Rapport Projet

TIA

Reseaux de neurones

Groupe:

NESR Mohamad

EL HIRACH Yassine

Enseignant: **LEFORT** Mathieu

Année universitaire **2021/2022**

Table des matieres

Etude "théorique" de cas simples **3**
3.1 Influence de η 3
3.2 Influence de σ 3
3.3 Influence de la distribution d'entrée 4

Etude pratique **5**
4.2 Implementation 5
4.3 Analyse de l'algorithme 5
Taux d'apprentissage η 5
η=0.01 5
η=0.5 5
η=0.99 6
η=2 6
Deduction 6
Largeur du voisinage σ 6
σ=1.4 7
σ=10 7
Deduction 7
Nombre de pas de temps d'apprentissage N 8
N=300 8
N=3000 8
N=30000 9
Deduction 9
Taille et forme de la carte 9
Ligne(1,5) 9
Ligne(5,1) 10
Ligne(1,50) 10
Ligne(1,25) 10
Carre(5,5) 11
Carre(20,20) 11
Rectangle(5,2) 11
Rectangle(2,25) 12
Deduction 12
Jeu de données 12
Ensemble de données 2 12
Ensemble de données 3 13
Ensemble robotique 13
Ensemble non-uniformément distribuées 1 13
Ensemble non-uniformément distribuées 2 14
Deduction 14
4.4 Bras robotique 14

Etude "théorique" de cas simples

3.1 Influence de η :

— Si η est nulle, ΔW_{ij} est nulle aussi.

$$\Delta W_{ij} = \eta e^{-\frac{||j-j^*||_c^2}{2\sigma 2}} (x_i - w_{ij}) = 0 * e^{-\frac{||j-j^*||_c^2}{2\sigma 2}} (x_i - w_{ij}) = 0$$

Le valeur du poid du neurone reste donc invariante durant l'apprentissage.

— Si η = 1, on a :

$$\Delta W_{ij} = \eta e^{-\frac{||j-j^*||_c^2}{2\sigma 2}} (x_i - w_{ij})$$

Sachant que la distance entre un neurone et lui même est de 0 :

$$j - j^* = 0 \quad Donc \quad \Delta W_{ij} = 1 * e^0 (x_i - w_{ij}) = (x_i - w_{ij})$$

La valeur du poid du prochain neurone sera donc celle du neurone actuel + $(x_i - w_{ij})$

— Dans le cas ou η ∈]0, 1[on a :

$$\Delta W_{ij} = \eta(x_i - w_{ij})$$

Si η tends vers 0:
le nouveau poids vas tendre vers W*
Si η tends vers 1:
le nouveau poids vas tendre vers X

— Dans le cas ou η >1:
Un taux ne peut pas être supérieur à 1. Ca sera traité comme si η =1.

3.2 Influence de σ :

— Si σ augmente alors $-\frac{||j-j^*||_c^2}{2\sigma^2}$ va diminuer et donc tendre de plus en plus vers 0.

Sachant que $e^0 = 1$, on obtient ΔW_{ij} = η * 1 * $(x_i - w_{ij}) = \eta (x_i - w_{ij})$
Les neurones vont donc apprendre plus de l'entrée courante.

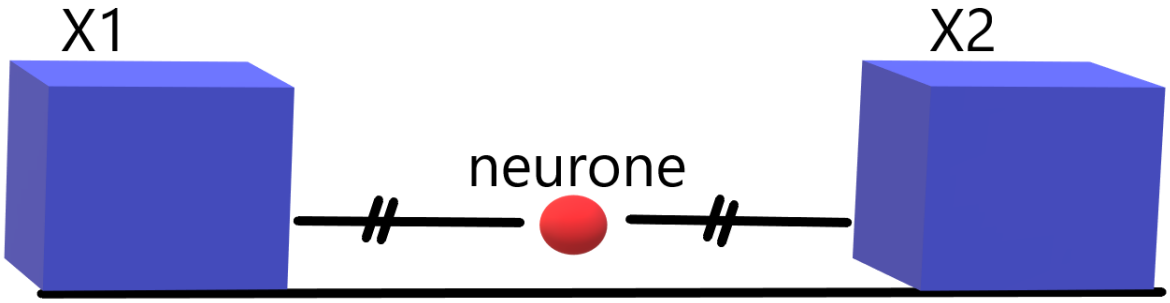
— L'auto-organisation obtenue sera plus resserrée car les poids des neurones vont se rapprocher de l'entrée ce qui donne a tout nos neurones des valeurs approchées.

— On pourra calculer la plus grande distance entre deux neurones et ensuite déduire si l'auto-organisation est plus resserrée (petite distance) ou lâche (grande distance).

3.3 Influence de la distribution d'entrée :

—

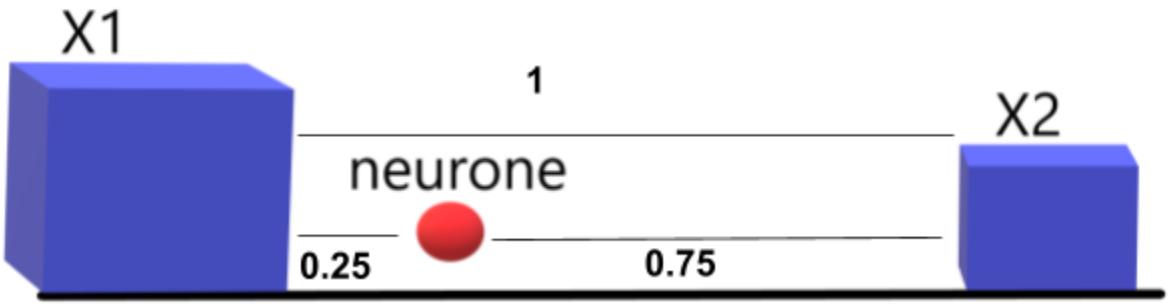
— Dans le cas où X1 et X2 sont présents autant de fois et en sachant que n est faible, le vecteur de poids du neurone convergera vers le centre d'inertie de X1 et X2 qui est dans ce cas le centre entre les deux entrées (isobarycentre).



