Faculté
des Sciences

Un réseau de neurones pour la

Rapport

Jason Bury
130538
Master 1 en
Sciences informatiques

Codirecteurs :
M. Pierre Hauweele
M. Hadrien Mélot
Rapporteur :
M. Tom Mens

Année 2016-2017

Table des matières

1 Introduction et objectif

1.1 Énoncé

Ce projet consiste à implémenter un *réseau de neurones artificiels* (RNA) pour commander la Sphero. Il devra être capable de gérer des trajectoires à grande vitesse mais également les dérapages. La Sphero sera commandée sur un sol plat sans obstacle.

1.2 Présentation de la Sphero

La Sphero est une boule robotisée téléguidée, connectée par Bluetooth. À l'intérieur, il y a deux moteurs : un pour faire tourner le contre-poids et l'autre pour modifier l'orientation de l'appareil. En faisant tourner le poids, la Sphero déplace son centre de gravité en dehors de sa base de sustentation [1], faisant ainsi rouler l'appareil.

1.3 Avantages d'un réseau de neurones

L'apprentissage et la génération de vecteur de sortie est massivement parallélisable dans un RNA. [1, 2] De plus, il répond de manière raisonnable à une entrée non rencontrée durant la phase d'apprentissage. Cela s'appelle la *généralisation*. [2,3] Les RNA présentés dans ce rapport sont des approximateurs universels de fonctions non-linéaires. [2] Un RNA avec apprentissage on-line s'adapte aux changements dans le système (*adaptabilité*). [2] Un RNA agit comme une boite noire. L'utilisateur n'a pas besoin de connaitre le fonctionnement du réseaux. Et enfin, Le bruit dans la phase d'apprentissage impacte peu les performances d'un RNA. [2]

Ces caractéristiques font d'un réseaux de neurones artificiels un candidat intéressant pour commander une Sphero. En effet, il est très difficile de concevoir, de manière analytique, un système de commande éfficace en pratique à cause des trop nombreux paramètres à gérer (Centre de gravité changeant, frottement, dérapages, équilibre, défauts,...). Un RNA, agissant comme une boite noire et étant un approximateur universel de fonction, nous permet de nous passer de la conception d'un simulateur ou d'un système de commande analytique. De plus, si l'apprentissage se fait on-line, le réseau de neurones pourra s'adapter en cas de changement de système (si le coefficient de frottement du sol change ou si la Sphero se déforme, par exemple). Enfin, du bruit sera inévitablement présent dans les données fournies par les capteurs.

1. Surface entre le solide et le sol au dessus duquel le centre de gravité doit se trouver pour que le solide garde l'équilibre.

Terminologie

Terme	Nom anglais complet
ANN	Artificial Neural Network
ELM	Extreme Learning Machine
FNN	Feedforward Neural Networks
MLP	Multilayer Perceptron
RBF	Radial Basis Function
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
SRN	Simple Recurrent Network
ESN	Echo State Network
LSM	Liquid State Machine
SNN	Spiking Neural Network
LSTM	Long Short-Term Memory
BRNN	Bi-directional Recurrent Neural Network
RMLP	Recurrent Multilayer Perceptron

TABLE 1 – Terminologie des réseaux de neurones

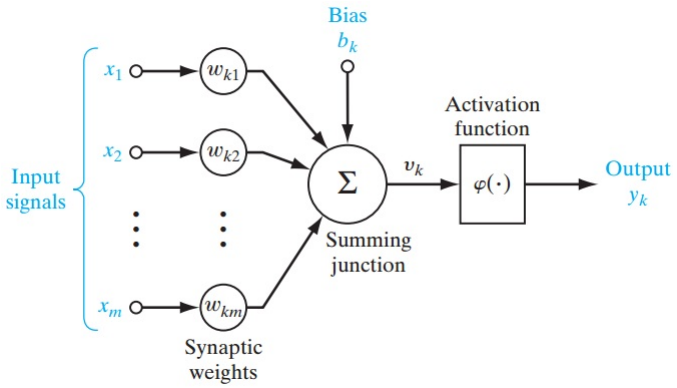


FIGURE 1 – Un neurone artificiel. **Source** : Haykin [2]

2 Théorie

2.1 Modèles de réseaux de neurones artificiels

Tout d'abord, analysons les différents modèles de réseau de neurones artificiels éxistants afin de pouvoir sélectionner celui qui convient le mieux à notre application. Les plus connus d'entre eux sont le perceptron multi-couche et la fonction à base radiale, tous deux non récurrents. Ensuite nous verrons les réseaux de neurones récurrents qui approximent des fonctions qui peuvent dépendre de toutes les entrées précédentes. Les terminologies utilisées proviennent de la littérature anglophone et tous ceux rencontrés sont répertoriés à la table 1.

2.1.1 Perceptron mutli-couches

Dans la suite, nous désignerons un *perceptron multi-couches* par sa terminologie anglaise : MLP pour « MultiLayer Perceptron ».

Neurone

La Figure 1 schématise le travail d'un neurone d'indice k. Un neurone effectue tout d'abord

Fonctions d'activations. (fonction de x)

Nom	Formule	Image
Identité	x	$]-\infty,\infty[$
Sigmoïde	$\frac{1}{1+\exp(-x)}$	$]0,1]$
Tangente hyperbolique	$\frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$	$]-1,1[$
Exponentielle	e^{-x}	$]0,\infty[$
Sinus	$\sin(x)$	$[-1,1]$
Softmax	$\frac{\exp(x_i)}{\sum \exp(x_i)}$ Ici x est un vecteur	$]0,1[$

TABLE 2 – Fonctions d'activation principalement utilisés dans un MLP. **Source** : STATISTICA Réseaux de Neurones Automatisés (SANN) [3]

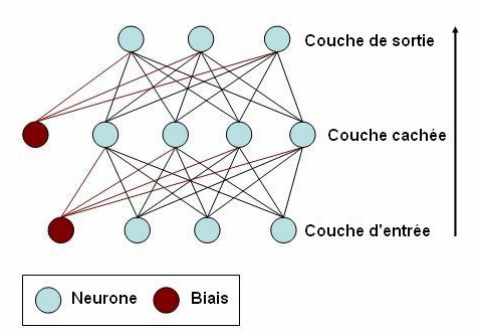


FIGURE 2 – Structure MLP à une couche cachée. **Source** : STATISTICA Réseaux de Neurones Automatisés (SANN) [3]

une somme pondérée de ses entrées

$$v_k = b_k + \sum_{j=1}^{m} x_j w_{kj}$$

où x est le vecteur d'entrée à m dimensions.
Chaque terme x_j est mutiplié par un poids w_{kj}. Ce sont les poids qui seront modifiés lors de la phase d'apprentissage. Le biais b_k est souvent ajouté à la pondération. Mais pour simplifier les formules, nous pouvons considèrer b_k comme étant l'entrée $x_0 = 1$ de poid fixe $w_{k0} = 1$. Et la somme pondérée est donc maintenant de la forme

$$v_k = \sum_{j=0}^{m} x_j w_{kj}$$

Ensuite le neurone applique la *fonction d'activation* ϕ sur la somme pondérée. Le domaine de ϕ est généralement $[0,1]$ ou $[-1,1]$. [2,3] La fonction ϕ utilisée dépend du problème qu'on veut résoudre (Table 2). Par exemple, la fonction Softmax est utilisée en classification. [3]

Structure

Un MLP est composé de plusieurs couches (Figure 2) :

couche d'entrée → couches cachées → couche de sortie.

Un neurone envoie sa sortie vers tous les neurones de la couche suivante. Il y a autant de neurones d'entrée que la dimension du vecteur d'entrée. Le $k^{\text{ième}}$ neurone d'entrée renvoie juste le $k^{\text{ième}}$ élément du vecteur d'entrée. Chaque neurone de sortie correspond à une dimension du vecteur de sortie. Les neurones cachés et neurones de sortie correspondent à la Figure 1.

Apprentissage supervisé

Il existe plusieurs algorithmes pour changer les poids sur un réseau. Mais le plus connu est l'algorithme de rétropropagation [2,4]. Il s'agit d'un algorithme d'*apprentissage supervisé*. L'apprentissage supervisé est une méthode visant à améliorer un approximateur grâce à un calcul d'erreur (appelé aussi mesure de performance [4]) comparant une sortie générée avec la sortie attendue. Nous avons donc besoin d'un ensemble de paires d'entrées/sorties. L'algorithme applique la formule développée ci-dessous à tous les neurones sauf les neurones d'entrée.

Soit ϕ_i, la fonction d'activation du neurone i. L'erreur entre la sortie produite et la sortie attendue est donnée par Q, l'*erreur quadratique* utilisée comme mesure de performance :

$$Q = \frac{1}{2}\sum_i (\phi_i - s_i)^2 \tag{1}$$

où i est l'indice d'un neurone de sortie et s_i la sortie attendue pour le neurone i.

Nous voulons modifier la valeur des poids pour que la prochaine fois que nous prenons la même entrée, l'erreur Q soit plus petite. Pour chaque poids W_{ij}, nous allons calculer le gradient $\frac{\partial Q}{\partial W_{ij}}$ qui, par définition, indique la façon dont Q varie si W_{ij} augmente d'une valeur δW_{ij} infiniment petite. Ensuite nous « ferons un pas » dans le sens opposé et proportionel au gradient en ajoutant $-\eta\frac{\partial Q}{\partial W_{ij}}$ à W_{ij} où η est une constante appelée *pas du gradient*. Donc la modification du poids W_{ij} que nous voulons appliquer vaut

$$\Delta W_{ij} = -\eta\frac{\partial Q}{\partial W_{ij}}$$

Déterminons à présent la valeur de ce gradient $\frac{\partial Q}{\partial W_{ij}}$. Nous savons que ϕ_i est une équation à une variable. Mais en entrée de cette variable est donnée la somme pondérée v_i et W_{ij} est un des poids de v_i. Par le théorème de dérivation des fonctions composées,

$$\frac{\partial Q}{\partial W_{ij}} = \frac{\partial Q}{\partial \phi_i}\frac{\partial \phi_i}{\partial v_i}\frac{\partial v_i}{\partial W_{ij}}$$

Théorème 1 (dérivation des fonctions composées)**.** *Soit* $f : A \to B : y \to f(y)$ *et* $g : B \to C : x \to g(x)$*. Alors la dérivée de* $f \circ g$ *en* x *vaut*

$$\frac{\partial f \circ g}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

Posons

$$\delta_i = \frac{\partial Q}{\partial \phi_i}\frac{\partial \phi_i}{\partial v_i}$$

qui est appelé *contribution à l'erreur* du neurone i. Nous verrons plus tard qu'elle sera utile pour la rétropropagation de la couche prédédente. Le dévellopement de la formule de rétropropagation a été fait en annexe A. Et voici ce que nous obtenons au final :

$$\Delta W_{ij} = -\eta\delta_i x_i \text{ où } \begin{cases} \delta_i &= (\phi_i - s_i)\phi'(v_i) & \text{Si } i \text{ est un neurone de sortie} \\ \delta_i &= \phi_i'(v_i)\sum_{k=1}^{n}\delta_k W_{ki} & \text{Si } i \text{ est un neurone caché} \end{cases} \tag{2}$$

Apprentissage non supervisé

Il existe également des algorithmes d'apprentissage non supervisé. Ces algorithmes ne cherchent pas à minimiser une erreur mais maximisent un score calculé à partir de la sortie du réseau.

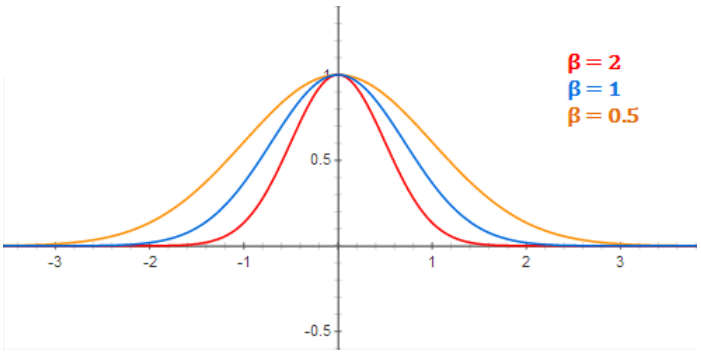


FIGURE 3 – Activation d'un neurone RBF avec différentes valeurs d'écart-type. $\beta = \frac{1}{2\sigma^2}$ **Source** : McCormick [6]

Apprentissage sur plusieurs MLP

Lorque l'une des dimensions du vecteur d'entrée est discrète et finie, il est conseillé d'utiliser un MLP différent par valeur différente sur cette dimension [4]. Cette technique peut aussi être utilisée si nous pouvons classer tous les états possibles selon le contexte. Par exemple, pour la Sphero nous pourrions utiliser un MLP pour le contexte « en train de déraper » et un autre MLP pour le contexte « ne dérape pas ».

Applications

Les MLP sont aussi utilisés en commande de robot par caméra [5]. Un MLP peut aussi être envisagé dans notre application. Nous lui fournissons une entrée composée d'informations sur l'état actuel du monde, l'état cible et ce réseau de neurones devra nous fournir en sortie la commande à appliquer pour atteindre l'état-cible.

2.1.2 Fonctions à base radiale

Nous désignerons un réseau de neurones *fonctions à base radiale* par sa terminologie anglaise : RBF pour « Rasial Basis Function ».

Neurone

Un neurone **caché** d'un RBF n'effectue pas de somme pondérée de ses entrées. Il applique directement sa fonction d'activation ϕ, une gaussienne de dimension n, de moyenne μ (appelé aussi prototype) et d'écart-type σ.

$$\phi(x) = e^{-\frac{1}{2}\sum_{k=1}^{n}\frac{(x_k-\mu_k)^2}{\sigma_k^2}} \tag{3}$$

Un neurone **de sortie** d'un RBF n'effectue pas non plus de somme pondérée de ses entrées. Sa fonction d'activation est :

$$\phi(x) = \frac{\sum_{j=1}^{m} W_j x_j}{\sum_{j=1}^{m} x_j} \tag{4}$$

où m est le nombre de dimension de x. La Figure 3 représente la sortie d'un neurone RBF où μ et x sont de dimension 1 et $\mu = 0$.
Pour résumer, un neurone RBF renvoie une valeur indiquant la similarité entre l'entrée et son prototype.

Structure

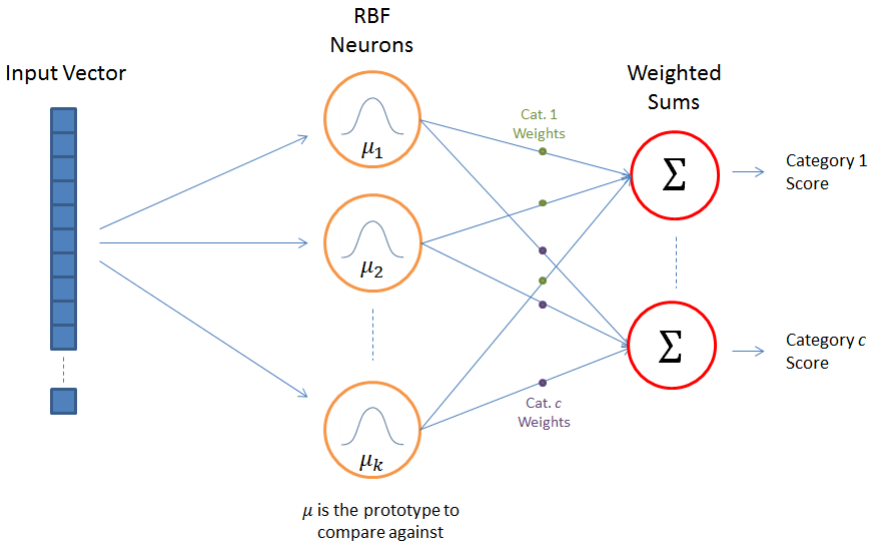


FIGURE 4 – Structure RBF. **Source** : McCormick [6]

La structure d'un RBF est comme celle du MLP sauf qu'il n'y a qu'une seule couche cachée (Figure 4).

