Mémoire

Sujet:

Place et mise en application des réseaux de neurones dans les jeux vidéo.

Problématique:

L'apprentissage numérique est-il présent dans les jeux vidéo?

Plan:

Apprentissage numérique
Présence actuelle dans les jeux vidéo
Description des différents modes d'apprentissage

- Mode supervisé
    - (les plus utilisés/courant)
    - exemples appliqués aux jeux
    - points/forts contraintes
- Mode non supervisé
    - (les plus utilisés/courant)
    - exemples appliqués aux jeux
    - points/forts contraintes
- Mode semi-supervisé ou hybride
    - (les plus utilisés/courant)
    - exemples appliqués aux jeux
    - points/forts contraintes

Acquisition des données par les réseaux de neurones

- Carte de Kohonen
- Perceptron simple et multicouches (PMC)

Projet

Apprentissage numérique

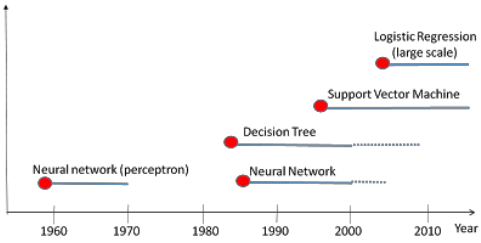
Définition:

L'apprentissage dit numérique (ou automatique) est un apprentissage qui est réalisé par un ordinateur (ou du moins une machine) à partir de différentes données qui lui sont donné en entrée. Le but principal de cette méthode est de passer outre les limites d'algorithmes classiques par exemple des arbres de décisions qui peuvent très compliqués pour la classification de données, on parle alors d'explosion combinatoire. Le traitement des entrées de ce système est réalisé à l'aide de différents algorithmes suivant la méthode d'apprentissage qu'on lui impose, mais il est toujours basé sur de la probabilité ce qui induit une possibilité d'erreur de sortie.

Il existe plusieurs modes d'apprentissage: le supervisé, le non supervisé et le semi-supervisé (appelé aussi hybride). Ces différents modes seront abordés plus tard dans ce mémoire.

Historique:

Arthur Samuel à été un des premiers à créer un programme permettant de jouer au jeu de dames pour IBM. Sa première version fonctionnelle fut délivrée en 1955 et était capable d'apprendre en jouant contre une autre personne. La technique d'apprentissage supervisée utilisée à l'époque, est celle qu'on appelle plus communément maintenant la méthode de recherche heuristique.



Après cette première avancée, de nombreuses découverte ont été faites, notamment dans les réseaux de neurones avec le Perceptron en 1957 (abordé plus tard dans ce mémoire). Le système ELIZA, permettant de simuler un psychothérapeute dans les années 1960. Puis, après une longue période sans grandes avancées, ce n'est que dans les années 1980 que de nombreuses découvertes ont été faites, notamment dans le domaine de l'intelligence artificielle. On a ainsi pu voir apparaître le système d'arbre décisionnel et les réseaux de neurones multi-couches très répandus dans le cadre de la finance ou d'autres secteur d'activité comme les systèmes postaux pour la reconnaissance des adresses. D'autres découvertes ont eut lieues jusque dans les années 2000, qui ont notamment notamment pues êtres faites grâce à l'arrivée d'internet et donc d'immenses quantités d'informations.

Présence actuelle dans les jeux vidéo

L'apprentissage numérique a commencé à apparaître dans les jeux vidéo en 1996 avec les réseaux de neurones dans *Battlecruiser 3000AD* et dans *Creatures*, puis en 2000 avec *Colin McRae Rally 2.0* (PS1/Dreamcast) et 2001 dans *Black & White*. Au fur et à mesures des années les réseaux de neurones se sont développés et ont commencé à être implémentés dans divers domaines tel que dans la reconnaissance d'objet ou de caractères, les moteurs de recherches, le médical, le secteur boursier, etc… Si l'apprentissage numérique s'est si bien développé ces dernières années dans de nombreux secteurs, il n'a cependant pas spécialement gagner sa place dans le monde du jeu vidéo. En effet, on dénombre seulement quelques grosses licences tel que *Forza Motorsport* (2005) sur Xbox qui avait la capacité d'apprendre à partir de la conduite du/des joueurs, ou bien *Halo 3* (2007) sur Xbox 360 pour la gestion des matchmaking sur internet en fonction des différents skills et statistiques des joueurs. A une échelle moins grande, quelques petits jeux tel que *Galactic Arms Race* ont implémenté la technologie des réseaux de neurones (cgNEAT) afin de créer de manière complètement indépendante des armes pour un vaisseau évoluant dans l'espace. A l'aide d'indication du joueur, le jeu génère les nouvelles armes selon si les anciennes lui ont plu ou non. Dans un autre registre des algorithmes ont été mis en place pour pouvoir résoudre des jeux de logique tel que les échecs ou le jeu de Go, un ancien jeu chinois dont une version digital est sortie sur le XBLA en 2010.

Différents modes d'apprentissage

Dans le monde de l'apprentissage numérique, 3 modes se distinguent : le supervisé, le non-supervisé et l'hybride (une combinaison des 2). Ces modes ont des caractéristiques différentes de fonctionnement, et permettent de regrouper l'ensemble des algorithmes et modèles d'apprentissage.

Notre étude se basant sur les jeux vidéos, seul le mode supervisé sera développé. En effet, les 2 autres modes ne sont que très peu utilisé, ou du moins pas par les développeurs faisant partie de studios.

