Algorithmes évolutionnistes: de l'optimisation de paramètres à la conception complète d'un système de contrôle.

Stéphane Doncieux
Université Pierre et Marie Curie
LIP6 - AnimatLab
E-mail: Stephane.Doncieux@lip6.fr

***Abstract*— Les algorithmes évolutionnistes, grâce à la généricité de leurs principes, peuvent être un outil d'optimisation mais aussi d'étude de controleurs pour des systèmes complexes dont la dynamique n'est pas encore maitrisée. Leur souplesse leur permet d'être utilisés à plusieurs niveaux, depuis l'optimisation de paramètres pour un contrôleur dont la structure a été prédéfinie, jusqu'à l'étude ou même la conception complète d'un système de contrôle.**

I. Introduction

La plupart des niches écologiques présentes sur terre sont occupées par des espèces vivantes qui ont su s'adapter à des conditions difficiles. Les solutions qu'elles ont apportées aux domaines de l'aérodynamique, de l'hydrodynamique ou même encore des matériaux sont encore aujourd'hui une importante source d'inspiration pour les ingénieurs. Le fil des araignées devrait permettre de créer des gilets pare-balles beaucoup plus solides et l'étude de la peau des dauphins a permis la création de revêtements de torpilles amortissant fortement les turbulences créées à partir d'une certaine vitesse. Toutes ces solutions innovantes ont été découvertes par la nature par essais et erreurs au fil des générations grâce aux interactions entre les animaux et leur environnement. La sélection naturelle des espèces qui leur a ainsi permis d'évoluer au cours du temps peut être exploitée dans un contexte d'ingénierie sur des problèmes pour lesquels des solutions systématiques ne sont pas encore connues. Cela constitue une méthode de recherche susceptible de faciliter l'étude et la résolution de problèmes complexes.

En effet, les algorithmes évolutionnistes, initialement conçus pour optimiser des paramètres, se sont révélés capables d'intervenir plus en amont dans la résolution d'un problème. Si dans le cas de l'optimisation de paramètres d'un contrôleur, la structure du dit contrôleur est donnée à l'avance, de nombreux travaux montrent que des algorithmes évolutionnistes peuvent être utilisés pour obtenir automatiquement cette structure. Ils peuvent intervenir dès la phase de conception du système de contrôle et optimiser à la fois les paramètres et la structure du système de contrôle.

Le fait de travailler sur un ensemble de solutions améliorées progressivement, plutôt que de n'en considérer qu'une, est particulièrement adapté aux problèmes multi-critères dans lesquels des compromis doivent être trouvés. Là où un algorithme d'apprentissage autre ne donnera qu'une seule solution à la fois sur un compromis imposé à l'avance, un AE va pouvoir matérialiser en une passe le front de Pareto, c'est à dire l'ensemble des solutions présentant les meilleurs compromis possibles. L'expérimentateur acquiert ainsi à la fois la connaissance de ce front ainsi que les solutions correspondantes [24], ce qui lui permet de réaliser ses choix sans avoir à relancer son algorithme d'optimisation à de multiples reprises.

Nous allons présenter les principes généraux des algorithmes évolutionnistes (AE), puis nous nous intéresserons à leur application au contrôle de robots, dans un premier temps pour optimiser des paramètres, puis d'un point de vue plus général pour concevoir un système de contrôle complet. Nous présenterons ensuite les différents contrôleurs utilisés couramment en conjonction avec des algorithmes évolutionnistes.

II. Principes généraux

A. Objectif

Les AE constituent une méthode d'exploration automatique d'un espace de recherche potentiellement très vaste. Contrairement à d'autres algorithmes partant d'une solution singulière et cherchant à remonter un gradient de performances, les AE utilisent un ensemble de solutions dont seule la performance ponctuelle est utilisée. Aucune autre propriété mathématique n'est nécessaire. La continuité ou la dérivabilité ne sont pas utilisées, seule la capacité à calculer une valeur ponctuelle de la fonction d'évaluation est nécessaire. La répartition, initialement aléatoire, de ces solutions sur l'espace de recherche limite les risques de convergence prématurée vers des extrema locaux. Les AE constituent ainsi une méthode de recherche guidée par les performances. Le champ des applications potentielles est très vaste compte tenu de la pauvreté des informations nécessaires et de la généricité des principes exploités. Des problèmes d'optimisation peuvent être traités par de tels algorithmes, mais aussi des problèmes d'apprentissage, ou même de conception complète de systèmes de contrôle de robot.

B. Inspiration

Les AE s'inspirent de manière très libre de la théorie darwinienne de l'évolution des espèces. Ces algorithmes travaillent sur une population d'individus. Ces individus contiennent l'information décrivant la solution qu'ils représentent. Cette information est désignée par le terme de *génotype* par référence à

la biologie alors que la solution qu'ils représentent, autrement dit le système cherché, sera nommée *phénotype*. Le génotype est l'information manipulée par l'AE alors que le phénotype en est la traduction après développement, c'est le système dont on cherche à optimiser ou concevoir le comportement. Dans cet objectif, les individus subissent un processus de sélection dont les critères sont définis par le concepteur. Génération après génération, seuls les plus performants survivent et se reproduisent dans l'objectif de générer des descendants plus performants que leurs parents.

L'objectif de la version artificielle de l'évolution est de maximiser une fonction d'adaptation (*fitness* en anglais). Cette fonction d'adaptation est représentative de l'efficacité des solutions générées sur un problème posé. Elle est le principal biais introduit par le concepteur pour guider l'évolution vers les solutions recherchées. La nature n'est pas aussi directive, c'est pourquoi l'analogie ne doit pas être prise au pied de la lettre: l'objectif n'est pas de reproduire, ni même de comprendre le fonctionnement de la sélection naturelle, le but est de disposer d'une méthode efficace de conception et d'optimisation de structures pour répondre à des problèmes particuliers, même si des interactions sont possibles avec des biologistes.

C. Détails d'un algorithme évolutionniste

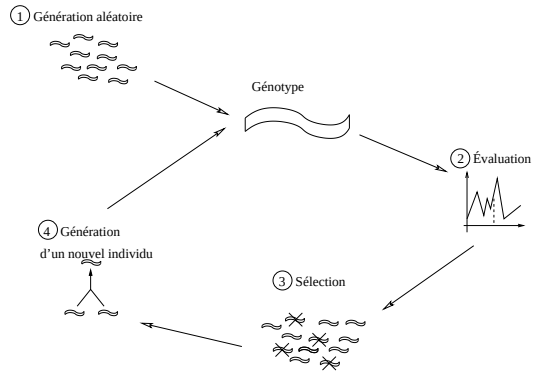

Fig. 1. Principe des algorithmes évolutionnistes. 1. Génération aléatoire de la population de génotypes. 2. Évaluation d'un génotype. 3. Remplacement d'un génotype peu performant par le nouveau génotype. 4. Copie d'un individu performant. 5. Application des opérateurs génétiques de mutation et de croisement.

Dans cette section, nous allons détailler chacune des parties d'un alorithme évolutionniste. La figure 1 présente les principes des algorithmes évolutionnistes.

1) Génération initiale d'une population: Les algorithmes évolutionnistes (AE) travaillent sur une population d'individus qui représentent les solutions potentielles au problème posé. À l'initialisation de l'algorithme, il est nécessaire de générer une population complète qui sera utilisée ensuite par l'algorithme.

La possibilité la plus couramment utilisée consiste à générer aléatoirement des individus. Cependant, il est possible de partir d'une population dont les individus ont été, au moins partiellement préparés manuellement, en incluant des connaissances a priori, de façon à restreindre l'espace de recherche et accélérer la convergence.

2) Évaluation: L'évaluation d'un génotype nécessite une étape de traduction appelée *développement* pendant laquelle le génotype est traduit en un phénotype décrivant tout ou partie d'un agent. Le comportement de l'agent est évalué et une note lui est attribuée selon des critères préétablis dépendant du problème à résoudre. La notation peut être très directive: notation précise de chacun des comportements élémentaires ou, au contraire, laisser plus de latitude à l'évolution en n'observant que le comportement global.

La notation est un élément capital des AE et est également le principal biais introduit par le concepteur. En effet, elle est la base du processus de sélection qui assure la survie des plus performants et leur utilisation pour la génération de nouveaux individus. Une notation très directive va imposer des étapes intermédiaires très précises, alors qu'une notation implicite ne s'intéressera qu'au résultat final.