Apprentissage

Dans ce réseau, les paramètres qui seront modifiés lors de l'apprentissage sont les poids des neurones de sortie et les moyennes et écart-types des neurones cachés. L'algorithme de rétropropagation peut être utilisé pour un apprentissage supervisé d'un RBF . Reprenons (1), l'erreur quadratique Q défini dans la section 2.1.1. En reprenant le même raisonnement que la rétropropagation dans un MLP , on va faire un pas de η dans le sens opposé et proportionnel au gradient pour chaque poids W_j des neurones de sortie mais aussi des prototypes μ_j et écart-types σ_j des neurones cachés. C'est à dire que les paramètres seront modifiés de la sorte pour le neurone i :

$$\Delta W_{ij} = -\eta \frac{\partial Q}{\partial W_{ij}}$$

$$\Delta \mu_{ij} = -\eta \frac{\partial Q}{\partial \mu_{ij}}$$

$$\Delta \sigma_{ij} = -\eta \frac{\partial Q}{\partial \sigma_{ij}}$$

Le dévellopement des formules de rétropropagation a été fait en annexe B. Et voici ce que nous obtenons au final :
Pour un neurone de sortie :

$$\Delta W_{ij} = -\eta(\phi_i - s_i)\frac{x_j}{\sum_{r=1}^{m} x_r}$$

Pour un neurone caché, posons x_r l'entrée de j provenant de i (c'est à dire $\phi_i = x_r$),
posons n la dimension de l'entrée de j,
posons aussi $R = \sum_{k=1}^{n} x_k$.

$$\Delta \mu_{ik} = -\eta \left[\sum_i (\phi_j - s_j) \frac{1}{R^2} \left(W_{jr} R - \sum_{k=1}^{n} W_{jk} x_k \right) \right] \phi_i \frac{x_k - \mu_{ik}}{\sigma_{ik}^2}$$

$$\Delta \sigma_{ik} = -\eta \left[\sum_i (\phi_j - s_j) \frac{1}{R^2} \left(W_{jr} R - \sum_{k=1}^{n} W_{jk} x_k \right) \right] \phi_i \frac{(x_k - \mu_{ik})^2}{\sigma_{ik}^3}$$

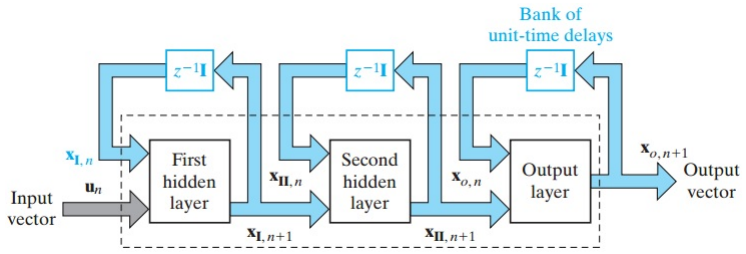


FIGURE 5 – Structure RLMP. **Source** : Haykin. p795 [2]

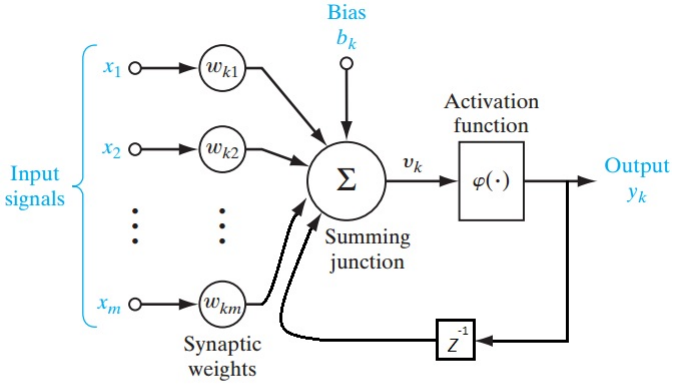


FIGURE 6 – Neurone caché d'un RMLP

Applications

Un RBF peut aussi servir pour la commande de robot [4]. Un des inconvénients des RBF est que le nombre de neurones cachés croît avec la dimension et la taille du vecteur d'entrée puisqu'un neurone caché s'active seulement pour des entrées dans le voisinage de son prototype. Nous pouvons raisonablement envisager ce modèle pour notre application puisque la description de l'état actuel et de l'état cible contiendra typiquement une coordonnée, un vecteur vitesse, une orientation et éventuellement l'accéleration, etc... La taille de l'entrée sera donc relativement petite. Ce sera lors de la phase pratique qu'on évaluera le nombre de neurones cachés nécessaires. De plus, les RBF sont moins sensibles au bruit [7].

2.1.3 Réseau de neurones récurrent (RNN)

Jusque maintenant, nous avons vu des réseaux de neurones *feedforward*. C'est à dire qu'ils ne présentent aucune boucle. Mais il éxiste des réseaux de neurones présentant des boucles. Cela leur permet d'approximer des fonctions qui dépendent non seulement de l'entrée actuelle mais aussi des entrées précédentes.

Structure

Un réseau de neurones récurrent est un réseaux présentant des boucles dans sa structure. Des *délais* notés Z^{-1} sont présents sur certains arcs dans le réseau afin de retarder d'une étape la transmission d'une valeur. Une étape dans un réseau consiste à recevoir une entrée et de générer une sortie. Il existe plusieurs réseaux de type récurrent (récurrent simple, machine à états liquide, etc). La Figure 5 présente un réseau de type perceptron multi-couches récurrent (RMLP). Il s'agit d'un MLP dont les neurones des couches cachées ont un arc qui boucle sur le neurone lui-même avec un Z^{-1} sur cet arc, comme sur la Figure 6.

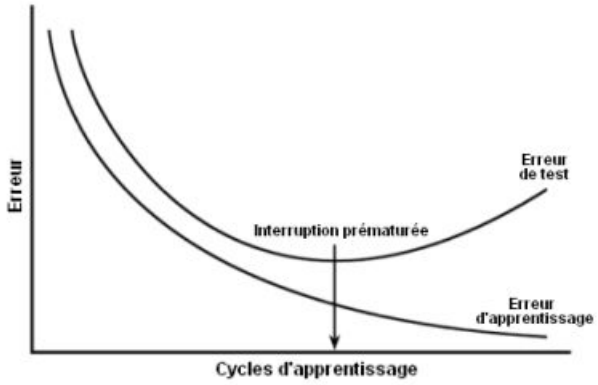


FIGURE 7 – Interruption prématurée. **Source** : STATISTICA Réseaux de Neurones Automatisés (SANN) [3]

Applications

Grâce aux délais, le réseau peut approximer des fonctions dont la sortie ne dépend pas seulement de l'entrée actuelle mais aussi des entrées précédentes. Mais cette fonctionnalité n'est pas utile pour notre application puisque les commandes à générer ne dépendent pas des états précédents l'état actuel.

2.2 Le problème du surajustement

2.2.1 Le phénomène

Lorsqu !un réseau de neurones à apprentissage supervisé apprend de trop nombreuses fois à partir du même set d'exemple, alors il peut se produire un phénomène appelé *surajustement* [3]. Le réseau devient dès lors de moins en moins efficace sur des entrées non rencontrées pendant les phases d'apprentissage. La Figure 7 montre qu'au fil des cycles d'apprentissage, l'erreur diminue. Mais à un certain moment, l'erreur sur des paires entrée/sortie non fournies pendant la phase d'apprentissage augmente. C'est à partir de ce moment là que le réseau de neurones surajuste.

2.2.2 Les solutions

Intérruption prématurée

Tout d'abord séparons notre ensemble de paires entrée/sortie en deux ensembles. L'un est nommé *ensemble d'apprentissage* et l'autre *ensemble de test*. Chacun de ces ensembles doit balayer toutes les valeurs d'entrées possibles. Ensuite à chaque cycle, on va d'abord effectuer un apprentissage pour chaque paires de l'ensemble d'apprentissage. Puis on génère des sorties à partir des entrées de l'ensemble de test. On compare l'erreur de test à celui de l'étape précédente. L'erreur de test est l'erreur calculée sur les sorties générées et les sorties attendues de l'ensemble de test. Si l'erreur de test est plus grand que celui du cycle précédent, alors on arrête les cycles d'apprentissage. Dans la Figure 7, on observe que l'interruption prématurée se produit au moment où son *pouvoir de généralisation*, c'est à dire son efficacité pour des entrées jamais rencontrées, est le plus élevé.

Dans la réalité, la courbe d'erreur de test n'est pas aussi lisse que dans la Figure 7 mais présente un certain bruit. Il faut donc pouvoir s'assurer que l'interruption prématurée a bien lieu

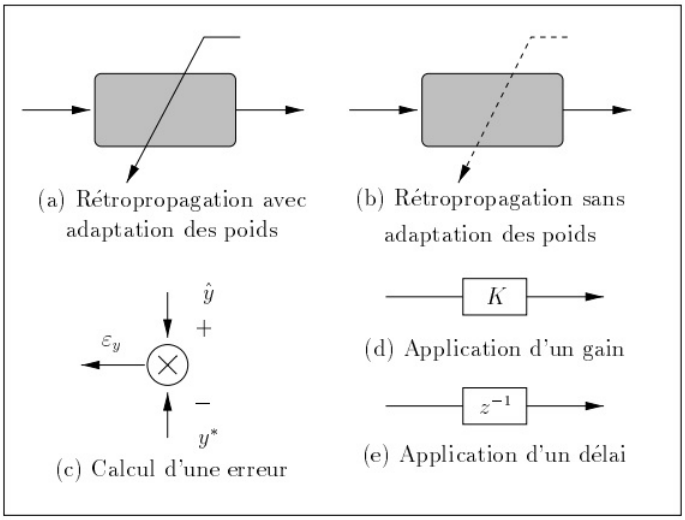


FIGURE 8 – Conventions graphiques pour la description de l'architecture d'une méthode utilisant des RNA. **Source** : Gauthier [4]

sur le minimum global de l'erreur de test. Comparer l'erreur de test à celui du cycle précédent permet de détecter seulement un minimum local.

Modération des poids

Cette méthode consiste à pénaliser l'utilisation de trop grand poids. Pour ce faire, on modifie le calcul d'erreur pour y prendre en compte leur valeur [3]. Soit E l'erreur, soit W un vecteur contenant tous les poids (ne pas prendre les biais en compte), la nouvelle valeur de l'erreur est

$$E_{new} = E + \frac{\sigma}{2} W^T W$$

où σ est une constante appelée *constante de modération*. Un σ trop petit ne permet pas d'éviter le surajustement et un σ trop grand empêche la généralisation.

2.3 Méthodes d'apprentissage avec maître distant

Nous allons observer différentes architectures intéressantes dans notre cas de génération de commandes. Le mot « architecture » utilisé ici ne désigne pas l'architecture d'un réseau de neurones mais l'architecture d'une méthode d'utilisation de réseaux de neurones. La convention graphique est montrée dans la Figure 8.

2.3.1 Le problème

Dans cette section, nous allons voir différentes méthodes pour entrainer un réseau de neurones à générer des commandes. Ces méthodes visent à contourner le *problème de l'apprentissage avec maître distant*. Dans l'apprentissage supervisé, un *maître* est ce qui permet de fournir une erreur entre la sortie d'un réseau de neurones et la sortie attendue. Et effectivement dans notre cas, nous n'avons rien qui puisse nous permettre de définir les commandes idéales à éxécuter pour passer d'un certain état à l'autre. Par contre nous pouvons avoir un maître qui compare l'état résultant de l'exécution d'une commande à l'état cible.

Dans les figures qui suivront, *Système* désigne l'exécution, dans la réalité, de la commande entrée. En sortie du Système nous avons l'état résultat de l'exécution des commandes.

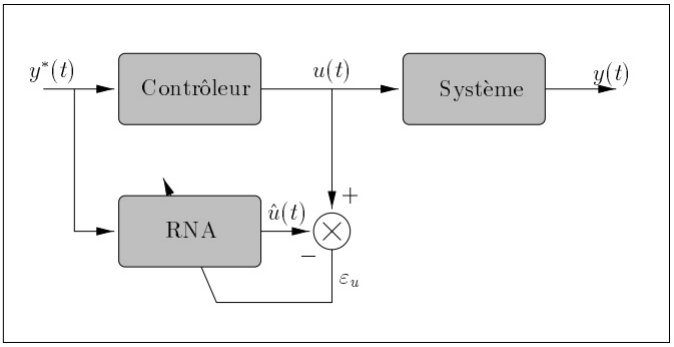


FIGURE 9 – Apprentissage par reproduction d'un contrôleur. **Source** : Gauthier [4]

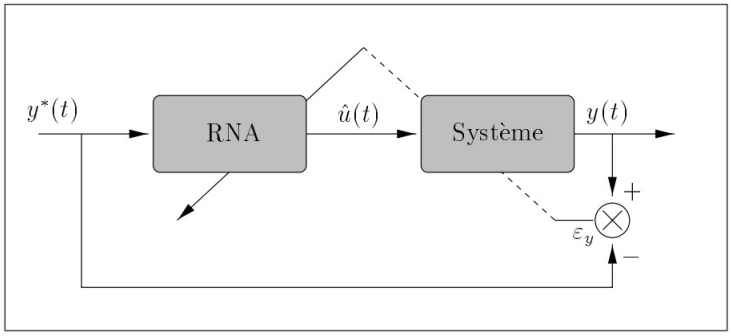


FIGURE 10 – Apprentissage spécialisé. **Source** : Gauthier [4]

2.3.2 Reproduction d'un contrôleur

Dans cette méthode d'apprentissage, le RNA apprend à reproduire les commandes d'un contrôleur existant (**Figure 9**). Cette méthode a été utilisée pour commander le volant d'une voiture pour suivre des virages où le controlleur est un humain [5].

2.3.3 Apprentissage spécialisé

Ici on applique l'algorithme de rétropropagation en démarrant depuis le maître distant et en passant par le système. Par exemple, nous avons vu que, dans le cas d'un MLP , section 2.1.1, l'apprentissage est l'application de la formule

$$\Delta W_{ij} = -\eta \frac{\partial Q}{\partial W_{ij}} = \frac{\partial Q}{\partial \phi_i} \frac{\partial \phi_i}{\partial v_i} \frac{\partial v_i}{\partial W_{ij}}$$

En reprenant la notation de la Figure 10,

$$\Delta W_{ij} = -\eta \frac{\partial \varepsilon_y}{\partial W_{ij}} = \frac{\partial \varepsilon_y}{\partial \hat{u}} \frac{\partial \hat{u}}{\partial v_i} \frac{\partial v_i}{\partial W_{ij}}$$

Où v_i est la somme pondérée des entrées du neurones i. Or $\frac{\partial \varepsilon_y}{\partial \hat{u}} = \frac{\partial \varepsilon_y}{\partial y} \frac{\partial y}{\partial \hat{u}}$. Et donc la formule devient

$$\Delta W_{ij} = \frac{\partial \varepsilon_y}{\partial y} \frac{\partial y}{\partial \hat{u}} \frac{\partial \hat{u}}{\partial v_i} \frac{\partial v_i}{\partial W_{ij}}$$

Et définir le facteur $\frac{\partial y}{\partial \hat{u}}$ est la difficulté de cette méthode : Il faut pouvoir modèliser le système par une fonction $y(\hat{u})$ dérivable en $\hat{u}$.

