Projet NET4529

Etude théorique du principe de *back-propagation*

Romain MAYER Aymeric SCHNEIDER

Dans le cadre du cours NET4529 Introduction à l'apprentissage par renforcement dans les réseaux, Aymeric et moi avons choisi comme projet de travailler sur la *back-propagation* en en faisant une étude théorique basé sur deux sources bibliographiques principales :

- Paul J. Werbos (1990) *Backpropagation Through Time: What It Does and How to Do It*, Proceedings of the IEEE, VOL. 78, NO. 10
- Renjie Liao, Yuwen Xiong, Ethan Fetaya, Lisa Zhang, KiJung Yoon, Xaq Pitkow, Raquel Urtasun, Richard Zemel (2017) *Reviving and Improving Recurrent Back-Propagation*

Pour introduire notre sujet, la back-propagation est simplement un moyen efficace et une méthode exacte de calculs de toutes les dérivées d'une quantité cible par rapport à un nombre important de quantité d'entrées.

Ce travail consistera dans un premier temps en l'étude de l'algorithme de *back-propagation* (BP) basique utilisé notamment pour la reconnaissance de *pattern*, puis nous utiliseront cette étude pour étendre la méthode au systèmes dynamiques et notamment les réseaux de neurones récurrents. On parlera alors de *recurrent back-propagation* (RBP). Les algorithmes informatiques permettant d'implémenter les neurones et les méthodes feront l'objet d'une troisième partie.

Nous nous attacherons à nous munir de la plus grande rigueur mathématique possible mais n'entreront pas dans des considérations mathématiques trop poussées. De plus les mots tirés de l'anglais seront notés en italique.

Première Partie : *Back-propagation* basique

Cette partie s'attachera à décrire le principe de back-propagation simple et son fonctionnement mathématique.

I) Contexte

Lorsque l'on travaille sur de l'apprentissage supervisé via la mise en place d'un réseau de neurones, il s'agit de soumettre à un ensemble de neurones, qui ne sont autres que des systèmes d'équations pondérées, un certain nombre de données d'entrée pour obtenir une donnée de sortie désirée. Par exemple, il peut s'agir de reconnaissance de pattern : l'algorithme devra reconnaitre certains motifs sur une image. Pour entrainer cet algorithme, lui apprendre à raisonner en quelque sorte, on utilise des données de test. A ces données d'entrées correspondent des données de sortie réelles, on souhaite donc que les résultats de sortie de l'algorithme convergent vers ces résultats réels. Cette notion de convergence est en général quadratique, il s'agit en fait de minimiser l'erreur quadratique moyenne. Or pour ce faire, il faut pouvoir expliquer au réseau qu'il s'est trompé et le corriger. C'est ici qu'intervient la *back-propagation*. Cette méthode va nous permettre de corriger les poids du système au regard de l'erreur calculée.

Formalisons un peu tout ça au travers d'un exemple. Supposons que l'on dispose d'une base de données contenant 2000 exemples de codes postaux manuscrits préalablement digitalisés, sous forme de grille 19x20 composée de 0 et de 1 représentant l'image de chaque chiffre du code postal (0 pour absence d'écriture, 1 pour présence), que l'on souhaite faire reconnaitre à notre réseau. On peut donner à chaque exemple un label t entre 1 et 2000. Pour chaque échantillon t, nous disposons d'un enregistrement du modèle d'entrée et de la classification correcte. On note $x(t)$ les entrées du système. Par rapport à notre exemple, $x(t)$ serait un vecteur de 19x20=380 composantes correspondant aux 380 éléments binaires de notre pattern. En sortie, on veut obtenir un vecteur $y(t)$ de 4 composantes binaires formant le chiffre observé. On observera en pratique en sortie une estimation $\hat{y}(t)$ que l'on souhaite aussi proche que possible, au sens de l'erreur quadratique, de $y(t)$. On se donne aussi un ensemble de poids, représentés par la matrice W, qui lie les entrées aux sorties.

Par la suite, on considèrera $x(t)$ comme un vecteur à m composantes $x_1(t), \dots, x_m(t)$ et $y(t)$ un vecteur à n composantes $y_1(t), \dots, y_n(t)$, W sera une matrice de taille $N + n$, $N + n$ où N est une constante déterminant le nombre de neurones que l'on souhaite implémenter et dont la seule contrainte est d'être supérieur à m.

