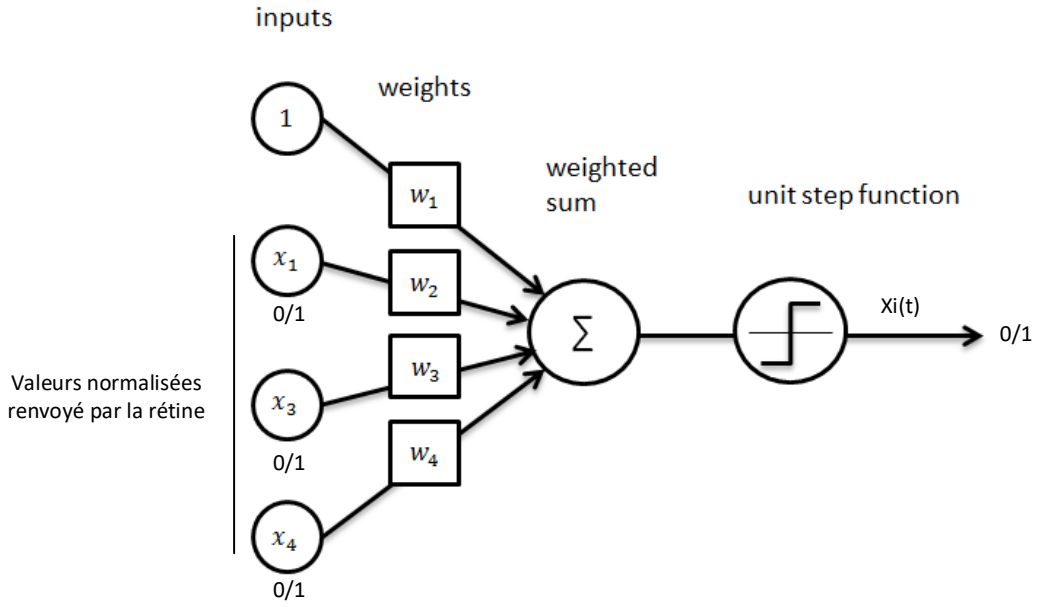
Question préliminaires



- On va coder le réseau en utilisant une fonction de propagation qui prendra en entrée le vecteur produit par la rétine et un autre vecteur représentant les poids des synapses et du biais. Elle renverra la valeur de classification : 1.0 (représentant la classe A) ou 0.0 (représentant la classe C).
- On va initialiser les poids du réseau en leur donnant une valeur réelle aléatoire entre -1 et 1. On ne veut pas donner une valeur trop écartée du poids. Cependant ce n'est pas la seule ou la meilleure façon, il existe plusieurs manières de procédé (déterministe ou non).
- La propagation de l'information s'effectuera de manière unidirectionnelle grâce à une fonction qui calculera la valeur de sortie des neurones de sorties en fonction de ces entrées, poids respectifs, biais, le tout pondéré avec la fonction d'activation.
- A chaque apprentissage (ou époché) les poids varieront en fonction de la valeur de sortie par rapport à celle attendu, la valeur du neurone d'entrée respectif et un coefficient d'apprentissage.
- L'apprentissage va continuer jusqu'à ce qu'il n'y est plus d'erreur (l'erreur quadratique étant binaire ici et non continue). Une fois l'erreur nulle les données sont correctement apprises (les données de départ seulement, la généralisation n'étant pas certaine). Si on alterne entre

