Neural Computing

Paul Aimé, Morgane Gaillard

19 décembre 2019

Table des matières

1 From perceptron to artificial neurons : basic principles **1**
1.1 Principes du neurone artificiel . . . 1
1.2 Apprentissage dans un réseau de neurones artificiels . . . 2
1.3 Cas d'étude . . . 3
1.3.1 Présentation du problème . . . 3
1.3.2 Apprentissage . . . 3
1.3.3 Étude du comportement lors de l'apprentissage . . . 4

2 Self-Organizing Maps **6**
2.1 Principe . . . 6
2.2 Apprentissage . . . 6
2.3 Cas d'étude . . . 7

1 From perceptron to artificial neurons : basic principles

1.1 Principes du neurone artificiel

Comme base pour construire un neurone artificiel, nous prendrons une simplification d'un neurone biologique, qui peut être vu comme une entité intégrant les potentiels post-synaptiques des synapses qui lui sont connectées en amont, et générant une réponse binaire (un *spike* ou non) suivant si la valeur de cette intégration dépasse un certain seuil qui est propre au neurone.

On notera z le résultat de l'intégration des potentiels post-synaptiques des synapses qui lui sont connectées en amont, notés a_i, et on l'exprimera mathématiquement sous la forme :

$$z = \sum_{i=1}^{N} w_i a_i \tag{1}$$

où N est le nombre de neurones connectés au neurone considéré, et w_i est le poids de la synapse reliant le neurone considéré à son neurone parent indexé i, et est donc modélisé comme un coefficient multiplicateur appliqué au potentiel post-synaptique de ce neurone parent. La sortie a du neurone considérée est alors :

$$a = \mathcal{N}(z) \tag{2}$$

où $\mathcal{N}$ est une fonction non linéaire, appelée fonction d'activation, et est ici une fonction seuil traduisant le caractère binaire de la réponse du neurone considérée.

On peut alors dire que le neurone réalise une classification binaire sur la valeur de z, selon qu'elle soit supérieure ou non au seuil du neurone, que l'on appellera b (comme *biais*). Réaliser un apprentissage sur un tel neurone revient alors à trouver les bons poids w_i, et le bon biais b, tels que la classification binaire dans l'espace $\mathbb{R}$ de son entrée z corresponde à une bonne classification dans l'espace $\mathbb{R}^N$ des N neurones qui lui sont parents. Avec une telle fonction de seuil, la réponse du neurone peut être explicitée comme suit :

$$\begin{cases} a = 1 & , \text{si } z > b \\ a = 0 & , \text{si } z < b \end{cases} \tag{3}$$

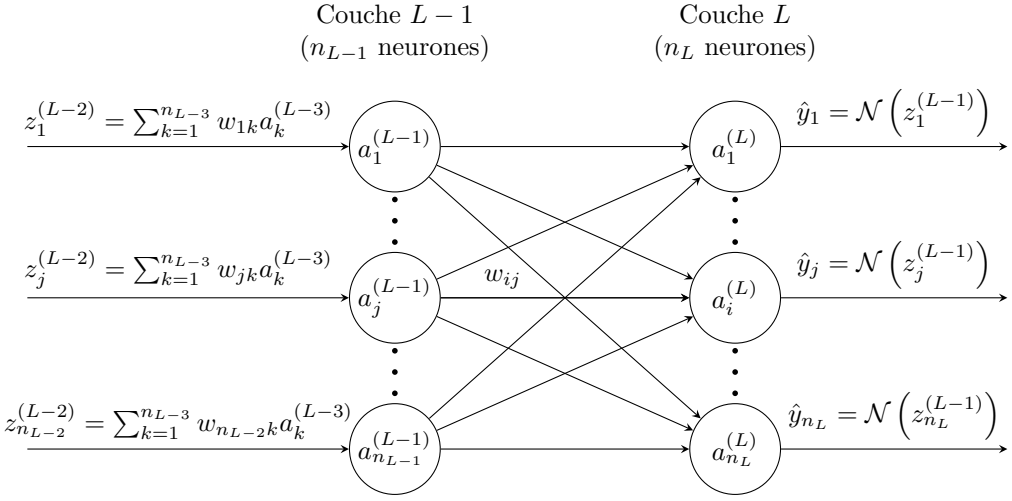
On peut alors reformuler l'inéquation qui conditionne l'activation du neurone de la manière suivante :

$$
\begin{array}{lll}
 & z > b & \\
\Leftrightarrow & \sum_{i=1}^{N} w_i a_i & > b \\
\Leftrightarrow & \sum_{i=1}^{N} w_i a_i - b & > 0 \\
\Leftrightarrow & \sum_{i=1}^{N+1} w_i a_i & > 0 \quad \text{, avec } a_{N+1} = -1 \text{ et } w_{N+1} = b
\end{array} \tag{4}
$$

