : array-like
            Valeurs d'entree
        lambda_param : float
            Parametre de raideur de la sigmoide
        """
        return 1 / (1 + np.exp(-lambda_param * x))

    def tanh_transfer(self, x):
        """
        Fonction de transfert tangente hyperbolique
        """
        return np.tanh(x)

    def bivalent(self, x):
        """
        Fonction de transfert bivalente
        """
        return (x > 0).astype(float)
```

Listing 1: Classe de base pour une FCM simple

8.2. Implémentation des Règles d'Inférence

Ajoutons maintenant les différentes règles d'inférence :

```
    def kosko_inference(self, transfer_func='sigmoid',
                        lambda_param=1):
        """
        Regle d'inference de Kosko originale
        (sans auto-influence)
```

```
6
7         Parametres:
8         -----------
9         transfer_func : str
10            Type de fonction de transfert
11        lambda_param : float
12            Parametre pour sigmoide
13        """
14        # Calcul de l'influence totale
15        total_influence = np.dot(self.weights, self.state)
16
17        # Application de la fonction de transfert
18        if transfer_func == 'sigmoid':
19            new_state = self.sigmoid(total_influence,
    lambda_param)
20        elif transfer_func == 'tanh':
21            new_state = self.tanh_transfer(total_influence)
22        elif transfer_func == 'bivalent':
23            new_state = self.bivalent(total_influence)
24        else:
25            raise ValueError(f"Fonction inconnue: {
    transfer_func}")
26
27        return new_state
28
29    def modified_kosko_inference(self, transfer_func='sigmoid
    ',
30                                 lambda_param=1):
31        """
32        Regle d'inference de Kosko modifiee
33        (avec auto-influence)
34        """
35        # Ajout de l'etat actuel
36        total_influence = self.state + np.dot(self.weights,
    self.state)
37
38        if transfer_func == 'sigmoid':
39            new_state = self.sigmoid(total_influence,
    lambda_param)
40        elif transfer_func == 'tanh':
41            new_state = self.tanh_transfer(total_influence)
42        elif transfer_func == 'bivalent':
43            new_state = self.bivalent(total_influence)
44        else:
45            raise ValueError(f"Fonction inconnue: {
    transfer_func}")
46
47        return new_state
48
```

```
    def rescaled_inference(self, transfer_func='sigmoid',
                           lambda_param=1):
        """
        Regle d'inference rescalee
        """
        # Transformation en [-1, 1]
        scaled_state = 2 * self.state - 1
        scaled_influence = np.dot(self.weights, scaled_state)
        total = scaled_state + scaled_influence

        if transfer_func == 'sigmoid':
            new_state = self.sigmoid(total, lambda_param)
        elif transfer_func == 'tanh':
            new_state = self.tanh_transfer(total)
        else:
            raise ValueError(f"Fonction inconnue: {
transfer_func}")

        return new_state
```

Listing 2: Méthodes d'inférence pour la classe FCM

8.3. Simulation et convergence

Implémentons maintenant la boucle de simulation :

```
    def simulate(self, iterations=50, inference='kosko',
                 transfer_func='sigmoid', lambda_param=1,
                 threshold=0.001):
        """
        Simule l'evolution de la FCM

        Parametres:
        -----------
        iterations : int
            Nombre maximum d'iterations
        inference : str
            Type d'inference ('kosko', 'modified', 'rescaled
')
        transfer_func : str
            Type de fonction de transfert
        lambda_param : float
            Parametre pour sigmoide
        threshold : float
            Seuil de convergence

        Retourne:
        ---------
        dict : Resultats de la simulation
        """
```

```
        # Choix de la methode d'inference
        if inference == 'kosko':
            inference_func = self.kosko_inference
        elif inference == 'modified':
            inference_func = self.modified_kosko_inference
        elif inference == 'rescaled':
            inference_func = self.rescaled_inference
        else:
            raise ValueError(f"Inference inconnue: {inference}")

        # Boucle de simulation
        converged = False
        for iteration in range(iterations):
            # Calculer le nouvel etat
            new_state = inference_func(transfer_func, lambda_param)

            # Verifier la convergence
            diff = np.abs(new_state - self.state)
            if np.all(diff < threshold):
                converged = True
                print(f"Convergence atteinte a l'iteration {iteration}")
                self.state = new_state
                self.history.append(self.state.copy())
                break

            # Mettre a jour l'etat
            self.state = new_state
            self.history.append(self.state.copy())

        if not converged:
            print(f"Pas de convergence apres {iterations} iterations")

        return {
            'converged': converged,
            'iterations': len(self.history) - 1,
            'final_state': self.state,
            'history': np.array(self.history)
        }

    def print_results(self):
        """
        Affiche les resultats de la simulation
        """
        print("\nEtat final des concepts:")
        print("-" * 40)
```

```
        for i, value in enumerate(self.state):
            print(f"Concept C{i+1}: {value:.4f}")
```

Listing 3: Méthode de simulation avec détection de convergence

8.4. Exemple d'utilisation : le système de santé

Utilisons notre classe pour simuler l'exemple du système de santé :

```
# Creation de la FCM avec 5 concepts
health_fcm = SimpleFCM(n_concepts=5)

# Definition des poids (matrice d'adjacence)
# C1 (Exercice) -> C5 (Sante) : +0.7
health_fcm.set_weight(from_concept=0, to_concept=4, weight
    =0.7)

# C2 (Alimentation) -> C5 (Sante) : +0.8
health_fcm.set_weight(from_concept=1, to_concept=4, weight
    =0.8)

# C3 (Stress) -> C5 (Sante) : -0.6
health_fcm.set_weight(from_concept=2, to_concept=4, weight
    =-0.6)

# C4 (Sommeil) -> C5 (Sante) : +0.7
health_fcm.set_weight(from_concept=3, to_concept=4, weight
    =0.7)

# C3 (Stress) -> C4 (Sommeil) : -0.5
health_fcm.set_weight(from_concept=2, to_concept=3, weight
    =-0.5)

# C1 (Exercice) -> C3 (Stress) : -0.4
health_fcm.set_weight(from_concept=0, to_concept=2, weight
    =-0.4)

# Definition de l'etat initial
# [Exercice, Alimentation, Stress, Sommeil, Sante]
initial_state = [0.8, 0.6, 0.7, 0.5, 0.4]
health_fcm.set_initial_state(initial_state)

# Simulation
print("=== Simulation du Systeme de Sante ===\n")
print("Etat initial:")
print("Exercice    : 0.8 (eleve)")
print("Alimentation: 0.6 (moyen)")
print("Stress      : 0.7 (eleve)")
print("Sommeil     : 0.5 (moyen)")
print("Sante       : 0.4 (faible)")
```

```
36
37 results = health_fcm.simulate(
38     iterations=30,
39     inference='kosko',
40     transfer_func='sigmoid',
41     lambda_param=1,
42     threshold=0.001
43 )
44
45 health_fcm.print_results()
46
47 # Analyse des resultats
48 print("\n=== Analyse ===")
49 print(f"Convergence: {'Oui' if results['converged'] else 'Non
       '}")
50 print(f"Iterations: {results['iterations']}")
```

Listing 4: Exemple complet : simulation du système de santé

9. Exemples complémentaires

Quelques exemples d'application.

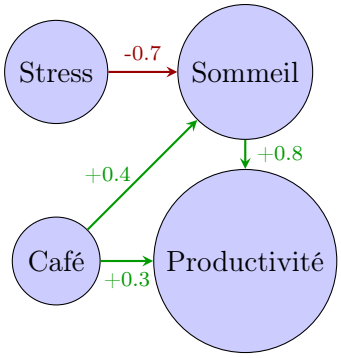


FIGURE 8: Exemple simple de FCM (stress, sommeil, café, productivité).

Le problème vient de la flèche C3 → C1 qui traverse C2. Il faut la courber pour passer au-dessus ou en-dessous.

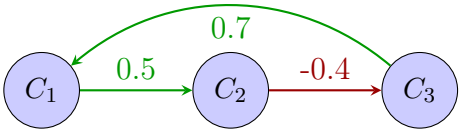


FIGURE 9: Représentation graphique d'une matrice d'adjacence (liens pondérés).

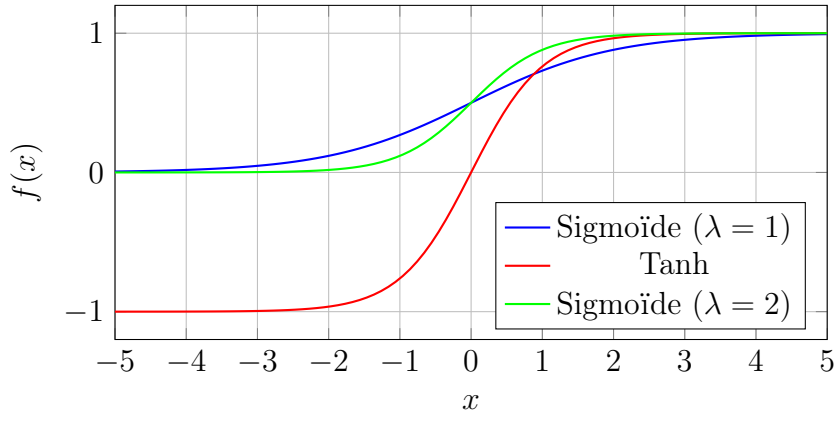


FIGURE 10: Comparaison de fonctions de transfert (sigmoïde vs tanh).

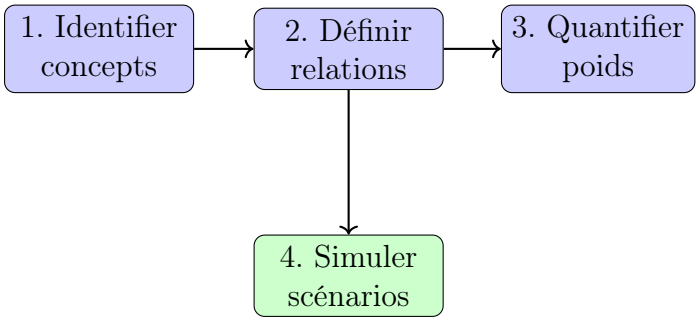


FIGURE 11: Processus méthodologique de construction d'une FCM (atelier experts).

10. Extraits de code complémentaires

Les extraits suivants complètent les implémentations du corps principal. Ils peuvent être copiés tels quels dans un notebook Python.

10.1. Création d'une FCM simple en Python sur le sommeil

Voir script complet : https://github.com/blue101010/fuzycognitivemap/blob/main/sommeil.py

```python
import numpy as np

# Concepts
concepts = ["Stress", "Sommeil", "Cafe", "Productivite"]

# Matrice de poids W
W = np.array([
    [0.0, -0.7, 0.0, 0.0],
    [0.0, 0.0, 0.4, 0.8],
    [0.0, 0.0, 0.0, 0.3],
    [0.0, 0.0, 0.0, 0.0]
])
```

Listing 5: Création d'une FCM simple en Python

```python
import numpy as np

def sigmoid(x, lambd=1):
    return 1 / (1 + np.exp(-lambd * x))

def simulate_fcm(W, A0, steps=10, lambd=1):
    A = A0.copy()
    history = [A.copy()]
    for _ in range(steps):
        A = sigmoid(A + W.T @ A, lambd)
        history.append(A.copy())
    return np.array(history)
```

```

A0 = np.array([0.5, 0.7, 0.3, 0.6])
history = simulate_fcm(W, A0)
print(history[-1])
```

Listing 6: Simulation d'une FCM en Python