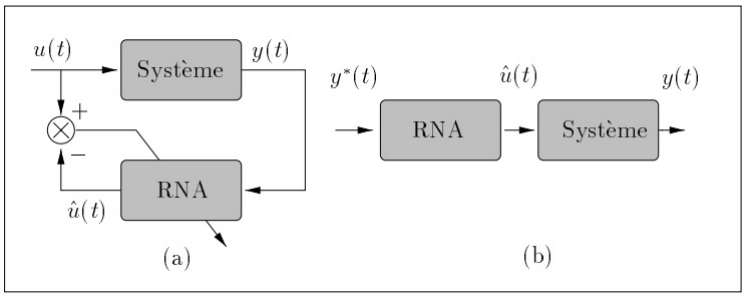


FIGURE 11 – Apprentissage en deux phases. **Source** : Gauthier [4]

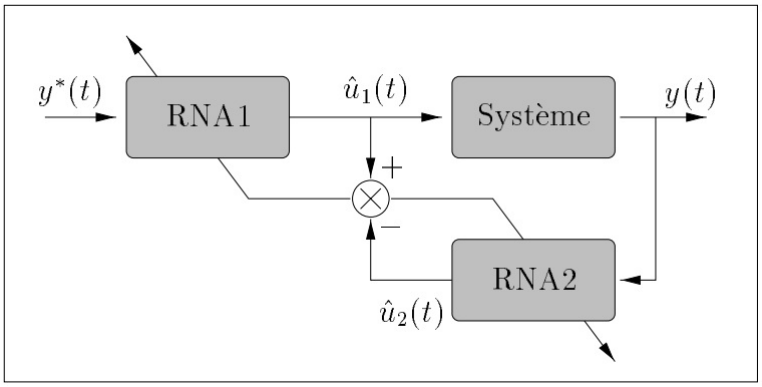


FIGURE 12 – Apprentissage indirect. **Source** : Gauthier [4]

2.3.4 Apprentissage en deux phases

Tout d'abord, (a) On génère des commandes u en boucle et on les applique au système (**Figure 11**). Soit y l'état du robot après avoir appliqué la commande u, on obtient en boucle des paires entrée/sortie où y est l'entrée et u la cible. On continue la boucle pour balayer toutes les configurations possibles de u.

Puis vient la phase d'utilisation (b) où on utilise le réseau cette fois pour génèrer les commandes. Mais sans apprentissage. L'adaptabilité est donc sacrifiée.

Cette méthode est inadaptée dans les cas où il existe plusieurs commandes idéales pour un même état cible. En effet, imaginons qu'il existe deux commandes idéales pour atteindre le même état cible et que ces deux commandes soient rencontrées lors de la phase (a). Alors les poids seront modifiés pour que la sortie converge vers la première commande idéale rencontrée, puis l'autre et ainsi de suite. Et finalement, la sortie du réseau pour cet état cible ne va pas converger vers une des commandes idéales.

2.3.5 Apprentissage indirecte

L'apprentissage indirecte, montré dans la **Figure 12**, est une méthode qui permet de retrouver l'adaptabilité (sacrifié dans l'apprentissage en deux phases, section 11). RNA1 et RNA2 ne sont pas deux réseaux différents. Il s'agit du même réseau de neurones. RNA1 et RNA2 partagent les mêmes poids. Mais selon Gauthier [4], cette méthode tend à toujours donner la même commande.

2.3.6 Apprentissage par modèle différentiable

L'idée est d'utiliser la méthode de l'apprentissage spécialisé à la section 2.3.3 mais en se servant d'un RNA comme modèle différentiable (**Figure 12**). Il faut donc, dans un premier

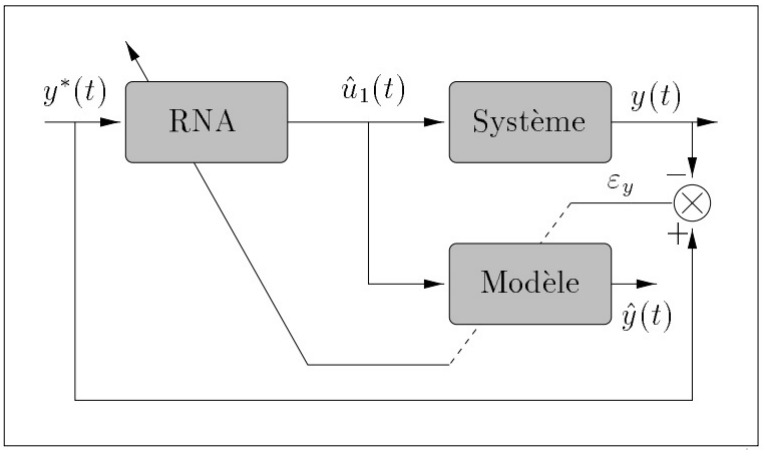
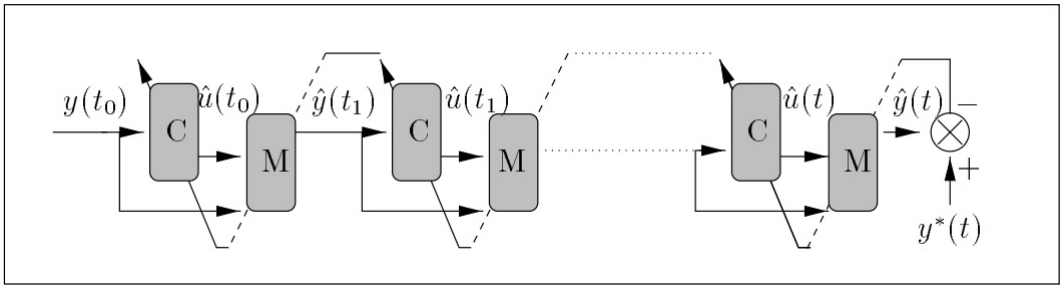


FIGURE 13 – Apprentissage par modèle différentiable. **Source** : Gauthier [4]



— **C** : Réseau qui génère les commandes
— **M** : Modèle

FIGURE 14 – Architecture de Nguyen et Widrow. **Source** : Gauthier [4]

temps, établir un RNA qui va apprendre à prédire le prochain état selon la commande à exécuter. Nous n'avons pas le problème de l'apprentissage avec maître distant pour ce RNA car sa sortie peut directement être comparée à la sortie du système. Ensuite, on l'utilise comme modèle. Et là nous pouvons connaître $\frac{\partial \varepsilon_y}{\partial \hat{u}_1}$ grâce aux formule de rétropropagation de la section 2.1.1.

L'adaptabilité est sacrifiée car si le système change et que le modèle n'a pas été entrainé pour ce nouveau système, l'algorithme de rétropropagation va faire converger la solution vers des commandes inadaptées pour le nouveau système.

2.3.7 Apprentissage sur plusieurs étapes

Un autre problème qui peut se présenter pour la commande par réseau de neurones est d'obtenir un maître distant seulement après plusieurs étapes. Par exemple, imaginons que nous voulons commander la marche d'un robot avec un réseau de neurones qui fournira des commandes toutes les T secondes. Si le robot tombe et que nous pouvons en obtenir un maître, nous ne savons pas à quelles étapes avant la chute le réseau a fourni des commandes responsables de cette chute. Une solution à ce problème est l'architecture de Nguyen et Widrow, Figure 14. Cela consiste à utiliser un modèle différentiable (qui peut être un réseau de neurone) pour pouvoir effectuer l'algorithme de rétropropagation jusque la première étape après l'obtention du maître précédent.

3 Implémentation et application

3.1 Design du vecteur d'entrée

3.1.1 Données utiles

Avant de choisir le réseau de neurones à utiliser pour générer des commandes pour la Sphero, il est important d'identifier ce que nous souhaitons obtenir en sortie et les informations que nous pouvons fournir en entrée en omettant les données inutiles. Si le vecteur d'entrée est de dimension très grande, alors un réseau RBF devra avoir un trop grand nombre de neurones cachés [4]. Si nous supposons que la sortie produite peut dépendre non seulement de l'input actuel mais aussi des entrées précédentes, alors un réseau récurrent devra être utilisé.

La Sphero peut nous fournir [8] :

— Sa position grâce à un odomètre (cm),
— Son vecteur vitesse (mm/s),
— Son vecteur d'accélération grâce à un accéleromètre (mG),
— Son orientation (-179→180 degrés),
— Son niveau de charge (Pas de poucentage mais label « Chargement », « OK », « Bas » et « Critique »),
— État des LEDs, évènement de collision, voltage de la batterie, nombre de charges, version ...

Parmis ces données, nous n'avons pas besoin de l'état des LEDs, version et nombre de charges car ils n'influencent pas la conduite de la Shero. Dans ce projet, nous ne prenons pas en compte les éventuelles collisions.

Le niveau de charge pourrait influencer la conduite. Nous ne savons pas si la puissance maximale des moteurs diminuent lorsque la charge est trop faible. Si cela s'avère être le cas, pour ne pas devoir effectuer un entrainement pour chaque niveau de charge différent, on entrainera la Sphero que pour les cas où le niveau de charge est à « OK » et on supposera donc que la conduite sera effectuée qu'avec ce niveau de charge.

Un odomètre n'est pas approprié pour la mesure de position dans notre application. En effet, la distance parcourue n'est mesurée qu'à partir du roulement du moteur permettant le déplacement. La distance parcourue par dérapage ne sera pas comptabilisée. Cela a été vérifié lors de l'expérience menée dans la section 3.1.2. Une camera est plus appropriée pour tracer la position réelle de la Sphero.

Pour l'accélération, cette donnée peut être utile pour anticiper la vitesse de la Sphero au moment où elle reçoit les commandes car, dû à la latence d'envoi de paquet, la vitesse à l'instant t n'est plus la même que celle communiquée. Si cette donnée impacte peu dans les résultats obtenus, alors nous pourrions décider de nous passer de ce paramètre. Ajouter cette dimension dans le vecteur d'entrée demandera un plus grand nombre d'exemples pour la phases d'apprentissage car il faut balayer toutes les configurations possibles de vecteur d'entrée.

3.1.2 Choix du design

On aligne l'axe des abscisses avec l'axe de l'orientation du moteur afin de reduire le domaine d'entrée et donc le nombre d'exemples à créer pour l'apprentissage. Le vecteur d'entrée contiendra donc la vitesse actuelle, l'accélération actuelle et la position relative à atteindre dans T secondes où T sera la période à laquelle on échantillonera les données des capteurs. Nous avons donc un vecteur d'entrée à 6 dimensions (vitesse actuelle en x et y, position relative cible en x et y, acccélération actuelle en x et y). Nous pourrions réduire d'avantage le domaine d'entrée si nous considérons que la direction et la vitesse à prendre pour atteindre une position est en symétrie orthogonale d'axe x.

Il y a deux façons différentes de commander la Sphero. [8] Donc deux possibilités s'offrent à nous pour le vecteur de sortie :

Vecteur	**Avantages**	**Inconvénients**
{puissance du moteur A puissance du moteur B}	• Plus précis • Permet plus de dérapage • Grande vitesse dans les virages	• Stabilisateur désactivé • Risque de commande « inutiles » • Risque une déstabilisation totale
{vitesse, orientation}	• Stabilisateur activé • Toutes les commandes ont un effet	• Moins précis • Évite les dérapages

Par commande « inutiles », nous entendons par là des commandes qui peuvent sembler absurdes. Par exemple, si la puissance donnée au moteur qui soulève le contre-poids est très élevée alors que la Sphero a une faible vitesse, le contre-poids tournera et se retrouvera dans le sens opposé à celui qui permettait de faire avancer la Sphero. Et la déstabilisation totale est une perte du repère qui sert à capturer l'orientation actuelle de la Sphero. En effet, l'orientation de la Sphero est donnée par rapport à un repère, 0 degré, qui est sensé resté constant. Si le stabilisateur est désactivé, la coque elle-même risque de changer d'orientation. Modifiant donc le repère. Ce qui biaisera toutes les orientations capturées par la suite. C'est pour cela que nous choisirons comme vecteur de sortie la vitesse et l'orientation.

Estimons la valeur optimale que nous pouvons donner à T. Une commande de streaming de données peut être envoyée à la Sphero. Grâce à cette commande, nous pouvons demander à la Sphero d'envoyer uniquement les données qui nous interessent à une fréquence donnée. Trois paramètres de cette commande peuvent influencer les performances d'échantillonage :

1. Un diviseur (entier) de la fréquence maximale d'échantillonage.
2. Le nombre d'échantillonages à garder en mémoire avant envoi.
3. Le nombre de capteurs à échantilloner.

Un échantillonage consiste à obtenir seulement les données qui nous intéressent à un instant précis. La fréquence maximale d'échantillonage, un échantillon à la fois, est de 400Hz. [8] La vitesse maximale de la Sphero est de 4,5 miles par heure. [9] Ce qui fait environ 2 mètres par seconde.

À la Figure 15 sont repris les histogrammes de temps de réceptions de 500 paquets pour différentes fréquences d'échantillonage. Les paquets invalides, donc de checksum incorrect ou de taille trop courte, sont rejetés. Ce qui explique les pics sur $2T$. En plus du streaming, pour chaque paquet reçu, un paquet est envoyé à la Sphero pour changer les leds. Cela nous permet d'obtenir des résultats qui se rapprocheront de ce que nous obtiendrons si nous envoyons une nouvelle commande à chaque période T. Les fréquences de streaming choisis sont des diviseurs de 400 car le démarage d'un streaming demande en paramètre un diviseur de 400. [8] La Sphero divise ensuite 400 par ce paramètre pour obtenir la fréquence demandée. La configuration de l'ordinateur utilisé est la suivante :

— AMD Athlon(tm) X2 Dual-Core QL-64
— Carte mère EI Capitan
— 3 Go de RAM DDR2
— disque dur ST9250827AS 5400rpm
— Adapteur USB bluetooth BT009x, transfert 1Mbs.
— OS : Linux 4.4.0-57-generic, Ubuntu 16.04.4

Pour les critères de choix d'une valeur pour T, deux critères sont pris en compte :

1. La stabilité du streaming. Moins la densité de probabilité de temps de réception de paquet est concentrée sur le voisinage de T, moins la solution sera précise.

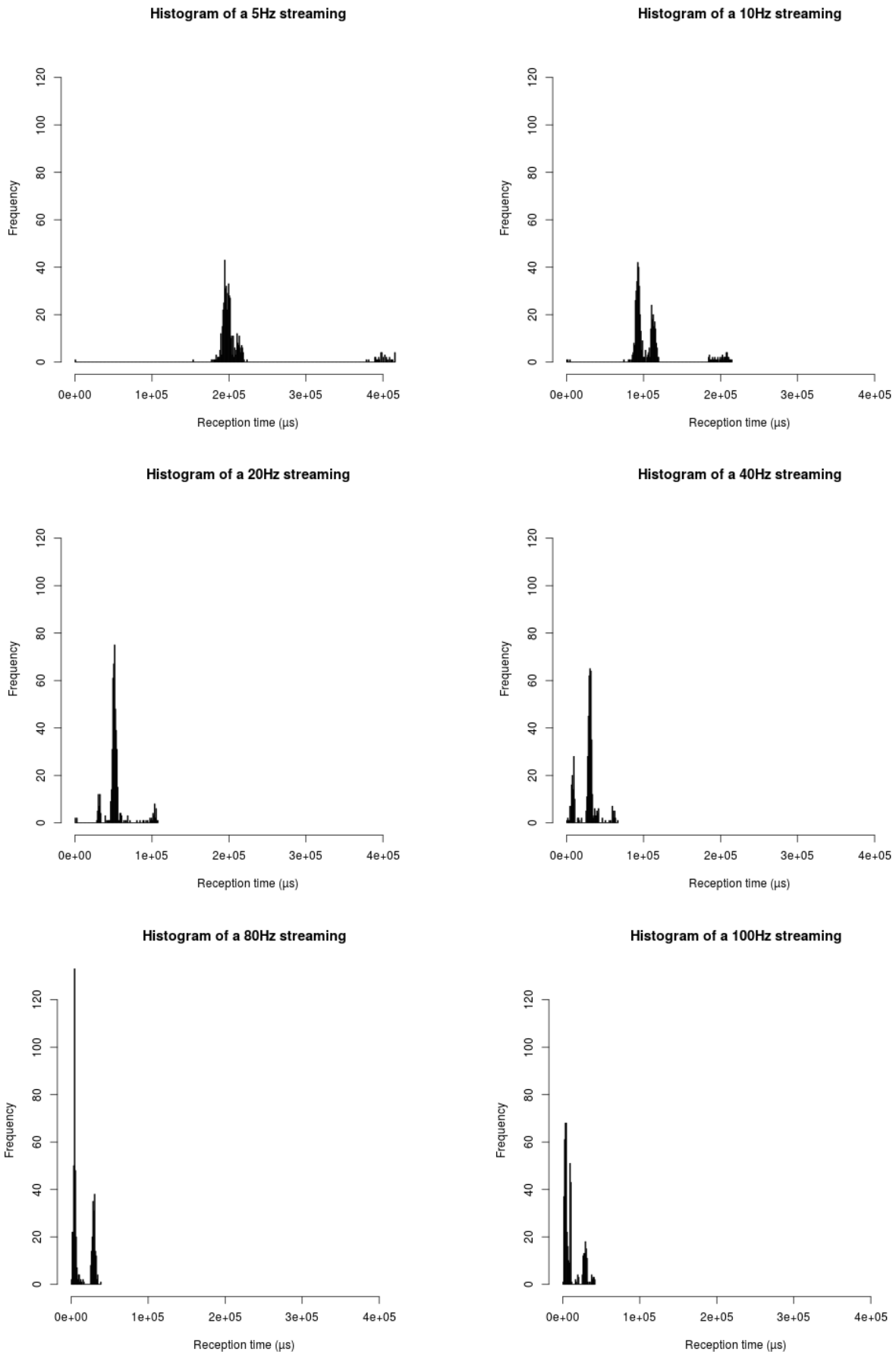


FIGURE 15 – Histogrammes de temps de réception de packet streamé. Générés par script R

2. Le temps de réaction. Plus T est grand, moins la trajectoire de la Sphero sera fidèle à la trajectoire voulue. Nous pouvons compenser ce manque de précision par un vecteur d'entrée plus complexe mais la solution converge alors moins vite et un ensemble d'apprentissage plus conséquent doit être fourni.

