Bordeaux INP
ENSC

# PROJET IA 2017

## Camélia

*Émilie ROGER, Camille TARTARE, Alexia TARTAS*

## PLAN DU RAPPORT

# 1. Partie 1

## 1.1. Question préliminaire

### 1.1.1. L'entrepôt

Nous avons choisi de créer un tableau d'entiers pour représenter l'entrepôt « virtuellement » et ainsi pouvoir connaître à tout moment sa configuration :

- `0` indique que la case est libre
- `-1` indique qu'il y a une étagère
- `-2` indique qu'il y a un chariot

Nous avons également utilisé un tableau de `PictureBox` pour représenter l'entrepôt. Celui-ci n'est pas destiné à être utilisé pour le traitement et les calculs, mais permet d'avoir un aperçu plus « joli » de l'entrepôt.

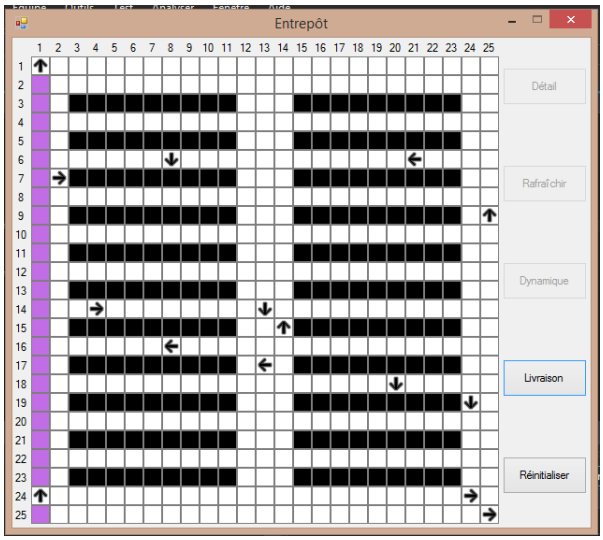



FIGURE 1 – Représentation visuelle de l'entrepôt

**Légende**

- Case blanche : case libre
- Case noire : étagère
- Case violette : zone de livraison
- Case rouge : point de départ d'un chariot
- Case verte : point d'arrivée d'un chariot
- Case orange : point d'arrêt temporaire d'un chariot (point intermédiaire entre le trajet de récupération d'un objet et le trajet de livraison d'un objet)
- Case bleue : case par laquelle le chariot va passer pour effectuer la récupération / livraison
- Flèche : chariot orienté suivant la direction indiquée par la flèche

**Interactions**

Pour faciliter l'interaction avec l'interface, l'utilisateur peut cliquer sur une case vide et blanche pour ajouter un chariot (dont l'orientation est générée aléatoirement). Il n'est donc pas nécessaire de lui demander le nombre de chariots à ajouter, puis ensuite de lui demander d'indiquer pour chaque chariot ses coordonnées. Cependant, c'est ce que nous avions fait au début, en créant un formulaire dédié. Par défaut, on place 15 chariots dans l'entrepôt. Lorsqu'on ajoute un chariot, il n'est pas possible de l'enlever.

Si l'utilisateur clique directement sur une flèche, c'est-à-dire un chariot, l'utilisateur peut choisir quel objet il doit aller chercher (en indiquant sa position, son orientation et sa hauteur) et suivant quel mode : chemin le plus court (distance), chemin le plus rapide (temps) ou en tenant compte de la trajectoire des autres chariots (réalité).

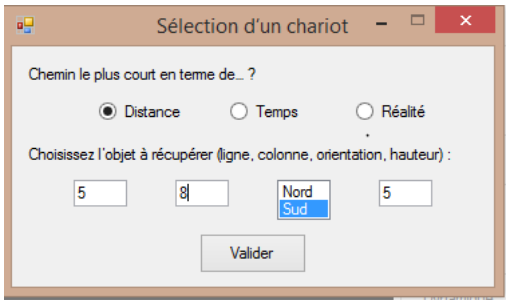



FIGURE 2 – Formulaire permettant d'indiquer l'objet à récupérer et le mode choisi

### 1.1.2. Les objets

Sur chaque étagère, se trouvent des objets que les chariots peuvent récupérer. Un objet est défini par sa position dans l'entrepôt (numéro de ligne et numéro de colonne), son orientation sur l'étagère et sa hauteur (comprise entre 0 et 10). L'orientation de l'objet permet de savoir si le chariot doit se placer au nord ou au sud de l'étagère pour pouvoir le récupérer. Ainsi, un objet peut soit être au nord de l'étagère (0), auquel cas le chariot devra se placer au nord de l'étagère, soit être au sud (1) de l'étagère auquel cas le chariot devra se placer au sud de l'étagère.

### 1.1.3. Les chariots

Un chariot est défini par sa position (numéro de ligne et numéro de colonne dans l'entrepôt) et une orientation (c'est-à-dire dans quel sens il est). Les chariots ne se déplaçant que verticalement et horizontalement, leur orientation ne peut être que nord (0), est (1), sud (2) ou ouest (3).

Un chariot met un certain temps pour changer de case :

- s'il va tout droit, il met 1 seconde
- s'il prend en virage, il met 4 secondes (3 pour se tourner et 1 pour changer de case)
- s'il fait demi-tour, il met 7 secondes (6 pour se tourner et 1 pour changer de case)

Par ailleurs, lorsqu'un chariot parvient à la position requise pour récupérer un objet, il met entre 0 et 20 secondes pour effectuer la récupération.

### 1.1.4. La zone de livraison

Lorsqu'un chariot a récupéré un objet, il doit revenir à la zone de livraison (qui est représentée en violet, il s'agit de la première colonne sur la grille entrepôt). Par ailleurs, un chariot doit normalement partir de la zone de livraison.

## 1.2. Question 1

### 1.2.1. Récupération de l'objet

Pour répondre à cette question, nous avons exploité les fichiers fournis, à savoir `Graphe.cs` et `Noeud.cs`. Nous avons également codé la classe `NoeudDistance`, héritant de `Noeud`, et qui permet de trouver les successeurs de l'instance suivant une contrainte de distance. Dans notre programme, un `Noeud` étant un chariot (position +

orientation), la classe `NoeudDistance` permet donc de trouver dans quelle position et quelle orientation pourra être le chariot au prochain déplacement. Pour cela il est nécessaire de connaître la configuration de l'entrepôt car on peut ainsi déterminer les positions possibles très facilement ; il suffit de vérifier si au nord, au sud, à l'est ou à l'ouest du chariot, il y a un `0` correspondant à un passage possible au niveau du tableau des entiers.

Par ailleurs, les chariots étant statiques (sauf celui qu'on déplace, bien sûr...), la configuration de l'entrepôt n'est pas vouée à changer après chaque déplacement. Elle est donc commune à chaque instance de la classe. L'entrepôt est donc un attribut statique de la classe.

De plus, comme on souhaite que le chariot atteigne une destination précise (en fonction de l'objet à aller récupérer) et que cet objectif ne change pas d'un déplacement à l'autre, la destination est également un attribut statique de classe.

### 1.2.2. Utilisation d'une heuristique

Pour « accélérer » la recherche du chemin le plus court, nous avons utilisé l'heuristique du calcul de la distance euclidienne entre le point de départ et le point d'arrivée, dont voici la formule : $d(AB) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$ avec $A$ le point de départ et $B$ le point d'arrivée.

### 1.2.3. Illustrations

**Chemins différents sans et avec l'heuristique :**

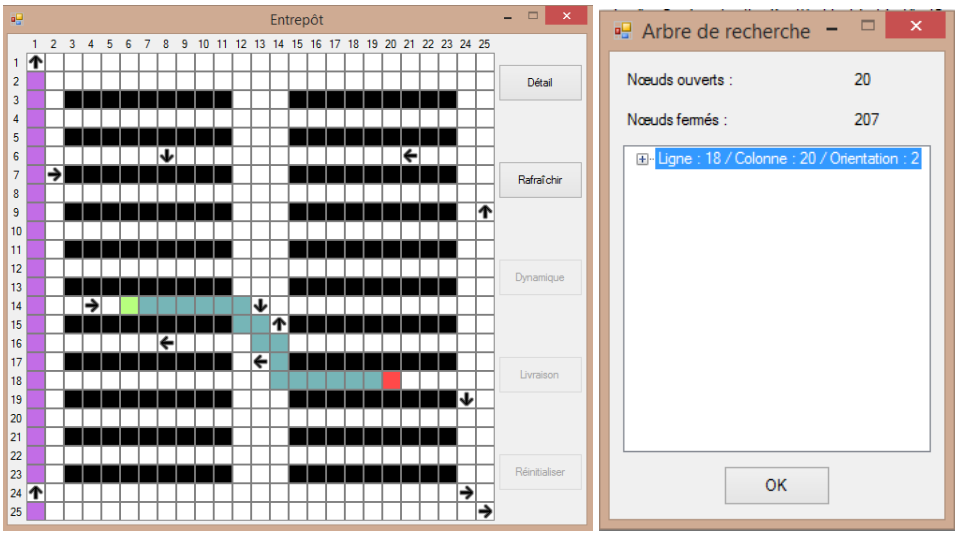



FIGURE 3 – Tracé du chemin et arbre de recherche sans l'heuristique

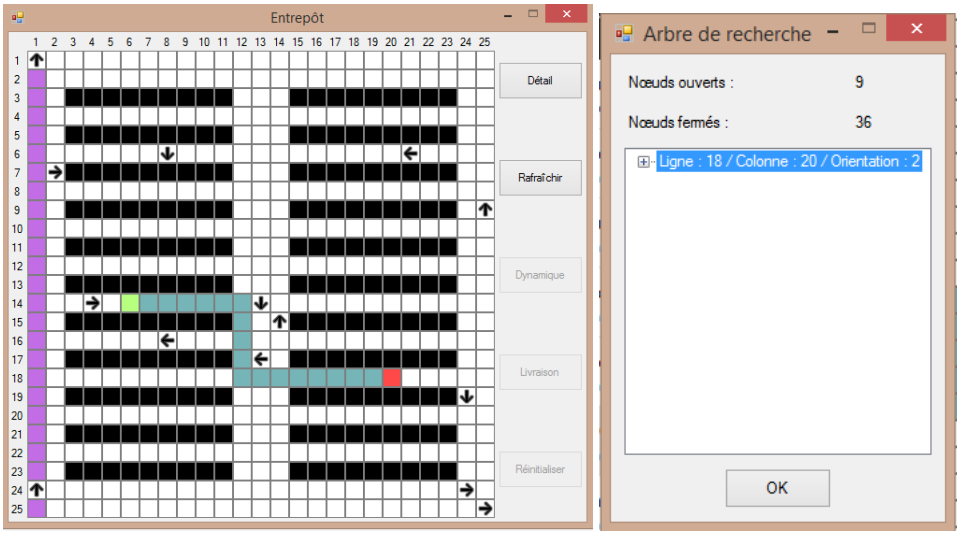



FIGURE 4 – Tracé du chemin et arbre de recherche avec l'heuristique

**Chemins identiques sans et avec l'heuristique :**

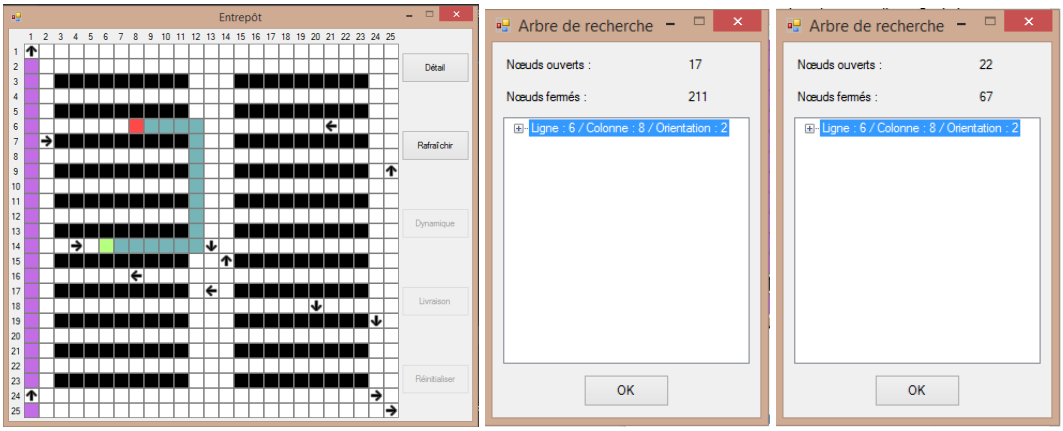



FIGURE 5 – Tracé du chemin et arbres de recherche sans et avec l'heuristique

### 1.2.4. Commentaires

On remarque que le chemin trouvé n'est pas toujours identique suivant si on utilise ou pas l'heuristique. Cependant, dans tous les cas, le nombre de pas (c'est-à-dire le nombre de déplacements) est identique. Le programme semble donc bien fonctionner.