Écrire l'équation de sortie du neurone de cette manière revient à ajouter un neurone parent avec un potentiel post-synaptique a_{N+1} toujours égale à -1. Cela permet de montrer que le paramètre b n'est pas fondamentalement différent des paramètres w_i, et donc que leurs apprentissages respectifs pourront alors se faire de la même manière. La différence subsiste dans le fait que, dans l'espace $\mathbb{R}^N$ des neurones parents, les poids w_i appliquent une rotation alors que le biais b réalise une translation (cf. figure 5a).

1.2 Apprentissage dans un réseau de neurones artificiels

Un apprentissage consiste à mettre à jour les paramètres d'apprentissage, et s'écrit donc tout simplement sous la forme : $w_{ij}^{(t+1)} \leftarrow w_{ij}^{(t)} + \Delta w_{ij}^{(t)}$. Il existe ensuite de nombreuses manière de décider de la manière pertinente de calculer $\Delta w_{ij}^{(t)}$ la quantité de mise à jour des paramètres d'apprentissage.



Dans le cas d'un apprentissage supervisé, nous disposons des sorties désirées, et nous pouvons donc nous servir d'une fonction de coût pour calculer l'erreur entre la sortie prédite et la sortie désirée. Là encore il existe plusieurs fonctions de coût, qui doivent être choisies de manière pertinente en fonction des données et de l'architecture du modèle.

Puisque l'on dispose d'une fonction de coût ($\mathcal{L} : (y, \hat{y}) \mapsto \mathcal{L}(y, \hat{y})$), nous pouvons alors nous servir de la méthode de la descente de gradient, particulièrement adaptée pour les réseaux de neurones. On a alors :

$$
\Delta w_{ij} = -\mu \times \frac{\partial \mathcal{L}(y, \hat{y})}{\partial w_{ij}} \quad \text{avec } \mu \text{ le taux d'apprentissage.} \tag{5}
$$

Il reste à calculer ce gradient, ce que l'on réalise en utilisant la règle de la dérivation en chaîne :

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_{ij}} &= \frac{\partial \mathcal{L}}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_i^{(L)}} \cdot \frac{\partial z_i^{(L)}}{\partial w_{ij}} \\
&= \frac{\partial \mathcal{L}}{\partial \hat{y}_i} \mathcal{N}'\left(z_i^{(L)}\right) \cdot \sum_{j'=1}^{N_{L-1}} \frac{\partial}{\partial w_{ij}} w_{ij'} a_{j'}^{(L-1)} \\
\boxed{\frac{\partial \mathcal{L}}{\partial w_{ij}} &= \frac{\partial \mathcal{L}}{\partial \hat{y}_i} \mathcal{N}'\left(z_i^{(L)}\right) a_j^{(L-1)}}
\end{aligned}
$$

On notera que puisque nous utilisons la méthode de la descente de gradient, alors en plus d'être non-linéaire et bornée, la fonction d'activation $\mathcal{N}$ se doit d'être dérivable. Nous remplaçons donc la fonction

de seuil précédente par une fonction sigmoïde. La sortie du neurone n'est alors plus binaire, mais est une valeur réelle comprise entre 0 et 1, avec une partie quasi-linéaire dans la partie où z est proche de zéro. Cette fonction est parfois appelée courbe d'apprentissage, et l'apprentissage se réalise dans cette partie linéaire, puisque les parties saturées témoigne d'un apprentissage "fini".

