Cartes de Kohonen:
Apprentissage auto-supervisé

Ce TP va vous permettre de découvrir la notion d'apprentissage *auto-supervisé (ou « non-supervisé »).* Nous allons en effet étudier ici comment un système plastique peut s'auto-modifier pour s'adapter aux données présentées en entrée sans intervention externe (donc sans supervision).

Un tel type de système sera ici illustré par un réseau de neurones classificateurs dits de **Kohonen.** Ce type de réseau apprend à minimiser la distance entre ses poids et les entrées, sans qu'aucune consigne sur la solution a priori ne soit donnée au système. Ce réseau permet ainsi de séparer des données d'entrées en catégories (ou classes) de façon autonome.

Dans ce TP, nous illustrerons ce cas d'apprentissage **non supervisé** par la programmation en langage C de ce réseau de Kohonen. Son test sur des exemples simples, puis son application sur l'exemple historique du voyageur de commerce et enfin sur son usage en compression d'image.

La carte de Kohonen:

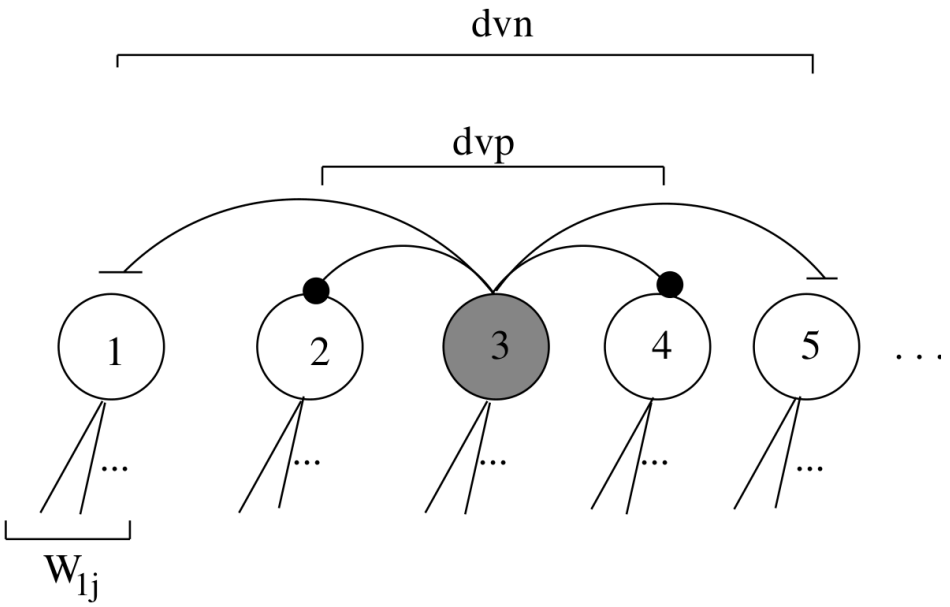

Illustration 1: Représentation d'un réseau de neurone de Kohonen en une dimension (1D). Une notion de topologie est garantie par les inter-connexions excitatrices ou inhibitrices latérales. ***dvp*** *est la distance au voisinage positif, c'est a dire la distance max des connections excitatrices.* ***dvn*** *est la distance au voisinage négatif, c'est a dire la distance max des connections inhibitrices. Les Entrées du réseau sont apprises par les connections montantes* ***Wij****.*

Algorithme :

Soit *E* l'ensemble des vecteurs d'entrée [e_1 , e_2 , … , e_n] de dimension *n*. Soit ε une constante d'apprentissage.

- Choisir aléatoirement un vecteur d'entrée ej dans E.

