Fondements de l'apprentissage machine

Automne 2014

Roland Memisevic

Leçon 4

Plan

- Les limites des fonctions de base fixes.
- Les réseaux de neurones.
- Rétropropagation d'erreurs ("backprop").

Fonctions de base non-linéaires

- Dans les leçons précédentes, nous avons examiné les modèles linéaires pour la régression et la classification :

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} = \sum_j w_j x_j$$

- Nous avons vu que nous pouvons les transformer en modèles non-linéaires par prétraitement des entrées à l'aide de fonctions de base non-linéaires :

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{\Phi}(\mathbf{x}) = \sum_i w_i \phi_i(\mathbf{x})$$

- Nous considérons maintenant *apprendre* les fonctions de base (ainsi que les paramètres du modèle).
- Avantage : Souvent, il est difficile de savoir quelles sont les fonctions les plus appropriées pour une tâche.

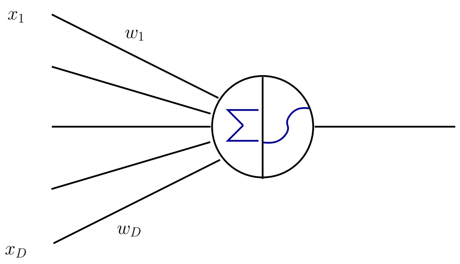
Réseaux de neurones

- Le type de modèle non-linéaire le plus commun est le **réseau de neurones feed-forward**.
- Dans les réseaux de neurones, les composants (dimensions) des fonctions de base sont modélisés comme des "neurones artificiels".
- Un "neurone" est défini comme suit :

$$h(\mathbf{w}^{\mathrm{T}}\mathbf{x})$$

  où $h(\cdot)$ est une fonction non-linéaire.
- Les paramètres s'appellent habituellement "poids".

Modèle de neurone



- Motivation : Selon le modèle le plus commun (mais très idéalisé) d'un neurone biologique, le neurone se déclenche avec une fréquence de décharge qui est une fonction non-linéaire d'une somme pondérée des fréquences de décharge d'autres neurones.

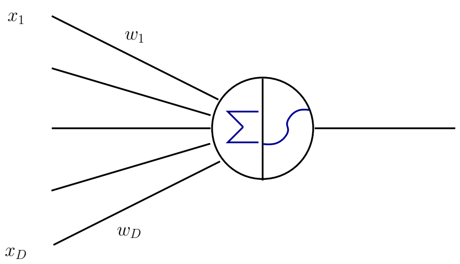
Réseau de neurones feed-forward

- La fonction non-linéaire s'appelle "fonction d'activation".
- La fonction d'activation la plus courante est le sigmoïde :

$$h(a) = \sigma(a) = \frac{1}{1 + \exp(-a)}$$

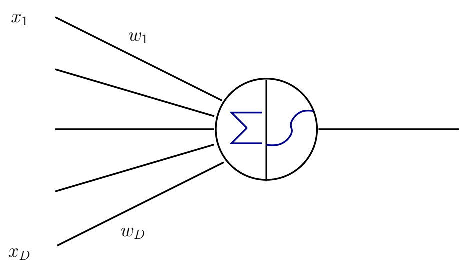
- Le sigmoïde est à la fois biologiquement assez plausible, et pratiquement commode.

Perceptron



- Notez que si on remplace le sigmoïde avec un seuil, on obtient un classifieur linéaire !
- Ce classifieur linéaire basé sur un seul neurone s'appelle perceptron (Rosenblatt, 1957).
- Le perceptron est un ancien modèle d'apprentissage inspiré par la biologie, et il est basé sur un algorithme d'apprentissage simple :