Une autre façon d'observer la stabilité d'un streaming est d'enregistrer la date de réception de chaque paquet, de leur soustraire la date esperée de réception et puis de reporter toutes ces valeurs dans un graphique. Toutes les parties du graphique représentant une droite horizontale représentent un moment pendant laquelle la période de réception de paquets valides est stable. Si on observe un bond vertical vers le haut de T secondes, alors un paquet a été rejeté à ce moment là. Ces graphiques sont reportés dans la Figure 16.

On observe que la fréquence de rejet de paquet est anormalement régulière. On observe presque toujours 10 paquets valides entre paquet rejeté. Tous ces paquets ont été rejeté car leur taille était insuffisante. Pour empêcher que ce manque de précision nuit à la convergence du réseau de neurones, nous pourrions, dans l'ensemble d'apprentissage, retirer les instances dont le temps mis pour atteindre l'état suivant est de environ $2T$. En rejetant donc toutes ces données, nous pouvons donc ne plus prendre en compte le pic sur $2T$ dans les histogrammes de la Figure 15. On observe dès lors que le plus stable celui à 80Hz car le pic en T est plus haut que celui des autres et avec une largeur presque pareille.

3.2 Choix du réseau

Nous n'avons pas besoin d'un réseau de neurones récurrent. Les commandes à appliquer ne dépendent pas de la vitesse, position et autres données fournies aux étapes précédentes. Le réseau le mieux adapté pour notre problème est un réseau de neurones à base radial. En effet, les RBF sont moins sensibles au bruit que les MLP [4, 7].Car dans la couche cachée, chaque neurone envoit une sortie indiquant la proximité entre le vecteur d'entrée et son prototype. Donc typiquement, la contribution à l'erreur est presque entièrement celle du neurone ayant son prototype le plus proche de l'entrée. Les autres neurones cachés auront une modification insignifiante de l'écart-type et du prototype.

Un des inconvénients des RBF est que le nombre de neurones cachés croît avec la dimension et la taille du vecteur d'entrée. Ceci ne pose pas de problème dans notre cas car le nombre de dimensions est faible.

3.3 Choix de la méthode

Vu toutes les inconnues mécaniques du mouvement de la Sphero, il s'avère difficile d'établir une équation permettant de prédire de manière précise l'état suivant en fonction des commandes exécutées. Nous préfèrons donc nous passer d'un modèle analytique. Dans ce cas, nous ne pouvons pas utiliser la méthode de l'apprentissage spécialisé. De plus, télécommander une Sphero pour suivre une trajectoire est très difficile pour un humain. Utiliser un apprentissage par reproduction de contrôleur n'est donc pas envisageable.
Nous pouvons obtenir un maître distant à chaque étape et donc nous n'avons pas besoin de l'architecture de Nguyen et Widrow.
Le problème de convergence vers une solution triviale, impliqué dans le cas d'un apprentissage indirect (section 2.3.5) exclu l'utilisation de cette méthode dans le cadre de ce projet.

Il nous reste deux méthodes pour lesquelles l'adaptabilité est sacrifiée : l'apprentissage en deux phases et apprentissage utilisant un modèle différentiable. Les deux sont envisageables.

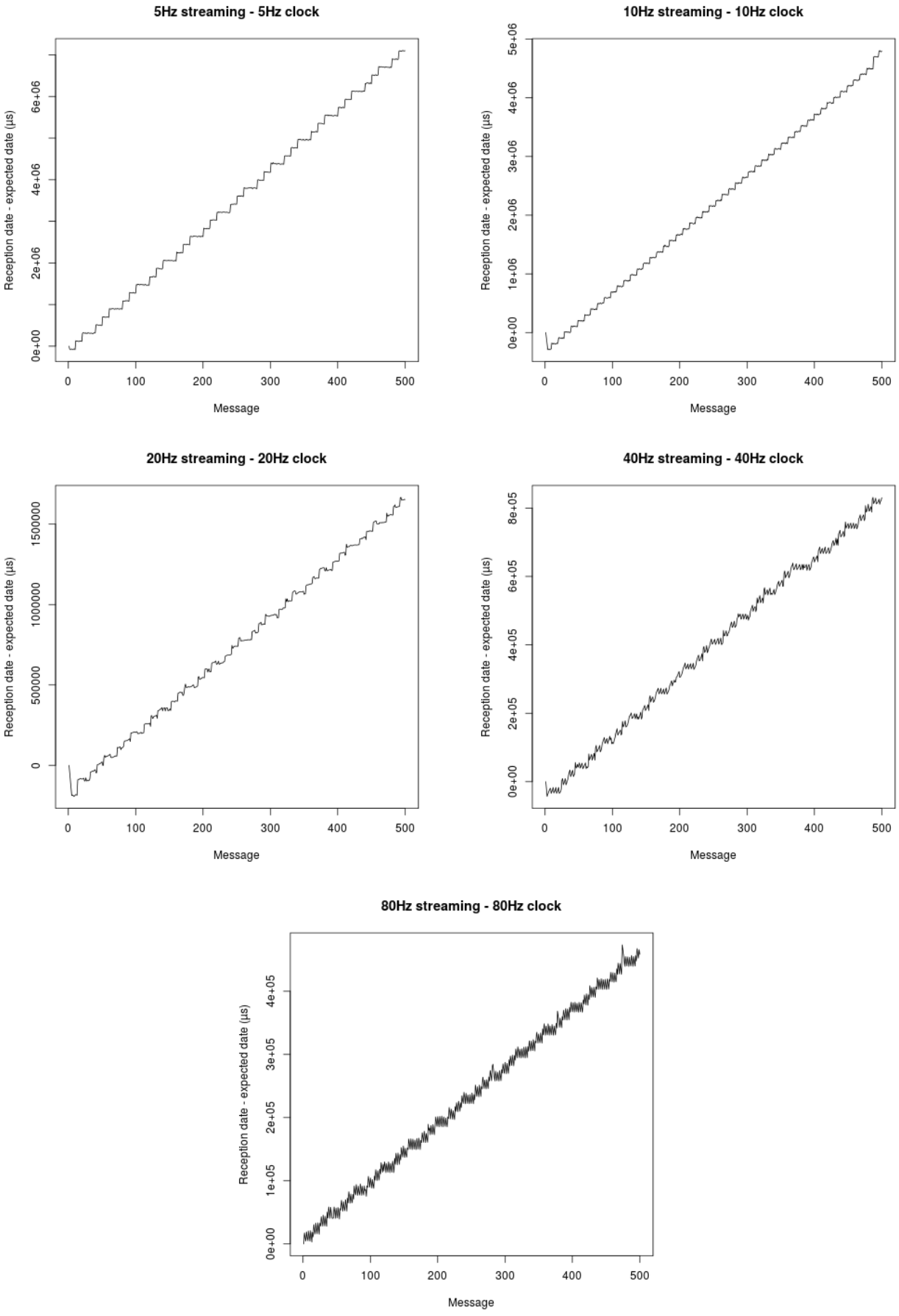


FIGURE 16 – Dates de réception de paquet - dates esperées. Générés par script R

Les tests seront effectués toujours sur le même sol horizontal.

La méthode avec modèle différentiable apportera un nouveau facteur influençant la réussite du projet : la fiabilité du modèle. Mais puisque le sytème ne change pas, lorsqu'on arrive à obtenir un modèle fiable, nous pouvons effectuer un apprentissage en-ligne qui permettra à la Sphero de parfaire le suivi d'une trajectoire si cette trajectoire est ammenée à être répétée. Nous utiliserons dans un premier temps la méthode en deux phases pour obtenir un ensemble de paires cible/commande où la précision dépend seulement des capteurs et pas de la fiabilité d'un modèle.

3.4 Les API

Certaines API permettant l'envoi de commande et réception de message provenant de la Sphero sont disponibles. Les APIs officiels sont disponibles en : **Objective-C**, **Swift**, **Android**. [8]
Il existe également des APIs créés par la communauté disponibles en [10] : **C#**, **JavaScript**, **Ruby**, **Python** [11], **Arduino**, **C++** [12].
L'API choisie est l'API **C++** pour le langage de programmation connu et performant exécutable sur PC. Malgré les apparences, cet API n'est pas modulaire concernant le streaming. En effet, le streaming est démarré dès la connection de la Sphero et la lecture des messages de streaming récupérés ne fonctionne que pour le streaming configuré pour l'application que l'auteur voulait en faire. C'est à dire que des accès à la mémoire se font sans considération des données demandées pour le streaming et ne permet donc que de récupérer la position et la vitesse. De plus l'écoute des messages de streaming sont non bloquants car ils sont effectués dans un thread à part. Il n'est donc pas possible, sans modifier l'API, d'utiliser les données reçues dès leur réception. L'API a donc été modifiée pour récupérer les données qui nous intéressent dans le cadre de ce projet mais aussi pour effectuer le design pattern observer et le faire hériter d'une interface d'objet capable de streamer. (Voir la section 3.6 sur l'implémentation du système de commande.)

Le langage **Qt** sera utilisé pour créer les interfaces graphiques. Cela nous permet de créer facilement des interfaces munies de boutons, menus et fenêtres afin de tracer la trajectoire. De plus, l'implémentation du réseau de neurones pourra être directement utilisée dans un script **Qt** puisqu'il s'agit d'une extension du c++.

L'API **Caffe** sera utilisé plus tard afin de modifier facilement l'architecture du réseau de neurones dans le module de commande sans recompilation. [13]

3.5 Implémentation d'un réseau de neurones

Un RBF a été implémenté. Bien que fonctionnel, ce n'est pas ce réseau qui est utilisé dans la version actuelle du système de commande de ce projet. Il a néanmoins été utilisé durant la phase d'expérimentation.

3.5.1 Propriété utilisée

Afin d'expliquer la structure de l'implémentation du réseau de neurone, nous allons tout d'abord expliquer pourquoi l'implémentation d'un neurone est indépendant de sa position et du modèle du réseau de neurones. Et grâce à cette propriété, nous pouvons donc connecter n'importe quelle couche de neurones avec une autre sans modifier l'implémentation. Nous pourrions donc facilement passer à une méthode d'apprentissage par modèle différentiable (Section 2.3.6) si cela s'avère nécessaire. De plus, l'implémentation d'un nouveau modèle non récursif comme le MLP nécésitera juste l'implémentation des nouveaux neurones. En résumé, l'implémentation

d'un réseau RBF se fera à partir d'une implémentation de réseau de neurones non récursif modulaire.

Deux fonctions sont nécessaires pour l'implémentation d'un neurone :

1. Une fonction qui permet de générer la sortie à partir de ses entrées (compute).
2. Une fonction permettant l'apprentissage par rétropropagation (backpropagation).

La fonction qui génère la sortie n'a besoin que des sorties des neurones de la couche précédente. Et si on observe le développement des formules de rétropropagation dans les annexes A et B, la fonction qui permet l'apprentissage n'a besoin que de la sortie générée précédemment (cela peut être gardé en mémoire) et la *contribution à l'erreur* des neurones de la couche suivante. Le terme de contribution à l'erreur fait réference au terme δ dans la formule de modification des poids d'un MLP (Section 2.1.1). En effet, pour tous les différents neurones que nous avons vu dans ce rapport, soit p un paramètre à modifier pour l'apprentissage du neurone i. Soit f_i la fonction appliquée aux entrées du neurone i afin de générer la sortie. p doit être modifié selon la formule suivante :

$$\Delta p_i = -\eta \frac{\partial Q}{\partial p_i}$$

Et afin de calculer la valeur de $\frac{\partial Q}{\partial p_i}$, deux facteurs sont à connaître : $\frac{\partial Q}{\partial f_i}$ et $\frac{\partial f_i}{\partial p_i}$. Pour connaître le facteur $\frac{\partial f_i}{\partial p_i}$, toutes les données dont nous avons besoin se trouvent dans l'implémentation du neurone i si nous y enregistrons sa sortie précédente. Tandis que pour le facteur $\frac{\partial Q}{\partial f_i}$, si le neurone i est un neurone de sortie, alors $\frac{\partial Q}{\partial f_i}$ est connu, il s'agit de la dérivée partielle de l'erreur entre f_i et sa valeur atendue. Sinon

$$\frac{\partial Q}{\partial f_i} = \sum_{k \in \text{couche suivate}} \frac{\partial Q}{\partial f_k} \frac{\partial f_k}{\partial f_i}$$

Et donc $\frac{\partial Q}{\partial f_i}$ peut être calculé à partir d'une implémentation dans les neurones de la couche suivante.

3.5.2 Diagramme de classe

Dans le diagramme de classe, Figure 17, dans la déclaration de la fonction `backpropagation()`, `errorContribution` est le facteur $\frac{\partial Q}{\partial f_i}$. Le tableau retourné est stocké dans `thisContribution`. Pour chaque entrée x_j du neurone i, la $j^{\text{ième}}$ valeur du tableau retourné est $\frac{\partial Q}{\partial f_i} \frac{\partial f_i}{\partial x_j}$. Ces valeurs sont utilisées pour effectuer la rétropropagation de la couche précédente.

3.5.3 Limitation actuelle

Dans la version actuelle de l'implémentation, les biais ne sont pas supportés, le MLP n'est pas encore fonctionnel et des fuites mémoires ont été détectés. Les fuites mémoires sont dû à l'allocation de mémoire pour transmettre des valeurs d'une couche à l'autre dans `Net.cpp`. L'opérateur `New` effectue une allocation dynamique et nécessite d'être libérée exlplicitement par le programmeur.

3.6 Implémentation d'un système de commande

Passons maintenant à la description des différentes classes et interfaces du système de commande.