On remarquera que le label t pourra, dans la deuxième partie de ce document, faire référence au temps, d'où sa dénomination.

Avant de passer à la correction d'erreur à proprement parler, donnons le lien mathématique entre les entrées, les sorties et les poids.

Pour se faire, on se donne déjà une fonction d'activation, fonction qui permettra de déterminer $\hat{y}(t)$ en fonction des $x(t)$. Le choix de cette fonction dépendra du problème observé. Ici, nous utiliserons celle donnée par la source [1] :

$$s(z) = \frac{1}{1 + e^{-z}}$$

On pose alors le système suivant :

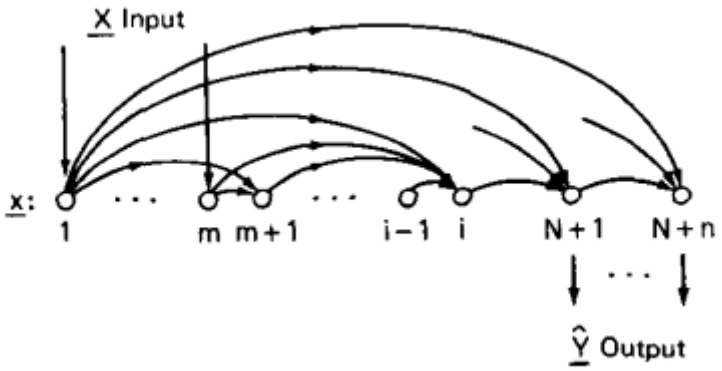
$$\forall i \in [m, N+n] \quad net_i = \sum_{j=1}^{i-1} W_{ij}\, x_i$$

$$\forall i \in [m, N+n] \quad x_i^{act} = s(net_i)$$

$$\forall i \in [1, n] \quad \hat{y}_i = x_{i+N}^{act}$$

On dira que x_i a été activé en x_i^{act} , net_i représente le niveau d'excitation électrique d'un neurone en référence aux neurosciences et au fonctionnement réel d'un neurone via ce qu'on appelle le potentiel d'action électrique qui permet aux neurones de communiquer.

Ce système peut être schématisé comme suit :

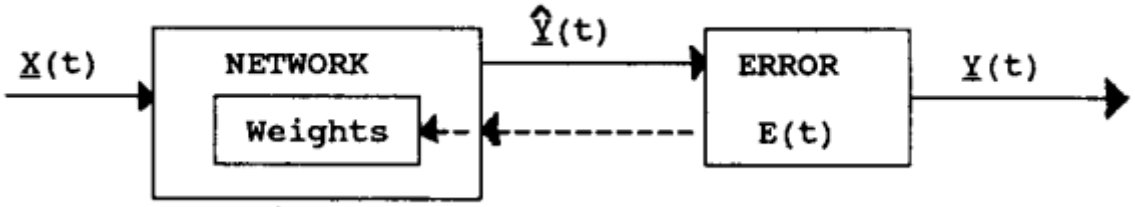


Il y a $N + n$ flèches correspondant aux $N + n$ neurones (en comptant les m premiers neurones qui sont juste uniquement des copies des entrées, dans le but de simplifier les notations). On remarque bien que la création de chaque neurone i est liée aux i-1 neurones d'avant. Chaque neurone est donc déterminé par tous ces prédécesseurs. On dit que ce réseau est entièrement connecté à l'extrême ou « fully-connected ». Cela rajoute donc un grand nombre de calcul et on préfère déconnecter les neurones entre eux en positionnant certains poids à 0. En pratique, les programmeurs préféreront disposer ces neurones en 3 couches : une couche d'entrée, une couche cachée et une couche de sortie. Il s'agit en fait de diviser les neurones en 3 groupes (m neurones d'entrée, N+n-m de neurones cachés et enfin n neurones de sortie par exemple) en mettant à 0 tous les poids de connexion entre les neurones successifs sauf les deux aux frontières des couches. Mathématiquement, cela veut dire que le neurone i ne sera pas déterminés par ces prédécesseurs récursivement.

Maintenant que notre réseau est construit, nous pouvons aborder le cœur de la back-propagation.

