Les réseaux de neurones

M1 MIAGE *Machine Learning & Applications*

Stéphane Airiau

LAMSADE

DAUPHINE
UNIVERSITÉ PARIS

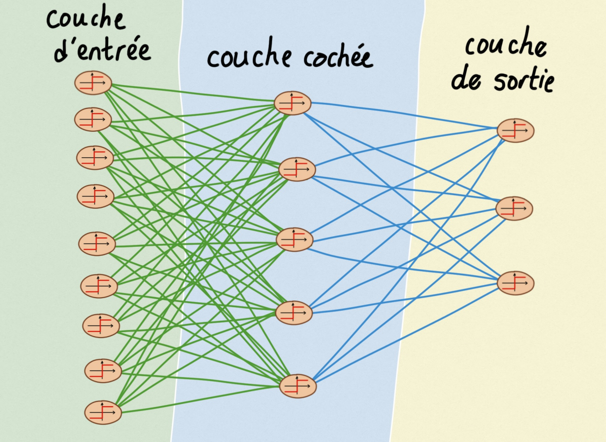
Passer au **réseau** de neurones!

Le **deep** learning consiste à avoir beaucoup de couches de neurones (ex alphaGo utilise des réseaux de 13 couches).

On ne va pas étudier les algorithmes pour pouvoir apprendre avec de tels réseaux, mais on va étudier un cas plus simple avec trois couches :

- une couche de neurones d'entrée ➾ on considère cette couche comme l'interface avec le capteur / les données
- une couche cachée
- une couche de sortie ➾ cette couche nous donne la prédiction du réseau

Réseau de neurones "Feed Forward"



Passer au **réseau** de neurones!

Evidemment, chaque on associe un **poids** à chaque arête du réseau.

- comme pour le perceptron, chaque neurone d'entrée est reliée à un capteur ou une valeur en entrée avec un poids.
- chaque arête entre un neurone d'entrée et un neurone de la couche cachée a un poids
- chaque arête entre un neurone de la couche cachée et un neurone de la couche de sortie a un poids.

Topologie du réseau

On va faire l'hypothèse que

- chaque neurone de la couche cachée est connecté à **tous** les neurones de la couche d'entrée
- chaque neurone de la couche de sortie est connecté à **tous** les neurones de la couche cachée

➾ on aura des notations plus simple

enlever des connexions correspondrait à forcer un poids nul.

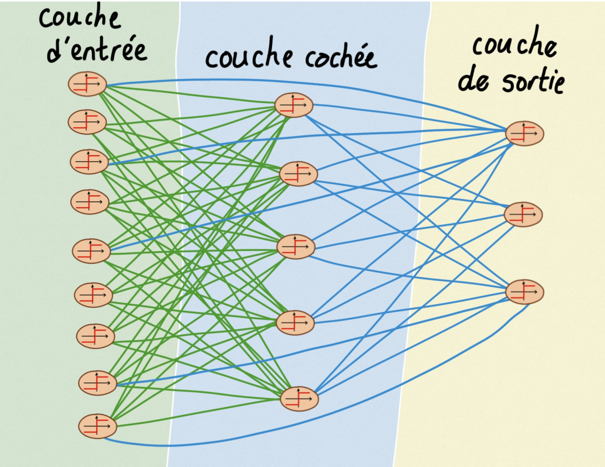
Ainsi, il n'y a pas de boucle dans le réseau : le "flot" d'information pourra seulement être

- couche d'entrée ➾ couche cachée ➾ couche de sortie
- couche de sortie ➾ couche cachée ➾ couche d'entrée

Nos algorithmes vont dépendre fortement de cette contrainte.

petite variante : on pourra aussi accepter que les neurones de la couche de sortie soient aussi directement connectés aux neurones d'entrée.

Réseau de neurone "Feed Forward"



On a ajouté quelques connexions entre la couche d'entrée et la couche de sortie.

NB. avec cette hypothèse, on pourrait aussi ajouter des couches cachées (les entrées d'un neurone d'une couche ne pouvant venir que des couches précédentes).

Exemple classique : base MNIST

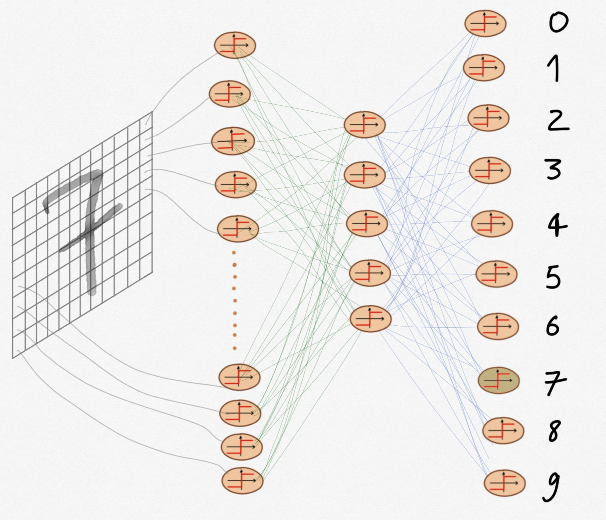
C'est une base de données d'images de chiffres écrits à la main.