Mode supervisé

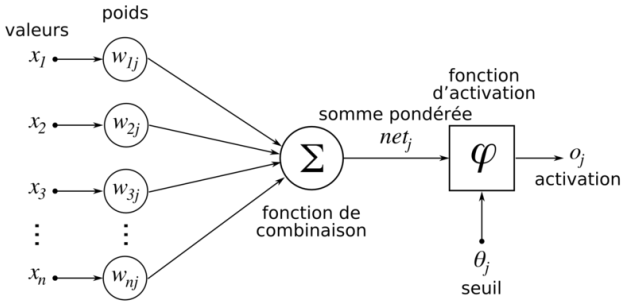
Le but du mode supervisé est d'amener l'ordinateur à apprendre un système de classification fourni au préalable. Celui-ci consiste à apprendre en minimisant le nombre d'erreur tout en respectant les entrées données. Les deux techniques les plus exploitées utilisant ce système, sont les réseaux de neurones et les arbres de décisions. Dans le cadre de la première, la classification est utilisée afin de déterminer le nombre d'erreurs du réseau puis de le modifier afin d'en réduire le nombre, alors que l'arbre décisionnel créé lui un modèle capable par la suite de prédire la valeur d'une variable cible basée sur les entrées reçues.

Les réseaux de neurones artificiels

Définition

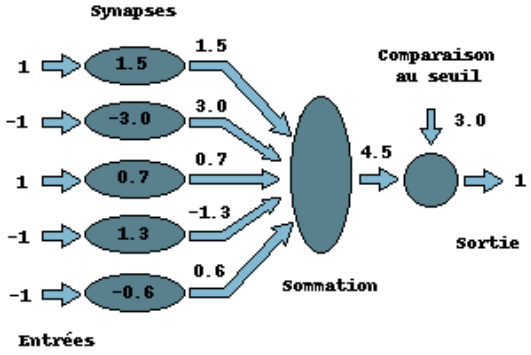
Un réseau de neurone artificiel (RNA) est une manière de traité l'information calqué sur le fonctionnement du système nerveux biologique, comme le cerveau. De le même manière que le cerveau humain utilise un système de réseaux inter-connectés pour résoudre des problèmes, un RNA est composé de nombreuses fonctions reliées entre elles, prenant des données en entrée, et renvoyant une ou plusieurs informations en sortie. Toujours sur le même schéma que le cerveau humain, un RNA à besoin d'être entraîné et doit subir des ajustements afin de répondre le mieux à une situation/problème donné(e).

Fonctionnement



Comme expliqué précédemment un RNA fonctionne grâce à l'inter-connexion de différentes fonctions. Ces connexions sont représentées sous formes de couches, chaque couche est composée d'un certain nombre de neurone recevant chacun leur entrées des neurones de la couche précédente. D'une couche à l'autre, les neurones sont tous reliés entre eux par des synapses qui sont elles-mêmes pondérées d'un poids différent selon l'importance du neurone. La sortie du neurone de la couche précédente est multipliée par ce poids et est ensuite sommée à l'ensemble des autres neurones multipliés par le poids de leur synapse. Arrivé à la dernière couche, la sortie du/des neurone(s) est/sont alors comparé(s) à une valeur seuil, et selon le résultat de la comparaison une décision ou un résultat sera alors renvoyé.

exemple :



Dans le monde du jeu vidéo

Les réseaux de neurones peuvent être utilisés de deux manières différentes dans les jeux vidéo. La première consiste à les utiliser au moment du développement du jeu; ils apprendront à partir de données fournies par les développeurs, puis le résultat final sera inclus dans le jeu. Les développeurs de Codemasters ont utilisé ce processus dans le jeu *Colin McRae Rally 2.0* sorti en 2000 afin d'entraîner une IA permettant à la voiture de rester sur la route, d'avoir un comportement cohérent, mais surtout de pouvoir s'adapter à tout type de circuit, sans avoir été entraîné dessus au préalable.

La deuxième technique, consiste à inclure le RNA directement dans le version finale du jeu, de cette manière, le jeu apprendra à partir du comportement du joueur, et les informations collectées seront alors restituées en jeu. Si cette technique est coûteuse en ressources, avec l'avancée des technologies et des supports il est désormais moins compliqué de l'implémenter dans les jeux. De nos jours, de nombreux jeux l'implémente afin d'offrir une meilleure expérience au joueur, de ce fait il y a moins d'effet de redondance et de linéarité car les PNJs (Personnage Non Joueur) sont capable de s'adapter aux routines du joueur, et par conséquent de l'obliger à changer ses habitudes. L'un des premiers jeux à l'avoir mis en oeuvre est *Creatures* en 1996. Le principe du RNA était de permettre une adaptation du comportement de la créature en fonction des ordres et actions du joueur. Au fur et mesure que le joueur avançait dans le jeu, le comportement et les réactions de la créature évoluaient, ainsi d'un joueur à l'autre des créatures pouvaient avoir des agissements complètement différents. En 2001 Lionhead sort *Black and White*, le but jeu étant là aussi de contrôler une créature, mais elle symbolisant une divinité sur Terre. De la même manière que *Creatures*, le personnage contrôlé s'adapte et change de comportement en fonction de l'éducation que le joueur lui inculque.

Avec l'avancée des technologies, les réseaux de neurones ont énormément évolué ainsi que la manière dont les développeurs s'en servent. En 2005, Microsoft sort le jeu de course *Forza Motorsport* et y inclus sa technologie développée pour l'occasion, *Drivatar*. Celle-ci est capable d'analyser le comportement du joueur et de s'y adapter, mais également de définir le chemin le plus optimisé pour parcourir un circuit puis d'altérer sensiblement ce chemin afin d'obtenir un comportement plus "aléatoire", donc plus proche de celui d'un humain. La récente sortie de la Xbox One (le 22 novembre 2013) ainsi que sa capacité à se connecter dans le cloud, a permis aux développeurs de pousser un pas plus loin l'utilisation de *Drivatar* dans leur dernier jeu *Forza Motorsport 5*. En effet, en plus des possibilités citées précédemment, Drivatar est capable d'analyser et de calquer le style de conduite du joueur. Grâce au cloud, le profil de conduite du joueur est alors disponible dans le monde entier. La résultante et le but de cette prouesse, est d'avoir la sensation de jouer contre son ami sur internet, alors qu'il ne s'agit en fait que d'un programme restituant son style de jeu.