Par ailleurs, on remarque que sans l'heuristique, le nombre de nœuds ouvertes et fermés et bien supérieur à celui qu'on obtient avec l'heuristique. L'heuristique est donc efficace et permet bien de trouver plus rapidement le chemin le plus court.

### 1.2.5. Cas où le chariot ne peut pas accéder à l'objet qu'il doit récupérer

Supposons par exemple qu'on veuille que le chariot situé en $(14, 13)$ aille chercher l'objet se trouvant en $(5, 7)$ en passant par le sud. Ceci n'est pas possible car un autre chariot bloque le passage. Une boîte de dialogue s'ouvrira en indiquant qu'il n'y a pas de solution.

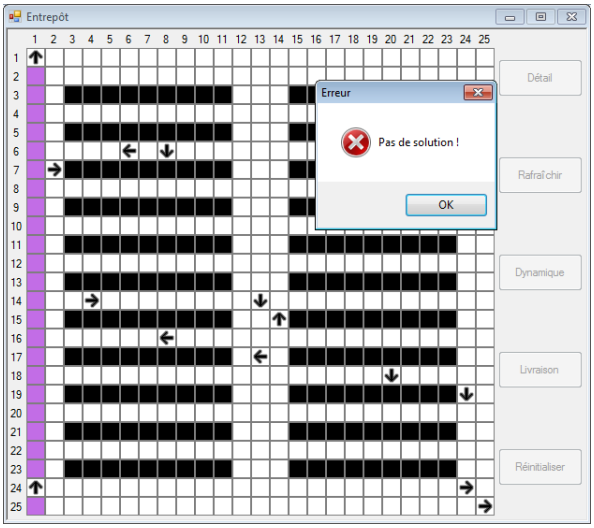



FIGURE 6 – Illustration du cas où il n'existe pas de chemin pour accéder à l'objet à récupérer

## 1.3. Question 2

### 1.3.1. Récupération de l'objet

Pour cette question, il s'agissait cette fois-ci de prendre en compte une contrainte de temps. Pour cela, nous avons codé la classe `NoeudTemps`, héritée de `Noeud`, dans laquelle nous avons modifié le coût du déplacement (qui était toujours à 1 précédemment) en fonction de la trajectoire pris par le chariot (1 seconde s'il continue tout droit, 4 s'il prend un virage et 7 s'il fait demi-tour). Pour le reste, elle ne diffère guère de la classe `NoeudDistance`.

### 1.3.2. Utilisation d'une heuristique

Pour « accélérer » la recherche du chemin le plus rapide, nous avons utilisé, comme pour la question précédente, la distance euclidienne entre le point de départ et le point d'arrivée.

### 1.3.3. Livraison de l'objet

Comme pour la récupération d'un objet, on a une contrainte de temps. Cependant, contrairement à la récupération, on ne connaît pas à l'avance la destination exacte du chariot. Nous avons donc choisi de créer la classe `NoeudLivraison`, héritée de `Noeud`, fort semblable à la classe `NoeudTemps`, mais dont la condition d'arrêt diffère : ici on vérifie juste que le chariot est sur une case de la zone de livraison (colonne 1).

### 1.3.4. Utilisation d'une heuristique

Pour « accélérer » la recherche du chemin le plus rapide, nous avons utilisé une heuristique qui dépend de l'orientation du chariot. Comme le chariot doit se rendre à la zone de livraison qui se trouve à l'ouest de l'entrepôt, on ajoute un coût de 0 si le chariot est orienté vers l'ouest, de 1 s'il est dirigé vers le nord ou le sud et de 2 s'il est orienté vers l'est.

### 1.3.5. Illustrations

**Tracé du chemin le plus court et du plus rapide :**

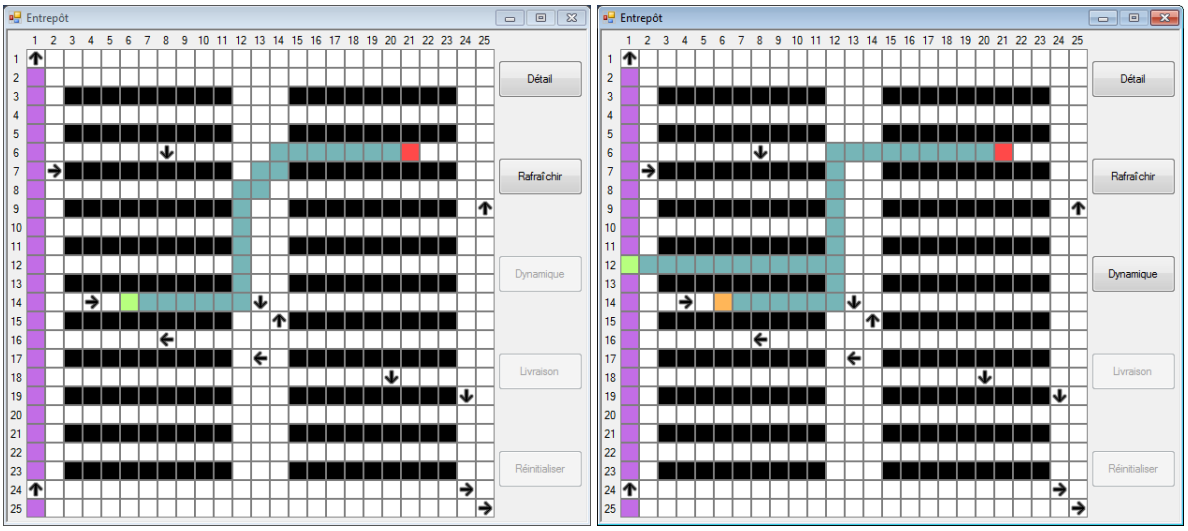



FIGURE 7 – Comparaison entre le chemin le plus court et le plus rapide

**Arbre de recherche avec ou sans l'heuristique de livraison :**

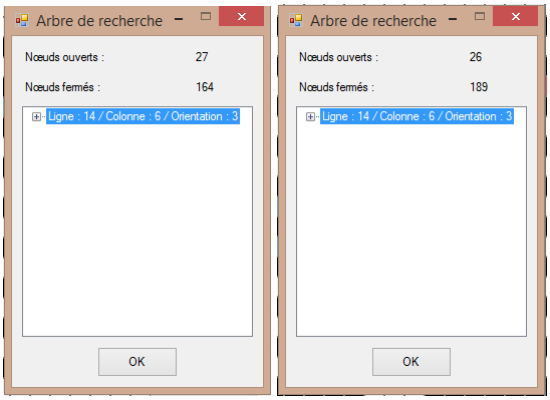



FIGURE 8 – Comparaisons avec ou sans heuristiques

### 1.3.6. Commentaires

On remarque que les chemins diffèrent lorsque la contrainte est temporelle ou en terme de distance. En effet, avec la contrainte de temps, le chariot effectue beaucoup moins de changement de direction car cela est plus chronophage. On remarque néanmoins que les deux trajets ont le même nombre de déplacements.

On remarque de plus très bien que sans les heuristiques, la somme des nœuds fermés et ouverts est beaucoup plus importante.

## 1.4. Question 3

Pour cette question, il faut, normalement, d'abord placer tous les chariots sur la première colonne (la zone de livraison). Ensuite, il faut sélectionner un des chariots sur la zone de livraison et indiquer l'objet à récupérer.

On va alors calculer les chemins les plus courts pour que chaque chariot aille chercher son objet. Les objets sont attribués aléatoirement aux chariots.

Nous avons utilisé les classes `NoeudTemps` et `NoeudLivraison`, mais nous avons également dû créer une nouvelle classe `NoeudRealite` (également héritée de la classe `Noeud`), utilisée dans notre première approche et qui permet de trouver le chemin d'un chariot en fonction des chemins déjà trouvés pour les précédents chariots.

#### 1.4.1. Première approche

Dans un premier temps, nous avons voulu que l'algorithme calcule le chemin le plus rapide pour le premier chariot, puis le chemin le plus court pour le deuxième en tenant compte du premier, et ainsi de suite jusqu'au chemin du chariot qu'on a sélectionné (en tenant compte des chemins de tous les autres chariots). Nous avons cependant rencontré des problèmes en procédant de cette manière. En effet, certains chariots fusionnaient entre eux. Nous avons donc abandonné cette première méthode (cependant toujours présent dans le code).

#### 1.4.2. Seconde approche

Dans un second temps, nous avons décidé de procéder en recalculant la trajectoire de chaque chariot à chaque seconde. Ainsi, tant que tous les chariots ne sont pas retournés livrer leur objet, on recalcule la position de tous les chariots à la seconde suivante Ensuite, on montre le déplacement des chariots dynamiquement à l'écran en utilisant un timer qui modifie l'image toutes les secondes (toutes les demi-secondes afin d'accélérer le processus).

#### 1.4.3. Utilisation d'une heuristique

Les heuristiques utilisées sont les mêmes que dans la question 2 puisqu'on utilise les mêmes classes que précédemment.

#### 1.4.4. Commentaires

Notre première approche ne fonctionne pas bien. En effet, certains chariots se passaient dessus (les chariots dont le chemin est calculé « avant » passent sur ceux dont le chemin a été calculé « après »).

Dans notre seconde approche, l'algorithme fonctionne bien. Les chariots ne se percutent et ne fusionnent pas. Ils respectent bien les contraintes de temps imposées pour les virages, les demi-tours et la récupération de l'objet lorsqu'il doit monter puis redescendre.

# 2. Partie 2

## 2.1. Question 1

Au départ, les $w_i$ sont générés de manière aléatoire (entre 1 et 5). Les méthodes utilisées sont présentes dans une classe `Neurone` et sont appelées depuis le formulaire.

On pose $y$ = poids de l'espèce et $x$ = poids de l'espèce. La droite est définie à partir de deux points : $y_1 = 0$ et $y_2 = 100$.

Pour $y_1 = 0$, on a $w_1 \cdot x_1 + w_3 = 0 \Leftrightarrow x_1 = -\frac{w_3}{w_1}$.

Pour $y_2 = 100$, on a $w_1 \cdot x_2 + w_2 \cdot y_2 + w_3 \cdot 1 = 0 \Leftrightarrow x_2 = \frac{-w3-w2 \cdot y_2}{w_1}$.

La convergence se fait en moyenne à partie de la douzième itération.

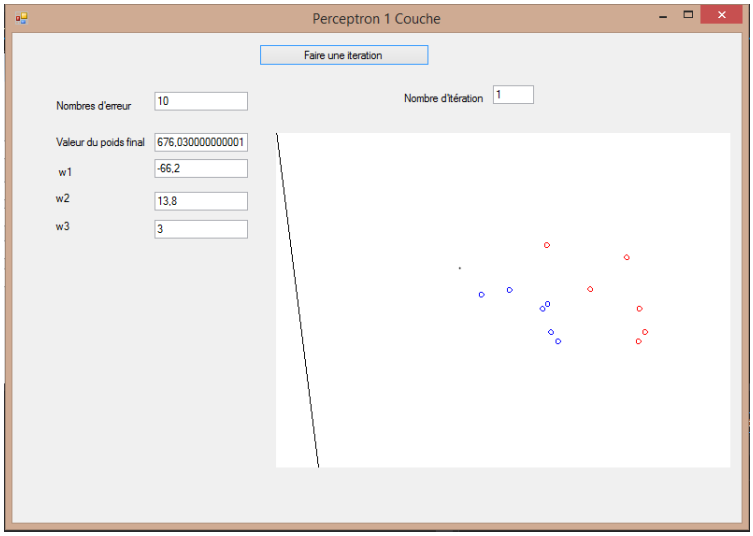



FIGURE 9 – Écran initial à la première itération

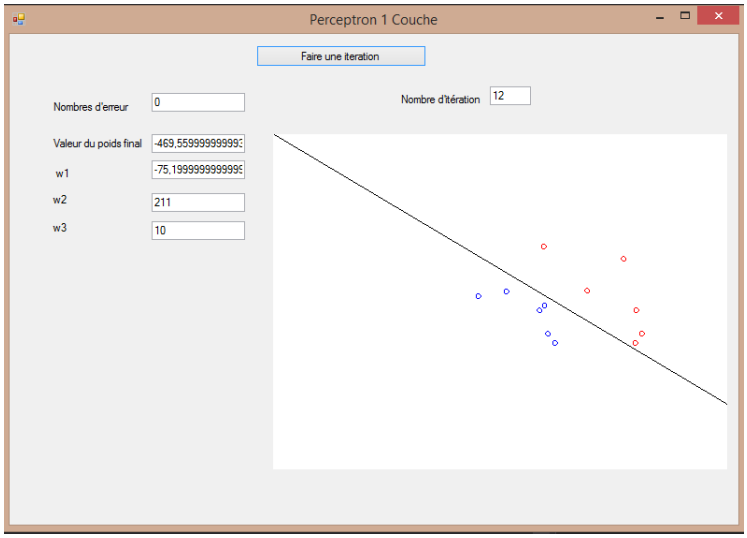
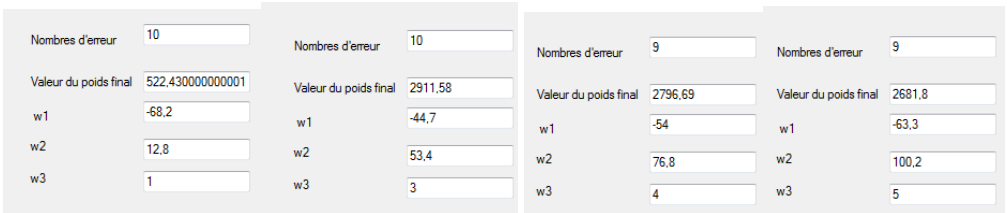



FIGURE 10 – Écran montrant la convergence au bout de 12 itérations

On remarque que la droite sépare bien les deux groupes de points.