- 60 000 images pour la base d'apprentissage
- 10 000 images pour la base de tests
- chaque image contient 28×28 pixels
- chaque pixel représente un niveau de gris
  (entiers entre 0 et 255, 0 est blanc, 255 est noir)

Un réseau de neurones pour apprendre à reconnaître chaque chiffre est :

- une couche d'entrée contenant $28 \times 28 = 784$ neurones
  chaque neurone est relié à un pixel de l'image
- une couche cachée contenant k neurones, chaque neurone est relié à chacun des 784 neurones de la couche d'entrée
- une couche de sortie qui contient 10 neurones
  chaque neurone représente un chiffre $0, 1, 2, \ldots, 9$.
  ➾ one hot encoding : si l'image représente un 7, le neurone 7 devrait avoir la valeur 1 en sortie, tous les autres neurones devraient avoir 0 en sortie.

Exemple de réseau pour la base MNIST



Obtenir une prédiction : propagation

Comment se passe le calcul de la prédiction du réseau ?

1- on part de la couche d'entrée ⇨ on propage le signal d'entrée

2- les neurones de la couche cachée prennent la valeur de sortie des neurones de la couche d'entrée et calculent chacun leur valeur de sortie

3- les neurones de sorties prennent pour entrée les valeurs de la couche cachée et calculent chacun leur sortie qui constitue la sortie du réseau de neurones

Vectorisation

Soit le vecteur ligne $a^{(k)}$ contenant les valeurs de chaque entrée pour l'exemple k : $a_1^{(k)}, a_2^{(k)}, \dots, a_n^{(k)}$

Soit le vecteur colonne $w^{(j)}$ contenant les poids du neurone j de la couche cachée : $w_1^{(j)}, w_2^{(j)}, \dots, w_n^{(j)}$

➯ le produit matriciel $a^{(k)} w^{(j)}$ nous donne la somme pondérée $\sum_{i=1}^{n} w_i^{(k)} a_i^{(j)}$

On rassemble le vecteur colonne de poids de chaque neurone en une seule matrice W_c

On rassemble le vecteur ligne de chaque exemple dans une matrice A_e

Le produit matriciel $A_e \times W_c$ nous donne la somme pondérée pour chaque neurone et chaque exemple

Soit f la fonction d'activation ➯ $A_c = f(A_e \times W_c)$ nous donne la valeur de sortie de chaque neurone pour chaque exemple.

➯ la valeur d'entrée pour la couche de sortie !

De même, on peut stocker dans une matrice W_s les vecteurs de poids des neurones de la couche de sortie.

$f(A_c \times W_s)$ est la valeur de sortie du réseau de neurones.

Au lieu d'utiliser des boucles pour faire nos calculs, on peut utiliser des produits matriciels
➡ pour réaliser ces calculs matriciels on peut utiliser

- des algorithmes optimisés
- du hardware spécialisé (carte graphiques sont faites pour faire du calcul matriciel !)

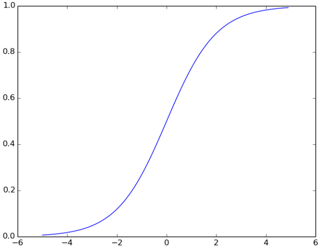
Pour l'algorithme d'apprentissage, on peut aussi l'écrire de façon vectoriel (pas dans les slides)
➡ permettra une exécution plus rapide.

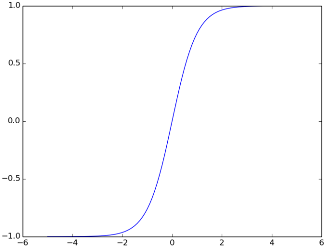
Fonction d'activation

- pour l'apprentissage – trouver la valeur de chaque poids – on va utiliser une méthode comme las descente de gradient
  ➾ on veut que la fonction d'activation soit dérivable
- On peut vouloir apprendre autre chose que des combinaisons de fonctions linéaires
  ➾ on peut utiliser des fonctions d'activation non linéaires qui se rapprochent de l'idée de la fonction seuil :
  - fonction logistique
  $$f : z \mapsto \frac{1}{1+e^{-z}}$$
  - fonction tangente hyperbolique tanh :
  $$\tanh z \mapsto \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Fonction d'activation

Fonction logistique	fonction tanh
$f : z \mapsto \frac{1}{1+e^{-z}}$	$\tanh : z \mapsto \frac{e^z - e^{-z}}{e^z + e^{-z}}$





Fonction d'erreurs

Alors qu'on avait une seule sortie pour le perceptron, on a maintenant une sortie pour chaque neurone de la couche de sortie.