Une notation explicite laisse moins de latitude à l'évolution, mais permet d'avoir un contrôle plus important sur les solutions qui seront générées. Elle permet de guider l'évolution en lui indiquant précisément quelle stratégie elle doit utiliser pour résoudre le problème. Une telle notation peut être utilisée lorsque la solution souhaitée doit exhiber un comportement connu précisément et pas un autre. Pour faire marcher un robot hexapode, une notation explicite prendra en compte, par exemple, les mouvements des pattes: leur rythme, leur amplitude, leur synchronisation.

Une notation implicite laisse plus de libertés à l'évolution. Sur le problème de la marche d'un robot hexapode, cela reviendra à noter uniquement la distance parcourue pendant un temps donné. Ce sera à l'évolution de trouver le rythme le plus efficace. C'est une description de plus haut niveau: on ne s'intéresse qu'au résultat sans donner de consignes particulières sur la façon de l'atteindre. Ce type de notation peut permettre de trouver des solutions originales au problème posé, car aucune contrainte n'est imposée sur la méthode à employer. Kodjabachian [33] a ainsi trouvé un rythme de marche très efficace pour un robot hexapode: le rythme tripode, utilisé par les insectes, sans avoir spécifié à l'évolution les caractéristiques d'un tel rythme.

Cependant, sur une tâche trop complexe, se pose le problème du démarrage. Si aucun individu de la population initiale ne parvient à résoudre, ne serait-ce que partiellement, le problème, l'algorithme peut rester bloqué dans un minimum local. Si on demande directement à l'évolution de générer un contrôleur capable de faire survivre ne serait-ce qu'un oiseau dans son environnement, il est fort probable, pour ne pas dire certain, que cela ne converge pas car cela nécessite de résoudre de nombreux problèmes à la fois: maîtrise du vol, recherche de nourriture, évitement de prédateurs... La solution, pour éviter un tel blocage, consiste à guider l'évolution en lui fournissant des étapes intermédiaires plus simples, dans une approche dite *incrémentale*: on décompose le problème et on utilise l'évolution pour résoudre tout d'abord un premier problème simple que l'on complète petit à petit jusqu'à atteindre l'objectif final.

3) Sélection: L'algorithme de sélection mis en oeuvre doit assurer la convergence vers une solution efficace tout en maintenant la diversité de la population, aspect caractéristique des AE par rapport aux autres algorithmes d'apprentissage ou d'optimisation. La diversité dans la population diminue les risques de convergence prématurée vers un extremum local et assure un taux de réponse plus rapide en cas de modification de

l'environnement pendant la durée de l'évolution.

Il y a plusieurs façons d'exercer une pression sélective sur la population. La plus radicale consiste à ne conserver que les meilleurs individus, mais les risques de perte de la diversité sont importants avec ce type de stratégie, aussi des stratégies plus souples ont été développées. L'algorithme de *la roue de la fortune*, par exemple, affecte à chaque individu une probabilité d'être conservé qui est proportionnelle à ses performances. Le meilleur a ainsi de grandes chances d'être conservé, mais des individus de performance plus faible peuvent également rester d'une génération à l'autre, ce qui peut aider à conserver la diversité de la population. Pour plus de détails sur les différents algorithmes de sélection, voir par exemple [11].

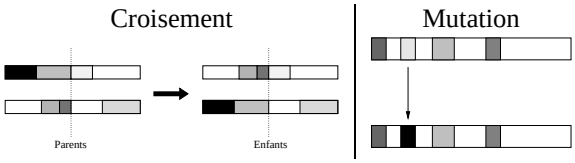

Fig. 2. Opérateurs génétiques

4) Génération d'un nouvel individu: Les nouveaux individus sont des copies d'individus parents sélectionnés en fonction de leurs performances. Deux types d'opérateurs leurs sont appliqués. Tout d'abord le *croisement*, qui va échanger du matériel génétique entre plusieurs individus. Cet opérateur permet d'explorer la zone de l'espace de recherche qui est intermédiaire entre les différents parents. Le second opérateur est la *mutation*, qui va modifier légèrement l'individu de façon aléatoire, explorant ainsi l'espace de recherche dans la zone voisine de l'individu de départ. Ces deux opérateurs sont appliqués avec des probabilités variables selon les algorithmes. Ces probabilités ne sont d'ailleurs pas toujours fixées à l'avance. Ainsi, dans les Stratégies Évolutionnistes [3], [56], l'évolution optimise également le taux de mutation.

Les deux opérateurs ne sont pas toujours utilisés simultannément, certains algorithmes n'en utilisent qu'un seul. Les Stratégies Évolutionnistes, par exemple, n'utilisent, dans leurs premières versions, que l'opérateur de mutation [55].

La génération d'un nouvel individu est la seule phase exploiratoire de l'algorithme, c'est le seul moment pendant lequel une solution est modifiée. Sauf exceptions en effet, les individus n'apprennent pas pendant leur durée de vie: leur comportement est entièrement déterminé par leur génotype et il n'y a pas de phase d'apprentissage intermédiaire[1].

D. Le génotype: le problème du codage de l'information

Le génotype représente l'information manipulée par l'AE au travers des opérateurs génétiques, c'est la représentation d'une solution. C'est un point crucial des AE: comment représenter une solution potentielle ? De ce choix découle directement l'espace de recherche, qui selon les circonstances peut être petit ou de très grande taille, ce qui va jouer directement sur le type de solutions trouvées ainsi que sur le temps de convergence.

Le génotype peut être très simple: une chaîne d'éléments binaires dans les Algorithmes Génétiques de Holland [23], ou bien beaucoup plus complexe et structuré comme les arbres de développement de réseaux de neurones de Gruau [15]. Il sera choisi en fonction de la tâche à résoudre et de ses propriétés dépendront pour beaucoup les performances de l'algorithme évolutionniste.

[1] de nombreux travaux s'orientent cependant dans cette voie [17], [8]

E. Description des principaux AE

Plusieurs éléments varient d'un AE à l'autre: l'algorithme de sélection, le codage des solutions et enfin les opérateurs génétiques utilisés. Ces derniers dépendent fortement du type de codage employé, mais un même codage peut être utilisé avec des opérateurs génétiques différents.

1) Algorithmes Génétiques: Les Algorithmes Génétiques (AG) de Holland [23] utilisent un codage composé d'éléments binaires. Le génotype est un ensemble ordonné de bits qui peuvent être convertis en nombres réels ou en un système cognitif plus complexe, comme un réseau de neurones, par exemple, selon l'étape de développement. Plusieurs possibilités de croisement existent. La plus simple consiste à choisir un emplacement dans l'ensemble ordonné des élements binaires et à échanger les sous-chaînes situées avant et après ce point (cas rerésenté sur la figure 2): c'est le croisement à un point. Il existe également des croisements à plusieurs points. Dans ce cas, on commence par recopier un des parents et, chaque fois que l'on atteint un point, on continue la copie en prenant en compte le génotype de l'autre parent. Les mutations consistent à inverser un ou plusieurs bits de la chaine binaire. Holland a défini plusieurs algorithmes de sélection. La note d'un individu ou son rang peuvent être utilisés directement pour déterminer sa probabilité de générer de nouveaux individus. Une autre méthode, plus radicale, consiste à ne garder d'une génération à l'autre qu'un certain pourcentage de la population, qui survivra et servira de géniteurs pour complèter la nouvelle population, c'est l'algorithme *"steady state"*. [11] contient une description complète des AG ainsi qu'un guide et des exemples permettant de les implémenter facilement.

2) Stratégies évolutionnistes: Les Stratégies Évolutionnistes [3], [56] utilisent directement un vecteur de nombres réels. Le croisement n'est pas utilisé dans les premières versions, mais des variantes plus élaborées inspirées des AG l'utilisent. Les mutations sont des modifications aléatoires gaussiennes des éléments du vecteur dont les paramètres sont également soumis à évolution alors que, pour les Algorithmes Génétiques, les paramètres de mutation sont fixes. Les relations entre les éléments d'un même vecteur sont également prises en compte de façon automatique par l'algorithme, grâce à l'utilisation d'une matrice de covariance également soumise à évolution.

La création d'une nouvelle population consiste à générer λ individus à partir de μ parents et à ne conserver que les μ meilleurs soit parmi les descendants uniquement (alors il est nécessaire que λ soit supérieur à μ, c'est la stratégie (λ, μ)), soit parmi les descendants ainsi que leurs parents (stratégie $(\lambda+\mu)$).

III. Contrôleurs concevables par évolution

Certains types de contrôleurs sont particulièrement adaptés à l'utilisation d'algorithmes évolutionnistes. L'évolution peut être utilisée pour les concevoir entièrement, que ce soient leur

structure ou leurs paramètres, en ne fournissant que les entrées et les sorties.