1.3 Cas d'étude

1.3.1 Présentation du problème

Nous prendrons le cas d'étude d'une classification binaire de tirages gaussiens dans l'espace 2D. Nous avons alors deux classes, qui peuvent être représentées dans un espace 2D. Nous prenons un dataset de 100 points par classe, donc 200 points au total. Nous découpons ensuite classiquement ce dataset en un set pour réaliser l'apprentissage, et un set pour réaliser le test. Le set d'apprentissage est lui même découpé en un set d'apprentissage à proprement parler, et un set de validation pour suivre une estimation des capacités de généralisation du réseau de neurones.

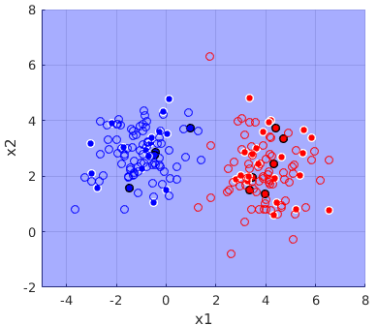
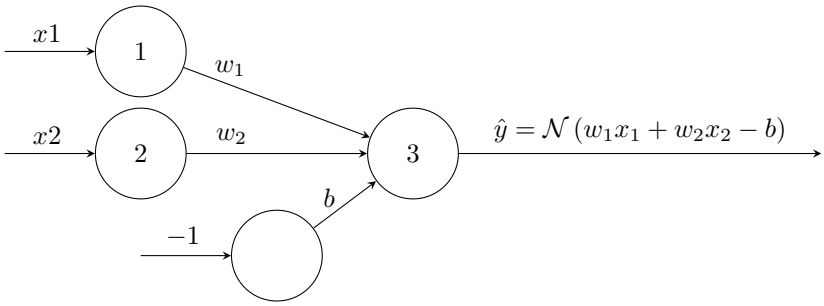


FIGURE 1 – Dataset. Tirage de gaussiennes de variance identité et de moyennes respectives $(4, 2)$ et $(-1, 3)$. *train set* → cercles vides ; *validation set* → disques pleins cerclés de blanc ; *test set* → disques pleins cerclés de noir.

Le *test set* comprend 5% des points, le *validation set* comprend 20% des 95% de points restants. Avec 200 points au total, il y a donc 152 points dans le *train set*, 38 dans le *validation set* et 10 dans le *test set*. Un exemple de tirage est visible sur la figure 1.

1.3.2 Apprentissage



Comme mentionné plus tôt, le choix de la fonction de coût $\mathcal{L}$ est important et doit être cohérent avec l'encodage et l'architecture du problème. Dans notre cas nous réalisons une classification binaire, et la sortie désirée est encodée sur un unique neurone par un label égal soit à 0 soit à 1. Nous utiliserons alors la fonction de coût entropie croisée binaire, et nous discuterons de sa pertinence.

$$\mathcal{L}(\hat{y}, y) = -\left(y \times log(\hat{y}) + (1-y) \times log(1-\hat{y})\right) \tag{6}$$

On observe qu'il est possible de réécrire cette fonction de la manière suivante, afin de mieux cerner son comportement :

$$\begin{cases} \mathcal{L}(\hat{y}, y) = -log(\hat{y}) & \text{, si } y = 1 \\ \mathcal{L}(\hat{y}, y) = -log(1-\hat{y}) & \text{, si } y = 0 \end{cases} \tag{7}$$

ou encore :

$$\mathcal{L}(\hat{y}, y) = -log(1 - |\hat{y} - y|) \tag{8}$$

On voit alors que la fonction de coût considérée renvoie naturellement une fonction de la distance absolue à la prédiction. Comme on peut le voir sur la figure 2, c'est une fonction croissante qui pénalise d'autant plus l'écart que celui ci est grand. Cela est un comportement souhaitable, dans le sens où lorsque l'erreur de prédiction est grande il est logique de se corriger d'autant plus, alors que lorsque l'erreur est faible la correction est moins nécessaire, et en tout cas doit être faite de manière plus précautionneuse.

En effet dans le cas limite où l'architecture du modèle et son nombre de paramètres libres ne permettent pas une séparation parfaite des données, alors de petites erreurs seront donc inévitables. Donner alors trop d'importance à la correction face à un point compris dans cette zone de non séparabilité (cf. figure 4) rendrait alors le système instable, puisque ce faisant il agrandirait alors l'erreur sur d'autres points du jeu de données, qu'il essaiera de corriger par la suite et ainsi de suite.

