PERCEPTION, MANIPULATION ET PROTECTION D'IMAGE

COMPTE RENDU DE TP3

Marie Guénon / Jean-Dominique Favreau / Arnaud Tanguy

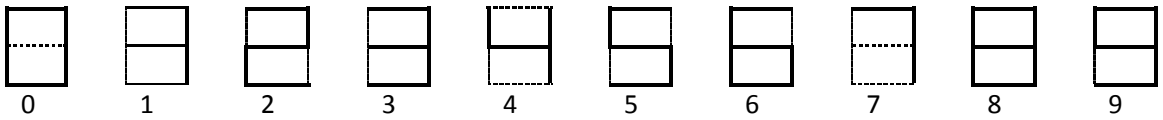
Table des matières

Description du sujet

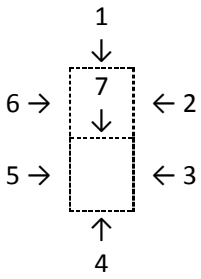
Le but de ce projet est de créer un algorithme automatique basé sur une descente de gradient et qui saurait dire si un chiffre affiché d'une manière particulière est pair ou impaire.

1. Affichage des chiffres

Les chiffes sur lesquels nous allons travailler suivent un affichage et une mise en forme bien particulière. Globalement, on peut dire que l'affichage des chiffres que nous allons utiliser suit l'affichage classique des horloges numérique. Ce qui nous donne :



Nous pouvons donc dire que chaque chiffre est constitué de 7 segments qui sont "allumé" (ici en gras) ou non et qui définit le chiffre affiché. Chaque segment a été numéroté comme suit :