— Si X1 est présent n fois de plus que X2, le vecteur de poids du neurone convergera forcément vers X1. On peut comparer cet exemple à celui du système solaire ou toutes nos planètes orbitent autour du soleil étant l'astre ayant la plus grande masse.

Par exemple: Supposons que X1 est présent 2 fois de plus que X2 et la distance totale entre X1 et X2 est de 1. Le neurone va donc se déplacer à $\frac{1}{2n}$ de X1.

On obtient donc une distance X1->neurone = 0.25 et X2->neurone=0.75



— Avec l'augmentation de la densité des données, nos neurones se rapprocheront de plus en plus vers le neurone gagnant. Ainsi, l'aire dans laquelle se retrouvent nos neurones sera plus petite (3.2 partie 3) et nos neurones seront donc plus "resserrés" et vont mieux apprendre.

Etude pratique

4.2 Implementation :

(voir code)

4.3 Analyse de l'algorithme :

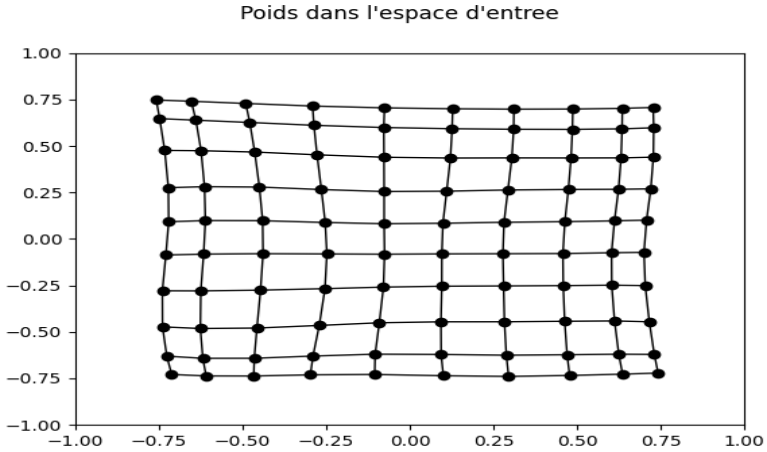
On cherche à prouver nos réponses, de la partie théorique, on va donc faire plusieurs tests pour chaque cas dans des différentes contraintes comme posées dans la partie 3.

Taux d'apprentissage η:

On va effectuer 4 tests, avec η=0.01, η=0.5, η=0.99 et η=2.

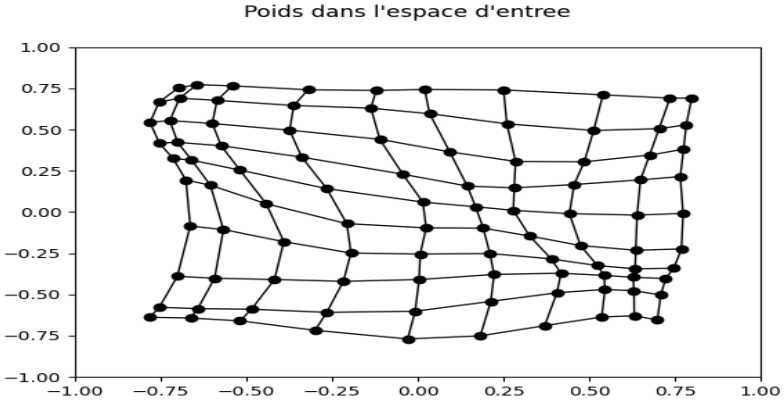
η=0.01:

erreur de quantification vectorielle moyenne 0.02699905028240309
plus grande distance entre 2 neurones 1.9953837595669928



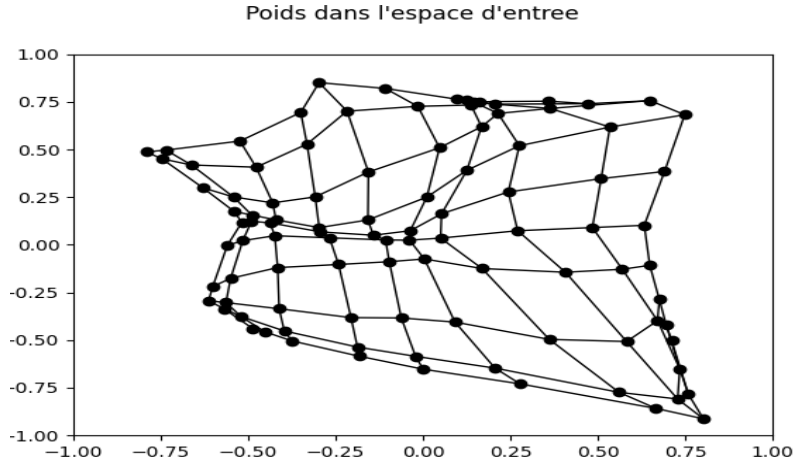
η=0.5:

erreur de quantification vectorielle moyenne 0.018161139199506213
plus grande distance entre 2 neurones 1.944785346796447



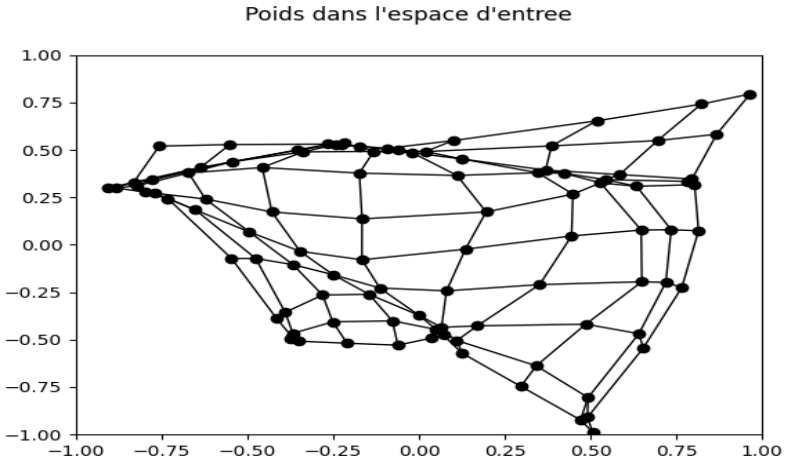
η=0.99:

erreur de quantification vectorielle moyenne 0.011553453130330519
plus grande distance entre 2 neurones 1.9294237523465945



η=2:

erreur de quantification vectorielle moyenne 0.03529676711749743
plus grande distance entre 2 neurones 2.173116982957775



Deduction:

On voit que quand η augmente, l'erreur de quantification vectorielle moyenne diminue. Cependant, la plus grande distance varie mais reste plutôt stable.