Point négatif

Comme pour tout algorithme d'apprentissage, il y a un danger; celui du surapprentisage. Cela arrive lorsque l'algorithme apprend trop, et par conséquent va par la suite trop restreindre son champs d'action. L'impact majeur est que le résultat obtenu n'est plus du tout conforme aux attentes.

En 2011, un développeur décide de mettre en place un RNA pour générer de bonnes IA pour les bots du jeu *Quake III*. Afin d'obtenir un comportement basé seulement sur le comportement des bots, il en place 16 dans une arène qui interagiront uniquement entre eux, sans intervention humaine, le tout placé sur un serveur à lui. Avec le temps, il oublie qu'il a lancé son expérience, et laisse fonctionner son programme pendant 4 ans. Lorsqu'il se rappel de son expérience et regarde les résultats, il s'attend à avoir des bots avec un comportement très réalistes et performants. Au lieu de ça, il constate qu'aucun des bots ne bougent ni attaquent les autres adversaires. Cela s'explique par le fait que l'algorithme a subit un surapprentissage, et à déterminer que le meilleur moyen de gagner était celui de ne pas attaquer, ni même de bouger.
Cet échec démontre bien la limite des RNA, et qu'il faut bien déterminer à quel moment l'algorithme à atteint le meilleur de son "raisonnement".

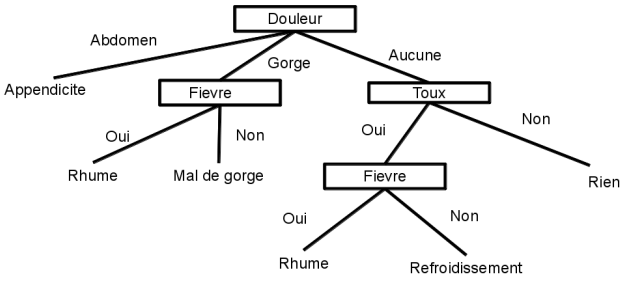
Les arbres décisionnels

Définition

Un arbre décisionnel est un graphe représentant une structure composée de décisions et d'actions reliées par des branches sur différents niveaux. Le but d'un arbre de décisions est de pouvoir mettre en place un système simple et clair aidant à faire des choix selon une situation donnée.

Fonctionnement

L'arbre de décision commence généralement avec un choix à faire. Chaque choix est représenté par une branche menant elle même à une autre décision à prendre, appelée plus communément noeud. Ainsi, un arbre est la représentation d'une succession de noeud reliés par des branches. Si un noeud n'a pas de branche, il est alors appelé feuille et représente la décision finale de l'arbre.



Dans un arbre décisionnel classique, un noeud peut avoir autant de branches que nécessaire, cependant il existe un autre type d'arbre, dit binaire, qui lui ne peut avoir qu'au maximum 2 branches.
Les points positifs d'un arbre de décision, sont qu'ils sont faciles à implémenter, rapide d'exécution et simple de compréhension.

Voici un pseudo code représentant l'image d'exemple ci-dessus:

```
Si douleur = abdomen alors
        problèmeSanté = Appendice
Sinon Si douleur = Gorge alors
```

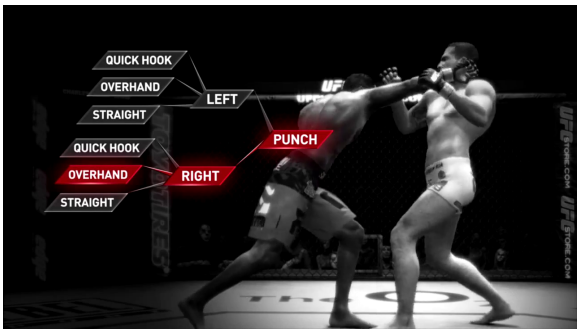
```
        Si Fièvre = Vrai alors
                problèmeSanté = Rhume
        Sinon
                ProblèmeSanté = MalDeGorge
        FinSi
Sinon
        Si Toux = Vrai alors
                Si Fievre = Vrai alors
                        problèmeSanté = Rhume
                Sinon
                        problèmeSanté  = Refroidissement
                FinSi
        Sinon
                problèmeSanté = Aucun
        FinSi
FinSi

Renvoyer problèmeSanté
```

Dans le monde du jeu vidéo

L'arbre de décisions n'est pas la technologie la plus répandue dans l'univers du jeu vidéo. Cependant elle a longtemps été utilisée dans les jeux de stratégie (STR) afin de faire du *path finding*. Même si les résultats pouvaient parfois être douteux (ex: une armée entière essayant de passer entre deux murs très rapprochés). Cependant, cette technique suffisait à remplir tant bien que mal ses tâches. Depuis d'autres algorithmes tel que A* ou Djikstra ont pris le relais dans ce domaine là.

C'est en 2012 que THQ sort le jeu UFC Undisputed 3. Le système de combat du jeu est basé sur des arbres décisionnels déclenchés selon certaines situations. Par exemple, si un des combattants ne cesse de se protéger au visage, l'adversaire va le percevoir, et parcourir un de ses arbres pour voir qu'elle action est la plus adaptée, ce qui pourrait être d'attaquer au niveau du ventre.