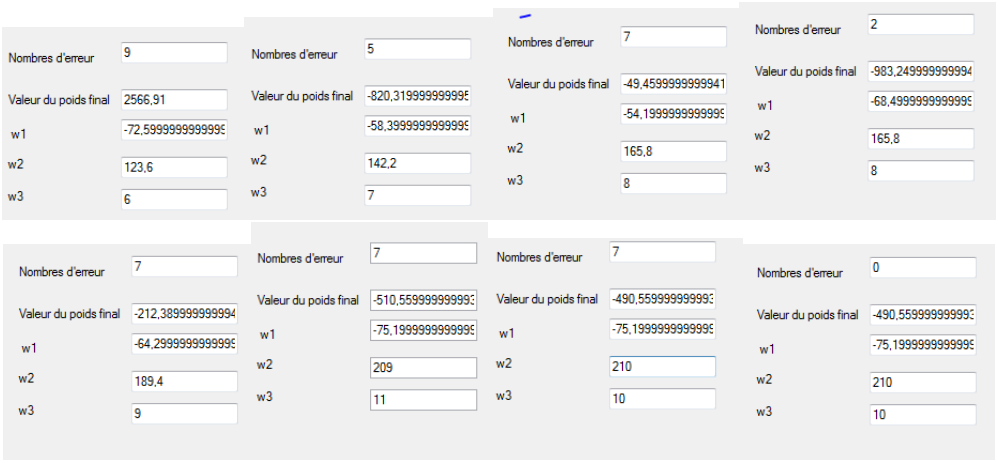



FIGURE 11 – Exemple de la valeur des poids à chaque itération pour une session convergent au bout de 12 itérations

## 2.2. Question 2

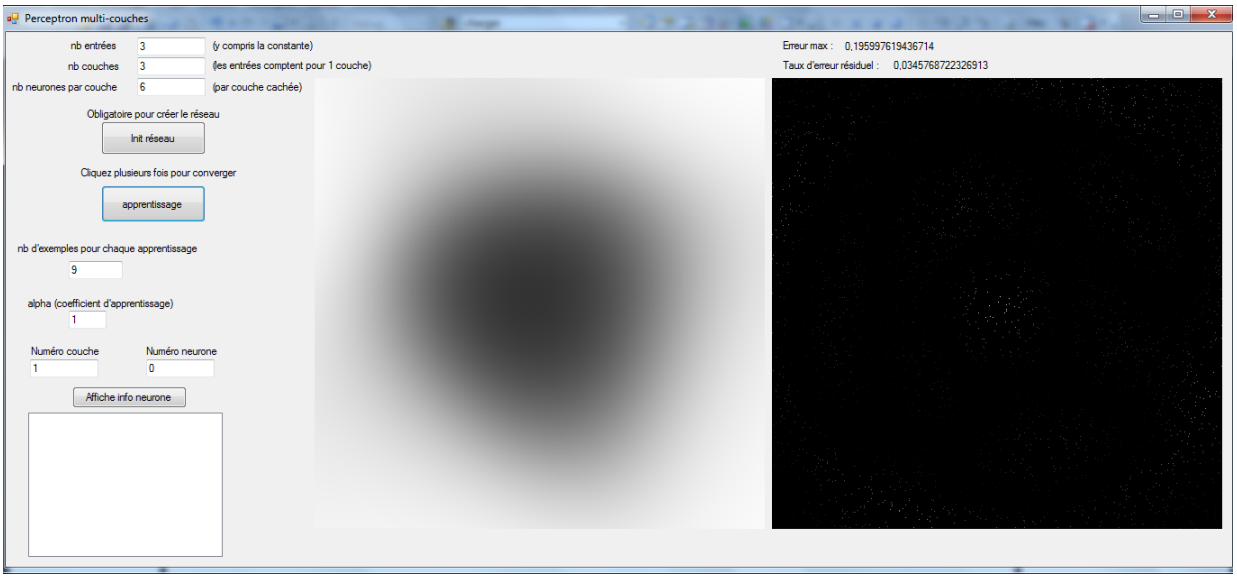
### 2.2.1. On fixe une référence

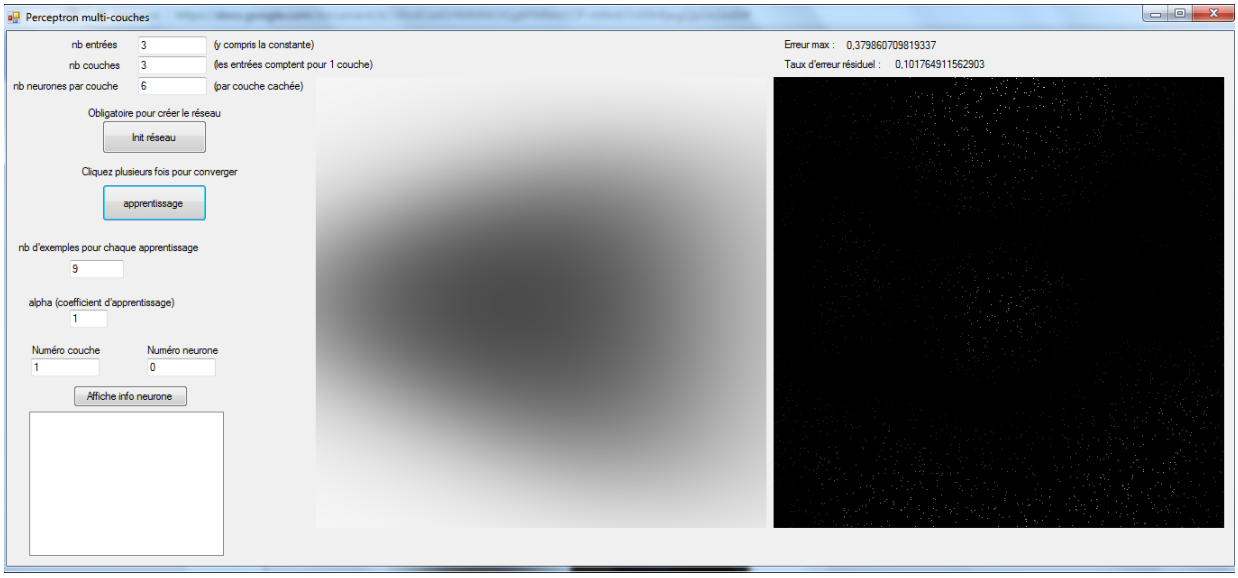
Il est intéressant de déterminer des valeurs minimales de référence, correspondant aux valeurs où le taux d'erreur résiduel est déjà stable, pour pouvoir effectuer les comparaisons des paramètres.

Pour pouvoir évaluer les valeurs de référence, il faut déterminer un nombre fixe d'itérations d'apprentissage. Afin de déterminer le nombre pertinent d'exemples pour chaque apprentissage, plusieurs tests ont été réalisés en ne faisant varier que ce nombre. Après ces nombreux tests, 10 exemples pour chaque apprentissage semblent être le plus pertinent. Le taux d'erreur résiduel stagne autour de 0,03. Si on prenait que 9 exemples pour chaque apprentissage, le taux d'erreur résiduel ne convergeait pas.

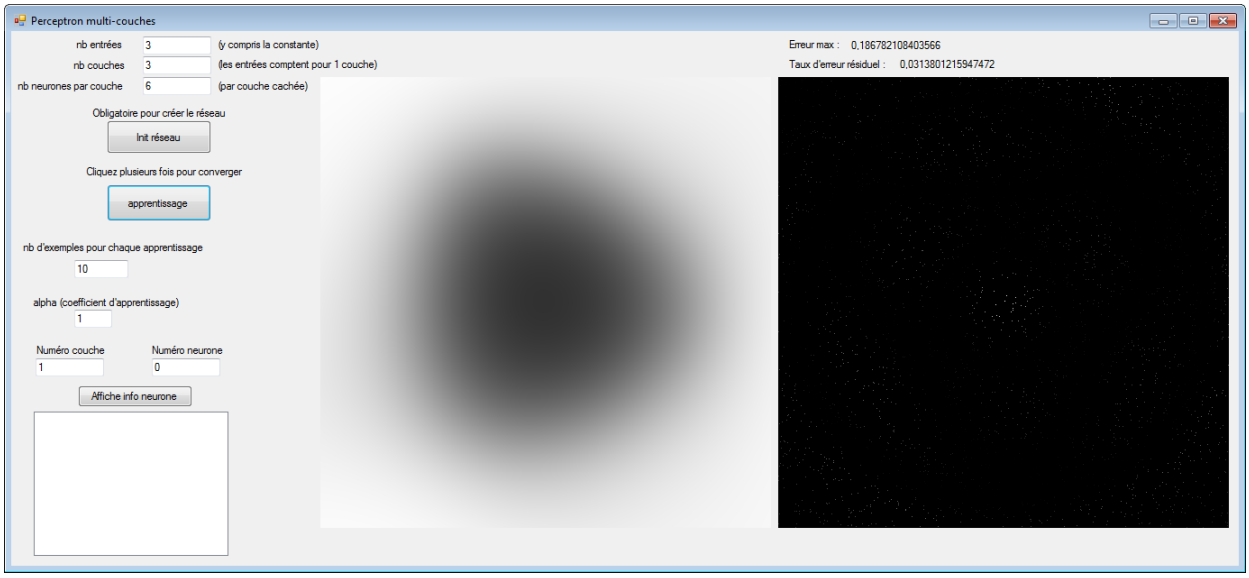
Voici les tests : (2 exemples d'initialisation pour chaque valeur)

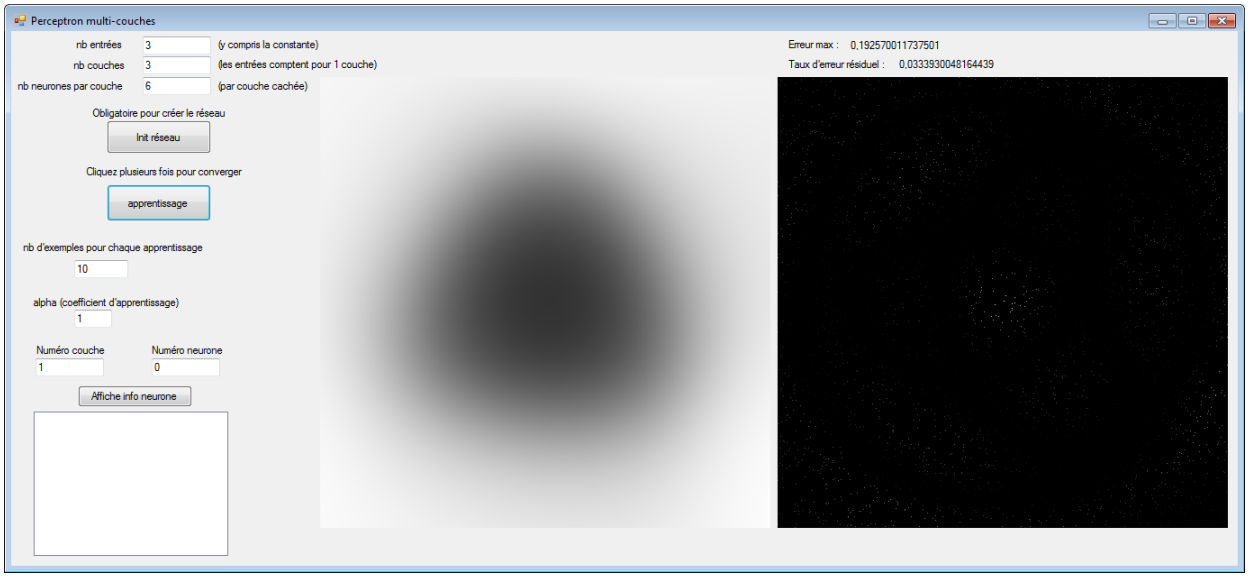
*Nombre d'exemples pour chaque apprentissage : 9*






