Cours 4 : Réseaux de Neurones et Apprentissage Profond (en OCaml)

Où l'on découvre que faire du machine learning sans PyTorch est une forme de méditation transcendantale

Attention : Ce cours contient des blagues plus/moins drôles. La responsabilité de l'auteur n'est pas engagée en cas de fou rire en plein examen.

Dernière compilation : June 27, 2026

Contents

1 Introduction : OCaml ? Sérieusement ?

«Professeur, pourquoi on n'utilise pas Python comme tout le monde ? »

«Parce que vous êtes en cours de programmation fonctionnelle, pas en cours de «taper `pip install` et prier». »

«Mais... TensorFlow... »

*«TensorFlow, c'est pour les gens qui ne savent pas faire une multiplication de matrices à la main. Et vous, vous allez la faire à la main. En OCaml. Avec des listes. Comme des **vrais**. »*

Ce cours est l'aboutissement d'une longue tradition d'implémentations «from scratch » (en français : «à partir de zéro, sans filet, sans pitié »). Tous les programmes présentés ici sont disponibles dans le répertoire du cours :

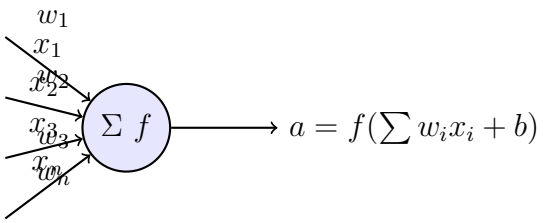
Fichier	**Contenu**
`regression.ml`	Régression polynomiale (moindres carrés, équations normales)
`classifier.ml`	Classification binaire (logistic regression, descente de gradient)
`encoder.ml`, `decoder.ml`	Encodeur et décodeur (réseaux feedforward simples)
`autoencoder.ml`	Autoencodeur complet avec rétropropagation
`rnn.ml`	RNN pour la parité binaire avec BPTT
`cnn.ml`	CNN pour la classification de motifs 2D
`graphiques.ml`	Générateur de graphiques SVG

Contrairement à une implémentation en Python avec PyTorch ou TensorFlow (où tout est fait pour vous), ici chaque multiplication de matrice, chaque dérivée, chaque mise à jour de poids est **explicitement écrite en OCaml**. C'est plus long, c'est plus dur, mais à la fin vous comprenez VRAIMENT comment ça marche.

2 Le Neurone Artificiel

2.1 Du neurone biologique au neurone formel

Un neurone biologique reçoit des signaux via ses dendrites, les accumule, et si le seuil est dépassé, envoie une décharge par l'axone. Le neurone artificiel fait exactement pareil, mais avec des nombres.



Mathématiquement, un neurone c'est deux opérations :

$$z = \sum_{i=1}^{n} w_i x_i + b = \mathbf{w}^T \mathbf{x} + b \qquad a = f(z)$$

2.2 Les fonctions d'activation

Le choix de f change radicalement le comportement du neurone :

Nom	Formule	Usage
Sigmoid	$\sigma(z) = \frac{1}{1+e^{-z}}$	Sortie entre 0 et 1 (probabilité)
Tanh	$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Cache des RNN (centré en 0)
ReLU	$\text{ReLU}(z) = \max(0, z)$	Couches cachées (efficace, simple)
Linéaire	$f(z) = z$	Sortie de régression

Sigmoid : *«Je compresse tout entre 0 et 1 ! »*

Tanh : *«Moi aussi, mais symétrique, et centré en 0. »*

ReLU : *«Les négatifs ? JAMAIS ENTENDU PARLER. Je les mets à 0. »*

Softmax : *«Bon les gars, on normalise tout ça pour que ça ressemble à des probabilités ? »*

2.3 Le code OCaml : le type `activation`

Dans tous nos programmes, les fonctions d'activation sont représentées par un type somme OCaml. C'est propre, c'est typé, c'est beau.

Listing 1: Type `activation` et fonction d'application (extrait de `autoencoder.ml`)

```
28 (*
       ----------------------------------------------------------------

29     2. Types du reseau
30     ----------------------------------------------------------------
           *)
31
32 type activation = Sigmoid | Tanh | Relu | Linear
```

Chaque variante du type `activation` correspond à une formule mathématique. Le pattern matching permet d'appliquer la bonne fonction sans `if` lourds, sans `switch`, sans risques d'oublier un cas (le compilateur vous le rappelle).

Voici les dérivées correspondantes, essentielles pour la rétropropagation :

Listing 2: Dérivées des fonctions d'activation (`autoencoder.ml`)

```
19 let sigmoid x = 1. /. (1. +. exp (-.x))
20 let dsigmoid x = let s = sigmoid x in s *. (1. -. s)
21
22 let tanh x = (exp (2. *. x) -. 1.) /. (exp (2. *. x) +. 1.)
23 let dtanh x = let t = tanh x in 1. -. t *. t
24
25 let relu x = if x > 0. then x else 0.
26 let drelu x = if x > 0. then 1. else 0.
```

3 La Rétropropagation : Le Cœur du Réseau

3.1 Descente de gradient

On a une fonction de coût $E(\mathbf{w})$ (l'erreur du réseau). On veut minimiser cette erreur. La méthode la plus simple : suivre la pente.

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \cdot \nabla E(\mathbf{w}^{(t)})$$

Le taux d'apprentissage η contrôle la vitesse de descente.

Si η est trop petit : on avance comme un escargot en PLS.
Si η est trop grand : on fait du hors-piste dans les Alpes.
Le bon η : celui qui vous fait descendre sans vous crasher.

3.2 La chaîne de dérivées (backprop)

Pour un réseau à L couches, on note $a^{(\ell)}$ l'activation de la couche ℓ et $z^{(\ell)}$ l'activation linéaire avant la non-linéarité.

La rétropropagation calcule $\delta^{(\ell)} = \frac{\partial E}{\partial z^{(\ell)}}$, l'erreur locale de chaque neurone, en propageant de la sortie vers l'entrée :

$$\delta^{(L)} = \nabla_a E \odot f'(z^{(L)}) \tag{1}$$
$$\delta^{(\ell)} = \left((W^{(\ell+1)})^T \delta^{(\ell+1)}\right) \odot f'(z^{(\ell)}) \tag{2}$$

Où $\odot$ est le produit de Hadamard (terme à terme).