II) Notion d'erreur et dérivée ordonnée

Il va donc maintenant s'agir de repropager des poids modifiés tels que l'erreur entre les résultats du réseau et ceux réels soit la plus petite possible. La figure suivante résume la situation :



En back-propagation basique, on choisit les poids tels qu'ils minimisent l'erreur quadratique :

$$E = \sum_{t=1}^{T} E(t) = \sum_{t=1}^{T} \sum_{i=1}^{n} \frac{1}{2} (\hat{y}(t) - y(t))^2$$

Ainsi pour obtenir les « meilleurs » poids, il va s'agir de calculer les dérivées de l'erreur E en fonction de tous les poids. Pour ce faire, plusieurs méthodes existent, plus ou moins laborieuse. Ici, nous utiliserons la dérivation ordonnée présentée dans [1], utilisée maintenant par la plupart des chercheurs.

On définit la dérivée ordonnée par :

$$\frac{\partial^{ord} E}{\partial y_i} = \frac{\partial E}{\partial y_i} + \sum_{j>i} \frac{\partial^{ord} E}{\partial y_j} \times \frac{\partial y_j}{\partial y_i}$$

Cette règle de la chaine est valide uniquement pour des systèmes ordonnés où les $y_1, y_2, \dots, y_n, E$ peuvent être calculés un par un dans l'ordre. Ainsi la dérivée ordonnée va exprimer l'impact total de tous les y_i précédant E. Par ce fait, on va donc pouvoir calculer l'impact de tous les poids sur l'erreur de notre réseau.

Par la suite, pour simplifier les notations comme nous étudierons à chaque fois la dérivée de E, on notera :

$$\frac{\partial^{ord} E}{\partial y_i} = F_{y_i}$$

III) Calcul des nouveaux poids et rétropropagation

A ce stade-là, nous avons tous les outils et les notations pour déterminer l'expression mathématiques des nouveaux poids.

En différenciant l'expression de E, on obtient facilement :

$$\frac{\partial E}{\partial \hat{y}_i(t)} = \hat{y}_i(t) - y_i(t)$$

En effet, tous les termes en $j \neq i$ s'annulent dans la somme sur j, et on évalue en un seul label t.

De plus on a :

$$\forall i \in [N+n, m+1] \;\; F_{x_i(t)} = \frac{\partial E}{\partial x_i(t)} + \sum_{j=i+1}^{N+n} F_{x_j(t)} \times \frac{\partial x_j(t)}{\partial x_i(t)}$$

$$= \frac{\partial E}{\partial \hat{y}_{i-N}(t)} + \sum_{j=i+1}^{N+n} F_{net_j(t)} \times W_{ji}$$

Et :

$$\forall i \in [N+n, m+1] \; F_{net_i(t)} = s'(net_i(t)) \times F_{x_i(t)}$$

On veillera à bien noter que i varie de $N+n$ à $m+1$ (*running backward*), ce qui correspond à la notion de back propagation : les poids sont propagés de l'erreur jusqu'au premier neurone.

Finalement on obtient :

$$F_{W_{ij}} = \sum_{t=1}^{T} F_{net_i(t)} \times x_i(t)$$

Pour simplifier le calcul de $F_{net_i(t)}$, peut remarquer que :

$$\forall z \; s'(z) = s(z) \times (1 - s(z))$$

Enfin, on obtient les nouveaux poids par :

$$New\ W_{ij} = W_{ij} - learning\_rate \times F_{w_{ij}}$$

Où $learning\_rate$ est une petite constante choisie ad hoc, souvent proche de 1.

Nous verrons en partie 3 l'implémentation algorithmique de cette méthode, sachant que l'on peut déjà ici spécifier 2 implémentations différentes :

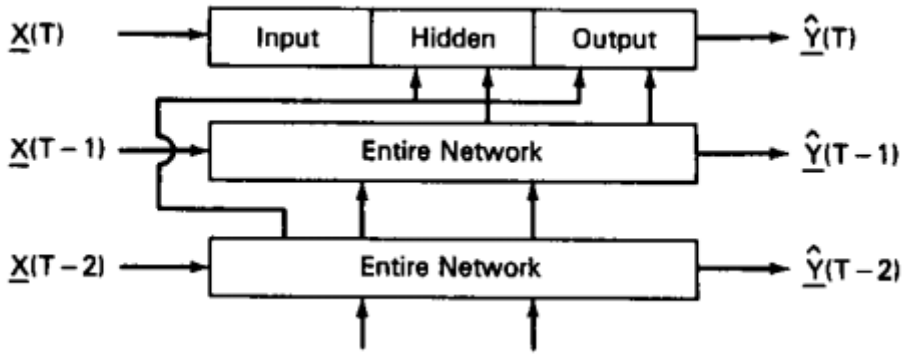
- Le *batch-learning* qui consiste à ajuster les poids après avoir observé tous les échantillons t.
- Le *pattern-learning* où les poids sont continuellement ajustés, à chaque observation.