```
def kosko_rule(A, W):
    return A + W.T @ A

def modified_kosko(A, W):
    return np.clip(A + W.T @ A, 0, 1)

def rescaled_rule(A, W):
    return (A + W.T @ A) / (1 + np.abs(W.T @ A))
```

Listing 7: Implementation des trois regles d'inference

```
from sklearn.metrics import mean_squared_error

def find_best_lambda(W, A0, target, lambdas):
    best_lambda, best_score = None, float("inf")
    for l in lambdas:
        hist = simulate_fcm(W, A0, steps=10, lambd=l)
        score = mean_squared_error(target, hist[-1])
        if score < best_score:
            best_score = score
            best_lambda = l
    return best_lambda, best_score
```

Listing 8: Calcul du lambda optimal

```
from fcmpy import Fcm

fcm = Fcm(concepts, W)
fcm.simulate(initial_state=A0, transfer="sigmoid", inference=
   "kosko")
print(fcm.state)
```

Listing 9: Construction d'une FCM avec FCMpy

```
A0_low_stress = np.array([0.2, 0.7, 0.3, 0.6])
A0_high_stress = np.array([0.9, 0.7, 0.3, 0.6])

hist_low = simulate_fcm(W, A0_low_stress)
hist_high = simulate_fcm(W, A0_high_stress)

print("Scenario faible stress:", hist_low[-1])
print("Scenario fort stress:", hist_high[-1])
```

Listing 10: Analyse de scenarios

```python
def monte_carlo_fcm(W, A0, runs=100):
    results = []
    for _ in range(runs):
        noise = np.random.normal(0, 0.05, W.shape)
        hist = simulate_fcm(W + noise, A0)
        results.append(hist[-1])
    return np.array(results)

mc_results = monte_carlo_fcm(W, A0)
print(mc_results.mean(axis=0), mc_results.std(axis=0))
```

Listing 11: Simulation Monte Carlo pour FCM

```python
def hebbian_learning(W, A, eta=0.01):
    for i in range(W.shape[0]):
        for j in range(W.shape[1]):
            if i != j:
                W[i,j] += eta * A[i] * A[j]
    return np.clip(W, -1, 1)
```

Listing 12: Apprentissage Hebbien Non-Lineaire

11. Analyse de Scénarios : questions « What-If »

Une des applications les plus puissantes des FCM est l'analyse de scénarios[12] : « Que se passe-t-il si... ? » [9]

11.1. Types d'interventions

Il existe deux types principaux d'interventions :

1. **Intervention ponctuelle** : Changer la valeur initiale d'un concept
2. **Intervention continue** : Maintenir un concept à une valeur fixée pendant toute la simulation

11.2. Implémentation des interventions

```python
class FCMWithInterventions(SimpleFCM):
    """
    Extension de SimpleFCM avec support des interventions
    """
    def __init__(self, n_concepts):
        super().__init__(n_concepts)
        self.fixed_concepts = {}  # Concepts fixes

    def set_fixed_concept(self, concept_id, value):
```

12. Analyse de scénarios : étude des effets de changements hypothétiques sur le système

```
        """
        Fixe un concept a une valeur constante
        (intervention continue)

        Parametres:
        -----------
        concept_id : int
            Indice du concept a fixer
        value : float
            Valeur fixe du concept
        """
        self.fixed_concepts[concept_id] = value

    def clear_fixed_concepts(self):
        """
        Supprime toutes les interventions continues
        """
        self.fixed_concepts = {}

    def simulate(self, iterations=50, inference='kosko',
                 transfer_func='sigmoid', lambda_param=1,
                 threshold=0.001):
        """
        Simule avec prise en compte des concepts fixes
        """
        if inference == 'kosko':
            inference_func = self.kosko_inference
        elif inference == 'modified':
            inference_func = self.modified_kosko_inference
        elif inference == 'rescaled':
            inference_func = self.rescaled_inference

        converged = False
        for iteration in range(iterations):
            # Calculer le nouvel etat
            new_state = inference_func(transfer_func,
    lambda_param)

            # Appliquer les concepts fixes
            for concept_id, value in self.fixed_concepts.
    items():
                new_state[concept_id] = value

            # Verifier la convergence
            diff = np.abs(new_state - self.state)
            if np.all(diff < threshold):
                converged = True
                print(f"Convergence atteinte a l'iteration {
    iteration}")
```

```
                self.state = new_state
                self.history.append(self.state.copy())
                break

            self.state = new_state
            self.history.append(self.state.copy())

        if not converged:
            print(f"Pas de convergence apres {iterations} iterations")

        return {
            'converged': converged,
            'iterations': len(self.history) - 1,
            'final_state': self.state,
            'history': np.array(self.history)
        }
```

Listing 13: Extension de la classe FCM pour les interventions

11.3. Exemple : Impact d'une intervention sur le stress

```
# Scenario 1 : Situation de base
print("=== Scenario 1 : Situation de Base ===")
health_fcm1 = FCMWithInterventions(n_concepts=5)

# Configuration identique a l'exemple precedent
# ... (memes poids)

initial_state = [0.8, 0.6, 0.7, 0.5, 0.4]
health_fcm1.set_initial_state(initial_state)

results1 = health_fcm1.simulate(iterations=30)
print(f"Sante finale: {results1['final_state'][4]:.4f}")

# Scenario 2 : Intervention anti-stress
print("\n=== Scenario 2 : Programme Anti-Stress ===")
health_fcm2 = FCMWithInterventions(n_concepts=5)

# ... (memes poids)

# Intervention : reduire le stress a 0.3
initial_state2 = [0.8, 0.6, 0.3, 0.5, 0.4]
health_fcm2.set_initial_state(initial_state2)

results2 = health_fcm2.simulate(iterations=30)
print(f"Sante finale: {results2['final_state'][4]:.4f}")

# Comparaison
```

```python
print("\n=== Comparaison ===")
improvement = results2['final_state'][4] - results1['
    final_state'][4]
print(f"Amelioration de la sante: {improvement:.4f}")
print(f"Pourcentage d'amelioration: {improvement/results1['
    final_state'][4]*100:.1f}%")
```

Listing 14: Analyse what-if : réduction du stress

12. Étude de cas approfondie : transition énergétique

Cette étude de cas illustre comment une FCM peut soutenir une discussion structurée entre parties prenantes (régulation, investissements, acceptabilité, coût, émissions). Le but n'est pas de prédire précisément, mais d'explorer des *interactions causales* et de comparer des scénarios cohérents [10, 11].

12.1. Structure conceptuelle

On définit un ensemble de concepts (investissement, régulation, taxe carbone, innovation, coût des renouvelables, acceptabilité, part de renouvelables, emplois verts, émissions) et une matrice de poids W.

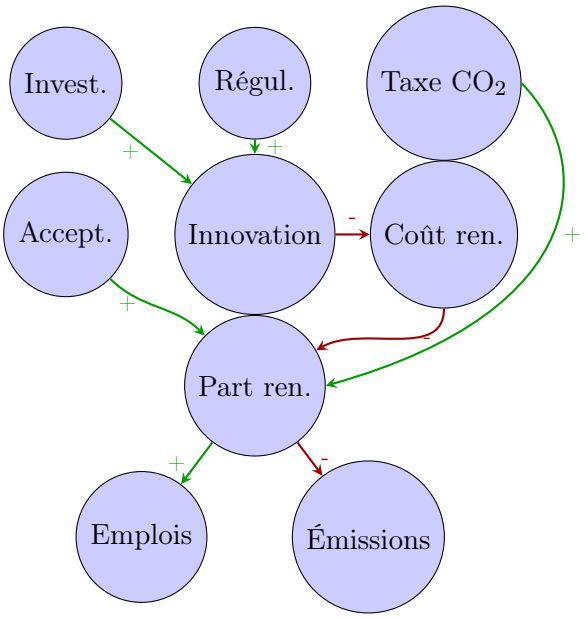


FIGURE 12: Schéma conceptuel simplifié pour l'étude de cas énergie.

12.2. Implémentation Python (extrait complet)

Le code ci-dessous reprend une implémentation et montre un squelette reproductible pour comparer un scénario *baseline* et un scénario *ambitieux*. On peut ensuite enrichir cette base en ajoutant une calibration ou une propagation d'incertitude.

```
# Concepts du modele energetique
concepts_energy = [
    'Investissement', 'Regulation', 'Taxe_carbone',
    'Innovation', 'Cout_renouv', 'Acceptabilite',
    'Part_renouv', 'Emplois_verts', 'Emissions'
]

# Matrice de poids
W_energy = np.zeros((9, 9))
W_energy[3, 0] = 0.8   # Investissement -> Innovation
W_energy[3, 1] = 0.6   # Regulation -> Innovation
W_energy[4, 3] = -0.5  # Innovation -> Cout (negatif)
...
W_energy[6, 4] = -0.6  # Cout -> Part_renouv (negatif)
W_energy[6, 2] = 0.4   # Taxe -> Part_renouv
W_energy[6, 5] = 0.6   # Acceptabilite -> Part_renouv
W_energy[7, 6] = 0.7   # Part_renouv -> Emplois
W_energy[8, 6] = 0.8   # Part_renouv -> Reduction emissions

# Scenarios
A0_base = np.array([0.5,0.5,0.3,0.3,0.6,0.5,0.3,0.4,0.3])
A0_ambitieux = np.array
    ([0.8,0.8,0.7,0.3,0.6,0.5,0.3,0.4,0.3])

# Simulation et comparaison
hist_base, _ = simulate_fcm(W_energy, A0_base)
hist_amb, _ = simulate_fcm(W_energy, A0_ambitieux)