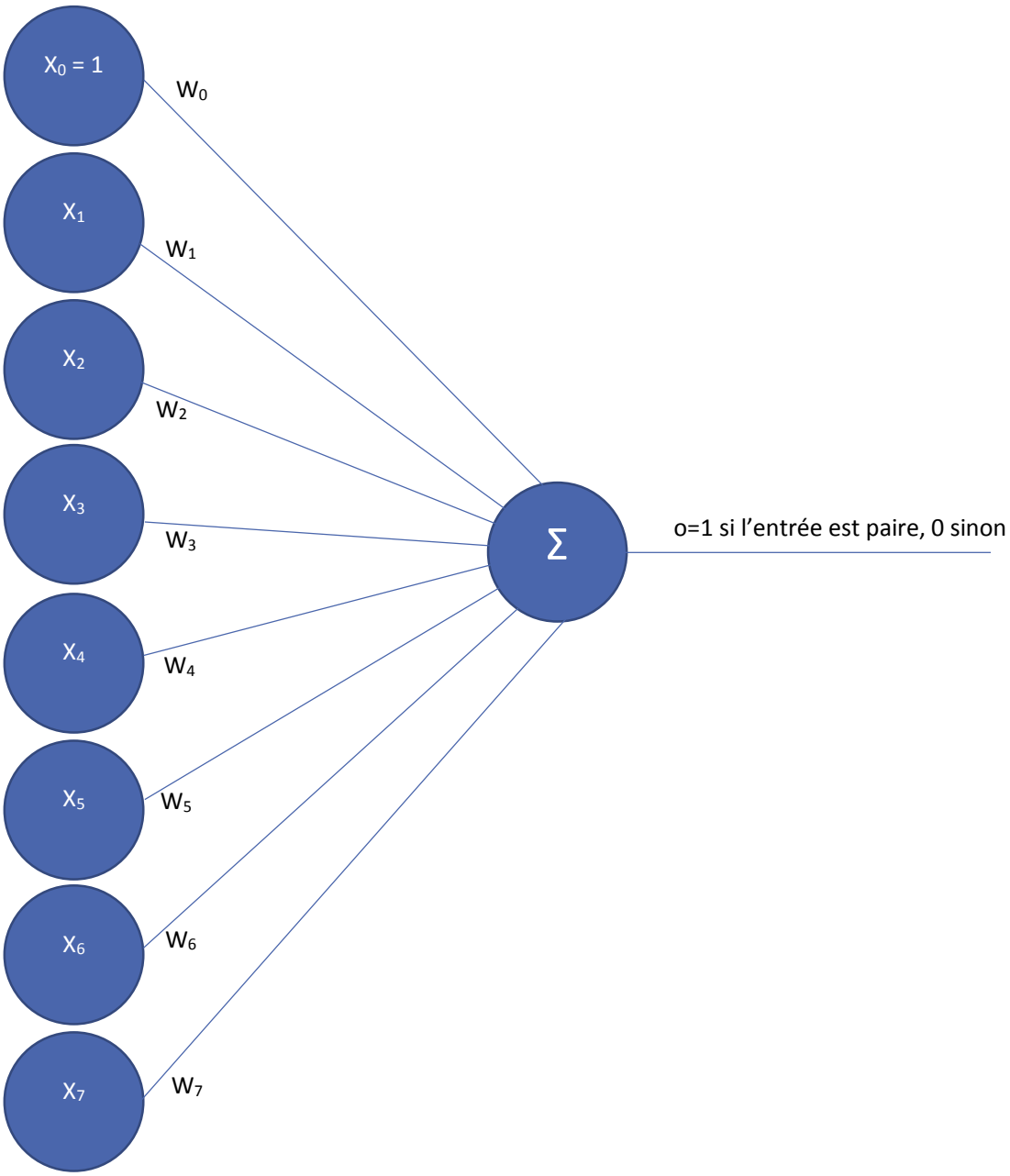


A partir de cette numérotation, on définit l'état de chaque segment de manière binaire : le segment est à 1 si il est "allumé", à 0 sinon. Puis concatène dans l'ordre l'état binaire de tous les segments. Ce qui nous donne :

0	1	2	3	4	5	6	7	8	9
1111110	0110000	1101101	1111001	0110011	1011011	1011111	1110000	1111111	1111011

2. Travail demandé

Le but est de détecter de manière automatique si les chiffres formatés comme décrit ci-dessus sont pairs ou impairs grâce à un algorithme de perceptron. Celui-ci sera de la forme suivante :



Où les x_i sont les entrées correspondant à l'état des segments et x_0 est fixé à 1.
Les w_i sont les poids attribués à chaque entrée et w_0 est le coefficient synaptique.

La sortie o est calculée selon la formule suivante :

$$o = \sum_{i=0}^{7} w_i x_i$$

Soit c la sortie qui était attendue. A chaque itération, les w_i sont recalculés suivant la formule :

$$\Delta w_i = \Delta w_i + \varepsilon(c - o)\sigma'(x_i)$$

Où ε est une constante influant la vitesse de convergence de notre algorithme.

Travail réalisé

Le travail que nous avons réalisé a été codé parallèlement en Matlab et en C++11. Cependant, les méthodes implémentées étant les même et suivant une mise en forme très similaire, elles seront expliquées simultanément.

1. Initialisation des x_i

Les x_i sont les entrées de notre algorithme. Ils représentent les différents cas possibles sur lesquels nous allons chercher à établir la parité du chiffre composé des x_i. Nous avons choisi de réunir tous les x_i dans une seule matrice

Cependant, comme il a été dit plus haut, nous allons appliquer une fonction σ à ces x_i. Nous avons choisi pour cela d'utiliser une sigmoïde :

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

Et donc de dérivée :

$$\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} * \left(\frac{e^{-x}}{1+e^{-x}}\right) = \frac{1}{1+e^{-x}} * \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right) = \frac{1}{1+e^{-x}} * \left(1 - \frac{1}{1+e^{-x}}\right)$$
$$= \sigma(x) * (1-\sigma(x))$$

Ce qui nous donne le code suivant (en Matlab à gauche, en C++ à droite) :

```
function result = sigma(x)
    result = 1./(1+exp(-x));
end

function result = sigma_p(x)
    s = sigma(x);
    %result=1;
    result = s.*(1-s);
end
```

```
auto sigma = [](double x) {
    return 1./(1.+exp(-x));
};
auto sigma_d = [&sigma](double x) {
    const double s = sigma(x);
    return s*(1-s);
};
```

Or, cette dérivée s'annule en 0, empêchant la variation des poids si l'entrée est nulle, ce pose un problème évidant et non-négligeable. C'est pourquoi, nous avons choisi d'utiliser un code binaire -1/1 pour éviter ce problème.