Deuxième Partie : *Recurrent Back-propagation*

Cette partie va s'attacher à étudier la RBP originale puis à en donner des variantes réactualisées

I) *Recurrent back-propagation* originale

La RBP s'attache à rétro propager les poids minimisant l'erreur quadratique pour des Réseaux de neurones récurrents, ou RNN, que l'on peut décrire par cette figure suivante :



Ainsi il s'agit de répéter des réseaux de neurones feed-forward, comme vu en partie 1, récursivement en fonction du temps. Le label dont nous avions parlé avant, et dont nous avions remarqué sa proche dénomination avec le temps, prend tout son sens ici.

Le défi de la RBP est donc de déterminer les poids corrigeant les erreurs a posteriori.

Notons d'abord que le travail de la première partie pourrait être entièrement repris et adapté ici, sous réserve de certaines conditions concernant le RNN, en considérant la fonction $net_i(t)$ suivante :

$$net_i(t) = \sum_{j=1}^{i-1} W_{ij} x_j(t) + \sum_{j=1}^{N+n} W'_{ij} x_j(t-1) + \sum_{j=1}^{N+n} W''_{ij} x_j(t-2)$$

Le but étant donc de déterminer les nouveaux poids $W_{ij}, W'_{ij}, W''_{ij}$. On utilisera pour cela :

$$F_{x_i(t)} = \frac{\partial E}{\partial \hat{y}_{i-N}(t)} + \sum_{j=m+1}^{N+n} F_{net_j(t-2)} \times W''_{ji} + \sum_{j=m+1}^{N+n} F_{net_j(t-1)} \times W'_{ji} + \sum_{j=i+1}^{n} F_{net_j(t)} \times W_{ji}$$

Puis on obtiendra les résultats en suivant la même méthode que pour la BP basique avec :

$$F_{W_{ij}} = \sum_{t=1}^{T} F_{net_i(t)} \times x_i(t)$$

$$F_{W'_{ij}} = \sum_{t=1}^{T} F_{net_i(t-1)} \times x_i(t)$$

$$F_{W''_{ij}} = \sum_{t=1}^{T} F_{net_i(t-2)} \times x_i(t)$$

Puis les nouveaux poids viennent facilement :

$$New\ W_{ij} = W_{ij} - learning\_rate \times F_{w_{ij}}$$

$$New\ W'_{ij} = W'_{ij} - learning\_rate \times F_{w'_{ij}}$$

$$New\ W''_{ij} = W''_{ij} - learning\_rate \times F_{w''_{ij}}$$

Cependant, tout redémontrer avec ces notations n'apporte que peu d'intérêt, c'est pourquoi nous nous intéressons à une formulation plus récente donnée par la source [2].

Nous commençons avec un algorithme entraîné par un feed forward noté légèrement différemment : si on note respectivement x les données d'entrées, h_0 les couches cachées initiales, F la fonction d'actualisation des couches de paramètre w_f, on itère la formule suivante :

$$h_{t+1} = F(x, w_F, h_t)$$

Le temps est alors représenté par les itérations et non dans x, on se restreint pour l'instant à un x fixé par rapport à t (les données d'entrées restent les même durant cette phase d'entraînement).

Sous hypothèse de convergence, on obtient alors :

$$h^* = F(x, w_F, h^*)$$

On calcule alors les y sorties prédites par le réseau ainsi :

$$y = G(x, w_G, h^*)$$

Où G est la fonction sortie paramétrée par w_G.

Puisque x est fixé, on peut introduire la fonction Ψ qui ne dépend que de w_G et h telle que :

$$\Psi(w_F, h) = h - F(x, w_F, h)$$

Pour la couche entraînée h^*, nous avons donc : $\Psi(w_F, h^*) = 0$

Il nous faut alors faire l'hypothèse que la fonction F est continue et dérivable par rapport à w_F et h pour dériver Ψ :

$$\begin{aligned}\frac{\partial \Psi(w_F, h^*)}{\partial w_F} &= \frac{\partial h^*}{\partial w_F} - \frac{\mathrm{d}F(x, w_F, h^*)}{\mathrm{d}w_F} \\ &= \left(I - J_{F,h^*}\right)\frac{\partial h^*}{\partial w_F} - \frac{\partial F(x, w_F, h^*)}{\partial w_F} \\ &= 0\end{aligned}$$