Une fois les δ calculés, les gradients des poids sont :

$$\frac{\partial E}{\partial W^{(\ell)}} = \delta^{(\ell)} (a^{(\ell-1)})^T \qquad \frac{\partial E}{\partial b^{(\ell)}} = \delta^{(\ell)}$$

3.3 Le code OCaml : la fonction `backward`

Voici le cœur de la rétropropagation, qui calcule les gradients pour toutes les couches en une seule passe arrière :

Listing 3: Fonction `backward` : calcule les gradients par rétropropagation (extrait de `autoencoder.ml`)

```
85 (* backward : calcule les gradients pour tout le reseau *)
86 let backward reseau couches_act sortie cible =
87   (* Erreur sur la sortie : dE/da = a - cible (MSE) *)
88   let da = List.map2 (-.) sortie cible in
89   (* Parcours des couches en sens inverse *)
90   let (_ , grads_poids, grads_biais) =
91     List.fold_right (fun (couche, (x, z, a)) (da_suiv, gp, gb) ->
92       let dact = derivee_actif couche.actif in
93       (* dE/dz = dE/da * dact(z) *)
94       let dz = List.map2 (fun da_i z_i -> da_i *. dact z_i) da_suiv
             z in
95       (* dE/dW = dz * x^T  (produit externe) *)
96       let gp_couche = List.map (fun dz_j ->
```

```
97          List.map (fun x_i -> dz_j *. x_i) x
98        ) dz in
99        (* dE/db = dz *)
100       let gb_couche = dz in
101       (* dE/dx = W^T * dz  (pour la couche precedente) *)
102       let n_entrees = List.length (List.hd couche.poids) in
103       let da_prec = List.init n_entrees (fun i ->
104         List.fold_left2 (fun s w_j_i dz_j -> s +. w_j_i *. dz_j)
105           0. (List.map (fun ligne -> List.nth ligne i) couche.poids
                ) dz
106       ) in
107       (da_prec, gp_couche :: gp, gb_couche :: gb)
108     ) (List.combine reseau (List.rev couches_act)) (da, [], [])
109   in
110   (grads_poids, grads_biais)
```

Détaillons ce code ligne par ligne :

- **Ligne 86-87** : On initialise $\delta = \frac{\partial E}{\partial a} = a - y$ (l'erreur de sortie, pour une fonction de coût quadratique).
- **Ligne 88-89** : `List.fold_right` parcourt les couches de la dernière à la première (grâce à `List.rev` qui remet les activations dans l'ordre du réseau, et `fold_right` qui traite de droite à gauche).
- **Ligne 91** : $\delta^{(\ell)} = \delta^{(\ell+1)} \odot f'(z^{(\ell)})$ (produit terme à terme entre l'erreur propagée et la dérivée de l'activation).
- **Ligne 92-94** : $\frac{\partial E}{\partial W^{(\ell)}} = \delta^{(\ell)}(a^{(\ell-1)})^T$ (produit extérieur : pour chaque neurone j, $\frac{\partial E}{\partial W_{ji}} = \delta_j \cdot x_i$).
- **Lignes 95-96** : $\frac{\partial E}{\partial b^{(\ell)}} = \delta^{(\ell)}$ (le gradient du biais est simplement l'erreur locale).
- **Lignes 98-102** : Calcul de $\delta^{(\ell)} = (W^{(\ell+1)})^T \delta^{(\ell+1)}$ pour propager l'erreur à la couche précédente. On prend la transposée de la matrice des poids et on la multiplie par δ.

Note culturelle : *Cette fonction a fait pleurer des générations d'étudiants. Si vous la comprenez du premier coup, vous êtes soit un génie, soit vous mentez.*

3.4 La mise à jour des poids

Une fois les gradients calculés, on met à jour les poids :

Listing 4: Fonction `mise_a_jour` : SGD sur tous les poids et biais (`autoencoder.ml`)

```
116 let mise_a_jour reseau grads_poids grads_biais taux =
117   List.iter2 (fun couche (gp, gb) ->
118     couche.poids <- List.map2 (fun ligne dW ->
119       List.map2 (fun w dw -> w -. taux *. dw) ligne dW
120     ) couche.poids gp;
121     couche.biais <- List.map2 (fun b db -> b -. taux *. db) couche.
          biais gb
122   ) reseau (List.combine grads_poids grads_biais)
```

C'est l'étape la plus simple : $W \leftarrow W - \eta \cdot \frac{\partial E}{\partial W}$.

4 Régression Polynomiale : Les Moindres Carrés

4.1 Le problème

On a des points (x_i, y_i) et on cherche un polynôme $P(x) = a_0 + a_1 x + \cdots + a_d x^d$ qui minimise l'erreur quadratique moyenne.

4.2 Solution analytique : équations normales

Contrairement aux réseaux de neurones qu'on entraîne par descente de gradient, la régression polynomiale a une **solution fermée** (analytique). On pose la matrice de Vandermonde X où $X_{i,j} = x_i^j$, et la solution est :

$$\beta = (X^T X)^{-1} X^T y$$

C'est exact, direct, et ça ne nécessite pas d'itérations. Pas de taux d'apprentissage, pas de mini-batchs, pas de prières nocturnes.

4.3 Le code OCaml

Listing 5: Régression polynomiale par équations normales (`regression.ml`)

```
let regression_poly (xs : float list) (ys : float list) (degre :
    int) : vect =
  let x = vandermonde xs degre in
  let xt = transposer x in
  let xtx = mul_mat xt x in
  let xty = mul_mat_vect xt ys in
  resous_systeme xtx xty
```

La matrice de Vandermonde se construit élégamment en OCaml :

Listing 6: Construction de la matrice de Vandermonde (`regression.ml`)

```
let vandermonde (xs : float list) (degre : int) : mat =
  List.map (fun x ->
    List.init (degre + 1) (fun j -> x ** float_of_int j)
  ) xs
```

