MANUEL D'UTILISATION DE YANNS
Yet Another Neural Network Simulator

Version non définitive

Romuald Boné
Michel Crucianu
{bone, crucianu}@univ-tours.fr
Laboratoire d'Informatique
Equipe Reconnaissance des Formes et Analyse d'Images
Ecole d'Ingénieurs en Informatique pour l'Industrie
Université François Rabelais
64 Av. Jean Portalis
37200 TOURS - FRANCE

Table des matières

Introduction

YANNS (*Yet Another Neural Network Simulator*) est un simulateur de réseaux de neurones qui permet d'utiliser les réseaux de neurones à propagation avant ou récurrents, avec connexions simples ou à délais. Dans sa version en mode texte, qui est décrite dans ce manuel, YANNS est constitué d'un noyau et d'un langage de description. Le noyau comporte plus de 10000 lignes de code C++ compilées. Pour utiliser ce noyau et rendre le simulateur accessible aux utilisateurs avec peu d'expérience de la programmation ou de la manipulation des réseaux de neurones, nous avons défini un langage de spécification simple et intuitif. Le langage fournit toutes les fonctions de spécification du réseau et toutes les fonctions de contrôle nécessaires. De nombreux contrôles de cohérence sont réalisés avant le début de l'apprentissage et des messages d'erreurs détaillés sont générés le cas échéant. Des réseaux de neurones complexes peuvent être créés rapidement et facilement, ce qui rend l'outil abordable pour des utilisateurs inexpérimentés qui veulent découvrir les modèles de réseaux de neurones avec l'aide du simulateur.

Dans la version en mode texte, YANNS se présente sous la forme d'un fichier exécutable (disponible sous Windows 95, Windows NT et Unix). L'utilisateur édite un fichier texte où, avec le langage de YANNS, il décrit l'architecture du réseau, indique l'emplacement des fichiers de données et choisit l'algorithme d'apprentissage et ses paramètres. Pour lancer la simulation, il suffit ensuite d'appeler l'exécutable dans un terminal en lui fournissant en paramètre le nom du fichier texte précédemment édité :

```
YANNS fichier.txt
```

La simulation commence et les résultats apparaissent à l'écran. Pour enregistrer ces résultats dans un fichier, il suffit de rediriger la sortie standard :

```
YANNS fichier.txt >resultat.txt
```

Le résultat de la simulation se trouve alors dans le fichier *resultat.txt*.

Ce manuel a pour but de présenter le langage de description de YANNS.

Conventions

Voici les conventions que nous avons adoptées dans ce manuel.

Les entrées et les sorties machine apparaissent dans `une police à espacement non proportionnel` et les valeurs remplaçables en *italiques* :

```
YANNS fichier.txt
```

Un mot clé est signalé en **gras** alors qu'une entité optionnelle apparaît entre `<caractère inférieur et caractère supérieur>`.

Modélisation neuronale

Un neurone artificiel (ou neurone) reçoit des signaux (entrées) en provenance de l'environnement et des autres neurones auxquels il est relié. A chacune des entrées est associé un poids représentatif de la « force » de la connexion interneuronale. Chaque neurone (Figure 1) possède sa propre fonction de transfert (ou fonction d'activation) qui lui permet de calculer sa sortie (ou son état) à partir du produit scalaire des entrées et des poids des connexions associées. La sortie peut être dupliquée en autant d'exemplaires que désiré, les duplicata transportant le même signal.

Un réseau de neurones artificiels (Figure 2) est constitué de neurones interconnectés par des arcs unidirectionnels appelés « connexions ». Chacun de ces neurones est d'un type spécifique :

- Les neurones d'entrée (ensemble de neurones généralement désigné par « rétine » du réseau) sont les neurones qui disposent de connexions entrantes reliées à l'extérieur du réseau. Ces neurones permettent à l'environnement d'agir sur le fonctionnement du réseau.
- Les neurones de sortie possèdent une connexion sortante vers l'extérieur du réseau. Ces neurones indiquent à l'environnement la réponse du réseau.
- Les neurones cachés ont des connexions entrantes qui ne proviennent que des neurones du réseau et possèdent une sortie non reliée vers l'exterieur.

Les liaisons entre les neurones sont définies par le graphe d'interconnexion du réseau, qui stipule l'orientation de ces liaisons. Les sommets du graphe sont les neurones et les arcs, les connexions internes au réseau; les connexions externes ne figurent pas dans ce graphe. Selon la forme de ce graphe, on distingue deux grandes classes de réseaux: les réseaux à couches et les réseaux dynamiques ou récurrents.

- Dans les réseaux à couches le graphe d'interconnexion peut être réparti en niveaux, ce qui signifie qu'il n'existe aucun « circuit », c'est à dire aucun chemin reliant un sommet à lui-même. La sortie d'un neurone ne dépend alors que des neurones appartenant à un niveau situé en amont. L'évaluation de la sortie du réseau ne peut se faire que couche par couche, par niveaux croissants, les neurones d'une même couche pouvant évoluer en parallèle.
- Pour les réseaux récurrents (ou dynamiques) le graphe d'interconnexion peut contenir des circuits. A un instant donné, la sortie d'un neurone peut donc dépendre des valeurs de sortie des neurones à un instant précédent.