Où $J_{F,h^*} = \frac{\partial F(x, w_F, h^*)}{\partial h}$ est la matrice Jacobienne de F évaluée au point h^*. Si $I - J_{F,h^*}$ est inversible, on obtient finalement :

$$\frac{\partial h^*}{\partial w_F} = \left(I - J_{F,h^*}\right)^{-1}\frac{\partial F(x, w_F, h^*)}{\partial w_F}$$

On introduit alors la fonction perte, qui nous indique à quel point les y estimés sont faux par rapports aux $\bar{y}$ réels, cette fonction est de la forme $L = l(\bar{y}, y)$, l étant une fonction de distance arbitraire. Avec le résultat précédent, nous pouvons déduire la dérivée de L par rapport à w_F grâce à la loi des dérivées de fonctions composées :

$$\frac{\partial L}{\partial w_F} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial h^*}\left(I - J_{F,h^*}\right)^{-1}\frac{\partial F(x, w_F, h^*)}{\partial w_F}$$

On peut de surcroît aussi dériver par rapport à w_G :

$$\frac{\partial L}{\partial w_G} = \frac{\partial L}{\partial y}\frac{\partial G(x, w_G, h^*)}{\partial w_G}$$

Cette dernière se calcule relativement facilement, cependant la première demande encore un peu de travail. Pour cela on pose le vecteur colonne :

$$z = \left(I - J_{F,h^*}^{\top}\right)^{-1}\left(\frac{\partial L}{\partial y}\frac{\partial y}{\partial h^*}\right)^{\top}$$

L'objectif est maintenant de calculer z afin d'atteindre $\frac{\partial L}{\partial w_F}$. La force brute n'est pas conseillée car $J_{F,h^*}^{\top}$ n'est pas forcément symétrique. On multiplie alors les termes par $\left(I - J_{F,h^*}^{\top}\right)$ et réarranger les termes donne :

$$z = J_{F,h^*}^{\top} z + \left(\frac{\partial L}{\partial y}\frac{\partial y}{\partial h^*}\right)^{\top}$$

On se retrouve alors avec un problème de point fixe de la fonction $f(z) = J_{F,h^*}^{\top} z + \left(\frac{\partial L}{\partial y}\frac{\partial y}{\partial h^*}\right)^{\top}$

Après calcul, nous nous retrouvons ainsi bien le gradient de L avec une grande précision, le tout avec et coût constant.

II) Variantes

En pratique, les hypothèses qui ont été faites (F continue et dérivable et $I - J_{F,h^*}$ inversible) peuvent être vérifiées pour beaucoup de types de RNN. Cependant, l'algorithme du point fixe de f(z), même s'il est garanti de converger, peut prendre beaucoup d'itérations pour obtenir une précision correcte.

Les variations suivantes proposent de calculer z avec exactitude et de manière plus optimisée.

La variation de Neumann se présente ainsi, on reprend la formule de z :

$$z = \left(I - J_{F,h^*}^{\top}\right)^{-1} \left(\frac{\partial L}{\partial y}\frac{\partial y}{\partial h^*}\right)^{\top}$$

En réarrangeant les termes on peut le formuler ainsi :

$$\left(I - J_{F,h^*}\right)\left(I - J_{F,h^*}^{\top}\right) z = \left(I - J_{F,h^*}\right) \left(\frac{\partial L}{\partial y}\frac{\partial y}{\partial h^*}\right)^{\top}$$

Le but est d'alors faire apparaître le terme$\left(I - J_{F,h^*}\right)\left(I - J_{F,h^*}^{\top}\right)$ qui est par construction symétrique. On peut alors résoudre ce système grâce à la méthode du gradient conjugué.