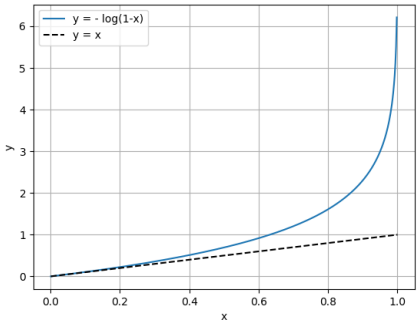


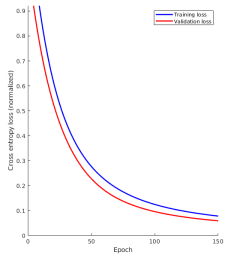
FIGURE 2 – $y = -log(1-x)$ pour $x \in]0, 1]$

Pour les mêmes raisons il est donc également primordial de choisir un taux d'apprentissage adéquat, ainsi qu'une taille de batch adaptée. En effet le problème cité précédemment advient lorsque n'est présenté au réseau de neurone qu'une partie du jeu de données, et que la mise à jour des poids pourrait alors entraîner une régression de performance sur le restant des données si celles si n'ont pas exactement la même distribution. Ce problème est donc amplifié par un taux d'apprentissage trop grand, ou par une taille de batch trop petite qui augmente les chances de séparer les données en plusieurs jeux de données aux distributions différentes (cf figure 5). Par ailleurs si l'on pourrait penser que ce problème ne se pose pas si le batch contient toutes les données, ce n'est en réalité jamais le cas si l'on considère l'existence de futurs points de test ; et la diminution de la taille du batch possède d'autres avantages non discutés ici, notamment celui de permettre au réseau de s'échapper de minima locaux.

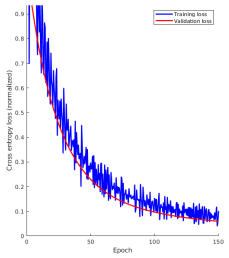
1.3.3 Étude du comportement lors de l'apprentissage

Instabilité de l'apprentissage

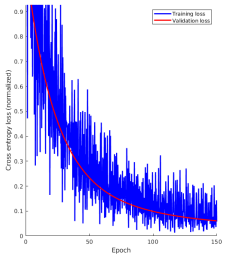
Comme développé ci-avant, les paramètres de taille du batch et de taux d'apprentissage peuvent rendre le système instable, en contrepartie du fait qu'il peuvent respectivement aider à sortir de minima locaux ou rendre l'apprentissage plus rapide. On retrouve alors ces effets dans la figure 5, sans plus de commentaires que ceux fait précédemment. On peux tout de même rajouter que le fait que la courbe de validation soit lisse indique que le jeu de validation est bien issue de la même distribution que le jeu d'apprentissage.



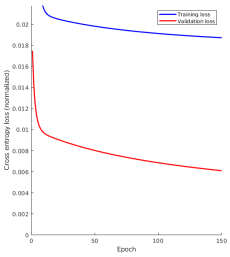
(a) `batch_ratio=1, lr=1e-4`



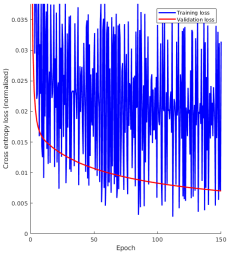
(b) `batch_ratio=.5, lr=1e-4`



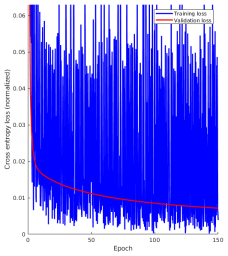
(c) `batch_ratio=.2, lr=1e-4`



(d) `batch_ratio=1, lr=1e-2`



(e) `batch_ratio=.5, lr=1e-2`



(f) `batch_ratio=.2, lr=1e-2`

FIGURE 3 – Comportement de l'apprentissage pour différentes valeurs de taille du batch et de taux d'apprentissage. Entropie croisée binaire normalisée par le nombre de points. 2 classes de distribution gaussiennes de variance identité et de moyennes respectives $(4, 2)$ et $(-1, 3)$. Courbe d'apprentissage en bleu, courbe de validation en rouge. Selon la taille du batch, la courbe d'apprentissage n'a pas le même nombre de points que celle de validation.