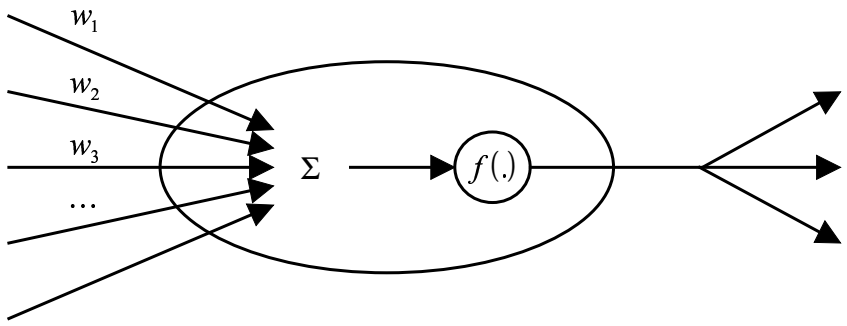

Figure 1 : Un neurone artificiel

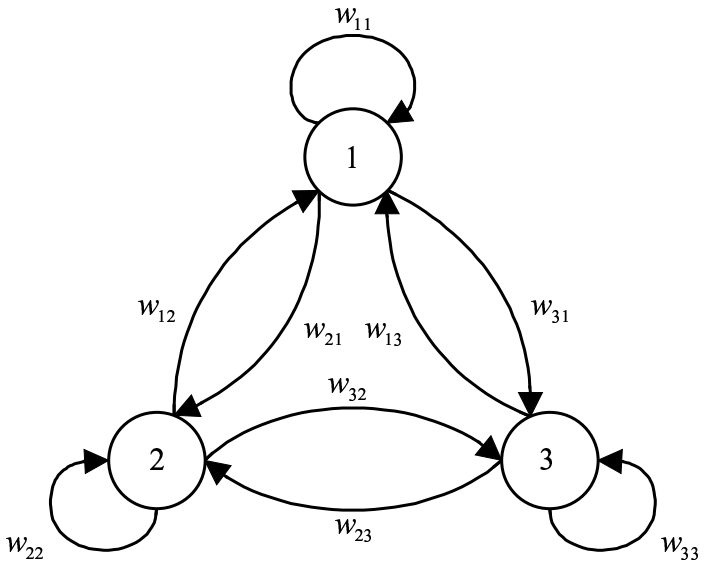

Figure 2 : Un réseau récurrent à 3 neurones

L'apprentissage est la phase pendant laquelle le réseau s'adapte (le plus souvent, les poids des connexions) afin de fournir sur ses neurones de sortie les valeurs désirées. Cet apprentissage nécessite un algorithme et des exemples désignés sous l'appellation d'ensemble (ou d'échantillon) d'apprentissage. Après initialisation des poids du réseau (en général par des valeurs aléatoires), il y a présentation des exemples au réseau et calcul des sorties correspondantes. Une valeur d'erreur est calculée et une correction des poids est appliquée.
L'apprentissage est dit supervisé lorsque les exemples sont constitués de couples de valeur du type : `<valeur d'entrée, valeur de sortie désirée>`.

Un réseau de neurones n'est réellement utile que si, après la phase d'apprentissage, il est capable de « généraliser », c'est-à-dire de produire des valeurs de sortie correctes lorsqu'on injecte sur sa rétine un vecteur d'entrée n'appartenant pas à l'ensemble d'apprentissage. Bien que l'on puisse utiliser un grand nombre de critères d'arrêt de l'apprentissage, le plus courant consiste à disposer d'un ensemble d'exemples, différent de l'ensemble d'apprentissage, sur lequel on mesure un critère d'erreur. Dans la plupart des cas, ce critère diminue en moyenne au début de l'apprentissage puis augmente par la suite. Cette augmentation est souvent décrite comme l'apprentissage par cœur où le réseau apprend quasi parfaitement sur l'ensemble d'apprentissage mais ne sait pas généraliser. En général, on choisit d'arrêter l'apprentissage quand l'erreur sur cet ensemble, appelé ensemble d'arrêt, commence à remonter.

Pour mesurer les capacités de généralisation d'un réseau, il est nécessaire d'utiliser un ensemble indépendant par rapport à la phase d'apprentissage. C'est l'ensemble de test (ou encore ensemble de validation), qui est également composé d'exemples d'entrées et de sorties associées.

En utilisation réelle, après apprentissage et validation du réseau, l'utilisateur ne dispose généralement que d'entrées et désire que le réseau lui fournisse des sorties estimées associées. Cet ensemble d'exemple qui ne possèdent que des entrées sera appelé ensemble d'utilisation.

Principes du langage de YANNS

Le langage de description du simulateur YANNS est basé sur le concept de blocs. Un bloc est défini par un mot clé, une accolade ouvrante, plusieurs déclarations et une accolade fermante. Les blocs sont de quatre types :

- Les blocs de définitions de neurones
- Les blocs de connections
- Le bloc script
- Le bloc apprentissage

Les blocs de définitions de neurones permettent de décrire les neurones qui seront utilisés dans le réseau. Un neurone est caractérisé par un numéro et une fonction de transfert. Les différentes fonctions disponibles sont les suivantes : identité, linéaire, sigmoïde et tangente hyperbolique (encore désignée sigmoïde symétrique).

Les blocs de connexions permettent de déclarer les différentes connexions qui relient les neurones entre eux, d'indiquer leur type et de fixer des valeurs précises aux poids. Les connexions simples possèdent un seul poids et correspondent à la présentation générale précédente. Les connexions FIR (Figure 3) sont des connexions avec retards (symboles Δ) qui peuvent posséder plusieurs poids. Par souci de généralité par rapport au modèle d'origine [Wan 1990; Back and Tsoi 1991], les retards ne sont pas nécessairement consécutifs.