Nous allons présenter différents types de contrôleurs couramment conçus par méthodes évolutionnistes. Nous les avons séparés en trois groupes: les programmes, les systèmes à base de règles et les réseaux de neurones artificiels.

La section suivante présentera les différentes méthodes d'utilisation de ces controleurs en conjonction avec un algorithme évolutionnistes ainsi que des exemples d'utilisation.

A. Programmes

Mis à part les contrôleurs fondus directement dans le silicium ou implémentés sous forme électronique, les contrôleurs se présentent sous forme de logiciels informatique. Les contrôleurs que nous allons décrire dans cette section sont exprimés directement dans le formalisme d'un programme informatique et sont manipulés en tant que tel par l'algorithme évolutionniste. Autrement dit, on ne passe pas par une métaphore intermédiaire, on manipule directement le programme.

Parmi les différents formalismes existants, deux ont été utilisés en conjonction avec des algorithmes évolutionnistes: les automates à états finis, la programmation fonctionnelle.

1) Automates à états finis: Les automates à états finis sont des programmes représentables sous la forme de graphes dont les noeuds sont des états et les arêtes des transitions conditionnelles auxquelles sont associées une action.

Le passage d'un état à un autre est régi par des lois de transition possèdant une partie condition, qui doit être remplie pour que la transition soit possible, et une partie action, qui est exécutée lors de l'exécution de la transition (voir figure 3 pour un exemple). À chaque pas de temps, une seule transition est effectuée.

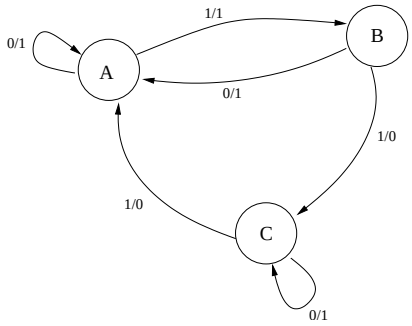

Fig. 3. Exemple d'automate à états finis avec 3 états: A, B et C. Le système dispose d'une entrée et une sortie. Les lois de transition sont de type <entrée>/<sortie>. Si le programme est dans l'état B, il passera à l'état A si l'entrée vaut 1, tout en émettant la valeur 1 en sortie, sinon il passera à l'état C tout en émettant la valeur 0 en sortie.

2) Programmation fonctionnelle: La programmation fonctionnelle est un formalisme dans lequel tout est appel de fonctions. Ce type de programme peut être représenté sous forme d'arbres dont les noeuds sont des instructions. La racine est la fonction principale et les feuilles en sont les paramètres. Ce type de formalisme a été utilisé dans la *programmation génétique*, que nous présenterons à la section IV-C.2.

De tels programmes peuvent être utilisés directement pour contrôler le fonctionnement de l'agent, mais ils peuvent aussi définir l'étape intermédiaire de construction du système de contrôle du robot. En effet, un programme génétique peut être utilisé pour contrôler le développement d'un réseau de neurone [15], comme nous le verrons à la section sur les réseaux de neurones, ou encore d'un automate à états finis [5].

B. Systèmes à base de règles

Une autre possibilité pour concevoir un contrôleur de robot consiste à décomposer le comportement souhaité en comportements élémentaires, qui ne seront appliqués que si certains critères sont remplis. Cela nous conduit à concevoir des systèmes composés de *règles* conditions-actions. Ce principe est à l'origine de plusieurs types de systèmes de contrôle en fonction de la nature des conditions et du type d'action possible.

1) Règles SI ... ALORS ...:

a) Les règles de production: sont utilisées dans un contexte numérique sur des problèmes de robotique. Grefenstette et al. ont ainsi utilisé dans leur système baptisé SAMUEL [14] des règles de type:

SI capteur3 est dans [35,45] ET $sonar2 < 20$ *ALORS* $tourner = -24$ *(force 0.8)*

Le système est composé d'un ensemble de ces règles qui sont appliquées si la partie condition est remplie. La force de la règle, optimisée par l'algorithme évolutionniste dans cet exemple, sert à lever des conflits possibles entre règles concurrentes.

2) Classeurs: Les *classeurs* définissent un formalisme précis pour les systèmes à base de règles de type <*condition*>-<*action*>. Ces règles sont inspirées de travaux de Holland liés aux algorithmes génétiques, c'est pourquoi, généralement, les règles utilisées ont une partie condition et une partie action binaires. Elles sont de la forme:

$1011\#\#10\#1\#00 -> 1001011$

La partie gauche est la condition de la règle. Le symbole $\#$ correspond à un joker, qui indique de ne pas prendre en compte l'entrée correspondante. La partie droite est la partie action.

3) Règles floues:

a) Les systèmes flous: ou systèmes à base de *logique floue*[68], sont composés de règles dont la partie condition fait intervenir des ensembles flous. Les ensembles flous sont des ensembles dont la fonction d'appartenance[2] n'est pas à valeurs binaires, mais à valeurs réelles et variables entre 0 et 1. La logique floue n'est ainsi pas uniquement composée de prédicats vraix ou faux, mais aussi de prédicats de valeurs intermédiaires. Elle est utilisée, par exemple, pour formaliser des concepts abstraits comme "grand", "petit" ou "moyen". Parlant d'êtres humains, par exemple, 1m70 commencera à être "grand", mais moins que 1m80 ou 1m90, voir figure 4.

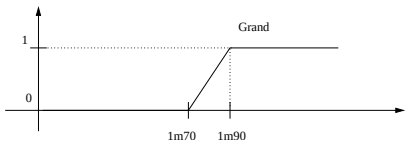

Fig. 4. Exemple de concept flou d'un "grand" humain. En abscisse, taille, en ordonnée, appartenance à l'ensemble flou.

L'appartenance à l'ensemble flou, aussi appelée *crédibilité*, peut être utilisée pour déterminer un poids à attribuer à la règle. L'action à effectuer sera choisie parmi les actions possibles

[2]La fonction d'appartenance d'un ensemble est une fonction booléenne associant à chaque élément la valeur "vrai" si l'élément fait partie de l'ensemble ou faux sinon.

en prenant cette pondération en compte. Il est, par exemple, possible d'appliquer l'action qui résulte de la moyenne des actions possibles pondérée par la crédibilité de chacune d'elles. Le passage d'une règle à une autre se fait alors de façon très souple.

b) Les règles de Mamdani: [42] sont des règles à plusieurs entrées et à sortie unique de la forme:

SI x_1 est A_1 ET ... ET x_n est A_n ALORS y est B

Où A_i et B sont des ensembles flous. En général, les systèmes robotiques nécessitent cependant une sortie continue, aussi ce type de contrôleur nécessite-t'il une étape délicate de *dé-flouification*, prenant en compte la sortie de chaque règle ainsi que sa crédibilité. Cette étape est relativement complexe et constitue la principale faiblesse de ce type de système. Cette méthode a été utilisée pour contrôler une voiture dans [10].

c) Règles TSK.: Takagi, Sugeno et Kang ont proposé des règles différentes, appelées *règles TSK* [60], inspirées des règles de Mamdani, mais avec une conclusion non floue qui permet d'éviter l'étape complexe de dé-flouification.

SI x_1 est A_1 ET ... ET x_n est A_n ALORS $y = \sum_i k_i \times x_i$

où les A_i sont des ensembles flous. La conclusion de la règle est la somme des entrées pondérées par des coefficients k_i. À chaque pas de temps, le système calcule la conclusion de chaque règle et la sortie globale est alors la somme des sorties des règles, pondérées par leur crédibilité.

C. Réseaux de neurones

1) Le neurone biologique: Le *neurone* biologique est une cellule ramifiée, recevant des informations provenant d'autres neurones grâce à ses *dendrites* et envoyant des signaux grâce à son *axone*. La connexion entre plusieurs neurones est réalisée par des *synapses* qui assurent la transmission d'un signal d'une cellule à l'autre par des moyens chimiques ou électriques. Les synapses disposent d'une efficacité de transmission de l'information variable: un neurone peut ainsi avoir plus ou moins d'influence sur les neurones auxquel il est connecté. Cette influence peut varier au cours du temps en fonction de mécanismes d'apprentissage.

Le signal transmis le long des axones, synapses puis dendrites est un *potentiel d'action*, une petite décharge électrique véhiculée en propageant une différence de potentiel entre l'intérieur et l'extérieur de l'axone ou de la dendrite. Le passage du potentiel d'action du neurone pré synaptique au neurone post synaptique s'effectue grace à des *neurotransmetteurs* émis par la terminaison synaptique du neurone pré synaptique et captés par la membrane du neurone postsynaptique.