4.4 Résultats

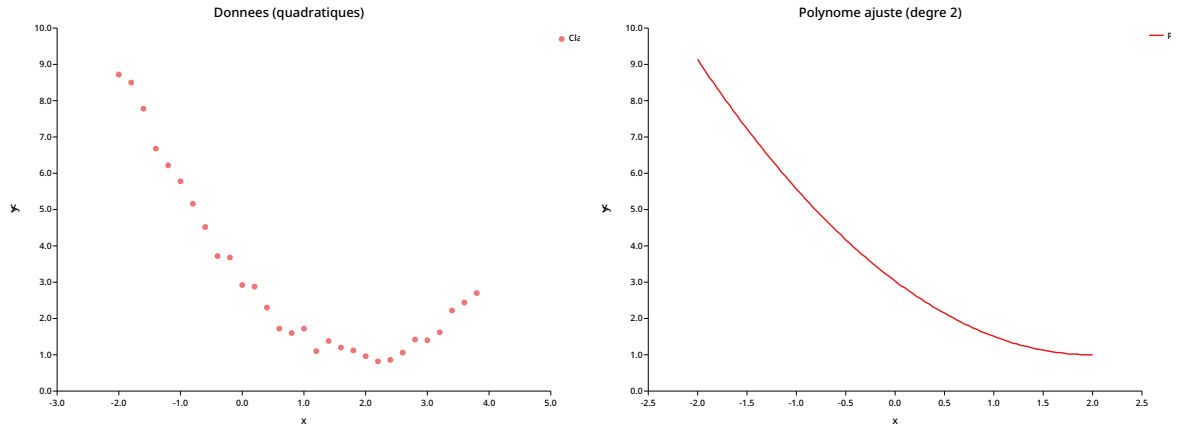


Figure 1: Régression quadratique : données brutes (gauche) et polynôme ajusté degré 2 (droite).

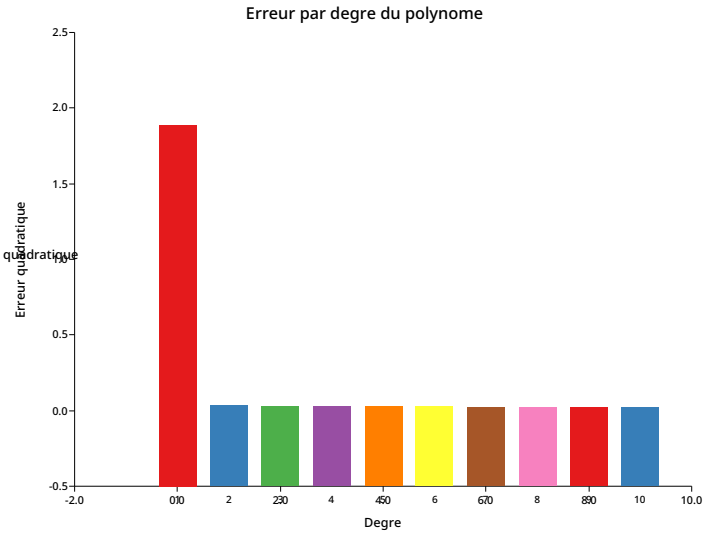


Figure 2: Erreur en fonction du degré du polynôme. Au-delà de degré 2, le gain est marginal.

5 Classification Linéaire : Logistic Regression

5.1 Du continu au discret

La régression prédit une valeur continue. Pour classifier en deux groupes, on utilise la **fonction sigmoid** qui transforme une valeur réelle en probabilité entre 0 et 1 :

$$P(\text{classe} = 1 \mid x) = \sigma(w_0 + w_1 x + w_2 y) = \frac{1}{1 + e^{-(w_0 + w_1 x + w_2 y)}}$$

5.2 La fonction de coût : cross-entropy

On ne peut pas utiliser l'erreur quadratique pour la classification (elle donne de mauvais gradients). On utilise la **cross-entropie** :

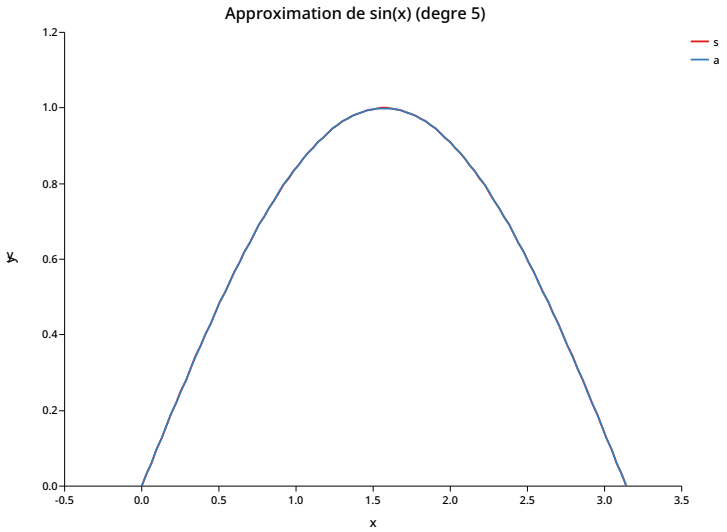


Figure 3: Approximation de $\sin(x)$ par un polynôme de degré 5 sur $[0, \pi]$.

$$E = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(p_i) + (1-y_i)\log(1-p_i)\right]$$

Listing 7: Fonction de coût cross-entropy (`classifier.ml`)

```
let cout (w0, w1, w2) donnees =
  let n = float_of_int (List.length donnees) in
  List.fold_left (fun somme (x, y, cible) ->
    let p = prediction (w0, w1, w2) (x, y) in
    (* Cross-entropy : -[cible*log(p) + (1-cible)*log(1-p)] *)
    let eps = 1e-15 in  (* evite log(0) *)
    let p = max eps (min (1. -. eps) p) in
    somme -. (cible *. log p +. (1. -. cible) *. log (1. -. p))
  ) 0. donnees /. n
```

5.3 Descente de gradient (code complet)

Listing 8: Descente de gradient avec taux d'apprentissage décroissant (`classifier.ml`)

```
let descente_gradient donnees taux pas_init n_iter =
  let rec boucle w t =
    if t >= n_iter then w
    else
      let taux_decay = taux /. (1. +. 0.001 *. float_of_int t) in
      let (g0, g1, g2) = gradient w donnees in
      let (w0, w1, w2) = w in
      boucle (w0 -. taux_decay *. g0,
              w1 -. taux_decay *. g1,
              w2 -. taux_decay *. g2) (t + 1)
  in
  boucle (0., 0., 0.) 0
```

Notez le `taux_decay` : le taux d'apprentissage diminue avec le temps ($\eta_t = \eta_0/(1 + 0.001t)$). C'est une technique classique pour converger plus proprement : on fait de grands pas au début, puis on affine.