- Présenter *ej* à chaque neurone *i* de la carte dont on calcule le potentiel (distance des poids aux entrées).

$$Pot^i = ||e_j, W_{ij}|| \quad (1)$$

Notez qu'il existe, dans le cas général, plusieurs façons de calculer ce potentiel, on peut faire la somme des distances de type 1 : abs (e_i-W_i) , pour ensuite les normaliser avec le nombre de liens qui ont été sommés. Ou on peut utiliser d'autres types de calcul de distance (gaussienne, euclidienne). Dans le premier exercice nous utiliserons simplement la distance euclidienne car c'est la plus évidente pour des entrées de dimensions 2.

- Calcul de l'activité des neurones : Le potentiel Pot^i étant basé sur une accumulation de distance, qui généralement peuvent varier de 0 à +∞ alors ce potentiel varie entre 0 et +∞. Pour nos neurones, nous préférerons observer des sorties normalisées entre 0 et 1. De plus, il est plus logique que le neurone gagnant (dont l'activité est à 1) soit celui qui est le plus proche des entrées (dont le potentiel est donc à 0). Ainsi il faut appliquer une formule de transformation adéquate au potentiel pour obtenir l'activité du neurone (Act^i) :

$$Act^i = \frac{1}{1+Pot^i} \quad (2)$$

Notons que dans le cas général n'importe quelle formule ayant le même comportement sera acceptable (par exemple toutes formules du type exp(-Pot/a) car exp(0) = 1 et exp(-∞) = 0).

- Recherche du neurone gagnant : neurone dont l'activité est maximum sur toute la carte.

$$Gagnant = argmax\left(Act^i\right) \quad (3)$$

- Mise à jour des poids du neurone gagnant ainsi que des neurones voisins :

$$W_{ij} = W_{ij} + \left(\varepsilon \cdot |ej - Wij| \cdot \varphi\left(Gagnant, i\right)\right) \quad (4)$$

Avec $\varphi(Gagnant,i)$ un coefficient associé au voisinage du neurone avec le gagnant et représentant les liens latéraux inhibiteur et excitateur. Par exemple, en une dimension, si le neurone gagnant est le « 5 » et le neurone actuel son voisin direct le « 4 » et qu'a une distance de 1 on choisit qu'il est bon d'activer les neurones voisins. Alors on pourrait définir $\varphi(5,4)=1$

En pratique on définira φ comme une fonction qui prends une distance entre neurone en entrée (ou « distance topologique » a ne pas confondre avec la « distance » utilisée pour obtenir « Pot ») et qui renvoi un coefficient inhibiteur ou excitateur.

Exercice 1 : Programmation d'une carte de Kohonen 1D et test sur une répartition aléatoire 2D

Dans cet exercice nous supposerons le réseau de neurones de taille fixe : 20 neurones. Nous supposerons que l'ensemble des données est composées de 20 vecteurs de deux dimensions. Chaque neurone possède par conséquence deux poids en entrée.
Nous utiliserons une fonction *φ(dist)* définit tel que dvp=1 et dvn=2, soit φ(0)=1 φ(1)=λ et φ(2)=-β où λ et β sont des coefficients positifs plus petits que 1 à déterminer.

- Prendre connaissance de l'environnement de programmation :

Dans ce TP nous utiliserons des fonctions d'affichage écrites en C et OpenGL, qui communiquent directement avec la carte graphique. OpenGL appelle les boucles d'exécution du programme et gère ses fonctions d'affichages. Au départ vous n'aurez qu'à modifier la boucle d'exécution hors-affichage, vous pouvez voir celle-ci comme une boucle « while » infinie appelée par OpenGl lorsqu'il ne gère pas l'affichage. Cette fonction sera donc votre fonction de travail, il s'agit de la fonction « idle » défini dans main.c. Par la suite il vous sera demandé de modifier les fonctions d'affichage OpenGL, ce qui devra se faire à la fin de la fonction affichage(), après le #endif (un bout de code est commenté pour vous donner un exemple, vous pouvez vous inspirer du reste du code également).

Commencez donc par vous familiariser avec le fichier de travail base_opengl.c qui contient ces fonctions main(), idle() et affichage().