print("Impact sur Part renouvelables:")
```

Listing 15: Etude de cas complete

13. Apprentissage automatique pour les FCM

Jusqu'à présent, nous avons construit nos FCM manuellement, en nous basant sur l'expertise humaine. Mais que faire quand nous avons des données historiques ? Les algorithmes d'apprentissage automatique peuvent nous aider à découvrir les poids (Papageorgiou, 2004) [12].

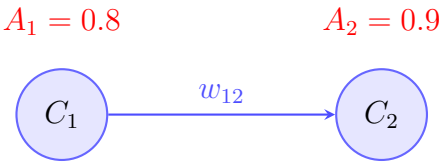
13.1. Apprentissage Hebbien : le principe de base

L'apprentissage Hebbien [13] est une approche non supervisée inspirée de la neurobiologie (Dickerson, 1994) [13].

Principe : Si deux concepts sont fréquemment actifs en même temps, leur connexion devrait se renforcer.

13.2. Principe de Hebb vulgarisé

Le principe de Hebb (1949) stipule que *"les neurones qui s'activent ensemble se connectent ensemble"*. Traduit pour les FCM : si deux concepts sont simultanément actifs, leur lien causal se renforce.



Règle de Hebb : Si C_1 et C_2 sont actifs, alors w_{12} augmente

13.3. Définition formelle

Une Carte Cognitive Floue est un graphe orienté pondéré $G = (C, W)$ où :

— $C = \{C_1, C_2, \ldots, C_n\}$ représente l'ensemble des **concepts** (nœuds)

— $W = [w_{ij}]_{n \times n}$ est la **matrice des poids** causaux

Chaque poids $w_{ij} \in [-1, 1]$ quantifie l'influence du concept C_i sur C_j :

$$w_{ij} = \begin{cases} > 0 & \text{influence positive (excitation)} \\ < 0 & \text{influence négative (inhibition)} \\ = 0 & \text{pas de relation causale} \end{cases} \tag{1}$$

13. Apprentissage Hebbien : inspiré de la neurobiologie, principe selon lequel « les neurones qui s'activent ensemble se connectent ensemble »

13.4. Dynamique d'inférence

L'état du système évolue selon la règle de mise à jour itérative :

$$A_i^{(t+1)} = f\left(\sum_{j=1}^{n} w_{ji} \cdot A_j^{(t)}\right) \tag{2}$$

où $A_i^{(t)} \in [0,1]$ est le niveau d'activation du concept C_i à l'itération t, et f est une fonction de transfert. La fonction sigmoïde est couramment utilisée :

$$f(x) = \frac{1}{1 + e^{-\lambda x}} \tag{3}$$

Le paramètre $\lambda > 0$ contrôle la pente de la sigmoïde.

13.5. Formulation mathématique du NHL

L'algorithme NHL, proposé par Papageorgiou et al. [14], étend la règle de Hebb pour les unités non-linéaires. La mise à jour des poids s'effectue selon :

$$w_{ij}^{(k+1)} = \gamma \cdot w_{ij}^{(k)} + \eta \cdot A_i^{(k)} \cdot \left(A_j^{(k)} - A_j^{(k-1)}\right) \tag{4}$$

où :

— $\gamma \in (0,1)$: (gamma) coefficient de **décroissance** (évite l'explosion des poids)

— $\eta \in (0,1)$: **(eta) taux d'apprentissage**

— $A_j^{(k)} - A_j^{(k-1)}$: **variation d'activation** (composante différentielle)

13.6. NHL expliqué

L'algorithme **Nonlinear Hebbian Learning** (NHL) permet d'ajuster automatiquement les poids d'une FCM à partir de données observées. La règle de mise à jour est :

$$\boxed{w_{ij}^{(k+1)} = \underbrace{\gamma \cdot w_{ij}^{(k)}}_{\text{Mémoire}} + \underbrace{\eta \cdot A_i^{(k)} \cdot \left(A_j^{(k)} - A_j^{(k-1)}\right)}_{\text{Apprentissage Hebbien}}} \tag{5}$$

Interprétation des termes :.

— **Terme de mémoire** ($\gamma \cdot w_{ij}^{(k)}$) : conserve une fraction du poids précédent, évitant les oscillations et l'oubli brutal.

— **Terme Hebbien** ($\eta \cdot A_i \cdot \Delta A_j$) : renforce le lien si le concept source C_i est actif *et* que le concept cible C_j change d'état.

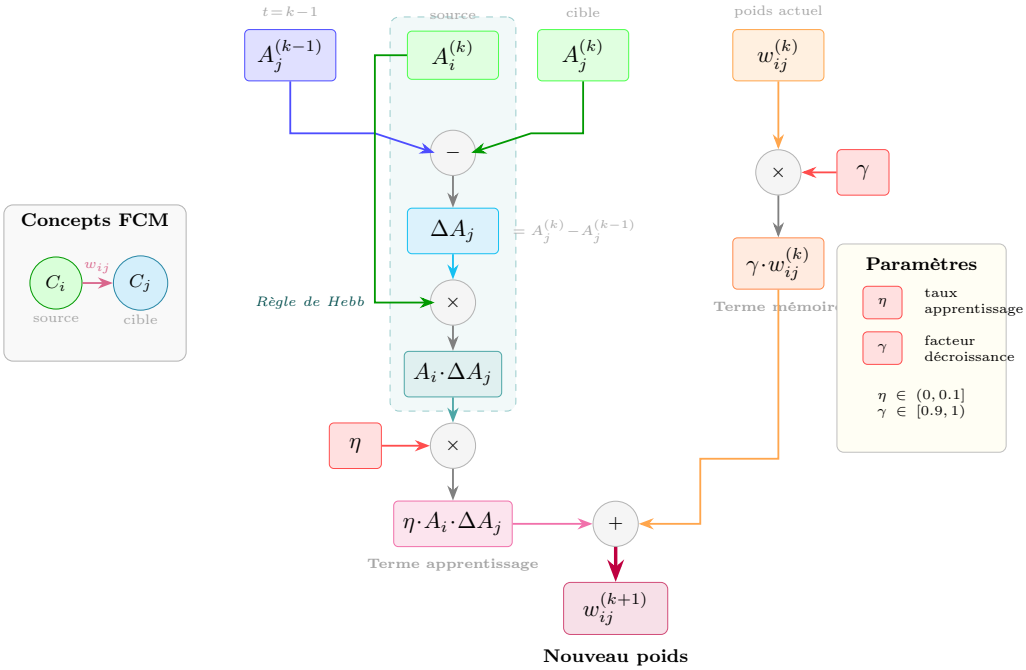


FIGURE 13: Décomposition du flux de calcul NHL : les activations aux instants $k-1$ et k sont combinées pour produire le terme d'apprentissage Hebbien, additionné au terme de mémoire.

14. Exemple de modélisation FCM pour la Priorisation de Vulnérabilités

14.1. Architecture du modèle

Exemple d'architecture FCM à trois couches modélisant l'écosystème de sécurité :

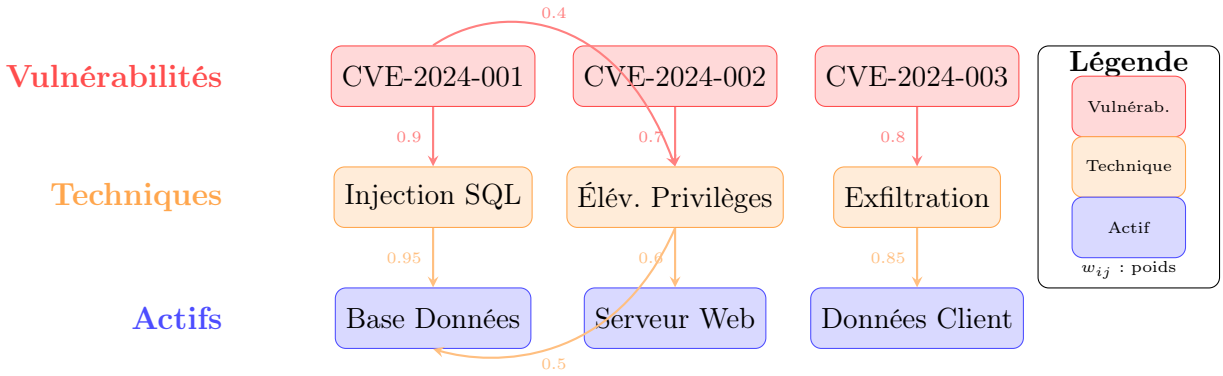


FIGURE 14: Architecture FCM tri-couche pour la modélisation des risques

14.2. Exemple Algorithme Nonlinear Hebbian Learning (NHL) python

L'algorithme NHL est une variante populaire pour les FCM (Papageorgiou, 2003) [6], exemple de mise en oeuvre :

```
class FCMWithLearning(FCMWithInterventions):
    """
    FCM avec capacites d'apprentissage
    """
    def __init__(self, n_concepts):
        super().__init__(n_concepts)

    def nonlinear_hebbian_learning(self, target_state,
                                   learning_rate=0.01,
                                   weight_decay=0.1,
                                   max_iterations=100):
        """
        Apprentissage Hebbien Non Lineaire

        Parametres:
        -----------
        target_state : array-like
            Etat cible desire
        learning_rate : float
            Taux d'apprentissage (eta)
        weight_decay : float
```

```
            Facteur de declin des poids (gamma)
        max_iterations : int
            Nombre maximum d'iterations
        """
        target = np.array(target_state)

        for iteration in range(max_iterations):
            # Simulation d'une etape
            new_state = self.kosko_inference(
                transfer_func='sigmoid',
                lambda_param=1
            )

            # Calcul de l'erreur
            error = target - new_state

            # Mise a jour des poids selon la regle de Hebb
            for i in range(self.n_concepts):
                for j in range(self.n_concepts):
                    if i != j:  # Pas d'auto-connexion
                        # Regle de Hebb non lineaire
                        delta_w = (learning_rate * error[i] *
                                   self.state[j] *
                                   (1 - abs(self.weights[i, j
]))))

                        # Declin du poids
                        self.weights[i, j] = (
                            (1 - weight_decay) * self.weights
[i, j] +
                            delta_w
                        )

            # Mise a jour de l'etat
            self.state = new_state

            # Verifier la convergence
            if np.allclose(new_state, target, atol=0.01):
                print(f"Apprentissage converge a l'iteration
{iteration}")
                break

        return self.weights
```

Listing 16: Implémentation de l'algorithme NHL

14.3. Concepts du modèle d'exemple précédent de priorisations de vulnérabilités

Le tableau 3 présente les concepts intégrés au modèle :

TABLE 3: Concepts FCM pour la priorisation de vulnérabilités

Catégorie	**Concept**	**Description**
Vulnérabilité	CVSS_Base	Score de base CVSS normalisé $[0, 1]$
	Exploitabilité	Probabilité d'exploitation (EPSS)
	Maturité_Exploit	Existence de code d'exploitation public
Technique	Complexité	Difficulté technique de l'attaque
	Chaînage	Potentiel de chaînage avec autres CVE
Contexte	Criticité_Actif	Importance de l'actif ciblé
	Exposition	Niveau d'exposition réseau
	Contrôles	Présence de contrôles compensatoires
Sortie	Priorité	Score de priorité résultant $[0, 1]$

15. Implémentation Python

Note sur l'apprentissage NHL : Dans l'exemple ci-dessus, la matrice de poids est définie manuellement par un expert. En présence de données historiques (vulnérabilités passées avec priorités connues), l'algorithme NHL (Section 13.5) peut être utilisé pour **apprendre** les poids optimaux avant l'étape d'inférence.

Voir scripts complets dans https://github.com/blue101010/fuzycognitivemap

15.1. Classe FCM de base

```
import numpy as np

class FCM:
    """Carte Cognitive Floue avec apprentissage NHL."""

    def __init__(self, n_concepts, lambda_=5.0):
```

```
        self.n = n_concepts
        self.W = np.zeros((n_concepts, n_concepts))
        self.lambda_ = lambda_
        self.A = np.zeros(n_concepts)

    def sigmoid(self, x):
        """Fonction de transfert sigmoide."""
        return 1 / (1 + np.exp(-self.lambda_ * x))

    def set_weights(self, weight_matrix):
        """Definit la matrice des poids."""
        self.W = np.array(weight_matrix)

    def infer(self, initial_state, max_iter=50, threshold=1e
-4,
              clamp_indices=None):
    """
    Inference FCM jusqu'a convergence.