Dans un autre registre, c'est en 2007 qu'est apparu le site web Akinator, un génie sorti tout droit de sa lampe pour nous dire à qui on pense. Le système est simple, il suffit que le joueur pense à quelqu'un et réponde à une série de question concernant le personnage. Les questions sont toutes de type fermées, c'est à dire qu'on peut seulement répondre par oui ou non. L'architecture de l'algorithme est gardé secrète par le développeur, Arnaud Megret, cependant on sait que le système est basé sur un arbre décisionnel binaire. De ce fait, pour chaque question posée par l'ordinateur, la réponse donnée par le joueur élimine un lot de possibilités, et lorsque l'algorithme "pense", avoir trouvé la bonne personne, il l'envoi au joueur. Il est cependant

possible qu'une mauvaise personne soit soumise au joueur, soit parce qu'Akinator ne connait pas la personne en question, ou bien parce que le joueur à répondue n'importe quoi. Dans le cas de la première hypothèse, l'algorithme est capable d'apprendre le personnage en demandant simplement la bonne réponse au joueur, ainsi avec la succession de réponse fournies par le joueur, le programme sera en mesure d'apprendre les caractéristique du personnage. Ce système bien que bluffant lorsqu'on y joue, repose sur un système simple, ou le parcours d'un simple arbre suffit à déterminer la meilleure solution. L'élément clé de cette application étant bien sur la base de données et la capacité d'apprentissage.

Acquisition des données par les réseaux de neurones

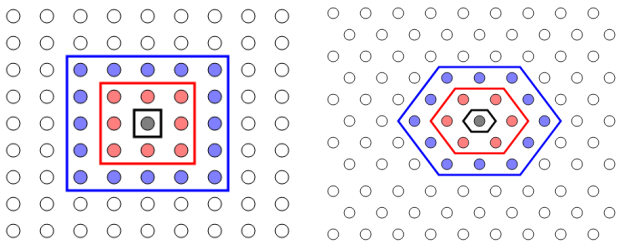
Carte de Kohonen

La carte de Kohonen, aussi appelée carte auto adaptive (Self Organizing Map - SOM) à été créée en 1984 par Teuvo Kohonen et se classe parmi les modes d'apprentissage non supervisé. En effet, le but est d'organiser des données discrétisées et représentées par des vecteurs associés à des neurones faisant partis d'une carte (grille). A chaque neurone est ensuite associer un poids qui sera amené à évolué pendant la phase d'apprentissage.

Le principe est de parcourir l'ensemble des données et de les comparer avec les différents neurones. Celui qui "réagira" (correspondra) le mieux, sera alors le neurone dit gagnant et représentera les données. L'ensemble des neurones entourant le neurone gagnant seront également affectés afin qu'ils se rapprochent d'avantage des données de ce dernier. La phase d'apprentissage se termine lorsque tous les neurones sont stables.

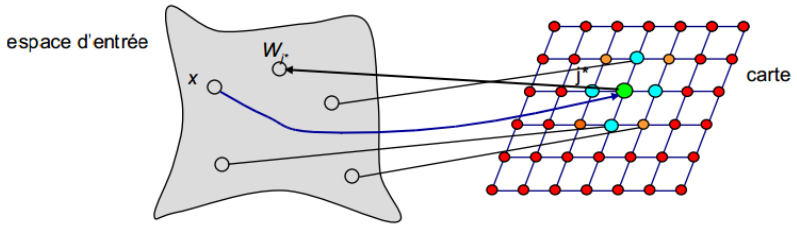
Les cartes associées peuvent être de dimensions différentes :

Unidimensionnelle

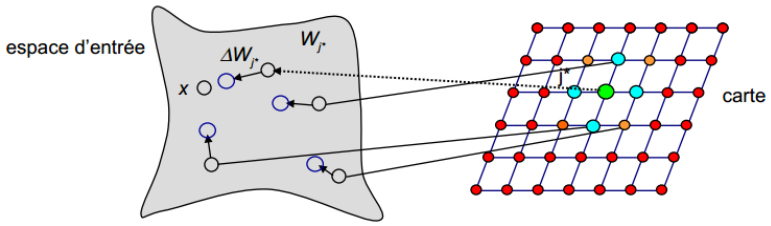


Bidimensionnelle

Exemple d'organisation sur une grille bi-dimensionnelle de type carrée, c'est à dire avec 4 voisins.



Sélection du neurone le plus proche de l'exemple X



Mise à jour des poids des neurones entourant le neurone gagnant

Comme expliqué précédemment, chaque neurone de la carte est associé à un poids noté *W*. Lors de la phase d'apprentissage, les itérations sont notées *t*.

A chaque itération *t*, un exemple d'apprentissage *X(t)* est choisi et comparé à l'ensemble des neurones. Le neurone gagnant *j** est celui dont le vecteur poids $W_{j^*}(t)$ est le plus proche de *X(t)*.

$$d_N [X(t) , W_{j^*}(t)] = min_j \, d_N [X(t) , W(t)]$$

avec :

- d_N distance dans l'espace d'entrée

La détermination des neurones voisins du neurone gagnant se fait de la manière suivante:

$$h_{j^*} (j , t) = h [d(j , j^*), t]$$

avec :

- *d* distance dans la carte
- *h* fonction de voisinage

Puis, la mise à jour des poids se fait ainsi :

$$W_j (t + 1) = W_j (t) + \Delta W_j (t)$$

$$\Delta W_j(t) = \varepsilon(t) \cdot h_{j^*}(j, t) \cdot (X(t) - W_j(t))$$

avec :

