Cours 4

Réseau de Neurones en OCaml

Introduction

Cours précédent (cours 3) :

- Monade `option` (gestion d'erreur)
- Monade `Writer` (journalisation)
- Pattern général `retour` / `»=`

Ce cours : on implémente un mini-réseau de neurones en OCaml pur : neurone, couche, réseau, gradient, descente.

Un réseau de neurones, c'est une tonne de multiplications de matrices avec un zeste de calcul différentiel. En OCaml, ça tient en 50 lignes.

Neurone artificiel

$f(\mathbf{x}, \mathbf{w}, b) = \sigma(\sum_i w_i x_i + b)$

```
type neurone = { w : float list; b : float }

let sigmoid x = 1.0 /. (1.0 +. exp (-. x))

let forward n x =
  let sum = List.fold_left (+.) 0.0
    (List.map2 ( *. ) n.w x) in
  sigmoid (sum +. n.b)
```

▶ **À faire :** Calculez la sortie du neurone $\mathbf{w} = [0.5; -0.3]$, $b = 0.1$ pour $\mathbf{x} = [1.0; 2.0]$. Vérifiez à la main.

Propagation avant (forward)

Une couche = une liste de neurones.

```
type layer = { neurones : neurone list }

let forward_layer l x =
  List.map (fun n →forward n x) l.neurones
```

Réseau = liste de couches.

```
type reseau = { couches : layer list }

let forward_network r x =
  List.fold_left (fun x layer →
    forward_layer layer x) x r.couches
```

▶ **À faire :** Écrivez `exemple` : réseau à 2 entrées, couche cachée de 3 neurones, couche sortie de 1 neurone. Testez avec `[0.5;0.8]`.

Fonction de coût (loss)

On mesure l'erreur entre prédiction et cible :

```
let mse pred target =
  List.map2 (fun p t →(p -. t) ** 2.0) pred target
  |>(fun l →List.fold_left (+.) 0.0 l)
  |>(fun s →s /. float_of_int (List.length pred))
```

Erreur quadratique moyenne (MSE) : $\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(p_i - t_i)^2$

▶ **À faire :** Calculez la MSE pour `pred = [0.8; 0.2]`, `target = [1.0; 0.0]`.

Gradient : dérivée de la sigmoïde

Pour corriger les poids, on suit le gradient.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad \sigma'(x) = \sigma(x) \times (1 - \sigma(x))$$

```
let sigmoid_deriv sigma = sigma *. (1.0 -. sigma)
```

Pourquoi c'est utile : $\sigma'(x)$ s'exprime facilement à partir de $\sigma(x)$.

▶ **À faire :** Vérifiez : si $\sigma(x) = 0.8$, que vaut $\sigma'(x)$? Si $\sigma(x) = 0.5$, que vaut $\sigma'(x)$? Que remarquez-vous ?

Gradient pas à pas

Pour un neurone de sortie :

- Erreur : $\delta_o = (p - t) \cdot \sigma'(x)$
- Mise à jour poids : $w_i \leftarrow w_i - \alpha \cdot \delta_o \cdot x_i$
- Mise à jour biais : $b \leftarrow b - \alpha \cdot \delta_o$

```
let gradient_sortie n x p t =
  let delta = (p -. t) *. sigmoid_deriv p in
  let dw = List.map (fun xi →-. alpha *. delta *. xi) x in
  let db = -. alpha *. delta in
  (dw, db)
\end{listing}
\exercisebox{Calculez $\delta_o$ pour $p=0.8$, $t=1.0$.
  Quel est le signe de la correction sur $w_i$~?}
\fin

\slide
\sectionbox{R\'etropropagation␣(backpropagation)}
\textbf{Pour␣une␣couche␣cach\'ee,␣$\delta_j$␣se␣propage~:}
```

```
$$\delta_j = \sigma'(x_j) \cdot \sum_k \delta_k \cdot w_{jk}$$
\begin{lstlisting}
let gradient_cache n inputs deltas_next w_next =
  let sum = List.fold_left2 (fun acc d w →
    acc +. d *. w) 0.0 deltas_next w_next in
  let out = forward n inputs in
  let delta = sum *. sigmoid_deriv out in
  let dw = List.map (fun i →-. alpha *. delta *. i) inputs in
  let db = -. alpha *. delta in
  (dw, db, delta)
```

▶ **À faire :** Expliquez la ligne `sum` : que représente-t-elle mathématiquement ?

Descente de gradient stochastique (SGD)

Algorithme complet :

1. Forward : calculer toutes les sorties
2. Backward : calculer les δ de la sortie vers l'entrée
3. Mettre à jour tous les poids et biais
4. Recommencer (epoch)

▶ **À faire :** Combien de poids total pour un réseau $2 \to 3 \to 1$?
$2 \times 3 + 3 \times 1 + 3 + 1 = 13$ paramètres.

SGD en OCaml : le squelette

```
let rec train r data epochs alpha =
  if epochs <= 0 then r
  else
    let r' = List.fold_left (fun r (x, t) →
      let activations = forward_all r x in
      let gradients = backward_all r x activations t alpha in
      update_weights r gradients
    ) r data in
    train r' data (epochs - 1) alpha
```

`forward_all` retourne toutes les activations. `backward_all` calcule les gradients. `update_weights` applique les corrections.

▶ **À faire :** Pourquoi `train` est-elle récursive ? Quelle serait la version itérative ?

Test : XOR avec un réseau $2 \to 2 \to 1$

```
let data = [
  ([0.0;0.0], [0.0]); ([0.0;1.0], [1.0]);
  ([1.0;0.0], [1.0]); ([1.0;1.0], [0.0]);
]

let reseau_xor = init_reseau [2; 2; 1]
let trained = train reseau_xor data 10000 0.5
```

Après entraînement :

```
# forward_network trained [1.0;0.0] →~0.95 (proche de 1)
# forward_network trained [0.0;0.0] →~0.05 (proche de 0)
```

▶ **À faire :** Testez le XOR appris. Pourquoi un seul neurone ne peut-il pas résoudre XOR ?

Conclusion

Points clés du cours 4 :

- Un neurone = combinaison linéaire + sigmoïde
- Forward : composition de fonctions
- Backprop : règle de dérivation en chaîne
- SGD : itérer sur les données
- Tout le code tient en une centaine de lignes OCaml

Félicitations ! Vous avez survécu aux 4 cours. Vous êtes capables d'écrire un réseau de neurones en OCaml pur, sans bibliothèque externe, sans boucle while, sans effet de bord.

Si vous pouvez faire du machine learning en OCaml pur, vous pouvez tout faire en OCaml pur. Maintenant, allez coder.

Exercices

1. Codez `init_reseau` qui génère un réseau avec des poids aléatoires entre $[-1, 1]$.

2. Codez `forward_all`, `backward_all`, `update_weights`.

3. Entraînez sur XOR. Tracez l'erreur au fil des epochs.

4. Remplacez sigmoïde par ReLU : $f(x) = \max(0, x)$. Recalculez la dérivée.

5. Ajoutez une couche cachée supplémentaire ($2 \to 4 \to 2 \to 1$). Le réseau converge-t-il mieux ?

6. ImplémenteZ `accuracy` et mesurez la précision sur le jeu de test.

7. Faites un mini-réseau pour reconnaître les chiffres (dataset `AND` : $[0;0] \rightarrow 0, [0;1] \rightarrow 0, [1;0] \rightarrow 0, [1;1] \rightarrow 1$).