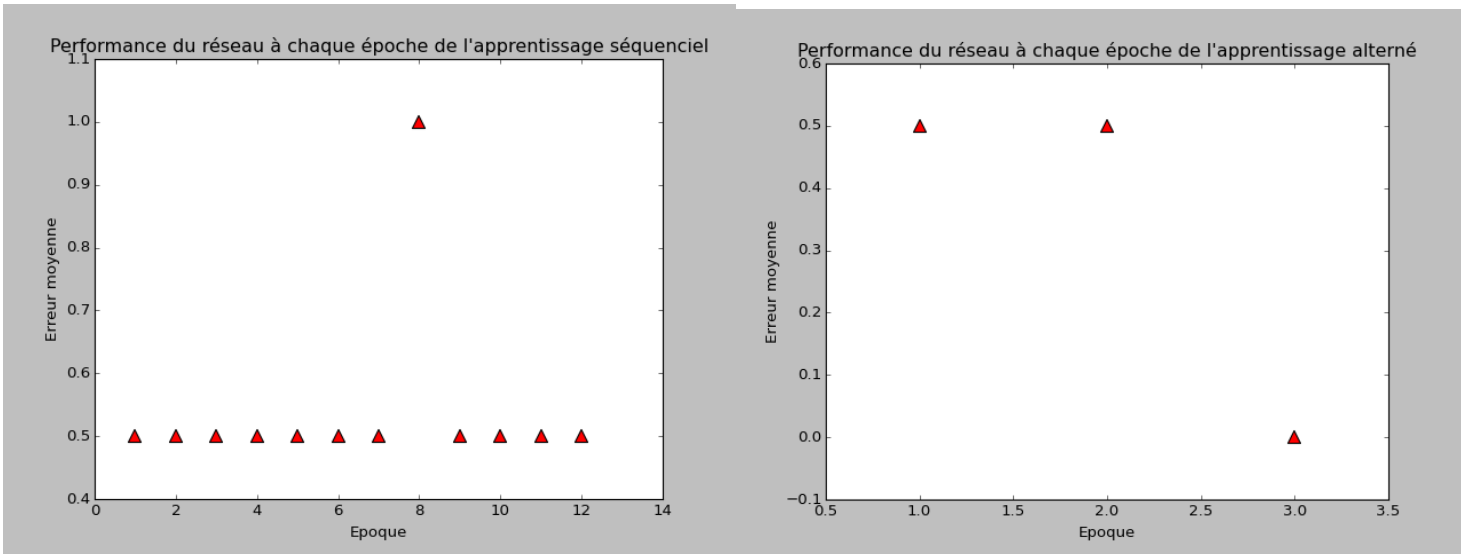
l'apprentissage de nos 2 motifs alors on doit s'assurer qu'il n'y est plus d'erreur sur ces 2.
Une fois l'erreur nulle cela ne sert de toute manière à rien de continuer car la fonction d'apprentissage ne changera rien (donc de toute manière pas de risque de sur-apprentissage).

Déroulement

5) Pour suivre l'évolution du réseau on pourrait tester aussi les capacités de généralisations du réseau pour savoir quand il fait le moins d'erreur face à des entrées inconnues.

On calcul l'erreur en testant les deux motifs et en fessant la moyenne de l'erreur quadratique des 2.

6)

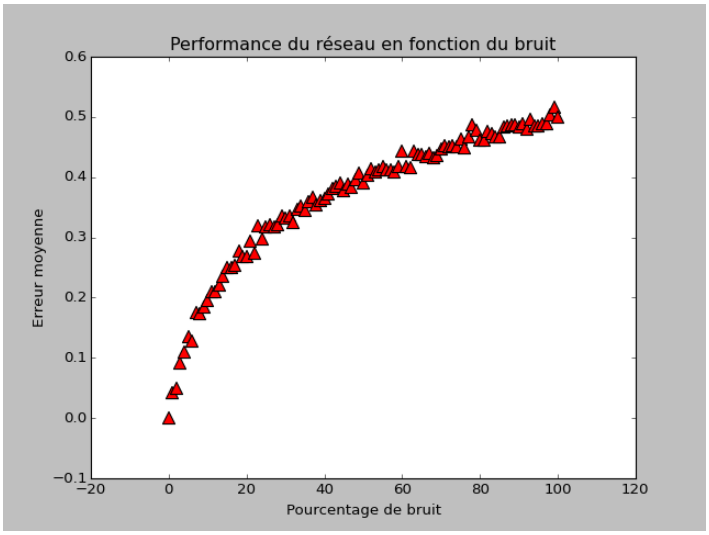


Il y a plusieurs méthodes d'apprentissages différentes et elles n'ont pas les mêmes avantages et inconvénients. Ici on a implémenté un apprentissage séquentiel (A puis C) et alterné (A et C, A et C,…).

On se rend rapidement compte que l'apprentissage alterné est quasiment toujours plus performant car il assure que l'erreur du réseau soit nulle après l'apprentissage complet de A et C en même temps. Ce qui n'est pas nécessairement le cas pour l'autre méthode, ici on remarque que l'erreur du réseau n'atteindra jamais 0, même après la fin d'apprentissage de C.

On peut notamment expliquer sa par le fait que une fois A à était appris, l'apprentissage de C peut entrainer une régression de l'apprentissage précédent.

