RDFIA TP1 : Introduction aux réseaux de neurones

Taylor Mordan, Thomas Robert, Matthieu Cord

19 octobre 2016

Pour les TP sur les réseaux de neurones, vous devrez rendre :

— Un compte rendu à la fin de chaque séance (manuscrit ou par mail à `taylor.mordan@lip6.fr` ou `thomas.robert@lip6.fr` selon votre encadrant de TP, les compte-rendus en retard ne seront pas acceptés). Ce compte-rendu devra décrire le travail effectué pendant la séance et les résultats importants obtenus.

— Un compte rendu global une semaine après la dernière séance (par mail), dont le but est de reprendre au propre le travail effectué pendant l'intégralité des séance de TP, en prenant du recul sur ce qui a été fait. Ce compte rendu représentera la majorité de la note de TP.

Les données et une version numérique du sujet sont accessibles
à l'adresse `http://webia.lip6.fr/~robert`

1 Objectifs

Le but du TP est de mettre en place un réseau de neurones simple afin de se familiariser avec ces modèles et la façon de les entrainer : la *backpropagation* (ou rétropropagation) du gradient.

Pour cela, nous allons implémenter un perceptron à une couche cachée et mettre en place la procédure d'apprentissage. On essayera d'abord notre perceptron sur un problème jouet pour vérifier son bon fonctionnement puis on l'appliquera au jeu de données classique MNIST.

Le site `http://playground.tensorflow.org` permet de visualiser le fonctionnement et l'apprentissage de petits réseau de neurones. Vous pouvez vous y rendre afin de mieux appréhender ce TP.

2 Rappels

Pour appliquer un perceptron à un problème de *machine learning* (en apprentissage supervisé), on a besoin de 4 choses :

— Un jeu de données et un problème associé ;

— L'architecture du réseau (à adapter aux données) ;

— Une fonction de coût que l'on cherche à minimiser (à adapter au problème) ;

— Un algorithme d'apprentissage pour résoudre le problème d'optimisation consistant à minimiser cette fonction de coût.

2.1 Jeu de données

Un jeu de données pour un problème de classification en apprentissage supervisé est constitué d'un ensemble de N couples $(x^{(i)}, y^{(i)}), i \in 1..N$ où $x^{(i)} \in \mathbb{R}^{n_x}$ est un vecteur de *features*

à partir duquel on veut prédire la vérité terrain $y^{(i)} \in \{0,1\}^{n_y}$ (indiquant la classe à laquelle appartient l'exemple).

Ce jeu de données est ensuite découpé en plusieurs ensembles : *train*, *test* et parfois *val*.

2.2 Architecture du réseau

Notre réseau sera constitué de :
- Un vecteur d'entrée de taille n_x ;
- Une couche cachée produisant un vecteur h de taille n_h avec :
  - Une matrice de poids W_h de taille $n_h \times n_x$;
  - Un biais b_h de taille n_h ;
  - La fonction d'activation tanh.
- Une couche de sortie produisant un vecteur $\hat{y}$ de taille n_y :
  - Une matrice de poids W_y de taille $n_y \times n_h$;
  - Un biais b_y de taille n_y ;
  - La fonction d'activation SoftMax.

2.3 Fonction de coût

Comme fonction de coût (*loss*), nous utiliserons l'entropie croisée. Pour un exemple seul de vecteur de vérité terrain y et de vecteur de prédiction $\hat{y}$, on a le coût unitaire :

$$\ell(y, \hat{y}) = -\sum_i y_i \log \hat{y}_i$$

On calcule généralement le coût pour un ensemble de données (X, Y) comme la somme des coûts unitaires :

$$\mathcal{L}(X, Y) = \sum_i \ell(y^{(i)}, \hat{y}^{(i)})$$

où $\hat{y}^{(i)}$ correspond à la sortie du réseau pour l'exemple $x^{(i)}$.