*Nombre d'exemples pour chaque apprentissage : 10*

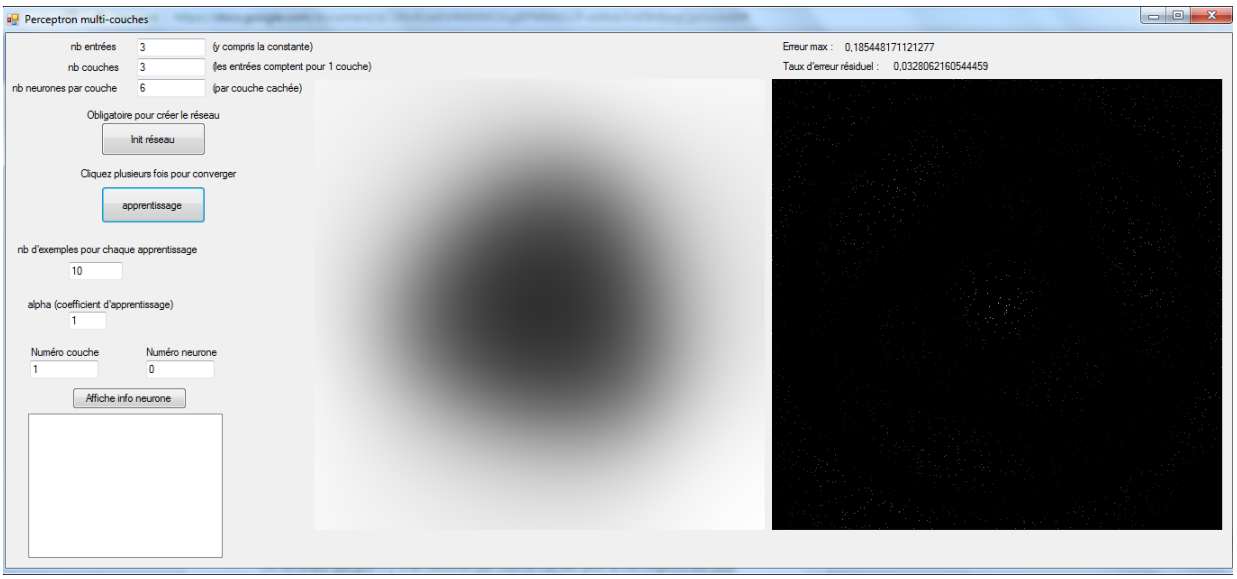


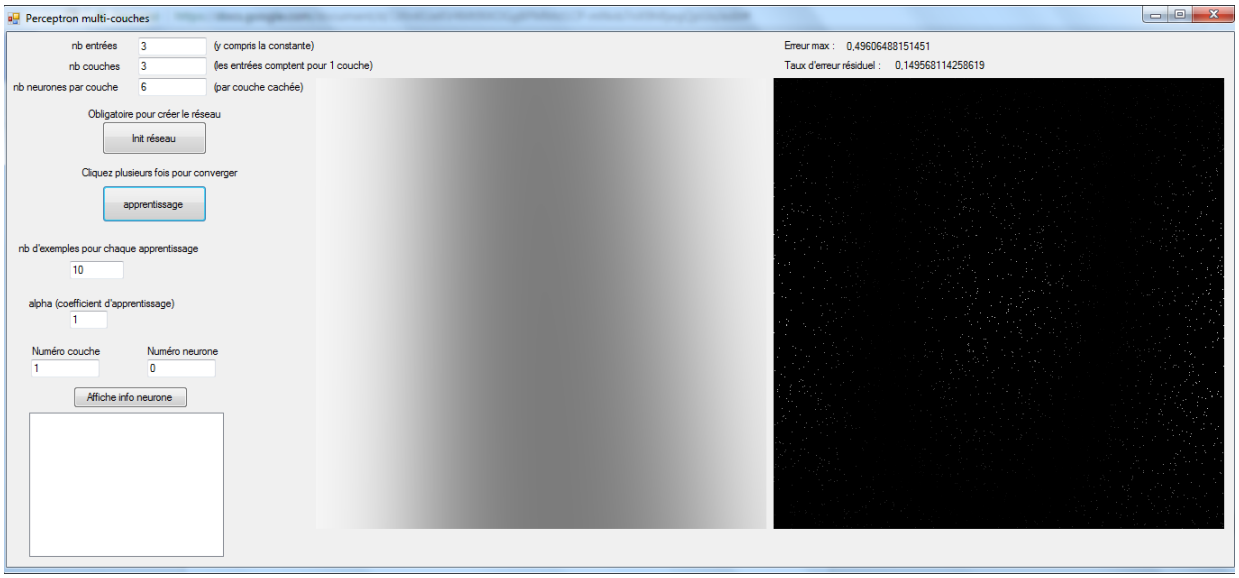



En faisant varier le nombre de neurones sur la couche cachée, on remarque que 7 neurones semblent être le plus pertinent. Après plusieurs initialisations, le taux d'erreur résiduel stagne autour de 0,03 avec 7 neurones.

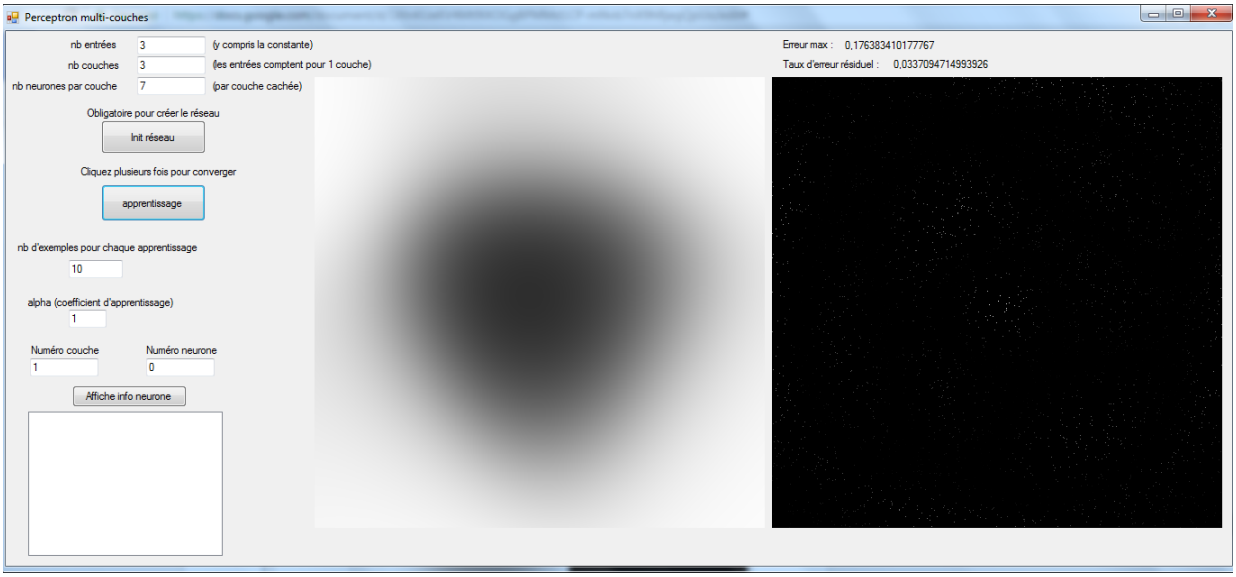
Voici les tests : (2 exemples d'initialisation pour chaque valeur)

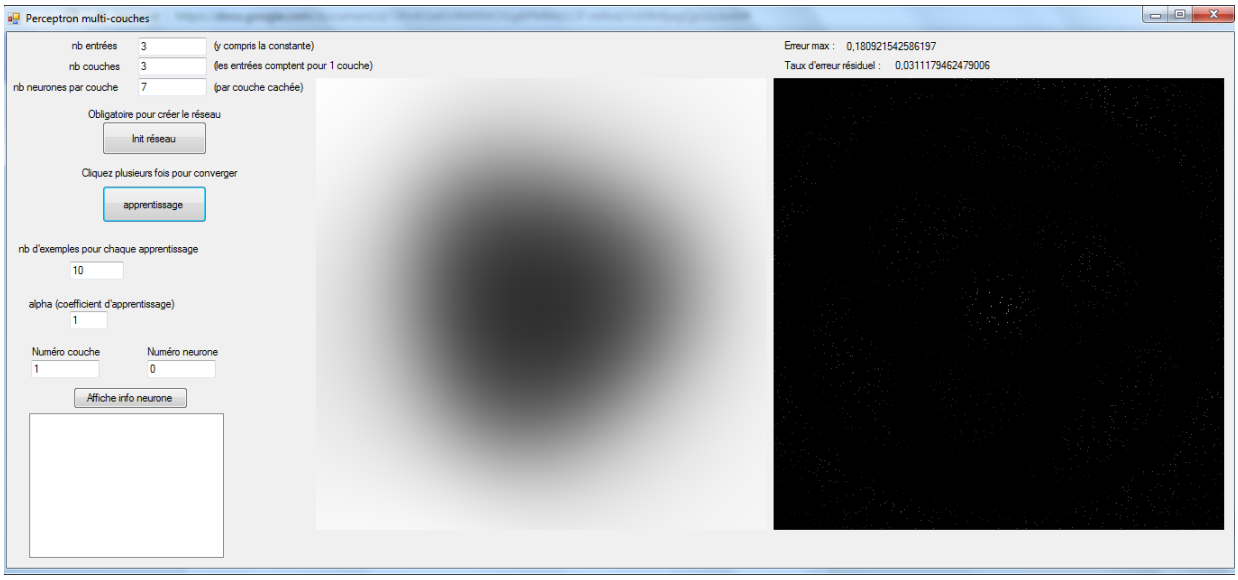
*Nombre de neurones par couche couchée : 6*



*Nombre de neurones par couche couchée : 7*

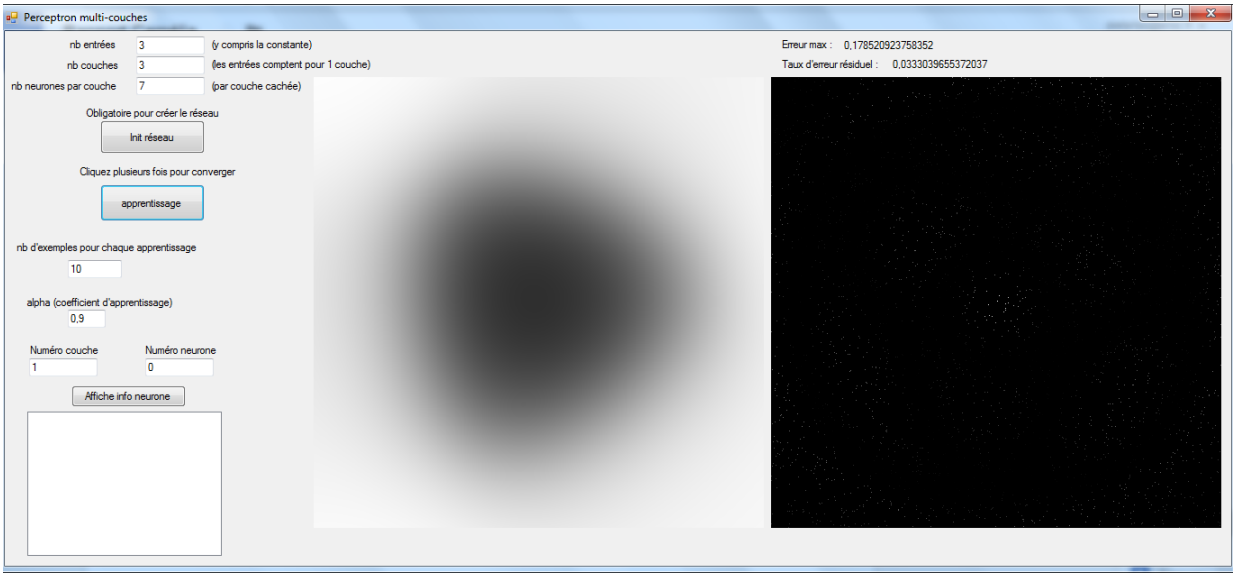


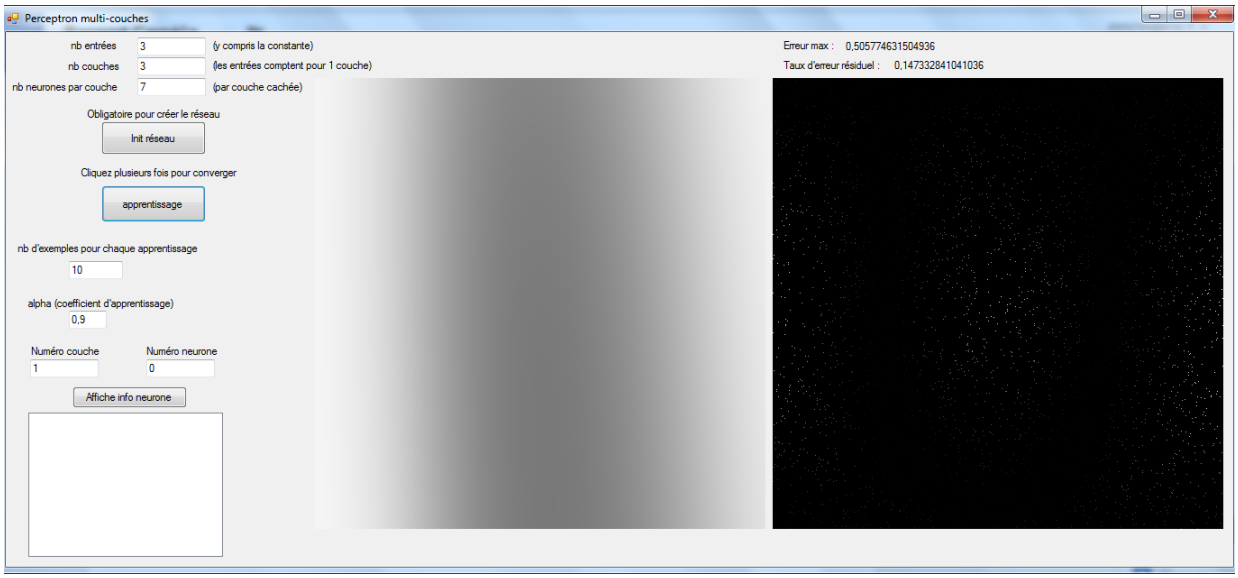




En faisant varier le coefficient d'apprentissage, on remarque que 0,95 est le coefficient le plus pertinent. Après plusieurs initialisations, le taux d'erreur résiduel stagne autour de 0,03 lorsque le coefficient vaut 0,95.