Perceptron

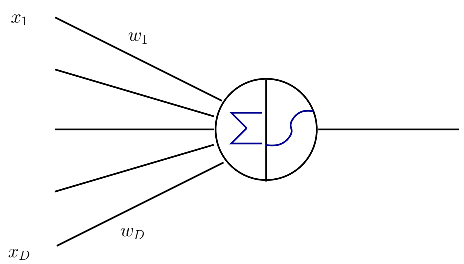


Apprentissage du perceptron (perceptron learning rule)

Encodez les cibles comme $-1/1$. Itérez sur l'ensemble d'entraînement $\{(\mathbf{x}_n, t_n)\}$ (dans n'importe quel ordre) en appliquant les mises à jour :

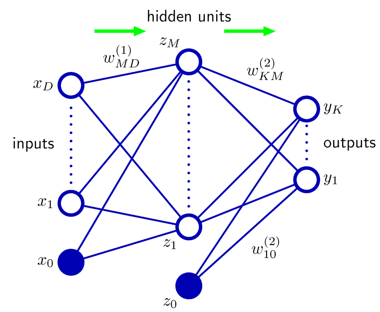
- $\mathbf{w} \leftarrow \mathbf{w} + (t_n - y(\mathbf{x}_n))\mathbf{x}_n$

Perceptron



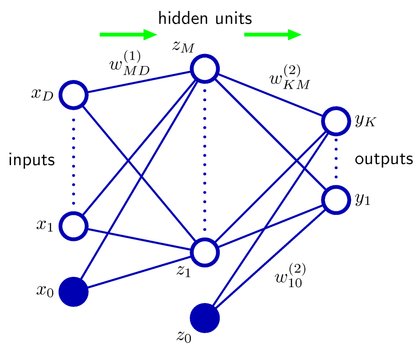
- Il existe plusieurs variantes de la règle d'apprentissage du perceptron.
- On peut montrer que la règle d'apprentissage converge si l'ensemble de données est linéairement séparable.

Réseau de neurones feed-forward



- Pour apprendre des fonctions non-linéaires, nous pouvons combiner plusieurs unités de calcul non-linéaires dans un réseau constitué de plusieurs couches.
- Notation : z_i désigne les sorties de la première couche, y_i désigne les sorties de la dernière couche, et $w_{ji}^{(k)}$ connecte le neurone j de la couche $k-1$ avec le neurone i de la couche k.

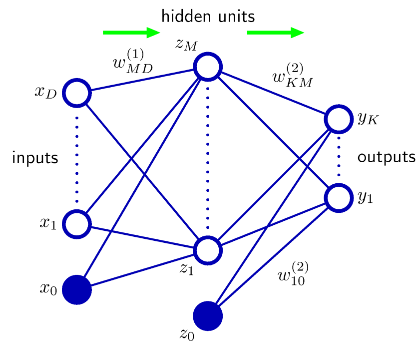
Réseau de neurones feed-forward



- Formellement :

$$y_k(\mathbf{x}) = \sum_{j=1}^{M} w_{kj}^{(2)} h\Big(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\Big) + w_{k0}^{(2)}$$

Réseau de neurones feed-forward



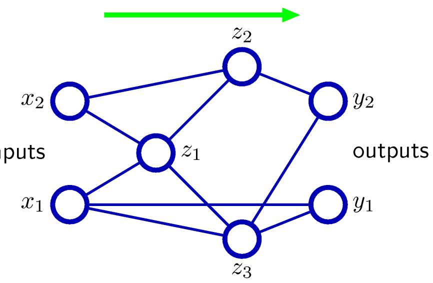
- En absorbant tous les biais (comme d'habitude) :

$$y_k(\mathbf{x}) = \sum_{j=0}^{M} w_{kj}^{(2)} h\Big(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\Big)$$

Réseau de neurones feed-forward

- Toutes les couches, sauf la première (entrées) et la dernière (sorties), s'appellent des "couches cachées", et les neurones des "unités cachées".
- En principe, on peut définir des architectures bien plus complexes que des réseaux feed forward. Cependant, l'apprentissage peut devenir beaucoup plus compliqué dans ce cas.