2.4 Méthode d'apprentissage

Backpropagation Pour rappel, le principe de *backpropagation* consiste à appliquer le théorème de dérivation des fonctions composées (*chain rule* en anglais) :

$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b}\frac{\partial b}{\partial a}.$$

Dans le cas vectoriel, si a, b, c sont des vecteurs, on a pour la dérivée de la composante c_i par rapport à la composante a_k :

$$\frac{\partial c_i}{\partial a_k} = \sum_j \frac{\partial c_i}{\partial b_j}\frac{\partial b_j}{\partial a_k}.$$

On visualiser schématiquement l'application de la *backpropagation* sur un réseau de neurones sur la figure 1.

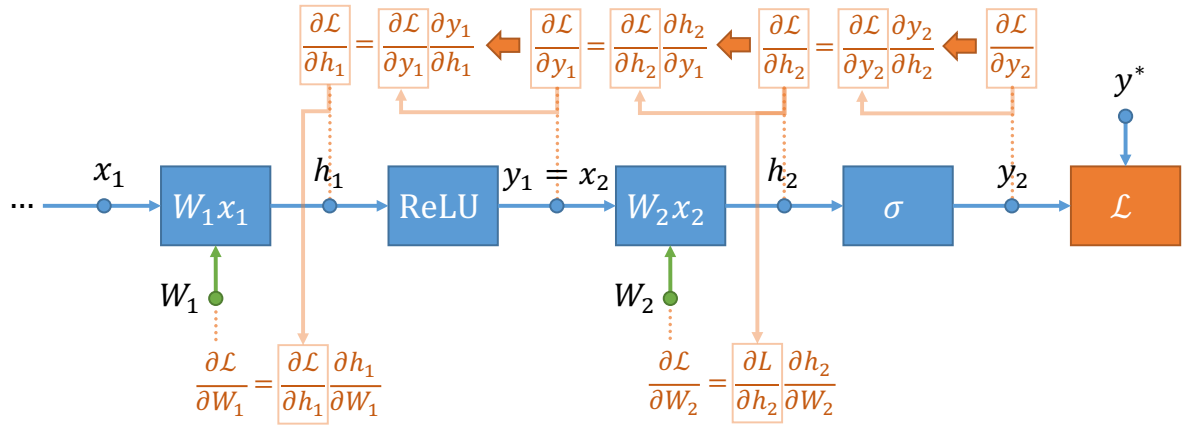

FIGURE 1 – Vue schématique de la backpropagation sur un réseau de neurones. On commence par calculer la dérivée du coût $\mathcal{L}$ par rapport à la sortie (y_2 sur le schéma) puis on remonte dans le réseau en réutilisant les dérivées calculées précédemment lors du calcul des nouvelles dérivées.

Descente de gradient La descente de gradient consiste à calculer la dérivée de la fonction de coût par rapport à un paramètre w, et à faire un *pas* (aussi appelé *learning rate*) η dans la direction du gradient, c'est à dire à modifier la valeur de w de η dans la direction du gradient :

$$w \leftarrow w - \eta \frac{\partial \mathcal{L}(X, Y)}{\partial w}$$

Dans notre cas, l'intégralité des matrices de poids et des biais sont concernés par cette procédure et devront être mis à jour ainsi à chaque passe de *backpropagation*.

Le gradient $\dfrac{\partial \mathcal{L}(X, Y)}{\partial w}$ peut être calculé sur différents ensembles, ce qui correspond à différentes variantes du l'algorithme de descente de gradient :

- l'ensemble d'apprentissage (*train*) complet, c'est la descente de gradient classique ;
- un petit sous-ensemble (généralement quelques dizaines) d'exemples d'apprentissage tirés aléatoirement, c'est la descente de gradient stochastique par *mini-batch* ;
- une seule paire d'exemple $(x^{(i)}, y^{(i)})$, on remplace donc $\mathcal{L}$ par le coût unitaire ℓ, c'est la descente de gradient stochastique *online*.