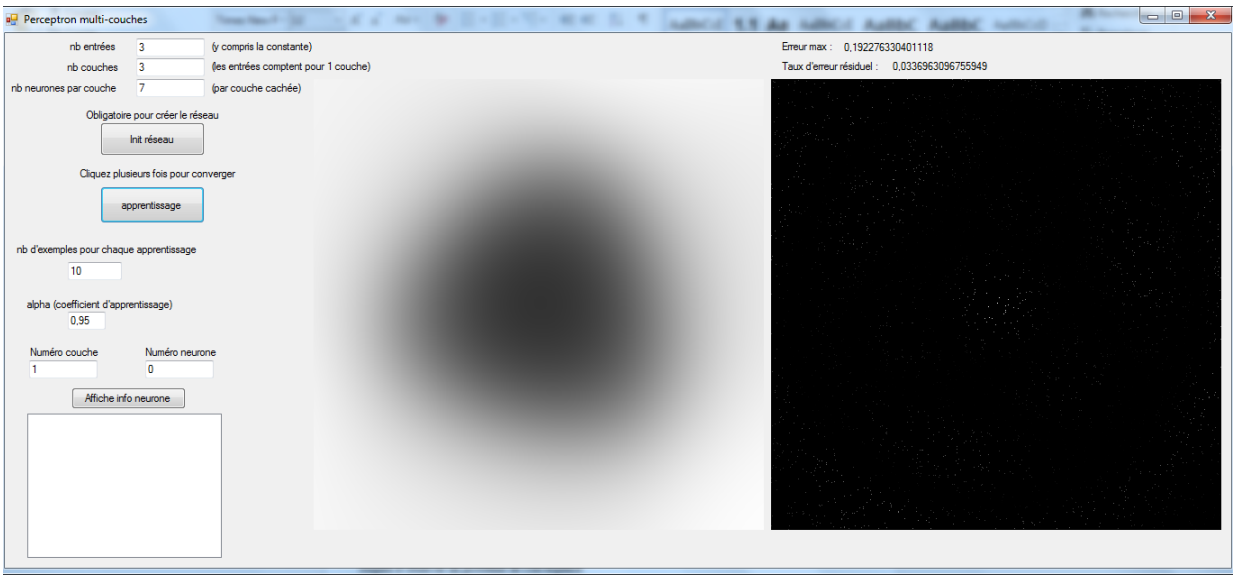
Voici les tests : (2 exemples d'initialisation pour chaque valeur)

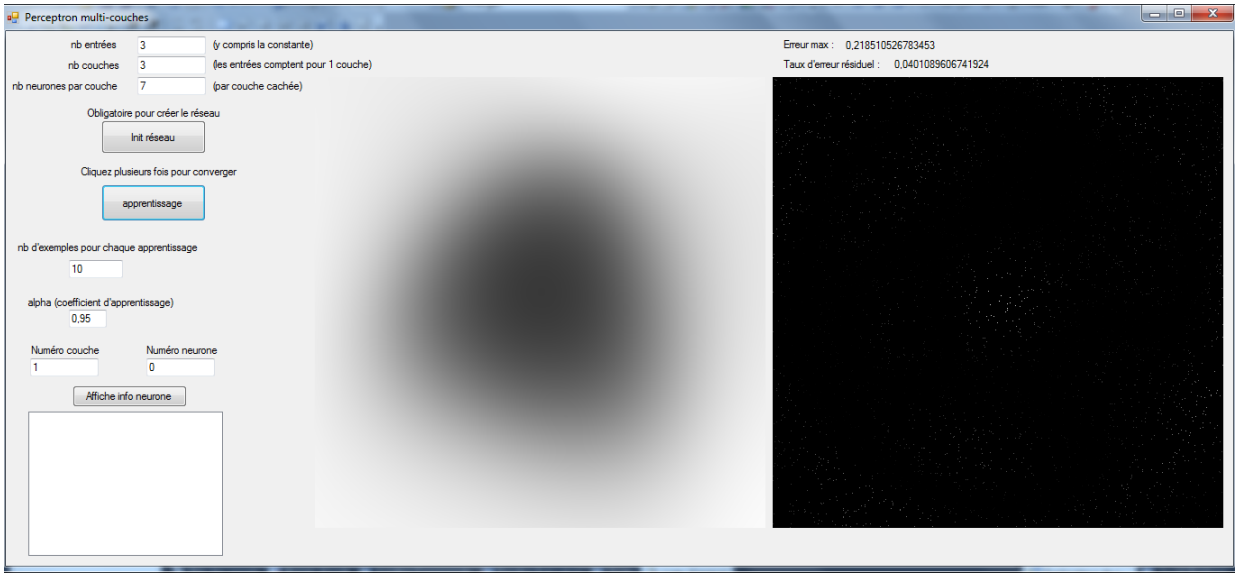
*Coefficient d'apprentissage : 0,9*




*Coefficient d'apprentissage : 0,95*






En résumé, les paramètres les plus pertinents pour obtenir un réseau efficace au bout de 10 itérations, sont :

| | |
|---|---|
| Nombre de neurones par couche | 7 |
| Coefficient d'apprentissage | 0,95 |

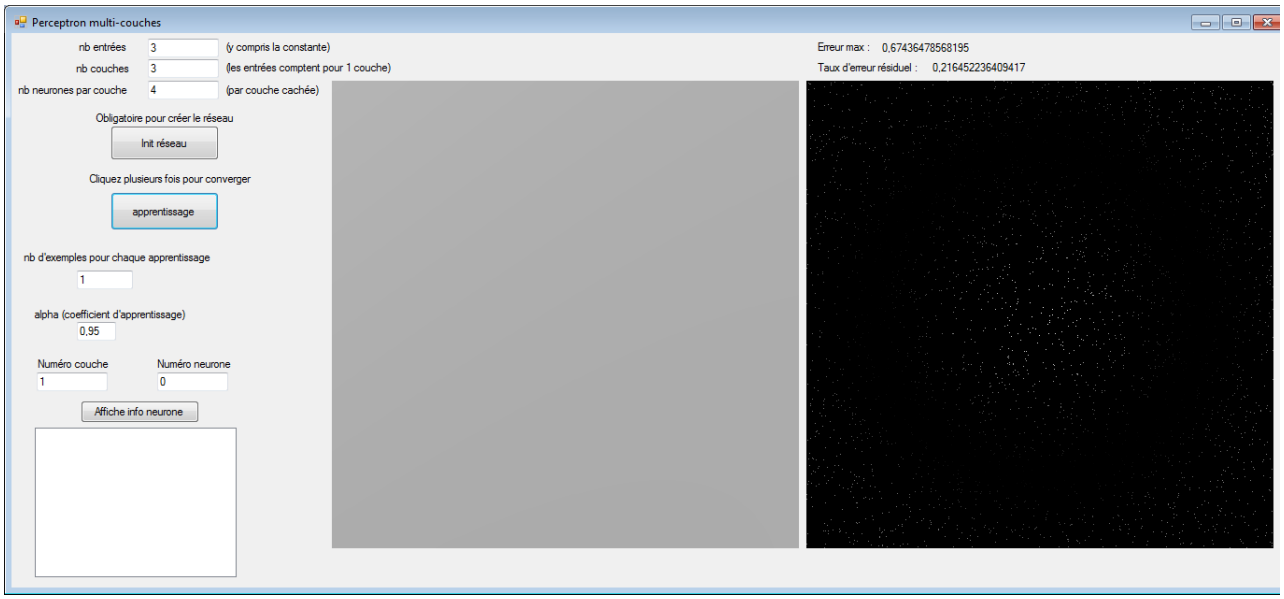
### 2.2.2. On étudie les vitesses de convergence

Dans cette partie, il est intéressant d'étudier les vitesses de convergence si on fait varier un seul paramètre, et en prenant l'autre paramètre avec la valeur de référence.

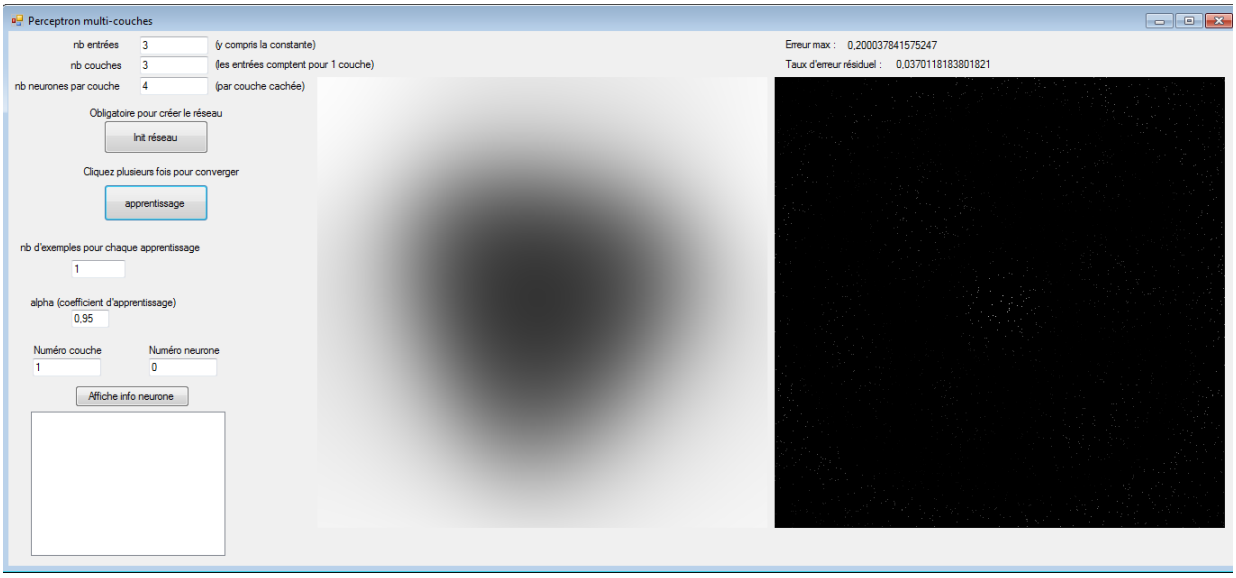
**Si on fait varier le nombre de neurones par couche**

*Nombre de neurones par couche : 4*

Lors de l'initialisation, on obtient les valeurs suivantes :

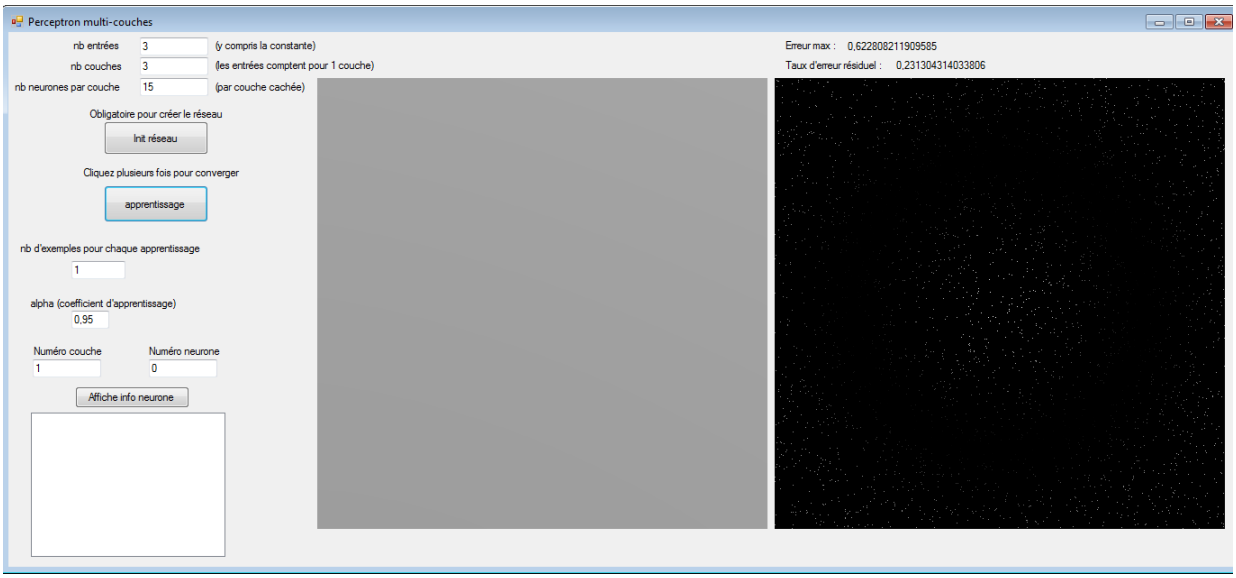



Pour arriver à une convergence du taux d'erreur résiduel, il faut effectuer 14 itérations. Voici les valeurs obtenues :