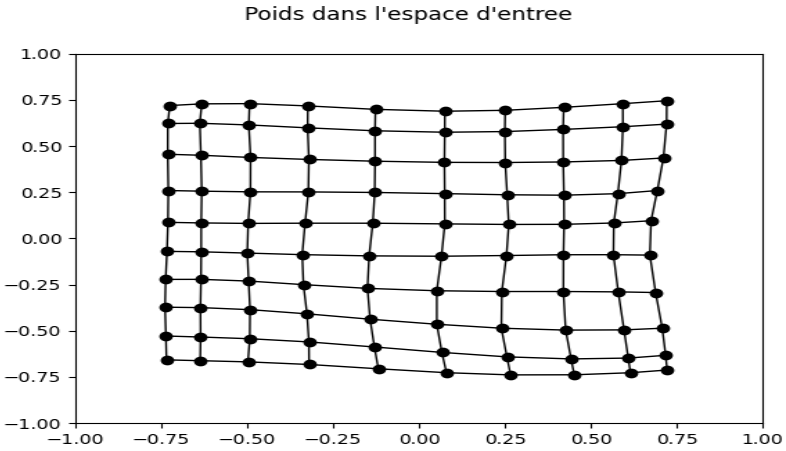
Largeur du voisinage σ:

On va effectuer 2 tests avec σ=1.4 et σ=10.

On pourra comme ça comparer notre cas par défaut à celui avec un voisinage plus grand pour confirmer nos déductions de la partie 3.2.

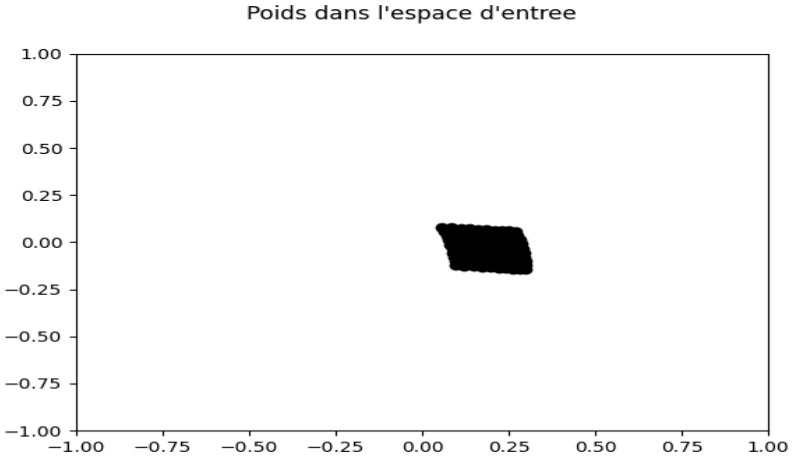
σ=1.4:

erreur de quantification vectorielle moyenne 0.026052095747022 87
plus grande distance entre 2 neurones 1.9705161848704522



σ=10:

erreur de quantification vectorielle moyenne 0.4978525159475215
plus grande distance entre 2 neurones 0.33820544165764094



Deduction:

On voit que quand σ est petite, l'erreur de quantification vectorielle moyenne est petite et la distance entre les neurones augmente. Tandis que quand σ est grande, l'erreur de quantification vectorielle moyenne est grande, la distance entre les neurones diminue. Et tout nos neurones bougent uniformément. C'est parce qu'on a plus de voisins qui apprennent en même temps ce qui entraîne leurs déplacements. Ceci confirme donc nos réponses dans la partie 3.2 ou on précise qu'avec l'augmentation de la largeur du voisinage, nos neurones seront plus resserrés. Également, la plus grande distance entre nos neurones diminue significativement.

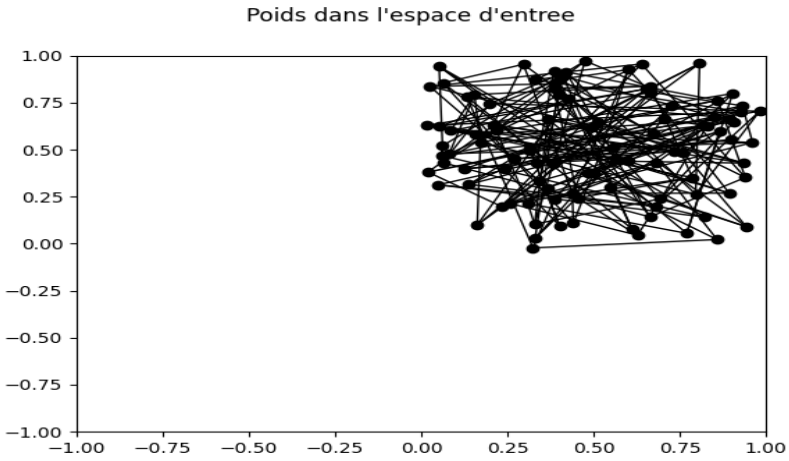
Nombre de pas de temps d'apprentissage N:

Hypothèse: Avec l'augmentation de N, on peut s'attendre à une hausse du temps d'exécution mais également une chute du taux d'erreur vu qu'on fait plus de pas (on avance doucement).