2) Le neurone formel: Le *neurone formel* de Mac Culloch et Pitts [43] est une version simplifiée du modèle biologique. C'est une unité de calcul recevant des entrées et calculant une valeur de sortie. L'*activation* du neurone est la somme des entrées reçues et sa valeur de sortie est obtenue en appliquant une *fonction de transfert* à l'activation. Les fonctions de transfert utilisées étaient initialement de simples fonctions-seuil manipulant des variables booléennes mais, à présent, ce sont souvent des fonctions sigmoïdes à valeurs réelles, comme par exemple $f(x) = tanh(x)$ ou encore $f(x) = \frac{1}{1+\exp^{-x}}$. La valeur de sortie est alors interprétée comme la fréquence de décharge du neurone.

Un *réseau de neurones* [54] est un graphe orienté dont chacun des noeuds est un neurone formel. À chaque lien est associé un *poids synaptique*, reflet de l'efficacité synaptique du neurone biologique.

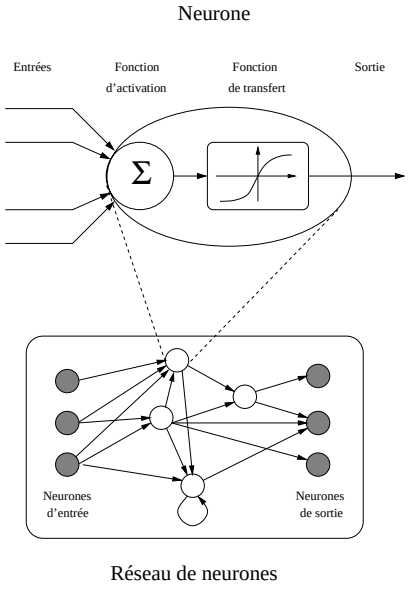

Fig. 5. Exemple de neurone et de réseau de neurones.

D'autres modèles plus proches du neurone biologique existent, notamment les neurones à décharge, mais leur étude dépasse largement le cadre de cet article. Pour plus de détails, voir par exemple [41].

La structure de réseau la plus étudiée et la plus utilisée jusqu'à présent est le *perceptron* [53], ou réseau à couche. Dans ce modèle, les neurones sont disposés par couche. Les neurones d'une couche reçoivent leurs entrées uniquement des neurones de la couche précédente. Un perceptron à une couche cachée [3] peut approximer n'importe quelle fonction continue s'il y a suffisamment de neurones cachés et en utilisant une fonction de transfert continue et différentiable [26], [7]. Avec deux couches cachées, un perceptron multi-couche peut approximer n'importe quelle fonction calculable [6], [37].

D. Contrôleurs P, PI, PD et PID

Lorsque le contrôle se ramène au maintien d'une variable à une valeur cible et que le domaine de fonctionnement est linéaire, il est possible d'appliquer des correcteurs de type Proportionnel (P), Porportionnel Intégral (PI), Proportionnel Dérivé (PD) ou encore Proportionnel Intégral Dérivé (PID). Le principe consiste alors à fournir en entrée du système l'erreur sur la variable à maintenir, autrement dit l'écart entre la valeur souhaitée et la valeur réelle, ainsi que, selon les cas, sa dérivée et son intégrale. La sortie du système vaut alors:

$$\begin{aligned}
\text{P:} \quad & s(t) = k_p\epsilon(t) \\
\text{PD:} \quad & s(t) = k_p\epsilon(t) + k_d\frac{d\epsilon(t)}{dt} \\
\text{PI:} \quad & s(t) = k_p\epsilon(t) + k_i\int\epsilon(t)dt \\
\text{PID:} \quad & s(t) = k_p\epsilon(t) + k_i\int\epsilon(t)dt + k_d\frac{d\epsilon(t)}{dt}
\end{aligned}$$

Ce type de contrôleur est particulièrement bien adapté au contrôle bas-niveau de manière générale et au contrôle de

[3] c'est-à-dire ayant une couche de neurones entre les entrées et les sorties

robots volants en particulier, dont l'objectif est le maintien de variables d'état à l'intérieur d'un domaine de viabilité[4]. Cette méthode est la stratégie la plus simple utilisée pour résoudre ce type de problèmes. Cependant, ce type de contrôleur n'accepte qu'une entrée et une sortie et ne fonctionne bien que dans un contexte linéaire.

IV. Évolution de contrôleurs

Lors de l'utilisation des algorithmes évolutionnistes pour l'optimisation ou la conception de contrôleurs de robots, il faut se doter d'un moyen d'évaluer le fonctionnement du contrôleur. Pour cela deux méthodes sont possibles: l'utilisation d'une simulation ou l'évolution en direct sur le système réel.

La simulation offre une méthode souple et rapide qui simplifie et accélère les calculs. Cependant, l'utilisation d'une simulation n'est pas toujours possible, car les équations régissant la dynamique du système ne sont pas toujours connues avec précision. De plus, le passage de la simulation à la réalité n'est pas toujours aisé. Pour faciliter ce passage, Jakobi a présenté une solution simple [28]: ne simuler que ce qui est indispensable et ajouter du bruit de façon à éviter que les solutions trouvées n'exploitent des caractéristiques peu robustes de l'environnement.

L'utilisation directe de systèmes réels supprime le besoin d'une simulation, ce qui simplifie la préparation des expériences et diminue également les connaissances nécessaires pour traiter le problème. Cependant, contrairement à des approches par simulation qui peuvent fonctionner en temps accéléré, l'évaluation d'un individu va nécessiter une période de temps importante et incompressible, ce qui va ralentir d'autant le temps de convergence. De plus, l'évaluation en réel peut entrainer des comportements dangereux pour l'intégrité du système, ce qui nécessite la préparation d'un banc de test évitant tout comportement non désiré.

Un compromis entre les deux approches peut cependant être trouvé: il est possible de commencer les calculs sur une simulation simplifiée puis d'achever d'optimiser les individus sur le système réel. On profite ainsi des avantages de la simulation pour accélérer les calculs au début, tout fonctionnant ensuite sur le système réel, lorsqu'un ensemble de solutions convenable a été trouvé, ce qui permet d'achever l'optimisation.

Nous allons à présent présenter l'utilisation de l'évolution pour concevoir et optimiser les contrôleurs que nous avons présentés dans la section précédente.

A. Optimisation de paramètres

Les Algorithmes Génétiques ou les Stratégies Evolutionnistes permettent d'optimiser une fonction. Si on considère une fonction évaluant les performances d'un robot sur un problème donné, ces algorithmes peuvent donc être utilisés pour optimiser les paramètres d'un système de contrôle quelconque, dont la structure a été au préalable fixée manuellement. L'information génétique est alors un vecteur de nombres, qui subira l'effet des opérateurs génétiques de mutation et de croisement. L'évaluation d'un génotype G consistera à tout d'abord positionner les paramètres du système de contrôle du robot en utilisant G et à observer et noter ses évolutions dans différentes conditions. Ces conditions doivent impérativement constituer un échantillon représentatif des conditions que le système devra ensuite affronter. Dans le cas contraire, l'évolution risque de trouver des solutions spécialisées dans des conditions particulières qui ne fonctionneront pas convenablement en dehors de ce champs de fonctionnement.

L'évolution peut ainsi être utilisée pour paramétrer des contrôleurs de type PID, problème souvent résolu de façon empirique, ou pour optimiser des contrôleurs plus spécifiques. Le contrôleur de marche du robot AIBO de Sony dans sa version commerciale a ainsi été optimisé grâce à des méthodes évolutionnistes [25].

B. Systèmes à base de règles

Si les systèmes à base de règles permettent d'inclure la connaissance d'experts dans un contrôleur automatique, leur définition manuelle se révèle rapidement fastidieuse, aussi des méthodes basées sur des algorithmes évolutionnistes ont été utilisées pour automatiser leur conception.

Les systèmes de classeurs peuvent être reliés à l'évolution de deux façons. Un ensemble de règle constituant un contrôleur peut être contenu dans un seul individu, ou bien chaque règle peut constituer un génotype à part entière, le contrôleur est alors la population dans son ensemble. Voir [11] pour plus de détails.