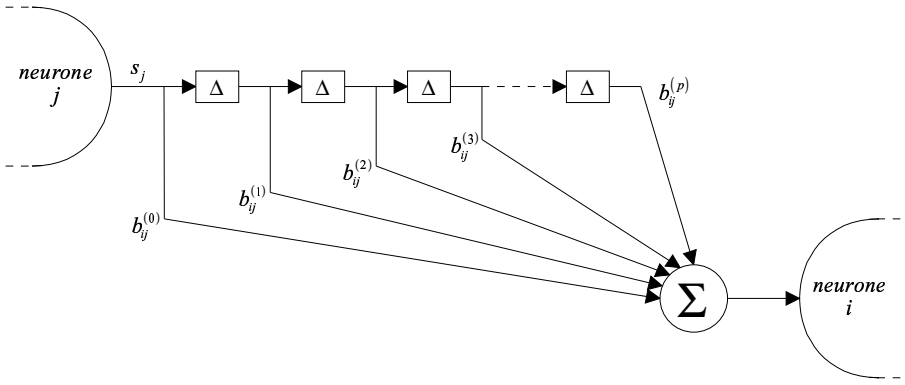

Figure 3 : une connexion FIR

Le bloc script permet de donner le chemin d'accès et les noms des fichiers contenant les ensembles (ou encore corpus) utilisés pour la simulation. On peut retrouver ici l'ensemble d'apprentissage, l'ensemble d'arrêt, l'ensemble de test et l'ensemble d'utilisation. Notons que le format de ces fichiers est des plus simples : les données sont stockées dans des fichiers textes, séparées par des espaces, des tabulations ou des sauts de ligne. Pour chaque simulation, il n'est pas nécessaire de disposer des 4 ensembles. Sur un réseau qui a déjà appris et qui a été sauvegardé, par exemple, on pourra utiliser seulement l'ensemble de test ou celui d'utilisation. Il ne peut y avoir qu'un seul bloc script dans un fichier de description.

Le bloc d'apprentissage permet de choisir l'algorithme d'apprentissage et ses paramètres, le ou les critères d'arrêt, le type d'initialisation des poids et le nom du fichier de sauvegarde du réseau appris.

Grammaire

Nous présentons ici la grammaire que doit respecter un fichier de description destiné à YANNS. Le caractère | indique une possibilité parmi plusieurs après le signe égal, les expressions entre caractères < et > sont optionnelles.

Un réel contient obligatoirement un point et éventuellement un exposant. Les nombres suivants sont reconnus comme des réels par YANNS : `7.2` `-0.2` `1.2E5` `1.2E-5`
Les nombres suivants ne sont pas reconnus : `1.` `.2`

Un commentaire débute avec deux barres obliques consécutives `//` et se termine à la fin de la ligne.

Pour les mots clés, YANNS ne différencie pas les minuscules des majuscules et partout où un mot apparaît au pluriel dans la grammaire, il est également accepté au singulier (les mots clés **`Neurons`** ou **`Neuron`** sont licites).

Une énumération ou une sélection permet de décrire un ensemble d'entiers, non nécessairement consécutifs, d'une manière rapide et compacte.

`1,3,4`	Entiers 1, 3 et 4
`1..8`	Entiers 1 à 8
`1..3,5..10,12`	Entiers 1 à 3, 5 à 10 et 12
`ALL`	Tous les entiers (dépend du contexte)

Un numéro de couche commence par un L majuscule accolé à un entier : `L3`
Une liste de couches énumère chaque couche, en les séparant par des virgules : `L2, L3.`

```
NoNeurons=Entier;

Neurons[Enumération] {
  TransfertFN= Identity;
             | Linear<(Réel)>;
             | Sigmoid<(Réel)>;
             | TanH;
             | SymSigmoid<(Réel)>;
< NoLayer=NuméroDeCouche ; >
}

// plusieurs blocs Neurons possibles

Inputs= [Enumération];
      | NuméroDeCouche;

Outputs= [Enumération];
       | NuméroDeCouche;

< Bias {
    Target= [Enumération];
```

```
          | NuméroDeCouche ;
          | ListeDeCouches ;
  < Weights= [ListeDePoids]; >
  }
>

Connections {
  Source= [Enumération];
        | NuméroDeCouche ;
        | ListeDeCouches ;

  Target= [Enumération];
        | NuméroDeCouche ;
        | ListeDeCouches ;

  Type = Simple;
       | FIR(Enumération);
< Weights= [ListeDePoids]; >
< FIRWeights= [ListeDePoids]; >
}

// plusieurs blocs Connections possibles

Script {
< Path = "CheminDesFichiersEnsembles";>
< Learning = "NomFichierApprentissage" <, [sélection]> ; >
< Stop = "NomFichierStop" <, [sélection]> <, BestNetwork> ; >
< Test = "NomFichierTest" <, [sélection]> ; >
< Use = "NomFichierUtil" <, [sélection]> <, StateDisplay> ; >
}

Learning {
  Algorithm = BackPropagation ( modeMAJ,  pas );
              | BackPropagation ( modeMAJ, pas, alpha, beta );
              | BPTT ( pas );
              | BPTT ( pas , alpha , beta );
              | BPTTH ( pas , H );
              | BPTTE ( pas, alpha, beta, Entier, Entier );
              | RL ( pas );
              | RL ( pas , alpha , beta );
              | RTRL ( pas );
              | RTRL ( pas , alpha , beta );
              | BPFIR1 ( modeMAJ, pas );
              | BPFIR1 ( modeMAJ, pas , alpha , beta );
              | BPFIR2 ( pas );
              | BPFIR2 ( pas , alpha , beta );
              | ALOPEX( pas, fenêtre );
< Momentum = Réel ; >
< WEIGHTDECAY = Réel ; >
< NoMaxStep = Réel ; >
< MinLearningError = Réel ; >
```

```
< MinStopError = Increase | Réel ; >
< ErrorDisplayPeriod = Réel ; >
< MAOStopError = Réel ; >
< MinStopError = Increase | Réel ; >
< Weightsinit =  Normal ;
                | Normal(Réel, Réel);
                | Uniform ;
                | Uniform(Réel, Réel); >
< AllWeightsInit = Normal ;
                | Normal(Réel, Réel);
                | Uniform ;
                | Uniform(Réel, Réel); >

< Save = "NomFichierSauvegarde" ; >
}
```

Valeurs par défaut

Pour une simplification de l'écriture des fichiers de description, de nombreux paramètres possèdent des valeurs par défaut et rendent leur déclaration optionnelle. Par ordre d'apparition, citons :

Bloc **`Neurons`**
Les valeurs par défaut des paramètres des fonctions de transfert sont toutes à 1.0

Bloc **`Script`**
Pour les différents ensembles, si un intervalle n'est pas précisé alors le contenu intégral de chaque ensemble est pris en compte.