5.4 Résultats

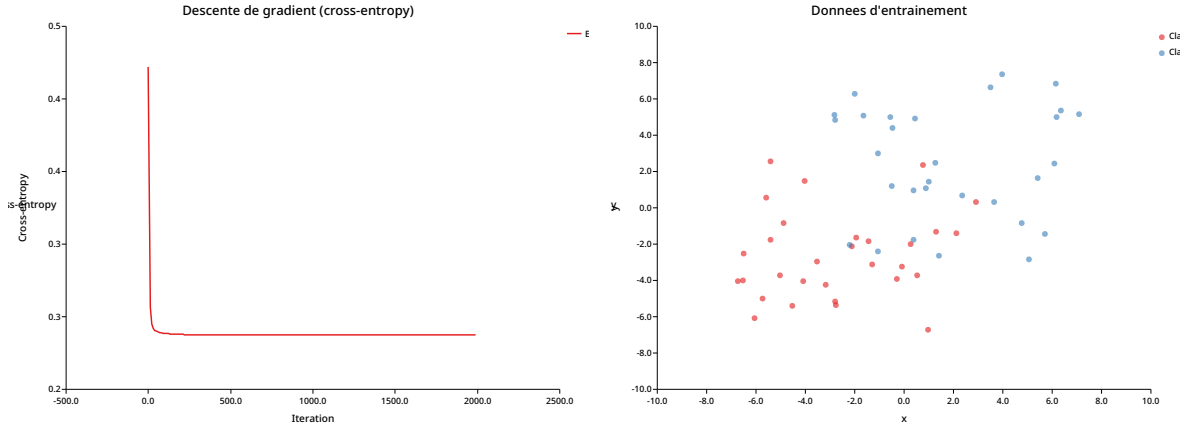


Figure 4: Gauche : évolution de la cross-entropy pendant l'entraînement. Droite : nuage de points des deux classes.

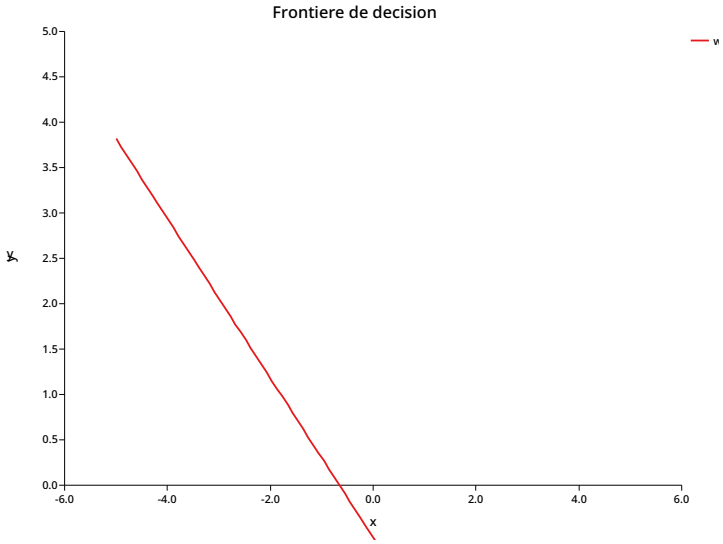


Figure 5: Frontière de decision $w_0 + w_1 x + w_2 y = 0$ séparant les deux classes.

6 Autoencodeur : Le Caméléon des Réseaux

6.1 Architecture encodeur-décodeur

Un autoencodeur est un réseau qui apprend à recopier son entrée en passant par un **goulot d'étranglement** (bottleneck) qui force la compression :

$$\xrightarrow{x} \underbrace{\text{Encodeur}}_{W_1,b_1} \xrightarrow{h=f(W_1x+b_1)} \underbrace{\text{Décodeur}}_{W_2,b_2} \xrightarrow{\hat{x}=g(W_2h+b_2)}$$

Avec comme objectif : $\hat{x} \approx x$. Le code latent h apprend une **représentation compacte** des données.

6.2 Les 4 couches de notre autoencodeur

Notre autoencodeur a 4 couches (encodeur 2 couches, décodeur 2 couches) :

$$\underbrace{2}_{\text{entrée}} \xrightarrow{\tanh} \underbrace{3}_{\text{cachée}} \xrightarrow{\text{lin}} \underbrace{1}_{\text{code}} \xrightarrow{\tanh} \underbrace{3}_{\text{cachée}} \xrightarrow{\text{lin}} \underbrace{2}_{\text{sortie}}$$

6.3 Le code : construction de l'autoencodeur

Listing 9: Construction de l'autoencodeur (`autoencoder.ml`)

```
149 (* Autoencodeur : 2 -> 3 (tanh) -> 1 (linear) -> 3 (tanh) -> 2 (
    linear) *)
150 let autoencodeur () =
151   [ cree_couche 2 3 Tanh;     (* encodeur *)
152     cree_couche 3 1 Linear;
153     cree_couche 1 3 Tanh;     (* decodeur *)
154     cree_couche 3 2 Linear;
155   ]
```

La fonction `cree_couche` initialise les poids aléatoirement avec une normalisation adaptée. `Tanh` pour les couches cachées (non-linéarité), `Linear` pour les couches de sortie.

6.4 Le passage avant (forward)

Listing 10: Passage avant complet : de l'entrée à la sortie en passant par chaque couche, avec sauvegarde des activations pour la rétropropagation (`autoencoder.ml`)

```
58 (* forward_entrainement : calcule les sorties ET sauvegarde
59    les entrees/sorties de chaque couche pour la retropropagation *)
60 let forward_entrainement reseau entree =
61   let _ = entree in  (* entrees du reseau *)
62   List.fold_left (fun (x, couches_act) couche ->
63     let z = List.map2 (fun ligne bias ->
64       List.fold_left2 (fun s w_ji x_i -> s +. w_ji *. x_i) bias
          ligne x
65     ) couche.poids couche.biais in
66     let a = match couche.actif with
67       | Sigmoid -> List.map sigmoid z
```

```
        | Tanh    -> List.map tanh z
        | Relu    -> List.map relu z
        | Linear  -> z
      in
      (a, (x, z, a) :: couches_act)
  ) (entree, []) reseau
```

Chaque étape calcule $z = Wx + b$ puis $a = f(z)$. On conserve (x, z, a) pour pouvoir rétropropager ensuite.