- ε pas d'apprentissage

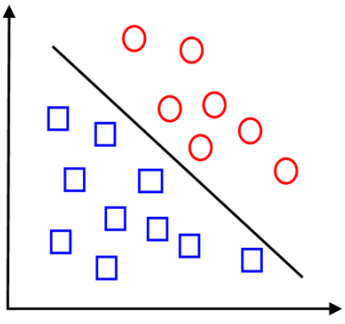
Perceptron simple et multicouches

Classifieur linéaire

Un classifieur linéaire représente un type d'algorithmes d'intelligence artificielles qui va permettre de trier des échantillons à partir d'une combinaison linéaire des propriétés de chaque échantillon. Cette combinaison est aussi appelée fonction de décision (ici *g(x)*, mais on la trouve souvent noté *h(x)*). Elle s'exprime donc par une fonction linéaire en fonction des propriétés de chaque échantillon (dans notre cas noté *x*). Chacune de ces propriétés est soumises à un poids qui évoluera au cours de l'apprentissage (appelé *w*), généralement on ajoute un biais comme propriété à chaque échantillon ainsi qu'un poids qui sera lui aussi modifié au cours du temps (qui est noté *w0* ou encore *b*). Et enfin, la fonction *f()* est une fonction qui permet de convertir le produit scalaire entre deux propriétés en la sortie attendue, cette fonction peut être la fonction de signe, la fonction de Heaviside ou encore la fonction sigmoïde. Ci-dessous vous pouvez voir la formule qui résume ce que fait un classifieur linéaire.

$$g(x) = f(w^T x + w_0) = f\left(\sum_{j=1}^{N} w_j x_j + w_0\right)$$

Finalement, la représentation graphique de ce genre classifieur linéaire correspond à faire une séparation avec un hyperplan entre les données de deux classes différentes, comme on peut le voir dans la figure suivante.

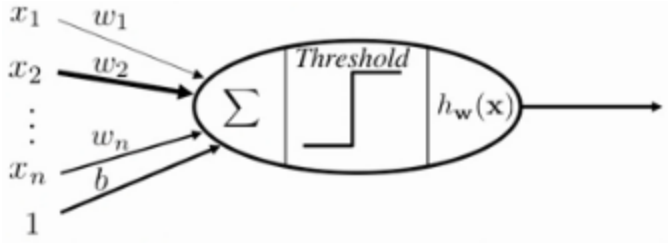


Cependant, les classifieurs sont basés sur les probabilités ce qui peut parfois introduire des erreurs de classement, mais plus le nombre d'échantillons est important et meilleur seront les résultats.

Perceptron

Le perceptron est un classifieur linéaire inventé en 1957 par Frank Rosenblatt qui a été inspiré par les théories cognitives de Friedrich Hayek et de Donald Hebb. Le perceptron est souvent considéré comme le plus simple des réseaux de neurones, et il est de type supervisé.

L'idée générale du perceptron est de reprendre modéliser une décision à partir d'une fonction linéaire et d'y ajouter une fonction de seuil (comme on l'a vu précédemment avec le classifieur linéaire). Nous avons donc en entrée un échantillon/vecteur que l'on nomme x, qui est décomposé en plusieurs éléments qui sont notés de x_1 à x_n. Sur chacun de ces éléments sont appliqués des poids de w_1 jusqu'à w_n, ensuite il faut appliquer le produit scalaire de tous ces poids aux les éléments du vecteur : $x_1.w_1,...,x_n.w_n$, et en faire la somme. Le résultat ainsi obtenu sera donnée en entrée à la fonction de seuil (noté ici Threshold) qui va permettre de savoir si cet élément correspond à la classe 1 si le résultat de la somme est supérieur ou égal à 0 sinon il appartient à la classe 0.



On ajoute également le biais comme on l'a vu pour le classifieur linéaire, afin de faire varier le seuil de comparaison et donc d'avoir une bonne classification des données.

Le perceptron a tout de même ses limitations, il ne peut résoudre que des problèmes linéaires séparables. Il ne peut par exemple pas résoudre le cas du XOR ("ou-exclusif") à lui seul, mais une utilisation d'un perceptron multi-couches permettra de résoudre ce problème.

Algorithme Perceptron simple couche

L'algorithme du perception simple couche est assez facile à mettre en place. Pour cela, il faut dans un premier temps définir la valeur du taux d'apprentissage et ensuite suivre les consignes suivantes :

Tant qu'on a pas obtenu le critère d'arrêt définit (soit le nombre d'erreurs vaut 0, soit le nombre maximal d'itérations est atteint)
    Pour chaque vecteur de la base d'entraînement x_t
        Calcule du produit vectoriel de chaque élément du vecteur par son poids
        Récupération de la valeur de sortie de Threshold(x_t . w) dans $h(x_t)$
        Si $h(x_t)$ est différent du résultat attendu y_t
            Pour tous les éléments du vecteur et le biais ($x_{t,i}$)
                Mettre à jour des poids avec la formule suivante
                $w_i \leftarrow w_i$ + TAUX_APPRENTISSAGE * (y_t - $h(x_t)$) * $x_{t,i}$
            Fin Pour
        Fin Si
    Fin Pour
Fin Tant que

Dans ce pseudo-code, on retrouve bien la formule mathématique qui a été expliquée précédemment. Vous pourrez retrouver directement cette implémentation d'algorithme en C++ dans l'annexe sur un exemple quelconque.

Rétropropagation du gradient de l'erreur

Cet algorithme a été inventé en 1984 par Paul J. Werbos et a été mis en place pour la première fois en 1986 par David Rumelhart. Le but de cet algorithme est de minimiser les erreurs dues à la différence entre la/les sorties attendues et celle(s) obtenue(s), en recalculant les poids des synapses de chaque neurone. Pour cela, on effectue une modification de ces poids en partant de la dernière couche vers la première.

Afin de réaliser la rétropropagation du gradient, la méthode à appliquer est celle mettant en place les dérivées partielles, ainsi que ce que l'on appelle le gradient.