Des systèmes à base de règles floues de type TSK ont été conçus par évolution pour contrôler un hélicoptère [22]. Ces règles associent une condition portant sur des ensembles flous à une fonction linéaire d'une ou plusieurs variables. L'algorithme procéde par étapes depuis un contrôleur composé d'une seule règle toujours valable avec un seul terme dans la partie conclusion, vers un système plus élaboré par divisions successives de l'espace des variables et/ou par ajouts de termes dans la partie conclusion. Les coefficients de la partie conclusion sont évolués en utilisant une Stratégie Évolutionniste. Les performances obtenues avec ce type de contrôleur ont été comparées à celles d'autres contrôleurs [57]. Il s'est avéré que le contrôleur à base de règles floues avait des performances similaires à ceux qui ont été conçus en utilisant des méthodes inspirées du contrôle robuste: ils avaient une grande robustesse à l'incertitude et au bruit, mais leur domaine de fonctionnement restait relativement restreint autour du vol stationnaire. Le contrôle nonlinéaire donnait des contrôleurs utilisables sur un domaine de fonctionnement plus vaste, mais nécessitait une connaissance beaucoup plus précise du système et était plus sensible aux perturbations. Cependant, à chaque domaine de vol particulier peut correspondre un ensemble de règles de contrôle adaptées, aussi, si la subdivision du domaine de vol est correcte et si les règles correspondantes soient bien configurées, un contrôleur à base de règles devrait théoriquement pouvoir se comporter de façon efficace sur un grand domaine de fonctionnement. Cependant, partitionner le domaine de vol est un problème complexe avec un espace de recherche très important, ce qui rend sa découverte délicate sans utiliser d'heuristiques ou de connaissances a priori. Cela explique sans doute les faibles performances de l'algorithme d'Hoffmann et al. sur ce point.

[4] appelé domaine de vol en aéronautique

C. Programmes

1) Automates à états finis: Fogel, un des pioniers des approches évolutionnistes, a fait évoluer des automates à états finis [38] pour apprendre à prévoir le prochain symbole dans une séquence fournie en entrée du système.

Landau et Picault ont utilisé un algorithme génétique pour faire évoluer des automates à états finis [36]. Un vecteur de valeurs binaires est soumis à évolution et subit ensuite une étape de développement pendant laquelle un automate va être généré puis utilisé pour contrôler un agent. Le développement consiste à traduire les éléments de la chaîne binaire en jetons qui sont des instructions de développement, empilées immédiatement après création. Après traduction, les instructions empilées sont exécutées pour obtenir l'automate. Elles peuvent manipuler la pile, ou créer une connexion ou un noeud. Les paramètres dont elles ont éventuellement besoin sont pris dans la pile et dépilés après usage.

Brave [5] a utilisé la programmation génétique, que nous présenterons à la section suivante, pour le développement d'automates à états finis.

2) Programmation génétique: Koza [34] a fait évoluer des programmes en utilisant un formalisme fonctionnel basé sur des arbres[5]. Les programmes manipulés sont des fonctions dont l'enchainement, ainsi que l'éventuel paramétrage, sont définis par évolution. L'expérimentateur doit fournir l'ensemble des *symboles terminaux* susceptibles d'être utilisés, ainsi que l'ensemble des *fonctions primitives* dont le programme sera composé. Les symboles terminaux sont les variables sur lesquelles porteront les calculs et les fonctions primitives sont les éléments de base permettant de construire les programmes. Pour résoudre des problèmes d'approximation de fonctions de type $z = f(x, y)$, par exemple, l'ensemble des symboles terminaux sera $\{x, y\}$ et l'ensemble des fonctions primitives pourra être $\{+, -, \times, /\}$. Le type de fonction obtenu sera de la forme (expression LISP):

$$(/\ (\times\ (+\ x\ 4.1)\ (-\ y\ 5.2))\ (\times\ (-\ x\ 1.3)\ (+\ y\ 7.1)))$$

ce qui correspond à la fonction suivante:

$$\frac{(x + 4.1) \times (y - 5.2)}{(x - 1.3) \times (y + 7.1)}$$

La figure 6 représente cette fonction sous la forme d'un arbre.

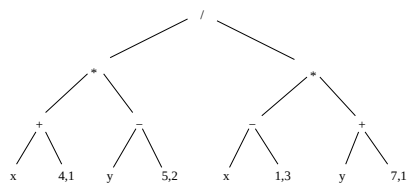

Fig. 6. Exemple de représentation de programme sous forme d'arbre.

Les programmes manipulés dans cette version simple de la programmation génétique ne permettent pas de réutiliser des fonctions à plusieurs endroit dans le programme, une possibilité couramment utilisée en programmation classique. Pour remrédier à ce manque, Koza [35] a ajouté la possibilité de définir automatiquement des fonctions susceptible d'etre appelées par le programme principal. Ces fonctions, appelées ADF pour *Automatically Defined Functions*, sont également soumises à évolution. Le programme est alors composé d'une ou plusieurs de ces fonctions auxquelles s'ajoute un programme principal, appelant ou non ces fonctions.

Les populations utilisées sont initialisées avec des milliers, ou même des millions, d'individus.

Le croisement permet d'échanger des sous-arbres entre programmes. Le croisement des programme ci-dessous selon les marques indiquées:

$$(/\ (\times\ (+\ x\ 4.1)\ (-\ y\ 5.2))\ (\times\ (-\ x\ 1.3)\ (+\ y\ 7.1)))$$

$$(\times\ (/\ (-\ x\ 3.2)\ (\times\ y\ 9.5))\ (/\ (+\ x\ 4.4)\ (/\ y\ 2.8)))$$

donne le résultat suivant:

$$(\times\ (\times\ (-\ x\ 1.3)\ (+\ y\ 7.1))\ (\times\ y\ 9.5))\ (/\ (+\ x\ 4.4)\ (/\ y\ 2.8)))$$

La mutation remplace un sous-arbre par un autre sous-arbre généré aléatoirement.

L'utilisation des ADF nécessite l'usage de 6 nouveaux opérateurs qui permettent la duplication, l'ajout ou la suppression de fonctions ou de paramètres de fonction.

D. Réseaux de neurones

Les travaux étudiant les liens possibles entre évolution et réseaux de neurones forment un domaine à l'heure actuelle très actif [66] avec de nombreuses applications en robotique [45]. L'évolution peut être utilisée à différents niveaux dans la conception d'un réseau de neurones et les problématiques posées peuvent largement dépasser le cadre de l'optimisation de paramètres tel qu'il est abordé dans les Algorithmes Génétiques ou les Stratégies Évolutionnistes.

1) Optimisation des poids uniquement: Une première possibilité consiste à figer la structure d'un réseau et à faire évoluer les poids uniquement dans une approche semblable à celle décrite dans la section IV-A.

La structure du réseau est déterminée à partir d'heuristiques et l'évolution se charge ensuite de trouver les poids adaptés [64], [62]. D'autres algorithmes, tels la rétropropagation de gradient[61], peuvent être utilisés dans ce contexte. Cependant, ils nécessitent de connaitre la solution idéale de façon à calculer l'erreur et ils ne sont adaptés qu'à des structures particulières de type perceptron. Des adaptations de ce type d'algorithmes permettent de réaliser un apprentissage sur certains types de structures récurrentes: rétropropagation temporelle, rétropropagation récurrente, mais l'évolution est à notre connaissance le seul algorithme applicable à n'importe quelle structure neuronale. Si les structures considérées se ramènent souvent à une variante de perceptron, des comportements dynamiques plus complexes peuvent nécessiter des structures récurrentes pour lesquelles aucun algorithme d'apprentissage ne convient. Dans ce cas, l'évolution est le seul algorithme disponible.

Les biologistes construisent ainsi des modèles de systèmes nerveux complexes, issus de leurs observations. Cependant, si la structure est identifiable, le paramétrage du modèle neuronale pose des problèmes d'apprentissage délicats. L'évolution peut

[5] il a utilisé pour cela le langage LISP

alors fournir une réponse efficace et permettre aux chercheurs de se concentrer dans un premier temps sur la modélisation de la structure pour vérifier leurs hypothèses avant de se concentrer éventuellement sur l'apprentissage. Ijspeert et al. [27] ont utilisé l'évolution pour améliorer un modèle neuronal de locomotion de lamproie dont les paramètres avaient été positionnés empiriquement. Ils ont ainsi montré que l'évolution peut aider à la conception de modèles biologiques en exploitant les connaissances acquises par l'observation et en comblant les lacunes grâce aux capacités de recherche par essais et erreurs de l'algorithme.

2) Conception de la structure et des paramètres du réseau: La génération de la structure d'un réseau de neurones demeure un problème ouvert, résolu généralement de manière empirique avec des heuristiques simples. L'évolution peut apporter une solution à ce problème en travaillant à la fois sur la structure et les paramètres du réseau.