Il suffit donc de sommer les erreurs sur chacune des sorties

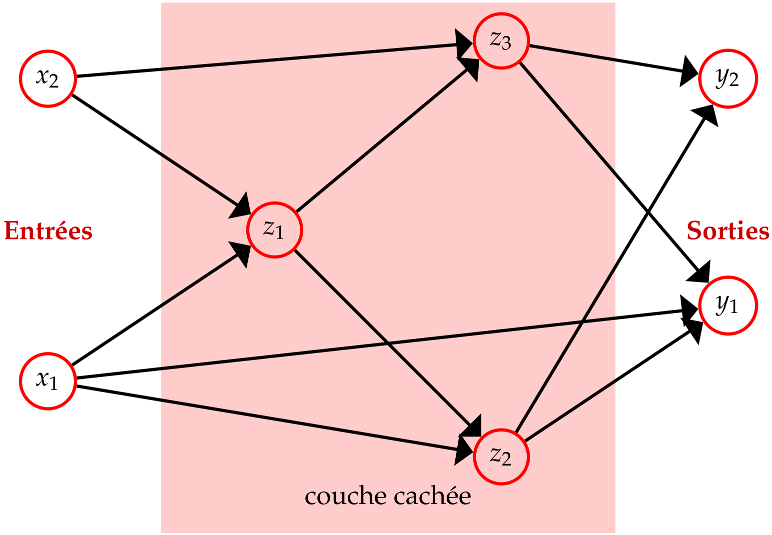
L'erreur d'apprentissage est mesurée par

$$E(\overrightarrow{w}) = \frac{1}{2}\sum_{d\in D}\sum_{k\in sorties}(t_{k,d} - o_{k,d})^2$$

où

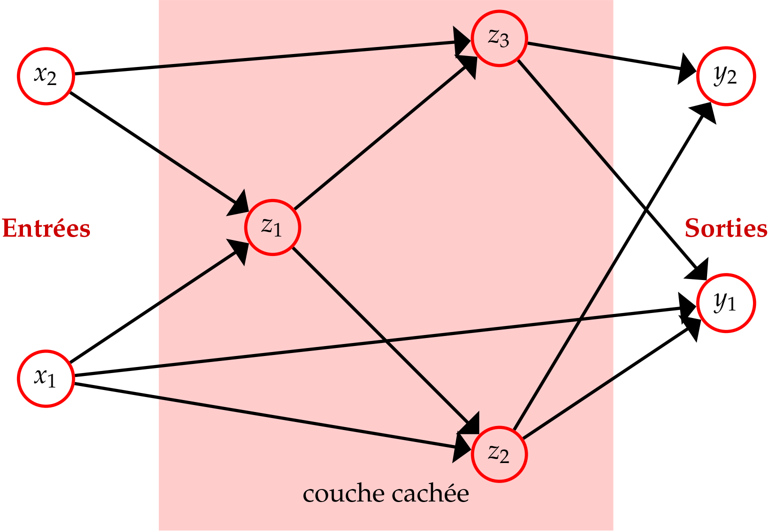
- D est l'ensemble des exemples d'apprentissage
- $t_{k,d}$ est la vraie classification de l'instance d pour la sortie k
- $o_{k,d}$ est la valeur de la sortie k pour l'instance d

Backpropagation



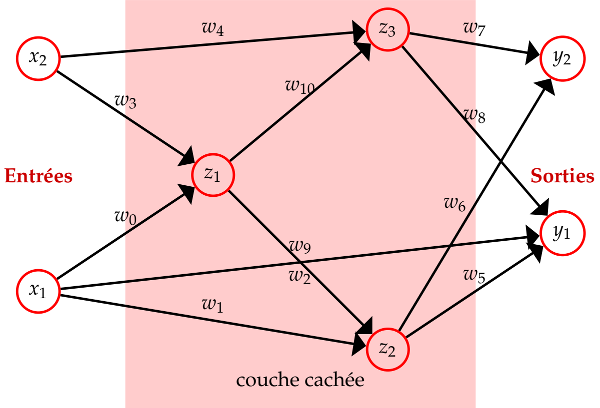
on peut calculer l'erreur aux sorties
mais il faut calculer une erreur pour les neurones de la couche interne !