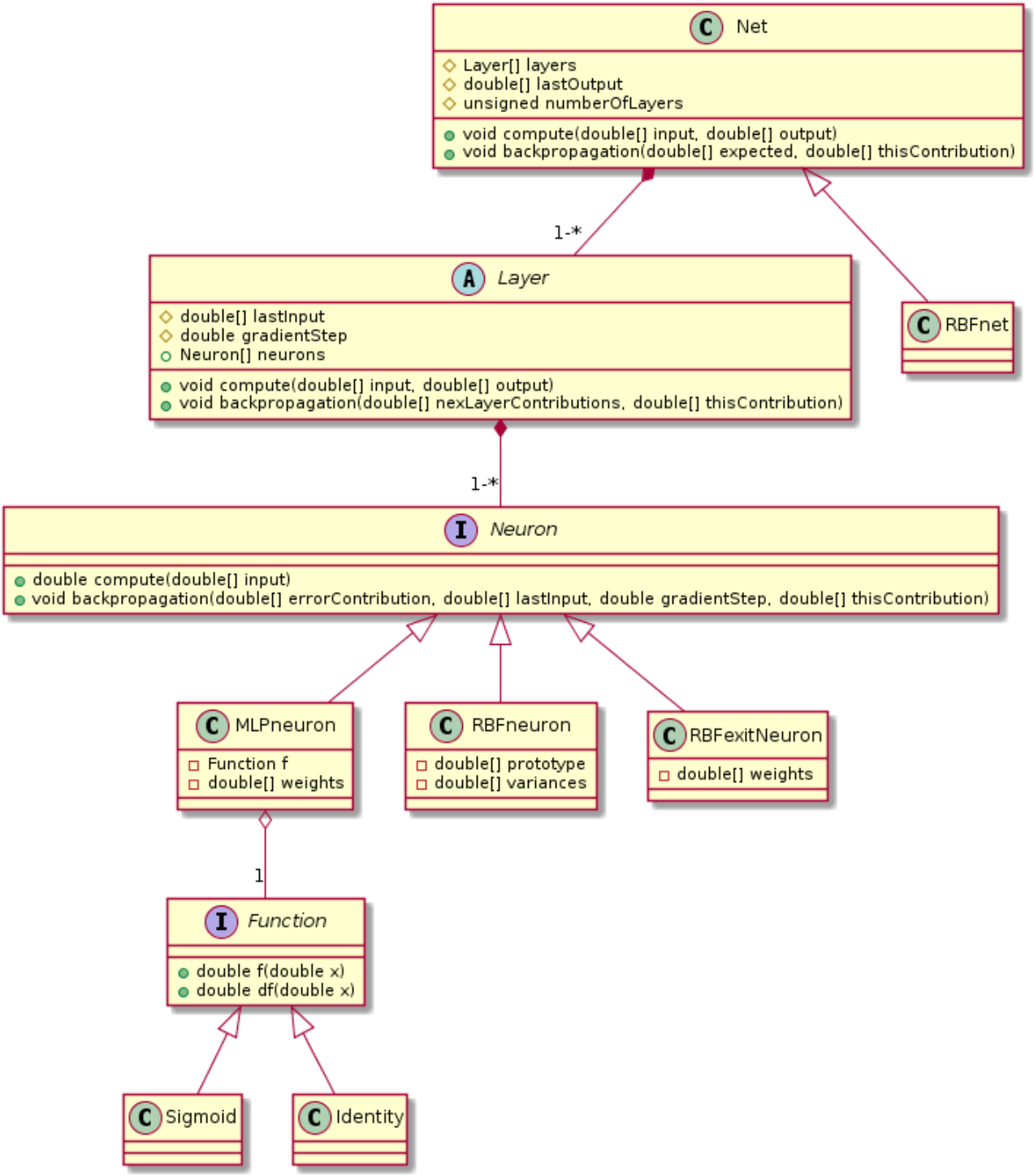


FIGURE 17 – Diagramme de classe du réseaux de neurones. Généré par plantUML

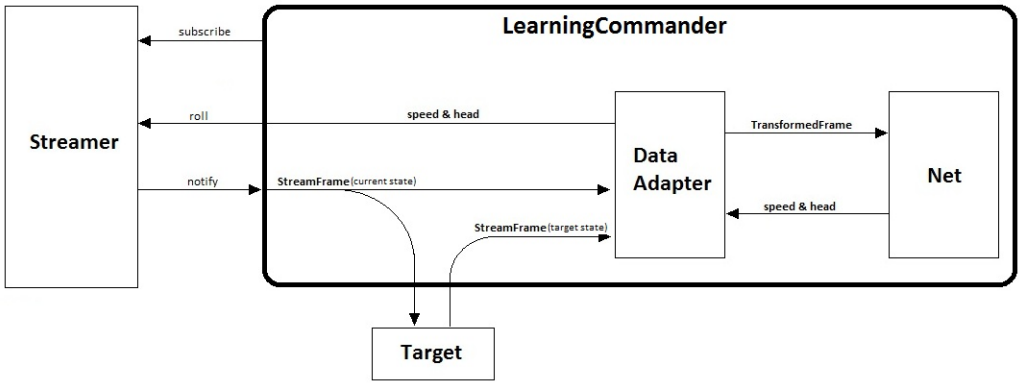


FIGURE 18 – Structure du système de commande

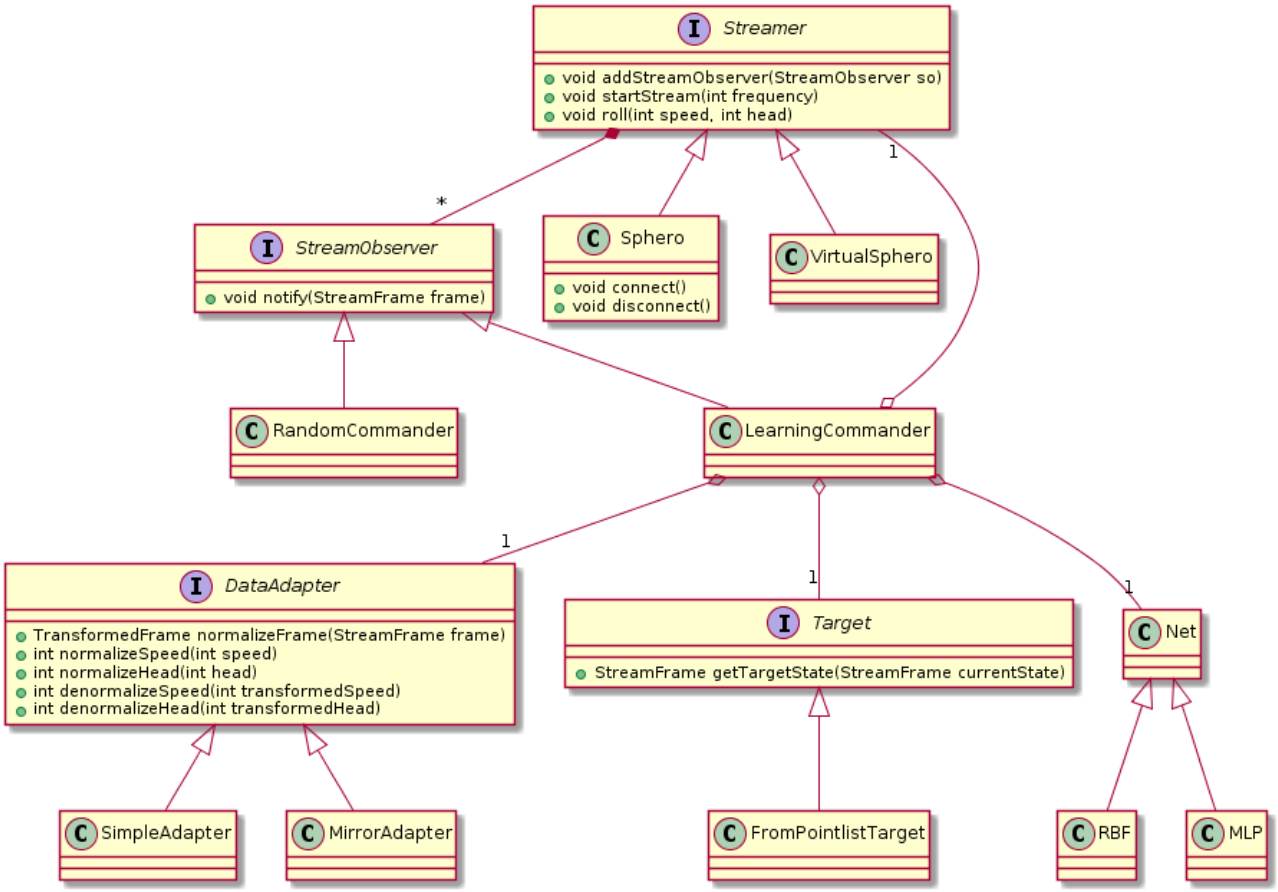


FIGURE 19 – Diagramme de classe dy système de commande.Généré par plantUML

3.6.1 Structure

En résumé, on inscrit le commander par machine learning à un Streamer. Ensuite on démare le streaming en donnant la fréquence f au Streamer. Le Streamer va périodiquement notifier le commander d'une nouvelle frame de données issues du streaming. Le commander va demander au Target l'état cible à atteindre dans $\frac{1}{f}$ secondes d'après l'état actuel du Streamer. Le commander Passe ensuite l'état actuel et l'état à atteindre dans un DataAdapter qui a pour but de résumer les données et les fournir comme input au réseau de neurones. Ce DataAdapter se charge aussi de réduire le domaine d'entrée pour avoir une convergence plus rapide. Le commander récupère le output du réseau de neurones et le passe au DataAdapter qui éffectuera une transformation afin d'obtenir les commandes à envoyer directement au Streamer. Cette procédure est illustré dans la Figure 18 est reprend les éléments suivants :

- **`StreamFrame`** : Une structure contenant les données brutes du streaming. C'est à dire l'orientation de la Sphero, sa position, son vecteur vitesse et son vecteur d'accélération mais aussi le temps entre la récéption de cette frame et la précédente.
- **`StreamObserver`** : Un objet qui pourra être notifié par un Streamer chaque fois que une `StreamFrame` est reçue. LearningCommander mais aussi le commander par commandes aléatoires sont des StreamObservers. (Figure 19)
- **`Streamer`** : Un objet capable de fournir des données issues d'un streaming et les envoyer au StreamObserver qui lui est abonné. La Sphero et VirtualSphero sont des streamers. (Figure 19)
- **`Target`** : Fourni un état à atteindre d'après l'état actuelle de la Sphero.
- **`LearningCommander`** : Il essaie d'atteindre l'état demandé par Target
- **`DataAdapter`** : Transforme les données de streaming et de l'état à atteindre afin de réduire le domaine d'entrée des données à fournir au réseau de neurones. Permet la transformation inverse.
- **`TransformedFrame`** : Une structure contenant les données de streaming mais relatif à la position actuelle de la Sphero et relatif à son orientation, avec les données nécessaires sur l'état à atteindre.
- **`Net`** : Le réseau de neurones.

3.6.2 Génération de commande aléatoire

Afin d'obtenir une base de données qui permetra d'entrainer préalablement le réseau de neurones, On va d'abord la piloter de manière aléatoire et récuperer ses données de streaming ainsi que les commandes fournies pour y arriver. Le comportement de ce générateur a un impact important sur la qualitée des données. Ceci a été observé expérimentalement.

Éviter les colisions

Tous les générateurs de commandes aléatoire implémentés durant ce projet ont été conçu pour éviter que la Sphero sorte d'une zone rectangulaire afin d'éviter les colisions. Lorsque la Sphero dépasse sa zone, des commandes sont envoyées afin que la Sphero fasse demi-tour avec une orientation sufisament proche de celle qui le permetra de revenir dans la zone. Par exemple, si la Sphero dépasse le bord droit de la zone, alors l'orientation commandée est dans l'intervale $270 \pm \Delta\text{max}_{\text{about-turn}}$ où $\Delta\text{max}_{\text{about-turn}}$ est un paramètre inférieur à 90 degrés.

Les erreurs influençant la qualité des données

Tout d'abord il faut éviter que la différence entre l'orientation actuelle de la Sphero et celle qui est commandée soit supérieur à ce que la Sphero est capable de faire avec sa vitesse angulaire maximale. En effet, supposons que la Sphero soit capable de changer son orientation

de maximum 30 degrés sur un temps de $\frac{1}{f}$ où f est la fréquence de streaming. Alors toutes les commandes dont le changement d'orientation est entre 30 et 180 degrés, à partir d'un même état initial, aboutiront au même état-cible. C'est à dire que plusieurs commandes de valeur significativement différentes sont optimales pour atteindre cet état-cible. Ce qui est mauvais pour la convergence du réseau de neurones. Si plusieurs outputs sont optimales pour le même input, alors la solution du réseau de neurones ne convergera pas vers un de ces outputs mais vers un outputs de valeur entre ces deux outputs optimals. Cela n'empêchera pas de produire un modèle performant au niveau de la commande puisque à partir du moment où l'orientation commandée est trop élevée pour que la Sphero puisse l'atteindre en $\frac{1}{f}$ secondes, sa valeur n'influence pas la conduite. Par contre c'est pour le calcul d'erreur que cela pose problème. Et non seulement cela empêche une bonne convergence mais cela élargi aussi le domaine d'entrée. Expérimentalement, sans limiter la différence d'orientation, ni le réseau de neurones implémenté, ni le MLP de Weka, ni celui du package nnet de R ne sont capables de fournir une solution correcte. Leur comportement est de retourner la moyenne des outputs, quelque soit l'input.

Ensuite, afin d'effectuer des trajectoires aléatoires mais naturelles et utiles pour l'apprentissage, il faut pouvoir simuler un humain qui effectuerait des mouvements aléatoires sur une télécommande. En effet, si la différence d'orientation entre l'état actuelle et l'orientation commandée est purement aléatoire, alors elle alterne rapidement entre positif et négatif. Ce qui provoque des tremblements, des mouvements balanciers et des trajectoires trop souvent en ligne droite. Les données de streaming issues d'un tel comportement ne permettaient pas non plus au réseau implémenté et au MLP de Weka de converger vers une autre solution que la moyenne de tous les outputs. Pour pouvoir simuler cette commande naturelle, le générateur va d'abord choisir une orientation aléatoire. Ensuite un nombre d'étapes aléatoires. Soit N le nombre d'étapes, soit d la différence d'angle (celle inférieur à 180°) entre l'orientation actuelle de la Sphero et la ciblée. Le générateur va commander une rotation de $\frac{d}{N}$ pour les N prochaines commandes sauf au moment où la Sphero quitte sa zone délimitée. Le nombre N est généré de sorte à éviter le premier problème cité. Ensuite, lorsque la Sphero atteind l'orientation cible, elle a une chance sur trois de changer d'orientation cible, afin d'éffectuer quand-même des mouvements droits. La génération de commande sur la vitesse se fait de la même manière.

Et pour finir, il y a une ambiguïté sur le format de l'angle obtenu par streaming. En effet, dans la documentation, il est écrit que pour commander une nouvelle orientation, celle-ci doit être un angle entre 0°et 359°et dont le sens est le suivant : 0°sigifie droit devant, 90°à droite, 180°en arrière et 270°à gauche. [8] L'angle obtenu par streaming (et uniquement par streaming) est entre -179°et 180°. Mais il n'y pas d'information concernant le sens. Il a été conclu que le sens des angles échantillonnés est l'inverse de l'angle à envoyer en commande en observant qu'en commandant l'orientation à 90°, la Sphero fini par échantillonner -90°.

3.6.3 Récupération de la trajectoire

Passons ensuite au commander utilisant le réseau de neurones. La Sphero ne sera pratiquemment jamais exactement sur la position cible. Actuellement, pour résoudre ce problème, l'objet Target cherche parmis les points de sa trajectoire, la plus proche de la position actuelle. Il renvoie ensuite l'état juste après celui dont la position est le plus proche de la position actuelle de la Sphero. Mais cette technique présente plusieurs inconvénients : Elle ne permet pas les trajectoires croisées comme les trajectoires en 8. De plus, la Sphero pourrait dévier assez fort de sa trajectoire cible. Dans ce cas, la position de la Sphero risque d'être plus proche d'un autre point de la trajectoire que celui qu'il est sensé atteindre.