6.5 Résultats

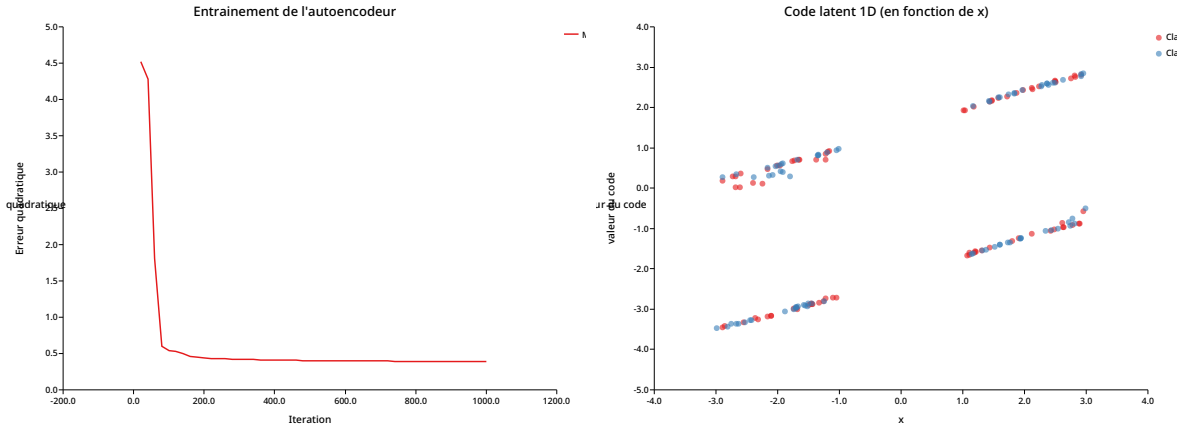


Figure 6: Gauche : l'erreur quadratique descend de 9 à 0,3 en 1000 itérations. Droite : le code latent 1D distingue les clusters de points.

Le code latent sépare les points par cluster : chaque groupe de points d'origine (définis par leur position dans le plan 2D) se voit attribuer une valeur de code différente. C'est exactement ce qu'on attend : l'autoencodeur apprend une **représentation discriminante** des données.

7 Réseau de Neurones Récurrent (RNN)

7.1 Le problème de la mémoire

Les réseaux feedforward (ceux vus jusqu'ici) prennent une entrée et produisent une sortie, point final. Pas de mémoire. Pour traiter des séquences (texte, audio, séries temporelles), il faut un état interne qui évolue.

Un feedforward qui traite une phrase : « Il était une fois … quoi déjà ? »

7.2 L'équation du RNN

Le RNN ajoute un **état caché** h_t qui se transmet d'un pas au suivant :

$$h_t = \tanh(W_{xh} x_t + W_{hh} h_{t-1} + b_h) \tag{3}$$
$$y_t = \sigma(W_{hy} h_t + b_y) \tag{4}$$

L'état h_t résume toute l'information vue jusqu'à l'instant t. C'est la **mémoire** du réseau.

7.3 Le code OCaml

Listing 11: Type `rnn` avec ses 5 matrices de poids et initialisation (`rnn.ml`)

```
41 let n_entree = 1
42 let n_cache  = 6
43 let n_sortie = 1
44
45 type rnn = {
46   mutable wxh : float list list;   (* n_cache × n_entree *)
47   mutable whh : float list list;   (* n_cache × n_cache  *)
48   mutable why : float list list;   (* n_sortie × n_cache *)
49   mutable bh  : float list;        (* n_cache             *)
50   mutable by  : float list;        (* n_sortie            *)
51 }
52
53 let rand () = Random.float 2. -. 1.
54
55 let init_rnn () = {
56   wxh = List.init n_cache (fun _ -> List.init n_entree (fun _ ->
       rand () /. 2.));
57   whh = List.init n_cache (fun _ -> List.init n_cache (fun _ ->
       rand () /. 2.));
58   why = List.init n_sortie (fun _ -> List.init n_cache (fun _ ->
       rand () /. 2.));
59   bh  = List.init n_cache (fun _ -> 0.);
60   by  = List.init n_sortie (fun _ -> 0.);
61 }
```

Listing 12: Passage avant du RNN sur une séquence : pour chaque pas de temps, on calcule h_t puis y_t (`rnn.ml`)

```
let forward rnn seq =
  let h0 = List.init n_cache (fun _ -> 0.) in
  let hs = Array.make (List.length seq + 1) h0 in
  let ys = Array.make (List.length seq) [] in
  List.iteri (fun i x ->
    let hp = hs.(i) in
    let z  = vec_add (mat_vec_mul rnn.wxh x)
                     (vec_add (mat_vec_mul rnn.whh hp) rnn.bh) in
    let h  = List.map tanh z in
    let y  = List.map sigmoid (vec_add (mat_vec_mul rnn.why h) rnn.
       by) in
    hs.(i + 1) <- h;
    ys.(i)     <- y
  ) seq;
  (Array.to_list ys, Array.to_list hs)
```

7.4 BPTT : Backpropagation Through Time

L'entraînement d'un RNN déroule le réseau dans le temps et propage l'erreur de la fin vers le début :

Listing 13: BPTT : rétropropagation dans le temps (`rnn.ml`)

```
let backward rnn seq cibles ys hs =
  let t = List.length seq in
  let g_wxh = ref (zero_mat n_cache n_entree) in
  let g_whh = ref (zero_mat n_cache n_cache)  in
  let g_why = ref (zero_mat n_sortie n_cache)  in
  let g_bh  = ref (zero n_cache) in
  let g_by  = ref (zero n_sortie) in
  let dh_next = ref (zero n_cache) in

  for i = t - 1 downto 0 do
    let x  = List.nth seq i in
    let hp = List.nth hs i in                    (* h_{t-1} *)
    let h  = List.nth hs (i + 1) in              (* h_t     *)
    let y  = List.nth ys i in
    let cible = List.nth cibles i in

    (* dL/dy = y - cible  (MSE) *)
    let dy = vec_sub y cible in

    (* contribution aux gradients de sortie *)
    let d_why = List.map (fun dy_j -> vec_smul dy_j h) dy in
    let d_by  = dy in
    g_why := matrix_add !g_why d_why;
    g_by  := vec_add !g_by d_by;

    (* rétropropagation dans l'état caché *)
```

```
    (* dh = (Why^T * dy) + dh_next  puis multiplié par dtanh(h) *)
    let why_t = List.init n_cache (fun j ->
      List.map (fun row -> List.nth row j) rnn.why
    ) in
    let dh_raw = vec_add (mat_vec_mul why_t dy) !dh_next in
    let dh = vec_mul dh_raw (List.map dtanh h) in

    (* gradients pour les poids cachés *)
    let d_wxh = List.map (fun dh_j -> vec_smul dh_j x) dh in
    let d_whh = List.map (fun dh_j -> vec_smul dh_j hp) dh in
    let d_bh  = dh in
    g_wxh := matrix_add !g_wxh d_wxh;
    g_whh := matrix_add !g_whh d_whh;
    g_bh  := vec_add !g_bh d_bh;

    (* propager au pas précédent *)
    dh_next := mat_vec_mul rnn.whh dh
  done;
  (!g_wxh, !g_whh, !g_why, !g_bh, !g_by)
```