    Parametres:
    -----------
    initial_state : array-like
        Vecteur d'activation initial
    max_iter : int
        Nombre maximum d'iterations
    threshold : float
        Seuil de convergence
    clamp_indices : list of int, optional
        Indices des concepts a verrouiller (valeurs fixes)

    Retourne:
    ---------
    tuple : (etat_final, historique)
    """
    self.A = np.array(initial_state, dtype=float)
    history = [self.A.copy()]

    # Sauvegarder les valeurs des noeuds verrouilles
    if clamp_indices is None:
        clamp_indices = []
    clamped_values = {i: self.A[i] for i in clamp_indices}

    for iteration in range(max_iter):
        # Calcul du nouvel etat (regle de Kosko)
        A_new = self.sigmoid(np.dot(self.W.T, self.A))

        # Restaurer les valeurs des noeuds verrouilles
        for idx, val in clamped_values.items():
            A_new[idx] = val
```

```
55
56          # Stocker l'etat AVANT verification de convergence
57          history.append(A_new.copy())
58
59          # Verification de convergence
60          if np.max(np.abs(A_new - self.A)) < threshold:
61              self.A = A_new
62              break
63          self.A = A_new
64
65      return self.A, np.array(history)
```

Listing 17: Implémentation de la classe FCM

15.2. Algorithme NHL

```
1  def nhl_learning(self, data_sequences, eta=0.1, gamma=0.9,
2                   epochs=100):
3      """
4      Apprentissage Hebbien Non-Lineaire.
5
6      Parametres:
7      -----------
8      data_sequences : list of np.array
9          Sequences temporelles d'activations observees
10     eta : float
11         Taux d'apprentissage (defaut: 0.1)
12     gamma : float
13         Facteur de decroissance (defaut: 0.9)
14     epochs : int
15         Nombre d'epoques d'entrainement
16     """
17     for epoch in range(epochs):
18         total_error = 0
19
20         for sequence in data_sequences:
21             for t in range(1, len(sequence)):
22                 A_prev = sequence[t-1]
23                 A_curr = sequence[t]
24                 delta_A = A_curr - A_prev
25
26                 # Mise a jour NHL (equation 4)
27                 for i in range(self.n):
28                     for j in range(self.n):
29                         if i != j:
30                             delta_w = eta * A_prev[i] *
    delta_A[j]
31                             self.W[i,j] = gamma * self.W[i,j]
    + delta_w
```

```
                            # Contrainte [-1, 1]
                            self.W[i,j] = np.clip(self.W[i,j
    ], -1, 1)

                # Calcul erreur
                A_pred = self.sigmoid(np.dot(self.W.T, A_prev
    ))
                total_error += np.mean((A_curr - A_pred)**2)

        if epoch % 10 == 0:
            print(f"Epoch {epoch}, MSE: {total_error:.6f}")

    return self.W
```

Listing 18: Implémentation de l'apprentissage NHL

15.2.1. Inférence avec verrouillage de nœuds

Dans certaines applications, il est nécessaire de maintenir certains concepts à des valeurs fixes pendant l'inférence, tandis que d'autres concepts évoluent librement vers l'équilibre. Cette technique, appelée **verrouillage de nœuds** (*node clamping*), permet de traiter les concepts d'entrée comme des *évidences* fixes et d'observer uniquement la propagation vers les concepts de sortie. Par exemple, dans un modèle de priorisation de vulnérabilités, les caractéristiques observées (CVSS, EPSS, criticité) sont verrouillées, et seul le concept de priorité est libre d'évoluer selon les influences causales définies dans la matrice de poids.

Le **verrouillage de nœuds** (*node clamping*) consiste à fixer les valeurs d'un sous-ensemble de concepts $\mathcal{C}_{\text{fix}} \subset \mathcal{C}$ pendant l'inférence itérative. À chaque itération k, après application de la règle de mise à jour $A_i^{(k+1)} = f(\sum_j w_{ji} A_j^{(k)})$, les concepts verrouillés sont restaurés à leur valeur initiale : $A_i^{(k+1)} \leftarrow A_i^{(0)}$ pour tout $i \in \mathcal{C}_{\text{fix}}$. Cette approche évite que le réseau converge vers un attracteur unique indépendant des entrées, garantissant ainsi que les concepts de sortie reflètent fidèlement les paramètres fournis.

15.3. Application à la priorisation

```
def prioritize_vulnerabilities(vulnerabilities, fcm_model):
    """
    Priorise une liste de vulnerabilites.

    Parametres:
    -----------
    vulnerabilities : list of dict
        Liste avec cles: 'cve_id', 'cvss', 'epss',
        'exploit_mature', 'asset_criticality', 'exposure'
```

```
    fcm_model : FCM
        Modele FCM entraine

    Retourne:
    ---------
    list : Vulnerabilites triees par priorite decroissante
    """
    results = []

    for vuln in vulnerabilities:
        # Construction du vecteur d'activation initial
        initial_state = np.array([
            vuln['cvss'] / 10.0,              # C0: CVSS
normalise
            vuln['epss'],                      # C1: EPSS
            vuln['exploit_mature'],            # C2: Maturite
 exploit
            0.5,                               # C3:
Complexite (defaut)
            vuln.get('chaining', 0.3),         # C4:
Potentiel chainage
            vuln['asset_criticality'],         # C5:
Criticite actif
            vuln['exposure'],                  # C6:
Exposition
            vuln.get('controls', 0.5),         # C7:
Controles
            0.0                                # C8: Priorite
 (sortie)
        ])

        # Inference FCM avec verrouillage des noeuds d'entree
 (C0-C7)
        # Seul le noeud de sortie (C8: Priorite) est libre d'
evoluer
        clamped_nodes = list(range(len(initial_state) - 1))
        final_state, history = fcm_model.infer(
            initial_state,
            clamp_indices=clamped_nodes
        )
        priority_score = final_state[-1]  # Dernier concept =
 Priorite

        results.append({
            'cve_id': vuln['cve_id'],
            'priority_score': priority_score,
            'final_state': final_state,
            'iterations': len(history) - 1
        })
```

```
48
49      # Tri par priorite decroissante
50      results.sort(key=lambda x: x['priority_score'], reverse=
        True)
51      return results
```

Listing 19: Priorisation de vulnérabilités avec FCM et verrouillage des nœuds d'entrée

15.4. Active Hebbian Learning (AHL)

L'algorithme AHL améliore NHL en permettant une séquence d'activation des concepts [12] :

```
    def active_hebbian_learning(self, training_sequences,
                                learning_rate=0.01,
                                weight_decay=0.1):
        """
        Active Hebbian Learning

        Parametres:
        -----------
        training_sequences : list of tuples
            Liste de (etat_initial, etat_cible)
        learning_rate : float
            Taux d'apprentissage
        weight_decay : float
            Facteur de declin des poids
        """
        for sequence_idx, (initial, target) in enumerate(
training_sequences):
            print(f"\nSequence {sequence_idx + 1}/{len(
training_sequences)}")

            # Initialiser avec l'etat initial
            self.set_initial_state(initial)
            target = np.array(target)

            # Apprentissage pour cette sequence
            for iteration in range(50):
                # Calculer le nouvel etat
                new_state = self.kosko_inference(
                    transfer_func='sigmoid',
                    lambda_param=1
                )

                # Calculer l'erreur
                error = target - new_state

                # Mise a jour des poids
                for i in range(self.n_concepts):
                    for j in range(self.n_concepts):
                        if i != j:
                            # Contribution de l'etat actuel
                            activation = self.state[j]

                            # Mise a jour selon AHL
                            delta_w = (learning_rate * error[
i] *
```

```
                                        activation *
                                        (1 - abs(self.weights[i
    , j])))

                            self.weights[i, j] = (
                                (1 - weight_decay) * self.
    weights[i, j] +
                                delta_w
                            )

                # Mise a jour de l'etat
                self.state = new_state

                # Verifier la convergence
                if np.allclose(new_state, target, atol=0.05):
                    print(f"  Converge a l'iteration {
    iteration}")
                    break

        return self.weights
```

Listing 20: Algorithme Active Hebbian Learning

15.5. Exemple : Apprendre à Partir de Données Historiques

```
# Creer une FCM vierge
learning_fcm = FCMWithLearning(n_concepts=5)

# Donnees d'entrainement : observations historiques
# Format: (etat_initial, etat_final_observe)
training_data = [
    # Observation 1
    ([0.8, 0.6, 0.3, 0.7, 0.5],
     [0.75, 0.65, 0.2, 0.8, 0.75]),

    # Observation 2
    ([0.5, 0.8, 0.6, 0.4, 0.4],
     [0.55, 0.82, 0.5, 0.5, 0.6]),

    # Observation 3
    ([0.9, 0.7, 0.2, 0.8, 0.6],
     [0.88, 0.72, 0.15, 0.85, 0.85]),
]

print("=== Apprentissage Automatique des Poids ===\n")

# Apprentissage
learned_weights = learning_fcm.active_hebbian_learning(
    training_sequences=training_data,
```

```
    learning_rate=0.05,
    weight_decay=0.05
)

print("\n=== Matrice de Poids Apprise ===")
print(learned_weights)

# Test sur une nouvelle situation
print("\n=== Test sur Nouvelle Situation ===")
test_state = [0.7, 0.75, 0.4, 0.6, 0.5]
learning_fcm.set_initial_state(test_state)
test_results = learning_fcm.simulate(iterations=20)
print("Etat predit:", test_results['final_state'])
```

Listing 21: Apprentissage à partir de données observées

16. Analyse d'incertitude avec Monte Carlo

Dans le monde réel, nos estimations des poids et des états initiaux sont rarement certaines. L'analyse Monte Carlo [14] nous permet de quantifier cette incertitude [15].

16.1. Principe de l'Analyse Monte Carlo

Au lieu d'une seule simulation avec des valeurs fixes, nous :

1. Définissons des distributions de probabilité pour les paramètres incertains
2. Générons de nombreux échantillons aléatoires de ces paramètres
3. Simulons la FCM pour chaque échantillon
4. Analysons statistiquement les résultats

16.2. Implémentation

```
import matplotlib.pyplot as plt
from scipy import stats

class FCMMonteCarloAnalysis:
    """
    Classe pour l'analyse Monte Carlo des FCM
    """
    def __init__(self, base_fcm):
        """
```

14. Méthode Monte Carlo : technique utilisant des échantillons aléatoires pour estimer des résultats probabilistes

```
        Parametres:
        -----------
        base_fcm : SimpleFCM
            FCM de base pour l'analyse
        """
        self.base_fcm = base_fcm
        self.results = []

    def run_monte_carlo(self, n_simulations=1000,
                        weight_std=0.1,
                        initial_state_std=0.05):
        """
        Execute une analyse Monte Carlo

        Parametres:
        -----------
        n_simulations : int
            Nombre de simulations Monte Carlo
        weight_std : float
            Ecart-type pour la perturbation des poids
        initial_state_std : float
            Ecart-type pour la perturbation de l'etat initial
        """
        self.results = []
        base_weights = self.base_fcm.weights.copy()
        base_initial = self.base_fcm.state.copy()

        print(f"Execution de {n_simulations} simulations
    Monte Carlo...")

        for sim in range(n_simulations):
            # Perturber les poids (distribution normale)
            perturbed_weights = base_weights + np.random.
    normal(
                0, weight_std, base_weights.shape
            )
            # Garder les poids dans [-1, 1]
            perturbed_weights = np.clip(perturbed_weights,
    -1, 1)

            # Perturber l'etat initial
            perturbed_initial = base_initial + np.random.
    normal(
                0, initial_state_std, base_initial.shape
            )
            # Garder dans [0, 1]
            perturbed_initial = np.clip(perturbed_initial, 0,
     1)

```

```
            # Creer une FCM avec parametres perturbes
            temp_fcm = SimpleFCM(self.base_fcm.n_concepts)
            temp_fcm.weights = perturbed_weights
            temp_fcm.set_initial_state(perturbed_initial)

            # Simuler
            result = temp_fcm.simulate(
                iterations=30,
                inference='kosko',
                transfer_func='sigmoid',
                threshold=0.001
            )

            self.results.append(result['final_state'])

            if (sim + 1) % 100 == 0:
                print(f"  {sim + 1}/{n_simulations}
simulations completees")

        self.results = np.array(self.results)
        return self.results

    def analyze_results(self):
        """
        Analyse statistique des resultats Monte Carlo
        """
        print("\n=== Analyse Statistique Monte Carlo ===\n")

        for concept_id in range(self.results.shape[1]):
            values = self.results[:, concept_id]

            print(f"Concept C{concept_id + 1}:")
            print(f"  Moyenne      : {np.mean(values):.4f}")
            print(f"  Ecart-type   : {np.std(values):.4f}")
            print(f"  Min          : {np.min(values):.4f}")
            print(f"  Max          : {np.max(values):.4f}")
            print(f"  Quartile 25% : {np.percentile(values,
25):.4f}")
            print(f"  Mediane      : {np.median(values):.4f}"
)
            print(f"  Quartile 75% : {np.percentile(values,
75):.4f}")
            print()

    def plot_distributions(self, concept_names=None):
        """
        Visualise les distributions des resultats
        """
        n_concepts = self.results.shape[1]
```