Bloc **`Learning`**
Les algorithmes qui disposent de l'option de mode de mise à jour des poids ont par défaut la valeur de l'option **`ONLINE`**.
Pour les lois d'initialisation, les valeurs par défaut sont pour la loi normale de 0.0 (moyenne) et de 1.0 (écart type) et pour la loi uniforme de –0.3 et +0.3 (bornes minimum et maximum). La loi par défaut est la loi uniforme.
Le nombre d'itération maximum pour l'apprentissage est fixé par défaut à 1000 itérations.
La période d'affichage de l'erreur est fixée à 100.

Gestion des erreurs

Le simulateur gère un certain nombre d'erreurs liées au fichier de description du réseau. Outre les erreurs lexicales ou syntaxiques, pour lesquelles un numéro de ligne est indiqué, YANNS effectue également des contrôles de cohérence et détecte :

- Un nombre de neurones différent entre la déclaration en début de fichier et les neurones déclarés dans les blocs **`Neurons`**
- Des neurones déclarés et non reliés à d'autres neurones
- Des neurones définis plusieurs fois
- Des couches déclarées pour un réseau récurrent
- Des couches vides pour un réseau à propagation avant

- Un mauvais chemin dans le bloc script
- Des fichiers pour l'apprentissage, le stop, le test ou l'utilisation qui ne comportent pas le bon nombre d'entrées
- Un problème de saturation mémoire

Explication du langage

Nombre de neurones

La déclaration du nombre de neurones (hormis le neurone de biais éventuel) du réseau à expérimenter s'effectue au début du fichier de description avec :

```
NoNeurons = Entier ;
```

Description des neurones

Les blocs **Neurons** permettent de définir les fonctions de transfert associées aux différents neurones du réseau. Si plusieurs neurones ont des caractéristiques communes, il est possible de regrouper leurs définitions dans le même bloc.

```
Neurons[Enumération] {
  TransfertFN= Identity;
               | Linear ;
               | Linear<(Réel)>;
               | Sigmoid ;
               | Sigmoid<(Réel)>;
               | TanH;
               | SymSigmoid ;
               | SymSigmoid<(Réel)>;
< NoLayer=NuméroDeCouche ; >
}
```

Exemple :

```
Neurons[1..12, 21] {
  TransfertFn=Linear;
}
```

Les définition mathématiques des fonctions sont les suivantes :

- **Identity** : Fonction identité $f(x) = x$
- **Linear(k)** : Fonction linéaire $f(x) = kx$ (par défaut k=1)
- **TanH** : Fonction tangente hyperbolique $f(x) = \tanh(x)$
- **Sigmoid(k)** : Fonction sigmoïde $f(x) = \frac{1}{1+e^{-kx}}$ (par défaut k=1, Figure 4)
- **SymSigmoid(k)** : Fonction sigmoïde symétrique $f(x) = \frac{e^{kx}-1}{e^{kx}+1}$ (par défaut k=1). Pour k=1 (Figure 5) cette fonction est équivalente à la fonction tangente hyperbolique.

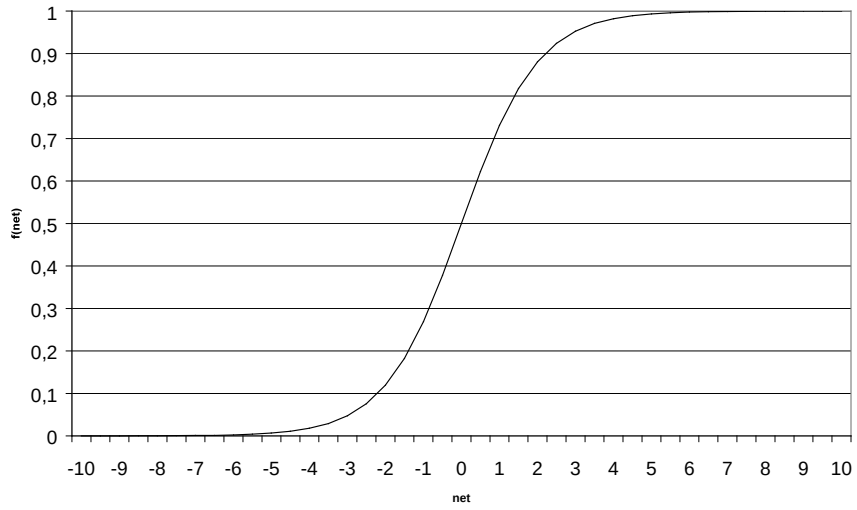

Figure 4 : Fonction sigmoïde

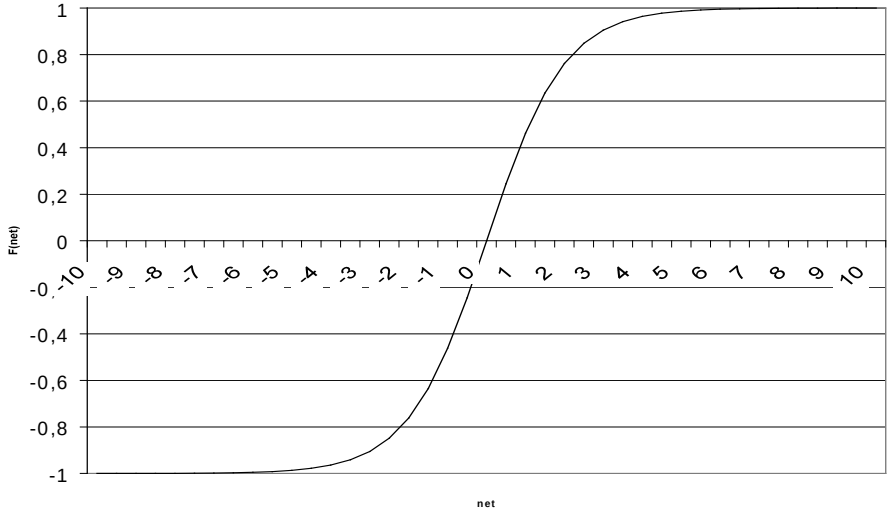

Figure 5 : Fonction sigmoïde symétrique

Ce bloc permet également de déclarer l'appartenance d'une neurone à une couche. Dans ce cas le réseau décrit est considéré comme un réseaux à propagation avant (*feed forward*) c'est à dire non récurrent par le simulateur. La propagation dans le réseau n'est plus synchrone mais de couche en couche de la première à la dernière.