- Gestion des données :
  - Identifier la structure de données nécessaire à la représentation d'un vecteur d'entrée, que l'on appellera Data. Créer un tableau de 20 Datas que l'on appellera DataSet.
  - Écrire la procédure InitialiseSet() qui initialise les vecteurs d'entrée par des valeurs appartenant à [0,200].
  - Écrire la fonction Data SortData() qui retourne un vecteur d'entrée tiré aléatoirement dans DataSet (attention, on veillera à ce que qu'une donnée déjà sélectionnée ne puisse être de nouveau tirée).
- Apprentissage :
  - Identifier la structure de données nécessaire à la représentation d'un neurone *(Wij* y compris), on l'appellera Neuron, créer un tableau de 20 neurones que l'on appellera NeuronSet.
  - Écrire la fonction qui calcule le potentiel de tous les neurones de la carte en fonction d'un vecteur d'entrée donné (eq.l).
  - Écrire la fonction qui calcule l'activité de tous les neurones de la carte (eq.2).
  - Écrire la fonction qui retourne l'indice du neurone gagnant (eq.3).
  - Écrire la fonction de mise à jour des poids (la distance des neurones est calculée grâce à leur indice dans NeuronSet) (eq.4).
  - Tester l'apprentissage du réseau de neurones.
  - Étendre l'IHM du programme grâce aux librairies graphiques fournies en modifiant la fonction « affichage » pour afficher la position des neurones dans l'espaces des entrées

ainsi que la position des données. Notez vos observations et commentez le résultat (des captures d'écran et un peu d'analyse sont recommandés)

- Facultatif : considérer une carte torique, c'est-à-dire rebouclant sur elle-même.

- Questions :
  - Après observation du résultat, quels sont selon vous les avantages et inconvénients de ce type réseau ?
  - Quel serait le comportement du réseau avec des dvp et dvn (donc une fonction φ) différente, par exemple avec un dvp et un dvn bien plus large ? (vous avez le droit de tester)
  - Avez-vous observé des phénomènes « négatifs » dans votre classification ? Si oui, décrivez les et nommez-les. Quel est l'origine de ces phénomènes. Dans quelle condition est-on certain que ce genre de phénomènes arrive ? Pourquoi ?

Exercice 2 : Application d'un réseau de Kohonen 1D au problème du voyageur de commerce.

Le problème du voyageur de commerce est un problème d'optimisation historique qui est difficile à résoudre de façon peu coûteuse et optimale.

Le problème est défini ainsi :
Soit un voyageur de commerce désirant visiter plusieurs villes, quel est le chemin le plus court que doit parcourir ce voyageur pour visiter toutes les villes ?

Illustration 2: La carte de France où le voyageur de commerce doit effectuer son trajet

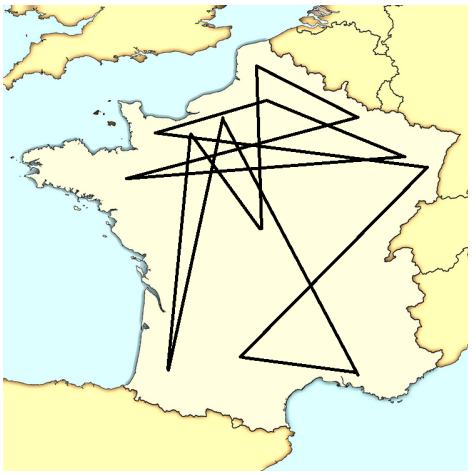

Illustration 3: Un trajet pas du tout optimal : on peut faire mieux !

Le réseau de Kohonen a historiquement donné la preuve qu'il était capable d'aboutir à d'assez bons résultats à ce problème pour un coût faible.

Exercice : Tenter de résoudre le problème du voyageur de commerce grâce au réseau de Kohonen de l'exercice 1 sur un problème d'itinéraire entre certaines des principales villes de France.