```
        if concept_names is None:
            concept_names = [f"C{i+1}" for i in range(
    n_concepts)]

        fig, axes = plt.subplots(2, 3, figsize=(15, 10))
        axes = axes.flatten()

        for i in range(n_concepts):
            ax = axes[i]
            values = self.results[:, i]

            # Histogramme
            ax.hist(values, bins=30, alpha=0.7,
                    color='steelblue', edgecolor='black')

            # Ligne verticale pour la moyenne
            mean_val = np.mean(values)
            ax.axvline(mean_val, color='red',
                       linestyle='--', linewidth=2,
                       label=f'Moyenne: {mean_val:.3f}')

            # Intervalles de confiance (95%)
            ci_low = np.percentile(values, 2.5)
            ci_high = np.percentile(values, 97.5)
            ax.axvline(ci_low, color='green',
                       linestyle=':', linewidth=1.5)
            ax.axvline(ci_high, color='green',
                       linestyle=':', linewidth=1.5)

            ax.set_title(f'{concept_names[i]}')
            ax.set_xlabel('Valeur finale')
            ax.set_ylabel('Frequence')
            ax.legend()
            ax.grid(True, alpha=0.3)

        plt.tight_layout()
        plt.savefig('monte_carlo_distributions.png', dpi=300)
        print("Graphique sauvegarde:
    monte_carlo_distributions.png")
```

Listing 22: Analyse Monte Carlo pour FCM

16.3. Exemple d'Utilisation

```
# Creer la FCM de base
health_fcm = SimpleFCM(n_concepts=5)

# Configuration (comme precedemment)
health_fcm.set_weight(0, 4, 0.7)  # Exercice -> Sante
```

```
health_fcm.set_weight(1, 4, 0.8)  # Alimentation -> Sante
health_fcm.set_weight(2, 4, -0.6) # Stress -> Sante
health_fcm.set_weight(3, 4, 0.7)  # Sommeil -> Sante
health_fcm.set_weight(2, 3, -0.5) # Stress -> Sommeil
health_fcm.set_weight(0, 2, -0.4) # Exercice -> Stress

initial = [0.8, 0.6, 0.7, 0.5, 0.4]
health_fcm.set_initial_state(initial)

# Analyse Monte Carlo
mc_analysis = FCMMonteCarloAnalysis(health_fcm)

# Executer 1000 simulations avec incertitude
results = mc_analysis.run_monte_carlo(
    n_simulations=1000,
    weight_std=0.1,      # 10% d'incertitude sur les poids
    initial_state_std=0.05  # 5% sur l'etat initial
)

# Analyser
mc_analysis.analyze_results()

# Visualiser
concept_names = ['Exercice', 'Alimentation', 'Stress',
                 'Sommeil', 'Sante']
mc_analysis.plot_distributions(concept_names)
```

Listing 23: Exemple complet d'analyse Monte Carlo

17. Applications et Cas d'Usage

Les FCM sont utilisées dans de nombreux domaines. Explorons quelques applications majeures [16].

17.1. Médecine et Santé

Diagnostic médical : Les FCM peuvent modéliser les relations entre symptômes, facteurs de risque et maladies [17].

Exemple : Diagnostic du Diabète

```
# Concepts:
# C1: Glycemie a jeun
# C2: IMC (Indice de Masse Corporelle)
# C3: Antecedents familiaux
# C4: Activite physique
# C5: Risque de diabete

diabetes_fcm = SimpleFCM(n_concepts=5)

```

```
# Relations causales
diabetes_fcm.set_weight(0, 4, 0.9)   # Glycemie -> Risque (forte)
diabetes_fcm.set_weight(1, 4, 0.7)   # IMC -> Risque
diabetes_fcm.set_weight(2, 4, 0.6)   # Antecedents -> Risque
diabetes_fcm.set_weight(3, 4, -0.5)  # Activite -> Risque (negative)
diabetes_fcm.set_weight(1, 0, 0.4)   # IMC -> Glycemie
diabetes_fcm.set_weight(3, 1, -0.3)  # Activite -> IMC (negative)

# Patient avec facteurs de risque eleves
patient = [0.8, 0.7, 0.9, 0.3, 0.0]
diabetes_fcm.set_initial_state(patient)

result = diabetes_fcm.simulate(iterations=20)
risk_score = result['final_state'][4]
print(f"Score de risque de diabete: {risk_score:.3f}")

if risk_score > 0.7:
    print("Risque ELEVE - Consultation recommandee")
elif risk_score > 0.5:
    print("Risque MODERE - Surveillance recommandee")
else:
    print("Risque FAIBLE - Continuer prevention")
```

Listing 24: FCM pour le diagnostic du diabète

17.2. Gestion Environnementale

Les FCM sont particulièrement utiles pour modéliser des écosystèmes complexes [9].

Exemple : Gestion d'un Écosystème Forestier

```
# Concepts:
# C1: Couverture forestiere
# C2: Biodiversite
# C3: Qualite du sol
# C4: Activites humaines (exploitation)
# C5: Precipitations
# C6: Temperature

forest_fcm = SimpleFCM(n_concepts=6)

# Relations ecologiques
forest_fcm.set_weight(0, 1, 0.8)   # Foret -> Biodiversite
forest_fcm.set_weight(0, 2, 0.7)   # Foret -> Sol
forest_fcm.set_weight(1, 2, 0.6)   # Biodiversite -> Sol
forest_fcm.set_weight(2, 0, 0.5)   # Sol -> Foret
forest_fcm.set_weight(3, 0, -0.7)  # Exploitation -> Foret
```

```
forest_fcm.set_weight(4, 0, 0.6)   # Pluie -> Foret
forest_fcm.set_weight(5, 0, -0.4)  # Temperature -> Foret
forest_fcm.set_weight(5, 4, -0.3)  # Temperature -> Pluie

# Scenario: changement climatique + exploitation
initial_state = [0.6, 0.5, 0.5, 0.7, 0.4, 0.8]
forest_fcm.set_initial_state(initial_state)

print("=== Scenario: Changement Climatique + Exploitation ===")
result = forest_fcm.simulate(iterations=25)
print(f"\nCouverture forestiere finale: {result['final_state'][0]:.3f}")
print(f"Biodiversite finale: {result['final_state'][1]:.3f}")
```

Listing 25: FCM pour la gestion forestière

17.3. Gestion d'Entreprise et Stratégie

Les FCM permettent de modéliser des stratégies d'entreprise complexes [18].

```
# Concepts:
# C1: Investissement R&D
# C2: Innovation produit
# C3: Satisfaction client
# C4: Part de marche
# C5: Revenus
# C6: Reputation de marque
# C7: Concurrence

business_fcm = SimpleFCM(n_concepts=7)

# Relations strategiques
business_fcm.set_weight(0, 1, 0.8)   # R&D -> Innovation
business_fcm.set_weight(1, 2, 0.7)   # Innovation -> Satisfaction
business_fcm.set_weight(2, 3, 0.8)   # Satisfaction -> Part de marche
business_fcm.set_weight(3, 4, 0.9)   # Part de marche -> Revenus
business_fcm.set_weight(4, 0, 0.6)   # Revenus -> R&D (reinvestissement)
business_fcm.set_weight(1, 5, 0.7)   # Innovation -> Reputation
business_fcm.set_weight(5, 3, 0.5)   # Reputation -> Part de marche
business_fcm.set_weight(6, 3, -0.6)  # Concurrence -> Part de marche

```

```python
# Scenario de croissance
initial = [0.7, 0.5, 0.6, 0.4, 0.5, 0.6, 0.7]
business_fcm.set_initial_state(initial)