Réseau de neurones feed-forward



- Un réseau feed forward peut avoir une connectivité éparse, où la plupart les connexions sont zéro, conduisant à des topologies plus intéressantes comme celle montrée ici.
- Pour que l'apprentissage reste simple, il suffit que le graphe du réseau soit un graphe orienté acyclique.

Fonctions d'activation

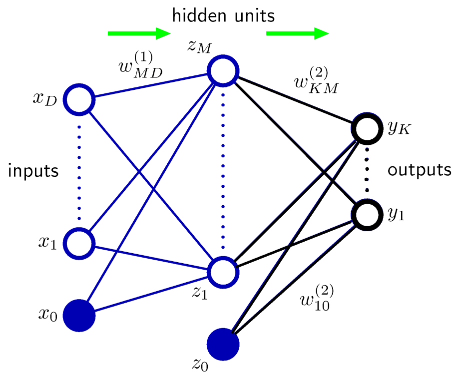
- Un réseau de neurones avec des fonctions d'activation linéaires produirait un modèle linéaire parce que la composition de fonctions linéaires est toujours linéaire.
- C'est la raison pour laquelle les non-linéarités sont nécessaires dans le réseau.
- Il y a d'autres fonctions d'activation communes, par exemple la fonction tangente hyperbolique $\tanh$ et le rectificateur $a \cdot [a > 0]$.

Fonctions d'activation

- La fonction d'activation de la dernière couche détermine la fonctionnalité du réseau.
  - Si nous utilisons une fonction d'activation linéaire, le modèle fera une régression non-linéaire : c'est un modèle de régression linéaire appliqué à des données prétraitées non-linéairement dans les couches précédentes.
  - Si nous utilisons un seul neurone sigmoïde, le modèle réalisera une régression logistique binaire.
  - Nous pouvons définir un modèle de régression softmax à l'aide de K neurones combinés par une fonction

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{\exp(y_k(\mathbf{x}))}{\sum_j \exp(y_j(\mathbf{x}))}$$

Fonctions d'activation

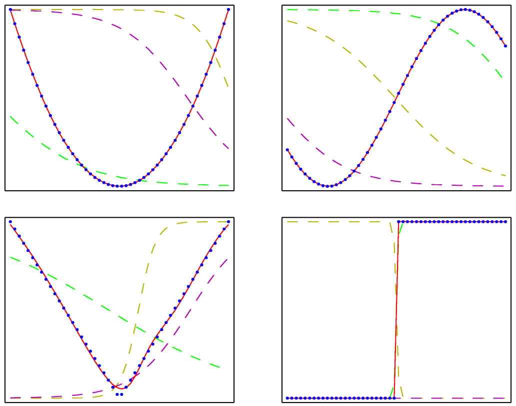


- La fonction d'activation de la dernière couche détermine la fonctionnalité du réseau.

Théorèmes d'approximation universelle

- On peut montrer (par exemple, Funahashi 1989) qu'un réseau avec une seule couche cachée sigmoïde peut modéliser une fonction non-linéaire (sous certaines conditions assez douces) avec une précision arbitraire.
- Cependant, le nombre d'unités requises peut être très grand.
- En pratique, les réseaux avec plusieurs couches fonctionnent souvent beaucoup mieux.

Exemples 1-d



- Données : bleu, sorties du modèle : rouge

Entraînement

- Pour entraîner un réseau de neurones pour la régression en utilisant les données $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}$, minimisez :

$$\frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2$$

- On peut toujours interpréter cela comme la maximisation de vraisemblance :

$$p(\mathcal{D}) = \prod_n \mathcal{N}(\mathbf{t}_n | \mathbf{y}(\mathbf{x}_n, \mathbf{w}), a\mathbf{I})$$

où la moyenne de gaussiennes est la sortie du réseau (elle était une fonction linéaire auparavant).