Cas de données non séparables

Il est possible que le nombre de paramètres libres, l'architecture du modèle, ou la nature des données ne permettent pas une séparation parfaite. C'est notre cas ici lorsque l'on choisit des gaussiennes qui se chevauchent (cf. figure 4), puisque, ne possédant pas de couche cachée, notre réseau de neurone ne peut tracer que des frontières linéaire dans l'espace 2D d'entrée. Une telle situation peut ajouter à l'instabilité du réseau, comme discuté plus tôt, notamment en cas de faible valeur de taille du batch.

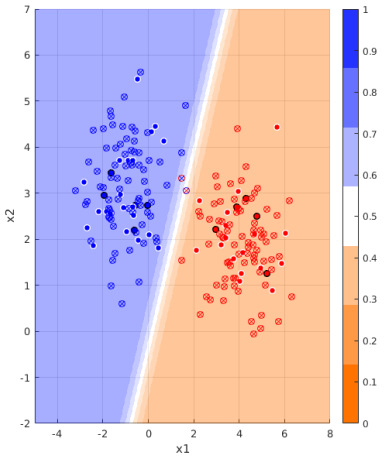
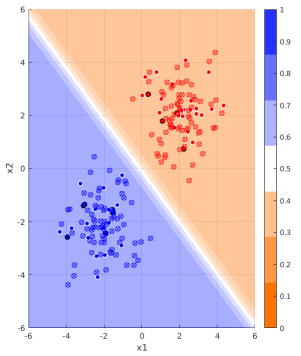


FIGURE 4 – Cas de données non séparables par le modèle

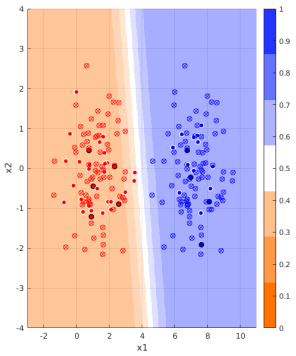
Cas de modèle sur-paramétré

A l'inverse, il est également possible que le modèle soit sur-paramétré au regard du jeu de données qu'il a à traiter. Dans notre exemple, deux cas spécifiques peuvent amener à cette situation.

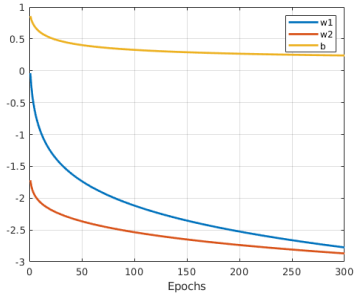
Tout d'abord lorsque les données ont pour point de symétrie l'origine (cf. figure 5a), alors le biais devient théoriquement inutile. En effet la frontière idéale passe alors par l'origine. Ainsi, si nous avons connaissance à priori de cette caractéristique du jeu de donnée, il est préférable de ne pas utiliser de biais, qui pourrait alors ralentir l'apprentissage. Dans le cas de la figure 5a), nous avons construit la classe 1 comme explicitement symétrique à la classe 2 par rapport à l'origine ($X2 = -X1$). La frontière parfaite parcimonieuse est donc pour $w_1 = w_2$ et $b = 0$, c'est le comportement que l'on observe sur la figure 5c. On note que cette équation possède une infinité de solutions, celle trouvée par le réseau est ici pour $w_1 \approx w_2 \approx -3$.



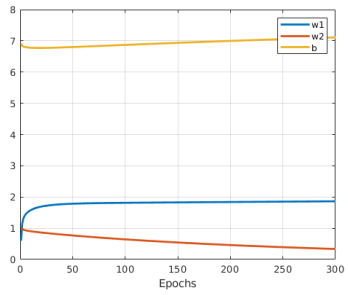
(a) `X2=X1`



(b) `X2=X1; X2(:,1)=X1(:,1)+4+mu1(1,1)`



(c) Évolution des poids pour les données 5a



(d) Évolution des poids pour les données 5b

FIGURE 5 – Exemple d'évolution des paramètres d'apprentissage pour des jeux de données rendant explicitement le modèle sur-paramétré.