L'évolution des poids du réseau peut se faire simplement en utilisant des algorithmes évolutionnistes classiques comme les *algorithmes génétiques* ou les *stratégies évolutionnistes* évoquées à la section précédente. L'évolution de la structure d'un réseau nécessite cependant un codage particulier, représentatif du graphe de connexion du réseau de neurones.

Pour être efficace, un codage nécessite plusieurs propriétés énoncées par Gruau [16]. Nous retiendrons essentiellement:

- **complétude**, tout réseau doit pouvoir être codé par un génome
- **fermeture** tout génome doit représenter un réseau
- **modularité**, un sous-réseau présent en plusieurs exemplaires dans le réseau complet doit pouvoir apparaitre qu'une seule fois dans le génome

Propriétés auxquelles ont peut ajouter les suivantes:

- **dynamique**, les dimensions de l'espace de recherche ne doivent pas être imposées, le codage doit permettre une recherche incrémentale
- **compatibilité avec des opérateurs génétiques**
- possibilité de spécifier certaines **entrées et sorties** du réseau (vers capteurs et effecteurs)

Le principal problème réside dans l'efficacité des opérateurs génétiques. La mutation doit effectuer une recherche statistiquement locale, un individu muté doit rester statistiquement proche de sa version initiale pour assurer une exploration fine de l'espace de recherche. Cependant, il doit rester possible d'exercer des sauts importants afin d'éviter de rester bloqué dans des extréma locaux. Le croisement doit permettre l'échange de blocs fonctionnels.

Deux types de codage sont possibles: un codage direct et un codage indirect. Un codage direct défini une représentation bijective simple du réseau : chaque neurone et chaque connexion y est représenté de façon individuelle. Un codage indirect est une représentation plus compacte, plus compressée du réseau nécessitant une étape de développement [32].

3) Codage direct:

a) Une représentation génétique: adaptée peut grandement faciliter la définition des opérateurs génétiques. Fullmer et Miikkulainen [9] ont utilisé un codage proche du génome biologique. C'est une suite d'entiers, semblables aux codons naturels, qui réprésentent les paramètres d'un noeud du réseau: un marqueur de début, une clé d'identification, la valeur de sortie initiale du réseau et une liste connexions de type clé de la source-poids de la connexion et enfin un marqueur de fin. Cette représentation facilite la définition des opérateurs génétiques, qui peuvent être identiques à ceux d'un algorithme génétique simple: les mutations changent un codon, le croisement échange des sous-chaines de codons entre chromosomes.

b) Le croisement: doit, dans des algorithmes évolutionnistes, permettre l'échange de blocs entre chromosomes. L'idéal serait de pouvoir échanger des blocs fonctionnels de façon à ce que le résultat du croisement soit susceptible de profiter de caractéristiques différentes issues de chacun de ses parents, ce qui est loin d'être évident avec un codage direct. La simple définition d'un opérateur de croisement n'est pas non plus un problème simple, à tel point que dans de nombreux cas, le croisement n'est même pas utilisé [50], [1], [67]. Pujol et Poli [51] utilisent une représentation sous forme de grille du réseau. Deux noeuds sont choisis et les liens, représentés sous forme relative (connexions au neurone de la ligne précédente (L-1), 2 colonnes après (C+2), par exemple), sont transposés d'un des noeuds à l'autre. Les auteurs n'ont fait évoluer que des réseaux de type perceptron, mais cette approche doit pouvoir se transposer à des structure quelconque. La principale limitation réside cependant dans l'obligation d'utiliser une grille de taille fixe, imposée initialement. *NEAT* [59] apporte une solution complètement différente et originale à ce problème. Chaque connexion dispose d'un numéro d'innovation, attribué de façon unique lors de l'apparition de la connexion. Ce numéro d'innovation, permet de définir un opérateur de similarité : sont similaires les chromosomes disposant d'un certain nombre de connexions ayant le même numéro d'innovation. Cette similarité permet de faire la correspondance entre des connexions et d'échanger l'information génétique de connexions historiquement issue d'une même mutation, connexions supposées exercer une fonction comparable dans les différents réseaux. La définition d'une similarité permet également de définir des niches écologiques permettant de protèger les nouvelles structures en utilisant le partage de fitness explicite (*explicit fitness sharing* en anglais) [12].

4) Codage indirect: Dans les méthodes décrites précédemment, la correspondance entre un chromosome et le réseau de neurones est directe et la traduction immédiate. Cependant, pour résoudre les problèmes cités ci-dessus ainsi que d'autres problèmes, comme le passage à l'échelle, il est intéressant de considérer des codages pour lesquels la correspondance génotype-phénotype n'est plus immédiate, mais nécessite une phase de développement.

Gruau a prouvé l'efficacité d'approches indirectes par rapport à des approches basées sur un codage direct [18], cependant de récents travaux [59] suggèrent qu'un codage direct peut être plus efficace qu'un codage indirect, montrant ainsi qu'au moins pour des réseaux de taille réduite, la différence entre codage direct et indirect n'est pas aussi nette que Gruau l'a suggéré.

a) Règles de réécriture: Un graphe peut être obtenu à partir d'une cellule initiale par exécution d'un programme ou de règles de production.

Lindenmayer a développé une classe de fractales appelées L-Systems [39] pour modéliser le développement des plantes. Un L-Système est un système de réécriture de chaines en par-

allèle. Il est composé d'un ensemble de règles de production qui définissent des transformations de chaînes élémentaires en d'autres chaînes élémentaires. Kitano [29] et Boers [4] ont exploité ce principe pour faire évoluer des réseaux de neurones.

b) Programmation génétique: La programmation génétique [34] a inspiré de nombreuses méthodes de développement de réseaux de neurones dont la plupart sont inspirées du *Codage Cellulaire* de Gruau [16], [63], [15].

Le codage cellulaire part d'un réseau simple contenant des cellules de développement reliées aux entrées et sorties. Ces cellules exécutent leur programme de développement [6]. Lorsque l'exécution atteint un symbole terminal, la cellule se transforme en neurone. Chaque cellule de développement dispose d'une copie du programme de développement ainsi que d'une tête de lecture qui pointe sur la prochaine instruction à exécuter. Les instructions de développement positionnent les différents paramètres du futur neurone et complexifient la structure par des instructions de division cellulaire. Ces instructions de division sont le point clé de cette méthode. Il en existe différents types. Elles répartissent les connexions entre cellule mère et cellule fille de différentes manières: elles peuvent les recopier dans la nouvelle cellule, transmettre les connexions sortantes et créer un lien entre la cellule mère et la cellule fille...

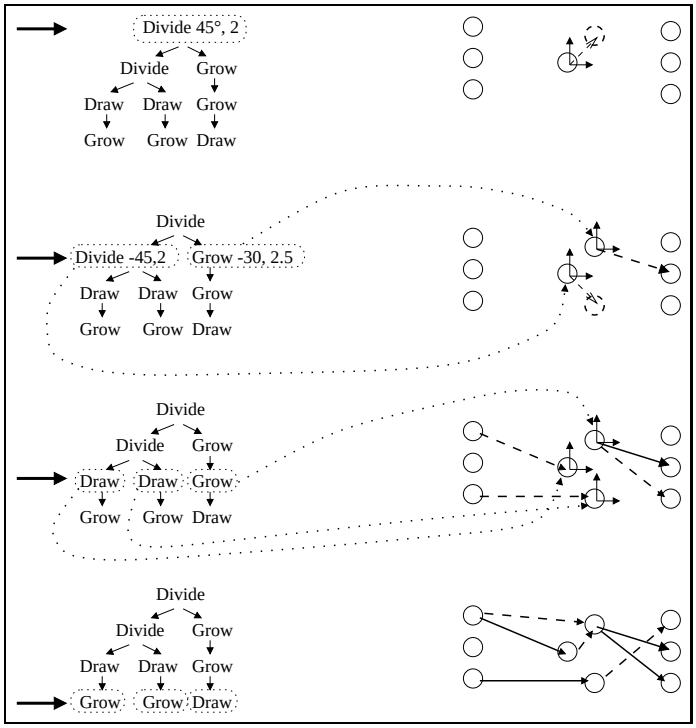

Fig. 7. Exemple de programme SGOCE. Les instructions à la même profondeur dans l'arbre du programme sont exécutées simultanément. Ce programme crée une réseau de trois neurones intermédiaires en partant d'une cellule initiale

SGOCE [32], [33], [31] est une méthode inspirée du codage cellulaire dans laquelle les cellules de développement sont situées dans un substrat à 2, voire 3, dimensions. Cela permet de créer des connexions depuis ou vers des neurones par des opérations autres que des divisions: il est possible de viser une zone du substrat et d'attirer ou de créer une connexion depuis ou vers la cellule la plus proche de cette zone (figure 7).