print("=== Strategie de Croissance par Innovation ===")
result = business_fcm.simulate(iterations=30)
print(f"\nPart de marche finale: {result['final_state'][3]:.3
    f}")
print(f"Revenus finaux: {result['final_state'][4]:.3f}")
print(f"Reputation finale: {result['final_state'][5]:.3f}")
```

Listing 26: FCM pour stratégie d'entreprise technologique

18. Bonnes Pratiques et Recommandations

18.1. Construction de FCM Efficaces

1. **Limitation du nombre de concepts** : Privilégier 5 à 15 concepts pour maintenir l'interprétabilité
2. **Validation avec experts** : Toujours valider la structure et les poids avec des experts du domaine
3. **Normalisation des poids** : Garder les poids dans $[-1, 1]$ pour faciliter l'interprétation
4. **Tests de sensibilité** : Tester l'impact de variations des paramètres
5. **Documentation** : Documenter clairement chaque concept et chaque relation

18.2. Choix des Paramètres

TABLE 4: Guide de sélection des paramètres

Paramètre	**Recommandation**
Fonction de transfert	Sigmoïde pour la majorité des cas ; Tanh si valeurs négatives importantes
λ (sigmoïde)	Commencer avec 1 ; augmenter (2-5) si convergence trop lente ; diminuer (0.5) si oscillations
Règle d'inférence	Kosko original pour systèmes sans inertie ; Modifié pour systèmes avec persistance
Seuil de convergence	0.001 typiquement ; ajuster selon la précision requise
Nombre d'itérations	30-50 suffisant dans la plupart des cas

18.3. Pièges à Éviter

- **Surcharge de concepts** : Trop de concepts rendent le modèle difficile à interpréter
- **Biais de confirmation** : Ne pas uniquement chercher à confirmer vos hypothèses
- **Poids arbitraires** : Justifier chaque poids par des données ou expertise
- **Ignorer l'incertitude** : Toujours considérer l'incertitude des estimations
- **Sur-apprentissage** : En apprentissage automatique, attention au surapprentissage

19. Cartes cognitives floues et théorie de Dempster–Shafer

Pour mieux modéliser le raisonnement incertain dans les cartes cognitives floues, (Jia *et al., 2020*) ont proposé [19] une extension intuitionniste des cartes cognitives floues en s'appuyant sur la **théorie de Dempster–Shafer** . Cette approche permet de combiner des degrés de croyance et d'hésitation, offrant ainsi un cadre plus expressif pour la représentation de l'incertitude dans des systèmes complexes.

20. Cartes cognitives floues et cybersécurité

Les cartes cognitives floues sont des outils de modélisation causale qui combinent la logique floue et les réseaux de neurones récurrents pour représenter des systèmes complexes sous forme de graphes dirigés pondérés. Dans le domaine de la cybersécurité, les FCM offrent une approche flexible et interprétable pour analyser les risques, modéliser les scénarios d'attaque, évaluer l'impact des vulnérabilités et simuler les conséquences de mesures de protection.

20.1. Analyse des risques et évaluation de la sécurité

Plusieurs travaux exploitent les FCM pour l'évaluation des risques de sécurité informatique. Szwed et Skrzyński (2014) [20] proposent une méthode légère d'évaluation des risques basée sur les FCM, appliquée à un système de télémédecine e-santé. Leur approche capture les dépendances entre les actifs du système et utilise le raisonnement FCM pour agréger les risques de manière efficace. De manière similaire, Poleto et al. [21] emploient les FCM pour identifier et simuler les facteurs de cybersécurité affectant les services de télésanté, permettant une meilleure compréhension des causes et conséquences des menaces dans un environnement médical connecté.

L'utilisation des FCM pour la sécurité dans le cloud computing est explorée par Napoles et al. [22], qui proposent un processus d'évaluation des risques dynamiques basé sur les cartes cognitives floues pour identifier et atténuer les risques liés aux données hébergées dans le cloud. Par ailleurs, (Soner et al., 2025) [23] étudient la propagation des risques de cybersécurité dans les infrastructures portuaires en utilisant les FCM pour modéliser la sensibilité du système aux facteurs de menace clés et simuler l'impact de scénarios de perturbation.

20.2. Détection d'intrusions et réponse aux incidents

Les FCM sont également appliquées à la détection d'intrusions et à la réponse aux incidents de sécurité. Jazzar et Jantan [24] présentent une approche utilisant les FCM pour réduire les fausses alertes dans les systèmes de détection d'intrusions basés sur les cartes auto-organisatrices (SOM). Leur modèle améliore la précision de la détection en intégrant des relations causales entre les événements suspects. Ramana et Rao [25] proposent une carte cognitive floue contextuelle pour les systèmes de réponse aux intrusions, permettant une prise de décision adaptative en fonction du contexte de l'attaque.

Dans une optique de visualisation et de planification, Oliinyk et Kuznetsov [26] utilisent les FCM comme outil de visualisation des scénarios de réponse aux incidents dans les systèmes de sécurité, facilitant l'analyse et la communication des stratégies de défense. Plus récemment, Shaburov et Vasilyev [27] développent un modèle cognitif flou pour automatiser l'analyse des incidents de cybersécurité, intégrant des mécanismes d'apprentissage pour améliorer la réactivité des équipes de sécurité opérationnelle.

20.3. Modélisation de scénarios d'attaque

Les FCM permettent également de modéliser des scénarios d'attaque complexes et de simuler leur impact sur les systèmes critiques. Shevchenko et al. [28] appliquent les cartes cognitives floues à l'analyse de scénarios pour l'évaluation des risques de sécurité de l'information, en utilisant le logiciel Mental Modeler pour construire des matrices d'influence et générer des scénarios de type *what-if*. Nikonov et Vasilyev [29] modélisent le vecteur d'attaque cybernétique sous forme de graphes hiérarchiques de FCM, permettant une analyse multi-échelle des scénarios d'attaque et l'identification des vulnérabilités critiques dans les infrastructures industrielles.

20.4. Conclusion en cybersécurité

L'ensemble de ces travaux démontre la pertinence et la flexibilité des cartes cognitives floues pour la cybersécurité. Les FCM permettent de capturer la complexité des systèmes, d'intégrer l'expertise humaine, de simuler

des scénarios dynamiques et de fournir des résultats interprétables pour la prise de décision. Leur application s'étend de l'évaluation des risques à la détection d'intrusions, en passant par la modélisation de scénarios d'attaque et la planification de réponses aux incidents.

21. Conclusion et Perspectives

21.1. Avantages des FCM

Les FCM offrent plusieurs avantages distinctifs [2] :

- **Interprétabilité** : La structure graphique rend le modèle transparent
- **Flexibilité** : Facilité d'adaptation à de nouveaux contextes
- **Gestion de l'incertitude** : Capacité à traiter l'imprécision inhérente aux systèmes complexes
- **Expertise + Données** : Possibilité de combiner connaissances d'experts et données empiriques
- **Analyse de scénarios** : Facilité de tester des hypothèses « what-if »

21.2. Limitations et Défis

Malgré leurs avantages, les FCM présentent certaines limitations :

- **Convergence non garantie** : Certaines configurations peuvent ne pas converger
- **Dynamique temporelle limitée** : Difficile de modéliser des délais temporels complexes
- **Validation difficile** : Manque de métriques standard pour évaluer la qualité d'une FCM
- **Subjectivité** : Dépendance forte à l'expertise humaine dans la construction

22. Construction automatique des FCM sans expertise humaine

22.1. Introduction et motivations

Traditionnellement, les Cartes Cognitives Floues (FCM) sont construites à partir des connaissances d'experts du domaine, ce qui présente des limitations en termes de subjectivité, de coût et de disponibilité des experts. Les approches récentes visent à automatiser ce processus, permettant aux FCM d'apprendre à partir des données et d'auto-organiser leur structure [30]. Cette capacité est particulièrement importante pour les applications où les connaissances expertes sont rares ou difficiles à obtenir.

On distingue trois paradigmes de construction des FCM :

— **Expert-based** : construction manuelle basée sur l'expertise humaine
— **Data-driven** : construction entièrement automatisée à partir des données
— **Hybride** : combinaison des deux approches précédentes

22.2. Apprentissage à partir de données temporelles

L'algorithme génétique à codage réel (RCGA) constitue une approche fondamentale pour générer automatiquement une matrice de connexion FCM à partir de données longitudinales. La fonction de fitness utilisée est définie par :

$$\text{Erreur} = \alpha \sum_{t=1}^{T-1} \sum_{j=1}^{N-1} |A_j(t) - \hat{A}_j(t)|^p \tag{6}$$

$$\text{Fitness} = \frac{1}{a \cdot \text{Erreur} + 1} \tag{7}$$

où $A_j(t)$ représente les données observées au temps t, $\hat{A}_j(t)$ le vecteur d'état prédit par la FCM candidate, et α, p, a sont des paramètres de normalisation.

22.3. Taxonomie des algorithmes d'apprentissage

Les techniques d'apprentissage automatique pour les FCM peuvent être catégorisées en trois groupes selon le paradigme d'apprentissage [30] :

22.3.1. Algorithmes basés sur Hebb

Ces algorithmes s'inspirent de la règle d'apprentissage hebbien en neurosciences. La mise à jour des poids suit le principe :

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} + \eta \cdot A_i^{(t)} \cdot A_j^{(t)} \tag{8}$$

où η est le taux d'apprentissage. Les variantes principales incluent :

— Apprentissage Hebbien Différentiel (DHL)
— Apprentissage Hebbien Non-Linéaire (NHL)
— Apprentissage Hebbien Actif (AHL)

22.3.2. Algorithmes basés sur les populations

Ces méthodes exploitent des populations de solutions candidates évoluant selon des mécanismes inspirés de la nature :

Table 5: Algorithmes métaheuristiques pour l'apprentissage automatique des FCM

Algorithme	**Abréviation**	**Inspiration**
Algorithme Génétique	GA	Évolution biologique
AG à Codage Réel	RCGA	Évolution biologique
Essaim de Particules	PSO	Comportement collectif
Évolution Différentielle	DE	Évolution biologique
Colonie de Fourmis	ACO	Intelligence en essaim
Recuit Simulé	SA	Thermodynamique

22.3.3. Algorithmes hybrides

Les approches hybrides combinent les avantages des méthodes hebbiennes (rapidité de convergence) et des méthodes populationnelles (exploration globale de l'espace de recherche).

22.4. Extraction automatique par fouille de textes et TAL

Une approche prometteuse consiste à extraire automatiquement les concepts et leurs relations causales à partir de corpus textuels [**?**]. Le processus comprend plusieurs étapes :

1. **Fouille de textes** (*text mining*) : identification des concepts pertinents dans les documents
2. **Règles d'association floues** (*fuzzy association rule mining*) : découverte des relations causales entre concepts
3. **Analyse des réseaux sociaux** : estimation des poids initiaux des connexions
4. **Inférence FCM** : raffinement des poids par simulation

Cette approche permet d'exploiter des *données prospectives*, c'est-à-dire une collection d'opinions orientées vers l'avenir extraites de communautés en ligne, traitant ainsi les problèmes de subjectivité et de myopie inhérents aux approches traditionnelles.

22.5. Utilisation des grands modèles de langage (LLM)

Les grands modèles de langage (LLM) représentent une avancée majeure pour la construction automatique de FCM. Ces systèmes, entraînés sur de vastes corpus textuels, possèdent des capacités de traitement du langage naturel permettant :