On va effectuer 3 tests, N = 300, N=3000, N=30000

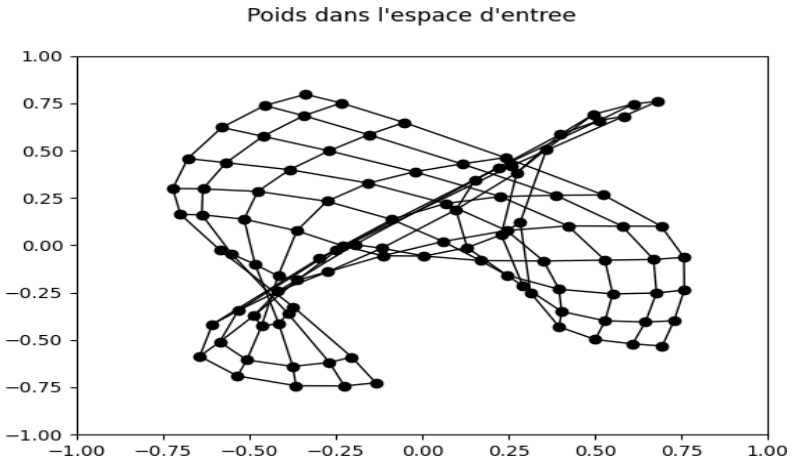
N=300:

erreur de quantification vectorielle moyenne 0.06563978054708124
plus grande distance entre 2 neurones 1.7981435140864646



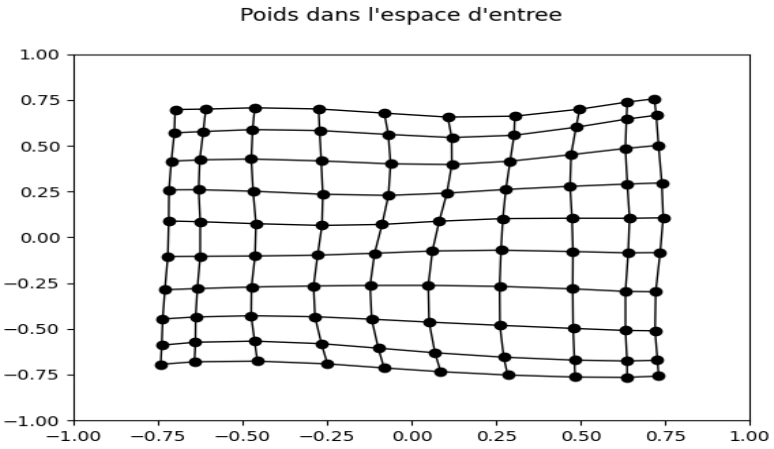
N=3000:

erreur de quantification vectorielle moyenne 0.021109587084699243
plus grande distance entre 2 neurones 1.8876589781525883



N=30000:

erreur de quantification vectorielle moyenne 0.01806688836540123
plus grande distance entre 2 neurones 2.086125716120889



Deduction:

Comme on a pu observer, quand le nombre de pas augmente, l'erreur de quantification vectorielle diminue mais le temps d'exécution augmente (chargement des graphes). Ceci confirme donc notre hypothèse posée en début de partie. Quand N est plus petit, la plus grande distance est plus petite. Ceci est dû au fait que notre grille n'a pas eu le temps de se former.

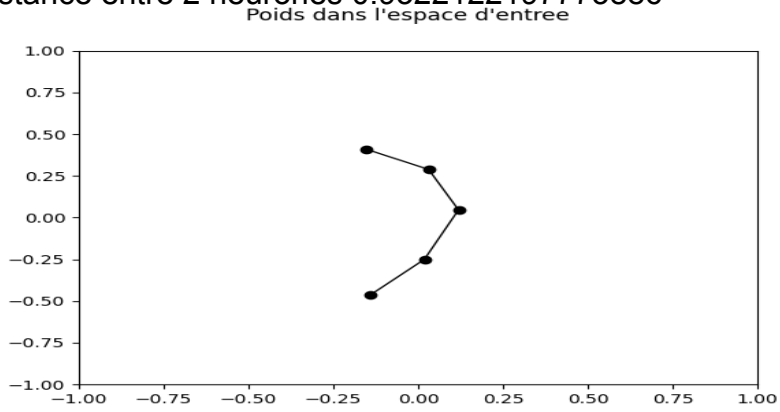
Taille et forme de la carte:

On n'a pas étudié ce cas auparavant, afin d'en tirer une conclusion, on gardera des valeurs similaires pour les dimensions dans de différents cas afin de faire des comparaisons. Logiquement, plus on rajoute des points, moins le taux d'erreur de quantification vectorielle est grand. Nous allons également vérifier si l'inversement des formes affecte nos résultats.

On va faire 9 tests: Ligne(1,5), Ligne(5,1) ,Ligne(1,50), ,Ligne(1,25) Carré(5,5), Carré(20,20), Rectangle(5, 2), Rectangle(25,2), Rectangle(2,25).

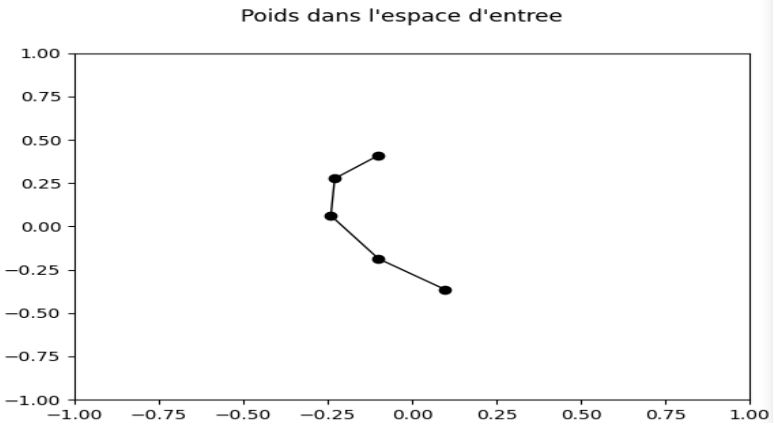
Ligne(1,5):

erreur de quantification vectorielle moyenne 0.35492737225291243
plus grande distance entre 2 neurones 0.9322122197773886



Ligne(5,1):