[6] Gruau utilise un programme génétique

L'*Edge Encoding* [40] est une autre variante du codage cellulaire dans laquelle le développement n'est plus basé sur des cellules de développement, mais sur des liens, des connexions entre cellules. Ce sont ces liens qui exécutent le programme de développement.

c) Neurogénèse: Si les méthodes évoquées précédemment restent relativement éloignées de la réalité biologique, certaines essaient d'y être beaucoup plus fidèles et mettent en place un développement dépendant d'émissions de médiateurs chimiques. L'évolution a alors pour tâche de régler ces émissions et de choisir le comportement adapté en fonction des concentrations locales de différents agents chimiques, voir par exemple [2], [47] et [20].

V. L'ÉVOLUTION COMME MÉTHODE DE CONCEPTION

A. Avantages

Les méthodes évolutionnistes disposent de plusieurs avantages sur les autres méthodes d'apprentissage. Tout d'abord, le retour d'information requis est très faible. En effet, le seul retour nécessaire est la note finale attribuée aux performances d'une solution potentielle. Cette note peut-être très qualitative et ne contenir aucune information concernant les moyens disponibles pour résoudre la tâche imposée. La distance parcourue pendant un temps donné [33] suffit pour noter un robot marcheur. Cette évaluation est très qualitative et ne nécessite pas d'entrer dans les détails du contrôleur. Aucune consigne n'est donnée sur le mouvement individuel des pattes, l'évolution doit trouver la solution la plus efficace.

Comme beaucoup d'autres algorithmes, l'évolution dispose de paramètres qui doivent être positionnés convenablement pour que l'algorithme fonctionne avec toute son efficacité. Cependant, l'évolution, contrairement aux autres méthodes, peut optimiser ces paramètres en cours de fonctionnement [21].

La fonction que l'évolution cherche à optimiser n'a pas besoin de propriétés mathématiques particulières. Il suffit de pouvoir connaitre sa valeur en un point, nul besoin de dérivée par exemple contrairement aux approches basées sur la remontée de gradient. Cela rend ces méthodes plus souples et applicables à des problèmes sur lesquels on ne dispose que de peu de connaissances. En effet, nul besoin d'une mise en équation complète du système, il suffit de pouvoir classer les solutions proposées par l'évolution.

Il n'est pas nécessaire non plus de connaitre la solution idéale ou même simplement des exemples, contrairement aux algorithmes supervisés basés sur la rétropropagation de l'erreur.

Les méthodes évolutionnistes n'ont donc besoin que de peu d'informations, l'algorithme en soi ne repose pas sur des propriétés particulières à un problème. Ce faible retour d'information nécessaire rend les méthodes évolutionnistes utilisables dans des applications très variées, qu'elles soient dans le domaine de l'ingénierie, comme la conception d'armatures métalliques pour l'industrie [46], ou dans le domaine des Systèmes Adaptatifs ou de la Vie Artificielle, comme la conception de la morphologie d'une créature virtuelle [58] ou plus généralement la robotique évolutionniste [49], [13], [19], [45], [44].

L'évolution ne fournit pas une solution unique mais un ensemble de solutions. En effet, en fin de calcul, la population

finale dans son ensemble peut être prise en compte plutôt que le meilleur individu uniquement. Dans la résolution de problèmes multi-critères et avec un algorithme de sélection adapté, cela permet d'obtenir un échantillonage du front de Pareto qui représente l'ensemble des meilleurs compromis possibles [24].

L'évolution a été ici présentée comme un outil de conception de systèmes de controle, cependant, elle peut également intervenir pendant la conception de la morphologie d'un robot [58].

B. Inconvénients

L'inconvénient principal découle directement des points cités ci-dessus: le retour d'information étant relativement faible et l'algorithme étant très général, les mécanismes à base d'évolution seront souvent plus lents à converger que les algorithmes optimisés pour un problème particulier qui vont exploiter au maximum les connaissances disponibles. Kitano a montré qu'un Algorithme Génétique était plus lent qu'un algorithme de rétropropagation pour apprendre les poids d'un réseau de type feed-forward à trois couches [30] sur un problème particulier. Cependant, dans plusieurs cas, ils se sont révélés plus efficaces que d'autres algorithmes classiques d'apprentissage [48], [65], [52].

L'évolution est une méthode de recherche par essais et erreurs. Elle ne pourra pas rivaliser avec des méthodes classiques sur des problèmes, soit dont on connait l'expression de la solution, soit dont on connait et utilise des propriétés particulières. C'est un algorithme général de recherche qui n'exploite aucune connaissance particulière et qui donc ne profite pas d'éventuelles propriétés intéressantes d'un problème particulier. A priori, l'évolution ne pourra donc pas rivaliser avec des méthodes conventionnelles de contrôle dans leur zone de fonctionnement habituelle. En revanche, elle deviendra intéressante lorsque l'on s'éloignera du domaine de fonctionnement de ces méthodes ou lorsque le fonctionnement du système ne sera pas connu.

VI. Conclusions

Un oiseau ne connait pas les équations de l'aérodynamique, pourtant il est capable de voler. Son corps, particulièrement adapté au vol, et son comportement ont été conçus au fil des générations par sélection successives des individus les plus performants. La nature a exploité une stratégie basée sur l'apprentissage par essais et erreurs qui peut être exploitée par les ingénieurs pour résoudre les problèmes auxquels ils sont confrontés.

Si la connaissance précise des équations d'un système demeure fondamentale pour maîtriser pleinement un processus, les algorithmes évolutionnistes dont nous avons présenté ici les principes, ainsi que quelques exemples d'utilisation, peuvent fournir un outil d'exploration intéressant dans des domaines partiellement ou totalement inconnus. Ils peuvent aider à étudier le système en trouvant des solutions astucieuses exploitant des caractéristiques spécifiques dont les justifications théoriques peuvent n'apparaître que bien plus tard.

References

[1] Peter J. Angeline, Gregory M. Saunders, and Jordan B. Pollack. An evolutionary algorithm that constructs recurrent networks, 1994.

[2] J. C. Astor. A developmental model for the evolution of artificial neural networks: Design, implementation, evaluation. Master's thesis, University of Heidelberg, September 1998.

[3] Thomas Bäck and Hans-Paul Schwefel. Evolution strategies i: Variants and their computational implementation. In *Genetic Algorithms in Engineering and Computer Science, Proc. First Short Course EUROGEN-95*. 1995.

[4] E. J. W. Boers, H. Kuiper, B. L. M. Happel, and I. G. Springhuizen-Kuiper. Designing modular artificial neural networks. Technical report, 1993.

[5] Scott Brave. Evolving deterministic finite automata using cellular encoding. In David B. Fogel John R. Koza, David E. Goldberg and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the first annual conference*, Stanford university, 28-31 july 1996. Cambridge, MA: MIT Press.

[6] G. Cybenko. Continuous valued neural networks with two hidden layers are sufficient. Technical report, Departement of computer science, Tufts University, Medford, MA, 1988.

[7] G. Cybenko. Approximation by superposition of a sigmoidal function. *Mathematics of Control, Signal and Systems*, (2):303–314, 1989.

[8] D. Floreano and J. Urzelai. Evolution of plastic control networks. *Autonomous Robots*, 11:311–317, 2001.

[9] Brad Fullmer and Risto Miikkulainen. Using marker-based genetic encoding of neural networks to evolve finite-state behaviour. In *Proceedings of the first European Conference on Artificial Life (ECAL-91*, Paris, 1991.

[10] Philippe Garnier and Thierry Fraichard. A fuzzy motion controller for a car-like vehicle. Technical Report 3200, INRIA, 1997.

[11] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[12] D.E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 148–154, San Francisco, 1987. CA: Morgan Kaufmann.

[13] T. Gomi and A. Griffith. Evolutionnary robotics – an overview. In IEEE Press, editor, *Proceedings of the IEEE Third International Conference on Evolutionnary Computation*, 1996.

[14] John Grefenstette and Alan Schultz. An evolutionary approach to learning in robots. In *Machine Learning Workshop on Robot Learning*, 1994.

[15] F. Gruau. *Synthèse de Réseaux de Neurones par Codage Cellulaire et Algorithmes Génétiques*. PhD thesis, ENS Lyon, Université Lyon I, 1994.

[16] Fédéric Gruau. Cellular encoding of genetic neural networks. Technical Report 9221, Laboratoire de l'informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 1992.

[17] Frédéric Gruau and Darrell Whitley. Adding learning to the cellular development of neural networks: Evolution and the baldwin effect, 1993.