Fonction de minimisation de l'erreur

On cherche à minimiser l'erreur entre la sortie attendue (y_t) et celle obtenue ($h_w(x_t)$). Pour la fonction de seuil (Threshold) que l'on a vu précédemment, on va donc effectuer la différence entre ces deux valeurs que l'on multiplie ensuite par le produit scalaire du poids de l'élément (w) par la valeur de cet élément (x_t).

$$Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) = -(y_t - h_{\mathbf{w}}(\mathbf{x}_t))\mathbf{w} \cdot \mathbf{x}_t$$

La fonction de minimisation est différente suivant le type de fonction d'activation du neurone. Si nous prenons la fonction sigmoïde qui permet de faire des probabilités sur l'appartenance d'un élément à une classe, nous allons obtenir une fonction de minimisation plus complexe que celle de la fonction de seuil.

$$h_{\mathbf{w}}(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

Fonction de décision de la fonction sigmoïde

$$Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) = -y_t \log h_{\mathbf{w}}(\mathbf{x}_t) - (1 - y_t) \log(1 - h_{\mathbf{w}}(\mathbf{x}_t))$$

Le résultat ainsi obtenu par cette fonction de seuil, nous permet de savoir si la prédiction a été correctement faite avec 0, sinon c'est que la prédiction est mauvaise, et par conséquent, on va pouvoir calculer le seuil à dépasser pour qu'elle soit bonne.

Les dérivées partielles

Une dérivée partielle correspond à l'état plus général d'une dérivée. Elle s'applique aux fonctions qui dépendent de plus d'un élément. Pour effectuer cette dérivée partielle, on va dériver notre fonction sur chacune des variables en considérant les autres variables comme des constantes.

$$\frac{\partial f(x,y)}{\partial x} = \lim_{\Delta \to 0} \frac{f(x+\Delta, y) - f(x,y)}{\Delta}$$

$$\frac{\partial f(x,y)}{\partial y} = \lim_{\Delta \to 0} \frac{f(x, y+\Delta) - f(x,y)}{\Delta}$$

Dans le cas ci-dessus, nous avons une fonction *f* qui dépend de deux variables. Nous allons donc faire la dérivation dans un premier temps sur *x*, puis dans un second temps sur *y*.

Dans le cas de la fonction de seuil, les dérivées partielles sont:

$$p(y_t = 1|\mathbf{x}) = 1 - h_{\mathbf{w}}(\mathbf{x}_t)$$
$$p(y_t = 0|\mathbf{x}) = 0 - h_{\mathbf{w}}(\mathbf{x}_t)$$

Tandis que dans le cas de la fonction sigmoïde, les dérivées partielles sont :

$$p(y_t = 1|\mathbf{x}) = h_{\mathbf{w}}(\mathbf{x}_t)$$
$$p(y_t = 0|\mathbf{x}) = 1 - h_{\mathbf{w}}(\mathbf{x}_t)$$

Le gradient

Il correspond à un vecteur contenant toutes les dérivées partielles d'une fonction f. Pour illustrer cela, on reprend notre exemple précédent, et on note le gradient ainsi obtenu.

$$\nabla f(x,y) \quad = \quad \left[\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y} \right]$$

Rétropropagation

Il faut maintenant calculer le gradient de ces pertes afin d'effectuer la rétropropagation. Pour cela, on peut utiliser plusieurs méthodes. La première méthode correspond à la forme basique de la rétropropagation qui nécessite de calculer la moyenne des dérivées partielles pour tous les vecteurs de la base de connaissance avant de pouvoir faire la mise à jour des poids des synapses, donc énormément de calcul avant de faire cette mise à jour.

La deuxième méthode, que l'on appelle la descente de gradient stochastique, permet de mettre à jour les poids en fonction du gradient d'une seule entrée de la base de connaissance prise au hasard (les poids étant bien entendu initialisés au départ de l'algorithme), on répète cette opération pendant un certain nombre d'itérations que l'on définit arbitrairement ou lorsque le seuil d'erreur est atteint. Lors de la mise à jour des poids, on applique également un facteur d'apprentissage (α) qui correspond à notre avancée pour obtenir l'erreur la plus petite possible. L'avantage de cette méthode est dotant plus efficace lorsque notre base de connaissance est importante, car on peut justement arrêter l'algorithme plus rapidement et faire beaucoup plus de mises à jour des poids.

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \quad \forall i$$

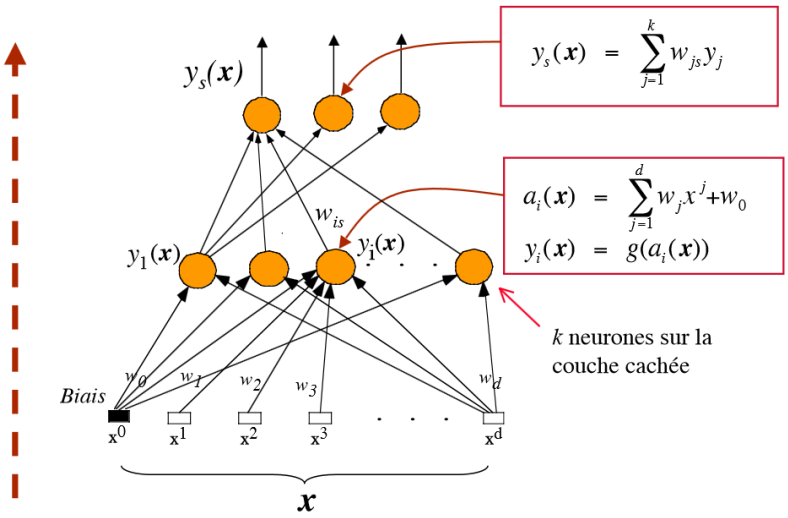
Un troisième algorithme existe qui est la descente de gradient stochastique par mini-lots. Cette méthode consiste à effectuer des moyennes de dérivées partielles d'un certain nombre d'éléments de la base de connaissance (10, 20, 100, un valeur que l'on trouve pertinente). L'avantage est d'effectuer non pas des produits d'un vecteur avec matrice (poids avec tous les éléments d'une entrée de la base), mais d'effectuer un produit matriciel entre les poids des éléments sélectionnés et la valeur de ces éléments. Cette solution est en réalité un mixte des deux précédents algorithmes, si on augmente trop le nombre d'éléments à traiter, on arrive dans le traitement de la rétropropagation basique, tandis que si on rabaisse le nombre d'éléments à 1, on passe l'algorithme devient de la rétropropagation stochastique.