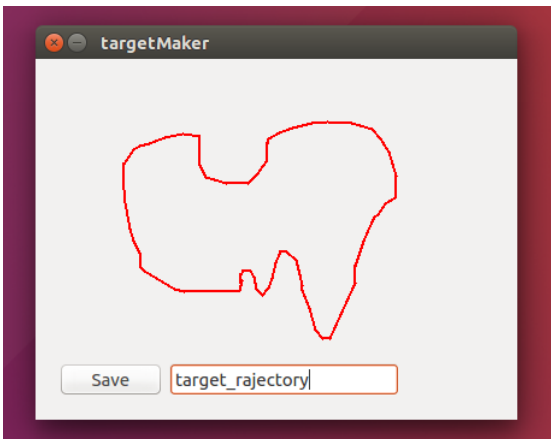


FIGURE 20 – Éditeur de trajectoire

3.6.4 Éditeur de trajectoire

Une interface graphique a été implémentée en Qt afin de créer des trajectoires à la souris. (Figure 20)

3.7 Expérimentation

Observons ensuite le résultat de l'apprentissage des réseaux de neurones sur les données issues de la commande aléatoire.

3.7.1 Test du réseau de neurones

Afin de vérifier si le réseau de neurones implémenté ne contient pas d'erreur, il sera d'abord utilisé pour une Sphero virtuelle. Un modèle très simple de Sphéro a été implémenté. Ce modèle ne simule pas la vitesse et l'inertie angulaire. Le modèle est le suivant : Soit

- $(X_{\text{actuelle}}, Y_{\text{actuelle}})$ la position actuelle en cm,
- V_{actuelle} la vitesse actuelle en cm/s
- V_{target} la vitesse commandée d'unité inconnue,
- O_{actuelle} l'orientation en degrés,
- O_{target} l'orientation commandée en degrés,
- T la période de streaming.

$$\text{acceleration} = a(V_{\text{target}} - V_{\text{actuelle}})$$

Où $a \in \mathbb{R}^+$ est un paramètre.

$$\text{Nouvelle vitesse} = V_{\text{actuelle}} + \text{acceleration} \times T$$

Posons d la différence entre O_{target} et O_{actuelle}, négatif si le sens de $O_{\text{actuelle}} \to O_{\text{target}}$ est horlogique. Alors

$$\text{Nouvelle orientation} = O_{\text{actuelle}} + d$$

Nouvelle vitesse et Nouvelle orientation sont limitées selon un paramètre.

$$\begin{aligned}\text{Nouvelle position} = \quad & (X_{\text{actuelle}} + (\text{Nouvelle vitesse})\cos(\text{Nouvelle orientation}),\\ & Y_{\text{actuelle}} + \text{Nouvelle vitesse}\sin(\text{Nouvelle orientation}))\end{aligned}$$

Deux réseaux de neurones ont été testés : le RBF implémenté et le MLP de Weka. La fonction d'activation des neurones cachés de Weka est la sigmoïde :

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

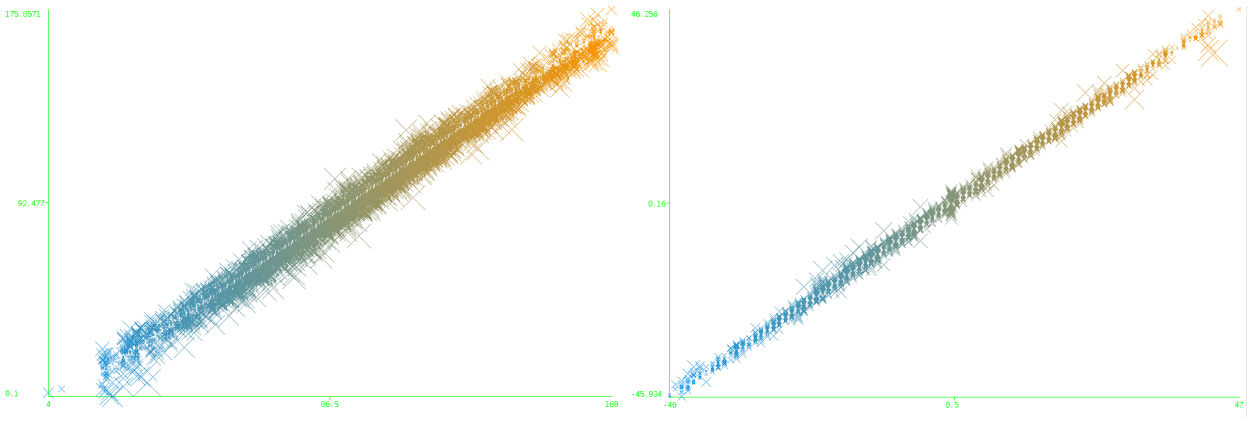


FIGURE 21 – Nuages de point prédit/espéré sur la vitesse et l'orientation sur les données de la Sphero virtuelle.(Généré par Weka 3.8.1)

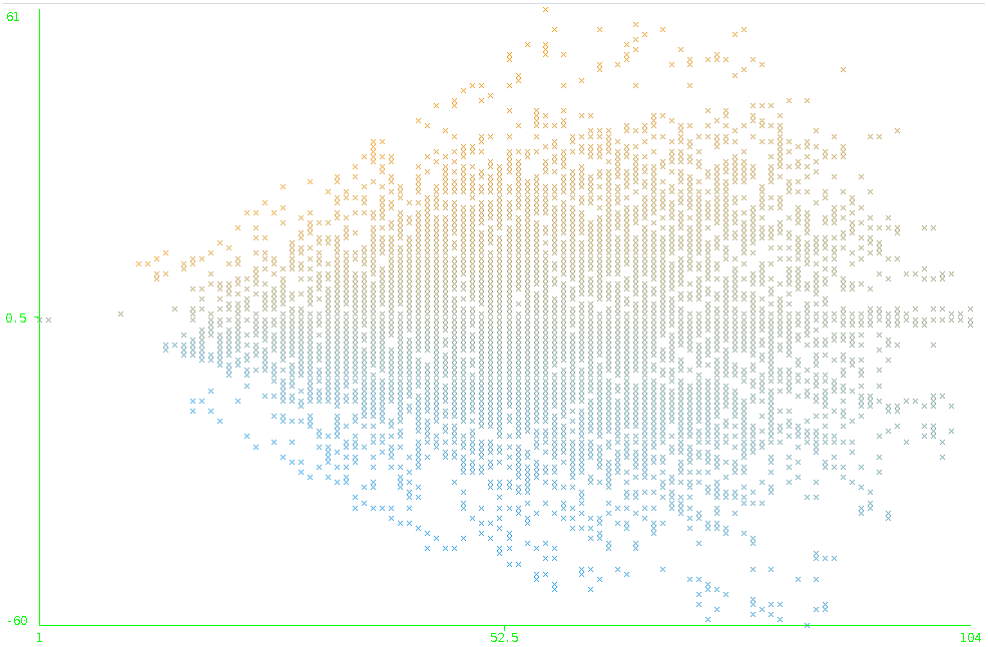


FIGURE 22 – Nuage des positions et de la direction à commander pour les atteindre. Pour la Sphero virtuelle.(Généré par Weka 3.8.1)

et la fonction d'activation des neurones de sortie est une fonction linéaire. [14]

Tous les deux étaient capables d'approximer ce modèle. Pour que les résultats observés ne soient pas biaisées par le surajustement, la technique de 10fold cross-validation a été utilisée. C'est-à-dire que la base de données est séparée en 10 ensembles de tailles égales. Pour chaque ensemble, on construit le modèle sur les 90 autres pourcents de données et on utilise ces 10% comme ensemble de test. À la fin, on obtient un nuage de points (sortie espérée, sortie aproximée par le modèle) où nous pouvons, par exemple, calculer le coefficient de corrélation pour avoir un indice sur la qualité du réseau de neurones pour ces données. Comme sur la Figure 21 où la corrélation est de 0,99 pour la vitesse et l'orientation. Deux modèles différents ont été construit car il a été observé que le RBF implémenté est plus éfficace si il n'y a qu'une sortie à générer au lieu de deux.

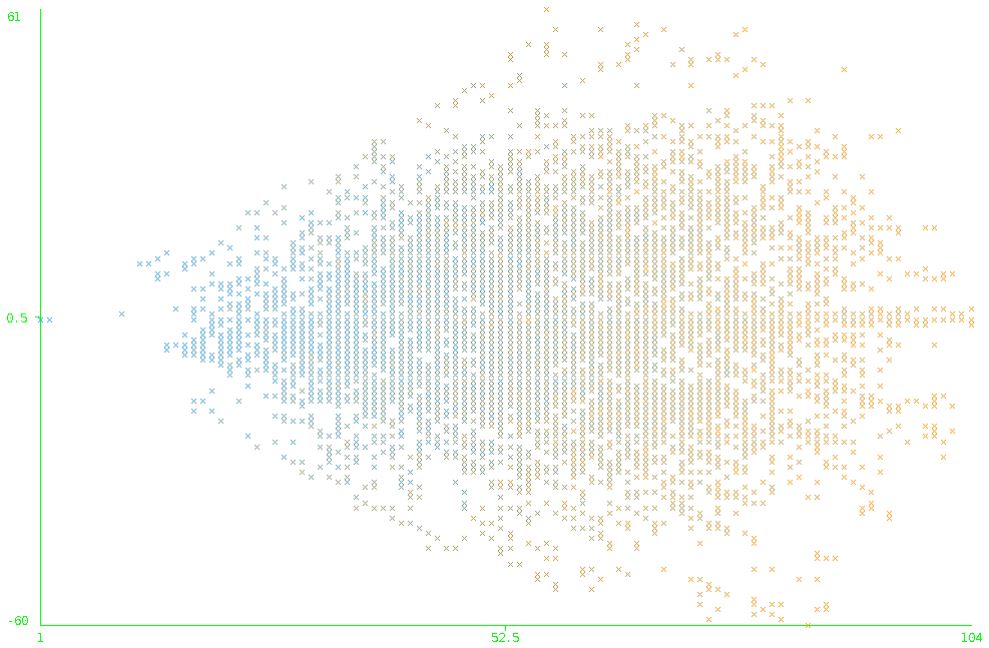


FIGURE 23 – Nuage des positions et de la vitesse à commander pour les atteindre. Pour la Sphero virtuelle.(Généré par Weka 3.8.1)

3.7.2 Observation des données réelles

Observons les données fournies par la commande aléatoire sur la véritable Sphero. Dans la figure 24 et 25, chaque point représentent la position que doit atteindre la Sphero $\frac{1}{f}$ secondes plus tard. Pour chaque point, la Sphero est en (0,0) et est dirigée vers la droite, parallèle à l'axe x. La couleur du point représente la valeur de l'orientation à commander. Plus le ton est orangé, plus la valeur est grande. On pourrait s'attendre à ce que plus le point est vers le haut, plus la valeur de head est grande, comme on peut l'observer sur les données de commande aléatoire sur la Sphero virtuelle. (Figure 22) On pourrait s'attendre aussi à ce que plus le point à atteindre est éloigné de la position actuelle, plus la vitesse commandée doit être élevée, comme on peut également le voir sur les données venant de la Sphero virtuelle. (Figure 23)

Concernant les données de la Sphero réelle, si on prend les attributs deux à deux, on n'observe pas de pattern quant à la direction à prendre selon les données acquises. Une de ces observation est illustrée dans la Figure 24. Dans cette figure, tous les points ont été légèrement déplacés aléatoirement afin de pouvoir tous les distinguer. Nous devrions voir en plus de 3 dimensions pour observer un éventuel pattern. Mais en tout cas, selon Wela, il y en a un. C'est ce que nous allons voir dans l'expérimentation sur les données réelles. Notons qu'il est possible d'obtenir des données indiquant qu'il faut tourner, par exemple, à droite pour une position qui se trouve plutôt vers la gauche. En effet, après un virage serré, à cause de la force centrifuge exercée sur le centre de gravité, la Sphero se met à se balancer de gauche à droite par rapport à sa direction. Juste avant la fin de ce virage, nous avons commandé à la Sphero de tourner vers la droite. Mais à la fin de son virage, l'inertie de son centre de gravité la fait basculer vers la gauche. Si la position est échantillonée pendant ce basculement, alors on a une position qui peut être vers la gauche par rapport à la position échantillonée juste avant alors qu'on avait commander à la Sphero de tourner vers la droite. Ce problème est analogue aux dérapages. Mais comme il ne s'agit pas d'un dérapage, la position réellement atteinte est effectivement échantillonée par l'odomètre. Afin de détecter ces cas, il nous faudrait observer la vitesse ou l'accélération en plus de la position à atteindre.

Tandis que pour la vitesse à commander, sur la Figure 25 on observe effectivement une

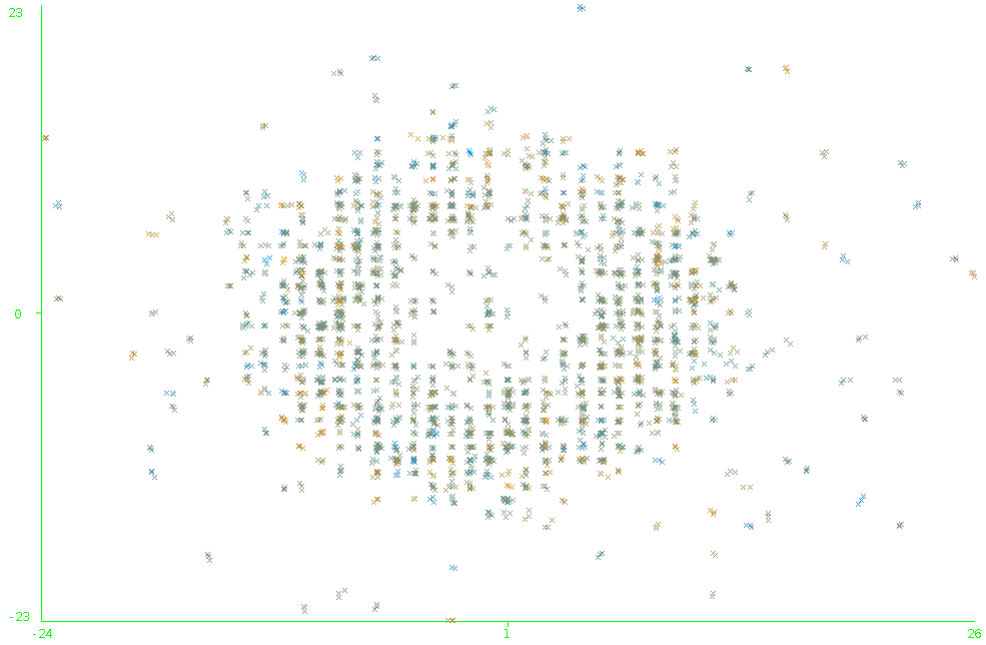


FIGURE 24 – Nuage des positions et de la direction à commander pour les atteindre. Pour la Sphero réelle.(Généré par Weka 3.8.1)