Il est possible de déclarer plusieurs blocs **`Neurons`** dans un même fichier de description.

Entrées / Sorties

Cette partie permet de déclarer les neurones d'entrée et les neurones de sortie.

```
Inputs=[Enumération];
      | NuméroDeCouche;

Outputs=[Enumération];
      | NuméroDeCouche;
```

Exemple :

```
Inputs=[1..12];
Output=[21];
```

Pour les réseaux récurrents il faut respecter un décalage entre les entrées et les sorties associées. Ce décalage correspond au nombre de connexions qui se trouvent sur le plus court chemin entre les neurones d'entrée et ceux de sortie dans le graphe d'interconnexion.

Par exemple, sur un problème de prévision mensuelle à l'aide d'un réseau récurrent dont le plus court chemin entre les neurones d'entrée et ceux de sortie est de longueur 2, le décalage entre une entrée et sa sortie associée est de 2 mais attention, la sortie associée étant la prévision du mois suivant, en final le décalage est de 1 — voir plus loin.

Fichier d'apprentissage (la première colonne pour les entrées, la seconde pour les sorties) :

```
Valeur mois d'avril          0.0
Valeur mois de mai           0.0
Valeur mois de juin          Valeur mois de mai (associée à
                                  l'entrée du mois d'avril)
Valeur mois de juillet       Valeur mois de juin
Valeur mois d'août           Valeur mois de juillet
...                          ...
```

Neurone de biais

Le bloc **Bias** permet la déclaration du neurone de biais. Il n'a pas d'entrée net (aucun neurone connecté en entrée) et sa sortie est constamment à 1. Les neurones qui lui sont connectés en sortie sont désignés avec le mot clé **Target**. Il est possible de fixer des valeurs précises à chacun des poids des connexions sortantes du neurones de biais en utilisant une déclaration optionnelle avec le mot clé **Weights**

```
Bias {
 Target=[Enumération] ;
        | NuméroDeCouche ;
        | ListeDeCouches ;
< Weights=[ListeDePoids] ; >
}
```

Exemple :

```
Bias {
  Target=[1,3,4] ;
  Weights=[0.5, 0.0, 0.1] ;
}
```

Le neurone de biais est ici connecté aux neurones 1, 3 et 4. Le poids de la connexion avec le neurone 1 est fixé à 0.5 et le poids de la connexion avec le neurone 4 est fixé à 0.1. Le poids

de la connexion avec la neurone 3 est destiné à être initialisé aléatoirement (valeur réservée 0.0).

Les connections avec la neurone biais sont des connexions simples.

Connexions

Les blocs **Connections** permettent la déclaration de l'ensemble des connexions présentes entre les différents neurones du réseau, ainsi que la déclaration de leur type, et éventuellement la valeurs des poids. Une connexion est un arc orienté qui part d'un neurone source et qui atteint un neurone cible. Le neurone source est désigné avec le mot clé **Source**, et le neurone cible est désigné par le mot clé **Target**.

```
Connections {
 Source=[Enumération];
        | NuméroDeCouche ;
        | ListeDeCouches ;
  Target=[Enumération];
        | NuméroDeCouche ;
        | ListeDeCouches ;
  < Type = Simple ;
        | FIR(Enumération) ; >
  < Weights = [ListedesPoids] ; >
  < FIRWeights = [ListedesPoids] ; >
}
```

Les connexions sont de type simple ou de type FIR (avec retards). Si le type de connexion n'est pas précisé, alors les connexions sont de type simple. Dans le cas des connexions FIR, il faut préciser les retards désirés. Selon le type de connexion, on pourra fixer la valeur de certains poids avec le mot clé **Weights** ou **FIRWeights**. Il est possible de déclarer plusieurs blocs **Connections** dans un même fichier de description.

Exemple 1 :

```
Connections {
  Source=[1, 2] ;
  Target=[5, 7] ;
  Type = FIR(0..3,5) ;
}
```

Les neurones 1 et 2 sont reliés aux neurones 5 et 7. Il y a donc 4 connexions de type FIR. Pour chacune d'elles, il y a 5 poids, le premier sans retard (retard 0) et les autres avec des retards respectivement d'1, 2, 3 et 5 unités de temps.

La liste des retards doit être ordonnée par ordre croissant. Pour spécifier des valeurs pour les poids, pour chaque neurone source on affecte les poids pour tous les neurones cibles et pour tous les retards.

Exemple 2 :

```
Connections {
  Source=[1, 2] ;
  Target=[5, 7] ;
  Type = Simple ;
  Weights = 0.1, 0.0, 0.2, 0.3;
}
```

Ici le bloc décrit quatre connexions simples, avec les valeurs initiales suivantes :
Connexion du neurone 1 au neurone 5 avec pour pondération initiale 0.1
Connexion du neurone 1 au neurone 7 avec tirage aléatoire de la pondération
Connexion du neurone 2 au neurone 5 avec pour pondération initiale 0.2
Connexion du neurone 2 au neurone 7 avec pour pondération initiale 0.3

Script

Le bloc **Script** permet de déclarer les noms des fichiers contenant les ensembles, ainsi que le chemin du répertoire où se trouve ces fichiers.
Le mot clé **Path** est utilisé pour indiquer le chemin des fichiers.
Le mot clé **Learning** est utilisé pour le corpus d'apprentissage.
Le mot clé **Stop** est utilisé pour le corpus de stop.
Le mot clé **Test** est utilisé pour le corpus de test.
Le mot clé **Use** est utilisé pour le corpus d'utilisation.

```
Script {
< Path = "CheminDesFichiersEnsembles";>
< Learning = "NomFichierApprentissage" <, [sélection]> ; >
< Stop = "NomFichierStop" <, [sélection]> <, BestNetwork> ; >
< Test = "NomFichierTest" <, [sélection]> > ;
< Use = "NomFichierUtil" <, [sélection]> <, StateDisplay> ; >
}
```

La sélection permet de choisir une partie des exemples contenus dans l'ensemble concerné. Cet intervalle est optionnel, en son absence, l'ensemble est intégralement pris en compte. La syntaxe de l'intervalle est la même que celle de l'énumération. Le mot clé **END** désigne la fin de l'ensemble.