Il y a encore un autre algorithme que l'on appelle Resilient Propagation (RProp) créé en 1992 par Martin Riedmiller et Heinrich Braun. qui lui ne prend pas en compte de facteur d'apprentissage. Il se base seulement sur les signes des dérivées partielles de tous les éléments du vecteur d'entrée, c'est à dire si le signe de la dérivée est le même qu'à la précédente itération alors on pondère le poids par le facteur η^+ qui est la plus part du temps mis à 1.2, si le signe est différent alors c'est qu'on a dépassé le minimum local et donc on va changer le signe du poids de notre neurone et le multiplier par η^- qui est généralement positionné à 0.5, et enfin le dernier cas c'est si les valeurs des dérivées n'ont pas changé alors on va continuer à appliquer les changements faits à l'itération précédente. Avec cette dernière méthode, on effectue moins de calcul et donc on gagne en performance.

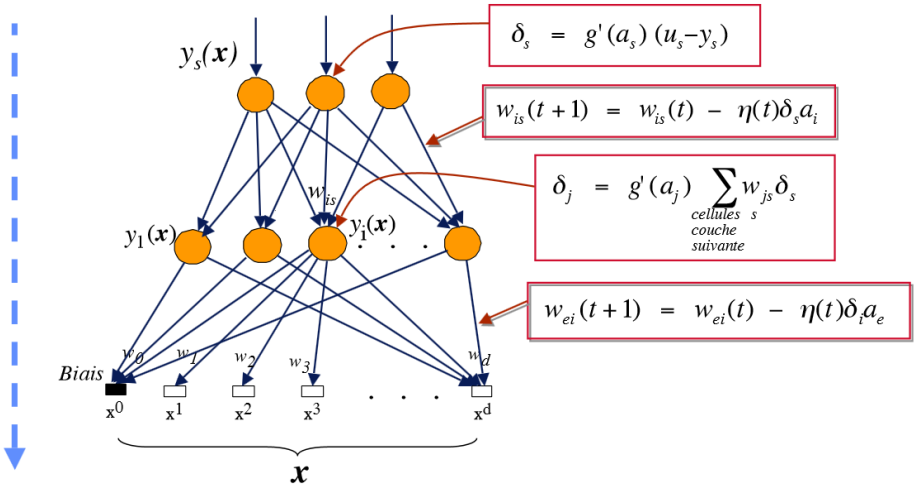
Algorithme Perceptron multi-couches

Le perceptron multi-couches est un réseau de neurones artificiels où les informations vont toujours vers la sortie. La toute première couche du réseau de neurones correspond aux éléments qui composent notre vecteur d'entrée. Tandis que la couche finale correspond à la sortie du réseau. Tous les neurones sont reliés par des liaisons pondérées, comme on l'a vu précédemment avec le perceptron simple couche entre les entrées et la fonction de seuil.

La partie apprentissage du perceptron multi-couches est basé sur la rétropropagation du gradient de l'erreur, on cherche donc à faire une recherche de minimisation de l'erreur entre la sortie obtenue et celle attendue.



Sens de la propagation



Sens de la rétropropagation de l'erreur

L'algorithme du perceptron est finalement assez simple à mettre en place, il peut être découpé en trois étapes :

1- La propagation des entrées jusqu'à la couche de sortie

2- Le calcul de l'erreur en sortie et rétropropagation de l'erreur

3- La correction des poids

Plusieurs paramètres rentrent en compte dans le déroulement de cet algorithme, il y a, dans un premier temps, le type de la fonction d'activation (de décision) des neurones, par exemple, la fonction de seuil ou sigmoïde, et dans un second temps, le nombre d'entrées de la base de connaissance qui va influencer le choix de la méthode de rétropropagation du gradient de l'erreur.

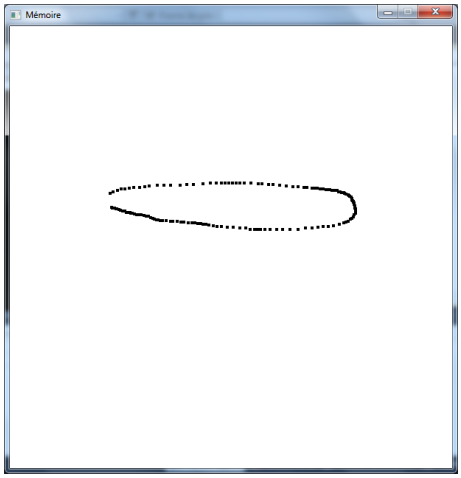
Un exemple d'un perceptron multi-couches est présent en annexe, dans cet exemple, nous avons intégré la possibilité de choisir entre la fonction de seuil et sigmoïde comme choix d'activation, et nous avons mis en place la rétropropagation du gradient stochastique. Son but est d'apprendre à la fonction XOR.

Le perceptron multi-couches est présent dans les jeux vidéo tels que Colin McRae 2.0 afin d'améliorer les IA du jeu et donc pour obtenir la meilleure conduite possible. Dans leur algorithme, les développeurs ont utilisé la méthode de rétropropagation du gradient RPROP car ils le considèrent comme le plus rapide. Ils ont décidé de mettre en place ce genre d'algorithme car le nombre de cas à prévoir pour la gestion de la conduite était trop énorme et trop complexe.

Projet

Description du projet

Le projet, que nous allons développer afin d'illustrer ces différentes méthodes de classification, consiste à réaliser de la reconnaissance de formes, ces dernières sont faites par un utilisateur. L'apprentissage des formes peut être réalisé par soit la carte de Kohonen soit le perceptron multicouches. Les formes, que l'utilisateur feront, seront des représentations de comportement d'ennemis, c'est à dire que l'utilisateur va par exemple dessiner la forme ci-dessous, et notre application pourra lui indiquer quel comportement il a représenté, c'est à dire un comportement de ronde.