3 Préparation : un peu de maths

1. Écrire les équations permettant de calculer la passe *forward* du réseau, c'est-à-dire calculer $\hat{y}$ à partir de x et des différents paramètres des couches. On explicitera au moins les étapes intermédiaires avant (on les notera $\tilde{h}$ et $\tilde{y}$) et après (h et $\hat{y}$) les fonctions d'activations.
2. En appliquant le principe de *backpropagation* à travers le réseau, écrire les dérivées du coût $\mathcal{L}$ par rapport à chacun des paramètres du réseau.
   Conseil : Ecrivez d'abord les dérivées pour chaque terme $(W_{y,ij}, b_{y,i}, W_{h,ij}, b_{h,i})$, puis essayez éventuellement de les regrouper sous la forme de gradients calculé par des opérations vectorielles.

4 Implémentation

1. Écrire une fonction `initMLP` initialisant un réseau à partir des tailles n_x, n_h et n_y. Le réseau sera stocké dans une structure de données `net` contenant les paramètres (poids et biais) du réseau. Tous les paramètres seront initialisés selon une loi normale de moyenne 0 et d'écart-type $0,3$.
   `net = initMLP(nx, nh, ny)`
2. Écrire une fonction `forward` qui calcule les sorties d'un réseau `net` pour un ensemble de k entrées x fournies sous la forme d'une matrice X de taille $k \times n_x$, qui renvoie donc une matrice $\hat{Y}$ de taille $k \times n_y$. Cette fonction retournera également une structure `out` qui contiendra les entrées, les sorties et les résultats intermédiaires dont on aura besoin par la suite pour calculer les gradients écrits dans partie précédente.
   Essayez d'écrire cette fonction avec des opérations vectorielles, sans itérer sur les différentes entrées.
   `[Yhat, out] = forward(net, X)`
3. Écrire une fonction `loss_accuracy` qui calcule la fonction de coût et la précision (taux de bonnes prédictions) pour chaque sortie d'une matrice $\hat{Y}$ (sortie de `forward`) vis-à-vis d'une matrice de vérité terrain Y de même taille, et retourne un vecteur de coût `L` de taille k et le scalaire `acc` correspondant à la prédiction.
   `[L, acc] = loss_accuracy(Yhat, Y)`
4. Écrire une fonction `backward` qui effectue une descente de gradient (grâce à la *backpropagation*) sur les paramètres d'un réseau `net` par rapport aux sorties `out` d'une passe *forward* correspondant aux vérités terrain Y. La descente de gradient utilisera un pas de gradient η.
   Cette fonction modifiera donc le contenu de `net` pour mettre à jour les paramètres du réseau.
   `net = backward(net, out, Y, eta)`

Note : Les structures en MATLAB s'utilisent comme ceci :

```
net = struct                    % initialisation d'une structure dans
                                % la variable net
net.nomChamp = randn(10, 2)     % rempli le champ "nomChamp" de la
                                % structure
tmp = net.nomChamp * 2          % meme syntaxe pour lire un champ de
                                % la structure
```

5 Application à un problème jouet

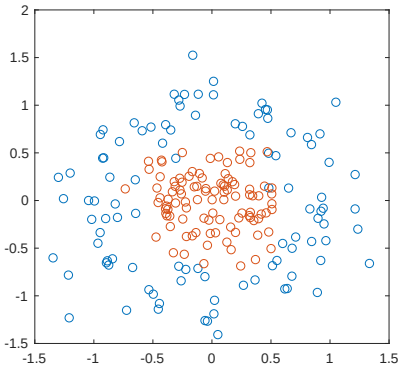

FIGURE 2 – Jeu de données *circles*

Nous allons commencer par essayer notre réseau sur un *dataset* jouet, voir figure 2. Pour cela, chargez le fichier `circles.mat` fourni. Ce fichier contient les *features* et les *targets* en *train* et en *test*, *i.e.* $X_{train}, y_{train}, X_{test}, Y_{test}$.