La nouvelle méthode appelée Neumann-RBP se base sur les séries de Neumann qui dit que sous certaines conditions sur une matrice A, on a :

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k$$

Le but étant d'appliquer ce résultat à $J_{F,h^*}^{\top}$et ainsi obtenir z. Une condition suffisante de convergence porte sur le rayon spectral (définit comme $\rho(A) = \max_i |\lambda_i|$ où λ_i sont les vecteurs propres de A), en effet les séries de Neumann convergent si $\rho(A) < 1$. Cette condition appliquée à $J_{F,h^*}^{\top}$ nous donne directement z tel que définit au tout début :

$$z = \left(I - J_{F,h^*}^{\top}\right)^{-1} \left(\frac{\partial L}{\partial y}\frac{\partial y}{\partial h^*}\right)^{\top}$$

Et une bonne approximation de $\left(I - J_{F,h^*}^{\top}\right)^{-1}$ s'obtient en calculant la somme partielle d'ordre K avec K suffisamment grand, on a alors :

$$\frac{\partial L}{\partial w_F} \approx \frac{\partial L}{\partial y} \sum_{k=0}^{K} \frac{\partial y}{\partial h^*} J_{F,h^*}^{k} \frac{\partial F(x, w_F, h^*)}{\partial w_F}$$

Troisième Partie : Codes

Cette partie s'attachera à donner les codes servant à l'implémentation des méthodes décrite dans les parties 1 et 2. Ils seront donnés en langage Python.

1) Basic BP

a) Implémentation du réseau (calcul des $\hat{y}_i$ et des x_i^{act})

```
import math as m
```

```
def reseau(X,W,Yhat,N): #X est un vecteur de taille m, W une matrice de taille N-n et Yhat un vecteur de taille n
    m=len(X)
    n=len(Yhat)
    net=[]
    xact=[]
    for i in range (m+1,N+n):
        for j in range(1,i-1):
            net.append(net[i]+W[i][j]*X[j])
        xact[i]=1/(1+m.exp(-net[i])
    for i in range (1,n):
        Yhat[i]=xact[i+N]
return xact,Yhat
```

b) Dérivée ordonnée

```
from sympy import *
def derivee_ordonnée_i(E,z,i): #on suppose z vecteur de taille n, et on suppose disposer d'une expression liant E aux z[i]
    terme_1=E.diff(z[i])
    terme_2=0
    n=len(z)
    for j in range (i+1,n):
        terme_2+=derivee_ordonnée_i(E,z,j)*z[j].diff(z[i])
    F_z[i]=terme_1 + terme_2
    return F_z[i]
def derivee_ordonnée(E,z):
    derivee_ordonnée=[]
    for i in range(n):
        derivee_ordonnée.append(derivee_ordonnée_i(E,z,i))
    return derivee_ordonnée
```

c) Batch-learning

```
import numpy as np
def batch_learning(W,N,learning_rate):
    x,Yhat,net=reseau(X,W,Yhat,N)
    F_Yhat,F_net=derivee_ordonnée(E,Yhat),derivee_ordonnée(E,W),dérivée_ordonnée(E,net)
    F_x=[]
    F_W=np.empty(N-n,N-n)
    n=len(Yhat)
    New_W=np.empty(N-n,N-n)
    for i in range(N):
        F_x[i]=0
    for i in range(n):
        F_x[i+N]=F_Yhat[i]
    for i in range(N+n,m+1,-1): #N>m
        for j in range(i+1,N+n):
            F_x[i]+=W[j][i]*F_net[j]
        F_net[i]=F_x[i]*x[i]*(1-x[i])
        for j in range (1,i-1):
            F_W[i,j]=F_net[i]*x[j]
    for i in range(N-n):
        for j in range(N-n):
            New_W[i][j]=W[i][j]-learning_rate*F_W[i][j]
    return New_W
```

d) Pattern-learning

```
def pattern_learning(T,K):#k une constante que l'on prend très grande, T nos echantillons dont on tire les labels t
    for k in range(K):
        for t in range(T):
            x,Yhat,net=reseau(X,W,Yhat,N)
            n=len(Yhat)
            F_Yhat=np.linespace(n)
            for i in range(n):
                F_Yhat[i]=Yhat[i]-Y(t,i)  #avec Y(t,i) les realisation réelles observées pour l'echantillon t
            F_x,F_net,F_W,New_W=batch_learning(W,N,learning_rate)
            for i in range (m+1,N+n):
                New_W[i][j]=W[i][j]-learning_rate*F_W[i][j]
    return New_W
```

2) RBP

On ne redonne pas les codes ici, en effet il suffit juste de calculer les nouveaux poids $New\ W_{ij} = W_{ij} - learning\_rate \times F_{w_{ij}}$, $New\ W'_{ij} = W'_{ij} - learning\_rate \times F_{w'_{ij}}$, $New\ W''_{ij} = W''_{ij} - learning\_rate \times F_{w''_{ij}}$ de la même façon.