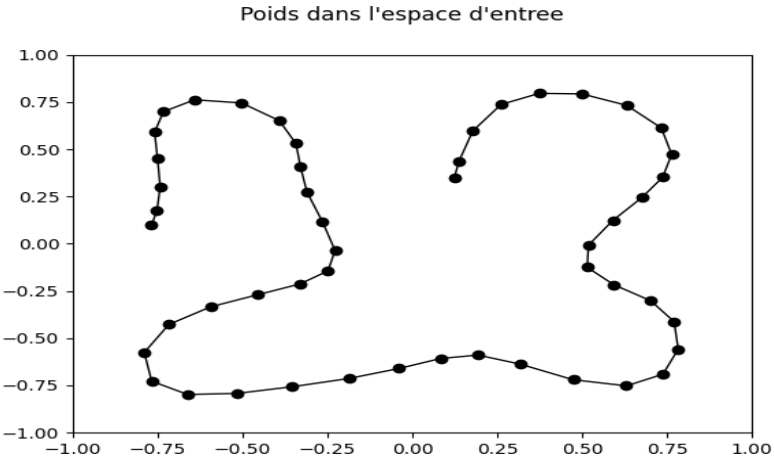
erreur de quantification vectorielle moyenne 0.3754112378008646
plus grande distance entre 2 neurones 1.0469907223947643



On remarque que l'inversement des dimensions n'a aucun effet, on peut s'en passer alors du test Rectangle(25,2).

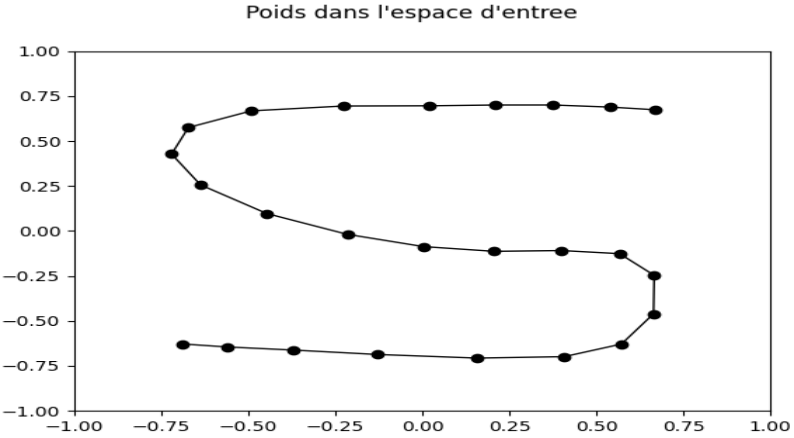
Ligne(1,50):

erreur de quantification vectorielle moyenne 0.02774220714760907
plus grande distance entre 2 neurones 2.1015859627974236



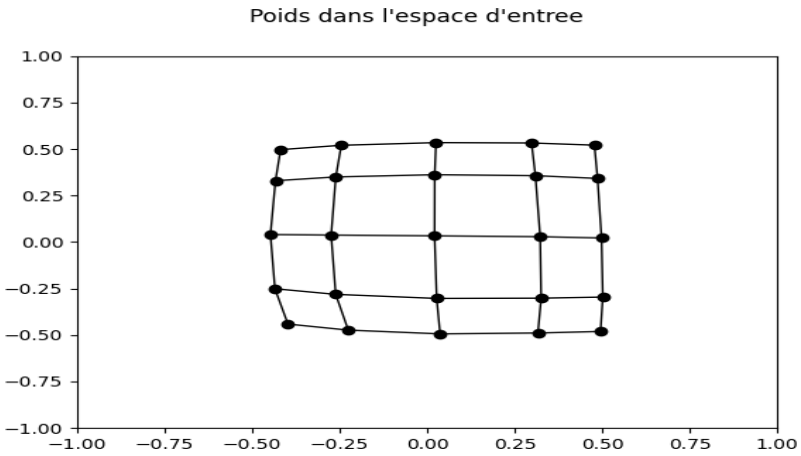
Ligne(1,25):

erreur de quantification vectorielle moyenne 0.05758802096720745
plus grande distance entre 2 neurones 1.8742136011715909



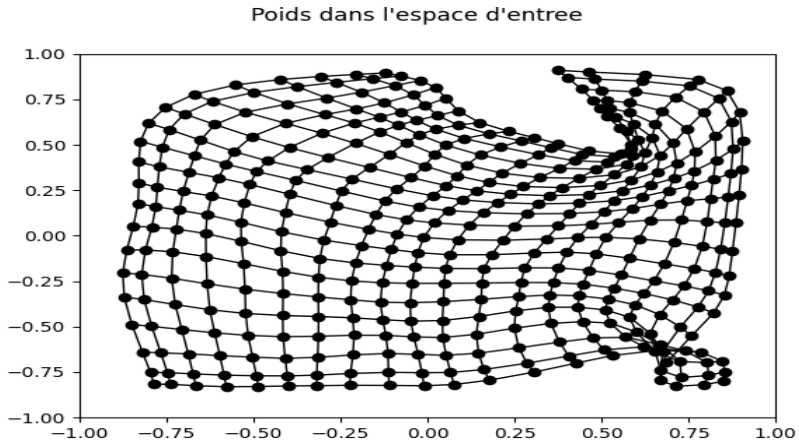
Carre(5,5):

erreur de quantification vectorielle moyenne 0.1018935562730385
plus grande distance entre 2 neurones 1.4013252667587357



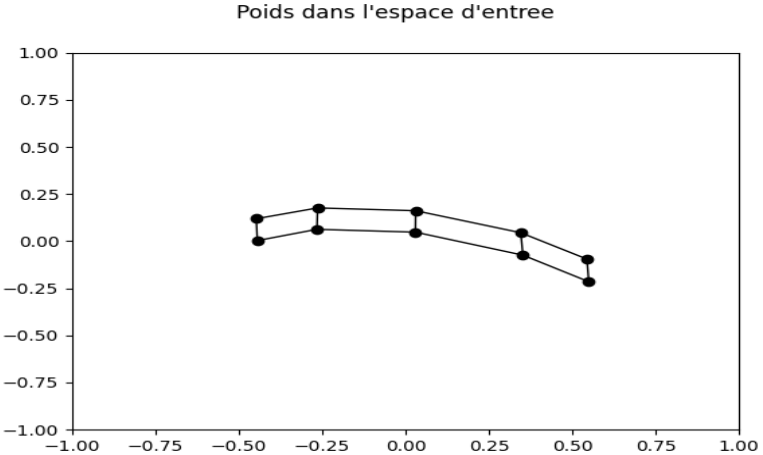
Carre(20,20):

erreur de quantification vectorielle moyenne 0.004660494989681395
plus grande distance entre 2 neurones 2.311949467592767