— La **reconnaissance d'entités nommées** (NER) pour l'identification des concepts
— L'**extraction de relations** pour déterminer les liens causaux
— La **génération de texte** pour verbaliser et valider les FCM construites

Une architecture récente propose un processus d'auto-encodage FCM-texte-FCM utilisant des agents LLM avec des instructions système successives pour :

1. Encoder une FCM en description textuelle (latent I)
2. Éditer le contenu pour le rendre plus naturel (latent II)
3. Décoder le texte en FCM par détection de noms, de nœuds et extraction d'arêtes

22.6. Approches par réseaux de neurones

Les réseaux de neurones artificiels offrent une alternative pour l'apprentissage des FCM, notamment en introduisant la notion de *biais* dans le modèle :

$$A_i^{(t+1)} = f\left(b_i + \sum_{j=1}^{n} A_j^{(t)} \cdot w_{ji}\right) \tag{9}$$

où b_i représente le biais du concept i, capturant les influences externes non modélisées par les relations causales explicites. Cette approche permet de :

— Maintenir des poids de connexion cohérents et interprétables
— Découvrir des biais uniques à un système complexe spécifique
— Améliorer la correspondance avec les valeurs réelles observées

22.7. Synthèse comparative des approches

Le tableau 6 présente une synthèse des différentes approches de construction automatique des FCM.

22.8. Défis et perspectives

Malgré les avancées significatives, plusieurs défis persistent dans la construction automatique des FCM :

1. **Passage à l'échelle** : les algorithmes existants sont souvent coûteux en temps de calcul pour les FCM de grande taille (centaines de nœuds)

TABLE 6: Comparaison des approches de construction automatique des FCM

Approche	Source de données	Automatisation	Avantages
RCGA	Données longitudinales	Complète	Optimisation globale
NHL/AHL	Données + FCM initiale	Semi-auto	Convergence rapide
Text Mining + NLP	Documents textuels	Semi-auto	Grande échelle
LLM	Textes descriptifs	Complète	Flexibilité, explicabilité
Réseaux de neurones	Données historiques	Complète	Gestion des biais
PSO/DE/ACO	Données temporelles	Complète	Exploration efficace

2. **Robustesse au bruit** : les données réelles contiennent souvent du bruit, affectant la qualité des FCM apprises
3. **Interprétabilité** : maintenir la signification sémantique des concepts et relations dans les approches entièrement automatisées
4. **Validation** : établir des critères objectifs pour évaluer la qualité des FCM générées automatiquement

Les recherches futures s'orientent vers :
— L'intégration de la théorie intuitionniste et de la théorie des systèmes gris pour gérer l'incertitude
— Le développement de FCM adaptatives ajustant leurs poids en réponse aux conditions changeantes
— L'exploitation des capacités des LLM pour une construction plus intuitive et explicable

23. Cartes Cognitives Floues : Fondements et Implémentation avec FCMpy

FCMpy [31] est un module Python open-source fournissant des outils pour la construction et l'analyse des FCM, incluant :
— La construction de FCM basées sur des données qualitatives d'experts
— La simulation du comportement du système
— Des algorithmes d'apprentissage automatique (NHL, AHL, RCGA)
— L'analyse de scénarios d'intervention

23.1. Construction des FCM basées sur l'expertise

23.1.1. Mesure de l'entropie informationnelle

Pour évaluer le degré d'accord entre les participants concernant les relations causales, on calcule l'entropie informationnelle :

$$R = -\sum_{i=1}^{n} p_i \log_2(p_i) \tag{10}$$

où p_i représente la proportion des réponses pour chaque terme linguistique concernant une relation causale donnée. Une entropie nulle indique un accord parfait entre les experts.

23.1.2. Fonctions d'appartenance floues

Les fonctions d'appartenance triangulaires sont utilisées pour mapper les termes linguistiques sur l'univers de discours $[-1, 1]$. Ces fonctions sont définies par trois paramètres : la borne inférieure, le centre et la borne supérieure du triangle.

23.1.3. Règles d'implication floues

Deux règles d'implication sont implémentées :

Règle du minimum de Mamdani [15] **:**

$$\mu_R(x, y) = \min(\mu_A(x), \mu_B(y)) \tag{11}$$

Règle du produit de Larsen :

$$\mu_R(x, y) = \mu_A(x) \cdot \mu_B(y) \tag{12}$$

où $\mu_A(x)$ et $\mu_B(y)$ désignent respectivement la valeur d'appartenance de x au terme linguistique A et de y au terme B.

23.1.4. Méthodes d'agrégation

Les fonctions d'appartenance activées sont agrégées par l'une des méthodes suivantes :

Maximum familial : $f(x, y) = \max(x, y)$

15. En termes simples, la règle de Ebrahim Mamdani fonctionne comme un « maillon faible » : lorsque plusieurs experts donnent leur avis sur une relation causale, on retient le niveau de certitude le plus faible parmi eux. Par exemple, si un expert est sûr à 80% et un autre à 50%, on conserve 50%. Cette approche prudente évite de surestimer la confiance dans une relation causale lorsqu'il existe des doutes.

Somme algébrique :

$$f(x, y) = x + y - x \cdot y \tag{13}$$

Somme d'Einstein :

$$f(x, y) = \frac{x + y}{1 + x \cdot y} \tag{14}$$

Somme de Hamacher :

$$f(x, y) = \frac{x + y - 2 \cdot x \cdot y}{1 - x \cdot y} \tag{15}$$

23.2. Simulation du comportement du système

23.2.1. Méthodes d'inférence

La dynamique du système est examinée par simulation sur des pas de temps discrets. Trois méthodes d'inférence sont disponibles :

Méthode de Kosko :

$$A_i^{(t+1)} = f\left(\sum_{j=1}^{n} A_j^{(t)} \cdot w_{ji}\right) \tag{16}$$

Méthode de Kosko modifiée :

$$A_i^{(t+1)} = f\left(A_i^{(t)} + \sum_{j=1}^{n} A_j^{(t)} \cdot w_{ji}\right) \tag{17}$$

Méthode rescalée :

$$A_i^{(t+1)} = f\left((2A_i^{(t)} - 1) + \sum_{j=1}^{n} (2A_j^{(t)} - 1) \cdot w_{ji}\right) \tag{18}$$

où $A_j^{(t)}$ est la valeur du concept j au pas de simulation t, w_{ji} est l'impact causal du concept j sur le concept i, et $f(x)$ est une fonction de transfert.

23.2.2. Fonctions de transfert

Sigmoïde :

$$f(x) = \frac{1}{1 + e^{-\lambda x}}, \quad x \in \mathbb{R}, \quad \text{borne les valeurs à } [0, 1] \tag{19}$$

Tangente hyperbolique :

$$f(x) = \tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}, \quad x \in \mathbb{R}, \quad \text{borne les valeurs à } [-1, 1] \tag{20}$$

Bivalente :

$$f(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases}, \quad \text{borne les valeurs à } \{0, 1\} \tag{21}$$

Trivalente :

$$f(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x = 0 \\ -1 & \text{si } x < 0 \end{cases}, \quad \text{borne les valeurs à } \{-1, 0, 1\} \tag{22}$$

23.2.3. Critère de convergence

La simulation s'arrête lorsque l'une des deux conditions est satisfaite :

$$\exists t \in \{1, 2, \ldots, T-1\} : |S^{(t+1)} - S^{(t)}| < \epsilon \tag{23}$$

où S est le vecteur d'activation des concepts d'intérêt, ou lorsqu'un nombre maximal d'itérations est atteint.

23.3. Algorithmes d'apprentissage

23.3.1. Apprentissage Hebbien Non-Linéaire (NHL) et Actif (AHL)

Ces algorithmes optimisent la matrice de connexion FCM pour que les concepts de sortie désirés (DOC - *Desired Output Concepts*) convergent vers une plage souhaitée.

Fonction de fitness F1 :

$$F_1 = \sqrt{\frac{|DOC_j^{(k)} - DOC_j^{min}| \cdot |DOC_j^{max}|^2}{2}} \tag{24}$$

Critère de stabilité F2 :

$$F_2 = |DOC_j^{(k+1)} - DOC_j^{(k)}| < \epsilon \tag{25}$$

où ϵ est un seuil recommandé entre 0.001 et 0.005.

23.3.2. Algorithme Génétique à Codage Réel (RCGA)

Le RCGA recherche une matrice de connexion FCM optimale à partir de données longitudinales. Chaque **chromosome** de la population encode une matrice de poids W candidate.

Calcul de l'erreur :

$$\text{Erreur} = \alpha \sum_{t=1}^{T-1} \sum_{j=1}^{N} |A_j(t) - \hat{A}_j(t)|^p \tag{26}$$

Fonction de fitness :

$$\text{Fitness} = \frac{1}{a \cdot \text{Erreur} + 1} \tag{27}$$

où :

— T : nombre de pas de temps dans la séquence de données observées
— N : nombre de concepts dans la FCM
— $A_j(t)$: valeur observée du concept j au temps t
— $\hat{A}_j(t)$: valeur prédite par la FCM candidate au temps t
— p : exposant définissant le type de norme ($p = 1$ pour norme L1, $p = 2$ pour norme L2)
— α : paramètre de normalisation
— a : paramètre d'échelle pour la fonction de fitness

23.4. Analyse de scénarios

L'analyse de scénarios dans le cadre FCM s'implémente de deux manières :

— **Interventions ponctuelles (single shot)** : modification des valeurs initiales des concepts
— **Interventions continues** : introduction d'un nouveau facteur dans la FCM avec spécification de son impact causal sur les concepts cibles

L'efficacité d'une intervention peut être ajustée par un paramètre dans l'intervalle $[0, 1]$, permettant de simuler des scénarios où l'intervention n'est pas délivrée à son plein potentiel.

24. Directions Futures des FCM

Le domaine des FCM continue d'évoluer. Voici quelques directions prometteuses :

1. **FCM Quantiques (Q-FCM)** : Exploitation de l'informatique quantique pour des FCM plus puissantes [32]
2. **Neuro-Fuzzy Cognitive Maps (NFCM)** : Intégration avec l'apprentissage profond [33]

3. **FCM Dynamiques** : Modèles capables de s'adapter automatiquement aux changements
4. **FCM Hiérarchiques** : Systèmes multi-niveaux pour des problèmes très complexes
5. **Explicabilité de l'IA** : Utilisation des FCM pour expliquer les décisions des systèmes d'IA

24.1. Ressources pour aller plus loin

Pour approfondir vos connaissances :

— **Bibliothèques Python** : FCMpy [34, 31], In-Cognitive [15], PyFCM [16]
— **Logiciels** : Mental Modeler [9], FCM Expert [35], FCM-VSS [36]
— **Littérature** : Articles dans *Fuzzy Sets and Systems* [37], revues de synthèse [16, 30]

24.2. Mot de la fin

Les Cartes Cognitives Floues représentent un outil puissant et accessible pour modéliser et comprendre les systèmes complexes. Leur force réside dans leur capacité à combiner intuition humaine et rigueur computationnelle, tout en restant interprétables. Une version modernisée de FCMpy (compatible Python 3.14) est disponible ici : https://github.com/blue101010/FCMpy/blob/master/tutorial-fr.md

16. Disponible sur https://github.com/mpuig/pyfcm

Références

[1] B. Kosko, Fuzzy cognitive maps, International Journal of Man-Machine Studies 24 (1) (1986) 65–75. doi:10.1016/S0020-7373(86)80040-2. (Cité pages 6 et 13.)

[2] E. I. Papageorgiou, J. L. Salmeron, Methods and algorithms for fuzzy cognitive map learning, Fuzzy Cognitive Maps for Applied Sciences and Engineering (2014) 1–30 doi:10.1007/978-3-642-39739-4_1. (Cité pages 6 et 56.)

[3] R. Axelrod, Structure of Decision: The Cognitive Maps of Political Elites, Princeton University Press, Princeton, NJ, 1976.
URL https://books.google.com/books?id=aKh9BgAAQBAJ&printsec=frontcover&source=gbs_atb (Cité page 7.)

[4] L. A. Zadeh, Fuzzy sets, Information and Control 8 (3) (1965) 338–353. doi:10.1016/S0019-9958(65)90241-X. (Cité page 7.)

[5] C. D. Stylios, P. P. Groumpos, Modeling complex systems using fuzzy cognitive maps, IEEE Transactions on Systems, Man, and Cybernetics, Part A : Systems and Humans 34 (1) (2004) 155–162. doi:10.1109/TSMCA.2003.818878. (Cité pages 7 et 13.)

[6] E. I. Papageorgiou, Fuzzy cognitive maps learning and evolution in complex systems, Soft Computing 7 (4) (2003) 245–254. doi:10.1007/s00500-002-0227-0. (Cité pages 11 et 36.)

[7] E. I. Papageorgiou, A new method for fuzzy cognitive map learning, Fuzzy Sets and Systems 166 (1) (2011) 81–99. doi:10.1016/j.fss.2010.12.015. (Cité page 13.)

[8] B. Kosko, Hidden patterns in combined and adaptive knowledge networks, International Journal of Approximate Reasoning 2 (4) (1988) 377–393. doi:10.1016/0888-613X(88)90025-1. (Cité page 14.)