De plus, nous avons choisi de mettre les chiffres à apprendre dans l'ordre croissant. Ce qui nous donne les matrices suivantes (en Matlab à gauche, en C++ à droite) :

```
numbers = [1,1,1,1,1,1,1,0;
           1,0,1,1,0,0,0,0;
           1,1,1,0,1,1,0,1;
           1,1,1,1,1,0,0,1;
           1,0,0,1,0,0,1,1;
           1,1,0,1,1,0,1,1;
           1,0,0,1,1,1,1,1;
           1,1,1,1,0,0,0,1;
           1,1,1,1,1,1,1,1;
           1,1,1,1,0,0,1,1 ]
numbers = numbers * 2 - 1;
```

```
Matrix<double, rows, cols> numbers;
numbers << 1,1,1,1,1,1,1,-1,
           1,-1,1,1,-1,-1,-1,-1,
           1,1,1,-1,1,1,-1,1,
           1,1,1,1,1,-1,-1,1,
           1,-1,-1,1,-1,-1,1,1,
           1,1,-1,1,1,-1,1,1,
           1,-1,-1,1,1,1,1,1,
           1,1,1,1,-1,-1,-1,1,
           1,1,1,1,1,1,1,1,
           1,1,1,1,-1,-1,1,1;
Matrix<double, rows, 1> out;
```

2. Initialisation des wi

Nous avons choisi d'initialiser les w_i par un chiffre tiré aléatoirement dans l'intervalle [0,1]. Pour cela, nous avons utilisé la fonction *randn* sous Matlab (à gauche) et la fonction *rand* en c++ (à droite) :

```
w = rand(size(in,2),1)*2-1;
```

```
Matrix<T, Cols, 1> w;
for (int i = 0; i < Cols; i++ ) {
    w(i, 0) = (double)rand()/RAND_MAX;
}
```

3. Descente de gradient

L'algorithme de descente de gradient suit la formule :

$$\Delta w_i = \Delta w_i + \varepsilon(c - o)\sigma'(x_i)$$

Il s'agit ensuite d'itérer un certain nombre de fois (max_iter) sur tous les poids w_i pour les mettre à jour, sachant qu'au bout d'un nombre assez grand d'itérations, les w_i devraient converger et se stabiliser.

Nous obtenons alors les algorithmes suivants, à gauche en Matlab et à droite en c++ :

```
while iter < max_iter
    delta_w = delta_w * 0;
    in_w = in * w;
    for i = 1 : size(w)
        delta_w(i) = delta_w(i) + epsilon*((out-in_w)'*(in(:,i)))
        * sigma_p(in_w(i,1));
    end
    w = w + delta_w;
    iter = iter + 1;
end
```

```
while (iter < max_iter) {
    delta_w = Matrix<T, Cols, 1>::Zero();
    Matrix<T, Rows, 1> in_w = in * w;
    for (int i = 0; i < Cols; i++ ) {
            auto s = ((out-in_w).transpose()*in.col(i));
            T sigm = sigma_d(in_w(i, 0));
            s *= epsilon * sigm;
            delta_w(i, 0) += s;
    }
    w = w + delta_w;
    ++iter;
}
```