7) On utilisera ici les poids calculés avec l'apprentissage alterné car on a vu plus haut qu'ils étaient bien plus performants.



L'erreur du réseau en fonction du bruit suit globalement une courbe logarithmique avec une augmentation rapide du taux d'erreur : avec seulement 20% de bruit on obtient en moyenne 0,3 erreur (le maximum étant aux alentours de 0,5 erreur).

Pour commencer on définit un pourcentage de bruit comme : la probabilité que chacun des bits d'une image soit transformé aléatoirement. Ainsi, un bruit de 100% revient à créer une image complètement aléatoire et un bruit de 0% est l'image de départ.

Ensuite pour chaque pourcentage de bruit on à effectuer 1000 itérations avec des entrées bruitées différentes pour approximer au mieux l'erreur moyenne correspondante. Cependant une amélioration pour avoir des résultats plus fiable consisterais a aussi recalculer la phase d'apprentissage avec des poids initiaux différents (cela n'a pas était fait car j'ai mis des affichages 'print' à l'intérieur de la fonction d'apprentissage).

Question de compréhension

1)

Exemple de poids finaux :

[0.9377768358108167, 0.5325030310409793, 0.6398204688697311, 0.7234894426713578, -0.4618593411356854, -0.542198494527805, 0.7438242523486243, -0.60308791333005, -0.7063341699672963, 0.14957173340516183, 0.14462130974254006, 0.1421310457374919, 0.018822568782338678, -0.22598868597412225, 0.9327402293488884, -0.20924382888199666, -0.17617301057827883, -1.0819966055992805, 0.4197777247151576, -1.020369437498306, 0.2733474527674936]

➔ (reconstruction en 2d)

[[0.9377768358108167, 0.5325030310409793, 0.6398204688697311, 0.7234894426713578, -0.4618593411356854],
[-0.542198494527805, 0.7438242523486243, -0.60308791333005, -0.7063341699672963, 0.14957173340516183],
[0.14462130974254006, 0.1421310457374919, 0.018822568782338678, -0.22598868597412225, 0.9327402293488884],
[-0.20924382888199666, -0.17617301057827883, -1.0819966055992805, 0.4197777247151576, -1.020369437498306]]
Biais :0.2733474527674936

On peut difficilement justifier avec précision pourquoi ces nombres exactes constituent les poids de la matrice, principalement parce que l'initialisation des poids n'est pas déterministe et n'est pas la même pour chaque poids. Mais on peut cependant percevoir une tendance : les poids les plus élevés sont souvent sur les bits positifs du A (mais pas du C), les poids intermédiaires sur les bits positifs de A et de C ou alors neutre ; les poids les plus faible sur les bits positifs de C (mais pas du A).

Cependant cela est loin d'être une règle absolu, mais cela partait assez logique étant donné que la sortie A est notamment créer par la présence de nombres positifs (voir la fonction d'activation) et la sortie de C, elle par des nombres négatifs.

On peut conclure que l'apprentissage ici consiste à assigner aux poids qui sont le plus corrélés aux motifs, la valeur correspondante en fonction : des autres poids, du biais et de la fonction d'activation.

2)

Dans la théorie, la technique d'apprentissage par perceptron n'a pas vocation à généraliser des cas qui ont était translaté ou subit une rotation. La matrice apprise est strictement lié à des pixels précis. On considère donc que le réseau ne généralise pas du tout ce genre de cas.

Dans la pratique cela peut quand même parfois marcher, notamment si des lettres ont des pixels qui sont 'allumés' dans la même zone.

En général on utilisera des réseaux plus complexes pour traiter ce genre de problèmes : perceptrons multicouches, réseaux de convolutions,…

3)

Pour résoudre ce problème (si l'on souhaite rester avec l'utilisation de perceptrons seulement), on pourrait utiliser 26 perceptrons indépendants reliés chacun aux mêmes entrées (la rétine). Pour chaque lettre en entrée, on aura qu'un seul des perceptrons qui doit s'activer (renvoyer un signal 1.0), ce perceptron correspond ainsi à la lettre correspondante en entrée.

Techniquement cela reviens à apprendre à chaque perceptron si l'entrée correspond à sa lettre correspondante ou pas.

On pourrait aussi renvoyer une valeur continue plutôt qu'un booléen pour assigner une probabilité à chaque entré de correspondre à tel ou tel lettre.