Ce projet sera développé en C++ avec OpengGL 2.0.

Bibliographie:

http://fr.wikipedia.org/wiki/R%C3%A9seau_de_neurones_artificiels
http://fr.wikipedia.org/wiki/Apprentissage_automatique
http://www.grappa.univ-lille3.fr/polys/apprentissage/sortie005.html
http://wikistat.fr/pdf/st-m-app-rn.pdf
http://www.trop.uha.fr/pdf/cours-wira.pdf
http://olivier.teytaud.pagesperso-orange.fr/publis/serpilliere.pdf
Apprentissage :
http://www-igm.univ-mlv.fr/~dr/XPOSE2002/Neurones/index.php?rubrique=Apprentissage
http://eric.univ-lyon2.fr/~ricco/cours/slides/reseaux_neurones_perceptron.pdf
http://fr.wikipedia.org/wiki/Carte_auto_adaptative
Apprentissage numérique :
http://veille-techno.blogs.ec-nantes.fr/index.php/2013/02/13/apprentissage-automatique-et-reseaux-de-neurones/
http://www.youtube.com/playlist?list=PL6Xpj9I5qXYGhsvMWM53ZLfwUInzvYWsm
http://www.researchgate.net/publication/228933107_Exploiting_the_fascination_Video_games_in_machine_learning_research_and_education/file/79e4151290bcba996c.pdf
http://videolectures.net/mlss05au_graepel_mlg/
http://research.microsoft.com/en-us/events/2011summerschool/jqcandela2011.pdf
http://www.cse.lehigh.edu/~munoz/CSE497/classes/MeganNeural.ppt
http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node109.html
http://www.mlplatform.nl/what-is-machine-learning/
http://videolectures.net/mlss05au_graepel_mlg/
http://thinkingmachineblog.net/use-of-machine-learning-in-video-games/
http://en.wikipedia.org/wiki/Decision_tree_learning
Apprentissage supervisé :
http://www.aihorizon.com/essays/generalai/supervised_unsupervised_machine_learning.htm
Neural network:
http://gamingbolt.com/forza-motorsport-5-wiki
http://research.microsoft.com/en-us/projects/drivatar/
http://www.ift.ulaval.ca/~lamontagne/ift17587/projets%20pass%C3%A9s%20GameAI.pdf
http://www.cplusplus.com/forum/lounge/105609/
http://www.idt.mdh.se/kurser/ct3340/ht11/MINICONFERENCE/FinalPapers/ircse11_submission_5.pdf
http://gameai.com/
http://i.imgur.com/dx7sVXj.jpg
Arbre décisionnel:
http://www.youtube.com/watch?v=G29LTOkZ8RA&feature=youtube_gdata
http://web.cs.wpi.edu/~rich/courses/imgd4000-d09/lectures/B-AI.pdf
http://www.grappa.univ-lille3.fr/polys/apprentissage/sortie004.html

http://fr.wikipedia.org/wiki/Arbre_de_d%C3%A9cision
http://web.cs.du.edu/~sturtevant/ai-s11/Lecture03.pdf
http://www.nuddz.com/questions/Comment-fonctionne-l-algorithme-de-Akinator/541
Carte de Kohonen
http://fr.wikipedia.org/wiki/Carte_auto_adaptative#Principe
http://en.wikipedia.org/wiki/Self-organizing_map
http://mimosa.cereq.fr/rousset/M2S3_octobre.pdf
http://eric.univ-lyon2.fr/~rias2006/presentations/VincentLemaire.pdf
http://beefchunk.com/documentation/ia/cours_IA/node101.html
http://asi.insa-rouen.fr/enseignants/~scanu/RdF/TP/Kohonen.htm
http://mnemstudio.org/neural-networks-kohonen-self-organizing-maps.htm
http://pro.chemist.online.fr/cours/koho1.htm
http://www.timotheecour.com/papers/vision_system.pdf
Classifieur linéaire
http://fr.wikipedia.org/wiki/Classifieur_lin%C3%A9aire
http://pageperso.lif.univ-mrs.fr/~guillaume.stempfel/Memoire.pdf
http://samifis.irisib.be/2011/01/07/classifieurs-lineaires-svm/
http://dspace.univ-tlemcen.dz/bitstream/112/322/11/ChapitreII.pdf
Perceptron
http://fr.wikipedia.org/wiki/Perceptron
http://wcours.gel.ulaval.ca/2010/a/GIF4101/default/5notes/ar-sem10-pmc.pdf
http://en.wikipedia.org/wiki/Multilayer_perceptron
http://fabien.tschirhart.free.fr/images/Docs/memoire_V129.pdf
http://wcours.gel.ulaval.ca/2009/h/19968/default/5notes/NotesDeCours/RetroPerceptron.pdf
http://themarvinproject.free.fr/final/node3.html#SECTION034000000000000000000
http://www.generation5.org/content/2001/hannan.asp
http://www.ai-junkie.com/misc/hannan/hannan.html
https://www.lri.fr/~antoine/Courses/ENSTA/Tr-ensta-nnx9.pdf
http://matlabgeeks.com/tips-tutorials/neural-networks-a-multilayer-perceptron-in-matlab/
Rétropropagation du gradient de l’erreur
http://en.wikipedia.org/wiki/Backpropagation
http://www.iro.umontreal.ca/~bengioy/ift6266/H12/html/gradient_fr.html#le-gradient
http://rfia2012.files.wordpress.com/2012/02/descente-stochastique.pdf
http://en.wikipedia.org/wiki/Rprop
http://www.heatonresearch.com/wiki/Resilient_Propagation
Projet
http://www.ifi.auf.org/site_data/rapports/tpe-promo10/tipe-pham_quang_dung.pdf