Backpropagation



idée : on partage l'erreur en fonction des poids

Backpropagation



- z_3 contribue à la décision sur y_2 avec un poids de w_7
  ➾ on attribue à z_3 une partie de l'erreur de y_2 avec un poids de w_7
- z_3 contribue à la décision sur y_1 avec un poids de w_8
  ➾ on attribue à z_3 une partie de l'erreur de y_1 avec un poids de w_8

L'erreur de z_3 sera donc $w_7 \times erreur(y_2) + w_8 * erreur(y_1)$

Backpropagation algorithm

- Cet algorithme suppose que la fonction d'activation est la fonction logistique.
- x_{ji} est l'entrée du noeud j venant du noeud i et w_{ji} est le poids correspondant.
- δ_n est l'erreur associée à l'unité n. Cette erreur joue le rôle du terme $t-o$ dans le cas d'une seule unité.

initialise chaque w_i avec une valeur au hasard
tant que l'erreur est trop grande
    Pour chaque example $(\overrightarrow{x}, t) \in D$
        1- calcul de l'état du réseau par propagation
        2- rétro propagation des erreurs
            a- **pour chaque** unité de sortie k, calcul du terme d'erreur
                $\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$
            b- **pour chaque** unité cachée h, calcul du terme d'erreur
                $\delta_h \leftarrow o_h(1-o_h) \sum\limits_{k \in \text{Neurones en aval de } j} w_{k,h}\delta_k$
            c- mise à jour des poids w_{ji}
                $w_{j,i} \leftarrow w_{j,i} + \eta\delta_j x_{ji}$

Backpropagation algorithm

```
initialise chaque w_i avec une valeur au hasard
tant que l'erreur est trop grande
    Pour chaque example (x⃗, t) ∈ D
        1- calcul de l'état du réseau par propagation
        2- rétro propagation des erreurs
            a- pour chaque unité de sortie k, calcul du terme d'erreur
                δ_k ← o_k(1 − o_k)(t_k − o_k)
            b- pour chaque unité cachée h, calcul du terme d'erreur
                δ_h ← o_h(1 − o_h) Σ_{k ∈ Neurones en aval de j} w_{k,h} δ_k
            c- mise à jour des poids w_ji
                w_{j,i} ← w_{j,i} + η δ_j x_ji
```

Le changement de poid δ_j dépend de la couche du neurone j.

Comme pour le perceptron avec fonction d'activation linéaire, on peut trouver la formule en calculant les dérivées partielles. C'est juste un peu plus compliqué que pour le perceptron.

En réalité, la forme de l'erreur de la couche cachée dont on a donné l'intuition vient de cette dérivation.

Propriétés

- backpropagation converge vers un minimum local (aucune garantie que le minimum soit global)
- en pratique, il donne de très bons résultats
  dans la pratique, le grand nombre de dimensions peut donner des opportunités pour ne pas être bloqué dans un minimum local
- pour le moment, on n'a pas assez de théorie qui explique les bons résultats en pratique
  - ajouter un terme de "momentum"
  - entrainer plusieur RNA avec les mêmes données mais différents poids de départ
    (boosting)

Quelles fonctions peut on représenter avec un RNA ?

- fonctions booléennes
- fonctions continues (théoriquement : toute fonction continue peut être approximée avec une erreur arbitraire par un réseau avec deux niveaux d'unités)
- fonctions arbitraires (théoriquement : toute fonction peut être approximée avec une précision aribitraire par un réseau avec 3 niveaux d'unités)

Exemple d'apprentissage avec un RNA

Jouer avec des exemples :
`https://playground.tensorflow.org`

- parfois, on peut voir que des neurones de la couche cachée apprennent un "pattern" particulier
- les neurones de la couche de sortie apprennent à combiner ces "patterns"
- mais dans certains cas, les RNA sont plus des boites noires et il est (souvent ?) difficile d'interpréter les poids

Implémentation

- from scratch : bon moyen pour comprendre l'algorithme
- pour plus de rapidité ➫ favoriser l'écriture vectorisée et utiliser des outils qui utilisent ces optimisations (ex python/numpy ou C++)
- utiliser des outils développés pour faire de l'apprentissage de réseaux de neurones
  - tensorflow/keras
  - pyTorch

Conclusion

- vous savez maintenant comment marche un réseau de neurones
- pour faire des problèmes intéressant, il faut souvent avoir accès à des GPU
- certains types de topologies vont pouvoir faire des tâches spécialisées

➯ cours de Deep learning en M2