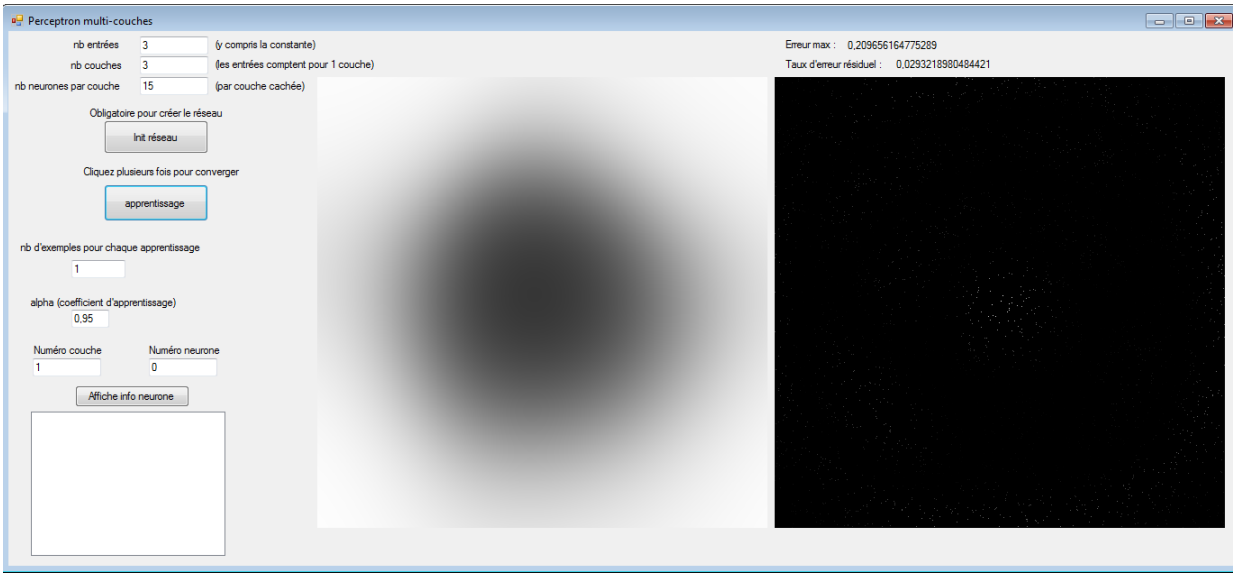



*Nombre de neurones par couche : 15*

Lors de l'initialisation, on obtient les valeurs suivantes :




Pour arriver à une convergence du taux d'erreur résiduel, il faut effectuer 14 itérations. Voici les valeurs obtenues :

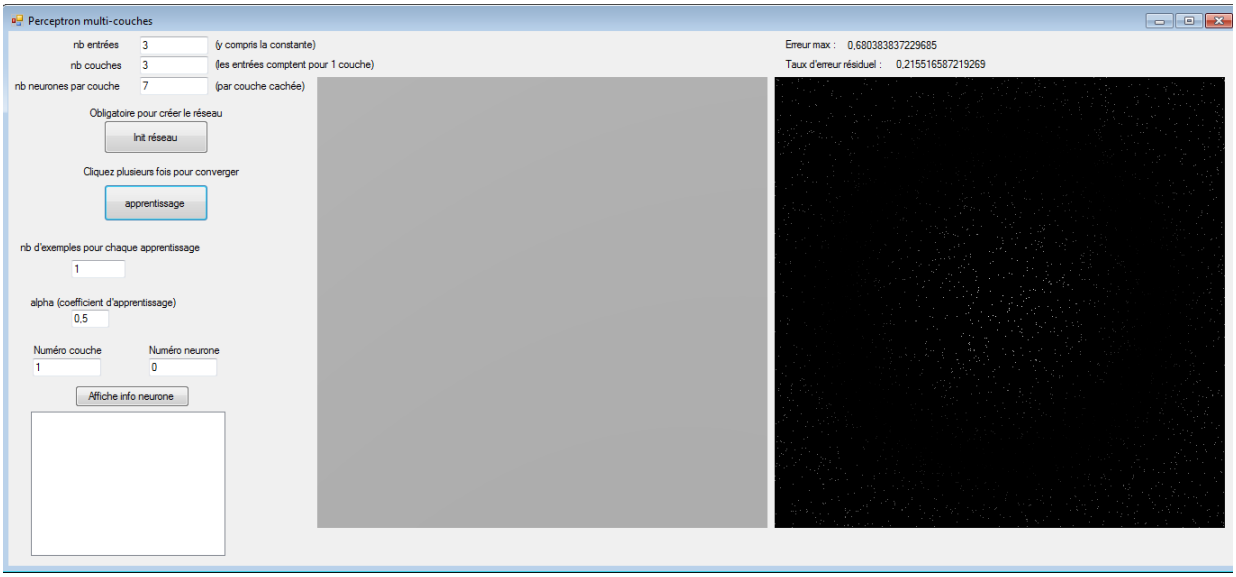



On remarque donc que plus le nombre de neurones par couche cachée est élevé, plus la vitesse de convergence est élevée.

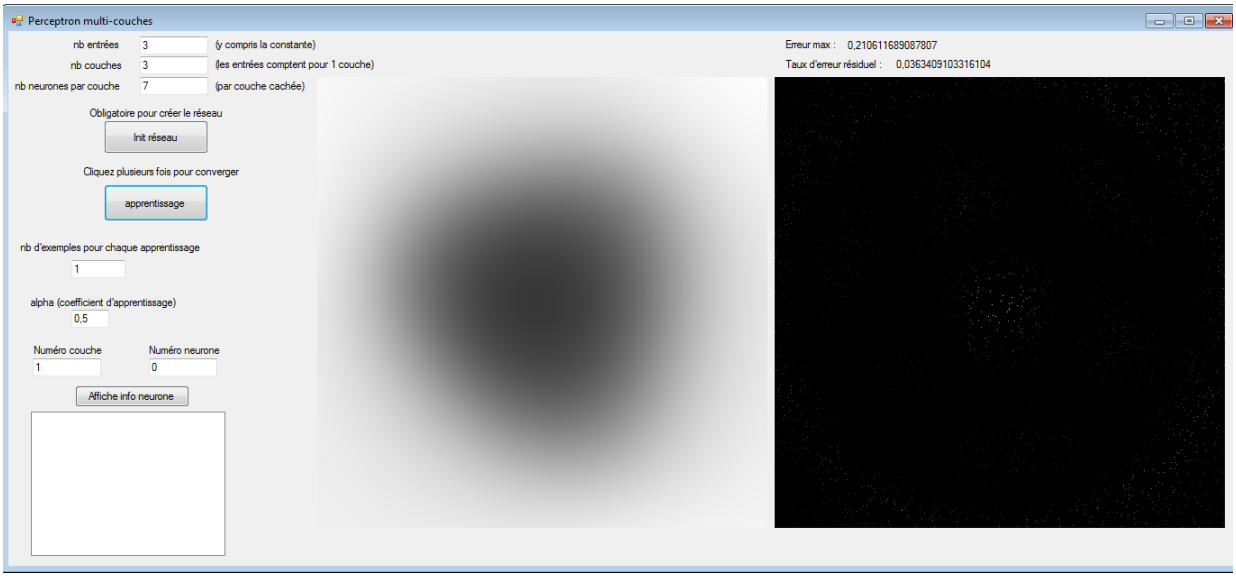
**Si on fait varier le coefficient d'apprentissage**

*Coefficient d'apprentissage : 0,5*

Lors de l'initialisation, on obtient les valeurs suivantes :

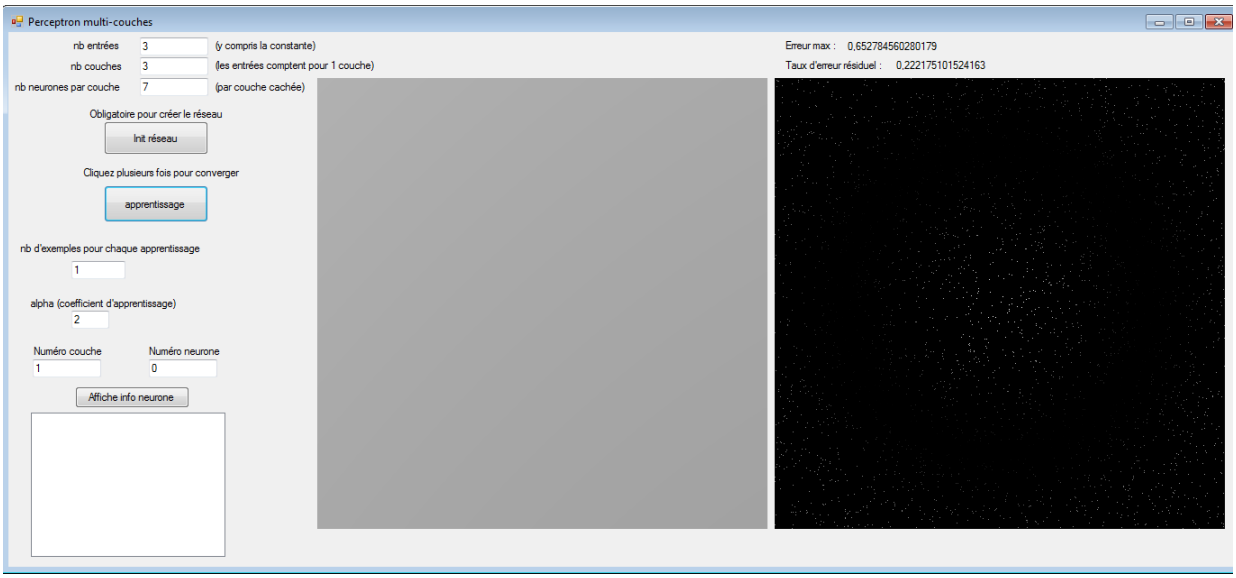



Pour arriver à une convergence du taux d'erreur résiduel, il faut effectuer 10 itérations. Voici les valeurs obtenues :

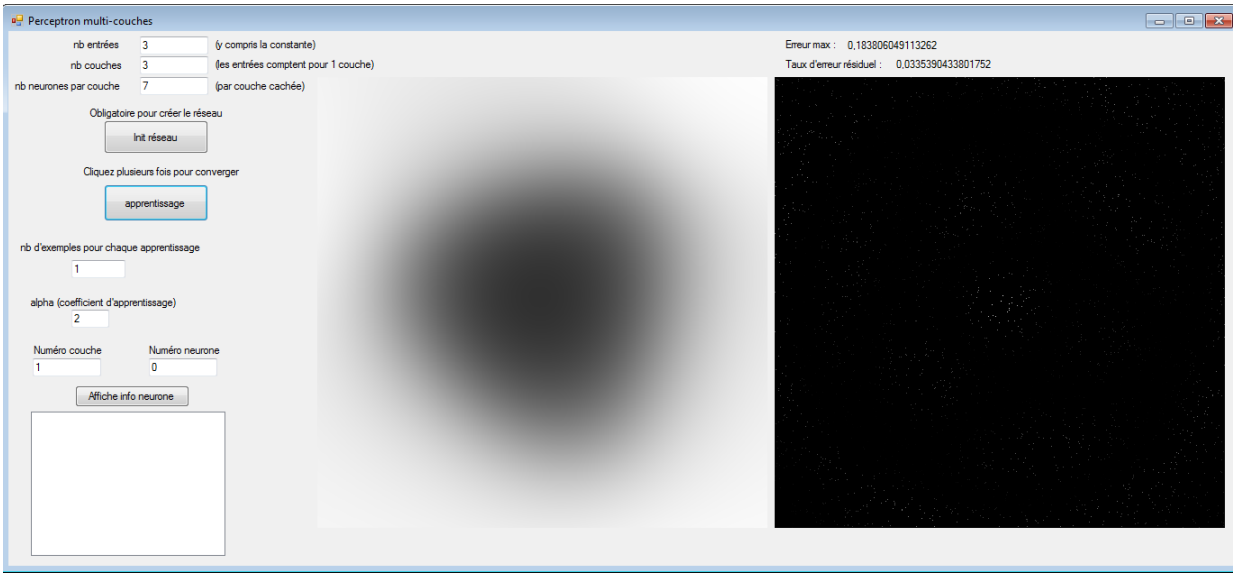



*Coefficient d'apprentissage : 2*

Lors de l'initialisation, on obtient les valeurs suivantes :




Pour arriver à une convergence du taux d'erreur résiduel, il faut effectuer 10 itérations. Voici les valeurs obtenues :




On remarque donc que plus le coefficient d'apprentissage est élevé, plus la vitesse de convergence est élevée.