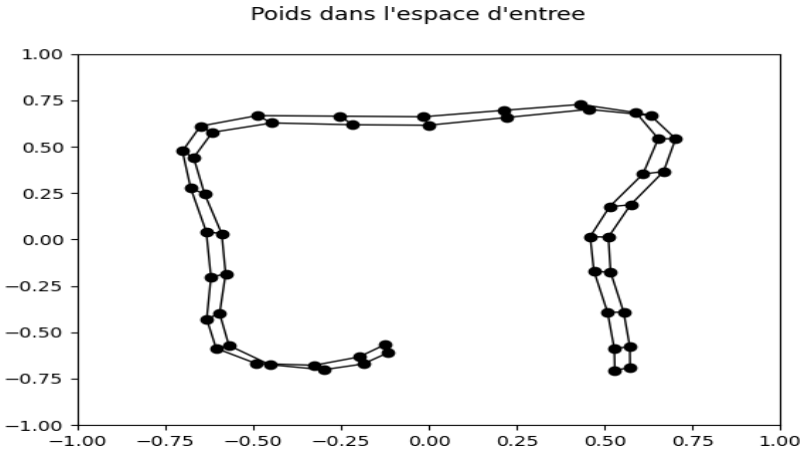
Rectangle(5,2):

erreur de quantification vectorielle moyenne 0.3172984468740671
plus grande distance entre 2 neurones 1.0137729225619831



Rectangle(2,25):

erreur de quantification vectorielle moyenne 0.052797920929379356
plus grande distance entre 2 neurones 1.8274729131472878



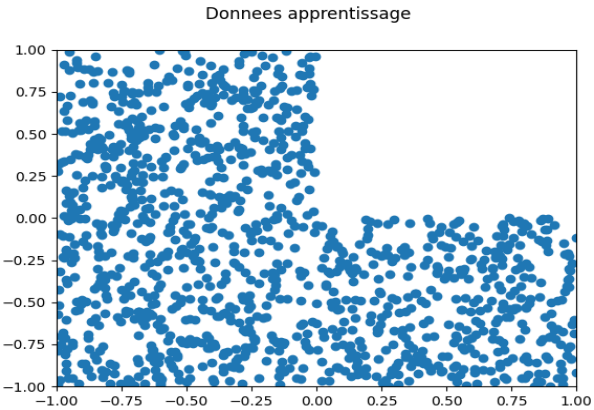
Deduction:

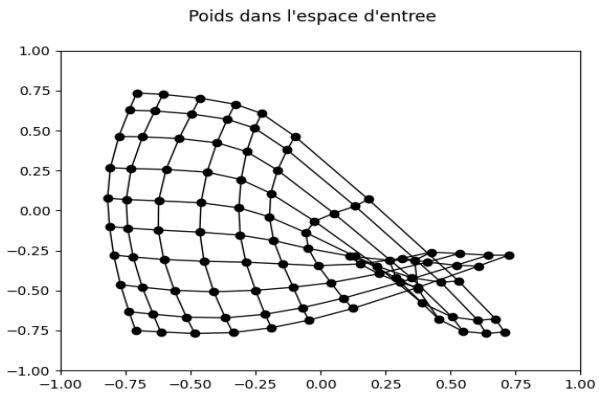
D'après les résultats ci-dessus, on peut déduire que certaines formes sont plus avantageuses que d'autres. C'est-à- dire, une forme qui permet de poser plus de points est plus intéressante d'une qui permet d'en poser moins. On remarque également qu'une ligne avec autant de points qu'un rectangle ou un carré a un taux d'erreur de quantification vectorielle moyenne moins important. L'objectif pour un meilleur résultat est donc d'aligner le plus grand nombre de points possible.

Jeu de données:

Ensemble de données 2:

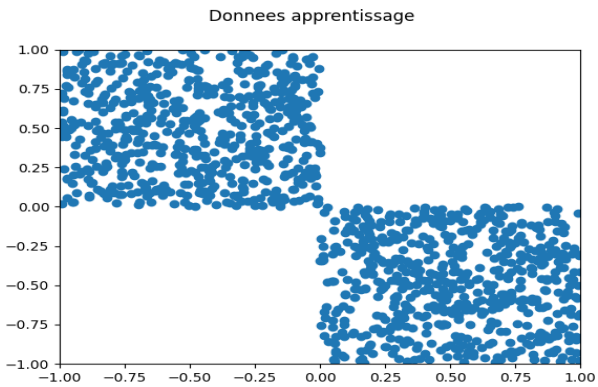
erreur de quantification vectorielle moyenne 0.0192765617885062
plus grande distance entre 2 neurones 2.0573531010216515

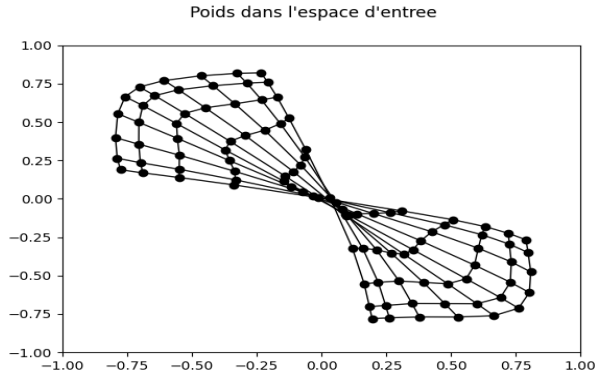




Ensemble de données 3:

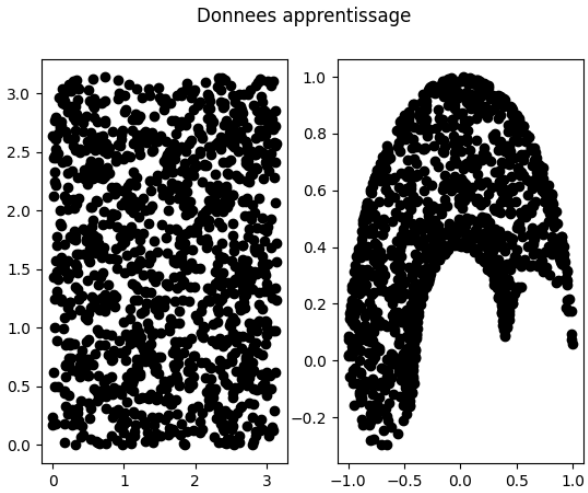
erreur de quantification vectorielle moyenne 0.012330753176460438
plus grande distance entre 2 neurones 2.0543047533264724

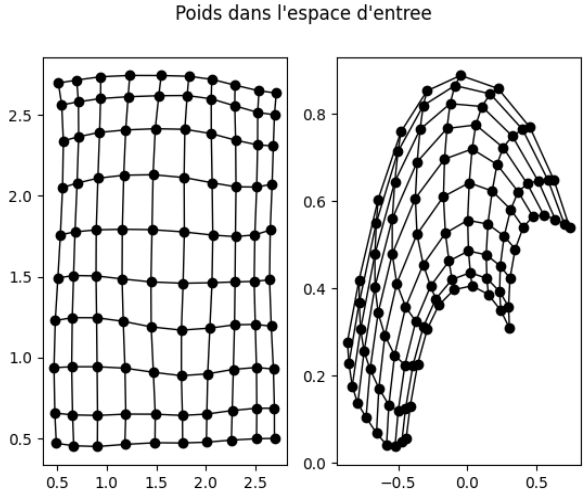




Ensemble robotique:

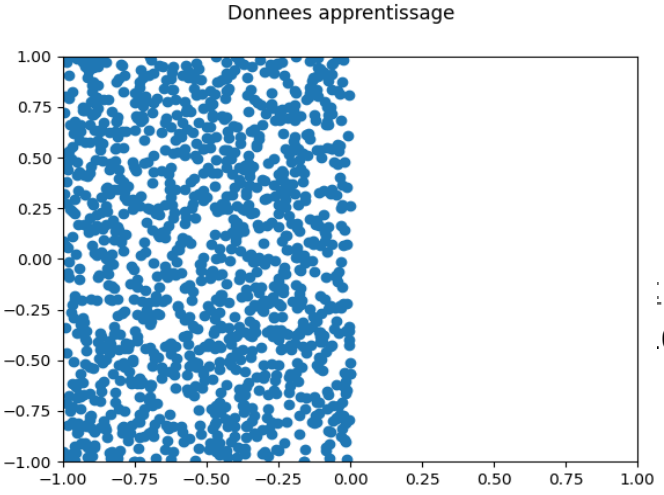
erreur de quantification vectorielle moyenne 0.07215218811820033
plus grande distance entre 2 neurones 3.347305702418091

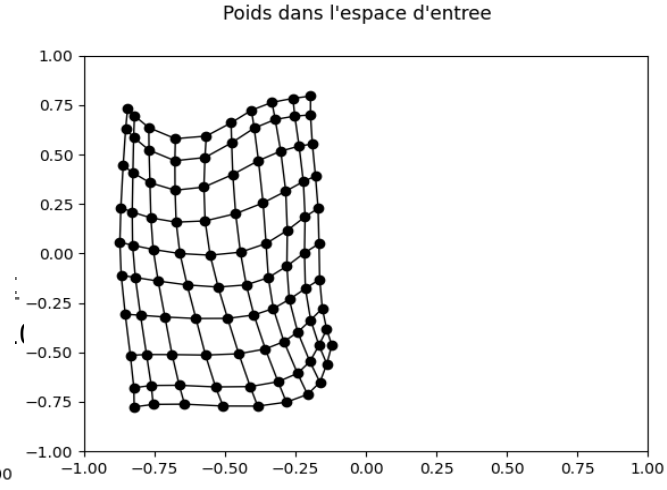




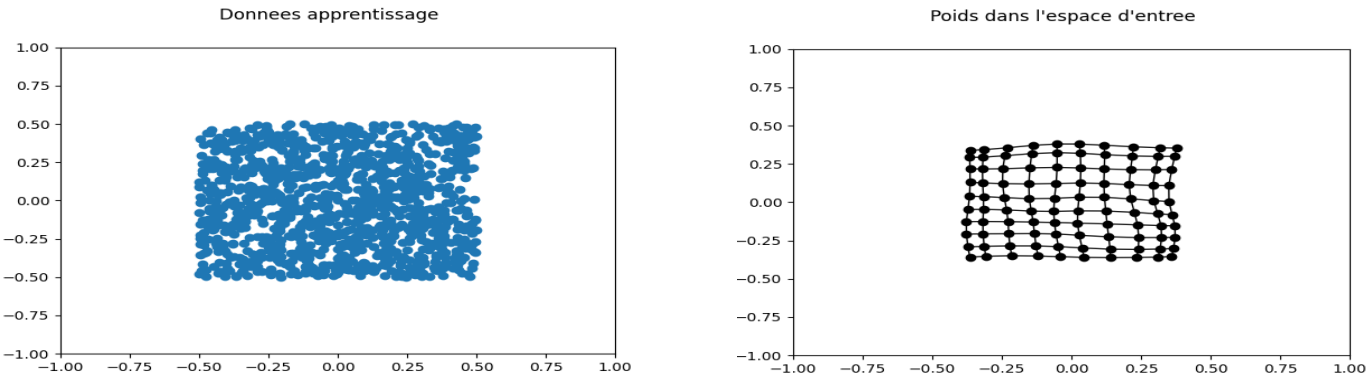
Ensemble non-uniformément distribuées 1:

erreur de quantification vectorielle moyenne 0.011218782895079904
plus grande distance entre 2 neurones 1.6906760212066199





plus grande distance entre 2 neurones 1.0283860732629244



Deduction:

On remarque que quand on met en place des contraintes et des limites à la répartition de l'ensemble de données, notre taux d'erreur de quantification vectorielle moyenne baisse parce qu'on limite les endroits où nos neurones sont repartis, cela va donc les obliger de se rapprocher les uns des autres et donc se resserrer. En observant notre ensemble non-uniformément distribué 2 on peut voir que quand nos neurones sont regroupés au centre et non pas sur tout le graphe, notre taux baisse significativement d'environ 50% parce qu'on a enlevé 0.5 de chaque côté. (on passe de 0.01 dans les autres cas jusqu'à 0.005 environ dans celui la). On remarque également que le bras robotique à un taux d'erreur élevé comparé aux autres données. On se demande donc si le jeu de données du bras robotique est assez crédible.