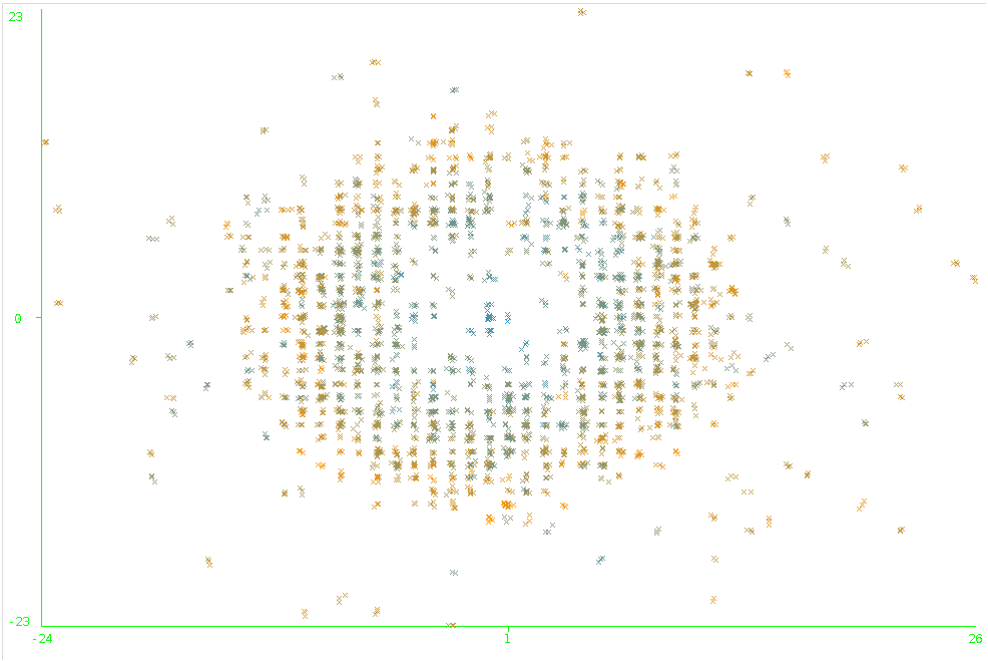


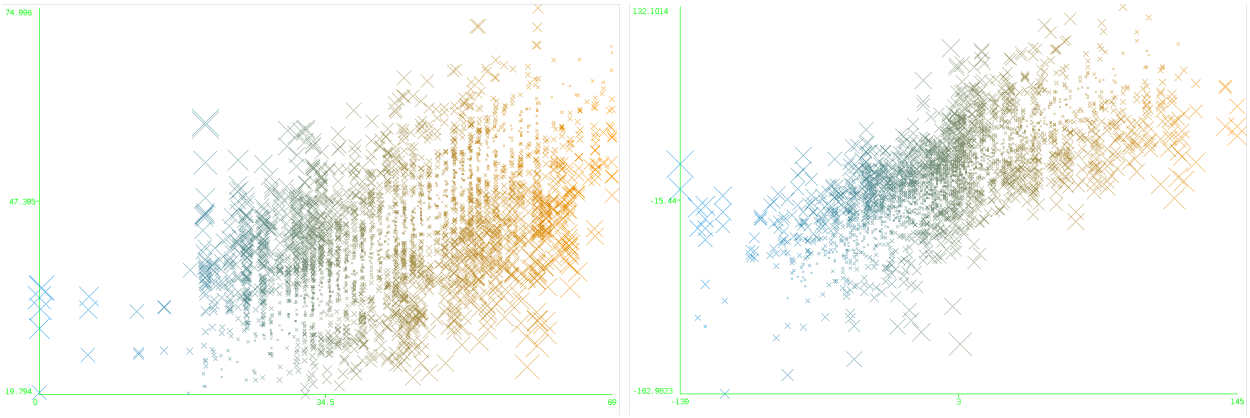
FIGURE 25 – Nuage des positions et de la vitesse à commander pour les atteindre. Pour la Sphero réelle.(Généré par Weka 3.8.1)

tendance à commander une vitesse plus élevée pour atteindre des positions plus éloignées. Plus la couleur du point est orangée, plus la vitesse est élevée.

3.7.3 Expérimentation sur données réelles

Puisque cette expérience s'est déroulée en même temps que la phase de conception et de test du générateur de commande aléatoire, il a fallut préparer un espace accessible sur une assez longue période. La vitesse a été bridée et aucune caméra n'a été utilisée. Dans ces conditions, une fréquence de 40Hz était trop élevée pour capturer les différences de positions car l'odométrie est capturée en nombre entier en cm. Une fréquence de 5Hz était assez basse pour observer les différences de position mais assez élevée pour effectuer des mouvements fluides. Pour toutes les obsevations effectuées, le RBF implémenté retournait toujours la moyenne des outputs.

Avec données relatives à la position et l'orientation

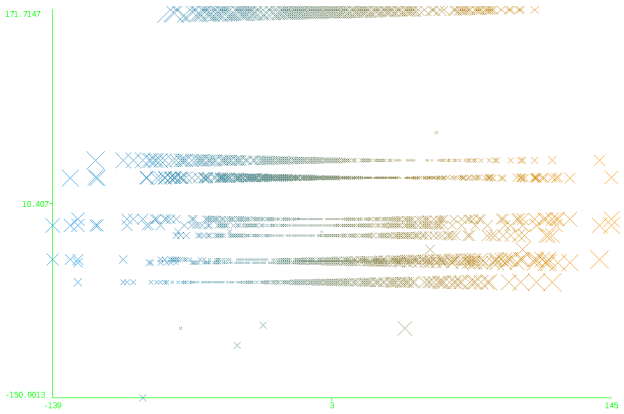


Vitesse **Orientation**

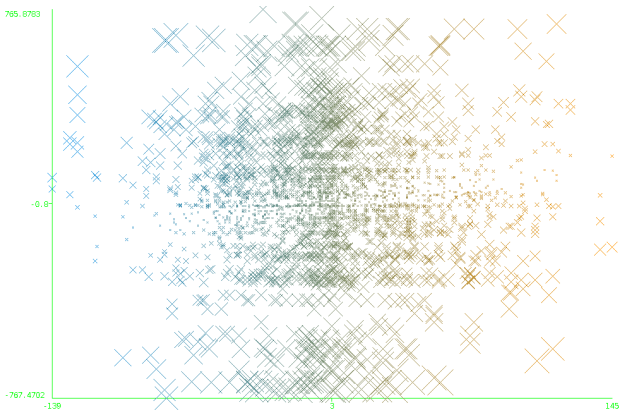
Ici, les données d'entrées sont relatives à la position de la Sphero et à son orientation. Les données sont normalisées avant des les utiliser dans le réseau de neurones. Si les valeurs de sortie ne sont pas normalisées, alors si elles ont un domaine différent, elles n'auront pas le même poids lors du calcul d'erreur. Si les valeurs d'entrée ne sont pas normalisées, alors nous devons paramètrer nous-mêmes l'initialisation de certains neurones. Comme les neurones du RBF, si ils sont initialisés avec un écart-type de 1 et une moyenne de 0 mais que les valeurs d'entrées peuvent atteindre plusieurs centaines, alors malgré la précision en 64bits du calcul de la fonction d'activation, elles retourneront souvent 0. Idem pour la fonction sigmoïde, elle retourna souvent 0 ou 1.

En construisant le modèle en règlant les paramètres, un optimum local a été atteint sur Weka. Il s'agit d'une seule couche de 20 neurones cachés avec un learning rate (i.e. pas de gradient) de 0,3 et un nombre d'époques de 1500. (C'est-à-dire que les données d'entrainement sont présentés 1500 fois aux réseaux de neurones). Le coefficient de corrélation du modèle construit par Weka est de 0.5404 pour la vitesse et de 0,69 pour l'orientation.

Sans normalisation des données



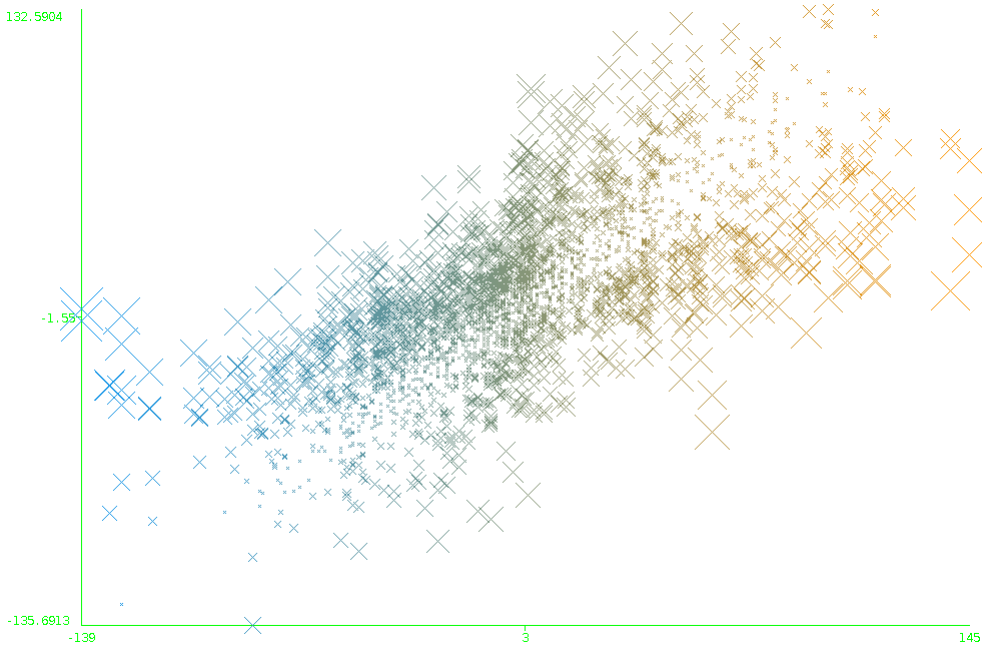
Vitesse



Orientation

Si nous n'avions pas normalisé les données, le coefficient de corrélation du modèle construit par Weka est de 0,0119 pour la vitesse et 0,0264 pour l'orientation. Nous pouvons d'ailleur ci-dessus observer un comportement étrange sur les prédictions données par le modèle. La configuration du réseau de neurones est la même que la précédente.

Ajout de la vitesse commandée en attribut



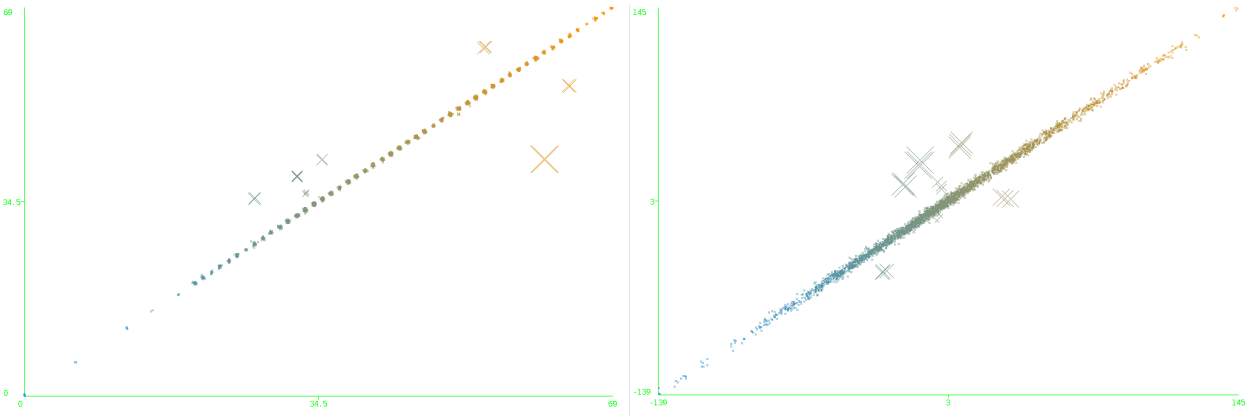
Revenons aux données normalisées et ajoutons la vitesse commandée en attribut. Toujours avec la même configuration du réseau de neurones. Puisque nous pouvons utiliser un réseau différent pas sortie, il est possible de d'abord prédire la vitesse à commander et d'ensuite ajouter cette donnée pour la prédiction de l'orientation à prendre si celle-ci améliore la précision. Dans ce cas, le coefficient de corrélation atteint 0,72 pour l'orientation. Un test est nécessaire afin de confirmer si cette augmentation de 0,03 est significatif. Par ailleur, si on ajoute l'orientation commandée comme attribut pour le modèle prédisant la vitesse à commander, la corrélation est de 0,4991.

En considérant la symétrie orthogonale

Dans la section 3.1.2 nous avons évoquer une possibilité d'une symétrie orthogonale d'axe x sur l'orientation et la vitesse à prendre. C'est d'ailleur ce qui est observé sur les données

de la Sphero virtuelle, Figure 22 et 23. Nous allons le vérifier pour les données réelles. Pour chaque instance, si l'ordonnée de la position à atteindre est négatif, on multiplie par -1 toutes les ordonnées des entrées. C'est-à-dire que si targety est <0, alors targety = -targety, currentSpeedy = -currentSpeedy et currentAccely = -currentAccely. Toujours avec la même configuration de réseau, le coefficient de corrélation devient 0,5123 pour la vitesse et 0,6339 pour l'orientation.

Résultats étranges avec le one nearest neighbor



Vitesse **Orientation**

Des résultats étranges ont été observés lorsque un modèle de régression est construit par le *one nearest neighbor* sur ces données via Weka. Pour prédire une valeur à partir des entrées, le modèle effectue un calcul de distance entre ces entrées et toutes les entrées des instances de l'ensemble d'apprentissage. Ensuite il retient l'instance la plus proche et retourne sa sortie. Ce modèle utilise des données normalisées. La corrélation atteint 0,9977 pour la vitesse et 0,9985 pour l'orientation ! Rappelons que par le principe du 10cross validation, aucune instance de test n'est reprise dans les instances d'entrainement.

Le one nearest neighbor a été implémenté dans le commander et testé sur les mêmes données utilisées dans Weka. Pour la validation, le Ncross validation a été appliqué où N est le nombre d'instance. Les modèles générés devraient donc être théoriquement plus performant que ceux construits durant un 10cross validation. Mais on observe une corrélation de ? pour la vitesse et ? pour l'orientation. Au vu de ces résultats, nous pouvons avoir des doutes quant aux résultats des modèles de réseaux de neurones produits pas Weka.

3.7.4 Déploiement

Nous avons donc trouvé un réseau de neurones convergeant vers une solution de corrélation de plus de 0,50 entre la vitesse à commander prédite et celle qui est attendue et de presque 0,70 pour l'orientation à commander. Au lieu d'implémenter le même réseau pour le commander, utiliser une API de réseau de neurones nous permetra de modifier plus rapidement la configuration du réseau et de tester de nouveaux types de réseau. L'API choisie est Caffe elle permet de mofidier aisément l'architecture via un fichier texte au format prototxt. [13] Les paramètres du modèle comme le nombre d'époques ou le pas de gradient peuvent aussi être modifiée via un tel fichier. De plus, étant en c++, le réseau peut rapidement être intégré dans l'impémentation actuelle du commander. L'enregistrement et le chargement d'un modèle est entièrement pris en charge par Caffe et ne dépend pas de l'architecture. Et enfin, l'implémentation de Caffe est multithreadé.

Avec ce nouveau réseau de neurones, les performances du commander sur les données de la Sphero virtuelle sont bonnes. Malheureusement, avec les données de la Sphero réelle, ce modèle converge vers la moyenne de toutes les sorties à prédire.

4 Conclusion

Malgré que le projet ne soit pas arrivé à son terme, nous avons découvert que pour pouvoir piloter une Sphero avec un réseau de neurone, nous avons besoin de nouveaux attributs ou d'une autre achitecture de réseau de neurones. En effet, malgré tous les problèmes rencontrés nuisant à la qualité des données récoltées (section 3.1.2 et 3.1.2) , un réseau de neurone devrait être capable d'approximer une fonction même avec des imprécisions, ou, du moins, de converger vers une autres solution que la moyenne de toutes les sorties. Les autres attributs qui peuvent être ajoutées sont la vitesse à atteindre, l'inclinaison gauche-droite, l'inclinaison arrière-avant et les commandes précédentes.

Le comportement étrange de Weka, section 3.7.3, nous a peut-être mis sur une mauvaise piste. Pensant que le problème vient du réseau de neurones implémenté alors qu'il peut provenir des données. Il faudrait donc à l'avenir éviter d'utiliser ce logiciel pour des tâches de régression.

Dans l'état actuel de ce projet, nous avons un système de commande commandant aléatoirement mais en effectuant des trajectoires naturelles. Nous pouvons désormais éditer facilement l'architecture du réseau de neurones pour le système de commandes, éditer des trajectoires, ou encore integrer de nouvelles transformations de données ou de nouveaux modèles de Sphero virtuelles.

Enfin, le développement d'un réseau de neurones nous a permis de comprendre en détails son fonctionnement et une propriété qui les rend très modulaires.

Remerciements

Je remercie Monsieur Pierre Hauweele de m'avoir proposé un projet qui est exactement dans le domaine que je voulais, de son aide et de la direction de mon projet.

Je remercie Monsieur Hadrien Mélot pour la direction de mon projet et de l'aide qu'il m'a apporté.

Je remercie Monsieur Tom Mens pour son feed-back et ses conseils sur la rédaction de ce rapport.