Le principe : pour chaque pas de temps t (de la fin vers le début), on accumule les gradients par rapport aux poids. La variable `dh_next` propage l'erreur à travers le temps via W_{hh}.

7.5 Tâche : parité binaire

Le RNN apprend à prédire la parité d'une séquence de bits :

$$[1, 1, 0, 1] \rightarrow [1, 0, 0, 1]$$

Chaque sortie y_t indique si le nombre de 1 vus depuis le début est pair (0) ou impair (1). C'est un problème simple en apparence, mais qui **exige une mémoire interne** : impossible à résoudre sans état caché.

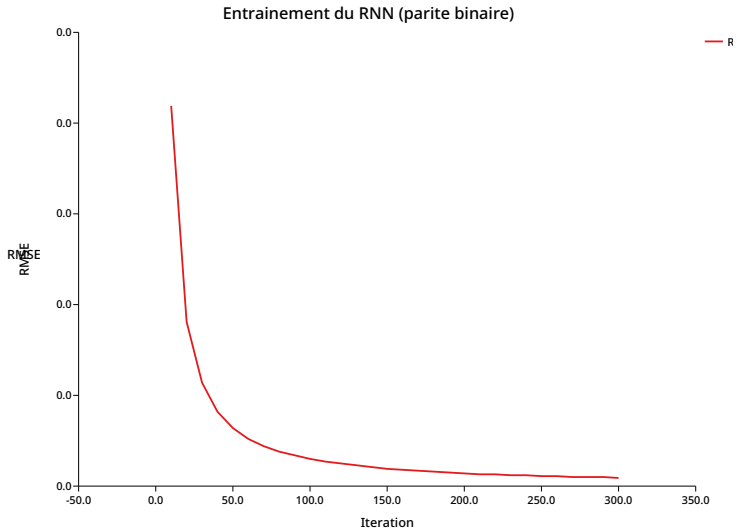


Figure 7: Courbe d'apprentissage du RNN. La RMSE descend à 10^{-5}, le RNN apprend parfaitement la parité.

8 Réseau Convolutionnel (CNN)

8.1 Pourquoi un CNN ?

Une image de 256×256 pixels, c'est 65 536 entrées. Connecter chaque neurone à tous les pixels crée des millions de paramètres. La convolution résout ce problème avec deux idées-clés :

1. **Partage des poids** : un même filtre est appliqué à toutes les positions de l'image.
2. **Connexions locales** : chaque neurone ne voit qu'une petite région de l'image (son champ récepteur).

8.2 La convolution 2D

$$Y[i,j,k] = \sum_{m=0}^{hk-1} \sum_{n=0}^{wk-1} X[i+m, j+n] \cdot K_k[m,n] + b_k$$

Soit en OCaml :

Listing 14: Convolution 2D : pour chaque filtre k, on glisse le noyau sur l'image et on calcule le produit scalaire (`cnn.ml`)

```
78 let conv2d_forward (c : conv2d) image =
79   (* image : hi × wi  (liste de listes) *)
80   let hi = List.length image in
81   let wi = List.length (List.hd image) in
82   let ho = hi - c.hk + 1 in
83   let wo = wi - c.wk + 1 in
84   List.init c.n_filtres (fun k ->
85     let noyau = List.nth c.noyaux k in
86     List.init ho (fun i ->
87       List.init wo (fun j ->
```

```
          let somme = ref (List.nth c.biais k) in
          for di = 0 to c.hk - 1 do
            for dj = 0 to c.wk - 1 do
              let vi = List.nth (List.nth image (i + di)) (j + dj) in
              let wk_val = List.nth (List.nth noyau di) dj in
              somme := !somme +. vi *. wk_val
            done
          done;
          !somme
        )
      )
    )
    (* résultat : n_filtres × ho × wo *)
```

8.3 Max Pooling

Après la convolution et ReLU, le **max pooling** réduit la taille des cartes en ne gardant que la valeur maximale par fenêtre 2×2 :

Listing 15: Max Pooling 2×2 avec stride 2 : on divise la taille par 2 (`cnn.ml`)

```
let max_pool_forward cartes =
  (* cartes : n_filtres × hi × wi *)
  let n_filtres = List.length cartes in
  let hi = List.length (List.hd cartes) in
  let wi = List.length (List.hd (List.hd cartes)) in
  let ho = hi / 2 in
  let wo = wi / 2 in
  List.map (fun carte ->
    List.init ho (fun i ->
      List.init wo (fun j ->
        let a = List.nth (List.nth carte (2*i)) (2*j) in
        let b = List.nth (List.nth carte (2*i)) (2*j+1) in
        let c = List.nth (List.nth carte (2*i+1)) (2*j) in
        let d = List.nth (List.nth carte (2*i+1)) (2*j+1) in
        max (max a b) (max c d)
      )
    )
  ) cartes
```

8.4 L'architecture complète

Notre CNN pour distinguer bandes verticales / horizontales :