Exemple :

`1,3,5`	Les exemples 1, et 5
`1..5`	Les exemples 1 à 5
`1..3, 8..10`	Les exemples 1 à 3 et 8 à 10
`5..END`	Tous les exemples à partir du cinquième

Le mot clé **BestNetwork** optionnel permet de conserver le réseau fournissant l'erreur d'apprentissage la plus basse sur l'ensemble de stop.

Le mot clé **StateDisplay** optionnel permet de connaître la sortie de tous les neurones du réseau en utilisation, pour l'ensemble des données du fichier USE.

Apprentissage

Le bloc **Learning** permet de définir les différents paramètres pour l'apprentissage du réseau. C'est dans ce bloc que l'on définit l'algorithme utilisé, le pas d'apprentissage, la ou les condition d'arrêt, l'initialisation des poids des connexions, et le chemin et le nom du fichier de sauvegarde du réseau.

```
Learning {
  Algorithm = BackPropagation ( modeMAJ,  pas );
            | BackPropagation ( modeMAJ,  pas , alpha , beta );
            | BPTT ( pas ) ;
            | BPTT ( pas , alpha , beta ) ;
            | BPTTH ( pas , H ) ;
            | BPTTH ( pas , alpha , beta , H ) ;
            | RL ( pas ) ;
            | RL ( pas , alpha , beta ) ;
            | RTRL ( pas ) ;
            | RTRL ( pas , alpha , beta ) ;
            | BPFIR1 ( modeMAJ,  pas );
            | BPFIR1 ( modeMAJ,  pas , alpha , beta );
            | BPFIR2 ( pas );
            | BPFIR2 ( pas , alpha , beta );
            | ALOPEX( pas, fenêtre ) ;
< Momentum = Réel ; >
< WeightDecay = Réel ;>
< NoMaxStep = Réel ; >
< MinLearningError = Réel ; >
< MinStopError = Increase | Réel ; >
< ErrorDisplayPeriod = Réel ; >
< MAOStopError = Réel ; >
< MinStopError = Increase | Réel ; >
< Weightsinit = Normal ;
              | Normal(Réel, Réel);
              | Uniform ;
              | Uniform(Réel, Réel); >
< AllWeightsInit = Normal ;
                 | Normal(Réel, Réel);
                 | Uniform ;
                 | Uniform(Réel, Réel); >

< Save = "NomFichierSauvegarde" ; >
}
```

Les différents algorithmes

- **BackPropagation** : Rétro-propagation du gradient [Rumelhart, Hinton et al. 1986], algorithme utilisé pour l'apprentissage des réseaux non récurrents (réseaux à couches). Ses paramètres sont :

  modeMAJ : Détermine la procédure de modification des poids. Il y a deux valeurs possibles, qui sont **ONLINE** ou **OFFLINE**. **ONLINE**, qui est la valeur par défaut, commande une modification des poids après chaque présentation d'un vecteur de

l'ensemble d'apprentissage. **OFFLINE** commande une modification des poids des connexions lorsque l'on a présenté tous les vecteurs de l'ensemble d'apprentissage.

pas : détermine l'importance des modifications qui seront apportées aux poids des connexions, lors de l'apprentissage.

Alpha et Beta : Paramètres optionnels permettant d'avoir un pas adaptatif.

- **RL** : *Recurrent Learning*, algorithme utilisé pour l'apprentissage des réseaux récurrents [Williams and Zipser 1989]. La modification est de type *offline*. Ses paramètres sont le pas ou le pas , alpha et beta.(voir algorithme **BackPropagation**).

- **RTRL** : *Real Time Recurrent Learning*, algorithme utilisé pour l'apprentissage des réseaux récurrents [Williams and Zipser 1989]. La modification est de type *online*. Ses paramètres sont le pas ou le pas , alpha et beta.(voir algorithme **BackPropagation**)

- **BPTT** : *Back Propagation Through Time* (Rétro-propagation du gradient dans le temps), algorithme utilisé pour l'apprentissage des réseaux récurrents [Rumelhart, Hinton et al. 1986]. Ses paramètres sont les mêmes que ceux de **BackPropagation.**

- **BPTTH** : Version de BPTT permettant de définir un dépliement sur H pas de temps. Ses paramètres sont ceux de **BackPropagation** auxquels, on ajoute H (nombre de pas de temps).

- **ALOPEX** : algorithme stochastique [Unnikrishnan and Venugopal 1992] utilisé pour l'apprentissage des réseaux non récurrents et des réseaux récurrents. Ses paramètres sont :

  pas : détermine l'importance des modifications qui seront apportées aux poids des connexions, lors de l'apprentissage.

  fenêtre : intervalle pour le calcul de la corrélation utilisée dans l'algorithme.

Paramètres globaux pour l'apprentissage

- **Momentum** : Permet de définir l'inertie (uniquement valable pour les algorithmes utilisant le gradient). L'inertie permet de lisser la variation des poids. On conserve l'ancienne modification des poids et on multiplie cette évolution par l'inertie avant de l'ajouter à la nouvelle modification des poids.
- **WeightDecay** : Cette option permet de mettre une contrainte pour faire tendre les poids vers zéro au cours de l'apprentissage.

Les critères d'arrêt

- **ErrorDisplayPeriod** : Cette option permet de définir la période d'affichage de l'erreur sur les ensembles d'apprentissage et de stop (l'erreur sur l'ensemble de stop est calculée à cette occasion).
- **MAOStopError** : Cette option permet de définir la fenêtre temporelle pour le calcul de la moyenne mobile de l'erreur calculée sur l'ensemble de stop.