[9] S. A. Gray, S. Gray, L. J. Cox, S. Henly-Shepard, Mental modeler: A fuzzy-logic cognitive mapping modeling tool for adaptive environmental management, in : 2013 46th Hawaii International Conference on System Sciences (HICSS), 2013, pp. 965–973. doi:10.1109/HICSS.2013.399.
URL https://ieeexplore.ieee.org/document/6479949 (Cité pages 16, 28, 51 et 65.)

[10] A. J. Jetter, K. Kok, Fuzzy cognitive maps for futures studies : A methodological assessment of concepts and methods, Futures 61 (2014) 45–57. doi:10.1016/j.futures.2014.05.002. (Cité page 31.)

[11] B. Kosko, Fuzzy Engineering, Prentice Hall, Upper Saddle River, NJ, 1997.
URL https://archive.org/details/fuzzyengineering0000kosk (Cité page 31.)

[12] E. I. Papageorgiou, C. D. Stylos, P. P. Groumpos, Active hebbian learning algorithm for fuzzy cognitive maps, International Journal of Approximate Reasoning 37 (3) (2004) 219–241. doi:10.1016/j.ijar.2004.06.006. (Cité pages 33 et 44.)

[13] J. A. Dickerson, B. Kosko, Virtual worlds as fuzzy cognitive maps, Presence : Teleoperators and Virtual Environments 3 (2) (1994) 173–189, author PDF : https://sipi.usc.edu/~kosko/Virtual_Worlds_FCM.pdf. doi:10.1162/pres.1994.3.2.173.
URL https://direct.mit.edu/pvar/article/3/2/173/58827/Virtual-Worlds-as-Fuzzy-Cognitive-Maps (Cité page 33.)

[14] E. I. Papageorgiou, C. D. Stylios, P. P. Groumpos, Fuzzy cognitive map learning based on nonlinear hebbian rule, in : AI 2003 : Advances in Artificial Intelligence, Vol. 2903 of Lecture Notes in Computer Science, Springer, 2003, pp. 256–268. doi:10.1007/978-3-540-24581-0_22.
URL https://link.springer.com/chapter/10.1007/978-3-540-24581-0_22 (Cité page 34.)

[15] T. Koutsellis, G. Xexakis, A. Karamaneas, A. Nikas, H. Doukas, Incognitive : A web-based python application for fuzzy cognitive map modeling, simulation, and uncertainty analysis based on the monte carlo method, SoftwareX 23 (2023) 101513. doi:10.1016/j.softx.2023.101513. (Cité pages 46, 65 et 72.)

[16] E. I. Papageorgiou, J. L. Salmeron, A review of fuzzy cognitive maps research during the last decade, IEEE Transactions on Fuzzy Systems 21 (1) (2013) 66–79. doi:10.1109/TFUZZ.2012.2201727. (Cité pages 50 et 65.)

[17] V. C. Georgopoulos, C. D. Stylios, P. P. Groumpos, Fuzzy cognitive maps in medical decision making, Applied Soft ComputingOften misciteds ; see later works by the same authors in medical FCM decision support (2003).

URL https://scholar.google.com/citations?user=1Rbv0LsAAAAJ (Cité page 50.)

[18] C. Mohr, Software for fuzzy cognitive maps, rensselaer Polytechnic Institute (1997). (Cité page 52.)

[19] Z. Jia, Y. Zhang, X. Dong, An extended intuitionistic fuzzy cognitive map via dempster-shafer theory, IEEE Access 8 (2020) 28359–28372, received January 13, 2020 ; accepted January 24, 2020 ; published January 29, 2020 ; current version February 6, 2020. doi:10.1109/ACCESS.2020.2970159.
URL https://ieeexplore.ieee.org/document/8974207 (Cité page 54.)

[20] P. Szwed, P. Skrzyński, A new lightweight method for security risk assessment based on fuzzy cognitive maps, International Journal of Applied Mathematics and Computer Science 24 (1) (2014) 213–225. doi:10.2478/amcs-2014-0016. (Cité page 54.)

[21] T. Poleto, A. P. H. d. G. Costa, C. R. d. S. d. C. Falcão, Fuzzy cognitive mapping for telehealth cybersecurity causes, Healthcare 9 (11) (2021) 1475. doi:10.3390/healthcare9111475. (Cité page 54.)

[22] G. Napoles, I. Grau, Y. Salgueiro, Dynamic security in cloud computing based on fuzzy cognitive maps, Journal of Intelligent & Fuzzy Systems 45 (2023) 1–12. doi:10.3233/JIFS-239398. (Cité page 55.)

[23] O. Soner, G. Kayisoglu, G. D. Ceyhun, Modeling and analyzing cybersecurity risk propagation in ports using fuzzy cognitive maps: System sensitivity to key threat factors, Ocean & Coastal Management 262 (2025) 107549, 10.1016/j.ocecoaman.2024.107549.
URL https://www.sciencedirect.com/science/article/abs/pii/S0964569125003199 (Cité page 55.)

[24] M. Jazzar, A. B. Jantan, Using fuzzy cognitive maps to reduce false alerts in som-based intrusion detection sensors, in : Second Asia International Conference on Modelling & Simulation (AMS 2008), IEEE Computer Society, 2008, pp. 54–59. doi:10.1109/AMS.2008.126. (Cité page 55.)

[25] K. Ramana, K. R. Rao, Contextual fuzzy cognitive map for intrusion response system, International Journal of Computer Information Technology 2 (3) (2013) 518–523.

URL https://ijcit.com/archives/volume2/issue3/Paper020318.pdf (Cité page 55.)

[26] A. Oliinyk, M. Kuznetsov, Fuzzy cognitive maps as a tool for visualizing incident response scenarios in security systems, Cybersecurity : Education, Science, Technique 44 (1) (2019) 34–42, searchable via Google Scholar and OAJI.net.
URL https://www.researchgate.net/publication/387402596_FUZZY_COGNITIVE_MAPS_AS_A_TOOL_FOR_VISUALIZING_INCIDENT_RESPONSE_SCENARIOS_IN_SECURITY_SYSTEMS (Cité page 55.)

[27] A. S. Shaburov, V. I. Vasilyev, Development of a fuzzy cognitive model for automating the analysis of cybersecurity incidents, in : 2024 International Russian Smart Industry Conference (SmartIndustryCon), IEEE, 2024, pp. 1–6. doi:10.1109/SmartIndustryCon61328.2024.10468453. (Cité page 55.)

[28] S. Shevchenko, Y. Zhdanova, O. Kryvytska, H. Shevchenko, S. Spasiteleva, Fuzzy cognitive mapping as a scenario approach for information security risk analysis, in : Workshop on Cybersecurity Providing in Information and Telecommunication Systems II (CPITS-II 2024), Vol. 3826, CEUR-WS.org, 2024, pp. 357–362.
URL https://ceur-ws.org/Vol-3826/short28.pdf (Cité page 55.)

[29] A. V. Nikonov, V. I. Vasilyev, Modeling the cyber attacks vector based on fuzzy cognitive maps, in : VII International Conference on Information Technology and Nanotechnology (ITNT 2021), 2021, pp. 1–5.
URL https://repo.ssau.ru/handle/Informacionnye-tehnologii-i-nanotehnologii/Modeling-the-cyber-attacks-vector-based-on-fuzzy-cognitive-maps-102945 (Cité page 55.)

[30] E. I. Papageorgiou, Learning algorithms for fuzzy cognitive maps—a review study, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42 (2) (2012) 150–163. doi:10.1109/TSMCC.2011.2138694.
URL https://ieeexplore.ieee.org/document/5766766 (Cité pages 57 et 65.)

[31] S. Mkhitaryan, Fcmpy : A python library for fuzzy cognitive maps, https://github.com/SamvelMkhitaryan/fcmpy (2022). (Cité pages 60 et 65.)

[32] A. Amirkhani, M. Aghaei, et al., Quantum fuzzy cognitive maps : A novel approach, Applied Soft Computing 84 (2019) 105733. doi:10.1016/j.asoc.2019.105733. (Cité page 64.)

[33] A. Amirkhani, et al., Neuro-fuzzy cognitive maps : A hybrid learning framework, Neurocomputing 401 (2020) 1–15. doi:10.1016/j.neucom.2020.03.051. (Cité page 64.)

[34] Mkhitaryan, Samvel and Giabbanelli, Philippe J. and Wozniak, Maciej K. and Nápoles, Gonzalo and de Vries, Nanne K. and Crutzen, Rik, FCMpy: A python module for constructing and analyzing fuzzy cognitive maps, PeerJ Computer Science 8 (2022) e1078. doi:10.7717/peerj-cs.1078.
URL https://peerj.com/articles/cs-1078/ (Cité page 65.)

[35] G. Nápoles, M. Leon, I. Grau, K. Vanhoof, Fcm expert: Software tool for scenario analysis and pattern classification based on fuzzy cognitive maps, International Journal on Artificial Intelligence Tools 27 (7) (2018) 1860010. doi:10.1142/S0218213018600102.
URL https://www.worldscientific.com/doi/10.1142/S0218213018600102 (Cité page 65.)

[36] V. Shrivastava, S. Shukla, Fcm-vss: An ai powered secured fuzzy cognitive maps management toolkit for visualization, simulation and summarization, SoftwareX 29 (2025) 102058, open-source implementation : https://github.com/vartul-shrivastava/FCM-VSS-10122024.
URL https://www.sciencedirect.com/science/article/pii/S2352711025000251 (Cité page 65.)

[37] Fuzzy sets and systems: An international journal in information science and engineering, https://www.journals.elsevier.com/fuzzy-sets-and-systems, iSSN : 0165-0114. DOI prefix : 10.1016/j.fss. Founded in 1978 by Hansürg Zimmermann. Founder editor-in-chief : Lotfi A. Zadeh. Current Impact Factor : 2.70. h-index : 191 (2025).
URL https://www.sciencedirect.com/journal/fuzzy-sets-and-systems (Cité page 65.)

[38] J. L. Salmeron, Modelling grey uncertainty with fuzzy grey cognitive maps, Expert Systems with Applications 37 (12) (2010) 7581–7588. doi:10.1016/j.eswa.2010.04.085.
URL https://www.sciencedirect.com/science/article/pii/S0957417410003854 (Non cité.)

[39] E. I. Papageorgiou, J. L. Salmeron, Fuzzy Cognitive Maps for Applied Sciences and Engineering, Springer, 2014. doi:10.1007/978-3-642-39739-4. (Non cité.)

[40] P. Georgopoulos, C. Stylios, Fuzzy cognitive maps in medical decision making, Studies in Health Technology and Informatics 210 (2015) 567–571.
URL https://pubmed.ncbi.nlm.nih.gov/25570329/ (Non cité.)

[41] G. Nápoles, et al., Fcm learning : A review and new directions, IEEE Transactions on Fuzzy Systems 26 (5) (2018) 2734–2750. doi:10.1109/TFUZZ.2018.2798662. (Non cité.)

[42] B. Kosko, Hidden patterns in combined and adaptive knowledge networks, International Journal of Approximate Reasoning 2 (4) (1988) 377–393. doi:10.1016/0888-613X(88)90025-1. (Non cité.)

[43] E. I. Papageorgiou, A new method for fuzzy cognitive map learning, Fuzzy Sets and Systems 166 (1) (2011) 81–99. doi:10.1016/j.fss.2010.12.015. (Non cité.)

[44] C. D. Stylios, P. P. Groumpos, Modeling complex systems using fuzzy cognitive maps, IEEE Transactions on Systems, Man, and Cybernetics, Part A 34 (1) (2004) 155–162. doi:10.1109/TSMCA.2003.818878. (Non cité.)

[45] B. Kosko, Fuzzy Thinking: The New Science of Fuzzy Logic, Flamingo, London, 1994, flamingo (HarperCollins) UK paperback edition.
URL https://www.scribd.com/document/770109834/Bart-Kosko-Fuzzy-Thinking-the-New-Science-of-Fuzzy-Logic-HarperPerennial-1994-Z-Lib-io (Non cité.)

[46] T. Koutsellis, G. Xexakis, A. Karamaneas, A. Nikas, H. Doukas, In-cognitive: A web-based python application for fuzzy cognitive map modeling, simulation, and uncertainty analysis based on the monte carlo method, SoftwareX 23 (2023) 101513. doi:10.1016/j.softx.2023.101513.
URL https://www.sciencedirect.com/science/article/pii/S2352711023001899 (Non cité.)

[47] T. Koutsellis, G. Xexakis, A. Karamaneas, A. Nikas, H. Doukas, In-cognitive: Web-based fcm modeling tool (software repository), https://github.com/ThemisKout/In-Cognitive, supplementary material

for [15]. Web application : https://in-cognitive.herokuapp.com/ (2022).
URL https://github.com/ThemisKout/In-Cognitive (Non cité.)

[48] G. A. Papakostas, Y. S. Boutalis, D. E. Koulouriotis, B. G. Mertzios, Fuzzy cognitive maps for pattern recognition applications, International Journal of Pattern Recognition and Artificial Intelligence 22 (08) (2008) 1461–1486. doi:10.1142/S0218001408006910. (Non cité.)

[49] W. Froelich, A. Wakulicz-Deja, Mining temporal medical data using adaptive fuzzy cognitive maps, in : Proceedings of the 2009 2nd Conference on Human System Interactions (HSI), IEEE, 2009, pp. 152–159, fCM-based temporal medical data mining ; DOI sometimes mis-cited as 10.1109/HSI.2009.5090992, which belongs to a different HSI'09 paper.
URL https://ieeexplore.ieee.org/document/5090946 (Non cité.)

[50] W. Froelich, W. Pedrycz, Fuzzy cognitive maps in the modeling of granular time series, Knowledge-Based Systems 115 (2017) 110–122. doi:10.1016/j.knosys.2016.10.018.
URL https://www.sciencedirect.com/science/article/pii/S0950705116303987 (Non cité.)

[51] Wikipedia contributors, Fuzzy cognitive map, https://en.wikipedia.org/wiki/Fuzzy_cognitive_map, wikipedia, The Free Encyclopedia. [Online ; accessed 27-December-2025] (2025). (Non cité.)