Entraînement

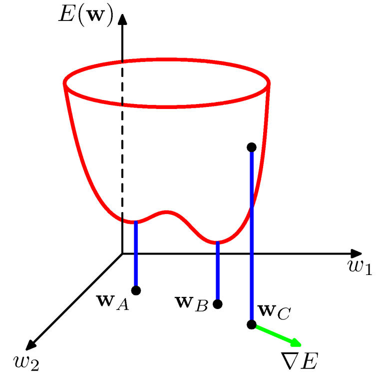
- Pour entraîner un réseau de neurones pour la classification en utilisant les données $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}$, minimisez :

$$-\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \log p(\mathcal{C}_k|\mathbf{x})$$

où les probabilités sur les classes sont données par une fonction "softmax" dont les entrées sont les sorties du réseau :

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{\exp(y_k(\mathbf{x}))}{\sum_j \exp(y_j(\mathbf{x}))}$$

Minima locaux



- La fonction objectif est non-convexe, donc il peut y avoir des minima locaux.
- L'optimisation conduira à des paramètres qui sont au moins mieux que l'initialisation, même s'ils ne sont pas l'optimum global.

Rétropropagation de l'erreur (error back-propagation)

- Pour effectuer la descente de gradient, il faut calculer les dérivées par rapport aux paramètres.
- Il y a une procédure générale qui permet de calculer les dérivées dans des réseaux de n'importe quelle topologie et n'importe quelle profondeur (au moins en théorie) : error back-propagation ou back-prop.

Error back-propagation

- Rappelons que la descente de gradient stochastique s'applique lorsque la fonction de perte se décompose en une somme sur les exemples entraînement :

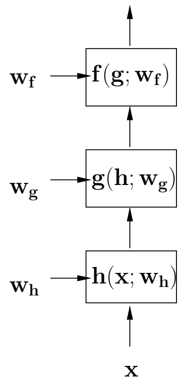
$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

- Il faut calculer les dérivées

$$\frac{\partial E_n}{\partial \mathbf{w}}$$

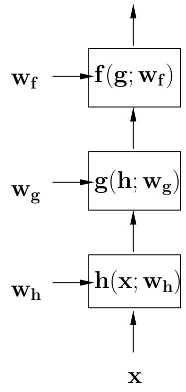
par rapport à des poids w_{ji} qui connectent deux neurones x_i, x_j.

Error back-propagation généralement



- Si le modèle, et de ce fait $E_n(\mathbf{w})$, se compose de plusieurs étapes exécutées en séquence, la dérivation en chaîne s'applique, ce qui rend le calcul des dérivées simples.

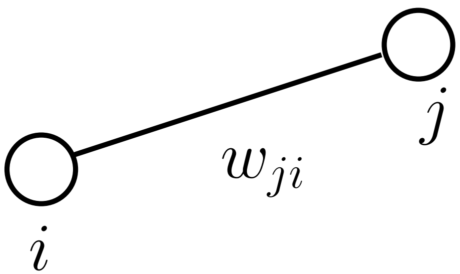
Error back-propagation généralement



$\mathbf{f}$	$\frac{\partial \mathbf{f}}{\partial \mathbf{g}}$	$\frac{\partial \mathbf{f}}{\partial \mathbf{w_f}}$
fprop	bprop	grad

- Si le modèle, et de ce fait $E_n(\mathbf{w})$, se compose de plusieurs étapes exécutées en séquence, la dérivation en chaîne s'applique, ce qui rend le calcul des dérivées simples.

Error back-propagation pour réseaux de neurones



- Pour calculer $\frac{\partial E_n}{\partial w_{ji}}$, nous utilisons les définitions suivantes :
- Soit z_i la sortie du neurone i.
- Soit $a_j = \sum_i w_{ji} z_i$ l'entrée nette d'un neurone ultérieur, j, connecté au neurone i par w_{ji}
- Donc $z_i = h(a_i)$

Error back-propagation pour réseaux de neurones

- Par la règle de dérivation en chaîne, nous avons :

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

- Le deuxième facteur est facile :

$$\frac{\partial a_j}{\partial w_{ji}} = z_i$$

et tous les z_i sont faciles à calculer : il suffit d'exécuter le réseau !