Listing 16: Initialisation et entraînement du CNN (`cnn.ml`)

```
let () =
  Printf.printf "=== CNN : Classification de motifs ===\n\n%!";
  Printf.printf "  Architecture :\n";
  Printf.printf "    Entrée : 6×6\n";
  Printf.printf "    Conv2D : 2 filtres 3×3 + ReLU\n";
  Printf.printf "    MaxPool : 2×2 (stride 2)\n";
```

```
  Printf.printf "␣␣␣␣FC␣␣␣␣␣␣:␣8␣↦␣2␣(softmax)\n\n%!";

  let conv = cree_conv2d 2 3 3 in
  let fc   = cree_fc 8 2 in
  let donnees = creer_donnees 30 in

  Printf.printf "␣␣%d␣images␣d'entraînement\n%!" (List.length
     donnees);

  Printf.printf "\n␣␣␣Exemples␣de␣motifs␣:\n%!";
  Printf.printf "\n␣␣␣---␣Bande␣verticale␣(classe␣0)␣---\n%!";
  affiche_image (List.hd (List.filter (fun (_, c) -> c = 0) donnees
     ) |> fst);
  Printf.printf "\n␣␣␣---␣Bande␣horizontale␣(classe␣1)␣---\n%!";
  affiche_image (List.hd (List.filter (fun (_, c) -> c = 1) donnees
     ) |> fst);

  Printf.printf "\n␣␣␣Entraînement␣en␣cours...\n%!";
  entrainer conv fc donnees 1.0 100;
```

8.5 Le passage backward complet

La rétropropagation pour un CNN implique trois gradients différents :

Listing 17: Backward FC : $dL/dW = \delta \cdot h^T$, $dL/dh = W^T \cdot \delta$ (`cnn.ml`)

Gradient pour la couche fully connected :

```
let backward_fc (f : fc) (c : cache) cible taux =
  (* dL/dlogits = probs - one_hot(cible) *)
  let n_c = List.length c.probs in
  let d_logits = List.init n_c (fun i ->
    List.nth c.probs i -. if i = cible then 1. else 0.
  ) in
  let d_poids = List.map (fun d_j ->
    List.map (fun x_i -> d_j *. x_i) c.plat
  ) d_logits in
  let d_biais = d_logits in
  let d_plat = List.init (List.length c.plat) (fun i ->
    List.fold_left2 (fun s w_ji d_j -> s +. w_ji *. d_j) 0.
      (List.map (fun row -> List.nth row i) f.poids) d_logits
  ) in
  (* Mise à jour des poids FC *)
  f.poids <- List.map2 (fun row d_row ->
    vec_sub row (vec_smul taux d_row)
  ) f.poids d_poids;
  f.biais <- vec_sub f.biais (vec_smul taux d_biais);
  d_plat
```

Listing 18: Backward MaxPool : le gradient passe uniquement par l'élément maximum de chaque fenêtre (`cnn.ml`)

Gradient pour le max pooling :

```
let backward_pool (c : cache) d_plat =
  let n_filtres = List.length c.conv_a in
  let hi = List.length (List.hd c.conv_a) in
  let wi = List.length (List.hd (List.hd c.conv_a)) in
  let d_carte = Array.init n_filtres (fun _ ->
    Array.init hi (fun _ -> Array.make wi 0.)
  ) in
  let idx_plateau = ref 0 in
  List.iteri (fun k idxs ->
    List.iteri (fun i ligne ->
      List.iteri (fun j (di, dj) ->
        let val_d = List.nth d_plat !idx_plateau in
        d_carte.(k).(2*i+di).(2*j+dj) <-
          d_carte.(k).(2*i+di).(2*j+dj) +. val_d;
        incr idx_plateau
      ) ligne
    ) idxs
  ) c.pool_idx;
  List.init n_filtres (fun k ->
    List.init hi (fun i ->
      List.init wi (fun j -> d_carte.(k).(i).(j))
    )
  )
```

Listing 19: Backward Conv2D : $dL/dK = X \star \delta$, $dL/dX = \delta \star \mathrm{rot180}(K)$ (`cnn.ml`)

Gradient pour la convolution :

```
let backward_conv (c : conv2d) (cache : cache) d_conv taux =
  let hi = List.length cache.image in
  let wi = List.length (List.hd cache.image) in
  (* dL/dnoyau *)
  let d_noyaux = List.init c.n_filtres (fun k ->
    List.init c.hk (fun di ->
      List.init c.wk (fun dj ->
        let somme = ref 0. in
        for i = 0 to hi - c.hk do
          for j = 0 to wi - c.wk do
            let dy = List.nth (List.nth (List.nth d_conv k) i) j in
            let x  = List.nth (List.nth cache.image (i + di)) (j +
               dj) in
            somme := !somme +. dy *. x
          done
        done;
        !somme
      )
    )
  ) in
  let d_biais = List.map (fun carte ->
    List.fold_left (fun s ligne ->
      s +. List.fold_left (+.) 0. ligne
    ) 0. carte
  ) d_conv in
```

```
  (* dL/dimage (pour info, pas utilisée ici) *)
  let d_image = List.init hi (fun i ->
    List.init wi (fun j ->
      let somme = ref 0. in
      for k = 0 to c.n_filtres - 1 do
        let noyau = List.nth c.noyaux k in
        for di = 0 to c.hk - 1 do
          for dj = 0 to c.wk - 1 do
            let pi = i - di in
            let pj = j - dj in
            if pi >= 0 && pi < hi - c.hk + 1 &&
               pj >= 0 && pj < wi - c.wk + 1 then
              let dy = List.nth (List.nth (List.nth d_conv k) pi)
                 pj in
              let w  = List.nth (List.nth noyau di) dj in
              somme := !somme +. dy *. w
          done
        done
      done;
      !somme
    )
  ) in
  (* Mise à jour des noyaux *)
  c.noyaux <- List.map2 (fun noyau d_no ->
    List.map2 (fun ligne d_ligne ->
      vec_sub ligne (vec_smul taux d_ligne)
    ) noyau d_no
  ) c.noyaux d_noyaux;
  c.biais <- vec_sub c.biais (vec_smul taux d_biais);
  d_image
```

8.6 Résultats

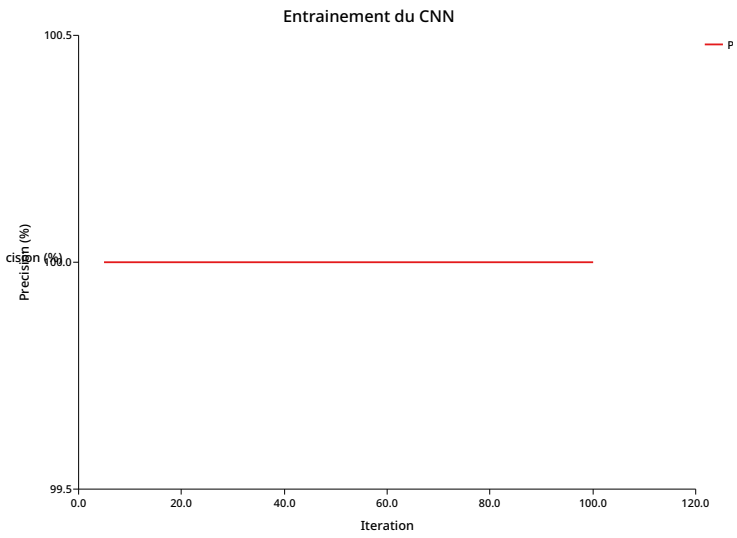


Figure 8: Précision du CNN : 100% atteint en moins de 20 itérations.