Exemple : Si **MAOStopError** = 5 alors on évalue la moyenne mobile des 5 dernières erreurs calculées sur le corpus de stop.

- **MinStopError** : Cette option peut recevoir une valeur réelle, ou le mot clé **Increase**. Ce dernier indique que l'on arrête l'apprentissage sur une augmentation de l'erreur (ou de la moyenne mobile de l'erreur si **MAOStopError** est utilisée) du corpus de stop. Si on donne une valeur réelle, il y a arrêt de l'apprentissage lorsque l'erreur sur le corpus de stop descend en dessous de cette valeur.
- **NoMaxStep** : Cette option permet d'indiquer au simulateur de s'arrêter au bout d'un nombre fixé d'itérations
- **MinLearningError** : Cette option permet d'arrêter l'apprentissage si l'erreur sur le corpus d'apprentissage descend en dessous de la valeur indiquée.

Le premier des différents critères d'arrêt atteint permet d'arrêter l'apprentissage. Par défaut le nombre d'itérations maximum pour l'algorithme d'apprentissage est fixé à 1000 et la période d'affichage à 100.

Les modes d'initialisation des poids

- **Weightsinit** : Cette option permet d'initialiser les poids des connexions, qui n'auraient pas été initialisées dans les blocs **Connections**. Cette initialisation peut se faire suivant deux lois :

  Normal(moyenne, écart type) : permet d'initialiser les poids suivant la loi normale. La moyenne et l'écart type sont des paramètres facultatifs, dont les valeurs par défaut sont respectivement 0 et 1.

  Uniform(valeur min, valeur max) : permet d'initialiser les poids suivant la loi uniforme. La valeur minimale et la valeur maximale sont des paramètres facultatifs, dont les valeurs par défaut sont respectivement –0,3 et 0,3.
- **AllWeightsInit** : Cette option permet d'initialiser tous les poids des connexions, y compris les poids avec une valeur spécifiée par l'utilisateur. Cette initialisation peut se faire suivant les deux même lois que pour **WeightsInit**.
- **Save** : Cette option permet d'indiquer le nom du fichier de sauvegarde du réseau appris, ainsi que le répertoire où il doit se trouver.

Exemples de fichiers de description

Réseau non récurrent

Le réseau définit ici est constitué de 8 neurones sans compter le neurone de biais, répartis en une couche d'entrée de 4 neurones, une couche cachée de 3 neurones et une couche de sortie constituée d'un seul neurone. Un neurone de biais est connecté aux neurones de la couche cachée et à celui de la couche de sortie.

```
NoNeurons=8;
```

La première couche est constituée de 4 neurones, de fonction de transfert identité.

```
Neurons [1..4] {
  TransfertFN=Identity;
  NoLayer=L1;
}
```

La seconde couche est constituée de 3 neurones, de fonction de transfert sigmoïde de paramètre par défaut 1.

```
Neurons [5..7] {
  TransfertFN=Sigmoid;
  NoLayer=L2;
}
```

La troisième couche est constituée d'une neurone, de fonction de transfert linéaire.

```
Neurons [8] {
 TransfertFN=Linear;
 NoLayer=L3;
}
```

On indique ici que les neurones 1 à 4 sont des entrées. Le neurone 8 est une sortie, donc la troisième couche est une couche de sortie.

```
Input=[1..4];   // equivalent a Input=L1;
Output=[8];     // equivalent a Output=L3;
```

On intègre une neurone de biais, que l'on relie aux neurones 5 à 8, c'est à dire à la couche cachée et à la couche de sortie.

```
Bias {
  target=[5..8];
}
```

On définit maintenant les connections entre les blocs. On connecte ici les neurones de la couche d'entrée aux neurones de la couche cachée.

```
Connections {
```

```
  source=[1..4];    // equivalent a source=L1;
  target=[5..7];    // equivalent a target=L2;
  type=Simple;
}
```

On connecte ici les neurones de la couche cachée aux neurones de la couche de sortie.

```
Connections {
  source=[5..7];    // equivalent a source=L2;
  target=[8];       // equivalent a target=L3;
  type=Simple;
}
```

On indique que les fichiers apprent.set, test.set et use.set se trouvent dans le répertoire `./reseau/non_recurrent/normal/` par rapport au répertoire courant, et que ces fichiers correspondent respectivement aux corpus d'apprentissage, de stop, de test et d'utilisation.

Le corpus d'apprentissage correspond aux lignes 10 à 150 du fichier apprent.set, et le corpus de stop correspond aux lignes 151 à 200 du fichier apprent.set.

```
Script {
  Path="./reseau/non-recurrent/normal/";
  Learning="apprent.set", 10..150;
  Stop="apprent.set", 151..200;
  Test="test.set";
  Use="use.set";
}
```

On indique les paramètres suivants pour l'apprentissage :

- Utilisation de l'algorithme d'apprentissage **`BackPropagation`** avec un pas de 0.005 et une inertie de 0.5. Pas défaut, on l'initialise en mode On-Line.
- La période de calcul de l'erreur sur les corpus d'apprentissage et de stop est de 5.
- La fenêtre temporelle pour le calcul de la moyenne mobile de l'erreur sur le corpus de stop est de 5.
- Arrêt de l'apprentissage sur augmentation de l'erreur sur le corpus de stop (déterminée à partir de la moyenne mobile).
- Nombre maximum d'itérations : 2000.
- Seuil minimal d'arrêt sur le corpus d'apprentissage : 0.001.
- Initialisation des poids de toutes les connexions par une loi normale avec les paramètres par défaut.
- Sauvegarde du réseau appris dans le fichier Model1.dsc contenu dans le répertoire /results/networks/

```
Learning {
  Algorithm=BackPropagation(Online , 0.005 );
  Momentum=0.5 ;
  ErrorDisplayPeriod=5;
  MAOStopError=5;
  MinStopError=Increase;
  NoMaxStep=1000;
```

```
  MinLearningError=0.001;
  AllWeightsInit=Normal;
  save="/results/networks/Model1.dsc";
}
```