4. Constantes

Comme nous avons pu le voir plus haut, nous utilisons deux constantes : le nombre d'itérations et ε qui influence la vitesse de convergence de l'algorithme. Après quelques tests, nous avons choisi de prendre

$$\epsilon = 0.1618 \text{ et } \max\_iter = 50$$

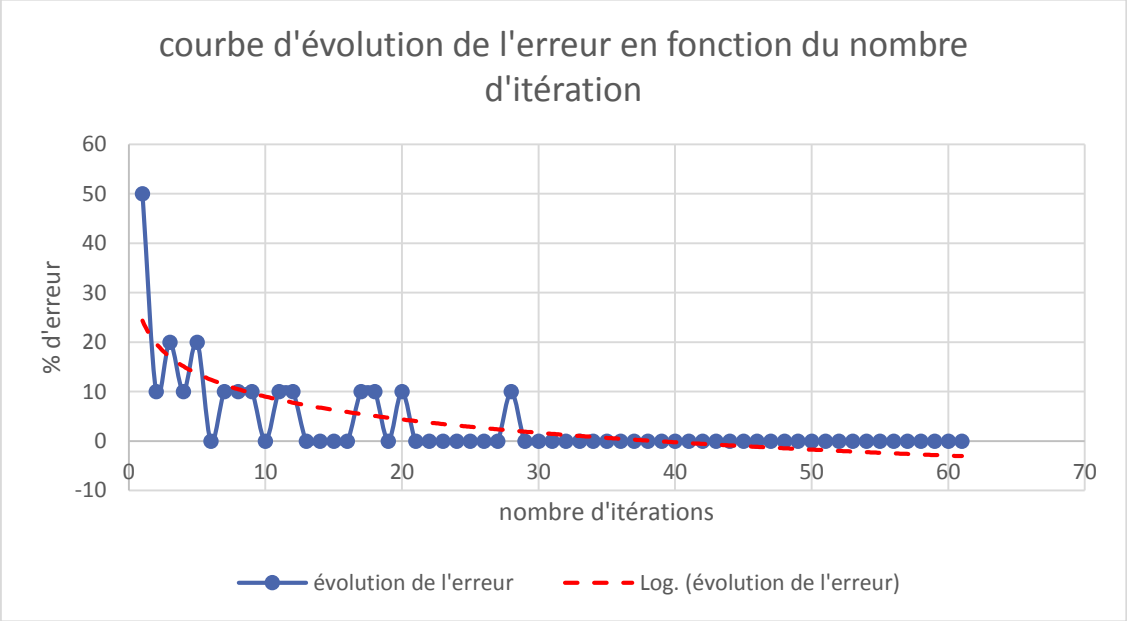
Ce choix a été motivé par une série de tests qui est détaillé ci-dessous.

Etude de l'influence du nombre d'itérations

Pour faire cette étude, nous avons décidé de lancer plusieurs fois de suite notre algorithme suivant un nombre croissant d'itérations et de calculer de manière systématique la "distance" entre le résultat obtenu et le résultat attendue. La distance a été calculée suivant la formule :

$$erreur = \frac{100 * \left(xor(entree * solution\ obtenue, solution\ attendue)\right)}{taille\ solution\ attendue}$$

Ce qui nous donne le résultat suivant avec ε= 0,1618 :