- Pour traiter le premier facteur, nous définissons $\delta_j := \frac{\partial E_n}{\partial a_j}$ et réécrivons la dérivée :

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$$

Error back-propagation pour réseaux de neurones

- Appliquer la règle de dérivation en chaîne une fois de plus pour obtenir

$$\delta_j = \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j}$$

  où la somme porte sur les neurones k connectés au neurone j.
- Intuitivement, cela reflète le fait que changer a_j affecte la fonction de coût à travers tous les a_k.
- Grâce à nos définitions, cela se simplifie à

$$\delta_j = h'(a_j)\sum_k w_{kj}\delta_k$$

- Donc, nous pouvons utiliser une récurrence pour calculer tous les δ à partir de la sortie !

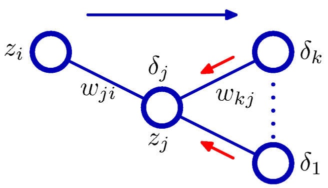
Error back-propagation pour réseaux de neurones

- Calculer les δ's pour les sorties est facile si la dernière couche est définie comme une régression linéaire ou logistique :
- Pour la fonction de coût d'erreur au carré (régression), on a

$$\frac{\partial}{\partial a_k}\frac{1}{2}\|\mathbf{y}(\mathbf{x},\mathbf{w}) - \mathbf{t}\|^2 = y_k - t_k$$

  parce que $y_k = a_k$ pour la régression.
- Il en est de même pour la régression logistique (activation softmax).

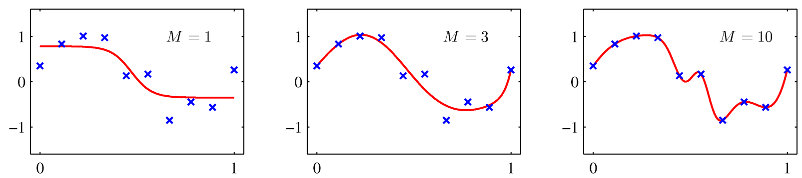
Error back-propagation pour réseaux de neurones



Sommaire (Bishop, page 244) :

1. Pour l'entrée $\mathbf{x}_n$, propagez vers l'avant ("forward propagate") pour trouver les activations de chaque neurone caché z_i et les sorties.
2. Calculez δ_k pour toutes les sorties.
3. Propagez les δ vers l'arrière ("backward propagate") pour obtenir les δ_j pour chaque neurone caché.
4. Calculez les dérivées pour chaque W_{ji} en multipliant les δ_j et les z_i.

Sur-apprentissage et régularisation

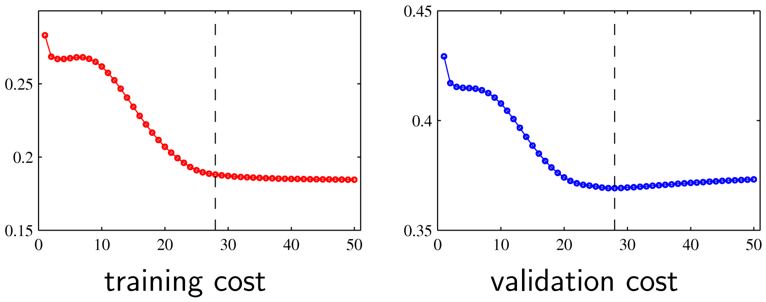


- Comme d'habitude, il est courant d'ajouter à l'objectif une pénalité pour prévenir le sur-apprentissage :

$$\frac{\lambda}{2}\|\boldsymbol{\theta}\|^2$$

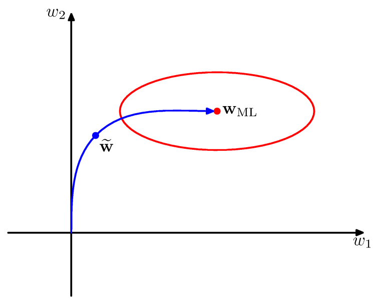
  où $\boldsymbol{\theta}$ est un vecteur contenant *tous* les poids.