### 2.2.3. Conclusion

Grâce à tous ces résultats, on peut conclure que plus il y a de neurones par couche cachée, plus la vitesse d'apprentissage est forte, autrement dit, que le taux d'erreur résiduel diminue alors que le nombre de neurones par couche augmente. De même avec le coefficient d'apprentissage : plus le coefficient d'apprentissage est élevé, plus la vitesse d'apprentissage est forte.

### 2.2.4. Question subsidiaire

Les données du fichier ont été obtenues grâce à la fonction : $z = x^2 + y^2$.

## 2.3. Question 3

### 2.3.1. Apprentissage supervisé

Dans cette partie, on utilise le perceptron multi-couches.

Dans un premier temps, on construit le perceptron multi-couches en fonction des souhaits de l'utilisateur : nombre de couches du réseau ($\geqslant 3$) et nombre de neurones sur les couches cachées.

Ensuite, on récupère les données du fichier `dataetclassif.txt` qu'on trie de façon à intercaler entre chaque donnée de la classe $A$, une donnée de la classe $B$.

Puis, on rentre dans la phase d'apprentissage du réseau, *via* la rétropropagation du gradient, suivant les paramètres entrés par l'utilisateur (nombre d'itérations et coefficient d'apprentissage). On lui demande ainsi de minimiser l'erreur commise entre les sorties obtenues lorsque le réseau les calcule à partir des données d'entrée, et les sorties théoriques.

Finalement, afin de vérifier si l'apprentissage a bien été réussi, on teste le réseau sur les pixels de l'image.

## Illustrations

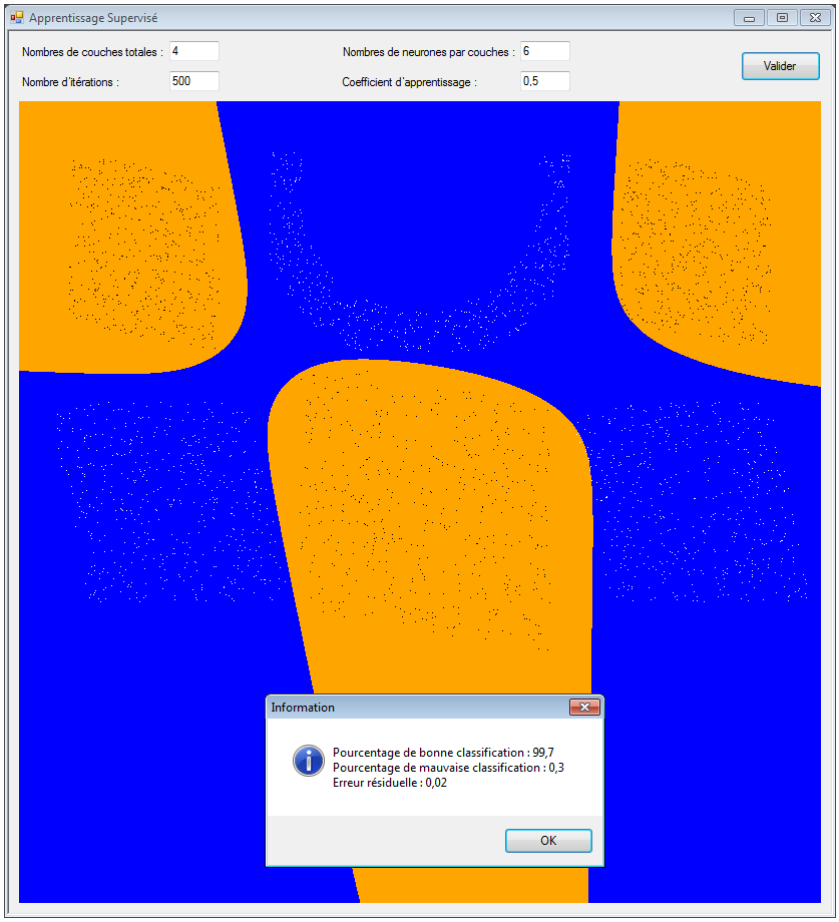



FIGURE 12 – Image de référence

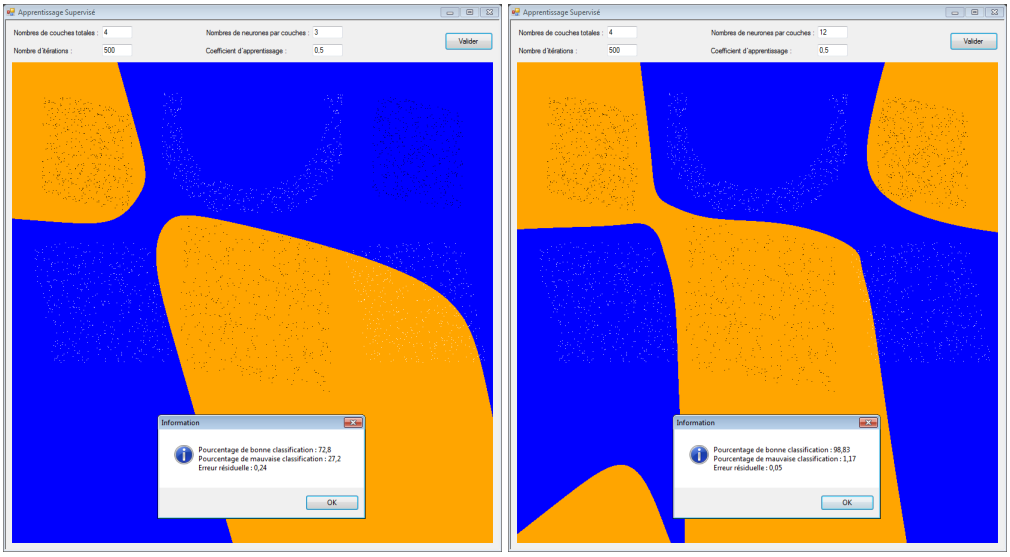


FIGURE 13 – Variation sur le nombre de neurones par couches cachées

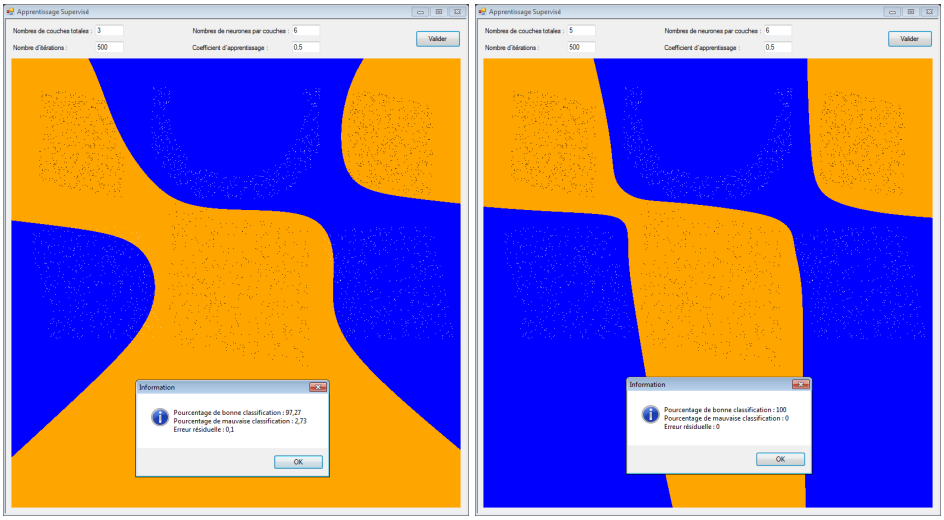



FIGURE 14 – Variation sur le nombre de couches

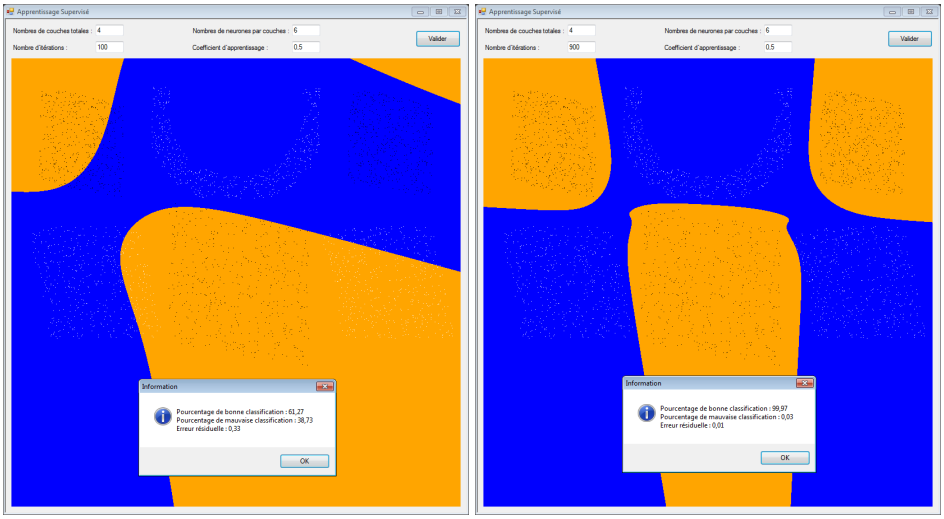



FIGURE 15 – Variation sur le nombre d'itérations

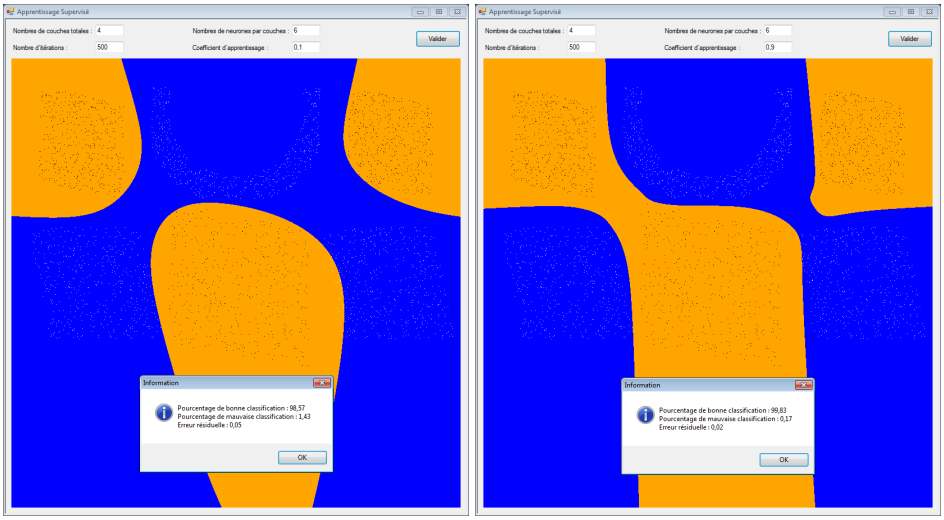



FIGURE 16 – Variation sur le coefficient d'apprentissage

**Commentaires**

***Performances d'apprentissage***

On évalue les performances d'apprentissage grâce à l'erreur résiduelle (moyenne des écarts entre sorties obtenues et sorties souhaitées) et aux pourcentages de bonne et mauvaise classification (proportion de données bien et mal classées).

On remarque que si le nombre d'itérations est trop faible, l'apprentissage du réseau sera assez médiocre. Par exemple, pour 100 itérations, il classifie quasiment toutes les entrées dans la même classe.

Si le coefficient d'apprentissage est trop faible ou trop fort, l'apprentissage sera également mauvais. Pour un coefficient trop faible, l'apprentissage est trop lent, et pour un coefficient trop élevé, il apprend et fait des rapprochements hasardeux.

Enfin, si le nombre de couches et le nombre de neurones par couches cachées sont trop faibles, les performances d'apprentissage du réseau seront moins bonnes. Dans tous les cas, plus il y aura de neurones au total dans le réseau, meilleur sera l'apprentissage.

***Temps d'apprentissage***

On évalue le temps d'apprentissage au moyen d'un `timer` (qui ne fonctionne malheureusement pas) qui compte le nombre de secondes écoulées entre le moment où le programme débute le calcul et le moment où le programme termine le calcul.