4.4 Bras robotique:

— Position motrice donnée:

On sait que chaque neurone a 4 poids. On pourra chercher le quadruplet ayant les valeurs de $\theta 1$ et $\theta 2$ pour ses 2 premiers poids. Ensuite, on pourra en déduire x1 et x2 en récupérant les 2 derniers poids qui correspondent à la position spatiale du bras.

```
def find_arm_pos_motrice(self, x1, x2):
    '''
    @summary: Renvoie theta1, theta2 quand on lui passe x1,x2.
    @param x1 : position en x
    @type x1 : int
    @param x2 : position en y
    @type x2 : int
    '''
    arm_theta= []
    for posx in range(self.gridsize[0]):
        for posy in range(self.gridsize[1]):
            if ((self.weightsmap[posx][posx][2] - x1 == 0)
            and (self.weightsmap[posx][posx][3] - x2 == 0)):
                arm_theta = [self.weightsmap[posx][posx][0], self.weightsmap[posx][posx][1]]
    return arm_theta
```

On sait que chaque neurone a 4 poids. On pourra chercher le quadruplet ayant les valeurs de x1 et x2 pour ses 2 derniers poids. Ensuite, on pourra en déduire $\theta 1$ et $\theta 2$ en récupérant les 2 premiers poids qui correspondent à une position motrice.

```
def find_arm_pos_spatial(self, theta1, theta2):
    '''
    @summary: Renvoie x1 et x2 quand on lui passe theta1, theta2 .
    @param theta1 : position en x
    @type theta1 : int
    @param theta2 : position en y
    @type theta2 : int
    '''
    arm_x = []
    for posx in range(self.gridsize[0]):
        for posy in range(self.gridsize[1]):
            if (numpy.abs(self.weightsmap[posx][posx][0] - theta1 == 0)
                    and (self.weightsmap[posx][posx][1] - theta2 == 0)):
                arm_x = [self.weightsmap[posx][posx][2], self.weightsmap[posx][posx][3]]
    return arm_x
```

— Cette méthode se rapproche du modèle de gaz neuronal qui est lui-même inspiré des cartes auto-adaptatives. L'avantage présenté est le fait que la quantification vectorielle ne prend pas en compte la forme de la carte comme pour kohonen. Vu que le positionnement de bras n'a pas une "forme spécifique", ce modèle ne s'en souciera pas de la compatibilité de la forme de la carte avec les données ce qui est très avantageux. Mais, le modèle prendra en compte également toutes les données y compris le bruit et le nombre de neurones va augmenter si l'espace d'entrée est grand ce qui pourrait limiter la qualité des performances (plus de neurones -> plus de calculs -> plus d'opérations -> plus de temps).

— Si l'objectif était de prédire uniquement la position spatiale à partir de celle motrice, le modèle qu'on aurait pu utiliser est celui du Perceptron multicouches. L'avantage de ce modèle est le fait qu'il puisse traiter des formes complexes (un bras) et la présence de plusieurs couches permet un meilleur filtrage des données quelle que soit leurs forme dans l'espace (ce n'est pas le cas pour le modèle de kohonen). Ce modèle fonctionne également très bien avec un grand nombre de données mais peut aussi retourner des résultats aussi précis avec moins de données. Les inconvénients du modèle sont le manque de garantie de convergence et d'efficacité(ça dépend de la qualité de l'entraînement). De plus, les calculs sont complexes et chronophages.

— Si on connaît les positions $(\theta 1\ ,\ \theta 2)$ et $(\theta 1'\ ,\ \theta 2')$, nous pourrons utiliser notre fonction implémentée find_arm_pos_spatial afin de retrouver (x1,x2) et (x1',x2'). On pourra

ensuite relier notre point de départ à notre point final. Tous les points présents sur cette droite sont des positions spatiales intermédiaires. De cette manière, nous obtiendrons une liste contenant toutes les positions spatiales possiblement prises par la main.

```
def predict_arm_trajectory(self, theta1, theta2, theta1p, theta2p, nb_etats):
    '''
        @summary: Renvoie la prediction de la suite de positions pris par la main .
        @param theta1 : position depart en x
        @type theta1 : int
        @param theta2 : position depart en y
        @type theta2 : int
        @param theta1p : position finale en x
        @type theta1p : int
        @param theta2p : position finale en y
        @type theta2p : int
        @param nb_etats : nombre d'etat intermediates souhaites
        @type nb_etats : int
        '''
    pos_init = numpy.append([theta1,theta2])
    pos_final = numpy.append([theta1p,theta2p])
    # On compte nos pas
    diff_theta1 = int((theta1p - theta1)/nb_etats)
    diff_theta2 = int((theta2p - theta2)/nb_etats)
    pos_inter = numpy.append(network.find_arm_pos_spatial(pos_init[0], pos_init[1]))
    for i in range(nb_etats+1):
        if (pos_inter[i - 1][0] + diff_theta1 < pos_final[0]) and (pos_inter[i - 1][1] + diff_theta2 < pos_final[1]):
            pos_inter.append([network.find_arm_pos_spatial(pos_init[0] + i*diff_theta1,pos_init[1] + i*diff_theta2)])
    pos_inter.append(network.find_arm_pos_spatial(pos_final[0], pos_final[1]))
    return pos_inter
```