Régularisation par "Early stopping"



- Normalement, le coût de validation commence à augmenter à un certain moment pendant l'apprentissage même si le coût d'entraînement diminue.
- Une méthode de régularisation est "early stopping" : enregistrez les modèles en cours d'entraînement et choisissez finalement celui qui donne la meilleure performance sur les données de validation.

Régularisation par "Early stopping"

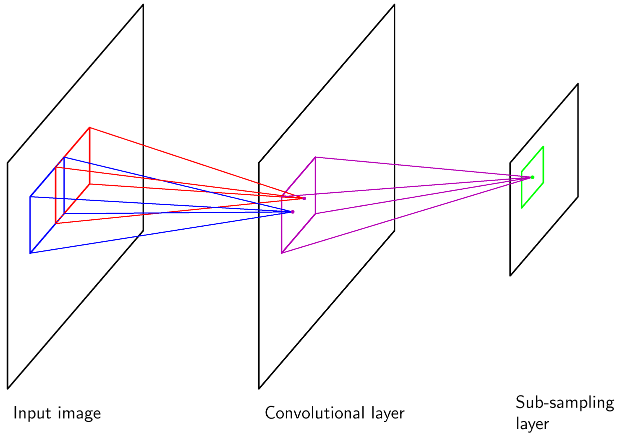


- Early stopping se comporte comme la régularisation par pénalité (weight decay) si les poids sont initialisés à des valeurs faibles.

Partage de poids

- Une troisième forme de régularisation dans un réseau de neurones est le partage de poids :
- Réduire le nombre de paramètres du modèle en forçant différentes parties du réseau à utiliser les mêmes paramètres.

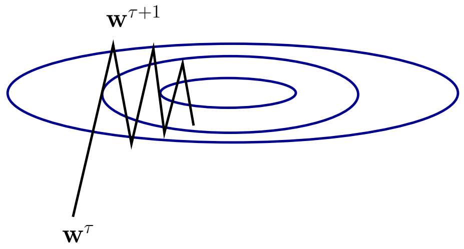
Partage de poids et réseaux convolutionnels



D'autres types de réseaux de neurones

- Radial basis function networks, Réseaux de neurones recurrents, Hopfield networks, Boltzmann machines, Helmholtz machines, echo-state networks, self-organizing maps, ...

Accélérer l'entraînement avec le momentum

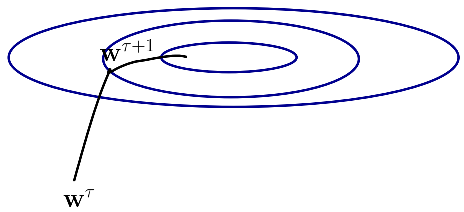


- La convergence peut être lente en raison de longues "vallées" dans la surface d'erreur, ce qui conduit à des oscillations pendant d'entraînement.
- Solution : ajoutez un terme "momentum"

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \frac{\partial E_n}{\partial \mathbf{w}} + \mu \Delta \mathbf{w}(\tau)$$

où $\Delta \mathbf{w}(\tau) = \left(\mathbf{w}^{(\tau)} - \mathbf{w}^{(\tau-1)}\right)$ et μ est une petite valeur (normalement entre 0 et 1)

Accélérer l'entraînement avec le momentum



- Avec le momentum, les oscillations orthogonales à la vallée sont atténuées et l'accumulation des directions similaires mène effectivement à de plus grands pas dans l'espace de paramètres.
- L'utilisation du momentum conduit souvant à une accélération très significative.