Le deuxième cas possible, présenté dans la figure 5b, est celui pour lequel la séparation des données est résolvable dans un espace qui est une simple projection othogonale de l'espace initial. C'est le cas ici où les données sont explicitement générées de sorte à ce que la classe 1 et la classe 2 soient parfaitement séparables et symétriques par rapport à la droite d'équation $x1 = 4$. Dans ce cas l'entrée x_2 est inutile à la classification, et la solution théorique la plus simple est $w_2 = 0$, $w_1 = 1$, $b = 4$. On voit sur la figure 5d que ce n'est pas le cas, et que le réseau a trouvé une autre combinaison parmi l'infinité rendue possible par la sur-paramétrisation du problème. Encore une fois cela peut ralentir l'apprentissage, voire le faire échouer dans des cas plus complexes. On notera que ces défauts de comportement d'apprentissage

peuvent en partie être résolus en introduisant un terme de régularisation, de telle sorte que la solution parcimonieuse soit préférée.

Notes :

- Pour un cas simple comme celui-ci, l'étude du comportement lors que l'apprentissage a été préférée à la présentation de métriques sur la base test.
- Le code (MATLAB) est disponible sur GitHub, et permet notamment de visualiser l'évolution de la segmentation de l'espace d'entrée en temps réel. Un GIF animé est d'ailleurs présent sur la page d'accueil. Le code est laissé à la libre utilisation.

2 Self-Organizing Maps

Dans cette section nous nous intéressons à une architecture de réseau de neurone permettant un apprentissage non-supervisé, en l'état des cartes auto-organisatrices de Kohonen [Koh90].

2.1 Principe

L'idée est de réaliser une projection de l'espace des entrées dans un espace réduit, représenté par une grille (le plus souvent rectangulaire) de neurones. Le but est que des entrées similaires activent préférentiellement la même zone de la carte, en ce sens la carte est alors bien une représentation de l'espace d'entrée.

Une composante essentielle est donc la préservation de la notion de proximité entre l'espace d'entrée est celui de la carte de neurones. Pour ce faire, la notion de proximité est transcrite au niveau de la carte par une co-activation des neurones proches. Cette propriété, couplée à celle d'apprentissage non supervisée, rend cette architecture de neurone particulièrement intéressante dans le sens où cela lui confère une compatibilité bien plus grande avec les réseaux de neurones biologiques que n'en ont les réseau de neurones de type supervisés à apprentissage par gradient de descente.

2.2 Apprentissage

La carte de neurones est de dimension $d1 \times d2$, et tous les neurones sont connectés à chacun des p neurones représentant l'espace d'entrée. Ainsi les poids de ce réseau peuvent être représentés par une matrice $W \in \mathbb{R}^{p \times d1 d2}$, où les colonnes correspondent donc aux poids de chaque neurone avec les entrées, et les lignes aux poids de chaque entrée avec l'ensemble des neurones de la carte.

Présentée à une entrée $x \in \mathbb{R}^p$, l'activation des neurones de la carte est alors $y = W^T x$. L'idée de l'apprentissage est de modifier le vecteur $w_{BMU} \in \mathbb{R}^p$ des poids du neurone le plus activé (BMU) pour le rapprocher du vecteur $x \in \mathbb{R}^p$ qui représente cette entrée, et dans une moindre mesure de faire de même pour les neurones voisins. La fonction de mise à jour des poids est alors :

$$w_k^{(t+1)} = w_k^{(t)} + \alpha_t \cdot \theta_t(i,k) \cdot (x - w_k^{(t)}) \tag{9}$$

où α_t est le taux d'apprentissage et $\theta_t(i,k)$ est la fonction de voisinage.