Réseau récurrent

```
NoNeurons=7;

Neurons [1..4] {
  TransfertFN=Identity;
}

Neurons [5..6] {
  TransfertFN=Sigmoid;
}

Neurons [7] {
  TransfertFN=Linear;
}

Input=[1..4];
Output=[7];

Bias {
  target=[5..7];
}

Connections {
  source=[1..4];
  target=[5..6];
  type=Simple;
}
```

Les connections suivantes sont à l'origine de la récurrence du réseau.

```
Connections {
  source=[5..6];
  target=[5..6];
  type=Simple;
}
```

On fixe les poids des connections qui relient les neurones 5 et 6 au neurone 7 à 0.5 et 0.2.

```
Connections {
  source=[5..6];
  target=[7];
  type=Simple;
  Weights=[0.5, 0.2]
}
```

On utilise l'option **StateDisplay**, pour obtenir en sortie la valeur de tous les neurones sur l'ensemble d'utilisation.

```
Script {
  Path="./reseau/recurrent/normal/";
  Learning="apprent.set";
  Stop="stop.set";
  Test="test.set";
  Use="use.set", StateDisplay;
}
```

On initialise seulement les poids qui ne possèdent pas encore de valeurs, en utilisant la loi normale de moyenne = 2.0 et d'écart type = 1.0

```
Learning {
  Algorithm=RTRL(0.005);
  Momentum=0.5 ;
  ErrorDisplayPeriod=5;
  MAOStopError=5;
  MinStopError=Increase;
  NoMaxStep=1000;
  MinLearningError=0.001;
  WeightsInit=Normal(2, 1);
  save="/results/networks/Model2.dsc";
}
```

Réseau récurrent à couches

Le réseau suivant est un réseau récurrent à couches Il possède une couche d'entrée avec un neurone, une couche de sortie avec un neurone, et une couche cachée de 3 neurones. Cette couche cachée est totalement récurrente.

```
NoNeurons=5;

Neurons [1] {
  TransfertFN=Identity;
  NoLayer = L1 ;
}

Neurons [2..4] {
  TransfertFN=Sigmoid;
  NoLayer = L2 ;
}

Neurons [5] {
  TransfertFN=Linear;
  NoLayer = L3 ;
}

Input=L1;
Output=L3;

Bias {
```

```
  target=L2, L3;
}

Connections {
  source=L1;
  target=L2;
  type=Simple;
}
```

Les connections suivantes sont à l'origine de la récurrence du réseau.

```
Connections {
  source=L2;
  target=L2, L3;
  type=Simple;
}
```

On utilise l'option **`StateDisplay`**, pour obtenir en sortie la valeur de tous les neurones sur l'ensemble d'utilisation.

```
Script {
  Path="./reseau/recurrent/normal/";
  Learning="apprent.set";
  Stop="stop.set";
  Test="test.set";
  Use="use.set", StateDisplay;
}
```

On initialise tous les poids du réseau en utilisant la loi normale de moyenne = 2.0 et d'écart type = 1.0

```
Learning {
  Algorithm=BPTT(0.005);
  ErrorDisplayPeriod=5;
  MAOStopError=5;
  MinStopError=Increase;
  NoMaxStep=1000;
  MinLearningError=0.001;
  AllWeightsInit=Normal(2, 1);
  save="/results/networks/Model3.dsc";
}
```

Format des fichiers de données (corpus)

Le format des fichiers de données (apprentissage, stop, test, use) est des plus simples : les données sont stockées dans des fichiers textes, sont séparées par des espaces, des tabulations ou des sauts de ligne.

Chaque donnée peut contenir un point décimal on non (alors qu'il est obligatoire dans les fichiers de description).

Des contrôles sont effectués par le simulateur pour vérifier que le nombre de données est un multiple du nombre de neurones visibles (entrées + sorties) pour les ensembles d'apprentissage, de stop et de test, et que le nombre de données est un multiple du nombre d'entrées pour un fichier use.

Dans la pratique, la lisibilité se trouve améliorée si chaque exemple (entrées, sorties) occupe une ligne et une seule.

Rappelons que pour les réseaux récurrents il faut respecter un décalage entre les entrées et les sorties associées. Ce décalage correspond au nombre de connexions qui se trouvent sur le plus court chemin entre les neurones d'entrée et ceux de sortie dans le graphe d'interconnexion.

Par exemple, sur un problème de prévision mensuelle à l'aide d'un réseau récurrent dont le plus court chemin entre les neurones d'entrée et ceux de sortie est de longueur 2, le décalage entre une entrée et sa sortie associée est de 2 mais attention, la sortie associée étant la prévision du mois suivant, en final le décalage est de 1 — voir ci-dessous.

Fichier d'apprentissage (la première colonne pour les entrées, la seconde pour les sorties) :

```
Valeur mois d'avril          0.0
Valeur mois de mai           0.0
Valeur mois de juin          Valeur mois de mai (associée à
                                  l'entrée du mois d'avril)
Valeur mois de juillet       Valeur mois de juin
Valeur mois d'août           Valeur mois de juillet
...                          ...
```

Bibliographie

Back, A. D. and A. C. Tsoi (1991). "FIR and IIR Synapses, a New Neural Network Architecture for Time Series Modeling." Neural Computation **3**(3): 375-385.
Rumelhart, D. E., G. E. Hinton, et al. (1986). Learning Internal Representations by Error Propagation, Parallel Distributed Processing, Explorations in the Microstructure of Cognition. Cambridge, MA, MIT Press.
Unnikrishnan, K. P. and K. P. Venugopal (1992). Learning in Connectionist Networks Using the Alopex Algorithm. International Joint Conference on Neural Networks, Baltimore, USA, IEEE.
Wan, E. (1990). Temporal backpropagation for FIR neural networks. International Joint Conference on Neural Networks, San Diego, USA.
Williams, R. J. and D. Zipser (1989). "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks." Neural Computation **1**: 270-280.