Au niveau du temps, plus le nombre d'itérations est important, plus le calcul est long. En effet, lorsqu'on multiplie par 10 le nombre d'itérations, on multiplie par 30000 le nombre de calculs (car il y a 3000 données). Ainsi, il faut faire un compromis entre performances et temps. On prendra donc 500.

Cependant, le coefficient d'apprentissage ne modifie pas le temps de calcul. Donc autant prendre un coefficient ni trop faible ni trop fort. On prendra donc 0,5.

Pour finir, plus il y aura de neurones (2 + (nombre de couches – 2) · nombre de neurones par couche cachée + 1), plus il y aura de calculs et donc plus le temps d'apprentissage sera long. On prendra donc 4 couches de 6 neurones par exemple.

### 2.3.2. Apprentissage non-supervisé

Dans cette partie, on utilise une carte de Kohonen.

Dans un premier temps, on construit la carte de Kohonen en fonction des souhaits de l'utilisateur : nombre de lignes et nombre de colonnes de la carte définissant le nombre de neurones (nombre de neurones = nombre de lignes x nombre de colonnes de la carte).

Ensuite, on récupère les données du fichier `dataetclassif.txt` (les observations) qu'on mélange de façon à ce que l'on est bien convergence et donc que les 6 classes soient bien détectées.

Puis, on rentre dans la phase d'apprentissage de la carte en fonction des paramètres entrés par l'utilisateur (coefficient d'apprentissage, distance pour laquelle on considère que 2 neurones sont voisins), *via* l'algorithme de Kohonen. On lui demande ainsi de modifier les poids de ses neurones de telle façon à ce qu'elles soient au plus proche des observations.

Une fois que la carte se stabilise, c'est-à-dire si les neurones n'ont plus que très peu de mouvements, on peut procéder au regroupement, c'est-à-dire à la détermination des différentes classes. Ici, on veut 6 classes. En fonction des distances entre les différents neurones de la carte, le programme va les regrouper dans 6 classes distinctes.

Finalement, afin de vérifier si la classification a bien été effectuée, on teste la carte sur les pixels de l'image.

## Illustrations pour 500 itérations

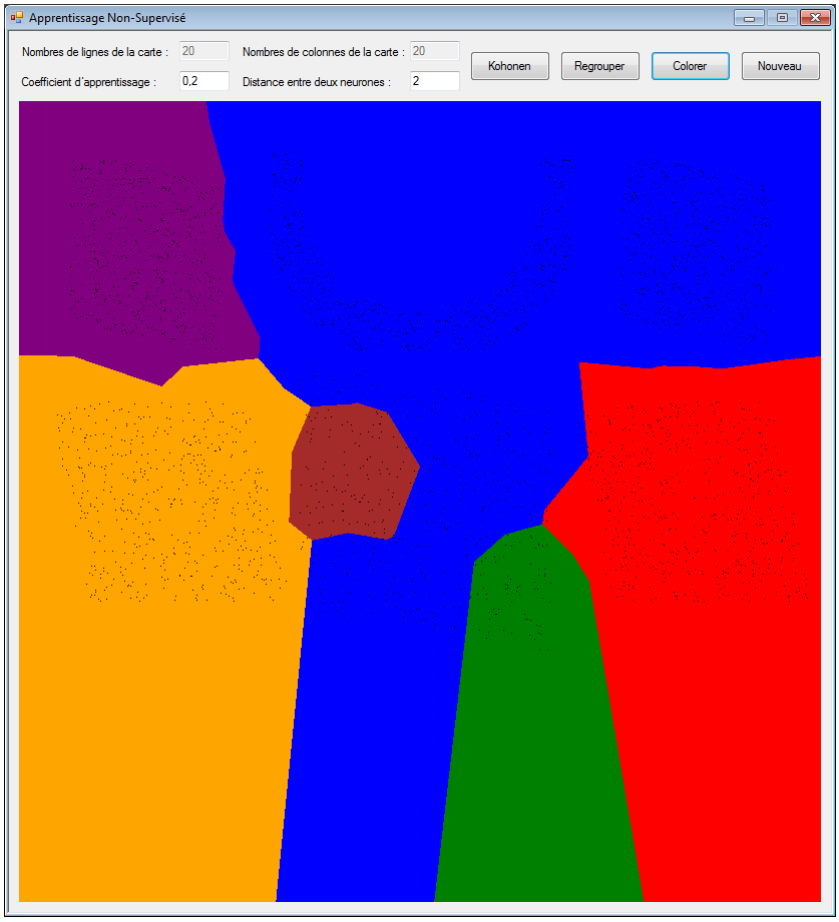



FIGURE 17 – Image de référence

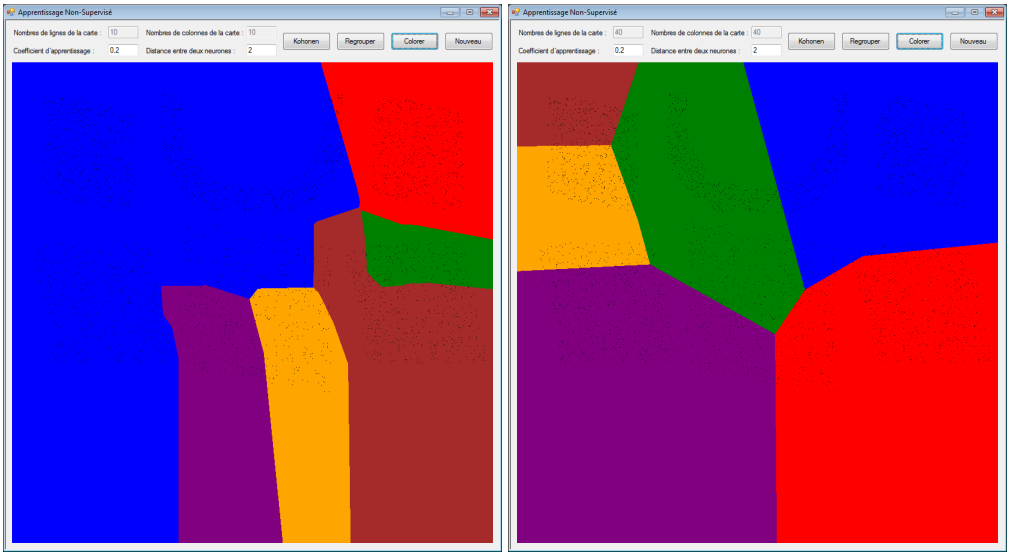


FIGURE 18 – Variations sur la taille de la carte

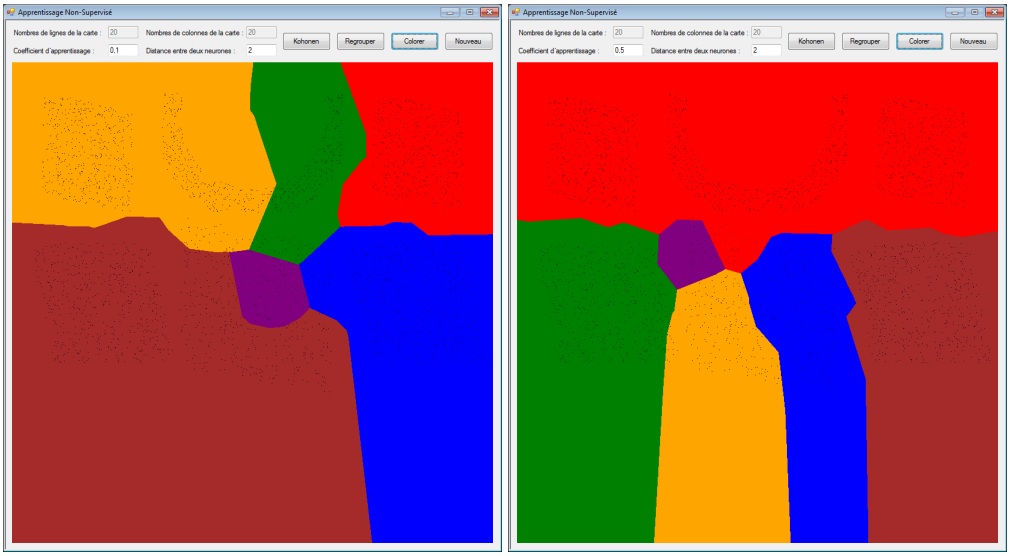



FIGURE 19 – Variations sur le coefficient d'apprentissage

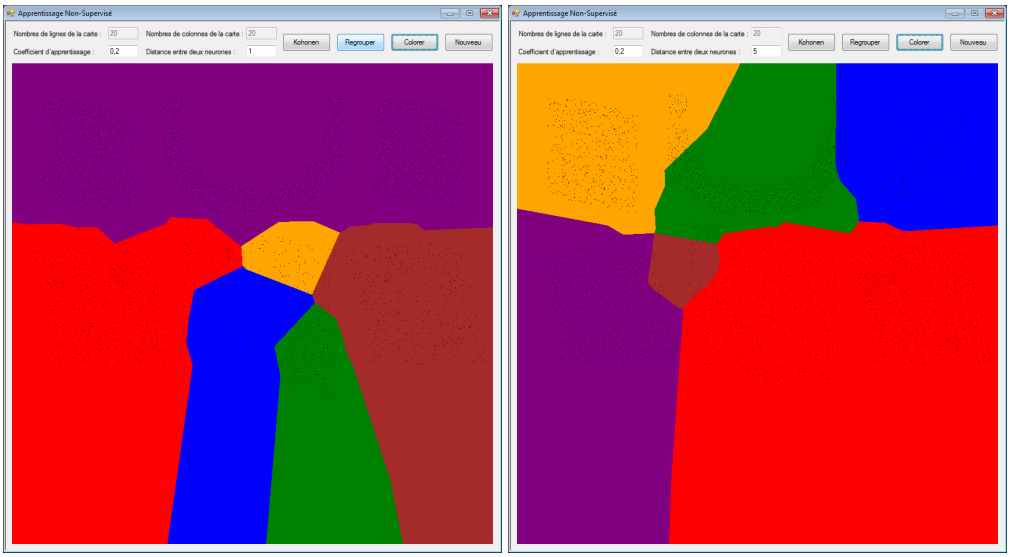



FIGURE 20 – Variations sur le coefficient d'apprentissage

**Commentaires**

Le nombre de classes reste constant (= 6) car il est difficile de dire si la classification est bonne ou mauvaise. Par exemple, pour 2 classes, est-ce jugé convenable si les 3 paquets du haut sont ensemble et les 3 du bas également, ou faut-il que l'écran soit coupé dans la diagonale, ou faut-il mettre 4 paquets à droite et 2 à gauche ? Il a donc été jugé préférable de ne se concentrer uniquement que sur la détermination de 6 classes.

***Performances d'apprentissage***

Les performances d'apprentissage ont été évaluées grâce au nombre d'itérations faites pour avoir une classification acceptable (vitesse de convergence). Le pourcentage de bonne et mauvaise classification donne une indication, mais elle n'est pas toujours fiable suivant le nombre de classes d'entrée.

Pour un coefficient d'apprentissage élevé, les performances sont moins bonnes car il faut plus d'itérations pour « stabiliser » la carte.

Pour une grande carte, c'est-à-dire une carte qui contient beaucoup de neurones, les performances ne sont pas non plus excellentes car peu de neurones sont gardés dans les classes (une dizaine pour 6 classes).

Si la distance est trop faible, les neurones ne se tirent pas assez et donc certains neurones n'arrivent pas à être

attirés dans une classe, mais comme du coup, ils ne gagnent jamais lors du regroupement, cela n'impacte pas vraiment les performances. Si la distance est trop forte, les neurones se tirent trop et sont tous attirés vers une même classe. Il faut donc mieux privilégier les faibles coefficients d'apprentissage : entre 1 et 3.

***Temps d'apprentissage***

Le temps d'apprentissage a été évalué au ressenti.

On peut ainsi remarquer que pour une carte plus grande, le temps de calcul est bien plus élevé.

Le coefficient d'apprentissage et la distance de voisinage n'ont pas l'air d'influencer beaucoup le temps d'apprentissage.

## 3. Répartition du travail

Nous nous sommes réparties le travail de la manière suivante :

| | Partie 1 | Partie 2 |
|---|---|---|
| Question 1 | Travail collectif | Camille |
| Question 2 | Travail collectif | Alexia |
| Question 3 | Travail collectif | Émilie |

Nous tenions à effectuer la première partie du projet toutes ensemble pour pouvoir confronter nos points de vue concernant l'architecture du code et de l'interface. Nous pensions qu'il était nécessaire de réfléchir toutes ensemble pour un meilleur résultat.

Pour la seconde partie du projet, nous nous sommes réparties chacune une question. Émilie ayant déjà connaissance des réseaux de neurones grâce à son projet informatique individuel, il a été décidé qu'elle ferait la question 3.

En revanche, à chaque séance, nous présentions le travail effectué pour avoir l'avis et les suggestions des camarades. Nous tenions également à bien comprendre le raisonnement pour être capable de reproduire le code.

Le rapport a été quant à lui rédigé par tout le monde.