La fonction de voisinage $\theta_t(i,k)$ est alors primordiale pour le bon déroulement de l'apprentissage. Elle doit être maximale pour $i = k$, et décroissante en fonction de $\|c_i - c_k\|$ la distance sur la grille entre les neurones i et k. De plus, afin que l'apprentissage converge, il faut que la notion de voisinage soit décroissante en fonction du temps pour une distance donnée. On l'écrit alors de la manière suivante :

$$\theta_t(i,k) = \exp\left(-h(t)\,\|c_i - c_k\|\right) \tag{10}$$

où $h(t)$ est alors une fonction croissante du temps qui contrôle la notion de voisinage (cf. figure 6). Pour un apprentissage sur un nombre d'époque T_{max}, on prend la fonction $h(t)$ suivante :

$$h(t) = \frac{1}{2 \times \left(\sigma_0 e^{-\frac{t}{T_{max}}}\right)^2} \tag{11}$$

de sorte à ce que $h(T_{max}, \sigma_0) = h(\sigma_0)$, une constante par rapport à T_{max}. Ainsi σ_0 devient le paramètre de contrôle de la dynamique de voisinage au cours du temps. Plus σ_0 est petit et plus le voisinage se ressert vite au cours du temps.

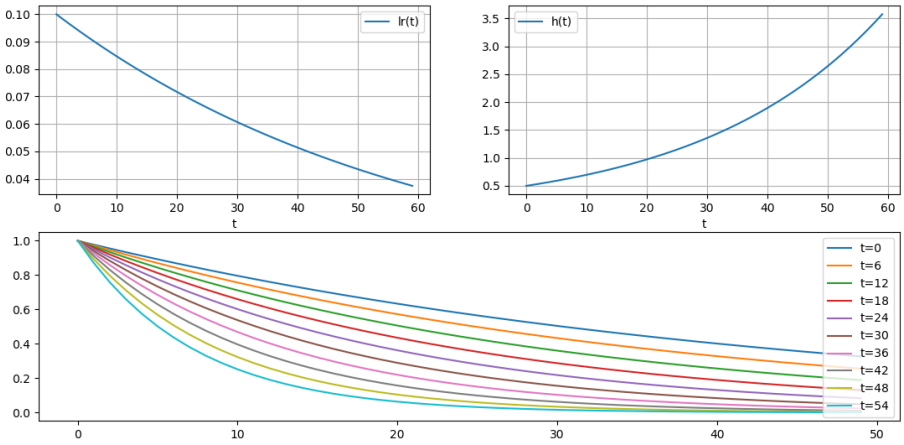


FIGURE 6 – Evolution des paramètres d'apprentissage. Taux d'apprentissage $\alpha(t)$ en haut à gauche, paramètre $h(t)$ de réduction de la taille du voisinage en haut à droite, et fonction de voisinage $\theta_t(d)$ en bas.

Finalement, pour atteindre une convergence, le pas d'apprentissage est aussi une fonction décroissante du temps, ici $\alpha_t = \alpha_0 \cdot e^{-t/T_{max}}$ (cf. figure 6). On observe sur la figure 6 la dynamique de ces différents coefficients et fonctions qui assurent le bon déroulement de l'apprentissage.

2.3 Cas d'étude

Nous étudierons l'application d'une carte auto-organisatrice au dataset "wine" de la libraire python scikit-learn [Ped+11]. Ce dataset comprend des données de dimensions 13 (dont les significations sont visibles sur la légende de la figure 7) réparties en 3 classes, avec un total de 178 exemples (59 pour la classe 0, 71 pour la classe 1, 48 pour la classe 2). Un apprentissage a été réalisé avec les paramètres $(d1, d2) = (2, 3)$, de telle sorte à réaliser une réduction de dimension, $\alpha_0 = 0.1$, $\sigma_0 = 1$, et $T_{max} = 60$.

Après l'apprentissage, on peut visualiser à quelle position de l'espace $\mathbb{R}^p$ d'entrée se retrouvent associés chacun des neurones de la carte (cf. figure 7). Dans notre cas, on observe que tous les neurones se trouvent très liés à la dimension `nonflavanoid_phenols` (les données sont bien normalisées), et que globalement il semblent se ressembler. Cela peut venir de la petite taille de la carte, qui fait que la notion de proximité, si primordiale, n'admet ici pas beaucoup de variation. Toutefois les neurones $(1, 3)$ et $(2, 3)$ se démarquent par le fait d'être également fortement positionné sur la dimension `hue`, et se démarque en plus l'un de l'autre sur la dimension `color_intensity`. Ainsi on peut s'attendre à ce que ces neurones soient discriminants.