- Questions préliminaires :
  - Combien de neurones sont nécessaires ici pour résoudre le problème : pourquoi ?
  - Quel est la partie du réseau qui sera le chemin à suivre pour notre voyageur de commerce ?
- Gestion des données :
  - Modifier le #define Mode 0 au début de votre code en #define mode 1 pour activer la prise en charge de cet exercice dans le code. En faisant ceci, vous activez le chargement du fichier « Villes_et_positions_dans_image.txt » qui contient les coordonnées des villes, ainsi que le chargement d'un fichier ppm qui représentera le fonds de la carte de France « carte_france_vierge.ppm ». Familiarisez-vous avec la nouvelle logique du code.
  - Modifier votre boucle d'apprentissage de Kohonen pour être capable de prendre en entrée des données de types « Points » (le type des villes) et non plus de type « Data »
  - Charger l'image de la carte de France en passant le nom du ppm de la carte en argument de votre programme (voir le début du main pour les curieux)
  - Procédez ensuite comme pour l'exercice 1 en visualisant les données (donc les villes) et vos prototypes et observer le déploiement de votre réseau durant l'apprentissage.
- Questions :
  - Quel est l'influence de l'initialisation des poids sur le résultat ? Tenter d'initialiser vos poids de Kohonen aléatoirement au départ. Quel impact cela a-t-il sur le résultat ? Proposez une solution optimale. Testez-la et illustrez-la.

Exercice 3 : Application d'un réseau de Kohonen au problème de la compression d'image

On souhaite compresser une image en couleur, pour ce faire nous allons diminuer le nombre de couleurs présentes dans l'image pour n'en garder qu'un nombre déterminé. Pour cet exercice nous commencerons par compresser l'image avec 16 couleurs.

La matrice d'entrée est une image de dimension L1 x L2 x 3.
Les pixels sont représentés dans l'espace de dimension 3 qui représente le R, le G et le B.

Pour cet exercice nous n'utiliserons pas la boucle d'affichage OpenGL, vous pouvez vous faire un nouveau main() dans un nouveau fichier ou utiliser la fonction idle vue precedemment sans prendre en compte l'affichage.

Illustration 4: Le perroquet qu'il faut compresser

- Questions préliminaires :
  - Comment utiliseriez-vous Kohonen pour résoudre ce problème ? Définir vos entrées et la façon d'exploiter vos sorties pour compresser l'image

- Si nous codons nos 16 couleurs de la façon la plus optimale, quel est alors le facteur que nous gagnons par rapport à la même image en noir et blanc ? Et par rapport à l'image d'origine ?

- Gestions des données :
  - Lire l'image « perroquet.ppm » grâce à la fonction « `readPPM` » définie dans ppm.c, qui renvoie un pointeur vers un tableau de pixels représentant l'image (jetez un œil à la structure `Image`)
  - Définir un nouveau tableau « image_compressee » pour votre résultat
  - Vous pourrez créer une image à partir de votre tableau résultat grâce à la fonction `writePPM` (définie dans ppm.h)
- Questions :
  - Observer votre image quantifiée résultante avec une compression de 16 couleurs, 32 couleurs et 256 couleurs et notez les résultats
  - Bonus : Observer la taille de l'image résultante, quel problème observez-vous, que faudrait-il faire pour le résoudre ?
  - Bonus 2 : Que se passe-t-il si au lieu d'apprendre sur les valeurs RGB des pixels de l'image on apprend sur les valeurs RGB + les valeurs X/Y représentant la position des pixels. Tester. Vous pouvez profiter des fonctions d'affichage OpenGL utilisez pour afficher la carte de France pour afficher le perroquet en les modifiant. Par la suite vous pouvez visualiser vos prototypes dans l'espace de l'image du perroquet en utilisant les valeurs des poids X/Y et en leur donnant une couleur en fonction de leurs poids RGB. Réussir ce bonus 2 donnera un très gros bonus sur la note du TP.