Je remercie Monsieur Gauvain Devillez pour m'avoir débloqué plusieurs fois dans la compilation et ses conseils.sec :realdataex

Références

[1] Arnon Amir, Pallab Datta, William P. Risk, Adrew S. Cassidy, Jeffrey A. Kusnitz, Steve K. Esser, Alexander Andreopoulos, Theodore M. Wong, Myron Flickner, Rodriguo Alvarez-Icaza, Emmet McQuinn, Ben Shaw et Dharmendra S. Modha : *Cognitive Computing Programming Paradigm : A Corelet Language for Composing Networks of Neurosynaptic Cores.* San Jose.

[2] Simon Haykin : *Neural Networks and Learning Machine.* McMaster University, New Jersey, troisième édition édition, 2008.

[3] Réseaux de neurones. `http://www.statsoft.fr/concepts-statistiques/reseaux-de-neurones-automatises/reseaux-de-neurones-automatises.htm`. [En ligne ; consulté le 26 novembre 2016].

[4] Eric Gauthier : *Utilisation des réseaux de neurones artificiels pour la commande d'un véhicule autonome.* Thèse de doctorat, Insitut National Polytechnique de Grenoble - INPG, 1999.

[5] Pomerleau Dean : *Neural Network Perception for Mobile Robot Guidance*. Thèse de doctorat, Carnegie Mellon University - CMU-CS, Pittsburgh, 1992.

[6] McCormick Chris : Radial basis function network RBFN. `http://mccormickml.com/2013/08/15/radial-basis-function-network-rbfn-tutorial/`, 2013. [En ligne ; consulté le 20 novembre 2016].

[7] Ian J.Goodfellow, Jonathon Shlens et Christian Szegedy : Explaining and harnessing adversarial examples. Cornell University Library, mars 2015.

[8] Sphero Docs | Robot Tech. `http://sdk.sphero.com/`, 2016. [En ligne ; consulté le 20 novembre 2016].

[9] Sphero®2.0 - sphero store. `https://store.sphero.com/collections/sphero/products/sphero-2-0`. [En ligne ; consulté le 23 novembre 2016].

[10] Sphero developer center. `https://developer.gosphero.com/`. [En ligne ; consulté le 24 novembre 2016].

[11] mmwise : A sphero ROS driver. `https://github.com/mmwise/sphero_ros`, 2016. [En ligne ; consulté le 20 novembre 2016].

[12] slock83 : Simple API for sphero 1&2.0 written in c++ for bluez stack. `https://github.com/slock83/sphero-linux-api`, 2016. [En ligne ; consulté le 20 novembre 2016].

[13] Blobs, layers, and nets : anatomy of a caffe model. `http://caffe.berkeleyvision.org/tutorial/net_layer_blob.html`. [En ligne ; consulté le 11 août 2017].

[14] Weka : Multilayerperceptron. `http://weka.sourceforge.net/doc.stable/weka/classifiers/functions/MultilayerPerceptron.html`. [En ligne ; consulté le 14 août 2017].

A Équations de rétropropagation d'un MLP

La modification à apporter au poids j du neurone i vaut

$$\Delta W_{ij} = -\eta \frac{\partial Q}{\partial W_{ij}} \tag{5}$$

$$\frac{\partial Q}{\partial W_{ij}} = \frac{\partial Q}{\partial \phi_i} \frac{\partial \phi_i}{\partial v_i} \frac{\partial v_i}{\partial W_{ij}} \tag{6}$$

Et on a posé

$$\delta_i = \frac{\partial Q}{\partial \phi_i} \frac{\partial \phi_i}{\partial v_i} \tag{7}$$

En reprenant (6), nous avons maintenant que

$$\frac{\partial Q}{\partial W_{ij}} = \delta_i \frac{\partial v_i}{\partial W_{ij}} \tag{8}$$

Nous allons déterminer la valeur de δ_i et de $\frac{\partial v_i}{\partial W_{ij}}$. Commençons par déterminer $\frac{\partial v_i}{\partial W_{ij}}$. Soit x_{ij} la $j^{\text{ième}}$ entrée du neurone i,

$$\begin{aligned} \frac{\partial v_i}{\partial W_{ij}} &= \frac{\partial (W_{i0} x_{i0})}{\partial W_{ij}} + \frac{\partial (W_{i1} x_{i1})}{\partial W_{ij}} + ... + \frac{\partial (W_{ij} x_{ij})}{\partial W_{ij}} + ... + \frac{\partial (W_{im} x_{im})}{\partial W_{ij}} \\ &= x_{ij} \end{aligned} \tag{9}$$

Ensuite, déterminons la valeur de δ_i selon deux cas : si i est un neurone de sortie et si i est un neurone caché.

Si le neurone i est un neurone de sortie,

$$\begin{aligned}\frac{\partial Q}{\partial \phi_i} &= \frac{1}{2}\left(\frac{\partial(\phi_0 - s_0)^2}{\partial \phi_i} + \frac{\partial(\phi_1 - s_1)^2}{\partial \phi_i} + ... + \frac{\partial(\phi_i - s_i)^2}{\partial \phi_i} + ... + \frac{\partial(\phi_l - s_l)^2}{\partial \phi_i}\right) \\ &= \frac{1}{2} 2(\phi_i - s_i)\frac{\partial(\phi_i - s_i)}{\partial \phi_i} \\ &= (\phi_i - s_i)\end{aligned} \tag{10}$$

Soit ϕ' la dérivée de ϕ,

$$\frac{\partial \phi_i}{\partial v_i} = \phi'(v_i) \tag{11}$$

Par (10) et (11), on a

$$\delta_i = (\phi_i - s_i)\phi'(v_i)$$

Si le neurone i est un neurone caché, alors soit n le nombre de neurones de la couche suivante (plus proche des neurones de sorties), soit k un neurone de la couche suivante, on sait que Q est fonction composée, dépendant de ϕ_k, lui-même dépendant de v_k, dépendant lui-même de ϕ_i. C'est pour cela qu'en appliquant le théorème de dérivée des fonctions composées,

$$\frac{\partial Q}{\partial \phi_i} = \sum_{k=1}^{n} \frac{\partial Q}{\partial \phi_k}\frac{\partial \phi_k}{\partial v_k}\frac{\partial v_k}{\partial \phi_i} \tag{12}$$

Si on subsitue (7) dans (12),

$$\frac{\partial Q}{\partial \phi_i} = \sum_{k=1}^{n} \delta_k \frac{\partial v_k}{\partial \phi_i} \tag{13}$$

Or ϕ_i est l'entrée du neurone k recevant la sortie du neurone i. Posons W_{ki} le poids que k attribue à ϕ_i. On peut alors continuer à développer (13) comme suit :

$$\frac{\partial v_k}{\partial \phi_i} = \frac{\partial W_{ki}\phi_i}{\partial \phi_i} = W_{ki} \tag{14}$$

Pour la valeur de $\frac{\partial \phi_i}{\partial v_i}$, nous pouvons reprendre (11). Et du coup,

$$\delta_i = \phi_i'(v_i)\sum_{k=1}^{n} \delta_k W_{ki}$$

Nous connaissons maitenant la modification à appliquer sur W_{ij}. Pour résumé, subsituons δ_i et (9) à (6),

$$\frac{\partial Q}{\partial W_{ij}} = \delta_i x_{ij} \tag{15}$$

Qu'on subsitue à (5) et nous avons enfin

$$\Delta W_{ij} = -\eta \delta_i x_{ij} \text{ où } \begin{cases} \delta_i &= (\phi_i - s_i)\phi'(v_i) & \text{Si } i \text{ est un neurone de sortie} \\ \delta_i &= \phi_i'(v_i)\sum_{k=1}^{n} \delta_k W_{ki} & \text{Si } i \text{ est un neurone caché} \end{cases} \tag{16}$$

B Équation de rétropropagation d'un RBF

Les modifications à apporter aux paramètres d'un neurone i vaut

$$\Delta W_{ij} = -\eta \frac{\partial Q}{\partial W_{ij}} \tag{17}$$

$$\Delta \mu_{ij} = -\eta \frac{\partial Q}{\partial \mu_{ij}} \tag{18}$$

$$\Delta \sigma_{ij} = \eta \frac{\partial Q}{\partial \sigma_{ij}} \tag{19}$$

D'abord déterminons le facteur $\frac{\partial Q}{\partial W_{ij}}$ dans (17). On est dans le cas où i est un neurone de sortie. Par (1), Q dépend de ϕ_i. Par (4), phi_i dépend de W_{ij}. Par le théorème de dérivée des fonctions composés,

$$\frac{\partial Q}{\partial W_{ij}} = \frac{\partial Q}{\partial \phi_i} \frac{\partial \phi_i}{\partial W_{ij}} \tag{20}$$

Par (10), on sait que

$$\frac{\partial Q}{\partial \phi_i} = (\phi_i - s_i)$$

Ensuite,

$$\begin{aligned} \frac{\partial \phi_i}{\partial W_{ij}} &= \frac{\partial \frac{\sum_{r=1}^{m} W_{ir} x_j}{\sum_{r=1}^{m} x_r}}{\partial W_{ij}} \qquad \text{par (4)} \\ &= \frac{1}{\sum_{r=1}^{m} x_r} \frac{\partial \sum_{r=1}^{m} W_{ir} x_j}{\partial W_{ij}} \\ &= \frac{x_j}{\sum_{r=1}^{m} x_r} \end{aligned} \tag{21}$$

Subsituons (10) et (21) dans (20).

$$\frac{\partial Q}{\partial W_{ij}} = (\phi_i - s_i) \frac{x_j}{\sum_{r=1}^{m} x_r} \tag{22}$$

Et (22) dans (17),

$$\Delta W_{ij} = -\eta (\phi_i - s_i) \frac{x_j}{\sum_{r=1}^{m} x_r}$$

Déterminons maintenant le facteur $\frac{\partial Q}{\partial \mu_{ik}}$ dans (18). On est dans le cas où i est un neurone caché. Q dépend de ϕ_i qui lui-même est fonction de μ_{ik} (3). Par le théorème de dérivée de fonction composés,

$$\frac{\partial Q}{\partial \mu_{ik}} = \frac{\partial Q}{\partial \phi_i} \frac{\partial \phi_i}{\partial \mu_{ik}} \tag{23}$$

On va déterminer la valeur de $\frac{\partial Q}{\partial \phi_i}$ et puis de $\frac{\partial \phi_i}{\partial \mu_{ik}}$. Soit j un neurone de sortie. On sait que Q dépend de tous les ϕ_j et les ϕ_j dépendent de ϕ_i. On sait donc que

$$\frac{\partial Q}{\partial \phi_i} = \sum_j \frac{\partial Q}{\partial \phi_j} \frac{\partial \phi_j}{\partial \phi_i} \tag{24}$$

Par (10), on sait que

$$\frac{\partial Q}{\partial \phi_j} = (\phi_j - s_j)$$

Et posons x_r l'entrée de j provenant de i, c'est à dire $\phi_i = x_r$. Posons n la dimension de l'entrée de j. Posons aussi $R = \sum_{k=1}^{n} x_k$. Pour trouver la valeur de $\frac{\partial \phi_j}{\partial \phi_i}$, nous aurons besoin de trouver $\frac{\partial \frac{1}{R}}{\partial x_k}$.

$$\begin{aligned} \frac{\partial \frac{1}{R}}{\partial x_k} &= \frac{-1}{R^2}\frac{\partial R}{\partial x_k} \\ &= \frac{-1}{R^2} \end{aligned} \tag{25}$$

Dès lors, utilisons (25) pour simplifier l'équation suivante ;

$$\begin{aligned} \frac{\partial \phi_j}{\partial \phi_i} &= \frac{\partial \phi_j}{\partial x_r} \\ &= \frac{\partial \left(\frac{\sum_{k=1}^{n} W_{jk}x_k}{\sum_{k=1} nx_k} \right)}{\partial x_r} \\ &= \frac{\partial \left(\frac{W_{j1}x_1}{R} \right)}{\partial x_r} + \frac{\partial \left(\frac{W_{j2}x_2}{R} \right)}{\partial x_r} + ... + \frac{\partial \left(\frac{W_{jr}x_r}{R} \right)}{\partial x_r} + .. + \frac{\partial \left(\frac{W_{jn}x_n}{R} \right)}{\partial x_r} \\ &= (W_{j1}x_1)\frac{-1}{R^2} + (W_{j2}x_2)\frac{-1}{R^2} + ... + \left[(W_{jr}x_r)\frac{\partial \frac{1}{R}}{\partial x_r} + \frac{\partial (W_{jr}x_r)}{\partial x_r}\frac{1}{R} \right] + ... + (W_{jn}x_n)\frac{-1}{R^2} \\ &= \left(\sum_{k=1}^{n} W_{jk}x_k \frac{-1}{R^2} \right) - W_{jr}x_r\frac{-1}{R^2} + \left[(W_{jr}x_r)\frac{-1}{R^2} + W_{jr}\frac{1}{R} \right] \\ &= \frac{-1}{R^2}\left[\left(\sum_{k=1}^{n} W_{jk}x_k \right) - W_{jr}x_r + (W_{jr}x_r) + W_{jr}(-R) \right] \\ &= \frac{1}{R^2}\left(W_{jr}R - \sum_{k=1}^{n} W_{jk}x_k \right) \end{aligned} \tag{26}$$

On substitue (10) et (26) dans (24) pour obtenir

$$\frac{\partial Q}{\partial \phi_i} = \sum_j (\phi_j - s_j)\frac{1}{R^2}\left(W_{jr}R - \sum_{k=1}^{n} W_{jk}x_k \right) \tag{27}$$

Si nous calculons $\frac{\partial \phi_i}{\partial \mu_{ik}}$, nous obtenons

$$\frac{\partial \phi_i}{\partial \mu_{ik}} = \phi_i \frac{x_k - \mu_{ik}}{\sigma_{ik}^2} \tag{28}$$

Et en substituant (27) et (28) dans (23),

$$\frac{\partial Q}{\partial \mu_{ik}} = \left[\sum_i (\phi_j - s_j)\frac{1}{R^2}\left(W_{jr}R - \sum_{k=1}^{n} W_{jk}x_k \right) \right] \phi_i \frac{x_k - \mu_{ik}}{\sigma_{ik}^2}$$

Qu'on substitue dans (18), on obtient la modification à appliquer sur μ_{ik} :

$$\Delta\mu_{ik} = -\eta \left[\sum_i (\phi_j - s_j)\frac{1}{R^2}\left(W_{jr}R - \sum_{k=1}^{n} W_{jk}x_k \right) \right] \phi_i \frac{x_k - \mu_{ik}}{\sigma_{ik}^2}$$

Enfin déterminons le facteur $\frac{\partial Q}{\partial \sigma_{ik}}$ dans (19). On est dans le cas où i est un neurone caché. Q dépend de ϕ_i qui lui-même est fonction de σ_{ik}. Par le théorème de dérivée de fonction composés,

$$\frac{\partial Q}{\partial \sigma_{ik}} = \frac{\partial Q}{\partial \phi_i} \frac{\partial \phi_i}{\partial \sigma_{ik}} \tag{29}$$

Si nous calculons $\frac{\partial \phi_i}{\partial \sigma_{ik}}$, nous obtenons

$$\frac{\partial \phi}{\partial \sigma_{ik}} = \phi_i \frac{(x_k - \mu_{ik})^2}{\sigma_{ik}^3} \tag{30}$$

On peut substituer (27) et (30) dans (29) :

$$\left[\sum_i (\phi_j - s_j) \frac{1}{R^2} \left(W_{jr} R - \sum_{k=1}^{n} W_{jk} x_k \right) \right] \phi_i \frac{(x_k - \mu_{ik})^2}{\sigma_{ik}^3}$$

Qu'on substitue dans (19), on obtient la modification à appliquer sur σ_{ik} :

$$\Delta \sigma_{ik} = -\eta \left[\sum_i (\phi_j - s_j) \frac{1}{R^2} \left(W_{jr} R - \sum_{k=1}^{n} W_{jk} x_k \right) \right] \phi_i \frac{(x_k - \mu_{ik})^2}{\sigma_{ik}^3}$$