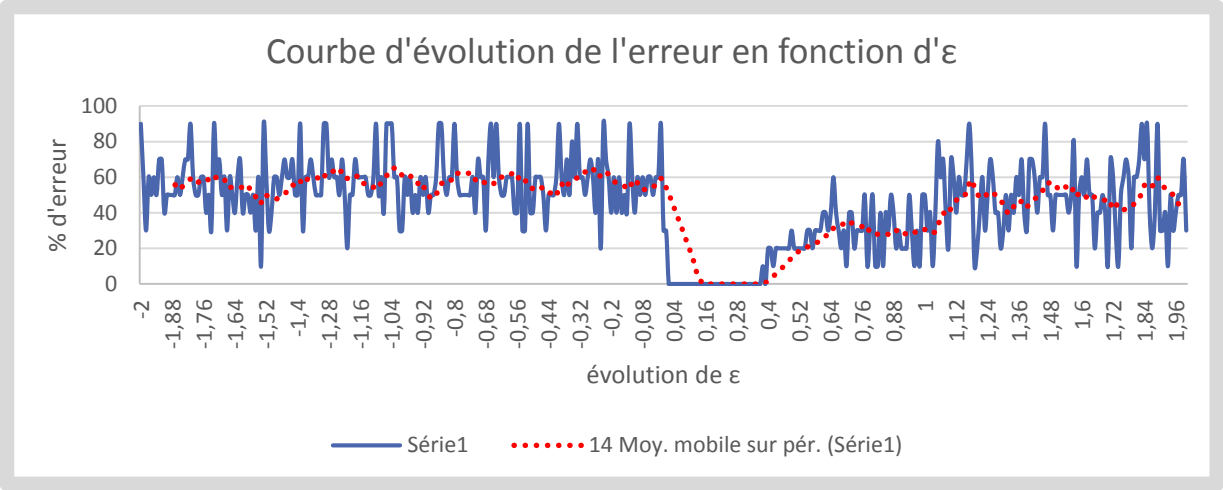


Nous pouvons constater (surtout sur la courbe de tendance, en rouge ici) que l'augmentation du nombre d'itération fait clairement baisser le pourcentage d'erreur dans le résultat. De plus, nous pouvons dire ici qu'à partir d'une quarantaine d'itérations, nous obtenons une solution stable qui n'est plus modifiée par la suite. Il est donc suffisant de prendre un maximum d'itération à 50, pour garder une marge d'erreur dans les cas les plus défavorables.

Etude de l'influence de ε

De la même manière que pour l'étude précédente, nous avons décidé de lancer plusieurs fois de suite notre algorithme suivant des valeurs croissantes d'ε et de calculer de manière systématique la "distance" entre le résultat obtenu et le résultat attendue. La distance a été calculée suivant la même formule que celle énoncée ci-dessus.

Ce qui nous donne le résultat suivant pour *max_iter*=50 :



Nous pouvons constater (surtout sur la courbe de tendance, en rouge ici) une baisse claire du pourcentage d'erreurs pour des valeurs d' ε comprises entre 0,04 et 0,4. C'est pourquoi nous avons choisi de prendre pour ε une valeur autour de la médiane de cet intervalle et donc ε=0,1618.

5. Résultats

Analyse temporelle

Dans a mesure où nous avons décidé d'implémenter notre algorithme dans deux langages de programmation différents, nous avons mesuré leur efficacité respective. Pour effectuer notre test, nous avons lancé 300 simulations ayant de pour paramètre ε= 0,1618 et *max_iter*=1000.

(Matlab à gauche, C++ à droite)

```
tic
for i=1:300
perceptron( numbers, out, .1618, 1000);
end
toc
```

```
std::chrono::time_point<std::chrono::system_clock> start, end;
start = std::chrono::system_clock::now();
for (int i = 0; i < 300; i++)
{
    perceptron<double, rows, cols>(numbers, out, sigma_d, .1618, 1000);
}
end = std::chrono::system_clock::now();

std::chrono::duration<double> elapsed_seconds = end - start;
std::time_t end_time = std::chrono::system_clock::to_time_t(end);
```

Dans ces conditions, nous obtenons un temps de 23,927583 secondes pour Matlab, contre 0,116787 en C++.

Nous pouvons donc dire que le C++ est environ 206 fois plus rapide que Matlab dans notre cas. Cette différence de performance et importante à noter, et surtout à exploiter. En effet, il n'y a ici pas beaucoup d'itérations et de données à traiter, mais si nous voulions élargir notre test à un set de données plus important et qui nécessiterait donc un temps de calcul bien plus long, il serait important de le faire en c++ pour optimiser au mieux le temps de calcul ainsi que l'utilisation des ressource de l'ordinateur.

Résistance aux variations