Apprenez et testez le modèle (voir procédure encadrée ci-dessous) en essayant différentes tailles de couche cachée n_h, différents *learning rate* η, voire différentes variantes de descente de gradient. Visualisez l'évolution du coût et de la précision au cours de l'apprentissage.

charger les données
pré-traiter les données (centrer-réduire) si nécessaire
initialiser le réseau
pour $i = 1..N_{iterations}$
    // Batch d'apprentissage, selon le type de descente de gradient
    $(X_{batch}, Y_{batch}) \leftarrow$ ensemble d'exemples de *train*
    passe forward sur X_{batch}
    passe backward avec les résultats du forward et Y_{batch}
    calcul de la loss et de l'accuracy sur (X_{batch}, Y_{batch})
    // Loss et accuracy sur tout l'ensemble de train, à faire si on utilise SGD
    si $i \ \% \ K_1 = 0$ *// toutes les K_1 (à choisir) itérations*
        calcul de la loss et de l'accuracy sur (X_{train}, Y_{train})
    // Loss et accuracy sur test
    si $i \ \% \ K_2 = 0$ *// toutes les K_2 (à choisir) itérations*
        calcul de la loss et de l'accuracy sur (X_{test}, Y_{test})
    Affichage de courbes et éventuellement de la frontière de décision

Vous pouvez également reproduire les visualisations de *Tensorflow Playground*, c'est à dire afficher une « carte de chaleur » des activations des neurones. Il suffit pour cela de créer un ensemble de données contenant des points formant une grille dans l'espace de départ, de faire passer ces points dans le réseau et d'afficher pour chaque neurone une image où la couleur de chaque pixel correspond à la valeur de l'activation du neurone pour le point correspondant au pixel.

```
% Générer une grille de points
[x1,x2] = meshgrid(-2:0.1:2);
x1 = reshape(x1, size(x1,1)^2, 1);
x2 = reshape(x2, size(x2,1)^2, 1);
Xgrid = [x1 x2];

% Affichage de la classification
[Ygrid, ~] = forward(net, Xgrid);
grid = Ygrid(:,1);
imagesc(reshape(Ygrid, sqrt(length(Ygrid)),sqrt(length(Ygrid))));
caxis([0.3, 0.7]);
plot((Xtrain(Ytrain(:,1)>.5,1) + 2) * 10, ...
  (Xtrain(Ytrain(:,1)>.5,2) + 2) * 10, 'o', 'color' , 'blue')
plot((Xtrain(Ytrain(:,2)>.5,1) + 2) * 10, ...
  (Xtrain(Ytrain(:,2)>.5,2) + 2) * 10, 'o', 'color', 'red')
```

6 Bonus : application à MNIST

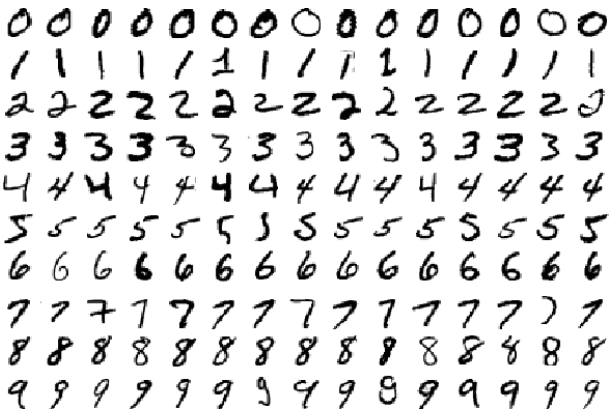

FIGURE 3 – Jeu de données MNIST

Si vous avez le temps, appliquez notre réseau au jeu de données MNIST. MNIST est un jeu d'images de chiffres manuscrits (voir figure 3), il y a donc 10 classes ($n_y = 10$). Les images font 28×28 pixels, mais elles seront représentées comme un vecteur de 784 valeurs. Pour utiliser ce jeu de données, chargez `mnist.mat`.

Pour ce jeu de données, il vaut mieux utiliser une descente de gradient stochastique par *mini-batch*. Pourquoi ?

Vous pouvez également essayer une initialisation des poids du réseau légèrement différente nommée Xavier, où les poids des matrices tirés selon une gaussienne sont divisés par la racine carrée du nombre d'entrées de la couche (*e.g.* $\sqrt{n_h}$ pour W_y), et où les poids des biais sont initialisés à 0. Qu'observez vous ?