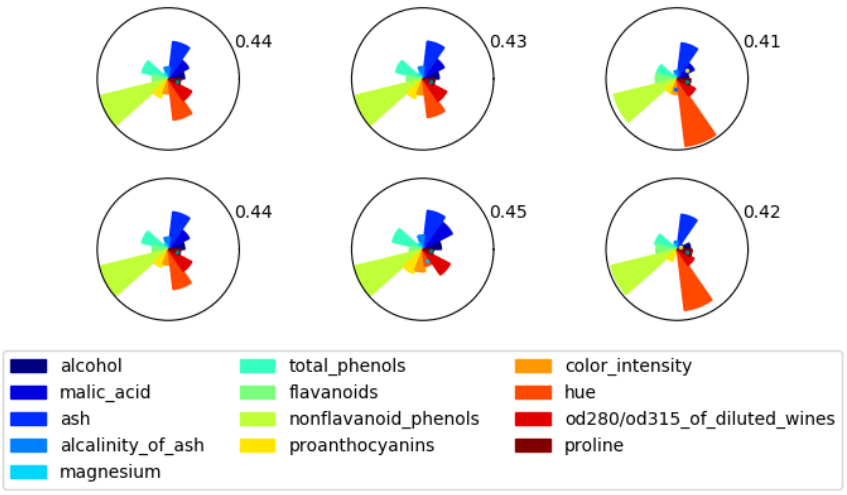


FIGURE 7 – Représentation des coordonnées de chaque neurone dans l'espace des entrées. Les valeurs des coordonnées sont représentées par les aires, et la valeur affichée est celle maximale. Un point de couleur indique une valeur négative.

Ce réseau de neurones ainsi formé nous permet alors de représenter les exemples d'entrée dans l'espace $\mathbb{R}^{d1d2}$ de la carte (cf. figure 8). Ici ce mapping est réalisé par simple attribution au neurone de maximum d'activation. On observe que, comme prédit, les neurones $(1, 3)$ et $(2, 3)$ sont les plus discriminants. Le neurone $(2, 3)$ semble être capable de discerner la classe 0, et les neurones $(1, 3)$ et $(2, 2)$ se chargent de discriminer les deux classes restantes, avec une petite ambiguïté. Les autres neurones ne sont quasiment pas utilisés, ce qui est plutôt pratique puisque le jeu de données comporte 3 classes.

De manière générale, puisque la propriété de la carte est de conserver la proximité de l'espace d'entrée dans l'espace de la carte, le clustering des points est alors sensé être rendu plus aisé par projection des données dans une carte auto-organisatrice entraînée. Pour réaliser ce clustering, des algorithmes classiques peuvent être utilisés.

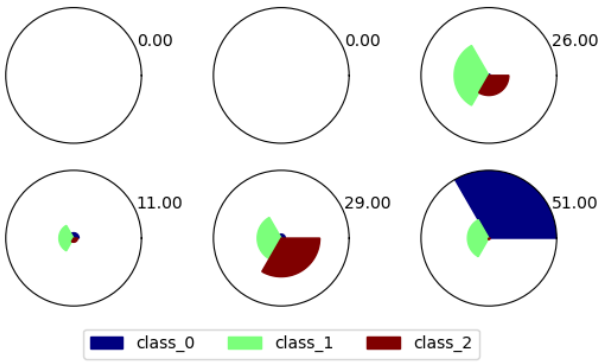


FIGURE 8 – Mapping des exemples sur leur neurone de plus forte activité. Le rayon représente le nombre d'exemples associés. Le chiffre correspond au nombre d'exemples de la classe majoritairement associée à chaque neurone.

Note

— Le code (python) est disponible sur GitHub, et contient notamment la fonction de visualisation de la matrice des poids. Le code est laissé à la libre utilisation.

Références

[Koh90] T. KOHONEN. "The self-organizing map". In : *Proceedings of the IEEE* 78.9 (sept. 1990), p. 1464-1480. ISSN : 1558-2256. DOI : 10.1109/5.58325. URL : https://sci2s.ugr.es/keel/pdf/algorithm/articulo/1990-Kohonen-PIEEE.pdf.

[Ped+11] F. PEDREGOSA et al. "Scikit-learn : Machine Learning in Python". In : *Journal of Machine Learning Research* 12 (2011), p. 2825-2830.