Figure 9: Filtres convolutifs 3×3 appris par le réseau. Chaque case représente un poids (rouge = positif, bleu = négatif).

9 Génération de Graphiques SVG

Tous les graphiques de ce cours sont générés automatiquement par le module `graphiques.ml`. Ce module produit des fichiers SVG (vectoriels) directement depuis OCaml.

Listing 20: Fonction de tracé de courbes d'apprentissage (`graphiques.ml`)

```
138 let courbe_entrainement ?(fichier = "courbe.svg") ?(titre = "")
139     ?(xlab = "Itération") ?(ylab = "Erreur")
140     courbes =
141   let oc = open_out fichier in
142   balise oc;
143
144   let (mx, my, _, _, _, _) =
145     if courbes = [] then ((fun x -> 0.), (fun y -> 0.), 0., 1., 0.,
          1.) else
146     let xs = List.concat_map (fun c -> List.map fst c.points)
          courbes in
147     let ys = List.concat_map (fun c -> List.map snd c.points)
```

```
        courbes in
    let xmin = List.fold_left min max_float xs in
    let xmax = List.fold_left max (-.max_float) xs in
    let ymin = List.fold_left min max_float ys in
    let ymax = List.fold_left max (-.max_float) ys in
    let xpad = (xmax -. xmin) *. 0.05 in
    let ypad = (ymax -. ymin) *. 0.1 in
    trace_axes oc (xmin -. xpad) (xmax +. xpad)
                  (max 0. (ymin -. ypad)) (ymax +. ypad)
                  xlab ylab
  in

  if titre <> "" then
    texte oc (float largeur /. 2.) 25. "middle" 18 "black" titre;

  List.iteri (fun i c ->
    let pts = List.map (fun (x, y) -> (mx x, my y)) c.points in
    polyline oc pts palette.(i mod Array.length palette) 2
  ) courbes;

  let y_leg = float marge_h +. 10. in
  List.iteri (fun i c ->
    let x_leg = float (largeur - marge_d + 10) in
    ligne oc x_leg (y_leg +. float i *. 20.) (x_leg +. 20.) (y_leg
       +. float i *. 20.)
      2 palette.(i mod Array.length palette);
    texte oc (x_leg +. 25.) (y_leg +. 5. +. float i *. 20.) "start"
       12 "black" c.legende
  ) courbes;

  fermer oc;
  close_out oc;
  Printf.printf "␣␣->␣graphique␣sauvegardé␣:␣%s\n%!" fichier
```

Les fonctions disponibles sont :

- `courbe_entrainement` : courbes d'apprentissage (une ou plusieurs)
- `carte_chaleur` : heatmap pour visualiser les filtres convolutifs
- `nuage` : nuage de points 2D avec séparation par classe
- `barres` : diagramme en barres pour comparer des valeurs

Guide d'utilisation

Prérequis : OCaml 5.x, `cairosvg` (pour convertir les SVG en PDF).

Compiler la bibliothèque de graphiques :

```
ocamlc -c graphiques.ml
```

Exécuter un programme avec graphiques :

```
ocaml graphiques.cmo autoencoder.ml
```

Exécuter sans graphiques (mode minimal) :

```
ocaml autoencoder.ml
```

Compiler le cours :

```
python3 -c "import␣cairosvg;␣[cairosvg.svg2pdf(url=f'{f}.svg',
␣␣write_to=f'{f}.pdf')␣for␣f␣in␣['regression_donnees',
␣␣'regression_courbe',␣'regression_comparaison',␣'regression_sin',
␣␣'classifieur_loss',␣'classifieur_nuage',␣'classifieur_frontiere',
␣␣'autoencodeur_loss',␣'autoencodeur_codes',␣'rnn_loss',
␣␣'cnn_precision',␣'cnn_filtre_0',␣'cnn_filtre_1']]"
pdflatex cours_reseaux_neurones.tex
```

10 Conclusion

10.1 Ce qu'on a appris

1. Un **réseau de neurones**, c'est juste des multiplications de matrices avec des non-linéarités entre les couches. Tout le reste n'est que détails d'implémentation (importants, certes).

2. La **rétropropagation**, c'est la règle de dérivation en chaîne bien appliquée. Les 10 000 lignes de PyTorch ne font que ça, mais en plus rapide.

3. Les **autoencodeurs** compressent l'information : utiles pour la réduction de dimension, le débruitage, la détection d'anomalies.

4. Les **RNN** apportent une mémoire : essentiels pour les séquences.

5. Les **CNN** réduisent drastiquement le nombre de paramètres pour les images grâce au partage des poids et à la connexion locale.

10.2 La blague finale

Un data scientist entre dans un bar. Il commande une bière.
Le barman sert 0,7 verre.
«Pourquoi seulement 0,7 ? demande le data scientist. »
«C'est le taux d'apprentissage optimal pour ne pas finir ivre mort après 3 pintes. »
Le data scientist réfléchit et dit :
«Vous devriez augmenter le nombre d'epochs. »
Le barman le regarde :
«Je sers des bières, pas des réseaux de neurones. »
Le data scientist : «C'est la même chose. »
— Blague de data scientist

10.3 Pour aller plus loin

- **LSTM / GRU** : RNN avec mécanismes d'oubli.
- **Transformers** : «Attention is all you need ».
- **GANs** : deux réseaux en compétition (générateur vs discriminateur).
- **Apprentissage par renforcement** : le réseau apprend par essais et erreurs, comme un humain (mais en moins long).

Fin du cours 4

Merci d'avoir suivi jusqu'au bout.

N'oubliez pas : `ocaml graphiques.cmo rnn.ml` *ça marche.*

(Et si ça ne marche pas, relisez le code.)