[18] Frederic Gruau, Darrell Whitley, and Larry Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89, Stanford University, CA, USA, 28–31 1996. MIT Press.

[19] A. Guillot and J.A. Meyer. The animat contribution to cognitive systems research. *Cognitive Systems Research*, 2(2):157–165, 2001.

[20] Inman Harvey. *The Artificial Evolution of Adaptive Behavior*. PhD thesis, University of Sussex, 1993.

[21] Robert Hinterding, Zbigniew Michalewicz, and Agoston E. Eiben. Adaptation in evolutionary computation: A survey. In *4th IEEE International Conference on Evolutionary Computation*, 1997.

[22] Frank Hoffmann, Tak John Koo, and Omid Shakernia. Evolutionary design of a helicopter autopilot. In *3rd On-Line World Conf. on Soft Computing (WSC3)*, 1998.

[23] J.H. Holland. *Adaptation in Natural and Artificial Systems*. MI: University of Michigan Press, 1975.

[24] Jeffrey Horn and Nicholas Nafpliotis. Multiobjective Optimization using the Niched Pareto Genetic Algorithm. Technical Report IlliGAl Report 93005, Urbana, Illinois, USA, 1993.

[25] G.S. Hornby, S. Takamura, J. Yokono, O. Hanagata, T. Yamamoto, and M. Fujita. Evolving robust gaits with aibo. In *IEEE International Conference on Robotics and Automation*, pages 3040–3045, 2000.

[26] K. Hornik. Some new results on neural network approximation. *Neural Networks*, (6):1069–1072, 1993.

[27] A.J. Ijspeert, J. Hallam, and D. Willshaw. Evolving swimming controllers for a simulated lamprey with inspiration from neurobiology. *Adaptive Behavior*, pages 151–172, 1999.

[28] N. Jakobi, P. Husbands, and I. Harvey. Noise and the reality gap: the use of simulation in evolutionary robotics. In F. Moran, A. Moreno, J. Merelo, and P. Chancon, editors, *Advances in Artificial Life: Proceedings of the Third European Conference on Artificial Life*. Berlin: Springer Verlag, 1995.

[29] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476, 1990.

[30] H. Kitano. Empirical studies on the speed of the convergence of neural network training using genetic algorithms. In *Proceedings of the Eighth Ntat'l Conf on AI (AAAI-90)*, pages 789–795. MIT Press, Cambridge, MA, 1990.

[31] J. Kodjabachian. *Développement et évolution de réseaux de neurones artificiels*. PhD thesis, Université Pierre et Marie Curie, 1998.

[32] J. Kodjabachian and J.A. Meyer. Evolution and development of control architectures in animats. *Robotics and Autonomous Systems*, 16:161–182, 1995.

[33] J. Kodjabachian and J.A. Meyer. Evolution and development of neural networks controlling locomotion, gradient-following, and obstacle-avoidance in artificial insects. *IEEE Transactions on Neural Networks*, 9:796–812, 1997.

[34] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[35] John R. Koza. *Genetic Programming II: Automatic Discovery of reusable Programs*. MIT Press, 1994.

[36] Samuel Landau, Sébastien Picault, and Alexis Drogoul. ATNoSFERES : a Model for Evolutive Agent Behaviors. In *Proc. of AISB'01 symposium on Adaptive Agents and Multi-agent +systems*, pages 95–99, march 2001. ISBN 1 902956 17 0.

[37] A. Lapedes and R. Farber. *Neural Information Processing Systems*, chapter How neural nets work. New York: American Institute of Physics, 1987.

[38] Alvin J. Owens Lawrence J. Fogel and Michael J. Walsh. *Artificial Intelligence Through Simulated Evolution*. Wiley, 1966.

[39] Aristid Lindenmayer. Mathematical models for cellular interaction in development, parts i and ii. *Journal of theoretical biology*, (18):280–315, 1968.

[40] Sean Luke and Lee Spector. Evolving graphs and networks with edge encoding: Preliminary report. In *Late Breaking Papers of the Genetic Programming 96 (GP96)*, 1996.

[41] Wolfgang Maas and Christopher M. Bishop, editors. *Pulsed Neural Networks*. MIT Press, 1999.

[42] E. H. Mamdani. Application of fuzzy algorithms for control of simple dynamic plants. In *Institution of Electrical Engineers*, 1974.

[43] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, (5):115–133, 1943.

[44] J.A. Meyer, P. Husbands, and I. Harvey. Evolutionary robotics: a survey of applications and problems. In P. Husbands and J.A. Meyer, editors, *Proceedings of The First European Workshop on Evolutionary Robotics - EvoRobot98*. Springer Verlag, 1998.

[45] Jean-Arcady Meyer. Evolutionary approaches to neural control in mobile robots. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, San Diego, 1998.

[46] Zbigniew Michalewicz, Dipankar Dasgupta, Rodolphe G LeRiche, and Marc Shoenauer. Evolutionary algorithms for constrained engineering problems, 1996.

[47] Olivier Michel, Manuel Clergue, and Philippe Collard. Artificial neurogenesis: Applications to the cart-pole problem and to an autonomous mobile robot, 1997.

[48] D. Montana and L. Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 762–767. Morgan Kaufmann, 1989.

[49] Stefano Nolfi and Dario Floreano. *Evolutionary Robotics*. MIT Press, 2000.

[50] Frank Pasemann and Ulrich Dieckmann. Evolved neurocontrollers for pole-balancing. In *Biological and Artificial Computation: From Neuroscience to Technology. IWANN'97*, 1997.

[51] Joao Carlos Figueira Pujol and Riccardo Poli. Evolving neural controllers using a dual representation. Technical Report CSRP-97-25, School of Computer Science, University of Birmingham, september 1997.

[52] N.J. Radcliffe. *Genetic Neural Networks on MIMD Computers*. PhD thesis, Dept. of Theoretical Phys., University of Edinburgh, UK, 1990.

[53] F. Rosenblatt. The perceptron : a probabilistic model for information storage and organization in the brain. *Psychological review*, pages 386–408, 1965.

[54] D.E. Rumelhart and J.L. McClelland. *Parallel Distributed Processing. Explorations in the Microstructure of Cognition*. The MIT Press/Bradford Books, 1986.

[55] Hans-Paul Schwefel. *Numerische Optimierung von ComputerModellen mittels der Evolutionsstrategie*. Birkhäuser, 1977.

[56] Hans-Paul Schwefel and Thomas Bäck. Evolution strategies ii: Theoretical aspects. In *Genetic Algorithms iEngineering and Computer Science, Proc. First Short Course EUROGEN-95*. 1995.

[57] H. Shim, T.J. Koo, F. Hoffmann, and S. Sastry. A comprehensive study of control design for an autonomous helicopter. In *IEEE conference on Decision and Control CDC'98 (submitted)*, 1998.

[58] K. Sims. Evolving 3d morphology and behavior by competition. In R.A. Brooks and P. Maes, editors, *Proceedings of the Fourth International Workshop on Artificial Life*. The MIT Press, 1994.

[59] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. Technical Report TR-AI-01-290, University of Texas at Austin, June 2001.

[60] T. Takagi and M. Sugeno. Fuzzy identification of systems and its application to modelling and control. In *IEEE Transactions on Systems, Man and Cybernetics*, volume 15. 1985.

[61] P.J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavior Sciences*. PhD thesis, Harvard University, Committee on Applied Mathematics, 1974.

[62] D. Whitley. Applying genetic algorithms to neuralnetwork learning. In *Proceedings of the 7th Conference for the Study of Artificial Intelligence and Adaptive Behavior*, 1989.

[63] D. Whitley, F. Gruau, and L Pyeatt. Cellular encoding applied to neurocontrol. In *Siwth International Conference on Genetic Algorithms*, 1995.

[64] D. Whitley and T. Hanson. Optimizing neural networks using faster, more accurate genetic search. In J.D. Shaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications*, pages 391–396. Morgan Kaufmann, 1989.

[65] D. Whitley, T. Starkweather, and C. Bogart. Genetic algorithm and neural networks: Optimizing connexions and connectivity. *Parallel Computing*, pages 347–361, 1990.

[66] Yao. Evolving artificial neural networks. *PIEEE: Proceedings of the IEEE*, 87, 1999.

[67] X. Yao and Y. Liu. A new evolutionary system for evolving artificial neural networks. *IEEE Transactions on Neural Networks*, 8(3):694–713, 1997.

[68] Lofti Zadeh. Fuzzy sets. *Journal of Information and Control*, 8:338–353, 1965.